# Incremental Random Forest Classifiers in Spark

by

Kathryn I. Siegel

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

~~Master of Science~~ in Computer Science and Engineering

[Master of Engineering] at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Signature redacted

Author ........................................................
Department of Electrical Engineering and Computer Science
May 20, 2016

Signature redacted

Certified by ........................................................
Samuel Madden
Professor, EECS
Thesis Supervisor

Signature redacted

Accepted by ........................................................
Dr. Christopher Terman
Chairman, Department Committee on Graduate Theses

# Incremental Random Forest Classifiers in Spark

by

Kathryn I. Siegel

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

## Abstract

The random forest is a machine learning algorithm that has gained popularity due to its resistance to noise, good performance, and training efficiency. Random forests are typically constructed using a static dataset; to accommodate new data, random forests are usually regrown. This thesis presents two main strategies for updating random forests incrementally, rather than entirely rebuilding the forests. I implement these two strategies—incrementally growing existing trees and replacing old trees—in Spark Machine Learning(ML), a commonly used library for running ML algorithms in Spark. My implementation draws from existing methods in online learning literature, but includes several novel refinements. I evaluate the two implementations, as well as a variety of hybrid strategies, by recording their error rates and training times on four different datasets. My benchmarks show that the optimal strategy for incremental growth depends on the batch size and the presence of concept drift in a data workload. I find that workloads with large batches should be classified using a strategy that favors tree regrowth, while workloads with small batches should be classified using a strategy that favors incremental growth of existing trees. Overall, the system demonstrates significant efficiency gains when compared to the standard method of regrowing the random forest.

Thesis Supervisor: Samuel Madden
Title: Professor, EECS

# Acknowledgments

I would like to express gratitude towards my master's thesis advisor, Professor Sam Madden, for providing guidance on my project for the past year. I would also like to thank Manasi Vartak, whose advice and assistance has greatly contributed towards my research and results.

THIS PAGE INTENTIONALLY LEFT BLANK

# Contents

**1 Introduction** **13**

1.1 Batched Workflows . . . . . 13

1.2 Random Forest Classifiers . . . . . 14

**2 Previous Work** **17**

2.1 Ensemble models . . . . . 17

2.2 Online random forests . . . . . 18

2.2.1 Saffari online random forests . . . . . 19

2.2.2 Denil online random forests . . . . . 20

2.2.3 Mondrian forests . . . . . 20

2.3 Concept drift . . . . . 21

2.4 Combined strategies . . . . . 22

**3 Methods** **25**

3.1 Incremental growth strategy . . . . . 25

3.2 Tree regeneration strategy . . . . . 27

3.3 Hybrid strategies . . . . . 27

3.4 Optimizations . . . . . 27

3.4.1 Bounding incremental growth . . . . . 28

3.4.2 Batched processing . . . . . 28

3.4.3 Tree reweighting . . . . . 28

3.5 API . . . . . 29

**4 Results** **31**

4.1 Experimental setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

4.2 Workload A: Large batches, no concept drift . . . . . . . . . . . . . . . 33

4.3 Workload B: Small batches, no concept drift . . . . . . . . . . . . . . . 37

4.4 Workload C: Large batches, concept drift . . . . . . . . . . . . . . . . . 41

4.5 Workload D: Small batches, concept drift . . . . . . . . . . . . . . . . . 45

**5 Conclusion** **51**

# List of Figures

4-1 Training time benchmarks for Homesite dataset . . . . . . . . . . . . 34

4-2 Error rate benchmarks for hybrid strategies on Homesite dataset . . . 36

4-3 Comparison of optimal hybrid strategies on Homesite dataset to control 37

4-4 Training time benchmarks for Otto Group dataset . . . . . . . . . . . 39

4-5 Error rate benchmarks for hybrid strategies on Otto Group dataset . 40

4-6 Comparison of optimal hybrid strategies on Otto Group dataset to control . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

4-7 Training time benchmarks for Airline Delay dataset . . . . . . . . . . 42

4-8 Error rate benchmarks for hybrid strategies on Airline Delay dataset 44

4-9 Comparison of optimal hybrid strategies on Airline Delay dataset to control . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

4-10 Training time benchmarks for Bike Sharing dataset . . . . . . . . . . 46

4-11 Error rate benchmarks for hybrid strategies on Bike Sharing dataset . 47

4-12 Comparison of optimal hybrid strategies on Bike Sharing dataset to control . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

5-1 Decision tree showing overview of results . . . . . . . . . . . . . . . . 52

THIS PAGE INTENTIONALLY LEFT BLANK

# List of Tables

3.1 Spark API for incremental random forests . . . . . . . . . . . . . . . . 30

4.1 Datasets used in experiments . . . . . . . . . . . . . . . . . . . . . . . . 33

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 1

# Introduction

Random forests are one of the most popular machine learning classifiers due to their robustness to noisy data, accuracy, and ability to handle "big data" workloads. However, the downside to using random forests is that, given additional training data, the random forest must be regrown from scratch. This work proposes several techniques to update a random forest incrementally without fully rebuilding the classifier, as well as insight into the performance and robustness of these techniques on different workloads. Specifically, I explore incremental methods that are responsive to shifts in overall data distribution. I develop a custom incremental random forest classifier and provide a Scala API through which users can call incremental random forest methods.

## 1.1 Batched Workflows

The work in this thesis is designed to handle the use case in which new batches of training data are constantly being added to a classifier. Such workloads are common in analytics, where observations are continuously collected in log entries, as well as in Internet of Things (IoT) networks, where extensive data collection takes place offline and data is transmitted periodically. Existing machine learning tools retrain a model on the entire dataset when new data is added. This entails reiterating over every single data point in the dataset, even when the added batch has a minimal effect on the resulting classifier. As such, incremental training in random forests should drastically improve the performance of these classifiers on batched workloads. This research seeks to implement and expose a Spark API that will allow data scientists to add data to the random forest using one of several incremental strategies.

## 1.2 Random Forest Classifiers

Before describing the details of my approach, it is important to introduce the mechanics of random forest classifiers. At a high level, random forests are collections of decision trees used for classification. Once grown, each decision tree classifies an unlabeled point by casting a vote, and the random forest reports the label with the most votes [11].

For this thesis, I used the Spark Machine Learning (ML) implementation of random forest classifiers. Apache Spark is a scalable data processing system that provides an engine for processing big data workloads. At its core is a structure called the resilient distributed dataset (RDD), which can be distributed over a cluster of machines and is fault-tolerant. A series of libraries run on Spark and take advantage of its cluster-computing capabilities. One such library is MLlib, which contains a wide array of machine learning tools. In this research, I model my incremental random forest classifier on the Spark random forest classifier, for the purposes of maintaining optimizations within the codebase that take advantage of Spark's strengths. Since each batch in a batched workload may contain a large dataset, implementing an incremental classifier in Spark allows us to take advantage of Spark's distributed computing capabilities for every batch.

The Spark ML random forest classifier favors batched and aggregated computation over single-datapoint processing. Each dataset is first preprocessed into RDDs with aggregated information about each point and its features. The Spark random forest classifier then randomly samples the data (with replacement) according to a Poisson distribution, and assigns a random sample to each decision tree in the forest. Each decision tree is grown from the root using its sampled dataset; the splitting criterion for each node in a decision tree is determined with an element of randomness. Specifically, at each node, the set of features is subsampled randomly. For each resulting set of features, the classifier examines all possible values for that feature on which the data can be split. Among all of these candidate splits, the classifier chooses the split that maximizes the decrease in Gini impurity, which is the probability that a point randomly selected from the node would be misclassified. The algorithm terminates when the maximum tree height is reached or no training points are misclassified within each individual decision tree.

The existing Spark ML random forest classifier performs well given a large dataset. However, like all implementations of random forests, a change in the training dataset would mean retraining the random forest from scratch. Retraining from scratch would lead to a lot of repetitive computation; the aggregate composition of the dataset might not change significantly with the new batch, especially if each batch has far fewer data points than the overall dataset. This thesis presents work that allows random forest classifiers to be updated with each new batch of training data more efficiently, therefore saving time for data scientists and other users of Spark MLlib.

The rest of this thesis is organized as follows. Chapter two describes the existing literature in online and incremental random forests. Chapter three describes the strategies I implemented for incrementally training random forests. Chapter four contains performance and robustness metrics for each of the random forest implementations, as well as a discussion of their implications.

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 2

# Previous Work

Since Leo Breiman introduced random forests to the data science community in 2001 [11], researchers have tested various refinements of the algorithm. One area of focus has been augmenting the algorithm for online learning. This chapter describes the existing literature on incremental and online random forests.

## 2.1 Ensemble models

A random forest is a type of ensemble model, which averages the predictions of many different "reasonably good" models to produce a prediction that better estimates the true hypothesis. Ensemble models are highly successful as machine learning tools, because they avoid the chance-dependent pitfalls of many singular models. For example, gradient descent methods can get stuck in local minima, but combining many models increases the chance that one will find the global minimum. Alternatively, even if none of the models in the ensemble produce the true hypothesis, averaging every prediction can lead to a a prediction that more closely matches the underlying truth [3].

There are several established methods for aggregating model predictions in ensembles. The Bayesian voting algorithm iterates through all hypotheses produced by models in the ensemble, and then combines the results based on how likely the hypothesis is given the sample. Another method is to manipulate the training data via bagging, known as bootstrap aggregation, and boosting, known as weighted training. Bagging involves randomly sampling points with replacement from a common dataset; the default Spark implementation of random forests uses bagging over boosting. Other ensemble methods exist but are not utilized in Spark ML.

