EnsemblePCReg: An R package for fully-automated ensemble learning with reproducible prediction using Principal Components Regression

Alireza S. Mahani
Scientific Computing Group
Sentrana Inc.

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

Abstract

EnsemblePCReg is an R package for fully-automated, heterogeneous ensemble regression. The learning process of **EnsemblePCReg** consists of generation and integration steps, both of which employ a rotating-partitions (RP) strategy to minimize 'data contamination' while maximizing data utilization. For ensemble generation, a set of base learners – each representing a specific class of learning algorithms – are trained over multi-dimensional grids of tuning parameters. For ensemble integration, Principal Components Regression (PCR) is applied to the output of the first step, and the optimal number of principal components is chosen by minimizing the RP error of PCR prediction. The software is composed from a set of atomic lego blocks, each implementing a 'train-predict interface'. The composable design patterns used in **EnsemblePCReg** go beyond the traditional pipeline model in data analytics to include more complex transformations needed for implementing RP-related operations. The integration step eliminates the need for selection and tuning of base learners, thereby reducing the risk of overfitting and expediting the model-building process. Furthermore, multi-core parallelization – with advanced thread scheduling – significantly improves processing speed, while utility functions for saving/loading trained models to/from disk reduce RAM usage. Thus, **EnsemblePCReg** broadens the reach of ensemble learning, from model-building on personal computers to cloud computing applications such as real-time prediction.

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

1. Introduction

Motivation: Ensemble learners combine a diverse collection of predictions from individual machine learning algorithms to produce a composite predictive model that is more accurate and robust than its components. Advantages of ensemble learning have been well known and documented (Krogh and Sollich 1997; Dietterich 2000; Zhang and Ma 2012), and while ensemble learning of homogeneous base learners is widely used in algorithms such as random forests (Breiman 2001) and gradient boosting machines (Friedman 2001), same is not true for heterogeneous ensemble learning. For example, despite the popularity of techniques such as stacked generalization (Wolpert 1992) in data science competitions (Bell and Koren 2007), winning solutions have been difficult to implement for real-world problems (Holiday 2012). For a review of techniques and literature on ensemble learning, see Rokach (2010) (classification) and Mendes-Moreira, Soares, Jorge, and Sousa (2012) (regression).

Heterogeneous ensemble learning – including base learners that can themselves be ensemble-based – can be a particularly rewarding effort since different flavors of base learners tend to be naturally diverse, each with their own strengths and weaknesses, thus satisfying a basic requirement for effective ensemble learning, i.e., weak correlation among components (Webb *et al.* 2004). Therefore, overcoming obstacles to their broader adoption for predictive-analytic applications is an important objective. These obstacles can be classified into three categories: 1) Achieving the central goal of ensemble learning, i.e., improved generalization, requires maximizing the use of limited data while avoiding 'data contamination', i.e., using within-sample predictions of a learning operation as input into the next operation. Striking this balance leads to a significant *methodological complexity*. 2) A byproduct of this complexity is that separating training and prediction steps becomes extremely difficult, posing a *reproducibility* challenge and making it impossible to use ensemble learning for real-time prediction. 3) Total training time, as well as size of trained objects, can grow rapidly for heterogeneous ensemble learners, creating a significant *computational burden*. This happens especially if some base learners are tree-based ensemble learners such as random forest.

Our contribution: **EnsemblePCReg** is an open-source R package for fully-automated ensemble learning and reproducible prediction in regression problems. It enjoys the following properties, which combine to provide a unique value proposition:

1. *Simple-to-use API*: Relying on carefully-selected default parameters to control the entire process, users can train and predict ensemble models using the familiar one-line API calls in R:

   ```
   R> est <- epcreg(formula, data)
   R> pred <- predict(est, newdata)
   ```

   Overriding default parameters provides great flexibility to experienced users.

2. *Good generalization*: Using a two-stage cross-validation-based approach for ensemble generation and integration, the ensemble meta-learner provided in **EnsemblePCReg** efficiently uses the data set while minimizing the risk of overfitting.

3. *Computationally efficient*: Multicore parallelization using appropriate thread scheduling policies during training and prediction, along with file methods for saving/loading base learner training objects to/from disk allow users to efficiently build ensemble models on personal computers.

4. *Reproducible*: Full separation of train and predict functions, combined with computational efficiencies of the software not only lead to reproducible software, but also allow for ensemble models to be applied in emerging applications such as online/stream computing.

Existing open-source software: We have found three open-source software packages for ensemble meta-learning: 1) the R package **caretEnsemble** (Mayer and Knowles 2015), based on **caret** (Kuhn *et al.* 2015); 2) the R package **subsemble** (LeDell *et al.* 2014), based on **SuperLearner** (Polley and van der Laan 2014), and 3) the C++ package Ensemble Learning

Framework (**ELF**)(Jahrer 2010). **caretEnsemble** provides wrappers for training and combining models wrapped in **caret**, using linear greedy optimization (via function `caretEnsemble`) and a **caret** model (via function `caretStack`). The **caret** training function automatically performs a grid search to tune the parameters of the base learners. Afterwards, a final set of base learners are trained on the entire data set, and passed to the ensemble learner. **subsemble** trains an underlying algorithm on different subsets of data, and combines them using a so-called 'V-fold cross-validation' approach (Sapp, van der Laan, and Canny 2014). **ELF** provides ensemble meta-learning through stacking (Wolpert 1992), cascade learning [ref], and residual learning (Friedman 2001). Similar to **caretEnsemble**, base learners in **ELF** are tuned via cross-validated selection. For a detailed discussion and comparison of these package vs. **EnsemblePCReg**, see Section 5.

Paper structure: The rest of this paper is organized as follows. In Section 2 we describe the main steps involved in ensemble PCR. In Section 3 we outline the major design objectives of **EnsemblePCReg**, and discuss key features that achieve these design goals. In Section 4 we provide several examples to illustrate the usage and features of **EnsemblePCReg**. Finally, in Section 5 we provide a summary of our work, discuss it in the context of existing ensemble learning software, and offer pointers for potential future work.

2. Methodology

Ensemble learning consists of three stages: generation, pruning and integration (Mendes-Moreira *et al.* 2012). In the generation step, many base learners are trained against the same data set. In pruning, a subset of these base learners are discarded. In integration, the selected models are combined to form a single, composite model. **EnsemblePCReg** absorbs pruning into integration to form a two-stage process. Below we describe each one.

2.1. Ensemble generation

Here the ideal outcome is a collection of diverse – i.e., uncorrelated – and accurate models (Brown and other 2005). **EnsemblePCReg** relies on three sources of diversity: 1) different base learners, 2) different sets of tuning parameters – i.e., configurations – for each base learner, and 3) different subsets of data to train each base learner configuration.

Base learners: **EnsemblePCReg** imports base learners from the **EnsembleBase** package (Mahani and Sharabiani 2015) (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, George, and McCulloch 2010).

Tuning parameters: 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

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)	param1 param2 param3 param4

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.

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.

Cross-validated training of base learners: Each of these 112 models is embedded in a rotating-partition (RP) pattern: Assuming 5-fold partitions (default), each model is trained 5 fives, each time on 4/5 of data. The *predict* method for this composite learner behaves differently for training data vs. new data: For training data, prediction is a *concatenation* of predictions of 5 underlying models, each one produced only for the 1/5 of data not seen by that model during training. For new data, prediction is the *average* of individual predictions from constituent models. The RP pattern ensures full data utilization while avoiding contamination, e.g., by passing within-sample predictions from base learners to the integration stage.

EnsemblePCReg provides the option of training base learners on more than one partitioning of data (via the `npart` argument in `epcreg` function). This parameter multiplies the number of base learners produced, which can improve the robustness of integration step, but at the expense of increased training time. See Example 4.2.

2.2. Ensemble integration

EnsemblePCReg uses Principal Components Regression (Jolliffe 1982) [better ref?], embedded in a cross-valdiation-based data splitting strategy – similar to that of the ensemble generation phase – for ensemble integration. It can be decomposed into the following sub-steps:

1. Within each of the K subsets of the data – consisting of (K-1)/K fraction of the entire data across (K-1) CV folds – PCR is applied to the predictions of base learners from the ensemble generation phase. This produces not a single predictor, but a collection of predictors, one for each value of the free parameter of the PCR operation, i.e., the number of principal components used in the regression step of PCR. As such, this step is referred to as a 'sweep' operation.

2. Predictions of each of the above K sweeps are assembled together for the training set, and averaged for the prediction set, similar to what was during ensemble generation for base learners.

3. Finally, RMSE error of concatenated prediction sweeps are calculated (for training set), and the number of principal components with minimum CV error is selected. This serves as the final ensemble model.

Motivation for using a CV strategy is similar to the ensemble generation step: If we use the entire training set to select the optimal number of PC's in the PCR step, we will always select the maximum possible number, i.e, the number of base learners instances. In other words, embedding PCR in cross-validation helps overcome the wll-known multi-collinearity problem of stacked generalization [ref].

EnsemblePCReg follows a modular and extensible approach to implementing the PCR integration software, as elaborated in Section 3.2. This allows for efficient implementation of other integrators such as one using L1/L2 penalized regression [ref] (instead of PCR) – implemented by co-authors in **EnsemblePenReg** (Sharabiani and Mahani 2014) – and the more traditional approach of selecting the base learner with the smallest cross-validation error – also implemented by co-authors in **EnsembleCV** (Sharabiani and Mahani 2015). A detailed discussion of various ensemble integration options can be found in Section 5.

3. Software features

As mentioned in Section 1, **EnsemblePCReg** pursues four design attributes: ease of use, minimize overfitting, computational efficiency, and reproducible predictions. In this section, we discuss the salient features of our software that support these objectives. These features fall under three broad categories: API design, performance optimization, and implementation.

