UNIVERSITY OF CALIFORNIA SAN DIEGO

Parallel Boosting and Learning from Diverse Datasets

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Julaiti Alafate

Committee in charge:

Professor Yoav Freund, Chair
Professor Sanjoy Dasgupta
Professor Arun Kumar
Professor Julian McAuley
Professor David T. Sandwell

2020

The dissertation of Julaiti Alafate is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California San Diego

2020

EPIGRAPH

The only real validation of a statistical analysis, or of any scientific enquiry, is confirmation by independent observations.

—Frank Anscombe

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

My advisor Yoav Freund was the primary reason I started my PhD. He has always made himself available to give advice and encouragement. His passion, curiosity, and hardworking spirit constantly inspire me. My gratitude is beyond expression for all his guidance and support.

I much appreciate the support of my committee. In particular, David Sandwell has been tremendously supportive with the bathymetry data editing project that is part of this thesis. Arun Kumar offered me many suggestions during the revisions of the parallel boosting paper.

I am grateful to many other faculty members of the Department of Computer Science and Engineering (CSE) with whom I have been fortunate to discuss research ideas, and sometimes, life lessons. Yannis Papakonstantinou supported me during the first two years of the graduate school while I was still exploring research ideas. Alex Orailoglu has been my source of wisdom on research and life since the day I knew him (in the thesis defense of a fellow graduate student). The conversations I had with Alex in his office and at the Faculty Club would be among the best memories of my time at UCSD.

Part of my research work is through the collaboration with the researchers at the Scripps Institution of Oceanography. Especially, I am thankful to Brook Tozer. Without his work, finishing the bathymetry editing project would not be possible.

I would like to thank my undergraduate advisor, Zhongzhi Zhang, who introduced me to research, and encouraged me to apply to graduate school. I am grateful to my high school teacher, Huirong Yang, who taught me how to program.

I cannot finish the research work in this thesis without the help from my peers. Rob McGuinness introduced me to the queue-based system architecture which is a critical component of the parallel boosting project. Chicheng Zhang put up with my stupid questions with great patience in the early years of my PhD (and he was a marvelous roommate). Songbai Yan helped me understand many concepts on the theory side of machine learning. Chunbin Lin taught me many things about how to approach a research problem and write a paper. Akshay Balsubramani

provided valuable insights on the research directions, and encouraged me to keep going.

I met many talented and friendly people in CSE. In particular, I would like to thank Zhengqin Li, Shengye Wang, Khalil Mrini, Shuang Liu, Sai Bi, Yao Qin, Zack Lipton, Sharad Vikram, Guo Li, and many others for their friendship and support. I would like to thank my lab mates Yizheng Wang, Joseph Geumlek, Casey Meehan, Mary Ann Smart, Robi Bhattacharjee, Jacob Geumlek, Zhifeng Kong, and Yaoyuan Yang for the great time we spent together.

Thanks to Hanjun Dai for the years of friendship and trust.

Thanks to Xiaorong Zhu for her support and kindness.

Thanks to Yan Shu and Sophia Sun for the pleasure of creating a podcast on machine learning together as a side project to research. The conversations gave me a broader view of the research topics (and a lot of fun).

Thanks to Yao Yu for her delightful personality and helping me survive the hard time in the year of graduation.

I did two internships during my graduate study with amazing mentors. I especially want to thank Mingxi Wu and Tian Wang for showing me how to turn knowledge into products.

My deepest gratitude goes to my parents and my sister for their love. I cannot imagine the sacrifices they made over the years. None of these would be possible without their support and the numerous opportunities that they gave me.

Chapter 2 through Chapter 7 contain material from the Conference on Advances in Neural Information Processing Systems, 2019 (“Faster Boosting with Smaller Memory,”, Alafate and Freund). The dissertation author was a primary investigator and author of this paper.

Chapter 4 contains material currently being prepared for submission: “Tell me something new: A new framework for asynchronous parallel learning”, Alafate and Freund. The dissertation author was a primary investigator and author of this paper.

Chapter 8 through Chapter 9 contain material from the ICML Workshop on Real World

Experiment Design and Active Learning, 2020 (“Experimental Design for Bathymetry Editing”, Alafate, Freund, Sandwell and Tozer). The dissertation author was a primary investigator and author of this paper.

Much of the above work was supported by the NIH under grant U19-NS107466.

VITA

2014	Bachelor of Science, Fudan University, China
2017	Master of Science, University of California San Diego
2020	Doctor of Philosophy, University of California San Diego

ABSTRACT OF THE DISSERTATION

Parallel Boosting and Learning from Diverse Datasets

by

Julaiti Alafate

Doctor of Philosophy in Computer Science

University of California San Diego, 2020

Professor Yoav Freund, Chair

This thesis is a study of boosting. It consists of two parts. In the first part, we develop a new way of parallelizing boosting. In the second part, we apply boosting to the problem of bathymetry data editing and study the issues of experimental design for diverse datasets.

The first part of this thesis presents a parallel boosting algorithm that achieves a significant speedup while keeping a small memory footprint. It combines two novel techniques. One is a method for parallelization with weak synchronous requirement which we call "Tell Me Something New" (TMSN). The other is a method we call stratified weighted sampling that significantly reduces the I/O load of boosting.

We implemented our algorithm using the Rust programming language and demonstrated

its superior performance when memory size is limited. Our experiments show a 10-100x speedup over two of the popular implementations of boosted trees, XGBoost [CG16] and LightGBM [KMF$^+$17], when training data is too large to fit in memory.

The second part of this thesis involves a project that uses boosting as an aid in the bathymetry data editing. Bathymetry is a study of the depths and shapes of underwater terrain. The objective of our project is to create a binary classifier that separates the correct depth measures from the incorrect ones. Our experimental results challenge the standard assumption that training and testing samples are both drawn i.i.d. from a fixed distribution.

First, we examine *spurious correlation*, where some training and testing samples are similar to each other because they are duplicates, near-duplicates, or sequentially collected. A simple memorization-based model could achieve a low in-sample validation error in these cases, but its out-of-sample test error is much worse.

Second, we examine *data diversity*, in which datasets are not diverse enough to be representative. It happens when the feature dimension is so high that collecting a representative sample is difficult. The models trained in these cases perform poorly on a new test set collected separately because of the domain shift problem.

Lastly, we propose an alternative framework from the perspective of experimental design and present a case study with modeling bathymetry data editing.

Chapter 1

Introduction

In this thesis, we study boosting and its application. Boosting [FS97, SF12], and in particular gradient boosted trees [Fri01], are some of the most popular learning algorithms used in practice. There are several highly optimized implementations of boosting, among which XGBoost [CG16] and LightGBM [KMF+17] are two of the most popular ones. These implementations can train models with hundreds of trees using millions of training examples in a matter of minutes.

The thesis is organized in two parts.

In Part I, we present a parallel boosting algorithm. It addresses a significant limitation of the existing boosting algorithm that all of the training examples are required to store in the main memory. Some workaround has been proposed. For example, XGBoost can operate in the disk-mode, which makes it possible to use machines with smaller memory than the training set size. However, it comes with a penalty in much longer training time.

We present a new implementation of boosted trees[1]. This implementation can run efficiently on machines whose memory sizes are much smaller than the training set. It is achieved with no penalty in accuracy, and with a speedup of 10-100x over XGBoost in disk mode.

[1]The source code of the implementation is released at `https://github.com/arapat/sparrow`.

Our method is based on on the observation that each boosting step corresponds to an estimation of the gradients along the axis defined by the weak rules. The common approach to performing this estimation is to scan *all* of the training examples so as to minimize the estimation error. This operation is very expensive especially when the training set does not fit in memory.

We reduce the number of examples scanned in each boosting iteration by combining two ideas. First, we use *early stopping* [Wal73] to minimize the number of examples scanned at each boosting iteration. Second, we keep in memory only a sample of the training set, and we replace the sample when the sample in memory is a poor representation of the complete training set. We exploit the fact that boosting tends to place large weights on a small subset of the training set, thereby reducing the effectivity of the memory-resident training set. We propose a measure for quantifying the variation in weights called the *effective number of examples*. We also describe an efficient algorithm, *stratified weighted sampling*.

We implemented a new boosted tree algorithm with these three techniques, called **Sparrow**. We compared its performance to the performance of XGBoost and LightGBM and show that Sparrow can achieve 10-20x speed-up over these existing algorithms especially in the limited memory settings.

In Part II, we use boosting as an aid in the bathymetry data editing, and explore the flaws of the common assumption on how training and testing samples are generated.

Bathymetry is a study of the depths and shapes of underwater terrain. The learning objective is to create a classifier that distinguish correct depth measures from the incorrect ones. The bathymetry editing data is a large-scale, multi-source dataset. In this project, we use a subset of the full data as our dataset. This dataset is over 100 GB in size. The examples in the dataset comes from multiple research institutions. Preliminary study shows that the samples from different data sources differ from each other statistically.

Our experiment results challenge the common consensus in machine learning that if the training data size is sufficiently large, the trained model would *generalize well* to the unseen

samples. The assumption underneath is that the training and testing samples are both i.i.d. from a fixed distribution. We study two problems when this assumption is not suited in practice.

First, we examine *spurious correlation*, which is a challenge to the "independence" assumption. Depending on how the data is collected, some training and testing samples are highly correlated to each other, because they are duplicates, near-duplicates, or sequentially collected. A simple memorization-based model could achieve a low in-sample validation error in these cases, but its out-of-sample test error is much worse.

Second, we examine *data diversity*, which is a challenge to the "identically distributed" assumption. It happens when the feature dimension is so high that collecting a representative sample is difficult. The models trained in these cases perform poorly on a new test set collected separately because of the domain shift problem.

Lastly, we propose an alternative framework from the perspective of experimental design and present a case study with modeling bathymetry data editing.

1.1 Preliminaries

Let $x \in X$ be the feature vectors and let the output be $y \in Y$ be the associated label or class. In this thesis, we assume binary classification task, i.e. $Y = \{0, 1\}$. For a joint distribution $\mathcal{D}$ over $X \times Y$, our goal is to find a hypothesis $H : X \to Y$ from a hypothesis class $\mathcal{H}$ that could achieve small error:

$$\mathrm{err}_{\mathcal{D}}(H) \doteq P_{(x,y)\sim\mathcal{D}}\left[l\left(y, H(x)\right)\right],$$

where the error function $l : Y \times Y \to \mathbb{R}$ quantifies the difference between the prediction $H(x)$ and the true label y. One typical error function is 0-1 loss $l(\hat{y}, y) = \mathbb{1}\left(\hat{y} \neq y\right)$.

As in the typical setting of supervised learning, the input to the learning algorithm is the training set $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ of n samples, where $(x_i, y_i) \in S$ is drawn i.i.d. (independent and identically distributed) from $\mathcal{D}$. Optimizing the performance of h on the training

set is not of much interest because we want h to perform well on unseen data. To do this, we need a *test set*, a separate set of labeled examples from the training set. If the two sets are completely independent from each other, predicting the labels of the test set using the training set is not possible. In general, we assume both training and testing examples are randomly sampled from a fixed but unknown distribution $\mathcal{D}$. Thus the objective of supervised learning algorithms is to minimize the expected loss $E_{(x,y)\sim\mathcal{D}}[l(h(x),y)]$, which also known as *generalization error*.

Part I

Lock-free parallel boosting

Chapter 2

An overview of boosting

We start with a brief overview of boosting. Boosting is a big and fascinating subject. This chapter highlights the topics that related to this thesis. For more detailed review and analysis of boosting, readers are referred to [SF12].

2.1 Strong and weak learnability

The intuition of boosting algorithms is to construct an accurate hypothesis H by combining the predictions of multiple "weaker" hypothesis h_i, each of them just needs to be at least slightly better than random guessing on the examples on which it was trained [SF12]. Before describe boosting algorithms in detail, we first need to describe weak and strong learnability.

Definition 1 *A class $\mathcal{H}$ is* strongly PAC learnable *if there exists a learning algorithm A such that for any distribution $\mathcal{D}$ over the instance space $\mathcal{X}$, and for any $c \in \mathcal{H}$ and for any positive value of ε and δ, algorithm A takes as input $m = m(\varepsilon, \delta)$ examples $(x_1, c(x_1)), \ldots, (x_m, c(x_m))$ where x_i is chosen independently at random according to D, and produces a hypothesis h such that*

$$P[err(h) > \varepsilon] \leq \delta,$$

where $err(h)$ is the generalization error of h with respect to distribution $\mathcal{D}$.

Definition 2 *A class $\mathcal{H}$ is* weakly PAC learnable *if for some $\gamma > 0$, there exists a learning algorithm A (often referred as γ-weak learner) such that for any distribution $\mathcal{D}$ over the instance space $\mathcal{X}$, and for any $c \in \mathcal{H}$ and for any positive value of δ, algorithm A takes as input $m = m(\varepsilon, \delta)$ examples $(x_1, c(x_1)), \ldots, (x_m, c(x_m))$ where x_i is chosen independently at random according to D, and produces a hypothesis h such that*

$$P[err(h) > \varepsilon] \leq \delta,$$

where $\varepsilon = \frac{1}{2} - \gamma$.

The only difference between *weak* and *strong* learnability is the relaxation to ε. In strong PAC learnability, ε can be arbitrarily small (yet positive), thus the learning algorithm is required to achieve arbitrarily small error as the number of training examples increases. In weak PAC learnability, we drop this requirement and fixed ε to $\frac{1}{2} - \gamma$, thus the learning algorithm is only required to have a small "edge" over random guessing.

Astonishingly, the weak and strong learnable models are equivalent such that the weak learners could be "boosted" into the strong learners. This is the promise of any boosting algorithm. The first provable boosting algorithm was proposed by [Sch90]. In next two sections, we describe two of the most popular variates of the boosting algorithm.

2.2 Confidence-rated boosting

In this section, we describe the confidence-rated boosting algorithm under the AdaBoost framework (Algorithm 9.1 on the page 274 of [SF12]).

We are given a set $\mathcal{H}$ of base classifiers (weak rules) $h : X \to [-1, +1]$. We want to

generate a *score function*, which is a *weighted* sum of T rules from $\mathcal{H}$:

$$H_T(\vec{x}) = \left(\sum_{t=1}^{T} \alpha_t h_t(\vec{x}) \right).$$

The term α_t is the weights by which each base classifiers contributes to the final prediction, and is decided by the specific boosting paradigm.

Finally, we have the strong classifier as the sign of the score function: $H_T = \text{sign}(H_T)$.

AdaBoost can be viewed as a coordinate-wise gradient descent algorithm [MBBF99]. The algorithm iteratively finds the direction (weak rule) which maximizes the decrease of the average potential function, and then adds this weak rule to H_T with a weight that is proportional to the magnitude of the gradient. The potential function used in AdaBoost is $\Phi(\vec{x}, y) = e^{-H_t(\vec{x})y}$. Other potential functions have been studied (e.g. [Fri01]). In this work we focus on the potential function used in AdaBoost.

We distinguish between two types of average potentials: the expected potential or true potential:

$$\Phi(H_t) = E_{(\vec{x},y)\sim\mathcal{D}} \left[e^{-H_t(\vec{x})y} \right],$$

and the average potential or empirical potential:

$$\widehat{\Phi}(H_t) = \frac{1}{n} \sum_{i=1}^{n} e^{-H_t(\vec{x}_i)y_i}.$$

The ultimate goal of the boosting algorithm is to minimize the expected potential, which determines the true error rate. However, most boosting algorithms, including **XGBoost** and **LightGBM**, focus on minimizing the empirical potential $\widehat{\Phi}(H_t)$, and rely on the limited capacity of the weak rules to guarantee that the true potential is also small. In Chapter 4, we present a different approach by using an estimator of the true edge (explained below) to identify the weak rules that reduce the *true* potential with high probability.

Consider adding a weak rule h_t to the score function H_{t-1}, we get $H_t = H_{t-1} + \alpha_t h_t$. Taking the partial derivative of the average potential Φ with respect to α_t we get

$$\left.\frac{\partial}{\partial \alpha_t}\right|_{\alpha_t=0} \Phi(H_{t-1} + \alpha_t h) = E_{(\vec{x},y)\sim\mathcal{D}_{t-1}}\left[h(\vec{x})y\right] \tag{2.1}$$

where

$$\mathcal{D}_{t-1} = \frac{\mathcal{D}}{Z_{t-1}} \exp\left(-H_{t-1}(\vec{x})y\right), \tag{2.2}$$

and Z_{t-1} is a normalization factor that makes $\mathcal{D}_{t-1}$ a distribution.

Boosting algorithms performs coordinate descent on the average potential where each coordinate corresponds to one weak rule. Using equation (2.1), we can express the gradient with respect to the weak rule h as a correlation, which we call the *true edge*:

$$\gamma_t(h) \doteq \mathrm{corr}_{\mathcal{D}_{t-1}}(h) \doteq E_{(\vec{x},y)\sim\mathcal{D}_{t-1}}\left[h(\vec{x})y\right]. \tag{2.3}$$

Given n i.i.d. samples, an unbiased estimate for the true edge is the *empirical edge*:

$$\hat{\gamma}_t(h) \doteq \widehat{\mathrm{corr}}_{\mathcal{D}_{t-1}}(h) \doteq \sum_{i=1}^{n} \frac{w_i}{Z_{t-1}} h(\vec{x}_i) y_i, \tag{2.4}$$

where $w_i = e^{-y_i H_{t-1}(\vec{x}_i)}$ and $Z_{t-1} = \sum_{i=1}^{n} w_i$. For clarity, we overload the notation and use γ_t and $\hat{\gamma}_t$ to denote $\gamma_t(h)$ and $\hat{\gamma}_t(h)$, respectively.

We can decompose $\hat{\gamma}_t$ as follows:

$$\begin{aligned}
\hat{\gamma}_t(h) &= \sum_{i=1}^{n} \frac{w_i}{Z_{t-1}} h(\vec{x}_i) y_i \\
&= \frac{1}{Z_{t-1}} \left[\sum_{i=1}^{n} w_i \mathbb{1}\left(h(x_i) = y_i\right) - \sum_{i=1}^{n} w_i \mathbb{1}\left(h(x_i) \neq y_i\right) \right] \\
&= (1 - \varepsilon_t) - \varepsilon_t = 1 - 2\varepsilon_t,
\end{aligned}$$

Given: $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \in \mathcal{X}, y_i \in Y$.
Initialize: $w_i^{(1)} = 1$ for $i = 1, \ldots, m$.
For $t = 1, 2, \ldots, T$:

1. Compute a distribution D_t with regards to $w_i^{(t)}$: $D_t(i) = \frac{w_i^{(t)}}{Z_t}$, where $Z_t = \sum_i w_i^{(t)}$ is a normalization factor.
2. Train weak learner using distribution D_t.
3. Get weak hypothesis $h_t : \mathcal{X} \to Y$.
4. Aim: select h_t with low weighted error:

$$\varepsilon_t = P_{i \sim D_t}[h_t(x_i) \neq y_i].$$

5. Choose $\alpha_t = \frac{1}{2} \ln \left(\frac{1-\varepsilon_t}{\varepsilon_t} \right)$.
6. Update, **For** $i = 1, \ldots, m$:

$$w_i^{(t+1)} = w_i^{(t)} \exp(-\alpha_t y_i h_t(x_i)).$$

Output: the final hypothesis:

$$H(x) = sign \left(\sum_{t=1}^{T} \alpha_t h_t(x) \right)$$

Figure 2.1: The boosting algorithm AdaBoost for binary classification.

where ε_t is the *weighted* training error defined as $\varepsilon_t = P_{i \sim D_t}[h_t(x_i) \neq y_i]$. Thus a weak rule with a large empirical edge also achieves low training error.

The pseudo-code for AdaBoost is given in Figure 2.1. In each iteration t, the weak learner trains a new weak rule that aims to achieve low *weighted* error ε_t with regard to sample distribution D_t. The new weak rule h_t is then added to the ensemble (the final hypothesis) with a weight factor that derives from its (weighted) error rate on the training data. At the end of each iteration, the distribution will be adjusted to "skew" towards examples that are difficult to predict.

Intuitively, AdaBoost always concentrates on the harder part of the training data in every iteration. It focuses the weak learners on the examples that previous weak rules cannot perform well, by skewing the sample distribution towards (or putting more weights on) the examples that are difficult to predict by current hypothesis. The difficulty of an example is quantified by its *margin*, $yH(x)$, which we will revisit in Section 2.3.

2.2.1 A bound on the training error

One of the key condition of a good learning model is fitting the training data, or achieve low training error. Theorem 1 shows that in AdaBoost, the training error drops exponentially fast as the size of the combined weak rules increases as long as the weak rules are slightly better than random guessing, i.e. $\forall t, \gamma_t > 0$.

Theorem 1 *(From Theorem 3.1 of [SF12], Page 54) Let D_1 be an arbitrary initial distribution over the training set. The weighted training error of the final hypothesis H with respect to D_1 is bounded as*

$$P_{i\sim D_1}[H(x_i) \neq y_i] \leq \prod_{t=1}^{T} \sqrt{1-4\gamma_t^2} \leq \exp\left(-2\sum_{t=1}^{T}\gamma_t^2\right)$$

Theorem 1 justifies the criteria for selecting h_t (Step 4 in Figure 2.1) that selects the weak rule with *large enough* edge γ_t (or low weighted error ε_t). It also shows that it is not necessary to select the weak rule with the largest edge to gain the exponential rate.

2.3 From AdaBoost to gradient boosting

In last section, AdaBoost is presented as a loss minimization problem, specifically minimizing a particular exponential (loss) function $\Phi(\vec{x}, y) = e^{-H_t(\vec{x})y}$. The initial design of AdaBoost [FS97] is not for this purpose. However, the advantage of this interpretation is that it provides a unified framework to explore and improve the various flavors of boosting algorithms. In this section, we further examine AdaBoost as loss minimization, and then briefly overview how it can be generalized to other potential functions and optimization algorithms.

In binary classification, the objective is to minimize the number of mistakes. The most direct approach to achieve it is using the 0-1 loss function, which derives following total loss function

$$\mathcal{L}(H) = \sum_{i=1}^{m} \mathbb{1}\left(H(x_i) \neq y_i\right) = \sum_{i=1}^{m} \mathbb{1}\left(y_i H(x_i) > 0\right).$$

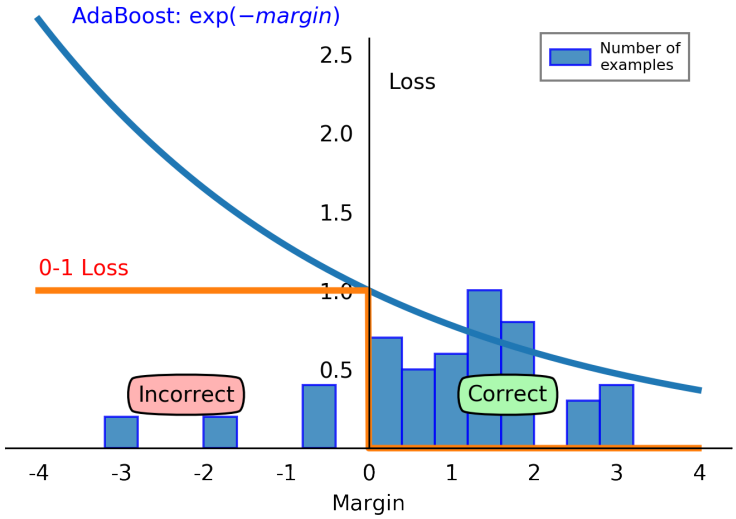

Figure 2.2: An illustration that justifies optimizing the exponential loss. AdaBoost optimizes the exponential loss over the *margin*. Margin is the product of the true label and predicted label $yH(x)$. The objective of the learning algorithm is to minimize the loss function by pushing examples to the right. Since the 0-1 loss is a step function and hard to optimize directly, some upper bound on it, e.g. the exponential loss, is used in optimization. The bars for number of examples are scaled to fit in the image.