## 2.2 Online random forests

Standard random forests are offline classifiers; trees are built on static datasets. Offline classifiers are poorly suited for datasets for which additional data points become available incrementally. A prevalent example is logging; logs are published as actions take place within a system, and a classifier for these logs must take into account the new information to avoid becoming inaccurate. Similar problems requiring an online classifier are found throughout industry. For example, Yahoo! uses an online classifier to characterize relevant articles to show each user on its homepage. In many of these use cases, retraining the classifier from scratch would take an excessive amount of time, given the enormous amount of existing training data. Since new batches of data are often far smaller than the size of the aggregate training data, the new data only shifts a classifier's behavior slightly. Retraining an entire classifier from scratch to capture slight shifts seems wasteful; online classifiers provide a far more efficient approach.

In this thesis, I study incremental random forest classifiers–forests that update themselves with new batches of data. Incremental forests are a subset online random forests; they are updated with new batches of data, rather than with one new point at a time. Batched updates can take advantage of Spark's strength, which is distributed, batched computation. Simply implementing an online random forest classifier in Spark would be computationally wasteful, as it would require a new Spark job for every additional point. Focusing on batched updates plays to Spark's strengths and addresses a sparser area of the machine learning literature. Existing online methods are easily extended into batched methods, and batched computation provides an opportunity for optimizations.

### 2.2.1 Saffari online random forests

The most well-known implementation of an online random forest is the Saffari implementation. [10] In offline mode, a Saffari random forest behaves like a typical random forest: each tree is created by sampling the original training set, the split at each node is chosen as the best among a random set of feature candidates, and predictions are made by summing conditional probabilities among all of the trees.

In online mode, trees receive a serial string of points and are grown in an extremely randomized fashion; at every node, tests and thresholds are chosen randomly. Specifically, when a node is created, it establishes a set of N random tests and maintains statistics on the left and right partitions created by each test. When a new point is added to the tree, the point's features place it in a leaf node. The algorithm then recalculates the gain $\mathrm{G_n}$ with respect to each test in that node $n$ according to the following equation:

$$\mathrm{G_n} = \ell_n - \frac{|samples_l|}{|samples_n|} * \ell_l - \frac{|samples_r|}{|samples_n|} * \ell_r, \tag{2.1}$$

where $\ell$ indicates loss, $l$ represents the left partition of a split, and $r$ represents the right partition of a split. The Saffari algorithm splits a leaf based on two hyperparameters: $\alpha$, the minimum number of samples a node must see, and $\beta$, the minimum gain a split must achieve. When both conditions are satisfied, the node splits.

The Saffari algorithm also specifies that trees can be discarded randomly, where the probability that a tree is discarded increases with its out-of-bag error. This trait allows the random forest to adapt to changes in the data distribution.

Saffari claims this algorithm is better than alternatives such as the Hoeffding tree algorithm (addressed later in this chapter), because it fits better to the inherent nature of decision trees. Empirically, Saffari random forests perform better than boosted forests. In my research, I test the efficacy of using the extremely randomized trees of the Saffari algorithm. I also experiment with the aforementioned split hyperparameters, tweaking my algorithm to select more optimal gain and points thresholds. Since the Spark random forest implementation closely matches the Saffari offline random forest implementation, many of my initial modifications to the Spark code to provide online learning capability are in line with the Saffari online-mode algorithm.

### 2.2.2 Denil online random forests

In "Consistency of Online Random Forests," Misha Denil et al. proposes and evaluates improvements to the Saffari online random forest algorithm. Denil online forests partition the sequence of data points into "structure" points and "estimation" points [2]. Structure points influence the structure of the tree but do not affect the predictions made in tree leaves. Estimation points do not influence the structure of the tree, but are used to re-estimate probabilities. The Denil algorithm uses the same split selection procedure as the Saffari algorithm. The paper shows that this refined implementation achieves a higher accuracy on complex datasets compared to a comparable implementation of the Saffari algorithm.

Much of the existing literature around online random forests acknowledges one primary setback: due to decision trees recursive structure, lower data cannot be used to correct earlier decisions. Both the Saffari and the Denil implementations have this flaw. The next few subsections address algorithms that involve regrowing part or all of select decision trees within a random forest. My research draws on both classes of techniques–incremental growth and regeneration.

### 2.2.3 Mondrian forests

Another type of online random forest is Mondrian forests [7], which are comprised of augmented extremely randomized decision trees. The algorithm for building each Mondrian decision tree is as follows. We start at the root and allocate a budget, $\lambda$, for this node. Then, we recursively process each node by randomly choosing split locations on the ranges of feature values. For each point $j$, dimension $d$, and dimension-wise maximum and minimum $u_{jd}$ and $l_{jd}$, let $E = \sum_d(u_{jd} - l_{jd})$. Then, these cuts each cost $\lambda' = \lambda - E$. If we can "afford" the cut, we split the node and assign budget $\lambda'$ to each subinterval. The tree stops growing when no more cuts can be afforded at any leaf node. Throughout the growing process, we denote a split hierarchy using a time variable. A node's time variable $\tau$ is set when a split is made within that node; the value of this variable is equivalent to $E + \tau_{parent}$.

To adapt Mondrian trees for online learning, we use the time parameter to determine where a new cut should be inserted. Starting at the root, we recurse down the tree until the cost coefficient $E_{new}$ for this new node is less than the cost coefficient $E_{old}$ for some node. We then insert the new node as the parent of this old node and adjust all children nodes accordingly.

Mondrian forests achieve an accuracy very close to offline random forests and extremely randomized forests trained on the same fraction of the data. The Mondrian algorithm significantly outperforms the Saffari algorithm when trained in online mode on the same fraction of data. Furthermore, Mondrian forests were shown to adapt to new data an order of magnitude faster than simply regrowing a forest from scratch.

The drawbacks of Mondrian forests include its relative intolerance of irrelevant data. Because splits are random, irrelevant and relevant features are equally likely to be chosen; noisy features can then harm the overall accuracy of each tree. Additionally, a Mondrian forest in online mode might choose to insert a node close to a root, thus initiating recalculation of a large section of the tree.

While Mondrian forests outperform Saffari online random forests, regrowing a tree section requires passing through all of the data points seen so far. As datasets grow larger, the overhead incurred by additional passes through the data will outweigh the efficiency gains of the algorithm. Mondrian forests have only been shown as more efficient than Saffari online random forests on datasets of a few thousand data points. As such, the optimal strategy should regrow parts of a random forest but should not require multiple passes over the data.

## 2.3 Concept drift

Concept drift describes changes in data distribution in an online learning setting that cause the mathematical relationships between the input variables and output predictions to change. These shifts cause ML classifiers trained on earlier data points to become inaccurate. The online random forest algorithms discussed earlier in this chapter adapt poorly to concept drift, as they accommodate new data by splitting leaves or regrowing small sections of trees in the forest. Concept drift can cause splits in nodes higher in the trees to become inaccurate, and a poor decision higher in the tree more significantly impacts performance than a mistake closer to the bottom. Online random forests adapt more poorly than other online classifiers to concept drift, as splits made in nodes are essentially permanent. In contrast, classifiers that use, for example, linear or logistic regression could just shift internal weights until the concept drift is accounted for.

Therefore, accounting for concept drift in online random forests requires regrowing trees–the decisions from these trees must counteract bad decisions from other trees resulting from wrong splits. Purely incremental strategies, or those that just split leaves, should not be the only methods by which online random forests adapt to change.

## 2.4 Combined strategies

Several papers in the literature use Hoeffding trees to grow online random forests. Hoeffding trees for online learning were first proposed in a paper by Domingos and Hulten; when growing, these trees maintain several candidate splits in each leaf, with the quality of each split estimated in an online manner [4]. In contrast with the minimum gain parameter controlling splits in Denil and Saffari random forests, Hoeffding trees use a measure of the Hoeffding bound to ensure that a split is optimal. Hoeffding trees split leaves when the Hoeffding bound indicates that the current best split is the optimal split, within reasonable certainty.

A paper by Bifet et al. describes an implementation of the Hoeffding tree algorithm that adapts to concept drift [1]. The algorithm grows Hoeffding trees of different sizes; the authors reason that smaller trees can adapt more quickly to concept drift, whereas larger trees are less sensitive to noisy deviations. The trees are occasionally either partly or fully regrown, depending on a metric called ADWIN that estimates drift. The Bifet paper's results show that the method is effective on small generated datasets with concept drift.

Abdusalam et al. developed an algorithm that grows random forests incrementally by using Hoeffding trees and selecting entire trees for replacement. Unlike the algorithm from the paper by Bifet et al., the system chunks a data stream, then processes data chunks serially. The algorithm detects concept drift using a two-window technique; if a tree's classification error between the two windows differs by an amount less than a threshold, it is not grown further. Otherwise, grow the tree incrementally. With every given batch, 25% of the trees are automatically regrown, with additional trees regrown if the system detects concept drift.

I draw from these algorithms when developing my system in Spark. Namely, my implementation involves a balance between incremental growth and tree replacement, much like the algorithms in the Bifet and Abdusalam papers. I refer back to these previously-developed algorithms in my discussion of the Spark incremental random forests system developed in this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 3

# Methods

This chapter describes the two main strategies I implemented for incrementally training random forests. I go into depth about the details of each implementation, hybrid strategies, and the API I exposed for use in Spark ML pipelines. My experiments seek to shed insight into which algorithms work best on different workloads.

## 3.1 Incremental growth strategy

The first implemented strategy involves incrementally growing existing trees in the forest, and is based off of the Saffari and Denil algorithms. A user can update an existing incremental random forest classification model with a new batch of data via the exposed Spark API. To adapt the existing Spark implementation of a random forest into an implementation with incremental growth capabilities, I augment the existing tree node and random forest classification model classes to hold additional metadata. Specifically, the new incremental tree node class holds aggregate statistics on every point it has seen thus far. When a node receives a new point, it calculates, for every feature, the value "bin" (i.e. subset) into which that feature falls. The boundaries between bins, called splits, are chosen by the Spark MLlib library to capture all possible differences between data points. For example, an numerical variable would have possible splits between all values that the variable assumes within the dataset, and a categorical variable would have possible splits between all groupings of categories.