3.1. User-friendly API

An ambitious goal for the API was to make ensemble learning and prediction look and feel as easy to use as any other machine learning algorithm in R. The end-result in **EnsemblePCReg** is that training, diagnostics, visualization, and prediction are as simple as the following familiar commands:

```
R> fit <- epcreg(formula, data)
R> summary(fit)
```

```
R> plot(fit)
R> pred <- predict(fit, newdata)
```

Several points are worth mentioning with regards to the API:

1. Supported by carefully-selected default values for parameters such as choice of base learners, configuration sets for each base learner, and integration controls, the software frees the users from spending time and energy selecting these values through a lengthy trial-and-error process. This makes ensemble modeling process much more efficient than ever before.
2. By encouraging the users towards high-level API calls, the software reserves the flexibility to change implementation details over time, e.g. for refactoring purposes, without breaking backward compatibility or forcing users to learn and adopt a new API.
3. In automating away such steps as selecting the tuning parameters of base learners, the software minimizes the need by practitioners to fiddle with data and create overfit models that suffer from poor generalization.
4. Despite using a sophisticated, two-stage CV-based methodology for improving its generalization performance, **EnsemblePCReg** fully separates train and predict functionalities. This has two important consequences: 1) trained objects can be stored, loaded and reused for new predictions, and thus saving on an unnecessary model training time, and 2) predictions can always be traced back to the model that produced them, which leads to reproducibility of results for both applied and research purposes.

Software implementation and architecture play an important role in making a simple API feasible depsite methodological complexities, as described next.

3.2. Composable architecture

A key architectural decision in **EnsemblePCReg** (and other members of the **Ensemble** software collection) is to decompose complex, multi-step operations into smaller lego blocks, all of which comply with a common interface that embodies the *train-predict (TP) design pattern*. According to this pattern, every operation must implement three methods: 1) `fit`, 2) `predict` for same data used in fit, and 3) `predict` for new data. Using a lego blocks of operations that conform to this interface, composite operations can be constructed, using a variety of *compositional* patterns. Each compositional pattern defines the recipes for how the three methods of the composite class can be constructed from the constituent class methods.

The most elementary example of a compositional pattern is 'chaining' of operations, which forms a pipeline. Figure 1 shows the pseudo-code for chaining two operations. **EnsemblePCReg** utilizes several, more complex compositional patterns such as 'batching', 'cross-validation', 'sweep', and 'select'. Figure 2 shows the call stack for the main function in the package, `epcreg`, illustrating how a complex ensemble learner can be composed from elementary operations through 1) strict enforcement of the train-predict interface, and 2) application of sophisticated compositional patterns.

The modular and composable architecture of **Ensemble** packages not only reduces complexity by breaking it down to manageable pieces, but also facilitates the process of innovating and

```
class A (or B) {
  // fields
  ...
  // methods
  void fit(Data data) {...}
  Data predict() {...}
  Data predict(Data newdata) {...}
}
class A_then_B() {
  A a;
  B b;
  void fit(Data data) {
    a.fit(data);
    b.fit(data + a.predict());
  }
  Data predict() {return b.predict();}
  Data predict(Data newdata) {
    return b.predict(newdata + a.predict(newdata));
  }
}
```

Figure 1: C++ pseudo-code for applying the 'chaining' compositional pattern to operations `A` and `B`, producing the composite operator `A_then_B`.

testing new integrators, whether vastly different from existing ones, or small variations of them. For example, consider the 'select' operation in Figure 2. This step is responsible for selecting the optimal number of principal components during ensemble integration. Some researchers have suggested a more conservative approach to selecting similar tuning parameters that control overfitting, e.g., by choosing a smaller number of principal components than the one that minimizes CV error (Friedman, Hastie, and Tibshirani 2001). In our framework, this can be done by implementing `Select.Fit` (and its prediction counterpart) for a type other than `MinErr`. The rest of the call stack remains unchanged.

To support our modular and extensible architecture, we have separated the core functionalities needed by all integrators – including base learner facilities – into **EnsembleBase**, which acts as the recipient of ongoing code refactoring work. Furthermore, we have made heavy use of `S4` generics in R, in addition to `S3` generics for functions such as `predict`, `summary` and `plot`.

3.3. Performance optimization

EsnemblePCReg uses a set of performance optimization strategies to reduce the processing time and memory consumed during ensemble learning and prediction. These strategies include 1) multicore parallelization with suitable thread scheduling policies for training and prediction functions, and 2) option to save/load base learner trained objects to/from disk – as temporary files during program execution, or permanent files in-between sessions – to reduce total size of ensemble trained object in RAM during training and prediction. We discuss the key concepts behind these features here, and illustrate how they can be used in Section 4.