As previously mentioned, the product of the true label and predicted label $yH(x)$ is called *margin*.

Unfortunately, 0-1 loss is a step function, which has derivative zero at all points other than at zero (where the derivative is undefined). Optimizing such function is difficult. To address this problem, an upper bound of the 0-1 loss could be used instead. In Figure 2.2, we illustrate using the exponential loss over the margin, $\mathcal{L}(H) = \sum_{i=1}^{m} \exp(-y_i H(x_i))$, as an upper bound for the 0-1 loss. It also visually shows the dynamics of the examples in terms of their margins during the boosting iterations. Intuitively, minimizing the loss functions pushes the examples to the right end of the horizontal axis, and increases their margin.

Once the loss function is defined, the next step is to search for the model parameters that minimize its value. Recall that the model output by AdaBoost in Figure 2.1 is in the form of (omitting the sign function) $H_T(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$. For simplicity, consider a finite-size hypothesis

class $\mathcal{H} = \{h_1, h_2, \ldots, h_m\}$. We can re-express the output model $H_T(x)$ as

$$H_\lambda(x) = \sum_{j=1}^{m} \lambda_j h_j(x),$$

for some set of weights λ. AdaBoost effectively learns the right weights λ by adjusting one λ_j in each iteration. This algorithmic technique, in a more general sense, is *coordinate descent* [FHH+07].

In AdaBoost, each iteration of the coordinate descent completes two tasks. First it decides the next "coordinate" to adjust using Equation 2.3, which finds some h_t with a large edge. Then it decides the proper weight adjustment α_t by taking the partial derivative of Φ with respect to h_t, which gives

$$\alpha_t(h) = \frac{1}{2} \ln \frac{1/2 + \gamma_t}{1/2 - \gamma_t}. \tag{2.5}$$

2.3.1 Gradient boosting

We analyzed AdaBoost as a coordinate descent algorithm optimizing an exponential loss function. This analysis allows to generalize AdaBoost and derive more general forms of boosting algorithm. This leads us to gradient boosting [FHT00]. Without going to much details, if we replace the exponential loss function with, as an example, cross entropy, and use gradient descent instead of coordinate descent, we can derive LogitBoost. Modern boosting packages like XGBoost [CG16] and LightGBM [KMF+17] often support a variety of loss functions, or allow users to define a custom loss function (and its derivative). It enables boosting to be a highly versatile learning algorithm and applied widely in industry (e.g. [HPJ+14a]).

2.4 Bounding the generalization error

Boosting is well-understood theoretically. In Section 2.2.1 we describe how fast the training error drops while more weak rules are being added to the additive model. In this section, we describe a bound on the *generalization error* of AdaBoost in terms of the margins of the examples. Let the *convex hull* $co(\mathcal{H})$ of the hypothesis class $\mathcal{H}$ be the set of all mappings that can be generated by taking a weighted average of the hypothesis from $\mathcal{H}$:

$$co(\mathcal{H}) \doteq \left\{ f : x \mapsto \sum_{t=1}^{T} \alpha_t h_t(x) \mid a_1, \ldots, a_T \geq 0; \sum_{t=1}^{T} a_t = 1; h_1, \ldots, h_t \in \mathcal{H}, T \geq 1 \right\}.$$

Note that the set of all possible additive models generated by AdaBoost is a subset of $co(\mathcal{H})$.

Theorem 2 *(From Theorem 5.1 of [SF12], Page 98) Let $\mathcal{D}$ be a distribution over $\mathcal{X} \times Y$, and let D_1 be the similar distribution over the training set. Assume that the hypothesis class $\mathcal{H}$ is finite, and let $\delta > 0$. Then with probability at least $1 - \delta$ over the random choice of the training set, every weighted average function $f \in co(\mathcal{H})$ satisfies the following bound*

$$P_{\mathcal{D}}[yf(x) \leq 0] \leq P_{i \sim D_1}[y_i f(x_i) \leq \theta] + O\left(\sqrt{\frac{\log|\mathcal{H}|}{m\theta^2} \cdot \log\left(\frac{m\theta^2}{\log|\mathcal{H}|} \right) + \frac{\log(1/\delta)}{m}} \right)$$

for all $\theta > \sqrt{(\ln|\mathcal{H}|)/(4m)}$.

As mentioned above, the generalization error is equivalent to the the term on the left. Theorem 2 says that for any $f \in co(\mathcal{H})$, the generalization error is small if the margins of the training examples are sufficiently large (larger than θ) for f. As one may expect, the tightness of the bound improves if the hypothesis class is small and the training sample size is large.

A generalization bound in the similar form also exists if the hypothesis class $\mathcal{H}$ is infinite, in which case the complexity of the hypothesis class can be measured by its VC dimension. For more analysis on the generalization bounds of AdaBoost, readers are referred to [SF12].

Chapter 2 through Chapter 7 contain material from the Conference on Advances in Neural Information Processing Systems, 2019 (“Faster Boosting with Smaller Memory,”, Alafate and Freund). The dissertation author was a primary investigator and author of this paper.

Chapter 3

Paradigms of distributed learning

Learning gigantic models on massive data has been a trend in recent years[1]. Ever larger learning models and training sets call for ever faster learning algorithms. On the other hand, computer clock rates are unlikely to increase beyond 4 GHz in the foreseeable future. As a result there is a keen interest in distributed, parallelized machine learning algorithms [BBL12].

There are two main motivations to train learning models in a distributed manner. One motivation is to address the challenge of storing data. Massive training data (in terms of both the number of examples and the number of features) is often distributed over multiple machines because it cannot fit in the memory of single machine. Thus learning models also have to be trained over multiple machines[2]. The other motivation to accelerate the training speed by utilizing more computation resources. In the ideal case, people want to obtain linear performance speed-up by concurrently executing the computation on multiple machines. However in practice, there are two reasons that prevent linear performance speed-up. First, message passing over network is much slower than accessing local memory. As a result, the cost of I/O involved during training

[1] As of the writing of this thesis, [BMR+20] proposes a new language model GPT-3 with 175 billions parameters trained on more than 45TB of compressed plain text, which exceeds the size of Microsoft's massive 17-billion-parameter Turning-NLG [Ros20] by one magnitude.

[2] Simple sub-sampling often leads to over-fitting and poor performance on testing data in practice, as showed in [SF10].

becomes more expensive. Second, some computers in the system might persistently fall behind others for various reasons, such as network congestion, latencies due to synchronization, laggards, and failing computers. As a result, adding more workers to the cluster could result in diminishing benefits [ZCF+10, MIM15].

3.1 Data parallel and model parallel

In large-scale machine learning, both data and model can be large. In the design of a distributed learning algorithm, we can choose to divide the learning task over data or over model. Data parallelism divide the learning task over data. It executes same computation on *different segments of data* concurrently on different machines. In most cases, some sort of communication is needed after the computation to aggregate the results generated by data segments. For example, distributed k-means can be implemented by partitioning data instances into parts, and iteratively, aggregate each part in parallel on different processors. Model parallelism divide the learning task over model. It partition *the parameters of the model* to parts, and update the parts that are independent from each other in parallel. For example, one can partition the model parameters of a large neural network to parts, and iteratively, update their weights simultaneously. The two choices of parallelism are not exclusive and could be used at the same time in the design of a distributed system.

Perhaps the most straightforward examples of distributed learning is bagging. In bagging, we train multiple models, each uses a new training set that generated from the original training set by sampling with replacement. These models could be easily trained in parallel since there is no dependencies (in terms of computation) between the training tasks.

Many learning algorithms have a sequential nature, meaning that the learning tasks are expected to proceed in a certain order because of some dependencies between them. For example, boosting iteration are sequential and each iteration depends on the previous one. For these

learning tasks, one common approach is based on the bulk synchronous parallel. Specifically to boosting, one can run the weak learner in parallel in each iteration, but synchronize the states of all computers before the next iteration starts. Because the consistency is guaranteed over the boosting iterations, the convergence bounds in the centralized setting still apply in the distributed setting.

Unfortunately, bulk synchronization does not scale well to more than 10–20 computers for reasons explained in the next section. There have been several attempts to break out of the bulk synchronous framework, most notably the work of Recht et al. on Hogwild [RRWN11] and Lian et al. on asynchronous stochastic descent [LHLL15]. Hogwild significantly reduces the synchronization penalty by using asynchronous updates and parameter servers. The basic idea is to decentralize the task of maintaining a global state and relying on sparse updates to limit the frequency of update clashes.

3.2 Trade-off between consistency and convergence

The most common approach to parallel ML is based on Valiant's bulk synchronous parallel (BSP) [Val90]. This approach calls for a set of workers and a master (Figure 3.1, Bulk synchronous parallel). All works can transit messages with the the master over network. The system works in (bulk) iterations. In each iteration the master sends a task to each worker and then waits for its response. Once *all* machines responded, the master proceeds to the next iteration. Thus the head node enforces synchronization (at the iteration level) and maintains a state that is shared by all of the workers.

BSP fits well into the iterative-convergent nature of most learning algorithms. The model parameters are guaranteed to be consistent by the end of each iteration. Any convergence guarantees that applies in the centralized setting are also valid in BSP. However, BSP suffers from the struggler problem. As showed in Figure 3.1, all fast workers must wait for the slowest

worker to complete before starting the next iteration. As a result, the time it takes to complete each iteration is up to the the slowest worker (i.e. the struggler).

Another approach is to make workers completely asynchronous. In full asynchronous parallel, workers communicate with the worker as soon as the computation is completed, without coordinating or waiting for the slower workers (Figure 3.1, Full asynchronous parallel). The problem with asynchronous parallel is that some workers could fall arbitrarily behind other workers. As a result, there is no convergence guarantee for most learning algorithms. Some notable examples of fully asynchronous learning algorithms include bagging [Bre96], in which the training tasks on different data samples are *embarrassingly parallel*, and Hogwild! [RRWN11], in which the convergence is achievable because the optimization problem is *sparse*.

To guarantee the convergence, the delay between the workers has to be bounded. One approach to this end is stale synchronous parallel (SSP) [HCC+13]. In SSP, the fastest worker and the slowest worker must not be more than τ clocks apart, where τ is the staleness parameter. The workers still run in an asynchronous manner and at their own pace. However, the fastest is forced to wait for the slowest worker to catch up if the gap between them is larger than τ. The parameter τ controls how much inconsistency is allowed in the system. In that sense, BSP and full asynchronous parallel could be seen as the special cases of SSP. Specifically, when $\tau = 0$, SSP is effectively bulk synchronous; when $\tau = \infty$, SSP is effectively full asynchronous.

There is a trade-off between convergence and consistency. In general, the convergence rate is worse as the consistency across the workers decreases [LSZ09, HCC+13], meaning that in SSP it would take more iterations for the model parameters to converge if τ is large. On the plus side, [HCC+13] shows empirically that allowing merely a small amount of staleness can speed-up the training significantly by reducing the delay caused by slow network communication and waiting for the strugglers.

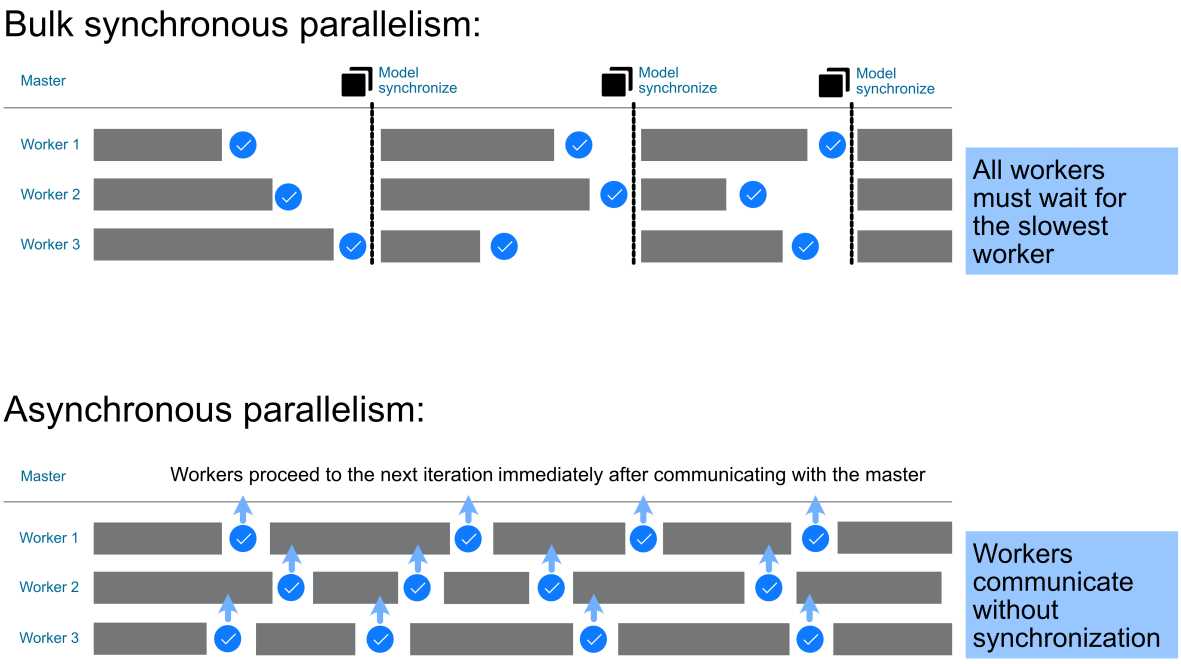

Figure 3.1: Comparison between bulk synchronous parallel (BSP) and full asynchronous parallel. In BSP, all workers synchronize their states at the end of every iteration. This paradigm fits well into the iterative-convergent nature of most learning algorithms. However, BSP is not efficient because the time it takes to finish one iteration is decided by the slowest worker (i.e. the struggler). In full asynchronous parallel, workers communicate with the master without coordinating with other workers. Thus fast workers no longer need to wait for the strugglers. The down side is that there is no any consistency guarantee among the workers, in which case proposing a convergence bound is hard.

3.2.1 Parameter Server

Classic frameworks for distributed computing, such as Hadoop and Spark [ZCF+10], are based on the bulk synchronous parallel. Thus it is difficult to implement learning algorithms with more flexible consistency requirements on these systems, such as the TMSN algorithm described in Section 4. Instead, we use a custom distributed system design that is similar to Parameter Server.

Parameter Server (PS) [SN10, LAP+14] is a general framework for implementing large-scale, distributed machine learning systems. The key idea is to store the model parameters in a

distributed hash table that is accessible through the network. PS is especially popular for training complex models with millions to billions of parameters.

Under the PS framework, the machines are divided into two roles: the servers and the workers. The servers store different partitions of the model (i.e. different subsets of the model parameters). The workers store different chunks of the training data. The servers and the workers communicate through the *push* and *pull* interfaces. The workers *pull* the model parameters as needed from the servers that store them, compute the (partial) model updates using its local data, then *push* the updated parameters to the servers.

3.3 The I/O bottleneck in parallel learning

In this section, we present a benchmark test we did on Amazon Web Services (AWS) to justify the design of our parallel boosting algorithm.

3.3.1 Network bandwidth

We start with a benchmark test for the network speed on AWS.

We created a cluster with 25, 50, 75, and 100 instances. The instance type is `c4.large`. We measure the time it takes to transmit a packet, network throughput, and the packet loss rate. There are two types of UDP packets being communicated. The first type of packet are called the workload packet. The workload packets are fixed-size (64 KB). It contains meta data including the source machine IP, the destination machine IP, and the timestamps when the packet was sent out and received. It also contains a randomly generated string to be exactly 64 KB in size. The second type of packet is called the echo packet. It contains the same meta data as the workload packet, but does not contain the random string. The echo packet is sent out from a destination machine to a source machine after a workload packet was received, so that we can confirm the transmission is successful.

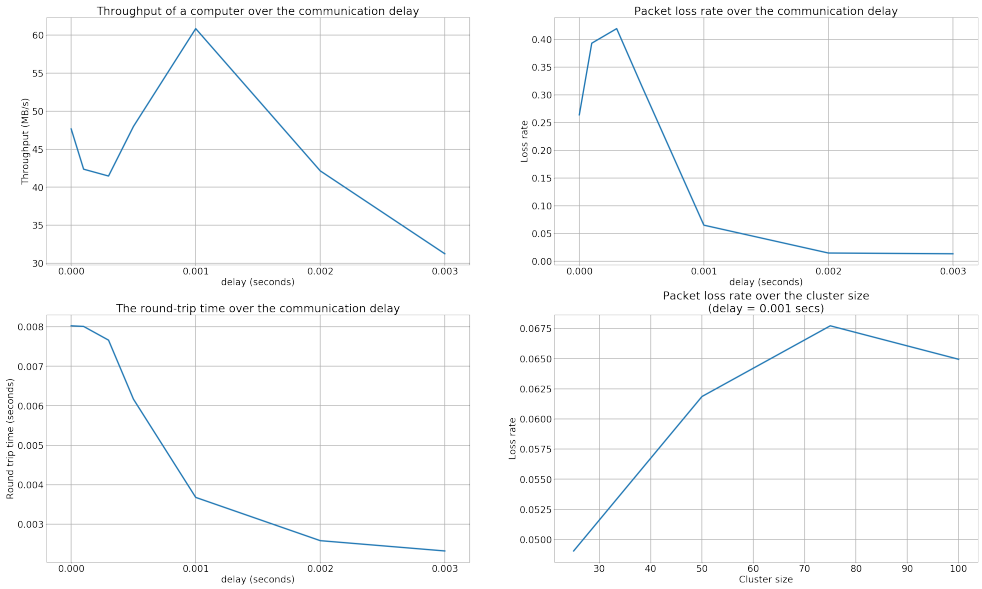

Figure 3.2: The disk loading speed on AWS ranges from 1.2 GB/s to 2.4 GB/s. Loading speed of the various-size files with random bits. The loading speed is less stable when the file size is small. If the same file is being read multiple times, the later reads could load 2x faster probably because of the page cache.

We run the program in two separate threads. A Sending Thread sends out the workload packets. Iteratively, the program sends a workload packet to every other machines in the cluster, and print a message in the log. After that, the thread sleeps for T seconds to keep intervals between sending out two consecutive packets. A Receiving Thread listens to a port where other machines would send in packets. For each packet it receives, it first prints a message in the log, then checks if the received packet is a workload packet. If so, a corresponding echo packet would be sent out to confirm that the workload was received.

The graph below shows the network throughput and loss as a function of delay between two consecutive packet sends. We can learn that the maximum throughput is achieved when the delay is just a bit larger than the average round trip time, 0.001 second. However, the packet loss rate is still quite high at this point (6%). The packet loss reduced to less than 1.4% when the delay is no smaller than 0.002 second.

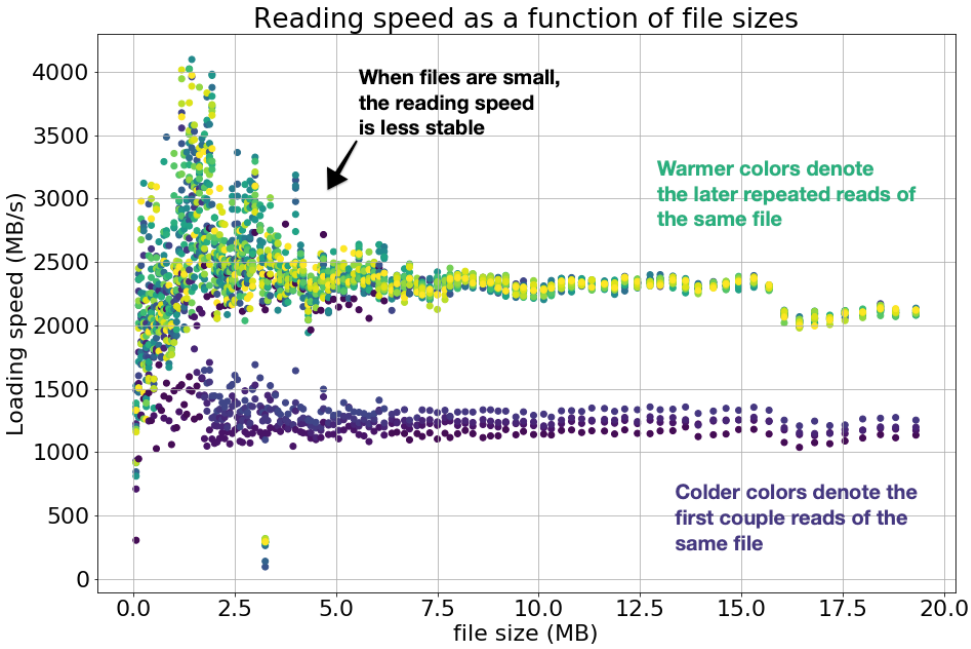

Figure 3.3: The disk loading speed on AWS ranges from 1.2 GB/s to 2.4 GB/s. Loading speed of the various-size files with random bits. The loading speed is less stable when the file size is small. If the same file is being read multiple times, the later reads could load 2x faster probably because of the page cache.

3.3.2 Disk I/O

In this benchmark test on AWS, we want to measure the I/O performance of the hard disk, specifically in Python programs. In all tests, we used the instance store, which are SSDs that physically attached to the EC2 instances. We didn't use the alternative Elastic Block Store (EBS), which is connected to the EC2 instances through network, because we observe much better I/O performance on the instance store.

We first generate a series of files with sizes ranging from 64 KB to 19.75 MB by writing random bits to disk. Then we read each files repeatedly and measure the wall clock time for each read from the moment reading starts to the moment it finishes.

In Figure 3.3, we see that the loading speed for the files on disk ranges from 1.2 GB/s to 2.4 GB/s. When we read a file multiple times, the initial reads (cold start) is usually slower than the later reads. The speed-up could be around 2 times. We believe it is due to the page caching

policy of the operating system, which loads recently used files from disk to memory for faster future access if some fraction of the memory is unused. Besides, we can also see that when the files are small, the variance of the loading speed is larger.

In conclusion, we read a peak network throughput at 60 MB/s, while the disk throughput stably at 1000 – 2500 MB/s.

