Random forests – Andy Liaw

# CART – classification and regression trees

Find best “gap” in a variable to split the data into two parts

Repeat until futile

Very naturally handles categorical and numerical vars, including when they’re mixed.

Very greedy algo, making it unstable. Tiny change in data can lead to v diff tree strs.

So not great interpretability.

Can be parallelized at diff levels

Finding “right-sized” tree requires cross val

Generally not v accurate

# Random forests

Similar to CART, but randomly sample vars to split rather than looking at all vars

We are *not* trying to find best split at each step, unlike in CART!

You don’t have a single tree, you have a whole lot of them.

Ensemble model: every model needs to be better than random guessing, try to have diff models make mistakes on diff datapoints.

## Tuning

Low correlation b/w base predictors -> high diversity.

Find good compromise b/w correlation of predictors and strength of each predictor.

Smaller *mtry* tuning parameter gives more diverse forest.

Median correlation vs median RMSE is a way to measure the tradeoff. Tune by altering mtry.

A larger node size forces the algo to produce smaller trees, thus larger neighborhoods

A smaller sampsize induces smaller and more diverse trees, but should be used w/ larger number of trees.

For large data, try small sampsize and large number of trees

Nearest neighbor classifier:

Terminal nodes in a decision tree rep groups of similar data, w/ sizes of neighborhoods decided from training data

RF averages terminal nodes from many regions, so neighborhoods are much less block and rectangular than the ones CART learns from training data. Smooths out the crude neighborhoods of a single tree.

## What controls RFs’ model complexity?

Increasing number of trees is like using smoother weights in NNs

Size of neighborhoods can also indicate model complexity

Smaller trees tend to work better for some data; and small trees tend to give you larger neighborhoods, bc stop splitting earlier and therefore each neighborhood is over larger part of the data

# Overfitting

Boosting and random forests are generally less likely to overfit as train them more; unlike nnets, where early stopping helps.

RF is less prone to overfitting than boosting is.

RF grows each tree to maximum size, thus have nearly 0 training error.

For boosting, test set error can keep decreasing as iterations go on even after training error reached 0. (But will eventually increase – can’t boost forever).

W/ random forests, *can* add as many more trees as you want and grow them as big as you want, w/in computational limits.

Takeaway message: gap between training error and test set error does not *necessarily* indicate overfitting. Increasing test set error with increasing model complexity is what does.

## RFs vs boosting

RF:

Trees independently grown, so can parallelize. Not the case in boosting

Use randomness to get diversity; in boosting, each tree tries to correct previous mistake

Grow trees to max sizes in RF, keep each tree small in boosting

Number of trees is not a tuning parameter in RF, number of trees should be tuned I boosting

Model size can be huge in RF, model size is usually small in boosting (so boosting can be better choice if you’re storage-limited)

Shrinkage is analogous to step size in nnets.

## RF vs DNN

RF: adding more trees doesn’t increase model complexity. DNN: network architecture pre-determines network archi.

No explicit optimization in RF, optimization with controlled greed in DNN.

Prediction can’t exceed range of training data in RF, prediction can be unbounded in DNN (depending on activation fct)

Hard to update an RF model w/ new data, trivial to update a DNN model w/ new data.

RF vs XGBoost vs DNN performance: better performance w/ XGBoost and DNNs, but took a lot more tuning. RF \*can\* train slower than DNN if you’re using an absurd tree but is usually faster. XGBoost trains fast.

# Sensitivity/ interpretability

Variable importance by permutation: a kind of sensitivity analysis. You can permute data one var at a time and see how accuracy drops.

Hold-out data, feed it to trained model, get error on original hold-out data. Then shuffle the k\* features in hold-out data n times, get error of the shuffled data on trained model.

Good approach for all black boxes, not just random forest

## Partial dependence plot

Every predictive model reps a fct w/ multiple vars. The marginal relation b/w y and a particular x can be examined using *partial dependence*.

e.g. for a model y = f(x1,x2)

The partial dependence on x1, rho(x1) , equals the integral of f(x1,x2) over dx2

i.e. all remaining variables are integrated out.

Computing partial dependence:

Replace orginal values in column x1 with a constant, like 1.2

Predict outcome using modified data

Compute the average prediction of y when x1 = 1.2

Repeat the process with different x1 values to obtain the partial dependence function rho(x1)

R package pdp implements this, as do ALEPlot and ICEbox

## Prediction intervals

Rarely used but Andy thinks it’s handy.

quantregForest package and grf package (which also gives local likelihood). Post-process a randomForest object.

For each new data point to be predicted, it lands in a leaf in each tree and is predicted by the mean of the (in-bag) data in that leaf, then averaged over all trees.

We can use the in-bag data that fell in the same terminal nodes as the new data point as a sample from the conditional distribution, use it to estimate the conditional quantiles.

Doing this by other methods (like customizing the loss fct to estimate quantiles) requires fitting a separate model for each quantile.