Multi-stage heterogeneous ensemble meta-learning with hands-off user-interface and stand-alone prediction using principal components regression: The R package EnsemblePCReg

Mansour T.A. Sharabiani
School of Public Health
Imperial College London

Alireza S. Mahani
Scientific Computing Group
Sentrana Inc.

Abstract

Despite the fact that ensemble meta-learning of a heterogeneous collection of base learners is an effective means to reduce the generalization error in predictive models, several factors have impeded a broad adoption of such techniques among practitioners. These factors include an intractable number of choices of base learners and their tuning parameters, complex methodology required for integration of base learners, the ensuing complexity of software needed to support stand-alone prediction, and significant CPU and memory consumption of heterogeneous ensemble meta-learning techniques. The R package **EnsemblePCReg** overcomes the above barriers by combining several features. Sensible base-learner parameter grids provide a hands-off API for non-experts while allowing expert users to exert control by overriding default settings. Sophisticated ensemble generation and integration methods, combining stacked generalization and principal components regression, offer favorable generalization performance. Finally, computational optimizations such as advanded thread scheduling for improved parallelization scaling and file methods for relieving memory consumption during training and prediction, significantly increase the range of data sizes that can be handled on personal computers. In combining these features, **EnsemblePCReg** significantly lowers the barrier for practitioners to apply heterogeneous ensemble meta-learning techniques to their everyday regression problems.

Keywords: heterogeneous ensemble learning, ensemble meta-learning, stacked generalization, principal components regression.

1. Introduction

While ensemble learning of homogeneous base learners such as decision trees is widely used in algorithms such as random forests (RF) (Breiman 2001) and gradient boosting machines (GBM) (Friedman 2001), heterogeneous ensemble meta-learning (HEML) – i.e., combining base learners of different types into a single meta-learner – has not yet achieved broad, real-world adoption (Holiday 2012), and applications of HEML techniques such as stacked generalization (Wolpert 1992) have been mostly limited to data science competitions (Bell and Koren 2007). This is unfortunate because – by virtue of combining diverse and specialized learners of different types – HEML achieves a central goal of ensemble learning, i.e., weak correlation among strong constituents (Webb *et al.* 2004).

In applying HEML techniques to real-world problems, practitioners face several challenges:

1. *Ensemble generation:* First, one must choose among the myriad base learners available. For example, the R package **caret** (Kuhn *et al.* 2015) exposes 213 base learners in its API, as of version 6.0-62! Using all these base learners in a HEML algorithm can be not only computationally time-consuming but also methodologically questionable, since many of these base learners are slight variants of each other, with high correlation among their predictions. In addition, for each base learner the user must select its tuning parameter(s). Once we step beyond simple base learners such as GLM models, most non-parametric algorithms include numerous tuning parameters. For example, the Bayesian Additive Regression Trees (BART) (Chipman *et al.* 2010) implementation in **BayesTree** (Chipman and McCulloch 2014) contains at least 7 non-trivial tuning parameters: `sigdf, sigquant, k, power, base, ntree, ndpost`. A brute-force, exhaustive grid search – e.g., using a cross-validated performance metric – quickly becomes computationally infeasible, and choosing a single set of default values provided in the package is by no means an optimal decision.

2. *Ensemble integration:* Combining base learner predictions into a single ensemble prediction is another challenge in HEML algorithms. While a simple average works well for homogeneous ensemble learners such as RF or BART, same is not true for HEML algorithms which combine different base learners with potentially very different performance profiles. Naively applying a second-stage model (meta-learner) to base learners that are trained on the full training set is also dangerous because it rewards overfit learners that have excellent within-sample performance, at the expense of poor out-of-sample accuracy. Furthermore, as more base learners are added to the ensemble, the inevitable multicollinearity of base-learner predictions must be properly handled by the meta-learner, requiring some form of regularization or shrinkage, whose parameter must be selected. The need for sophisticated ensemble integration techniques in HEML, while maximizing data utilization and avoiding data contamination, creates significant methodological challenges.

3. *Stand-alone prediction:* A key shortcoming of many HEML-based models emerging from data science competitions has been their inability to produce stand-alone predictions: The price paid for the highly convoluted methodologies used in such models is that the ensemble model must be re-trained each time a new prediction is needed. This lengthy process, while not a serious problem during competitions where (unlabeled) test set is often provided to the participants at the outset, makes such models unfit for emerging applications such as cloud-based analytics services, or real-time recommender engines (Johnston 2012). Furthermore, the need to re-train for each new prediction means results cannot be reproduced by researchers due to the stochasticity of training process in many base learners.

4. *CPU and memory consumption:* Finally, HEML algorithms are computationally demanding, thanks to their need for training a large number of base learners, each of which can itself be an ensemble learner. This means not only high CPU usage, but also high memory consumption, especially if using tree-based base learners such as RF. It is easy for HEML algorithms to require tens of Gigabytes of memory, even for small data

sizes, thus exceeding the resources of many PC's and restricting their use to powerful servers and/or distributed clusters. Scheduling of base-learner training jobs on available threads is another complexity, thanks to the highly uneven training times across different base-learner types.

These challenges combine to put HEML techniques squarely within the niche of machine learning and high-performance computing experts, require allocation of significant computing resources, and incur significant delays for modeling projects. For many real-world applications with serious resource and time constraints, such requirements render HEML models impractical.

Such difficulties are reflected in the open-source software landscape. While numerous R packages exist for homogeneous ensemble learning, including **randomForest** (Liaw and Wiener 2002), **gbm** (Ridgeway 2015), **BayesTree** (Chipman and McCulloch 2014), **adabag** (Alfaro *et al.* 2013), **mboost** (Hothorn *et al.* 2016) and **subsemble** (LeDell *et al.* 2014) among many others, yet – to our knowledge – there are few open-source software options available for HEML: the R packages **caretEnsemble** (Mayer and Knowles 2015) and **SuperLearner** (Polley and van der Laan 2014), and the C++ library **ELF** (Jahrer 2010). While these packages have indeed reduced the barrier to using HEML by practitioners, yet there remains a significant gap between HEML and homogeneous ensemble learners in terms of broad usability.

EnsemblePCReg offers a unique set of features for overcoming the above-mentioned barriers to building HEML models, thus significantly broadening the reach and utility of such techniques for practitioners:

1. *Easy-to-use API:* In **EnsemblePCReg**, training a model is as simple as the standard single-line call used in many base learners implemented in various R packages. Similarly, prediction, diagnostics and visualization are all one-line API calls. Despite using a sophisticated methodology (see Section 2), prediction can be done without re-training the model, thus allowing **EnsemblePCReg** to be used for on-demand prediction, and making the results fully reproducible.

2. *Sophisticated ensemble generation and integration:* **EnsemblePCReg** combines two advanced techniques for ensemble generation (stacked generalization (Wolpert 1992)) and ensemble integration (rotating-partition principal components regression (Merz and Pazzani 1999)) into a two-stage process, which we refer to as the 'double-rotating-partition' – or DRP – framework. The result is maximal use of data while avoiding contamination and overfitting, thus leading to superior generalization performance, i.e., low error on test data.

3. *Computational optimizations:* Multicore parallelization using advanced thread scheduling policies during both training and prediction, along with memory optimization via file methods for saving/loading base learner training objects to/from disk, allow users to efficiently build ensemble models on PC's with limited processing power and memory.

The rest of this paper is organized as follows. In Section 2 we motivate and describe the ensemble generation and integration stages used in **EnsemblePCReg**. In Section 3 we present the computational optimization techniques utilized in **EnsemblePCReg** for efficient CPU and RAM utilization. Section 4 acts as a tutorial by presenting several examples that illustrate key features and use-cases of **EnsemblePCReg**. In Section 5 we further discuss several important topics raised in the paper, and offer concluding remarks.

2. Ensemble meta-learning in EnsemblePCReg

Ensemble learning consists of three stages (Mendes-Moreira *et al.* 2012): generation, pruning and integration. In ensemble *generation*, many models are generated using different base learners, tuning parameters, or data subsets. During *pruning*, a subset of these models are eliminated and the remainder is passed on to the *integration* stage, where they are combined using the meta-learner to form the final model. In **EnsemblePCReg**, the regularization effect of the PCA component of the integration algorithm (Section 2.2) allows us to absorb the pruning stage into the integration stage.

2.1. Ensemble generation using SG

The ideal outcome in ensemble generation is a collection of diverse – i.e., uncorrelated – and accurate base learners. Ensemble generation techniques broadly fall into two categories (Mendes-Moreira *et al.* 2012): 1) data manipulation techniques such as subsampling of training cases, random (feature) subspace (Ho 1998), and output smearing (Breiman 2000) among others; and 2) modeling process manipulation techniques such as changing tuning parameters and learning algorithm, and post-bagging (Jorge and Azevedo 2005). **EnsemblePCReg** uses elements from each category by combining models produced with different base learners, tuning parameters, and data partitions. Appendix A contains a list of base learners and their default parameter grids used during ensemble generation in **EnsemblePCReg**. Below, we focus on motivating and describing our data partitioning strategy.

In ensemble integration, the predictions of base learners – trained during the ensemble generation stage – are used as predictors in the meta-learning process. If we use the same data set for training both the base learners and the meta-learner in a naive manner, this can lead the meta-learner to favor base learners that overfit to the training set – i.e., those base learners whose within-sample predictions are the closest to the observed values – at the expense of out-of-sample performance. We refer to this as 'data contamination', suggesting that the data (or process) used for base-learner training has contaminated the data (or process) used by the meta-learner (Figure 1a).

A simple way to avoid data contamination is to split the training set into two subsets, using one for base-learner training and the other for meta-learner training (Figure 1b). Such simple setups, however, do not utilize the data efficiently, which can be especially problematic for small data sets. An improvement is to reverse the subset labels and repeat the process, using an average of the resulting two models as the final ensemble model (Figure 1c).