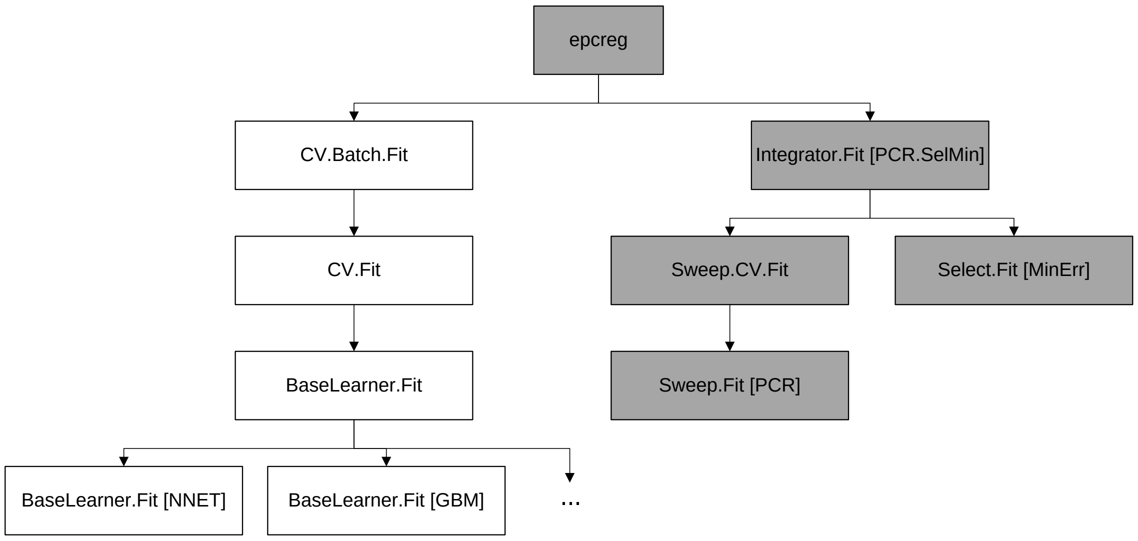

Figure 2: Call stack for `epcreg` function in **EnsemblePCReg** package, illustrating the modular and extensible design of the software. Square brackets indicate specialization of a `S4` generic class. White boxes indicate functions implemented in **EnembleBase**. For brevity, `Regression` and `Config` have been dropped from function and class names. See source code for full names. A similar diagram can be created for `predict.epcreg`, following the train-predict duality discussed in Section 3.2.

Multicore parallelization: During training, many base-learner instances must be trained. These training tasks are completely independent, and can be parallelized. Similarly, during prediction, the predictions of these base learners must first be collected before combining them in ensemble integration stage. Using the `ncores` parameter, users can control the number of parallel threads to use in training (`epcreg`) and prediction (`predict.epcreg`). Parallelizing prediction may appear unnecessary in many applications – especially in batch processing – since it takes only a fraction of training time. However, in many applications such as online prediction services where a pre-trained model is exposed to users for obtaining predictions for their data points, fast response times can be critical.

Thread scheduling policy: In assigning tasks to parallel threads, we seek the dual objectives of load balancing and parallel overhead minimization (Chandra 2001). Load balancing seeks to distribute tasks across threads such that total task durations are as even as possible. Otherwise, a weak link – i.e., 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 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. 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. **EnsemblePCReg** offers three flavors of scheduling – accessed via the `schedule.method` argument – within each of these two broad categories: 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. [describe more, or refer to source code]

Figure 3 compared 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 assignment 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. In Section xx, we offer examples of how to toggle between scheduling modes, including how to generate and use task lengths for effective parallelization.

File method: Some base learners produce large training objects, especially ensemble tree-based algorithms such as RF and GBM. Multiplied by (number of configurations per partition

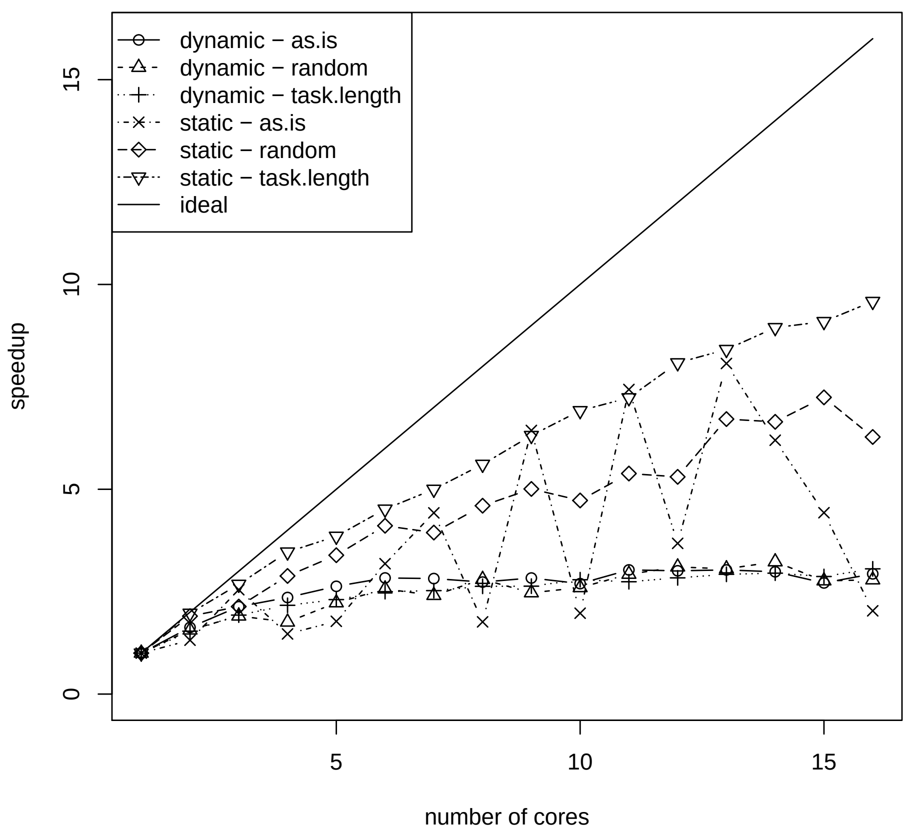

Figure 3: Impact of scheduling policy on scaling behavior of ensemble training as a function of number of parallel cores used.

x number of partitions x 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, for a data set of xx observations and yy features (dataset zz), using `npart=10` with default base learners and configurations produces a xx GB object, exceeding the total RAM available on most personal computers.

