Final project for computational statistics

Manuel Bottino, Patrick Poetto, Jacopo Spagliardi

2024-10-27

Contents

1 References

[1] C. M. Bishop. *Pattern Recognition and Machine Learning*. Ed. by M. Jordan. Information Science and Statistics. Springer, 2006.

[2] L. Breiman. "Bagging predictions". In: *Machine Learning* 24.2 (1996), pp. 123-140.

[3] e. a. Han Jiawei. *Data Mining Concepts and Techniques*. Morgan Kaufmann Publishers, 2023.

[4] e. a. Hastie Trevor. *The Elements of Statistical Learning*. Vol. 27. 2. 2009. Chap. 9, p. 745.

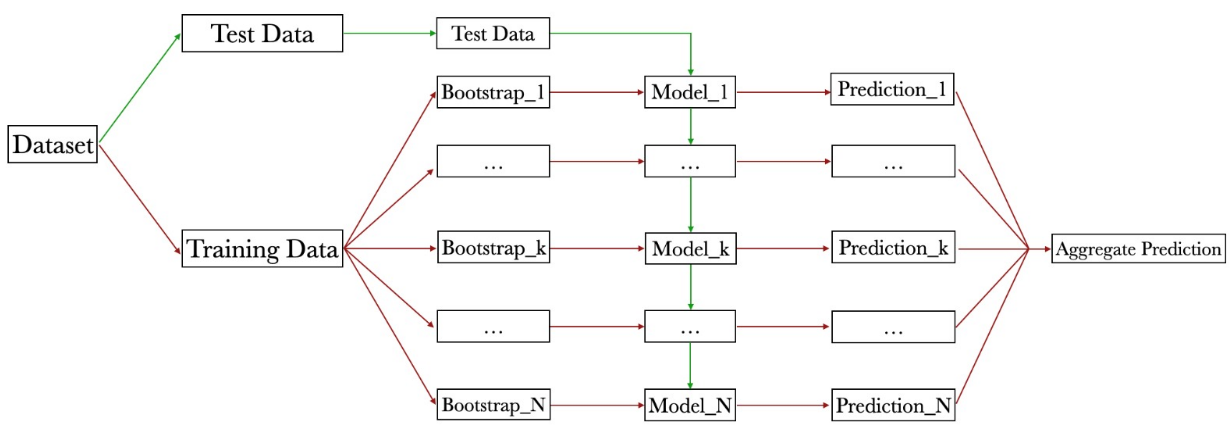
[5] e. a. Kuhn Max. *Applied Predictive Modeling*. Springer, 2016.

[6] e. a. Seni Giovanni. *Ensemble Methods in Data Mining: Improving Accuracy Through Combining Predictions*. Ed. by C. Robert Grossman University of Illinois. Synthesis Lectures on Data Mining and Knowledge Discovery. Morgan&Claypool Publishers, 2010.

2 Review

The underlying idea of bagging is to create multiple instances of the base model (e.g., a decision tree) trained on randomly selected samples of data with replacement (bootstrapping, that's why this technique is also called Bootstrap aggregating). Each base model is trained on subsets of the original data, independently of the others, producing a prediction. The predictions of the individual models are then combined using a majority voting rule (for classification problems) or averaging (for regression problems) to obtain a final prediction. By creating different instances of the base model and combining their predictions, bagging can reduce variability and improve overall predictive performance. Let's see how this type of ensemble technique works:

- Divide our dataset randomly into two different groups of rows:
  - Training Data: the larger part of our group that will be used to produce new datasets.
  - Testing Data: the smaller part that will be used to see the performance of our technique.
- Produce with bootstrap technique new datasets using Training Data. Each bootstrap sample is created by randomly selecting with replacement from the original sample. This means that some rows may be selected multiple times, while others may be excluded to reach the dimension of the original sample. It produced a number N of samples
- Using each sample we produce N models (always the same type of model, for example, linear regression model) that we can call M_k with $k = 1, ..., N$
- Then we can use each model to produce predictions using Testing Data.
- Assembling all the predictions in a democratic way if the variable we are predicting is categorical, and in an averaging way if the variable is continuous
- This assembled prediction is more accurate than the prediction made by the same model but create only uses the original dataset.



