WorkedExample

April 17, 2021

1 Brief user guide for the RandomForest2 package

This companion notebook briefly explains how to use the main functionalities provided by the `R` package **RandomForest2** and presents the code needed to reproduce one 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 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 variable,s 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.

RandomForest2 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/RandomForest2")
```

We start by loading `RandomForest2` and `VALUE9`, an 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(RandomForest2)
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). `$train.y` and `$test.y` correspond to the predictands; `$train.x` (training set) and `$test.x` (testing set) are the predictors. For example:

[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 select the sixth station and first fold:

[3]:
```
train.x <- VALUE9[[6]]$f1_79_84$train.x
train.y <- VALUE9[[6]]$f1_79_84$train.y
```

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

[4]:
```
rf <- randomForestTrain(x = train.x, y = train.y)
```

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)
```

By default, random forests consider the root mean squared error as split function. However, RandomForest2 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 this work we restrict ourselves to the 2-parameter gamma distribution family, with different estimators (see the paper for details).

[6]:
```
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`. The package has to be installed independently, otherwise parallelization will not be an option. Use `install.packages(future.apply)` to install it. This will also install the dependency `future`. The simplest way to use parallelization is setting `parallel.plan = "auto"`. This 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")
```

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

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)

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

})

future::plan(future::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 random forest is trained, the corresponding predictions can be obtained with the function `randomForestPredict()`. We use the out-of-sample predictors (`$test.x`) for the models above.

[10]:
```
test.x <- VALUE9[[6]]$f1_79_84$test.x
test.y <- VALUE9[[6]]$f1_79_84$test.y
```

The default configuration of `randomForestPredict()` considers the mean as aggregation function (this is the standard procedure in random forests; see the paper for details):

[11]:
```
pr <- randomForestPredict(rf, newdata = test.x)
head(pr)
```

1 9.10497585076127 **2** 3.08777004435932 **3** 5.17882580778721 **4** 2.66586621188115 **5** 3.9547599269327 **6** 8.1807959768411

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.

[12]:
```
pr.median <- randomForestPredict(rf, newdata = test.x, bagging.function =␣
↪median)
pr.alltrees <- randomForestPredict(rf, newdata = test.x, bagging.function = NA)
pr.alltrees[1:10,] # Show the 10 first predictions returned by the 100 trees
```

A matrix: 10 Œ 25 of type dbl

1	19.707692	4.0280000	7.080000	5.1758621	7.163158	8.7222222
2	2.870794	0.9833333	5.180000	0.8485714	1.685185	0.8636364
3	3.415873	6.5454545	8.586957	8.2928571	2.744000	8.7222222
4	1.666667	4.0280000	1.537500	1.0771429	1.074419	8.7222222
5	2.589809	2.7656716	5.180000	3.4556338	5.417647	10.3533333
6	8.097778	11.6461538	8.023810	4.7428571	8.012338	12.6585366
7	8.097778	7.0206897	12.450000	24.7100000	4.129474	7.1603053
8	5.293103	3.1166667	3.523246	6.9045455	1.832353	4.0812500
9	8.865517	14.1545455	6.813333	20.8800000	4.942105	7.8437500
10	8.865517	14.1241379	18.035714	9.5526316	4.942105	10.9934211