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 `epcreg.save` and `epcreg.load` methods are provided to save/load `epcreg` objects to/from permanent files for intersession continuity.

It must be noted that, executing training and prediction in parallel mode partially negates the memory savings offered by the file method, since as many base learner objects could be loaded into RAM at any given time as the number of parallel threads.

4. Using EnsemblePCReg

4.1. Example 1: Training and prediction for ensemble models

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

```
R> library("EnsemblePCReg")
R> my.seed <- 0
R> set.seed(my.seed)
R> data(servo)
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 esnemble 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:

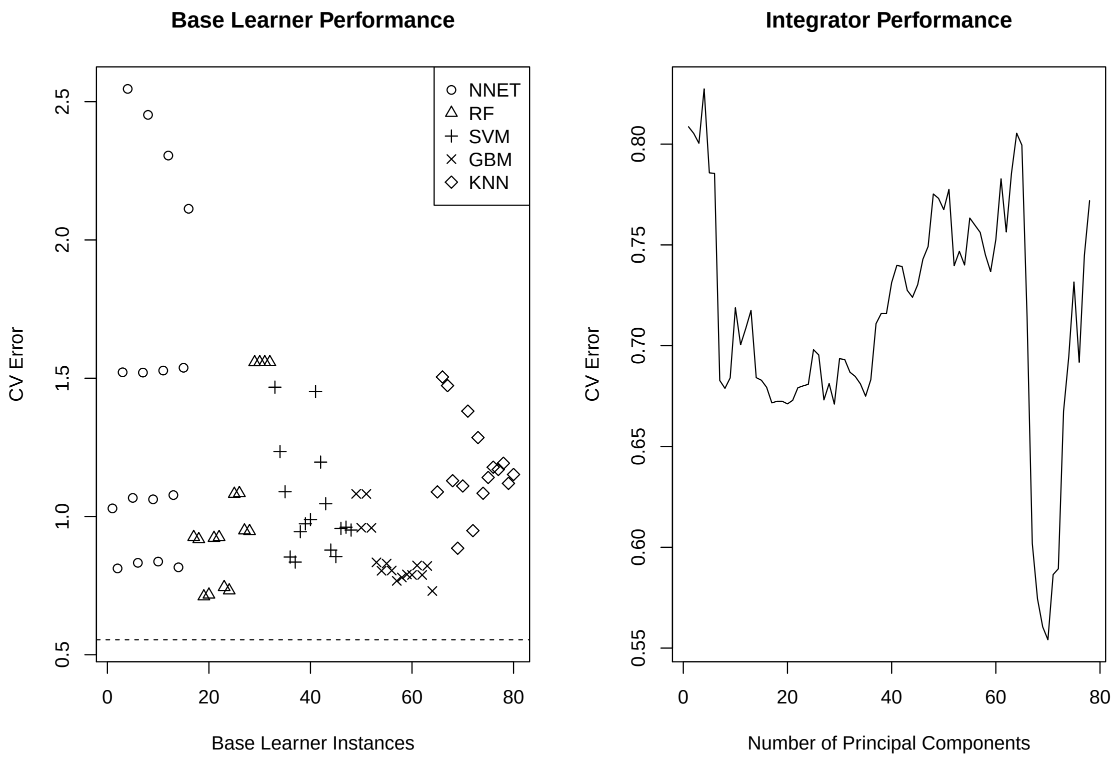

Figure 4: Performance of base learners (left) and PCR integrator (right) for `servo` data set, using two-step 5-fold cross-validation as described in Section **??**.

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: Changing default settings

In the first example, we saw that building an ensemble model using the default settings of the `epcreg` function, is extremely easy. Users can exert more control over these settings, 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, it might be tempting to exclude 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 end of pack so as to generate identical models for other base learners by fixing the random seed before each run:

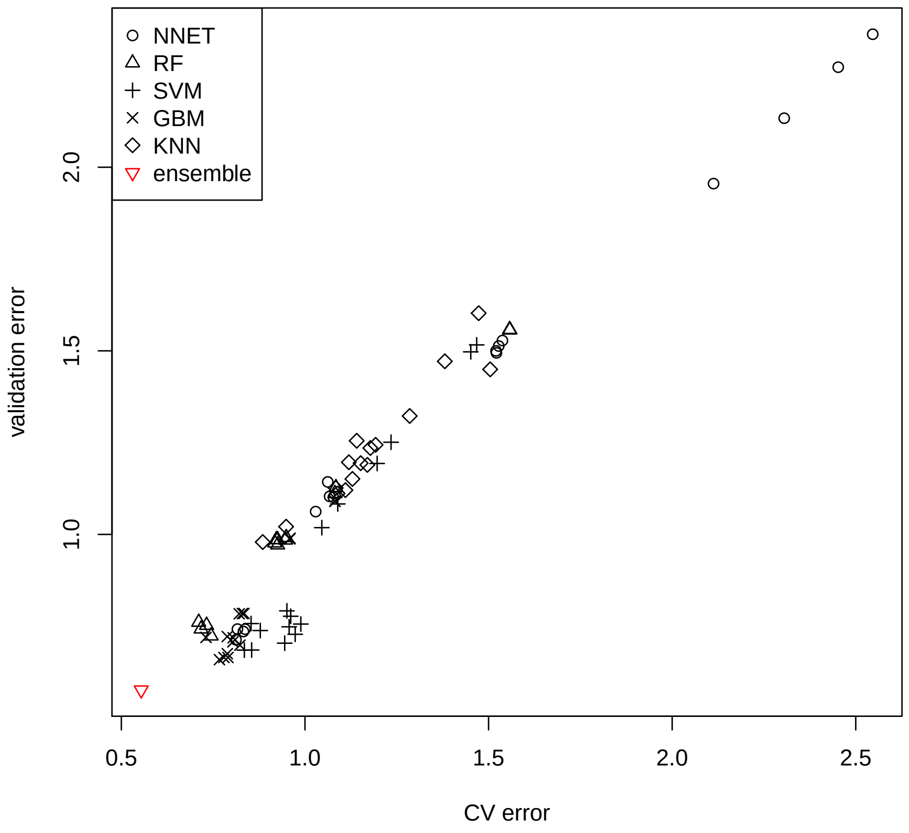

Figure 5: Comparison of CV error (70% of data) against validation error (30% of data) for base learners as well as ensemble model trained on `servo` data set.

