

# Introduction to the R-package: **ranger**

A Fast Implementation of Random Forests



## **Group B**

Data and Text Mining  
QFin SS22

Date: 2022-04-21





- ranger = "**random forest generator**"
- fast implementation of random forests (RF's) for high dimensional data, optimised by *extensive runtime and memory profiling*
- core is implemented in C++ and R package Rcpp was employed to make the new implementation available as R package
- most of the features of the randomForest package are available and new ones were added
  - in addition to classification and regression forests, survival forests can be grown
  - class probabilities for multi-class classification problems can be estimated

# Why need ranger?

Package / implementation	Limitation(s)
original implementation (Fortran 77)	has to be recompiled in case of data/parameter value changes
randomForest (R)	not optimised for the use with high-dimensional data
Willows	not optimised for number of features
party (R)	shares weaknesses of randomForest and Willows
bigrf (R)	only classification forests can be grown
RandomForests	licensing costs and closed source code
Random Jungle	only available as C++ application
Rborist (R) and randomForestSRC (R)	relatively large runtime when growing RF's on high-dimensional data

## Example

# Example - Set-up

Loading in ranger and german:

```
library(ranger)
load("~/WU/QFin/S3Q2/DataTextMining/R_Package_Project/Data/german.rda")
```

Constructing train and test sets (70/30 split):

```
set.seed(123)
train.index = sample(seq_len(nrow(german)),
                     size = floor(0.7 * nrow(german)))
train.german = german[train.index, ]
test.german = german[-train.index, ]
```

## Example - Interfaces

First, we train a classification random forest, with Class as the dependent variable and all variables included in the set of potential predictors.

We can call ranger, specifying formula and data:

```
rf = ranger(formula = Class ~ ., data = train.german)
```

Alternatively, we can specify dependent.variable.name and data:

```
rf = ranger(dependent.variable.name = "Class", data = train.german)
```

Else, we can supply predictor data (x) and response vector (y):

```
rf = ranger(x = train.german[, -21], y = train.german[, 21])
```

## Example - Hyperparameter values

Note, that we can set our own hyperparameter values to fit a RF by specifying the corresponding argument values in ranger:

```
rf = ranger(formula = Class ~ ., data = train.german,
             num.trees = 1000,
             mtry = 5,
             min.node.size = 3,
             max.depth = 5)
```

As mentioned in one of the lectures, for bagging we set mtry equal to the total number of predictors.

Let's stick to the original specification with default values for simplicity:

```
rf = ranger(formula = Class ~ ., data = train.german)
```



# Example - ranger output

Applying the print method (for ranger objects) to rf yields a clean, informative summary:

```
rf

## Ranger result
##
## Call:
##  ranger(formula = Class ~ ., data = train.german)
##
## Type:                                Classification
## Number of trees:                     500
## Sample size:                         700
## Number of independent variables:     20
## Mtry:                                4
## Target node size:                     1
## Variable importance mode:             none
## Splitrule:                           gini
## OOB prediction error:                 25.43 %
```

## Example - ranger output

A `ranger.forest` object is written by default; to save memory one can set `write.forest = FALSE` if no predictions are intended. Applying the print method (for `ranger.forest` objects) yields:

```
rf$forest

## Ranger forest object
##
## Type:                      Classification
## Number of trees:          500
```

Conveniently, we can access the train set confusion matrix, stored in the output:

```
rf$confusion.matrix

##      predicted
## true   good bad
## good  449  47
## bad   131  73
```

## Example - Test set predictions

Objects of class ranger have their corresponding predict method. We call predict and provide our ranger object and data:

```
test.ranger.prediction = predict(rf, data=test.german)
```

To obtain a matrix with terminal nodeIDs for all observations in dataset and trees, include type = "terminalNodes".

Note, that we obtain an object from a different class:

```
class(test.ranger.prediction)
```

```
## [1] "ranger.prediction"
```

## Example - Test set predictions ctd.

Applying the corresponding print method to `test.ranger.prediction` yields:

```
test.ranger.prediction
## Ranger prediction
##
## Type:                      Classification
## Sample size:               300
## Number of independent variables: 20
```

Now, to extract the predictions we can call `predictions` and provide this `ranger.predict` object as the only input:

```
test.pred = predictions(test.ranger.prediction)
head(test.pred)

## [1] good good bad  good good bad
## Levels: good bad
```

# Example - Train set predictions

If we wish to make predictions on the train set, we can conveniently access these as follows:

```
train.pred = predictions(rf)
```

# Example - Variable importance

By default, variable importance is not computed.

```
importance(rf)
```

```
## Error in importance.ranger(rf): No variable importance found.  
Please use 'importance' option when growing the forest.
```

For classification, there are 3 choices for the value of importance argument in ranger - "impurity", "impurity\_corrected" and "permutation".

