WorkedExample_rodri

April 19, 2021

1 Brief user guide for the RandomForestDist package

This companion notebook briefly explains how to use the main functionalities provided by the `R` package RandomForestDist and presents the code needed to reproduce part of the experiments presented in the article *Adapting Random Forests to Reliably Predict the Probability Distribution of Local Rainfall*, submitted to *Water Resources Research* by *Legasa et al.* in 2021. In that work, random forests (RFs) are applied to the problem of statistical downscaling of rainfall. In particular, the authors analyze the suitability of different split functions which allow to work with non-normally distributed variables and propose a novel a posteriori (AP) approach which permits to accurately estimate the shape and rate parameters of the underlying rainfall distribution, which in turn allows for generating reliable stochastic rainfall series.

RandomForestDist requires a modified version of rpart which can be found here. In order to run the examples provided below, these two packages need to be first installed. This can be easily done from GitHub using the `devtools` package:

[ ]:
```
devtools::install_github("MNLR/rpart")
install.packages(c("progressr", "qmap", "fitdistrplus"))
devtools::install_github("MNLR/RandomForestDist")
```

We start by loading `RandomForestDist` and `VALUE9`, a dataset included in the package which contains daily series of rainfall for 9 illustrative meteorological stations over Europe along with the 17 large-scale reanalysis predictors used for statistical downscaling (see the paper for details).

[1]:
```
library(RandomForestDist)
data(VALUE9)
```

```
Loading required package: rpart

Loading required package: progressr
```

`VALUE9` is a list of 9 elements (one per station), being the data already divided into a stratified 5-fold (F1 = 1979-1984, F2 = 1985-1990, F3 = 1991-1996, F4 = 1997-2002, F5 = 2003-2008). `$train.y`(training set) and `$test.y` (test set) correspond to the predictands; `$train.x` (training set) and `$test.x` (test set) are the predictors. For example, for the first station and fold:

[2]:
```
head(cbind(VALUE9[[1]]$f1_79_84$train.y, VALUE9[[1]]$f1_79_84$train.x))
```

A matrix: 6 Œ 69 of type dbl

5.4	100934.56	101076.31	101029.38	101127.6	0.9725891	-0.5552429	3
3.0	99861.81	100098.66	99890.75	100026.2	4.0751587	2.3417603	6
5.0	100774.52	101000.39	101343.70	101556.2	1.7720581	0.8526245	3
12.9	100790.47	100810.84	101059.16	101159.5	4.7946411	4.9987427	7
11.0	99843.41	99968.94	100306.59	100510.2	8.2443481	8.8000122	9
1.0	99936.30	99915.36	100108.30	100281.5	7.5472656	9.0443359	1

1.1 Training the models

To illustrate the training process, let's just focus on the sixth station and first fold:

[3]:
```
train.x <- VALUE9[[6]]$f1_79_84$train.x  # predictors for the train period
train.y <- VALUE9[[6]]$f1_79_84$train.y  # predictand (rainfall) for the train␣
 ↪period
```

The function `randomForestTrain()` builds the model for the predictors `x` and predictands `y`.

[4]:
```
rf <- randomForestTrain(x = train.x, y = train.y)  # building/training a␣
 ↪default RF
```

The user can specify the desired values for the parameters `mtry` (number of predictors to randomly use as candidate split variables), `ntree` (number of trees in the forest), `minbucket` (minimum number of elements each leaf is required to have), `minsplit` (minimum number of elements each node is required to have to attempt the splitting) and `maxdepth` (maximum depth allowed for each tree).

[5]:
```
rf <- randomForestTrain(x = train.x, y = train.y, ntree = 25, mtry = 22,
                        minbucket = 10, minsplit = 30, maxdepth = 10)  #␣
 ↪building/training a customized RF
```

By default, RFs consider the root mean squared error as split function. However, RandomForestDist allows for using different split functions (`split.function` argument), which have been defined in the modified version of rpart. Currently supported options are `"anova"`, `"poisson"`, `"class"`, `"exp"`, `"gammaLLMME"`, `"gammaLLmean"`, `"bernoulliGammaLLMME"`, `"gammaDeviation"`, `"gammaLLBC3"`, `"bernoulliLL"` and `"binaryCrossEntropyGammaDeviation"`.