Wolpert (1992) proposed 'stacked generalization' (SG) to overcome data contamination. In SG, data is partitioned into – often equal or nearly-equal – subsets or folds. For each fold, base learners are trained on its 'complement', i.e. all cases not present in that fold, and predictions of the complement subset for the cases in the subset are collected. After this is repeated for all folds, the out-of-sample predictions are assembled to form a data set covering all original cases, and the resulting matrix is used as the feature matrix for meta-learning. This is illustrated – for two partitions – in Figure 1d. As Wolpert pointed out, cross-validated (CV) selection of a single base learner can be considered a special case of SG, where the meta-

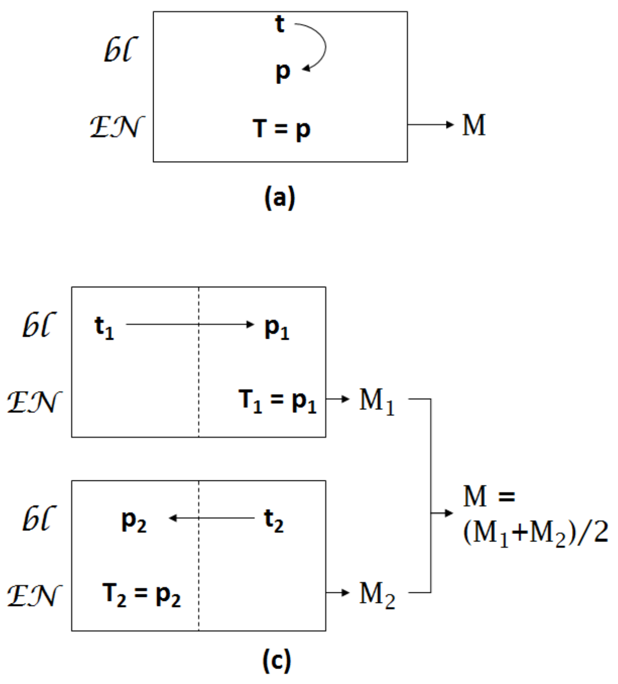

Figure 1: Progression of data-splitting strategies for minimizing data contamination while maximizing data utilization: a) Within-sample predictions of base learners (bl) are provided as predictors into the ensemble integration (EN) stage, causing 'contamination'; b) A simple scheme to avoid contamination, where half of data is used for training base learners and the other half is used for training the ensemble learner; c) Improvement upon (b), where the roles of two data subsets are reversed, producing two ensemble models which must be combined (e.g., by averaging); d) Rotating-Partition (RP) strategy, inspired by cross-validation, where out-of-sample base learner predictions are assembled into a full prediction for entire training set, followed by ensemble training of a single model.

learner simply selects the base learner corresponding to the meta-feature with smallest error. However, we have labeled the term 'rotating-partition' to described this pattern in order to clearly differentiate it from CV selection.

2.2. Ensemble integration using PCR*

In principal, any base learner – including non-parametric or ensemble learners – can be used as a meta-learner during ensemble integration. However, weighted averaging of base learner predictions is generally more robust:

$$f_E(\mathbf{x}) = \sum_{i=1}^{K} h_i(\mathbf{x}) \times f_i(\mathbf{x}), \tag{1}$$

where $\mathbf{x}$ is the feature vector, $f_i(.)$ is the prediction function for base learner i, $h_i(.)$ is the weighting function for base learner i, and $f_E(.)$ is the resulting prediction function for the ensemble meta-learner. Weighted-average models fall under constant and non-constant (dynamic) weight categories, with constant weights being more common, and the method used in **EnsemblePCReg**.

Any ensemble integration method must deal with the fact that base learner predictions can be strongly correlated, for two reasons: 1) Same base learner, trained with different tuning parameters, tends to produce somewhat similar models and predictions, and 2) Accurate base learners – by definition – are all correlated with the response variable, and hence with each other. Collinearity of base learners tends to become more pronounced as we increase the number of base learners, especially for small data sets, and can lead to an ensemble model that is overfit to the training data.

Many techniques exist for dealing with collinear predictors (Hastie *et al.* 2009), including forward-backward variable selection, regularized regression such as ridge (L2) and Lasso (L1), and Principal Components Regression (PCR). Regularized regression and PCR are similar in that they each have a tuning parameter that controls the degree of shrinkage applied to the coefficients of the resulting linear regression model. In ridge/Lasso, the tuning parameter controls the relative importance of the weight penalty term in the error function, while in PCR the tuning parameter is the number of principal components (ordered by eigenvalue) used in regression. While regularized regression has received more attention as an ensemble integration technique, PCR has the slight advantage that the tuning parameter has a natural grid, thus removing the need for careful grid construction to ensure that the optimal value is an interior point of the grid and adjusting computation time through grid resolution.

It must be noted that, as discussed in (Hastie *et al.* 2009, Section 3.4.1), there is a close connection between ridge regression and PCR. While PCR projects model predictions into the subspace spanned by the eigenvectors corresponding to the largest eigenvalues of the covariance matrix ($\mathbf{X}^T\mathbf{X}$):

$$\mathbf{X}\boldsymbol{\beta}^{\text{pcr}} = \sum_{j=1}^{P} \mathbf{u}_j \mathbf{u}_j^T \mathbf{y}, \qquad (2)$$

(where P denotes the number of principal components kept for regression) ridge regression applies a soft version of the PCR by weighting the contribution of each eigenvector ($\mathbf{u}$) with inverse proportion to its eigenvalue (d_j):

$$\mathbf{X}\boldsymbol{\beta}^{\text{ridge}} = \sum_{j=1}^{J} \mathbf{u}_j \, \frac{d_j^2}{d_j^2 + \lambda} \, \mathbf{u}_j^T \mathbf{y}, \qquad (3)$$

where λ is the regularization (or shrinkage) parameter in ridge regression. While this paper focuses on **EnsemblePCReg**, interested users can refer to **EnsemblePenReg** (Sharabiani and Mahani 2014) for a penalized-regression version of this **EnsemblePCReg**.

The number of components used in PCR is selected using the same RP pattern seen in SG (Section 2.1). Details are described in Merz and Pazzani (1999). We follow their terminology and refer to the PCR method embedded in the RP pattern as PCR*.

2.3. Combining SG and PCR*: The DRP framework

We discussed data contamination and multi-collinearity as two key challenges in heterogeneous ensemble meta-learning, and presented two techniques used in **EsnemblePCReg** for addressing these problems: SG and PCR*. **EnsemblePCReg** combines these techniques into a single learner that produces superior predictive performance compared to using each technique in isolation. The result is called the Double-Rotating-Partition (DRP) framework, referring to application of the rotating-partition (RP) pattern twice in succession, once for ensemble generartion, and once for ensemble integration. Next we present empirical evidence supporting the advantage of DRP over simpler alternatives, i.e., using RP only for generation (SG) or integration (PCR*) stages.

We used 22 publicly-available data sets, listed in Appendix C. Performance comparison of DRP against SG and PCR* is based on their out-of-sample performance on 112 random 70-30 splits of each data set into training and test sets. Default settings of `epcreg` were used. 'Generalization improvement' is defined as the ratio of error in the test set for either of the alternative models (SG and PCR*), divided by error of the DRP model. This number is calculated for each of the 112 partitions within each data set. Figure 2 shows a summary of results, plotting mean +/- 2 se of the metric for each data set. Data sets are sorted from left to right in order of increasing size, i.e., number of observations (matching the appendix table). We observe the following:

1. Using the RP pattern for generating base learners (DRP vs. PCR*) offers significant improvement across all data sizes.

2. Using the RP pattern during integration – to select the number of PC's in PCR – (DRP vs. SG) offers improvement for small data sets, and little to no improvement for large data sets.

3. DRP is rarely significantly worse than either alternative.

Given that embedding algorithms such as PCR or regularized regression in RP does not add significant time to overall training time – since base learner training consumes the majority of training time – it is a safe strategy to use DRP across all data sizes.

It must be noted that, the objective of this benchmarking test was not to prove that ensemble meta-learning is better than any individual base learner, though this will be briefly discussed in Section 4.1.

The price we pay for methodological superiority of the DRP framework is its complexity, and thus a modular implementation of the DRP framework – with train and predict functions decoupled – is highly non-trivial. Appendix B describes the approach used in **EnsemblePCReg** for composing the DRP framework from simpler components.

3. Computational optimizations in EnsemblePCReg

In addition to utilizing the parallelizable nature of training and predicting on base learners, **EnsemblePCReg** has two additional optimization features: advanced thread scheduling for improved parallelization speedup, and file methods for reducing RAM pressure. We discuss these two optimization techniques in this section, and provide examples of their usage in Section 4.3.