Chapter 2 through Chapter 7 contain material from the Conference on Advances in Neural Information Processing Systems, 2019 (“Faster Boosting with Smaller Memory,”, Alafate and Freund). The dissertation author was a primary investigator and author of this paper.

Chapter 4

TMSN and boosting by early stopping

In this chapter we describe a new approach for parallelizing ML algorithms which reduces communication and synchronization between workers. We call this approach "Tell Me Something New" (**TMSN**) [AF18]. To explain **TMSN** we start with an analogy.

Consider a team of a hundred human investigators that is going through thousands of documents to build a criminal case. Suppose that the information in the documents is highly overlapping. Therefor, when an investigator reads a new document she is likely to find no new information. How should the investigators organize their work and communicate their findings to each other?

We contrast the bulk synchronous parallel (BSP) approach and the **TMSN** approach. In BSP, each investigator takes a stack of documents to their cubicle and reads through it. Then all of the investigator meet in a room and tell each other what they found. Once they are done, the process repeats. One problem with this approach is that the fast readers have to wait for the slow readers. Another is that a decision needs to be made on how many documents or pages, to put in each stack. Too many and the iterations would be very slow, too few and all of the time would be spent in meetings.

The **TMSN** approach is radically different. In this approach, each investigator has a queue

of incoming documents that is replenished with new documents when the queue is short. There is no meeting either. Instead, when an investigator finds a piece of information that she believes is new, she sends an email to all other investigators, updating them on her finding. The emails are read by the others in the order in which they were received. This has several advantages. First, the investigators work at their own pace, meaning that no one is waiting for the others. Second, the new information is shared as soon as it is available in a non-blocking manner (emails to be read later) instead of stopping people from their work (a meeting). Third, the system is fault resilient — somebody falling asleep has little effect on the others. The analogy to parallel ML maps investigators to computers, "case" to "model", "documents" to "training data", and "new information" to "improved model".

More concretely, **TMSN** for model learning works as follows. Each worker has a model H and an upper bound L on the true loss of H. The worker searches for a better model H' whose loss upper bound is L'. If L' is significantly smaller than L, then the worker takes two actions. First, it uses (H', L') to replace (H, L). Second, it broadcasts (H', L') to all other workers. Each worker also listens to the broadcast channel. If a pair (H', L') is received, the worker checks whether L' is significantly lower than its own upper bound L. If so, the worker replaces its current model with the incoming one. Otherwise, the worker discards the incoming pair.

We apply the idea of **TMSN** on designing boosting algorithm. The proposed boosting algorithm combines in-memory processing and out-of-memory sampling. In this section, we introduce the in-memory processing. The proposed method is based on a modification of boosting by filtering, or FilterBoost [BS07]. Our method is different from FilterBoost for (1) applying a tighter stopping rule, and (2) estimating the edges of the new rules with a sample set instead of repeatedly drawing fresh examples. We are able to quantify the "effectiveness" of the sample set using the effective sample size, which we explain in Section 5.

4.1 Sequential analysis

Sequential analysis was introduced by Abraham Wald in the 1940s [Wal73] during the Second World War. It is studied to provide an efficient quality control measure for testing anti-aircraft missile [Wal80]. The problem is formulated as "calculating the probability of a hit by anti-aircraft fire on a directly approaching dive bomber". Prior statistical methods were not satisfactory because of the large sample size (missile fires) required to guarantee the degree of precision and confidence of the estimates. In comparison, an experienced military personnel could stop an experiment after significantly fewer rounds of fire and conclude that the experiment does not need to be complete because the test subject is either obviously superior or obviously inferior to the testing objective. In the hindsight, it suggests that a sequential test could be much more efficient than a batch test. To demonstrate this intuition, we start with a simplified case of searching for a biased coin.

4.1.1 Simplified case: searching for a biased coin

Suppose we are given n coins, and asked to find a coin among them that is biased towards head. We formulate the problem as a hypothesis test. Consider a single coin with a bias p towards head. By flipping this coin, we could observe a sequence of independent and identically distributed (i.i.d.) random variables: $X_1, X_2, \ldots \in \{0,1\}$, with $P(X_i = 1) = p$. The sample mean of n coin flips is defined as

$$\hat{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} X_i.$$

By definition, $\forall i > 0, E(\hat{\mu}_i) = p$. Suppose that the probability of the biased coin tossing up head is at least $\frac{1}{2} + \gamma$ for some constant $\gamma \in (0, \frac{1}{2}]$. We seek a hypothesis test to decide between the null $H_0(\gamma) : p = \frac{1}{2}$ and the alternative $H_1(\gamma) : p \geq \frac{1}{2} + \gamma$.

Next we contrast the batch test and the sequential test. For the sake of simplicity, assume that we are given n workers/computers so that we can run the tests for n coins in parallel.

Algorithm 1 Batch test for finding the biased coin

Input Sample size N, confidence parameter α

Compute Θ with regards to N

Initialize $S_0 = 0$

for $k := 1 \ldots N$ **do**

$\mathbf{S}_k = \mathbf{S}_{k-1} + X_k$

end for

if $\hat{\mu}_N = S_N/N > \Theta$ **then**

Return the biased coin (reject H_0)

else

Failed to reject H_0

end if

Algorithm 2 Sequential test for finding the biased coin

Input confidence parameter α

Initialize $S_0 = 0$

for $k := 1, 2, \ldots$ **do**

$\mathbf{S}_k = \mathbf{S}_{k-1} + X_k$

Compute θ_k

if $\hat{\mu}_k = S_k/k > \theta_k$ **then**

Return the biased coin i (reject H_0)

end if

end for

Batch test

In batch test, the sample size N is fixed beforehand. The sample size N could be set using any concentration bound. For example, one can use the Hoeffding bound and show that

$$P(\hat{\mu}_N - p > \frac{\gamma}{2}) \leq e^{-2N\left(\frac{\gamma}{2}\right)^2} \tag{4.1}$$

with probability $\geq 1-\delta$. Following Equation 4.1, the batch test reads at least $N = \frac{2}{\gamma^2}\ln\frac{1}{\delta}$ samples, and compares the sample mean $\hat{\mu}_N$ against the threshold $\Theta = \frac{\gamma}{2}$. The hypothesis test rejects the null if $\hat{\mu}_N > \Theta$, and accept the null otherwise, as showed in Algorithm 1. The test is *embarrassingly parallel* [HS11]. Thus with n workers, the tests on all n coins finish in $O\left(\frac{1}{\gamma^2}\ln\frac{1}{\delta}\right)$ time.

The batch test sets the sample size such that the deviation of the test statistics is bounded even in the worst case scenario. However, if there exists a coin such that its bias towards head is significantly larger than $\frac{1}{2}+\gamma$, the hypothesis test would require, potentially, fewer than N samples, and terminate early. In contrast to the batch test, we can use sequential test to run hypothesis test without fixing the sample set size in advance.

Sequential test

The sequential test uses a stopping rule, which is based on a sequence of thresholds $\{\theta_1, \theta_2, \ldots\}$. The stopping rule can stop the test *early*, if at any time the test statistics (i.e. the sample mean) is larger than the corresponding threshold. More specifically, the test takes the samples one at a time. In each step k, the test updates the test statistics $\hat{\mu}_k$, and compares it against the threshold θ_k. The stopping rule triggers at any time if $\hat{\mu}_k > \theta_k$, in which case the test can immediately return without seeing the remaining samples, as showed in Algorithm 2. One choice of setting θ_k is based on Theorem 3, which depends on both the mean and the variance of X_i.

Theorem 3 (based on Balsubramani [Bal14] Theorem 4) *Let M_t be a martingale $M_t = \sum_i^t X_i$, and suppose there are constants $\{c_k\}_{k\geq 1}$ such that for all $i \geq 1$, $|X_i| \leq c_i$ w.p. 1. For $\forall\sigma > 0$, with*

probability at least $1-\sigma$ *we have*

$$\forall t : |M_t| \leq C\sqrt{\left(\sum_{i=1}^{t} c_i^2\right)\left(\log\log\left(\frac{\sum_{i=1}^{t} c_i^2}{|M_t|}\right)+\log\frac{1}{\sigma}\right)}, \tag{4.2}$$

where C is a universal constant.

Suppose we know how bias is the coin with the largest probability to turn up head, we can derive an optimal sample size n^* beforehand. However, since we don't have this information, it is hard to set a *right* sample size n in the batch test, If n is set much smaller than n^*, the null may be rejected incorrectly, meaning that a fair coin could seem to be biased just because of chance. If n is set much larger than n^*, it leads to unnecessary waste of the computational resources.

The sequential test addresses this problem by dynamically decide to stop when it sees about n^* samples, while delivering same error guarantees as in the batch setting.

4.1.2 Sequential analysis in machine learning

Suppose we want to estimate the expected loss of a model. In the standard large deviation analysis, we assume that the loss is bounded in some range, say $[-M,+M]$, and that the size of the training set is n. This implies that the standard deviation of the training loss is at most $M/\sqrt{n}$. In order to make this standard deviation smaller than some $\varepsilon > 0$, we need that $n > (M/\varepsilon)^2$. While this analysis is optimal in the worst case, it can be improved if we have additional information about the standard deviation. We can glean such information from the observed losses by using the following sequential analysis method.

Instead of choosing n ahead of time, the algorithm computes the loss one example at a time. It uses a *stopping rule* to decide whether, conditioned on the sequence of losses seen so far, the difference between the average loss and the true loss is smaller than ε with large probability. The result is that when the standard deviation is significantly smaller than $M/\sqrt{n}$, the number of examples needed in the estimate is much smaller than $(M/\varepsilon)^2$.

Applying the approach of **TMSN**, we can use the stopping rule that allows the workers to update the model *with confidence* using only its local data, and communicate with other workers only if the improvement on the model is statistically significant. Since the workers communicate at their own pace, the parallelization is asynchronous. In **TMSN**, the significance of the model updates on each worker is validated individually using a sequential test. The test reads as many samples as it is required from the local data available on each worker. The false discovery of test is guaranteed by the chosen stopping rule.

TMSN has two advantages over the synchronous approaches (such as bulk synchronous parallel (BSP)). First, **TMSN** does not suffer from the struggler problem because each workers update the models at their own pace independently. Second, **TMSN** reduces the amount of communication over the network because the workers only communicate when it is necessary.

In next section, we propose a **TMSN**-based approach to boosting.

4.2 Finding weak rules with a sequential test

Section 2.2.1 shows that, to decrease the expected potential in AdaBoost, we want to find a weak rule with a large edge (and add it to the score function). The common approach to do this by searching for the weak rule with the largest *empirical edge*. In this section, we show a different approach that finds a weak rule which, with high probability, has a significantly large *true edge*.

The setup of the problem is similar to how we detect a biased coin in Section 4.1.1. Let $\gamma > 0$ be a parameter such that, for any weak rule h_t with a true edge γ_t, we say γ_t is significantly if $\gamma_t > \gamma$. We formulate a hypothesis test, where the null is $\gamma_t \leq \gamma$, and the alternative is $\gamma_t > \gamma$.

Fixing the current strong rule H (i.e. the score function), we define a (unnormalized) weight for each example, denoted as $w(\vec{x}, y) = e^{-H(x)y}$. Consider a particular candidate weak rule h_t and a sequence of labeled examples $\{(\vec{x}_1, y_1), (\vec{x}_2, y_2), \ldots\}$. For any weak rule h_t and the corresponding γ_t, we define two cumulative quantities (after seeing n examples from the

sequence):

$$M_t \doteq \sum_{i=1}^{n} w(\vec{x}_i, y_i)(h_t(\vec{x}_i)y_i - \gamma), \text{ and } V_t \doteq \sum_{i=1}^{n} w(\vec{x}_i, y_i)^2. \tag{4.3}$$

M_t is an estimate of the difference between the true correlation of h and the parameter γ. V_t quantifies the variance of this estimate.

The goal of the stopping rule is to identify h_t if $\gamma_t > \gamma$. Based on Theorem 3, the stopping rule is defined to be $t > t_0$ and

$$M_t > C\sqrt{V_t(\log\log\frac{V_t}{M_t} + B)}, \tag{4.4}$$

where t_0, C, and B are parameters. If both conditions of the stopping rule are true, we would reject the null, and accept that the true edge of h_t is larger than γ with high probability. The proof of this test can be found in [Bal14].

If the stopping rule is triggered for some weak rule h_t, meaning that $\gamma_t > \gamma$ with high probability, we would then add h_t to the score function. The associated edge of h_t is γ as an lower bound to γ_t which is unknown. Apparently, the associated weight of h_t in the score function (α_t in Figure 2.1) could then be underestimated. When it happens, the weak rule h_t could trigger the stopping rule again in the later boosting iterations, and its weight α_t in the score function would be increased subsequently.

Note that our stopping rule is correlated with the cumulative variance V_t, which is basically the same as $1/n_{\text{eff}}$. If n_{eff} is large, say $n_{\text{eff}} = n$ when a new sample is placed in memory, the stopping rule stops quickly. On the other hand, when the weights diverge, n_{eff} becomes smaller than n, and the stopping rule requires proportionally more examples before stopping.

The relationship between martingales, sequential analysis, and stopping rules has been studied in previous work [Wal73]. Briefly, when the edge of a rule is smaller than γ, then the sequence is a supermartingale. If it is larger than γ, then it is a submartingale. The only assumption is that the examples are sampled i.i.d.. Theorem 3 guarantees two things about the

stopping rule defined in Equation 4.4: (1) if the true edge is smaller than γ, the stopping rule will never fire (with high probability); (2) if the stopping rule fires, the true edge of the rule h is larger than γ.

Setting the value of γ

An important parameter in the stopping rule is γ. The role of γ is similar to a filter. It prevents the weak rules with a smaller edge from triggering the stopping rule. When γ is too small, it takes fewer examples for the stopping rule to trigger, yet the selected weak rule h_t would be added to the score function with a weight α_t that is much smaller than what it should be (because we significantly underestimate the edge of h_t). As a result, the algorithm takes more iterations to converge since the training error decreases slower. On the other hand, when γ is too large, we may not find any weak rule that could trigger the stopping rule.

In the experiment, we set a large initial value for γ, e.g. 0.25. We then gradually decrease the value of γ every time when none of the workers detects a weak rule with a significantly large edge after scanning all or most of their local examples.

Empirical study on the stopping rule

Another key parameter in the stopping rule is the shrinking factor C. In the experiments, we observed that the stopping rule in Theorem 3 could be too loose empirically. We addressed this problem by setting a shrinking factor C on the upper bound used in the stopping rule, so that the stopping rule is more likely to be triggered and be trigger early. Unavoidably, a tighter, more aggressive stopping rule could increase the false positive rate, or the probability that the null is incorrectly rejected. To get a sense of the impact of C on the false positive rate, we conducted an experiment using synthetic data.

We repeated the following experiment. A sequence of i.i.d. random variables, $X_1, X_2, \ldots X_n \in \{0,1\}$, is generated with the size of the sequence $n = 1,000,000$ and $P(X_i = 1) = 0.5$. We then

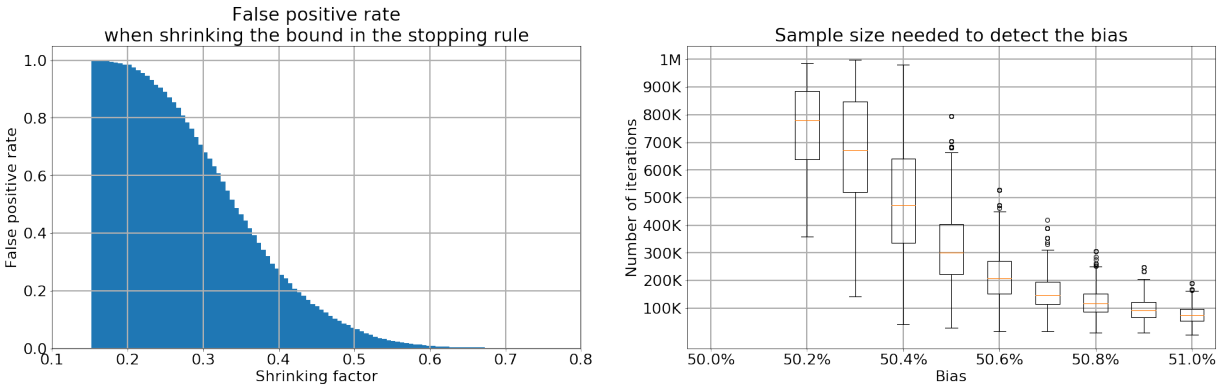

Figure 4.1: In the left plot, we show how false positive rate responses to the different values of the shrinking factor C. The false positive rate increases as C decreases. It is almost zero when $C > 0$. In the right plot, we set $C = 0.8$, and evaluate, empirically, how many samples it takes for the stopping rule to detect a bias. The result shows that the number of samples needed decreases as the bias increases. For detecting a bias of 0.1%, or $P(X_i = 1) = 0.501$, the stopping rule reads less than 1 M examples. For detecting a bias of 1%, or $P(X_i = 1) = 0.510$, the stopping rule reads less than 200 K examples.

observe if the stopping rule with the shrinking factors C would be triggered on the sequence (while reading at most n samples), which would falsely reject the null and lead to a false positive. The shrinking factor C is set to be $0.1, 0.2, \ldots, 1.0$. In the left plot of Figure 4.1, we see that the false positive rate increases as the shrinking factor decreases, which match our expectation. Meanwhile, the false positive rate is almost 0 as long as $C \geq 0.7$.

Lastly, to get a sense, empirically, on how many samples would be required for the stopping rule to trigger, we repeated the experiment above for 10 times, each time setting $P(X_i = 1)$ to the different value ranging from 0.50 to 0.51 with an interval of 0.001. With what we learned from last experiment, we set $C = 0.8$. The result in the right plot of Figure 4.1 shows, as one would expect, the stopping triggers with fewer samples as the bias $P(X_i = 1)$ increases. With this particular choice for C, we are able to detect a 0.1% bias with less than 1 M examples, and a 1% bias with less than 200 K examples. In the context of our boosting algorithm, it means that the stopping rule could trigger with fewer examples if the gap between the true edge of a weak rule and the parameter γ, namely $\gamma_t - \gamma$, is large.

Chapter 2 through Chapter 7 contain material from the Conference on Advances in Neural Information Processing Systems, 2019 (“Faster Boosting with Smaller Memory,”, Alafate and Freund). The dissertation author was a primary investigator and author of this paper.

Chapter 4 contains material currently being prepared for submission: “Tell me something new: A new framework for asynchronous parallel learning”, Alafate and Freund. The dissertation author was a primary investigator and author of this paper.

Chapter 5

Efficient weighted sampling

Recent work on machine learning often addresses enormous amount datasets. It has been showed that simple methods could outperform sophisticate ones given sufficiently large training data size [HNP09]. Though there is no limit on how much data to collect for training, the actual computation is limited by the available computer memory.

When the training data does not fit in the memory, one technique is to leverage the disk and use the disk space as a memory cache. For example, XGBoost supports the external memory version [XGB20] in which it puts both training data and all internal data structures in a cache file on disk instead of in memory. Given the performance gap between the disk and the memory (see Section 3.3), the training often takes much longer time with this technique than its in-memory counterpart for the same algorithm.

Another popular technique is simple sub-sampling. The downside is that simple sub-sampling often leads to over-fitting and poor performance on testing data in practice, as showed in, for example, [SF10].

In this chapter, we propose an efficient weighted sampling strategy that addresses this challenge for our boosting algorithm. We keep in memory only a sample of the training set. In addition, we resample when the sample in memory is a poor representation of the complete

training set. We exploit the fact that boosting tends to place large weights on a small subset of the training set, thereby reducing the effectiveness of the memory-resident training set. We propose a measure for quantifying the variation in weights called *effective sample size*. We also describe an efficient algorithm called *stratified weighted sampling*.

5.1 Effective sample size

To gain some intuition regarding the sample effectiveness, consider the following setup of an imbalanced classification problem. Suppose that the training set size is $N = 100,000$, of which $0.01N$ are positive and $0.99N$ are negative. Suppose we can store $n = 2,000$ examples in memory. The number of positive examples in memory is $0.01n = 20$. Apparently for the first weak rule h_1, the boosting algorithm should and will add an (almost) all negative rule to the score function. The initial memory-resident sample, though it is highly skewed, is effective for identifying h_1.

With the first weak rule h_1 added, we then reweigh the examples using the AdaBoost rule, and give half of the total weight to the positives and the other half to the negatives. As a result, for the memory-resident sample, the 20 positive examples have a combined sum of weights that is equivalent to the total weight of the remaining 1980 negative examples. With such a small number of positive examples, there is a danger of over-fitting, especially if the second weak rule tries to make an prediction over the positives.

In contrast, if a (weighted) resampling is called after adding the first weak rule, it would generate a new training sample with 1000 positives and 1000 negatives. The new sample allows us to find additional weak rules with little danger of over-fitting. The insight of this effect is that the absolute number of examples is not necessarily a right *effectiveness* measure of a sample in terms of estimating a statistics in order to find a weak rule, Next, we try to quantify such effectiveness using the example weights in boosting.

Recall in Section 2.2, we define the empirical edge $\hat{\gamma}(h)$ of a weak rule (Equation 2.4),

which is an unbiased estimate of its true edge $\gamma(h)$. How accurate is this estimate?

A standard quantifier is the variance of the estimator. Following the notations in Section 2.2, consider a sample $S = \{(x_1, y_1, w_1), \ldots, (x_n, y_n, w_n)\}$, where $w_i = e^{-y_i H(\vec{x}_i)}$ is the weight of the corresponding example given current score function H. Suppose the true edge of a weak rule h is γ. Then the expected (normalized) correlation between the predictions of h and the true labels is

$$\begin{aligned} E_S\left[\frac{w}{Z} y h(x)\right] &= E_S\left[\frac{w}{Z} y h(x) | y = h(x)\right] + E_S\left[\frac{w}{Z} y h(x) | y \neq h(x)\right] \\ &= \left(\frac{1}{2} + \gamma\right) - \left(\frac{1}{2} - \gamma\right) = 2\gamma, \end{aligned} \tag{5.1}$$

where Z is a normalization factor that makes $\{w_1, \ldots, w_n\}$ a distribution. The variance of this correlation can be written as

$$\mathrm{Var}\left[\frac{w}{Z} y h(x)\right] = \frac{1}{n^2} \frac{E\left(w^2\right)}{E^2(w)} - 4\gamma^2. \tag{5.2}$$

Ignoring the second term (because γ is usually close to zero) and the variance in $E(w)$, we approximate the variance in the edge to be

$$\mathrm{Var}(\hat{\gamma}) \approx \frac{\sum_{i=1}^n w_i^2}{\left(\sum_{i=1}^n w_i\right)^2}. \tag{5.3}$$

If all of the weights are equal then $\mathrm{Var}(\hat{\gamma}) = 1/n$. It corresponds to a standard deviation of $1/\sqrt{n}$ which is the expected relation between the sample size and the error.