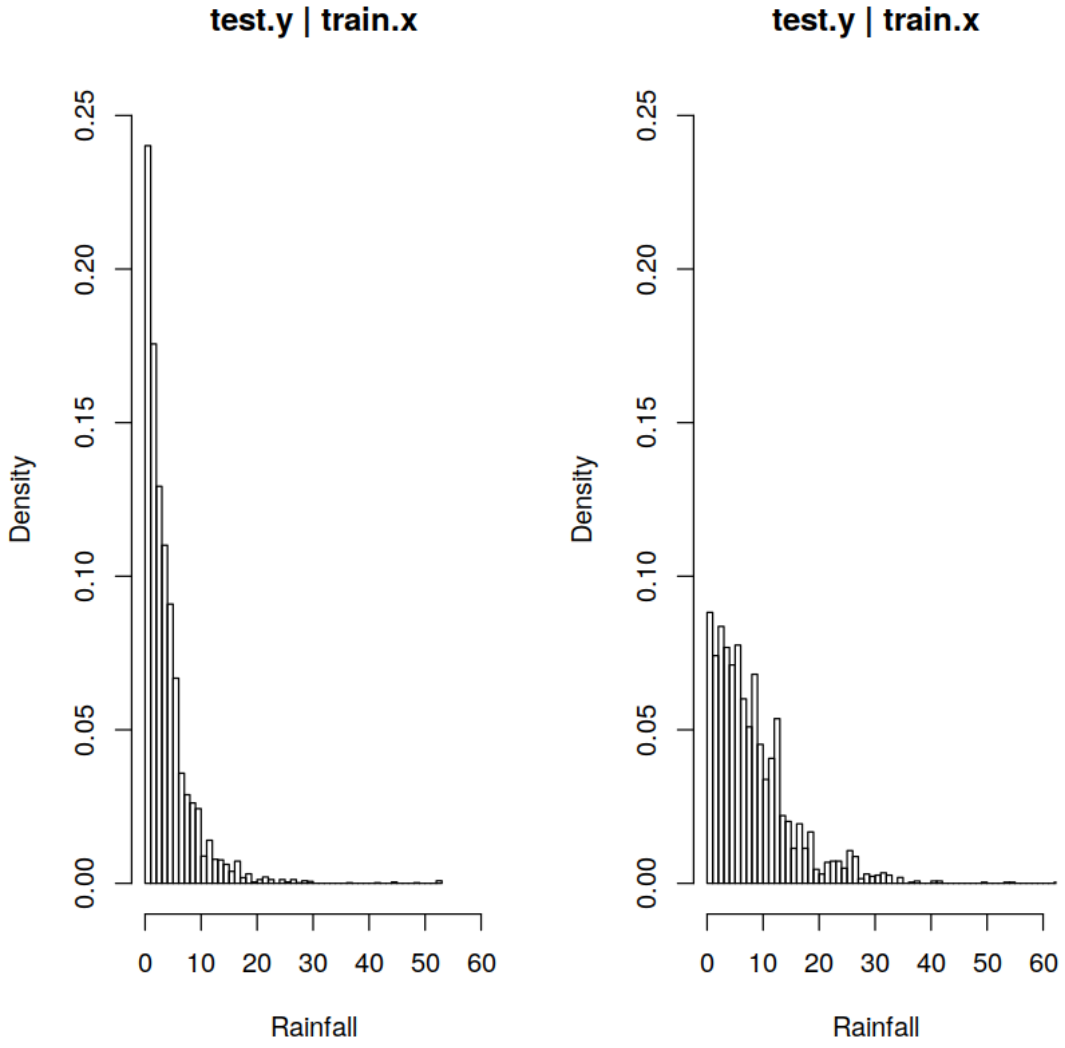
The parameter `method` controls the *a posteriori* approach (the reader is referred to the paper for details). The available options are described in the help page (`?randomForestPredict`), and we summarize them here. If `method` is not `NULL`, the argument `bagging.function` is ignored.

- `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, the shape and rate parameters of the distribution are returned. `method = "bc3"` uses BC3 estimators for the gamma distribution (see Legasa et al.). 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:

[13]:
```
pr.leaves <- randomForestPredict(rf, newdata = test.x, method = "leaves")

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:

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

	mean	sd
	8.567043	8.929944
	2.057696	2.355131
	4.390519	4.311260
	2.370429	2.563702
A matrix: 10 Œ 2 of type dbl	3.430964	3.611757
	8.167465	6.278448
	7.929120	5.945889
	3.136086	3.263602
	8.423594	5.928816
	10.706578	5.797082

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"`:

[15]:
```
pr.mle <- randomForestPredict(rf, newdata = test.x, method = "mle")
pr.distr[1:10,] # We show the first 10 predicted outcomes
```

	mean	sd
	8.567043	8.929944
	2.057696	2.355131
	4.390519	4.311260
	2.370429	2.563702
A matrix: 10 Œ 2 of type dbl	3.430964	3.611757
	8.167465	6.278448
	7.929120	5.945889
	3.136086	3.263602
	8.423594	5.928816
	10.706578	5.797082

Finally, the predicted distribution parameters can be used to simulate a series of stochastic rainfall values. This can be done using the function `randomForestSimulate()`. It 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).

[16]:
```
simulation <- randomForestSimulate(pr.distr)
```

We simulate 1000 series:

[17]:
```
simulations <- randomForestSimulate(pr.distr, n = 1000)
head(simulations)
```

	19.130835	21.746864	2.5620741	24.4728991	-8.599635	0.9378861	3.
	7.006280	6.521129	-5.8547134	1.6765677	3.276943	2.7772007	0.
A matrix: 6 Œ 1000 of type dbl	2.076019	10.101233	0.4358133	5.2178859	7.616047	0.8833983	2.
	4.343131	-1.113647	4.7924359	-0.5219222	2.309342	-1.7117722	3.
	5.317898	1.443337	1.0277138	8.4330255	2.390635	7.8563778	3.
	1.687543	2.783779	4.4966187	4.2025480	14.675236	0.4887514	10

2 Reproducibility code for the final model in *Legasa et al. 2020*

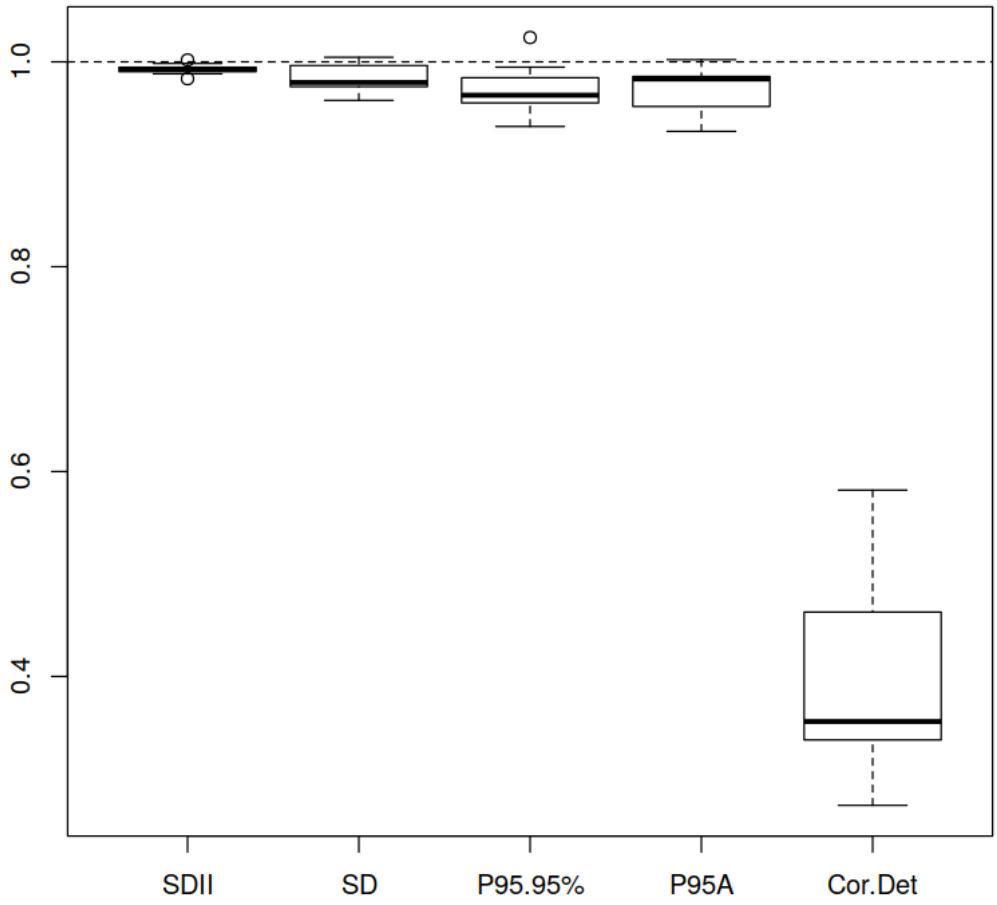
Finally, we use the code below to train the final model in *Legasa et al. 2020*, and obtain a Figure similar to Figure 6 in *Legasa et al. 2020* (we reproduce here only the continuous part). We use a random forest with 25 trees, minimum leaf size 5 which considers the *deviation* as split function.

[18]:
```
future::plan(future::multisession, workers = 6) ## Sets parallelization to 6␣
↪cores

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

        #Use the function to predict for $test.x, an also return the observed␣
↪values $test.y:
        return(list(predicted = randomForestPredict(md, newdata = fold$test.x,␣
↪method = "mme"),
                    observed = fold$test.y))
    })

    # rearange 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) ## Reverts plan to normal
```

We end up with the downscaled probability distribution for each station and day. We validate them for the selected measures as follows:

[19]:
```
validation <-
    lapply(predictions, function(station.predictions){ #for each meteorological␣
↪station
        # Generate 250 simulations

        observed <- station.predictions$observed
        simulations <- randomForestSimulate(prediction = station.
↪predictions$predicted, distr = "gamma", n = 250)
```

```r
        # validate each simulation
        stochastic.validation <-
            apply(simulations, MARGIN = 2, FUN = function(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])
                     )
                )
            })

        # For each series we obtained the measures. We average them:
        stochastic.validation <- rowMeans(stochastic.validation)

        # We compute the deterministic correlation. 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))

# We plot the results:
boxplot(validation)
abline(h = 1, lty = 2)
```


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