3.1. Advanced thread scheduling

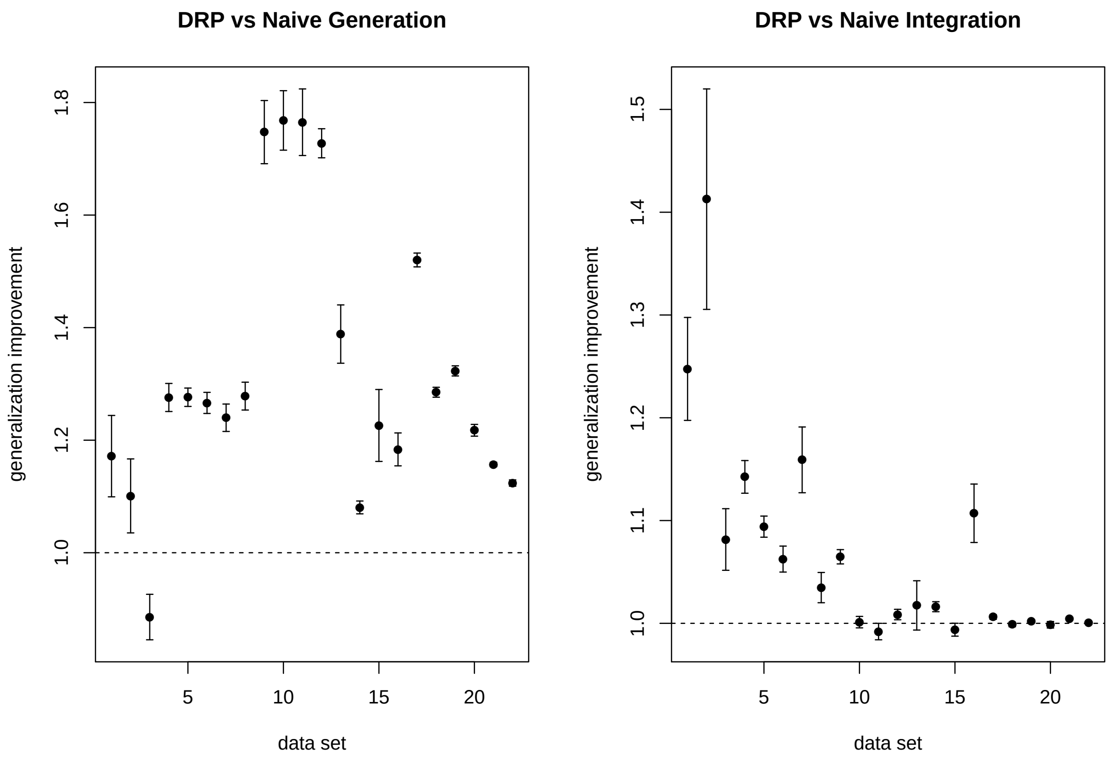

Figure 2: Improvement in generalization (out-of-sample) performance of DRP framework (using PCR for integration stage) compared to two alternative methods, for each of the 22 data sets in Table 2. Left: DRP is compared to an ensemble model that does not use the RP pattern for training the base learners, but is otherwise identical to DRP (Naive-BL). Right: DRP is compared to an ensemble model that does not use RP pattern for selecting the number of PC's in PCR-based integration (i.e. always using the maximum number of PC's which is equivalent to standard linear regression), but is otherwise identical to DRP (Naive-Ens). Each point is based on 112 random, 70-30 splits of data. Vertical bars indicate 2 times standard error.

Thread scheduling describes the process – dictated by a combination of the application and the operating system – of assigning parallelizable tasks to concurrent threads. In doing so, we seek the dual objectives of load balancing and parallel overhead minimization (Chandra 2001). Load balancing is an attempt to distribute tasks across threads such that total task durations are as even as possible. Otherwise, a weak link – i.e., a queue of tasks with long total execution time – will become the bottleneck and reduce parallelization speedup. At the same time, we would like to minimize the need for threads to synchronize their actions since this also imposes a performance hit.

There are two general categories of thread scheduling policies: static (or pre-scheduled) and dynamic. In static scheduling, tasks are assigned to threads before entering the parallel region, while in dynamic scheduling tasks are put in a single queue and grabbed by threads as they finish previous tasks. Compared to dynamic policies, static scheduling imposes lower thread synchronization costs since threads do not need to coordinate their action while fetching and executing tasks. However, if static task assigment is uneven, it can lead to load imbalance. Static vs. dynamic scheduling can be controlled via the `preschedule` flag in `epcreg` function. For both static and dynamic scheduling, **EnsemblePCReg** offers three flavors, accessed via the `schedule.method` argument: 1) `as.is`: In dynamic scheduling, this option leaves the job queue unchanged. In static scheduling, jobs are assigned to threads in a round-robin fashion; 2) `random`: This can be considered similar to `as.is`, but jobs are first shuffled randomly; 3) `task.length`: In dynamic scheduling, jobs are first sorted in decreasing order of expected duration (using the `task.length` argument), while in static scheduling, a simple but effective algorithm is used to assign jobs to threads, also based on expected durations, to achieve good load balance. (Interested readers can examine the source code for function `Regression.CV.Batch.Fit` in the file `baselearners.R`, from R package **EnsembleBase**, which contains the actual implementation of thread scheduling policies described above.)

Figure 3 compares the performance of various combinations of `preschedule` and `schedule.method` flags in parallel training of base learners for the `servo` data set (available in **EnsembleBase**), using 10 partitions of data for ensemble generation (`npart = 10`). We see that all three dynamic methods scale poorly as we increase the number of cores. This is a reflection of the small data size (167 observations), leading to small job durations, which in turn makes thread synchronization overhead quite comparable to them. Performance of `as.is` method in static scheduling is quite erratic, depending on the specific distribution of long tasks across threads. The `random` method performs better, only surpassed by the `task.length` method. We expect that, as the number of base learner instances is increased (e.g. by increasing `npart` or number of configurations per base learner), the performance of `random` method will approach that of `task.length`. Based on the above results, we have selected static scheduling with `random` method as default for both train and predict stages. See Section 4.3 for usage examples.

3.2. Memory optimization

Some base learners, especially ensemble, tree-based algorithms such as RF and GBM, produce large training objects. Multiplied by number of configurations per partition times number of partitions times number of folds per partition used in **EnsemblePCReg** (default: 16 x 1 x 5 = 80), this can lead to very large training objects produced by `epcreg`. For example, even for a small data set of 337 observations and 16 features, using `npart=50` with default base learners and configurations produces a ∼15GB object, approaching/exceeding the total RAM available on many PC's. A particular tuning parameter that contributes linearly towards estimation object size for forest learners such as RF, GBM and BART is the number of trees in the forest.

To overcome this RAM bottleneck, we have provided facilities for saving/loading `epcreg` objects to/from disk, during model training and prediction. This feature is activated by the `file.method` flag. Behind the scene, the software saves each base learner training object to a temporary file (after training for that instance is finished), removes the object from memory, and calls R's garbage collector to reclaim that space. During prediction, training objects are loaded from the temporary files as needed to produce base learner predictions. Also, special functions `epcreg.save` and `epcreg.load` are provided to save/load `epcreg` objects to/from permanent files for intersession continuity. In the above example, using file methods reduces the R object size corresponding to the trained model by a factor of 40x. See Section 4.4.

It must be noted that, executing training and prediction in parallel mode partially negates

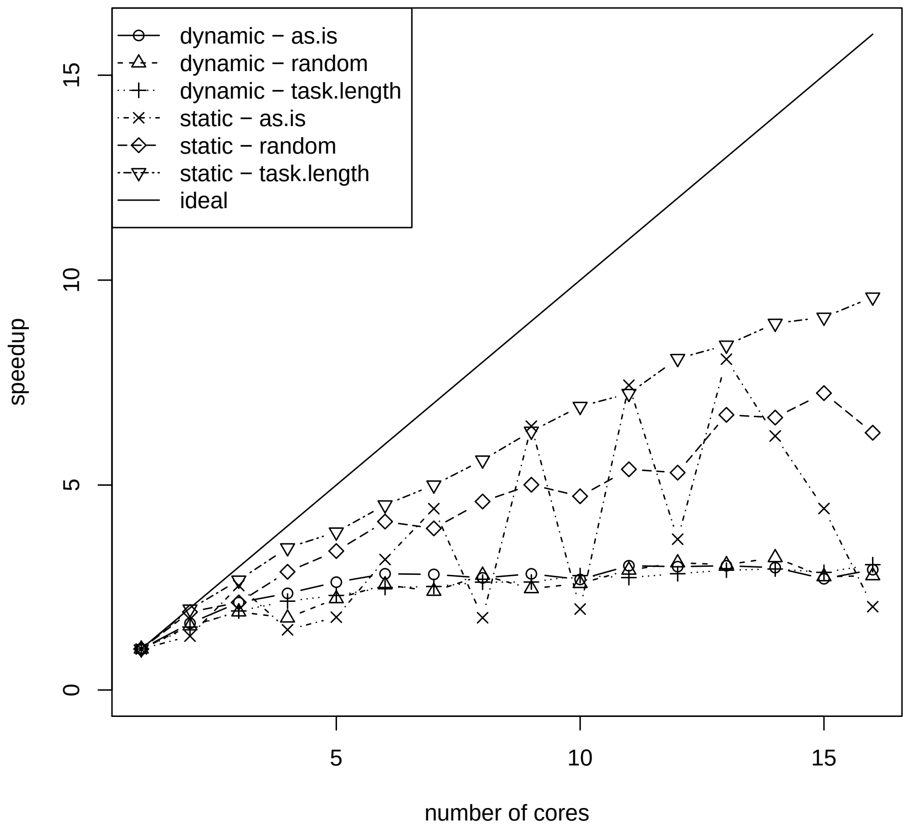

Figure 3: Impact of scheduling policy on scaling behavior of ensemble training as a function of number of parallel cores used. 'Dynamic' corresponds to `preschedule` flag set to `FALSE`.

the memory savings offered by the file method, since at any time as many base learner objects could be loaded into RAM as the number of parallel threads.

4. Using EnsemblePCReg

The package consists of two major components: a high-level user API, and a low-level developer API. In this tutorial we focus on the user API as it is more stable and the primary concern for the majority of users of **EnsemblePCReg**. The user API consists of the following components:

1. EML training (`epcreg`) and prediction (`predict.epcreg`)
2. Configuring base learners (`ecpreg.baselearner.control` and `make.configs`) and ensemble meta-learner (`epcreg.integrator.control`)
3. Diagnostics (`summary.epcreg`) and visualization (`plot.epcreg`)
4. Disk write I/O (`epcreg.save` and read (`epcreg.load`)

All these functions are quite intuitive, and their use is illustrated via examples in the remainder of this section.