The existing bin-finding algorithm caps the maximum number of bins at around 1000, merging bins if the number of splits calculated by the algorithm exceed the ceiling. The aggregate statistics data structure for each node takes the form of an array that, for every feature bin, stores a count of the number of points with a feature that fell within the bounds of that bin. Similarly, the incremental random forest classification model is augmented with metadata containing the candidate splits for each feature from the first training run. These same splits are used to sort new batches of points into feature bins. As a result, the tree does not have to store and re-iterate through previous batches of points to search for new optimal splits.

The specific algorithm used to incrementally grow the random forests is as follows. Upon receiving a new batch, the random forest bins each point based on its feature values, with each point falling into one bin per feature. The random forest determines the boundaries of these bins using the stored splits metadata in the random forest classification model, as described previously in this section. For example, if a feature ranges in integer values from 1 to 10, then the splits metadata could contain a split value of 5.5, which would divide that feature distribution into two ranges: 1-5 and 6-10. The system constructs a data structure holding the feature bins for each point. A random sample of points is selected for consideration in each tree. Then, the system finds all of the leaf nodes in the random forest and loads these nodes onto a queue. For every node in the queue, the system determines the subset of new points that, when traversing the node's tree from the root, fall in that node. As described earlier in this section, each leaf node is augmented with metadata about the previous points captured by that node. A similar metadata structure storing aggregate feature bin counts is calculated for the new batch and then merged into the existing metadata stored within the node. Then, the system runs through all potential splits within the leaf and determines whether a split that decreases Gini impurity exists. If so, that split is made and the two new leaf nodes are loaded back onto the queue.

## 3.2 Tree regeneration strategy

The second implemented strategy involves regrowing trees within the random forest. Upon update, a set proportion of trees are randomly selected for replacement. The system then grows a random forest with the number of trees selected, creating a new hybrid model with the maintained trees from the old model and the newly grown trees from the new model.

## 3.3 Hybrid strategies

The metadata required for incremental growth is still maintained with the tree regeneration strategy. As a result, my implementation can intermix tree regeneration with incremental growth, meaning that a tree that is regrown after one batch can be incrementally grown after the next, since regrowing the tree still maintains the necessary leaf metadata do detect future node splits. To create hybrid strategies, I select a proportion of trees without replacement to be regrown, and then select another proportion of trees to be grown incrementally. So, after the random forest receives a new batch, each tree can either be regrown or incrementally grown by the algorithm.

To create many different hybrid approaches, I vary the proportion of trees that are grown with each strategy. Those not selected for regeneration or incremental growth are left unchanged. I hypothesize that the inherent characteristics of each data workload will affect the optimal hybrid strategy for an incremental random forest classifier on my test datasets. This thesis will test this hypothesis by exploring the performance of these various hybrid strategies in chapter 4.

## 3.4 Optimizations

In this section, I detail the optimizations I made to my implementation of the incremental growth and tree replacement strategies. These optimizations are additions to the memory and speed optimizations already existing in ordinary Spark random forest classifiers. An initial comparison of the classification performance increase due to these optimizations revealed a 0-2% increase in accuracy, depending on the tested dataset.

### 3.4.1 Bounding incremental growth

Since datasets can have many features, unbounded tree growth can result in extremely deep trees after just a few batches. The largest tree height supported by Spark ML is 30, after which trees can no longer be grown incrementally. Additionally, splitting leaves in tall trees imposes a large performance overhead, as the algorithm must examine a large number of candidate leaves. As a result, the system restricts the maximum initial tree height to a set value $h$, relaxing this restriction by one for each incrementally grown tree that receives a new batch of data. In all hybrid strategies, a tree is replaced when it reaches a certain threshold depth. When a tree is regrown, its maximum height is reset to the original value $h$. This technique prevents the incremental random forest from overfitting to earlier batches and allows the forest to better adapt to concept drift, as it ensures that later batches can significantly impact tree predictions.

### 3.4.2 Batched processing

The incremental random forest implementations in the Saffari and Denil papers describe accumulating points in leaves until the leaf is split. In the purely online setting, accumulating points one at a time is the only option. Since batched workloads allow for many points to be processed at one time, we can preprocess the batch into metadata structures that are passed to the relevant leaves and merged with the existing leaf metadata. This optimization prevents the system from having to run back over all previous batches to decide on a split.

### 3.4.3 Tree reweighting

In the batched incremental learning setting, the classifier receives an unlabeled batch of points, predicts the label for each point, and then views the labels to assess its accuracy. Each tree in the random forest has a different accuracy on a particular new batch of data; if there is concept drift, newer trees likely have better classification performance than older trees. To account for this concept drift, I weight trees by age. With every new batch, trees that are not regrown are weighted to have less voting power. As a result, the classifier captures information from previous batches, but uses information from the most recent batch more heavily in the classification of the next batch.

## 3.5 API

My custom incremental random forest classifier class, IncrementalRandomForestClassifier, exposes a Scala API for data scientists to use. I introduce a new model class, the IncrementalRandomForestClassificationModel class, which is an augmented adaptation of the Spark ML RandomForestClassificationModel class that allows for incremental optimizations. Table 3.1 details the methods available for use in Spark ML pipelines. The code for my implementation of incremental random forest classifiers has been open-sourced on Github.

| **Method** | **Description** |
|---|---|
| def train(df: DataFrame): IncrementalRFClassificationModel | Trains a model for future online learning by maintaining tree and leaf metadata. |
| def update( old: IncrementalRFClassificationModel, df: DataFrame): IncrementalRFClassificationModel | Updates an existing model with a new batch of data by regrowing some trees and incrementally growing others. |
| def addTrees( oldModel: IncrementalRFClassificationModel, df: DataFrame, addedTrees: Int): IncrementalRFClassificationModel | Trains a model using a warm start by adding more trees to the existing forest. |
| def setRegrowProportion(prop: Int) | Set the proportion of trees in the forest that should be regrown with every new batch. |
| def setIncrementalProportion(prop: Int) | Set the proportion of trees in the forest that should be regrown with every new batch. |
| def setInitialMaxDepth(depth: Int) | Set the initial depth of newly-grown trees. |

Table 3.1: Spark API for the incremental random forest (RF) classifier developed in this thesis.

# Chapter 4

# Results

In this chapter, I analyze the benchmark results and discuss their implications for data scientists using incremental random forest classifiers. In each trial, each incremental random forest classifier received batches sequentially, retraining itself after every new batch. For each tested workload, I first contrast sample training times of several incremental growth and tree replacement strategies. Then, I analyze a range of hybrid strategies and explore the ideal hybrid strategy for each workload based on concept drift and batch size.

## 4.1 Experimental setup

The tree regeneration and incremental growth strategies are compatible; both can be used simultaneously within an incremental random forest. This thesis explores various hybrid approaches and provides insight into which approaches work best on datasets with and without concept drift. For each hybrid approach, I vary the proportion of trees that are grown with each strategy. The system selects trees without replacement to be replaced or grown incrementally; the remaining trees are unchanged.

In my research, I experiment with several different parameters that affect the error rate of my incremental random forest classifier. I hypothesize that certain parameters used to build each incremental random forest will affect the optimal incremental strategy for that forest on my test datasets. I vary the following parameters in my experiments and report the effect on the optimal hybrid incremental algorithm.

1. **Tree depth.** I vary the initial maximum height of each tree in the random forest. In my experiments, I use two initial heights: shallow and deep, which represent heights of 5 and 10, respectively. I chose 10 to be the sample height for deep trees, as it is the most commonly used height in the literature to obtain a high accuracy without overfitting. A shallow tree height of 5 still provides some accuracy, but allows the tree to adapt more significantly to concept drift in incremental batches.

2. **Concept drift.** I test the performance of the incremental random forest classifier on four datasets: two with concept drift and two without concept drift. The datasets with concept drift show seasonal changes over the course of many months, whereas the datasets without concept drift are randomized as to ensure that there is no significant shift in data distribution.

3. **Batch size.** I vary the batch size of the data workloads to test the impact of batch size on classifier performance and optimal strategy. Specifically, I select 100 as the "small" batch size and 2,000 as the "large" batch size.

4. **Hybrid strategy.** I vary the proportion of trees grown incrementally between 0.0 and 1.0, using intermediate steps of 0.1. I also vary the proportion of trees regrown after each batch from 0.0 to 1.0 with step size 0.1. Trees are selected to be regrown, grown incrementally, or unchanged, depending on the hybrid strategy. One tree cannot be selected to be grown by more than one strategy after any particular batch. Consider, for example, a hybrid strategy with a 0.1 proportion of trees regenerated, a 0.2 proportion of trees grown incrementally, and 100 trees in the forest. After receiving a batch, 10 trees are first selected without replacement to be regrown. Then, another 20 trees are selected among the remaining trees to be incrementally grown. Since trees are selected without replacement, the sum of the two proportions cannot exceed 1.0.