If the weights are not equal then the variance is larger and thus the estimate is less accurate. We define the *effective number of examples* n_{eff} to be $1/\mathrm{Var}(\hat{\gamma})$, specifically,

$$n_{\text{eff}} \doteq \frac{\left(\sum_{i=1}^n w_i\right)^2}{\sum_{i=1}^n w_i^2}. \tag{5.4}$$

To see that the name "effective number of examples" makes sense, consider n weights

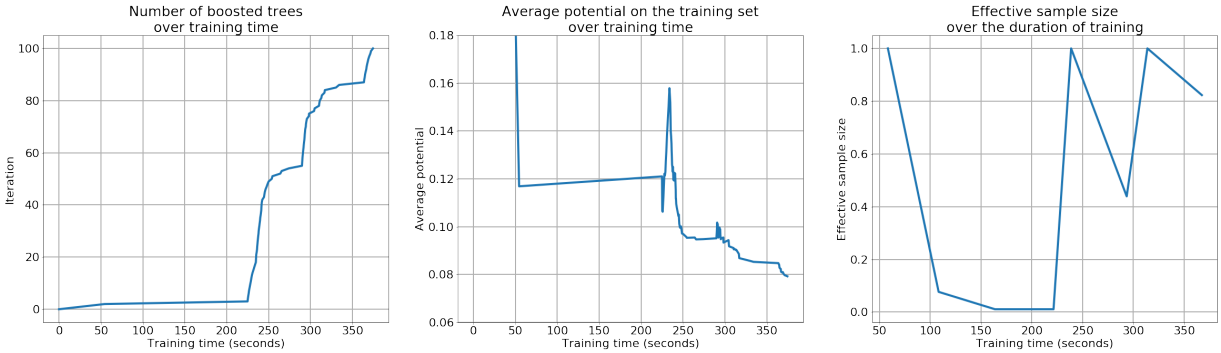

Figure 5.1: From left to right: (1) the number of boosted trees generated over the duration of the training; (2) the decrease of the average potential evaluated on the training set over the duration of the training; (3) the changes in the effective sample size of the sample over the duration of the training, where the two jumps at around 238 seconds and at around 313 seconds are the results of resampling. The first plot shows that a new sample, with a high effective sample size, could drastically reduces the training time of per boosting iteration, thus the new boosted trees are added to the score function at a much faster rate. The potential also decreases at a faster rate as showed in the second plot. The spikes seen in the average potential is caused by overfitting when the algorithm adds new rules that are learned from a sample with a very small effective sample size.

where $w_1 = \cdots = w_k = 1/k$ and $w_{k+1} = \cdots = w_n = 0$. It is easy to verify that in this case $n_{\text{eff}} = k$ which agrees with our intuition, namely the examples with zero weights have no effect on the estimate.

Suppose the memory is only large enough to store n examples. If $n_{\text{eff}} \ll n$ then we are wasting valuable memory space on examples with small weights, which can significantly increase the chance of over-fitting. We can fix this problem by using weighted sampling. In this way we repopulate memory with n equally weighted examples, and make it possible to learn without over-fitting.

Getting a new sample also reduces the number of examples needed for the stopping rule to trigger, because the bound on the right side of Equation 4.4 is tighter as a result of the smaller variance of the weights V_t. Figure 5.1 shows that, empirically, when the effective sample size is large (e.g. when a new sample replaces the old one), the training time of per boosting iteration drastically decreases, meaning that the algorithm is able to add new rules to the score function at a much faster rate. The same phenomena is also observed in the decrease of the average potential

on the training set.

5.2 Stratified weighted sampling

While there are well known methods for weighted sampling, all of the existing methods (that we know of) are inefficient when the weights are highly skewed. In such cases, most of the scanned examples are rejected, which leads to very slow sampling. To increase the sampling efficiency, we introduce a technique we call *stratified weighted sampling*. It generates same sampled distribution while guaranteeing that the fraction of rejected examples is no large than $\frac{1}{2}$.

The stratified weighted sampling is a modification on the *minimal variance weighted sampling* [Kit96]. This method reads from disk one example (x, y) at a time, calculates the weight for that example w, and accepts the example with the probability proportional to its weight. Accepted examples are stored in memory with the initial weights of 1. This increases the effective sample size back to $n_{\text{eff}} = n$, thereby reduces the chance of over-fitting. This process continues, in parallel to the boosting process, as long as the boosting algorithm is making progress and the weights are becoming increasingly skewed. When n_{eff} is much smaller than the memory size n, we clear the memory and collect a new sample with uniform weights to replace the current memory-resident sample.

The downside of the sampling method in [Kit96] is the sampling efficiency. When example weights are highly skewed and concentrated, selective sampling becomes inefficient because of the high rejection rate. The sampling method has to read each example and calculate its weight, which is computational expensive. However, when 1% of the examples amount to 99% of the weight, majority of the examples would be rejected from the sample, meaning that the samples would be collected at $1/100$ of the rate that they are read.

Stratified weighted sampling solves this inefficiency by partitioning examples into "strata" according to their weights. Each stratum i holds examples whose weights are in the range of

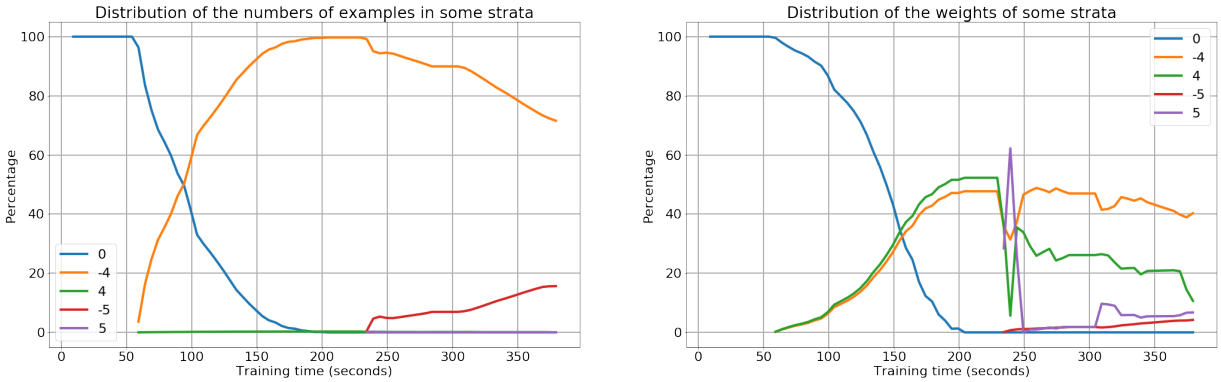

Figure 5.2: The plots show the dynamics of some strata over the training time in terms of the number of the examples in the strata and the weights of the strata, both after normalized to 100% to make them distributions. Some strata are hidden in the plots so that the plots are more readable. The legend shows the index of each stratum, which is the binary logarithm of the weights of the examples in it. In the left plot, we see that all examples are assigned to the stratum with the index 0 at time 0, because the initial weights of all examples are equal to 1 and $\lfloor \log_2 1 \rfloor = 0$. Over the duration of training, the easy examples are being moved to the stratum with the indices -4 and -5, and the hard ones are being moved to the stratum with the indices 4 and 5. In the right plot, we see the same procedure from the perspective of the weights distribution of the strata.

$\left[2^i, 2^{i+1}\right)$. The strata is maintained as follows. The sampling method takes one example out of a selected stratum, removes it from this stratum, calculates its current weight w using the current score function, and writes it back to the stratum with the index $\lfloor \log_2 w \rfloor$. For each stratum, we also keep track of the estimated sum of the weights of all examples resided in this stratum, which we refer as the weight of the stratum[1].

Stratified weighted sampling, effectively, separates the hard examples from the easy ones, and assigns them to the same blocks on disk. Figure 5.2 shows the dynamics of this process. Initially, all examples are assigned to the same stratum with the index 0 since their initial weights are all equal to 1. Over the duration of the boosting process, the hard examples, whose margins are small, are moved to the strata with a larger index, while the easy examples, whose margins are large, are moved to the strata with a smaller index.

The sampling mechanism proceeds in two steps. First, it samples a stratum following the

[1]The weight of the stratum is an estimate because the weights of the examples in the stratum are not strictly up to date with the current score function.

distribution normalized over the weights of all strata. Second, it samples an example from the sampled stratum using the minimal variance weighted sampling. The rejection rate in sampling is below $1/2$ because the weights of the examples in each stratum are within a factor of 2 from each other. It is a significant improvement in contrast to the standard approach, especially when the weights of the examples are highly skewed. For example, in Figure 5.2, the two strata with the hard examples, the stratum 4 and 5, contain less than 1% of the examples combined, but amount to nearly 20% of the total weights of all training examples.

Chapter 2 through Chapter 7 contain material from the Conference on Advances in Neural Information Processing Systems, 2019 ("Faster Boosting with Smaller Memory,", Alafate and Freund). The dissertation author was a primary investigator and author of this paper.

Chapter 6

Architecture of an asynchronous distributed booster

In this chapter we propose a new boosted tree algorithm called **Sparrow**. It combines the three techniques from Chapter 4 and Chapter 5, *early stopping*, *effective number of examples*, and *stratified weighted sampling*. The source code of Sparrow is available at `https://github.com/arapat/sparrow`.

As Sparrow consists of a number of concurrent threads and many queues, we chose to implement it using the **Rust** programming language for the benefits of its memory-safety and thread-safety guarantees [KN18].

The main procedure of **Sparrow** generates a sequence of weak rules $h_1, \ldots, h_k$ and combines them into a strong rule H_k. It calls two subroutines that execute in parallel: a **Scanner** and a **Sampler**. Next, we explain these two subroutines. The bold letters in parenthesis denote the corresponding component in the workflow diagram in Figure 6.1. The pseudo-code of the system is provided in the Appendix A.

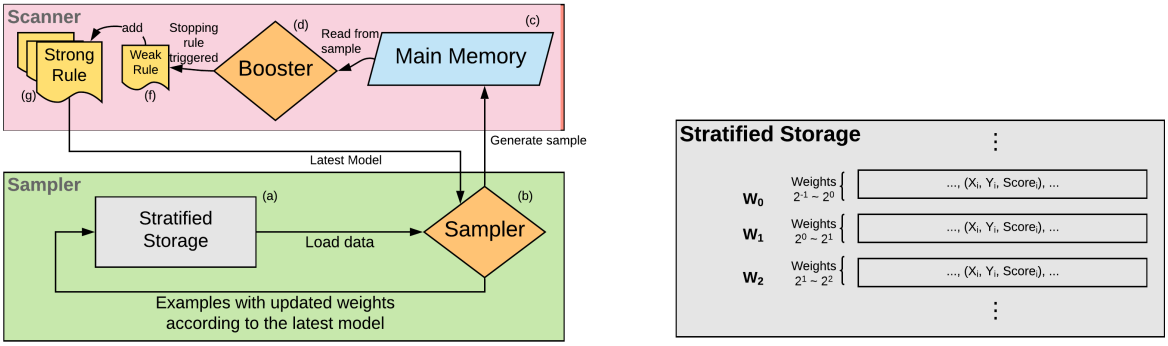

Figure 6.1: The **Sparrow** system architecture. Left: The workflow of the Scanner and the Sampler. Right: Partitioning of the examples stored in disk according to their weights.

Scanner

The task of a scanner (the upper part of the workflow diagram in Figure 6.1) is to read training examples sequentially and stop when it has identified one of the rules to be a *good* rule.

At any point of time, the scanner maintains the current strong rule H_t, a set of candidate weak rules $\mathcal{W}$, and a target edge γ_{t+1}. For example, when training boosted decision trees, the scanner maintains the current strong rule H_t which consists of a set of decision trees, a set of candidate weak rules $\mathcal{W}$ which is the set of candidate splits on all features, and $\gamma_{t+1} \in (0.0, 0.5)$.

Inside the scanner, a booster **(d)** scans the training examples stored in main memory **(c)** sequentially, one at a time. It computes the weight of the read examples using H_t and then updates a running estimate of the edge of each weak rule $h \in \mathcal{W}$ accordingly. Periodically, it feeds these running estimates into the stopping rule, and stop the scanning when the stopping rule fires.

The stopping rule is designed such that if it fires for h_t, then the true edge of a particular weak rule $\gamma(h_{t+1})$ is, with high probability, larger than the set threshold γ_{t+1}. The booster then adds the identified weak rule h_{t+1} **(f)** to the current strong rule H_t to create a new strong rule H_{t+1} **(g)**. The booster decides the weight of the weak rule h_{t+1} in H_{t+1} based on γ_{t+1} (lower bound on its accuracy). It could underestimate the weight. However, if the underestimate is large, the weak rule h_{t+1} is likely to be "re-discovered" later which will effectively increase its weight.

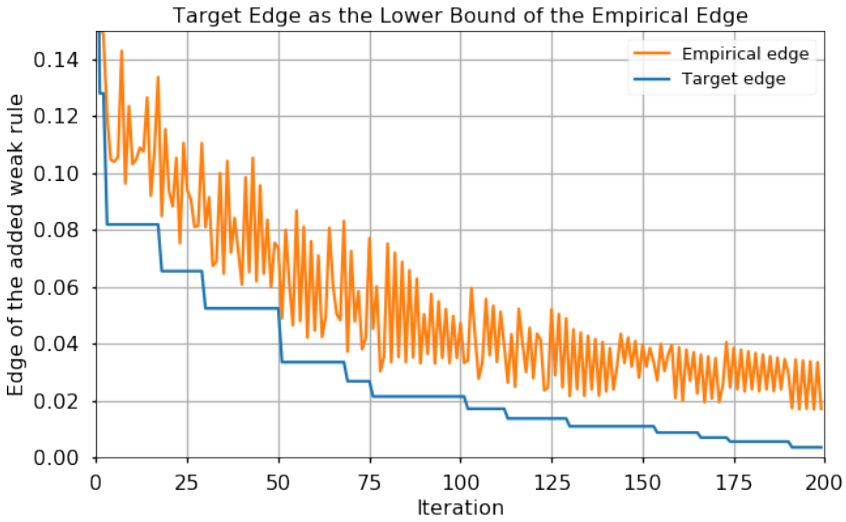

Figure 6.2: The empirical edge and the corresponding target edge γ of the weak rules being added to the ensemble. Sparrow adds new weak rules with a weight calculated using the value of γ at the time of their detection, and shrinks γ when it cannot detect a rule with an edge over γ.

Lastly, the scanner falls into the *Failed* state if after exhausting all examples in the current sample set, no weak rule with an advantage larger than the target threshold γ_{t+1} is detected. When it happens, the scanner shrinks the value of γ_{t+1} and restart scanning. More precisely, it keeps track of the empirical edges $\hat{\gamma}(h)$ of all weak rules h. When the failure state happens, it resets the threshold γ_{t+1} to just below the value of the current maximum empirical edge of all weak rules.

To illustrate the relationship between the target threshold and the empirical edge of the detected weak rule, we compare their values in Figure 6.2. The empirical edge $\hat{\gamma}(h_{t+1})$ of the detected weak rules are usually larger than γ_{t+1}. The weak rules are then added to the strong rule with a weight corresponding to γ_{t+1} (the lower bound for their true edges) to avoid over-estimation. Lastly, the value of γ_{t+1} shrinks over time when there is no weak rule with the larger edge exists.

Sampler

Our assumption is that the entire training dataset does not fit into the main memory and is therefore stored in an external storage **(a)**. As boosting progresses, the weights of the examples become increasingly skewed, making the dataset in memory effectively smaller. To counteract

that skew, Sampler prepares a *new* training set, in which all of the examples have equal weights, by using selective sampling. When the effective sample size n_{eff} associated with the old training set becomes too small, the scanner stops using the old training set and starts using the new one[1].

The sampler uses selective sampling by which we mean that the probability of an example (x, y) being added to the sample is proportional to its weight $w(x, y)$. Each added example is assigned an initial weight of 1. There are several known algorithms for selective sampling. The best known one is rejection sampling in which a biased coin is flipped for each example. We use a method known as *minimal variance sampling* [Kit96] because it produces less variation in the sampled set.

Stratified Storage and Stratified Sampling

The standard approach to sampling reads examples one at a time, calculates the weight of the example, and accepts the example into the memory with the probability proportional to its weight, otherwise rejects the example. Let the largest weight be w_{max} and the average weight be w_{mean}, then the maximal rate at which examples are accepted is $w_{\text{mean}}/w_{\text{max}}$. If the weights are highly skewed, then this ratio can be arbitrarily small, which means that only a small fraction of the evaluated examples are then accepted. As evaluation is time consuming, this process becomes a computation bottleneck.

We proposed a stratified-based sampling mechanism to address this issue (the right part of Figure 6.1). It applies incremental update to reduce the computational cost of making prediction with a large model, and uses a stratified data organization to reduce the rejection rate.

To implement incremental update we store for each example, whether it is on disk or in memory, the result of the latest update. Specifically, we store each training example in a tuple (x, y, H_l, w_l), where x, y are the feature vector and the label, H_l is the last strong rule used to calculate the weight of the example, and w_l is the weight last calculated. In this way both the

[1]The sampler and scanner can run in parallel on a multi-core machine, or run on two different machines. In our experiments, we keep them in one machine.

scanner and sampler only calculate over the incremental changes to the model since the last time it was used to predict examples.

To reduce the rejection rate, we want the sampler to avoid reading examples that it will likely to reject. We organize examples in a stratified structure, where the stratum k contains examples whose weights are in $[2^k, 2^{k+1})$. This limits the skew of the weights of the examples in each stratum so that $w_{\text{mean}}/w_{\text{max}} \leq \frac{1}{2}$. In addition, the sampler also maintains the (estimated) total weight of the examples in each strata. It then associates a probability with each stratum by normalizing the total weights to 1.

To sample a new example, the sampler first samples the next stratum to read, then reads examples from the selected stratum until one of them is accepted. For each example, the sampler first updates its weight, then decides whether or not to accept this example, finally writes it back to the stratum it belongs to according to its updated weight. As a result, the reject rate is at most $1/2$, which greatly improves the speed of sampling.

Lastly, since the stratified structure contains all of the examples, it is managed mostly on disk, with a small in-memory buffer to speed up I/O operations.

Chapter 2 through Chapter 7 contain material from the Conference on Advances in Neural Information Processing Systems, 2019 (“Faster Boosting with Smaller Memory,”, Alafate and Freund). The dissertation author was a primary investigator and author of this paper.

Chapter 7

Experimental results

In this section we describe the experiment results of **Sparrow**. In all experiments, we use trees as weak rules. First we use the forest cover type dataset [GRM03] to evaluate the sampling effectiveness. It contains 581 K samples. We performed a 80/20 random split for training and testing.

In addition, we use two large datasets to evaluate the overall performance of Sparrow on large datasets. The first large dataset is the splice site dataset for detecting human acceptor splice site [SF10, ACDL14]. We use the same training dataset of 50 M samples as in the other work, and validate the model on the testing data set of 4.6 M samples. The training dataset on disk takes over 39 GB in size. The second large dataset is the bathymetry dataset for detecting the human mislabeling in the bathymetry data [JAM16]. We use a training dataset of 623M samples, and validate the model on the testing dataset of 83M samples. The training dataset takes 100 GB on disk. Both learning tasks are binary classification.

The experiments on large datasets are all conducted on EC2 instances with attached SSD storages from Amazon Web Services. We ran the evaluations on five different instance types with increasing memory capacities, ranging from 8 GB to 244 GB (for details see Section 7.3).

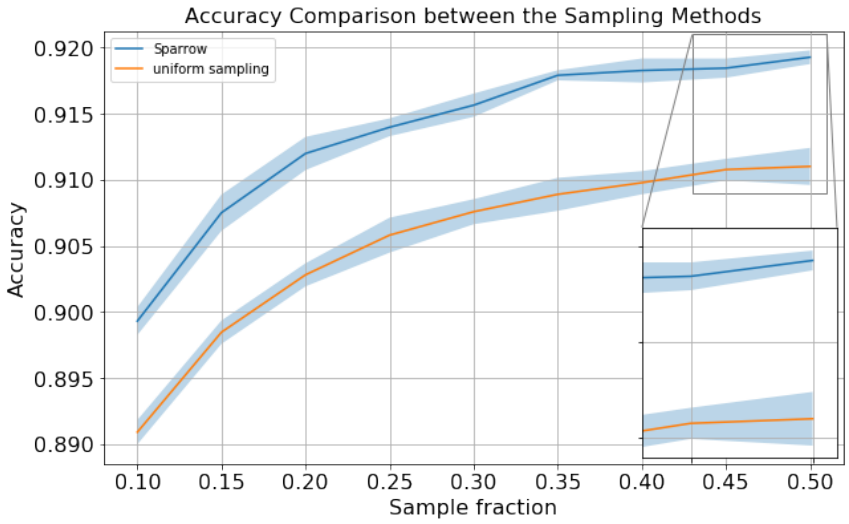

Figure 7.1: Accuracy comparison on the CoverType dataset. For uniform sampling, we trained XGBoost on a uniformly sampled dataset with the same sample fraction set in Sparrow. The accuracy is evaluated with same number of boosting iterations.

7.1 Effectiveness of the weighted sampling

We evaluate the effectiveness of weighted sampling by comparing it to uniform sampling. The comparison is over the model accuracy on the testing data when both trained for 500 boosting iteration on the cover type dataset. For both methods, we generate trees with depth 5 as weak rules. In uniform sampling, we first randomly sample from the training data with each sampling ratio, and use XGBoost to train the models. We evaluated the model performance on the sample ratios ranging from 0.1 to 0.5, and repeated each evaluation for 10 times. The results are showed in Figure 7.1. We can see that the accuracy of Sparrow is higher with the same number of boosting iteration and same sampling ratio. In addition, the variance of the model accuracy is also smaller. It demonstrates that the weighted sampling method used in Sparrow is more effective and more stable than uniform sampling.

7.2 Training on the large datasets

We compare Sparrow on the two large datasets, and use XGBoost and LightGBM for the baselines since they out-perform other boosting implementations [CG16, KMF$^+$17]. The comparison was done in terms of the reduction in the exponential loss, which is what boosting minimizes directly, and in terms of AUROC, which is often more relevant for practice. We include the data loading time in the reported training time.

There are two popular tree-growth algorithms: depth-wise and leaf-wise [Shi07]. Both **Sparrow** and LightGBM grow trees leaf-wise. XGBoost uses the depth-wise method by default. In all experiments, we grow trees with at most 4 leaves, or depth two. We choose to train smaller trees in these experiments since the training take very long time otherwise.

For XGBoost, we chose approximate greedy algorithm which is its fastest training method. LightGBM supports using sampling in the training, which they called *Gradient-based One-Side Sampling* (GOSS). GOSS keeps a fixed percentage of examples with large gradients, and randomly sample from remaining examples. We selected GOSS as the tree construction algorithm for LightGBM. In addition, we also enabled the `two_round_loading` option in LightGBM to reduce its memory footprint.