```
R> baselearners.1 <- c("rf", "svm", "gbm", "knn", "nnet")
R> control.1 <- epcreg.baselearner.control(baselearners = baselearners.1)
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)
R> summary(est.2)

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

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
```

[mention that this could be due to random fluctuations in a small data set, and we need more systematic analysis, both within this data set and also using 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.

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> est.npart <- epcreg(myformula, data.train, print.level = 0,
+    baselearner.control = control.npart)
R> plot(est.npart)
```

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: Multicore parallelization

On our machine [describe specs here or elsewhere], for `npart=10`, training time was approximately 6 minutes, depsite the fact that `servo` is a very small data set (117 observations and 4 variables in training set). Indpendence of base learners offers an obvious parallelization opportunity, which can be exploited via the `ncores` argument of `epcreg` function:

```
R> est.npart.par <- epcreg(myformula, data.train, print.level = 0,
+    baselearner.control = control.npart, ncores = 4)
```

This reduced training time to 140sec, representing a nearly 2.6x [?] speedup, which is reasonable for 4 cores [no it's not that great; explain why]. 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 general, 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

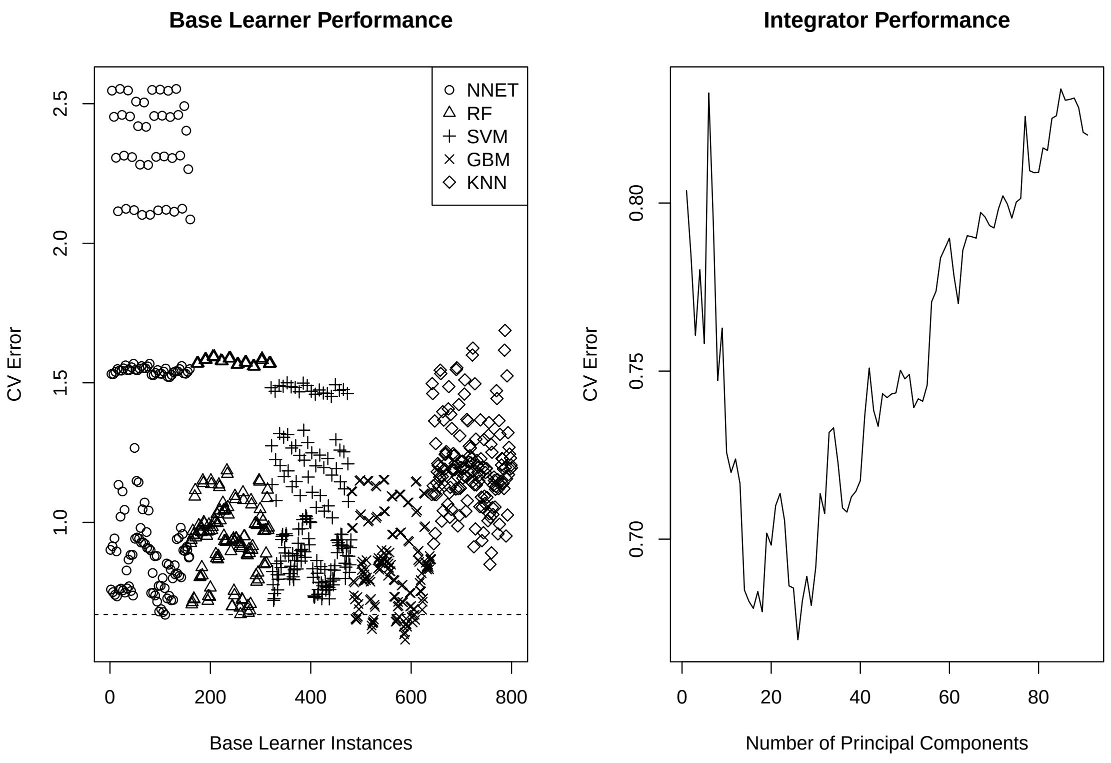

Figure 6: Base learner and integrator performance for `servo` data set, using `10` partitions instead of the default value of `npart=1`.

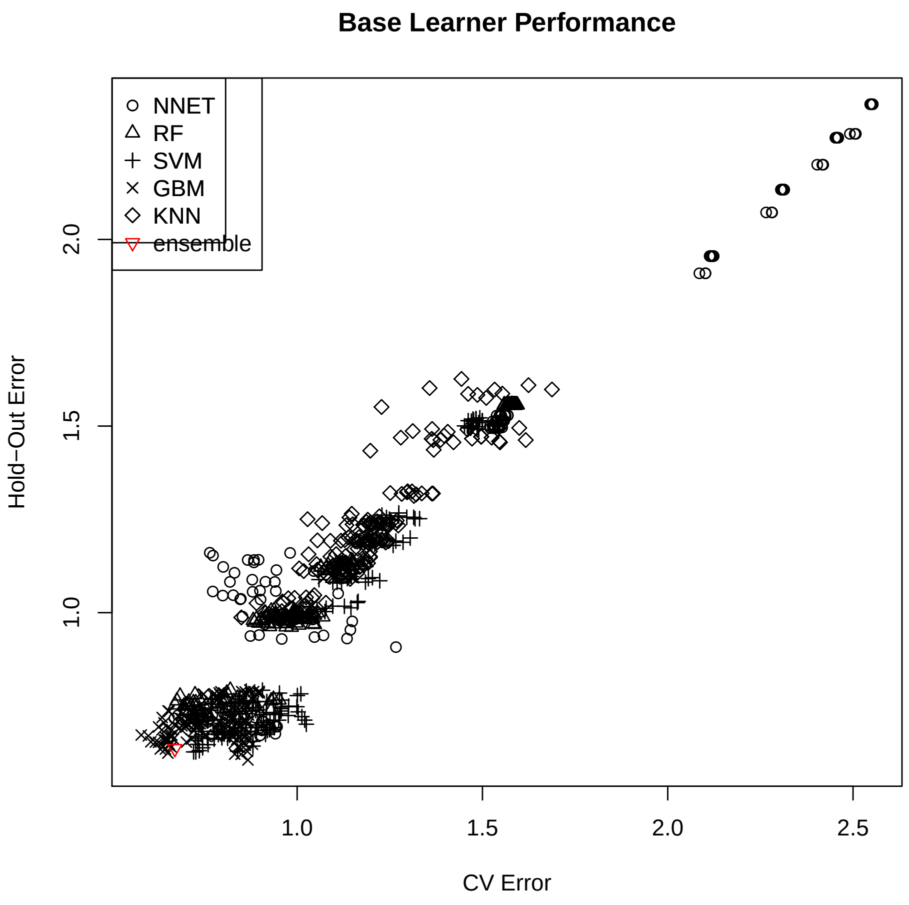

Figure 7: Comparison of CV and validation errors for base learners and ensemble model, for `servo` data set, using `10` partitions.

available via the `EnsembleBase` package, and is based on the multi-threading implementation of the R package **doParallel** (Analytics and Weston 2015).

For ensemble models, prediction can also be time-consuming:

```
R> t.pred.npart <- proc.time()[3]
R> pred.npart <- predict(est.npart, newdata = data.predict)
R> t.pred.npart <- proc.time()[3] - t.pred.npart
R> cat("t.pred.npart:", t.pred.npart, "\n")
```

```
prediction time - serial: 13.579 sec
```

While in some use-cases 14sec may not be a long time, in other cases such as online prediction services, or real-time and/or stream processing, sub-second response times may be desirable or even required. Similar to training, prediction can also benefit from multicore parallelization:

```
R> t.pred.npart.par <- proc.time()[3]
R> pred.npart.par <- predict(est.npart, newdata = data.predict, ncores = 4)
R> t.pred.npart.par <- proc.time()[3] - t.pred.npart.par
R> cat("t.pred.npart.par:", t.pred.npart.par, "sec\n")
```

```
prediction time - parallel: 4.204
```

Speedup (using 4 cores) is reasonable:

```
R> cat("parallelization speedup - predict:",
+    t.pred.npart / t.pred.npart.par, "\n")
```

```
parallelization speedup - predict: 3.230019
```

An important parameter in configuring the multicore parallelization for training and prediction of ensemble models is the scheduling policy, namely whether it should be static or dynamic. Static scheduling means jobs are pre-allocated to threads before entering the parallel region, while in dynamic scheduling, each thread grab the next job from a queue once each time it is finished with the previous job. (For a more detailed explanation of different scheduling policies in multi-threading and their impact on application performance, see Chandra (2001).) Static scheduling incurs a smaller thread synchronization overhead; however, if job durations are non-uniform and unpredictable, it can lead to load imbalance and suboptimal parallelization speedup. In ensemble models, the training jobs are often more time-consuming and non-uniform and therefore better suited for dynamic scheduling. On the other hand, prediction jobs are faster and more honogeneous, and therefore static scheduling is more appropriate for them. Default scheduling policies in **EnsembleBase** reflect this analysis, which is backed by empirical results [maybe expand]. This topic is further discussed in Section 5. [move this paragraph to Section **??**.]

4.4. Example 4: Using file methods

Ensemble models are not only time-consuming, but also memory-consuming. This is due to two reasons: 1) The training object for some base learners, especially ensemble base learners such as random forest, are large objects, sometimes much larger than the raw data. In **EnsemblePCReg**, we build 16 base learners of each kind, for a total of 80 models. If we inrease the number of partitions, as we did in Example 3, the tally further rises. In that example, when we used 10 partitions, even for a small data set of 117 observations (training set), the resulting ensemble trained object reaches about 250MB. For a data set of 10,000 observations, this number would become 25GB, exceeding the memory (RAM) capacity of most personal computers, and even some servers.

EnsembleBase 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 objects contain the temporary file information containing the trained object (on disk). Upon calling `predict` against the trained object, the core estimation object is loaded from disk, used for prediction, and discarded afterwards [implement this]. As usual, using this feature is extremely easy:

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

[continue example with load and save methods.]

[discuss memory savings in serial and parallel modes.]

5. Discussion

5.1. Summary

In this paper, we presented **EnsemblePCReg**, an R package for fully-automated ensemble meta-learning of regression models. Using a two-stage, cross-validation-based approach for ensemble generation and integration, the software combines multiple base learners, each with multiple sets of values for their tuning parameters into a composite model that is often more accurate and reliable than a standard cross-validated selection strategy. Combined with good default values and simple-to-use API, the fully-automated process of **EnsemblePCReg** minimizes the risk of overfitting the model to training data, while achieving high efficiency in data utilization. Separation of training and prediction functionalities allows for reusable models and reprducible results. Performance optimization techniques such as multicore parallelization and file methods for saving and loading base learner trained models, allow **EnsemblePCReg** to be a practical tool for ensemble meta-learning on personal computers, and for emerging data-analytic applications such as stream processing and online prediction services.

A. Setup

```
R> sessionInfo()
```

```
R version 3.2.2 (2015-08-14)
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
 [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_1.0.0 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_3.2.2       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_3.2.2         nnet_7.3-8
[16] parallel_3.2.2      pbkrtest_0.4-4      grid_3.2.2
[19] glmnet_2.0-2        nlme_3.1-122        mgcv_1.8-7
[22] quantreg_5.19       e1071_1.6-7         MatrixModels_0.4-1
[25] iterators_1.0.8     class_7.3-14        survival_2.38-3
[28] lme4_1.1-10         randomForest_4.6-12 bartMachine_1.2.0
[31] Matrix_1.2-2        rJava_0.9-7         nloptr_1.0.4
[34] codetools_0.2-14    SparseM_1.7
```

References

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.

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

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

Brown G, other (2005). "Diversity creation methods: a survey and categorisation." *Information Fusion*, **6**(1), 5–20.

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

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

Dietterich TG (2000). "Ensemble methods in machine learning." In *Multiple classifier systems*, pp. 1–15. Springer.

Friedman J, Hastie T, Tibshirani R (2001). *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin.

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.

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

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.

Jahrer M (2010). "ELF - Ensemble Learning Framework. An open source C++ framework for supervised learning." `http://elf-project.sourceforge.net`.

Jolliffe IT (1982). "A note on the use of principal components in regression." *Applied Statistics*, pp. 300–303.

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

Krogh A, Sollich P (1997). "Statistical mechanics of ensemble learning." *Physical Review E*, **55**(1), 811.

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

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, Soares C, Jorge AM, Sousa JFD (2012). "Ensemble approaches for regression: A survey." *ACM Computing Surveys (CSUR)*, **45**(1), 10.

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`.

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

Rokach L (2010). "Ensemble-based classifiers." *Artificial Intelligence Review*, **33**(1-2), 1–39.

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

Sapp S, van der Laan MJ, Canny J (2014). "Subsemble: an ensemble method for combining subset-specific algorithm fits." *Journal of applied statistics*, **41**(6), 1247–1259.

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`.

Sharabiani MT, Mahani AS (2015). *EnsembleCV: Extensible Package for Cross-Validation-Based Integration of Base Learners.* R package version 0.7.1, URL `http://CRAN.R-project.org/package=EnsembleCV`.

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

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

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.

Zhang C, Ma Y (2012). *Ensemble Machine Learning.* Springer.

Affiliation:

Alireza S. Mahani
Scientific Computing Group
Sentrana Inc.

1725 I St NW
Washington, DC 20006
E-mail: alireza.s.mahani@gmail.com