There are some scenarios in which it may not work or provide significant improvements; for example:

- When the dataset is small: we need a large training dataset to create different istances of the base model.
- When there is strong dependence among training data: create different instances in this case during bootstrap may not introduce enough diversity.

- When the predictors are uninformative: bagging may not be able to improve model performance. In fact, bagging could increase the importance of a predictor.

An application example of issues bagging may produce is provided using the bike sharing dataset from the kaggle repository, applied to a simple linear regression problem.

```r
ols = lm(cnt ~ ., data = data)
summary(ols)
```

```
##
## Call:
## lm(formula = cnt ~ ., data = data)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -11723  -1927     -6   1748  27894
##
## Coefficients:
##                         Estimate Std. Error t value Pr(>|t|)
## (Intercept)           20047.6414 28328.5743   0.708 0.479624
## t1                     1176.6899   445.8862   2.639 0.008696 **
## t2                     -330.9358   343.7709  -0.963 0.336396
## hum                    -123.3030    36.7792  -3.353 0.000890 ***
## wind_speed             -219.6227    38.2706  -5.739 2.10e-08 ***
## seasonspring          -3753.3208   825.7186  -4.546 7.61e-06 ***
## seasonsummer          -1087.3169   984.2647  -1.105 0.270066
## seasonwinter          -2646.4438   685.4697  -3.861 0.000135 ***
## weather_codeclear      -939.2478   662.0169  -1.419 0.156878
## weather_codecloudy     1454.9593  1275.5175   1.141 0.254801
## weather_codefew clouds  -707.8169   637.6232  -1.110 0.267742
## weather_coderain      -3216.5621   837.2999  -3.842 0.000146 ***
## is_holiday_we         -5516.5725   436.3260 -12.643  < 2e-16 ***
## sunshine                 29.7677   174.3640   0.171 0.864544
## global_radiation         41.0780    10.5490   3.894 0.000119 ***
## max_temp                -93.4597    88.1941  -1.060 0.290027
## min_temp               -176.4441   121.6067  -1.451 0.147713
## precipitation          -196.5107    54.9000  -3.579 0.000394 ***
## pressure                  0.1083     0.2695   0.402 0.688031
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3695 on 342 degrees of freedom
## Multiple R-squared:  0.8162, Adjusted R-squared:  0.8065
## F-statistic: 84.36 on 18 and 342 DF,  p-value: < 2.2e-16
```

Watching this naive model example we immediately see that there are some uninformative predictors, so we want to verify if bagging doesn't work very well in this case as expected.

```r
# Devide the dataset
train_indices <- sample(1:nrow(data), 0.9 * nrow(data))
train_data <- data[train_indices, ]
test_data <- data[-train_indices, ]
```

```r
# Bootstrap
n_bootstraps <- 100
train_datasets <- lapply(1:n_bootstraps, function(i) train_data[sample(1:nrow(train_data),
                                                                    replace = TRUE), ])

# Models
models <- map(train_datasets, function(data)   lm(cnt ~ ., data = data))

# Extraction of betas
coefficients <- lapply(models, coef)

# Bagging predictions
predictions <- list()

for (i in 1:length(models)) {
  predictions[[i]] <- predict(models[[i]], newdata = test_data)
}

predictions_bag <- rowMeans(do.call(cbind, predictions))

# Ols predictions
predictions_ols = predict(ols, newdata = test_data)

# Comparison
residuals_bag = test_data$cnt - predictions_bag
residuals_ols <- test_data$cnt - predictions_ols

std_error_ols <- sqrt(sum(residuals_ols^2) /
                        (length(residuals_ols) - length(ols$coefficients)))

std_error_bag <- sqrt(sum(residuals_bag^2) /
                        (length(residuals_bag) - length(ols$coefficients)))
```

```r
std_error_bag
```

```
## [1] 5101.75
```

```r
std_error_ols
```

```
## [1] 4765.122
```

We immediately see that the standard error of the bagged is bigger than the standard error of the predictions produced by the original ols model because of the presence of uninformative predictors.

