An Infra-Structure for Performance Estimation and Experimental Comparison of Predictive Models in R

Luis Torgo
FCUP - LIAAD/INESC Tec
University of Porto
ltorgo@dcc.fc.up.pt, ltorgo@inesctec.pt

June 16, 2015

Abstract

This document describes an infra-structure provided by the R package `performanceEstimation` that allows to estimate the predictive performance of different approaches (workflows) to predictive tasks. The infra-structure is generic in the sense that it can be used to estimate the values of any performance metrics, for any workflow on different predictive tasks, namely, classification, regression and time series tasks. The package also includes several standard workflows that allow users to easily set up their experiments limiting the amount of work and information they need to provide. The overall goal of the infra-structure provided by our package is to facilitate the task of estimating the predictive performance of different modeling approaches to predictive tasks in the R environment.

1 Introduction

The goal of this document is to describe the infra-structure that is available in package `performanceEstimation` to estimate the performance of different approaches to predictive tasks. The main goal of this package is to provide a general infra-structure that can be used to estimate the performance using several predictive performance metrics of any modelling approach to different predictive tasks, with a minimal effort from the user. The package provides this type of facilities for classification, regression and time series tasks. There is no limitation on the type of approaches to these tasks for which you can estimate the performance - the user just needs to provide a workflow function (following some interface rules) that implements the approach for which the predictive performance is to be estimated. The package includes some standard workflow functions that implement the standard learn+test approach that most users will be interested in. This means that if you just want to estimate the performance of some variants of a method already implemented in R (e.g. an SVM), on some particular tasks, you will be able to use these standard workflow functions and thus your required input will be limited to the minimum. The package also provides a series of predictive performance metrics for different tasks. Still, you

are not limited to these metrics and can use any metric as long as it exists in R or you provide a function calculating it. Finally, the most recent versions of the package also include several standard data pre-processing and predictions post-processing steps that again can be incorporated into the workflows being evaluated.

This infra-structure implements different methods for estimating the predictive performance. Namely, you can select among: (i) cross validation, (ii) hold-out and random sub-sampling, (iii) leave one out cross validation, (iv) bootstrap (ϵ_0 and .631) and also (v) Monte-Carlo experiments for time series forecasting tasks. For each of these tasks different options are implemented (e.g. use of stratified sampling).

Most of the times experimental methodologies for performance estimation are iterative processes that repeat the modeling task several times using different train+test samples with the goal of improving the accuracy of the estimates. The estimates are the result of aggregating the scores obtained on each of the repetitions. For each of these repetitions different training and testing samples are generated and the process being evaluated is "asked" to: (i) obtain the predictive model using the training data, and then (ii) use this model to obtain predictions for the respective test sample. These predictions can then be used to calculate the scores of the performance metrics being estimated. This means that there is a workflow that starts with a predictive task for which training and testing samples are given, and that it should produce as result the predictions of the workflow for the given test sample. There are far too many possible approaches and sub-steps for the implementation of this workflow. To ensure full generality of the infra-structure, we allow the user to provide a function that implements this workflow for each of the predictive approaches she/he wishes to compare and/or evaluate. This function can be parameterizable in the sense that there may be variants of the workflow that the user wishes to evaluate and/or compare. Still, the goal of this workflow functions is very clear: (i) receive as input a predictive task for which training and test samples are given, as well as any eventual workflow specific parameters; and (ii) produce as result a set of predictions for the given test set. These predictions will then be used to obtain the scores of the predictive metrics for which the user is interested in obtaining reliable estimates.

The infra-structure we describe here provides means for the user to indicate: (i) a set of predictive tasks with the respective data sets; (ii) a set of workflows and respective variants; and (iii) the information on the estimation task. The infra-structure then takes care of all the process of experimentally estimating the predictive performance of the different approaches on the tasks, producing as result an object that can be explored in different ways to obtain the results of the estimation process. The infra-structure also provides several utility functions to explore these results objects, for instance to obtain summaries of the estimation process both in textual format as well as visually. Moreover, it also provides functions that carry out statistical significance tests based on the outcome of the experiments.

Finally, the infra-structure provides utility functions implementing frequently used workflows for common modelling techniques, several data pre-processing steps and prediction post-processing operations, as well as functions that facilitate the automatic generation of variants of workflows by specifying sets of parameters that the user wishes to consider in the comparisons.

2 A Simple Illustrative Example

Let us assume we are interested in estimating the predictive performance of several variants of an SVM on the **Iris** classification problem. More specifically, we want to obtain a reliable estimate of the error rate of these variants using 10-fold cross validation. The following code illustrates how these estimates could be obtained with our proposed infra-structure.

```
library(performanceEstimation)  # Loading our infra-structure
library(e1071)                  # A package containing SVMs

data(iris)                      # The data set we are going to use

res <- performanceEstimation(
         PredTask(Species ~ .,iris),
         Workflow("standardWF",learner="svm"),
         EstimationTask(metrics="err",method=CV())
         )

##
##
## ##### PERFORMANCE ESTIMATION USING  CROSS VALIDATION  #####
##
## ** PREDICTIVE TASK :: iris.Species
##
## ++ MODEL/WORKFLOW :: svm
## Task for estimating  err  using
##  1 x 10 - Fold Cross Validation
##   Run with seed =  1234
## Iteration :  1  2  3  4  5  6  7  8  9  10
```

This simple example illustrates several key concepts of our infra-structure. First of all, we have the main function - `performanceEstimation()`, which is used to carry out the estimation process. It has 3 main arguments: (i) a vector of predictive tasks (in the example a single one); (ii) a vector of workflows (in the example a single one); and (iii) the specification of the estimation task (essentially the metrics to be estimated and the estimation methodology). The package defines three classes of objects for storing these three concepts: predictive tasks (S4 class **PredTask**); workflows (S4 class **Workflow**); and estimation tasks (S4 class **EstimationTask**).

PredTask objects can be created by providing information on the formula defining the task, the source data set (an R data frame or the name of such object), and an optional ID (a string) to give to the task.

Workflow objects include information on the name of the function implementing the workflow and any number of parameters to be passed to this function when it will be called with different train and test sets.

EstimationTask objects provide information on the metrics for which we want a reliable estimate and also the methodology to be used to obtain these estimation. In case of metrics not defined with the package `performanceEstimation` we also need to supply the name of the function that can be used to calculate these specific metrics.

In the above simple example we are using the data frame **iris** to create a task consisting of forecasting the variable *Species* using all remaining variables. We are solving this classification task by using the function `standardWF` with the parameter **learner** set to "svm". This workflow function is already provided

by the package and essentially can be used for the most frequent situations where a user just wants to apply and existing learning method to solve some task. This avoids the need for the user to have to write the functions solving the task. The `standardWF` function simply applies the modeling function indicated through the parameter **learner** (in this case the function `svm` defined in package **e1071** [MDH+12]) to the training set, and uses the resulting model to obtain the predictions for the test set. As we will see later the `standardWF` function also implements a series of common data pre-processing steps (e.g. filling in unknown values), and several common prediction post-processing steps (e.g. applying some transformation to the predicted values). Finally the above example specifies the estimation task as using 10-fold Cross Validation to obtain estimates of the error rate.

The result of the call to `performanceEstimation()` is an S4 object of the class **ComparisonResults**. These objects tipically are not directly explored by the end-user so we ommit their details here[1]. There are several utility functions that allow the users to explore the results of the experimental comparisons, as shown by the next few illustrative examples.

```
summary(res)

##
## == Summary of a  Cross Validation Performance Estimation Experiment ==
##
## Task for estimating  err  using
##  1 x 10 - Fold Cross Validation
##   Run with seed =  1234
##
## * Predictive Tasks ::  iris.Species
## * Workflows  ::  svm
##
## -> Task:  iris.Species
##   *Workflow: svm
##             err
## avg     0.03333
## std     0.04714
## med     0.00000
## iqr     0.06667
## min     0.00000
## max     0.13333
## invalid 0.00000
```

The generic function `summary` allows us to obtain the estimated scores for each compared approach on each predictive task. For each performance metric (in this case only the error rate), the function shows several descriptive statistics of the performance of the workflow in the different iterations of the estimation process. Moreover, information is also given on eventual failures on some of the iterations.

The generic function `plot` can be used to obtain a graphical display of the distribution of performance metrics across the different iterations of the estimation process using box-plots, as show in Figure 1.

[1]Interested readers may have a look at the corresponding help page - `class?ComparisonResults` .

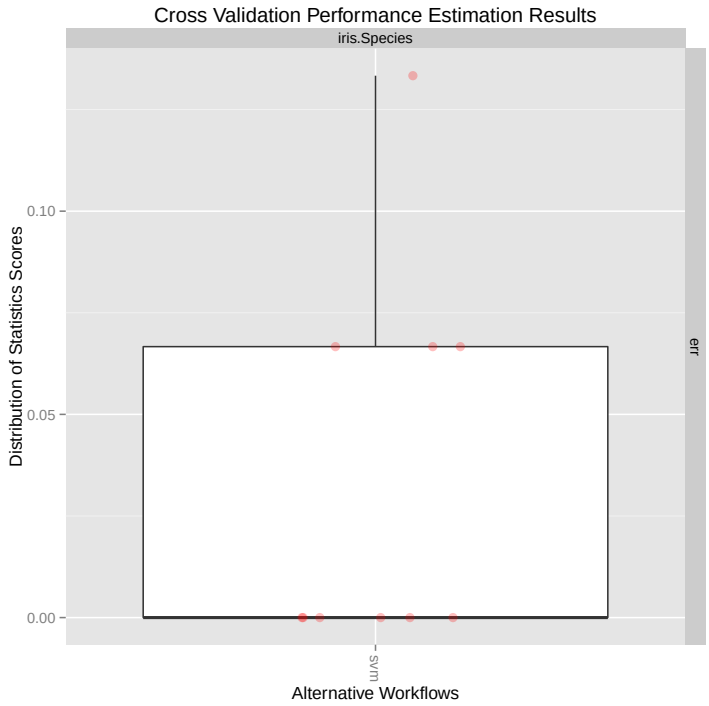

Figure 1: The distribution of the error rate on the 10 folds. Red dots show the performance on each individual iteration of the estimation process.

```
plot(res)
```

The above example is just a simple illustration of the key concepts of the package `performanceEstimation` . Most of the times you will use the package for more complex experiments, usually involving several tasks, several workflows or workflow variants, and eventually estimating several performance metrics. Before, we provide illustrations for these more complex setups we describe in more detail the key concepts of the package `performanceEstimation` .

3 Predictive Tasks

Predictive tasks are data analysis problems where we want to obtain a model of an unknown function $Y = f(X_1, X_2, \cdots, X_p)$ that relates a target variable Y with a set of p predictors $X_1, X_2, \cdots, X_p$. The model is usually obtained using a sample of n observations of the mapping of the unknown function, $D = \{\langle \mathbf{x}_i, Y_i \rangle\}_{i=1}^{n}$, where $\mathbf{x}_i$ is a vector with the p predictors values and Y_i the respective target variable value. These data sets in R are usually stored in data frames, and formula objects are used to specify the form of the functional dependency that we are trying to model, i.e. which is the target variable and the predictors.

Objects of class **PredTask** encapsulate the information of a predictive task,

i.e. the functional form and the data required for solving it. For convenience they also allow the user to assign a name to each task. These S4 objects can be created using the construtor function `PredTask()`, as seen in the following example:

```
data(iris)
PredTask(Species ~ .,iris)

## Prediction Task Object:
##  Task Name         :: iris.Species
##  Task Type         :: classification
##  Target Feature    :: Species
##  Formula           :: Species ~ .
##  Task Data Source  :: iris
```

We should remark that, by default and for memory economy particularly if you are using large data sets, the objects of this class do not copy internally the data of the provided data frame. This means that they assume that the data frame provided in the second argument will still exist when the estimation experiments will be executed. If you want the data to be copied into the **PredTask** object then you should call the constructor as follows: only store the data required for the specified task, as it should be clear from this other simple example:

```
data(iris)
PredTask(Species ~ .,iris,copy=TRUE)

## Prediction Task Object:
##  Task Name         :: iris.Species
##  Task Type         :: classification
##  Target Feature    :: Species
##  Formula           :: Species ~ .
##  Task Data Source  :: internal  150x5 data frame.
```

Note the difference on the task data source information to indicate that the data set is stored internally (i.e. a copy was made from the provided object).

4 Workflows and Workflow Variants

Estimation methodologies work most of the times by re-sampling the available data set D in order to create different train and test samples from D (an exception being a single repetition of hold-out). The goal is to estimate the predictive performance of a proposed workflow to solve the task, by using these different samples to increase our confidence on the estimates. This workflow consists on the process of obtaining a model from a given training sample and then use it to obtain predictions for the given test set. This process can include several steps, e.g. specific data pre-processing steps, and may use any modelling approach, eventually developed by the user.

We distinguish two types of workflows: (i) generic, and (ii) user-defined. The first are probably the most common setups where the user simply wants to apply methods that already exist in R to some task. The second covers situations where the user has developed her/his own specific approach to solve a task and wants to apply it to some concrete problems. The package covers both situations as we will see.

Independently of the workflows being generic or user-defined, they will typically have parameters that we can tune to try to improve the results. Frequently, we want to estimate and compare the performance of these parameter variants over some set of tasks. Our package also facilitates these setups by providing a function to easily specify these alternatives.

4.1 Workflow Variants

The function `workflowVariants` provides means to easily create a vector of **Workflow** objects, each resulting from a variation of some parameter values of a certain workflow. For instance, suppose you want to evaluate the performance of SVMs on a certain task and you wish to consider a certain range of values for the parameters **cost** and **gamma** of the function `svm` from package **e1071**. You could obviously create a vector of **Workflow** objects, one for each combination of **cost** and **gamma** you wish to consider in your experiment. This could become very easily an annoying task if you want to consider a large number of combinations. Instead you can use the `workflowVariants` function.

The following example illustrates its use by considering 15 variants of SVMs applied to the Boston Housing data:

```
library(performanceEstimation)
library(e1071)
data(Boston,package="MASS")

exp <- performanceEstimation(
         PredTask(medv ~ .,Boston),
         workflowVariants(wf="standardWF",learner="svm",
                          learner.pars=list(cost=1:5,gamma=c(0.1,0.05,0.01))),
         EstimationTask(metrics="mae",method=CV())
         )
```

The function accepts any workflow function in parameter **wf**. All remaining parameters are taken as parameters of that workflow function. If you include any of these parameters with a vector of values (as it is the case of **cost** and **gamma** in the above example), the values in these vectors will be used to generate variants of the workflow. The variants are essentially all possible combinations of the values in each vector. In the above example we provide 5 values for **cost** and 3 for **gamma** and thus we will have 15 **Workflow** objects. It may happen that some of the parameters of the workflow, actually take as valid values a vector. In those situations you do not want the function to use the values to generate variants. You can achived this effect by including the name of that parameter in the parameter **as.is** of the function `workflowVariants`.

4.2 Generic Workflows

Most of the times users want to evaluate standard workflows on some tasks. This means that they want to use existing out-of-the-box tools within R and apply them to some data sets. As such, the most frequently used workflows will essentially build a model using some existing algorithm and obtain its predictions for the test set. To facilitate this type of approaches, package `performanceEstimation` includes generic workflow functions. The idea is to save the user from having to write her/his own workflow functions provided her/his workflow fits this generic schema.

4.2.1 Classification and Regression Tasks

Function `standardWF()` implements a typical workflow for both classification and regression tasks. Apart from a formula, a training set data frame and a test set data frame, this function has the following main parameters that help the user to specify the intended approach:

learner - the name of an R function that obtains a model from the training data. This function will be called with a formula in the first argument and the training set data frame in the second. This will typically be the name of some existing learning algorithm in R.

learner.pars - a list specifying any extra parameter settings that should be added to the formula and training set, at the time the learner function is called (defaults to `NULL`).

predictor - the name of an R function that is able to obtain the predictions of the model obtained with `learner`. This function will be called with the object resulting from the `learner` call on the first argument and the test set data frame in the second (it defaults to function `predict`). Most of the times you will not change this default because most modeling techniques in R have a predict method that can be applied to the models.

predictor.pars - a list specifying any extra parameter settings that should be added to the model and test set, at the time the predictor function is called (defaults to `NULL`). This is sometimes handy has some predict methods of some algorithms allow for instance to either return class labels or probabilities, and the option is made through some extra parameter when calling the predict function.

Below you find another simple example that illustrates the use of some of these parameters, this time to estimate the accuracy of 3 variants of a classification tree using 2×5-CV.

```
library(performanceEstimation)
library(DMwR)
data(iris)

res <- performanceEstimation(
         PredTask(Species ~ .,iris),
         workflowVariants(learner="rpartXse",
                          learner.pars=list(se=c(0,0.5,1)),
                          predictor.pars=list(type="class")),
         EstimationTask(metrics="acc",method=CV(nReps=2,nFolds=5))
         )
```

Notice how we have omitted the specification of the workflow name in the call to `workflowVariants`. If you do not specify it through parameter **wf** the function will assume that you want to use the workflow defined by the function `standardWF`, unless there is a parameter **type** in the call which will lead to assume a time series generic workflow provided by function `timeseriesWF` that will be described in Section 4.2.2. In effect, you can also use the same speedup trick when calling directly the **Workflow** constructor, i.e.
`Workflow("standardWF",learner="svm")`

is equivalent to
`Workflow(learner="svm")`.

We now present an example of one of the most frequent type of comparisons users carry out - checking which is the "best" model for a given predictive task. Let us restrict the search to a small set of models for illustrative purposes and let us play with the well-known Boston housing regression task:

```
data(Boston,package='MASS')
library(DMwR)
library(e1071)
library(randomForest)
bostonRes <- performanceEstimation(
  PredTask(medv ~ .,Boston),
  workflowVariants(learner=c('rpartXse','svm','randomForest')),
  EstimationTask(metrics="mse",method=CV())
  )
```

Notice that on this simple example we have used all modeling tools with their default parameter settings which is not necessarily a good idea when we are looking for the best performance. Still, the goal of this illustration is to show you how simple this type of experiments can be if you are using a standard workflow setting. In case you want to use the modelling tools with other parameter settings then you should separate them in different `workflowVariants` calls because each learner will most probably use different parameters, as shown in the following example:

```
data(Boston,package='MASS')
library(DMwR)
library(e1071)
library(randomForest)
bostonRes <- performanceEstimation(
  PredTask(medv ~ .,Boston),
  c(workflowVariants(learner='rpartXse',
                     learner.pars=list(se=c(0,1))),
    workflowVariants(learner='svm',
                     learner.pars=list(cost=c(1,5),gamma=c(0.01,0.1))),
    workflowVariants(learner='randomForest',
                     learner.pars=list(ntree=c(500,1000)))),
  EstimationTask(metrics="mse",method=CV())
  )
```

Notice that this code involves estimating the mean squared error on the Boston Housing task for 8 different models through 10-fold cross validation (the `CV` function assumes `nFolds=10` by default).

On top of using out-of-the-box existing algorithms users also frequently apply standard data pre-processing steps before the models are obtained. Our `standardWF` function also implements a few examples of these steps that you can include in your comparisons. The following is a list of the parameters of this function that control these steps:

pre - A vector of function names that will be applied in sequence to the train and test data frames, generating new versions, i.e. a sequence of data pre-processing functions.

pre.pars - A named list of parameter values to be passed to the pre-processing functions.

We have implemented a few of these functions. Namely, in the **pre** argument you may use the following strings:

scale - that scales (subtracts the mean and divides by the standard deviation) any numeric features on both the training and testing sets. Note that the mean and standard deviation are calculated using only the training sample.

centralImp - that fills in any `NA` values in both sets using the median value for numeric predictors and the mode for nominal predictors. Once again these centrality statistics are calculated using only the training set although they are applied to both train and test sets.

na.omit - that uses the R function `na.omit` to remove any rows containing `NA`'s from both the training and test sets.

undersampl - this undersamples the training data cases that do not belong to the minority class (this pre-processing step is only available for classification tasks!). It takes the parameter **perc.under** that controls the level of undersampling (defaulting to 1, which means that there would be as many cases from the minority as from the other(s) class(es)).

smote - this operation uses the SMOTE [CBHK02] resampling algorithm to generate a new training sample with a more "balanced" distributions of the target class (this pre-processing step is only available for classification tasks!). It takes the parameters **perc.under**, **perc.over** and **k** to control the algorithm. Read the documentation of function `SMOTE` to know more details.

Note that you can also write your own data pre-processing functions provided you follow some protocol, and then use the name of your functions in the **pre** argument. Check the help page of function `standardPRE` to know the details on this simple protocol to write your own pre-processing functions.