In *Legasa et al. 2021*, we restrict ourselves to the 2-parameter gamma distribution family, with different estimators (see the paper for details).

[6]:
```
# NOTE: This chunk may take a while:
rf.dev <- randomForestTrain(x = train.x, y = train.y, ntree = 100, minbucket =␣
 ↪10, method = "gammaDeviation")
rf.LLMME <- randomForestTrain(x = train.x, y = train.y, ntree = 100, minbucket␣
 ↪= 10, method = "gammaLLMME")
rf.LLBC3 <- randomForestTrain(x = train.x, y = train.y, ntree = 100, minbucket␣
 ↪= 10, method = "gammaLLBC3")
```

1.2 Parallelization

The training process can be parallelized by means of the package `future.apply`. This package has to be installed independently, otherwise parallelization will not be an option. Use `install.packages(future.apply)` to install it, which will also install the dependency `future`.

The simplest way to use parallelization is by setting `parallel.plan = "auto"`, which will automatically use all the available cores to train the model in parallel.

[7]:
```
rf <- randomForestTrain(x = train.x, y = train.y, ntree = 100, minbucket = 10,
                        method = "gammaDeviation",
                        parallel.plan = "auto")  # training a RF with␣
 ↪parallelization
```

If not set to `"auto"`, `paralell.plan` will use the execution plan stored in the `strategy` argument of `future::plan()`. The interested reader is referred to the packages `future` and `future.apply` for details on the available options. As an example, the following block of code shows how parallelization in 4 cores can be achieved in combination with the argument `workers` and setting `parallel.plan` to `multisession`:

[8]:
```
rf <- randomForestTrain(x = train.x, y = train.y, ntree = 100, minbucket = 10,
                        method = "gammaDeviation",
                        parallel.plan = future::multisession, workers = 4)  #␣
 ↪training a customized RF with parallelization using 4 workers
```

Also, if missing or set to `NULL` (default), the function will use the current execution plan. This is useful to configure a customized plan outside the function, as we do in the next example. In particular, we set the execution to `multisession()` prior to launching the loop and train several models using this plan (do not forget to close the sessions by reverting to `sequential()`).

[9]:
```
future::plan(future::multisession, workers = 4) # sets the plan to multisession␣
 ↪and 4 workers

rfs <- lapply(VALUE9[[1]], function(fold){ # we iterate over the folds...
    return(
        randomForestTrain(x = fold$train.x, y = fold$train.y, ntree = 25,␣
 ↪minbucket = 10,
                          method = "gammaDeviation",
                          parallel.plan = NULL) #... and train the models
    )

})

future::plan(future::sequential) # reverts plan to sequential()
```

Note that using `parallel.plan = NA` will avoid parallelization altogether. This is the default if `future.apply` is not available.

1.3 Generating the predictions

Once the RF is trained, out-of-sample predictors (`$test.x`) can be used to obtain the corresponding downscaled predictions with the function `randomForestPredict()`. The default configuration of `randomForestPredict()` considers the mean as aggregation function (this is the standard procedure in RFs; see the paper for details):

[10]:
```
test.x <- VALUE9[[6]]$f1_79_84$test.x  # predictors for the test period
test.y <- VALUE9[[6]]$f1_79_84$test.y  # predictand (rainfall) for the train␣
 ↪period
```

```
# We use the RF with 100 trees and gammaDeviation split function that we have␣
↪already learnt to predict daily rainfall:
pr <- randomForestPredict(rf, newdata = test.x)  # predictions for the test␣
↪period, considering the mean as aggregation function
head(pr)
```

1 6.69288220375256 **2** 2.60350666864955 **3** 4.30276790536605 **4** 2.22515326732951 **5** 4.25758322163732 **6** 10.8015920530665

The parameter `bagging.function` can be used to apply other aggregation functions (e.g.the median). Also, if set to `NA`, `randomForestPredict()` will provide the predictions returned by all the individual trees.