4.1. Example 1: Using default settings

First, we load the package and a sample data set, and split it randomly into training and prediction sets.

```
R> library("EnsemblePCReg")
R> data(servo)
R> my.seed <- 0
R> set.seed(my.seed)
R> myformula <- class ~ motor + screw + pgain + vgain
R> perc.train <- 0.7
R> index.train <- sample(1:nrow(servo), size = round(perc.train*nrow(servo)))
R> data.train <- servo[index.train,]
R> data.predict <- servo[-index.train,]
```

Training the ensemble regression model is as simple as a one-line call to `epcreg` function:

```
R> est <- epcreg(myformula, data.train, print.level = 0)
```

Performance of base learners, as well as ensemble integrator step can be easily plotted (Figure 4):

```
R> plot(est)
```

The horizontal dotted line in the left panel indicates the final ensemble error, and corresponds to the bottom of the curve in the right panel. A few observations are worth mentioning:

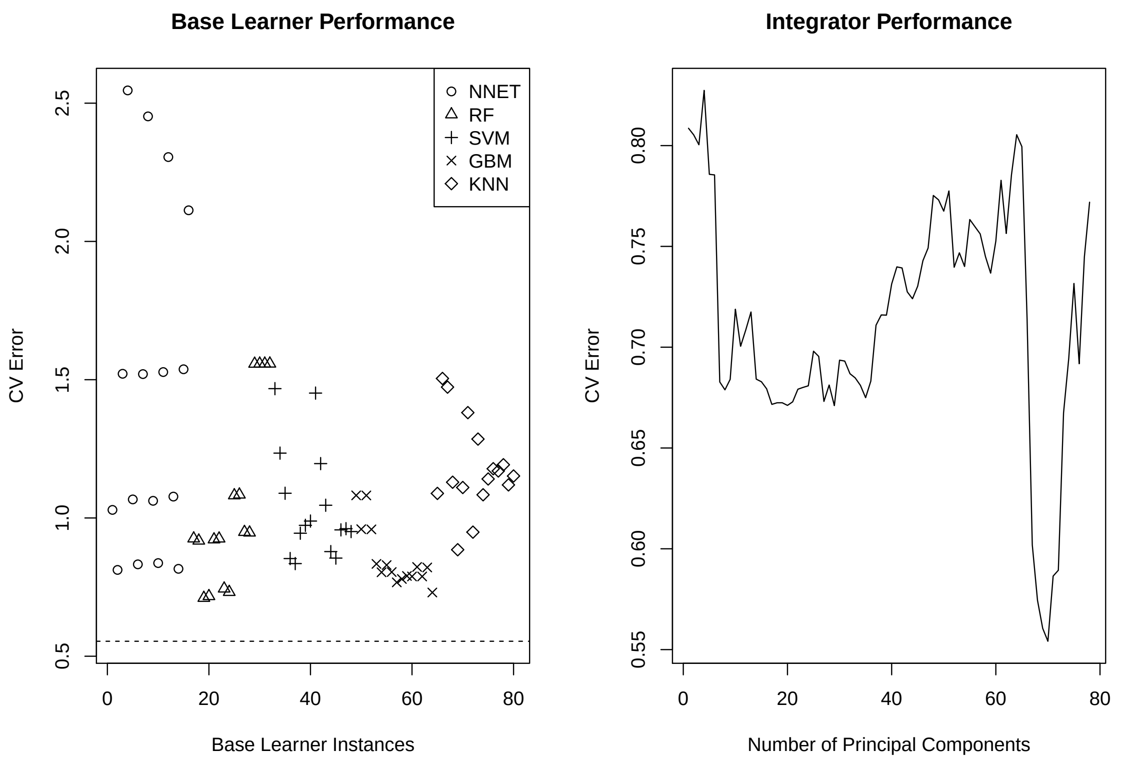

Figure 4: Performance of base learners (left) and PCR integrator (right), corresponding to the example discussed in Section 4.1.

1. As a whole, among the 5 base learners used, the two ensemble methods, i.e., random forest (RF) and gradient boosting machines (GBM), have the best cross-validated performance, compared to the 3 non-ensemble techniques, i.e., neural networks (NNET), support vector machines (SVM) and k-nearest neighbors (KNN). This confirms our premise that ensemble methods are generally superior in terms of prediction accuracy.

2. Within each base learner, CV performance of different sets of tuning parameters have significant spreads. This is especially true for NNET.

3. The PCR integrator (ensemble) outperforms all individual base learners – including ensemble base learners, RF and GBM – by a significant margin. However, looking at the plot of CV error vs. number of PC components (right panel), there could be cause for concern due to the sharp drop in error around the point of minimum error; this could indicate a small-sample fluke.

We can use the 30% hold-out sample to validate whether the superior performance of the ensemble method carries to data sets unseen by the algorithm. As shown in Figure 5, this is indeed the case. Overall, we see good correlation between CV and validation errors across base learners, although for instances with smaller errors, the correlation weakens.

The ensemble model, `est`, can be easily used for prediction on new data sets, using the usual R syntax:

```
R> newpred <- predict(est, data.predict)
R> cat("first 5 predictions:", head(newpred, 5), "\n")

first 5 predictions: 5.800048 0.675943 3.605809 0.7802629 0.6644396
```

The ability to re-use a trained ensemble model for new predictions, rather than requiring the prediction set to be available during training, allows for ensemble models to be trained, stored, and applied on demand, e.g., in response to streaming data, and without the significant delay imposed by the training process.

Overall, we see that **EnsemblePCReg** has successfully encapsulated and hidden all the comlexities involved in training and prediction for ensemble models, exposing a familiar and easy-to-use API for practitioners.

4.2. Example 2: Overriding default settings

Experienced users can exert more control over the model, including selection of base learners and their configuration grids, number of partitions, and number of folds per partition. These can be done via the utility function `epcreg.baselearner.control`, as illustrated next.

Looking at Fig. 4, we may consider excluding neural networks (NNET) from the ensemble, since its 16 instances have generally high CV errors. To test this hypothesis, we perform a comparison of ensemble performance, with and without NNET. We begin by creating two base learner control arguments, taking care to move NNET to the end of the pack so as to generate identical models for other base learners by fixing the random seed before each run:

```
R> baselearners.1 <- c("rf", "svm", "gbm", "knn", "nnet")
R> control.1 <- epcreg.baselearner.control(baselearners = baselearners.1)
```

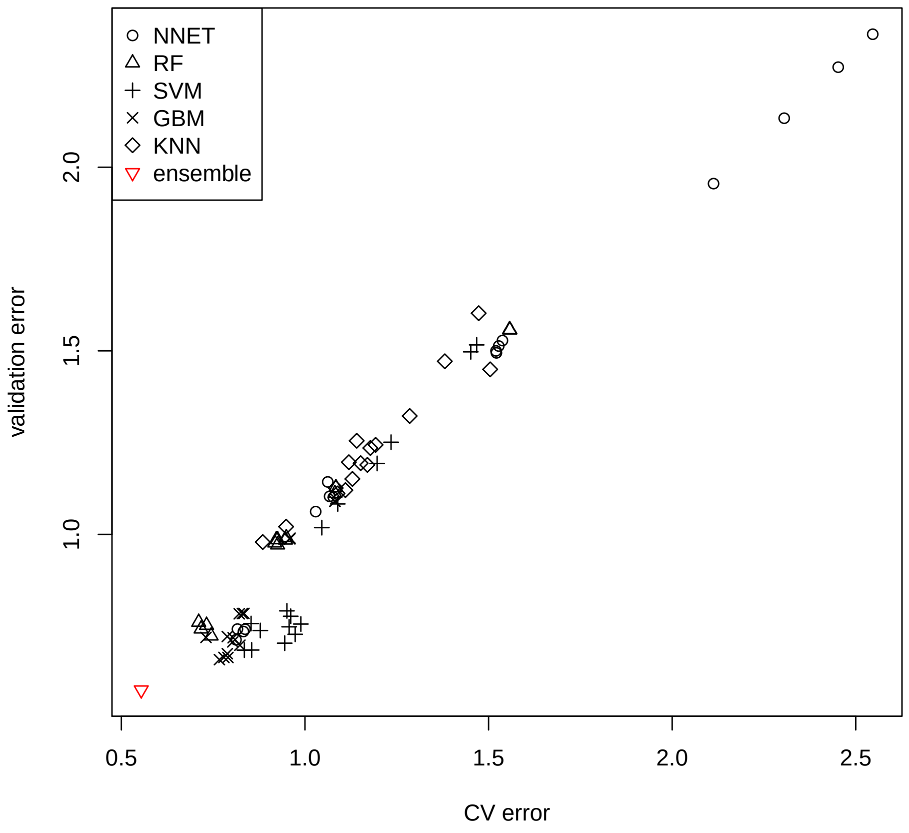

Figure 5: Comparison of CV error (70% of data) against validation error (30% of data) for base learners as well as ensemble model trained on `servo` data set.

```
R> baselearners.2 <- c("rf", "svm", "gbm", "knn")
R> control.2 <- epcreg.baselearner.control(baselearners = baselearners.2)
```

Next, we train ensemble models under both settings:

```
R> set.seed(my.seed)
R> est.1 <- epcreg(myformula, data.train, print.level = 0,
+    baselearner.control = control.1)
R> set.seed(my.seed)
R> est.2 <- epcreg(myformula, data.train, print.level = 0,
+    baselearner.control = control.2)
```

and summarize them:

```
R> summary(est.1)

number of base learner instances: 96
maximum number of PC's considered: 84
optimal number of PC's: 11
minimum error: 0.6543877

R> summary(est.2)

number of base learner instances: 80
maximum number of PC's considered: 72
optimal number of PC's: 49
minimum error: 0.6325347
```

We see that the CV error is slightly lower after removing NNET. Validation errors reveal an even more pronounced error improvement:

```
R> pcr.newerror.1 <- rmse.error(predict(est.1, data.predict), data.predict$class)
R> pcr.newerror.2 <- rmse.error(predict(est.2, data.predict), data.predict$class)
R> cat("validation error - all learners:", pcr.newerror.1, "\n")

validation error - all learners: 0.6150482

R> cat("validation error - all learners minus nnet:", pcr.newerror.2, "\n")

validation error - all learners minus nnet: 0.4776699
```

We see that, in this case, removing the bad-performing base learners from the ensemble meta-learner helped the out-of-sample performance, but this result cannot be relied upon without more extensive testing, especially due to the very small data size which can cause severe random fluctuations in performance depending on how the RNG is seeded. Weak learners can indeed improve ensemble performance if they are relatively uncorrelated with other, stronger learners. A reliable conclusion can only be drawn after mosr extensive testing of this and several other data sets.