The following is a simple example of using data pre-processing steps within our provided generic workflows:

```
library(performanceEstimation)
data(algae,package="DMwR")

res <- performanceEstimation(
         PredTask(a1 ~ .,algae[,1:12],"AlgaA1"),
         Workflow(learner="lm",pre=c("centralImp","scale")),
         EstimationTask(metrics="mae",method=CV())
         )

##
##
## ##### PERFORMANCE ESTIMATION USING  CROSS VALIDATION  #####
##
## ** PREDICTIVE TASK :: AlgaA1
##
## ++ MODEL/WORKFLOW :: lm
## Task for estimating  mae  using
##  1 x 10 - Fold Cross Validation
##    Run with seed =  1234
## Iteration :  1  2  3  4  5  6  7  8  9  10
```

Finally, standard workflows may also include some post-processing steps to be applied to the predictions of the model. These may include for instance some re-scaling of these predictions or even minimizing the risk of predictions through some cost-based approach. Function `standardWF` also accepts some parameters that control these post-processing steps. The following is a list of the parameters of this function that control these steps:

post - A vector of function names that will be applied in sequence to the predictions of the model, generating a new version, i.e. a sequence of data post-processing functions.

post.pars - A named list of parameter values to be passed to the post-processing functions.

As with pre-processing steps you may also write your own prediction post-processing functions (check the help page of `standardPOST` for details). Still, we currently provide the following alternatives:

na2central - this function fills in any `NA` predictions into either the median (numeric targets) or mode (nominal targets) of the target variable on the training set. Note that this is only applicable to predictions that are vectors of values.

onlyPos - in some numeric forecasting tasks the target variable takes only positive values. Nevertheless, some models may insist in forecasting negative values. This function casts these negative values to zero. Note that this is only applicable to predictions that are vectors of numeric values.

cast2int - in some numeric forecasting tasks the target variable takes only values within some interval. Nevertheless, some models may insist in forecasting values outside of this interval. This function casts these values into the nearest interval boundary. This function requires that you supply the limits of this interval through parameters **infLim** and **supLim**. Note that this is only applicable to predictions that are vectors of numeric values.

maxutil - maximize the utility of the predictions [Elk01] of a classifier. This method is only applicable to classification tasks and to algorithms that are able to produce as predictions a vector of class probabilities for each test case, i.e. a matrix of probabilities for a given test set. The method requires a cost-benefit matrix to be provided through the parameter **cb.matrix**. For each test case, and given the probabilities estimated by the classifier and the cost benefit matrix, the method predicts the classifier that maximizes the utility of the prediction. This approach [Elk01] is a slight 'evolution' of the original idea [BFOS84] that only considered the costs of errors and not the benefits of the correct classifications as in the case of cost-benefit matrices we are using here. The parameter **cb.matrix** must contain a (square) matrix of dimension $NClasses \times NClasses$ where entry $X_{i,j}$ corresponds to the cost/benefit of predicting a test case as belonging to class j when it is of class i. The diagonal of this matrix (correct predicitons) should contain positive numbers (benefits), whilst numbers

outside of the matrix should contain negative numbers (costs of misclassifications). See the Examples section of the help page of the function `standardPOST` for an illustration.

In the next example we illustrate the use of the post-processing routines by "correcting" the predictions of a linear regression model regards the frequency of occurrence of an alga, which can not be below zero:

```
library(performanceEstimation)
data(algae,package="DMwR")

res <- performanceEstimation(
         PredTask(a1 ~ .,algae[,1:12],"AlgaA1"),
         c(Workflow(wfID="lm",
                    learner="lm",
                    pre=c("centralImp","scale")),
           Workflow(wfID="lmOnlyPos",
                    learner="lm",
                    pre=c("centralImp","scale"),
                    post=c("onlyPos"))),
         EstimationTask(metrics="mae",method=CV())
         )

##
##
## ##### PERFORMANCE ESTIMATION USING  CROSS VALIDATION  #####
##
## ** PREDICTIVE TASK :: AlgaA1
##
## ++ MODEL/WORKFLOW :: lm
## Task for estimating  mae  using
##  1 x 10 - Fold Cross Validation
##    Run with seed =  1234
## Iteration :  1  2  3  4  5  6  7  8  9  10
##
## ++ MODEL/WORKFLOW :: lmOnlyPos
## Task for estimating  mae  using
##  1 x 10 - Fold Cross Validation
##    Run with seed =  1234
## Iteration :  1  2  3  4  5  6  7  8  9  10

summary(res)

##
## == Summary of a  Cross Validation Performance Estimation Experiment ==
##
## Task for estimating  mae  using
##  1 x 10 - Fold Cross Validation
##    Run with seed =  1234
##
## * Predictive Tasks ::  AlgaA1
## * Workflows  ::  lm, lmOnlyPos
##
## -> Task:  AlgaA1
##   *Workflow: lm
##            mae
## avg     14.309
## std      1.858
## med     14.340
## iqr      2.448
## min     11.652
```

```
## max     17.452
## invalid  0.000
##
##   *Workflow: lmOnlyPos
##            mae
## avg     13.116
## std      1.690
## med     13.173
## iqr      2.273
## min      9.956
## max     15.303
## invalid  0.000
```

As you see this simple post-processing step improved the performance of the model considerably.

4.2.2 Time Series Tasks

Our infra-structure also includes another generic workflow function that is specific for predictive tasks with time-dependent data (e.g. time series forecasting problems). This workflow function implements two different approaches to the problem of training a model with a set of time-dependent data and then use it to obtain predictions for a test set in the future. These two approaches contrast with the standard approach of learning a model with the available training sample and then use it to obtain predictions for all test period. This standard approach could be applied using the previously described `standardWF()` function. However, there are alternatives to this procedure, two of the most common being the sliding and growing window approaches, which are implemented in another workflow function developed specifically for time series tasks.

Predictive tasks for time-dependent data are different from standard classification and regression tasks because they require that the test samples have time stamps that are more recent then training samples. In this context, experimental methodologies handling these tasks should not shuffle the observations to maintain the time ordering of the original data. The most common setup is that we have a L time steps training window containing observations in the period $[t_1, t_L]$ and a F time steps test window typically containing the observations in the time window $[t_{L+1}, t_{L+F}]$.

The idea of the sliding window method is that if we want a prediction for time point t_k belonging to the test interval $[t_{L+1}, t_{L+F}]$ then we can assume that all data from t_{L+1} till t_{k-1} is already past, and thus usable by the model. In this context, it may be wise to use this new data in the interval $[t_{L+1}, t_{k-1}]$ to update the original model obtained using only the initial training period data. This is particularly advisable if we suspect that the conditions may have changed since the training period has ended. Model updating using the sliding window method is carried out by using the data in the L last time steps, i.e. every new model is always obtained using the last L data observations, as if the training window was slided forward in time. Our `timeseriesWF()` function implements this idea for both time series with a numeric target variable and a nominal target variable. This function has a parameter (`type`) that if set to "slide" will use a sliding window approach. As with the `standardWF()` function, this `timeseriesWF()` function also accepts parameters specifying the learner, predictor, and their respective parameters, as well as the previously defined pre-

and post-processing steps, all with exactly the same names and possible values (c.f. Section 4.2.1). Moreover, this function also includes an extra parameter, named `relearn.step`, which allows the user to establish the frequency of model updating. By default this is every new test sample, i.e. 1, but the user may set a less frequent model-updating policy by using higher values of this parameter to avoid high computation costs.

The idea of the growing window approach is very similar. The only difference is on the data used when updating the models. Whilst sliding window uses the data occurring in the last L time steps, growing window keeps increasing the original training window with the newly available data points, i.e. the models are obtained with increasingly larger training samples. By setting the parameter `type` to "grow" you get the `timseriesWF()` function to use this method.

The following code illustrates these two approaches by comparing them to the standard approach of using a single model to forecast all testing period:

```
library(performanceEstimation)
library(quantmod)
library(randomForest)
getSymbols('^GSPC',from='2008-01-01',to='2012-12-31')
data.model <- specifyModel(
  Next(100*Delt(Ad(GSPC))) ~ Delt(Ad(GSPC),k=1:10))
data <- as.data.frame(modelData(data.model))
colnames(data)[1] <- 'PercVarClose'
spExp <- performanceEstimation(
  PredTask(PercVarClose ~ .,data,'SP500_2012'),
  c(Workflow(wf='standardWF',wfID="standRF",
             learner='randomForest',
             learner.pars=list(ntree=500)),
    Workflow(wf='timeseriesWF',wfID="slideRF",
             learner='randomForest',
             learner.pars=list(ntree=500),
             type="slide",
             relearn.step=30),
    Workflow(wf='timeseriesWF',wfID="growRF",
             learner='randomForest',
             learner.pars=list(ntree=500),
             type="grow",
             relearn.step=30)
   ),
  EstimationTask(metrics=c("mse","theil"),
                 method=MonteCarlo(nReps=5,szTrain=0.5,szTest=0.25))
  )
```

The above example applies 3 different workflows to the task of trying to forecast the percentage daily variation of the prices of S&P 500, using some information of the previous prices as predictors. Namely, all workflows use a random forest with 500 trees but the predictions for each test set of the 5 repetitions Monte Carlo estimation methodology (c.f. Section 5.3.5), are obtained differently. The first workflow, named "standRF", obtains a single random forest with the training set and uses it to obtain predictions for the full test set. The other two approaches, instead of using this standard workflow, take advantage of the workflow provided by `timeseriesWF` and use either sliding or growing windows to obtain these predictions. For both these two latter approaches a new random forest is obtained after each 30 new test cases (set by parameter `relearn.step`).

4.3 User-defined Workflows

With the goal of ensuring that the proposed infra-structure is able to cope with all possible usage scenarios, we also allow the user to write and provide her/his own workflow functions to be used in the estimation tasks, provided they follow some protocol. Namely, these user-defined workflow functions should be written such that the first three parameters are: (i) the formula defining the predictive task; (ii) the provided training sample; and (iii) the test sample for which predictions are to be obtained. The functions may eventually accept other arguments with specific parameters of the user-defined workflow. The following is a general sketch of a user-defined workflow function:

```
myWorkFlow <- function(form,train,test,...) {
  require(mySpecialPackage,quietly=TRUE)
  ## cary out some data pre-processing
  myTrain <- mySpecificPreProcessingSteps(train)
  ## now obtain the model
  myModel <- myModelingTechnique(form,myTrain,...)
  ## obtain the predictions
  preds <- predict(myModel,test)
  ## cary out some predictions post-processing
  newPreds <- mySpecificPostProcessingSteps(form,train,test,preds)
  ## finally produce the worflow output object with the predictions
  trues <- responseValues(form,test)
  res <- WFoutput(rownames(test),trues,newPreds)
  return(res)
}
```

Not all workflows will require all these steps, though some may even require more. This is clearly something that is up to the user. The only strict requirements for these functions are: (i) the first 3 parameters of the workflow function should be the formula, train and test data frames; and (ii) the result of the function should be an object of the S4 class **WFoutput**.

