randomForestSRC: Speedup Random Forest Analyses Vignette

Hemant Ishwaran, Min Lu and Udaya B. Kogalur

Subsampling

Subsampling can be used to reduce the size of the in-sample data used to grow a tree and therefore can greatly reduce computational load. Subsampling is implemented using options `sampsize` and `samptype` in `rfsrc()`. See `rfsrc.fast()` in the next subsection for a fast forest implementation using subsampling.

Fast approximate random forests using subsampling

Our function `rfsrc.fast()` adopts fast approximate random forests using subsampling that applies to all families.

```
library("randomForestSRC")
## load the Iowa housing data
data(housing, package = "randomForestSRC")

## do quick and *dirty* imputation
housing <- impute(SalePrice ~ ., housing,
         ntree = 50, nimpute = 1, splitrule = "random")

## grow a fast forest
obj <- rfsrc.fast(SalePrice ~ ., housing)
print(obj)
```

```
>                          Sample size: 2930
>                     Number of trees: 500
>           Forest terminal node size: 5
>       Average no. of terminal nodes: 83.776
> No. of variables tried at each split: 27
>              Total no. of variables: 80
>       Resampling used to grow trees: swor
>     Resample size used to grow trees: 398
>                             Analysis: RF-R
>                               Family: regr
>                      Splitting rule: mse *random*
>       Number of random split points: 10
>                     (OOB) R squared: 0.8761255
>                   (OOB) Error rate: 790552487
```

Reducing number of variables for big data set

We could adjust the parameters in functions to encourage computational speed.

```
## ------------------------------------------------------------
## big data set, reduce number of variables using simple method
## ------------------------------------------------------------
library("randomForestSRC")
## use Iowa housing data set
data(housing, package = "randomForestSRC")

## original data contains lots of missing data, use fast imputation
## however see impute for other methods
housing2 <- impute(data = housing, fast = TRUE)

## run shallow trees to find variables that split any tree
xvar.used <- rfsrc(SalePrice ~., housing2, ntree = 250, nodedepth = 4,
                   var.used="all.trees", mtry = Inf, nsplit = 100)$var.used

## now fit forest using filtered variables
xvar.keep  <- names(xvar.used)[xvar.used >= 1]
o <- rfsrc(SalePrice~., housing2[, c("SalePrice", xvar.keep)])
print(o)

>                          Sample size: 2930
>                     Number of trees: 500
>           Forest terminal node size: 5
>       Average no. of terminal nodes: 400.552
> No. of variables tried at each split: 16
>              Total no. of variables: 46
>       Resampling used to grow trees: swor
>     Resample size used to grow trees: 1852
>                             Analysis: RF-R
>                               Family: regr
>                      Splitting rule: mse *random*
>       Number of random split points: 10
>                     (OOB) R squared: 0.9038358
>                   (OOB) Error rate: 613708802
```

Randomized splitting rules

Computational speed can be significantly improved with randomized splitting invoked by the option `nsplit`. When `nsplit` is set to a non-zero positive integer, a maximum of `nsplit` split points are chosen randomly for each of the candidate splitting variables when splitting a tree node, thus significantly reducing the cost from having to consider all possible split-values. To revert to traditional deterministic splitting (all split values considered) use `nsplit`=0.

Randomized splitting for trees has a long history. See Ishwaran (2015). Geurts, Ernst, & Wehenkel (2006) recently introduced extremely randomized trees using what they call the extra-trees algorithm. We can see that this algorithm essentially corresponds to randomized splitting with `nsplit`=1. In our experience however this is not always the optimal value for `nsplit` and may often be too low.

Finally, for completely randomized (pure random) splitting use `splitrule`="random." In pure splitting, nodes are split by randomly selecting a variable and randomly selecting its split point (Cutler & Zhao, 2001).

Unique time points for survival analysis

Setting `ntime` to a reasonably small value such as 100 or 250 (the default) constrains survival ensemble calculations to a restricted grid of time points and significantly improves computational times.

Large number of variables

The default setting `importance="none"` turns off variable importance (VIMP) calculations and considerably reduces computational times. Variable importance can always be recovered later using functions `vimp` or `predict`. Also consider using `max.subtree` which calculates minimal depth, a measure of the depth that a variable splits, and yields fast variable selection (Ishwaran, Kogalur, Gorodeski, Minn, & Lauer, 2010).

Increasing `nodesize`

Another way to speed up things considerably is increasing `nodesize`. When users have very big data sets, you can actually get better performance with larger `nodesize`. The function `tune.nodesize()` can be used to set `nodesize`.

C-index calculation for big survival data sets

The C-index is computationally expensive and requires approximately $O(n^2)$ calculations where n is the sample size. This makes it infeasible for large survival data sets. The solution is to turn of the C-calculation by turning off all performance requests when training the forest. This is accomplished using the option `perf.type='none'`. Thus a generic training call would look something like:

```
o <- rfsrc(Surv(my.time,my.cens)~.,my.data,perf.type="none")
```

It is also recommended to set `nodesize` to a reasonably large value. Also, assuming the user is interested in measuring performance of their model, consider using the Brier score which is very fast. This can be obtained using the function `get.brier.survival`:

```
bs <- get.brier.survival(o)
```

See the vignette for Random Survival Forests for more details about the Brier score.

Save memory

Use option `save.memory="TRUE"` for big data competing risk and survival models. By default the package stores terminal node quantities to be used in prediction for test data but this can be memory intensive for big data.

Block size

Make sure `block.size="NULL"` (or set to number of trees) so that the cumulative error is calculated only once.

Cite this vignette as

H. Ishwaran, M. Lu, and U. B. Kogalur. 2021. "randomForestSRC: speedup random forest analyses vignette." http://randomforestsrc.org/articles/speedup.html.

```
@misc{HemantSpeed,
    author = "Hemant Ishwaran and Min Lu and Udaya B. Kogalur",
    title = {{randomForestSRC}: speedup random forest analyses vignette},
    year = {2021},
    url = {http://randomforestsrc.org/articles/speedup.html}
}
```

References

Cutler, A., & Zhao, G. (2001). Pert-perfect random tree ensembles. *Computing Science and Statistics*, *33*, 490–497.

Geurts, P., Ernst, D., & Wehenkel, L. (2006). Extremely randomized trees. *Machine learning*, *63*(1), 3–42. Springer.

Ishwaran, H. (2015). The effect of splitting on random forests. *Machine learning*, *99*(1), 75–118. Springer.

Ishwaran, H., Kogalur, U. B., Gorodeski, E. Z., Minn, A. J., & Lauer, M. S. (2010). High-dimensional variable selection for survival data. *Journal of the American Statistical Association*, *105*(489), 205–217. Taylor & Francis.