Boruta for those in a hurry

Miron B. Kursa

November 19, 2024

1 Overview

Boruta [4] is a feature selection method; that is, it expects a standard information system you'd fed to a classifier, and judges which of the features are important and which are not. Let's try it with a sample dataset, say `iris`. To make things interesting, we will add some nonsense features to see if they get filtered out; to this end, we randomly mix the order of elements in each of the original features, wiping out its interaction with the decision, `iris$Species`.

```
> set.seed(17)
> data(iris)
> irisE<-cbind(
+  setNames(
+   data.frame(apply(iris[,-5],2,sample)),
+   sprintf("Nonsense%d",1:4)
+  ),
+  iris
+ )
```

Now, time for Boruta:

```
> library(Boruta)
> Boruta(Species~.,data=irisE)->BorutaOnIrisE
> BorutaOnIrisE
```

```
Boruta performed 17 iterations in 1.11302 secs.
 4 attributes confirmed important: Petal.Length, Petal.Width,
Sepal.Length, Sepal.Width;
 4 attributes confirmed unimportant: Nonsense1, Nonsense2, Nonsense3,
Nonsense4;
```

As one can see, the method had *rejected* nonsense features and *confirmed* (retained) the original ones, as it was to be expected. What is important is that Boruta does a sharp classification of features rather than ordering, which is in contrast to many other feature selection methods. The other substantial difference is that Boruta is an *all relevant* method, hence aims to find all features

connected with the decision — most other methods are of a *minimal optimal* class, consequently aims to provide a possibly compact set of features which carry enough information for a possibly optimal classification on the reduced set [5]. What does it mean in practice is that Boruta will include redundant features, that is ones which carry information already contained in other features.

As an example, let's add a feature which contains all the information in the decision in a most accessible form — namely, a copy of the decision, and push it into Boruta.

```
> irisR<-cbind(
+  irisE,
+  SpoilerFeature=iris$Species
+ )
> Boruta(Species~.,data=irisR)

Boruta performed 18 iterations in 0.3712935 secs.
 5 attributes confirmed important: Petal.Length, Petal.Width,
Sepal.Length, Sepal.Width, SpoilerFeature;
 4 attributes confirmed unimportant: Nonsense1, Nonsense2, Nonsense3,
Nonsense4;
```

We see that `SpoilerFeature` has not supplanted any of the original features, despite making them fully redundant. One may wonder, however, how come anyone would need something which is clearly redundant? There are basically three reasons behind this:

- One may perform feature selection for an insight in which aspects of the phenomenon in question are important and which are not. In such cases subtle effects possess substantial explanatory value, even if they are masked by stronger interactions.
- In some sets, especially of a $p \gg n$ class, nonsense features may have spurious correlations with the decision, arising purely by chance. Such interactions may rival or even be stronger than actual mechanisms of the underlying phenomenon, making them apparently redundant. All relevant approaches won't magically help distinguish both, but will better preserve true patterns.
- Minimal optimal methods are generally cherry-picking features usable for classification, regardless if this usability is significant or not, which is an easy way to overfitting. Boruta is much more robust in this manner.

2 Mechanism

Under the hood, Boruta uses feature importance scores which are provided by certain machine learning methods; in particular Random Forest [1], which happens to be used by default (using the `ranger` package [6] implementation).