Objects of class **WFoutput** contain the result of applying the workflow to a train+test partition. They are created using the constructor function **WFoutput()** that takes as first argument the row ids of the test set, as second the true values of the target variable in the test set and as third the predictions of the workflow for these test cases. These objects may optionally contain any other information the creator of the workflow function deems important to return. This optional information is "attached" to the object using the replacement function **workflowInformation()**. This function accepts as value a list whose content is completely free and left to the creator of the workflow function. As example of the usage of this function will be given later in this section.

The sketch shown above also illustrates the use of the function `responseValues()` that can be used to obtain the values of the target variable given a formula and a data frame.

On top of the 3 mandatory parameters (formula, training and test sets), user-defined workflow functions may also accept any other arguments. As we have seen in Section 4.1 we provide the function `workflowVariants()` to facilitate the specification of different variants of any workflow function by trying all combinations of several of its specific parameters. For instance, if the modelling function in the above example workflow (function `myModelingTechnique()`) had an integer parameter `x` and a Boolean parameter `y`, we could generate several **Work-**

flow objects to be evaluated/compared using the `performanceEstimation()` function, as follows:

```
workflowVariants('myWorkFlow',x=c(0,3,5,7),y=c(TRUE,FALSE))
```

This would generate 8 variants of the same workflow with all combinations of the specified values for the 2 parameters.

Let us see a concrete example of a user supplied workflow function. Imagine we want to evaluate a kind of ensemble model formed by a regression tree and a multiple linear regression model on an algae blooms data set [Tor10]. We write a workflow function that implements our intended workflow:

```
RLensemble <- function(f, tr, ts, weightRT=0.5, step=FALSE, ..., .models=FALSE) {
  require(DMwR,quietly=TRUE)
  ## Getting the column id of the target variable
  tgtCol <- which(colnames(tr) == as.character(f[[2]]))
  ## filling in NAs using knnImputation
  noNAsTR <- tr
  noNAsTS <- ts
  noNAsTR[,-tgtCol] <- knnImputation(tr[,-tgtCol])
  noNAsTS[,-tgtCol] <- knnImputation(ts[,-tgtCol],distData=tr[,-tgtCol])
  r <- rpartXse(f,tr,...)
  l <- lm(f,noNAsTR)
  if (step) l <- step(l,trace=0)
  pr <- predict(r,ts)
  pl <- predict(l,noNAsTS)
  ps <- weightRT*pr+(1-weightRT)*pl
  res <- WFoutput(rownames(ts),responseValues(f,ts),ps)
  if (.models) workflowInformation(res) <- list(linearModel=l,tree=r)
  res
}
```

This workflow starts by building two modified samples of the training and testing sets, with the `NA` values being filled in using a nearest neighbour strategy (see the help page of the function `knnImputation()` for more details). These versions are to be used by the `lm()` function that is unable to cope with cases with missing values. After obtaining the two models and their predictions the function calculates a weighted average of both predictions. Note how we have used the function `workflowInformation` to attach a list with the two models to the **WFoutput** results object, in case the user calls the function with `.models=TRUE`.

To evaluate different variants of this workflow we could run the following experiment:

```
data(algae,package='DMwR')
expRes <- performanceEstimation(
    PredTask(a1 ~ .,algae[,1:12],'alga1'),
    workflowVariants('RLensemble',
                     se=c(0,1),step=c(TRUE,FALSE),weightRT=c(0.4,0.5,0.6)),
    EstimationTask("mse",method=CV()))
```

5 Estimation Tasks

The third argument of the main function `performanceEstimation` defines the estimation task we want to carry out. Its value is an S4 object of class **EstimationTask** that can be created through the constructor function with the same

name. The main arguments of this constructor are **metrics** and **method**. The first is a vector with names of metrics for which we want reliable estimates, while the second is the method to be used to obtain these estimates. Other arguments of the constructor are **evaluator** and **evaluator.pars** that allow the specification of functions for calculating user-defined evaluation metrics. Finally, the last parameter of the constructor is **trainReq** that is a Boolean value indicating whether the training data should also be "sent" to the evaluation functions. This is useful for metrics that require the training data for being calculated (e.g. some normalized metrics use the average value of the target variable in the training set)

5.1 Performance Metrics

The package implements a reasonable set of the most common performance metrics. These include classification, regression and time series metrics. These are internally calculated by the functions `classificationMetrics` and `regressionMetrics`. The help pages of these two functions include an exhaustive list of this metrics together with their definitions. Depending on the predictive tasks being used one of these functions will be called to calculate the metrics the user indicates in the parameter **metrics** of the `EstimationTask` constructor. On top of the metrics implemented in these functions the user may also include the strings "trTime", "tsTime" and "totTime" to obtain estimates of the training, testing and total computation times taken by the workflow.

In case the user wants to estimate some metric not implemented in our package she/he needs to write a function implementing its calculation. This function needs to be written following a certain protocol that can be checked in the help page of the class **EstimationTask**. The name of the special purpose metric calculation function should be provided to the `EstimationTask` constructor through the parameter **evaluator**. Any parameters that are required to be passed to this user-defined function can also be passed through the parameter **evaluator.pars**.

5.2 User-defined Performance Metrics

Although the functions `classificationMetrics` and `regressionMetrics` implement a large set of performance metrics, it is inevitable that some applications may require some domain-specific metrics for which users want reliable estimates. Our package allows the indication of user-defined functions that calculate such domain-specific metrics. This is achieved through parameters **evaluator** and **evaluator.pars** of the **EstimationTask** constructor. The first allows the user to specify the name of such function, whilst the second is a list with parameter values to use when calling such function. In order to be usable by our package such functions need to obey some input/output protocol. Specifically, the user-defined functions should have as the first 3 parameters: (i) **trues** that receives the vector of true values of the target variable in the test set; (ii) **preds** that receives the vector of model predictions for such test cases; and (iii) **stats** that is a vector with the names of the metrics to be calculated (the user-defined functions may be able to calculate several metrics). Any further parameters of the function are up to the user and may be set through values in the **evaluator.pars** list. The user-defined functions must return as a result a

(named) vector with as many scores as the number of metrics specified in the parameter **stats**.

Suppose we want to calculate the error of a regression model as the difference between true and predicted values raised to some power. We could define a function to calculate such metric following our package input/output protocol as:

```
powErr <- function(trues,preds,stats,pow=3) {
    if (stats != "pow.err") stop("Unable to calculate that metric!")
    c(pow.err = mean((trues-preds)^pow))
}
```

Using such function in the context of some 10-fold cross validation estimation experiment would involve creating an estimation task as:

```
EstimationTask(metrics="pow.err",method=CV(),
               evaluator="powErr",evaluator.pars=list(pow=4))
```

5.3 Estimation Methodologies

There are different ways of providing reliable estimates of the predictive performance of a workflow. Our infra-structure implements some of the most common estimation methods. In this section we briefly describe them and provide short illustrative examples of their use.

The parameter **method** of the **EstimationTask** constructor allows the user to specify the estimation methodology that will be used. The next sections explain the options available.

5.3.1 Cross Validation

k-Fold cross validation (CV) is one of the most common methods to estimate the predictive performance of a model. By including an S4 object of class **CV** in the parameter **method** we can carry out experiments of this type.

The constructor function `CV()` can be used to obtain objects of class **CV**. It accepts the following parameters:

nReps - the number of repetitions of the k-fold CV experiment (default is 1)

nFolds - the number of k folds to use (default is 10)

seed - the random number generator seed to use (default is 1234)

strat - whether to use stratified samples on the different iterations (default is `FALSE`)

dataSplits - a list containing the data splits to use on each repetition of a k-folds CV experiment (defaulting to 'NULL'). Check the help page of the class **CV** for further details.

Bellow you can find a small illustration using the Breast Cancer data set available in package **mlbench**. On this example we compare some variants of an SVM using a $3 \times 10-$fold cross validation process with stratified sampling because one of the two classes has a considerably lower frequency.

```
data(BreastCancer,package='mlbench')
library(e1071)
bcExp <- performanceEstimation(
  PredTask(Class ~ .,BreastCancer[,-1],'BreastCancer'),
  workflowVariants('standardWF',
           learner='svm',
           learner.pars=list(cost=c(1,5),gamma=c(0.01,0.1))
          ),
  EstimationTask(metrics=c("F","prec","rec"),
                 evaluator.pars=list(posClass="malignant"),
                 method=CV(nReps=3,nFolds=10,strat=TRUE)))
```

Please note the use of the `evaluator.pars` parameter of the `EstimationTask` constructor function. We have used it to indicate which of the class labels of this problem should be considered the "positive" class, which is required to compute the values of the F, recall and precision metrics we are estimation through $3 \times 10-$fold cross validation. This parameter setting is passed to the `classificationMetrics` function that is the default for classification tasks like Breast Cancer.

5.3.2 Bootstrapping

Bootstrapping or bootstrap resampling is another well-known experimental methodology that is implemented in our package. Namely, we implement two of the most common methods of obtaining bootstrap estimates: ϵ_0 and .632 bootstrap. By including an S4 object of class **Bootstrap** in the **method** argument we can carry out experiments of this type.

Function `Bootstrap()` can be used as a constructor of objects of class **Bootstrap**. It accepts the following arguments:

type - a string with the type of bootstrap estimates: either "e0" for ϵ_0 bootstrap, or ".632" for .632 bootstrap (default is "e0")

nReps - the number of repetitions of the bootstrap experiment (default is 200)

seed - the random number generator seed to use (default is 1234)

dataSplits - a list containing user-supplied data splits for each of the repetitions (check the help page of the class for further details). This parameter defaults to `NULL`, i.e. no user-supplied splits, they are decided internally by the infra-structure.

Bellow you can find a small illustration using the Servo data set available in package **mlbench**. On this example we compare some variants of an artificial neural network using 100 repetitions of a bootstrap experiment.

```
data(Servo,package='mlbench')
library(nnet)
nnExp <- performanceEstimation(
  PredTask(Class ~ .,Servo),
  workflowVariants(learner='nnet',
                   learner.pars=list(trace=F,linout=T,
                       size=c(3,5),decay=c(0.01,0.1))
          ),
  EstimationTask("mse",method=Bootstrap(nReps=100)))
```

5.3.3 Holdout and Random Sub-sampling

The Holdout is another frequently used experimental methodology, particularly for large data sets. To carry out this type of experiments in our infra-structure we can include an S4 object of class **Holdout** in the third argument of function `performanceEstimation()`.

