CS6240: Final Project Report

Tariq Anwar – Yasaman Ahrabi

Library and Language Used:

Our project is using the **MLlib**, a machine learning library built on top of Apache Spark and the solution is implemented in **Scala** language.

Features used:

All the core covariates and extended covariates (except NLCD_YYYY_FS_* features) and (Latitude, Longitude, Year, Month, Day, Time) from checklist were used.

Model used and parameters:

We started with the Decision Tree model and got ACC: 0.75. Then we chose to explore the Random Forest ensemble model to see whether we can get higher accuracy.

Spark decision tree classification algorithm takes some parameters that can significantly influence quality of created model. There is no simple theory that tells what values should be used. Answer can be found by iterations, and selecting model with best characteristics.

The parameters for Random Forest are as follows:

- impurity measure- we used gini measure.
- maxDepth: Maximum depth of each tree in the forest.
- maxBins: Number of bins used when discretizing continuous features.
- numTrees: Number of trees in the forest.

To find the best parameters we ran a tuning program to explore each configuration and found the best model by looking at the maximum ACC of each configuration. We used cross validation method with **(80/20) split ratio**. We explored **numTrees from 2 – 20** and **maxDepth in the range {10,15,20,25}**. The ACC results for each of the configurations explored are as below:

The ACC is calculated as follows:

(Number of true positives + Number of true negatives) / (Number of true positives + Number of true negatives + Number of false positives + Number of false negatives)

Test 1:

maxDepth	NumTrees	ACC
10	2	0.760
	4	0.774
	6	0.778
	8	0.775
	10	0.776
	12	0.776
	14	0.777

	16	0.777
	18	0.776
	20	0.777

Test 2:

maxDepth	NumTrees	ACC
15	2	0.785
	4	0.806
	6	0.811
	8	0.812
	10	0.813
	12	0.814
	14	0.811
	16	0.814
	18	0.813

Test 3:

maxDepth	NumTrees	ACC
20	2	0.802
	4	0.831
	6	0.839
	8	0.842
	10	0.843
	12	0.844
	14	0.846
	16	0.845
	18	0.846
	20	0.845

Test 4:

maxDepth	NumTrees	ACC
25	20	0.859

As it is evident from the above statistics, the greater the depth and number of trees, the better the ACC. So the best parameters for the chosen model is: **20 trees and depth 25.**

Psuedo-Code:

* Partitioning and worker assignment is added in red color.

```
Class Tuning {
      labeledDataRDD = Read the labeled data file
      //each worker gets a split of data file
```

```
      Parse the labeledDataRDD and create maps(String -> Int) for categorical
features
      (e.g. column#: 962)
      labeledPointRDD = labeledDataRDD.map( x =>
            label = x.label
            features = x.features.map(f => f.toDouble if continuous or
categoricalMap(f) if categorical)
            LabeledPoint(label, features))
      // each worker works on a set of rows in labeledDataRDD to create
labeldPointRDD by mapping the rows into LabeledPoints
      Random split labeledPointRDD into trainData and testData (80/20)
      for numTrees in Range (2 – 20):
            model = train a RandomForest(numTrees, maxDepth, trainData)
            // Read MLlib description section
            for each testData:
                  predictRDD += model.predict(testData)
                  // Read MLlib description section
            compute ACC for testData
            print (numTrees, ACC)
}
```

```
Class Training {
      labeledDataRDD = Read the labeled data file
      //each worker gets a split of data file
      Parse the labeledDataRDD and create maps(String -> Int) for categorical
features
      (e.g. column#: 962)
      labeledPointRDD = labeledDataRDD.map( x =>
            label = x.label
            features = x.features.map(f => f.toDouble if continuous or
categoricalMap(f) if categorical)
            LabeledPoint(label, features))
      // each worker works on a set of rows in labeledDataRDD to create
labeldPointRDD by mapping the rows into LabeledPoints
      Random split labeledPointRDD into trainData and testData (80/20)
      numTrees = 20
      maxDepth = 25
      model = RandomForest(numTrees, maxDepth, trainData)
      // Read MLlib description section
      save model to file
}
```

```
Class Predicting {
      unlabeledDataRDD = Read the unlabeled data file
      //each worker gets a split of data file
```

```
	model = Load the model from file system
	Parse the unlabeledDataRDD and create maps(String -> Int) for
categorical features
	(e.g. column#: 962)
	resultRDD = labeledDataRDD.map( x =>
		label = 0
		features = x.features.map(f => f.toDouble if continuous or
categoricalMap(f) if categorical)
		lp = LabeledPoint(label, features)
		predict = model.predict(lp)
		// Read MLlib description section
		(x.sampling_event_id , predict))
	Save resultRDD to output csv file
}
```