[11]:
```
pr.median <- randomForestPredict(rf, newdata = test.x, bagging.function =␣
↪median)  # median as aggregation function
pr.alltrees <- randomForestPredict(rf, newdata = test.x, bagging.function = NA)
pr.alltrees[1:10,]  # 10 first predictions returned by the 100 trees
```

A matrix: 10 Œ 100 of type dbl

1	1.8250000	5.2857143	2.640000	5.9166667	4.6578947	7.18750
2	0.9636364	2.0363636	4.282353	0.7727273	1.5142857	4.80000
3	4.3529412	5.2857143	2.133333	5.9166667	4.8800000	1.96153
4	1.1357143	5.2857143	2.133333	1.2909091	0.9818182	1.61666
5	2.4230769	2.0700000	2.761538	1.9100000	3.2315789	1.60909
6	13.3000000	9.5800000	8.190909	9.7200000	6.3125000	13.5235
7	12.5000000	11.6894737	9.240000	14.2166667	2.1142857	13.5235
8	1.7764706	0.8666667	2.747059	2.4857143	1.5916667	4.68000
9	8.8666667	6.4153846	10.507143	12.2454545	9.6153846	4.09000
10	11.2888889	14.5333333	12.300000	14.0666667	13.6181818	16.2900

The parameter `method` controls the *a posteriori* approach (the reader is referred to the paper for details). The available options are described in the help page (`?randomForestPredict`), and we summarize them here. If `method` is not `NULL`, the argument `bagging.function` is ignored.

