Parallel computing with R

Bédouret Baptiste, Cellitti Francesco, Negroni Edoardo

October 2023

1 Introduction

The goal of this project is to perform a reproducible simulation by using parallel computing. For this project we decided to work on the topic of **Non-parametric classification**. To apply the classification we chose two models to study. SVM and Random Forest. Thus, the objective is to try to detect which is the best non parametric classifier among those two in term of performance and computation time while applying a simulation.

2 The data

To be able to test our algorithms and to apply parallelization, we generated our own data.

2.1 Generation of the covariates

We first generated 100 independent variables denoted as X using the *mvrnorm* function. It is typically used for simulating data from a multivariate normal distribution. As we have 500 observations, the dimension of X is 500×100.

2.2 Generation of y1 and y2

Then, we generated the labels y_1 and y_2 depending on the predictor variables X.

- For y_1, we used a linear approach which corresponds to the simple case. We created a linear regression model and calculated the probabilities using the logistic function. The logistic function transforms the input variables into a probability value between 0 and 1, which represents the likelihood of the dependent variable being 1 or 0. Finally, we got our binary response y_1 based on those probabilities.
- For y_2, we used a different approch more complicated. To be able to class the label either 0 or 1, we applied a dimension reduction PCA and then

created a circular function based on the sum of the squares of the first principal component and the second principal component.

$$PC_1^2 + PC_2^2 < 1$$

A dataset was built merging the vectors y1, y2 and the matrix X. Please note that these graphs are for illustrative purposes only, as the linear relationship is defined in p-dimensional space and therefore cannot be visualized by plotting only two covariates.

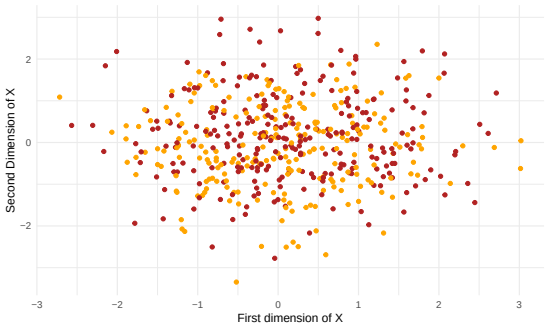


Figure 1: Data visualization of first two dimension of X and y_1

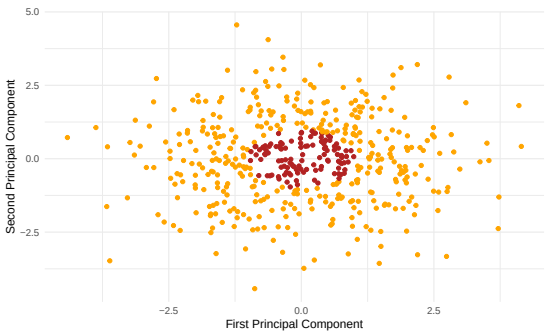


Figure 2: Data visualization of first two PC of X and y_2

3 Non parametric classifier

Nonparametric classifiers are machine learning models that do not make specific assumptions about the functional form of the data distribution or the relationship between variables. Unlike parametric classifiers, which have a fixed number

of parameters and assume a particular functional form, nonparametric classifiers are more flexible and can adapt to a wider range of data.

3.1 Support vector machine

Now we have to classify with respect to our binary lebel, with respect to our X. The first technique that we develop is based on the Support Vector Machine. The SVM method is based on the enlarging the feature space using kernels in order to solve the problem of non-linear boundary in the classes. From the classical support vector classifier, combining it with kernels we will obtain support vector machines on the form:

$$f(x) = \beta_0 + \sum_{i \in S} \alpha_i K(x, x_i)$$

Generating in two different ways the labels for our y as it's said before, let's check also the types of possible Kernels, from three different kernels:

- Linear:
$$K(x_i, x_{i'}) = x_i^T x_{i'}$$
- Polynomial:
$$K(x_i, x_{i'}) = (x_i^T x_{i'} + c)^d$$
- Radial:
$$K(x_i, x_{i'}) = exp\left\{-\gamma \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2\right\}$$

From that, if a test observation is far from a training x_i in terms of Euclidean distance, so the summation on the exponent will be large and such as a consequence, the kernel is small. Therefore, observation far from x will not be considered for our purpose in order to estimate x. The use of kernels allows us to study the behavior only of a certain amount of distinct pairs with respect to the entire enlarged feature space. The function used in order to implement the Support Vector Machine is *svm* from the package e1071:

```
svm_first_trial = svm(y ~., data = datasettwo,
                  scale = F, type   = "C-classification",
                  kernel = "polynomial",
                  cost = 100000, cross = 5, gamma = 0.0001,
                  degree = 3)
```

In this function we can see the choice for parameter γ of the radial Kernel function and the **cost** of the following. The cost is defined as a threshold C such that:

$$\sum_{i=1}^{n} \epsilon_i \leq C \tag{1}$$

It means that the greater is C, the more tolerant is the violations to the the margin and is, such as gamma and the degree of the polynomial, a tuning parameter to select due to a cross-validation, as follows:

```
tuning = tune(svm, y ~.,
          type = "C",
          scale = F,
          data   = datasettwo,
          kernel = c("radial","linear","polynomial"),
          ranges = list(cost = c(0.001, 0.01, 0.1, 1, 10, 100),
                    gamma = c(0.0001, 0.001, 0.01, 0.1)),
          tunecontrol = tune.control(sampling = "fix"))
tuning$best.parameters
```

In order to select the best hyper parameters for the model we decide to apply a cross-validation, so computing (in parallel) the tuning for these different type of parameters. The results show us that, for y labels generated linearly, the best kernel is 11 times out of 20 linear and otherwise polynomial for the remaining 9. For the non linear y the best kernel is the polynomial (12 out of 20) with 8 tuning of radial.

3.2 Random forest

The other method exploited in this analysis is the well known Random forest. It is an ensemble learning method, which means it combines the predictions of multiple individual models to make more accurate and robust predictions. Random Forest is built upon decision trees.

A decision tree is a flowchart-like structure where an internal node represents a feature or attribute, the branches represent a decision rule, and each leaf node represents an outcome or class label.

To overcome the fact that decision trees suffer from high variance, Bootstrap aggregation (or Bagging) is exploited: in this approach we generate B different bootstrapped training data sets. We then train our method on the b-th bootstrapped training set in order to get $\hat{f}^{*b}(x)$, and finally average all the predictions, to obtain:

$$\hat{f}_{BAG} = \frac{1}{B}\sum_{b=1}^{B} f^{*b}(x) \tag{2}$$

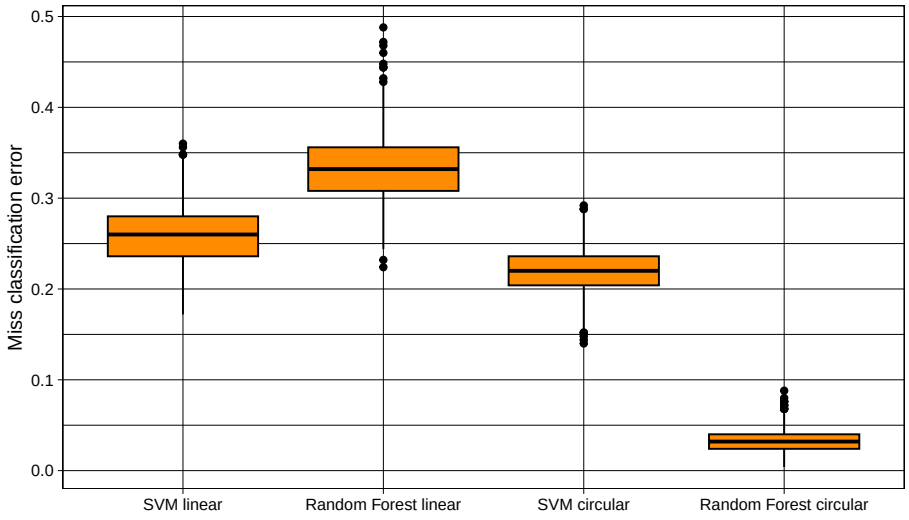
These trees are grown deep, and are not pruned. Hence each individual tree has high variance, but low bias. Averaging these B trees reduces the variance. Random forests provide an improvement over bagged trees by way of a small tweak that decorrelates the trees. We build a number of decision trees on bootstrapped training samples. But when building these decision trees, each time a split in a tree is considered, a random sample of m predictors is chosen

as split candidates from the full set of p predictors. The split is allowed to use only one of those m predictors. A fresh sample of m predictors is taken at each split, and typically we choose $m \approx \sqrt{p}$, that is, the number of predictors considered at each split is approximately equal to the square root of the total number of predictors. We can think of this process as decorrelating the trees, thereby making the average of the resulting trees less variable and hence more reliable.
When you want to make a prediction using a Random Forest, each individual tree in the ensemble makes its prediction. In classification problems, it's a majority vote among the trees, and in regression problems, it's an average of the tree predictions. Finally, calculate the average of all the predictions as shown previously.
The function used in order to implement the Random Forest is *randomForest* from the package randomForest:

```
randomForest(formula, data=NULL, ..., subset, importance = T)
```

4 Parallelization and results

After describing all the algorithm used in the simulation, we're going to deal with the parallelization process. We performed a reasonably large number of replications in our simulation study (e.g 1000 replications). The library *doParallel* was applied and we used the loop *for each %dopar%* to process the 1000 replications. The loop contains the generation of the data which is used in SVM function and randomForest function. It is possible to note that the code care about the reproducibility by using a different seed for each replication.



From the misclassification error distributions computed 1000 times for all methods and data, it's evident that for our linear y data, SVM outperforms Random Forest significantly. Meanwhile, the most effective estimator for nonlinear labels (circular structure) is a Random Forest. As observed in the boxplots, the entire distribution of Missclassification Error (MCE) is tightly concentrated within a small interval (first quantile = 0.0240, second quantile = 0.040).

In the graph, it is evident that there are numerous outliers for the misclassification error in different replicates. This phenomenon could lead to more pronounced variations in misclassification values, even when observations come from the same distribution. Therefore, it can be concluded that, for the type of data generated, in general, Support Vector Machines (SVM) appear to be more robust than Random Forests (RF).

Conversely, when dealing with nonlinearly computed y data, the linear SVM does not exhibit superior results. In fact, the average error is notably higher, exceeding 0.0 compared to Random Forest. In the linear domain, errors are generally more pronounced for both non-parametric classifiers. The critical distinction lies in the linear SVM's ability to predict outcomes more accurately than Random Forest in this scenario.

	SVM_log	SVM_cir	RF_log	RF_cir
MC mean error	0.26	0.22	0.33	0.03
MC sd error	0.032	0.024	0.036	0.012
Time in sec (for each iteration)	17,68	2,340	22,65	10,36