Through my experiments, I seek to diagnose the optimal hybrid approach for each

| Workload | Batch size | Concept Drift |
|---|---|---|
| Homesite Quote Conversion | large | no |
| Otto Group Product Classification | small | no |
| Airline Delay Causes | large | yes |
| Bike Sharing | small | yes |

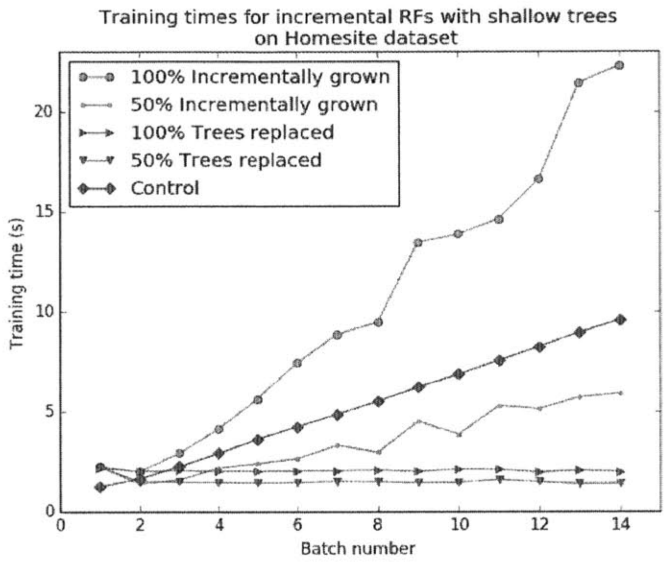
Table 4.1: I run experiments on the four example workloads listed in this table. These workloads span the combinations of small and large batch size with existent or nonexistent concept drift.

of four data workloads, shown in Table 4.1. To evaluate the efficiency and accuracy of a classifier, I measure the training time and predictive error rate on the next batch. I compare the metrics taken on random forest classifiers running a variety of incremental strategies to the control setting, which involves regrowing the entire random forest from scratch on all batches previously seen. The results chapter of this thesis discusses the results for each workload separately, and then explains an overarching method for selecting a general hybrid strategy for a new workload.

## 4.2 Workload A: Large batches, no concept drift

The Homesite Quote Conversion dataset is a Kaggle dataset that provides over fifty numerical and categorical metrics on each of 200,000 customers. The classification goal is to determine whether a potential customer will purchase home insurance given the provided metrics about their quoted price, previous activity, coverage information, and more [5]. To study how incremental random forests would perform on workloads with large batch sizes and no concept drift, I randomly divided the Homesite dataset into fifteen batches of 2,000 data points each. Then, I ran tests using both deep and shallow random forests, measuring the training time required by each sample strategy to accommodate each new batch of data.

As seen in Figure 4-1, retraining the entire tree from scratch on all data caused the training time for the control setting to be much higher on later batches than the training time for the experimental settings, with one notable exception. When all trees are grown incrementally after each batch, the training time is higher than the



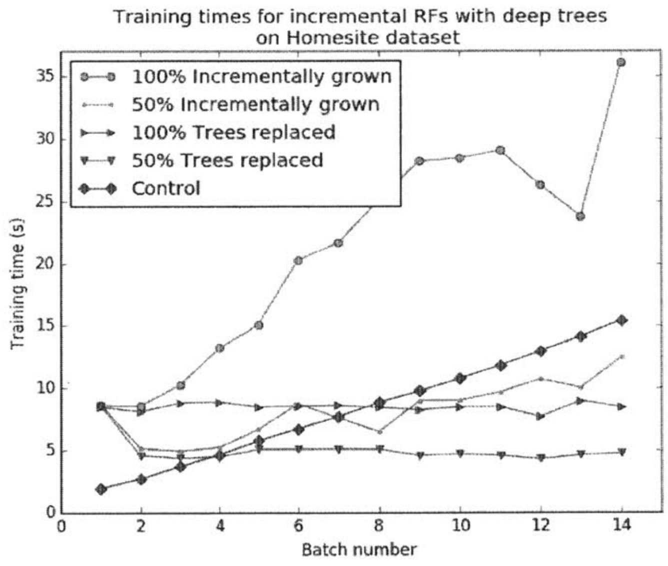


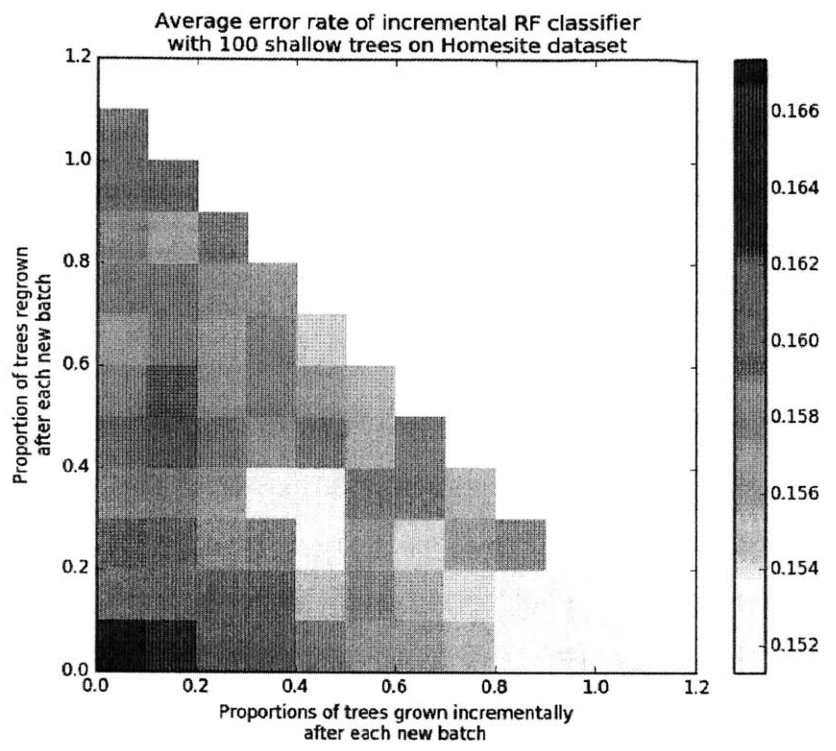
Figure 4-1: These graphs show the training times for the incremental growth strategy, the tree replacement strategy, and the control setting on the batched Homesite Quote Conversion dataset.

control. This occurs because every full added level of nodes doubles the number of leaves in the tree. As a result, the next incremental training step must examine twice as many leaves for potential new splits. While Spark random forests mitigate this exponential growth by batch processing leaves, the results still reveal the significant overhead. As a result, my findings indicate that no more than around 50% of trees should be incrementally grown after every batch if training time is a primary concern. These findings are consistent in both random forests with deep trees and those with shallow trees.

In contrast to the incremental growth strategy and the control, the tree replacement strategy shows similar training times across all batches. This occurs because the system grows the same number of trees on the same quantity of data after each timestep. While a data scientist would likely not wish to regrow 100% of trees, I consider the training times for a 100% tree replacement strategy to demonstrate the worst-case training time for a random forest classifier using this strategy. Even when the forest is entirely regrown on each new batch, the training time eventually drops below the monotonically increasing control training times. As seen, the random replacement strategy will always be more efficient than the control after a certain batch number on workloads such as the batched Homesite Quote Conversion dataset.

Figure 4-2 shows the results of measuring the average predictive error rates of 50 different hybrid incremental random forests. In random forests with shallow trees, hybrid strategies with a high proportion of trees incrementally grown performed better than other strategies. In general, using shallow trees should favor incremental strategies, as shallow trees are better able adapt to new information in batches than deep trees. Since forests with deep trees are generally more accurate than those with shallow trees, a data scientist would likely use shallow trees if training time was a primary concern. Considering the training time data shown in Figure 4-1, the optimal strategy for this setting seems to be regrowing approximately 40% of trees and incrementally growing another 40%.

In random forests with deep trees, hybrid strategies with a nonzero proportion of trees regrown after every new batch show the lowest error rate. This trend reveals that



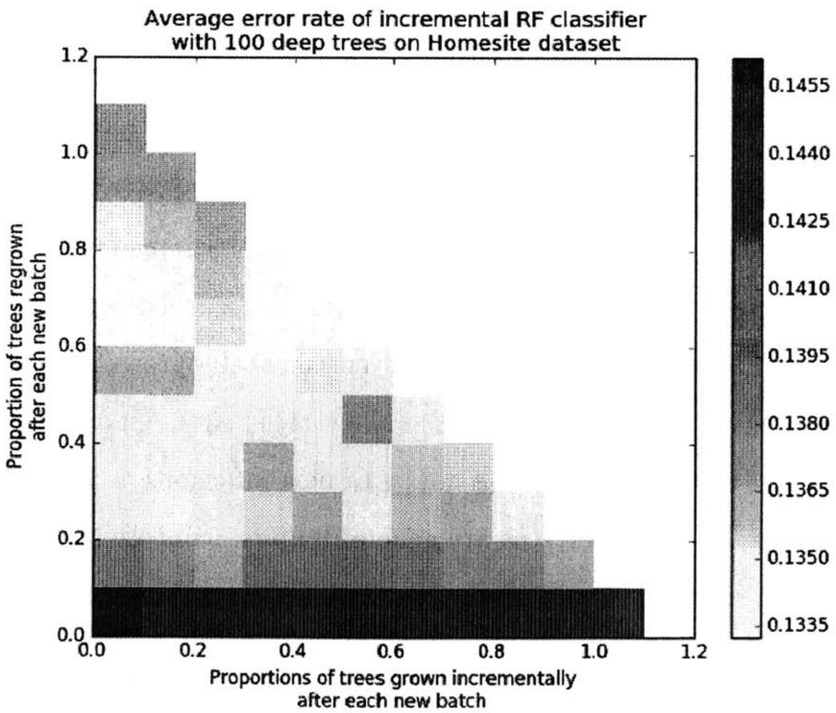


Figure 4-2: These two plots show the average error rates of various hybrid tree replacement and incremental growth strategies on the Homesite batched dataset. The axes indicate the percentage of trees that are modified according to each strategy.

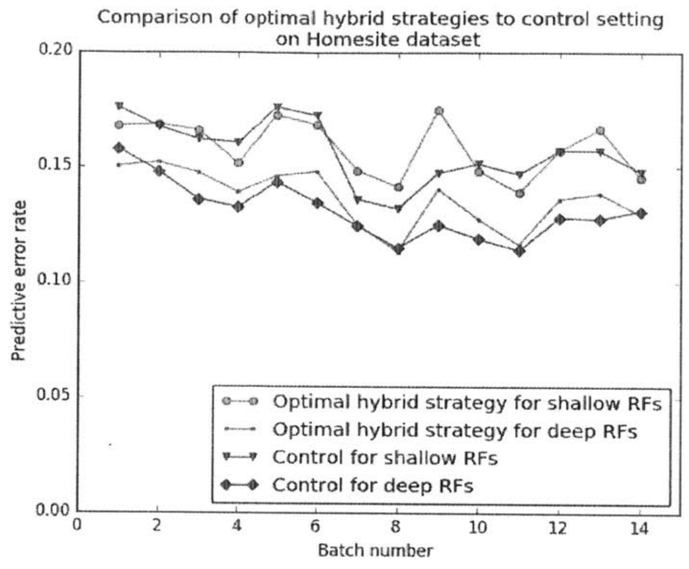


Figure 4-3: This graph compares the error rates of the control setting to the error rates of the optimal hybrid strategy for the Homesite dataset.

deep decision trees poorly accommodate additional information from later batches. While these deep forests with purely incremental strategies have lower error rates than shallow forests, they perform significantly worse than all other hybrid strategies. Therefore, the optimal strategy for this setting seems to be to regrow around 30% of trees with each new batch. This strategy minimizes both training time and predictive error rate.

Figure 4-3 compares each optimal hybrid strategy to the control across all batches of the workload. Comparing these optimal strategies to the control implementation, which regrows the entire forest on the cumulative dataset, I find that the experimental settings obtain error rates roughly equivalent to the control's.

## 4.3 Workload B: Small batches, no concept drift

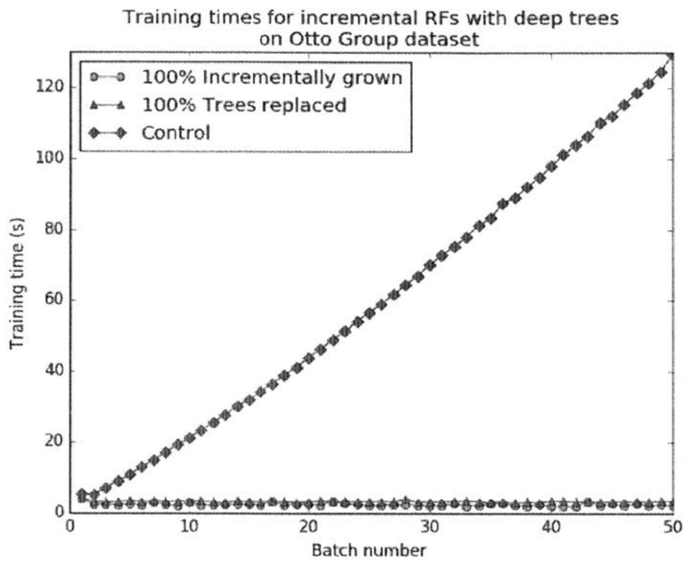
The Otto Group Product Classification dataset contains 200,000 data points representing products sold by the Otto Group. Each data point is characterized by nearly 100 numerical features representing qualities of each product; the classification task is to distinguish one particular product category, "Class 2," from the others. By randomly downsampling the dataset, I segmented the data into batched workloads of approximately 100 points per batch, much smaller than the 2,000-point batches of the Homesite dataset [6]. My initial analysis, as with the Homesite dataset, involved contrasting the training times of the two incremental random forest strategies on the data. I trained random forest classifiers using each of two strategies on fifty 100-point batches of the Otto dataset to demonstrate any trends over time.

As seen in Figures 4-4,the control demonstrates the largest training time after each batch, as it must be retrained from scratch on the aggregate data. In comparison, the training times for the experimental strategies are far smaller, even when every tree in the forest is modified after each batch. This trend is consistent in both random forests with deep trees and those with shallow trees. With small batches, the number of leaves that will be incrementally split will also be small, accounting for the low training time of the incremental strategy. This differs from the results of the Homesite dataset, where a large batch size caused a larger number of new splits, resulting in a distinct increase in training time.

Figure 4-5 displays the predictive error rate of various hybrid incremental random forest classifiers. As shown, for smaller batch sizes such as 100, strategies that involve only incremental growth generally perform better than strategies that incorporate some proportion of regrown trees. This trend is present in both random forest classifiers comprised of deep trees and those with shallow trees, though the trend is stronger among forests with shallow trees.

Trees grown on one batch naturally overfit to that batch. With a small batch size, there is a higher likelihood that a batch's data distribution differs from the overall data distribution of the full dataset. As such, tree replacement strategies could grow trees that fit well to one batch, but that fit poorly to the rest of the data, increasing the error rate. Incremental growth incorporates the data from multiple batches into each tree, mitigating the effect of overfitting to the wrong data distribution.

These results demonstrate that data scientists using incremental random forests on workloads with small data batches should utilize a strategy that only involves incremental growth. Figure 4-5 indicates that the specific proportion of trees that



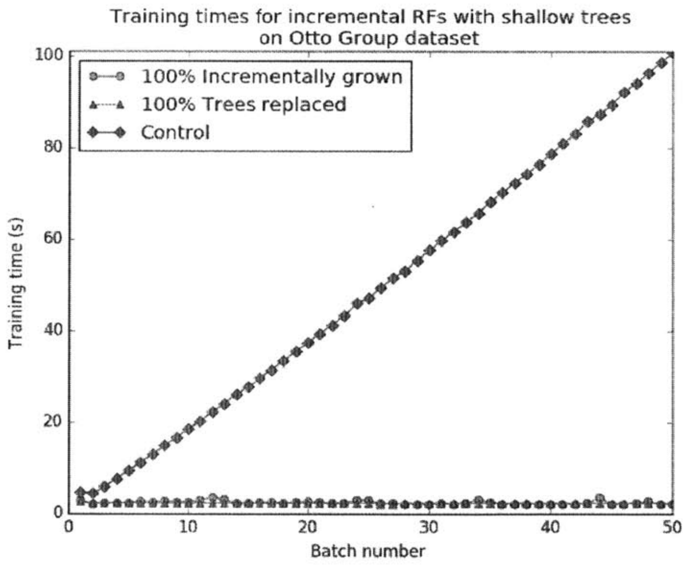
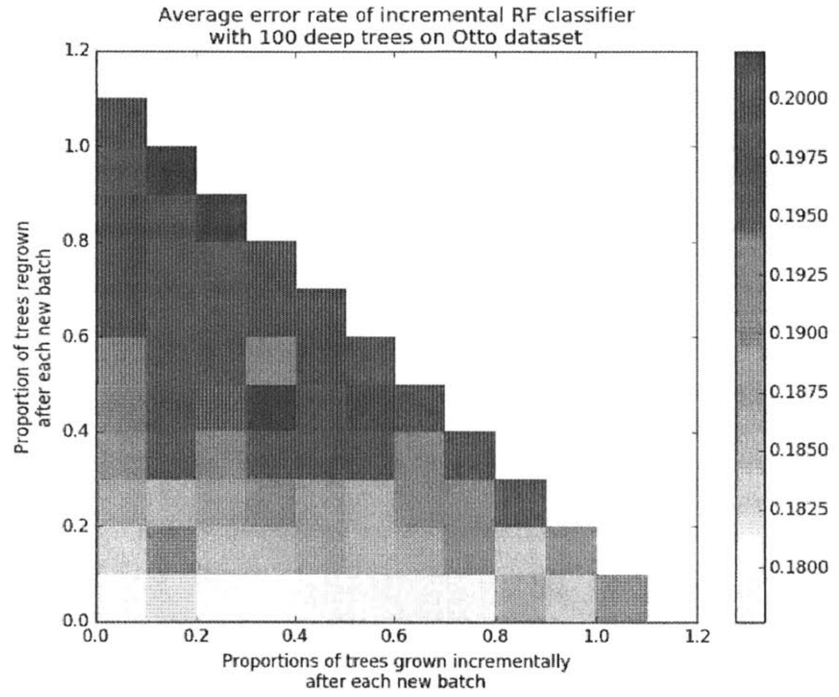


Figure 4-4: These graphs show the training times for the incremental growth strategy, the tree replacement strategy, and the control setting on the batched Otto Group Product Classification dataset.



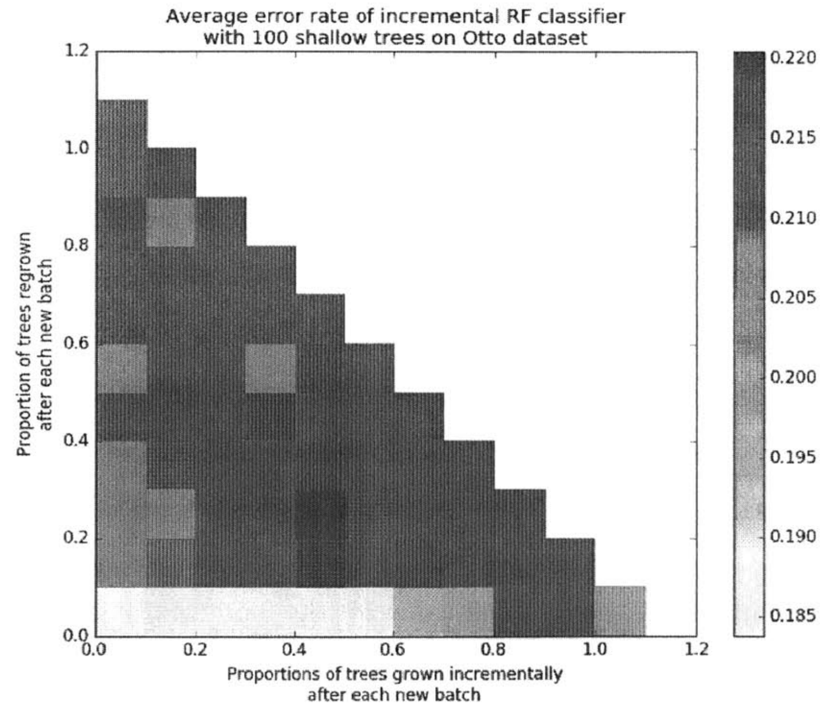


Figure 4-5: These two plots show the average error rates of various hybrid tree replacement and incremental growth strategies on the Otto Group batched dataset. The axes indicate the percentage of trees that are modified according to each strategy.