Both XGBoost and LightGBM take advantage of the data sparsity for further speed-up training. Sparrow does not deploy such optimizations in its current version, which puts it in disadvantage.

The memory requirement of **Sparrow** is decided by the sample size, which is a configurable parameter. XGBoost supports external memory training when the memory is too small to fit the training dataset. The in-memory version of XGBoost is used for training whenever possible. If it runs out of memory, we trained the model using the external memory version of XGBoost instead. Unlike XGBoost, LightGBM does not support external memory execution. However, LightGBM uses various techniques to optimize the representation of data features,

which improves the memory efficiency. For example, it uses a technique called *Exclusive Feature Bundling*. This technique merges two feature dimensions into one if they are *exclusive* to each other, meaning that they never take nonzero values simultaneously. As a result, LightGBM might be able to train a model in some cases even if the memory size is smaller than the *original* training data size. Nevertheless, if the memory size is too small to even fit the optimized feature representation, LightGBM could still run out of memory and crash.

As for the stopping rule in Sparrow, we follow the definition in Equation 4.4, and set $C = 1$ and $\sigma = \frac{0.001}{|\mathcal{H}|}$, where $\mathcal{H}$ is the set of base classifiers (weak rules).

Lastly, all algorithms in this comparison optimize the exponential loss as defined in AdaBoost.

7.3 Evaluate Sparrow on the large datasets

The experiments on large datasets are all conducted on EC2 instances with attached SSD storages from Amazon Web Services. We ran the evaluations on five different instance types with increasing memory capacities, specifically 8 GB (`c5d.xlarge`, costs $0.192 hourly), 15.25 GB (`i3.large`, costs $0.156 hourly), 30.5 GB (`i3.xlarge`, costs $0.312 hourly), 61 GB (`i3.2xlarge`, costs $0.624 hourly), and 244 GB (`i3.8xlarge`, costs $2.496 hourly).

We evaluated each algorithm in terms of (1) the AUROC as a function of training time, and (2) the decrease of the exponential loss as a function of training time, both evaluated on the testing data.

The evaluations in terms of AUROC are given in Figure 7.2 and Figure 7.3.

On the splice site dataset, **Sparrow** is able to run on the instances with as small as 8 GB memory. The external memory version of XGBoost can execute with reasonable amount of memory (but still needs to be no smaller than 15 GB) but takes about 3x longer training time. However, we also noticed that **Sparrow** does not have an advantage over other two boosting

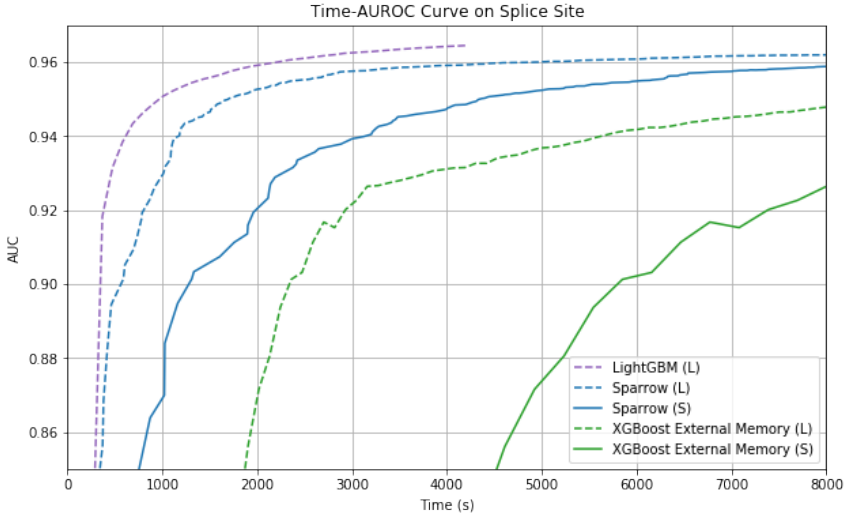

Figure 7.2: Time-AUROC curve on the splice site detection dataset, higher is better, clipped on right and bottom The (S) suffix is for training on 30.5 GB memory, and the (L) suffix is for training on 61 GB memory.

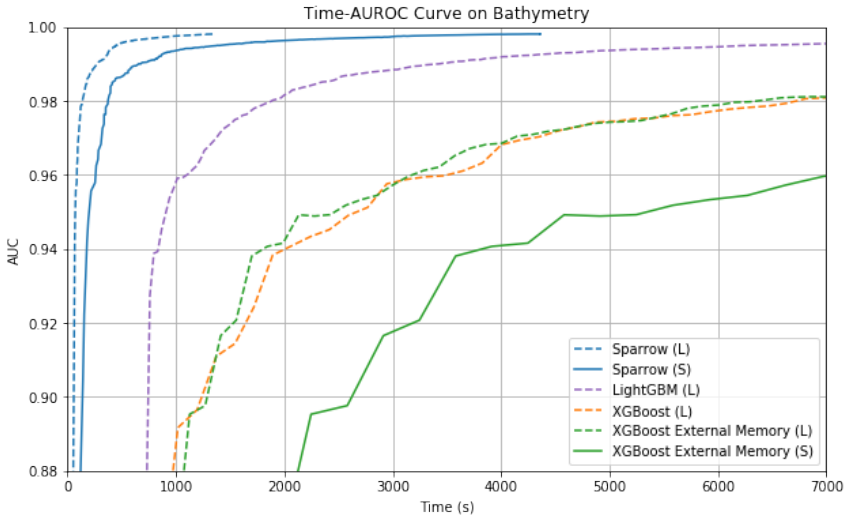

Figure 7.3: Time-AUROC curve on the bathymetry dataset, higher is better, clipped on right and bottom. The (S) suffix is for training on 61 GB memory, and the (L) suffix is for training on 244 GB memory.

Table 7.1: Training time (hours) on the splice site dataset. The (m) suffix is trained in memory. The (d) suffix is trained with disk as external memory.

Training time until the loss convergences

Memory	Sparrow	XGB	LGM
8 GB	**2.9** (d)	OOM	OOM
15 GB	**8.4** (d)	$>$ 50 (d)	OOM
30 GB	**10.4** (d)	0.6 (d)	OOM
61 GB	4.4 (d)	12.8 (d)	**1.2** (m)
244 GB	1.3 (d)	1.1 (m)	**0.5** (m)
Converged	0.057	0.055	**0.053**

Training time until the average loss reaches 0.06

Memory	Sparrow	XGB	LGM
8 GB	**1.4** (d)	OOM	OOM
15 GB	**7.1** (d)	$>$ 50 (d)	OOM
30 GB	**2.3** (d)	9.3 (d)	OOM
61 GB	1.3 (d)	4.6 (d)	**0.3** (m)
244 GB	0.5 (d)	0.3 (m)	**0.2** (m)

Table 7.2: Training time (hours) on the bathymetry dataset. The (m) suffix is trained in memory. The (d) suffix is trained with disk as external memory.

Training time until the loss convergences

Memory	Sparrow	XGB	LGM
8 GB	The disk cannot fit the data		
15 GB	**2.5** (d)	OOM	OOM
30 GB	**1.9** (d)	41.7 (d)	OOM
61 GB	**1.2** (d)	38.6 (d)	OOM
244 GB	**0.4** (d)	20.0 (m)	4.0 (m)
Converged	**0.046**	0.054	0.054

Training time until the average loss reaches 0.06

Memory	Sparrow	XGB	LGM
8 GB	The disk cannot fit the data		
15 GB	**1.0** (d)	OOM	OOM
30 GB	**0.6** (d)	41.7 (d)	OOM
61 GB	**0.6** (d)	38.4 (d)	OOM
244 GB	**0.2** (d)	16.9 (m)	3.3 (m)

implementations when the memory size is large enough to load in the entire training dataset.

On the bathymetry dataset, **Sparrow** consistently out-performs XGBoost and LightGBM, even when the memory size is larger than the dataset size. In extreme cases, we see that **Sparrow** takes 10x-20x shorter training time and achieves better accuracy. In addition, both LightGBM and the in-memory version of XGBoost crash when trained with less than 244 GB memory.

We observed that properly initializing the value of γ and setting a reasonable sample set size can have great impact on the performance of **Sparrow**. If stopping rule frequently fails to fire, it can introduce a significant overhead to the training process. Specific to the boosted trees, one heuristic we find useful is to initialize γ to the maximum advantage of the tree nodes in the previous tree. A more systematic approach for deciding γ and sample set size is left as future work.

In Table 7.1 and Table 7.2, we compared the training time it takes to reduce the exponential loss as evaluated on the testing data. Specifically, we compared the values of the average loss when the training converges and the corresponding training time. In addition, we observed that the average losses converge to slightly different values, because two of the algorithms in comparison, Sparrow and LightGBM, apply sampling methods during the training. Therefore, we also compared the training time it takes for each algorithm to reach the same threshold for the average loss.

We use "XGB" for XGBoost, and "LGM" for LightGBM in the tables. In addition, we observe that the training speed on the 8 GB instances is better than that on 15 GB instances, because the 8 GB instance has more CPU cores than the 15 GB instance.

We concluded that, in terms of reducing the exponential loss, Sparrow converges significantly faster than XGBoost when the memory is not sufficient to fit the training data. In these cases, LightGBM crashes after running out of memory. When the memory is sufficiently large, Sparrow converges much faster than Sparrow, but slower than LightGBM.

Chapter 2 through Chapter 7 contain material from the Conference on Advances in Neural Information Processing Systems, 2019 ("Faster Boosting with Smaller Memory,", Alafate and Freund). The dissertation author was a primary investigator and author of this paper.

Part II

Learning from heterogenous data

Chapter 8

When randomly splitting is not enough

One of the basic assumptions in machine learning is that data is generated independently at random from a fixed but unknown distribution. We refer to this as the IID ("independently and identically distributed") assumption. The IID assumption justifies the common practice of splitting a dataset at random into a training set (RTrain) and a test set (RTest), we train the model on RTrain, and report its performance on RTest.

Reporting the test error on RTest is generally accepted as a reliable criteria for comparing the accuracy of different models. However, in some real world applications, performance on RTest is sometimes a poor predictor of the performance on a new test sample (NTest), which is collected in a related, but different environment. We identify two reason for the difference:

- **Domain shift:** In some applications of machine learning, RTest and NTest are drawn according to different distributions. For example, suppose we train a classifier to classify people from a photo of their faces. Suppose a large dataset is collected and split into RTrain and RTest, suppose further that the performance on RTest is good. The performance on a new dataset, taken several months after training of the classifier, might be significantly poorer. It may due to the changes of haircut, the changes of lighting, or capturing by a different camera. This problem is often referred to as "domain shift" and the solutions for

it are called “transfer learning”. [Ng16] declared transfer learning as an important open problem.

- **Spurious correlations:** Datasets sometimes contain spurious correlations which appear in RTrain and RTest but do not carry over to NTest.

One of the conceptual foundations of machine learning is the randomized train-set/test-set methodology (RTTM). This methodology dictates a randomized method for creating training testing sets. A large dataset is collected and then partitioned randomly into a training set and a test set. The classifier is trained on the training set and then tested on the test set. The main contribution of this paper is the identification of a fundamental problem with RTTM and the proposal of an approach for addressing this problem.

Most of the theory of machine learning is based on RTTM. Specifically, we assume that the training set and the test set are both drawn IID from the same distribution. Under this assumption the train error and the test error of a classifier are unbiased estimates of the same quantity: the generalization error.

However, in practice often one cannot justify this assumption. To make this concrete, consider a traffic sign classifier that is to be used as part of a self-driving car. Suppose we train our classifier on data that was collected in a hundred different locations. How should we collect a test set? If we collect the test set video from the same one hundred location, then our accuracy guarantees apply only to the same one hundred locations. If we want our guarantees to apply to other locations, we must use data collected from those locations. In this case the training set and the test set are likely to have different distribution. Making most theoretical results inapplicable. We call this the *data diversity problem*.

There is another problem with RTTM, which relates to the fact that videos consist of frames, and neighboring frames are usually very similar to each other. Suppose that we have a one second stretch of a video in which a stop sign appears in view. The thirty or so frames

corresponding to that view are likely to be very similar to each other. Now suppose 15 of these frames appear in the training set and the other 15 in the test set. The images in the training set would be labeled as "containing stop sign". As the images in the test set are very similar, simple memorization based methods, such as k-nearest-neighbors, are likely to perform well on this task. We call this problem the *data sequentiality problem*

In this chapter we demonstrate the data sequentiality and the data diversity problem in a particular large scale study. We then describe a heuristic we used that can reduce the data diversity problem.

8.1 The standard framework of machine learning

The standard framework of machine learning is as follows.

Let $X \in \mathcal{X}$ be the input features, and $y \in \mathcal{Y}$ be the labels. The data instance (X, y) are independent and identically distributed (i.i.d.) random variables from a fixed but unknown joint distribution $\mathcal{D}$ over $X \times \mathcal{Y}$. Let $h : \mathcal{X} \to \mathcal{Y}$ be the candidate hypothesis from some hypothesis set $\mathcal{H}$. The objective of learning is to find the optimal hypothesis h^* that minimizes the *true* error:

$$\mathrm{err}_{\mathcal{D}}(h) \doteq P_{(X,y)\sim\mathcal{D}}\left[L(y, h(X))\right], \tag{8.1}$$

where $L(y, h(X))$ is the loss function used to measure the difference between the the true label y and the predicted label $h(X)$.

The learning algorithm is given two sets of i.i.d. data instances, a training set $\mathcal{T} = \{(X_1, y_1), \ldots, (X_n, y_n)\}$ and a test set $\mathcal{U} = \{X_1, y_1), \ldots, (X_m, y_m)\}$. The objective of the learning algorithm is to find the hypothesis $\widehat{h}$ that minimizes the *training error*,

$$\widehat{\mathrm{err}}(h; \mathcal{T}) \doteq \frac{1}{n} \sum_{i=1}^{n} L(y_i, h(X_i)), \qquad (X_i, y_i) \in \mathcal{T} \tag{8.2}$$

If the training set size is small and/or the hypothesis set size is large, the algorithm may find a hypothesis that *overfits* the training set, in which case the true error of $\widehat{h}$ is likely to be significantly larger than the true error of h^*. To estimate the true error of $\widehat{h}$, the algorithm calculates the *generalization error* $\widehat{\mathrm{err}}(\widehat{h}; \mathcal{U})$ on the test set (also referred to as the *test error*). Under the standard framework, the test error is an unbiased estimate of the true error and can be used to compare different learning algorithms.

In this chapter, we discuss the workflow of a machine learning project in two phases: the training phase and the production phase. The training phase includes training the model on $\mathcal{T}$ and evaluating its performance on $\mathcal{U}$. The production phase starts after the model is deployed into production. The error rate in the production phase is evaluated on the newly collected data instances that are unseen in the training phase. In the light of the i.i.d. assumption, the error rate in production is expected to be close to the test error.

Randomized train/test split

The *de facto* standard for generating the training set $\mathcal{T}$ and test set $\mathcal{U}$ is the randomized train/test split (RTTS) method. Starting with a sufficiently large dataset, one randomly divides the dataset into two to three parts: a training set $\mathcal{T}$, a test set $\mathcal{U}$, and sometimes a validation set $\mathcal{V}$. There is no agreed upon way for selecting the relative proportion of the three splits. [HTF09] suggests reserving 50% of the dataset for $\mathcal{T}$, and 25% each for $\mathcal{V}$ and $\mathcal{U}$. Note that for the sake of simplicity, we will assume only a training set and a test set and no validation set in this chapter.

The standard framework is at the heart of machine learning theory and practice. The assumption on how one collects the data instances (i.e. the i.i.d. assumption) is usually taken for granted. However, there are at least two ways in which practical data collection fails this assumption: the data diversity problem and the spurious correlation problem. In next two sections, we first explain these two problems with examples, then discuss them in the context of experimental design in machine learning.

8.2 Two problems of the standard framework

In this section we describe two ways in which common data collection procedures are inconsistent with the standard framework. The standard framework assumes that the examples are *independent and identically distributed* (i.i.d.). These inconsistencies, in turn, create biases in the error estimates used in RTTS. We present the inconsistencies in the two following subsections: in Section 8.2.1 we describe the problems in the *identically distributed* assumption; in Section 8.2.2 we describe the problems with the *independently distributed* assumption.

8.2.1 Data diversity problem

The purpose of the test set in RTTS is to provide an estimate of the error rate of the learned model when deployed "in the wild". For this estimate to be unbiased, the test set should be *representative* of the "in-the-wild" distribution. However, collecting a representative set of examples is often hard, as described below.

A data-set is representative if it has the same distribution as that of data collected "in the wild". When the data is low dimensional, it is easy to ensure that a dataset is representative. In practice, many applications are based on very high dimensional data, such as images or video. The diversity of inputs is so large that collecting a *representative sample* is difficult.

Consider collecting a dataset for training an autonomous driver. The variables in the context of driving include: geographical locations[1], driving conditions[2], traffic conditions[3], different time of the day[4], seasonal and weather conditions[5]. Ideally, we assume the dataset should be both sufficiently large and sufficiently diverse so that it covers the various *combinations* of the variables above. The diversity of the data instances depends on what data sources are

[1]For example, different states, cities, towns.
[2]For example, highways, residential areas, primitive roads.
[3]For example, peak hours and off-peak hours, with and without the presence of pedestrians.
[4]For example, dawn, daytime, dusk, and nighttime.
[5]For example, spring, winter, rainy days, snowy days.

Figure 8.1: The imaginary from the highway Camera mounted over California State Route 1, randomly sampled from a continuous 2-minute video clip.

considered in the data collection step.

Contrast the image samples of the video streams from three cameras on the road. The stream A (Figure 8.1) is generated by a fixed camera monitoring a segment of California State Route 1. The stream B (Figure 8.2) is generated by the Google Street View project ([ADF+10]), from which the images are sampled as follows: select a geographic coordinate in United States uniformly at random, collect the imagery of Google Street View at that location if it exists, otherwise select another random coordinate. The stream C (Figure 8.3) is generated by the video cameras mounted on the cars from the San Francisco Bay Area and New York City ([YXC+18]).

In the stream A, the diversity is limited, only a small part of the frame changes from moment to moment, such as the cars driving in the lanes. In the stream B, the diversity is higher, as we start to explore different *types* of roads located at different parts of the country. We also observe the differences in weather conditions. However, these roads are mostly empty, likely located in the less populated areas. In the stream C, the diversity is very high. The driving conditions are complicated with the presence of other vehicles and pedestrians. The images also

Figure 8.2: The images sampled from Google Street View by repeatedly selecting a geographic coordinates in United States until there is an image taken at that location by Google Street View.

Figure 8.3: The samples from BDD100K selected uniformly at random.

cover different time of the day.

Along with many other applications of machine learning, the driving datasets are drawn from a *long tail* distribution. It requires careful experimental design to collect a diverse dataset. For example, if the learning task involves detecting traffic signs, the data instances solely from the stream A or the stream B are not diverse enough because there are few images that contain the traffic signs mostly seen on the city streets. On the other hand, if the learning task involves detecting lanes or the edges of the roads, the data instances solely from the stream A or the stream C are not diverse enough because there are few images with the primitive roads and the roads located in rural areas.

Another challenge in these applications is that it is hard to acquire samples that represent the further end of the long tail. In the context of driving, it may be very *unusual* situations, such as the extreme weathers (e.g. heavy fog) and unexpected pedestrian behaviors (e.g. pedestrian walking on the highway).

As we see in these examples, if the training data is not diverse enough, then simply increasing the size of the training data will not fundamentally improve the model performance in the production phase. It is because the model has little prediction power over the data instances generated by the data sources that are drastically different from those seen in training.

In the standard framework, the data diversity problem is not a concern because the data instances are assumed to be drawn identically from a fixed distribution. In practice, the data instances are often drawn from multiple data sources each potentially with different distributions. The RTTS method addresses the presence of multiple data sources by shuffling the data instances before splitting them into the training set $\mathcal{T}$ and the test set $\mathcal{U}$, so that the distributions that generate $\mathcal{T}$ and $\mathcal{U}$ are identical. However, the data shuffling does not effectively improve the data diversity. Thus, the model trained on $\mathcal{T}$ cannot predict well for the data instances from previously unseen data sources. Consequently, the generalization error evaluated on $\mathcal{U}$ is no longer a reliable estimate for the model performance in the production phase, which is a more critical measurement

for most practical machine learning projects. Instead, we will see a change in the distributions from the training set to the data instances collected in the production phase. This phenomena is referred as *distribution shift*. Many algorithmic solutions are proposed to mitigate distribution shift (see [KL18]).

The data diversity problem is not unique to machine learning. It is generally observed in the scientific studies that use experiments and data to validate theories. [Lan88] examines machine learning in the lens of experimental science. By definition, an experiment involves observing the behavior of a system while varying one or more of its components and keeping others constant. The methodology enables us to study the interactions between the variables in a system. Designing controlled experiments is straightforward with a few independent variables. However, in practical problems, controlling all variables are infeasible. As a result, the recommended method is to collect observations from each cell in experimental design, and justify the differences of the results between cells using the data. In the context of machine learning, it means one should investigate a diverse assembly of data sources that are systematically selected.

The diversity in data sources is presence in many datasets in which the instances could be partitioned into subpopulations. For instance, consider the following two examples.

- **Medical image segmentation** Data diversity is a major challenge in deploying learning-based model in medical image segmentation to new environments ([dB16, VOIVDB14]). The model performance is promising in a controlled, standardized environment. However, the accuracy decreases when the model is deployed in a new environment (e.g. a new hospital), due to the variance in imaging protocol, scanner model, or different patient population. In this example, we can see that the data distribution may change from one location to another.

- **Advertising clicks prediction** Predicting advertising clicks is central to most online advertising system. [HPJ+14a] has observed in practice that the prediction accuracy of the

model decreases over time. The decreases is caused by the change in the behaviors of the advertisers and the website users over time. Such changes cannot be captured by a fixed model trained on historical data. In this example, the data instances are from a non-stationary distribution. The distribution changes from one period of time to the next.

8.2.2 Spurious correlation problem

The power of machine learning methods is in their exceptional capability of discovering the statistical correlations between the data features and the data labels. In many cases, there is no cause-effect relationship between the two variables that are correlated. This type of correlations is referred as *spurious correlations* in statistics ([Sim77]). The spurious correlations exist when there is a third random variable Z that is a common cause of the two random variables that are correlated. The common cause Z is also referred as the *confounding factor*.

An assumption in machine learning is that if the training set is sufficiently large, the correlations discovered in the training set are likely to be persist over the population. We argue that this conclusion is not guaranteed in practice because of the mutual dependencies between the training instances.

Under the standard framework, we assume the data instances in the training set are mutually independent. In practice, this assumption is difficult to satisfy for many reasons. One of them is that these data instances are collected by the same mechanism, meaning that they are from the same data sources, collected in some fixed order, or other reasons that could introduce a common causal factor between the data features and the data labels. If the common cause factor only associates with the data collection process but does not exist across the population in general, the correlations due to these causal factors are unlikely to exist beyond the training set.