Function `Holdout()` can be used as a constructor of objects of class **Holdout**. It accepts the following arguments:

nReps - the number of repetitions of the Holdout experiment (default is 1)

hldSz - the percentage of cases (a number between 0 and 1) to leave as holdout (test set) (default is 0.3)

seed - the random number generator seed to use (default is 1234)

strat - whether to use stratified samples (default is `FALSE`)

dataSplits - a list containing user-supplied data splits for each of the repetitions (check the help page of the class for further details). This parameter defaults to `NULL`, i.e. no user-supplied splits, they are decided internally by the infra-structure.

Please note that for the usual meaning of Holdout the number of repetitions should be 1 (the default), while larger values of this parameter correspond to what is usually known as random subsampling.

The following is a small illustrative example of the use of the random subsampling with the LetterRecognition classification task from package **mlbench**.

```
data(LetterRecognition,package='mlbench')
ltrExp <- performanceEstimation(
    PredTask(lettr ~ .,LetterRecognition),
    workflowVariants(learner='rpartXse',
                     learner.pars=list(se=c(0,1)),
                     predictor.pars=list(type='class')
                     ),
    EstimationTask(metrics="err",method=Holdout(nReps=3,hldSz=0.3)))
```

Please note the use of the `predictor.pars` parameter of our `standardWF()` function to be able to cope with the fact that the `predict` method for classification trees requires the use of `type="class"` to get actual predicted class labels instead of class probabilities.

5.3.4 Leave One Out Cross Validation

Leave one out cross validation is a type of cross validation method that is mostly used for small data sets. You can think of leave one out cross validation as a k-fold cross validation with k equal to the size of the available data set. To carry out this type of experiments in our infra-structure we can include an S4 object of class **LOCV** in the third argument of function `performanceEstimation()`.

Function `LOOCV()` can be used as a constructor of objects of class **LOOCV**. It accepts the following arguments:

seed - the random number generator seed to use (default is 1234)

dataSplits - a list containing user-supplied data splits for each of the repetitions (check the help page of the class for further details). This parameter defaults to `NULL`, i.e. no user-supplied splits, they are decided internally by the infra-structure.

The following is a small illustrative example of the use of the LOOCV with the Iris classification task.

```
data(iris)
library(e1071)
irisExp <- performanceEstimation(
    PredTask(Species ~ .,iris),
    workflowVariants(learner='svm',
                     learner.pars=list(cost=c(1,10))
                     ),
    EstimationTask(metrics="acc",method=LOOCV()))
```

5.3.5 Monte Carlo Experiments

Monte Carlo experiments are similar to random sub-sampling (or repeated Hold-out) in the sense that they consist of repeating a learning + testing cycle several times using different and eventually overlapping data samples. The main difference lies on the way the samples are obtained. In Monte Carlo experiments the original order of the observations is respected and train and test splits are obtained such that the testing samples appear "after" the training samples, thus being the methodology of choice when you are comparing time series forecasting models. The idea of Monte Carlo experiments is the following: (i) given a data set spanning from time t_1 till time t_N, (ii) given a training set time interval size w_{train} and a test set time interval size w_{test}, such that $w_{train} + w_{test} < N$, (iii) Monte Carlo experiments generate r random time points from the interval $[t_{1+w_{train}}, t_{N-w_{test}}]$, and then (iv) for each of these r time points they generate a training set with data in the interval $[t_{r-w_{train}+1}, t_r]$ and a test set with data in the interval $[t_{r+1}, t_{r+w_{test}}]$. Using this process r train+test cycles are carried out using the user-supplied workflow function, and the experiment estimates result from the average of the r scores as usual. The overall process is depicted in Figure 2.

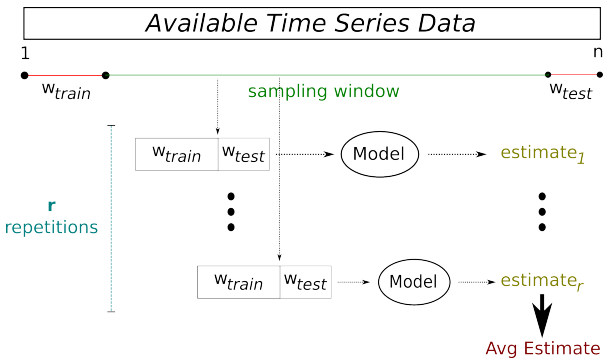

Figure 2: The Monte Carlo estimation methodology.

To carry out this type of experiments in our infra-structure we can include an S4 object of class **MonteCarlo** in the third argument of function `performanceEstimation()`.

The function `MonteCarlo()` can be used as a constructor of objects of class **MonteCarlo**. It accepts the following arguments:

nReps - the number of repetitions of the Monte Carlo experiment (default is 10)

szTrain - the percentage (a number between 0 and 1) or the actual number of cases to use in the training samples (default is 0.25)

szTest - the percentage (a number between 0 and 1) or the actual number of cases to use in the test samples (default is 0.25)

seed - the random number generator seed to use (default is 1234)

dataSplits - a list containing user-supplied data splits for each of the repetitions (check the help page of the class for further details). This parameter defaults to `NULL`, i.e. no user-supplied splits, they are decided internally by the infra-structure.

The following is a small illustrative example using the quotes of the SP500 index. This example compares two random forests with 500 regression trees, one applied in a standard way, and the other using a sliding window with a relearn step of 5 days. The experiment uses 10 repetitions of a train+test cycle using 50% of the available data for training and 25% for testing.

```
library(quantmod)
library(randomForest)
getSymbols('^GSPC',from='2008-01-01',to='2012-12-31')
data.model <- specifyModel(
  Next(100*Delt(Ad(GSPC))) ~ Delt(Ad(GSPC),k=1:10))
data <- modelData(data.model)
colnames(data)[1] <- 'PercVarClose'
spExp <- performanceEstimation(
  PredTask(PercVarClose ~ .,data,'SP500_2012'),
  c(Workflow('standardWF',wfID="standRF",
             learner='randomForest',learner.pars=list(ntree=500)),
    Workflow('timeseriesWF',wfID="slideRF",
             learner='randomForest',
             learner.pars=list(ntree=500,relearn.step=30))
   ),
  EstimationTask(metrics="theil",
                 method=MonteCarlo(nReps=10,szTrain=0.5,szTest=0.25)))
```

Note that in the above example we have not tried any variants of the two workflows that are applied to the task. This means that we have used directly the `Workflow` constructor to create our workflow. Note also the use of the `wfID` parameter of this constructor to allow you to give a particular workflow ID to some approach.

6 Statistical Significance of Differences

The estimation methodologies that we have presented in the previous sections allow the user to obtain estimates of the mean predictive performance of different workflows or variants of these workflows, on different predictive tasks. We have seen that by applying the `summary` method to the objects resulting from the estimation experiments we can obtain the average performance for each candidate workflow on each task. These numbers are estimates of the expected mean performance of the workflows on the respective tasks. Being estimates, the obvious next question is to check whether the observed differences in performance between the workflows are statistically significant. More formally, we want to know that confidence level of rejecting the null hypothesis that the difference between the estimated averages is zero.

That is the goal of the function `pairedComparisons()`. This function implements a series of statistical hypothesis tests that can be used in different types of setups. We follow the general recommendations of Demsar [Dem06]. The function `pairedComparisons()` implements a series of statistical tests that can be used under different conditions. The function calculates them all (whenever possible) and it is up to the user to use the ones that are more adequate to answer her/his research questions (see Demsar [Dem06] for a series of guidelines on how to proceed).

Function `pairedComparisons()` returns a list with as many components as there are metrics estimated. For each metric another list includes the different tests that are carried out as well as several other information. Namely, for each error metric the returned list includes the following components:

- **setup** - with generic information on the estimation task
- **avgScores** - a matrix with the average scores on the metric of each workflow on each task
- **medScores** - another matrix with information like the previous one, but instead of the average we get the median scores
- **rks** - a matrix with the rank positions corresponding to the average scores
- **avgRksWfs** - a vector with the average of the above ranks across all tasks
- **t.test** - a list containing information concerning paired t tests
- **WilcoxonSignedRank.test** - a list containing information concerning paired *Wilcoxon Signed Rank* tests
- **F.test** - a list containing information concerning the F test
- **Nemenyi.test** - a list containing information concerning the post-hoc *Nemenyi* test
- **BonferroniDunn.test** - a list containing information concerning the post-hoc *Bonferroni-Dunn* test

For both the t and *Wilcoxon Signed Rank* tests the components contain and array with 3 dimensions, the third dimension being the task, which means

that for each task you get a matrix of results of paired comparisons. Namely, paired comparisons between each workflow and the baseline workflow. Each row of the matrix contains the average (or median in the case of Wilcoxon) score of the respective workflow, the difference between this score and the score of the baseline workflow, and the p value associated with the hypothesis that this difference is distributed around mean (median in Wilcoxon). Please note that although our function calculates the results of the t test, this is not recommended for most settings given the lack of independence of the values on each iteration (for instance on k-fold cross validation each of the k scores is obtained with training sets that have a string overlap).

The other 3 tests can be used in the following two settings: (i) test if there are significant differences among any pair of workflows; and (ii) test the significance of the differences between all workflows against a baseline. In both of these two settings we should start by checking the results of the F test to check if we can reject the null hypothesis that the average ranks of all workflows are equivalent. In case this hypothesis is rejected (component `rejNull` of the list `F.test`) we can move to the post-hoc tests. In the setup (i) we should use the *Nemenyi* test to find the critical difference among ranks above which we can say that the respective difference is statistically significant. The *Nemenyi.test* component is a list with components containing this information (critical difference value, ranking differences among all pairs of workflows, and whether these differences are or not significant). Under the setup (ii) the post-hoc test that should be used is the *Bonferroni-Dunn* test. In this case we just want to check if the difference between the average rank of each workflow and that of the baseline is or not significant (thus much less paired comparisons). The result is again given as a list where we have information on the critical difference, the baseline workflow name, the vector of average differences and whether these differences are different or not.

