Subsemble: A Flexible Subset Ensemble Prediction Method

by

Stephanie Karen Sapp

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Statistics

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, Chair
Professor Nicholas Jewell
Professor John Canny

Spring 2014

Subsemble: A Flexible Subset Ensemble Prediction Method

Abstract

Subsemble: A Flexible Subset Ensemble Prediction Method

by

Stephanie Karen Sapp

Doctor of Philosophy in Statistics

University of California, Berkeley

Professor Mark van der Laan, Chair

Ensemble methods using the same underlying algorithm trained on different subsets of observations have recently received increased attention as practical prediction tools for massive data sets. We propose Subsemble, a general subset ensemble prediction method, which can be used for small, moderate, or large data sets. Subsemble partitions the full data set into subsets of observations, fits one or more user-specified underlying algorithm on each subset, and uses a clever form of V-fold cross-validation to output a prediction function that combines the subset-specific fits through a second user-specified metalearner algorithm. We give an oracle result that provides a theoretical performance guarantee for Subsemble. Through simulations, we demonstrate that Subsembles with randomly created subsets can be beneficial tools for small to moderate sized data sets, and often have better prediction performance than the same underlying algorithm fit just once on the full data set. We also describe how to include Subsembles as candidates in a SuperLearner library, providing a practical way to evaluate the performance of Subsembles relative to the same underlying algorithm fit just once on the full data set.

Since the final Subsemble estimator varies depending on the data within each subset, different strategies for creating the subsets used in Subsemble result in different Subsembles, which in turn have different prediction performance. To study the effect of subset creation strategies, we propose supervised partitioning of the covariate space to learn the subsets used in Subsemble. We highlight computational advantages of this approach, discuss applications to large-scale data sets, and develop a practical Supervised Subsemble method using regression trees to both create the covariate space partitioning, and select the number of subsets used in Subsemble. Through simulations and real data analysis, we show that this subset creation method can provide better prediction performance than the random subset version.

Finally, we develop the `R` package **subsemble** to make the Subsemble method readily available to both researchers and practitioners. We describe the `subsemble` function, discuss implementation details, and illustrate application of the Subsemble algorithm for prediction with **subsemble** through an example data set.

To Karen L. Nelson

Contents

List of Figures

List of Tables

Acknowledgments

The author wishes to acknowledge several individuals and groups whose support made this work possible. Special thanks to committee chair Professor Mark van der Laan for his advice and guidance. Thanks also to committee members Professor Nicholas Jewell and Professor John Canny for their assistance and input. Additional thanks to Google's Quantitative Marketing Team for providing industry experience and inspiration from real-world applications. Finally, sincere thanks to family and friends, especially Karen L. Nelson and Ming K. Yuen, for their invaluable encouragement and support.

Chapter 1

Introduction

Procedures using subsets of observations from a full available data set are promising tools for prediction with large-scale data sets. By operating on subsets of observations, computations can be parallelized, taking advantage of modern computational resources. Much recent research work has focused on proposing and evaluating the performance of different subset prediction procedures.

Subset prediction procedures create subsets of the full available data set, train the same underlying algorithm on each subset, and finally combine the results across the subsets. The method used to obtain the subsets, and the method used to combine the subset-specific results, differ depending on the procedure.

A classic subset prediction method is bagging, or bootstrap aggregating, developed in Breiman 1996a. In bagging, one subsamples a large number of fixed size bootstrap samples, and fits the same prediction algorithm on each bootstrap sample. The final prediction function is given by the simple average of the subset-specific fits. This approach has several drawbacks. First, some observations will never be used, while others will be selected multiple times. Second, taking a simple average of the subset fits does not differentiate between the quality of each fit.

Recently, Zhang, Duchi, and Wainwright 2013 proposed two subset methods for estimating the parameter of a parametric prediction model: an average mixture (AVGM) procedure, and a bootstrap average mixture (BAVGM) procedure. Both procedures first partition the full data set into disjoint subsets, and estimate the parameter of interest within each subset. To obtain the final parameter estimate, AVGM takes a simple average of the subset-specific estimates. BAVGM first draws a single bootstrap sample from each partition, re-estimates the parameter on the bootstrap sample, and combines the two estimates into a so-called bootstrap bias corrected estimate. To obtain the final parameter estimate, BAVGM takes a simple average of the subset-specific bootstrap bias-corrected estimates. The AVGM and BAVGM procedures have shortcomings. The approaches are only designed for parametric models, and the theoretical results provided rely on using parametric models. AVGM does not account for fit quality differences at all, since it simply averages the subset fits. BAVGM's approach to bias correction estimates the bias of a partition's parameter estimate by reusing data that was already used in the fit of that parameter. Finally, both methods are only proposed for use with large data sets. That is, the methods are proposed due to their computational attractiveness, rather than their statistical performance.

Another recent classification method using subsets was discussed in Lin and Kolcz 2012. This case study explored using subsets of observations to train classification algorithms, and combining the results linearly. The authors mention the possibility of weighting each classifier if different underlying algorithms are used, but recommend simple averaging if the same underlying classifier is trained on different subsets of observations. As their work is a case study, no theoretical performance guarantees are provided. Furthermore, the approach is only evaluated for a single algorithm (logistic regression), with a single data set, using very large subsets. Finally, the method is again only proposed by the authors for use with large data sets.

While not a subset method, boosting, formulated by Freund and Schapire 1997, is an example of an ensemble methods that differentiates between the quality of each fit. 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. Another drawback of boosting is that it is a sequential algorithm, and hence cannot be parallelized.

Another ensemble method that differentiates between the quality of each fit, but is not a subset method, is the SuperLearner method of van der Laan, Polley, and Hubbard 2007, which generalizes the stacking algorithms developed by Wolpert 1992 and extended by Breiman 1996b. SuperLearner learns the optimal weighted combination of a library of candidate learner algorithms by using cross-validation. SuperLearner generalizes stacking by allowing for general loss functions and hence a broader range of estimator combinations. Like boosting, SuperLearner is not a subset method because the ensemble combines fits of the candidate algorithms trained on the full data set. As with boosting, training on the full data set cannot be parallelized.

A key drawback of each of the above subset methods is that they do not differentiate between the quality of each subset-specific fit. To address this, we propose a novel ensemble method, Subsemble, for combining results from fitting the same underlying algorithm on different subsets of observations. Subsemble is a general subset ensemble prediction algorithm that partitions a full data set into subsets of observations, fits one or more underlying algorithm on each subset, and combines the subset-specific fits through a second metalearner algorithm using a clever form of V-fold cross-validation, thus accounting for the quality of each subset-specific fit.

The remainder of this work is organized as follows. In Chapter 2, we introduce the Subsemble procedure and study its statistical performance. In Chapter 3, we propose creating the subsets used in Subsemble through supervised partitioning of the covariate space. In Chapter 4, we describe the R package **subsemble** and demonstrate its usage for prediction applications. Finally, we conclude and discuss future research directions in Chapter 5.

Chapter 2

The Subsemble Method

2.1 Introduction

In this chapter, we introduce the Subsemble method. Subsemble ensembles together fits of the same underlying algorithm on different subsets of observations, while also accounting for the quality of each subset-specific fit. Our approach has many benefits and differs from existing methods in a variety of ways. 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 and learn the weighted combination. Finally, Subsemble has desirable statistical performance and can improve prediction quality on both small and large data sets.

This chapter focuses on the statistical performance of Subsemble. We provide an oracle result for Subsemble, showing that Subsemble performs as well as the best possible combination of the subset-specific fits. We describe how to choose between Subsemble and the underlying algorithm fit just once on the full data set, resulting in a weighted combination of the procedures. Through simulation studies, we demonstrate the desirable performance of Subsemble as a prediction procedure for moderate sized data sets. We show that Subsembles with randomly created subsets often provide better prediction performance than fitting the same underlying algorithm only once on the full available data set, and that including both the usual and Subsemble versions of algorithms in a SuperLearner library provides superior results to including only the usual versions of algorithms.

The remainder of this chapter is organized as follows. Subsemble is presented in Section 2.2. We describe how to choose between fitting an algorithm just once on the full data set versus various Subsemble fits, through including both the Subsemble and usual versions of the algorithm as candidates in a SuperLearner library, in Section 2.3. Simulation study and real data analysis results appear in Section 2.4. We summarize and conclude in Section 2.5.

2.2 Subsemble

2.2.1 The Subsemble Algorithm

Assume the full data set consists of n independent and identically distributed observations $O_i = (X_i, Y_i)$ of $O \sim P_0$. Our goal is to predict the outcome Y_i given the covariate vector X_i. Given an algorithm $\hat{\Psi}$, which is a mapping from an empirical probability distribution P_n into the parameter space space Ψ of functions of X, the usual approach to prediction using $\hat{\Psi}$ applies $\hat{\Psi}$ to the empirical distribution P_n, resulting in the estimator $\hat{\Psi}(P_n)$.

The Subsemble procedure takes a different approach to forming a prediction function using $\hat{\Psi}$. Instead of using the entire data set to obtain a single fit of $\hat{\Psi}$, Subsemble applies $\hat{\Psi}$ to multiple empirical distributions, each consisting of a subset of the available observations, created from a partitioning of the entire data set into J disjoint subsets. We refer to these J subsets of the entire data set at the *final subsets*. Subsemble then obtains the optimal combination of the final subset-specific fits by minimizing cross-validated risk through V-fold cross-validation.

Note that the cross-validation within Subsemble is used as an estimator selection tool. It is used to find the best combination of subset-specific fits by minimizing cross-validated risk. Risk estimates are based on obtaining subset-specific fits on cross-validation training sets, and estimating risk using the corresponding test sets. For this procedure to yield accurate risk estimates, the jth subset-specific estimator in the cross-validation training sets needs to be similar to the final jth subset-specific estimator of the full data set. Otherwise, the risk estimate of the jth estimator does not reflect its true risk, and the resulting combination of the J estimators is also meaningless.

The jth estimator is defined as applying the underlying algorithm $\hat{\Psi}$ to the jth final subset. In fact, the only difference between the J estimators is the particular data used to train the underlying algorithm. We thus need to define the jth estimator in the cross-validation training sets to be very similar to the jth final estimator. This is accomplished by using very similar data in the jth cross-validation and final subsets.

To motivate the construction of the V folds used in Subsemble, consider randomly splitting the entire data set into V folds. Now, suppose that at each cross-validation step, the training data were randomly assigned to the J subsets. With this approach, the data used in subset j in a cross-validation training set has no relationship to the data used in the final subset j. A partial solution would be, at each cross-validation step, to assign the training data to subsets based on each observation's assignment in the final subsets. This construction guarantees that each observation used in the subset-specific fit j during cross-validation is contained in the data used in the final subset-specific fit j. However, undefined estimates could occur if all data in the final subset j happened to fall in the same fold v.

Subsemble instead selects the V folds to preserve the subset structure: we first partition each subset j into V folds, and then create the overall vth fold by combining the vth folds from all the J subsets. This cross-validation approach has several benefits. First, very similar data is used in the cross-validation subset assignments and the final subset assignments. Second, since only $1/V$ of each final subset is left out at each cross-validation step, the potential problem of undefined estimates in the cross-validation steps is avoided. Finally, creating the cross-validation training sets does not require combining data across the subsets. This is due to the fact that, since the final subsets are partitioned into V folds, and the subset assignments in the cross-validation steps are the same as the final subset assignments, leaving a fold v out of subset j produces all the data assigned to the jth subset in the cross-validation training set. See Figure 2.1 for an illustration.

Subsemble also requires specifying a second metalearner algorithm $\hat{\Phi}$ to be used for

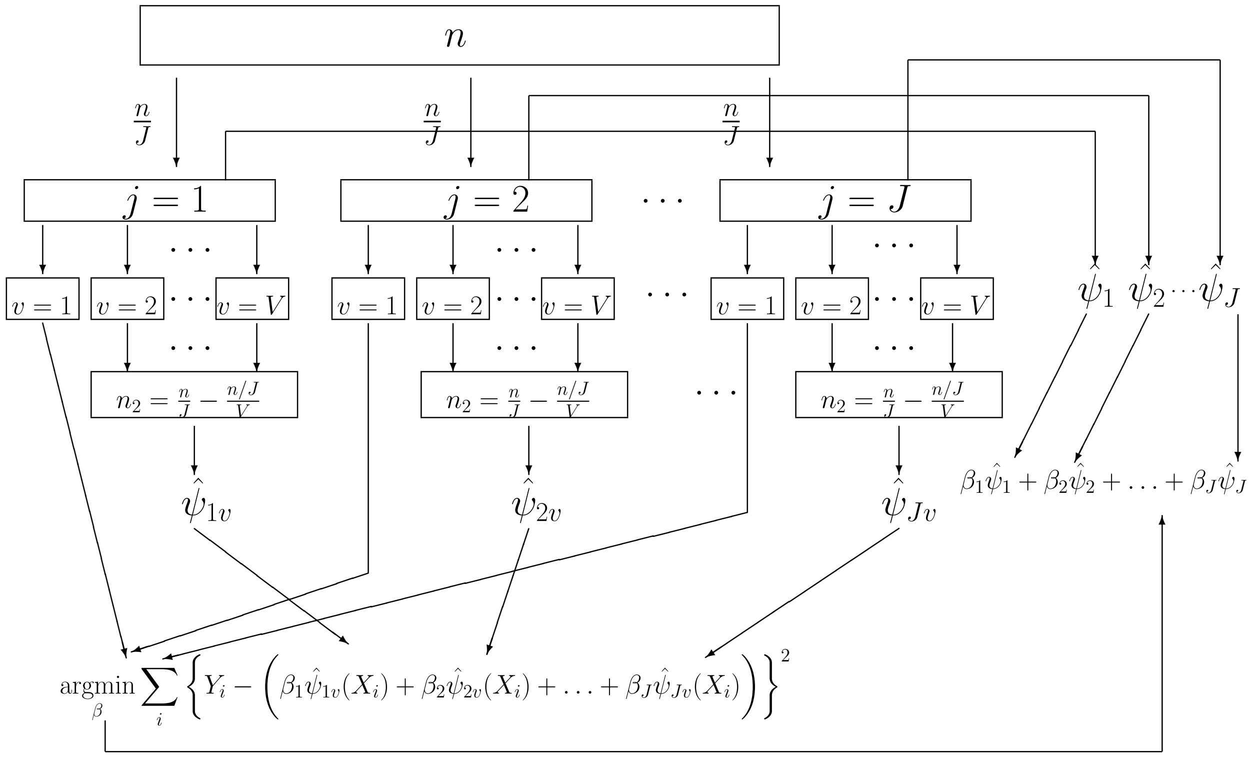
n
$\frac{n}{J}$
$\frac{n}{J}$
$\frac{n}{J}$
$j = 1$
$j = 2$
$j = J$
$v = 1$
$v = 2$
$v = V$
$v = 1$
$v = 2$
$v = V$
$v = 1$
$v = 2$
$v = V$
$n_2 = \frac{n}{J} - \frac{n/J}{V}$
$n_2 = \frac{n}{J} - \frac{n/J}{V}$
$n_2 = \frac{n}{J} - \frac{n/J}{V}$
$\hat{\psi}_{1v}$
$\hat{\psi}_{2v}$
$\hat{\psi}_{Jv}$
$\hat{\psi}_1 \; \hat{\psi}_2 \cdots \hat{\psi}_J$
$\beta_1\hat{\psi}_1 + \beta_2\hat{\psi}_2 + \ldots + \beta_J\hat{\psi}_J$
$\underset{\beta}{\text{argmin}} \sum_i \left\{ Y_i - \left(\beta_1\hat{\psi}_{1v}(X_i) + \beta_2\hat{\psi}_{2v}(X_i) + \ldots + \beta_J\hat{\psi}_{Jv}(X_i) \right) \right\}^2$

Figure 2.1: Diagram of the Subsemble procedure using linear regression as the metalearner to combine the subset-specific fits. The full data set, consisting of n observations is partitioned into J disjoint subsets. The same underlying algorithm $\hat{\psi}$ is applied to each subset, resulting in J subset-specific fits $\hat{\psi}_1, \hat{\psi}_2, \ldots, \hat{\psi}_J$. V-fold cross-validation, where the V folds are constructed to preserve the subset structure, is used to learn the best weighted linear combination of the subset-specific fits.

combining the subset-specific fits. For example, the metalearner algorithm $\hat{\Phi}$ could be a linear regression, random forest, or support vector machine. Figure 2.1 shows the Subsemble procedure when metalearner $\hat{\Phi}$ is specified as linear regression.

Pseudocode for the Subsemble algorithm is shown in Figure 2.2. More formally, Subsemble proceeds as follows. Given the user-specified number of subsets J, the n observations are partitioned into J disjoint subsets. Define the algorithm $\hat{\Psi}_j$ as $\hat{\Psi}$ applied to the jth subset. Each of the J algorithms $\hat{\Psi}_j$ are applied to P_n, resulting in J subset-specific estimators $\hat{\Psi}_j(P_n)$. V-fold cross-validation is then used to select the optimal combination of the subset-specific fits based on minimizing the cross-validated risk.

```
Algorithm 1: Subsemble
input :
  • $n$ observations $(X_i,\ Y_i)$
  • partitioning of the $n$ observations into $J$ disjoint subsets
  • underlying learner algorithm $\hat{\Psi}$
  • metalearner algorithm $\hat{\Phi}$
output: optimal combination $\hat{\Phi}^*(\hat{\Psi}_1, \ldots, \hat{\Psi}_J)$

for $j \leftarrow 1 : J$ do
    // create subset-specific learner fits
    $\hat{\Psi}_j \leftarrow$ apply $\hat{\Psi}$ to observations $i$ such that $i \in j$
    // create V folds
    randomly partition each subset $j$ into $V$ folds
end

// cross-validation
for $v \leftarrow 1 : V$ do
    // CV fits
    $\hat{\Psi}_{j,-v} \leftarrow$ apply $\hat{\Psi}$ to observations $i$ such that $i \in j$, $i \notin v$
    for $i : i \in v$ do
        // predicted values
        $\tilde{X}_i \leftarrow \big(\hat{\Psi}_{1,-v}(X_i), \ldots, \hat{\Psi}_{J,-v}(X_i)\big)$
    end
end

$\hat{\Phi}^* \leftarrow$ apply $\hat{\Phi}$ to training data $(Y_i, \tilde{X}_i)$
$\hat{\Phi}^*(\hat{\Psi}_1, \ldots, \hat{\Psi}_J) \leftarrow$ final prediction function
```

Figure 2.2: Pseudocode for the Subsemble algorithm.

The V folds are selected as follows. Each subset $j = 1, \ldots, J$ is first partitioned into V folds. Each full fold v is then obtained by combining the vth folds across the J subsets. 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 metalearner algorithm $\hat{\Phi}$ to the following redefined set of n observations: $(\tilde{X}_i, Y_i)$, where $\tilde{X}_i = \{\hat{\Psi}_j(P_{n,v(i)})(X_i)\}_{j=1}^J$. That is, for each i, the redefined input vector $\tilde{X}_i$ consists of the J predicted values obtained by evaluating the J subset-specific estimators trained on the data excluding the $v(i)$th fold, at X_i. As an example, specifying $\hat{\Phi}$ as linear regression would result in selecting the best linear combination $\sum_{j=1}^J \beta_j \hat{\Psi}_j$ of the subset-specific fits, by regressing Y_i onto the J values of $\hat{\Psi}_j(P_{n,v(i)})(X_i)$.

While this chapter primarily discusses Subsembles which combine different subset-specific fits of a single underlying algorithm, the procedure can also readily accommodate multiple underlying algorithms. To illustrate this point, instead of a single underlying algorithm $\hat{\Psi}$, consider L underlying algorithms $\hat{\Psi}^1, \ldots, \hat{\Psi}^L$. Then, instead of finding the optimal $\sum_{j=1}^J \beta_j \hat{\Psi}_j$, for example, the Subsemble procedure can be used to find the optimal $\sum_{j=1}^J \sum_{\ell=1}^L \beta_{\ell,j} \hat{\Psi}_j^\ell$, where $\hat{\Psi}_j^\ell$ denotes the fit of the ℓth algorithm $\hat{\Psi}^\ell$ on the jth subset.

2.2.2 Theoretical Performance Guarantee for Subsemble

The following oracle result, following directly from the work of van der Laan, Polley, and Hubbard 2007, gives a theoretical guarantee of Subsemble's performance.

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 $P(\hat{\Psi}(P_n) \in \mathbf{Y}) = 1$.*

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 van der Laan, *Dudoit, and van der Vaart 2006) 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* $\hat{\Phi}_{\tilde{\beta}_n}$*:*

$$\frac{E\frac{1}{V}\sum_{v=1}^{V} E_0\left[\left\{E_0[Y|X] - \hat{\Phi}_{\beta_n}(P_{n,v})\right\}^2\right]}{E\frac{1}{V}\sum_{v=1}^{V} E_0\left[\left\{E_0[Y|X] - \hat{\Phi}_{\tilde{\beta}_n}(P_{n,v})\right\}^2\right]} \rightarrow 1 \text{ as } n \rightarrow \infty$$

Otherwise, the cross-validation estimator $\hat{\Phi}_{\beta_n}$ *achieves a near parametric* $\frac{\log n}{n}$ *rate:*

$$E\frac{1}{V}\sum_{v=1}^{V} E_0\left[\left\{E_0[Y|X] - \hat{\Phi}_{\beta_n}(P_{n,v})\right\}^2\right] = O\left(\frac{\log n}{n}\right)$$

The results of this Theorem hold even if the number of subsets J grows at up to a polynomial rate in n.

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. As a result, the main lesson from this Theorem is, since usually the underlying algorithm used won't convergence at parametric rate, Subsemble performs as well as the best possible combination of subset-specific fits. That is, since their ratio of risk differences converges to one, the Subsemble not only has the same rate of convergence as the oracle procedure, as well as the same constant. Our result is even stronger: the risk difference of the Subsemble is literally asymptotically indistinguishable from that of the oracle procedure.

Note that Theorem 1 doesn't tell us how many subsets are best, or how Subsemble's combination of many subset-specific fits will perform relative to fitting the single algorithm $\hat{\Psi}$ just once on the full available data set. In Section 2.3, we provide a practical way to select between Subsemble and a single fit of the same underlying algorithm on the full data set, and also to select among different types of Subsembles. We further show through simulations in Section 2.4 that there is often a range of subsets which are better than the full fit.

2.3 Learning When to Use Subsembles

2.3.1 Subsembles as Candidates in SuperLearner

While the oracle result for Subsemble given in Section 2.2.2 provides a theoretical basis for the performance of Subsemble, it doesn't tell us whether or not Subsemble will outperform the standard single fit of an algorithm only once on the entire data set. The oracle result also provides no guidance about the best number of partitions to use in Subsemble. Here, we provide a practical approach to select between these options, describing how to include Subsembles with different numbers of subsets, as well as the usual version of the specified algorithm, as candidate algorithms in a SuperLearner library.

SuperLearner, developed in van der Laan, Polley, and Hubbard 2007, is a powerful prediction algorithm that finds the optimal weighted combination of a set of candidate prediction algorithms by minimizing cross-validated risk. SuperLearner generalizes stacking algorithms developed by Wolpert 1992 and extended by Breiman 1996b, and was named based on the theoretical performance results discussed in van der Laan, Polley, and Hubbard 2007. SuperLearner extends this prior work by allowing for general loss functions, thus allowing for different parametric combinations of estimators.

SuperLearner takes as input a library of K prediction algorithms, as well as another cross-validated risk predictor algorithm $\hat{\Theta}$, and outputs the optimal weighted combination, through $\hat{\Theta}$, of the K algorithms fit on the full data set. To select the optimal weights, SuperLearner uses V-fold cross-validation. As an example, with $\hat{\Theta}$ specified as linear regression, SuperLearner selects the optimal linear combination of the K candidate predictor algorithms. The current implementation of SuperLearner uses non-negative linear regression for $\hat{\Theta}$.

The SuperLearner algorithm proceeds as follows. Propose a library of K candidate prediction algorithms. Split the data set into V blocks of equal size. For each block $v = 1, \ldots, V$, fit each of the K candidate algorithms on the observations not in the vth block, and obtain K predictions for each observation in the vth block using these fits. Select the optimal combination by applying the user-specified minimum cross-validated risk predictor algorithm $\hat{\Theta}$: regressing the true outcome of the n observations on the K predictions to obtain a combination of the K algorithms. Finally, fit the K algorithms on the complete data set. Predictions are then obtained by using these final fits combined as specified by $\hat{\Theta}$ obtained in the previous step. For additional details, we refer the reader to van der Laan, Polley, and Hubbard 2007.

SuperLearner can be used to evaluate between Subsembles using different number of subsets, and underlying algorithms fit just once on the entire data set. Simply include Subsembles as candidates in a SuperLearner library, as well as the underlying algorithms fit once on all data as other candidates. The SuperLearner will then learn the optimal weighted combination of these candidates.

2.3.2 Oracle Result for SuperLearner

The SuperLearner algorithm has its own oracle result. As developed in van der Laan, Polley, and Hubbard 2007, we have the following Theorem.

Theorem 2. *Assume the minimum cross-validated risk predictor algorithm $\hat{\Theta} = \hat{\Theta}_\alpha$ is indexed by a finite dimensional parameter $\alpha \in \mathbf{A}$. Let K be the total number of algorithms included in the SuperLearner library, including both full and Subsemble versions. Let $\mathbf{A}_n$ be a finite set of values in $\mathbf{A}$, 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 $P(\hat{\Psi}_k(P_n) \in \mathbf{Y}) = 1$.*

Define the cross-validation selector of α as

$$\alpha_n = \arg\min_{\alpha \in \mathbf{A}_n} \sum_{i=1}^{n} \left\{ Y_i - \hat{\Theta}_\alpha(\tilde{X}_i) \right\}^2$$

and define the oracle selector of α as

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

Then, for every $\delta > 0$, there exists a constant $C(\delta) < \infty$ (defined in van der Laan, Dudoit, and van der Vaart 2006) such that

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

As a result, if none of the learners included in the library converge at a parametric rate, then the oracle selector does not converge at a parametric rate, and the cross-validation estimator $\hat{\Theta}_{\alpha_n}$ is asymptotically equivalent with the oracle estimator $\hat{\Theta}_{\tilde{\alpha}_n}$:

$$\frac{E\frac{1}{V}\sum_{v=1}^{V} E_0\left[\left\{ E_0[Y|X] - \hat{\Theta}_{\alpha_n}(P_{n,v}) \right\}^2 \right]}{E\frac{1}{V}\sum_{v=1}^{V} E_0\left[\left\{ E_0[Y|X] - \hat{\Theta}_{\tilde{\alpha}_n}(P_{n,v}) \right\}^2 \right]} \rightarrow 1 \text{ as } n \rightarrow \infty$$

Otherwise, the cross-validation estimator $\hat{\Theta}_{\alpha_n}$ achieves a near parametric $\frac{\log n}{n}$ rate.

$$E\frac{1}{V} \sum_{v=1}^{V} E_0\left[\left\{ E_0[Y|X] - \hat{\Theta}_{\alpha_n}(P_{n,v}) \right\}^2 \right] = O\left(\frac{\log n}{n} \right)$$

The results of this Theorem hold even if the number of algorithms K grows at up to a polynomial rate in n.

Similar to the oracle result for Subsemble, Theorem 2 tells us that the risk difference, based on squared-error loss, of the SuperLearner from the true $E_0[Y|X]$ can be bounded from above by a function of the risk difference of the oracle procedure. The oracle procedure results in the best possible combination of the candidate algorithms, since the oracle procedure chooses α to minimize the *true* risk difference. Typically, none of the candidate algorithms will converge at a parametric rate. As a result, SuperLearner will perform as well as best possible combination of candidates. That is, since their ratio of risk differences converges to one, the risk difference of the SuperLearner is literally asymptotically indistinguishable from that of the oracle procedure.

2.4 Data Analysis

2.4.1 Description of Data Sets

The oracle results of Theorems 1 and 2 show the benefits of Subsemble for large sized data sets. In this section, we investigate Subsemble's statistical performance for small to moderate sized samples.

In the studies that follow, we used four small to moderate sized data sets (Simulated 1, Simulated 2, Yacht, Diamond) to evaluate the practical performance of Subsemble. All data sets have one real-valued output variable, and no missing values.

The first two data sets are simulated, and generated as below. The Sim 1 data set has 20 input variables. The sim 2 data set has 200 input variables.