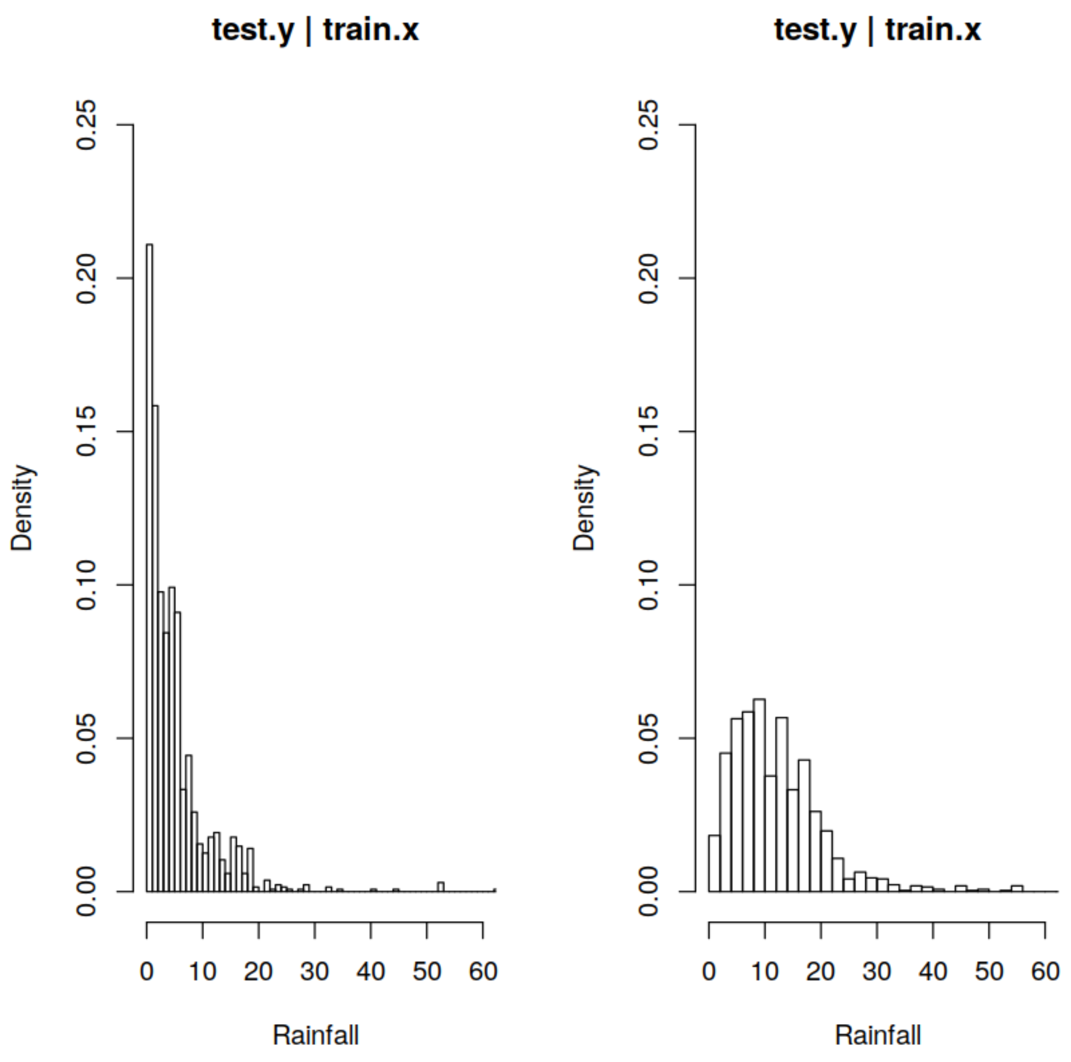
- `method = "leaves"` provides all the observations for each prediction.
- `method = "random.sample"` draws a random sample from the leaves for each prediction.
- `method = "aposteriori"` returns the parameter of the probability distribution of `train.y | train.x`, as estimated using Moments Matching Estimation (refer to `fitdistrplus::fitdist()`). In the case of the gamma distribution, its shape and rate parameters are returned. `method = "bc3"` uses BC3 estimators for the gamma distribution. Additionally, this parameter can be used to select the estimation method (see `fitdistrplus::fitdist()`), i.e., one of the following: `"mme"` (Moments Matching Estimation; default),`"mle"` (Maximum Likelihood Estimation), `"qme"` (Quantile Matching Estimation), `"mge"` (Maximum Goodness-of-fit Estimation), `"mse"` (Maximum Spacing Estimation). Note that some of these options require additional parameters (use `...`), and some of them may not converge.

We provide some examples below.

Setting the parameter `method = "leaves"` allows us to plot `train.y | train.x`, for instance:

[12]:
```
pr.leaves <- randomForestPredict(rf, newdata = test.x, method = "leaves") # we␣
↪predict the observations falling on leaves
```

```r
# and plot the histogram of their distribution:
par(mfrow = c(1,2))
hist(pr.leaves[[100]], breaks = 50, xlab = "Rainfall", freq = FALSE,
     main = "test.y | train.x", ylim = c(0, 0.25), xlim = c(0,60))
hist(pr.leaves[[300]], breaks = 50, xlab = "Rainfall", freq = FALSE,
     main = "test.y | train.x",ylim = c(0, 0.25), xlim = c(0,60))
```



As the model was trained using the *deviation* for the 2 parameter gamma distribution, using `method = "aposteriori"` provides the shape and rate parameters for each prediction:

[13]:
```r
pr.distr <- randomForestPredict(rf, newdata = test.x, method = "aposteriori")
pr.distr[1:10,] # We show the first 10 predicted outcomes
```

A matrix: 10 Œ 2 of type dbl

shape	rate
0.9677957	0.1419155
0.7862478	0.2982816
1.3759968	0.3128242
0.7537063	0.3403316
1.0151516	0.2371556
1.4834291	0.1391585
1.5673480	0.1938595
1.2676882	0.3518086
2.6937874	0.2916631
4.7145554	0.3804880

As stated above, the user can also specify the estimator used. For instance, rather than the Moments Matching Estimation (default) we could use the Maximum Likelihood Estimation by setting `method = "mle"`:

[14]:
```
pr.mle <- randomForestPredict(rf, newdata = test.x, method = "mle")
pr.distr[1:10,]   # 10 first predicted outcomes
```

A matrix: 10 Œ 2 of type dbl

shape	rate
0.9677957	0.1419155
0.7862478	0.2982816
1.3759968	0.3128242
0.7537063	0.3403316
1.0151516	0.2371556
1.4834291	0.1391585
1.5673480	0.1938595
1.2676882	0.3518086
2.6937874	0.2916631
4.7145554	0.3804880

Finally, the predicted distribution parameters can be used to simulate a series of stochastic rainfall values using the function `randomForestSimulate()`, which takes as inputs the object returned by `randomForestPredict()` and the number of simulated series, `n`. The parameter `distr` allows for manually selecting the distribution of interest (by default, it is automatically selected based on the training split function considered).

[15]:
```
simulation <- randomForestSimulate(pr.distr)
```

We simulate 1000 series:

[16]:
```
simulations <- randomForestSimulate(pr.distr, n = 1000)
head(simulations)
```

A matrix: 6 Œ 1000 of type dbl

3.3957831	2.6261201	4.2000766	23.5097911	0.2244423	2.4337384
0.9729068	0.3553539	7.1575646	0.7050184	3.0437166	0.4486807
13.5193071	1.1896116	1.8297097	4.7100415	1.8269712	1.5602529
1.0754117	0.4053252	0.1027564	6.8429978	0.8373085	0.1429419
5.5282513	5.4668850	6.2051241	0.5221806	13.2525883	0.1781004
17.9463134	0.6897609	3.2023356	4.5290871	6.5932239	21.9272342

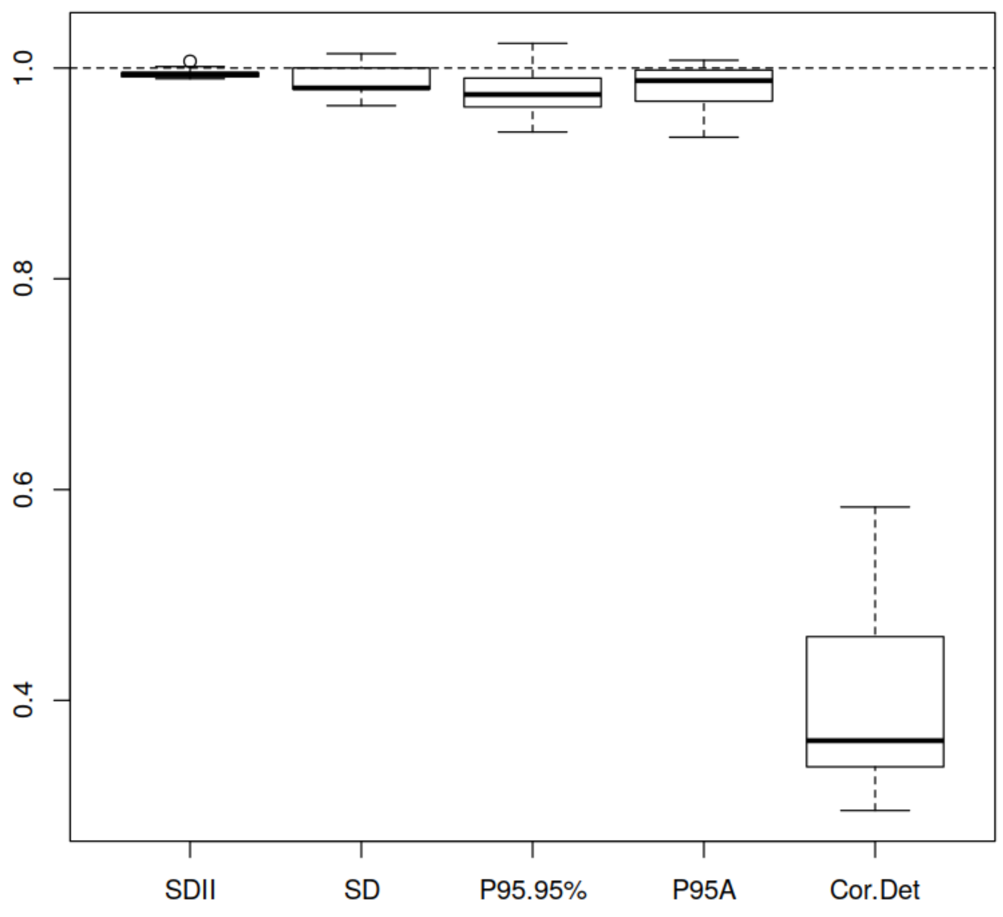
2 Reproducibility code for Figure 6 in *Legasa et al. 2021*

The code provided below can be used to obtain a similar to Figure 6 in *Legasa et al. 2021* (we reproduce here only the continuous part of rainfall) based on a RF with 25 trees and minimum leaf size 5 which considers the `gammaDeviation` as split function.

[17]:
```
future::plan(future::multisession, workers = 6)  # parallelizing using 6 cores

predictions <-
lapply(VALUE9, function(meteorological.station){
    predictions <- lapply(meteorological.station, function(fold){
        # training the model
        md <- randomForestTrain(x = fold$train.x, y = fold$train.y,
                                ntree = 25, minbucket = 5,
                                method = "gammaDeviation",
                                parallel.plan = NULL)

        # applying the trained model to $test.x and returning the observed␣
 ↪values $test.y:
        return(list(predicted = randomForestPredict(md, newdata = fold$test.x,␣
 ↪method = "mme"),
                     observed = fold$test.y))
    })

    # rearranging the series
    predicted <- do.call(rbind, lapply(predictions, function(pr) pr$predicted))
    attr(predicted, "class") <- "RandomForest2.prediction.simulable" # recovers␣
 ↪the class
    observed <- do.call(c, lapply(predictions, function(pr) pr$observed))

    return(list(predicted = predicted, observed = observed))
})

future::plan(future::sequential) # reverting plan to normal
```

We end up with the downscaled probability distribution (parameters *shape* and *rate*) for each station and day, which are then used to generate an ensemble of 250 stochastic rainfall values. With illustrative purposes, these values are subsequently validated in terms of the SDII, SD, P95 and P95A metrics (see the paper for details).

[18]:
```
validation <-
    lapply(predictions, function(station.predictions){  # loop covering each␣
 ↪meteorological station
        # generating 250 simulations from the predicted shape and rate␣
 ↪parameters
        observed <- station.predictions$observed
        simulations <- randomForestSimulate(prediction = station.
 ↪predictions$predicted,
                                            distr = "gamma", n = 250)
        # validating each simulation
        stochastic.validation <-
            apply(simulations, MARGIN = 2, FUN = function(simulation){# for␣
↪each simulation

                P95.simulated <- quantile(simulation, probs = 0.95)
                P95.observed <- quantile(observed, probs = 0.95)

                return(
                    c(SDII = mean(simulation)/mean(observed),
                      SD = sd(simulation)/sd(observed),
                      P95 = P95.simulated/P95.observed,
                      P95A = sum(simulation[simulation >= P95.simulated])/
↪sum(observed[observed >= P95.observed])
                     )
                )
            })

        # averaging the 250 values obtained for each validation metric
        stochastic.validation <- rowMeans(stochastic.validation)

        # correlation is computed for the deterministic series of predicted␣
↪rainfall; recall E[X] = alpha/beta
        alphas <- station.predictions$predicted[ , 1]
        betas <- station.predictions$predicted[ , 2]
        correlation <- cor(alphas/betas, observed, method = "spearman")

        return(c(stochastic.validation, Cor.Det = correlation))

    })

validation <- t(simplify2array(validation))

# plotting the results
boxplot(validation)
abline(h = 1, lty = 2)
```


1.0
0.8
0.6
0.4
SDII
SD
P95.95%
P95A
Cor.Det