Configuration grids for base learners can also be adjusted by overriding the default `baselearner.configs` argument passed to `epcreg.baselearner.control`. The `make.configs` collection of utility functions can be used for this purpose. For example, to change the `n.trees` parameter values in GBM (from their default value of `1000,2000`) to `500, 750`, we do:

```
R> my.configs.gbm <- make.configs(baselearner = "gbm"
+    , config.df = expand.grid(
+      n.trees=c(500, 750)
+      , interaction.depth=c(3,4)
+      , shrinkage=c(0.001,0.01,0.1,0.5)
+      , bag.fraction=0.5))
R> my.configs <- c(make.configs(baselearner = c("nnet", "rf", "svm", "knn")),
+    my.configs.gbm)
R> my.control <- epcreg.baselearner.control(baselearner.configs = my.configs)
```

Unless users have a solid reason to change default configuration grids, we recommend against doing so, as these default values have been created based on empirical results and literature review in order to induce sufficient diversity while maintaining a high likelihood of the optimal set of parameters being an interior point on the grid for each base learner.

Perhaps a more rewarding override of default settings is to increase the number of partitions for base learners (`npart` argument passed to function `epcreg.baselearner.control`), from the default value of `1`. This can even out random effects, especially in small data sets, and result in smoother, more reliable, PCR curves to be used in the integration step. This larger ensemble model can be easily estimated and visualized as follows:

```
R> control.npart <- epcreg.baselearner.control(npart = 10)
R> set.seed(my.seed)
R> t.npart <- proc.time()[3]
R> est.npart <- epcreg(myformula, data.train, print.level = 0,
+    baselearner.control = control.npart)
R> t.npart <- proc.time()[3] - t.npart
R> cat("training time:", t.npart, "\n")

training time: 360.69
```

As we see in Figure 6, integrator curve is somewhat smoother. It also appears that cross-validation best selection of a single base learner instance achieves lower error than the ensemble. However, validation set errors indciate that the ensemble model has superior performance, compared to the individual base learner that would have been selected by minimizing cross-validation error (see Figure 7).

4.3. Example 3: Parallelization and advanced thread scheduling

On our test machine (Intel Xeon E5-2670, 16 cores, 128GB of memory), training time was approximately 6 minutes for `npart=10`, depsite the fact that `servo` is a very small data

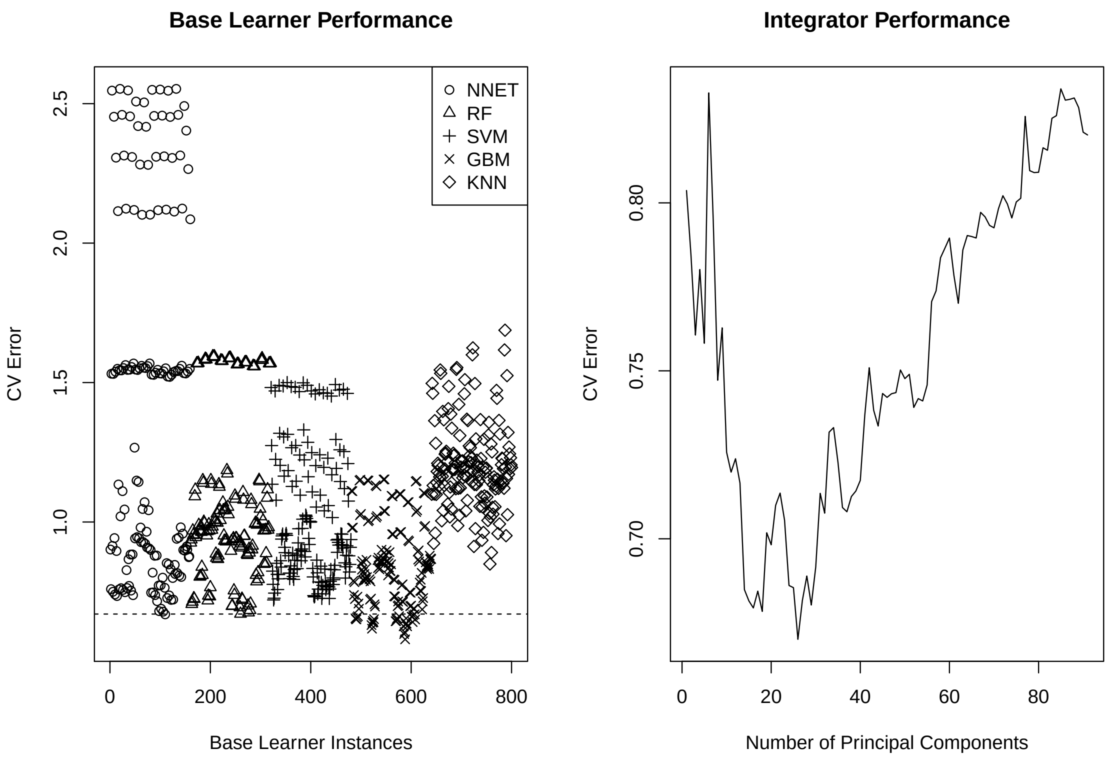

Figure 6: Base learner and integrator performance for `servo` data set, using `10` partitions instead of the default value of `npart=1`.

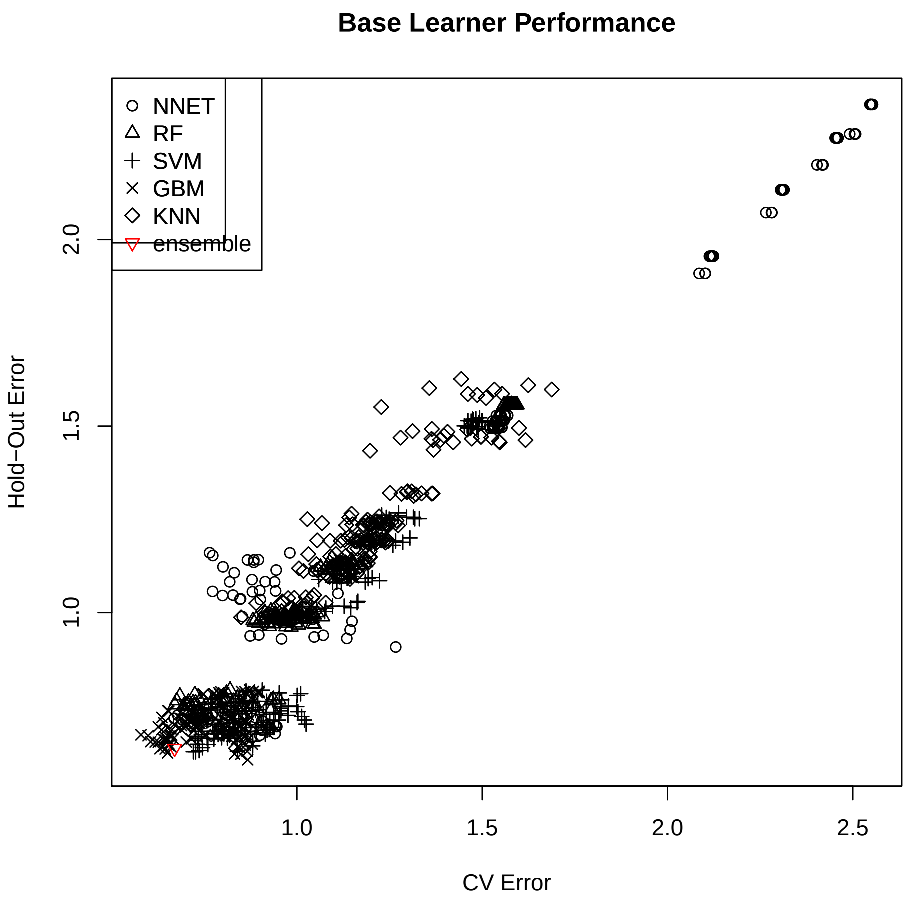

Figure 7: Comparison of CV and validation errors for base learners and ensemble model, for `servo` data set, using `10` partitions. See example in Section 4.2.

set (117 observations and 4 variables in training set). Independence of base learners offers an obvious parallelization opportunity, which can be exploited via the `ncores` argument of `epcreg` function:

```
R> t.npart.par <- proc.time()[3]
R> est.npart.par <- epcreg(myformula, data.train, print.level = 0,
+    baselearner.control = control.npart, ncores = 4)
R> t.npart.par <- proc.time()[3]
R> cat("training time - parallel:", t.npart.par, "\n")
R> cat("\tspeedup:", t.npart / t.npart.par, "\n")
```

```
training time - parallel: 117.189

        speedup: 3.077849
```

Same parameter can be used to parallelize the `predict` function. While absolute times for prediction are much smaller than training times, further reducing prediction times can be highly valuable for online / streaming analytics.

It must be noted that some base learners such as RF offer their own parallelization opportunities (e.g. due to complete independence of tree-building process in a random forest), but the outer parallelization is more generally applicable, and also likely to be more efficient since it is at a coarser level, i.e., each parallel job lasts long enough to amortize parallelization overhead. The parallelization functionality is made available via the `EnsembleBase` package, and is based on the multi-threading implementation of the R package **doParallel** (Analytics and Weston 2015).

A 3.1x speedup is decent (and an improvement over the naive scheduling approach, using `as.is` for `schedule.method`) but significantly below the ideal 4x when using 4 cores. As discussed in Section 3.1, this can be improved by using advanced thread scheduling techniques. To do so, we extract the vector of base learner training times from the estimation object using the `summary` function, and pass this as the `task.length` parameter to `epcreg` while setting the `schedule.method` parameter to `"task.length"`:

```
R> tvec <- summary(est.npart)$tvec
R> t.npart.par.opt <- proc.time()[3]
R> est.npart.par.opt <- epcreg(myformula, data.train, print.level = 0,
+    baselearner.control = control.npart, ncores = 4
+    , schedule.method = "task.length"
+    , task.length = tvec)
R> t.npart.par.opt <- proc.time()[3] - t.npart.par.opt
R> cat("training time - parallel & task-length-based load balance:"
+      , t.npart.par.opt, "\n")
R> cat("\tspeedup:", t.npart / t.npart.par.opt, "\n")
```

```
training time - parallel & task-length-based load balance: 101.293

        speedup: 3.560858
```

We see a meaningful improvement in performance by using estimated task lengths for thread scheduling. This improvement could have been more dramatic of we had fewer tasks (e.g., if using fewer partitions).

4.4. Example 4: Using file methods

As mentioned in Section 3.2, ensemble models are not only time-consuming, but also memory-consuming. **EnsemblePCReg** provides a functionality for relieving this RAM pressure by saving base learner estimation objects to temporary files, using R's `tmpfile()` service. This functionality can be activated by setting the argument `filemethod` to `TRUE`. In this case, the trained object returned by `epcreg` contains the temporary file information holding the actual base-learner trained objects (on disk). Upon calling `predict` against the trained object, these base-learner estimation objects are loaded from disk, used for prediction, and discarded afterwards. As usual, using this feature is straightforward:

```
R> est.filemethod <- epcreg(myformula, data.train, print.level = 0
+    , filemethod = TRUE)
```

We can compare estimation object size with and without file methods:

```
R> size.mb <- object.size(est) / 1024^2
R> size.filemethod.mb <- object.size(est.filemethod) / 1024^2
R> cat("object size - without file method:", size.mb, "\n")
R> cat("object size - with file method:", size.filemethod.mb, "\n")

object size - without file method: 175.3192 MB

object size - with file method: 3.791229 MB
```

Estimation objects created using file method can be saved to / load from disk via the special utility functions provide in the package: `epcreg.save` and `epcreg.load`. Same functions can also be used without file methods, allowing for a single call to handle both cases.

5. Discussion

In this paper, we presented the R package **EnsemblePCReg** for heterogeneous ensemble meta-learning of regression models. The package offers a hands-off solution via a unique combination of easy-to-use API, sophisticated EML methodology, and advanced computational optimization techniques. This combination significantly lowers the barrier for practitioners to apply HEML techniques to real-world problems. We close out this paper by comparing **EnsemblePCReg** to alternative open-source software for HEML, and discussing some of the limitations and development opportunities of our package.

5.1. Alternative EML software

EnsemblePCReg offers advantages over existing HEML software along its three design objectives: 1) hands-off solution, 2) advanced HEML methodology, and 3) computational efficiency.

Hands-off solution: As we saw in Section 4.1, training a regression HEML model in **EnsemblePCReg** is as simple as 1) installing and loading the package in R, and 2) calling `epcreg` while leaving all other parameters at default values. In contrast, in **caretEnsemble** (Mayer and Knowles 2015) one must 1) choose a subset of the 213 base learners available in **caret** (Kuhn *et al.* 2015), 2) for each base learners, define a tuning-parameter grid, 3) either prune the grid for each base learner via cross-validation in **caret** or train each base learner on all grid points, and 4) pass the trained objects to `caretEnsemble` or `caretStack`. Aside from having to write many more lines of code, the burden of choosing base learners and their tuning-parameter grids is often enough to turn away many users. In **SuperLearner** (Polley and van der Laan 2014), there is even more effort needed for implementing multiple wrappers around the same base learner to override default values of tuning parameters. The C++ library **ELF** adds further steps to the process, by requiring a detailed project setup. Also, being a standalone console application, **ELF** it does not benefit from being part of the R ecosystem (Section 5.3).

Our focus on delivering a hands-off solution in **EnsemblePCReg** comes at a price: We offer a smaller selection of base learners, and expose a subset of tuning parameters in the grids. This decision is driven by several factors: First, adding more base learners would require choosing sensible default parameter grids for each one. This requires extensive benchmarking work, as we have done for the current set of base learners. Secondly, adding new base learners must be done carefully and with consideration of similarity between them and existing base learners. If we emphasize a particular class of base learners in our mix, the ensemble will be biased towards their predictions, which may not be a desirable outcome if the dominant base learners are not among the best performers. Finally, adding more base learners would have diminishing returns, and the current set of diverse base learners in **EnsemblePCReg** is likely to capture the majority of benefits from ensemble meta learning. That being said, the limited choice of base learners in **EnsemblePCReg** compared to alternatives such as **caret** means that the latter package is more suitable for studying and comparing performance of specific base learners, while **EnsemblePCReg** must be used for efficiently generating accurate HEML models.

Advanced EML methodology: Another unique feature of **EnsemblePCReg** is utilizing a robust HEML technique – the DRP framework – with excellent generelization performance, while avoiding the need for re-training the model prior to prediction tasks. Consider the **caretEnsemble** package. It accepts a list of base-learner / tuning-parameter combinations and submits each one to the `train` function of **caret**, which in turn trains each combination on the full training set, before being passed to the meta-learner used in **caretEnsemble**. This means there is no RP pattern used during ensemble generation which – as we saw in Section 2.3 – has a significant disadvantage compared to the DRP method in terms of predictive accuracy. The C++ library **ELF** also does not use RP for ensemble generation. On the other hand, **SuperLearner** uses RP for ensemble generation, but not in ensemble integration. (using RP for integration has been mentioned as an option in the accompanying paper (Van der Laan, Polley, and Hubbard 2007).) Instead, it relies on weight constraints to control any potential overfitting caused by muticollinearity of base learners.

Computational efficiency: While most ensemble software – homogeneous or heterogeneous – take advantage of obvious parallelization opportunities available in ensemble learning, yet none tackled the load imbalance problem, which is acute in HEML algorithms due to the highly non-uniform duration of learning jobs. As we saw in Section 3.1, using estimated task durations for thread scheduling leads to much better load balance and performance scaling in multi-core parallelization, and **EnsemblePCReg** is unique in taking advantage of this optimization strategy. Similarly, memory optimization via file methods is another unique feature of **EnsemblePCReg**.

5.2. Extending EnsemblePCReg to classification

EnsemblePCReg is dedicated to regression problems, i.e., when the response variable is continuous. A significant contribution would be to extend the package (or create new packages) for classification (binary and categorical response) as well as other domains such as survival analysis (time-to-event response with potential censoring). The DRP framework used for regression in **EnsemblePCReg** is likely to prove valuable for other domains as well. Choice of the particular ensemble integration technique is one area of difference, as PCR is naturally suited for regression. For classification, an alternative is regularized (logistic or multinomial) regression. Another area of work is default tuning-parameter grids for classification base learners. While many packages and techniques used for regression can also target classification problems – in fact, many were designed originally for classification – yet their tuning parameters for classification has significant differences from regression.

5.3. Advantages and limitations of R

Choosing R as the implementation language for the **Ensemble** collection was driven by several factors: First, it provides ready access to open-source machine learning libraries in R's package ecosystem. This allowed us to avoid 're-inventing the wheel', and focus on novel aspects of our framework. Secondly, R's friendly syntax for matrix and vector manipulation provides for speedy implementation of our packages. Thirdly, the language enjoys a rich interface with compiled code such as C/C++ and Fortran, allowing developers to offload computationally-demanding parts of an algorithm to native code, often leading to significant speedup over interpreted R scripts. Finally, R provides automatic cross-platform compatibility for our package, making it accessible to all major platforms such as Linux, Windows and Mac. These advantages made R an ideal environment for a first implementation of our framework.

However, there are also important limitations to the current implementation in R. First, the parallelization overhead in R is significantly more than its equivalent in C, thus limiting parallelization speedup, especially for collections of small tasks. Secondly, R is inherently inefficient in handling memory, e.g., due to data duplication and automatic garbage collection. Thirdly, using high-level API calls provided by other packages for base learner training prevents certain types of optimization, such as data locality, NUMA-awarness and memory mapping to reduce total RAM usage and relieve pressure on L3 cache (Mahani and Sharabiani 2015b). Finally, support for object-oriented programming in R is limited, even with S4 classes.

References

Alfaro E, *et al.* (2013). "adabag: An R Package for Classification with Boosting and Bagging." *Journal of Statistical Software*, **54**(2), 1–35. URL http://www.jstatsoft.org/v54/i02/.

Analytics R, Weston S (2015). *doParallel: Foreach Parallel Adaptor for the 'parallel' Package*. R package version 1.0.10, URL http://CRAN.R-project.org/package=doParallel.

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

Bell RM, Koren Y (2007). "Lessons from the Netflix prize challenge." *ACM SIGKDD Explorations Newsletter*, **9**(2), 75–79.

Breiman L (2000). "Randomizing outputs to increase prediction accuracy." *Machine Learning*, **40**(3), 229–242.

Breiman L (2001). "Random forests." *Machine learning*, **45**(1), 5–32.

Chandra R (2001). *Parallel programming in OpenMP*. Morgan Kaufmann.

Chipman H, McCulloch R (2014). *BayesTree: Bayesian Additive Regression Trees*. R package version 0.3-1.2, URL https://CRAN.R-project.org/package=BayesTree.

Chipman HA, *et al.* (2010). "BART: Bayesian additive regression trees." *The Annals of Applied Statistics*, pp. 266–298.

Friedman J, Hastie T, Tibshirani R (2010). "Regularization paths for generalized linear models via coordinate descent." *Journal of statistical software*, **33**(1), 1.

Friedman JH (2001). "Greedy function approximation: a gradient boosting machine." *Annals of statistics*, pp. 1189–1232.

Hastie T, *et al.* (2009). *The elements of statistical learning 2nd edition*. New York: Springer.

Hechenbichler KSK (2015). *kknn: Weighted k-Nearest Neighbors*. R package version 1.3.0, URL http://CRAN.R-project.org/package=kknn.

Ho TK (1998). "The random subspace method for constructing decision forests." *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, **20**(8), 832–844.

Holiday R (2012). "What the Failed $1M Netflix Prize Says About Business Advice." http://www.forbes.com/sites/ryanholiday/2012/04/16/what-the-failed-1m-netflix-prize-tells-us-about-business-advice/. Accessed: 2016-01-01.

Hornik K, *et al.* (1989). "Multilayer feedforward networks are universal approximators." *Neural networks*, **2**(5), 359–366.

Hothorn T, *et al.* (2016). *mboost: Model-Based Boosting*. R package version R package version 2.6-0, URL http://CRAN.R-project.org/package=mboost.

Jahrer M (2010). "ELF Ensemble Learning Framework." http://mloss.org/software/view/260/.

Johnston C (2012). "Netflix Never Used Its $1 Million Algorithm Due To Engineering Costs." http://www.wired.com/2012/04/netflix-prize-costs/. Accessed: 2016-06-03.

Jorge AM, Azevedo PJ (2005). "An experiment with association rules and classification: Post-bagging and conviction." In *Discovery science*, pp. 137–149. Springer.

Kapelner A, Bleich J (2014). "bartMachine: Machine Learning With Bayesian Additive Regression Trees." *ArXiv e-prints.*

Kuhn M, *et al.* (2015). *caret: Classification and Regression Training.* R package version 6.0-62, URL `http://CRAN.R-project.org/package=caret`.

LeDell E, *et al.* (2014). *subsemble: An Ensemble Method for Combining Subset-Specific Algorithm Fits.* R package version 0.0.9, URL `http://CRAN.R-project.org/package=subsemble`.

Liaw A, Wiener M (2002). "Classification and Regression by randomForest." *R News*, **2**(3), 18–22. URL `http://CRAN.R-project.org/doc/Rnews/`.

Mahani AS, Sharabiani MT (2015a). *EnsembleBase: Extensible Package for Parallel, Batch Training of Base Learners for Ensemble Modeling.* R package version 0.7.2.

Mahani AS, Sharabiani MT (2015b). "SIMD parallel MCMC sampling with applications for big-data Bayesian analytics." *Computational Statistics & Data Analysis*, **88**, 75–99.

Mayer ZA, Knowles JE (2015). *caretEnsemble: Ensembles of Caret Models.* R package version 1.0.0, URL `http://CRAN.R-project.org/package=caretEnsemble`.

Mendes-Moreira J, *et al.* (2012). "Ensemble approaches for regression: A survey." *ACM Computing Surveys (CSUR)*, **45**(1), 10.

Merz CJ, Pazzani MJ (1999). "A principal components approach to combining regression estimates." *Machine learning*, **36**(1-2), 9–32.

Meyer D, *et al.* (2015). *e1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien.* R package version 1.6-7, URL `http://CRAN.R-project.org/package=e1071`.

Polley E, van der Laan M (2014). *SuperLearner: Super Learner Prediction.* R package version 2.0-15, URL `http://CRAN.R-project.org/package=SuperLearner`.

R Core Team (2016). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. URL `https://www.R-project.org/`.

Ridgeway G (2015). *gbm: Generalized Boosted Regression Models.* R package version 2.1.1, URL `http://CRAN.R-project.org/package=gbm`.

Samworth RJ, *et al.* (2012). "Optimal weighted nearest neighbour classifiers." *The Annals of Statistics*, **40**(5), 2733–2763.

Sharabiani MT, Mahani AS (2014). *EnsemblePenReg: Extensible Classes and Methods for Penalized-Regression-based Integration of Base Learners.* R package version 0.6, URL `http://CRAN.R-project.org/package=EnsemblePenReg`.

Smola A, Vapnik V (1997). "Support vector regression machines." *Advances in neural information processing systems*, **9**, 155–161.

TBD, *et al.* (2013). "KEEL Data-Mining Software Tool: Data Set Repository, Integration of Algorithms and Experimental Analysis Framework." URL `http://sci2s.ugr.es/keel/datasets.php`.

Tibshirani R (1996). "Regression shrinkage and selection via the lasso." *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288.

Torgo L (2013). "Regression Data - LIAAD." URL `http://www.dcc.fc.up.pt/~ltorgo/Regression/DataSets.html`.

Van der Laan MJ, Polley EC, Hubbard AE (2007). "Super learner." *Statistical applications in genetics and molecular biology*, **6**(1).

Venables WN, Ripley BD (2002). *Modern Applied Statistics with S*. Fourth edition. Springer, New York. ISBN 0-387-95457-0, URL `http://www.stats.ox.ac.uk/pub/MASS4`.

Webb G, *et al.* (2004). "Multistrategy ensemble learning: Reducing error by combining ensemble learning techniques." *Knowledge and Data Engineering, IEEE Transactions on*, **16**(8), 980–991.

Wolpert DH (1992). "Stacked generalization." *Neural networks*, **5**(2), 241–259.

A. List of base learners and tuning parameters

EnsemblePCReg imports base learners from the **EnsembleBase** package (Mahani and Sharabiani 2015a) (co-developed by the authors). Currently, seven base learners are available in **EnsembleBase**, each based on an existing implementation in R (Table 1) and thinly wrapped in a uniform interface: 1) random forests (RF) (Breiman 2001), 2) gradient boosting machines (GBM) (Friedman 2001), 3) feedforward neural networks (NNET) (Hornik *et al.* 1989), 4) support vector regression machines (SVM) (Smola and Vapnik 1997), 5) K-nearest neighbors (KNN) (Samworth *et al.* 2012), 6) penalized (L1/L2) regression (PENREG) (Tibshirani 1996), and 7) Bayesian additive regression trees (BART) (Chipman *et al.* 2010).

A subset of the tuning parameters for each base learner are deemed 'configurable' and exposed to the user (Table 1). Various combinations of values for these tuning parameters can be formed to create a configuration set, i.e., a multi-dimensional collection of tuning-parameter combinations used for training each base learner. By default, a grid of 16 points is created for each base learner. Definition of these default configuration grids can be seen by typing `?make.configs` in an R console (after loading **EnsembleBase**). The configurable subset of tuning parameters as well as the default grid for ech base learner are chosen so as to 1) cover the likely optimal combination in most problems, and 2) induce diversity across different configurations. Experienced users can override the default settings to define their own grids, and select a subset of available base learners. By default, 6 of the seven base learner are included (`bart` not included due to its generally-long training time), and each one is assigned a 16-point grid, bringing the total number of models generated during the first step to 6 x 16 = 96.

B. Composing the DRP framework using learner templates

Algorithm	R package	Configuration parameters
Neural Network	**nnet** (Venables and Ripley 2002)	weight decay hidden-layer size maximum iterations
Support Vector Machine	**e1071** (Meyer *et al.* 2015)	cost of constraints violation epsilon in insensitive-loss function kernel type
K-Nearest Neighbors	**kknn** (Hechenbichler 2015)	number of neighbors kernel type
Random Forest	**randomForest** (Liaw and Wiener 2002)	number of trees multiplier of `mtry` minimum size of terminal nodes
Gradient Boosting Machine	**gbm** (Ridgeway 2015)	number of trees maximum interaction depth shrinkage bagging fraction
Penalized Regression	**glmnet** (Friedman, Hastie, and Tibshirani 2010)	Relative weight of L1 vs. L2 loss shrinkage parameter
Bayesian Additive Regressio Trees	**bartMachine** (Kapelner and Bleich 2014)	num_trees k q nu

Table 1: List of base learners wrapped in **EnsembleBase** and imported by **EnsemblePCReg**, along with their R implementation, and the subset of tuning parameters that are configurable. For definition of parameters and other details, see documentation for **EnsembleBase** and base learner packages.

Interfaces implemented by predictive models can be classified into two groups, according to whether they offer an integrated or a stand-alone prediction functionality.

Using the R (R Core Team 2016) syntax for illustration, an interface with integrated prediction can be described as:

```
newpred <- model(X, y, newX)
```

where `model` refers to a learner, `X` is the training feature matrix, `y` is the training response vector, `newX` is the feature matrix for test cases, and `newpred` is the vector of predictions for test cases. Such models are especially attractive for data science competitions, where `newX` is often available at the outset, and where the complex ensemble techniques do not easily lend themselves to a decoupling of train and predict functions.

On the other hand, an interface with stand-alone prediction (SAP) has the following form:

```
est <- model.train(X, y)
newpred <- model.predict(est, newX)
```

where the predictive `model` has been decoupled into `train` and `predict` functions. In addition to a more efficient prediction process, such interfaces enhance the reproducibility of results. This is because for many base learners, the training process is stochastic, i.e., it produces a – perhaps slightly – different trained model, if starting with a different random seed. Therefore, if the only way to make a prediction is to re-train the model, we can never guarantee the same results in two runs. (Prediction process is, in contrast, often non-stochastic, given a trained model.)

```
model_A(train_A, predict_A)
model_B(train_B, predict_B)

model_AthenB(train_AthenB
  , predict_AthenB):

train_AthenB <- function(X, y) {
  est_A <- train_A(X, y)
  est_B <- train_B(predict_A(est_A, X), y)
  return (list(A = est_A, B = est_B))
}

predict_AthenB <- function(est, newX) {
  pred_A <- predict_A(est$A, newX)
  return (predict_B(est$B, pred_A))
}
```

Figure 8: A prototypical pipeline pattern for chaining two models, `A` and `B`, implementing the SAP interface. The prediction matrix from `A` is used as feature matrix in `B`. The design pattern illustrates how the the composite model, `AthenB`, implements the SAP interface, `(train_AthenB, predict_AthenB)`, using the implementations of the two constituent models, `(train_A, predict_A)` and `(train_B, predict_B)`. R syntax is used for illustration.