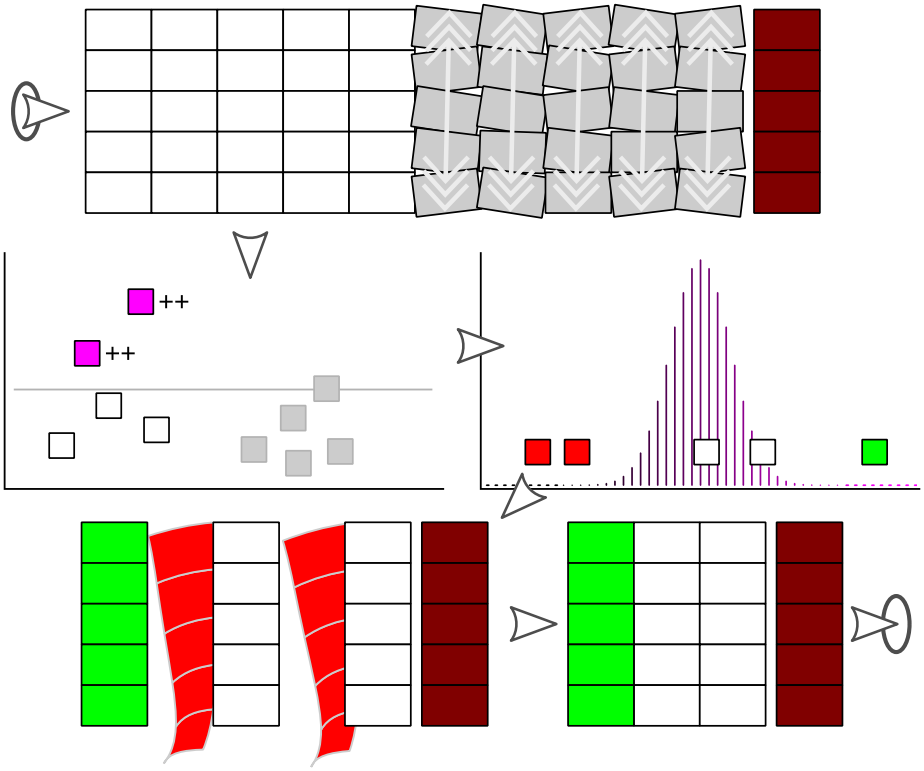

Figure 1: Illustration of the main loop of the Boruta algorithm.

Such scores only contribute to the ranking of features, though — to separate relevant features, we need some reference of what is a distribution of importance of an irrelevant feature. To this end, Boruta uses *shadow features* or *shadows*, which are copies of original features but with randomly mixed values, so that their distribution remains the same yet their importance is wiped out.

As importance scoring is often stochastic and can be degraded due to a presence of shadows, the Boruta selection is a process. In each iteration, first shadows are generated, and such an extended dataset is fed to an importance provider. Original features' importance is then compared with the highest importance of a shadow; and those which score higher are given a *hit*. Accumulated hit counts are finally assessed; features which significantly outperform best shadow are claimed confirmed, while those which significantly under-perform best shadow are claimed rejected and removed from the set for all subsequent iterations. This loop is illustrated on Figure 1.

The algorithm stops when all features have an established decision, or when a pre-set maximal number of iterations (100 by default) is exhausted. In the latter case, the remaining features are claimed *tentative*.

The process can be observed live with `doTrace` argument set to 1 (report

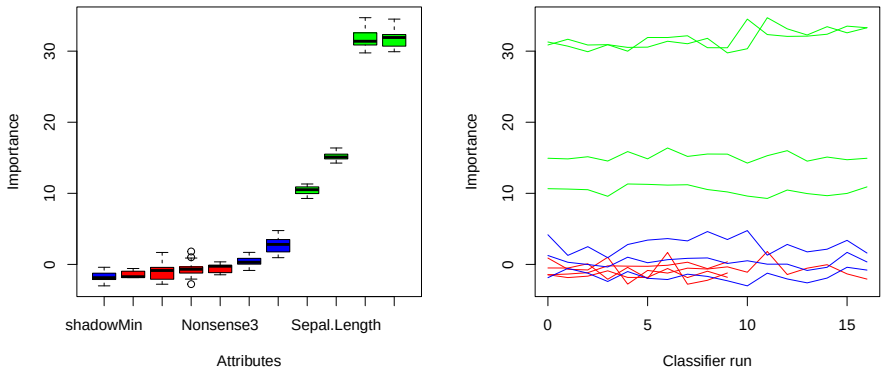

Figure 2: The result of calling `plot` (left) and `plotImpHistory` (right) on the `BorutaOnIrisE` object.

after each decision), 2 (report after each iteration) or 3 (also report hits); importances in each iteration are also stored in the `ImpHistory` element of the Boruta object. The graphical summary of a run can be obtained using `plot` and `plotImpHistory` on the Boruta result object, as shown on Figure 2 for the extended iris example. First function uses boxplots to show the distribution of features' importance over Boruta run, using colours to mark final decision; it also draws boxplots for the importance of worst, average and best shadow in each iteration (in blue). Second function visualises the same data, but as a function of the iteration number. The summary of feature importance and hit counts can be extracted using the `attStats` convenience function.

```
> attStats(BorutaOnIrisE)
```

```
               meanImp  medianImp    minImp     maxImp  normHits  decision
Nonsense1    -0.6223617 -0.6799732 -2.772898  1.8188905 0.1176471  Rejected
Nonsense2    -0.9948939 -0.8577297 -2.786295  1.6816756 0.0000000  Rejected
Nonsense3    -0.4736442 -0.2560304 -1.457678  0.3584504 0.0000000  Rejected
Nonsense4    -1.4697104 -1.7284141 -1.853175 -0.5765322 0.0000000  Rejected
Sepal.Length 15.1547848 15.1036042 14.253937 16.3782501 1.0000000 Confirmed
Sepal.Width  10.4050117 10.5075698  9.277403 11.3140607 1.0000000 Confirmed
Petal.Length 31.7638457 31.9196862 29.916815 34.5062408 1.0000000 Confirmed
Petal.Width  31.7153185 31.3917818 29.752987 34.7062734 1.0000000 Confirmed
```

3 Importance sources

Building Random Forest multiple times on substantially enlarged dataset may easily become very time consuming, especially for larger sets for which us-

ing Boruta makes most sense. It is also possible that RF importance is not best suited to catch the information content of features because of the dataset specifics.

Anyhow, Boruta allows you to switch importance source to an arbitrary function which gets an information system and returns a vector of numeric importance scores of all features. The package already includes adapters for several importance scorers and their various configurations; all start with a `getImp` prefix.

In particular, there is one for rFerns [2], an implementation of random ferns, a purely stochastic ensemble classifier which can usually provide similar importance scores as Random Forest, but in substantially shorter time:

```
> library(rFerns)
> Boruta(Species~.,data=irisE,getImp=getImpFerns)

Boruta performed 14 iterations in 0.07461381 secs.
 4 attributes confirmed important: Petal.Length, Petal.Width,
Sepal.Length, Sepal.Width;
 4 attributes confirmed unimportant: Nonsense1, Nonsense2, Nonsense3,
Nonsense4;
```

You can pass arguments to the importance provider by providing it to the Boruta call; for instance, ranger, the default importance provider, makes use of all available CPU threads, won't always be the optimal choice. Setting `num.threads` in the Boruta call will cause it to relay this argument to the `ranger` function, and hence limit the training process parallelism.

Importance adapters may also be modified by transdapters (functions with names ending with `Transdapter`) to achieve some additional goals; consult the transdapter vignette or manual for more details.

4 Caveats

Few things worth noting before using Boruta in production:

- Boruta is a heuristic; there are no strict guarantees about its output. Whenever possible, try to assess its results, especially in terms of selection stability as classification accuracy may be deceiving [3].
- For datasets with lots of features, the default configuration of the importance source is likely insufficient; in the particular case of Random Forest the number of trees is often not large enough to allow the importance scores to stabilise, which in turn often leads to false negatives and unstable results.
- Boruta is a strictly serial algorithm, and spends most time waiting for the importance provider — hence, tweaking this element brings the best chance to speed up the selection. If speed is a concern, one should also

avoid the formula interface and directly pass predictor and decision parts of the information system.

- Elimination of tentative features becomes practically impossible if they turn out to have very similar importance distribution to the best shadow, and the presence of such does not make the overall Boruta result useless.
- Importance history for bigger problems may take an impractically huge amount of memory; hence its collection can be turned off with `holdHistory` argument of the `Boruta` function. This will disable some functionality, though, most notably plotting.
- Treatment of missing values and non-standard decision forms (like survival problems) depends on the capacity of the information source.
- The original Boruta paper describes the 1.0 version, and the algorithm has undergone substantial changes since then, namely the initial, warm-up rounds were removed, the multiple testing correction was introduced, finally the nomenclature has been clarified.

References

[1] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[2] Miron B. Kursa. rFerns: An Implementation of the Random Ferns Method for General-Purpose Machine Learning. *Journal of Statistical Software*, 61(10), 2014.

[3] Miron B. Kursa. Robustness of Random Forest-based gene selection methods. *BMC Bioinformatics*, 15(1), 2014.

[4] Miron B. Kursa and Witold R. Rudnicki. Feature selection with the boruta package. *Journal of Statistical Software*, 36(11), 2010.

[5] Roland Nilsson, José M. Peña, Johan Björkegren, and Jesper Tegnér. Consistent Feature Selection for Pattern Recognition in Polynomial Time. *The Journal of Machine Learning Research*, 8:589–612, 2007.

[6] Marvin N. Wright and Andreas Ziegler. ranger: A Fast Implementation of Random Forests for High Dimensional Data in C++ and R. *Journal of Statistical Software*, 77(1), 2015.