Let us see a concrete example of using this statistical tests. Suppose we want to apply several variants of an SVM to three classification tasks and we want to check if there are significant differences among them. The following test carries out the performance estimation task:

```
library(e1071)
data(PimaIndiansDiabetes,package='mlbench')
data(iris)
data(Glass,package="mlbench")
res <- performanceEstimation(
    c(PredTask(diabetes ~ .,PimaIndiansDiabetes,"Pima"),
      PredTask(Type ~ ., Glass),
      PredTask(Species ~ .,iris)),
    workflowVariants(learner="svm",
                     learner.pars=list(cost=1:5,gamma=c(0.1,0.01,0.001))),
    EstimationTask(metrics="err",method=CV()))
```

The object `res` can be passed to `pairedComparisons()` to calculate the tests described before:

```
pres <- pairedComparisons(res)
```

As we have several workflows being compared on different tasks we will start by checking the results of the Friedman F test:

```
pres$err$F.test
```

```
## $chi
## [1] 18.57
##
## $FF
## [1] 1.586
##
## $critVal
## [1] 0.8892
##
## $rejNull
## [1] TRUE
```

We can see that the null hypothesis that the average ranks of the workflows are equivalent can be rejected and thus we can proceed to the post-hoc test. In case we wish the check the hypothesis that there are statistically significant differences among the workflows we should proceed with a Nemenyi post-hoc test. The results of this test can be inspected as follows:

```
pres$err$Nemenyi.test
```

```
## $critDif
## [1] 12.38
##
## $rkDifs
##         svm.v1 svm.v2 svm.v3 svm.v4 svm.v5 svm.v6 svm.v7 svm.v8 svm.v9 svm.v10 svm.v11
## svm.v1  0.0000 1.5000 1.0000 1.8333 1.5000 0.6667 0.8333 0.1667 1.3333   2.833   7.667
## svm.v2  1.5000 0.0000 0.5000 0.3333 0.0000 2.1667 2.3333 1.3333 0.1667   1.333   9.167
## svm.v3  1.0000 0.5000 0.0000 0.8333 0.5000 1.6667 1.8333 0.8333 0.3333   1.833   8.667
## svm.v4  1.8333 0.3333 0.8333 0.0000 0.3333 2.5000 2.6667 1.6667 0.5000   1.000   9.500
## svm.v5  1.5000 0.0000 0.5000 0.3333 0.0000 2.1667 2.3333 1.3333 0.1667   1.333   9.167
## svm.v6  0.6667 2.1667 1.6667 2.5000 2.1667 0.0000 0.1667 0.8333 2.0000   3.500   7.000
## svm.v7  0.8333 2.3333 1.8333 2.6667 2.3333 0.1667 0.0000 1.0000 2.1667   3.667   6.833
## svm.v8  0.1667 1.3333 0.8333 1.6667 1.3333 0.8333 1.0000 0.0000 1.1667   2.667   7.833
## svm.v9  1.3333 0.1667 0.3333 0.5000 0.1667 2.0000 2.1667 1.1667 0.0000   1.500   9.000
## svm.v10 2.8333 1.3333 1.8333 1.0000 1.3333 3.5000 3.6667 2.6667 1.5000   0.000  10.500
## svm.v11 7.6667 9.1667 8.6667 9.5000 9.1667 7.0000 6.8333 7.8333 9.0000  10.500   0.000
## svm.v12 6.6667 8.1667 7.6667 8.5000 8.1667 6.0000 5.8333 6.8333 8.0000   9.500   1.000
## svm.v13 2.3333 3.8333 3.3333 4.1667 3.8333 1.6667 1.5000 2.5000 3.6667   5.167   5.333
## svm.v14 1.0000 2.5000 2.0000 2.8333 2.5000 0.3333 0.1667 1.1667 2.3333   3.833   6.667
## svm.v15 1.0000 2.5000 2.0000 2.8333 2.5000 0.3333 0.1667 1.1667 2.3333   3.833   6.667
##         svm.v12 svm.v13 svm.v14 svm.v15
## svm.v1    6.667   2.333  1.0000  1.0000
## svm.v2    8.167   3.833  2.5000  2.5000
## svm.v3    7.667   3.333  2.0000  2.0000
## svm.v4    8.500   4.167  2.8333  2.8333
## svm.v5    8.167   3.833  2.5000  2.5000
## svm.v6    6.000   1.667  0.3333  0.3333
## svm.v7    5.833   1.500  0.1667  0.1667
## svm.v8    6.833   2.500  1.1667  1.1667
## svm.v9    8.000   3.667  2.3333  2.3333
## svm.v10   9.500   5.167  3.8333  3.8333
## svm.v11   1.000   5.333  6.6667  6.6667
## svm.v12   0.000   4.333  5.6667  5.6667
## svm.v13   4.333   0.000  1.3333  1.3333
## svm.v14   5.667   1.333  0.0000  0.0000
## svm.v15   5.667   1.333  0.0000  0.0000
##
## $signifDifs
##         svm.v1 svm.v2 svm.v3 svm.v4 svm.v5 svm.v6 svm.v7 svm.v8 svm.v9 svm.v10 svm.v11
## svm.v1   FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE   FALSE   FALSE
## svm.v2   FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE   FALSE   FALSE
## svm.v3   FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE   FALSE   FALSE
## svm.v4   FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE   FALSE   FALSE
## svm.v5   FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE   FALSE   FALSE
## svm.v6   FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE   FALSE   FALSE
## svm.v7   FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE   FALSE   FALSE
## svm.v8   FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE   FALSE   FALSE
## svm.v9   FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE   FALSE   FALSE
## svm.v10  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE   FALSE   FALSE
## svm.v11  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE   FALSE   FALSE
## svm.v12  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE   FALSE   FALSE
## svm.v13  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE   FALSE   FALSE
## svm.v14  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE   FALSE   FALSE
## svm.v15  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE   FALSE   FALSE
##         svm.v12 svm.v13 svm.v14 svm.v15
## svm.v1    FALSE   FALSE   FALSE   FALSE
## svm.v2    FALSE   FALSE   FALSE   FALSE
## svm.v3    FALSE   FALSE   FALSE   FALSE
## svm.v4    FALSE   FALSE   FALSE   FALSE
## svm.v5    FALSE   FALSE   FALSE   FALSE
## svm.v6    FALSE   FALSE   FALSE   FALSE
## svm.v7    FALSE   FALSE   FALSE   FALSE
```

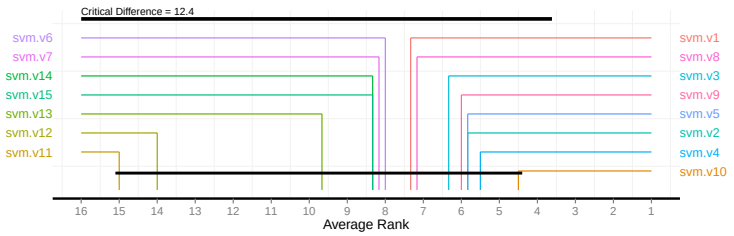

Figure 3: The CD diagram of the Nemenyi post-hoc test.

```
## svm.v8    FALSE   FALSE   FALSE   FALSE
## svm.v9    FALSE   FALSE   FALSE   FALSE
## svm.v10   FALSE   FALSE   FALSE   FALSE
## svm.v11   FALSE   FALSE   FALSE   FALSE
## svm.v12   FALSE   FALSE   FALSE   FALSE
## svm.v13   FALSE   FALSE   FALSE   FALSE
## svm.v14   FALSE   FALSE   FALSE   FALSE
## svm.v15   FALSE   FALSE   FALSE   FALSE
```

This is obviously a too extensive set of information though the conclusions are centered in the `signifDifs` component. An even better alternative can be obtained through CD diagrams, which we have implemented in function `CDdiagram.Nemenyi` whose result is shown in Figure 3.

```
CDdiagram.Nemenyi(pres)
```

At the top left of the diagram you can see a horizontal line with the critical difference for the differences between average ranks to be considered statistically significant. Then each workflow is represented by a line with origin in its respective average rank (in the X axis). If the lines of any pair of workflows are connected by a horizontal black line, it means that the difference between them is not statistically significant. In the diagram of Figure 3 we can observe that all lines are connected meaning that none of the paired differences between all workflows can be seen as statistically significant.

7 Larger Examples

The main advantage of the infra-structure we are proposing is to automate large scale performance estimation experiments. It is on these very large setups that the use of the infra-structure saves more time to the user. However, in these contexts the objects resulting from the estimation process are very large and some of the tools we have shown before for exploring the results may produce over-cluttered output. In effect, if you have an experiment involving dozens of predictive tasks and eventually hundreds of workflow variants being compared

on several evaluation metrics, doing a plot of the resulting object is simply not possible as the graph will be unreadable. This section illustrates some of these cases and presents some solutions to overcome the difficulties they bring.

Extremely large experiments may take days or weeks to complete, depending on the available hardware. In this context, it may not be wise to run the experiments on a single call to the `performanceEstimation()` function because if something goes wrong in the middle you may loose lots of work/time. Using the random number generation seeds that are available in all experimental settings objects we can split the experiments in several calls and still ensure that the same data folds are used in all estimation experiments. If the seeding process is not enough due to the usage of different hardware for instance, then you may still resort to the user supplied data splits to make sure all methods are compared on the same data. We will see that when all experiments are finished we will be able to merge the objects of each call into a single object as if we had issued a single call.

Another way of improving performance on large experiments is through parallel computation. package `performanceEstimation` includes facilities to run experiment in parallel through the existing parallel back-ends in R. The simplest of options, that has the advantage of not requiring you to know anything at all about parallel computation in R, is to call the `performanceEstimation` function with a fourth parameter that will make all experiments to run in parallel on the different cores of any standard multicore computer. The following is a simple example of achieving this effect:

```
library(performanceEstimation)
library(e1071)
data(Satellite,package="mlbench")
pres <- performanceEstimation(
    PredTask(classes ~ .,Satellite),
    Workflow(learner="svm"),
    EstimationTask("err",CV()),
    cluster=TRUE
    )

##
##
## ##### PERFORMANCE ESTIMATION USING  CROSS VALIDATION  #####
##
## ** PREDICTIVE TASK :: Satellite.classes
##
## ++ MODEL/WORKFLOW :: svm
## cvEstimates: Running in parallel with 2 worker(s)
## Task for estimating  err  using
##  1 x 10 - Fold Cross Validation
##   Run with seed =  1234
```

By adding this extra parameter the lower level functions that implement the estimation procedures (in this case cross validation but the concept is applicable to any of the other methods), will automatically create a (local) parallel backend using half of the cores of your computer. On reasonably large data sets this simple extra setting will typically lead to cut execution times by a significant amount.

Other more complex parallel settings are also possible but they require you to know how to create the clusters before calling our function. For instance, you could create a cluster (using function `makeCluster` of package **parallel**) that involved several computers. The object of class `cluster` that results from these steps should then be sent through parameter `cluster` of our `performanceEstimation` function so that the function can take advantage of this parallel back-end. The following is a simple illustrative example:

```
library(parallel)
myclust <- makeCluster(c("localhost","192.168.2.10","192.168.2.13"),"SOCK")
library(performanceEstimation)
library(e1071)
data(Satellite,package="mlbench")
pres <- performanceEstimation(
    PredTask(classes ~ .,Satellite),
    Workflow(learner="svm"),
    EstimationTask("err",CV()),
    cluster=myclust
    )
stopCluster(myclust)
```

Please note that there are a few assumptions on the above example, namely concerning access of your username in localhost to the remote machines whose IP's are given. This means that the username exists on all machines and that appropriate SSH keys have been set so that no passwords are required for accessing the remote machines.