With the above in mind, we can state the key software design goals for reproducible ensemble learning as follows:

1. Define SAP lego blocks, i.e. small, atomic operations (such as base learners) that decouple prediction from training.
2. Assemble these lego blocks using patterns or templates that preserve SAP property: This can be accomplished by defining the template in terms of how it generates the train and predict functions of the composed learner from its constituent learners' train and predict functions.

Note that composing complex learners from simpler ones in the above approach is recursive, i.e., composite learners can be embedded in templates and recombined to form even more complex learners. Next, we describe the templates that are needed to compose the DRP framework.

Pipeline template: This is the most fundamental and ubiquitous template for connecting operations in analytic workflows. It roughly means that 'the output of the first operation is used as input into the second operation'. This is a broad definition, since the exact way in which the second stage consumes the output of the first stage is not specified. A useful, prototypical case is where the first stage produces a matrix prediction, and this matrix is used as the feature matrix (`X`) for the second stage. More formally, consider models A and B, each having the SAP property. Our prototypical pipeline template defines a composite model `AthenB` with the SAP implementation shown in Figure 8.

```
model_A(train_A, predict_A)

model_ADP(train_ADP, predict_ADP):

train_ADP <- function(X, y) {
  pred <- double(length(y))
  for (k in 1:K) {
    est <- train_A(X[part != k, ],
      y[part != k])
    pred[part == k] <- predict_A(est,
      X[part == k, ])
  }
  est_full <- train_A(X, y)
  return (list(est = est_full,
    pred = pred))
}

predict_ADP <- function(est,
  newX = NULL) {
  if (is.null(newX)) return (est$pred)
  return (predict_A(est$est, newX))
}
```

Figure 9: Rotating-Partition (RP) design pattern: When applied to a SAP base learner, `A`, it produces a composite SAP learner, `ADP`. The pattern assumes that a partition vector, `part` is defined, with `K` folds. This `R` pseudo-code assumes that the predict method for `A` returns a vector, but it can be generalized to other cases, e.g., when it returns a matrix.

Rotating-Partition (RP) Pattern: In Section 2, we discussed stacked generalization as a technique for overcoming data contamination, and PCR* as a technique for dealing with multicollinearity of base learner predictions. These two techniques share the same pattern of rotating-partitions for doing the seemingly impossible: producing out-of-sample predictions on the training data. When applied to a SAP base learner, the RP pattern produces a composite SAP learner. Figure 9 presents a formal definition of the RP pattern. The RP pattern is the central piece of the DRP framework for ensemble regression.

Batch Pattern: The batch pattern does what the name suggests: It batches together a collection of models and generates a composite model. There are no interactions between models in the batch pattern. In the DRP framework, the batch pattern is used during the generation stage. The definition is listed in Figure 10.

Select Learner: This learner also performs a simple task: it selects the feature – from the columns of its feature matrix – that is the closest to the response vector, i.e., it chooses the feature vector with the smallest error. The definition is listed in Figure 11.

DRP Framework: Having developed all the necessary SAP lego blocks, we can now assemble them into an end-to-end ensemble regression framework, referred to as the DRP framework. The resulting composite SAP learner is illustrated in Figure 12. This figure clearly illustrates

```
model_<n>(train_<n>, predict_<n>),
  n = 1, ..., N

model_Batch(train_Batch, predict_Batch):

train_Batch <- function(X, y) {
  est <- list()
  for (n in 1:N) {
    est[[n]] <- train_<n>(X, y)
  }
  return (est)
}

predict_Batch <- function(est, newX) {
  pred <- matrix(nrow = nrow(newX)
    , ncol = N)
  for (n in 1:N) {
    pred[, n] <- predict_<n>(est[[n]]
      , newX)
  }
  return (pred)
}
```

Figure 10: Batch design pattern: A collection of base learners, `model_<n>, n = 1, ..., N`, are batched together to produce a composite model, `model_Batch`. It is assumed that base learners have vector predictions, and thus the batch learner has a matrix prediction.

```
train_select <- function(X, y) {
  N <- length(y)
  err <- double(N)
  for (n in 1:N) {
    err[n] <- sqrt(mean((X[, n] - y)^2))
  }
  return (which.min(err))
}
predict_select <- function(est, newX) {
  return (newX[, est])
}
```

Figure 11: Select Learner: It chooses the feature vector with smallest error, e.g., by calculating the RMSE error of each feature vis-a-vis the response vector.

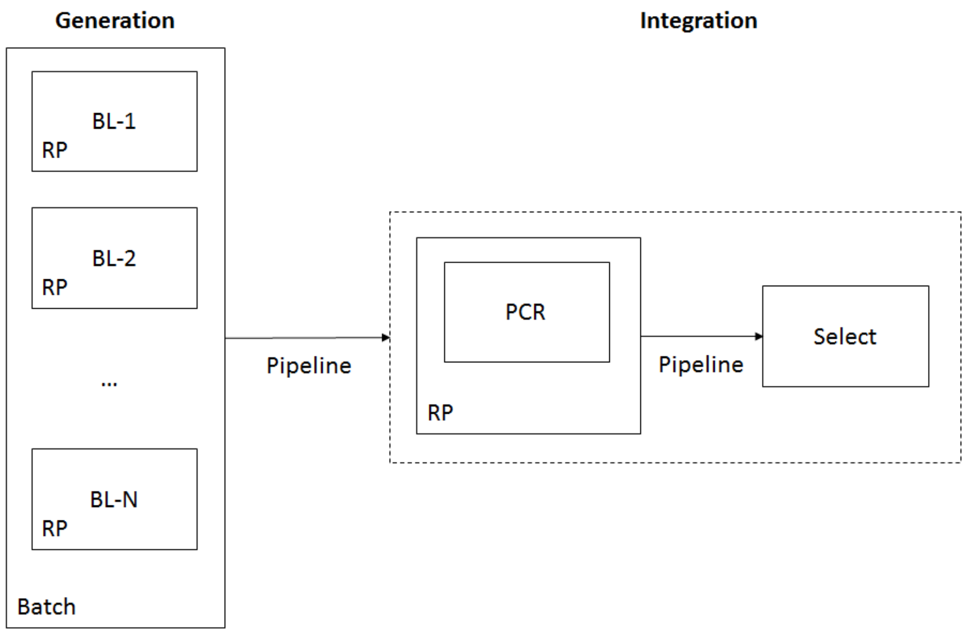

Figure 12: Double-Rotating-Partition framework for heterogeneous ensemble regression. The generation stage involves a batch of base learners, itself embedded in an RP pattern. The integration stage consists of an RP-embedded PCR (or penalized regression) operation – followed by a select operation. Pipeline pattern is used to chain steps together.

the 'composable' nature of our design patterns: they can be combined recursively into increasingly composite learners, thanks to all lego blocks and design patterns implementing the SAP interface.

C. Data sets

Table 2 contains a list of 22 publicly-available data sets used to produce benchmarking results reported in Section 2.3.

D. R session information

```
R> sessionInfo()

R version 3.2.5 (2016-04-14)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 14.04 LTS

locale:
 [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8           LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=en_US.UTF-8       LC_MESSAGES=en_US.UTF-8
```

Table 2: Data sets used in the benchmarking test described in Section **??**, sorted by size. Sources: UCI Machine Learning Repository Bache and Lichman (2013), Luis Torgo's Repository Torgo (2013), KEEL TBD *et al.* (2013)

Dataset	nobs	nvar	source
servo	167	4	UCI
machine	209	6	UCI
yacht	308	6	UCI
baseball	337	16	UCI
dee	365	6	KEEL
autompg	392	7	Luis Torgo
ele1	495	2	KEEL
housing	506	13	UCI
wages	534	9	UCI
energy-heating	768	8	UCI
energy-cooling	768	8	UCI
stock	950	9	KEEL
laser	993	4	KEEL
concrete	1030	8	UCI
ele2	1056	4	KEEL
solar flare - c	1066	10	UCI
friedman	1200	5	KEEL
wine - red	1599	11	UCI
plastic	1650	2	KEEL
election	3107	6	UCI
abalone	4177	8	UCI
wine - white	4898	11	UCI

```
 [7] LC_PAPER=en_US.UTF-8          LC_NAME=en_US.UTF-8
 [9] LC_ADDRESS=en_US.UTF-8        LC_TELEPHONE=en_US.UTF-8
[11] LC_MEASUREMENT=en_US.UTF-8    LC_IDENTIFICATION=en_US.UTF-8

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] EnsemblePCReg_1.1  EnsembleBase_1.0.1 kknn_1.3.0

loaded via a namespace (and not attached):
 [1] Rcpp_0.12.1         igraph_1.0.1        magrittr_1.5
 [4] itertools_0.1-3     splines_3.2.5       MASS_7.3-44
 [7] missForest_1.4      doParallel_1.0.10   gbm_2.1.1
[10] lattice_0.20-33     foreach_1.4.3       minqa_1.2.4
[13] car_2.1-1           tools_3.2.5         nnet_7.3-8
[16] parallel_3.2.5      pbkrtest_0.4-4      grid_3.2.5
[19] glmnet_2.0-2        nlme_3.1-127        mgcv_1.8-12
[22] quantreg_5.19       e1071_1.6-7         MatrixModels_0.4-1
[25] iterators_1.0.8     class_7.3-14        survival_2.39-2
[28] lme4_1.1-10         randomForest_4.6-12 bartMachine_1.2.0
[31] Matrix_1.2-5        rJava_0.9-7         nloptr_1.0.4
[34] codetools_0.2-14    SparseM_1.7
```

Affiliation:

Alireza S. Mahani
Scientific Computing Group
Sentrana Inc.
1725 I St NW
Washington, DC 20006
E-mail: alireza.s.mahani@gmail.com