2.1 Why it works

First of all let's introduce the general idea of bagging. Given a dataset $L = \{(x_n, y_n),\ n = 1, \dots, N\}$ we try to improve a predicting procedure in a "naive" way, ideally we would like to have a sequence of datasets $\{L_k\}$ each containing N independent observations, then take the average of the sequence $\{\phi(x, L_k)\}$.

$$\phi_A(x, P) = E_L[\phi(x, L)] \tag{1}$$

Because ϕ_A depends not only on x but also on the underlying probability distribution P from which L is drawn. Since we are missing this last information the only tool we are left with is our dataset L, we instead use P_L the bootstrap approximation of P, it can be defined as the probability mass function with equal probability of extracting each element of the dataset (x_n, y_n). Finally we can define the bootstrap aggregate predictor as:

$$\phi_B(x) = \phi_A(x, P_L) \tag{2}$$

In order to understand why bagging works theoretically it can be proved that mean-squared error (MSE) of $\phi_A(x)$ is lower than the mean-squared error averaged over L of $\phi(x, L)$. How much lower the two sides are depends on the inequality:

$$[E_L \phi(x, L)]^2 \leq E_L \phi^2(x, L) \tag{3}$$

This result is true taking advantage of the Jensen's inequality for the specific case in which $g(X) = X^2$, this function is convex ($g'' > 0$), thus $E[Z^2] \geq (E[Z])^2$

2.2 Instability

The inequality 3 is a nice starting point to explain what role instability plays. In fact, If $\phi(x, L)$ does not change too much with replicate L the two sides will be nearly equal, and aggregation will not help. The more highly variable the $\phi(x, L)$ are, the more improvement aggregation may produce. Applying this reasoning to our $\phi_B(x)$, if the procedure is unstable, it can give improvement through aggregation. However, if the procedure is stable, then $\phi_B(x) = \phi_A(x, P_L)$ will not be as accurate for data drawn from P as $\phi_A(X, P) \sim \phi(x, L)$.

Let's see how this concept can be translated when we look at a more specific context, like classification. In this instance the predictor $\phi(\boldsymbol{x}, L)$ predicts a label $j \in \{1, ..., J\}$. We first define $Q(j|x) = P(\phi(x, L) = j)$, over many independent replicates of the learning set L, ϕ predicts class label j at input x with relative frequency $Q(j|x)$, and let $P(j|x)$ be the probability that input x generates class j. At this point we can set $\phi^*(x) = \arg\max_j P(j|x)$ (the Bayes predictor) which leads to the following expression for $Q(j|x)$:

$$\begin{cases} 1 \ if \ P(j|x) = max_i P(i|x) \\ 0 \ elsewhere \end{cases}$$

Now we have all the ingredients to show the maximum classification rate for:

$$r^* = \int \max_j P(j|x) P_X(dx) \tag{4}$$

where $P_X(dx)$ is the probability distribution of X.

Also, call ϕ order-correct at the input $\boldsymbol{x}$ if:

$$\arg\max_j Q(j|x) = \arg\max_j P(j|x) = \tag{5}$$

If we now focus on the aggregate of ϕ and define it following the procedure described above $\phi_A(x) = \arg\max_j Q(j|x)$. We have the maximum attainable correct-classification rate (accuracy) of $\phi(x)$, we are missing the corret-classification for x, which for $\phi_A(x)$ is $\sum_j I(\arg\max_j Q(j|x) = j) P(j|x)$ Where I is the indicator function.

Putting all the pieces together the correct-classification rate for $\phi_A(x)$ is:

$$r_A = \int_{\boldsymbol{x}\in C} max_j P(j|\boldsymbol{x})P_X(d\boldsymbol{x}) + \int_{\boldsymbol{x}\in C'} [\sum_j (I(\phi_A(\boldsymbol{x}) = j)P(j|\boldsymbol{x})]P_X(\boldsymbol{x}) \quad (6)$$

Even if ϕ is order correct at x its correct classification rate can be far from optimal, but ϕ_A is optimal. Only if ϕ is order-correct for most input of x (it is good), then aggregation can transform it into a nearly optimal predictor. On the other hand, unlike the numerical prediction situation, poor predictors can be transformed into worse ones if we use bagging.

2.3 Model - Decision Tree-Based Methods

Tree-based methods partition the feature space into a set of rectangles, and then fit a simple model, say a constant, in each one. This is a simple yet powerful procedure since it is able to disentangle a model into simpler and smaller models and describe his features more accurately than global models do basically using binary conditional clustering. The geometric perspective described before can be seen as a tree where data are run and at each node a test is conducted to see what is the path an observation should follow until reaching a leaf, which represents the final prediction explained by the constant model. For example, let's say we have p inputs and a response, for each of N observations: that is, (x_i, y_i) for $i = 1, 2, ..., N$, with $x_i = (x_{i1}, x_{i2}, ..., x_{ip})$. The algorithm has decide on the splitting variables and split points, as well as what shape the tree should have. Suppose first that we have a partition into M regions $R_1, R_2, ..., R_M$, and we model the response as a constant c_m in each region: As a criterion for optimal partitional we can minimize the sum of squares $\sum(y_i - f(x_i))^2$. In this way the best $\hat{c}_m$ is just the average of y_i in region R_m: $\hat{c}_m = av(y_i|x_i \in R_m)$. Our model will be then: $f(x) = \sum_{m=1}^{M} c_m \cdot I(x \in R_m)$.

Now finding the best binary partition in terms of minimum sum of squares is generally computationally infeasible so we can set up a CART (classification and regression tree) algorithm starting with the data, a splitting variable j, a split point s, and defining the half planes as: $R_1(j, s) = \{X \mid X_j \leq s\}$ and $R_2(j, s) = \{X \mid X_j > s\}$. Then we seek j and s that solve

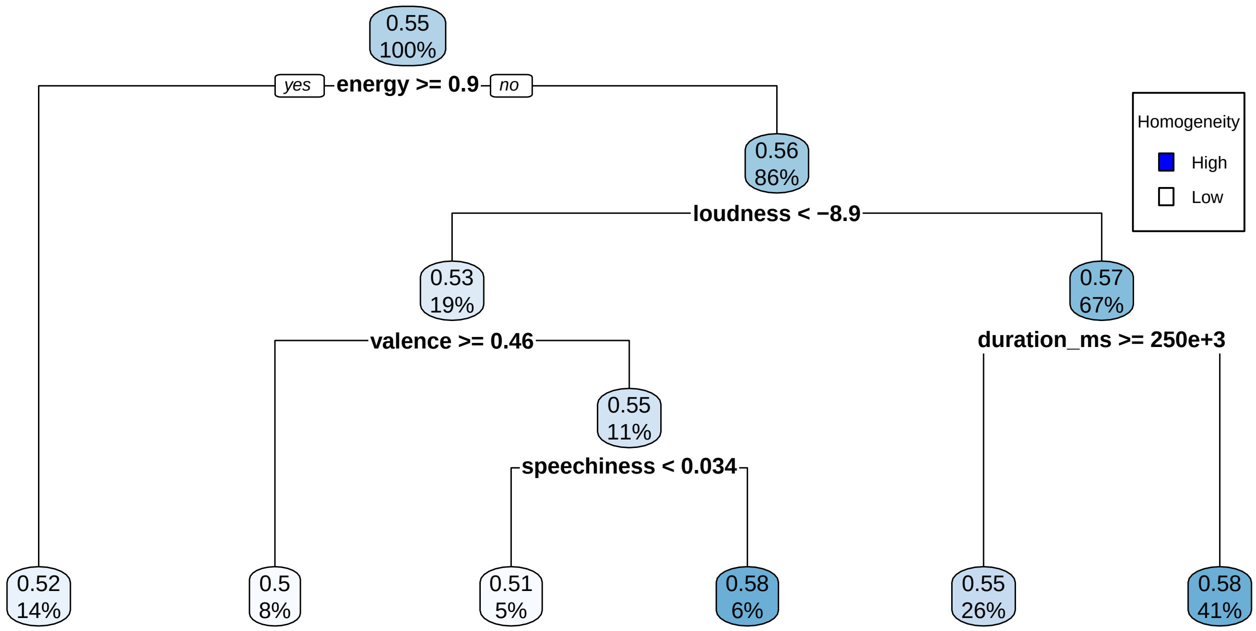
$$\min_{j,s} \left[\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \right]$$

For any choice j and s, the inner minimization is solved by:

$$\hat{c}_1 = \text{av}(y_i \,|\, x_i \in R_1(j, s)) \quad \text{and} \quad \hat{c}_2 = \text{av}(y_i \,|\, x_i \in R_2(j, s))$$

For each j, the split point s can be found very quickly and hence determination of the best pair (j, s) is feasible by brute force. Having found the best split, we partition the data into the two resulting regions and repeat the splitting process on each of the two regions. Then this process is repeated on all of the resulting regions. The question now become: how large should we grow the tree? It is pretty straightforward that too many nodes (splits) may overfit the data while doing vice versa may end up not being able to capture them well, resulting in misprediction. One strategy could be to set a lower threshold for the decrease of the sum of squares and stop the splitting when this is reached. However this strategy is too short sighted since a seemingly worthless split may lead to a very good one below. A more robust strategy may be to do kind of the opposite: grow a very large tree and then use a cost-complexity pruning criterion to collapse one internal node at a time from the full tree until the single node tree so that we find a sequence. It is intuitive that the optimal tree must be somewhere in the sequence. Now, with a cross validation selection method, we can find the actual optimal tree just minimizing the cross validated sum of squares.

This is how the CART algorithm for growing decision trees basically works. Note that decision trees are divided in classification and regression trees if the response is a factor or a numerical or continous variable.



Predicted Value	Median Value
0.5845508	0.55

We used the well known Spotify dataset to grow a basic tree with the function "rpart". The thresholds at each node are defined by the algorithm which minimizes the MSE of the model. The color of each nodes represent homogeneity across data which have been classified in a node. Each node show the percentage of observations that it contains and the outcome prediction for that node. Then we used the function "predict" applied to our grown tree to make prediction on a virtual new observation, providing the median of each variable as a new value and getting predicted popularity of a song which have the median as each feature available in the dataset. Not surprisingly, the predicted value for popularity is near the median of the covariate popularity itself.

2.4 Algorithm Setup

Bootstrap aggregating Regression Trees can be done following the procedure proposed by **(Breiman, 1996)**:

1. Randomly divide a real dataset into a 10% test $\mathcal{T}$ and a 90% learning set $\mathcal{L}$. If the dataset is simulated then we can consider a $15-85$ or a $20-80$ proportion.

2. Grow a regression tree from $\mathcal{L}$ using a 10-fold cross validation, then run $\mathcal{T}$ down the tree to obtain the squared error $e_S(L, T)$. Using che k-folds cross validation means that $\mathcal{L}$ is divided into 10 folds, and for each iteration, a regression tree is trained on 9 folds and evaluated on the remaining fold. The process is repeated 10 times, such that each fold is utilized as the test set once. Ultimately, the best model, i.e. the regression tree with the best average performance across all test folds, is selected as the final result. At the end of the 10 iterations, the squared errors obtained for each test fold are used to calculate the mean squared error $e_S(L, T)$, which represents the average squared difference between the predictions of the regression tree and the actual values of the test set.