Data duplications and near-duplications are among the most straightforward examples of spurious correlation in the dataset. For instance, [BD19] finds that 10% of the images from the CIFAR-100 test sets have duplicates in the training set, while the ratio is 3.3% for CIFAR-10

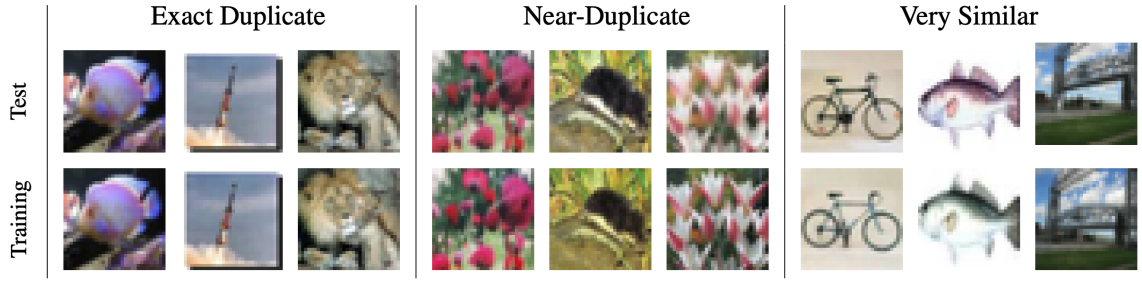

Figure 8.4: The example is from [BD19]. Samples of different types of duplicates discovered in CIFAR-100 test and train sets. The top row shows images from the test set. The bottom row shows the corresponding similar images in the training set. Exact duplicate images are identical almost pixel-by-pixel. Near-duplicates are same images that post-processed differently. Very similar images contain different contact but look highly similar.

(see Figure 8.4). In ImageNet ([DDS$^+$09]), 1.78% of the validation images have near-duplicate images in the training set ([SSSG17]), even though the benchmark dataset has gone through de-duplication.

These large-scale image datasets are collected by crawling images from popular image search engines, such as Google and Flickr, using a set list of keywords assembled from sources like WordNet ([Mil98]). Unavoidably, some of the search query results contain duplicate images for reasons like same images being uploaded by more than one user or being used on more than one webpages. De-duplicating is not trivial in high dimension. Two data instances may not be exact, byte-by-byte match to each other, but still be very similar to each other before or after the feature extraction. As a result, some duplicate instances end up in these datasets.

Not all data dependencies are in the form of data duplication. [NMK19] studies the spurious correlation between the training set and the test set for the image segmentation algorithms on the satellite data. As it shows in Figure 8.5, the training set and the test set are generated using the RTTS method over the random selections drawn from a fixed region on map. Some of these selections could have overlapping pixels, which essentially gives away the labeling of the overlapped region for the corresponding selections appeared in the test set. As we will see in Section 9, in some cases the selections does not even need to be overlaps, but merely to be close

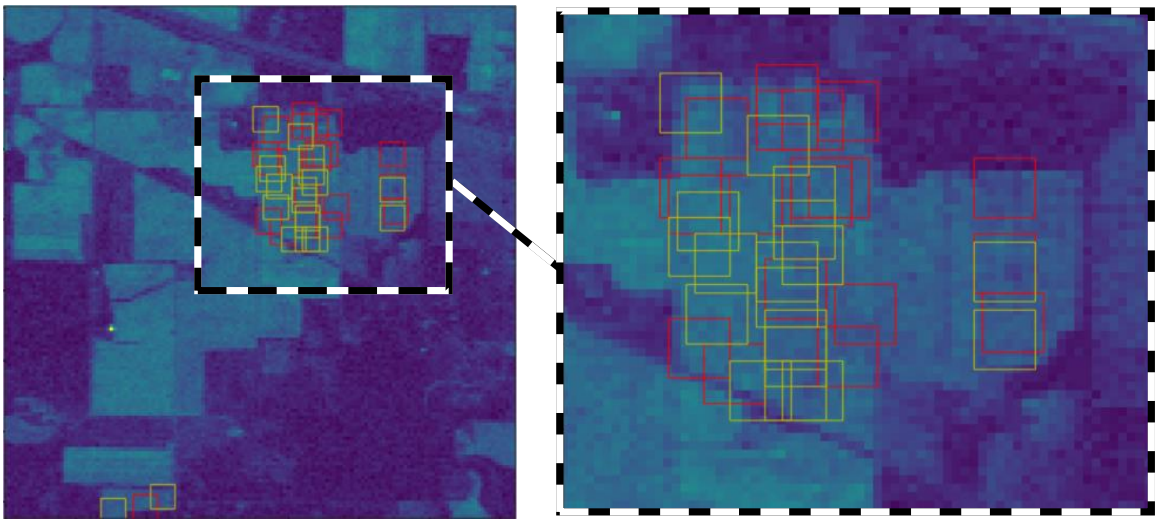

Figure 8.5: The example is, with permission, from [NMK19]. The left figure shows the map from Indian Pines, while the right figure is zoom into the region of interest. Random selections are drawn from the region of interest, and then split into the training set and the test set, denoted by the yellow are red squares, respectively.

enough in distance, to introduce spurious correlations.

Another source of the spurious correlation is the common underlying properties that should not be considered in the prediction rules because they are inaccessible at the time of inference. For example, it is usually required to create both the temporal boundary and the entity-level boundary when working with the financial datasets. For example, when training a classifier for fraud detection using a dataset of credit card transactions, the data instances should be split into the training set and the test set in a way such that (1) all transactions in the training set happened before all transactions in the test set, and (2) the transactions belong to the same account holder should appear in either the training set or the test set but not both. The first requirement is to eliminate the mutual dependencies between the consecutive transactions. For example, a malicious party may attempt to put several purchases consecutively on the stolen credit cards before the accounts get frozen. The second requirement is to eliminate the mutual dependencies between the transactions due to the economic patterns of a specific account holder. For example, some credit card owners may have never experienced fraudulent charges. The model may learn

specific patterns that could identify such owners, and predict all future transactions from their cards as normal.

In all the examples that have spurious correlation, we could use simple memorization-based method and perform well on the test instances that have duplicates in the training set. However, these prediction rules cannot be generalized to the population, thus have little utility in application.

In the context of predictive modeling, spurious correlation is a type of information leakage. [RPŚ+10] describes the information leakage as the "unintentional introduction" of the predictive information. Similar to other forms of information leakage, spurious correlation leads to an over-estimation of the model performance based on the performance on the test set. It is especially problematic when the learning model has a strong "memorization" capacity. For example, [BD19] finds that, after removing the duplicate and near-duplicate images, the error rate of the state-of-the-art model increases from 3.56% to 3.95% on CIFAR-10, and increases from 17.05% to 19.38% on CIFAR-100.

8.3 Experimental design in machine learning

The standard framework described in Section 8.1 deviates from the real-world machine learning projects on how the data instances are collected. In this section, we discuss an alternative framework that better fits the actual data collection process. We are not attempting to propose the resolutions for the data diversity problem and the spurious correlation problem, but to explore possible future work directions.

8.3.1 Collect data from environments

Under the standard framework, all data instances are drawn from a fixed distribution. However, as showed in the examples described in this chapter, the data distributions are often not

identical in training and in testing, because the data sources have changed.

Alternatively, we should treat each data source as a distinct *environment* from which we can draw data instances. Each environment e is governed by its own distributions $\mathcal{D}_e$ over (X, y). As a result, when we switch the environment from training to testing, we could see a distribution shift. In the driving example, the camera mounted over the vehicle could capture images from multiple environments, e.g. driving in the cities and driving in the rural areas.

Informally, we assume the data is generated by a two-step hierarchical model: first an environment e is drawn, then the data instances are drawn from its corresponding distribution $\mathcal{D}_e$.

In practice, we have no control over the data sources that would be used in the production time. In other words, the testing environment of a model is likely to be different from its training environment. This is very different from the assumption in the standard framework, where the training environment and the testing environment is identical statistically. In the design of the machine learning experiments, we should reflect this challenge.

8.3.2 Experimental design in machine learning

Machine learning is an experimental science ([Lan88]). By definition, an experiment involves observing the behavior of a system while varying one or more of its components and keeping others constant. The methodology enables us to study the interactions between different variables.

Comparing to other fields of science, it is easier to design control experiments in machine learning because the practitioners have complete control over the data sources (while in training) and the learning algorithms.

A basic controlled experiment can be designed as follows. Given a set of k data sources, we have k separate environments $E = \{e_1, e_2, \ldots, e_k\}$ by treating the instances from each data sources as from separate environments. Collect separate pairs of the training set $\mathcal{T}_i$ and the test set $\mathcal{U}_i$ from each environment e_i. Then with each training set $\mathcal{T}_i$, we can train a model h_i. Finally

for each trained model h_i, we can test its performance on all environments using all the tests set $\mathcal{U}_j, j = 1, \ldots, k$.

Generally, we expect to see that the model h_i performs best in the environment e_i and vice versa. More interesting results, however, are its performance on all other environments. The justification is to by using examples from separate environments, we are likely to better estimate the model performance in a non-identical environment, and also reduce or even eliminate the boost in model performance due to the spurious correlation embedded in the model.

8.3.3 Model ensemble and active learning

Under the standard framework, given k data sources, we collect examples from all of them and generate one training set and one test set using the RTTS method. If, instead, we follow the proposal in Section 8.3.2, we can create a model ensemble with k distinct models. We think the ensemble of k model is a better fit for the practical learning tasks because it enables us to measure the prediction confidence.

Evaluating the prediction confidence is more feasible with an ensemble than with a single model. For example, in a binary classification task with $0/1$ label, we would be very confident in the model prediction if all k models predict the same, while we would be uncertain if $k/2$ of the models predict 0, and the other $k/2$ predict 1. When the model is uncertain about its prediction, it can abstain from prediction, which is a safer approach for many applications where the cost incorrect inference is high.

Delegate when uncertain

In many high-stake applications of machine learning, such as autonomous driving and medical diagnoses, it is an over-reaching goal to replace the human experts entirely with computers. Instead, we should treat computers as an enhancement to human intelligent. Specifically, the models should make prediction only if it has a high confidence. For complicated examples on

which the models are uncertain, it should print “I don’t know” and delegate the prediction task to a human expert. The hybrid approach could reduce the risk of making high-cost mistakes, while still benefits from the automation.

Active learning

The key idea of active learning is that the model can make improvement at lower cost by selectively choosing which data instances to use for training. It is best suited in the scenarios where it is cheap to acquire a large amount of data instances, but expensive to label them. It is suited in the context of ensemble learning for the following reasons. The learning models are expected to perform well on “easy examples”. One way to define easiness of an example is by how likely the similar examples may be appear in the training set. The models are likely to perform well on the most commonly seen examples, but less confident on the rarely seen ones. Many learning tasks work with a long tail distribution, which means a vast pool of the data instances may not be collected in training but appear in testing. For example, it is rare but possible that pedestrians may be walking on the highway. In the data collection for model training, it is unlikely that such scenarios would be captured. But when the model is deployed at scale, the probability of encountering these rare cases is significant. These instances can be labeled by human experts, and later be added to the training set for improving the model.

8.4 Related work

8.4.1 Randomized train/test split

The Randomized train/test split (RTTS) method is used widely across the academic research and the real-world applications of machine learning. In the academic community, we rely on the performance reported on the test set generated by RTTS to decide which algorithms/architectures are the best. On the application side, RTTS is a default step in the data

preprocessing, as seen in the popular machine learning packages such as scikit-learn ([PVG+11]) and TensorFlow ([ABC+16]).

The RTTS method is justified only if the data is drawn under the i.i.d. assumption. [DKP+15] identifies three reasons why the i.i.d. assumption could be violated in practical applications: (1) the distribution can be non-stationary and change over time; (2) consecutive data samples can be inter-dependent; (3) the data samples are generated by an adversary, instead of a distribution.

8.4.2 Learning from multiple environments

We refer to the term *environment* to recognize that the data instances may be drawn from separate, distinct data sources. This concept has been previously studied in [ABGLP19], [PBM16], and [HDM17], which aim to identify the invariant features or rules across different environments, and to show that the invariants are related to the causal explanation of the learning objectives. The end product of these work is a single model that generalizes well in all varieties of the environment. We propose a different learning objective in Section 8.3, which uses an ensemble approach, training multiple models that fit best in each environment.

8.4.3 Distribution shift

In practice, the training environment is often different from the testing environment. As a result, machine learning practitioners often observe the statistical difference at various scale between the environment that the model is trained, and the environment that the model is to be applied, i.e. a distribution shift.

Distribution shift, also been referred as data shift, is a general term to describe the discrepancy between the two distributions $\mathcal{D}_S$ and $\mathcal{D}_T$: $\mathcal{D}_S$ generates the examples for training the model, and $\mathcal{D}_T$ generates the examples that we want to make predictions on. The discrepancy

causes the model to make bias estimations on the examples from $\mathcal{D}_{\mathcal{I}}$. There are a number of approaches to address the distribution shift.

Transfer learning and domain adaptation

In transfer learning and domain adaptation, the training data is sampled from the source domain, while the fresh examples are sampled from the target domain. The data distribution shifts from one domain to the other. [KL18] describes the three types of data shift that happens, namely priority shift, covariate shift, and concept shift. Except for covariate shift, data labels are required to fit the model to the new domain.

Exploit invariance and causality

[HDM17, ABGLP19] proposed methods that prompt the invariant part (or core features) of the data in the model training process. The term *environment* is used to describe the distributions where the data can be sampled from. The idea is that by focusing on the data properties that persist across all environments, the model can fit the invariant casual associations between the variables, and generalize better to the fresh examples from an unseen distribution that shares these properties. For example, [ABGLP19] introduce a regularizer to penalize the feature representations that cannot generalize well in different environments.

Periodic retraining

Perhaps more popular in the applied side of machine learning, periodic retraining is a common routine to combat the distribution shift over time. [HPJ$^+$14b] uses the term *data freshness* to describe the delay between the dates that the training and testing data sets are collected. They observe that the prediction accuracy of the model decreases as the delay between the training and testing increases. Thus to improve the prediction accuracy, they retrain the model on a daily basis so that the delay between the training and testing is as small as possible.

8.4.4 Spurious correlation and information leakage

In the context of predictive modeling, spurious correlation is a type of information leakage. [KRPS12] argues that there are two causes of information leakage: leakage in features and leakage in training examples. Spurious correlation is a type of leakage in training examples.

To the best of our knowledge, there is no systematic method that can universally prevent the spurious correlation between the training and testing sets. In fact, even detecting the spurious correlation is a difficult task.

[RPŚ+10] proposes three general approaches for detecting spurious correlation.

Exploratory data analysis

As a general data analysis approach, exploratory data analysis covers a wi de variety of statistical tools and techniques, and provides direct insights on the underlying structure of the data set. Many instances of information leakage, including those caused by the spurious correlation, can be identified by exploratory data analysis. We discuss a specific example of revealing the spurious correlation using a simple data visualization technique in Section 9.

Critical model evaluation

In many real-world problems, the domain experts and the modelers often have a rough expectation for the model performance. When the model performance is too good to be true, it should raise concerns. In Section 9, we discuss how we saw stellar results on the prediction accuracy, which raised a red flag, and motivated us to discover the spurious correlation between the training and testing sets.

Exploration of usage scenarios

Finally, one can break the fallacy of the overly optimistic testing performance by testing the models in the true application settings. In other words, the information leakage can be detected

by using the fresh new data gathered after the model is trained.

[KRPS12] proposes a sufficient condition for preventing information leakage in general. In short, the condition says that all training examples must be observable for the purpose of inferring all testing examples. More formally, they define a legitimate set $legit\{u\}$ of a random variable u. A second random variable v is said to be in $legit\{u\}$ if v is observable at the time of inferring u. Then, for all the feature and label pair $(\tilde{X}, \tilde{y})$ in the training set, and for all label y in the test set, the condition is defined as

$$\tilde{X} \in legit\{y\} \text{ and } \tilde{y} \in legit\{y\}.$$

However, this condition may be hard or expensive to validate on instance-by-instance basis for all training and testing examples pair. In addition, it requires a precise definition for *observable* according to the application scenario. In some cases, *observable* is an ill-defined concept. For example, the data instances may be collected over time but the precise timestamp information is not available.

Chapter 8 through Chapter 9 contain material from the ICML Workshop on Real World Experiment Design and Active Learning, 2020 ("Experimental Design for Bathymetry Editing", Alafate, Freund, Sandwell and Tozer). The dissertation author was a primary investigator and author of this paper.

Chapter 9

Experimental design for bathymetry

As a case study of how to design experiments in machine learning, we study the problem of bathymetry data editing in this chapter.

Bathymetry is a study of the depths and shapes of underwater terrain. One of the key objectives in the bathymetry study is to produce a high-definition map of the global seafloor by aggregating the available data from multiple research institutions around the world. There is some percentage of the bathymetry data that are corrupted, which should be eliminated from the final seafloor map. Therefore, it is critical to have a careful data editing step before merging the new data sources for eliminating the corrupted data. In this section, we describe how we use machine learning to assist the data editing process, and discuss the challenges posed by the data diversity problem and the spurious correlation.

9.1 Bathymetry data editing

Modern bathymetry drawn on two sources of information: Satellite altimeters that have global coverage but poor resolution, and shipboard echo sounders, which provide potentially more accurate data only along the path of the ship. The quality of the data from the echo sounders varies widely. As a consequence, extensive quality assurance is performed on the data after

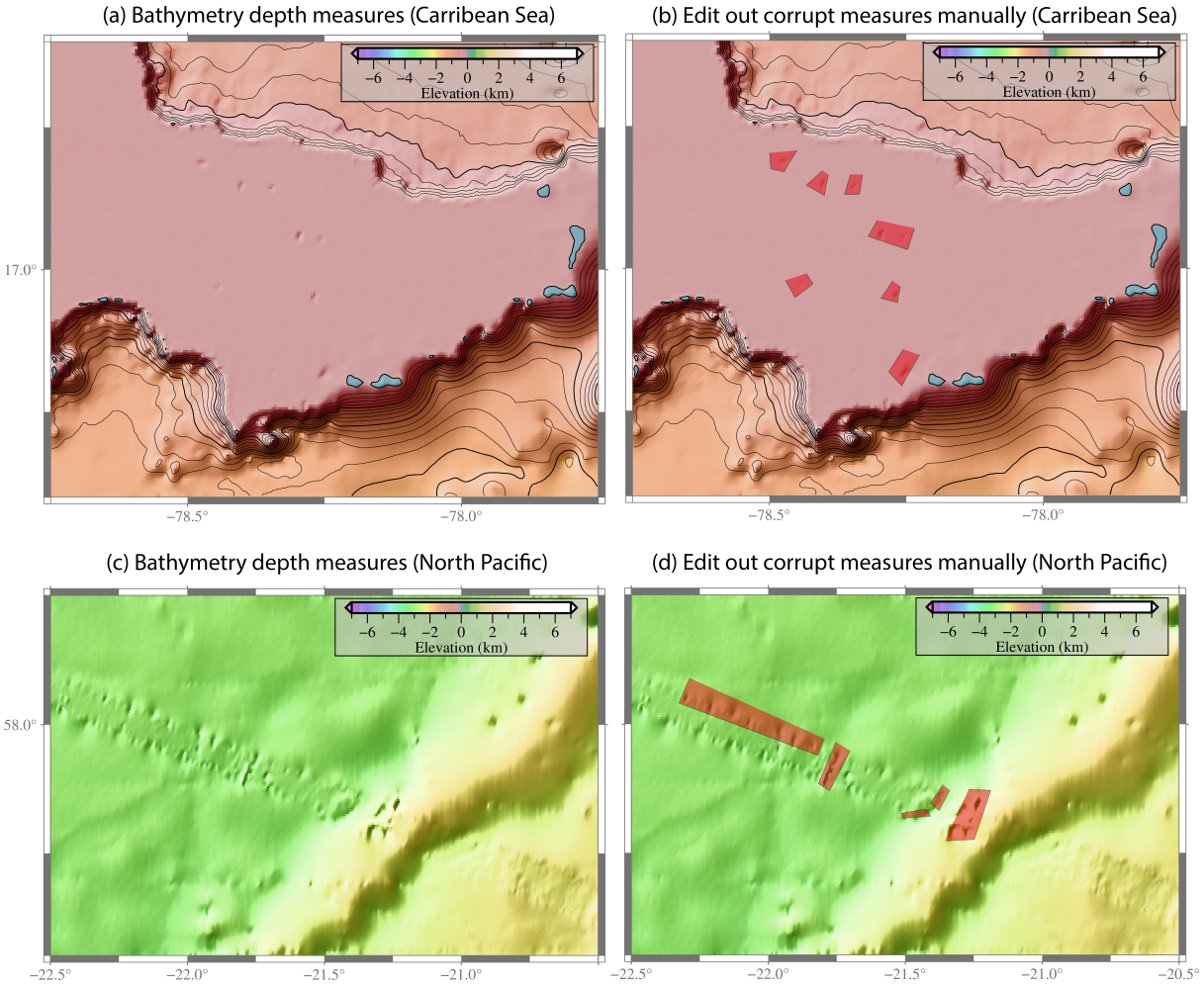

Figure 9.1: Manual bathymetry editing: **(a) (c)** A segment of ocean sea floor which includes from bad measurements. **(b) (d)** Manual editing requires drawing small polygons to identify the measurements to be removed.

it is collected from the ships and before it is integrated into the map. This process is called "editing" and corresponds to assigning a binary label to each measurement. Measurements labeled "good" are incorporated into the bathymetry map while "bad" measurements are discarded. This process has traditionally been done manually by analyzing the map after a full update (using all measurements), then identifying and removing suspicious outliers in the updated map (Figure 9.1).

Manual editing is labor intensive and the accuracy of the results vary from person to person. The goal of our project is to develop a "computer assisted editing" method in which the computer assigns a confidence-rated prediction to each measurement. The human editor can then select a large area and instruct the computer to remove all bad measurements within that area (Figure 9.2).

The original bathymetric data comes with three attributes: longitude, latitude, and the seafloor depth measurement at the specified location. In the preprocessing step, we also have access to other meta data related to the location of the measurements, which we use to enrich the feature representation.

The bathymetry dataset has a clear hierarchical structure. The dataset is aggregated from multiple data sources (17 sources at the time of the writing), each corresponding to a research institution. Within each institution, the data is gathered from multiple vessel cruises. In addition, based on how the depth was measured, we can further split the data instances into two classes: measured by a single-beam echo sounder or a multi-beam echo sounder.

9.2 Data description

We use a bathymetry datasets from 7 different research institutions around the globe (see Table 9.1). Each research institution collects their data using hundreds to thousands of cruises. The numbers of data samples collected by different cruises are not uniform. We use *imbalance ratio* to denote the percentage of samples from the largest 20% of the cruises.

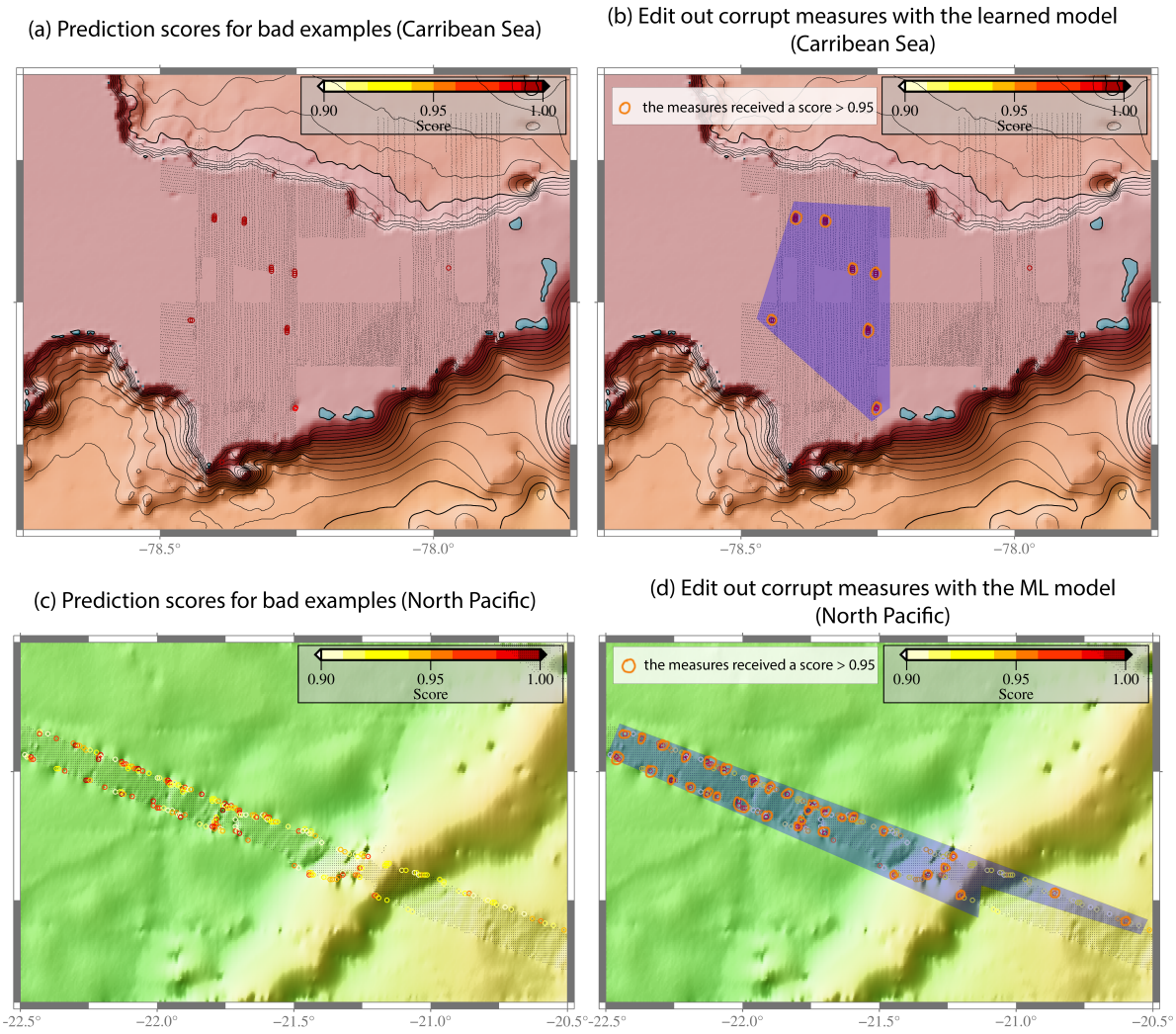

Figure 9.2: Computer assisted bathymetry editing: **(a) (c)** Scores generated by the boosted trees algorithm. **(b) (d)** computer-assisted editing, the editor draws a larger rectangle and specifies a minimal threshold for points to be removed.

Table 9.1: Overview of the bathymetry editing dataset

Institution	% edited[1]	num. of cruises	num. of measures	imbalance ratio[2]
all	5.105	7881	287 M	93
AGSO	1.961	127	15 M	48
JAMSTEC	4.306	541	81 M	81
NGA	19.340	1374	4 M	92
NGDC	4.751	1033	116 M	69
NOAA geodas	11.063	4079	20 M	72
SIO	13.137	247	21 M	76
US multi	5.301	480	30 M	73

Note 1. Percent edited refers to the percentage of data removed from each data set by human editors.

Note 2. Imbalance ratio refers to the percentage of data from the top 20% of the cruises ranked by the number of measurements collected by each cruise.

The samples from different research institutions differ statistically. The sample sizes range from 4 millions to over 100 millions. The quality of samples varies from one institution to another. For example, 1.961% of the samples from AGSO are labeled as corrupted and removed, while for NGA 19.340% of the samples are labeled as removed. There are a number of reasons that might explain these differences. Probably one of the more important ones is that the datasets are collected from different geographic areas. For instance, JAMSTEC data focuses on the area near Japan, while SIO data concentrates near US west coast [TSS$^+$19].

The original depth measurements has 3 features: measured depth, latitude, and longitude. we added 32 additional features to describe the region where the data is collected, for example, the medium seafloor depth of the region, the seafloor variance of the region, the year in which the data was collected, and more. The full list of feature descriptions could be found in Appendix B.

The data label is a binary bit, where 0 means the human labeler marked the measurement as corrupted or incorrect, and 1 means that the labeler marked the measurement as valid. The learned classifier should ensure a low false negative rate (so that as few valid measurements are removed as possible), while having close to zero false positive rate (so that the corrupt data would not be integrated into the final map).

9.3 Learning from multiple data sources

Under the standard framework, the data from different research institutions would be merged into one and create a large data sample S. Then the sample S would be split into a training set and a test set using the RTTS method. The common consensus in machine learning says that, if the sample S is sufficiently large, the trained model would *generalize well* to the unseen samples. This claim is disputable for this dataset, because there are wide varieties among the samples from different research institutions as we show in Section 9.2. Consider an extreme case as follows. If the samples from one institution amount to 99% of the samples in S, the sample S would not be representative of the *global bathymetry* because of its heavy bias towards the data from one specific institution.

Instead, we treat the data from different research institutions as they are from different data sources, and train one model for each one of them. For each research institution i, we merge all of its measurements and create a sample S_i. Then then shuffle the examples in S_i and split it to a training set $\mathcal{T}_i$ and a test set $\mathcal{U}_i$. Lastly, we train a model $H^{(i)}$ with each $\mathcal{T}_i$, and then test it with $\mathcal{U}_i$.

The experiment results are exceptional. For all research institution, the trained model achieves almost perfect separation between the examples that are labeled good and that are labeled bad. In Figure 9.3, we select three research institutions, and present their ROC (receiver operating characteristic) curves and the scores distribution for the good examples and the bad examples. The three institutions are JAMSTEC, NGA, and SIO. For all three institutions, the areas under ROC curve of the trained models are above 0.99. The plots of the score distributions show that the models are able to separate the good examples from the bad ones almost perfectly.

The evaluation results are too good to be true. Upon further examination, we realize that the models would not have utility in practice since their exceptional performance is achieved by exploiting the spurious correlation between the training and testing examples.

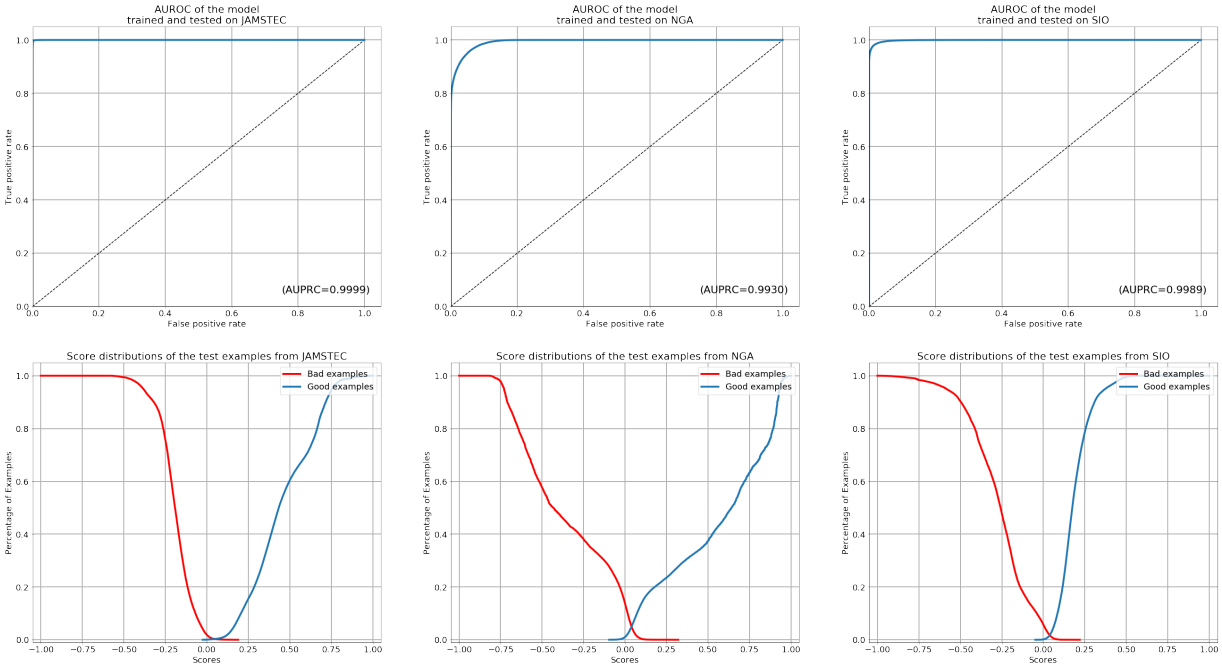

Figure 9.3: The ROC curves and the scores distributions of three models that exploit the spurious correlation between the training set and the test set. The scores distribution plot show the percentage of examples that exceed certain threshold. Specifically, the blue curve shows the percentage of good examples (y-axis) that receive a score below the corresponding threshold (x-axis); the red curve shows the percentage of bad examples (y-axis) that receive a score above the corresponding threshold (x-axis). The overlapping area in the middle denote the fraction of examples that the model is unable to separate. For all three models, the values of AUROC are all above 0.99. The score distributions show that the model is able to separate the good examples from the bad examples almost perfectly.

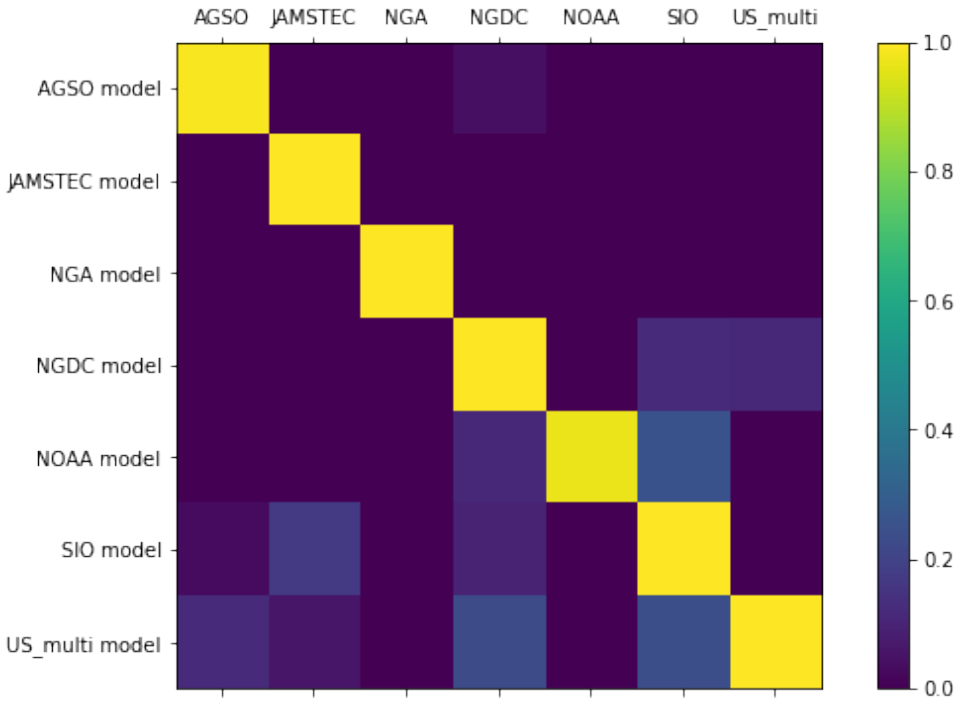

Figure 9.4: Evaluating the Area under the ROC curve (AUROC) of a model with the data from different research institutions. The results show that the model $H^{(i)}$ achieves close to 1.0 AUROC on the test data $\mathcal{U}_i$ (i.e. the samples from the same research institution where the training data was drawn), but cannot generalize to the test data from other regions in general.

To see why the model is impractical, we evaluate the performance of $H^{(i)}$ over the test sets $\mathcal{U}_1, \mathcal{U}_2, \ldots, \mathcal{U}_k$ individually. The performance is again evaluated by the area under the ROC curve for each combination of the model $H^{(i)}$ and the test set $\mathcal{U}_j$, and visualize the evaluations in Figure 9.4. The result is surprising: the values of AUROC are close to 1.0 when $i = j$, but almost 0.0 when $i \neq j$. In other words, the model H_i predicts almost perfectly if the test sample is drawn from the same research institution where its training data was drawn, but cannot generalize to the test data from any other research institutions in general.

9.4 The danger of overly fine partitioning

In the common practice, data sets are split randomly, which might not be adequate. To see that, we describe the problem it causes if we apply this type of data splitting in the bathymetry data editing task.

Individual-based splitting

As in the common practice, we merge all measures from all the cruises, without noting to which research institution they belong. Then we decide the proportion of the measurements to be put into the training set, ρ. Finally, we iterate through all measurements, and assign them to the training set with the probability of ρ, and to the testing set with the probability of $1-\rho$. An additional validation set can be generated similarly. We call this way of data splitting "individual-based splitting".

Individual-based splitting results in a classifier that delivers an exceptional accuracy rate. However, the classifier cannot generalize beyond the test set. Upon further examination, we discover that the classifier is effectively memorizing the location of the measurements in the training sets and their corresponding labels.

As it shows in Figure 9.5, the label of one measurement is highly correlated with the

label of the previous measure. In the data region marked out by the green box, we can see a clear split between the corrupted measurements (in red) and the valid measurements (in blue). The clear separation may have multiple explanation, for example, the echo sounder might have malfunctioned while traveling the ill-measured regions, but got fixed and worked properly after leaving the regions. If a machine learning model with the strong "memorizing" capability is applied in this case, it might exploit this phenomena and overfit in these regions. When it happens, the testing examples sampled from the same region will perform exceptionally, but we cannot trust this performance to be generalized to the regions not presented in our training data.

As a general principal, a reliable testing set should be acquired *independently* from the acquisition of the training set while preserving certain workflow involved in the data collection process. In the bathymetry data editing task, the measurements in the testing set should be collected by different cruises that are not observed in the training set. In certain task involving the video or other forms of sequential images, the testing data and the training data should not be sampled from the same clips. We argue that we cannot obtain a reliable generalization performance by simply guaranteeing that the training and testing examples do not overlap.

In Figure 9.5, we visualize the geolocation of the measurements, and use two different colors to denote their labels: the blue dots for the valid ones, and the red dots for the corrupted ones. We can see that there is a high correlation in the correctness of the measurements over the consecutive measurements. The reason is as follows. The cruises took the depth measurements sequentially over time. The measurements were predominantly accurate, until some abnormal activity, such as device malfunction, happened at some point t_1. Following the time t_1, all the measurements taken were all corrupted. Lastly, the cruises started to take correct measurements again at some point t_2 once the malfunction was fixed.

In the first iteration of our work, we created the training and testing sets using RTTM over the data instance. Specifically, we aggregate all measurements from all cruises, shuffle them, and randomly partition them into the training set and the testing set. We discovered that the accuracy

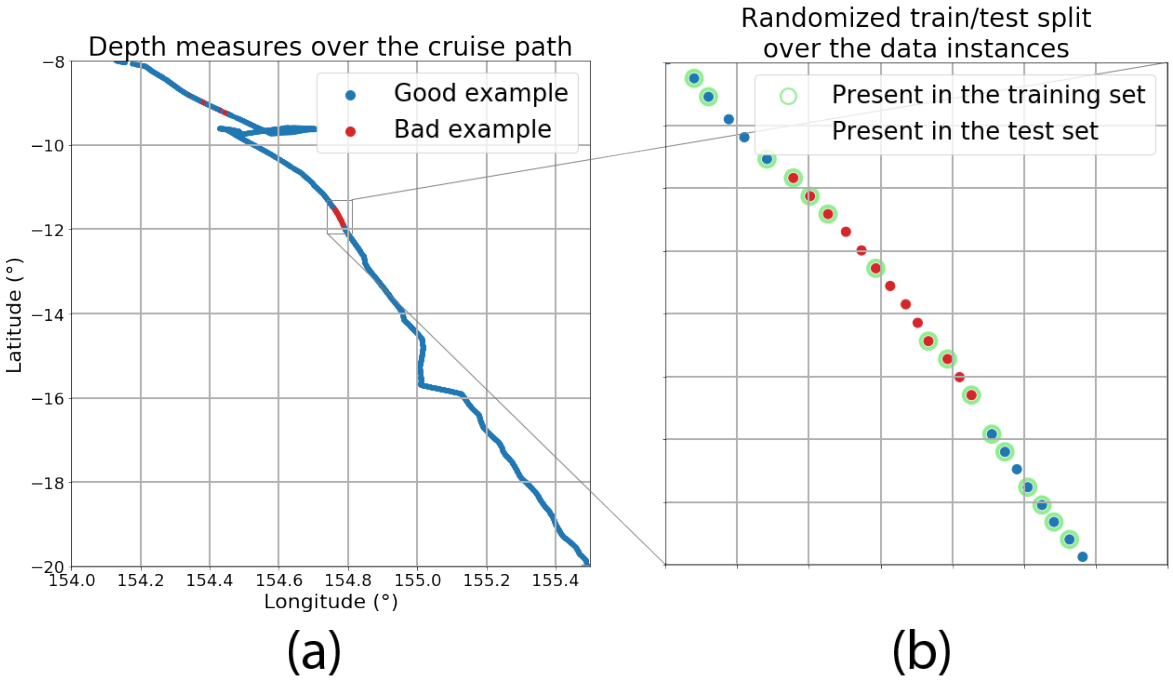

Figure 9.5: An example of the sequentiality problem in bathymetry. **(a)** is the route of a typical ship cruise. **(b)** is a zoomed-in part of the route, depicting the partition of examples into training and testing.

of the trained model was almost perfect on the testing set (see Figure 9.3). We then suspected that the model was able to exploit the sequentiality of the measurement correctness, and inferring the labels of the testing examples by comparing to the neighboring examples in the training set.

To test our hypothesis, we conducted the second iteration of the modeling which uses cruise-based splitting.

Cruise-based splitting

We use cruise-based splitting method to address the overly-fined partitioning problem that we had in the individual-based splitting. To avoid the model to memorize and exploit the sequentiality in the measurements collected by the same cruise, we make the training/testing data split on the cruise level: all measurements from the same cruise will either be in the training set or in the testing set but not both.

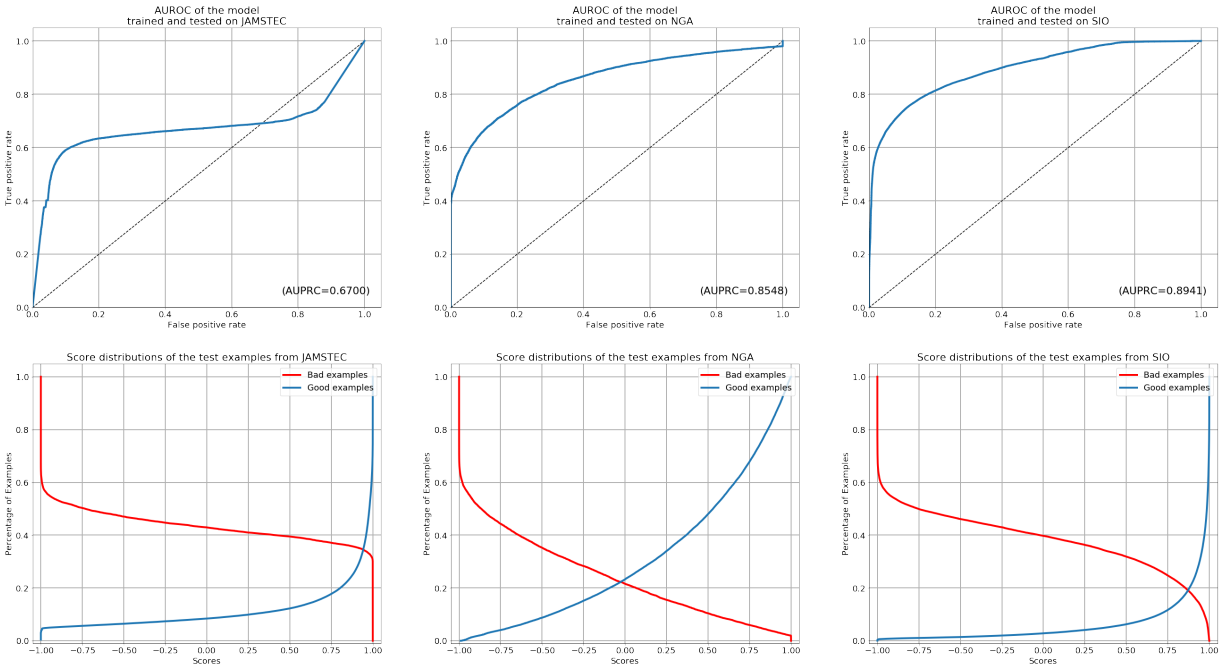

Figure 9.6: The ROC curves and the scores distributions of three models after we split the data at the cruise level. The scores distribution plot show the percentage of examples that exceed certain threshold. The overlapping area in the middle denote the fraction of examples that the model is unable to separate. The AUROC scores are more realistic. However, for some fraction of the test samples, the model seems to be predicting worse than random guessing (e.g. in JAMSTEC).

Similar to the instance-based splitting, we first decide the proportion of *the number of cruises* to be put into the training set, ρ. Then we iterate through all cruise in random order, and assign them to the training set with the probability of ρ, and to the testing set with the probability of $1 - \rho$.

We repeated the same experiments, and obtained a more realistic results. In Figure 9.6, we see that the models trained on the NGA and SIO datasets achieve reasonably good AUROC scores, while the AUROC of the model trained on the JAMSTEC dataset decreases significantly.