Let us now focus on the issue of splitting a large experiment in partial calls to `performanceEstimation()` and at the end merge the separate results into a single object as if the experiments were run all together. Please note that further efficiency gains could be achieved if some parallel setup involving different computers as the one described above was used. The following example addresses several regression tasks using several workflows that are variants of different regression algorithms. We illustrate how to create the different **PredTask** objects programmatically, how to create the different workflow variants, and how to call `performanceEstimation()` with different task/workflow combinations storing the intermediate objects on temporary files for later reading and merging,

```
library(performanceEstimation)
library(e1071)
library(randomForest)

data(algae,package="DMwR")
DSs <- sapply(names(algae)[12:18],
         function(x,names.attrs) {
           f <- as.formula(paste(x,"~ ."))
           PredTask(f,algae[,c(names.attrs,x)],x,copy=TRUE)
         },
         names(algae)[1:11])

WFs <- list()
```

```
WFs$svm <- list(learner.pars=list(cost=c(10,150,300),
                                  gamma=c(0.01,0.001),
                                  epsilon=c(0.1,0.05)),
                pre="centralImp",post="na2central")
WFs$randomForest <- list(learner.pars=list(mtry=c(5,7),
                                           ntree=c(500,750,1500)),
                         pre="centralImp")

for(d in seq_along(DSs)) {
  for(w in names(WFs)) {
    resObj <- paste(names(DSs)[d],w,'Res',sep='')
    assign(resObj,
           performanceEstimation(
                  DSs[d],
                  c(
                    do.call('workflowVariants',
                            c(list(learner=w),WFs[[w]]))
                    ),
                   EstimationTask(metrics=c('mse','mae'),method=CV(nReps=3)),
                   cluster=TRUE)
           )

    save(list=resObj,file=paste(names(DSs)[d],w,'Rdata',sep='.'))
  }
}
```

The above code compares 12 SVM variants with 6 random forest variants, on 7 algae blooms regression tasks, using $3 \times 10-$fold cross validation. Although this is not a very large experimental comparison it still includes applying 18 different workflow variants on 7 different prediction tasks, 30 times, i.e. 3780 train+test cycles. Instead of running all these experiments on a single call to the function `performanceEstimation()` (which would obviously still be possible), we have made different calls for each workflow type (SVM and random forest) and for each predictive task. This means that each call will run all variants of a certain workflow on a certain predictive task. The result of each of these calls will be assigned to an object with a name composed of the task name and workflow learner (the `resObj` variable). In the end each of these objects is saved on a file with a similar name, for future loading and results analysis. For instance, in the end there will be a file with name "a1.svm.Rdata" which contains an object of class **ComparisonResults** named `a1svmRes`. This object contains the MSE and MAE estimated scores of the SVM variants on the task of predicting the target variable "a1" (one of the seven algae in this data set).

Later on, after the above experiment has finished you can load the saved objects back into R and moreover, join them into a single object, as shown below:

```
nD <- paste('a',1:7,sep='')
nL <- c('svm','randomForest')
res <- NULL
for(d in nD) {
  resD <- NULL
  for(l in nL) {
    load(paste(d,l,'Rdata',sep='.'))
```

```
    x <- get(paste(d,l,'Res',sep=''))
    resD <- if (is.null(resD)) x else mergeEstimationRes(resD,x,by='workflows')
  }
  res <- if (is.null(res)) resD else mergeEstimationRes(res,resD,by='tasks')
}
save(res,file='allResultsAlgae.Rdata')
```

The `mergeEstimationRes()` function when applied to objects of class **ComparisonResults** allows merging of these objects across different dimensions. Namely, such objects have the individual scores of all experiments spread across 3 dimensions: the metrics, the workflows and the tasks. The argument `by` of the `mergeEstimationRes()` function allows you to specify how to merge the given objects. The most common situations are: (i) merging the results of different workflows over the same data sets - you should use "`by='workflows'`", or (ii) merging the results of the same workflows across different tasks - you should use "`by='tasks'`".

The following code can be used to check that the merging was OK, and also to illustrate a few other utility functions whose purpose should be obvious:

```
res
```

```
##
## ==  Cross Validation Performance Estimation Experiment ==
##
## Task for estimating  mse,mae  using
##  3 x 10 - Fold Cross Validation
##   Run with seed =  1234
##
##  18  workflows applied to  7  predictive tasks
```

```
taskNames(res)
```

```
## [1] "a1" "a2" "a3" "a4" "a5" "a6" "a7"
```

```
workflowNames(res)
```

```
##  [1] "svm.v1"          "svm.v2"          "svm.v3"          "svm.v4"
##  [5] "svm.v5"          "svm.v6"          "svm.v7"          "svm.v8"
##  [9] "svm.v9"          "svm.v10"         "svm.v11"         "svm.v12"
## [13] "randomForest.v1" "randomForest.v2" "randomForest.v3" "randomForest.v4"
## [17] "randomForest.v5" "randomForest.v6"
```

```
metricNames(res)
```

```
## [1] "mse" "mae"
```

With such large objects the most we can do is obtaining the best scores or rankings of the workflows:

```
topPerformers(res)
```

```
## $a1
##           Workflow Estimate
## mse randomForest.v3  265.374
## mae randomForest.v1   11.066
##
```

```
## $a2
##            Workflow Estimate
## mse randomForest.v5   94.463
## mae         svm.v11    5.917
##
## $a3
##     Workflow Estimate
## mse   svm.v3   44.442
## mae  svm.v10    3.911
##
## $a4
##            Workflow Estimate
## mse randomForest.v5   18.238
## mae         svm.v10    1.882
##
## $a5
##            Workflow Estimate
## mse randomForest.v1   44.415
## mae          svm.v7    4.037
##
## $a6
##     Workflow Estimate
## mse   svm.v6  111.757
## mae  svm.v12    5.548
##
## $a7
##            Workflow Estimate
## mse randomForest.v3   26.208
## mae         svm.v10    2.383
```

```
rankWorkflows(res)
```

```
## $a1
## $a1$mse
##          Workflow Estimate
## 1 randomForest.v3    265.4
## 2 randomForest.v5    265.6
## 3 randomForest.v1    266.0
## 4 randomForest.v6    272.3
## 5 randomForest.v4    272.6
##
## $a1$mae
##          Workflow Estimate
## 1 randomForest.v1    11.07
## 2 randomForest.v3    11.07
## 3 randomForest.v5    11.08
## 4 randomForest.v4    11.19
## 5 randomForest.v2    11.20
##
##
## $a2
## $a2$mse
##          Workflow Estimate
## 1 randomForest.v5    94.46
## 2 randomForest.v6    94.88
## 3 randomForest.v3    94.91
## 4 randomForest.v1    94.95
## 5 randomForest.v4    95.03
##
## $a2$mae
##   Workflow Estimate
## 1  svm.v11    5.917
## 2   svm.v1    5.938
## 3  svm.v10    5.941
## 4   svm.v4    5.954
## 5   svm.v5    5.968
##
##
## $a3
## $a3$mse
##          Workflow Estimate
## 1          svm.v3    44.44
```

```
## 2          svm.v2    44.54
## 3          svm.v8    45.27
## 4          svm.v9    45.31
## 5 randomForest.v5    46.31
##
## $a3$mae
##   Workflow Estimate
## 1  svm.v10    3.911
## 2   svm.v4    3.915
## 3   svm.v7    3.943
## 4   svm.v1    3.956
## 5   svm.v5    4.007
##
##
## $a4
## $a4$mse
##          Workflow Estimate
## 1 randomForest.v5    18.24
## 2 randomForest.v3    18.25
## 3 randomForest.v1    18.26
## 4          svm.v4    18.37
## 5 randomForest.v2    18.56
##
## $a4$mae
##   Workflow Estimate
## 1  svm.v10    1.882
## 2   svm.v4    1.883
## 3   svm.v7    1.984
## 4   svm.v1    1.994
## 5  svm.v11    2.022
##
##
## $a5
## $a5$mse
##          Workflow Estimate
## 1 randomForest.v1    44.42
## 2 randomForest.v3    44.43
## 3 randomForest.v5    44.50
## 4 randomForest.v4    45.51
## 5 randomForest.v2    45.53
##
## $a5$mae
##   Workflow Estimate
## 1   svm.v7    4.037
## 2  svm.v10    4.056
## 3   svm.v4    4.061
## 4   svm.v1    4.074
## 5   svm.v5    4.077
##
##
## $a6
## $a6$mse
##          Workflow Estimate
## 1          svm.v6    111.8
## 2         svm.v12    113.1
## 3          svm.v5    121.9
## 4         svm.v11    123.2
## 5 randomForest.v3    123.3
##
## $a6$mae
##   Workflow Estimate
## 1  svm.v12    5.548
## 2   svm.v7    5.572
## 3   svm.v6    5.587
## 4   svm.v4    5.633
## 5   svm.v1    5.644
##
##
## $a7
## $a7$mse
##          Workflow Estimate
## 1 randomForest.v3    26.21
## 2 randomForest.v5    26.22
```

```
## 3 randomForest.v1    26.33
## 4 randomForest.v2    26.45
## 5 randomForest.v4    26.50
##
## $a7$mae
##   Workflow Estimate
## 1  svm.v10    2.383
## 2   svm.v4    2.408
## 3   svm.v7    2.419
## 4   svm.v1    2.440
## 5   svm.v6    2.440
```

Notice that both `topPerformers()` and `rankWorkflows()` assume that the evaluation metrics are to be minimized, i.e. they assume the lower the better the scores. Still, both functions have a parameter named `maxs` that accepts a vector with as many Boolean values as there are evaluation metrics being estimated, which you may use to indicate that some particular metric is to be maximized and not minimized (the default). So for instance, if you had an experiment where the 1st and 3rd metrics are to be minimized, whilst the second is to be maximized, you could call these functions as `rankWorkflows(resObj,maxs=c(F,T,F))`.

