Radboud University Nijmegen

BSc. Thesis

SOW-BKI300

Exploring the Impact of BudgetPrune on Apache Spark Random Forest Performance

Author:
Koen Dercksen
s4215966

Supervisors:
Johan Kwisthout,
Arjen de Vries

July 4, 2017

Radboud University

Abstract

This thesis explores the impact of a previously proposed pruning algorithm for random forest ensembles called BudgetPrune [9]. BudgetPrune tries to optimize the tradeoff between prediction accuracy and feature acquisition cost, allowing for accurate prediction in resource-constrained environments. Using Apache Spark ML's random forest model as a baseline, the influence of the pruning step on prediction accuracy and cost is examined.

1 Introduction

Modern classification systems such as search-engines and recommendation systems face prediction-time budget constraints. These budgets arise due to computation time or data transfer time used in extracting features from data. It is beneficial to minimize these acquisition costs while maintaining prediction accuracy, in order to save resources and/or speed up the classification system. BudgetPrune [9] is an algorithm used to prune random forest classifiers in order to minimize these costs while maintaining the prediction accuracy of the original classifier.

Random forest classifiers are scalable to large datasets; they use a random subset of features per tree, and the combination of trees makes up for any overfitting that might happen when using individual decision trees. BudgetPrune takes a constructed random forest as a baseline and prunes features that are sparsely used across trees, leading to cost reduction with minimal decrease in accuracy. Optimal pruning drives the model to either use features a large number of times, allowing for complex decision boundaries in the space of those features, or not use features at all and avoiding the cost associated with acquiring them.

In this thesis I implement BudgetPrune for Apache Spark [6] random forest classifiers, aiming to investigate the effect the algorithm has on the performance of these models. Since Spark often deals with large datasets distributed across multiple servers, reducing the average feature acquisition cost can decrease runtime and memory usage dramatically.

2 Theory

The research by Nan et al. [9] that my work is based on considers solving the Lagrangian relaxed problem of finding an optimal model considering prediction-time resource constraints, also known as the error-cost tradeoff problem:

$$\min_{f \in \mathcal{F}} E_{(x,y)\sim\mathcal{P}}[e(y, f(x))] + \lambda E_{x\sim\mathcal{P}_x}[C(f, x)] \tag{1}$$

where sample/label pairs (x, y) are drawn from a distribution $\mathcal{P}$, $e(y, \hat{y})$ is the error function ($\hat{y}$ is the predicted label, y is the actual label), $C(f, x)$ is the cost of running sample x through classifier f and λ is the tradeoff parameter (i.e. how heavy will feature cost influence the final solution). In the case of this thesis, $\mathcal{F}$ is the space of possible random forest (RF) classifiers. Each RF is made up of trees $\mathcal{T}_0, \ldots, \mathcal{T}_n$.

2.1 Lagrangian dual problem

The method of Lagrange multipliers is a strategy for optimizing a function subject to equality constraints. As an example, imagine the following problem:

$$\max f(x, y) \text{ s.t. } g(x, y) = c \tag{2}$$

We can introduce a new variable λ called a *Lagrange multiplier* and define a new function:

$$\mathcal{L}(x, y, \lambda) = f(x, y) - \lambda(g(x, y) - c) \tag{3}$$

We can then find a λ such that we minimize $\mathcal{L}$.

The duality principle is that we can view an optimization problem from two perspectives; the *primal* and *dual* problem. A solution to the dual problem gives a lower bound to the solution of the primal problem. The two solutions do not have to be equal; the difference between them is called the duality gap, and an optimization algorithm will typically stop when this gap is small enough.

2.2 Formal definition of pruning

I will be using the notation defined in Section 3 of Nan et al. [9]. In order to formulate random forest pruning in terms of Eq. (1), we will define pruning as a 0-1 integer problem. Given a decision tree $\mathcal{T}$, we will index its nodes as $h \in \{1, \ldots, |\mathcal{T}|\}$.

First, introduce the binary variable z for each node h in tree $\mathcal{T}$:

$$z_h = \begin{cases} 1 & \text{if node } h \text{ is a leaf in the pruned tree} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Let $p(h)$ be the set of nodes that lie on the path from the root to and including h. Consider that every pruned tree $\mathcal{T}^p$ should be a subset of the original tree $\mathcal{T}$; thus, all valid pruned trees satisfy the constraints:

$$\sum_{u \in p(h)} z_u = 1 \tag{5}$$

where h is any leaf node in the tree. Take for example the tree in Fig. 1; $p(2)$ would be made up of $\{z_0, z_1, z_2\}$. The constraint specified in Eq. (5) holds for this combination of nodes since only $z_2 = 1$ because it is a leaf node. We can repeat this for every leaf node in the tree and the constraint will hold; hence, this tree is valid. Were we to prune the tree at node 1, $z_0 + z_1 = 1$ holds and thus this pruned tree is valid, as well as a subset of the original tree ($\{0, 1, 4, 5, 6\} \subset \{0, 1, 2, 3, 4, 5, 6\}$).

We can define the expected error as the number of misclassified training samples in the leaf nodes:

$$e_h = \sum_{i \in S_h} \mathbb{1}_{[y^{(i)} \neq Pred_h]}$$

where S_h is the subset of total samples that is routed through node h and $Pred_h$ is the predicted label at h. To illustrate, imagine that we run 100 samples through the tree in Fig. 1 and the first 50 of those samples end up going down the left subtree. At node 0 we have $S_0 = \{s_1, \ldots, s_{100}\}$. At its child nodes we would have $S_1 = \{s_1, \ldots, s_{50}\}$, and $S_4 = \{s_{51}, \ldots, s_{100}\}$.

Lastly, given the binary variables:

$$w_{k,i} = \begin{cases} 1 & \text{if feature } k \text{ is used by sample } i \text{ in any tree } \mathcal{T} \\ 0 & \text{otherwise} \end{cases}$$

we define the expected feature cost of an example as $\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} w_{k,i}$ (we always assume a feature cost of 1.0).

Using these definitions, we can combine the pruning constraints, error and costs into an integer program based on the way the variables are related to each other.

$$\min_{z_h^{(t)}, w_{k,i}^{(t)}, w_{k,i} \in [0,1]} \overbrace{\frac{1}{NT} \sum_{t=1}^{T} \sum_{h \in \mathcal{T}_t} e_h^{(t)} z_h^{(t)}}^{\text{error}} + \lambda \left(\overbrace{\frac{1}{N} \sum_{k=1}^{K} \sum_{i=1}^{N} w_{k,i}}^{\text{feature acquisition cost}} \right) \tag{6}$$

For additional details on variables and their meaning that may not be defined in this paper, refer back to Nan et al. [9].

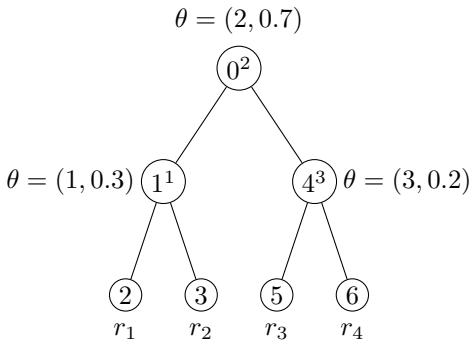


Figure 1: Example of tree with 7 nodes and 3 feature splits. The r_n below the leaf nodes correspond to rows in Eq. (7). $\theta(f, t)$ routes samples with feature $f \leq t$ to the left subtree.

2.2.1 LP relaxation

Nan et al. [9] show that Eq. (6) can be relaxed to a linear programming problem given Lemma 3.1 in their paper. I will provide an example of the transformation of constraints into a network matrix. A network matrix has a single -1 and a single 1 in each column, the rest of the values in that column being zero. First we define the rows for the sum constraint on the node variable defined in Eq. (4):

$$\begin{array}{c} \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{array} \begin{array}{c} \begin{array}{ccccccc} z_0 & z_1 & z_2 & z_3 & z_4 & z_5 & z_6 \end{array} \\ \left[\begin{array}{ccccccc} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{array}\right] \end{array} \tag{7}$$

Each row in Eq. (7) defines Eq. (5) on a possible path through the tree displayed in Fig. 1; for example, if we take the path from the root node to leaf node 2, this is translated to the constraint $z_0 + z_1 + z_2 = 1$. The second step is to expand this matrix to include the expanded constraint as defined in Lemma 3.1 of Nan et al. [9], where each feature split is considered a type of extra child node:

$$
\begin{array}{c}
 & \begin{array}{ccccccccccc} z_0 & z_1 & z_2 & z_3 & z_4 & z_5 & z_6 & w_{1,1}^{(1)} & w_{2,1}^{(1)} & w_{2,2}^{(1)} & w_{3,2}^{(1)} \end{array} \\
\begin{array}{c} r_1 \\ r_2 \\ fr_1 \\ r_3 \\ r_4 \\ fr_4 \\ fr_2 \\ fr_3 \end{array} &
\left[\begin{array}{ccccccccccc}
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}\right]
\end{array}
\tag{8}
$$

The extra four rows fr_n in Eq. (8) define the feature acquisition constraints for two examples, example 1 being routed to leaf node 2 and example 2 to leaf node 5. For example, $s_1 = (.2, .5, 0)$ and $s_2 = (0, .8, .1)$ would result in the proposed routings.

Now, assuming that the rows are ordered so that the leaf nodes are in a post-order fashion (in the case of Fig. 1, post-order would be 2-3-1-5-6-4-0), we can transform the matrix in Eq. (8) into an equivalent network matrix Eq. (9) through row operations:

$$
\begin{array}{c}
 & \begin{array}{ccccccccccc} z_0 & z_1 & z_2 & z_3 & z_4 & z_5 & z_6 & w_{1,1}^{(1)} & w_{2,1}^{(1)} & w_{2,2}^{(1)} & w_{3,2}^{(1)} \end{array} \\
\begin{array}{c} -r_1 \\ r_1{-}r_2 \\ r_2{-}fr_1 \\ fr_1{-}r_3 \\ r_3{-}r_4 \\ r_4{-}fr_4 \\ fr_4{-}fr_2 \\ fr_2{-}fr_3 \\ fr_3 \end{array} &
\left[\begin{array}{ccccccccccc}
-1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & -1 & -1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}\right]
\end{array}
\tag{9}
$$

You could extend this matrix to hold a complete RF, but the desired property of this approach is that it is now possible to solve the systems that these matrices represent for each tree individually. The network matrices can be converted to shortest path problems and solved easily in parallel (see Fig. 2).

2.3 Primal-dual approach

The problem defined in Eq. (6) is NP-hard to solve, and even though its LP relaxation is solvable in polynomial time it can still be way too large to solve within a feasible timeframe. The complexity comes down to $O(T \times |T_{max}| + N \times T \times K_{max})$ [9] where T is the number of trees in the RF, $|T_{max}|$ is the maximum number of nodes in a tree, N is the number of samples in the used dataset and K_{max} is the maximum number of features a sample uses within a tree. In other words, we are looking at $O(T^3)$ scaling, which is really bad when our ensembles

become large. The primal-dual approach proposed by Nan et al. [9] decomposes the optimization into many subproblems that can be parallelized; each one is then solved as a shortest path problem. This improves the runtime complexity to $O(\frac{T}{p}(|T_{max}| + N \times K_{max}) \log(|T_{max}| + N \times K_{max}))$ where p is the number of processors available. Now the complexity scales only with $\frac{T}{p}$ and we can use much larger ensembles.

To accomplish this, Nan et al. [9] improve the problem of Eq. (6) into a primal-dual problem:

$$\max_{\substack{\beta_{k,i}^{(t)} \geq 0 \\ w_{k,i}^{(t)} \in [0,1] \\ w_{k,i} \in [0,1]}} \min_{z_h^{(t)} \in [0,1]} \frac{1}{NT} \sum_{t=1}^{T} \sum_{h \in \mathcal{T}_t} \hat{e}_h^{(t)} z_h^{(t)} + \lambda \left(\frac{1}{N} \sum_{k=1}^{K} \sum_{i=1}^{N} w_{k,i} \right) + \sum_{t=1}^{T} \sum_{i=1}^{N} \sum_{k \in K_{t,i}} \beta_{k,i}^{(t)} (w_{k,i}^{(t)} - w_{k,i}) \tag{10}$$

Intuitively, the transformation from Eq. (6) to Eq. (10) can be explained as follows: rather than trying to minimize error/cost over the ensemble, we minimize error/cost for each individual tree first (primal problem). Then we can determine the difference between the number of times the feature is used in a single tree and the number of times it is used in the ensemble ($w_{k,i}^{(t)} \leq w_{k,i}$); the dual problem. From the solution to the dual problem, we can derive new feature costs for every tree (Eq. (11) and Eq. (12)) and repeat the same steps until the solutions converge. To clarify, $\mathcal{T}_{k,i}$ is the set of trees where sample i encounters feature k.

$$\mu_{k,i} = \frac{\lambda}{N} - \sum_{t \in \mathcal{T}_{k,i}} \beta_{k,i}^{(t)} \tag{11}$$

$$w_{k,i} = \begin{cases} 0 & \text{if } \mu_{k,i} > 0 \\ 1 & \text{otherwise} \end{cases} \tag{12}$$

Eq. (10) is also subject to constraints mentioned in Section 4 of Nan et al. [9] (the algorithm used to solve this problem is mentioned there as well). Also note the lack of c_k here, too, since the feature cost in my experiments is always set to 1.0.

3 Implementation

The implementation in Nan et al. [9] uses IBM's CPLEX [2] solver for the primal update step, whereas I wrote all the code from parsing the Spark RFs to prediction with the pruned RFs myself. At the time of writing the code, there was no open source code for BudgetPrune released by Nan et al., however it was released recently [7].

3.1 Baseline model

The model that I started out with is an Apache Spark RF classifier. I decided on using the RF classifier from the ML API, since the old MLlib API will be deprecated in the future. The process is basically as follows:

Listing 1: Train a basic RF classifier.

```
// Load libsvm data from file
val data = spark.read.format("libsvm").load("path/to/data.txt")

// Split data into train/test chunks
val Array(train, test) = data.randomSplit(Array(0.7, 0.3))

// Train a random forest classifier on the training split
val rf = new RandomForestClassifier()
    .setLabelCol("label")
    .setFeaturesCol("features")
    .setImpurity("gini")
    .setFeatureSubsetStrategy("auto")
    .setMaxDepth(5)
    .setMaxBins(32)
    .setNumTrees(20)
    .fit(train)

// Predict classes for the test split
val predictions = rf.transform(test)

// Create an evaluator to calculate the prediction accuracy
val ev = new MulticlassClassificationEvaluator()
    .setLabelCol("label")
    .setPredictionCol("prediction")
    .setMetricName("accuracy")

println("Prediction accuracy: " + ev.evaluate(predictions))

// Save model to file for reuse
rf.save("path/to/model")
```

The values listed here are the ones that I used for each of the base models. In principle one could use any random forest as input for the pruning algorithm,

so I did not try to optimize anything about the values used; I did however keep them the same across the different base models that I trained in order to get consistent results when the time came to compare to the pruned models.

3.2 BudgetPrune in Spark

Implementing the pruning algorithm was the meat of the thesis. I encountered quite a few issues along the way, and I will detail my work in this section.

3.2.1 Parsing Spark RFs

The ML API is a highlevel abstraction meant to be easy to use, but this means that the core data of the model is hard to get to. When saving a RF to file, it is serialized in a format that can be loaded as a *DataFrame*; see the ML documentation [14] for more details.

First, I defined some datatypes to hold the RF structure.

Listing 2: Data structures to hold Spark RF

```
case class PruneTree(
  // Different ID for every tree in the RF
  treeID: Int,
  // List of nodes in this tree
  nodes: Array[PruneNode]
)

case class PruneNode(
  id: Int,
  // The gini coefficient calculated at this node
  impurity: Double,
  // Class to predict if this node were a leaf node
  prediction: Double,
  // Count per class of all the examples coming through this node
  impurityStats: Array[Double],
  // Information gain at this node
  gain: Double,
  // ID of left/right child
  leftChild: Int,
  rightChild: Int,
  // Feature split information
  split: PruneSplit
)

case class PruneSplit(
  // Feature that is used to split on in this node
  featureIndex: Int,
  // When this feature is categorical, the array holds a list of values
  // that go down the left subtree. When the feature is continuous,
  // array[0] holds the threshold value: f <= threshold goes down the
```

```
  // left subtree
  leftCategoriesOrThreshold: Array[Double],
  // -1 if the feature is continuous, else holds the number of
  // categories for this feature
  numCategories: Int
)

// Define a RF as a list of trees
type PruneRF = Array[PruneTree]
```

These datatypes copy the internal structure of the Spark datatypes to make it easy to parse the models from file. The actual parsing code can be found in the complete implementation (`TreeIO.scala`).

3.2.2 Building the constraint matrix

I approached this preparatory part of the algorithm in two main steps.

- Create node constraints (Eq. (7)).
- Create feature constraints and stack them with the node constraints, preserving post-order arrangement of the rows (Eq. (8)).

The node constraints are fairly straightforward to create. Simply find all possible root to leaf paths in the tree; then for each path, create a row in the constraint matrix, setting the node variables in the path to 1 and the rest to zero.

Listing 3: Find all paths in the tree.

```
def findAllPaths(tree: PruneTree): Seq[PruneNode] = {
  val leafs = tree.nodes.filter(_.leftChild == -1)
  for (leaf <- leafs) yield ShortestPath.pathToNode(leaf.id, tree)
}
```

We can then build the first part of the matrix by iterating over all the paths, creating rows as previously mentioned. The next step is to create the feature constraints. This is a little more involved and requires some additional preparatory work. For every sample in the dataset, we need to work out which features it uses when running through a tree (the set of features will thus be different per individual tree). We also need to keep in mind that BudgetPrune assumes that a feature, once acquired, is cached in memory and does not rack up additional acquisition cost when it is inspected more than once. I implement this caching per tree; i.e. if a tree looks at a certain feature more than once for a single sample, only the first time incurs an acquisition cost. There is reason to believe that this caching should be forest-wide; see Section 5 for more information.

To accomplish this I wrote a slightly altered prediction function that, given a `Prunetree` and a sample, returns the unique features acquired in order to

route the sample to a leaf node on the given tree (see `TransformTree.scala`). This also includes the tree nodes that were visited to get to the feature: for example, on row 5 of Eq. (8) we see that for sample 1 using feature 1 results in $w_{1,1} = 1$ as well as $z_0 = z_1 = 1$. Once we have this data, the code to create the additional matrix rows is almost identical to the code for the node constraints, with the exception of also setting $w_{k,i}$ variables to 1.

Matrix datatype It should be noted that Scala/Spark does not supply a Matrix datatype that fit my needs during this thesis. Datatypes such as `CoordinateMatrix` or `SparseMatrix` are supplied by Spark, but do not offer functionality for adding rows, iterating over the matrix or other utility functions that were necessary. Dealing with extremely sparse matrices, I decided to implement my own matrix datatype that was suitable to do row operations and sparse storage. An outline can be found below.

Listing 4: Vector and Matrix traits.

```
sealed trait Vector[T] extends Iterable[T] {
  // Length of vector
  def numElements: Int
  // Get element at index i
  def apply(i: Int): T
  // Invert sign of every element
  def inverted: Vector[T]
  // Get underlying sorted map structure
  def getMap: TreeMap[Int, T]
  // Get iterator to loop over vector
  def iterator: Iterator[T]
  // Addition with another vector (element-wise)
  def +(that: Vector[T]): Vector[T]
  // Subtraction with another vector (element-wise)
  def -(that: Vector[T]): Vector[T] = this + that.inverted
}

sealed trait Matrix[T] extens Iterable[(Int, Vector[T])] {
  // Number of rows in matrix
  def numRows: Int
  // Get row at index i
  def apply(i: Int): Vector[T]
  // Add a row at index i
  def addRow(i: Int, row: Vector[T]): Matrix[T]
  // Loop over rows
  def iterator: Iterator[(Int, Vector[T])]
  // Check if every column contains a single (-)1 and rest zeroes
  def isNetworkMatrix: Boolean
}
```

Based on these traits I implemented a `SparseMatrix` datatype backed by a `TreeMap` (a sorted map datatype available in Scala) in an effort to save memory

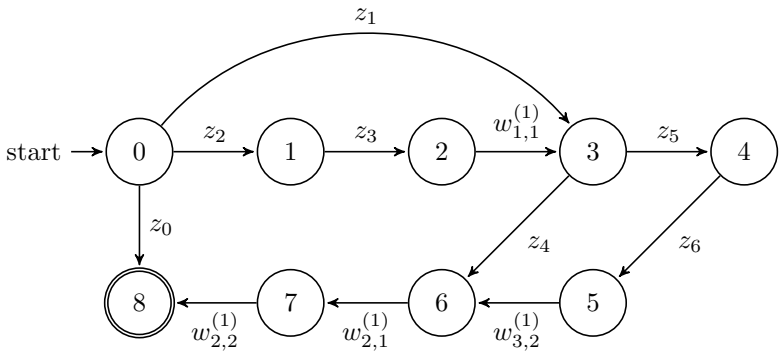


Figure 2: Example graph for Eq. (9). Note that the vertex numbers correspond to a row index in Eq. (9); i.e. vertex 0 corresponds to the first row $-r1$ in the matrix.

and not use fullsize arrays. Especially with larger datasets, this makes quite a difference in memory usage. For the full implementation, see `Matrices.scala`.

3.2.3 Minimization by shortest-path algorithm

The network matrix that results from transforming the constraint matrix rows of a tree as shown in Eq. (9) can be converted into a graph. Every row in the network matrix becomes a vertex in the graph, and the edges are defined by the location of the -1 and 1 values (see Fig. 2). Every edge has a cost associated with it; the edges with node variables use prediction error as their cost (these costs stay constant), and the edges with feature variables use costs computed from the dual variables (these costs thus change after every iteration).

Due to the fact that the rows in the network matrix are sorted in a post-order arrangement, the graph that results from the conversion is a *directed acyclic graph* (DAG) [12]. Finding the shortest path in a DAG can be done in linear time $O(N + M)$ where N is the number of edges and M is the number of vertices [1], in contrast to arbitrary graphs where shortest path algorithms are slower.

Topological ordering DAGs can always be topologically sorted. This means that the vertices are ordered such that for every directed arc uv from vertex u to v, vertex u comes before v in the ordering. As can be easily seen, the vertices in Fig. 2 are already numbered in topological order due to the order of the matrix rows in Eq. (9).

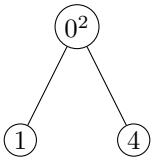
We can now find an optimal path through the DAG using the algorithm listed in Algorithm 1 (pseudocode, for real implementation see `TransformTree.scala`).

We can extract the optimal path from the `path` array returned in Algorithm 1. This path corresponds to a specific tree pruning. For example, if we find the shortest path in Fig. 2 to be $0 - 3 - 6 - 7 - 8$ we can see that

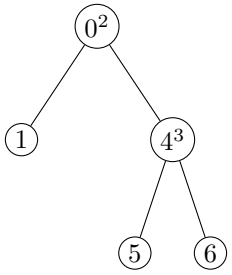
Algorithm 1: DAG shortest path algorithm in pseudocode.

```
input  : list of vertices V, source vertex s
output: array dist of the same length as V, containing for each vertex
        v ∈ V the shortest distance from s to v
let dist be an array of the same length as V; set dist[s] = 0 and all other
dist[v] = ∞;
let path be an array of the same length as dist, with all elements set to
null; each path[v] will hold the parent vertex in the path from s to v;
foreach vertex u ∈ V starting from s do
    foreach vertex v such that there exists an edge from u to v do
        let w be the cost associated with edge uv;
        if dist[v] > dist[u] + w then
            let dist[v] = dist[u] + w;
            let path[v] = u;
        end
    end
end
```

$z_1 = z_4 = w^{(1)}_{2,1} = w^{(1)}_{2,2} = 1$; the other variables are set to zero. This corresponds to pruning the tree in Fig. 1 at nodes 1 and 4, resulting in the tree seen in Fig. 3a. For another example, see also Fig. 3b.



(a) Tree from Fig. 1 pruned based on solution $0-3-6-7-8$ in Fig. 2.



(b) Tree from Fig. 1 pruned based on solution $0-3-4-5-6-7-8$ in Fig. 2.

Figure 3: Examples of tree prunings based on shortest-path solutions.

3.2.4 Ensemble optimization

Optimizing each tree gives us a list of plausible solutions that we can use to find a more accurate lowerbound solution to the primal problem. The (slightly simplified) code of the full algorithm looks like this:

Listing 5: For the full implementation, refer to `TransformTree.scala`

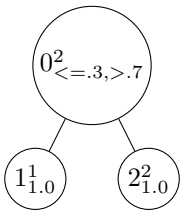
```
// Initialize dual variables to zero
var duals = Array.ofDim[Double](numTrees, numSamples, numFeatures)

// Create the network problems with some default feature costs
var solutions = findSolutions(ensemble, defaultCosts)

// Initialize duality gap to something big
var dualityGap = 1e8

// Alternate between primal/dual updates until converged or max
// iterations is reached
while (i < maxIterations || dualityGap < epsilon) {
    // Calculate primal variables
    val primals = updatePrimalVars(solutions, lambda, duals)
    // Find new solution to networks with updated primal variables
    solutions = findSolutions(ensemble, primals)
    // Update dual variables based on new primal solution
    duals = updateDualVars(primals, duals, solutions, stepsize)
    // Calculate duality gap based on difference in primal/dual
    // variables
    dualityGap = calculateGap(primals, duals)

    i += 1
}
```



(a) Pure leaf nodes. 70% of samples have class 2. Both leaf nodes classify all samples routed through them 100% correctly.

(b) Impure collapsed node resulting from Fig. 4a. This node now only classifies 70% of samples correctly.

Figure 4: Disadvantage of using majority voting with pruned trees. Predicted class at a node is in superscript, the accuracy of that prediction in subscript.

The `findSolutions` function is parallelized using Scala's builtin `Future` framework; see Listing 6.

Listing 6: Illustrating parallelization.

```
def findSolutions(ensemble, primals): Seq[Solution] = {
    // Distribute tree jobs over all available cores
    solutions = for (tree <- ensemble) yield {
        Future { solveForTree(tree, primals) }
    }
    // Blocking execution until all computations are done.
    // The timeout is infinite because the function is guaranteed to
    // return eventually, and we cannot continue the algorithm if
    // it would not.
    val futureSolutions = Future.sequence(solutions)
    Await.result(futureSolutions, Duration.Inf)
}
```

3.2.5 Alternate prediction rule

Spark's RF classifier by default uses a majority vote to classify unseen samples (a sample is routed through every tree and the class label that was predicted most often is the RFs classification for that particular sample). Depending on the tree depth, leaf nodes in trees that are not pruned are often pure or very close to pure (meaning that each leaf contains only one class of samples). When pruning the tree, internal nodes turn into leaves with mixed classes, making a simple majority vote less reliable. This is illustrated in Fig. 4.

Nan et al. [9] suggest the following prediction rule:

$$p = \frac{1}{T} \sum_{t=1}^{T} p_t \tag{13}$$

For each sample x, we acquire the probability distribution over label classes p_t at the leaf node that x is routed to. Then we aggregate those classes with Eq. (13) and return the class with the highest probability as prediction for x. This means that, in the case of Fig. 4b, we now know that the prediction is only 70% certain. If some other prediction ends up with a higher probability after running a sample through the whole ensemble, we can thus avoid some prediction errors potentially introduced by majority voting. Nan et al. [9] claim that their prediction rule consistently gives a lower prediction error than majority voting with pruned trees.

The code for this is relatively straightforward:

Listing 7: Alternate prediction rule.

```
// Simplified version of prediction function
def predict(forest: IndexedSeq[PruneTree], sample: Array[Double]):
    Double = {
    // Get a list with class probability distributions for each
    // tree in the forest
    val predictions = for (t <- forest) yield t.predict(sample)
    val summed = predictions.transpose.map(_.sum)
    // Return the class associated with the highest probability
    summed.zipWithIndex.maxBy(_._1)._2
}
```

4 Experiment

Depending on the tradeoff parameter λ, we expect to see the prediction accuracy drop when we put heavy emphasis on reducing feature acquisition cost; tree size will decrease resulting in lower feature acquisition cost. Likewise, a small λ will result in high prediction accuracy but a small-to-none decrease in feature acquisition cost.

4.1 Datasets

In order to test the performance of a pruned model versus the Apache ML baseline model, I selected three different datasets with a varying number of samples and features. The chosen sets are all fairly small; due to the complexity of the algorithm, trying to run it on large sets took extremely long. All datasets come from the UCI machine learning repository [5].

Dataset	Accuracy
Iris	97%
Glass	76.6%
Vehicle	71.2%

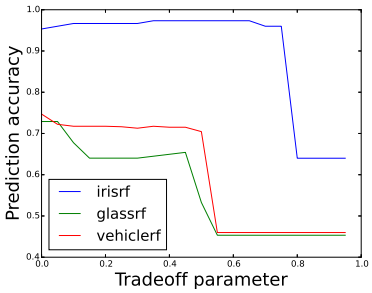
Figure 5: Prediction accuracy with base RF models.

Iris Dataset: Multivariate dataset with three different classes and 50 samples of each class. One class is linearly separable from the other two; the latter are not linearly separable from each other. Each sample has four continuous (real) features. The problem is to classify the type of iris flower for a particular sample.

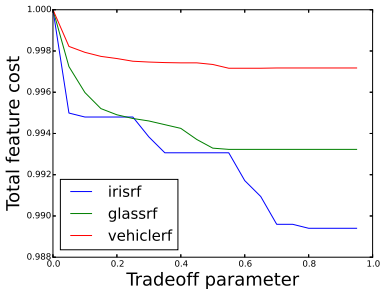
Glass Dataset: Multivariate dataset with six different classes and 214 total samples. Each sample has 10 continuous (real) features. The problem is to classify the type of glass for a particular sample.

Statlog (Vehicle Silhouettes) Dataset [10]: Multivariate dataset with four different classes and 946 total samples. Each sample has 18 continuous (integer) features. The problem is to classify a given sample as a type of vehicle.

The feature costs in these three datasets are all uniformly initialized to 1.0 in my tests. The feature values for all these sets have been normalized to $[-1, 1]$.



(a) Tradeoff curve for each of the three datasets.



(b) Impact of tradeoff on total feature cost (normalized to 0-1).

Figure 6: Tradeoff curves for prediction accuracy and total feature cost.

4.2 Results

The baseline RF models provided an accuracy baseline that can be found in Fig. 5. I ran my implementation on each model obtaining various points on the error/cost tradeoff curve, showing the impact of favouring accuracy over resource consumption or vice versa in Fig. 6a. The reduction of total feature cost with increase of λ can be seen in Fig. 6b. I also compared the custom prediction rule Eq. (13) to a regular majority vote rule on the Iris dataset, but I did not find any significant differences in prediction error.

5 Discussion

5.1 Unexpected results

Whereas the impact of error/cost tradeoff on prediction accuracy that I found was expected, I was surprised by the magnitude of the reduction in feature cost. As can be seen in Fig. 6b, the use of features only shrinks approximately 1% in the largest dataset and even less in the smaller sets. I suspect this has to do with either my implementation, the datasets used or a mixture of both factors.

Implementation The main problem area is the caching of acquired features. Nan et al. [9] assume that acquisition cost for a single feature is only incurred once; I implemented this for each tree individually, but I now suspect that caching has to be applied for the whole ensemble. Individual tree caching does not have a large impact unless the trees are fairly large and a single feature is acquired many times, whereas in hindsight global caching makes more sense to judge feature cost over an ensemble.

To illustrate, imagine a sample s made up of 4 features. Now consider a tree that has multiple splits on the same feature. In my implementation, only the first split that s encounters for a certain feature incurs a cost; the subsequent checks on that same feature do not incur additional cost in that tree. However, this process repeats itself for the next tree that s runs through. If every tree in the forest looks at the first feature in s, the total cost would add up to the feature cost times the number of trees in the forest. Global caching would mean that the acquisition cost is only incurred the first time s encounters a feature in *any* tree, and all other times that feature is used do not cause additional cost.

This difference in caching mechanism results in different network problems, which in turn results in a different optimal solution to Eq. (10). It would explain why I did not find the performance increase claimed by Nan et al. [9].

Datasets used As mentioned in Section 4, I used relatively small datasets for my tests in order to deal with slow code. The tiny number of features may have had an impact in not being able to reduce feature acquisition drastically, simply due to the fact that there were no features that could be 'disregarded' without compromising prediction accuracy. However, it should be noted that the large decrease in prediction accuracy coupled with relatively minimal change in feature acquisition points to the problem lying in the implementation.

5.2 Future research & improvements

Due to various problems along the way, this project has not covered everything it set out to do. There are some topics worthy of further investigation that I will discuss here.

Correct caching The first improvement on my work is to fix the caching problem highlighted in Section 5. Unfortunately, I did not have the time to fix

this issue myself; assuming that it is indeed the problem causing my findings, fixing it should improve pruned models to match up to the results found by Nan et al. [9].

Optimize code As mentioned in Section 3, I wrote all my code from scratch. This resulted in a severely impaired runtime compared to the runtimes that Nan et al. [9] reported (minutes versus hours with my implementation). Their code uses optimized libraries such as IBM's CPLEX [2] to solve the problem, and there is no way to compete with that unfortunately. I made everything from scratch as there is no package similar to CPLEX for Scala/Spark, however it is very likely that those packages will exist in the future and can be used to improve upon my work.

Prediction rule As mentioned in Section 4, I did not find any differences in prediction error using two different prediction rules on the Iris dataset. This is probably due to the small number of features and the relatively 'easy' dataset; maybe on a set with more features, the custom rule would make for a significant improvement. Unfortunately, there was no time to test this further.

Gradient descent for dual problem stepsize The current implementation uses a constant stepsize value for the dual variable update; see Section 3.2.4. Using a static value means that even though the difference between the dual and primal solutions is very large, the steps towards convergence have the same size. Thus, it may take way longer than needed to reach a solution when the dual and primal values start very far apart. Using gradient descent, you could project a suitable stepsize for each iteration based on the duality gap for that iteration, making the solutions converge faster. This means that the running time could be drastically decreased, especially for large datasets where the duality gap starts out with a gigantic value.

Explore Spark-specific properties The original intent of this project was to explore the scalability of BudgetPrune in a distributed framework like Spark. Unfortunately, I never got to this part. It would be interesting to see how BudgetPrune could be adapted to deal with distributed data, and how it could perform with a pipeline-suitable implementation.

Improve on memory requirements BudgetPrune is quite heavy on memory use. Consider a dataset with S samples and k features per sample, used by an ensemble of T trees; worst case, we store $2 \times T \times S \times k_{max}$ values ($w_{k,i}^{(t)}$ and $\beta_{k,i}^{(t)}$) plus another $S \times k$ for the primal variables $w_{k,i}$. For large datasets, this can really be a problem. Intuitively, it should be able to alter the algorithm such that we do not have to keep track of values for every feature of every sample. This would be a major improvement since it allows for better scaling to huge datasets that you would typically use in an environment like Spark.

References

[1] Nicos Christofides. *Graph Theory: An algorithmic approach*. Academic Press, Inc., New York, 1975.

[2] IBM ILOG CPLEX. V12. 1: User's manual for cplex. *International Business Machines Corporation*, 46(53):157, 2009.

[3] Marshall L Fisher. The lagrangian relaxation method for solving integer programming problems. *Management science*, 27(1):1–18, 1981.

[4] Yasser Ganjisaffar, Rich Caruana, and Cristina Lopes. Bagging gradient-boosted trees for high precision, low variance ranking models. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information*, SIGIR '11, pages 85–94, New York, NY, USA, 2011. ACM.

[5] M. Lichman. UCI machine learning repository, 2013.

[6] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *CoRR*, abs/1505.06807, 2015.

[7] Feng Nan. Budgetprune code release.

[8] Feng Nan, Joseph Wang, and Venkatesh Saligrama. Feature-budgeted random forest. In David Blei and Francis Bach, editors, *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1983–1991. JMLR Workshop and Conference Proceedings, 2015.

[9] Feng Nan, Joseph Wang, and Venkatesh Saligrama. Pruning random forests for prediction on a budget. *arXiv preprint arXiv:1606.05060*, 2016.

[10] B. Shepherd P. Mowforth. Statlog (vehicle silhouettes) dataset.

[11] Hanif D Sherali, Antoine G Hobeika, and Chawalit Jeenanunta. An optimal constrained pruning strategy for decision trees. *INFORMS Journal on Computing*, 21(1):49–61, 2009.

[12] K Thulasiraman and MNS Swamy. 5.7 acyclic directed graphs. *Graphs: Theory and Algorithms*, page 118, 1992.

[13] Joseph Wang, Kirill Trapeznikov, and Venkatesh Saligrama. Efficient learning by directed acyclic graph for resource constrained prediction. In *Advances in Neural Information Processing Systems*, pages 2152–2160, 2015.

[14] Matei Zaharia. Spark ml documentation.