3. Select a bootstrap sample $\mathcal{L}_B$ from $\mathcal{L}$ and use it to grow a tree. Then use $\mathcal{L}$ to prune the tree avoiding overfitting. Repeat this step 25 times to obtain predictors $\phi_1(\mathbf{x}), ..., \phi_{25}(\mathbf{x})$.

4. For $(y_n, \mathbf{x_n}) \in \mathcal{T}$, the bagged predictor will be $\hat{y}_n = av_k \phi_k(\mathbf{x_n})$, and the squared error $e_B(\mathcal{L}, \mathcal{T}) = av_n(y_n - \hat{y}_n)^2$.

5. Divide the data into $\mathcal{L}$ and $\mathcal{T}$ for 100 times and average the errors to obtain $\bar{e}_S$ and $\bar{e}_B$.

2.5 Application of algorithm

Try to replicate the algorithm provided by the paper for regression trees using 10-fold cross-validation and bagging

```r
set.seed(655)
# Function that randomize learning and testing datasets, point 5 of algorithm
# performs this 100 time. To each division the author applies both methods (cross
# validation and bagging).

random_training_test=function(df,k, bin_80_20=0){
  fold_indices <- sample(cut(seq(1, nrow(df)), breaks = k, labels = FALSE,
                             ordered_result = FALSE))
  if (bin_80_20==1) {
    test_indices <- which(fold_indices == 1 | fold_indices == 2)
  }else {
    test_indices <- which(fold_indices == 1)}

  train_i <- df[-test_indices, ]
  test_i <- df[test_indices, ]
  return(list(train_i,test_i))
}



k_fold_reg_tree= function(df,k, y){
  formula <- as.formula(paste(y, "~", "."))

  # 1. take the dataset and divide it into learning and testing (90% - 10%)
  # which is already done by the function random_training_test
  train_i=df[[1]]
  test_i=df[[2]]


  # 2. take the learning set and apply 10-fold cv to find the model with lowest MSE

  fold_indices <- sample(cut(seq(1, nrow(train_i)), breaks = k, labels = FALSE,
                             ordered_result = FALSE) )

  cv_result=Inf

  for (i in 1:k) {


  # Split the data into training and testing sets (10-fold cross validation)
  test_indices <- which(fold_indices == i)
  train_data <- train_i[-test_indices, ]
  test_data <- train_i[test_indices, ]
```

```r
  # Train a regression tree on the training set
  model <- rpart(formula, data = train_data)

  # Predict the target variable for the testing set
  predictions <- predict(model, newdata = test_data)


  #Calculate the evaluation metric (mean squared errors)
  result <- mean((predictions - test_data[[y]])^2)
  # 3. For every candidate we compute the MSE compered to the original test

  if(result<cv_result){
    cv_result=result
    # Compute mse of this model using the original test set as comparison
    predictions <- predict(model, newdata = test_i)
    final_res=mean((predictions - test_i[[y]])^2)
  }
  }
  return(final_res)
}

reg_tree_boot= function(df,b, y){
  formula <- as.formula(paste(y, "~", "."))

  # 1. take the dataset and divide it into learning and testing (90% - 10%)
  # which is already done by the function
  train_i=df[[1]]
  test_i=df[[2]]
  predictions=rep(0, nrow(test_i))

  # 2. compute 25 predictions with different bootstrap samples
  for (i in 1:b) {
  train_data=slice_sample(train_i,n=nrow(train_i), replace=TRUE)
  model <- rpart(formula, data = train_data)

  # point three of the algorithm is also about pruned trees using the original
  # training set. the function prune determines a nested sequence of subtrees
  # of the supplied rpart object by recursively snipping off the least important splits,
  # based on the complexity parameter (cp).
  pruned_tree <- rpart::prune(model, cp=0.01, newdata=train_i)

  # Predict the target variable for the testing set
  predictions <- predictions + predict(pruned_tree, newdata = test_i)

  }
  # 3. take the mean of these predictors and compute the MSE
  predictions<-predictions/b
  final_res=mean((predictions - test_i[[y]])^2)
  return(final_res)
}

# I wrote down y instead of the standard MEDV (median of prices) so my lapply below
# can be computed
```

```r
housing <- read.table("data/housing.csv", quote="\"", comment.char="")
colnames(housing)=c('CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS',
                    'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'y')

# I wrote down y instead of the standard O3 (ozone levels) so my lapply below
# can be computed
colnames(ozone)[1]="y"

B=1000

friedman1=t(replicate(B, {
  fried1=mlbench.friedman1(n=1200, sd=1)
  data.frame(cbind(fried1$x,fried1$y))}
  ))
colnames(friedman1)=c("x_1","x_2","x_3","x_4","x_5","x_6","x_7","x_8","x_9","x_10","y")

friedman2=t(replicate(B, {
  fried2 <- mlbench.friedman2(n = 1200)
  data.frame(cbind(fried2$x,fried2$y))}
  ))
colnames(friedman2)=c("x_1","x_2","x_3","x_4","y")

friedman3=t(replicate(B, {
  fried3 <- mlbench.friedman3(n = 1200)
  data.frame(cbind(fried3$x,fried3$y))}
  ))
colnames(friedman3)=c("x_1","x_2","x_3","x_4","y")




datasets=list(housing, ozone, friedman1, friedman2, friedman3)


# the list in which we will store all of our mse (both from cross validation and
# bagging) for each dataset, and the decrease in mse (computed below).
e_bar_global=list()

# Smart lapply to perform 10-fold cv and bagging for all the datasets
e_bar_global=lapply(datasets, function(data){
  colMeans(t(replicate(B, {
  # the only utility of this if is to apply 80-20 instead of 90-10 to the simulated
  # datasets (friedman from 1 to 3)
  if (ncol(data)==5 | ncol(data)==11){
    i=sample(1:B, 1)
    df=random_training_test(data.frame(data[i,]),10,1)
  }else{(df=random_training_test(data,10))
  }
  mse_cv=k_fold_reg_tree(df,10,"y")
  mse_bag=reg_tree_boot(df,25,"y")
  c(mse_cv, mse_bag)
})))
})
```

```r
# Smart lapply to compute the decrease in MSE for all of the datasets
e_bar_global=lapply(e_bar_global,
                    function(data){data[3]=((data[1]-data[2])/data[1])*100
                                  round(data,2)
  })
```

Dataset name	$\bar{e}_S$	$\bar{e}_B$	Decrease
Boston Housing	23.16	16.58	28.4%
Ozone	23.11	18.46	20.14%
Friedman #1	9.64	6.94	28.05%
Friedman #2	3.017648×10^4	2.241992×10^4	25.7%
Friedman #3	0.03	0.02	32.51%

Original results from **(Breiman, 1996)**:

Dataset name	$\bar{e}_S$	$\bar{e}_B$	Decrease
Boston Housing	20.0	11.6	42%
Ozone	23.9	18.8	21%
Friedman #1	11.4	6.1	46%
Friedman #2	31,100	22,100	29%
Friedman #3	0.0403	0.0242	40%

3 Conclusion

The goal of bagging is to reduce the variance of the base model, which is often associated with high receptiveness to the training data. By creating different instances of the base model and combining their predictions, bagging improves overall predictive performance.

It could be applied to different types of models, in particular, we analyze how it works with Decision Tree Based Methods; machine learning algorithms used for classification and regression tasks that organize the data into a tree-like structure, where each node represents a decision criterion, and each branch corresponds to a possible response or action.

The procedure proposed by Breiman firstly divides a real dataset in learning (or training) and a test set; grows, on the learning set, regression trees and using cross-validation selects the tree with the best average performance, cut tree to avoid overfitting by using the learning set and repeat this for a large number of division learning-test.

The application of this algorithm to 5 different datasets produces important decreases in the squared errors consistent with the results computed by Breiman himself. The discrepancy will be attributed to small changes in the datasets considered and the use of different machines (Braiman produced these works in 1996)