MLlib Random Forest:

Parallelization of training process in MLlib:

To parallelize training, MLlib's implementation of decision tree partitions the input data across its workers. During training, each node that needs to be split is put onto a queue. At each iteration, some number of nodes is pulled from the queue, depending on the available memory. Then, depending on the impurity function, statistics are collected across the partitioned dataset. The statistics for the same node are then shuffled to the same worker, which decides the best splits for the node. Finally, these best splits are sent back to the master, which grows the tree by creating more nodes, adding them into the queue as needed.

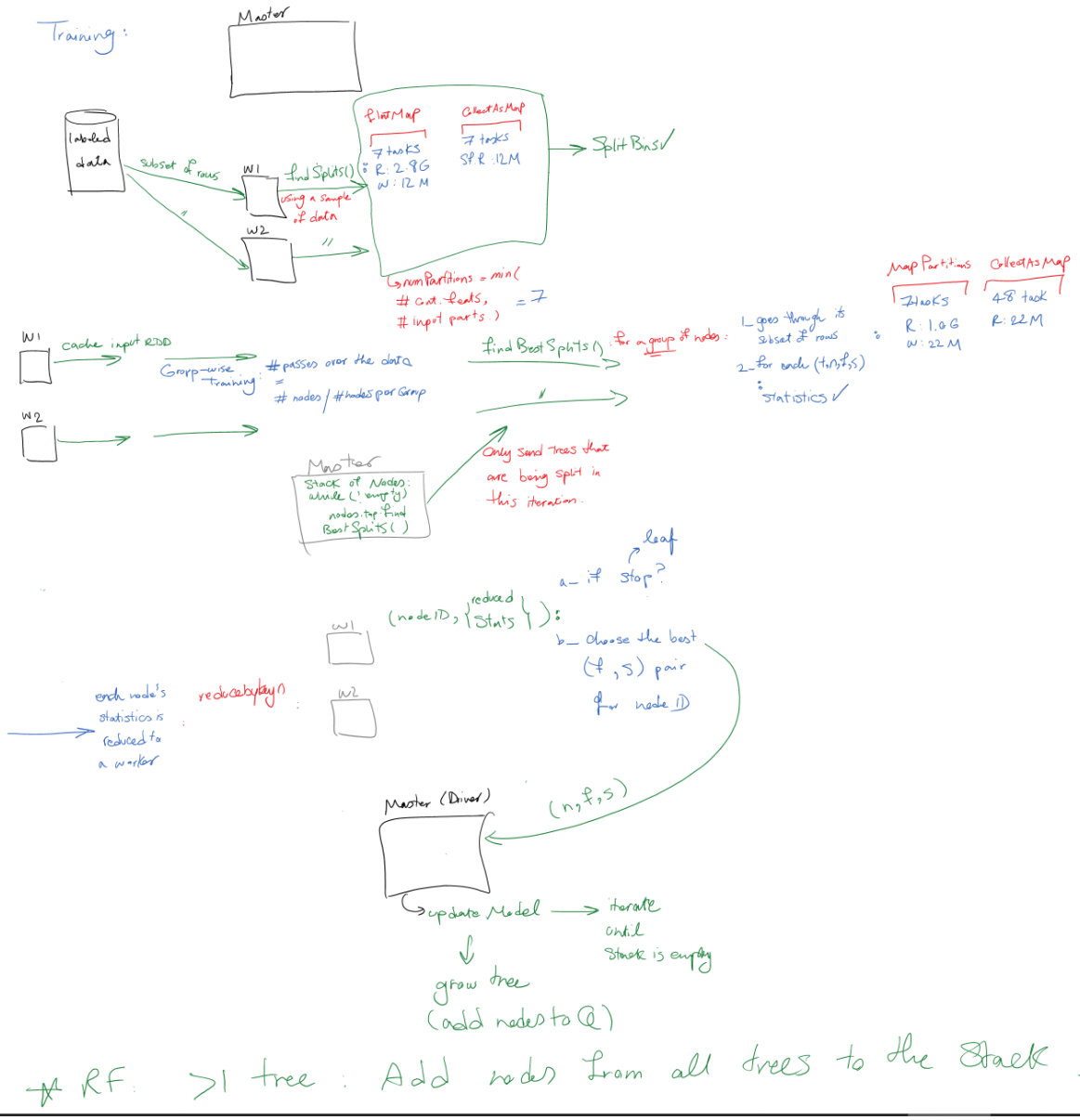
Under the cover, a decision tree is treated as a random forest with one tree. MLlib trains multiple trees at once by adding nodes from all of them to the queue.

Pseudo-code for Decision Tree training in MLlib:

```
function MLlibTrainDecisionTree(S){
	splitBins = findSplitBins(S)
	nodeQueue = Queue()
	nodeQueue.enqueue(Node.emptyNode())
	while nodeQueue not empty do
		nodes = dequeueNodes(nodeQueue)
		findBestSplits(S, splitBins, nodes, nodeQueue)
}
function findBestSplits(S, splitBins, nodes, nodeQueue) {
	statistics = collectStatistics(S, splitBins, nodes)
	splits = findBestSplit(S, splitBins, nodes, statistics)
	for (node, split) in zip(nodes, splits) do
		if StoppingCriteriaMet(node, split) then
			makeLeaf(node, split)
		else
			node.split()
			nodeQueue.enqueue(node.leftChild)
```

```
                    nodeQueue.enqueue(node.rightChild)
}
```

The following diagram shows the flow of data in MLlib's Random Forest implementation. Detailed information about partitioning and worker assignment is sketched below.



Prediction:

Function predictByVoting in TreeEnsembleModels.scala: Classifies a single data point based on majority votes. The vote of each tree is calculated navigating the decision tree using the features of the data point as a guide.

The model we used was about 80 MB which fits in the memory of a single machine. Thus one machine is used to run the prediction program. So increasing the number of machines/partitions won't affect the performance.

Affecting the performance of MLlib Random Forest:

MLlib's Random Forest partitions the input labeled data RDD so that each worker machine gets a split of input. After that the amount of data being shuffled among machines depends on the amount of statistics for nodes, which is about 20 MB for our input. To affect the performance, one could change the partitioning of input RDD, or the following parameters in Random Forest:

- **maxMemoryinMB**: Amount of memory to be used for collecting sufficient statistics.
  The default value is conservatively chosen to be 256 MB to allow the decision algorithm to work in most scenarios. Increasing maxMemoryInMB can lead to faster training (if the memory is available) by allowing fewer passes over the data. However, there may be decreasing returns as maxMemoryInMB grows since the amount of communication on each iteration can be proportional to maxMemoryInMB.
- **useNodeIdCache**: If this is set to true, the algorithm will avoid passing the current model (tree or trees) to executors on each iteration.
  This can be useful with deep trees (speeding up computation on workers) and for large Random Forests (reducing communication on each iteration).

Running Time results:

Tuning:

Program:	Number of machines:	Duration:
Trees : (2 – 20) , Depth = 20	6 (1 master, 5 workers)	2.2 hrs
Trees : (2 – 20) , Depth = 15	6 (1 master, 5 workers)	51 mins
Trees : (2 – 20) , Depth = 10	11 (1 master, 10 workers)	36 mins

Training:

Program:	Number of machines:	Duration:
Trees : 20 , Depth = 20	6 (1 master, 5 workers)	32 mins
Trees : 20 , Depth = 25	6 (1 master, 5 workers)	1.0 hr
Trees : 20, Depth = 25	1 (Standalone)	5.4 hrs

Prediction:

Program:	Number of machines:	Duration:
Trees : 20 , Depth = 25	6 (1 master, 5 workers)	3.2 mins

References:

- To calculate the statistics for ACC, … we referred:
  https://grzegorzgajda.gitbooks.io/spark-examples/
- MLLib Spark documentation:
  http://spark.apache.org/docs/latest/mllib-ensembles.html#random-forests
  https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/ml/tree/impl/RandomForest.scala