Table 1: Table of the miscclassification errors

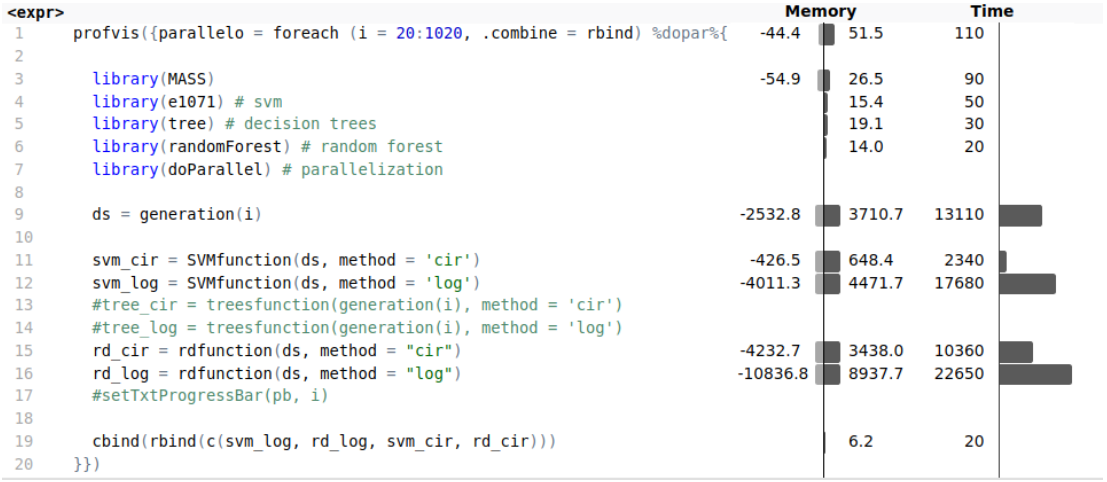
5 Profiling

It is good to point out that the function for tuning parameters in the SVM model is usually applied for each dataset (so for each replication). We noticed that, when we applied the function created to estimate the SVM model, much of the time spent was due to the estimation of hyperparameters. For each dataset we created at each iteration, the X which are generated from the same distribution (Multivariate normal with mean=0 and var=Identity matrix), although with different seeds (so the data are different); the same reasoning applies to all "linear" Y_1 and "circular" Y_2. As we expected in our code, for most datasets (distinguished between "log" and "cir") we got the same hyperparameters. So we thought it was appropriate to use them as "defaults" within the SVM function. Thus, in the code, we separated the tuning part and we created a smaller test run (20 iterations, i.e., 20 different datasets tested), on which we applied the hyperparameter tuning function in parallel. By using profvis, the tuning of SVM without parallelization was 6 min 50 sec. With parallelization it is much

faster, in fact it takes 1 min 17 sec.

In summary, the utility of the profvis function proves valuable in assessing the time required for parallelization.

The execution time for computing the four algorithms 1000 times, each time generating different data based on a seed i, totals 10.56 minutes. It's noteworthy that the Random Forest function stands out as the most time-consuming, averaging 10 seconds for each core. This emphasizes the computational cost associated with its implementation. Furthermore, the observed trend suggests that linear classification poses greater difficulty and exhibits higher computational requirements compared to linear values. This insight underscores the need for careful consideration and optimization when dealing with linear classification scenarios.



Positive values represent an increase in memory usage. This could be due to the allocation of memory for variables, data structures, or other resources used by your R code. Negative values represent a decrease in memory usage. This occurs when memory that was previously allocated is released or freed. For example, when objects are overwrited, or variables go out of scope, it can result in a negative change in memory.

6 Conclusion

In conclusion, our analysis of misclassification error distributions, computed 1000 times across various methods and datasets, reveals noteworthy insights. For linear y data, SVM emerges as the superior choice, outperforming Random Forest. Conversely, when dealing with nonlinear labels, a Random Forest employing a circular decision boundary proves to be the most effective estimator.

The boxplots illustrate that, in the linear domain, the misclassification error is tightly concentrated within a narrow interval, emphasizing the robustness of

our chosen methodologies. However, when y is computed nonlinearly, the linear SVM demonstrates a performance gap, exhibiting an average error exceeding 0.18 compared to Random Forest.

Overall, these findings highligt the behavior of classifiers in different contexts, underscoring the importance of selecting appropriate methodologies based on the nature of the data. While SVM excels in these linear scenarios, Random Forest emerges as a reliable choice for capturing these nonlinear relationships. As a final consideration, it was noticeable during this analysis that having to run a very large number of simulations, resolving bottlenecks and using process parallelization greatly improved the code and its speed.