An Introduction to Recursive Partitioning for Instrumental Variable Tree Using `IVTree` package

Guihua Wang
Jun Li
Wallace J. Hopp

November 22, 2019

Contents

1 Introduction

This document is a brief introduction of `causalTree` package, which includes the `causalTree` function and the `honest.causalTree` function. These implement the methods from *Recursive Partitioning for Heterogeneous Causal Effects* [1]. The package also includes the functions `causalForest` and `propensityForest` which implement versions of the causal forest algorithm from *Estimation and Inference of Heterogeneous Treatment Effects using Random Forests* [3]

The `causalTree` function builds a regression model and returns an `rpart` object, which is the object derived from `rpart` package, implementing many ideas in the CART (Classification and Regression Trees), written by Breiman, Friedman, Olshen and Stone [2]. Like `rpart`, `causalTree` builds a binary regression tree model in two stages, but focuses on estimating heterogeneous treatment effects. The function requires the user to specify a binary treatment variable in addition to the outcome variable and the features that are usually passed to `rpart`.

Following `rpart`, in the first stage, the tree is grown from the root node based on a specified splitting rule. In each node, the data in a leaf will be split into two groups to best minimize the risk function. Next, in the left sub-node and right sub-node, the splitting routine will be applied separately and so on recursively until no improvements can made,

or until some limits are reached (e.g. the routine will stop if it cannot make splits that have at least `minsize` of treated observations and `minsize` control observations in each terminal node.) The risk for each node is calculated using a risk function associated with the splitting rule.

Unlike `rpart`, which estimates the average of the outcome variable in each leaf, `causalTree` estimates a treatment effect in each leaf by taking the difference of the sample average of the treated group and the sample average of the control group within the leaf. The user selects one of several splitting rules for determining the optimal split at each step.

The user also specifies a cross-validation method. Cross-validation is implemented similar to `rpart`, where the cross-validation penalty parameter penalizes the number of nodes in the tree. The main difference arises in the cross-validation criterion, which is selected by the user. The `cptable` of the `rpart` object includes the cross-validation error. The user can choose to prune the tree using a specified cross-validation method, where The leaves to be pruned are selected according to the risk function calculated while the tree is built.

The `causalTree` package incorporates an additional function not included in `rpart`, which is honest re-estimation `honest.causalTree` of causal effects. Honest here means that we estimate causal effects in the leaves of a given tree on an independent estimation sample rather than the data used to build and cross-validate the tree. The user first builds the tree with `causalTree`, specifying the training data for building the tree, and then passes the tree object as well as the estimation sample data into `honest.causalTree`, which replaces the leaf estimates from the input tree with new estimates in each leaf, calculated on the estimation sample.

To allow the user more control, there is another function, `estimate.causalTree`, that takes in an `rpart` object (which could have been estimated using the `rpart` package) and replaces the leaf estimates with sample average treatment effects within each leaf, using a new dataset passed in by the user. So it is possible to build a tree in either `causalTree` or `rpart` using any one of a number of methods, and then use `estimate.causalTree` to estimate treatment effects in each leaf on an arbitrary dataset.

The package has a few additional features not present in `rpart`. One is that the `minsize` parameter requires that there are at least `minsize` treated and `minsize` control observations in each leaf, so that a sample average treatment effect can be calculated.

Another feature is that we allow the user the option to restrict the set of potential split points considered, and further, in the splitting process we rescale the covariate values within each leaf and each treatment group in order to ensure that when moving from one potential split point to the next, we move the same number of treatment and control observations from the right leaf to the left leaf.

The rpart algorithm considers every value of every feature as a possible split point. An obvious disadvantage of this approach is that computation time can grow prohibitively large in models with many observations and features. But there are some more subtle disadvantages as well. The first is that there will naturally be sampling variation in calculating the

risk function as we vary the possible split points. A problem akin to a multiple hypothesis testing problem arises: since we are looking for the maximum value of an estimated criterion across a large number of possible split points, as the number of split points tested grows, it becomes more and more likely that one of the splits for a given covariate appears to improve the fit criterion even if the true value of the criterion would indicate that it is better not to split. One way to mitigate both the computation time problem and the multiple-testing problem is to consider only a limited number of split points.

A third problem is specific to considering treatment effect heterogeneity. To see the problem, suppose that a covariate strongly affects the mean of outcomes, but not treatment effects. Within a leaf, some observations are treated and some are control. If we consider every level of the covariate in the leaf as a possible split point, then shifting from one split point to the next shifts a single observation from the right leaf to the left leaf. This observation is in the treatment or the control group, but not both; suppose it is in the treatment group. If the covariate has a strong effect on the level of outcomes, the observation that is shifted will be likely have an outcome more extreme than average. It will change the sample average of the treatment group, but not the control group, leading to a large change in the estimated treatment effect difference. We expect the estimated difference in treatment effects across the left and right leaves to fluctuate greatly with the split point in this scenario. This variability around the true mean difference in treatment effects occurs more often when covariates affect mean outcomes, and thus it can lead the estimators to split too much on such covariates, and also to find spurious opportunities to split.

To address this problem, we incorporate the following modifications to the splitting rule. We include a parameter `bucketNum`, the target number of observations per "bucket." For each leaf, before testing possible splits for a particular covariate, we order the observations by the covariate value in the treatment and control group separately. Within each group, we place the observations into buckets with `bucketNum` observations per bucket. If this results in less than `minsize` (another user-set parameter) buckets, then we use fewer observations per bucket (to attain `minsize` buckets). If this results in more than `bucketMax` buckets, we use more observations per bucket to obtain `bucketMax` buckets. We number the buckets, and considering potential splits by bucket number rather than the raw values of the covariates. This guarantees that when we shift from one split point to the next, we add both treatment and control observations, leading to a smoother estimate of the goodness of fit function as a function of the split point. After the best bucket number to split on is selected, we translate that into a split point by averaging the maximum covariate value in the corresponding treatment and control buckets.

The `CausalForest` function code extends the `CausalTree` method, specifically by building an ensemble of trees, based on a user specified number. The data is repeatedly sampled to build each tree (with replacement).This function returns a causalForest object, which is a list of `rpart` objects. Note that this function always performs honest estimation, by using one subset of the data to build the tree ensemble and another subset of the data to

estimate the treatment effects at the leaves of the built trees. To predict, call R's predict function with new test data and the causalForest object (estimated on the training data) obtained after calling the causalForest function. During the prediction phase, the average value over all tree predictions is returned as the final prediction.

An alternative method to build an ensemble of trees is also implemented using the function `propensityForest`. In this function, an ensemble of trees are built, and splitting nodes for each tree are determined using the treatment variable (along with the input covariates) as a proxy for the output variable. To evaluate the tree ensemble built, treatment effects are estimated using the true output variable along with the input covariates.

2 Notation

References

[1] Susan Athey and Guido Imbens. Machine learning methods for estimating heterogeneous causal effects. *arXiv preprint arXiv:1504.01132*, 2015.

[2] L. Breiman, J.H. Friedman, R.A. Olshen, , and C.J Stone. *Classification and Regression Trees*. Wadsworth, Belmont, Ca, 1983.

[3] Stefan Wager and Susan Athey. Estimation and inference of heterogeneous treatment effects using random forests. *arXiv preprint arXiv:1510.04342*, 2015.