# Example - "impurity" importance

For the simple Gini impurity measure, we set importance = "impurity":

```
rf = ranger(Class ~ ., data = german, importance = "impurity")
head(importance(rf))
```

```
## Status_of_checking_account      Duration
##           46.89292                41.14497
##           History                Purpose
##           25.66421                25.55567
##           Amount                 Savings
##           55.28643                20.38094
```

However, there is no method of getting p-values for this method:

```
importance_pvalues(rf)
```

```
## Error in importance_pvalues(rf): Impurity variable importance found. Please
use (hold-out) permutation importance or corrected impurity importance to use this
method.
```

# Example - "impurity\_corrected" importance and "janitza" p-values

This method takes care of the bias towards the variables, which have large possible numbers of splits associated with them.

```
rf = ranger(Class ~ ., data = german, importance = "impurity_corrected")
```

Printing the variable importance values and p-values, based on "janitza" method:

```
head(importance_pvalues(rf, method = "janitza"))
```

```
## Warning in importance_pvalues(rf, method = "janitza"): Only few negative  
importance values found, inaccurate p-values. Consider the 'altmann' approach.
```

##	importance	pvalue
## Status_of_checking_account	24.202150	0
## Duration	7.455544	0
## History	8.570162	0
## Purpose	1.804617	0
## Amount	7.169496	0
## Savings	4.842300	0



## Example - "janitza" limitations

Why do we get a warning?

- With an unbiased variable importance measure, the importance values of non-associated variables vary randomly about zero.
- Thus, the method assumes that all variables with negative importance values belong to the set of variables not associated with the response.
- They are used to construct a distribution of importance under the null of no association with the response.
- In our case, there are too few of them and the constructed distribution is probably inaccurate.
- The "janitza" method is really meant for high-dimensional data, where there would be more of such variables and hence, higher accuracy could be achieved.

# Example - "permutation" importance and "altmann" p-values

The warning suggests to try the "altmann" method instead.  
In order to do so, the variable importance needs to be measured via the permutation approach.

```
rf = ranger(Class ~ ., data = german, importance = "permutation")
head(importance_pvalues(rf, formula = Class ~ ., data = german, method = "altmann"))
```

	importance	pvalue
## Status_of_checking_account	0.037091650	0.00990099
## Duration	0.018105326	0.00990099
## History	0.013182394	0.00990099
## Purpose	0.004803444	0.01980198
## Amount	0.012086962	0.00990099
## Savings	0.007027413	0.00990099

## Example - Regularisation

Regularisation may be applied as a feature selection method. The smaller the `regularization.factor` value, the harsher the penalty; additionally, we can use `depth` to intensify the penalty with increasing tree depth (by setting `regularization.usedepth = TRUE`).

```
rf = ranger(Class ~., data = german,
             regularization.factor = 0.5, regularization.usedepth = T,
             importance = "impurity")
```

```
## Warning in ranger(Class ~ ., data = german, regularization.factor = 0.5, :
Paralellization deactivated (regularization used).
```

# Example - `respect.unordered.factors`

By default, `respect.unordered.factors = "ignore"` (FALSE) and all factors are regarded ordered. These variables are essentially treated as numeric and `ranger` appears to run faster. However, not all categorical variables (if any) are known to have ordered relations with the outcome.

```

formula = "Duration ~ History + Status_of_checking_account + Purpose + Savings"

set.seed(123)
rf1 = ranger(formula = formula, data = train.german,
              respect.unordered.factors = TRUE)
set.seed(123)
rf2 = ranger(formula = formula, data = train.german)

rf1$prediction.error; rf2$prediction.error

## [1] 152.7619
## [1] 153.0551

rf1$r.squared; rf2$r.squared

## [1] 0.01127661
## [1] 0.009378965
  
```

## Example - replace and sample.fraction

We can specify both arguments to control the size of bootstrap samples and whether we are sampling with or without replacement. By default, `replace = TRUE` and `sample.fraction = 1`, i.e. we are sampling from the whole data set with replacement.