In order to obtain further results from these large objects one usually proceeds by analyzing parts of the object, for instance focusing on a particular task or metric, or even a subset of the workflows. To facilitate this we can use the generic function `subset()` that can also be applied to objects of class **ComparisonResults**. An example of its use is given below, which results in a graph of the performance of the different workflows in the predictive task "a1", in terms of "MAE", which is show in Figure 4.

```
plot(subset(res, tasks='a1', metrics='mae'))
```

As before we are using the generic function `plot()` but this time applied to a subset of the original object with all results. This subset is obtained using the generic function `subset()` that accepts several parameters to specify the subset we are interested on. In this case we are using the parameters `tasks` and `metrics` to indicate that we want to analyze only the results concerning the task "a1" and the metric "mae". Other possibility is the parameter `workflows` for indicating a subset of the workflows. Both `workflows`, `tasks` and `metrics` accept as values a character string containing a regular expression that will be used internally with the R function `grep()` over the vector of names of the respective objects (names of the workflows, names of the tasks and names of the metrics, respectively). For instance, if you want to constrain the previous graph even further to the workflows whose name ends in "4" (absurd example of course!), you could use the following:

```
plot(subset(res, tasks='a1', workflows='4$'))
```

If you are more familiar with the syntax of "wildcards" you may use the R function `glob2rx()` to convert to regular expressions, as show in the following example:

```
summary(subset(res, tasks='a1', workflows=glob2rx('*svm*'),metrics='mse'))

##
## == Summary of a  Cross Validation Performance Estimation Experiment ==
```

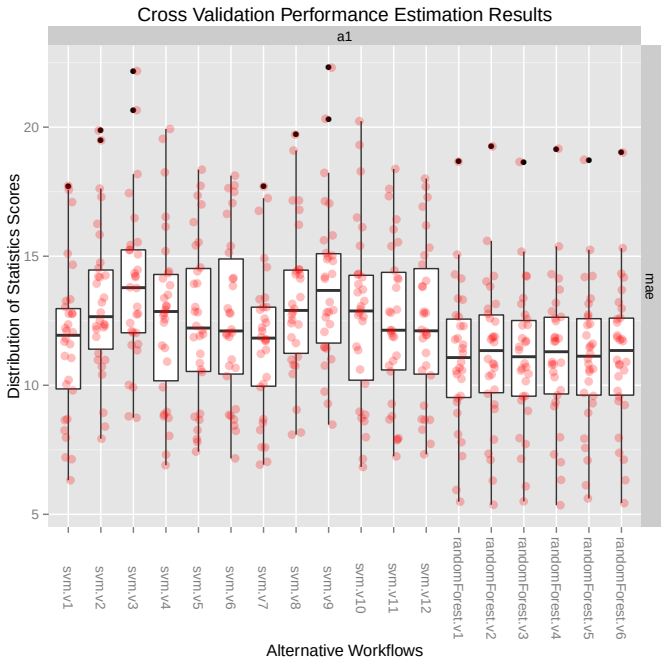

Figure 4: The MAE results for the task "a1".

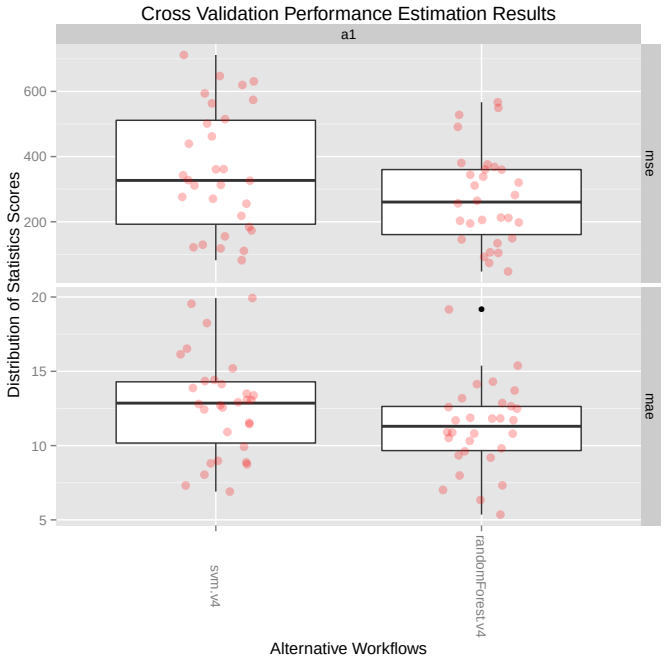

Figure 5: Illustration of the use of regular expressions in sub-setting the results objects.

```
##
## Task for estimating  mse  using
##  3 x 10 - Fold Cross Validation
##   Run with seed =  1234
##
## * Predictive Tasks ::  a1
## * Workflows  ::  svm.v1, svm.v2, svm.v3, svm.v4, svm.v5, svm.v6, svm.v7, svm.v8, svm.v9, svm.v10,
##
## -> Task:  a1
##   *Workflow: svm.v1
##           mse
## avg     305.37
## std     147.63
## med     275.08
## iqr     211.22
## min      79.09
## max     605.37
## invalid   0.00
##
##   *Workflow: svm.v2
##           mse
## avg     349.2
## std     171.6
## med     301.5
## iqr     196.5
## min     121.0
## max     821.4
## invalid   0.0
##
##   *Workflow: svm.v3
##           mse
## avg     367.2
## std     180.4
## med     345.7
## iqr     203.3
## min     142.1
## max     916.0
## invalid   0.0
##
##   *Workflow: svm.v4
##           mse
## avg     356.54
## std     187.29
## med     326.87
## iqr     318.84
## min      82.04
## max     711.67
## invalid   0.00
##
##   *Workflow: svm.v5
##           mse
## avg     376.4
## std     235.4
```

```
## med     283.0
## iqr     308.7
## min     105.0
## max     975.8
## invalid   0.0
##
##   *Workflow: svm.v6
##            mse
## avg      396.2
## std      287.6
## med      285.4
## iqr      294.4
## min       98.4
## max     1260.5
## invalid    0.0
##
##   *Workflow: svm.v7
##            mse
## avg     304.24
## std     144.76
## med     271.52
## iqr     212.01
## min      78.69
## max     587.87
## invalid   0.00
##
##   *Workflow: svm.v8
##            mse
## avg     355.6
## std     176.4
## med     313.4
## iqr     191.0
## min     130.2
## max     862.9
## invalid   0.0
##
##   *Workflow: svm.v9
##            mse
## avg     375.2
## std     188.1
## med     350.2
## iqr     195.9
## min     143.0
## max     954.9
## invalid   0.0
##
##   *Workflow: svm.v10
##            mse
## avg     359.46
## std     188.58
## med     331.29
## iqr     320.81
## min      82.85
```

```
## max     721.56
## invalid   0.00
##
##   *Workflow: svm.v11
##            mse
## avg     367.1
## std     216.2
## med     282.4
## iqr     288.6
## min     101.3
## max     959.6
## invalid   0.0
##
##   *Workflow: svm.v12
##              mse
## avg      376.06
## std      244.88
## med      284.02
## iqr      278.97
## min       94.37
## max     1179.99
## invalid    0.00
```

The following are some illustrations of the use of other available utility functions.

Obtaining the scores on all iterations and metrics of a workflow on a particular task:

```
getScores(res, 'svm.v6','a3')
```

```
##           mse   mae
##  [1,]  58.773 4.040
##  [2,]  63.421 4.836
##  [3,] 124.213 6.630
##  [4,]  20.818 2.902
##  [5,]  28.508 3.202
##  [6,]  18.425 2.774
##  [7,]  14.446 2.509
##  [8,]  54.975 4.584
##  [9,]  23.522 2.871
## [10,] 135.025 6.735
## [11,] 112.234 5.329
## [12,]  81.117 5.570
## [13,]  34.504 3.779
## [14,]   9.791 2.163
## [15,]  30.126 3.477
## [16,]  58.715 4.655
## [17,]  23.793 3.171
## [18,]  26.481 3.237
## [19,]  99.573 5.928
## [20,]  27.569 3.294
## [21,]   8.496 2.129
## [22,]  27.933 3.197
## [23,]  20.164 2.851
## [24,]  72.373 5.365
## [25,]  69.470 4.814
## [26,] 126.076 5.477
## [27,]  65.167 5.459
```

```
## [28,]   9.438 2.300
## [29,]  22.697 3.512
## [30,]  90.988 5.114
```

Getting the summary of the results of a particular workflow on a predictive task :

```
estimationSummary(res,'svm.v3', 'a7')

##             mse   mae
## avg      30.516 3.157
## std      29.266 1.213
## med      23.096 3.017
## iqr      31.305 1.053
## min       3.084 1.394
## max     141.577 7.616
## invalid   0.000 0.000
```

Finally, the `metricsSummary()` function allows you to apply any summary function (defaulting to `mean()`) to the iterations estimates. The following calculates the median of the results of the SVMs on the task "a1",

```
metricsSummary(subset(res, workflows=glob2rx('*svm*'), tasks='a1'),
               summary='median')

## $a1
##     svm.v1 svm.v2 svm.v3 svm.v4 svm.v5 svm.v6 svm.v7 svm.v8 svm.v9 svm.v10 svm.v11
## mse 275.08 301.55 345.65 326.87 282.99  285.4 271.52  313.4 350.18  331.29  282.41
## mae  11.94  12.66  13.78  12.86  12.22   12.1  11.83   12.9  13.67   12.88   12.13
##     svm.v12
## mse  284.02
## mae   12.11
```

8 Conclusions

We have presented package `performanceEstimation` that aims at being a general package for estimating and comparing the performance of any approach to any predictive task in R. The package allows users to very easily carry out standard out-of-the-box comparative experiments between existing modeling tools in R. However, it also allows the users to supply their own functions implementing special workflows to solve tasks. This means that the package should cover the needs of occasional users as well as advanced users that which to try and compared their own proposed workflows.

The package `performanceEstimation` also includes several facilities for testing the statistical significance of the observed differences, namely implementing the current state of the art in this subject as described in [Dem06], including CD diagrams for both the Nemenyi and Bonferroni-Dunn post-hoc tests.

Finally, we have provided a few illustrations of how to use package `performanceEstimation` for larger experiments, namely taking advantage of parallel computation that is available in R.

References

[BFOS84] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Statistics/Probability Series. Wadsworth & Brooks/Cole Advanced Books & Software, 1984.

[CBHK02] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *JAIR*, 16:321–357, 2002.

[Dem06] J. Demsar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30, 2006.

[Elk01] Charles Elkan. The foundations of cost-sensitive learning. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI'2001)*, 2001.

[MDH+12] David Meyer, Evgenia Dimitriadou, Kurt Hornik, Andreas Weingessel, and Friedrich Leisch. *e1071: Misc Functions of the Department of Statistics (e1071), TU Wien*, 2012. R package version 1.6-1.

[Tor10] Luis Torgo. *Data Mining with R: learning with case studies*. Chapman & Hall/CRC Press, 2010.