Scalable Ensemble Learning and Computationally Efficient Variance Estimation

by

Erin E. LeDell

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Biostatistics

and the Designated Emphasis

in

Computational Science and Engineering

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Mark van der Laan, Co-chair
Assistant Professor Maya Petersen, Co-chair
Professor Alan E. Hubbard
Professor in Residence Phillip Colella

Spring 2015

Scalable Ensemble Learning and Computationally Efficient Variance Estimation

Copyright 2015
by
Erin E. LeDell

Abstract

Scalable Ensemble Learning and Computationally Efficient Variance Estimation

by

Erin E. LeDell

Doctor of Philosophy in Biostatistics
and the Designated Emphasis in
Computational Science and Engineering

University of California, Berkeley

Professor Mark van der Laan, Co-chair

Assistant Professor Maya Petersen, Co-chair

Ensemble machine learning methods are often used when the true prediction function is not easily approximated by a single algorithm. The Super Learner algorithm is an ensemble method that has been theoretically proven to represent an asymptotically optimal system for learning. The Super Learner, also known as stacking, combines multiple, typically diverse, base learning algorithms into a single, powerful prediction function through a secondary learning process called metalearning. Although ensemble methods offer superior performance over their singleton counterparts, there is an implicit computational cost to ensembles, as it requires training multiple base learning algorithms. We present several practical solutions to reducing the computational burden of ensemble learning while retaining superior model performance, along with software, code examples and benchmarks.

Further, we present a generalized metalearning method for approximating the combination of the base learners which maximizes a model performance metric of interest. As an example, we create an AUC-maximizing Super Learner and show that this technique works especially well in the case of imbalanced binary outcomes. We conclude by presenting a computationally efficient approach to approximating variance for cross-validated AUC estimates using influence functions. This technique can be used generally to obtain confidence intervals for any estimator, however, due to the extensive use of AUC in the field of biostatistics, cross-validated AUC is used as a practical, motivating example.

The goal of this body of work is to provide new scalable approaches to obtaining the highest performing predictive models while optimizing any model performance metric of interest, and further, to provide computationally efficient inference for that estimate.

To my loved ones.

Contents

Contents ii

List of Figures iv

List of Tables v

1 Introduction 1

2 Scalable Super Learning 3
2.1 Introduction 3
2.2 The Super Learner algorithm 3
2.3 Super Learner software 8
2.4 Online Super Learning 16
2.5 Super Learner in practice 19
2.6 Conclusion 20

3 Subsemble: Divide and Recombine 21
3.1 Introduction 21
3.2 Subsemble ensemble learning 23
3.3 Subsembles with Subset Supervision 33
3.4 Conclusion 36

4 AUC-Maximizing Ensembles through Metalearning 37
4.1 Introduction 37
4.2 Ensemble metalearning 38
4.3 AUC maximization 41
4.4 Benchmark results 44
4.5 Conclusion 52

5 Computationally Efficient Variance Estimation for Cross-validated AUC 53
5.1 Introduction 53
5.2 Cross-validated AUC as a target parameter 54
5.3 Influence curves for variance estimation 56

5.4 Confidence intervals for cross-validated AUC 58
5.5 Generalization to pooled repeated measures data 63
5.6 Software 67
5.7 Coverage probability of the confidence intervals 70
5.8 Conclusion 74

6 Glossary **75**

Bibliography **79**

List of Figures

2.1 H2O Ensemble Runtime Performance Benchmarks 16

3.1 Diagram of the Subsemble algorithm with one learner. 25
3.2 Pseudocode for the Subsemble algorithm. 26
3.3 Model Performance Benchmarks: Subsemble vs Super Learner. 31
3.4 Runtime Performance Benchmarks: Subsemble vs Super Learner. 32

4.1 Example of how to use the `method.AUC` function to create customized AUC-maximizing metalearning functions. 43
4.2 Example of how to use the custom `method.AUC_optim.4` metalearning function with the `SuperLearner` function. 44
4.3 Example of how to update an existing `"SuperLearner"` fit by re-training the metalearner with a new method. 44
4.4 Model CV AUC offset from the best Super Learner model for different metalearning methods. (No color == top metalearner) 48
4.5 CV AUC gain by Super Learner over Grid Search winning model. (Loess fit overlays actual points.) 49
4.6 Model CV AUC offset from the best Super Learner model for the subset of the metalearning methods that enforce box-constraints on the weights. (No color == top metalearner) 51

5.1 Plots of the coverage probabilities for 95% confidence intervals generated by our simulation for training sets of 1,000 (left) and 5,000 (right) observations. In the case of a 95% confidence interval, the coverage probability should be close to 0.95. For the smaller dataset of $n = 1,000$ observations, we see that the coverage is slightly lower (92-93%) than specified, whereas for $n = 5,000$, the coverage is closer to 95%. 72

List of Tables

2.1 Base learner model performance (test set AUC) compared to **h2oEnsemble** model performance performance using 2-fold CV (ensemble results for both GLM and NNLS metalearners). 15
2.2 Training times (minutes) for **h2oEnsemble** with a 3-learner library using various cluster configurations, including a single workstation with 32 vCPUs. The number of vCPUs for each cluster is noted in parenthesis. Results for $n = 5$ million are not available for the single workstation setting. 15

4.1 Default metalearning methods in **SuperLearner** R package version 2.0-17. 40
4.2 Metalearning methods evaluated. 42
4.3 Example base learner library representing a small, yet diverse, collection of algorithm classes. Default model parameters were used. 45
4.4 Top metalearner performance for HIGGS datasets, as measured by cross-validated AUC ($n = 10,000$; CV $= 10 \times 10$). 47
4.5 Top metalearner performance for HIGGS datasets, as measured by cross-validated AUC ($n = 100,000$; CV $= 2 \times 2$). 47
4.6 CV AUC for HIGGS datasets ($n = 10,000$; CV $= 10 \times 10$) 50
4.7 CV AUC for HIGGS datasets ($n = 100,000$; CV $= 2 \times 2$) 50

5.1 Coverage probability for influence curve based confidence intervals for CV AUC using training sets of various dimension. 72
5.2 Influence curve based standard errors for CV AUC for training sets of various dimensions. 73
5.3 Standard deviation of 5,000 CV AUC estimates for training sets of various dimensions. 73
5.4 Average CV AUC across 5,000 iterations for training sets of various dimensions. 73
5.5 Bootstrap confidence interval coverage probability using B bootstrapped replicates of a training set of $n = 1,000$ observations. 74

Acknowledgments

I want to thank my advisors, family, friends and the open source software community.

Chapter 1

Introduction

The machine learning technique of combining multiple algorithms into an *ensemble* has long been used to provide superior predictive performance over single algorithms. While ensembles achieve greater performance than a single algorithm, the computational demands of ensemble learning algorithms can be prohibitive in the era of "big data." The computational burden of an ensemble algorithm is the aggregate of the computation required to train all the constituent models. While some ensemble learners, e.g. Random Forest [16], can be described as "embarrassingly parallel" in nature, there are others, such as *boosting* [29] methods, that require much more sophistication to effectively scale. Although there are several algorithms that fall into the category of ensemble learning, *stacking* [17, 86] or *Super Learning* [84] in particular, has been theoretically proven to represent an asymptotically optimal system for learning and will be a main focus of this dissertation.

The *Super Learner* algorithm combines multiple, typically diverse, learning algorithms into a single, powerful prediction function through a secondary learning process called *metalearning*. Although the Super Learner may require a large amount of computation to perform all of the training required in cross-validating each of the constituent algorithms, many of the steps in the algorithm are computationally independent. We can exploit this computational independence by taking advantage of modern parallel computing architectures. In addition to exploiting the natural parallelism inherent to the Super Learner algorithm, we explore novel approaches to scaling the Super Learner to big data. In Chapter 2, we use cluster-distributed base learning algorithms as well as online or sequential base learners and metalearners. In Chapter 3, we discuss a subset-based ensemble method called the *Subsemble* [72] algorithm, including its theoretical properties and parallelized implementation. With each of these approaches, we present open source software packages and benchmarks for our methods.

In the field of biostatistics, and in industry at large, binary classification or ranking problems are quite common. In many practical situations, the number of available examples of the majority class far outweighs the number of minority class examples, producing training sets with a rare outcome. In the medical field, binary classification algorithms can be trained to diagnose a rare health condition using patient health data [65]. In industry, fraud detection

is a canonical example of a rare outcome problem. Some of the most important data science or applied machine learning problems involve predicting a rare, and sometimes disastrous, event using a limited number of positive training examples. In rare outcome problems, it may not be appropriate to use a model performance metric such as *mean squared error* or *classification error*, so different types of performance metrics are needed. The *Area Under the ROC Curve (AUC)* is probably the most well-known example of these performance measures, however *Partial AUC* [62, 44], F_1*-score* [70] and *H-measure* [37] are also examples of performance measures that may be used in practice. In Chapter 4, we propose a generic computational approach, based on nonlinear optimization, to construct a Super Learner ensemble which will maximize any desired performance metric. In biostatistics, AUC is still one of the most widely-used performance metrics, so AUC is used as a motivating and practical example.

Although the past decade has produced significant efforts towards the goal of scaling of machine learning algorithms to big data, there has been considerably less research on the topic of variance estimation for big data. This lack of inference is noticeably absent. In the machine learning community, journal articles are consistently published containing model performance estimates with no mention of the corresponding confidence intervals for these estimates. The statistics community has a tradition of computing confidence intervals for statistical estimates using a variety of techniques, such as the nonparametric bootstrap. However, in the case of model performance estimates, training a single model on a large dataset may already be a computationally arduous task. Repeating the entire training process hundreds or thousands of times to obtain a bootstrap estimate of variance is simply not feasible in many cases. Although there has been success at reducing the computational burden of traditional bootstrapping, as in the case of the *Bag of Little Bootstraps (BLB)* [1] method, re-sampling based methods for variance estimation still require repeated re-estimation of the quantity of interest. In Chapter 5, we demonstrate how to generate a computationally efficient, *influence function* based estimate of variance, using cross-validated AUC as an example. Influence function based variance estimation removes the requirement of repeated re-estimation of the parameter of interest, and simplifies the variance estimation task into a computationally simple calculation. Since variance estimation via influence functions relies on asymptotic theory, the technique is perfectly suited for large n.

This dissertation establishes a modern and practical framework for high-performance ensemble machine learning and inference on big data. Using AUC as an example performance metric of interest, we present theoretically sound methods as well as efficient implementations in open source software. The goal of this body of work is to provide new scalable approaches to obtaining the highest performing predictive models while optimizing any model performance metric, and further, to provide computationally efficient inference for that estimate.

Chapter 2

Scalable Super Learning

2.1 Introduction

Super Learning is a generalized loss-based ensemble learning framework that was theoretically validated in [84]. This template for learning is applicable to and currently being used across a wide class of problems including problems involving biased sampling, missingness, and censoring. It can be used to estimate marginal densities, conditional densities, conditional hazards, conditional means, conditional medians, conditional quantiles, conditional survival functions, among others [48]. Some applications of Super Learning include the estimation of propensity scores, dose-response functions [47] and optimal dynamic treatment rules [60], for example.

When used for standard prediction, the Super Learner algorithm is a supervised learning algorithm equivalent to "generalized stacking" [17, 86], an ensemble learning technique that combines the output from a set of base learning algorithms via a second-level metalearning algorithm. The Super Learner is built on the theory of cross-validation and has been proven to represent an asymptotically optimal system for learning [84]. The framework allows for a general class of prediction algorithms to be considered for the ensemble.

In this chapter, we discuss some of the history of stacking, review the basic theoretical properties of Super Learning and provide a description of *Online Super Learning*. We discuss several practical implementations of the Super Learner algorithm and highlight the various ways in which the algorithm can scale to big data. In conclusion, we present examples of real-world applications that utilize Super Learning.

2.2 The Super Learner algorithm

The Super Learner algorithm is a generalization of the stacking algorithm introduced in context of neural networks by Wolpert [86] in 1992 and adapted to regression problems by Breiman [17] in 1996. The "Super Learner" name was introduced due to the theoretical *oracle* property and its consequences as presented in [24] and [84].

The distinct characteristic of the Super Learner algorithm in the context of stacking, is its theoretical foundation, which shows that the Super Learner ensemble is asymptotically equivalent to the oracle. Under most conditions, the theory also guarantees that the Super Learner ensemble will perform as least as well as the best individual base learner. Therefore, if model performance is the primary objective in a machine learning problem, the Super Learner with a rich candidate learner library can be used to theoretically and practically guarantee better model performance than can be achieved by a single algorithm alone.

Stacking

Stacking or *stacked generalization*, is a procedure for ensemble learning where a second-level learner, or a *metalearner*, is trained on the output (i.e., the cross-validated predictions, described below) of a collection of *base learners.* The output from the base learners, also called the *level-one* data, can be generated by using cross-validation. Accordingly, the original training data set is often referred to as the *level-zero* data. Although not backed by theory, in the case of large training sets, it may be reasonable to construct the level-one data using predictions from a single, independent test set. In the original stacking literature, Wolpert proposed using *leave-one-out cross-validation* [86], however, due to computational costs and empirical evidence of superior performance, Breiman suggested using 10-fold cross-validation to generate the level-one data [17]. The Super Learner theory requires cross-validation (usually *k-fold cross-validation*, in practice) to generate the level-one data.

The following describes in greater detail how to construct the level-one data. Assume that the training set is comprised of n independent and identically distributed observations, $\{O_1, ..., O_n\}$, where $O_i = (X_i, Y_i)$ and $X_i \in \mathbb{R}^p$ is a vector of covariate or feature values and $Y_i \in \mathbb{R}$ is the outcome. Consider an ensemble comprised of a set of L base learning algorithms, $\{\Psi^1, ..., \Psi^L\}$, each of which is indexed by an algorithm class and a specific set of model parameters. Then, the process of constructing the level-one data will involve generating an $n \times L$ matrix, $\boldsymbol{Z}$, of k-fold *cross-validated predicted values* as follows:

1. The original training set, $\boldsymbol{X}$, is divided at random into $k = V$ roughly-equal pieces (validation folds), $\boldsymbol{X}_{(1)}, ..., \boldsymbol{X}_{(V)}$.
2. For each base learner in the ensemble, Ψ^l, V-fold cross-validation is used to generate n cross-validated predicted values associated with the lth learner. These n-dimensional vectors of cross-validated predicted values become the L columns of $\boldsymbol{Z}$.

The level-one data set, $\boldsymbol{Z}$, along with the original outcome vector, $(Y_1, ..., Y_n) \in \mathbb{R}^n$, is used to train the metalearning algorithm. As a final task, each of the L base learners will be fit on the full training set, and these fits will be saved. The final "ensemble fit" is comprised of the L base learner fits, along with the metalearner fit. To generate a prediction for new data using the ensemble, the algorithm first generates predicted values from each of the L base learner fits, and then passes those predicted values as input to the metalearner fit, which returns the final predicted value for the ensemble.

The historical definition of stacking does not specify restrictions on the type of algorithm used as a metalearner, however the metalearner is often a method which minimizes cross-validated risk of some loss function of interest. For example, ordinary least squares (OLS) can be used to minimize the sum of squared residuals, in the case of a linear model. The Super Learner algorithm can be thought of as the theoretically-supported generalization of stacking to any estimation problem where the goal is to minimize the cross-validated risk of some bounded loss function, including loss functions indexed by nuisance parameters.

Base learners

It is recommended that the *base learner library* include a diverse set of learners (e.g., Linear Model, Support Vector Machine, Random Forest, Neural Net, etc.), however the Super Learner theory does not require any specific level of diversity among the set of the base learners. The library can also include copies of the same algorithm, indexed by different sets of model parameters. For example, the user could specify multiple Random Forests [16], each with a different splitting criterion, tree depth or "mtry" value. Typically, in stacking-based ensemble methods, the prediction functions, $\hat{\Psi}^1, ..., \hat{\Psi}^L$, are fit by training each of base learning algorithms, $\Psi^1, ..., \Psi^L$, on the full training data set and then combining these fits using a metalearning algorithm, Φ. However, there are variants of Super Learning, such as the Subsemble algorithm [72], described in Chapter 2, which learn the prediction functions on subsets of the training data.

The base learners can be any parametric or nonparametric supervised machine learning algorithm. Stacking was originally presented by Wolpert who used neural networks as base learners. Breiman extended the stacking framework to regression problems under the name, *stacked regressions* and experimented with different base learners. For base learning algorithms, he evaluated ensembles of decision trees (with different numbers of terminal nodes), and generalized linear models (GLMs) using subset variable regression (with a different number of predictor variables) or ridge regression [40] (with different ridge parameters). He also built ensembles by combining several subset variable regression models with ridge regression models and found that the added diversity among the base models increased performance. Both Wolpert and Breiman focused their work on using the same underlying algorithm (i.e., neural nets, decision trees or GLMs) with unique tuning parameters as the set of base learners, although Breiman briefly suggested the idea of using heterogeneous base learning algorithms such as "neural nets, nearest-neighbor, etc."

Metalearning algorithm

The metalearner, Φ, is used to find the optimal combination of the L base learners. The $\boldsymbol{Z}$ matrix of cross-validated predicted values, described previously, is used as the input for the metalearning algorithm, along with the original outcome from the level-zero training data, $(Y_1, ..., Y_n)$.

The metalearning algorithm is typically a method designed to minimize the cross-validated risk of some loss function. For example, if your goal is to minimize mean squared prediction error, you could use the least squares algorithm to solve for $(\alpha_1, ..., \alpha_L)$, the weight vector that minimizes the following:

$$\sum_{i=1}^{n}(Y_i - \sum_{l=1}^{L} \alpha_l z_{li})^2$$

Since the set of predictions from the various base learners may be highly correlated, it is advisable to choose a metalearning method that performs well in the presence of collinear predictors. Regularization via Ridge [39] or Lasso [79] regression is commonly used to overcome the issue of collinearity among the predictor variables that make up the level-one data set. Empirically, Breiman found that using Ridge regression as the metalearner often yielded a lower prediction error than using unregularized least squares regression. Of the regularization methods he considered, a linear combination achieved via non-negative least squares (NNLS) [50] gave the best results in terms of prediction error. The NNLS algorithm minimizes the same objective function as the least squares algorithm, but adds the constraint that $\alpha_l \geq 0$, for $l = 1, ..., L$. Le Blanc and Tibshirani [51] also came to the conclusion that non-negativity constraints lead to the most accurate linear combinations of the base learners.

Breiman also discussed desirable theoretical properties that arise by enforcing the additional constraint that $\sum_l \alpha_l = 1$, where the ensemble is a *convex combination* of the base learners. However, in simulations, he shows that the prediction error is nearly the same whether or not $\sum_l \alpha_l = 1$. The convex combination is not only empirically motivated, but also supported by the theory. The oracle results for the Super Learner require a uniformly bounded loss function, and restricting to the convex combination implies that if each algorithm in the library is bounded, the convex combination will also be bounded. In practice, truncation of the predicted values to the range of the outcome variable in the training set is sufficient to allow for unbounded loss functions.

In the Super Learner algorithm, the metalearning method is specified as the minimizer of the cross-validated risk of a loss function of interest, such as *squared error loss* or *rank loss* (1-AUC). If the loss function of interest is unique, unusual or complex, it may be difficult to find an existing machine learning algorithm (i.e., metalearner) that directly or indirectly minimizes this function. However, the optimal combination of the base learners can be estimated using a nonlinear optimization algorithm such as those that are available in the open source **NLopt** library [45]. This particular approach to metalearning, described in detail in Chapter 4, provides a great deal of flexibility to the Super Learner, in the sense that the ensemble can be trained to optimize any complex objective. Historically, in stacking implementations, the metalearning algorithm is often some sort of regularized linear model, however, a variety of parametric and non-parametric methods can used as a metalearner to combine the output from the base fits.

Oracle properties

The *oracle selector* is defined as the estimator, among all possible weighted combinations of the base prediction functions, which minimizes risk under the true data-generating distribution. The oracle result for the cross-validation selector among a set of candidate learners was established in [82] for general bounded loss functions, in [25] for unbounded loss functions under an exponential tail condition, and in [84] for its application to the Super Learner. The oracle selector is considered to be optimal with respect to a particular loss function, given the set of base learners, however it depends on both the observed data and the true distribution, P_0, and thus is unknown. If the true prediction function cannot be represented by a combination of the base learners in the library, then "optimal" will be the closest combination that could be determined to be optimal if the true data-generating mechanism were known. If a training set is large enough, it would theoretically result in the oracle selector. In the original stacking literature, Breiman observed that the ensemble predictor almost always has lower prediction error than the single best base learner, although a proof was not presented in his work [17].

If one of the base learners is a parametric model that happens to contain the true prediction function, this base learner will achieve a parametric rate of convergence and thus the Super Learner achieves an almost parametric rate of convergence, $\log(n)/n$.

Comparison to other ensemble learners

In the machine learning community, the term *ensemble learning* is often associated with *bagging* [15] or *boosting* [29] techniques or particular algorithms such as Random Forest [16]. Stacking is similar to bagging due to the independent training of the base learners, however there are two notable distinctions. The first is that stacking uses a metalearning algorithm to optimally combine the output from the base learners instead of simple averaging, as in bagging. The second is that modern stacking is typically characterized by a diverse set of strong base learners, where bagging is often associated with a single, often weak, base learning algorithm. A popular example of bagging is the Random Forest algorithm which bags classification or regression trees trained on subsets of the feature space. A case could be made that bagging is a special case of stacking which uses the mean as the metalearning algorithm.

From a computational perspective, bagging, like stacking, is a particularly well-suited ensemble learning method for big data since models can be trained independently on different cores or machines within a cluster. However, since boosting utilizes an iterative, hence sequential, training approach, it does not scale as easily to big data problems.

A Bayesian ensemble learning algorithm that is often compared to stacking is Bayesian Model Averaging (BMA). A BMA model is a linear combination of the output from the base learners in which the weights are the posterior probabilities of models. Both BMA and Super Learner use cross-validation as part of the ensemble process. In the case where the true prediction function is contained within the base learner library, BMA is never worse

than stacking and often is demonstrably better under reasonable conditions. However, if the true prediction function is not well approximated by the base learner library, then stacking will significantly outperform BMA [22].

2.3 Super Learner software

This section serves as an overview of several implementations of the Super Learner ensemble algorithm and its variants. The original Super Learner implementation is the **SuperLearner** R package [68], however, we present several new software projects that aim to create more scalable implementations of the algorithm that are suitable for big data. We will discuss a variety of approaches to scaling the Super Learner algorithm:

1. Perform the cross-validation and base learning steps in parallel since these are computationally independent tasks.
2. Train the base learners on subsets of the original training data set.
3. Utilize distributed or parallelized base learning algorithms.
4. Employ online learning techniques to avoid memory-wise scalability limitations.
5. Implement the ensemble (and/or base learners) in a high-performance language such as C++, Java, Scala or Julia, for example.

Currently, there are three implementations of the Super Learner algorithm that have an R interface. The **SuperLearner** and **subsemble** [55] R packages are implemented entirely in R, although they can make use of base learning algorithms that are written in compiled languages as long as there is an R interface available. Often, the main computational tasks of machine learning algorithms accessible via R packages are written in Fortran (e.g., **randomForest**, **glmnet**) or C++ (e.g., **e1071**'s interface to **LIBSVM** [21]), and the runtime of certain algorithms can be reduced by linking R to an optimized BLAS (Basic Linear Algebra Subprograms) library such as **OpenBLAS** [88], **ATLAS** [85] or **Intel MKL** [42]. These techniques may provide additional speed in training, but do not necessarily curtail all memory-related scalability issues. Typically, since at least one copy of the full training data set must reside in memory in R, this is an inherent limitation to the scalability of these implementations.

A more scalable implementation of Super Learner algorithm is available in the **h2oEnsemble** R package [53]. The **H2O Ensemble** project currently uses R to interface with distributed base learning algorithms from the high-performance, open source Java machine learning library, **H2O** [35]. Each of these three Super Learner implementations are at a different stage of development and have benefits and drawbacks compared to the others, but all three projects are being actively developed and maintained.

The main challenge in writing a Super Learner implementation is not implementing the ensemble algorithm itself. In fact, the Super Learner algorithm simply organizes the

cross-validated output from the base learners and applies the metalearning algorithm to this derived data set. Some thought must be given to the parallelization aspects of the algorithm, but this is typically a straightforward exercise, given the computational independence of the cross-validation and base learning steps. One of the main software engineering tasks in any Super Learner implementation is creating a unified interface to a large collection of base learning and metalearning algorithms. A Super Learner implementation must include a novel or third-party machine learning algorithm interface that allows users to specify the base learners in a common format. Ideally, the users of the software should be able to define their own base learning functions that specify an algorithm and set of model parameters in addition to any default algorithms that are provided within the software. The performance of the Super Learner is determined by the combined performance of the base learners, so a having a rich library of machine learning algorithms accessible in the ensemble software is important.

The metalearning methods can use the same interface as the base learners, simplifying the implementation. The metalearner is just another algorithm, although it is common for a non-negative linear combination of the base algorithms to be created using a method like NNLS. However, if the loss function of interest to the user is unrelated to the objective functions associated with the base learning algorithms, then a linear combination of the base learners that minimizes the user-specified loss function can be learned using a nonlinear optimization library such as **NLopt**. In classification problems, this is particularly relevant in the case where the outcome variable in the training set is highly imbalanced. **NLopt** provides a common interface to a number of different algorithms that can be used to solve this problem. There are also methods that allow for constraints such as non-negativity ($\alpha_l \geq 0$) and convexity ($\sum_{l=1}^{L} \alpha_l = 1$) of the weights. As discussed in Chapter 4, using a nonlinear optimization algorithm such as L-BFGS-B, Nelder-Mead or COBYLA, it is possible to find a linear combination of the base learners that specifically minimizes the loss function of interest.

SuperLearner R package

As is common for many statistical algorithms, the original implementation of the Super Learner algorithm was written in R. The **SuperLearner** R package, first released in 2010, is actively maintained with new features being added periodically. This package implements the Super Learner algorithm and provides a unified interface to a diverse set of machine learning algorithms that are available in the R language. The software is extensible in the sense that the user can define custom base-learner function wrappers and specify them as part of the ensemble, however, there are about 30 algorithm wrappers provided by the package by default. The main advantage of an R implementation is direct access to the rich collection of machine learning algorithms that already exist within the R ecosystem. The main disadvantage of an R implementation is memory-related scalability.

Since the base learners are trained independently from each other, the training of the constituent algorithms can be done in parallel. The embarrassingly parallel nature of the

cross-validation and base learning steps of the Super Learner algorithm can be exploited in any language. If there are L base learners and V cross-validation folds, there are $L \times V$ independent computational tasks involved in creating the level-one data. The **SuperLearner** package provides functionality to parallelize the cross-validation step via multicore or SNOW (Simple Network of Workstations) [80] clusters.

The R language and its third-party libraries are not particularly well known for memory efficiency, so depending on the specifications of the machine or cluster that is being used, it is possible to run out of memory while attempting to train the ensemble on large training sets. Since the **SuperLearner** package relies on third-party implementations of the base learning algorithms, the scalability of **SuperLearner** is tied to the scalability of the base learner implementations used in the ensemble. When selecting a single model among a group of candidate algorithms based on cross-validated model performance, this is computationally equivalent to generating the level-one data in the Super Learner algorithm. If cross-validation is already being employed as a means of grid-search based model selection among a group of candidate learning algorithms, the addition of the metalearning step is a computationally minimal burden. However, a Super Learner ensemble can result in a significant boost in overall model performance over a single base learner model.

subsemble R package

The **subsemble** R package implements the Subsemble algorithm [72], a new variant of Super Learning, which ensembles base models trained on subsets of the original data. Specifically, the disjoint union of the subsets is the full training set. As a special case, where the number of subsets = 1, the package also implements the Super Learner algorithm. The algorithm and the package are described in greater detail in Chapter 3, however we will briefly mention the algorithm and software in this chapter.

The Subsemble algorithm can be used as a stand-alone ensemble algorithm, or as base learning algorithm in the Super Learner algorithm. Empirically, it has been shown that Subsemble can provide better prediction performance than fitting a single algorithm once on the full available dataset [72], although this is not always the case.

An oracle result shows that Subsemble performs as well as the best possible combination of the subset-specific fits. The Super Learner has more powerful asymptotic properties; it performs as well as the best possible combination of the the base learners trained on the full data set. However, when used as a stand-alone ensemble algorithm, Subsemble offers great computational flexibility, in that the training task can be scaled to any size by changing the number, or size, of the subsets. This allows the user to effectively "flatten" the training process into a task that is compatible with available computational resources. If parallelization is used effectively, all subset-specific fits can be trained at the same time, drastically increasing the speed of the training process. Since the subsets are typically much smaller than the original training set, this also reduces the memory requirements of each node in your cluster. The computational flexibility and speed of the Subsemble algorithm offers a unique solution to scaling ensemble learning to big data problems.

In the **subsemble** package, the J subsets can be created by the software at random, or the subsets can be explicitly specified by the user. Given L base learning algorithms and J subsets, a total of $L \times J$ subset-specific fits will be trained and included in the Subsemble (by default). This construction allows each base learning algorithm to see each subset of the training data, so in this sense, there is a similarity to ensembles trained on the full data. To distinguish the variations on this theme, this type of ensemble construction is referred to as a "cross-product" Subsemble. The **subsemble** package also implements what are called "divisor" Subsembles, a structure that can be created if the number of unique base learning algorithms is a divisor of the number of subsets. In this case, there are only J total subset-specific fits that make up the ensemble, and each learner only sees approximately n/J observations from the full training set (assuming the subsets are of equal size). For example, if $L = 2$ and $J = 10$, then each of the two base learning algorithms would be used to train five subset-specific fits and would only see a total of 50% of the original training observations. This type of Subsemble allows for quicker training, but will typically result in less accurate models. Therefore, the "cross-product" method is the default Subsemble type in the software.

An algorithm called Supervised Regression Tree Subsemble or "SRT Subsemble" [71] is also on the development road-map for the **subsemble** package. SRT Subsemble is an extension of the regular Subsemble algorithm, which provides a means of learning the optimal number and constituency of the subsets. This method incurs an additional computational cost, but can provide greater model performance for the Subsemble.

H2O Ensemble

The **H2O Ensemble** project contains an implementation of the Super Learner ensemble algorithm which is built upon the distributed, open source, Java-based machine learning learning platform for big data, **H2O**. **H2O Ensemble** is currently implemented as a stand-alone R package called **h2oEnsemble** which makes use of the **h2o** package, the R interface to the **H2O** platform. There are a handful of powerful supervised machine learning algorithms supported by the **h2o** package, all of which can be used as base learners for the ensemble. This includes a high-performance method for deep learning, which allows the user to create ensembles of deep neural nets or combine the power of deep neural nets with other algorithms such as Random Forest or Gradient Boosting Machines (GBMs) [30].

Since the **H2O** machine learning platform was designed with big data in mind, each of the **H2O** base learning algorithms is scalable to very large training sets and enables parallelism across multiple nodes and cores. The **H2O** platform is comprised of a distributed in-memory parallel computing architecture and has the ability to seamlessly use datasets stored in Hadoop Distributed File System (HDFS), Amazon's S3 cloud storage, NoSQL and SQL databases in addition to CSV files stored locally or in distributed filesystems. The **H2O Ensemble** project aims to match the scalability of the **H2O** algorithms, so although the ensemble uses R as its main user interface, most of the computations are performed in Java via **H2O** in a distributed, scalable fashion.

There are several publicly available benchmarks of the **H2O** algorithms. Notably, the **H2O** GLM implementation has been benchmarked on a training set of 1 billion observations [34]. This benchmark training set is derived from the "Airline Dataset" [77], which has been called the "Iris dataset for big data". The 1 billon row training set is a 42GB CSV file with 12 feature columns (9 numerical features, 3 categorical features with cardinalities 30, 376 and 380) and a binary outcome. Using a 48-node cluster (8 cores on each node, 15GB of RAM and 1Gb interconnect speed), the **H2O** GLM can be trained in 5.6 seconds. The **H2O** algorithm implementations aim to be scalable to any size dataset so that all of the available training set, rather than a subset, can be used for training models.

h2oEnsemble takes a different approach to scaling the Super Learner algorithm than the **subsemble** or **SuperLearner** R packages. Since the **subsemble** and **SuperLearner** ensembles rely on third-party R algorithm implementations which are typically single-threaded, the parallelism of these two implementations occurs in the cross-validation and base learning steps. In the **SuperLearner** implementation, the ability to take advantage of multiple cores is strictly limited by the number of cross-validation folds and number of base learners. With **subsemble**, the scalability of the ensemble can be improved by increasing the number of subsets used, however, this may lead to a decrease in model performance. Unlike most third-party machine learning algorithms that are available in R, the **H2O** base learning algorithms are implemented in a distributed fashion and can scale to all available cores in a multicore or multi-node cluster. In the current release of **h2oEnsemble**, the cross-validation and base learning steps of the ensemble algorithm are performed in serial, however, each serial training step is maximally parallelized across all available cores in a cluster. The **h2oEnsemble** implementation could possibly be re-architected to parallelize the the cross-validation and base learning steps, however, it is unknown at this time how that may affect runtime performance.

h2oEnsemble R code example

The following R code example demonstrates how to create an ensemble of a Random Forest and two Deep Neural Nets using the **h2oEnsemble** R interface. In the code below, an example shows the current method for defining custom base learner functions. The **h2oEnsemble** package comes with four base learner function wrappers, however, to create a base learner with non-default model parameters, the user can pass along non-default function arguments as shown. The user must also specify a metalearning algorithm, and in this example, a GLM wrapper function is used.

```
library("SuperLearner") # For "SL.nnls" metalearner function
library("h2oEnsemble")

# Create custom base learner functions using non-default model params:
h2o_rf_1 <- function(..., family = "binomial",
                          ntree = 500,
                          depth = 50,
                          mtries = 6,
                          sample.rate = 0.8,
                          nbins = 50,
                          nfolds = 0)
     h2o.randomForest.wrapper(..., family = family, ntree = ntree,
     depth = depth, mtries = mtries, sample.rate = sample.rate,
     nbins = nbins, nfolds = nfolds)
}

h2o_dl_1 <- function(..., family = "binomial",
                          nfolds = 0,
                          activation = "RectifierWithDropout",
                          hidden = c(200,200),
                          epochs = 100,
                          l1 = 0,
                          l2 = 0)
     h2o.deeplearning.wrapper(..., family = family, nfolds = nfolds,
     activation = activation, hidden = hidden, epochs = epochs,
     l1 = l1, l2 = l2)
}

h2o_dl_2 <- function(..., family = "binomial",
                          nfolds = 0,
                          activation = "Rectifier",
                          hidden = c(200,200),
                          epochs = 100,
                          l1 = 0,
                          l2 = 1e-05)
     h2o.deeplearning.wrapper(..., family = family, nfolds = nfolds,
     activation = activation, hidden = hidden, epochs = epochs,
     l1 = l1, l2 = l2)
}
```

The function interface for the `h2o.ensemble` function follows the same conventions as the other **h2o** R package algorithm functions. This includes the `x` and `y` arguments, which are the column names of the predictor variables and outcome variable, respectively. The

`data` object is a reference to the training data set, which exists in Java memory. The `family` argument is used to specify the type of prediction (i.e., classification or regression). The `predict.h2o.ensemble` function uses the `predict(object, newdata)` interface that is common to most machine learning software packages in R. After specifying the base learner library and the metalearner, the ensemble can be trained and tested:

```
# Set up the ensemble
learner <- c("h2o_rf_1", "h2o_dl_1", "h2o_dl_2")
metalearner <- "SL.nnls"

# Train the ensemble using 2-fold CV to generate level-one data
# More CV folds will increase runtime, but should increase performance
fit <- h2o.ensemble(x = x, y = y, data = data, family = "binomial",
                    learner = learner, metalearner = metalearner,
                    cvControl = list(V = 2))

# Generate predictions on the test set
pred <- predict(fit, newdata)
```

Performance benchmarks

h2oEnsemble was benchmarked on Amazon's Elastic Compute Cloud (EC2) in order to demonstrate the practical use of the Super Learner algorithm on big data. The instance type used across all benchmarks is EC2's "c3.8xlarge" type, which has 32 virtual CPUs (vCPUs), 60 GB RAM and 10 Gigabit interconnect speed. Since **H2O**'s algorithms are distributed and the benchmarks were performed on multi-node clusters, the node interconnect speed is critical to performance. Further computational details are given in the next section. A few different cluster architectures were evaluated, including a 320 vCPU and 96 vCPU cluster (ten and three nodes each, respectively), as well as a single workstation with 32 vCPUs.

The **h2oEnsemble** package currently provides four base learner function wrappers for the **H2O** algorithms. The following supervised learning algorithms are currently supported: Generalized linear models (GLMs) with Elastic Net regularization, Gradient Boosting (GBM) with regression and classification trees, Random Forest and Deep Learning (multi-layer feed-forward neural networks). These algorithms support both classification and regression problems, although this benchmark is a binary classification problem. Various subsets of the "HIGGS" dataset [8] (28 numeric features; binary outcome with balanced training classes) were used to assess the scalability of the ensemble. An independent test set of 500,000 observations (the same test set as in [8]) was used to measure the performance.

The base learner library consists of the three base learners: a Random Forest of 500 trees and two Deep Neural Nets (one with dropout [38] and the other with L_2-regularization). In the ensemble, 2-fold cross-validation was used to generate the level-one data and both a GLM and NNLS metalearner were evaluated. An increase in the number of validation folds will

likely increase ensemble performance, however, this will increase training time (2-fold is the recommended minimum required to retain the desirable asymptotic properties of the Super Learner algorithm). Although the performance is measured by AUC in the benchmarks, the metalearning algorithms used (GLM and NNLS) are not designed to maximize AUC. By using a higher number of cross-validation folds, an AUC-maximizing metalearning algorithm and a larger and more diverse base learning library, the performance of the ensemble will likely increase. Thus, the ensemble AUC estimates shown in Table 2.1 are conservative example of performance.

	RF	DNN-Dropout	DNN-L_2	Ensemble: GLM, NNLS
$n = 1,000$	0.730	0.683	0.660	0.729, 0.730
$n = 10,000$	0.785	0.722	0.707	0.786, 0.788
$n = 100,000$	0.825	0.812	0.809	0.818, 0.819
$n = 1,000,000$	0.823	0.812	0.838	0.841, 0.841
$n = 5,000,000$	0.839	0.817	0.845	0.852, 0.851

Table 2.1: Base learner model performance (test set AUC) compared to **h2oEnsemble** model performance performance using 2-fold CV (ensemble results for both GLM and NNLS metalearners).

	Cluster (320)	Cluster (96)	Workstation (32)
$n = 1,000$	2.1 min	1.1 min	0.5 min
$n = 10,000$	3.3 min	2.5 min	2.0 min
$n = 100,000$	3.5 min	5.9 min	11.0 min
$n = 1,000,000$	14.9 min	42.6 min	102.9 min
$n = 5,000,000$	62.3 min	200.2 min	-

Table 2.2: Training times (minutes) for **h2oEnsemble** with a 3-learner library using various cluster configurations, including a single workstation with 32 vCPUs. The number of vCPUs for each cluster is noted in parenthesis. Results for $n = 5$ million are not available for the single workstation setting.

Memory profiling was not performed as part of the benchmarking process. The source code for the benchmarks is available on GitHub [52].

Computational details

All benchmarks were performed on 64-bit linux instances (type "c3.8xlarge" on Amazon EC2) running Ubuntu 14.04. Each instance has 32 vCPUs, 60 GB RAM and uses Intel Xeon E5-2680 v2 (Ivy Bridge) processors. In the 10-node (320 vCPU) cluster and 3-node (96 vCPU) cluster that were used, the nodes have a 10 Gigabit interconnect speed. Since the

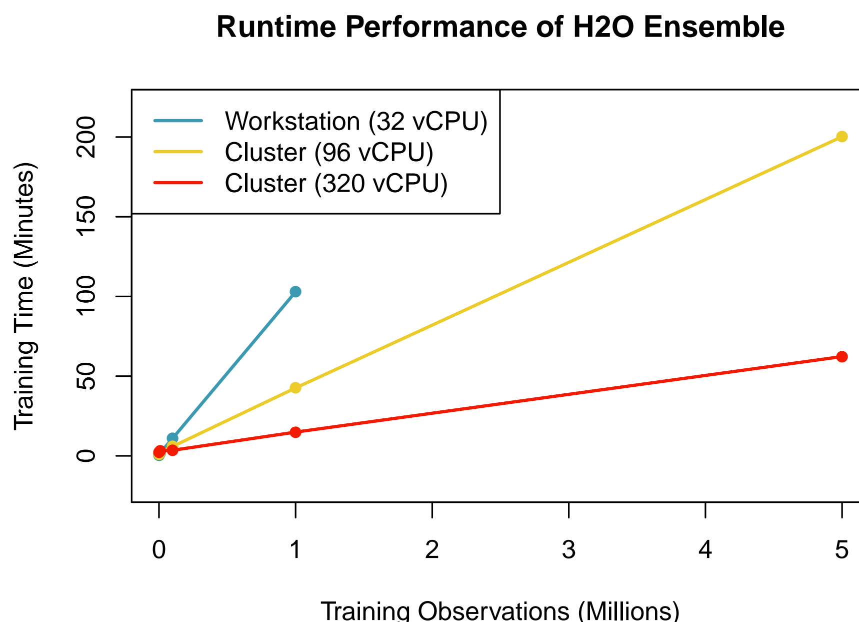

Figure 2.1: Training time for **h2oEnsemble** for training sets of increasing size (subsets of the HIGGS dataset). This ensemble included the three base learners, listed previously.

H2O base learner algorithms are distributed across all the cores in a multi-node cluster, it is recommended to use 10 Gigabit interconnect (or greater). These results are for **h2oEnsemble** R package version 0.0.3, R version 3.1.1, **H2O** version 2.9.0.1593 and Java version 1.7.0_65 using the **OpenJDK** Runtime Environment (IcedTea 2.5.3).

2.4 Online Super Learning

Another approach to creating a scalable Super Learner implementation is by using sequential, or *online learning* techniques. The Online Super Learner uses both online base learners and an online metalearner to achieve out-of-core performance. Online learning methods offer a solution to the demanding memory requirements of *batch learning* (where the algorithm sees all the data at once), which typically requires the full training set to fit into RAM. A unique advantage of online learning, as opposed to batch learning, is that the algorithm fit can respond to changes or drift within the data generating mechanism over time.

Optimization in online learning

Training an estimator typically reduces to some sort of optimization problem. There are many ways to solve different optimization problems, and one such way is gradient descent (GD). The algorithm is parameterized by γ, which controls the step size. If the number of training examples is very large, one iteration of the algorithm may take a long time because computing the gradient on the full data set is slow.

An alternative to GD is stochastic gradient descent (SGD). The algorithm is similar, except a step is taken using an estimate of the gradient based on one, or in the case of the mini-batch version, $m \ll n$, observations. In [13], it is shown that SGD converges at the same rate as GD. Though SGD steps are noisy and more are required, each step only uses a small fraction of the data and therefore can be computed very quickly. Because of this, reasonably good results can be obtained in only a few passes through the data, which may take many passes in traditional GD. This results in a much faster algorithm. Another advantage of SGD is that it can be used in an online setting, where an essentially infinite stream of observations are being collected and consumed by the algorithm in a sequential process.

The Online Super Learner algorithm

Assume a finite set, $X_1, ..., X_n$, or stream of observations, $X_1, ..., X_n, X_{n+1}, ...$, from some some distribution, P_0, or some sequence of distributions. In the Online Super Learner algorithm, the base learning algorithms are online learners. We will consider the sequential learning case where we update each of the base learners after observing each new training point, X_i.

Since the online base learners are all observing the same training point at the same time, all of the base learners can be updated simultaneously at each iteration (either serially, or in parallel). In the batch version of Super Learning, after cross-validation is used to generate the level-one data from the base learners, a metalearning algorithm is fit using the level-one data. Since standard k-fold cross-validation is a batch operation, we must consider an alternative sequential approach to generating the level-one data and updating the metalearner fit in the Online Super Learner algorithm. There may be many possible ways to do this, but we will discuss one particular method, which was implemented in **Vowpal Wabbit Ensemble** [56].

Consider two types of sequential learning – single-pass and multi-pass mode, in which the training process is completed in one pass or multiple passes through the data, respectively. If the training samples are streaming in sequentially (in other words, the training set is not a previously-collected set of examples), then the single-pass mode is necessary. However, if all of the training samples have already been collected, then it is possible to take multiple passes through the data.

In both the single and multi-pass mode, there is an option to designate a holdout set of observations that are never used in training. This is not strictly necessary, but can be performed in order to collect information about estimated model performance (where model

performance is evaluated incrementally on the holdout set). In practice, for single-pass mode, sequential validation of each observation is sufficient, and thus a designated holdout set is not required. This is because in single-pass mode, each observation can be used first as a test sample for the current fit, and then as a training sample in the next iteration of the algorithm. However, in multi-pass mode, a holdout set is required in order to generate an honest estimate of model performance.

To distinguish between two types of observations that are not used in training – we will reserve the term "holdout set" for observations used strictly for model evaluation. A second group of held-out observations, the "validation set", will be set aside and used to generate predicted values using the existing model fit. The resulting predicted values will serve as the level-one data, which is used to train the metalearner.

We define two tuning parameters, ρ and ν, to control the construction of the level-one data for the metalearner. In multi-pass mode, the parameter ρ controls the "validation period", or the number of training examples between each (non-holdout) validation sample, as observed in sequence. For example, if $\rho = 10$, then every 10th non-holdout sample would be set aside to be included in the rolling validation set. In this case, the algorithm sees 10 training samples for every validation sample. All non-validation, non-holdout sample points are used in training.

The second tuning parameter is ν, the "running validation size", or the number of most recent (and non-holdout, if multi-pass) validation samples to be retained (at any iteration) for updating the metalearning fit. This parameter controls the amount of data that needs to reside in memory at any given time – bigger ν translates into a more informed metalearning fit, but demands higher memory requirements. In theory, the ν parameter could be adaptive, however, in this specificiation of the algorithm, we will consider it to be a fixed value.

Fixing or limiting the size of the level-one data for the ensemble at a given iteration via the ν parameter allows the user to make use of batch learning algorithms in the metalearning process. Regardless, this implementation of the Online Super Learner algorithm is still considered to be a sequential learner, since the ensemble fit is learned incrementally. When using a batch algorithm as part of the metalearning process, it is advisable to choose a large enough ν, or validation set size, in order to successfully train the metalearner. This is similar to mini-batch learning in SGD, where $m > 1$ examples are retained at any given time for training (in that case, the m samples are used to estimate the gradient).

In an alternative formulation, where sequential learners are used for both the base learners and the metalearning algorithm, then $\nu = 1$ and the metalearner fit will be updated with one training sample at a time, in sync with the updates of the base learner fits. Or, in the case of mini-batch SGD, then $\nu = m$ and m training observations are processed at each iteration.

A practical Online Super Learner

Vowpal Wabbit (VW) [49] is a fast out-of-core learning software developed by John Langford that was first released in 2007 and is still very actively maintained. **VW** implements SGD (and a few other optimization routines) for a variety of estimators with a primary goal of

being computationally very fast and scaling well to large datasets. The software also allows for estimators to easily be fit on different subsets and cross products of subsets of independent variables. **VW** is written in C++ and can be used as a library in other software.

The default learning algorithm in **VW** is a variant of online gradient descent. Various extensions, such as conjugate gradient (CG), mini-batch, and data-dependent learning rates, are included. **Vowpal Wabbit** is very useful for sparse data, so a **VW**-based Online Super Learner will also be useful for sparse data.

A proof-of-concept version of the Online Super Learner algorithm, **Vowpal Wabbit Ensemble**, was implemented in C++, and uses the **Vowpal Wabbit** machine learning library to provide the base learners. The additional dependencies are the **Boost** C++ library [23] and a C implementation (**f2c** translation from Fortran) [73] of the NNLS algorithm for the metalearning process. The tuning parameters mentioned in Section 2.4 give the user fine-grained control over the sequential learning process.

To demonstrate computational performance, the Online Super Learner was trained on the "Malicious URL Dataset" [61] which has 2.4 million rows and 3.2 million sparse features. Using three algorithms to make up the ensemble, the Online Super Learner made a single pass over the data. The training process on a single 2.3 GHz Intel Core i7 processor took approximately 25 seconds.

2.5 Super Learner in practice

There are many applications of the Super Learner algorithm, however, due to its superior performance over single algorithms and other ensemble learners, it is often used in situations where model performance is valued over other factors, such as training time and model simplicity. The algorithm has been used in a wide variety of applications in field of biostatistics. In the context of prediction, Super Learner has been used to predict virologic failure among HIV-infected individuals [65], HIV-1 drug resistance [76] and mortality among patients in intensive care units [66], for example.

Super Learning can be used at iterative time points to evaluate the the relative importance of each measured variable on an outcome. This can provide continuously changing prediction of the outcome and evaluation of which clinical variables likely drive a particular outcome. [41]

In the context of learning the optimal dynamic treatment rule, a non-sequential Super Learner seeks to directly maximize the mean outcome under the two time point rule [60]. This implementation relies on sequential candidate estimators based on various loss functions. Super Learner has also been used to estimate both the generalized propensity score and the dose-response function [47].

2.6 Conclusion

In this chapter, we discussed the Super Learner algorithm, a theoretically-backed non-parametric ensemble prediction method. Super Learner fits a set of base learners, and combines the fits through a second-level metalearning algorithm using cross-validation. We discussed several software implementations of the algorithm, and provided code examples and benchmarks of a distributed, scalable implementation called **H2O Ensemble**. Further, an online implementation of the Super Learner algorithm was presented as an alternative to the batch version as another approach to achieving scalability for big data. Lastly, we described examples of practical applications of the Super Learner algorithm.

Chapter 3

Subsemble: Divide and Recombine

3.1 Introduction

As massive datasets become increasingly common, new scalable approaches to prediction are needed. Given that memory and runtime constraints are common in practice, it is important to develop practical machine learning methods that perform well on big data sets in a fixed computational resource setting. Procedures using subsets from a training set are promising tools for prediction with large-scale data sets [90]. Recent research has focused on developing and evaluating the performance of various subset-based prediction procedures. Subsetting procedures in machine learning construct subsets from the available training data, then train an algorithm on each subset, and finally combine the results across the subsets to form a final prediction. Prediction methods operating on subsets of the training data can take advantage of modern computational resources, since machine learning on subsets can be massively parallelized.

Bagging [15], or bootstrap aggregating, is a classic example of a subsampling prediction procedure. Bagging involves drawing many bootstrap samples of a fixed size, fitting the same underlying algorithm on each bootstrap sample, and obtaining the final prediction by averaging the results across the fits. Bagging can lead to significant model performance gains when used with weak or unstable algorithms such as classification or regression trees. The bootstrap samples are drawn with replacement, so each bootstrap sample of size n contains approximately 63.2% of the unique training examples, while the remainder of the observations contained in the sample are duplicates. Therefore, in bagging, each model is fit using only a subset of the original training observations. The drawback of taking a simple average of the output from the subset fits is that the predictions from each of the fits are weighted equally, regardless of the individual quality of each fit. The performance of a bagged fit can be much better compared to that of a non-bagged algorithm, but a simple average is not necessarily the optimal combination of a set of base learners.

An *average mixture* (AVGM) procedure for fitting the parameter of a parametric model has been studied by [90]. AVGM partitions the full available dataset into disjoint subsets,

estimates the parameter within each subset, and finally combines the estimates by simple averaging. Under certain conditions on the population risk, the AVGM can achieve better efficiency than training a parametric model on the full data. A *subsampled average mixture* (SAVGM) procedure, an extension of AVGM, is proposed in [90] and is shown to provide substantial performance benefits over AVGM. As with AVGM, SAVGM partitions the full data into subsets, and estimates the parameter within each subset. However, SAVGM also takes a single subsample from each partition, re-estimates the parameter on the subsample, and combines the two estimates into a so-called "subsample-corrected" estimate. The final parameter estimate is obtained by simple averaging of the subsample-corrected estimates from each partition. Both procedures have a theoretical backing, however, the results rely on using parametric models.

An ensemble method for classification with large-scale datasets, using subsets of observations to train algorithms, and combining the classifiers linearly, was implemented and discussed in the case study of [59] at Twitter, Inc.

While not a subset method, boosting, formulated by [29], is an example of an ensemble method that differentiates between the quality of each fit in the ensemble. Boosting iterates the process of training a weak learner on the full data set, then re-weighting observations, with higher weights given to poorly classified observations from the previous iteration. However, boosting is not a subset method because all observations are iteratively re-weighted, and thus all observations are needed at each iteration. Boosting is also a sequential algorithm, and thus cannot be easily parallelized, although distributed implementations of boosting algorithms do exist.

Another non-subset ensemble method that differentiates between the quality of each fit is the Super Learner algorithm of [84], which generalizes and establishes the theory for stacking procedures developed by [86] and extended by [17]. Super Learner learns the optimal weighted combination of a base learner library of candidate base learner algorithms by using cross-validation and a second-level metalearning algorithm. Super Learner generalizes stacking by allowing for general loss functions and hence a broader range of estimator combinations.

The Subsemble algorithm is a method proposed in [72], for combining results from fitting the same underlying algorithm on different subsets of observations. Subsemble is form of supervised stacking and is similar in nature to the Super Learner algorithm, with the distinction that base learner fits are trained on subsets of the data instead of the full training set. Subsemble can also accommodate multiple base learning algorithms, with each algorithm being fit on each subset. The approach has many benefits and differs from other ensemble methods in a variety of ways.

First, any type of underlying algorithm, parametric or nonparametric, can be used. Instead of simply averaging subset-specific fits, Subsemble differentiates fit quality across the subsets and learns a weighted combination of the subset-specific fits. To evaluate fit quality and determine the weighted combination, Subsemble uses cross-validation, thus using independent data to train the base learners and learn the weighted combination. Finally, Subsemble has desirable statistical performance and can improve prediction quality on both

small and large datasets.

This chapter focuses on the statistical properties and performance of the Subsemble algorithm. We present an oracle result for Subsemble, showing that Subsemble performs as well as the best possible combination of the subset-specific fits. Empirically, it has been shown that Subsemble performs well as a prediction procedure for moderate and large sized datasets [72]. Subsemble can, and often does, provide better prediction performance than fitting a single base algorithm on the full available dataset.

3.2 Subsemble ensemble learning

Let $X \in \mathbb{R}^p$ denote a real valued vector of covariates and let $Y \in \mathbb{R}$ represent a real valued outcome value with joint distribution, $P_0(X, Y)$. Assume a training set consists of n independent and identically distributed observations, $O_i = (X_i, Y_i)$ of $O \sim P_0$. The goal is to learn a function $\hat{f}(X)$ for predicting the outcome, Y, given the input X.

Assume that there is a set of L machine learning algorithms, $\Psi^1, ..., \Psi^L$, where each is indexed by an algorithm class and a specific set of model parameters. These algorithms can be any class of supervised learning algorithms, such as a Random Forest, Support Vector Machine or a linear model. The base learner library can also include copies of the same algorithm, specified by different sets of tuning parameters. Typically, in stacking-based ensemble methods, functions, $\hat{\Psi}^1, ..., \hat{\Psi}^L$, are learned by applying base learning algorithms, $\Psi^1, ..., \Psi^L$, to the full training data set and then combining these fits using a metalearning algorithm, Φ, trained on the cross-validated predicted values from the base learners. Historically, in stacking methods, the metalearning method is often chosen to be some sort of regularized linear model, such as non-negative least squares (NNLS) [17], however a variety of parametric and non-parametric methods can be used to learn the optimal combination output from the base fits. In the Super Learner algorithm, the metalearning algorithm is specified as a method that minimizes the cross-validated risk of some particular *loss function* of interest, such as *negative log-likelihood loss* or squared error loss.

The Subsemble algorithm

Instead of using the entire dataset to obtain a single fit, $\hat{\Psi}^l$, for each base learner, Subsemble applies algorithm Ψ^l to multiple different subsets of the available observations. The subsets are created by partitioning of the entire training set into J disjoint subsets. The subsets are typically created randomly and of the same size. With L unique base learners and J subsets, the ensemble is then comprised of a total of $L \times J$ subset-specific fits, $\hat{\Psi}^l_j$. As in the Super Learner algorithm, Subsemble obtains the optimal combination of the fits by minimizing cross-validated risk through cross-validation.

In stacking algorithms, k-fold cross-validation is often used to generate what is called the level-one data. The level-one data is the input data to the metalearning algorithm, which is different from the level-zero data, or the original training data set. Assume that the

number of cross-validation folds is chosen to be $k = V$ folds. In the Super Learner algorithm, the level-one data consists of the V sets of cross-validated predicted values from each base learning algorithm. With L base learners and a training set of n observations, the level-one data will be an $n \times L$ matrix, and serve as the design matrix in the metalearning task.

In the Subsemble algorithm, a modified version of k-fold cross-validation is used to obtain the level-one data. Each of the J subsets are partitioned further into V folds, so that the vth validation fold spans across all J subsets. For each base learning algorithm, Ψ^l, the $(j,v)^{th}$ iteration of the cross-validation process is defined as follows:

1. Train the $(j,v)^{th}$ subset-specific fit, $\hat{\Psi}^l_j$, by applying Ψ^l to the observations that are in folds $\{1,...,V\} \setminus v$, but restricted to subset j. The training set used here is a subset of the jth subset and contains $\frac{n(V-1)}{JV}$ observations.

2. Using the subset-specific fit, $\hat{\Psi}^l_j$, predicted values are generated for the entire v^{th} validation fold, including those observations that are not in subset j. The size of the validation set for the $(j,v)^{th}$ iteration is $\frac{Jn}{V}$.

This unique version of cross-validation generates predicted values for all n observations in the full training set, while only training on subsets of data. A total of $L \times J$ learner-subset models are cross-validated, resulting in a $n \times (L \times J)$ matrix of level-one data that can be used to train the metalearning algorithm, Φ. A diagram depicting the Subsemble algorithm using a single underlying base learning algorithm, ψ, is shown in Figure 3.2.

More formally, define $P_{n,v}$ as the empirical distribution of the observations not in the vth fold. For each observation i, define $P_{n,v(i)}$ to be the empirical distribution of the observations not in the fold containing observation i. The optimal combination is selected by applying the metalearning algorithm Φ to the following redefined set of n observations: $(\tilde{X}_i, Y_i)$, where $\tilde{X}_i = \{\tilde{X}^l_i\}_{l=1}^L$, and $\tilde{X}^l_i = \{\hat{\Psi}^l_j(P_{n,v(i)})(X_i)\}_{j=1}^J$. That is, for each i, the level-one input vector, $\tilde{X}_i$, consists of the $L \times J$ predicted values obtained by evaluating the $L \times J$ subset-specific estimators trained on the data excluding the $v(i)^{\text{th}}$ fold, at X_i. Note that the level-one dataset in Subsemble has J times as many columns as the level-one dataset generated in the Super Learner algorithm.

The cross-validation process is used only to generate the level-one data, so as a separate task, $L \times J$ final subset-specific fits are trained, using the entire subset j as the training set for each $(l,j)^{th}$ fit. The final Subsemble fit is comprised of the $L \times J$ subset-specific fits, $\hat{\Psi}^l_j$, and a metalearner fit, $\hat{\Phi}$. Pseudocode for the Subsemble algorithm is shown in Figure 3.2.

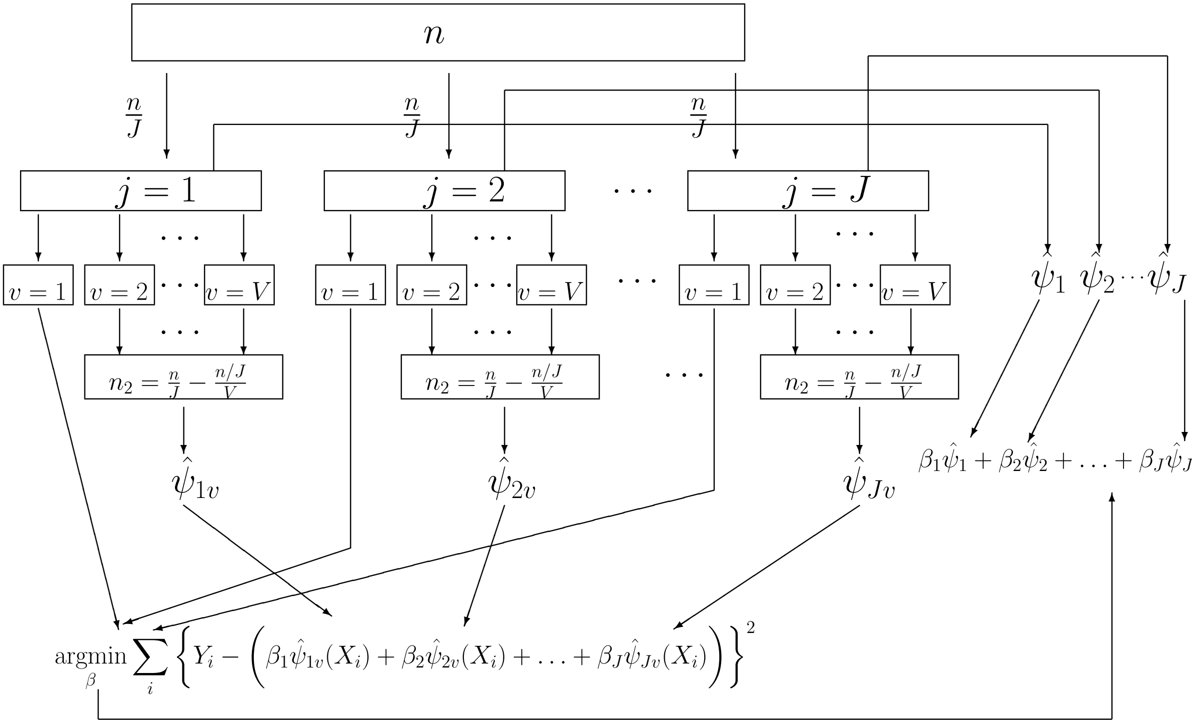

Figure 3.1: Diagram of the Subsemble procedure using a single base learner ψ, and linear regression as the metalearning algorithm.

Algorithm 1 Subsemble

- Assume n observations $(X_i,\ Y_i)$
- Partition the n observations into J disjoint subsets
- Base learning algorithms: $\Psi^1, \ldots, \Psi^L$
- Metalearner algorithm: Φ
- Optimal combination: $\hat{\Phi}(\{\hat{\Psi}^l_1, \ldots, \hat{\Psi}^l_J\}_{l=1}^L)$

for $j \leftarrow 1 : J$ **do**
  `// Create subset-specific base learner fits`
  for $l \leftarrow 1 : L$ **do**
    $\hat{\Psi}^l_j \leftarrow$ apply Ψ^l to observations i such that $i \in j$
  end for
  `// Create V folds`
  Randomly partition each subset j into V folds
end for

for $v \leftarrow 1 : V$ **do**
  `// CV fits`
  for $l \leftarrow 1 : L$ **do**
    $\hat{\Psi}^l_{j,v} \leftarrow$ apply Ψ^l to observations i such that $i \in j,\ i \notin v$
  end for
  for $i : i \in v$ **do**
    `// Predicted values` $\tilde{X}_i \leftarrow \left(\{\hat{\Psi}^l_{1,v}(X_i), \ldots, \hat{\Psi}^l_{J,v}(X_i)\}_{l=1}^L\right)$
  end for
end for

$\hat{\Phi} \leftarrow$ apply Φ to training data $(Y_i, \tilde{X}_i)$

$\hat{\Phi}(\{\hat{\Psi}^l_1, \ldots, \hat{\Psi}^l_J\}_{l=1}^L) \leftarrow$ final prediction function

Figure 3.2: Pseudocode for the Subsemble algorithm.

Oracle result for Subsemble

The following oracle result gives a theoretical guarantee of Subsemble's performance, was proven in [72] and follows directly from the work of [84]. Theorem 1 has been extended from the original formulation in order to allow for L base learners instead of a single base learner. The squared error loss function is used as an example in Theorem 1.

Theorem 1. *Assume the metalearner algorithm $\hat{\Phi} = \hat{\Phi}_\beta$ is indexed by a finite dimensional parameter $\beta \in \mathbf{B}$. Let $\mathbf{B}_n$ be a finite set of values in $\mathbf{B}$, with the number of values growing at most polynomial rate in n. Assume there exist bounded sets $\mathbf{Y} \in \mathbb{R}$ and Euclidean $\mathbf{X}$ such that $P((Y,X) \in \mathbf{Y} \times \mathbf{X}) = 1$ and $P(\hat{\Psi}^l(P_n) \in \mathbf{Y}) = 1$ for $l = 1, \ldots, L$.*

Define the cross-validation selector of β as

$$\beta_n = \arg\min_{\beta \in \mathbf{B}_n} \sum_{i=1}^{n} \left\{ Y_i - \hat{\Phi}_\beta(\tilde{X}_i) \right\}^2$$

and define the oracle selector of β as

$$\tilde{\beta}_n = \arg\min_{\beta \in \mathbf{B}_n} \frac{1}{V} \sum_{v=1}^{V} E_0 \left[\left\{ E_0[Y|X] - \hat{\Phi}_\beta(P_{n,v}) \right\}^2 \right]$$

Then, for every $\delta > 0$, there exists a constant $C(\delta) < \infty$ (defined in [82]) such that

$$\begin{aligned} &E \frac{1}{V} \sum_{v=1}^{V} E_0 \left[\left\{ E_0[Y|X] - \hat{\Phi}_{\beta_n}(P_{n,v}) \right\}^2 \right] \\ &\leq (1+\delta) E \frac{1}{V} \sum_{v=1}^{V} E_0 \left[\left\{ E_0[Y|X] - \hat{\Phi}_{\tilde{\beta}_n}(P_{n,v}) \right\}^2 \right] + C(\delta) \frac{V \log n}{n} \end{aligned}$$

As a result, if none of the subset-specific learners converge at a parametric rate, then the oracle selector does not converge at a parametric rate, and the cross-validation estimator $\hat{\Phi}_{\beta_n}$ is asymptotically equivalent with the oracle estimator $\hat{\Phi}_{\tilde{\beta}_n}$. Otherwise, the cross-validation estimator $\hat{\Phi}_{\beta_n}$ achieves a near parametric $\frac{\log n}{n}$ rate.

Theorem 1 tells us that the risk difference, based on squared error loss, of the Subsemble from the true $E_0[Y|X]$ can be bounded from above by a function of the risk difference of the oracle procedure. Note that the oracle procedure results in the best possible combination of the subset-specific fits, since the oracle procedure selects β to minimize the true risk difference. In practice, the base learning algorithms are unlikely to converge at a parametric rate, so it follows that Subsemble performs as well as the best possible combination of subset-specific fits. It has also been shown empirically that Subsembles can perform at least as well as, and typically better than simple averaging [72] of the subset-specific fits. Since averaging, or bagging, is an example of combining the subset-specific fits, it follows from Theorem 1

that the Subsemble algorithm is asymptotically superior than bagging the subset-specific fits.

Note that Theorem 1 doesn't specify how many subsets are best, or how Subsemble's combination of many subset-specific fits will perform relative to fitting a single algorithm, Ψ^l, just once on the full training set. The *Supervised Regression Tree (SRT) Subsemble* algorithm [71], discussed in Section 3.3, will provide further insight into how to data-adaptively select the optimal number of subsets.

A practical Subsemble implementation

The Subsemble algorithm offers a practical "divide-and-conquer" approach to supervised ensemble learning with big data. The original training data set can be partitioned into J disjoint subsets, each of which can reside in memory on one node of a cluster. Assuming that the subsets contain roughly the same number of training examples, the computational burden of training with the full dataset of n observations is reduced to training on a dataset of size n/J. This can greatly reduce both the (per-node) memory and total runtime requirements of the training process, while still allowing each base learning algorithm to see all of the original training samples. Subsemble offers an advantage over approaches that use only a single subset of the data, thereby wasting valuable training data.

The base learners are trained independently of each other, so much of the Subsemble algorithm is embarrassingly parallel in nature. The V-fold cross-validation process of generating the level-one predicted values involves training and testing a total of $L \times J \times V$ models. These models can be trained and evaluated simultaneously, in parallel, across $L \times J \times V$ nodes. After generating the cross-validated level-one data, the metalearning algorithm is trained using this data.

The metalearning step is performed after the cross-validation step, as it uses the level-one data produced by the cross-validation step as input. This step involves training a metalearning algorithm on a design matrix of dimension, $n \times (L \times J)$, however, a subset of the level-one design matrix can be used if prohibited by memory constraints. Alternatively, an online algorithm can be used for a metalearner if the level-one data does not fit into memory.

The final ensemble model will consist of the metalearner fit and $L \times J$ base learner fits. These final base learner fits can be can be fit in parallel. Alternatively, the $L \times J$ subset-specific fits from any v^{th} iteration of the cross-validation step can be saved instead of fitting $L \times J$ new models, however, these models will be fit using approximately $\left(\frac{V-1}{V}\right) n/J$, instead of n/J, training observations. Therefore, if resources are available, it is preferable to additionally fit the $L \times J$ models (each trained using n/J observations), separate from the cross-validation step. Given appropriate parallel computing resources (at least $L \times J \times V$ cores), the entire ensemble can be trained in parallel in just two or three steps.

A parallelized implementation of the Subsemble algorithm is available in the **subsemble** R package [55]. Support for both multicore and multi-node clusters is available in the software. This package currently uses the **SuperLearner** R package's [68] machine learning algorithm

wrapper interface to provide access to approximately 30 machine learning algorithms that can be used as base learners. The user can also define their own algorithm wrappers by utilizing the **SuperLearner** wrapper template.

The Supervised Regression Tree Subsemble algorithm, discussed in Section 3.3, provides a way to determine the optimal number of subsets. However, in practice, the number of subsets may be determined by the computational infrastructure available to the user. By increasing the number of subsets, the user can re-size the learning problem into a set of computational tasks that fit within specific resource constraints. This is particularly useful if compute nodes are easy to come by, but the memory on each node is limited. The larger the memory (RAM) on each compute node, the bigger your subsets can be.

subsemble R code example

In this section, we present an R code example that shows the main arguments required to train a Subsemble fit using the **subsemble** R package. The interface for the `subsemble` function should feel familiar to R users who have used other modeling packages in R. The user must specify the base learner library, the metalearning algorithm, and optionally, the number of desired data partitions (subsets).

Algorithm functions (e.g. `"SL.gam"`) from the **SuperLearner** package are used to specify the base learner library and metalearner. Support for other types of base learner and metalearner functions, such as those specified using the **caret** [43] R package, is on the development road-map for the **subsemble** package. In contrast to the **SuperLearner** package, the **subsemble** package allows the user to choose any algorithm for the metalearner and does not require the metalearner to be a linear model that minimizes the cross-validated risk of a given loss function. Most machine learning algorithms work by minimizing some loss function (or surrogate loss function), so this functionality is not that different in practice, however it does allow the user to specify a non-parametric metalearning algorithm.

The code example uses a three-learner library consisting of a Generalized Additive Model (`"SL.gam"`), a Support Vector Machine (`"SL.svm.1"`) and Multivariate Adaptive Regression Splines (`"SL.earth"`). The metalearner is specified as generalized linear model with Lasso regularization (`"SL.glmnet"`). A number of algorithm wrapper functions are included in the **SuperLearner** package. However, the user may want to define non-default model parameters for a base learner. To illustrate this, one custom base learner (the SVM) is included in the example ensemble. To create a custom wrapper function, the user may pass along any non-default arguments to the default function wrapper for that algorithm.

```r
library("subsemble") # Also loads the SuperLearner package
library("cvAUC") # For model evaluation

# Create one non-default base learner function to use in the library
SL.svm.1 <- function(..., type.class = "C-classification") {
   SL.svm(..., type.class = type.class)
}

# Set up the Subsemble:
# Use two default learners from SuperLearner package, one custom wrapper
learner <- c("SL.gam", "SL.svm.1", "SL.earth") # Base learners
metalearner <- "SL.glmnet" # Metalearner
subsets <- 4  # Number of subsets
```

By default, a "cross-product" type Subsemble will be created using the `subsemble` function, but this can be modified using the `learnControl` argument. Since four subsets and three base learners are specified by the user, this will result in an ensemble of $4 \times 3 = 12$ subset-specific fits.

```r
# Train and evaluate the Subsemble fit:
fit <- subsemble(x = x, y = y, newx = newx,
                 family = binomial(),
                 learner = learner,
                 metalearner = metalearner,
                 subsets = subsets)

# Evaluate model performance on a test set
auc <- cvAUC::AUC(predictions = fit$pred, labels = newy)
```

Performance benchmarks

This section presents benchmarks of the model performance and runtime of the Subsemble algorithm as implemented in the **subsemble** R package, version 0.0.9. This example uses the same three-algorithm base learner library and metalearner that was specified in the R code example above.

Binary outcome training data sets with 10 numerical features were simulated using the `twoClassSim` function in the **caret** R package [43]. Training sets of increasing size were generated, and a independent test set of 100,000 observations was used to estimate model performance, as measured by Area Under the ROC Curve (AUC). Confidence intervals for the test set AUC estimates were generated using the **cvAUC** R package [54].

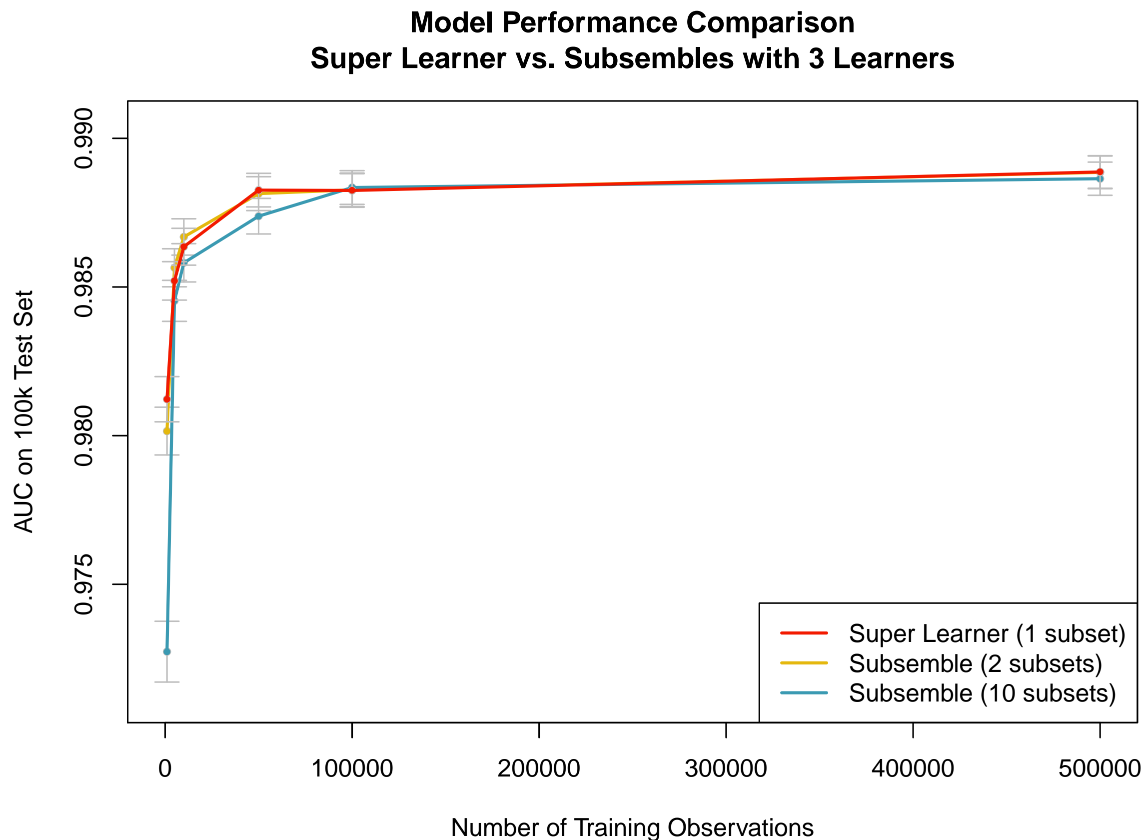

Figure 3.3: Model performance for Subsemble and Super Learner for training sets of increasing size. 95%-confidence intervals from the **cvAUC** package are shown.

Model performance

As shown in Figure 3.3, the performance of the 2-subset and 10-subset Subsemble is comparable to the Super Learner algorithm. In this example, after about 100,000 training observations, the 95% confidence intervals for AUC for the three models overlap. For smaller training sets, Subsemble model performance can dip below Super Learner if too many subsets are used. Once the subsets become too small, they may fail to sufficiently estimate the distribution of the full training set.

Computational performance

These benchmarks were performed on a single "r3.8xlarge" instance on Amazon's Elastic Compute Cloud (EC2) with 32 virtual CPUs and 244 GB of RAM using R version 3.1.1. Both single-threaded and multicore implementations of Subsemble and Super Learner were

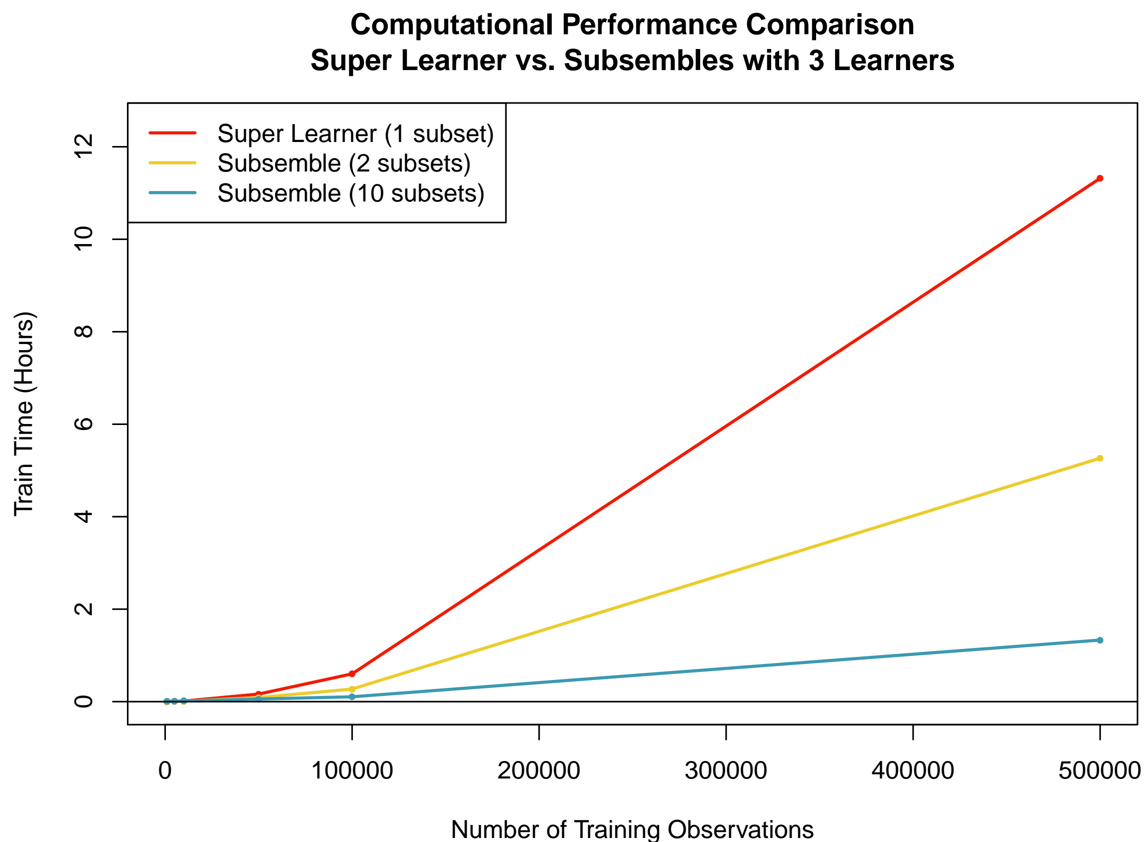

Figure 3.4: Training time for Subsemble and Super Learner for training sets of increasing size.

measured. For this example, there are a total of $3 \times J$ subset-specific fits since three base learners were used in the Subsemble. In the case of $J = 10$, there are 30 subset-specific models that need to be trained, which can be fit all at once on a 32-core machine using the multicore option in the software. Subsembles with a larger number of subsets and base learners can also be trained in parallel on multi-node clusters. The cross-validation and base learning tasks are computationally independent tasks, so as long as a big enough cluster is used, the subset-specific models can all be trained at the same time. For maximum efficiency, a cluster with at least $L \times J$ cores should be used. The training time for multicore Subsemble and multicore Super Learner is shown in Figure 1.4 for training sets of increasing size.

3.3 Subsembles with Subset Supervision

Different methods for creating Subsemble's subsets result in different Subsembles, since the performance of the final Subsemble estimator can vary depending on the specific partitioning of the data into subsets. The naive approach to creating subsets is to randomly partition the n training observations into J subsets of size n/J, however, this still requires the user to choose a value for J, or to try several values for J.

In this section, we introduce the "Supervised Subsembles" method [71] for partitioning a data set into the subsets used in Subsemble. Supervised Subsembles create subsets via supervised partitioning of the *covariate space*, and use a form of *histogram regression* [64] as the metalearner. We also discuss a practical supervised Subsemble method called "Supervised Regression Tree Subsemble," or "SRT Subsemble", which employs *regression trees* to both partition the observations into the subsets used in Subsemble, and select the number of subsets to use. In each subsection, we highlight the computational independence properties of these methods which are advantageous for applications involving big data.

Supervised Subsembles

The subsets used in supervised Subsembles are obtained by a supervised partitioning of the covariate space, $\mathcal{X} = \bigcup_{j=1}^{J} S_j$, to create J disjoint subsets $S_1, \ldots, S_J$, such that any given vector of covariates belongs to exactly one of the J subsets. There are many unsupervised methods that can be used to partition the covariate space, however, supervised partitioning methods will create a partitioning that is predictive of the outcome. Given the number of clusters, J, this technique will partition the covariate space, however it does not directly solve the problem of choosing an optimal value for J.

Compared to randomly selecting the subsets, constructing the subsets to be internally similar, yet distinct from each other, results in locally smoothed subset-specific fits. The added diversity among the subset-specific fits can improve the overall performance of the ensemble. In this case, each subset-specific fit is optimal for a distinct partition of the covariate space. Since a subset-specific fit, $\hat{\Psi}_j^l$, is tailored for the subspace S_j, it will not necessarily be the case that $\hat{\Psi}_j^l$ will be a good predictor for observations belonging to the subspace, $S_{j'}$, where $j \neq j'$.

To account for this, supervised Subsembles use a modified version of histogram regression as the metalearner in order to combine the subset-specific fits. The usual form of histogram regression, applied to the J subsets $\{S_j\}_{j=1}^{J}$, produces a local average of the outcome Y within each subset. In contrast, the modified version of histogram regression outputs the associated $\hat{\Psi}_j^l$ for each subset S_j. In addition, this version of histogram regression includes a coefficient and intercept within each subset. The histogram regression metalearning fit, $\hat{\Phi}$, for combining the subset-specific fits is defined as follows:

$$\hat{\Phi}(\hat{\Psi}^1, \ldots, \hat{\Psi}^L)(x) = \sum_{j=1}^{J} \left[I(x \in S_j) \left(\beta_j^0 + \sum_{l=1}^{L} \beta_j^l \hat{\Psi}_j^l(x) \right) \right] \tag{3.1}$$

These supervised Subsembles have the benefit of preserving the subset computational independence. That is, if subsets are known a priori, by keeping the subset assignments fixed, computations on the subsets of the Supervised Subsemble described in this section remain computationally independent across the entire procedure.

To see this, let β_n to be the cross-validation selector of β and use the squared error loss as an example. As shown in [71],

$$\begin{aligned} \beta_n &= \arg\min_{\beta} \sum_{i=1}^{n} \left\{ Y_i - \hat{\Phi}_\beta(\tilde{X}_i) \right\}^2 \\ &= \left\{ \arg\min_{\beta_j} \sum_{i:i\in S_j} \left(Y_i - \left[\beta_j^0 + \sum_{l=1}^{L} \beta_j^l \hat{\Psi}_{j,v(i)}^l (X_i) \right] \right)^2 \right\}_{j=1}^{J} \end{aligned}$$

Thus, each term β_j can be estimated by minimizing cross-validated risk using only the data in subset S_j. Unlike when using randomly constructed subsets, supervised subsets remove the requirement of recombining data in order to produce the final prediction function for the Subsemble algorithm.

The Supervised Regression Tree (SRT) Subsemble algorithm

Supervised Regression Tree (SRT) Subsemble is a practical supervised Subsemble algorithm which uses regression trees to determine both of the number of subsets, J, and the partitioning of the covariate space.

With large-scale data, reorganizing a data set into different pieces can be a challenge from an infrastructure perspective. In a cluster setting, this requirement could require copying a large amount of data between nodes. As a result, it is preferable to avoid approaches that split and then recombine data across the existing subsets. After the initial partitioning of the data across nodes, it is more computationally efficient to create additional splits that divide an already existing partition.

Constructing and selecting the number of subsets

The Classification and Regression Tree (CART) algorithm [18], recursively partitions the covariate space by creating binary splits of one covariate at a time. Concretely, using covariate vector $X_i = (X_i^1, \ldots, X_i^K)$, the first iteration of CART selects a covariate X^k, and then creates the best partition of the data based on that covariate. As a metric to measure and select the best covariate split for a continuous outcome, some *splitting criterion* is used. For classification trees, Gini impurity or information gain is often used as a splitting criterion. For regression trees, typically, the split that minimizes the overall sum of squares is selected. For additional details, we refer the reader to [18].

The first iteration of CART creates the first partition of the data based on two regions $S_1^1 = I(X^k \leq c_1)$ and $S_1^2 = I(X^k > c_1)$. Subsequent splits are obtained greedily, by

repeating this procedure on each new partition. For example, the second iteration of CART selects a covariate, $X^{k'}$, and partitions S_1^1 into $S_2^1 = I(X^k \leq c_1,\ X^{k'} \leq c_2)$ and $S_2^2 = I(X^k \leq c_1,\ X^{k'} > c_2)$. For a given partitioning, the standard prediction function from CART outputs the local average of the outcome Y within each subset.

To partition the covariate space and select the number of subsets, SRT Subsemble applies CART as follows:

1. Run the CART algorithm on the data set, resulting in a sequence of nested partitionings of the covariate space. That is, the CART algorithm outputs a sequence of sub-trees: a first tree with a single root node, a second tree with two nodes, a third tree with three nodes, and so on, ending with the full tree with M nodes. The m nodes of each mth sub-tree are treated as a candidate partitioning into m subsets.

2. Next, explore the sequence of M possible partitions (sequence of sub-trees) produced by CART, beginning at the root. For each candidate number of subsets $1, \ldots, M$, a supervised Subsemble model is fit. Moreover, with m subsets, create L subset-specific fits, $\hat{\Psi}_j^l$, for each subset $j = 1, \ldots, m$, and create the overall prediction function according to Equation 3.1. Note that CART is used to create the subsets S_j that appear in Equation 3.1.

3. Finally, choose the number of subsets that produces the supervised Subsemble with minimum cross-validated risk.

SRT Subsemble in practice

SRT Subsemble has desirable computational independence (by subset) and provides a mechanism to learn the optimal number and constituency of the subsets to use in the Subsemble algorithm. Note that as a consequence of this independence, fitting a sequence of Subsembles in a series of sub-trees, each subsequent Subsemble only requires computation for the two new nodes at each step. That is, given the Subsemble fit with, say, m subsets, computing the next Subsemble with $m + 1$ subsets only requires computation for the two new nodes formed in the $(m + 1)^{st}$ split of the tree. This is a result of the fact that the nodes are computationally independent in the SRT Subsemble framework, and also that at each split of the tree, all nodes remain the same, except for the single node in the mth tree (that is split into two new nodes in the $(m + 1)^{st}$ tree).

There are several choices available for practitioners applying the SRT Subsemble process in practice. These user-selected options allow SRT Subsemble to be quite flexible, with options to suit the application at hand. There is no single, best approach; instead the options will be determined based on the application, computational constraints and desired properties of the estimator.

The first consideration relates to building and exploring the tree. One possibility is to simply build a very large tree (resulting in a full tree with M nodes), build a Subsemble for

each sub-tree, $1, \ldots, M$, and through this process simply locate the Subsemble with the lowest cross-validated risk among the sequence of sub-trees outputted by CART. Alternatively, a greedy process can be used. Instead of calculating cross-validated risk for all sub-trees of a very large tree, the cross-validated risk can be computed while the tree is being built. That is, after each additional split to the tree, build the associated Subsemble, calculate the associated cross-validated risk, and refrain from making additional splits once some stopping criteria is achieved. As a simple example, the stopping criteria could be an increase of the cross-validated risk.

Second, the user must decide where in the tree to start building Subsembles. The most obvious approach is to start with building a Subsemble at the root node of the tree, meaning the Subsemble is built with only one subset containing all observations. A Subsemble with one subset is equivalent to the Super Learner algorithm [84]. For small to moderate-sized data sets, where computational considerations are of less of a concern, this is a good choice. However, for large-scale data sets, it may be preferable to first split the data into partitions of some desired size, and then begin the Subsemble process on the subsets. This approach would allow the user to take advantage of multiple independent computational resources, since each partition of data could be transferred to a dedicated computational resource (all subsequent computations remain independent from other partitions).

3.4 Conclusion

In this chapter, we presented the Subsemble algorithm, a flexible subset ensemble prediction method. Subsemble partitions a training set into subsets of observations, fits one or more base learning algorithm on each subset, and combines the subset-specific fits through a second-level metalearning algorithm using a unique form of k-fold cross-validation. We provided a theoretical performance guarantee showing that Subsemble performs as well as the best possible combination of the subset-specific fits. Further, we described the practical implementation of the Subsemble algorithm which is available in the **subsemble** R package, and presented performance benchmarks that demonstrate desirable predictive performance with significant runtime improvements as compared with full-data ensembles such as generalized stacking. We described using a supervised partitioning of the covariate space to create Subsemble's subsets. We discussed the computational advantages of this supervised subset creation approach, and described the practical SRT Subsemble algorithm which will construct the covariate partitioning and learn the optimal number of subsets.

Chapter 4

AUC-Maximizing Ensembles through Metalearning

4.1 Introduction

In the field of biostatistics, binary classification or ranking problems arise in many applications, for example, in diagnostic testing. There are also many problems for which the outcome is rare, or imbalanced, meaning the number of positive cases far outweighs the number of negative cases, or vice versa. In this type of prediction problem, the Area Under the ROC Curve (AUC) is frequently used to measure model performance, due to its robustness against prior class probabilities. When AUC maximization is the goal, a classifier that aims to specifically maximize AUC can have significant advantages in these types of problems.

However, most commonly used classification algorithms work by optimizing an objective function that is unrelated to AUC – for example, accuracy or error rate. If the training dataset has an imbalanced outcome, this can lead to classifiers where the majority class has close to 100% accuracy, while the minority class has an accuracy of closer to 0-10% [87]. In practice, the accuracy of the minority class is often more important than the accuracy of the majority class. Therefore, unless some type of intervention (e.g., under-sampling, over-sampling) is used to help alleviate this issue, or AUC maximization is inherent to the algorithm, class imbalance may negatively impact the performance of a binary classification algorithm.

In this chapter, we introduce an ensemble approach to AUC maximization for binary classification problems. Ensemble methods are algorithms that combine the output from a group of base learning algorithms, with the goal of creating an estimator that has predictive performance over the individual algorithms that make up the ensemble. The Super Learner algorithm [84] is an ensemble algorithm which generalizes stacking [51, 17, 86], by allowing for more general loss functions and hence a broader range of estimator combinations. The Super Learner is built on the theory of cross-validation and has been proven to represent an

asymptotically optimal system for learning [84].

Super Learner, described in further detail in Section 4.2, estimates the optimal combination of the base learning algorithms in the ensemble, with respect to a user-defined loss function. The "metalearning step" in the Super Learner algorithm is the process of data-adaptively determining the optimal combination a specific group of base learner fits via a second-level metalearning algorithm. With respect to model performance, this leads to estimators that have superior (or at worst, equal) performance to the top base algorithm in the ensemble. Even if none of the base learners specifically maximize AUC, it is possible to inject AUC-maximization directly into imbalanced data problems via the metalearning step of the Super Learner algorithm.

Any type of parametric or nonparametric algorithm (which is associated with a bounded loss function) can be used in the metalearning step, although in practice, it is common to estimate the optimal linear combination of the base learners. Since the Super Learner framework allows for any loss function (and corresponding risk function), to be used in the metalearning step, it is possible to create ensemble learners that specifically aim minimize a user-defined loss function of interest.

The loss function associated with AUC, also called "rank loss," measures the bipartite ranking error, or disagreement between pairs of examples. The associated risk is calculated as $1.0 - \text{AUC}$. In the Super Learner algorithm, minimization of the rank loss or, equivalently, maximization of the AUC, can be approached directly by using an AUC-maximimining metalearning algorithm. In Section 4.3, we discuss how AUC maximization can be formulated as a nonlinear optimization problem. We have implemented the AUC-maximizing metalearning algorithm as an update to the **SuperLearner** R package and demonstrate its usage with a code example.

In Section 4.4, we evaluate the effectiveness of a large number of nonlinear optimization algorithms to maximize the cross-validated (CV) AUC of a Super Learner fit. We compare the cross-validated AUC of the AUC-optimized ensemble fits to the cross-validated AUC of the ensembles that do not attempt to optimize AUC. Super Learner fits using various metalearning algorithms are benchmarked using training sets with varying levels of class imbalance. The results provide evidence that AUC-maximizing metalearners typically outperform non-AUC-maximizing metalearning methods, with respect to ensemble AUC. The results also demonstrate that as the level of class imbalance increases in the training set, the Super Learner ensemble out-performs the top base algorithm by a larger degree.

4.2 Ensemble metalearning

The Super Learner prediction is the optimal combination of the predicted values from the base learners, which is the motivation behind the name, "Super Learner." The optimal way of combining the base learning algorithms is precisely what is estimated in the metalearning step of the Super Learner algorithm. The output from the base learners, also called "level-one" data in the stacking literature [86], serves as input to the metalearner algorithm. Super

Learner theory requires cross-validation to generate the level-one dataset, and in practice, k-fold cross-validation is often used.

The following describes how to construct the level-one dataset. Assume that the training set is comprised of n independent and identically distributed observations, $\{O_1, ..., O_n\}$, where $O_i = (X_i, Y_i)$ and $X_i \in \mathbb{R}^p$ is a vector of covariate or feature values and $Y_i \in \mathbb{R}$ is the outcome. Consider an ensemble comprised of a set of L base learning algorithms, $\{\psi^1, ..., \psi^L\}$, each of which is indexed by an algorithm class, as well as a specific set of model parameters. Then, the process of constructing the level-one dataset will involve generating an $n \times L$ matrix, $\boldsymbol{Z}$, of k-fold cross-validated predicted values as follows:

1. The original training set, $\boldsymbol{X}$, is divided randomly into $k = V$ roughly-equal pieces (validation folds), $\boldsymbol{X}_{(1)}, ..., \boldsymbol{X}_{(V)}$.
2. For each base learner in the ensemble, ψ^l, V-fold cross-validation is used to generate n cross-validated predicted values associated with the lth learner. These n-dimensional vectors of cross-validated predicted values become the L columns of $\boldsymbol{Z}$.

The level-one dataset, $\boldsymbol{Z}$, along with the original outcome vector, $(Y_1, ..., Y_n) \in \mathbb{R}^n$, is then used to train the metalearning algorithm, Φ.

Base learner library

Super Learner theory does not require any specific level of diversity among the set of base learners, however, a diverse set of base learners (e.g., Linear Model, Support Vector Machine, Random Forest, Neural Net) is encouraged. The more diverse the library is, the more likely it is that the ensemble will be able to approximate the true prediction function. The "base learner library" may also include copies of the same algorithm, indexed by different sets of model parameters. For example, the user can specify multiple Random Forests [16], each with a different splitting criterion, tree depth or "mtry" value.

The base learner prediction functions, $\{\hat{\psi}^1, ..., \hat{\psi}^L\}$, are trained by fitting each of L base learning algorithms, $\{\psi^1, ..., \psi^L\}$, on the training set. The base learners can be any parametric or nonparametric supervised machine learning algorithm. Once the level-one dataset is generated by cross-validating the base learners, the optimal combination of these fits is estimated by applying the metalearning algorithm, Φ, to these data.

Metalearning algorithms

In the context of Super Learning, the metalearning algorithm is a method that minimizes the cross-validated risk associated with some loss function of interest. For example, if the goal is to minimize mean squared prediction error, the ordinary least squares (OLS) algorithm can be used to solve for $\boldsymbol{\alpha} = (\alpha_1, ..., \alpha_L)$, the weight vector that minimizes the following:

$$\sum_{i=1}^{n}(Y_i - \sum_{l=1}^{L} \alpha_l z_{il})^2$$

In the equation above, z_{il} represents the (i, l) element of the $n \times L$ level-one dataset, $\boldsymbol{Z}$. If desired, a non-negativity restriction i.e., $\alpha_l \geq 0$, can be imposed on the weights. There is evidence that this type of regularization increases the predictive accuracy of the ensemble [17]. In this case, the Non-Negative Least Squares (NNLS) algorithm [50] can be used as a metalearner. Both OLS and NNLS are suitable metalearner choices to use when the goal is to minimize squared prediction error. In the **SuperLearner** R package [68], there are five pre-existing metalearning methods available by default, and these are listed in Table 4.1.

Method	Description	R Package
NNLS	Non-negative Least Squares	**nnls**
NNLS2	Non-negative Least Squares	**quadprog**
CC_LS	Non-negative Least Squares	**nloptr**
NNloglik	Negative Log-likelihood (Binomial)	Base R
CC_nloglik	Negative Log-likelihood (Binomial)	**nloptr**

Table 4.1: Default metalearning methods in **SuperLearner** R package version 2.0-17.

However, in many prediction problems, the goal is to optimize some *objective function* other than the objective function associated with ordinary or non-negative least squares. For example, in a ranking problem, if the goal is to maximize the AUC of the model, then an AUC-maximizing algorithm can be used in the metalearning step. Unlike the *accuracy* metric for classification problems, AUC is a performance measure that is unaffected by the prior class distributions [14]. Accuracy-based performance measures implicitly assume that the class distribution of the dataset is approximately balanced and the misclassification costs are equal [37]. However, for many real world problems, this is not the case. Therefore, AUC may be a suitable performance metric to use when the training set has an imbalanced, or rare, binary outcome. Multi-class versions of AUC exist [36, 69], however, we will discuss AUC in the context of binary classification.

Although we use AUC-maximization as the primary, motivating example, the technique of targeting a user-defined loss function in the metalearning step can be applied to any bounded loss function, $L(\psi)$. It is worth noting that the loss function, $L(\psi)$, not just risk, $E_0 L(\psi)$, must be bounded. The AUC-maximizing metalearning algorithm that we have contributed to the **SuperLearner** package can be reconfigured so that the Super Learner minimizes any loss function that is possible to implement in code. For binary classification, other performance measures of interest may be F_1-*score* (or F_β) [70], *Partial AUC* [62, 44], or *H-measure* [37]. A Super Learner ensemble that optimizes any of these metrics can be constructed following the same procedure that we present for AUC-maximization.

4.3 AUC maximization

Given a set of base learning algorithms, the linear combination of the base learners that maximizes the cross-validated AUC of the Super Learner ensemble can be found using nonlinear optimization.

Nonlinear optimization

A nonlinear optimization problem is an optimization problem that seeks to minimize (or maximize) some objective function, $f(\alpha)$, where $f : \mathbb{R}^d \rightarrow \mathbb{R}$, $\alpha \in \mathbb{R}^d$, and the solution space is subject various constraints. Box constraints enforce an upper or lower bound on each of the d optimization parameters. In other words, upper and/or lower bound vectors, $lb = (lb_1, ..., lb_d)$ and $ub = (ub_1, ..., ub_d)$, can be defined, such that $lb_j \leq \alpha_j \leq ub_j$ for $j = \{1, ..., d\}$. In partially constrained, or unconstrained, optimization problems, one or both of these bounds may be $\pm\infty$. There may also exist m separate nonlinear inequality constraints, $f_{c_j}(\alpha) \leq 0$ for $j = 1, ..., m$, for constraint functions, $f_{c_j}(\alpha)$. Lastly, a handful of algorithms are capable of supporting one or more nonlinear equality constraints, which take the form, $h(\alpha) = 0$.

Nonlinear optimization in R

The optimization methods available in base R (via the `optim` function), as well as algorithms from the **nloptr** package [89], can be used to approximate the linear combination of the base learners that maximizes the AUC of the ensemble.

The `optim` function supports general-purpose optimization based on Nelder-Mead [63], quasi-Newton and conjugate-gradient algorithms. There is also a simulated annealing method [9], however, since this method can be quite slow, we did not consider its use as metalearning method to be practical. With the `optim` function, there is only one method, L-BFGS-B [20, 92], that allows for box-constraints on the weights. The `optim`-based methods do not allow for equality constraints such as $\sum_j \alpha_j = 1$, however, normalization of the weights can be performed as an additional step (after the optimal weights are determined by the metalearner) to provide added interpretability to the user. Since AUC is a ranking-based measure, normalization of the weights will not affect the AUC value.

The **nloptr** package is an R interface to the **NLopt** [45] software project from MIT. **NLopt** is an open-source library for nonlinear optimization, which provides a common interface for a number of different optimization routines. We evaluated 16 different global and local optimization algorithms from **NLopt** for the purpose of metalearning. The complete list of **NLopt**-based algorithms is documented in Table 4.2. A lower bound of 0 is imposed for the weights using the **NLopt** methods to avoid learning negative weights for the base learner contribution. We also evaluated the effect of adding an upper bound of 1 in comparison to leaving the upper bound undefined, i.e., ∞.

Method	Description	min$\{\alpha_j\}$	max$\{\alpha_j\}$
NNLS	Non-negative Least Squares (`nnls`)	0	Inf
NNLS2	Non-negative Least Squares (`quadprog`)	0	Inf
CC_LS	Non-negative Least Squares (`nloptr`)	0	1
NNloglik	Negative Log-likelihood (Binomial) (`optim`)	0	Inf
CC_nloglik	Negative Log-likelihood (Binomial) (`nloptr`)	0	1
AUC_nlopt.1	DIRECT	0	1
AUC_nlopt.2	DIRECT	0	Inf
AUC_nlopt.3	DIRECT-L	0	1
AUC_nlopt.4	DIRECT-L	0	Inf
AUC_nlopt.5	DIRECT-L RAND	0	1
AUC_nlopt.6	DIRECT-L RAND	0	Inf
AUC_nlopt.7	DIRECT NOSCAL	0	1
AUC_nlopt.8	DIRECT NOSCAL	0	Inf
AUC_nlopt.9	DIRECT-L NOSCAL	0	1
AUC_nlopt.10	DIRECT-L NOSCAL	0	Inf
AUC_nlopt.11	DIRECT-L RAND NOSCAL	0	1
AUC_nlopt.12	DIRECT-L RAND NOSCAL	0	Inf
AUC_nlopt.13	ORIG DIRECT	0	1
AUC_nlopt.14	ORIG DIRECT	0	Inf
AUC_nlopt.15	ORIG DIRECT-L	0	1
AUC_nlopt.16	ORIG DIRECT-L	0	Inf
AUC_nlopt.17	Controlled Random Search with Local Mutation	0	1
AUC_nlopt.18	Controlled Random Search with Local Mutation	0	Inf
AUC_nlopt.19	Improved Stochastic Ranking Evolution Strategy	0	1
AUC_nlopt.20	Improved Stochastic Ranking Evolution Strategy	0	Inf
AUC_nlopt.21	Principal Axis (PRAXIS)	0	1
AUC_nlopt.22	Principal Axis (PRAXIS)	0	Inf
AUC_nlopt.23	Constrained Opt. by Linear Approximations	0	1
AUC_nlopt.24	Constrained Opt. by Linear Approximations	0	Inf
AUC_nlopt.25	Bounded NEWUOA	0	1
AUC_nlopt.26	Bounded NEWUOA	0	Inf
AUC_nlopt.27	Nelder-Mead	0	1
AUC_nlopt.28	Nelder-Mead	0	Inf
AUC_nlopt.29	Sbplex	0	1
AUC_nlopt.30	Sbplex	0	Inf
AUC_nlopt.31	BOBYQA	0	1
AUC_nlopt.32	BOBYQA	0	Inf
AUC_optim.1	L-BFGS-B	0	1
AUC_optim.2	L-BFGS-B	0	Inf
AUC_optim.3	Nelder-Mead	-Inf	Inf
AUC_optim.4	BFGS	-Inf	Inf
AUC_optim.5	Conjugate Gradient (CG)	-Inf	Inf

Table 4.2: Metalearning methods evaluated.

AUC-maximizing ensembles

The **SuperLearner** R package was used to evaluate various metalearning methods. The five pre-existing methods, described in Table 4.1, were compared against a new set of AUC-maximizing metalearning methods. As part of this exercise, we implemented a new metalearning function, `method.AUC`, for the **SuperLearner** package. In the `method.AUC` function, the weights are initialized as $\alpha_{init} = (\frac{1}{L}, ..., \frac{1}{L})$, where L is the number of base learners. The function will execute an optimization method that maximizes AUC, and will return the optimal weights found by the algorithm.

The `method.AUC` function currently supports four `optim`-based and sixteen **nloptr**-based optimization algorithms by default. Many of these algorithms support box-constraints, so additional configurations of the existing methods can also be specified using our software. An example of how to specify a custom metalearning function by wrapping the `method.AUC` function is shown in Figure 4.1. In the example, we specify the metalearner to use the unbounded, `optim`-based, BFGS method [19, 28, 33, 74], with post-optimization normalization of the weights.

```
# Create a customized AUC-maximizing metalearner
#  using the new method.AUC() function:

library("SuperLearner")

method.AUC_optim.4 <- function(optim_method = "BFGS", ...) {
  method.AUC(nlopt_method = NULL,
             optim_method = optim_method,
             bounds = c(-Inf, Inf),
             normalize = TRUE)
}
```

Figure 4.1: Example of how to use the `method.AUC` function to create customized AUC-maximizing metalearning functions.

A complete list of the default **SuperLearner** metalearning methods plus the new `optim` and **nloptr**-based AUC-maximizing metalearning methods are listed in Table 4.2. A total of 37 new AUC-maximizing metalearning functions were evaluated. The `AUC_nloptr.X` (where `X` is an integer between 1 and 20) functions implement global optimization routines and the remainder (21-32) are local, derivative-free, methods. The `AUC_optim.X` (where `X` is an integer between 1 and 5) functions implement the variations of the `optim`-based methods.

To reduce the computational burden of evaluating a large number of metalearners, we implemented two functions for the **SuperLearner** package which simplify the processes of re-combining the base learner fits using a new metalearning algorithm. These functions, `recombineSL` and `recombineCVSL`, take as input an existing `"SuperLearner"` or

```
# Example base learner library:
SL.library <- list(c("SL.glmnet"),
                   c("SL.gam"),
                   c("SL.randomForest"),
                   c("SL.polymars"),
                   c("SL.knn"))

# Then the "method.AUC_optim.4" function can be used as follows:
fit <- SuperLearner(Y = Y, X = X, newX = newX,
                    family = binomial(),
                    SL.library = SL.library,
                    method = "method.AUC_optim.4")
```

Figure 4.2: Example of how to use the custom `method.AUC_optim.4` metalearning function with the `SuperLearner` function.

`"CV.SuperLearner"` fit, and a new metalearning function. They re-use the existing level-one dataset and base learner fits stored in the model object, re-run the metalearning step and return the updated fit. A simple example of how to use the `recombineSL` function is shown in Figure 4.3.

```
# Assume that ‘fit’ is an object of class, "SuperLearner"
newfit <- recombineSL(object = fit, Y = Y, method = "method.NNloglik")
```

Figure 4.3: Example of how to update an existing `"SuperLearner"` fit by re-training the metalearner with a new method.

4.4 Benchmark results

This section contains the results from the benchmarks of various metalearning methods in a binary classification problem. The purpose of these benchmarks is twofold. We evaluate whether AUC-maximizing metalearners, compared to other methods such as NNLS, actually maximize the cross-validated AUC of the ensemble. We also investigate the training set characteristics that lead to the greatest performance gain, as measured by cross-validated (CV) AUC. In particular, we measure the effect that class imbalance, as well as training size, has on the performance of Super Learner, under various metalearning methods.

After computing the performance of Super Learner using the 42 metalearners under consideration, we identify the metalearning algorithm that yields the best CV AUC. The remaining metalearners are compared by calculating the offset, in terms of cross-validated

AUC, between the Super Learner utilizing the top metalearing method and the other methods. For example, if the top method produced a model with a CV AUC of 0.75, the offset between the top method and a model with 0.73 CV AUC would be 0.02. The AUC offsets for each dataset-metalearner combination are displayed in a grid-format using heatmaps.

For each training set, we contrast the performance, as measured by CV AUC, of the Super Learner ensemble with the best base model, as determined by a cross-validated grid search over the set of algorithms in the library. In the grid search method, the estimator with the best cross-validated performance (as measured by a given loss criterion), is selected as the winning algorithm, and the other algorithms are discarded. The term “grid search” is often used to describe a search through a manually specified subset of the hyperparameter space of a particular learning algorithm, however the term can also be used to describe the search among a set of different learning algorithms (and parameters) under consideration. In the Super Learning, or stacking, context, the process of generating the level-one dataset using cross-validation is equivalent to a cross-validated grid search of the base learners. Moreover, in the Super Learner literature, the grid search technique is referred to as the Discrete Super Learner algorithm [83].

It is quite common for machine learning practitioners to use a grid search to evaluate the performance of a set of unique candidate learning algorithms, or unique sets of model parameters within a single algorithm class, as a means to select the best model from those under consideration. Since the metalearning step requires a only small amount of computation as compared to the computation involved in generating level-one data, executing a grid search and training the Super Learner algorithm are computationally similar tasks. Although grid search and Super Learning require a similar amount of work, the Super Learner ensemble, by optimally combining the set of candidate estimators, can provide a boost in performance over the top base model. For context, we also provide the cross-validated AUC for each of the base models in the ensemble.

In the benchmarks, we use a single, diverse base learner library which contains the following five algorithms: Lasso Regression, Generalized Additive Models, Random Forest (with 1,000 trees), Polynomial Spline Regression and K-Nearest Neighbor ($k = 10$). The R packages that were used for each these methods are listed in Table 4.3.

	Algorithm	R Package	Function
ψ^1	Lasso Regression	**glmnet**	`glmnet`
ψ^2	Generalized Additive Model	**gam**	`gam`
ψ^3	Random Forest (1,000 trees)	**randomForest**	`randomForest`
ψ^4	Polynomial Spline Regression	**polyspline**	`polymars`
ψ^5	K-Nearest Neighbor ($k = 10$)	**class**	`knn`

Table 4.3: Example base learner library representing a small, yet diverse, collection of algorithm classes. Default model parameters were used.

HIGGS dataset

The benchmarks use training sets derived from the HIGGS dataset [8], a publicly available training set that has been produced using Monte Carlo simulations. There are 11 million training examples, 28 numeric features and a binary outcome variable. The associated binary classification task is to distinguish between a background process ($Y = 0$) and a process where new theoretical Higgs bosons are produced ($Y = 1$).

Although this simulated dataset is meant to represent a data generating distribution with a rare outcome (evidence of the Higgs particle), the class distribution of the outcome variable in the simulated data is approximately balanced, with $P(Y = 1) \approx 0.53$. Two groups of subsets of the HIGGS dataset where created, one with $n = 10,000$ observations and the other with $n = 100,000$ observations. Within each group, ten datasets of fixed size, n, but varying levels of class imbalance, were constructed from the original HIGGS dataset. The following levels of outcome imbalance, $P(Y = 1)$, were evaluated: 1-5%, 10%, 20%, 30%, 40% and 50%.

The **SuperLearner** [68] and **cvAUC** [54] R packages were used to train and cross-validate the AUC of the Super Learner fits. For the $n = 10,000$ sized training sets, 10-fold cross-validation was used to estimate the cross-validated AUC values. Within the Super Learner algorithm, 10-fold cross-validation was used to generate the level-one data, where the folds were stratified by the outcome variable. For the $n = 100,000$ training sets, 2-fold cross-validation was used to estimate cross-validated AUC, and 2-fold, stratified, cross-validation was also used to generate the level-one data.

Negative log-likelihood vs. AUC optimization

The benchmark results in Table 4.4 and Table 4.5 suggest that AUC-maximizing metalearners and the negative log-likelihood method perform best among all reviewed metalearning algorithms, as measured by cross-validated AUC. Although the AUC-based metalearners usually perform better than the non-AUC methods for the datasets that were evaluated, there are some examples where using the loss function associated with the negative log-likelihood (of the binomial distribution) yields the top Super Learner.

As expected, there is not one single metalearning method that performs best across all datasets, so we recommend evaluating various metalearners. Since the metalearning step is rather fast in comparison to any of the L base learning steps, this is a reasonable approach to creating the highest performing Super Learner using the available training dataset.

Effect of class imbalance

By creating a sequence of training sets of increasing class imbalance, we evaluate the ability of an AUC-maximizing Super Learner to outperform the non-AUC-maximizing metalearning methods. We also measure the gain, in terms of cross-validated AUC of the ensemble, that the Super Learner provides over the top base model, or grid search winner.

No.	$P(Y=1)$	Metalearner	SL AUC Gain	SL AUC	GS AUC	GS Model
1	0.01	CC_nloglik	0.003	0.721	0.718	gam
2	0.02	AUC_optim.4	0.016	0.739	0.723	gam
3	0.03	NNloglik	0.023	0.764	0.741	gam
4	0.04	AUC_optim.4	0.038	0.770	0.732	gam
5	0.05	AUC_optim.4	0.032	0.759	0.727	randomForest
6	0.10	NNloglik	0.018	0.768	0.750	polymars
7	0.20	AUC_nlopt.11	0.014	0.778	0.764	randomForest
8	0.30	AUC_optim.4	0.007	0.783	0.775	randomForest
9	0.40	AUC_nlopt.23	0.005	0.788	0.783	randomForest
10	0.50	NNloglik	0.003	0.787	0.783	randomForest

Table 4.4: Top metalearner performance for HIGGS datasets, as measured by cross-validated AUC ($n = 10,000$; CV $= 10 \times 10$).

No.	$P(Y=1)$	Metalearner	SL AUC Gain	SL AUC	GS AUC	GS Model
1	0.01	NNloglik	0.027	0.754	0.727	gam
2	0.02	AUC_nlopt.19	0.021	0.766	0.745	polymars
3	0.03	AUC_optim.5	0.024	0.772	0.748	polymars
4	0.04	AUC_optim.4	0.019	0.777	0.758	polymars
5	0.05	AUC_optim.3	0.020	0.779	0.759	randomForest
6	0.10	AUC_nlopt.13	0.014	0.786	0.772	randomForest
7	0.20	AUC_nlopt.5	0.007	0.793	0.786	randomForest
8	0.30	AUC_nlopt.13	0.003	0.795	0.792	randomForest
9	0.40	AUC_nlopt.21	0.002	0.798	0.796	randomForest
10	0.50	AUC_nlopt.11	0.001	0.800	0.798	randomForest

Table 4.5: Top metalearner performance for HIGGS datasets, as measured by cross-validated AUC ($n = 100,000$; CV $= 2 \times 2$).

What the results demonstrate in Table 4.4, Table 4.5 and Figure 4.5 is that the AUC gain achieved by using a Super Learner (SL) ensemble versus choosing the best base algorithm, as selected by grid search (GS), is highest in the $P(Y = 1) \leq 10\%$ range. In particular, for the $P(Y = 1) = 4\%$ dataset in Table 4.4, there is nearly a 0.04 gain in cross-validated AUC achieved from using an AUC-maximizing Super Learner, as opposed to selecting the top base learning algorithm, which in this case is a Generalized Additive Model (GAM). For completeness, the cross-validated grid search performance for each of the base algorithms are provided in Table 4.6 and Table 4.7 at the end of this subsection.

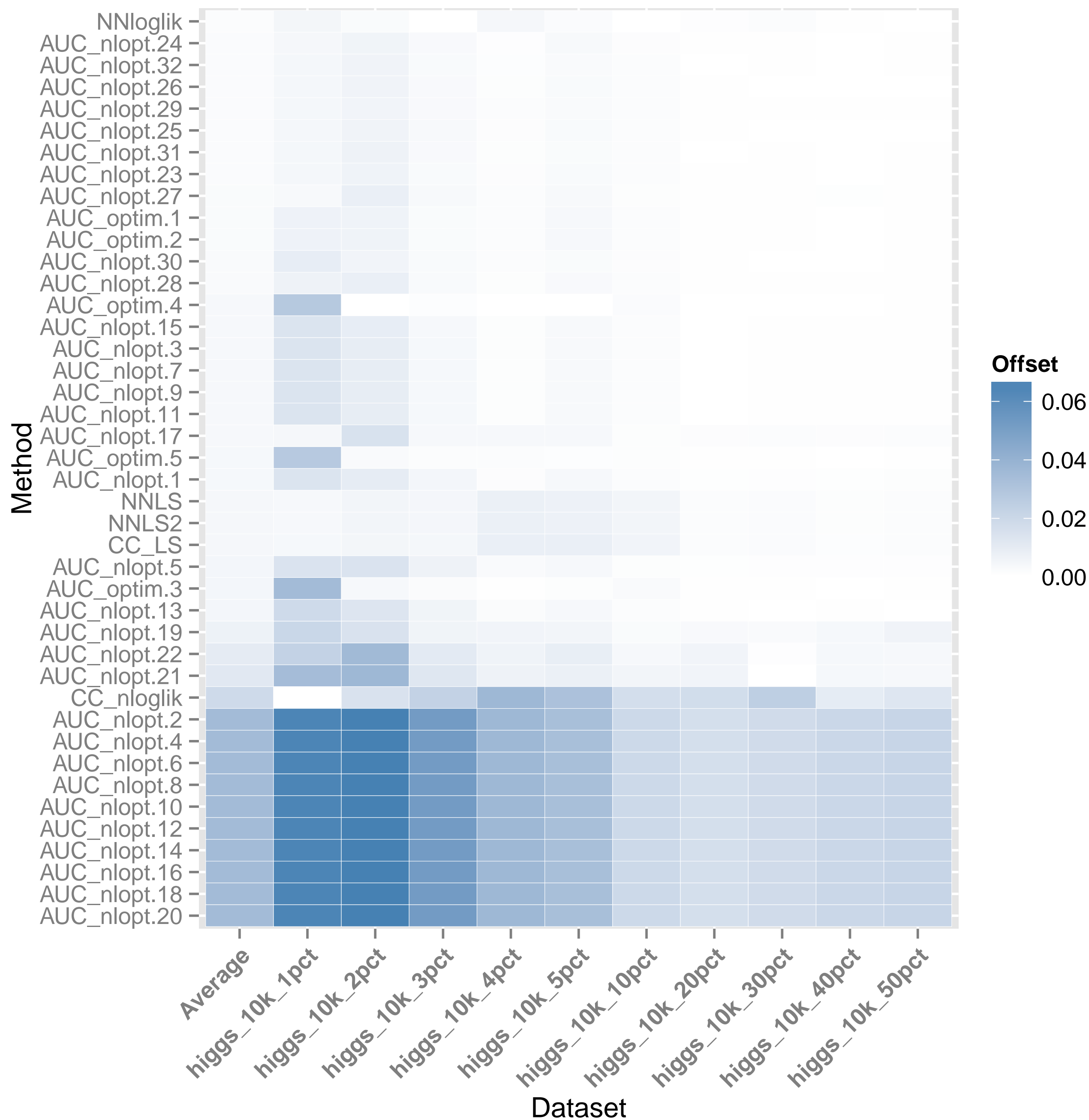

Figure 4.4: Model CV AUC offset from the best Super Learner model for different metalearning methods. (No color == top metalearner)

Figure 4.5 is a plot of the Super Learner AUC gain versus the value for $P(Y = 1)$ for both HIGGS datasets, along with a Loess fit for each group of points. With both datasets, the general trend is that the Super Learner AUC gain increases as the value of $P(Y = 1)$ decreases. However, with the $n = 10,000$ training set, there is a sharp decrease in AUC gain near $P(Y = 1) = 1\%$. This trend is not present in the $n = 100,000$ dataset, so it may be a an artifact of the $n = 10,000$ dataset, due to the global rarity of minority class examples in the training set (there are only 100 minority class observations in total). Future investigation could include examining the level of imbalance with higher granularity, and in particular, in the $P(Y = 1) \leq 1\%$ range, using larger n.

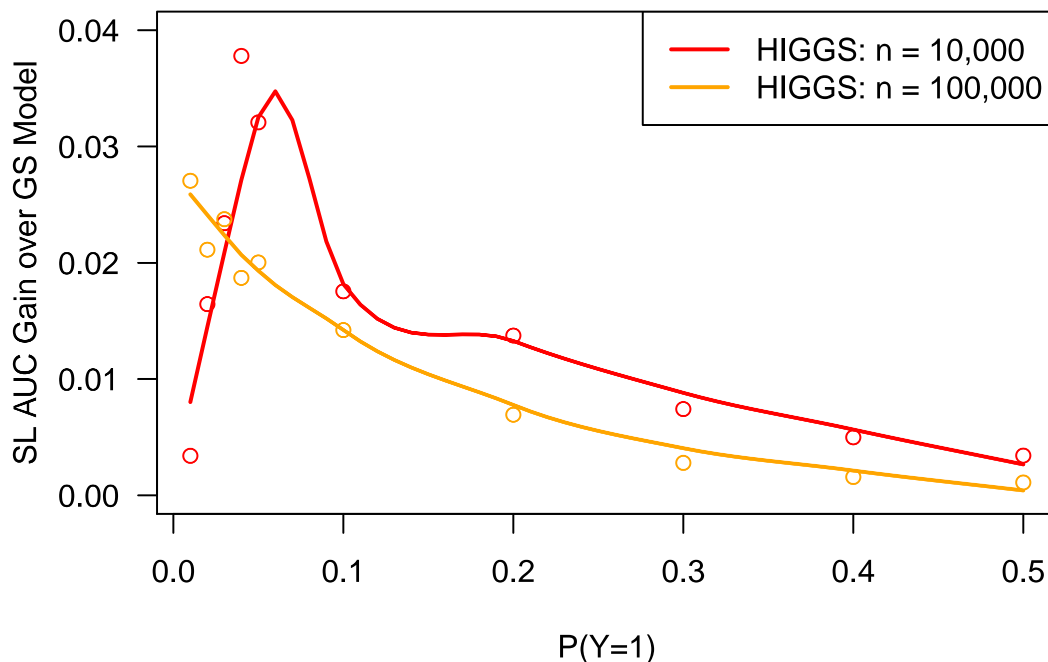

Figure 4.5: CV AUC gain by Super Learner over Grid Search winning model. (Loess fit overlays actual points.)

No.	$P(Y=1)$	SL	glmnet	gam	randomForest	polymars	knn
1	0.01	0.721	0.675	0.718	0.600	0.504	0.487
2	0.02	0.739	0.667	0.723	0.671	0.530	0.490
3	0.03	0.764	0.692	0.741	0.717	0.500	0.508
4	0.04	0.770	0.686	0.732	0.731	0.579	0.540
5	0.05	0.759	0.675	0.723	0.727	0.569	0.540
6	0.10	0.768	0.672	0.720	0.744	0.750	0.554
7	0.20	0.778	0.679	0.720	0.764	0.760	0.579
8	0.30	0.783	0.679	0.721	0.775	0.757	0.593
9	0.40	0.788	0.682	0.722	0.783	0.763	0.595
10	0.50	0.787	0.683	0.722	0.783	0.756	0.600

Table 4.6: CV AUC for HIGGS datasets ($n = 10,000$; CV = 10×10)

No.	$P(Y=1)$	SL	glmnet	gam	randomForest	polymars	knn
1	0.01	0.754	0.679	0.727	0.708	0.500	0.530
2	0.02	0.766	0.683	0.724	0.729	0.745	0.537
3	0.03	0.772	0.684	0.728	0.742	0.748	0.544
4	0.04	0.777	0.684	0.724	0.749	0.758	0.554
5	0.05	0.779	0.687	0.729	0.759	0.753	0.563
6	0.10	0.786	0.689	0.728	0.772	0.771	0.588
7	0.20	0.793	0.684	0.725	0.786	0.776	0.607
8	0.30	0.795	0.682	0.722	0.792	0.767	0.617
9	0.40	0.798	0.680	0.720	0.796	0.769	0.627
10	0.50	0.800	0.682	0.721	0.798	0.771	0.624

Table 4.7: CV AUC for HIGGS datasets ($n = 100,000$; CV = 2×2)

Effect of regularization on the base learner weights

The odd-numbered **NLopt**-based AUC-maximizing metalearning methods enforce box constraints – weights are bounded below by 0 and above by 1. As shown in Table 4.2, the even-numbered **NLopt**-based methods are the partially unconstrained counterparts of the odd methods, where the upper bound is ∞. The top metalearner offset heatmap in Figure 4.4 shows that all of partially unconstrained AUC-maximizing metalearning methods do poorly in comparison to their fully constrained counterparts.

In Figure 4.4, the partially unconstrained NLopt methods are removed in order to more clearly demonstrate the differences between the box-constrained methods.

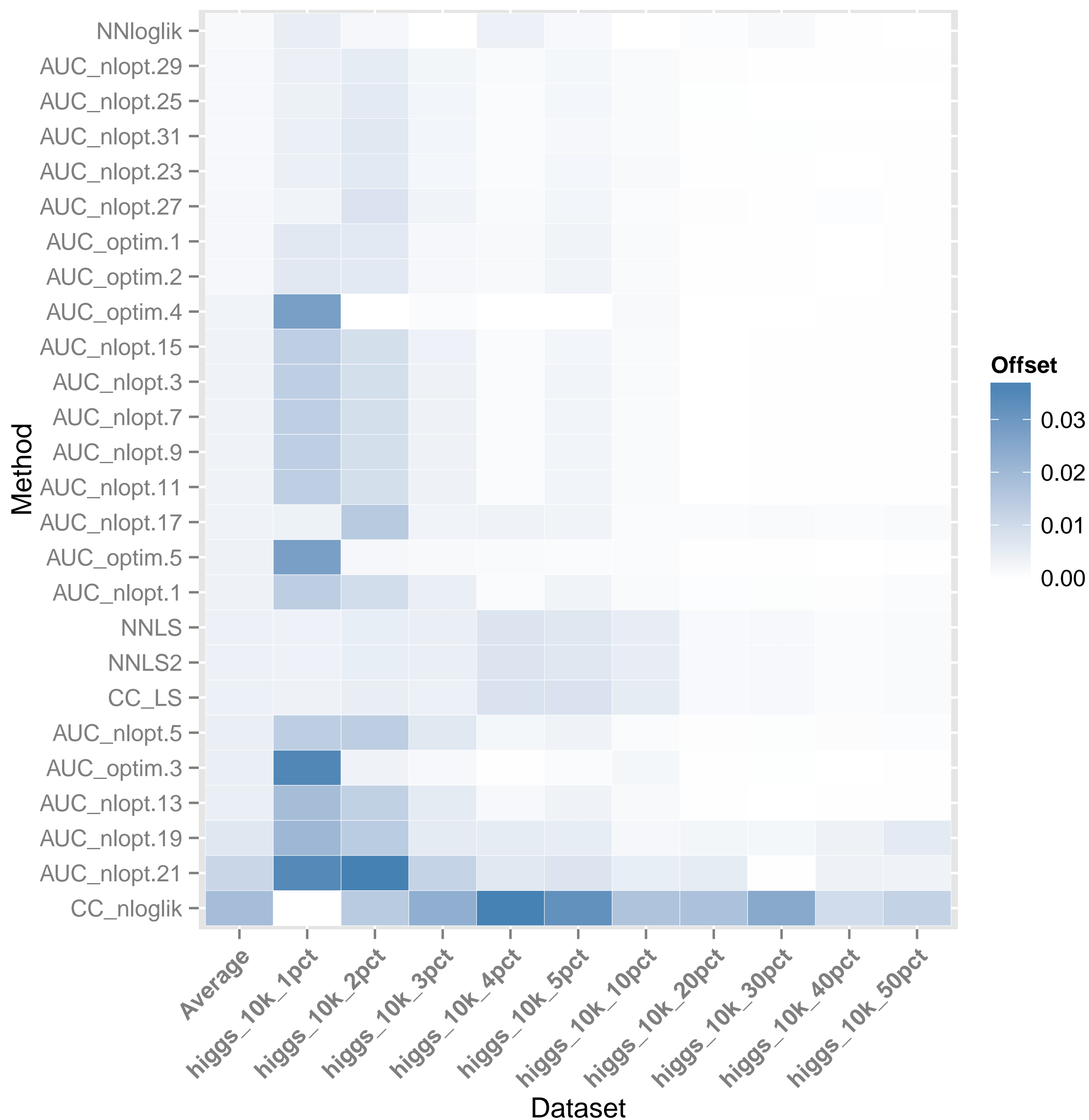

Figure 4.6: Model CV AUC offset from the best Super Learner model for the subset of the metalearning methods that enforce box-constraints on the weights. (No color == top metalearner)

4.5 Conclusion

In this chapter, we describe an implementation of an AUC-maximizing Super Learner algorithm that can be replicated for any loss function of interest. Due to the popularity of AUC as a model performance estimator in binary classification problems, we chose rank loss (1 - AUC) as an example to demonstrate this technique. The new AUC-maximizing metalearning functionality, as well as utility functions for efficient re-estimation of the metalearning algorithm in an existing Super Learner fit, have been contributed to the **SuperLearner** R package.

We benchmarked the AUC-maximizing metalearning algorithms against various existing metalearners that do not target AUC. The results suggest that the use of an AUC-maximizing metalearner in the Super Learner algorithm will typically lead to higher performance, with respect to cross-validated AUC of the ensemble. The benchmarks indicate that AUC-targeting is most effective when the outcome is imbalanced – in particular, in the $P(Y = 1) \leq 10\%$ range. The benchmarks also show that NNLS, which has been widely used as a metalearning method in stacking algorithms since it was first suggested by Breiman, consistently performs worse than AUC-maximizing metalearners.

In the current implementation of `method.AUC`, the weights are initialized as $\alpha_{init} = (\frac{1}{L}, ..., \frac{1}{L})$, where L is the number of base learners. It would also be interesting to investigate the effect of initializing the weights using existing information about the problem, for example, let $\alpha_{init} = (0, ..., 0, 1, 0, ..., 0)$, where the 1 is located at the index of the top base learner, as determined by cross-validation. Further, it may increase performance to use the global optimum as input for a local optimization algorithm, although this technique for chaining algorithms together was not evaluated in our study.

Chapter 5

Computationally Efficient Variance Estimation for Cross-validated AUC

5.1 Introduction

The area under the ROC curve, or AUC, is a ranking-based measure of performance in binary classification problems. Its value can be interpreted as the probability that a randomly selected positive sample will rank higher than a randomly selected negative sample. AUC is a more discriminating performance measure than accuracy [2], and is invariant to relative class distributions [3].

In practice, we are generally concerned with how well our results will generalize to new data. Cross-validation is a means of obtaining an estimate that is generalizable to data drawn from the same distribution but not used in the training set. Common types of cross-validation procedures include k-fold [4], leave-one-out [78, 7, 4], and leave-p-out [75] cross-validation. Given the advantages of AUC as a performance measure, along with the desire to produce generalizable results, cross-validated AUC is frequently used in binary classification problems.

An important task in any estimation procedure is to rigorously quantify the uncertainty in the estimates. In many cases, specification of a parametric model known to contain the truth is not possible, and approaches to inference which are robust to model misspecification are therefore needed. Two approaches to robust inference include inference based on resampling methods, and inference based on influence curves (also known as influence functions). In practice, resampling methods such as the nonparametric bootstrap [26, 27], are commonly used due to their generic nature and simplicity. However, when data sets are large or when methods for training a prediction model are complex, bootstrapping can quickly become a computationally prohibitive procedure.

Although cross-validation lends itself well to parallelization, it can still take a very long time to generate a cross-validated performance measure, such as cross-validated AUC, depending on the complexity of the algorithm used to train the prediction model or the size of

the training set. In machine learning, ensemble methods are prediction methods that make use of, or combine, several or many candidate learning algorithms to obtain better predictive performance. This boost in performance is often accompanied by an increase in the time it takes to generate cross-validated predictions. Alternatively, given massive data sets, even simple prediction methods can be computationally expensive. In cases where obtaining a single estimate of cross-validated AUC requires a significant amount of time and/or resources, the bootstrap is either not an option, or at the very least, a undesirable option for obtaining variance estimates.

As a response to the computational costs of the bootstrap, variations of the bootstrap have been developed that achieve a more desirable computational footprint, such as the "m out of n bootstrap" [11] and subsampling [67]. Another recent advancement that has been made in this area is the "Bag of Little Bootstraps" (BLB) method [1]. Unlike previous variations, BLB simultaneously addresses computational costs, statistical correctness and automation, which appears to be a promising generalized method for variance estimation on massive data sets.

Regardless of the reduction in computation that different variations of the bootstrap offer, all bootstrapping variants require repeated estimation on at least some subset of the original data. By using influence curves for variance estimation, we avoid the need to re-estimate our parameter of interest, which in the case of cross-validated AUC, requires fitting additional models. In order to estimate variance using influence curves, you must first, unsurprisingly, calculate the influence curve for your estimator. For complex estimators, it can be a difficult task to derive the influence curve. However, once the derivation is complete, variance estimation is reduced to a simple and computationally negligible calculation. This is the main motivation for our use of influence curves as a means of variance estimation.

The main goal of this chapter is to establish an influence curve based approach for estimating the asymptotic variance of the cross-validated area under the ROC curve estimator. We first define true cross-validated AUC along with a corresponding estimator and then provide a brief overview of influence curve based variance estimation. We derive the influence curve for the AUC of both i.i.d. data and pooled repeated measures data (multiple observations per independent sampling unit, such as a patient), and demonstrate the construction of influence curve based confidence intervals. We conclude with a simulation that evaluates the coverage probability of the confidence intervals and provide a comparison to bootstrapped based confidence intervals. The methods are implemented in a publicly available R package called **cvAUC** [54].

5.2 Cross-validated AUC as a target parameter

In this section, we formally introduce AUC. We then define the estimator for cross-validated AUC, as well as the target that it is estimating, the true cross-validated AUC.

Consider some probability distribution, P_0, that is known to be an element of a statistical model, $\mathcal{M}$. Let $O = (X, Y) \sim P_0 \in \mathcal{M}$, where Y is a binary outcome variable, and $X \in \mathbb{R}^p$

represents one or more covariates or predictor variables ($p \geq 1$). Without loss of generality, we will denote $Y = 1$ as the positive class and $Y = 0$ as the negative class, and ψ as a function that maps X into $(0, 1)$. The quantity, $\psi(X)$, is the predicted value or score of a sample. The Area Under the ROC curve can be defined as the following:

$$AUC(P_0, \psi) = \int_0^1 P_0\left(\psi(X) > c \mid Y = 1\right) P_0\left(\psi(X) = c \mid Y = 0\right) dc. \tag{5.1}$$

Alternatively, we can define AUC as

$$AUC(P_0, \psi) = P_0\left(\psi(X_1) > \psi(X_2) \mid Y_1 = 1, Y_2 = 0\right), \tag{5.2}$$

where (X_1, Y_1) and (X_2, Y_2) are i.i.d. samples from P_0. The quantity, $AUC(P_0, \psi)$, the true AUC, equals the probability, conditional on sampling two independent observations where one is positive ($Y_1 = 1$) and the other is negative ($Y_2 = 0$), that the predicted value (or rank) of the positive sample, $\psi(X_1)$, is higher than the predicted value (or rank) of the negative sample, $\psi(X_2)$.

Consider $O_1, ..., O_n$, i.i.d. samples from P_0, such that $O_i = (X_i, Y_i)$ for each i, and let P_n denote the empirical distribution. Let n_0 be the number of observations with $Y = 0$ and let n_1 be the number of observations with $Y = 1$. In the context of machine learning, the ψ function is what is learned by a binary prediction algorithm using the training data. The AUC of the empirical distribution can be written as follows:

$$\begin{aligned} AUC(P_n, \psi) &= \frac{1}{n_0 n_1} \sum_{i=1}^{n} \sum_{j=1}^{n} I\left(\psi(X_j) > \psi(X_i)\right) I\left(Y_i = 0, Y_j = 1\right) \\ &= \frac{1}{n_0 n_1} \sum_{i=1}^{n_0} \sum_{j=1}^{n_1} I\left(\psi(X_j) > \psi(X_i)\right), \end{aligned}$$

where I is the indicator function.

We focus on estimating cross-validated AUC. We do not require that the cross-validation be any particular type; however, in practice, k-fold is common. We will use a generalized notation to encode the data splitting procedure, where a binary indicator vector is used to specify which observations belong to the validation set at each iteration of the cross-validation process. Let $B_n^1, ..., B_n^V$ be the collection of random splits that define our cross-validation procedure, where $B_n^v \in \{0, 1\}^n$. In the case of k-fold cross-validation, $k = V$, and each of the B_n^v encodes a single fold. The vth validation fold is the set of observations indexed by $\{i : B_n^v(i) = 1\}$, and the remaining observations belong to the vth training set, $\{i : B_n^v(i) = 0\}$.

Let $\mathcal{M}_{NP}$ denote a nonparametric model that includes the empirical distribution, P_n, and let $\hat{\Psi} : \mathcal{M}_{NP} \to \mathbb{R}$ be an estimator of target parameter, ψ_0, true cross-validated AUC. We assume that $\hat{\Psi}(P_0) = \psi_0$.

For each B_n^v, we define $\psi_{B_n^v} = \hat{\Psi}(P^0_{n,B_n^v})$, where P^0_{n,B_n^v} is the empirical distribution of the observations contained in the vth training set. The function $\psi_{B_n^v}$, which is learned from the

v^{th} training set, will be used to generate predicted values for the observations in the vth validation fold. We define n_1^v and n_0^v to be the number of positive and negative samples in the vth validation fold, respectively. Formally, $n_1^v = \sum_{i=1}^n I(Y_i = 1) I(B_n^v(i) = 1)$ and $n_0^v = \sum_{i=1}^n I(Y_i = 0) I(B_n^v(i) = 1)$. We note that n_1^v and n_0^v are random variables that depend on the value of both B_n^v and $\{Y_i : B_n^v(i) = 1\}$. The AUC for a single validation fold, $\{i : B_n^v(i) = 1\}$, is:

$$AUC(P^1_{n,B_n^v}, \psi_{B_n^v}) = \frac{1}{n_0^v n_1^v} \sum_{i=1}^n \sum_{j=1}^n I\left(\psi_{B_n^v}(X_j) > \psi_{B_n^v}(X_i)\right) I(Y_i = 0, Y_j = 1) I(B_n^v(i) = B_n^v(j) = 1).$$

Then the V-fold cross-validated AUC estimator is defined as:

$$\begin{aligned} E_{B_n} AUC(P^1_{n,B_n}, \psi_{B_n}) &= \frac{1}{V} \sum_{v=1}^V AUC(P^1_{n,B_n}, \psi_{B_n^v}) \\ &= \frac{1}{V} \sum_{v=1}^V \frac{1}{n_0^v n_1^v} \sum_{i=1}^n \sum_{j=1}^n I\left(\psi_{B_n^v}(X_j) > \psi_{B_n^v}(X_i)\right) \\ &\times \; I(Y_i = 0, Y_j = 1) I(B_n^v(i) = B_n^v(j) = 1). \end{aligned} \tag{5.3}$$

The target, ψ_0, of the V-fold cross-validated AUC estimator is defined as:

$$\begin{aligned} E_{B_n} AUC(P_0, \psi_{B_n}) &= \frac{1}{V} \sum_{v=1}^V AUC(P_0, \psi_{B_n^v}) \\ &= \frac{1}{V} \sum_{v=1}^V P_0\left(\psi_{B_n^v}(X_1) > \psi_{B_n^v}(X_2) \mid Y_1 = 1, Y_2 = 0\right), \end{aligned} \tag{5.4}$$

where (X_1, Y_1) and (X_2, Y_2) are i.i.d. samples from P_0. In other words, our target parameter, the true cross-validated AUC, corresponds to fitting the prediction function on each training set, evaluating its true performance (or true probability of correctly ranking two randomly selected observations, where one is a positive sample and the other a negative sample) in the corresponding validation set, and finally, taking the average over the validation sets. The true value of this target parameter is random, in that it depends on the split of the sampled data into training sets and corresponding fits of the prediction function. We now wish to construct confidence intervals for our estimator of cross-validated AUC, $E_{B_n} AUC(P^1_{n,B_n}, \psi_{B_n})$.

5.3 Influence curves for variance estimation

We provide a brief overview of influence curves and their relation to variance estimation. We outline the general procedure for obtaining confidence intervals using the influence curve

of an estimator. This section serves as a gentle introduction to concepts and notation used throughout the remainder of the chapter.

Suppose that $O \equiv O_1, ..., O_n$ are i.i.d. samples from a probability distribution, P_0, that is known to be an element of a statistical model, $\mathcal{M}$. Let $\mathcal{F}$ be some class of functions of O. Throughout this chapter, we will use the notation Pf, where P is a probability distribution, to denote $\int f(x)dP(x)$. We consider the empirical process, $(P_0 f : f \in \mathcal{F})$, which is a "vector" of true means. Let $\Psi : \mathcal{M} \to \mathbb{R}^d$ be a parameter of interest, and let $\psi_0 = \Psi(P_0) \equiv \Psi(P_0 f : f \in \mathcal{F})$ be the true parameter value; ψ_0 is a function of true means. Now let $\mathcal{M}_{NP}$ denote a nonparametric model that includes the empirical distribution, P_n, of $O_1, ..., O_n$. We consider the empirical process, $(P_n f : f \in \mathcal{F})$, which is a "vector" of empirical means. Let $\hat{\Psi} : \mathcal{M}_{NP} \to \mathbb{R}^d$ be an estimator of ψ_0 that maps the empirical distribution, P_n, or rather, a "vector" of empirical means, into an estimate $\hat{\Psi}(P_n) \equiv \hat{\Psi}(P_n f : f \in \mathcal{F})$. We assume that $\hat{\Psi}(P_0) = \psi_0$, so that the estimator targets the desired target parameter, ψ_0. This estimate is *asymptotically linear* at P_0 if

$$\hat{\Psi}(P_n) - \hat{\Psi}(P_0) = \frac{1}{n}\sum_{i=1}^{n} IC(P_0)(O_i) + o_P(1/\sqrt{n}) \tag{5.5}$$

for some zero-mean function, $IC(P_0)$, of O (i.e. $P_0 IC(P_0) = 0$). The function, $IC(P_0)$, is called the influence curve, or influence function, of the estimator, $\hat{\Psi}$. The main task in the process of constructing influence curve based confidence intervals is demonstrating the asymptotic linearity of your estimator.

By the Central Limit Theorem, we find that $\sqrt{n}\,\left(\hat{\Psi}(P_n) - \hat{\Psi}(P_0)\right) \xrightarrow{d} \mathcal{N}(0, \Sigma_0)$, where $\Sigma_0 = P_0 IC(P_0) IC(P_0)^T$. This covariance matrix can be estimated with the empirical covariance matrix, $\widehat{IC}(O_i)$, $i = 1, ..., n$ where $\widehat{IC}$ is an estimate of $IC(P_0)$. When our target parameter is one-dimensional, as in cross-validated AUC, we can write the following:

$$\sqrt{n}\,\left(\hat{\Psi}(P_n) - \hat{\Psi}(P_0)\right) \xrightarrow{d} \mathcal{N}\left(0, \Phi^2(P_0)\right), \tag{5.6}$$

where $\Phi^2(P_0) = \int IC(P_0)(x)^2 dP_0(x)$. We can estimate $\Phi^2(P_0)$ as

$$\Phi_n^2 \equiv \Phi^2(P_n) = \frac{1}{n}\sum_{i=1}^{n} IC(P_n)(O_i)^2, \tag{5.7}$$

however, other estimators of the variance of the influence curve can be considered. Letting z_r denote the rth quantile of the standard normal distribution, it follows that for any estimate $\Phi_n^2 \equiv \Phi^2(P_n)$ of $\Phi^2(P_0)$, we have that

$$\left(\hat{\Psi}(P_n) - z_{1-\alpha/2}\frac{\Phi_n}{\sqrt{n}}\ ,\ \ \hat{\Psi}(P_n) + z_{1-\alpha/2}\frac{\Phi_n}{\sqrt{n}}\right) \tag{5.8}$$

forms an approximate $100 \times (1-\alpha)\%$ confidence interval for $\psi_0 \equiv \hat{\Psi}(P_0)$.

In order to assume that asymptotically linear estimators of ψ_0 exist, we must assume that the parameter Ψ is pathwise differentiable [12]. This method for establishing the asymptotic linearity and normality of the estimator is called the *functional delta method* [81, 32], which is a generalization of the classical delta method for finite dimensional functions of a finite set of estimators.

5.4 Confidence intervals for cross-validated AUC

In this section, we establish the influence curve for AUC and show that the empirical AUC is an asymptotically linear estimator of the true AUC. Using these results, we follow the methodology from Section 5.3 to derive confidence intervals for cross-validated AUC. Then we provide a description of the practical construction of the confidence intervals from an i.i.d. data sample.

Theorem 2. *Let $O = (X, Y) \sim P_0$, where $X \in \mathbb{R}^p$ represents one or more variables and Y is binary. Without loss of generality, assume $Y \in \{0, 1\}$ and that ψ is a function that maps X into (0,1). Define $AUC(P_0, \psi)$ as*

$$\int_0^1 P_0\left(\psi(X) > c \mid Y = 1\right) P_0\left(\psi(X) = c \mid Y = 0\right) dc.$$

The efficient influence curve of $AUC(P_0, \psi)$, evaluated at a single observation, $O_i = (X_i, Y_i)$, for a nonparametric model for P_0 is given by

$$\begin{aligned} IC_{AUC}(P_0, \psi)(O_i) = & \frac{I(Y_i = 1)}{P_0(Y = 1)} P_0\left(\psi(X) < w \mid Y = 0\right)\Big|_{w=\psi(X_i)} \\ & + \frac{I(Y_i = 0)}{P_0(Y = 0)} P_0\left(\psi(X) > w \mid Y = 1\right)\Big|_{w=\psi(X_i)} \\ & - \left\{ \frac{I(Y_i = 0)}{P_0(Y = 0)} + \frac{I(Y_i = 1)}{P_0(Y = 1)} \right\} AUC(P_0, \psi). \end{aligned}$$

For each ψ, the empirical $AUC(P_n, \psi)$ is asymptotically linear with influence curve, $IC_{AUC}(P_0, \psi)$. Let $B_n \in \{0, 1\}^n$ be a random split of the observations into a training and validation set. Let P^1_{n,B_n} and P^0_{n,B_n} be the empirical distributions of the validation set, $\{i : B_n(i) = 1\}$, and training set, $\{i : B_n(i) = 0\}$, respectively. We assume that B_n has only a finite number of values uniformly in n, as in k-fold cross-validation. We assume that $p = \sum_i B_n(i)/n$ is bounded away from a $\delta > 0$, with probability 1. Define the cross-validated area under the ROC curve as

$$\hat{R}(\hat{\Psi}, P_n) = E_{B_n} AUC\left(P^1_{n,B_n}, \hat{\Psi}(P^0_{n,B_n})\right). \tag{5.9}$$

We also define the target of this cross-validated area under the ROC curve as

$$\tilde{R}(\hat{\Psi}, P_n) = E_{B_n} AUC\left(P_0, \hat{\Psi}(P^0_{n,B_n})\right). \tag{5.10}$$

We assume that there exists a $\psi_1 \in \mathbf{\Psi}$ *so that* $P_0 \left\{ IC_{AUC}\left(P_0, \hat{\Psi}(P_n)\right) - IC_{AUC}(P_0, \psi_1) \right\}^2$ *converges to zero in probability as* $n \to \infty$*. We also assume that* $\sup_{\psi \in \mathbf{\Psi}} \sup_O |IC_{AUC}(P_0, \psi)(O)| < \infty$*, where the supremum over* O *is over a support of* P_0*. Then,*

$$\hat{R}(\hat{\Psi}, P_n) - \tilde{R}(\hat{\Psi}, P_n) = \frac{1}{n} \sum_{i=1}^{n} IC_{AUC}(O_i) + o_P(1/\sqrt{n}). \tag{5.11}$$

In particular, $\sqrt{n}\left(\hat{R}(\hat{\Psi}, P_n) - \tilde{R}(\hat{\Psi}, P_n)\right)$ *converges to a normal distribution with mean zero and variance,* $\sigma^2 = P_0 \left\{ IC_{AUC}(P_0, \psi_1) \right\}^2$*. Thus, one can construct an asymptotically* 0.95*-confidence interval for* $\tilde{R}(\hat{\Psi}, P_n)$ *given by* $\hat{R}(\hat{\Psi}, P_n) \pm 1.96 \frac{\sigma_n}{\sqrt{n}}$*, where* σ_n^2 *is a consistent estimator of* σ^2*. A consistent estimator of* σ^2 *is obtained as*

$$\sigma_n^2 = E_{B_n} P_{n,B_n}^1 \left\{ IC_{AUC}\left(P_{n,B_n}^1, \hat{\Psi}(P_{n,B_n}^0)\right) \right\}^2. \tag{5.12}$$

Proof. In order to derive influence curve based confidence intervals for cross-validated AUC, we must first derive the influence curve for AUC and show that $AUC(P_n, \psi)$ is an asymptotically linear estimator of $AUC\left(P_0, \psi\right)$ with influence curve as specified in the theorem. For that purpose we use the functional delta method [81, 32]. The asymptotic linearity of $AUC(P_n, \psi)$ is an immediate consequence of the compact differentiability of functionals $(F_1, F_2) \to \int F_1(x) dF_2(x)$ for cumulative distribution functions (F_1, F_2) [32], so the functional delta method can be applied here as well. Therefore it only remains to determine the actual influence curve which is defined in terms of the Gateaux derivative of $P \to AUC(P, \psi)$ in the direction of the empirical distribution for a single observation, O.

We define $F_a(c) = P_0(\psi(X) < c \mid Y = a)$ for $a \in \{0, 1\}$. Therefore, we can alternatively express true AUC as

$$AUC\left(P_0, \psi\right) = \Phi(F_0, F_1) = \int (1 - F_1(c)) dF_0(c). \tag{5.13}$$

The Gateaux derivative of $\Phi(F_0, F_1)$ in direction (h_0, h_1) is given by:

$$\frac{d}{d\epsilon} \Phi(F_0 + \epsilon h_0, F_1 + \epsilon h_1) \Big|_{\epsilon=0} = \int -h_1(c) dF_0(c) + \int (1 - F_1(c)) dh_0(c)$$

Therefore, we have the following linear approximation:

$$\Phi(F_{0n}, F_{1n}) - \Phi(F_0, F_1) = \int -(F_{1n} - F_1) dF_0 + \int (1 - F_1) d(F_{0n} - F_0)$$

Let F_{0n}, F_{1n} be the empirical distributions of F_0, F_1. Next we derive the linear approximations of $F_{1n} - F_1$ and $F_{0n} - F_0$. Note that for $a \in \{0, 1\}$,

$$F_{an}(c) = P_n(\psi(X) < c \mid Y = a) = \frac{P_n(\psi(X) < c, Y = a)}{P_n(Y = a)} \tag{5.14}$$

It follows that $F_{an}(c) - F_a(c) \sim$

$$\begin{aligned}
&\frac{(P_n - P_0)(\psi(X) < c, Y = a)}{P_0(Y = a)} - \frac{P_0(\psi(X) < c, Y = a)}{[P_0(Y = a)]^2}(P_n(Y = a) - P_0(Y = a)) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left\{\frac{I(\psi(X_i) < c, Y_i = a)}{P_0(Y = a)} - \frac{F_a(c)I(Y_i = a)}{P_0(Y = a)}\right\}
\end{aligned}$$

So the influence curve of $F_{an}(c)$ for a single observation, $O_i = (X_i, Y_i)$, is:

$$\frac{I(\psi(X_i) < c, Y_i = a)}{P_0(Y = a)} - \frac{F_a(c)I(Y_i = a)}{P_0(Y = a)} \tag{5.15}$$

We can substitute this for h_a in the linear approximation above resulting in the desired influence curve, $IC_{AUC}(P_0, \psi)$, as presented in the theorem. For that, it is helpful to observe that:

$$\begin{aligned}
\int I(\psi(X_i) < c, Y_i = 1)dF_0(c) &= I(Y_i = 1)\int_{\psi(X_i)<c} dF_0(c) && (5.16)\\
&= I(Y_i = 1)(1 - F_0(\psi(X_i)) && (5.17)
\end{aligned}$$

This is the influence curve for $AUC(P_n, \psi)$, and, since the model $\mathcal{M}$ for P_0 is nonparametric, this is also the efficient influence curve of parameter $AUC(P_0, \psi)$ on a nonparametric model.

Using the notation that was defined in Section 5.2, it follows that

$$\begin{aligned}
&E_{B_n} AUC(P^1_{n,B_n}, \hat{\Psi}(P^0_{n,B_n})) - E_{B_n} AUC(P_0, \hat{\Psi}(P^0_{n,B_n})) \\
&\qquad = E_{B_n}(P^1_{n,B_n} - P_0)IC_{AUC}(P_0, \hat{\Psi}(P^0_{n,B_n})) + E_{B_n} R(P^1_{n,B_n}, \hat{\Psi}(P^0_{n,B_n})) \\
&\qquad \approx E_{B_n}(P^1_{n,B_n} - P_0)IC_{AUC}(P_0, \hat{\Psi}(P^0_{n,B_n})) + o_P(1/\sqrt{n}) \\
&\qquad = E_{B_n}(P^1_{n,B_n} - P_0)IC_{AUC}(P_0, \psi_1) \\
&\qquad\quad + E_{B_n}(P^1_{n,B_n} - P_0)\left\{IC_{AUC}(P_0, \hat{\Psi}(P^0_{n,B_n})) - IC_{AUC}(P_0, \psi_1)\right\} \\
&\qquad\quad + o_P(1/\sqrt{n}) \\
&\qquad = E_{B_n}(P^1_{n,B_n} - P_0)IC_{AUC}(P_0, \psi_1) + o_P(1/\sqrt{n}) \\
&\qquad = (P_n - P_0)IC_{AUC}(P_0, \psi_1) + o_P(1/\sqrt{n}).
\end{aligned}$$

At the first equality we apply the previously established asymptotic linearity of $AUC(P_{n,B^1_n}, \psi)$, conditional on the training sample, which proves that each B_n-specific remainder,

$R(P^1_{n,B_n}, \hat{\Psi}(P^0_{n,B_n}))$, is $o_P(1/\sqrt{n})$. Since there are only a finite number of possible B_n, this also proves the next equivalence stating that the average across the different B_n-splits of the remainder is also $o_P(1/\sqrt{n})$. In the third equality, we just carry out a simple split of the empirical process in two terms. In the statement of the theorem, we assume that $P_0 \left\{ IC_{AUC}(P_0, \hat{\Psi}(P_n)) - IC_{AUC}(P_0, \psi_1) \right\}^2$ converges to zero in probability as $n \to \infty$ for some ψ_1. Using a result from [91] involving the application of empirical process theory (specifically Lemma 2.14.1, which references [81]), the term,
$E_{B_n}(P^1_{n,B_n} - P_0) \left\{ IC_{AUC}(P_0, \hat{\Psi}(P^0_{n,B_n})) - IC_{AUC}(P_0, \psi_1) \right\}$, is shown to be $o_P(1/\sqrt{n})$, which results in the fourth equality.

Finally, $E_{B_n}(P^1_{n,B_n} - P_0) IC_{AUC}(P_0, \psi_1) = (P_n - P_0) IC_{AUC}(P_0, \psi_1)$, proving the asymptotic linearity of the cross-validated AUC estimator as stated in the final equality. In particular,

$$\sqrt{n} \left(E_{B_n} AUC(P^1_{n,B_n}, \hat{\Psi}(P^0_{n,B_n})) - E_{B_n} AUC(P_0, \hat{\Psi}(P^0_{n,B_n})) \right)$$

converges to a normal distribution with mean zero and variance, $\sigma^2 = P_0 \left\{ IC_{AUC}(P_0, \psi_1) \right\}^2$. A consistent estimator of σ^2 is obtained as

$$\sigma^2_n = E_{B_n} P^1_{n,B_n} \left\{ IC_{AUC} \left(P^1_{n,B_n}, \hat{\Psi}(P^0_{n,B_n}) \right) \right\}^2 .$$

For σ^2_n, we estimate the unknown conditional probabilities of the influence curve IC_{AUC} with the empirical distribution of the validation set, so that $P^1_{n,B_n}(\psi(X) > w \mid Y = 0)$ will be consistent at $\psi = \hat{\Psi}(P^0_{n,B_n})$ under no conditions on the estimator $\hat{\Psi}$. This is why we replaced P_0 in $IC_{AUC}(P_0, \psi)$ by the empirical distribution of the validation set. However, the probabilities $P_0(Y = 1)$ and $P_0(Y = 0)$ can be estimated using the whole sample.

Thus, one can construct an asymptotically 0.95-confidence interval for $E_{B_n} AUC(P_0, \hat{\Psi}(P^0_{n,B_n}))$ given by $E_{B_n} AUC(P^1_{n,B_n}, \hat{\Psi}(P^0_{n,B_n})) \pm 1.96 \frac{\sigma_n}{\sqrt{n}}$.

□

A practical implementation for i.i.d. data

For further clarity, we provide a description of the practical construction of the confidence intervals from an i.i.d. data set, as implemented in our software package. Consider an i.i.d. sample of size n with a binary outcome Y. For each observation, $O_i = (X_i, Y_i)$, we have a p-dimensional numeric vector, $X_i \in \mathbb{R}^p$, and a binary outcome variable, Y_i. Without loss of generality, let $Y_i \in \{0, 1\}$, for all $i = 1, ..., n$, however, Y can be any ordered two-class variable. In this example, we will use k-fold cross-validation for $k = V > 1$ and define the splits as $B^1_n, ..., B^V_n$. Calculating the V-fold cross validated AUC estimate corresponds to:

1. Building or fitting the prediction function on each of V validation sets.

2. Generating a predicted outcome for each observation in the vth validation set. The predictions are generated using a fit that was trained on the $\{1, ..., V\} \setminus v$ folds.

3. For each validation fold, using these predicted values, together with the observed outcomes for each observation, to generate an estimate of the AUC for that validation fold.

4. Average these estimates across the V validation folds to calculate the V-fold cross-validated AUC.

Recall that P^1_{n,B^v_n} and P^0_{n,B^v_n} are the empirical distributions of the vth validation and training set, respectively and P_n is the empirical distribution of the whole data sample. The V-fold cross-validated AUC estimate, denoted $\hat{R}(\hat{\Psi}, P_n)$, is given by $\frac{1}{V}\sum_{v=1}^{V} AUC\left(P^1_{n,B_n}, \psi_{B^v_n}\right)$. In order to construct influence curve based confidence intervals for $\hat{R}(\hat{\Psi}, P_n)$, we estimate the asymptotic variance as:

$$\sigma^2_n = E_{B_n} P^1_{n,B_n} \left\{ IC_{AUC}\left(P^1_{n,B_n}, \hat{\Psi}(P^0_{n,B_n})\right)\right\}^2 \tag{5.18}$$

$$= \frac{1}{V}\sum_{v=1}^{V}\left\{\frac{1}{n}\sum_{i=1}^{n}\left\{IC_{AUC}\left(P^1_{n,B^v_n}, \hat{\Psi}(P^0_{n,B^v_n})\right)(O_i)\right\}^2 I\left(B^v_n(i)=1\right)\right\}, \tag{5.19}$$

where $\psi_{B^v_n} = \hat{\Psi}(P^0_{n,B^v_n})$, and for each $v \in \{1, ..., V\}$ and $i \in \{1, ..., n\}$, we have

$$\begin{aligned} IC_{AUC}(P^1_{n,B^v_n}, \hat{\Psi}(P^0_{n,B^v_n}))(O_i) & \\ = \frac{I(Y_i=1)}{P_n(Y=1)} P^1_{n,B^v_n}\left(\psi_{B^v_n}(X) < w \mid Y=0\right)\Big|_{w=\psi_{B^v_n}(X_i)} & \\ + \frac{I(Y_i=0)}{P_n(Y=0)} P^1_{n,B^v_n}\left(\psi_{B^v_n}(X) > w \mid Y=1\right)\Big|_{w=\psi_{B^v_n}(X_i)} & \\ - \left\{\frac{I(Y_i=0)}{P_n(Y=0)} + \frac{I(Y_i=1)}{P_n(Y=1)}\right\} AUC\left(P^1_{n,B_n}, \psi_{B^v_n}\right). & \end{aligned}$$

Despite the density of the notation above, each of the components in the influence curve can be calculated very easily from the data. The terms, $P_n(Y=1) \equiv \frac{1}{n}\sum_{j=1}^{n} I(Y_j=1)$ and $P_n(Y=0) \equiv \frac{1}{n}\sum_{j=1}^{n} I(Y_j=0)$, are the proportions of positive and negative samples, respectively, in the empirical distribution. Let $n^v_1 = \sum_{j=1}^{n} I(Y_j=1)I(B^v_n(j)=1)$ be the number of positive samples in the vth validation set and let $n^v_0 = \sum_{j=1}^{n} I(Y_j=0)I(B^v_n(j)=1)$ be the number of negative samples in the vth validation set. Also, recall that $\psi_{B^v_n}$ is the function learned by the vth training set, which maps a vector, X, of covariates, to a predicted value, $\psi_{B^v_n}(X) \in (0,1)$. For a given sample, $O_i = (X_i, Y_i)$, we calculate the predicted value, $\psi_{B^v_n}(X_i)$, and note whether Y_i is labeled as positive ($Y_i = 1$) or negative ($Y_i = 0$). Above, each of the terms in the expression for the influence curve contains an indicator function, conditional on the value of Y_i. Therefore, given the value of Y_i, we need only to evaluate the non-zero part of the expression.

When $Y_i = 1$, we need to evaluate:

$$P^1_{n,B^v_n}\left(\psi_{B^v_n}(X) < w \mid Y = 0\right)\Big|_{w=\psi_{B^v_n}(X_i)}$$
$$= \frac{1}{n^v_0}\sum_{j=1}^{n} I\left(X_j < \psi_{B^v_n}(X_i)\right) I\left(Y_j = 0\right) I\left(B^v_n(j) = 1\right)$$

This sum counts the number of negative samples in the validation set that have a predicted value less than $\psi_{B^v_n}(X_i)$, the predicted value for sample i. Then, we divide by the total number of negative samples in the validation set. Similarly, when $Y_i = 0$, we need to evaluate:

$$P^1_{n,B^v_n}\left(\psi_{B^v_n}(X) > w \mid Y = 1\right)\Big|_{w=\psi_{B^v_n}(X_i)}$$
$$= \frac{1}{n^v_0}\sum_{j=1}^{n} I\left(X_j > \psi_{B^v_n}(X_i)\right) I\left(Y_j = 1\right) I\left(B^v_n(j) = 1\right)$$

This sum counts the number of positive samples in the validation set that have a predicted value greater than $\psi_{B^v_n}(X_i)$, the predicted value for sample i. Then, we divide by the total number of positive samples in the validation set. The remaining term in the expression for the influence curve is simply $AUC\left(P^1_{n,B_n}, \psi_{B^v_n}\right)$, given in Section 5.3, multiplied by inverse probability of $P_n(Y = 1)$ or $P_n(Y = 0)$, depending on the value of the indicator function at Y_i. Thus, for fixed $v \in \{1, ..., V\}$ and $i \in \{1, ..., n\}$, we have demonstrated how to calculate the quantity, $IC_{AUC}\left(P^1_{n,B^v_n}, \hat{\Psi}(P^0_{n,B^v_n})\right)(O_i)$, from an i.i.d. data set. Then we square this term and sum over i.i.d. samples, i, and cross-validation folds, v, to get

$$\sigma^2_n = \frac{1}{V}\sum_{v=1}^{V}\left\{\frac{1}{n}\sum_{i=1}^{n}\left\{IC_{AUC}\left(P^1_{n,B^v_n}, \hat{\Psi}(P^0_{n,B^v_n})\right)(O_i)\right\}^2 I\left(B^v_n(i) = 1\right)\right\},$$

an estimate for the asymptotic variance of $\hat{R}(\hat{\Psi}, P_n)$, our V-fold cross-validated AUC estimator. The target of this estimator is

$$\tilde{R}(\hat{\Psi}, P_n) = E_{B_n} AUC\left(P_0, \hat{\Psi}(P^0_{n,B_n})\right) = \frac{1}{V}\sum_{v=1}^{V} AUC\left(P_0, \hat{\Psi}(P^0_{n,B^v_n})\right),$$

the true V-fold cross-validated AUC. Then, as in Theorem 2, one can construct an asymptotically 0.95-confidence interval for $\tilde{R}(\hat{\Psi}, P_n)$ as $\hat{R}(\hat{\Psi}, P_n) \pm 1.96\frac{\sigma_n}{\sqrt{n}}$.

5.5 Generalization to pooled repeated measures data

Above, we derived a consistent influence curve based estimator of the asymptotic variance of cross-validated AUC for the simple setting in which there are n i.i.d. observations. Each of

these observations, O_i has a predictor variable, X_i, coupled with a binary outcome variable, Y_i, that we wish to predict. Now we consider the common setting in which there are repeated measures for each observation. This data structure arises frequently in medical studies, where each patient is measured at multiple time points. We focus on the case where the order of these measures is not meaningful, and one simply wishes to obtain a single summary of classifier performance pooled over all measures. We begin by providing a formal definition of the target parameter, the pooled cross-validated AUC, for such cases. We then extend the results presented in the previous sections to derive an influence curve based variance estimator for the cross-validated AUC of a pooled repeated measures data set.

As before, we let $P_0 \in \mathcal{M}$ and $\Psi : \mathcal{M} \to \boldsymbol{\Psi}$. We denote the target parameter $\Psi(P_0)$ as ψ_0. Let $O = (X(t), Y(t) : t \in \tau) \sim P_0$ for a possibly random index set $\tau \subset \{1, ..., T\}$, where t corresponds to a single time-point observation. Here $Y(t)$ is binary for each t. We observe n i.i.d. copies $O_i = (X_i(t), Y_i(t) : t \in \tau_i)$, $i = 1, ..., n$ of O. Let $\mathcal{M}_{NP}$ denote a nonparametric model that includes the empirical distribution, P_n, of $O_1, ..., O_n$ and let $\hat{\Psi} : \mathcal{M}_{NP} \to \mathbb{R}$ be an estimator of ψ_0. We assume that $\hat{\Psi}(P_0) = \psi_0$. We consider the case where t is not a meaningful index, and that either $\psi_0(t, x) = E_0 \left(Y(t) \mid X(t) = x\right)$ does not depend on t, or that the investigator has no interest in understanding the dependence on t. Consider the distribution,

$$\bar{P}_0(x, y) = \frac{1}{E_0|\tau|} \sum_{t=1}^{T} P_0(t \in \tau) P_0 \left(X(t) = x, Y(t) = y \mid t \in \tau\right).$$

This represents the limit distribution of the empirical distribution $\bar{P}_n$ of the pooled sample:

$$\bar{P}_n(x, y) = \frac{1}{\sum_{i=1}^{n} |\tau_i|} \sum_{i=1}^{n} \sum_{t \in \tau_i} I\left(X_i(t) = x, Y_i(t) = y\right).$$

One could define as a measure of interest for evaluation a predictor ψ, the area under the ROC curve one would obtain if one treats the pooled sample as N i.i.d. observations. That is, we define

$$\overline{AUC}\left(\bar{P}_0, \psi\right) = \int_0^1 \bar{P}_0 \left(\psi(X) > c \mid Y = 1\right) \bar{P}_0 \left(\psi(X) = c \mid Y = 0\right) dc, \qquad (5.20)$$

where, without loss of generality, we let the positive class be represented by $Y = 1$ and the negative class be represented by $Y = 0$. The pooled repeated measures AUC can be interpreted as the probability that, after pooling over all independent sampling units and all time points, a randomly sampled positive outcome will be ranked more highly than a randomly sampled negative outcome.

The AUC for the empirical distribution of the pooled sample can be expressed explicitly as follows. Let $n_0 = \sum_{i=1}^{n} \sum_{t \in \tau_i} I(Y_i(t) = 0)$ and let $n_1 = \sum_{j=1}^{n} \sum_{s \in \tau_j} I(Y_j(s) = 1)$. Then

we have

$$\overline{AUC}\left(\bar{P}_n, \psi\right)$$
$$= \frac{1}{n_0 n_1} \sum_{i=1}^{n} \sum_{t \in \tau_i} \sum_{j=1}^{n} \sum_{s \in \tau_j} I\left(\psi(X_j(s)) > \psi(X_i(t))\right) I\left(Y_i(t) = 0, Y_j(s) = 1\right).$$

Now we consider the cross-validated AUC of a pooled repeated measures data set. Let $B_n \in \{0,1\}^n$ be a random split of the n independent observations into a training and validation set. Let $\bar{P}^1_{n,B_n}$ and $\bar{P}^0_{n,B_n}$ be the empirical distributions of the pooled data within the validation set, $\{i : B_n(i) = 1\}$, and training set, $\{i : B_n(i) = 0\}$, respectively. Again, we assume that B_n has only a finite number of values uniformly in n, as in k-fold cross-validation. Given a random split, B_n, we define $\psi_{B_n} \equiv \hat{\Psi}(\bar{P}^0_{n,B_n})$.

As in the i.i.d. example in the previous section, we will walk through the case of V-fold cross-validation. Let $B^1_n, ..., B^V_n$ be the collection of random splits that define our cross-validation procedure such that each of the B^v_n encodes a single fold. The vth validation fold is $\{i : B^v_n(i) = 1\}$, and the remaining samples belong to the vth training set, $\{i : B^v_n(i) = 0\}$. Note that since our independent units are collections of pooled time points, $O_i = (X_i(t), Y_i(t) : t \in \tau_i)$, that all pooled samples from each i.i.d. sample, O_i will be contained within the same validation fold.

For each B^v_n, we define $\psi_{B^v_n} \equiv \hat{\Psi}(\bar{P}^0_{n,B^v_n})$, where $\bar{P}^0_{n,B^v_n}$ is the empirical distribution of the pooled data contained in the vth training set. The function, $\psi_{B^v_n}$, which is learned from the vth training set, will be used to generate predicted values for the observations in the vth validation fold. We define n^v_1 and n^v_0 to be the number of positive and negative samples in the vth validation fold, respectively. Formally, $n^v_1 = \sum_{i=1}^n \sum_{t \in \tau_i} I\left(Y_i(t) = 1\right) I\left(B^v_n(i) = 1\right)$ and $n^v_0 = \sum_{i=1}^n \sum_{t \in \tau_i} I\left(Y_i(t) = 0\right) I\left(B^v_n(i) = 1\right)$. We note that n^v_1 and n^v_0 are random variables that depend on the value of both B^v_n and $\{Y_i : B^v_n(i) = 1\}$.

The AUC for a single validation fold, $\{i : B^v_n(i) = 1\}$, for pooled repeated measures data, is

$$\overline{AUC}\left(\bar{P}^1_{n,B^v_n}, \psi_{B^v_n}\right) = \frac{1}{n^v_0 n^v_1} \sum_{i=1}^{n} \sum_{t \in \tau_i} \sum_{j=1}^{n} \sum_{s \in \tau_j} h(n, v, i, t, j, s) \tag{5.21}$$

where $h(n, v, i, t, j, s) =$

$$I\left(\psi_{B^v_n}(X_j(s)) > \psi_{B^v_n}(X_i(t))\right) I\left(Y_i(t) = 0, Y_j(s) = 1\right) I\left(B^v_n(i) = B^v_n(j) = 1\right).$$

In other words, it is the probability that, after pooling over units and time, a randomly drawn positive sample will be assigned a higher predicted value than a randomly drawn negative sample in the same validation fold by the prediction model fit using the corresponding training set.

Then the V-fold cross-validated AUC estimator, for pooled repeated measures data, is defined as

$$E_{B_n}\overline{AUC}\left(\bar{P}^1_{n,B_n}, \psi_{B_n}\right) = \frac{1}{V}\sum_{v=1}^{V}\overline{AUC}\left(\bar{P}^1_{n,B_n^v}, \psi_{B_n^v}\right) \tag{5.22}$$

$$= \frac{1}{V}\sum_{v=1}^{V}\left\{\frac{1}{n_0^v n_1^v}\sum_{i=1}^{n}\sum_{t\in\tau_i}\sum_{j=1}^{n}\sum_{s\in\tau_j} h(n,v,i,t,j,s)\right\}. \tag{5.23}$$

We also define the target, ψ_0, of the V-fold cross-validated AUC estimate as

$$E_{B_n}\overline{AUC}\left(\bar{P}_0, \psi_{B_n}\right) = \frac{1}{V}\sum_{v=1}^{V}\overline{AUC}\left(\bar{P}_0, \psi_{B_n^v}\right) \tag{5.24}$$

$$= \frac{1}{V}\sum_{v=1}^{V}\bar{P}_0\left(\psi_{B_n^v}(X_1) > \psi_{B_n^v}(X_2) \mid Y_1 = 1, Y_2 = 0\right), \tag{5.25}$$

where $(X_1, Y_1) \equiv (X_1(t), Y_1(t))$ and $(X_2, Y_2) \equiv (X_2(t), Y_2(t))$ are single time-point observations. The following theorem is the pooled repeated measures analogue to Theorem 2.

As in the i.i.d. data version, this target represents the average across validation folds of the true probability (under P_0) that a randomly sampled positive observation would be ranked higher than a randomly sampled negative observation in the same validation fold by the prediction function fit in the corresponding training set. Again, the true value of this target parameter is random – it depends on the random split of the sample into V folds and corresponding fits of the prediction function. However, it nonetheless provides a meaningful measure of the performance of the prediction function on independent data.

Theorem 3. *The efficient influence curve of $\overline{AUC}\left(\bar{P}_0, \psi\right)$, evaluated at $O_i = (X_i(t), Y_i(t)) : t \in \tau_i)$, for a nonparametric model for P_0 is given by:*

$$IC_{\overline{AUC}}\left(\bar{P}_0, \psi\right)(O_i) = \frac{1}{E_0|\tau|}\sum_{t\in\tau} IC_{AUC}\left(\bar{P}_0, \psi\right)(X_i(t), Y_i(t)),$$

where

$$\begin{aligned} IC_{AUC}&\left(\bar{P}_0, \psi\right)(X_i(t), Y_i(t)) \\ &= \frac{I(Y_i(t)=1)}{\bar{P}_0(Y=1)}\bar{P}_0\left(\psi(X) < w \mid Y(t) = 0\right)\Big|_{w=\psi(X_i(t))} \\ &+ \frac{I(Y_i(t)=0)}{\bar{P}_0(Y=0)}\bar{P}_0\left(\psi(X) > w \mid Y(t) = 1\right)\Big|_{w=\psi(X_i(t))} \\ &- \left\{\frac{I(Y_i(t)=0)}{\bar{P}_0(Y=0)} + \frac{I(Y_i(t)=1)}{\bar{P}_0(Y=1)}\right\} AUC\left(\bar{P}_0, \psi\right), \end{aligned}$$

Directly above, $(X, Y) \equiv (X(s), Y(s))$ represents a single time-point observation. For each ψ, the estimator $\overline{AUC}\left(\bar{P}_n, \psi\right)$ obtained by plugging in the pooled empirical distribution $\bar{P}_0$, is asymptotically linear with influence curve $IC_{\overline{AUC}}\left(\bar{P}_0, \psi\right)$.

Let $B_n \in \{0,1\}^n$ be a random split and let P^1_{n,B_n} and P^0_{n,B_n} be the empirical distributions of the validation $\{i : B_n(i) = 1\}$ and training set $\{i : B_n(i) = 0\}$, respectively. Let $\bar{P}^1_{n,B_n}$ be the empirical distribution of the pooled data within the validation set. We assume that B_n has only a finite number of values uniformly in n, as in k-fold cross-validation. We assume that $p = \sum_i B_n(i)/n$ is bounded away from a $\delta > 0$, with probability 1. Define the cross-validated area under the ROC curve as

$$\hat{R}(\hat{\Psi}, P_n) = E_{B_n}\overline{AUC}\left(\bar{P}^1_{n,B_n}, \hat{\Psi}(P^0_{n,B_n})\right). \tag{5.26}$$

We also define the target of this cross-validated area under the ROC curve as

$$\tilde{R}(\hat{\Psi}, P_n) = E_{B_n}\overline{AUC}\left(\bar{P}_0, \hat{\Psi}(P^0_{n,B_n})\right). \tag{5.27}$$

We assume that there exists a $\psi_1 \in \mathbf{\Psi}$ so that $P_0\left\{IC_{\overline{AUC}}\left(\bar{P}_0, \hat{\Psi}(P_n)\right) - IC_{\overline{AUC}}\left(\bar{P}_0, \psi_1\right)\right\}^2$ converges to zero in probability as $n \to \infty$. We also assume that $\sup_{\psi \in \mathbf{\Psi}} \sup_O \left|IC_{\overline{AUC}}\left(\bar{P}_0, \psi\right)(O)\right| < \infty$, where the supremum over O is over a support of P_0. Then,

$$\hat{R}(\hat{\Psi}, P_n) - \tilde{R}(\hat{\Psi}, P_n) = \frac{1}{n}\sum_{i=1}^{n} IC_{\overline{AUC}}\left(\bar{P}_0, \psi_1\right)(O_i) + o_P(1/\sqrt{n}). \tag{5.28}$$

In particular, $\sqrt{n}\left(\hat{R}(\hat{\Psi}, P_n) - \tilde{R}(\hat{\Psi}, P_n)\right)$ converges to a normal distribution with mean zero and variance, $\sigma^2 = P_0\left\{IC_{\overline{AUC}}\left(\bar{P}_0, \psi_1\right)\right\}^2$. Thus, one can construct an asymptotically 0.95-confidence interval for $\tilde{R}(\hat{\Psi}, P_n)$ given by $\hat{R}(\hat{\Psi}, P_n) \pm 1.96\frac{\sigma_n}{\sqrt{n}}$ where σ_n^2 is a consistent estimator of σ^2. A consistent estimator of σ^2 is obtained as

$$\sigma_n^2 = E_{B_n}P^1_{n,B_n}\left\{IC_{\overline{AUC}}\left(\bar{P}^1_{n,B_n}, \hat{\Psi}(P^0_{n,B_n})\right)\right\}^2. \tag{5.29}$$

Proof. This is the pooled repeated measures analogue of Theorem 2, so the proof follows the exact same format and arguments as the proof of Theorem 2. □

5.6 Software

We implemented the influence curve based confidence intervals for cross-validated AUC for i.i.d. data as well as for pooled repeated measures data, as an R package. The package, called **cvAUC** [54], has the same function interface as the popular **ROCR** package [5].

For each observation, the user provides a cross-validated predicted value, as generated by a binary prediction algorithm, and a corresponding binary class label. If the user has pooled

repeated measures data instead of i.i.d. data, then the user must also provide an id for each observation. The user must also indicate which observations belong to each cross-validation fold. To be clear, the user must provide for each observation, i:

1. The value of the outcome, Y_i.
2. The validation fold, $v \in \{1, ..., V\}$, that observation, i, is associated with.
3. The predicted probability of the outcome, $\psi(X_i)$, based on plugging in that observation's covariates, W_i, into a fit trained on the observations associated with folds: $\{1, ..., V\} \backslash v$.

The main functions of the package calculate the confidence intervals (confidence level supplied by the user; defaults to 95%) for cross-validated AUC and AUC estimates calculated using i.i.d. and pooled repeated measures training data. The package also includes utility functions to compute AUC and cross-validated AUC from a set of predicted values and associated true labels.

To provide some context to the computational efficiency of our methods, the influence curve based CV AUC variance calculation for i.i.d. data takes less than half a second to execute for a sample of 100,000 observations on a 2.3 GHz Intel Core i7 processor (package version 1.0.3). For 1 million observations, it currently takes 13 seconds. The **cvAUC** R package [54] is available on CRAN and GitHub. More information and code examples can be found in the user manual for the package.

cvAUC R code example

Below is a simple example of how to use the **cvAUC** R package. This i.i.d. data example does the following:

1. Load a data set with a binary outcome. For the i.i.d. case we use a simulated "toy" dataset of 500 observations, included with the package, of graduate admissions data.
2. Divide the indices randomly into 10 folds, stratifying by outcome. Stratification is not necessary, but is commonly performed in order to create validation folds with similar distributions. Store this information in a list called `folds`.
3. Define a function to fit a model on the training data and to generate predicted values for the observations in the validation fold, for a single iteration of the cross-validation procedure. We use a logistic regression fit.
4. Apply this function across all folds to generate predicted values for each validation fold. The concatenated version of these predicted values is stored in vector called `predictions`. The outcome vector, Y, is the `labels` argument.

The following utility functions will create stratified (by outcome) validation folds, train and test a GLM and then run the entire example.

```
# Create CV folds (stratify by outcome)
.cvFolds <- function(Y, V){
  Y0 <- split(sample(which(Y == 0)),
            rep(1:V, length = length(which(Y == 0))))
  Y1 <- split(sample(which(Y == 1)),
            rep(1:V, length = length(which(Y == 1))))
  folds <- vector("list", length = V)
  for (v in seq(V)) {folds[[v]] <- c(Y0[[v]], Y1[[v]])}
  return(folds)
}

# Train/test glm for each fold
.doFit <- function(v, folds, data){
  fit <- glm(Y ~ ., data = data[-folds[[v]],], family = "binomial")
  pred <- predict(fit, newdata = data[folds[[v]],], type = "response")
  return(pred)
}

iid_example <- function(data, V = 10){

  # Create folds
  folds <- .cvFolds(Y = data$Y, V = V)
  # CV train/predict
  predictions <- unlist(sapply(seq(V), .doFit,
                         folds = folds, data = data))
  # Re-order pred values
  predictions[unlist(folds)] <- predictions
  # Get CV AUC and confidence interval
  out <- ci.cvAUC(predictions = predictions, labels = data$Y,
             folds = folds, confidence = 0.95)
  return(out)
}
```

The following is an example which returns output from the `ci.cvAUC` function:

```
# Load a training set with a binary outcome
library(cvAUC)
data(admissions)

# Get cross-validated performance
set.seed(1)
out <- iid_example(data = admissions, V = 10)
```

The output is given as follows:

```
> out
$cvAUC
[1] 0.9046473

$se
[1] 0.01620238

$ci
[1] 0.8728913 0.9364034

$confidence
[1] 0.95
```

In the i.i.d. example above, we provided cross-validated predicted values, fold indices, and class labels (0/1) to the `ci.cvAUC` function while using a default confidence level of 95%. The cross-validated AUC is shown to be approximately 0.905, with an estimated standard error of 0.016. The corresponding 0.95% confidence interval for the CV AUC is approximately [0.873, 0.936].

5.7 Coverage probability of the confidence intervals

In this section, we present results from a simulation which demonstrates the coverage probability of our influence curve based confidence intervals as implemented in our R package, **cvAUC**. The *coverage probability* of a confidence interval is the proportion of the time, over repetitions of the identical experiment, that the interval contains the true value of interest. Our true value of interest is true cross-validated AUC, defined in equation 5.4. In the simulation below, we consider a variety of training set sizes. We show that when n is small (for example, $n = 1,000$), the coverage probability of the influence curve based confidence interval may drop below the specified rate. However, in the example below, by the time $n = 5,000$, the 95% confidence intervals achieve very good coverage (94-95%). Therefore, if you have a small sample size, bootstrapping may serve as a computationally-reasonable alternative variance estimation technique. To quantify the computational advantage of the influence curve approach, we calculate the number bootstrap replicates that are required in order to achieve 95% coverage.

Simulation to evaluate coverage probability

Let $n \times p$ represent the dimensions of our training set design matrix, $\boldsymbol{X}$. We considered training sets where $n = \{500, 1000, 5000, 10000, 20000\}$ and $p = \{10, 50, 100, 200\}$. The the number of covariates that are correlated with the outcome is fixed at 10. The remaining $p-10$

covariates are random noise. For the 10 informative covariates, we generate 100,000 points from $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, and for each these observations, we let $Y = 0$. Similarly, we generate 100,000 observations from $\mathcal{N}(\boldsymbol{\nu}, \boldsymbol{\Sigma})$ and let $Y = 1$ for all these observations. For this simulation, we let $\mu_i = 0$ and $\nu_i = 0.3$, for $i \in \{1, ..., 10\}$ and we let $\boldsymbol{\Sigma}$ represent the identity covariance matrix. These combined 200,000 observations represent our true data distribution, P_0. We note that our target parameter, true cross-validated AUC, is itself random, but that it represents a true target. We are interested in the confidence interval that contains this random target 95% of the time. The samples were generated using the `mvrnorm` function of the R package, **MASS** [6].

To calculate the coverage probability of our influence curve based confidence intervals, we generate the CV AUC and corresponding confidence intervals 5,000 times and report the proportion of times that the confidence interval contains the true CV AUC. For each iteration, we sample n points from the same distribution as our population data and use that as a training set.

We perform 10-fold cross-validation by splitting these n observations into 10 validation folds, stratifying by outcome, Y. For each validation fold, we train a Lasso-regularized logistic regression fit using the **glmnet** R package [31] using the observations from the remaining 9 folds. Using the fit model, we then generate predictions for each of the samples in the validation fold and calculate the empirical AUC. We will call this the “fold AUC.” We also calculate the “true AUC” by generating predicted values for all of the 200,000 data points in our population data and calculating the empirical AUC among this population.

This process is repeated for each of the 10 validation folds, at which point we average the fold AUCs to get the estimate for cross-validated AUC. We also average the 10 true AUCs to get the true cross-validated AUC. We then calculate a 95% confidence interval for our CV AUC estimate and note whether or not the true CV AUC falls within the confidence interval.

For each value of $p \in \{10, 50, 100, 200\}$, this process is repeated 5,000 times to obtain an estimate of the coverage probability of our confidence intervals. The coverage probability is the proportion times that the true CV AUC fell within our confidence interval. For 95% confidence intervals, we expect the coverage probability to be close to 0.95. The coverage probabilities for each training set is shown in Table 5.1.

The results of the simulation indicate that for a relatively small sample size (e.g. $n = 1,000$), the coverage probability of the confidence intervals are slightly lower (92-93%) than specified (95%). However, when $n \geq 5,000$, we have coverage between 94-95%. These simulations use just one particular data generating distribution, but the results can serve as a rough benchmark of coverage probability rates over various n.

In Table 5.2, we summarize the standard errors estimated using the influence curve based variance estimation technique, as implemented in the **cvAUC** package. For comparison, in Table 5.3 we report the standard deviation of the CV AUC estimates across the 5,000 iterations of the simulation. We see that for $n \geq 5,000$, the standard errors and standard deviations are identical, however, for smaller n, the influence curve based standard errors are slightly conservative compared to the standard deviation across the 5,000 iterations. This is

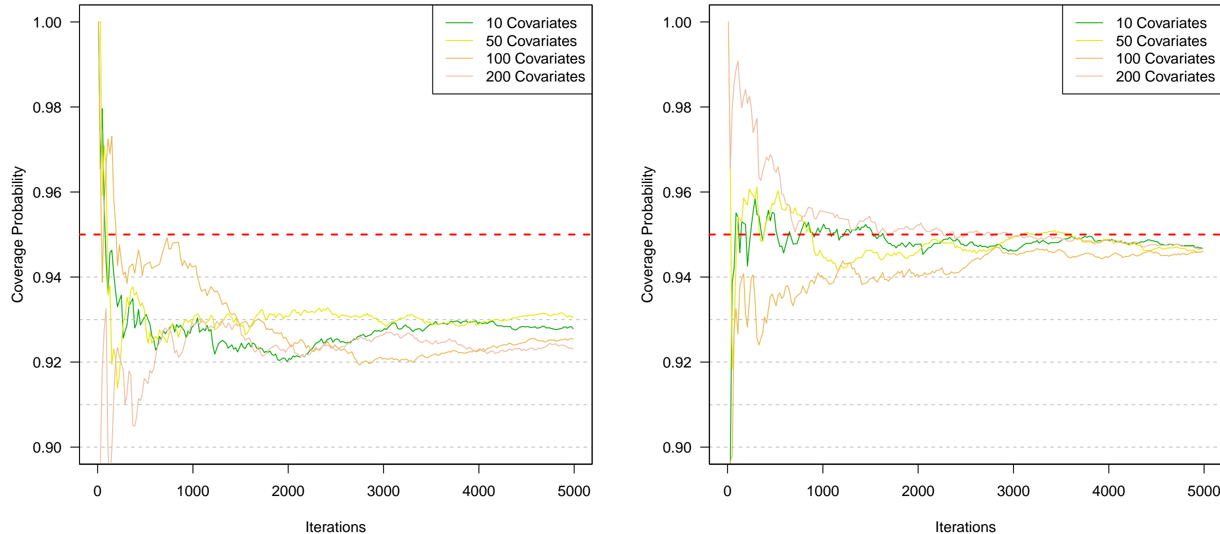

Figure 5.1: Plots of the coverage probabilities for 95% confidence intervals generated by our simulation for training sets of 1,000 (left) and 5,000 (right) observations. In the case of a 95% confidence interval, the coverage probability should be close to 0.95. For the smaller dataset of $n = 1,000$ observations, we see that the coverage is slightly lower (92-93%) than specified, whereas for $n = 5,000$, the coverage is closer to 95%.

	$n = 500$	$n = 1,000$	$n = 5,000$	$n = 10,000$	$n = 20,000$
$p = 10$	0.909	0.928	0.946	0.943	0.943
$p = 50$	0.891	0.931	0.946	0.950	0.941
$p = 100$	0.885	0.925	0.946	0.946	0.949
$p = 200$	0.878	0.923	0.947	0.937	0.940

Table 5.1: Coverage probability for influence curve based confidence intervals for CV AUC using training sets of various dimension.

expected, based on the coverage probabilities reported in Table 5.1.

For reference, we provide the average CV AUC estimate across 5,000 iterations for training sets of various dimensions in Table 5.4. A total of $20 \times 5,000 = 500,000$ cross validated AUC estimates were generated for the entire simulation. The number of individual models that were trained across all 10 folds was $500,000 \times 10 = 5$ million.

	$n = 500$	$n = 1,000$	$n = 5,000$	$n = 10,000$	$n = 20,000$
$p = 10$	0.023	0.015	0.007	0.005	0.003
$p = 50$	0.023	0.016	0.007	0.005	0.003
$p = 100$	0.024	0.016	0.007	0.005	0.003
$p = 200$	0.024	0.016	0.007	0.005	0.003

Table 5.2: Influence curve based standard errors for CV AUC for training sets of various dimensions.

	$n = 500$	$n = 1,000$	$n = 5,000$	$n = 10,000$	$n = 20,000$
$p = 10$	0.028	0.017	0.007	0.005	0.003
$p = 50$	0.033	0.018	0.007	0.005	0.003
$p = 100$	0.034	0.019	0.007	0.005	0.003
$p = 200$	0.038	0.019	0.007	0.005	0.003

Table 5.3: Standard deviation of 5,000 CV AUC estimates for training sets of various dimensions.