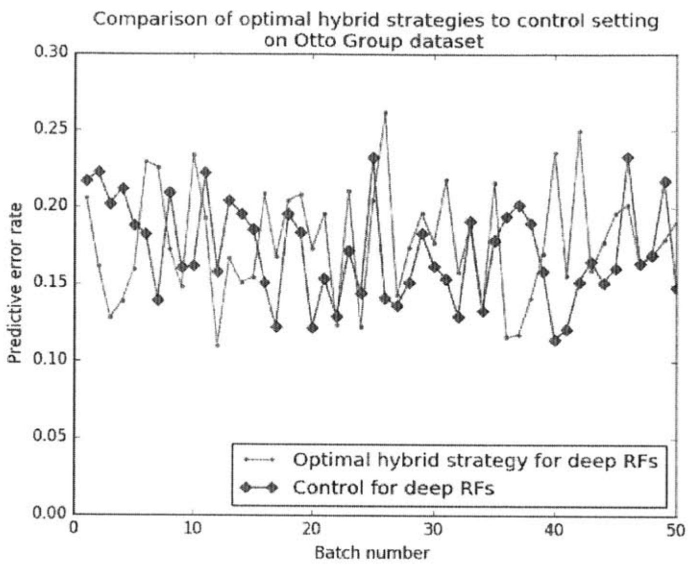


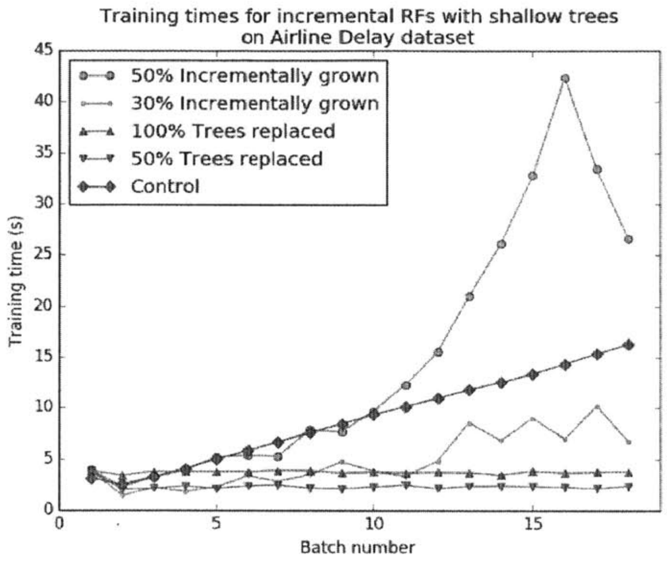
Figure 4-6: This graph compares the error rates of the control setting to the error rates of the optimal hybrid strategy for the Otto Group dataset. I show a comparison of the performance of forests with deep trees; the results for forests with shallow trees look similar.

are incrementally grown can vary with little impact on accuracy, so data scientists should choose a smaller ratio to minimize training time.

I compare the optimal hybrids to the control in Figure 4-6, finding that the experimental settings obtain error rates indistinguishable to the control's across all batches. Given the similar error rates, the significantly smaller training times of the hybrid strategy indicate that incremental classifiers are more optimal than the control for workloads such as the Otto Group batched dataset.

## 4.4 Workload C: Large batches, concept drift

The US Department of Transportation Airline On-Time Statistics and Delay Causes dataset contains information about every flight that took place in the last decade. I used 18-months of this dataset, beginning in January 2014, for my analysis. Because the distribution of airline delay data varies seasonally, this dataset exhibits concept drift. Each month of data translated to a 10,000-point batch in an 18-batch data workload [8].



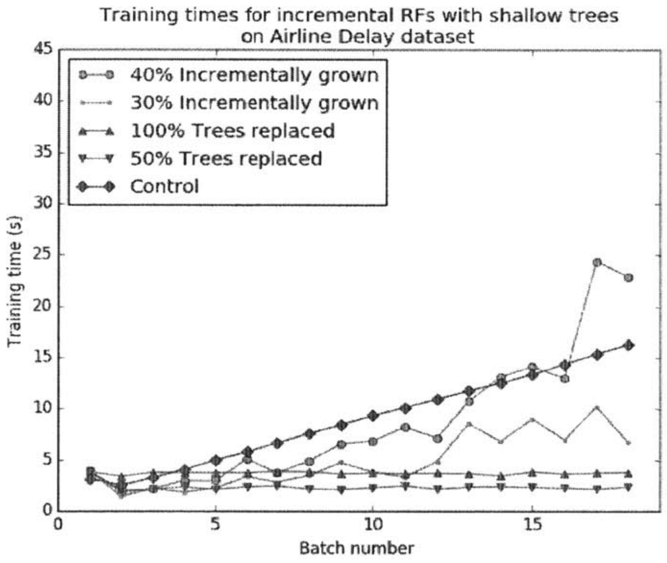


Figure 4-7: These graphs show the training times for the incremental growth strategy, the tree replacement strategy, and the control setting on the batched Airline Delay dataset.

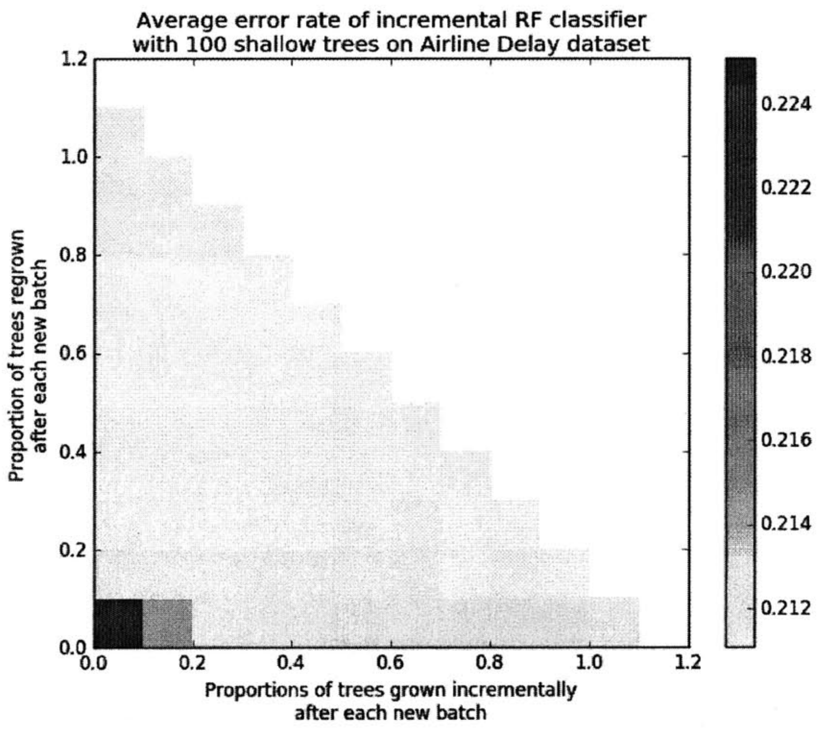
In Figure 4-7, I contrast the training times of several incremental random forest classifiers using a variety of pure strategies. In the first graph, I show the training times for the classifiers incrementally growing 50% and 40% of trees, as all training time progressions for classifiers incrementally growing a higher percentage rapidly increase to a time significantly larger than the control. If the percentage of trees incrementally grown is 40% or lower, the classifier shows a lesser training time than the control.

The classifier performance results from the Homesite dataset are similar these Airline Delay dataset results. In both, incrementally growing every tree in the random forest quickly becomes inefficient compared to the control. The effect is augmented when the dataset has concept drift, since more leaves will likely be split to accommodate the shifts in data distribution. As such, data scientists using batched incremental random forests on data workloads with a large batch size should not incrementally grow a large proportion of trees if efficiency is a major concern.

Figure 4-8 demonstrates a distinct trend—incremental random forest classifiers applied to workloads with large batch sizes and concept drift perform far more poorly when using a purely incremental strategy than when using any other strategy. This trend is more pronounced in forests with deep trees. Again, the results show that deep trees have a harder time adapting to shifts in data distribution than shallow trees. In a dataset with concept drift, such as the Airline Delay dataset, this ability to adapt becomes a larger factor in classification accuracy. The effect is mitigated in shallow trees when a percentage of trees greater than 20% is grown incrementally after every batch.

The results in Figure 4-7 and 4-8 indicate that data scientists using incremental random forest classifiers on workloads with large batches and concept drift should regrow around 20% of the trees in the forest after each batch. This strategy minimizes predictive error while also maintaining a low incremental training time.

Figure 4-9 compares these optimal hybrids and the control, showing that the experimental settings have essentially identical error rates to the controls. Given the similar error rates, the smaller training times of the optimal hybrid strategies indi-



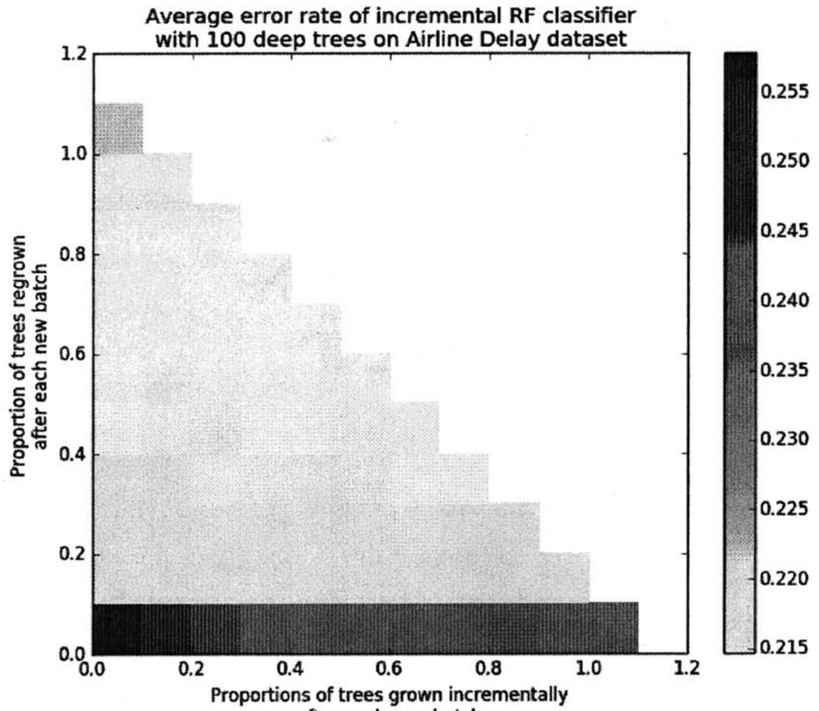


Figure 4-8: These two plots show the average error rates of various hybrid tree replacement and incremental growth strategies on the Airline Delay batched dataset. The axes indicate the percentage of trees that are modified according to each strategy.

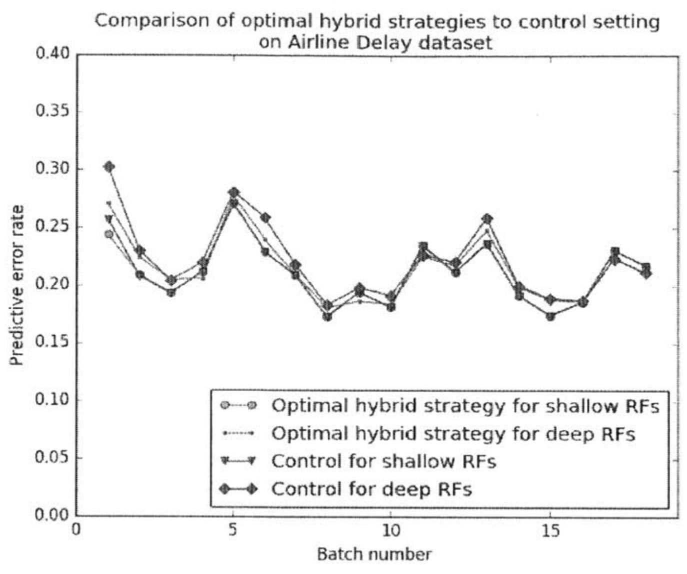


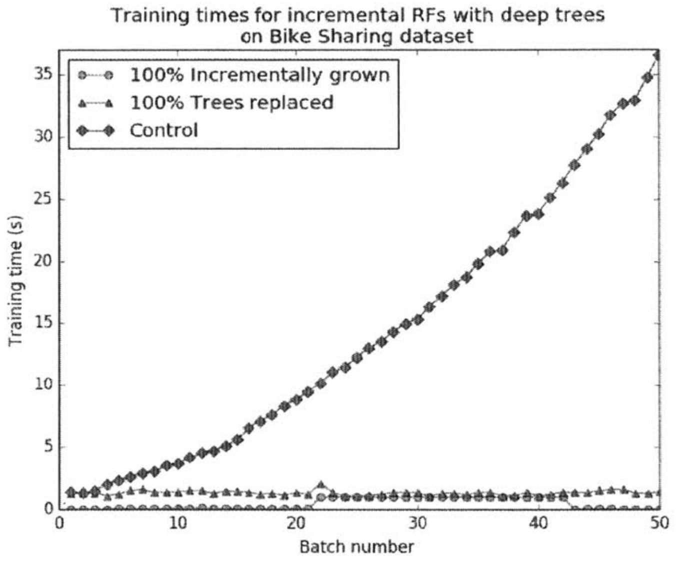
Figure 4-9: This graph compares the error rates of the control setting to the error rates of the optimal hybrid strategy for the Airline Delay dataset.

cate that incremental classifiers provide a more time-efficient alternative to ordinary random forest classifiers on this workload.

## 4.5 Workload D: Small batches, concept drift

The UCI Bike Sharing Dataset captures the number of rental bikes used hourly in the Capital bikeshare system over two years. The dataset provides general information about the temperature, weather, and other aspects of a particular day; the classification task is whether the number of used rental bikes exceeds 120 in a given hour. Over the course of a year, the data distribution of the Bike Sharing dataset shifts due to seasonal changes; fewer individuals generally ride bikes in the winter, whereas the summer sees a significant spike in rentals [9].

Figure 4-10 shows that the training time for the control increases dramatically as it accumulates tens of batches. These results are similar to the results from the Otto Group dataset, showing that when the batch size is small, incremental strategies allow random forest classifiers to register and adapt to new data far more quickly than ordinary random forests. Therefore, if time is a primary concern, incremental random



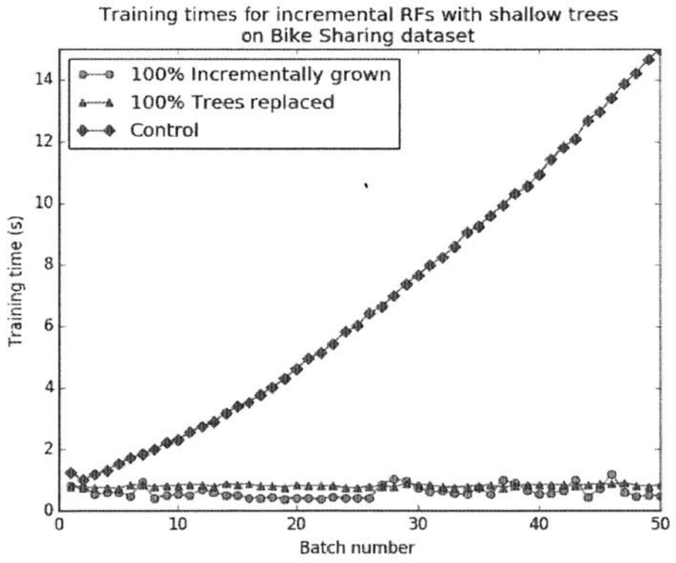
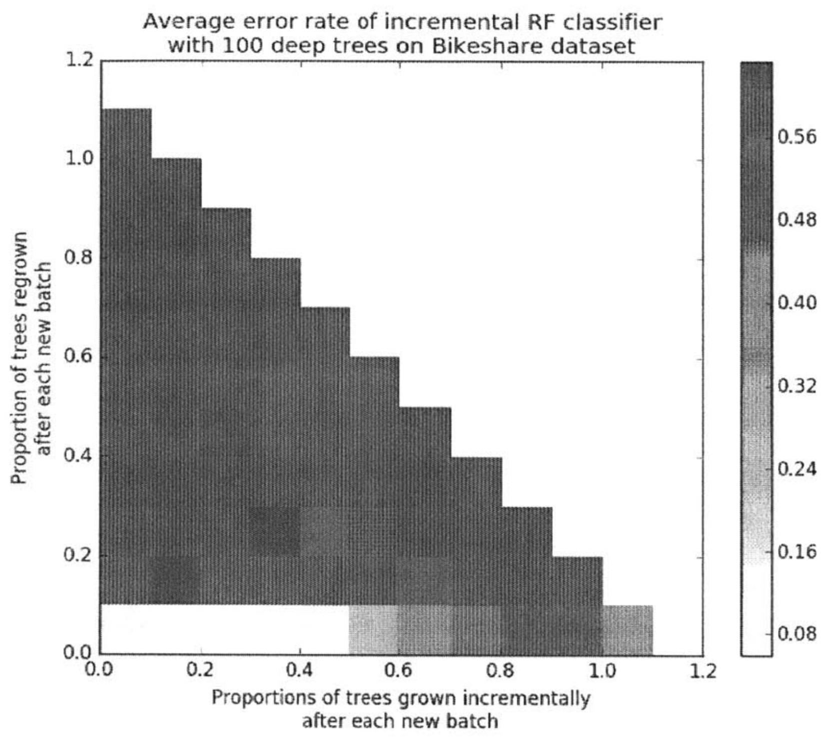


Figure 4-10: These graphs show the training times for the incremental growth strategy, the tree replacement strategy, and the control setting on the batched Homesite Quote Conversion dataset.



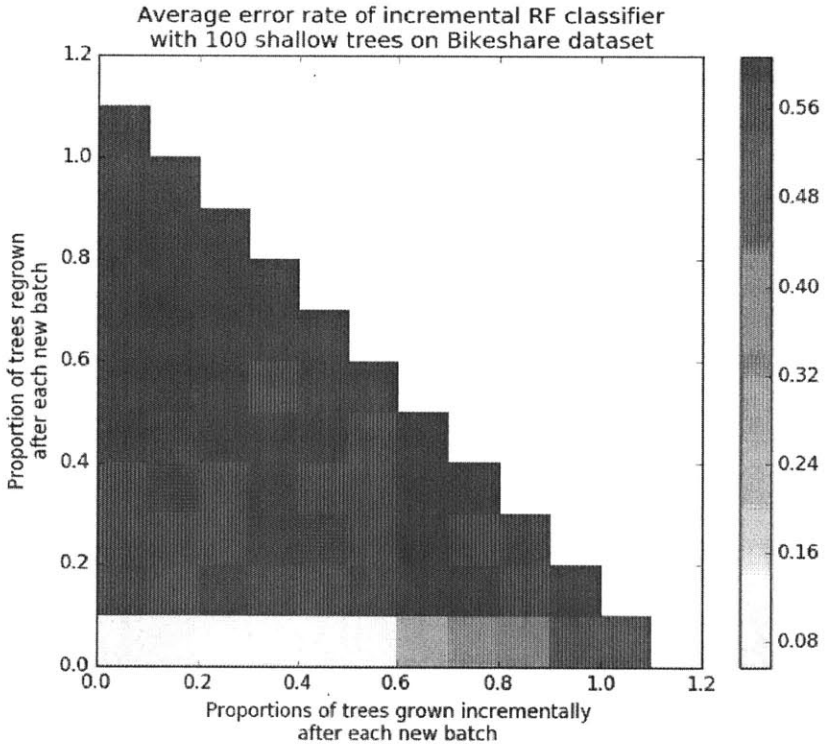


Figure 4-11: These two plots show the average error rates of various hybrid tree replacement and incremental growth strategies on the Bike Sharing batched dataset. The axes indicate the percentage of trees that are modified according to each strategy. As seen, the best strategy is to never regrow trees, and to always grow them incrementally.

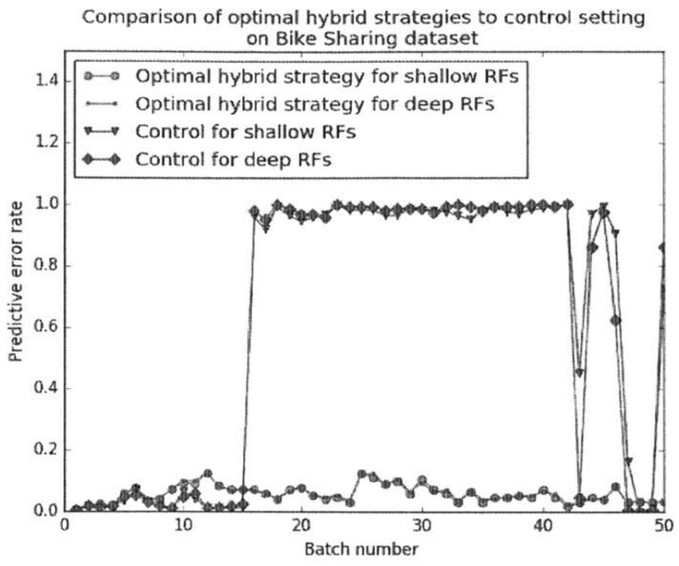


Figure 4-12: This graph compares the error rates of the control setting to the error rates of the optimal hybrid strategy for the Airline Delay dataset.

forests become more desirable when the batch size of a workload is small.

Figure 4-11 demonstrates that pure incremental growth strategies are far more accurate than any other strategy on this workload. The bikeshare dataset exhibits extreme concept shift; very few individuals rent bikes in the middle of winter, whereas the number of rentals over the summer nearly always passes the threshold of 120. However, since the dataset is cyclical, a classifier that shifts too signficantly to fit, for example, the summer trend will perform very poorly on data batches from the subsequent winter months. The results reveal this trend is consistent in both incremental random forests with deep trees and those with shallow trees. As a result, data scientists should use a pure incremental growth strategy when using incremental random forests on a data workload with small batches and large concept drift.

Figure 4-12 compares the hybrid strategies with the lowest error rates to the control. In this case, the control performs exceptionally poorly. The Bike Sharing dataset is batched by date, so it exhibits dramatic concept drift resulting from seasonal shifts. The favored incremental strategy consists of incrementally growing a small proportion of the trees, which both takes into account information from many batches and down-weights previous batches. In contrast, the control weights all previous data equally, so is less adaptible to rapid and large concept drift. As a result, the incremental strategy is far preferable to the control for classifying workloads such as the Bike Sharing batched dataset.

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 5

# Conclusion

In this thesis, I examined four datasets that differ in batch size and presence of concept drift, and determined the optimal incremental strategy for each dataset. While the dominant predictor of strategy was batch size, the presence of concept drift also affected the optimal incremental strategy.

Figure 5-1 captures the observed trends. For most datasets, with both time and error rate taken into account, a pure incremental growth or pure tree regeneration strategy seemed optimal. The most significant trend was that incremental strategies were optimal with small batch sizes, and that tree regrowth strategies were optimal with large batch sizes. Moreover, my results show that not all trees must be modified with each batch for the random forest to adapt to new information. In many cases, incrementally growing or regenerating too many trees led to overfitting on a particular batch, and consequently led to a high predictive error rate. Since regenerating or incrementally growing a larger percentage of trees also increases training time, modifying a smaller percentage of trees—between 20 and 50%—is optimal.

Overall, this thesis presented the two main strategies for growing random forest classifiers incrementally, and compared them over four differing datasets: the Homesite, Otto Group, Airline Delay, and Bike Sharing datasets. The purpose of this analysis was to provide insights into the behavior of incremental random forests that data scientists can generalize to other datasets. Figure 5-1 can stand as a reference as to what strategies are likely to be optimal for other workloads, as long as the batch

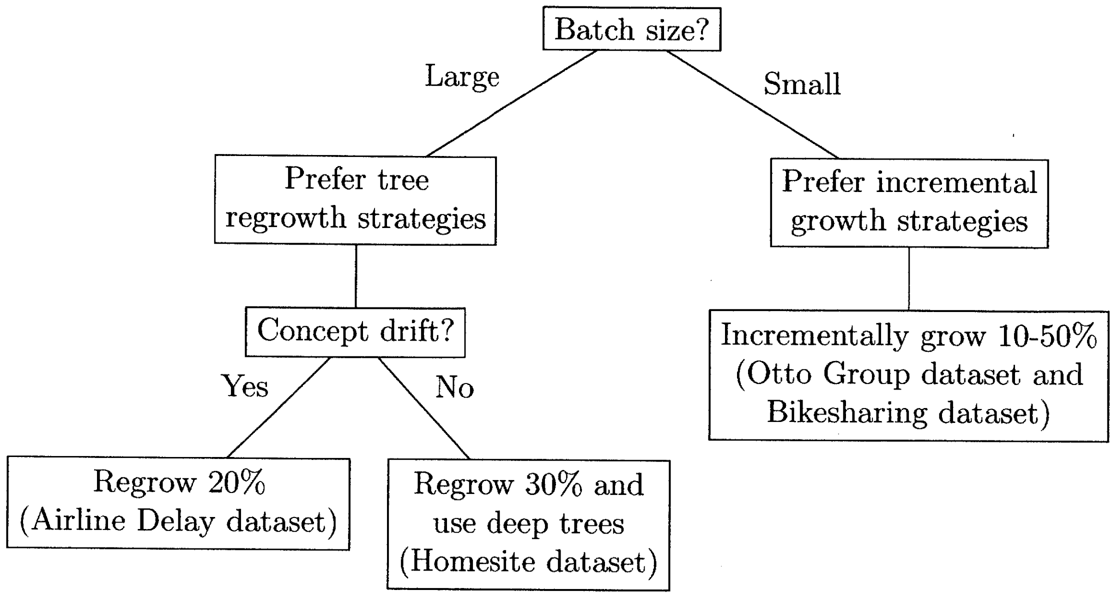

Batch size?
Large
Small
Prefer tree regrowth strategies
Prefer incremental growth strategies
Concept drift?
Yes
No
Incrementally grow 10-50% (Otto Group dataset and Bikesharing dataset)
Regrow 20% (Airline Delay dataset)
Regrow 30% and use deep trees (Homesite dataset)

Figure 5-1: This tree summarizes my results for the four tested datasets, and provides loose rules on what incremental strategy is likely optimal for other datasets

size and degree of concept drift are known.

Moreover, I provide a novel implementation of these two strategies in Spark Machine Learning (ML). My implementation is different from existing incremental random forest libraries, as it runs within Spark and handles data in batches. Other implementations, such as those from the Saffari and Denil papers, process each point individually, which would be inefficient in Spark. As such, my implementation takes advantage of several optimizations only possible when processing batched workloads. Overall, my system demonstrates significant efficiency gains when compared to the standard method of entirely regrowing random forests. I open sourced my implementation of Spark incremental random forests at

https://github.com/kathrynsiegel/incremental-random-forest

so that others may use it in the future.

# Bibliography

[1] Albert Bifet, Geoff Holmes, Bernhard Pfahringer, Richard Kirkby, and Ricard Gavaldà. New ensemble methods for evolving data streams. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 139–148, New York, NY, USA, 2009. ACM.

[2] Misha Denil, David Matheson, and Nando de Freitas. Consistency of online random forests. In *ICML (3)*, volume 28 of *JMLR Proceedings*, pages 1256–1264. JMLR.org, 2013.

[3] Thomas G. Dietterich. Ensemble methods in machine learning. In *Proceedings of the First International Workshop on Multiple Classifier Systems*, MCS '00, pages 1–15, London, UK, UK, 2000. Springer-Verlag.

[4] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pages 71–80, New York, NY, USA, 2000. ACM.

[5] Kaggle. Homesite quote conversion. Accessed: 2016-04-14.

[6] Kaggle. Otto group product classification challenge. Accessed: 2016-04-28.

[7] Balaji Lakshminarayanan, Daniel M. Roy, and Yee W. Teh. Mondrian forests: Efficient online random forests. In Z. Ghahramani, M. Welling, C. Cortes, N.d. Lawrence, and K.q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3140–3148. Curran Associates, Inc., 2014.

[8] US Department of Transportation. Airline on-time statistics and delay causes, 2015.

[9] UCI Machine Learning Repository. Bike sharing dataset data set. Accessed: 2016-05-04.

[10] Amir Saffari, Christian Leistner, Jakob Santner, Martin Godec, and Horst Bischof. On-line random forests.

[11] Leo Breiman Statistics and Leo Breiman. Random forests. pages 5–32, 2001.