Sim 1:

$$\begin{aligned}
X_i &\sim N(0,9),\ i=1,\ldots,20 \\
\epsilon &\sim N(0,9) \\
Y &= \epsilon + X_1 + \sin(X_2) + \log(|X_3|) + X_4^2 + X_5X_6 + I(X_7X_8X_9 < 0) + I(X_{10} > 0) \\
&\quad +X_{11}I(X_{11} > 0) + \sqrt{|X_{12}|} + \cos(X_{13}) + 2X_{14} + |X_{15}| + I(X_{16} < -1) \\
&\quad +X_{17}I(X_{17} < -1) - 2X_{18} - X_{19}X_{20}
\end{aligned}$$

Sim 2:

$$\begin{aligned}
X_i &\sim N(0,16),\ i=1,\ldots,200 \\
\epsilon &\sim N(0,25) \\
Y &= -1 + \epsilon + \sum_{i=1}^{200} \log(|X_i|)
\end{aligned}$$

The second two data sets are publicly available real-world data. The yacht data set, available from Bache and Lichman 2013, has 308 observations and 6 input variables. The diamond data set, described by Chu 2001, has 308 observations and 17 input variables.

2.4.2 Subsemble Performance Comparison

In this study, we compare the performance of Subsemble with two alternatives: fitting the underlying algorithm just once on all data, and a naive subset method which simply averages the same subset-specific fits used in the Subsemble instead of learning a weighted combination.

We used four underlying algorithms: linear regression, lasso, regression tree, and random forest. We selected these algorithms because they are well-known and commonly used methods, and examples that cover a range of algorithm properties: both adaptive (regression tree) and non-adaptive (linear regression) methods, as well as regularized (lasso) and ensemble (random forest) versions. Note that these algorithms merely serve for the purpose of demonstration, as the oracle results given in Theorems 1 and 2 show that using more algorithms, and particularly more diverse algorithms, will result in even better statistical performance.

For each of the four algorithms, we first fit the algorithm just once on the training set (the 'Full' fit). We then divided the training set into 2, 3, 4, and 5 randomly created subsets. For each subset division, we fit each of the four algorithms on the subsets, and combined the results across the subsets in two ways: naive simple averaging across the subset-specific fits, and the Subsemble procedure with linear regression as the metalearner.

Tuning details of the underlying algorithms were defaults and set as follows. For lasso, we used 10-fold cross-validation to select the regularization weight among a grid of 100 possible values, ranging from very close to zero up to a maximum of the smallest data derived weight for which all coefficients were zero. For regression tree, we set 20 observations as the minimum needed for a split to be attempted, the minimum number of observations in any leaf node to 7, maximum depth to 30, the ANOVA between-groups sum-of-squares metric to measure and select the best covariate split, minimum R-squared increase at each step to 0.01, and 10-fold cross-validation for pruning. For random forest, we used 1000 trees, with the same parameters used for each tree in the forest: one third the number of variables as candidates randomly selected among for each split, and minimum number of terminal nodes as 5.

For the simulated data sets, we simulated training sets of 1,000 observations and test sets of 10,000 observations, and repeated the experiment 10 times. For the real data sets, we split the data sets into 10 folds, and let each fold serve as the test set. Mean Squared Prediction Error (MSPE) results were averaged across the 10 trials for both simulated and real data sets. We also performed a t-test for the difference in means between each subset method (naive and Subsemble, for each number of subsets) and the 'Full' fit. Results are presented in Table 2.1.

With simulated data set 1, for both linear regression and lasso, the full algorithm fit, Subsembles, and naive versions have essentially the same performance. For regression tree and random forest, all the Subsembles significantly outperform the full fit. For regression tree, the naive versions have essentially the same performance as the corresponding Subsembles, and also significantly outperform the full fit. However, for random forest, the naive versions are much worse than the Subsembles, and the naive versions perform significantly

Table 2.1: MSPE comparison of three different methods using the same underlying algorithm: the 'Full' fit of the algorithm only once on the entire data set, Subsembles with randomly created subsets, and naive averages of the same fits used in the Subsembles. The underlying algorithm used in each row appears in The Algorithm column. J indicates the number of subsets. The method with lowest MSPE for each underlying algorithm is in bold. The number of symbols in the superscript indicates the significance level of a t-test for the difference in means between the subset method and the full fit: 0.10 (1 symbol), 0.05 (2 symbols), 0.01 (3 symbols). Asterisks (*) are used when the subset method MSPE is significantly lower, and tick marks ($'$) are used when the subset method MSPE is significantly higher.

Dataset	Algorithm	Full	Method	$J = 2$	$J = 3$	$J = 4$	$J = 5$
Sim 1							
	Linear	**347.6**	Subsemble	**347.6**	347.8	347.7	348.1
			Naive	**347.6**	348.0	347.7	348.2
	Lasso	**341.4**	Subsemble	341.6	342.0	342.2	343.1
			Naive	342.7	343.6	345.5	347.6$'$
	Tree	265.6	Subsemble	254.7**	253.6***	253.6***	**249.9*****
			Naive	254.9**	253.2***	254.4***	251.7***
	Forest	229.3	Subsemble	**195.1*****	195.6***	196.5***	198.5***
			Naive	246.5$'''$	258.4$'''$	270.3$'''$	279.6$'''$
Sim 2							
	Linear	340.8	Subsemble	271.7***	271.7***	**271.4*****	277.6***
			Naive	362.9$'''$	408.5$'''$	549.2$'''$	3.10 e6$'''$
	Lasso	274.0	Subsemble	274.1	**273.8**	274.2	275.1
			Naive	**273.8**	274.1	273.9	274.1
	Tree	349.9	Subsemble	271.1***	**270.9*****	**270.9*****	271.7***
			Naive	334.4***	316.8***	302.3***	295.1***
	Forest	263.0	Subsemble	**252.6*****	253.3***	253.7***	255.1***
			Naive	264.4	265.4$''$	266.2$'''$	267.0$'''$
Yacht							
	Linear	83.42	Subsemble	57.95*	58.18*	57.38*	**55.66****
			Naive	72.44	72.17	71.49	72.17
	Lasso	80.82	Subsemble	57.34	58.94	58.01	**55.71***
			Naive	74.17	74.74	75.15	75.16
	Tree	**4.296**	Subsemble	6.866	15.60$'''$	22.39$'''$	17.52$'''$
			Naive	7.349	18.75$'''$	24.30$'''$	20.41$'''$
	Forest	14.54	Subsemble	**7.213***	8.460	8.760	8.977
			Naive	21.13	28.28	35.35$''$	43.29$''$
Diamond							
	Linear	3.07 e5	Subsemble	**2.61 e5****	2.73 e5*	2.67 e5*	2.72 e5*
			Naive	2.74 e5*	2.75 e5*	2.94 e5	2.76 e5
	Lasso	3.13 e5	Subsemble	**2.73 e5*****	2.75 e5**	2.74 e5***	2.96 e5
			Naive	2.78 e5**	2.91 e5**	3.05 e5	2.90 e5
	Tree	1.15 e6	Subsemble	1.10 e6	**1.01 e6***	1.10 e6	1.07 e6
			Naive	1.11 e6	1.06 e6	1.18 e6	1.13 e6
	Forest	**5.05 e5**	Subsemble	6.00 e5$''$	6.81 e5$'''$	7.80 e5$'''$	8.26 e5$'''$
			Naive	6.54 e5$'''$	7.50 e5$'''$	8.04 e5$'''$	8.60 e5$'''$

worse than the full fit.

For simulated data set 2, the lasso once again has essentially the same performance across the full fit, Subsembles, and naive versions. With linear regression, regression tree, and random forest, the Subsembles significantly outperform the full fit. The naive version has poorer performance. With linear regression and random forest, the naive version is significantly worse than the full fit. With regression tree, the naive version does significantly improve on the full fit, but still has much worse performance than the Subsembles. In this simulation, we also see an important problem with the naive version: there is no way to account for a poor subset-specific fit. This is likely the reason why the MSPE results for the naive versions with linear regression are so high.

With the yacht data set, the full fit of the regression tree fit was significantly better than both the Subsembles and the naive versions. For the other three underlying algorithms, at least one Subsemble significantly outperformed the full fit, while the naive versions were either not significantly different, or had significantly worse performance than the full fit.

For the diamond data set, with linear regression and lasso, most Subsembles and naive versions has significantly better performance than the full fit, with the Subsembles being more significantly better. With regression tree, one Subsemble was significantly better than the full fit, while all naive versions were not significantly different. With random forest, both Subsembles and naive versions were significantly worse than the full fit.

Across all the data sets, we see that the Subsembles can often significantly outperform the full algorithm. Note that performance of the Subsemble depends on both the underlying algorithm and the distribution generating the data. None of the underlying algorithms always had the best performance by using the full fit, or by using Subsembles. As a result, for real data sets in which the generating distribution is unknown, we cannot predict ahead of time whether the full fit or Subsembles of a given underlying algorithm will have better performance.

Subsembles also perform at least as well as, and usually better than, the corresponding naive averaging versions. This result is not only practical: it is also predicted by the theoretical oracle inequality in Section 2.2.2. The oracle result tells us that Subsemble performs as well as the best possible combination of subset-specific fits. Since naive averaging *is* a possible combination of subset-specific fits, it follows that Subsemble is asymptotically superior.

2.4.3 SuperLearner Performance Comparison

In this study, we compare the performance of the SuperLearner using two different libraries of candidate algorithms: a library including only algorithms fit on the full data set, and a library including both Subsembles and algorithms fit on the full data set. We again created the subsets randomly, used linear regression as metalearner, and used the following underlying algorithms: linear regression, lasso, regression tree, and random forest. In the library with Subsembles versions, we included Subsembles with 2 and 5 subsets for each of the 4 algorithms, as well as the full algorithms.

Table 2.2: MSPE comparison of SuperLearners with two different libraries: one using only algorithms fit once on the entire data set, and the other using both algorithms fit once on the entire data set and two Subsemble versions of each algorithm. Underlying algorithms used were: linear regression, lasso, regression tree, and random forest. The method with lowest MSPE for each each data set is in bold. The Significance column indicates the significance level of a t-test for the difference in means between the two methods.

Dataset	No Subsembles	Subsembles	Significance
Sim 1	228.4	**194.7**	< 0.01
Sim 2	263.9	**250.7**	< 0.01
Yacht	4.827	**4.046**	0.07
Diamond	284171	**248882**	0.02

For the simulated data sets, we simulated training sets of 1,000 observations and test sets of 10,000 observations, and repeated the experiment 10 times. For the real data sets, we split the data sets into 10 folds, and let each fold serve as the test set. Mean Squared Prediction Error (MSPE) results were averaged across the 10 trials for both simulated and real data sets. We also performed a t-test for the difference in means between the two SuperLearner library results. Results are presented in Table 2.2.

Across all data sets, the SuperLearner whose library included Subsembles outperformed the SuperLearner whose library used only full algorithm versions.

2.5 Summary

In this chapter, we introduced the Subsemble procedure for fitting the same underlying algorithm on different subsets of observations, and learning the optimal weighted combination using V-fold cross-validation. We provided a theoretical statistical result, showing that Subsemble performs as well as the best possible combination of the subset-specific fits. Through simulation studies and real data analysis, we illustrated that Subsembles with randomly created subsets can provide practical performance improvements on moderate sized data sets.

Chapter 3

Learning Subsemble's Subsets

3.1 Introduction

In Chapter 2, we studied the performance of Subsembles created from randomly selected subsets of observations. In particular, we demonstrated that such random subset Subsembles have good performance in practice, and often provide better prediction performance than fitting the underlying algorithm only once on a full available data set.

Note that the final Subsemble estimator varies depending on the data used to create each subset-specific fit. As a result, different strategies for creating Subsemble's subsets will result in different Subsembles. In this chapter, we introduce a different method for partitioning a data set into the subsets used in Subsemble. In particular, we propose the use of Supervised Subsembles, which create subsets through supervised partitioning of the covariate space, combined with a form of histogram regression as the metalearner used to combine these subset-specific fits. We also develop a practical Supervised Subsemble method, which employs regression trees to both partition the observations into the subsets used in Subsemble, and select the number of subsets to use. We discuss computational independence properties of our proposed methods that are advantageous for applications involving big data, and show through simulations that our proposed version of Subsemble can result in further improved prediction performance.

The remainder of this chapter is organized as follows. The Supervised Subsemble approach for creating subsets, along with associated metalearner, is presented in Section 3.2. We describe the practical Supervised Subsemble method using regression trees in Section 3.3. Simulation and real data analysis results are discussed in Section 3.4. We summarize and conclude in Section 3.5.

3.2 Supervised Subsemble

3.2.1 Supervised Partitioning of the Covariate Space

To obtain the subsets used in Subsemble, we propose partitioning of the covariate space to create J disjoint subsets of covariates, $S_1, \dots, S_J$, such that any given vector of covariates belongs to exactly one of the J subsets. Numerous methods to achieve a partitioning of the covariate space are available. For example, the unsupervised k-means algorithm could be used to first cluster the observations into J clusters based on only their covariates, with these J clusters also forming the J subsets. In general, supervised partitioning methods also consider the outcome variable, thus creating a partitioning that is directly predictive of the outcome.

Compared to randomly selecting the subsets, constructing the subsets to be similar internally and different from each other results in locally smoothed subset-specific fits when fitting the same algorithm on each subset. In particular, each subset-specific fit is in fact tailored for the associated partition of the covariate space. More specifically, the subset-specific fit $\hat{\Psi}_j$ associated with S_j is tailored for the partition S_j, and we would not expect $\hat{\Psi}_j$ to be a good fit for observations with covariates in some other $S_{j'}$, where $j \neq j'$.

3.2.2 Modified Histogram Regression Metalearner

To reflect the above observation, we propose using a modified version of histogram regression as the metalearner used to combine the subset-specific fits. The usual form of histogram regression, applied to our J subsets S_j, would output a local average of the outcome Y within each subset. Instead of this standard form of histogram regression, our version of histogram regression outputs the associated $\hat{\Psi}_j$ for each subset S_j. In addition, our version of histogram regression includes a coefficient and intercept within each subset. Concretely, we define our proposed histogram regression metalearner $\hat{\Phi}$ for combining the subset-specific fits as follows:

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

The value of the functional form of Equation 3.1 is its generalization to using more than one underlying algorithm. That is, applying multiple prediction algorithms to each subset. That is, instead of applying a single underlying algorithm $\hat{\Psi}$ to each subset, the Subsemble procedure readily accommodates applying L underlying algorithms $\hat{\Psi}^1, \dots, \hat{\Psi}^L$ to each subset. The generalization of Equation 3.1 gives us the following histogram regression metalearner $\hat{\Phi}$ for combining these multiple subset-specific fits as follows:

$$\hat{\Phi}(\hat{\Psi}^1, \dots, \hat{\Psi}^L)(x) = \sum_{j=1}^{J} \left[I(x \in S_j)\left(\beta_j^0 + \sum_{\ell=1}^{L} \beta_j^\ell \hat{\Psi}_j^\ell(x)\right)\right] \tag{3.2}$$

3.2.3 Computational Independence of Subsets

Note that the computations Subsemble performs on each subset are *always* independent, even in the cross-validation training steps. This is because the partitioning of the n observations into J subsets remains the same during cross-validation. As a result, leaving out a fold v from a subset j produces all the data assigned to the j-th subset in the cross-validation training set. However, with randomly constructed subsets, minimizing the cross-validated risk to learn the optimal combination of the subset-specific fits requires access to all the data.

The Supervised Subsembles proposed here have the additional benefit of preserving the subset computation 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 completely computationally independent across the entire procedure.

To see this, let β_n to be the cross-validation selector of β. Then by definition of the cross-validation selector,

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

Using the fact that each observation i belongs to exactly one subset S_j, we rearrange terms as follows:

$$= \arg\min_{\beta} \sum_{j=1}^{J} \sum_{i:i\in S_j} \left\{ Y_i - \hat{\Phi}_{\beta}(\tilde{X}_i) \right\}^2$$

From the definitions of $\hat{\Phi}$ and $\tilde{X}_i$,

$$= \arg\min_{\beta} \sum_{j=1}^{J} \sum_{i:i\in S_j} \left\{ Y_i - \sum_{j'=1}^{J} \left[I(X_i \in S_{j'}) \left(\beta_{j'}^0 + \sum_{\ell=1}^{L} \beta_{j'}^{\ell} \hat{\Psi}_{j',v(i)}^{\ell}(X_i) \right) \right] \right\}^2$$

Again, since each observation i belongs to exactly one S_j,

$$= \arg\min_{\beta} \sum_{j=1}^{J} \sum_{i:i\in S_j} \left\{ Y_i - \left[\beta_j^0 + \sum_{\ell=1}^{L} \beta_j^{\ell} \hat{\Psi}_{j,v(i)}^{\ell}(X_i) \right] \right\}^2$$

Finally, since the observations $i : i \in S_j$ are disjoint for different j, terms involving each β_j can be minimized independently:

$$= \left\{ \arg\min_{\beta_j} \sum_{i:i\in S_j} \left(Y_i - \left[\beta_j^0 + \sum_{\ell=1}^{L} \beta_j^{\ell} \hat{\Psi}_{j,v(i)}^{\ell}(X_i) \right] \right)^2 \right\}_{j=1}^{J}$$

Thus, each term β_j can be estimated by minimizing cross-validated risk using only the data in subset S_j.

We are thus able to estimate the coefficient associated with each subset independently, using only data within that subset. Consequently, unlike the randomly constructed subsets, we avoid needing to recombine data to produce the final prediction function.

3.3 SRT Subsemble: Supervised Regression Tree Subsemble

3.3.1 Motivation

The Supervised Subsembles proposed in the previous section maintain computational independence across subsets. However, this assumes that both the number of subsets, and the partitioning of the covariate space, are provided. In this section, we propose the a practical Supervised Regression Tree Subsemble (SRT Subsemble) algorithm, which uses regression trees with Subsemble as a practical and computationally feasible way to determine both of the number of subsets, and the partitioning of the covariate space to create the subsets.

To motivate our approach, we first discuss relevant concerns about constructing covariate space partitions when dealing with large-scale data set applications. For big data, splitting data up can be a significant computational concern. As a result, it is preferable to avoid approaches that, when creating different numbers of subsets, first split the full data set into, for example, two partitions, and then recombine the data in order to determine a way to split the data into three partitions. Instead, it is better to work with greedy methods, which enforce that once the data has been split, any further splits will only divide an already existing partition.

3.3.2 Constructing and Selecting the Number of Subsets

Classification and Regression Trees (CART), developed by Breiman et al. 1984, recursively partition 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, CART fits an ANOVA model and uses the between-groups sum-of-squares metric. The best split is then selected to maximize this between-groups sum-of-squares. For additional details, we refer the reader to Breiman et al. 1984.

The first iteration of CART thus 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 for Subsemble, we apply CART as follows. First, we simply 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. We treat the m nodes of each m-th sub-tree as a candidate partitioning into m subsets.

Next, we explore the sequence of M possible partitions (sequence of sub-trees) produced by CART, beginning at the root. For each candidate number of subsets $1, \ldots, M$, we fit the associated Supervised Subsemble. Concretely, with m subsets, we create L subset-specific fits $\hat{\Psi}_j^\ell$ for each subset $j = 1, \ldots, m$, and create the overall prediction function according to Equation 3.2. Note that we use CART to create the subsets S_j that appear in Equation 3.2. Finally, we select the best number of subsets as the Subsemble with minimum cross-validated risk.

3.3.3 Computational Advantages

Our proposed SRT Subsemble retains the desirable subset computational independence discussed in Section 3.2, while also creating the subsets used in Subsemble in a computationally friendly way, as well as providing a criteria for choosing the number of subsets to use in Subsemble.

As a consequence of this subset computational independence, note that in 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 due the fact that the nodes are computationally independent in the SRT Subsemble framework, plus the fact that at each split of the tree, all nodes remain the same, except for the single node in the m-th tree that is split into two new nodes in the $m + 1$-st tree.

3.3.4 Implementation Flexibility

There are several paths that can be taken when applying the SRT Subsemble process in practice. These user-selected options allow SRT Subsemble to be quite flexible, with decisions made to suit the application at hand. There is no one best approach, instead the options will be determined based on the application/constraints/desired properties. We briefly discuss these options.

First, the user must decide how to build and explore 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 tracked while the tree is being built-out. That is, after each additional split to the tree, build the associated Subsemble, calculate its associated cross-validated risk, and stop adding additional splits when some stopping criteria is achieved. As an 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. That is, building a Subsemble with only one subset containing all observations. For small- to moderate-sized data sets, where computational considerations are of less 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 only then start building Subsembles. 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, since all subsequent computations remain independent from other partitions.

3.4 Data Analysis

3.4.1 Preliminary Observations

We next present results from comparing the performance of SRT Subsemble with the version of Subsemble using randomly created subsets. We discuss further implementation details in the next subsection.

Before presenting our simulated and real data set results, we first discuss a few preliminary observations, through simple examples, which show that there are certainly scenarios in which SRT Subsemble results in better performance than the random subset version of Subsemble. To see this, consider an actual histogram with a single covariate, where the outcome is simply the mean in various subsets of that covariate. Further, suppose we use a single underlying algorithm $\hat{\Psi}$, which simply takes the mean outcome among observations. In this case, it is clear that the SRT Subsemble procedure will produce superior prediction performance, as compared to the random subset version of Subsemble.

We also note that using homogeneous subsets presents desirable behavior for more subsets with fewer observations. With SRT Subsemble, as subsets become smaller, they also become more homogeneous. As a result, data-adaptive prediction algorithms can still result in good fits. In contrast, when using random subsets, aggressive data-adaptive algorithms often sacrifice performance from over-fitting when subsets become too small. We also note that using homogeneous subsets allow less data-adaptive prediction algorithms to achieve good fits within each subset. In particular, we do not need to use aggressive algorithms within each subset to get a good fit.

3.4.2 Implementation Details

As mentioned above, we compared the performance of SRT Subsemble with the version of Subsemble using randomly created subsets. In both methods, we estimated β coefficients to minimize cross-validated risk, and used the same four underlying algorithms for the purposes of this demonstration: linear regression, lasso, regression tree, and random forest. Additional details about the methods compared are discussed below.

For SRT Subsemble, we used the following procedure to build the tree used in SRT Subsemble. We used the `R` package `rpart` (Therneau, Atkinson, and Ripley 2012) to build a full tree. In the `rpart` package, the splits and size of the tree built are controlled by four parameters: `minsplit` (minimum number of observations needed for a node to split), `minbucket` (minimum number of observations in any node), `cp` (complexity parameter), and `maxdepth` (maximum depth of any node in the tree). Since we used $V = 10$-fold cross-validation to select the optimal number of subsets, we set `minbucket`$= 2V$ to ensure a sufficient number of observations in each node to perform this cross-validation. For the remaining parameters, we used `rpart`'s defaults: `minsplit` $= 3\times$`minbucket`, `cp` $= 0.01$, and `maxdepth` $= 30$.

We then selected the best number of subsets for the SRT Subsemble by building a Subsemble as in Equation 3.2 at each sub-tree outputted by `rpart`, starting at the root node with only one subset, and selecting the Subsemble with the lowest estimated cross-validated risk among the sequence of sub-trees outputted by `rpart`.

For the random subset Subsembles, we first built the same number of Subsembles as those that we explored with the SRT Subsemble. That is, if the SRT Subsemble explored a full tree with M nodes, we built random subset Subsembles with $1, \ldots, M$ nodes. We used combined the subset-specific fits according the the following equation:

$$\hat{\Phi}(x) = \sum_{j=1}^{J} \left(\beta_j^0 + \sum_{\ell=1}^{L} \beta_j^\ell \hat{\Psi}_j^\ell \right)$$

To select the optimal number of subsets for the finally selected random subset version, we simply selected the Subsemble with the lowest *oracle* risk; that is, the lowest true risk on the test data. Note that this is not possible in practice, and is included here only for illustration purposes. Observe that we are comparing the SRT Subsemble to the *best conceivable* random subset version of Subsemble. In particular, this includes the version with only a single subset: a combination of the underlying algorithms fit on the full available data set (i.e., the SuperLearner method of van der Laan, Polley, and Hubbard 2007).

3.4.3 Description of Data Sets

In the studies that follow, we used four small to moderate sized data sets (Synthetic 1, Synthetic 2, Yacht, Diamond) to evaluate the practical performance of Subsemble. All data sets have one real-valued output variable, and no missing values.

The first two data sets are simulated. We created this pair of synthetic data to demonstrate one scenario in which the SRT Subsemble provides better performance (Synth 1), and another scenario in which the random subset Subsemble yields better performance (Synth 2). The first synthetic data set exhibits significant greater local behavior than the second synthetic data set. Both simulated data sets have 2 input variables $X_1, X_2 \sim N(0,9)$, and random error $\epsilon \sim N(0,1)$. The outcome variable for each simulated data set is generated as follows:

Synth 1:

$$Y = \epsilon + \sin(X_1) + 2\log(|X_1|) + 3\sqrt{|X_1|} + \sin(0.5\pi X_1)$$

Synth 2:

$$Y = 2 + \epsilon + \sin(X_1)$$

Note that in practical applications, we do not know the true data generating distribution a priori. In particular, for data sets with many covariates, it becomes challenging even to visualize the data, and thus impractical to determine ahead of time whether or not the data exhibits significant local behavior. As a result, the two synthetic data sets presented here merely serve for illustrative purposes, to provide the reader with some general intuition for the performance of the SRT Subsemble method.

The second two data sets are publicly available real-world data, and are the same data sets studied in the analysis of Chapter 2. These data sets illustrate actual data from applications in which the SRT Subsemble method performs better than the random subset Subsemble. The yacht data set, available from Bache and Lichman 2013, has 308 observations and 6 input variables. The diamond data set, described by Chu 2001, has 308 observations and 17 input variables.

3.4.4 Results

For the synthetic data sets, we simulated training and test sets of 1,000 observations, and repeated the experiment 10 times. For the real data sets, we split the data sets into 10 folds, and let each fold serve as the test set. Mean Squared Prediction Error (MSPE) results were averaged across the 10 trials for both simulated and real data sets. We also performed a t-test for the difference in means between the two methods. The average number of subsets used in each method, as well as the maximum number of subsets (size of the full tree built) are also included. Results are presented in Table 3.1.

From Table 3.1, we see that the SRT Subsemble method significantly performs better than the oracle-selected random subset version on three of the four data sets: synthetic data set 1, the yacht data, and the diamond data. In fact, for all data sets, the two methods are significantly different at the 0.01 level.

Table 3.1: MSPE comparison of SRT Subsemble versus an oracle-selected version of Subsemble with random subsets. Underlying algorithms used were: linear regression, lasso, regression tree, and random forest. The method with lowest MSPE for each each data set is in bold. The Sig / Max column indicates the significance level of a t-test for the difference in means between the two methods, and the maximum number of subsets considered.

Dataset		SRT	Random Oracle	Sig / Max
Synth 1	MSPE	**1.21**	1.45	<0.01
	# Subsets	6.3	2.5	7.5
Synth 2	MSPE	2.84	**1.40**	<0.01
	# Subsets	7.4	1.4	9.8
Yacht	MSPE	**1.19**	2.96	0.01
	# Subsets	3.2	1.7	3.5
Diamond	MSPE	**1.30 e5**	2.10 e5	<0.01
	# Subsets	3.1	2.1	5.0

We emphasize that while the SRT Subsemble method is viable in practice, the comparison oracle-selected random subset Subsemble is not, since it utilizes the test data to select the number of subsets to minimize the MSPE *on the test data*. The fact that the SRT Subsemble method achieves lower MPSE than this oracle-selected random subset version show clearly that SRT Subsemble can often significantly outperform any random subset version, including a single subset.

3.5 Summary

In this chapter, we introduced Supervised Subsembles and proposed the SRT Subsemble method. Supervised Subsembles partition the covariate space to obtain subsets, and use a modified form of histogram regression as the metalearner used to combine the subset-specific fits. We described how Supervised Subsembles preserve computational independence of the subsets throughout the entire algorithm. SRT Subsemble is a practical method to both construct the covariate partitioning, and select the optimal number of subsets. We explained the desirable computational properties of SRT Subsemble, as well as the flexibility it provides in implementation. Finally, we presented simulated and real data set results, which demonstrated that the SRT Subsemble method can result in better prediction performance than comparable random subset Subsembles.

Chapter 4

The R Package subsemble

4.1 Introduction

To make the Subsemble method easily applicable and accessible to both practitioners and the research community, we developed the `R` package **subsemble**. In this chapter, we describe the **subsemble** package and functionality, discuss our algorithm implementation, and demonstrate the package's usage for prediction applications through an example data set.

The remainder of this chapter is organized as follows. Implementation details are discussed in Section 4.2. We describe the features of the `subsemble` function in Section 4.2.1, and the algorithm implementation in Section 4.2.2. We illustrate the application of **subsemble** for prediction with an example data set in Section 4.3. We summarize and conclude in Section 4.4.

4.2 Package Description

The **subsemble** package leverages several functions from the **SuperLearner** package of Polley and van der Laan 2012. In particular, we use the **SuperLearner** package's easy to use prediction algorithm wrappers and built-in data partitioning functions. The `subsemble` function is also similar in syntax to the `SuperLearner::SuperLearner` function.

While numerous prediction algorithms are available in `R`, the default versions of these algorithms follow different syntax conventions. Since **subsemble** takes as input user-specified learner and metalearner algorithms, common syntax is important. Fortunately, the algorithm API provided in the **SuperLearner** package provides this common syntax through a variety of prediction algorithm wrappers.

The **subsemble** package utilizes the **SuperLearner** package's data partitioning function, `CVFolds`. We use the `SuperLearner::CVFolds` function to assign training observation indices to (optionally shuffled and stratified) V-fold cross-validation folds.

When the user does not specify the exact subset assignments, the `SuperLearner::CVFolds` function is also used to assign training observation indices to subsets, since it is a general-purpose data partitioning function. The subset assignment process is determined by the `subControl` argument of the `subsemble` function. Additional details are discussed in Sections 4.2.1 and 4.2.2.

4.2.1 The `subsemble` Function

The `subsemble` function implements the Subsemble algorithm. It is used to train a subset-ensemble model and is the primary function of the package. In this section, we describe the most interesting arguments of the `subsemble` function.

Underlying algorithm (`learner`)

The underlying prediction algorithm(s) trained on each of the subsets is specified by the `learner` argument in `subsemble`. We currently use the SuperLearner algorithm API from the **SuperLearner** package, which identifies the algorithms by wrapper function name (e.g. `"SL.glm"`) from the `SuperLearner` package. A complete list of available algorithms is available via the `SuperLearner::listWrappers` function. The currently available algorithms are shown in Table 4.1.

Table 4.1: Learning algorithms for the `subsemble` function supported by default via the **SuperLearner** package.

	Function Name	Package	Tuning Parameters	Description

1	`SL.bart`	**BayesTree**	`ntree`
			`sigdf`
			`sigquant`
			`k`
			`power`
			`base`
			`binaryOffset`
			`ndpost`
			`nskip`
				Bayesian Regression Tree

2	`SL.bayesglm`	**arm**	-	Bayesian GLM

3	`SL.caret`	**caret**	`method`
			`tuneLength`
			`trControl`
			`metric`
				Interface to the **caret** package

4	`SL.caret.rpart`	**caret**	`tuneLength`
			`trControl`
			`metric`
				caret Regression Tree

5	`SL.cforest`	**party**	-	Conditional Tree Forest

6	`SL.earth`	**earth**	`degree`
			`penalty`
			`nk`
				Adaptive Regression Splines

7	`SL.gam`	**gam**	`deg.gam` `cts.num`	Generalized Additive Model
8	`SL.gbm`	**gbm**	`gbm.trees` `interaction.depth`	Gradient Boosting
9	`SL.glm`	**stats**	-	Generalized Linear Model
10	`SL.glm.interaction`	**stats**	-	GLM with Interaction Terms

11	`SL.glmnet`	**glmnet**	`alpha`
			`nfolds`
			`nlambda`
			`useMin`
				Elastic Net

12	`SL.ipredbagg`	**ipred**	`nbagg` `control`	Bagging Trees
13	`SL.knn`	**class**	`k`	K-Nearest Neighbors
14	`SL.leekasso`	**sva**	-	Leekasso
15	`SL.loess`	**stats**	`span` `l.family`	Local Polynomial Spline Regression

16	`SL.logreg`	**LogicReg**	`ntrees`
			`nleaves`
			`kfold`
				Logic Regression

17	`SL.mean`	**stats**	-	Weighted Mean
18	`SL.nnet`	**nnet**	`size`	Neural Network
19	`SL.polymars`	**polspline**	-	Adaptive Polynomial

20	`SL.randomForest`	**randomForest**	`ntree`
			`mtry`
			`nodesizes`
				Random Forest

21	`SL.ridge`	**MASS**	`lambda`	Ridge Regression

22	`SL.rpart`	**rpart**	`cp`
			`minsplit`
			`xval`
			`maxdepth`
			`minbucket`
				Regression Tree

23	`SL.rpartPrune`	**rpart**	`cp`
			`minsplit`
			`xval`
			`maxdepth`
			`minbucket`
				Pruned Regression Tree

24	`SL.step`	**stats**	`direction`
			`trace`
			`k`
				Stepwise Regression

25	`SL.step.forward`	**stats**	`trace` `k`	Forward Stepwise Regression

26	`SL.step.interaction`	**stats**	`direction`
			`trace`
			`k`
				Forward Stepwise Regression
				with Interaction Terms

27	`SL.stepAIC`	**MASS**	`direction`
			`steps`
			`k`
				Stepwise Regression

28	`SL.svm`	**e1071**	`type.reg`
			`type.class`
			`nu`
				Support Vector Machine

Combination Algorithm (`metalearner`)

The prediction algorithm used to learn the optimal combination of the subset-specific fits is specified by the `metalearner` argument. The `metalearner` argument uses the same list of function wrapper names as the `learner` argument. Unlike the `learner` argument, the `metalearner` argument must specify exactly one algorithm.

Subset Description or Creation (`subsets`)

The `subsets` argument supports three ways to specify the assignment of training observations to subsets. The user may provide a vector assigning each training observation to a subset. Alternatively, the user may provide a list of vectors, with each vector identifying the observations belonging to the same subset. Finally, the user may simply specify the number of subsets into which the training data should be partitioned. In this final case, `subsemble` will handle creation of the subsets, while the user can still control how these subsets are created by using the `subControl` argument.

Subset Process Parameters (`subControl`)

If the user specifies `subsets` as a number, the creation of subsets is controlled by the list of two logical parameters (`stratifyCV` and `shuffle`) specified in the `subControl` argument. The user may specify whether the training observations should be stratified by a binary response, and assigned to subsets to preserve the same response ratio across subsets, via the `stratifyCV` parameter. The user may specify whether the training observations should be shuffled before assignment to subsets through the `shuffle` parameter. The last element of the `subControl` list is `subControl[["supervised"]]` which, in the future, will support the supervised learning of subsets. Currently this is set to `NULL`.

Cross-Validation Process Parameters (`cvControl`)

The cross-validation process can be controlled through the list of three parameters (`V`, `stratifyCV`, and `shuffle`), specified in the `cvControl` argument. The number of cross-validation folds is specified by the `V` parameter. The `stratifyCV` and `shuffle` parameters are the same as in the `subControl` list and also both default to `TRUE`.

Learning Process Parameters (`learnControl`)

Currently, the only parameter controlled by the `learnControl` list is `multiType`, which is only used if there is more than one learner specified by the `learner` argument. The two supported values for `multiType` are `"crossprod"` (the default) and `"divisor"`. The `"crossprod"` type will train all of the learners on each of the subsets. For the `"divisor"` type, one learner will be trained on each subset, and thus the length of the `learners` vector must be a divisor of the number of subsets. If `length(learner)` equals the number of subsets, each learner will be applied to a single subset. If `length(learner)` is a divisor of the number of subsets, then the learners will be repeated as necessary (to equal the number of subsets).

Parallelization (`parallel`)

The `parallel` argument is specified by a character string and defaults to `"seq"` for sequential computation. Using the `"multicore"` option will parallelize several pieces of code using the built-in **parallel** library. In particular, the internal cross-validation step, as well as the final model fitting of individual sub-models (underlying algorithms trained on the subsets of the training data), will be performed in parallel. This is discussed in greater detail in Section 4.2.2

4.2.2 Algorithm Implementation

The Subsemble algorithm implementation can be broken up into four main phases. The first task is to partition (the row indices of) the training data into J subsets and assign the indices to cross-validation folds. The next three phases are: internal cross-validation, metalearning, and the final model fitting phase.

Data Partitioning Step

If the subsets are not explicitly defined by the user via the `subsets` argument, they will be created as specified by the `subControl` argument. Currently, the package supports partitioning the indices at random (with optional stratification by a binary outcome, when applicable) using the `SuperLearner::CVFolds` function, but in the future we hope to include additional functionality for learning subsets.

After the indices have been partitioned into subsets, we further partition each subset into V cross-validation folds as specified by the `cvControl` argument. This list of lists, `subCVsets`, is returned as part of the `subsemble` function output.

Cross-Validation Step

The cross-validation step involves generating cross-validated predictions for the learning algorithm on each of the subsets. When there are multiple learners defined by the `learner` argument, the “cross-product” multi-learning type involves training all of the L unique learners on each of the J subsets, for a total of $L \times J$ distinct models that make up the ensemble. As mentioned previously, this is the default behavior for `subsemble` and can be modified using the `learnControl` parameter. The “divisor” multi-learning method will train one learner on each of the J subsets, repeating unique learners across subsets (assuming that L is a divisor of J), for a total of J unique models.

The V iterations of the cross-validation step can be performed in parallel using the `parallel` option. During the v^{th} iteration, for each subset, j, we fit a model, $\hat{\Psi}_{j,-v}^{(l)}$, where $l \in \{1, ..., L\}$ indexes the specific learner. The training indices will be those that are in subset j, but not in fold v of subset j. Then we generate predictions on the test set that is defined by the indices from fold v of all subsets.

In the case of cross-product multi-learning, we repeat this process across V folds, J subsets, and L learners. By stacking the cross-validation predictions back together, we construct the matrix $\mathbf{Z}$, of dimension $n \times (J \times L)$, where n is the number of observations in the original training set. Define $M := J \times L$. Then each of the M columns of the $\mathbf{Z}$ matrix correspond to each of the unique subset-specific learner models that make up the final ensemble model.

For a single learner algorithm or divisor multi-learning, we repeat this process across V folds and J subsets, while using either the same (in the case of a single underlying learner) or different (in the case of divisor multi-learning) underlying learner. In this case, our $\mathbf{Z}$ matrix has dimension $n \times J$.

The $\mathbf{Z}$ matrix is required for the next step, metalearning.

Metalearning Step

As described previously, the metalearning algorithm is specified using the `metalearner` argument. The metalearning step simply fits the metalearner algorithm $\hat{\Phi}$, using the matrix $\mathbf{Z}$ of cross-validated predictions as the training data, and the original outcome vector $\mathbf{Y}$. The resulting metalearner fit is the function which combines the output from the M individual models, which are fit in the next step. This fit is saved as part of the function output in an object called `metafit`.

Final Model-Fitting Step

The last step involves training M models, where $M = J \times L$ in the case of cross-product multi-learning and $M = J$ in the case of divisor multi-learning. The final models can be fit in parallel using the `parallel` argument. These objects are saved in a list called `subfits` and returned as part of the output of the `subsemble` function.

The final Subsemble model takes the predicted values generated from the individual underlying models and combines them together into a single predicted value using the metalearner fit. The test set predictions are stored in a vector called `pred` and returned as part of the output of the `subsemble` function. For reference, the predicted values from each of the individual M models are also saved in a data.frame called `subpred`. Like many other machine learning packages in R, we also provide a `predict.subsemble` function that takes as input a subsemble fit object along with a test set, and generates predictions for the observations in the test set.

4.3 Examples

4.3.1 Set Up the Subsemble

We first present a simple example showing how to use the `subsemble` function with a single underlying learning algorithm and mostly default arguments. In this example, our underlying algorithm is random forest, our metalearner is lasso, and our Subsemble is composed of 3 subsets. We begin by specifying these arguments:

```
learner <- c("SL.randomForest")
metalearner <- c("SL.glmnet")
subsets <- 3
```

4.3.2 Training

With the important arguments specified above, we now train our Subsemble. In this example, we use training data `X` with binary outcome vector `Y`. Using mostly default argument values, we train our Subsemble as follows:

```
fit <- subsemble(Y=Y, X=X, newX=newX, family=binomial(),
                 learner = learner, metalearner = metalearner,
                 subsets = subsets)
```

4.3.3 Testing

Since we specified a test data set `newX` during the above training phase, the `subsemble` function returns predicted values for that test set via the `pred` value. We can use these predictions on the test data (`fit$pred`), combined with the true binary outcome vector of the test data (`newY`), to evaluate the Subsemble model performance using AUC. In this example, the test set AUC is 0.925.

```
auc <- cvAUC(predictions=fit$pred, labels=newY)$cvAUC
print(auc)
```

Alternatively, we can also use `subsemble`'s `predict` function to generate predictions on a test set after creating the Subsemble `fit` object. The format is similar to most machine learning algorithm interfaces in R, as shown in the following example:

```
pred <- predict(fit, newdata=newX)
auc <- cvAUC(predictions=pred$pred, labels=newY)$cvAUC
```

4.3.4 Binary Outcome Data Example

We next present a binary outcome example using a simulated example data set from the **cvAUC** package of LeDell, Petersen, and van der Laan 2013. The data set represents admissions information for a graduate program in the sciences. The binary outcome represents 1 for "admitted" and 0 for "not admitted." There are three continuous predictors and two binary predictors for a total of 5 feature columns.

```
library(cvAUC)
data(admissions)
```

We first created an example training data set using the first 400 observations, and an example test data set using the final 100 observations:

```
X <- subset(admissions, select=-c(Y))[1:400,]
newX <- subset(admissions, select=-c(Y))[401:500,]
Y <- admissions$Y[1:400]
newY <- admissions$Y[401:500]
```

For this demonstration, we use two underlying learners (random forest and GLM) to illustrate the two currently implemented types of learning with multiple underlying algorithms. We begin by setting up the Subsemble.

```
learner <- c("SL.randomForest", "SL.glm")
metalearner <- c("SL.glm")
subsets <- 2
```

Cross-Product Multi-Learning

In this example, the use the two underlying algorithms specified above, with the remaining arguments allowed to use their default values (including `learnControl`). With `learnControl[["multiType"]]` set to `"crossprod"` (the default), we ensemble four models together – a random forest on both of the two subsets, and a GLM on both of the two subsets.

```
fit <- subsemble(Y=Y, X=X, newX=newX, family=binomial(),
                 learner = learner, metalearner = metalearner,
                 subsets = subsets)
```

We then evaluate the Subsemble model performance on a test set. In this example, the test set AUC is 0.937.

```
auc <- cvAUC(predictions=fit$pred, labels=newY)$cvAUC
print(auc)
```

Divisor Multi-Learning

In the next Subsemble example, we modify the parameters to train a different type of Subsemble model. For this example, we set `learnControl[["multiType"]]` to `"divisor"`, which means we will ensemble a total of two models instead of four. In this example, a random forest is trained on the first subset, and a GLM is trained on the second subset. When using multiple learners, the "divisor" type of multi-learning will always be faster than cross-product multi-learning, so it can be used to get quick results.

We begin by training the Subsemble:

```
fit <- subsemble(Y=Y, X=X, newX=newX, family=binomial(),
                 learner = learner, metalearner = metalearner,
                 subsets = subsets,
                 learnControl = list(multiType="divisor"))
```

We then evaluate performance on a test set. In this example, the test set AUC is 0.922.

```
auc <- cvAUC(predictions=fit$pred, labels=newY)$cvAUC
print(auc)
```

4.4 Summary

The `R` package **subsemble** implements the general Subsemble prediction algorithm for creating an ensemble of subset-specific algorithm fits. In this chapter, we described the functionality of the `subsemble` function of the package, detailed our algorithm implementation and discussed its operations, and demonstrated the usage of `subsemble` through an example data set.

Chapter 5

Discussion

In this work, we introduced the flexible subset ensemble prediction method Subsemble. Subsemble partitions a full data set into subsets of observations, fits one or more underlying algorithm on each subset, and combines the subset-specific fits through a second metalearner algorithm using a clever form of V-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, and illustrated the desirable practical performance of Subsembles composed of randomly created subsets. We next proposed creating the subsets used in Subsemble through supervised partitioning of the covariate space. We described the computational advantages of this Supervised Subsemble approach, developed the practical SRT Subsemble algorithm to both construct the covariate partitioning and learn the optimal number of subsets, and demonstrated that the SRT Subsemble performs well in practice. Finally, we developed the R package **subsemble**, described the package and our algorithm implementation, and demonstrated the application of the package for prediction with an example data set.

There are many promising directions for future research with the Subsemble method. Study applying Subsemble, and specifically SRT Subsemble, to large-scale data sets would be a valuable area for future work. In particular, future research should explore more practical suggestions and concrete recommendations for implementing and using SRT Subsemble in practice on large-scale data sets. For example, with big data, it is probably impractical to build a very large tree. While we give some ideas in this work regarding stopping criteria, additional work should be done to determine improved stopping criteria. Determining the starting node is another interesting area for future work. That is, for very large data sets, starting at the root node is likely not feasible. As a result, future study should examine criteria for selecting this starting point.

Modifying the CART algorithm used in SRT Subsemble is another interesting area for future work. Rather than simply using the default CART algorithm to build a tree, and then exploring the Subsembles associated with the resulting sub-trees, future work should consider instead directly incorporating Subsembles into the CART procedure. For example, future work could consider using a different metric to determine the best covariate split, such as the associated Subsemble's cross-validated risk.

Characterizing when the SRT Subsemble method performs better and/or worse than Subsembles with randomly created subsets should also be the subject of future study. In this work, we provided preliminary intuition comparing the performance of these methods. However, thorough simulation studies should be conducted to provide further insight into the differences between the two methods for varying data generating distributions, underlying learner algorithms, and metalearner algorithms.

Developing other methods for creating the subsets used in Subsemble is another topic for future research. While the random subsets and SRT Subsemble methods presented in this work demonstrate desirable practical performance, there are certainly plenty of opportunities to explore alternative approaches. Comparing the relative statistical performance, run time, and computational properties of the varying subset creation techniques would also be of interest.

Finally, the **subsemble** R package should continue development, incorporating new research advances. For example, future work should implement the SRT Subsemble method within the **subsemble** package.

Bibliography

Bache, K. and M. Lichman (2013). *UCI Machine Learning Repository.* URL: `http://archive.ics.uci.edu/ml`.

Breiman, L. et al. (1984). *Classification and Regression Trees.* Monterey, CA: Wadsworth and Brooks.

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

— (1996b). "Stacked Regressions". In: *Machine Learning* 24.1, pp. 49–64.

Chu, Singfat (2001). "Pricing the C's of Diamond Stones". In: *Journal of Statistical Education* 9.2.

Freund, Yoav and Robert E. Schapire (1997). "A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting". In: *Journal of Computer and System Sciences* 55, pp. 119–139.

LeDell, Erin, Maya Petersen, and Mark van der Laan (2013). *cvAUC: Cross-Validated Area Under the ROC Curve Confidence Intervals.* R package version 1.0-0. URL: `http://CRAN.R-project.org/package=cvAUC`.

Lin, Jimmy and Alek Kolcz (2012). "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, pp. 793–804.

Polley, Eric and Mark van der Laan (2012). *SuperLearner: Super Learner Prediction.* R package version 2.0-9. URL: `http://CRAN.R-project.org/package=SuperLearner`.

Therneau, Terry, Beth Atkinson, and Brian Ripley (2012). *rpart: Recursive Partitioning.* R package version 4.1-0.

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

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

Wolpert, David H. (1992). "Stacked Generalization". In: *Neural Networks* 5.2, pp. 241–259.

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