	$n = 500$	$n = 1,000$	$n = 5,000$	$n = 10,000$	$n = 20,000$
$p = 10$	0.720	0.737	0.747	0.748	0.748
$p = 50$	0.706	0.733	0.747	0.748	0.748
$p = 100$	0.699	0.731	0.747	0.748	0.748
$p = 200$	0.689	0.728	0.747	0.748	0.748

Table 5.4: Average CV AUC across 5,000 iterations for training sets of various dimensions.

Comparison to bootstrapped confidence intervals

We implemented quantile (or percent) bootstrapped confidence intervals in Julia [10] (version 0.0.3) to compare the coverage probability of bootstrap derived confidence intervals to influence curve derived confidence intervals. The same data generating distributions [58] as the influence curve based simulations were used, and again we used Lasso-regularized logistic regression [46]. For each iteration of the experiment, we generate an original training set and B bootstrapped replicates of the this training set. Using the B training sets, we generate B cross-validated AUC estimates [57]. We use the 0.025 and 0.975 quantiles of the B cross-validated AUCs to estimate the 95% confidence intervals. In this simulation, the computation time for bootstrapped confidence intervals is $O(B)$ times greater than the runtime of the influence curve based confidence intervals since each bootstrap replicate requires a complete re-calculation of CV AUC. Some methods of bootstrapping (e.g. m of out n bootstrap [11] and "Bag of Little Bootstraps" [1]) make computational improvements on $O(B)$, however all bootstrapping methods require generating repeated estimations of CV AUC.

On an example training set of $n = 1,000$ observations, we evaluated how many boot-

strapped replicates, B, are required to obtain 95% coverage. In this simulation, we found that at least 400 bootstrap replicates were required to obtain a coverage probability of 0.95. The coverage probabilities for increasing values of B are shown in Table 5.5.

	$B = 100$	$B = 200$	$B = 300$	$B = 400$
$p = 10$	0.906	0.930	0.929	0.958

Table 5.5: Bootstrap confidence interval coverage probability using B bootstrapped replicates of a training set of $n = 1,000$ observations.

Since the bootstrap confidence interval coverage probability estimate converged after approximately 1,000 iterations of the experiment, the coverage probability estimates in Table 5.5 are averaged over 1,000 iterations instead of 5,000.

5.8 Conclusion

Cross-validated AUC represents an attractive and commonly used measure of performance in binary classification problems. However, resampling based approaches to constructing confidence intervals for this quantity can be computationally expensive. In this chapter, we established the asymptotical linearity of the cross-validated AUC estimator and derived its influence curve for both the i.i.d. and pooled repeated measures cases. We then presented a computationally efficient approach to constructing confidence intervals based on estimating this influence curve, which is implemented as a publicly available R package called **cvAUC**. A simulation demonstrated that we were able to achieve the expected coverage probability for our confidence intervals, however, for small sample sizes, the coverage probability can dip below the desired rate. We have demonstrated a computationally efficient alternative to bootstrapping for estimating the variance of cross-validated AUC estimates. This technique for generating computationally efficient confidence intervals can be replicated for another estimator by following the same procedure.

Chapter 6

Glossary

accuracy: In binary classification problems, accuracy is the proportion of correctly classified samples divided by the total number of samples.

Area Under the ROC Curve (AUC): The AUC is the area under the receiver operator characteristic (ROC) curve. In binary classification or ranking problems, the AUC is a measure of performance which is equal to the probability that a classifier will rank a randomly chosen positive example higher than a randomly chosen negative example.

asymptotically linear: An estimator, $\hat{\Psi}(P_n)$, of $\hat{\Psi}(P_0)$, is asymptotically linear if $\hat{\Psi}(P_n) - \hat{\Psi}(P_0) = \frac{1}{n}\sum_{i=1}^{n} IC(P_0)(O_i) + o_P(1/\sqrt{n})$ with influence curve, $IC(P_0)$ of O.

average mixture (AVGM) algorithm: A procedure for estimating a parameter in a parametric model using subsets of training data. Given m different machines and a data set of size N, partition the data into samples of size $n = N/m$, and distribute n samples to each machine. Then compute the empirical minimizer on each partition of the data, and average all the parameter estimates across the machines to obtain a final estimate.

Bag of Little Bootstraps (BLB): A procedure which incorporates features of both the bootstrap and subsampling to obtain a robust, computationally efficient means of assessing estimator quality.

bagging: Bootstrap aggregating, or bagging, is an ensemble algorithm designed to improve the stability and accuracy of machine learning algorithms used in statistical classification and regression. Bagging is a special case of the model averaging approach.

base learner: A supervised machine learning algorithm (with a specific set of tuning parameters) used as part of the ensemble.

base learner library: A set of base learners that make up the ensemble.

batch learning: Any algorithm that learns by processing the entire dataset at once (which typically requires the entire dataset to fit in memory). This is the opposite of online or sequential learning.

boosting: A boosting algorithm iteratively learns weak classifiers and adds them to a final strong classifier.

classification error: In classification problems, where the aim is to correctly classify examples into one of two or more classes, the classification error is rate at which examples have been classified incorrectly.

convex combination: A convex combination is a linear combination where the coefficients are non-negative and sum to 1.

covariate space: The space, $\mathcal{X}$, that the input data, X, is sampled from. For example, $\mathcal{X}$ could be equal to $\mathbb{R}^p$ or $\{0,1\}^p$.

coverage probability: The coverage probability of a confidence interval is the proportion of the time, over repetitions of the identical experiment, that the interval contains the true value of interest.

cross-validated predicted values: Assuming n i.i.d. training examples and k-fold cross-validation where $k = V$, the cross-validated predicted values for a particular learner is the set of n predictions obtained by training on folds, $\{1, ..., V\} \setminus v$, and generating predictions on the held-out validation set, fold v. In the context of stacking, this is called the "level-one" data.

ensemble, ensemble learner: A machine learning algorithm that uses the input from multiple base learners to inform its predictions.

F_1-score: In binary classification, the F_1-score can be interpreted as the weighted average of the precision and recall.

functional delta method: A generalization of the classical delta method for finite dimensional functions of a finite set of estimators.

H-measure: The H-measure is an alternative to AUC for measuring binary classification performance which can incorporate different costs for false positives and false negatives.

H2O: An open source machine learning library with a distributed, Java-based back-end.

histogram regression: A prediction method that, given a partitioning of the covariate space into J subsets, models each subset as the average outcome among the training data within that subset.

influence function, influence curve: In the field of robust statistics, the influence function of an estimator measures the effect that one sample has on the estimate. Influence functions can be used to derive asymptotic estimates of variance.

k-fold cross-validation: In k-fold cross-validation, the data is partitioned into k folds, and then a model is trained using the observations from $k-1$ folds. Next, the model is evaluated on the held out set. This is repeated k times and estimates are averaged over the k-folds.

leave-one-out cross-validation: Similar to k-fold cross-validation, with the exception that k is equal to the number of training observations.

level-one data: The independent predictions generated from validation (typically k-fold cross-validation) of the base learners. This data is the input to the metalearner. This is also called the set of "cross-validated predicted values".

level-zero data: The original training data set which is used to train the base learners.

loss function, objective function: A loss function is a function that maps an event or values of one or more variables onto a real number intuitively representing some "cost" associated with the event. An optimization problem seeks to minimize a loss function.

mean squared error (MSE), squared error loss: The mean squared error or squared error loss of an estimator measures the average of the squares of the error, or the difference between the estimator and what is estimated.

metalearner: A supervised machine learning algorithm that is used to learn the optimal combination of the base learners. This can also be an optimization method such as non-negative least squares (NNLS), COBYLA or L-BFGS-B for finding the optimal linear combination of the base learners.

metalearning: In stacking or Super Learning, metalearning is the process of fitting a secondary learning algorithm to the level-one dataset.

negative log-likelihood loss: The negative log of the probability of the data given the model.

objective function: The objective function is either a loss function or its reciprocal inverse.

online (or sequential) learning: Online learning, as opposed to batch learning, involves using a stream of data for training examples. In online methods, the model fit is updated, or learned, incrementally.

Online Super Learner (OSL): An online implementation of the Super Learner algorithm that uses stochastic gradient descent for incremental learning.

oracle, oracle selector: The estimator, among all possible weighted combinations of the base prediction functions, which minimizes risk under the true data-generating distribution.

Partial AUC: The area under the ROC curve, defined over a restricted range of false positive rates.

rank loss: The rank loss is a name for the quantity, 1 - AUC, where "AUC" is the area under the ROC curve.

regression trees: A prediction method that recursively partitions the covariate space of the training data. The terminal nodes are modeled using linear regression.

splitting criterion: In learning a regression tree, the criterion used to determine where to split.

stacking, stacked generalization, stacked regression: Stacking is a broad class of algorithms that involves training a second-level metalearner to ensemble a group of base learners. For prediction, the Super Learner algorithm is equivalent to generalized stacking.

subsampled average mixture (SAVGM): A bias-corrected version of the AVGM algorithm with substantially better performance.

Subsemble: Subsemble is a general subset ensemble prediction method which partitions the full dataset into subsets of observations, fits a specified underlying algorithm on each subset, and uses a unique form of k-fold cross-validation to output a prediction function that combines the subset-specific fits. An oracle result provides a theoretical performance guarantee for Subsemble.

Super Learner (SL), Super Learning: Super Learner is an ensemble algorithm takes as input a library of supervised learning algorithms and a metalearning algorithm. SL uses cross-validation to data-adaptively select the best way to combine the algorithms. It is general since it can be applied to any loss function $L(\psi)$ or $L_\eta(\psi)$ (and thus corresponding risk $R_0(\psi) = E_0 L(\psi)$), or any risk function, $R_{P_0}(\psi)$. It is optimal in the sense of asymptotic equivalence with oracle selector as implied by oracle inequality.

Supervised Regression Tree (SRT) Subsemble: A practical supervised Subsemble algorithm which uses regression trees to determine both of the number of subsets, J, and the partitioning of the covariate space.

Vowpal Wabbit (VW): An open source, out-of-core, online machine learning library written in C++.

Bibliography

[1] Kleiner, A., Talwalkar, A., Sarkar, P., and Jordan, M. (2013). A scalable bootstrap for massive data. *Journal of the Royal Statistical Society, Series B*.

[2] Ling, C., Huang, J., and Zhang, H. (2003). AUC: a statistically consistent and more discriminating measure than accuracy. *Proceedings of IJCAI 2003*.

[3] Bradley, A. P. (1997). The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition 30*, 1145–1159.

[4] Geisser, S. (1975). The predictive sample reuse method with applications. *Amer. Statist. Assoc. 70*, 320–328.

[5] Sing, T., Sander, O., Beerenwinkel, N., and Lengauer, T. (2005). ROCR: Visualizing classifier performance in R. *Bioinformatics* **21**, 20, 3940–3941.

[6] Venables, W. N. and Ripley, B. D. (2002). *Modern Applied Statistics with S*, Fourth ed. Springer, New York.

[7] David M. Allen. “The relationship between variable selection and data augmentation and a method for prediction”. In: *Technometrics* 16 (1974), pp. 125–127. issn: 0040-1706.

[8] P. Baldi, P. Sadowski, and D. Whiteson. “Searching for Exotic Particles in High-energy Physics with Deep Learning”. In: *Nature Communications* 5 (2014). url: `http://archive.ics.uci.edu/ml/datasets/HIGGS`.

[9] C. J. P. Belsie. “Convergence theorems for a class of simulated annealing algorithms on Rd”. In: *Journal of Applied Probability* 29 (1992), pp. 885–892.

[10] Jeff Bezanson et al. “Julia: A Fast Dynamic Language for Technical Computing”. In: *CoRR* abs/1209.5145 (2012). url: `http://arxiv.org/abs/1209.5145`.

[11] P. J. Bickel, F. Götze, and W. R. van Zwet. “Resampling fewer than n observations: gains, losses, and remedies for losses”. In: *Statist. Sinica* 7.1 (1997). Empirical Bayes, sequential analysis and related topics in statistics and probability (New Brunswick, NJ, 1995), pp. 1–31. issn: 1017-0405.

[12] Peter J. Bickel et al. *Efficient and adaptive estimation for semiparametric models*. Johns Hopkins Series in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, 1993, pp. xxii+560. isbn: 0-8018-4541-6.

[13] Léon Bottou. "Large-scale machine learning with stochastic gradient descent". In: *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[14] Andrew P. Bradley. "The use of the area under the ROC curve in the evaluation of machine learning algorithms". In: *Pattern Recognition* 30 (1997), pp. 1145–1159.

[15] Leo Breiman. "Bagging Predictors". In: *Machine Learning* 24.2 (1996), pp. 123–140.

[16] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (2001), pp. 5–32.

[17] Leo Breiman. "Stacked Regressions". In: *Machine Learning* 24.1 (1996), pp. 49–64.

[18] L. Breiman et al. *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks, 1984.

[19] C. G. Broyden. "The convergence of a class of double-rank minimization algorithms". In: *Journal of the Institute of Mathematics and Its Applications* 6 (1970), pp. 76–90.

[20] Richard H. Byrd, Peihuang Lu, and Jorge Nocedal. *A Limited-Memory Algorithm for Bound-Constrained Optimization*. 1995.

[21] C.C. Chang and C.J. Lin. *LIBSVM: A library for Support Vector Machines*. 2001. URL: `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`.

[22] Bertrand Clarke and Bin Yu. "Comparing Bayes model averaging and stacking when model approximation error cannot be ignored". In: *Journal of Machine Learning Research*. 2003, p. 2003.

[23] Beman Dawes, David Abrahams, and Nicolai Josuttis. *Boost C++ Libraries*. URL: `http://www.boost.org`.

[24] Sandrine Dudoit and Mark J. van der Laan. "Asymptotics of cross-validated risk estimation in estimator selection and performance assessment". In: *Statistical Methodology* 2.2 (2005), pp. 131–154.

[25] Sandrine Dudoit, Mark J. van der Laan, and Aad W. van der Vaart. "The Cross-validated Adaptive Epsilon-Net Estimator". In: *Statistics and Decisions* 24.2 (2006), pp. 373–395.

[26] B. Efron. "Bootstrap methods: another look at the jackknife". In: *Ann. Statist.* 7.1 (1979), pp. 1–26. ISSN: 0090-5364.

[27] Bradley Efron and Robert J. Tibshirani. *An introduction to the bootstrap*. Vol. 57. Monographs on Statistics and Applied Probability. Chapman and Hall, New York, 1993, pp. xvi+436. ISBN: 0-412-04231-2.

[28] R. Fletcher. "A New Approach to Variable Metric Algorithms". In: *Computer Journal* 13.3 (1970), pp. 317–322.

[29] Yoav Freund and Robert E. Schapire. "A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting". In: *Journal of Computer and System Sciences* 55.1 (1997), pp. 119–139.

[30] Jerome H. Friedman. "Greedy Function Approximation: A Gradient Boosting Machine". In: *Annals of Statistics* 29 (1999), pp. 1189–1232.

[31] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. "Regularization Paths for Generalized Linear Models via Coordinate Descent". In: *Journal of Statistical Software* 33.1 (2010), pp. 1–22. URL: `http://www.jstatsoft.org/v33/i01/`.

[32] Richard D. Gill. "Non- and semi-parametric maximum likelihood estimators and the von Mises method. I". In: *Scand. J. Statist.* 16.2 (1989). With a discussion by J. A. Wellner and J. Præstgaard and a reply by the author, pp. 97–128. ISSN: 0303-6898.

[33] D. Goldfarb. "A Family of Variable Metric Updates Derived by Variational Means". In: *Mathematics of Computation* 24.109 (1970), pp. 23–26.

[34] H2O. *H2O Performance Datasheet.* 2014. URL: `http://docs.h2o.ai/resources/h2odatasheet.html`.

[35] H2O.ai. *H2O Machine Learning Platform.* version 2.9.0.1593. 2014. URL: `https://github.com/h2oai/h2o`.

[36] DavidJ. Hand and RobertJ. Till. "A Simple Generalisation of the Area Under the ROC Curve for Multiple Class Classification Problems". English. In: *Machine Learning* 45.2 (2001), pp. 171–186. ISSN: 0885-6125. DOI: `10.1023/A:1010920819831`.

[37] D.J. Hand. "Measuring classifier performance: a coherent alternative to the area under the ROC curve". In: *Machine Learning* 77 (2009), pp. 103–123.

[38] Geoffrey E. Hinton et al. "Improving neural networks by preventing co-adaptation of feature detectors". In: *CoRR* abs/1207.0580 (2012). URL: `http://arxiv.org/abs/1207.0580`.

[39] A. E. Hoerl and R. W. Kennard. "Ridge Regression: Bias estimation for nonorthogonal problems". In: *Technometrics* (1970), pp. 55–67.

[40] Arthur E. Hoerl. "Application of ridge analysis to regression problems". In: *Chemical Engineering Progress* (1958), pp. 54–59.

[41] Alan Hubbard et al. "Time-dependent prediction and evaluation of variable importance using superlearning in high-dimensional clinical data". In: *Journal of Trauma and Acute Care Surgery* 75 (2013), S53–S60.

[42] *Intel Math Kernel Library (MKL).* Intel Corporation. URL: `https://software.intel.com/en-us/intel-mkl/details`.

[43] Max Kuhn. Contributions from Jed Wing et al. *caret: Classification and Regression Training.* R package version 6.0-35. 2014. URL: `http://CRAN.R-project.org/package=caret`.

[44] Y. Jiang, C. E. Metz, and R. M. Nishikawa. "A receiver operating characteristic partial area index for highly sensitive diagnostic tests". In: *Radiology* 201 (1996), pp. 745–750.

[45] Steven G. Johnson. *The NLopt nonlinear-optimization package*. 2014. URL: `http://ab-initio.mit.edu/nlopt`.

[46] Simon Kornblith. *GLMNet.jl: Julia wrapper for fitting Lasso/ElasticNet GLM models using glmnet*. Commit version 0526df8455. 2014. URL: `https://github.com/simonster/GLMNet.jl`.

[47] Noemi Kreif, Ivan Diaz Munoz, and David Harrison. "Health econometric evaluation of the effects of a continuous treatment: a machine learning approach". In: *The Selected Works of Ivan Diaz* (2013).

[48] Mark J. van der Laan and Sherri Rose. *Targeted Learning: Causal Inference for Observational and Experimental Data*. First. Springer Series in Statistics. Springer, 2011.

[49] John Langford, Alex Strehl, and Lihong Li. *Vowpal Wabbit*. 2007. URL: `http://mloss.org/software/view/53/`.

[50] Charles L. Lawson and Richard J. Hanson. *Solving Least Squares Problems*. SIAM, 1974.

[51] Michael LeBlanc and Robert Tibshirani. *Combining Estimates in Regression and Classification*. Tech. rep. Journal of the American Statistical Association, 1993.

[52] Erin LeDell. *h2oEnsemble Benchmarks*. 2015. URL: `https://github.com/ledell/h2oEnsemble-benchmarks/releases/tag/big-data-handbook`.

[53] Erin LeDell. *h2oEnsemble: H2O Ensemble Learning*. R package version 0.0.3. 2014. URL: `https://github.com/h2oai/h2o/tree/master/R/ensemble`.

[54] Erin LeDell, Maya Petersen, and Mark van der Laan. *cvAUC: Cross-Validated Area Under the ROC Curve Confidence Intervals*. R package version 1.1.0. 2015. URL: `http://CRAN.R-project.org/package=cvAUC`.

[55] Erin LeDell, Stephanie Sapp, and Mark van der Laan. *subsemble: An Ensemble Method for Combining Subset-Specific Algorithm Fits*. R package version 0.0.9. 2014. URL: `http://CRAN.R-project.org/package=subsemble`.

[56] Samuel Lendle and Erin LeDell. *Vowpal Wabbit Ensemble: Online Super Learner*. 2013. URL: `http://www.stat.berkeley.edu/~ledell/vwsl`.

[57] Dahua Lin. *A set of functions to support the development of machine learning algorithms*. v0.4.2. 2014. URL: `https://github.com/JuliaStats/MLBase.jl`.

[58] Dahua Lin and John Myles White. *A Julia package for probability distributions and associated functions*. v0.5.4. 2014. URL: `https://github.com/JuliaStats/Distributions.jl`.

[59] Jimmy Lin and Alek Kolcz. "Large-Scale Machine Learning at Twitter". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA: ACM, 2012, pp. 793–804.

[60] Alexander R. Luedtke and Mark J. van der Laan. "Super-Learning of an Optimal Dynamic Treatment Rule". In: *U.C. Division of Biostatistics Working Paper Series* 326 (2014).

[61] Justin Ma et al. *Malicious URL Dataset (UCSD)*. 2009. URL: http://www.sysnet.ucsd.edu/projects/url/.

[62] D. K. McClish. "Analyzing a Portion of the ROC Curve". In: *Med Decis Making* 9 (1989), pp. 190–195.

[63] J. A. Nelder and R. Mead. "A simplex method for function minimization". In: *The Computer Journal* 4 (1965), pp. 308–313.

[64] Andrew Nobel. "Histogram regression estimation using data-dependent partitions". In: *The Annals of Statistics* 24.3 (June 1996), pp. 1084–1105. DOI: 10.1214/aos/1032526958. URL: http://dx.doi.org/10.1214/aos/1032526958.

[65] Maya L. Petersen et al. "Super learner analysis of electronic adherence data improves viral prediction and may provide strategies for selective HIV RNA monitoring". In: *Journal of Acquired Immune Deficiency Syndromes (JAIDS)* 69.1 (2015), pp. 109–18.

[66] Romain Pirracchio et al. "Mortality prediction in intensive care units with the Super ICU Learner Algorithm (SICULA): a population-based study". In: *Statistical Applications in Genetics and Molecular Biology* 3.1 (2015), pp. 42–52. URL: http://dx.doi.org/10.1016/S2213-2600(14)70239-5.

[67] Dimitris N. Politis, Joseph P. Romano, and Michael Wolf. *Subsampling*. Springer Series in Statistics. Springer-Verlag, New York, 1999, pp. xvi+347. ISBN: 0-387-98854-8. DOI: 10.1007/978-1-4612-1554-7. URL: http://dx.doi.org/10.1007/978-1-4612-1554-7.

[68] Eric Polley and Mark van der Laan. *SuperLearner: Super Learner Prediction*. R package version 2.0-9. 2010. URL: http://CRAN.R-project.org/package=SuperLearner.

[69] F. Provost and P. Domingos. "Well-Trained Pets: Improving Probability Estimation Trees". In: *CeDER Working Paper: IS-00-04, Stern School of Business, New York Univ.* (2000).

[70] C. J. Van Rijsbergen. *Information Retrieval*. 2nd ed. London, UK: Butterworth, 1979.

[71] Stephanie Sapp and Mark J. van der Laan. *A Scalable Supervised Subsemble Prediction Algorithm*. Tech. rep. 321. U.C. Berkeley Division of Biostatistics Working Paper Series, Apr. 2014. URL: http://biostats.bepress.com/ucbbiostat/paper321.

[72] Stephanie Sapp, Mark J. van der Laan, and John Canny. "Subsemble: an ensemble method for combining subset-specific algorithm fits". In: *Journal of Applied Statistics* 41.6 (2014), pp. 1247–1259. DOI: 10.1080/02664763.2013.864263. URL: http://www.tandfonline.com/doi/abs/10.1080/02664763.2013.864263.

[73] Ed Schmahl. *NNLS C Implementation*. 2000. URL: http://hesperia.gsfc.nasa.gov/~schmahl/nnls/nnls.c.

[74] D. F. Shanno. “Conditioning of quasi-Newton methods for function minimization”. In: *Mathematics of Computation* 24.111 (1970), pp. 647–656.

[75] Jun Shao. “Linear model selection by cross-validation”. In: *J. Amer. Statist. Assoc.* 88.422 (1993), pp. 486–494. ISSN: 0162-1459.

[76] Sandra E. Sinisi et al. “Super Learning: An Application to the Prediction of HIV-1 Drug Resistance”. In: *Statistical Applications in Genetics and Molecular Biology* 6.1 (2007). URL: `http://works.bepress.com/mark_van_der_laan/196/`.

[77] ASA Sections on Statistical Computing. *Airline Dataset (1987-2008).* URL: `http://stat-computing.org/dataexpo/2009/the-data.html`.

[78] M. Stone. “Cross-validatory choice and assessment of statistical predictions”. In: *J. Roy. Statist. Soc. Ser. B* 36 (1974). With discussion by G. A. Barnard, A. C. Atkinson, L. K. Chan, A. P. Dawid, F. Downton, J. Dickey, A. G. Baker, O. Barndorff-Nielsen, D. R. Cox, S. Giesser, D. Hinkley, R. R. Hocking, and A. S. Young, and with a reply by the authors, pp. 111–147. ISSN: 0035-9246.

[79] R. Tibshirani. “Regression shrinkage and selection via the lasso”. In: *Journal of the Royal Statistical Society. Series B* 58.1 (1996), pp. 267–288.

[80] Luke Tierney. *Simple Network Of Workstations for R (SNOW).* URL: `http://homepage.stat.uiowa.edu/~luke/R/cluster/cluster.html`.

[81] Aad W. van der Vaart and Jon A. Wellner. *Weak convergence and empirical processes.* Springer Series in Statistics. With applications to statistics. Springer-Verlag, New York, 1996, pp. xvi+508. ISBN: 0-387-94640-3.

[82] Mark J. van der Laan, Sandrine Dudoit, and Aad W. van der Vaart. “The Cross-Validated Adaptive Epsilon-Net Estimator”. In: *Statistics and Decisions* 24.3 (2006), pp. 373–395.

[83] Mark J. van der Laan and Eric C. Polley. “Super Learner in Prediction”. In: *U.C. Berkeley Division of Biostatistics Working Paper Series* 266 (2010). URL: `http://biostats.bepress.com/ucbbiostat/paper266`.

[84] Mark J. van der Laan, Eric C. Polley, and Alan E. Hubbard. “Super Learner”. In: *Statistical Applications in Genetics and Molecular Biology* 6.1 (2007).

[85] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. *Automatically Tuned Linear Algebra Software (ATLAS).* 27, pp. 3–35. URL: `https://en.wikipedia.org/wiki/Automatically_Tuned_Linear_Algebra_Software`.

[86] David H. Wolpert. “Stacked Generalization”. In: *Neural Networks* 5.2 (1992), pp. 241–259.

[87] K. Woods et al. “Comparative Evaluation of Pattern Recognition Techniques for Detection of Microcalcifications in Mammography”. In: *International J. Pattern Recognition and Artificial Intelligence* 7.6 (1993), pp. 1417–1436.

[88] Zhang Xianyi, Wang Qian, and Werner Saar. *OpenBLAS*. 2015. URL: http://www.openblas.net/.

[89] Jelmer Ypma, Hans W. Borchers, and Dirk Eddelbuettel. *R interface to NLopt*. version 1.0.4. 2014. URL: http://cran.r-project.org/web/packages/nloptr/index.html.

[90] Yuchen Zhang, John C. Duchi, and Martin J. Wainwright. "Communication-Efficient Algorithms for Statistical Optimization". In: *Journal of Machine Learning Research* 14.1 (2013), pp. 3321–3363.

[91] W. Zheng and M. J. van der Laan. *Targeted Maximum Likelihood Estimation of Natural Direct Effect*. Tech. rep. 288. U.C. Berkeley Division of Biostatistics Working Paper Series, 2011.

[92] Ciyou Zhu et al. "Algorithm 778: L-BFGS-B: Fortran Subroutines for Large-scale Bound-constrained Optimization". In: *ACM Trans. Math. Softw.* 23.4 (Dec. 1997), pp. 550–560. ISSN: 0098-3500. DOI: 10.1145/279232.279236. URL: http://doi.acm.org/10.1145/279232.279236.