Now, set `replace = FALSE` and `sample.fraction = 1`... What happens?

```
rf = ranger(formula = Class ~ ., data = train.german,
             replace = F, sample.fraction = 1)
rf$prediction.error

## [1] NaN
```

The OOB prediction error is NaN, as each bootstrap sample was identical to the full data set.

## Other ranger arguments

- `probability`: grow probability forests as in Malley et al. (2012) [2]
- `case.weights`: weights for sampling of training observations
- `class.weights`: weights for the outcome classes in the splitting rule
- `always.split.variables`: variable names to be always selected in addition to the `mtry` variables
- `splitrule`: splitting rule to be used
- `split.select.weights`: probabilities to select variables for splitting
- `num.threads`: number of threads to be used
- `save.memory`: slower, but uses less memory



## **Run case-specific random forest** (algorithm by Xu et al. (2014) [4]):

1. Grow a random forest on the training data.
2. For each observation of interest (test data), the weights of all training observations are computed by counting the number of trees in which both observations are in the same terminal node.
3. For each test observation, grow a weighted random forest on the training data, using the weights obtained in Step 2. Predict the outcome of the test observation as usual.

```
csrf(formula, train.german, test.german,  
      params1 = list(),  
      params2 = list(), verbose = FALSE)
```



# holdoutRF() and parse.formula()

The `holdoutRF()` function grows two random forests on two CV folds. Instead of out-of-bag data, the other fold is used to compute permutation importance ([1]).

- **Inputs:** All arguments are passed to `ranger()` (except `importance`, `case.weights`, `replace` and `holdout`).
- **Outputs:** Hold-out random forests with variable importance.

The `parse.formula()` function parses formula and returns dataset containing selected columns.

- **Inputs:** `formula` (the model to fit), `data` (data frame with training data), `env` (for LHS evaluation).
- **Outputs:** Dataset including selected columns and interactions.

# treeInfo()

The `treeInfo()` function extracts the tree information of a `ranger` object.

- **Inputs:** `object` (a `ranger` object), `tree` (number of the tree of interest).
- **Outputs:** `data.frame` with each row being a node and the following columns:
  - `nodeID` (0-indexed)
  - `leftChild`: ID of the left child (0-indexed)
  - `rightChild`: ID of the right child (0-indexed)
  - `splitvarID`: ID of the splitting variable (0-indexed), the variable order changes if the formula interface is used.
  - `splitvarName`: Name of the splitting variable
  - `terminal`: logical, TRUE for terminal nodes,
  - `prediction`: the predicted class (factor) for classification and the predicted numerical value for regression



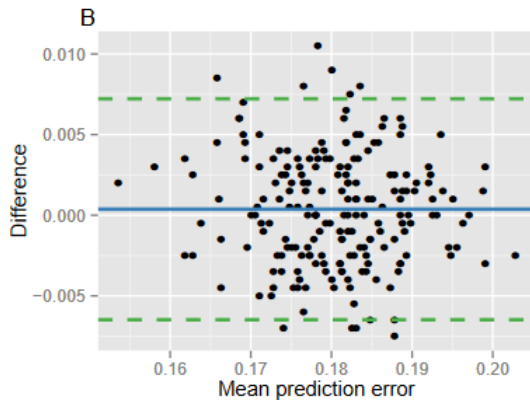
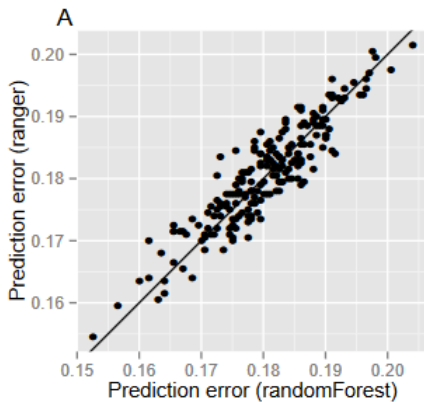
The authors of the package compared the results to those obtained by **randomForest**.