9.5 Imbalance within the coherent subsets

Within one "coherent" subset, there can be an imbalance among the samples. For example, in the bathymetry data provided by one of the agencies (NGA), the top 5 cruises sorted by the number of measures (out of 1376 cruises) contain about the same number of measures that the rest of the cruises combined.

To mitigate this problem we changed the way we partition the data into training and testing. Namely, we divided the data sequences into long sub-sequences and then place each sub-sequence at either the training set or the test set at random. We compared the two ways of forming sub-sequences. The first way is to take whole cruises as sub-sequences, the second is to take non-overlapping sub-sequences of 100,000 measurements. These correspond to about 2500KM for the multi-beam cruises. Most cruises are shorter than 100,000 and are included whole, but some are much longer. Partitioning the sequences in these ways reduces the AUROC significantly, as is shown in Figure 9.7. However, this performance is a better predictor of future performance.

Identifying the coherent subsets of the training datasets can potentially improve the prediction accuracy in two ways. First, if we individually train a model for every data subset, we expect the model accuracy to improve since the data distribution is "easier" to learn. Second, we

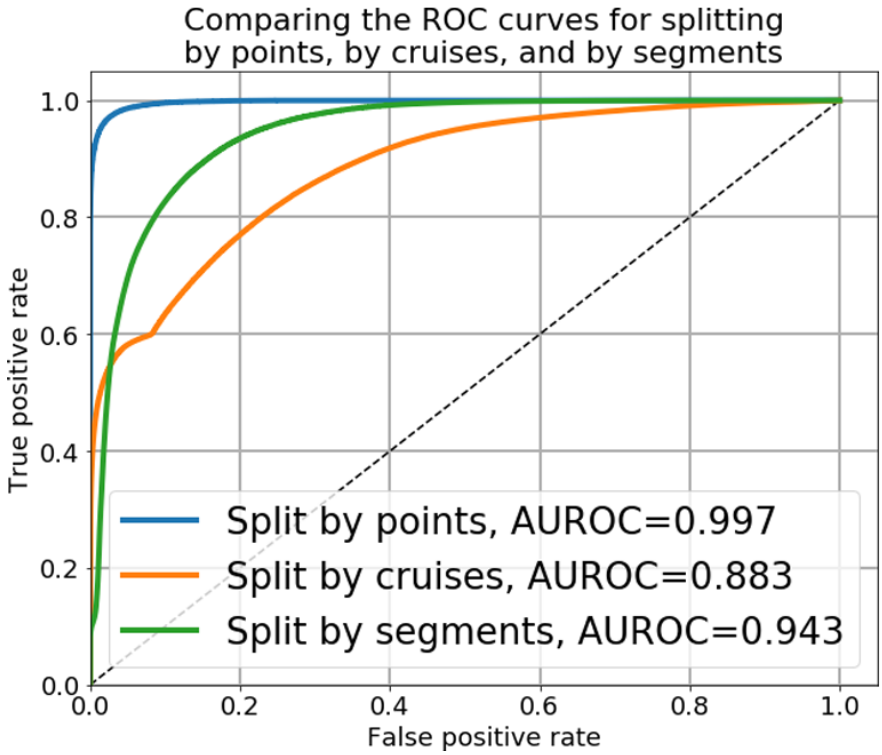

Figure 9.7: The ROC curves for three different ways of partitioning the data into train and test: individual examples, whole cruises, and segments of 100,000 measurements.

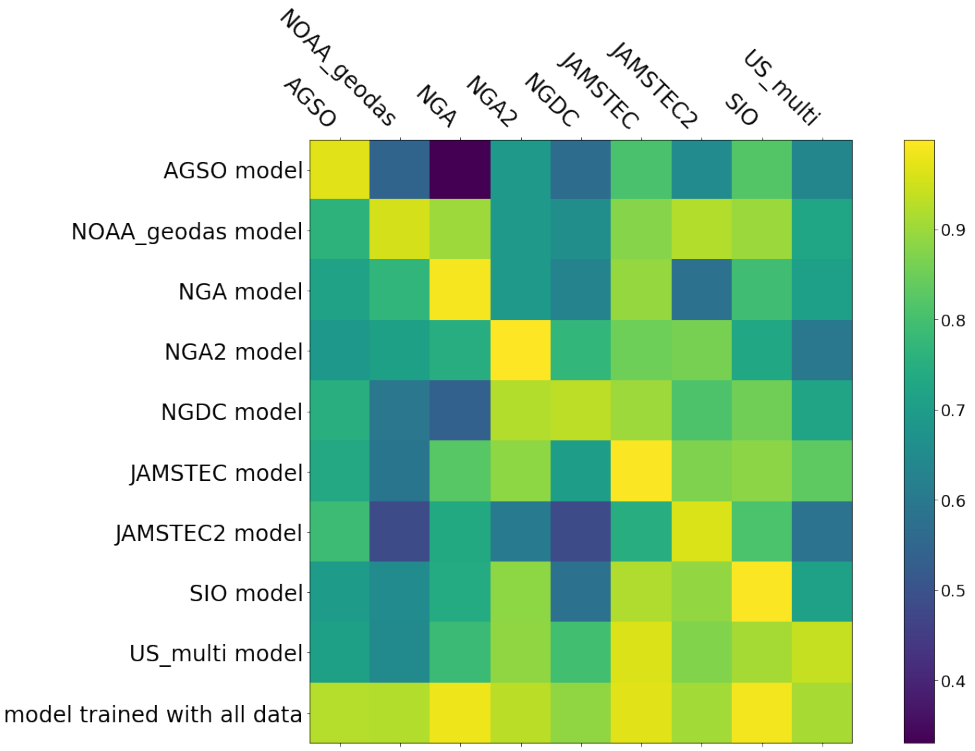

Figure 9.8: Area under ROC (AUROC) for the models trained using data from one region and tested on the data from another region. Higher values are better.

can create an ensemble of the models trained on each coherent subsets, and get a better prediction accuracy than any single classifier.

The general principal of identifying subsets within a dataset is to split data in the way that improves the model performance, and to stop further splitting when it causes overfitting.

The bathymetry datasets is aggregated from various research institutes around the globe. A natural assumption is that the examples collected from the same agencies are more likely to be coherent.

The data splitting is similar to the general cruise-based splitting, with the addition that we now create separate training and testing data sets for the data from each research institutes.

With every training set that contains examples from only one institution, we can train a model that is supposed to be performed best on the examples from the same coherent set. To further validate our assumption, we *cross test* the models using the examples collect by the other

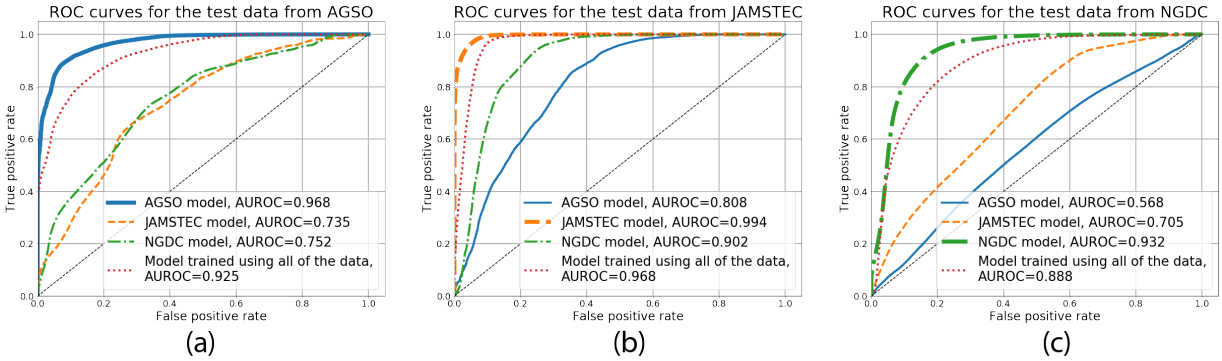

Figure 9.9: Comparison of the test performance using the models trained using the data from different sources. The plots show the test performance on the test data sampled from the datasets provided by *AGSO* (Australia), *JAMSTEC* (Japan), and *NGDC* (US-based data center with global data coverage). In all three cases, the optimal models are the one trained on the data sampled from the same origin as the test data. The model trained using all of the data is also included for comparison.

Table 9.2: Improvement in terms of AUROC of a model trained on the data sampled from the same region as the test data over the model trained on all of the data.

AGSO	geodas[1]	NGA	NGA2	NGDC	JAM[2]	JAM2[2]	SIO	US_multi
4.29%	3.38%	0.78%	6.94%	4.38%	2.59%	5.57%	0.73%	2.95%

Note 1. NOAA_geodas, abbreviated to fit the page.
Note 2. JAMSTEC and JAMSTEC2, abbreviated to fit the page.

agencies, which under our assumption are sampled from a similar but different distribution.

We also reported the model performance on each of the data sources. Each row of the matrix in Figure 9.8 corresponds to a model (classifier) trained using data from one of the 17 regions, the bottom row correspond to the model trained using all of the data. The columns correspond to the region from which the test set is taken (train and test sets from each region are randomly partitioned). The colors correspond to correspond to the AUROC when the row model is tested on the column test data. First, observe that some pairs, such as model *AGSO* and test *NGA*, or *JAMSTEC* model and test *NGDC* have a very poor AUROC of 0.5 or smaller. In other words, the distributions corresponding to different regions are very different, which is why we call this data source diverse. This contradicts the identically distributed part of the IID assumption.

Further, the highest AUROC in each column corresponds to the the model trained on

data from the same region. The second best is usually the classifier trained using all of the data. However, as shown in table 9.2, the classifier using just the data corresponding to the the same region is, in all cases, better than the classifier using all of the data.

We show the full ROC curves of three specific regions in Figure 9.9. We sampled the test sets from *AGSO* (Australia-based organization), *JAMSTEC* (Japan-based research institution), and *NGDC* (US-based data center). We then used these test sets to evaluate four models: first three are trained using the data from each one of the three regions, and the fourth one trained using all of the data. In all three cases, the best performance (in terms of AUROC) is achieved by the model trained using the data from the same source as the test data, while the performances of the models trained using the data from difference sources are significantly worse. Finally, the performance of the model trained using all of the data falls in between.

We conclude that for the bathymetry editing task, combining all data into one large training set is inferior to using only data that is similar to the data in the test set. In other words, maximizing the size of the training set should be balanced against the similarity of the training set to that of the expected test set.

Chapter 8 through Chapter 9 contain material from the ICML Workshop on Real World Experiment Design and Active Learning, 2020 (“Experimental Design for Bathymetry Editing”, Alafate, Freund, Sandwell and Tozer). The dissertation author was a primary investigator and author of this paper.

Appendix A

Pseudocode for Sparrow

Algorithm 3 Main Procedure of **Sparrow**

Input Sample size n

A threshold θ for the minimum n_{eff}/n ratio for training weak learner

Initialize $H_0 = 0$

Create initial sample S by calling SAMPLE

for $k := 1 \ldots K$ **do**

Call **Scanner** on sample S generate weak rule h_k, γ_k

$H_k \leftarrow H_{k-1} + \frac{1}{2} \log \frac{1/2+\gamma}{1/2-\gamma} h_k$

if $n_{\text{eff}}/n < \theta$ **then**

Receive a new sample $S \leftarrow$ from **Sampler**

Set $S \leftarrow S'$

end if

end for

Algorithm 4 Scanner

Input An iterator over in-memory sampled set S

Initial advantage target $\gamma_0 \in (0.0, 0.5)$

static variable $\gamma = \gamma_0$

loop

if sample S is scanned without firing stopping rule **then**

Shrink γ by $\gamma \leftarrow 0.9\widehat{\gamma}$

Reset S to scan from the beginning

end if

$(x, y, w_l) \leftarrow S.next()$

$w \leftarrow \text{UPDATEWEIGHT}(x, y, w_l, H)$

for $h \in \mathcal{W}$ **do**

Compute $h(\vec{x})y$

Update M_t, V_t (Eqn 4.3)

if Stopping Rule (Eqn 4.4) fires **then**

return h, γ

end if

end for

end loop

Algorithm 5 Sampler

Input Randomly permuted, disk-resident training-set

Disk-resident stratified structure $D \leftarrow \{\}$

Weights of the strata $W \leftarrow \{\}$

Construct new sample $S \leftarrow \{\}$

loop

With the probability proportional to W,

sample a strata R

$(x, y, w_l) \leftarrow R.next()$

Delete (x, y, w_l) from R, **update** W

Receive new model H from MAINPROCEDURE

$w \leftarrow$ UPDATEWEIGHT(x, y, w_l, H)

With the probability proportional to w,

$S \leftarrow S + \{(x, y, w)\}$.

Append (x, y) to the right stratum with regard to w,

$D \leftarrow D + \{(x, y, w)\}$

Update W

if S is full **then**

Send S to MAINPROCEDURE

$S \leftarrow \{\}$

end if

end loop

Appendix B

Bathymetry editting dataset

The bathymetry editing dataset has 32 features and a binary label. The learning task is to predict whether a depth measurement is valid or corrupted. The original bathymetry data comes with only three features, the measured depth in addition to the latitude and the longitude of the measured location. We enrich the feature representation by adding relevant features derived from the existing bathymetry map. The list of features and their descriptions are described in Table B.1.

Table B.1: Data features of the bathymetry editing dataset

Name	**Example**	**Description**
lon	143.92	longitude
lat	-43.99	latitude
depth	-4637	the depth measure
pred	-4633	the predicted depth from the gravity model
(pred-depth)/depth	0.00	the relative difference between the depth measure and the prediction from the gravity model
d10	0.98	average depth of the sea floor in the 10km grid
d20	0.97	average depth of the sea floor in the 20km grid

Continued on next page

Table B.1: Data features of the bathymetry editing dataset *(Continued)*

Name	**Example**	**Description**
d60	0.95	average depth of the sea floor in the 60km grid
age	39.35	age of the oceanic plate
VGG	20.92	vertical gravity gradient
rate	1773.15	plate half-spreading rate
sed	1002.69	sediment thickness
roughness	23.16	seafloor roughness
G:T	0.58	free air gravity anomaly to topography ratio
NDP2.5m	352.59	the depth measure minus the median depth in 2.5km blocks
NDP5m	1227.86	the depth measure minus the median depth in 5km blocks
NDP10m	4867.36	the depth measure minus the median depth in 10km blocks
NDP30m	28191.80	the depth measure minus the median depth in 30km blocks
STD2.5m	22.27	the standard deviation of the depth in 2.5km blocks
STD5m	42.23	the standard deviation of the depth in 5km blocks
STD10m	86.56	the standard deviation of the depth in 10km blocks
STD30m	188.40	the standard deviation of the depth in 30km blocks
MED2.5m	-17.31	the median depth in 2.5km blocks
MED5m	-30.94	the median depth in 5km blocks
MED10m	-0.91	the median depth in 10km blocks
MED30m	1.92	the median depth in 30km blocks

Continued on next page

Table B.1: Data features of the bathymetry editing dataset *(Continued)*

Name	**Example**	**Description**
D-MED2.5m/STD2.5m	-0.77	the difference between the depth measure and the median depth divided by the standard deviation of the depth, in 2.5km blocks
D-MED5m/STD5m	-0.73	the difference between the depth measure and the median depth divided by the standard deviation of the depth, in 5km blocks
D-MED10m/STD10m	-0.01	the difference between the depth measure and the median depth divided by the standard deviation of the depth, in 10km blocks
D-MED30m/STD30m	0.01	the difference between the depth measure and the median depth divided by the standard deviation of the depth, in 30km blocks
year	2000	the year in which the depth was measured
kind	G	the type of the echo sounder used for the measurement

Bibliography

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

[ABGLP19] Martin Arjovsky, Léon Bottou, Ishaan Gulrajani, and David Lopez-Paz. Invariant risk minimization. *arXiv preprint arXiv:1907.02893*, 2019.

[ACDL14] Alekh Agarwal, Oliveier Chapelle, Miroslav Dudík, and John Langford. A Reliable Effective Terascale Linear Learning System. *Journal of Machine Learning Research*, 15:1111–1133, 2014.

[ADF+10] Dragomir Anguelov, Carole Dulong, Daniel Filip, Christian Frueh, Stéphane Lafon, Richard Lyon, Abhijit Ogale, Luc Vincent, and Josh Weaver. Google street view: Capturing the world at street level. *Computer*, 43(6):32–38, 2010.

[AF18] Julaiti Alafate and Yoav Freund. Tell me something new: A new framework for asynchronous parallel learning. *arXiv preprint arXiv:1805.07483*, 2018.

[Bal14] Akshay Balsubramani. Sharp Finite-Time Iterated-Logarithm Martingale Concentration. *arXiv:1405.2639 [cs, math, stat]*, May 2014.

[BBL12] Ron Bekkerman, Mikhail Bilenko, and John Langford. *Scaling Up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press, 2012.

[BD19] Björn Barz and Joachim Denzler. Do we train on test data? purging cifar of near-duplicates. *arXiv preprint arXiv:1902.00423*, 2019.

[BMR+20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin

Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[Bre96] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[BS07] Joseph K. Bradley and Robert E. Schapire. FilterBoost: Regression and Classification on Large Datasets. In *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, pages 185–192, USA, 2007. Curran Associates Inc.

[CG16] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[dB16] Marleen de Bruijne. Machine learning approaches in medical image analysis: From detection to diagnosis. *Medical Image Analysis*, 33:94 – 97, 2016.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[DKP+15] Trevor Darrell, Marius Kloft, Massimiliano Pontil, Gunnar Rätsch, and Erik Rodner. Machine learning with interdependent and non-identically distributed data (dagstuhl seminar 15152). In *Dagstuhl Reports*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[FHH+07] Jerome Friedman, Trevor Hastie, Holger Höfling, Robert Tibshirani, et al. Pathwise coordinate optimization. *The annals of applied statistics*, 1(2):302–332, 2007.

[FHT00] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting (With discussion and a rejoinder by the authors). *The Annals of Statistics*, 28(2):337–407, April 2000.

[Fri01] Jerome H. Friedman. Greedy Function Approximation: A Gradient Boosting Machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.

[FS97] Yoav Freund and Robert E Schapire. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences*, 55(1):119–139, August 1997.

[GRM03] João Gama, Ricardo Rocha, and Pedro Medas. Accurate decision trees for mining high-speed data streams. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 523–528. ACM, 2003.

[HCC+13] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 1223–1231. Curran Associates, Inc., 2013.

[HDM17] Christina Heinze-Deml and Nicolai Meinshausen. Conditional variance penalties and domain shift robustness. *arXiv preprint arXiv:1710.11469*, 2017.

[HNP09] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[HPJ+14a] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, and et al. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, page 1–9, New York, NY, USA, 2014. Association for Computing Machinery.

[HPJ+14b] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, pages 1–9, 2014.

[HS11] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2009.

[JAM16] Japan Agency for Marine-Earth Science and Technology JAMSTEC. Data and sample research system for whole cruise information in jamstec (darwin). `http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm`, 2016.

[Kit96] Genshiro Kitagawa. Monte Carlo Filter and Smoother for Non-Gaussian Nonlinear State Space Models. *Journal of Computational and Graphical Statistics*, 5(1):1–25, 1996.

[KL18] Wouter M Kouw and Marco Loog. An introduction to domain adaptation and transfer learning. *arXiv preprint arXiv:1812.11806*, 2018.

[KMF+17] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3146–3154. Curran Associates, Inc., 2017.

[KN18] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2018.

[KRPS12] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(4):1–21, 2012.

[Lan88] Pat Langley. Machine learning as an experimental science. *Machine Learning*, 3(1):5–8, 1988.

[LAP+14] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014.

[LHLL15] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2737–2745. Curran Associates, Inc., 2015.

[LSZ09] John Langford, Alexander Smola, and Martin Zinkevich. Slow learners are fast. *arXiv preprint arXiv:0911.0491*, 2009.

[MBBF99] Llew Mason, Jonathan Baxter, Peter Bartlett, and Marcus Frean. Boosting Algorithms As Gradient Descent. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 512–518, Cambridge, MA, USA, 1999. MIT Press.

[Mil98] George A Miller. *WordNet: An electronic lexical database*. MIT press, 1998.

[MIM15] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! But at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[Ng16] Andrew Ng. Nuts and bolts of building ai applications using deep learning. *NIPS Keynote Talk*, 2016.

[NMK19] Jakub Nalepa, Michal Myller, and Michal Kawulok. Validating hyperspectral image segmentation. *IEEE Geoscience and Remote Sensing Letters*, 16(8):1264–1268, 2019.

[PBM16] Jonas Peters, Peter Bühlmann, and Nicolai Meinshausen. Causal inference by using invariant prediction: identification and confidence intervals. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 78(5):947–1012, 2016.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[Ros20] Corby Rosset. *Turing-NLG: A 17-billion-parameter language model by Microsoft*, 2020.

[RPŚ+10] Saharon Rosset, Claudia Perlich, Grzergorz Świrszcz, Prem Melville, and Yan Liu. Medical data mining: insights from winning two competitions. *Data Mining and Knowledge Discovery*, 20(3):439–468, 2010.

[RRWN11] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011.

[Sch90] Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, June 1990.

[SF10] Soeren Sonnenburg and Vojtěch Franc. COFFIN: A Computational Framework for Linear SVMs. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 999–1006, USA, 2010. Omnipress.

[SF12] Robert E. Schapire and Yoav Freund. *Boosting: Foundations and Algorithms*. MIT Press, 2012.

[Shi07] Haijian Shi. *Best-first decision tree learning*. PhD thesis, The University of Waikato, 2007.

[Sim77] Herbert A Simon. Spurious correlation: A causal interpretation. In *Models of Discovery*, pages 93–106. Springer, 1977.

[SN10] Alexander Smola and Shravan Narayanamurthy. An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, 3(1-2):703–710, 2010.

[SSSG17] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. Revisiting unreasonable effectiveness of data in deep learning era. In *Proceedings of the IEEE international conference on computer vision*, pages 843–852, 2017.

[TSS+19] B Tozer, DT Sandwell, WHF Smith, C Olson, JR Beale, and P Wessel. Global bathymetry and topography at 15 arc sec: Srtm15+. *Earth and Space Science*, 6(10):1847–1864, 2019.

[Val90] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, August 1990.

[VOIVDB14] Annegreet Van Opbroek, M Arfan Ikram, Meike W Vernooij, and Marleen De Bruijne. Transfer learning improves supervised image segmentation across imaging protocols. *IEEE transactions on medical imaging*, 34(5):1018–1030, 2014.

[Wal73] Abraham Wald. *Sequential Analysis*. Courier Corporation, 1973.

[Wal80] W Allen Wallis. The statistical research group, 1942–1945. *Journal of the American Statistical Association*, 75(370):320–330, 1980.

[XGB20] XGBoost. *Using XGBoost External Memory Version*, 2020.

[YXC+18] Fisher Yu, Wenqi Xian, Yingying Chen, Fangchen Liu, Mike Liao, Vashisht Madhavan, and Trevor Darrell. Bdd100k: A diverse driving video database with scalable annotation tooling. *arXiv preprint arXiv:1805.04687*, 2018.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.