- binary outcome was simulated with 2000 samples and 50 features (5 with non-zero effect)
- 200 data sets were generated
- 5000 trees were fitted on each data set with both packages
- out-of-bag prediction errors for each data set were compared  
→ *No systematic difference could be observed.*

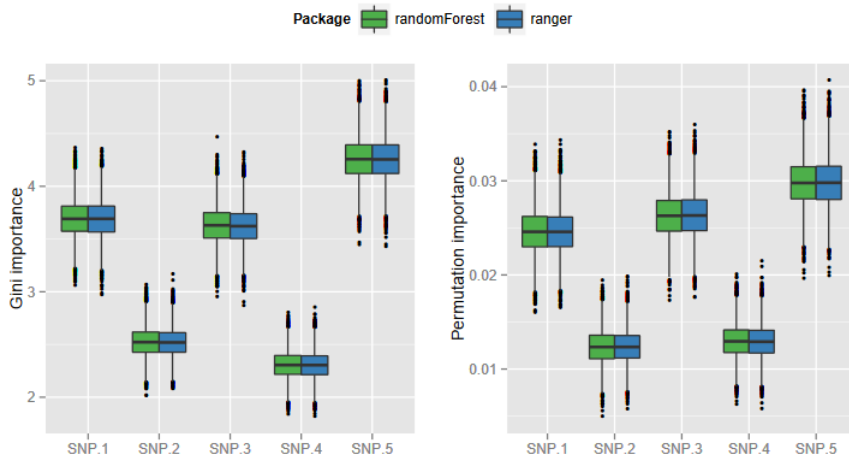
A similar experiment was conducted to measure differences in variable importance.

→ *Variable importance results are very similar for both packages.*

# Validation II



# Validation III





- Simulation study to compare runtime and memory usage
- 5 different implementations of random forests
- all implementations were run with 12 CPU cores (if possible)

Implementations:

- R package **randomForest**
- R package **randomForestSRC**
- R package **Rborist**
- C++ application **Random Jungle**
- R package **ranger**



# Runtime and memory usage II

Runtime and memory usage for **high**-dimensional data:  
10K observations, 150K features, 1K trees, mtry = 5K, 15K, 135K

Package	Runtime in hours			Memory usage in GB
	mtry = 5000	mtry = 15,000	mtry = 135,000	
randomForest	101.24	116.15	248.60	39.05
randomForest (MC)	32.10	53.84	110.85	105.77
bigrf	NA	NA	NA	NA
randomForestSRC	1.27	3.16	14.55	46.82
Random Jungle	1.51	3.60	12.83	0.40
Rborist	NA	NA	NA	>128
ranger	0.56	1.05	4.58	11.26
ranger (save.memory)	0.93	2.39	11.15	0.24
ranger (GWAS mode)	0.23	0.51	2.32	0.23

# Runtime and memory usage III

Runtime and memory usage for **low**-dimensional data:  
100K observations, 100 features, 1K trees, binary and continuous  
features

Package	Runtime in minutes		Memory usage in GB
	dichotomous features	continuous features	
randomForest	25.88	34.78	7.76
randomForest (MC)	2.98	3.75	9.17
bigrf (memory)	5.22	5.72	11.86
bigrf (disk)	24.46	26.39	11.33
randomForestSRC	8.94	9.45	8.85
Random Jungle	0.87	1367.61	1.01
Rborist	1.40	2.13	0.84
ranger	0.69	5.49	3.11



# Runtime and memory usage V

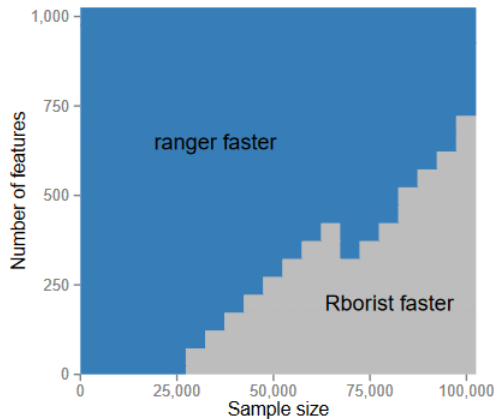





Figure: [3]





-  Silke Janitza, Ender Celik, and Anne-Laure Boulesteix. “A computationally fast variable importance test for random forests for high-dimensional data”. In: *Advances in Data Analysis and Classification* 12.4 (2018), pp. 885–915.
-  James D Malley et al. “Probability machines”. In: *Methods of information in medicine* 51.01 (2012), pp. 74–81.
-  Marvin N Wright and Andreas Ziegler. “ranger: A fast implementation of random forests for high dimensional data in C++ and R”. In: *arXiv preprint arXiv:1508.04409* (2015).



Random Forests". In: *Journal of Computational and Graphical Statistics* 25.1 (2016), pp. 49–65. DOI: