The Journal of Open Source Software

policytree: Policy learning via doubly robust empirical welfare maximization over trees

Erik Sverdrup[1], **Ayush Kanodia**[1], **Zhengyuan Zhou**[2], **Susan Athey**[1], **and Stefan Wager**[1]

1 Stanford Graduate School of Business **2** NYU Stern

DOI: 10.21105/joss.02232

Software

- Review
- Repository
- Archive

Editor: Arfon Smith

Reviewers:

- @nhejazi
- @jjharden

Submitted: 20 May 2020
Published: 22 June 2020

Summary

The problem of learning treatment assignment policies from randomized or observational data arises in many fields. For example, in personalized medicine, we seek to map patient observables (like age, gender, heart pressure, etc.) to a treatment choice using a data-driven rule.

There has recently been a considerable amount of work on statistical methodology for policy learning, including Manski (2004), Zhao, Zeng, Rush, & Kosorok (2012), Swaminathan & Joachims (2015), Kitagawa & Tetenov (2018), van der Laan & Luedtke (2015), Luedtke & van der Laan (2016), Mbakop & Tabord-Meehan (2016), Athey & Wager (2017), Kallus & Zhou (2018), and Zhou, Athey, & Wager (2018). In particular, Kitagawa & Tetenov (2018) show that if we only consider policies π restricted to a class Π with finite VC dimension and have access to data from a randomized trial with n samples, then an empirical welfare maximization algorithm achieves regret that scales as $\sqrt{\mathrm{VC}(\Pi)/n}$. Athey & Wager (2017) extend this result to observational studies via doubly robust scoring, and Zhou et al. (2018) further consider the case with multiple treatment choices (in particular, the regret will depend on the tree depth, feature space, and number of actions).

The package `policytree` for `R` (R Core Team, 2020) implements the multi-action doubly robust approach of Zhou et al. (2018) in the case where we want to learn policies π that belong to the class Π of depth-k decision trees. In order to use `policytree`, the user starts by specifying a set of doubly robust scores for policy evaluation; the software then carries out globally optimal weighted search over decision trees.

It is well known that finding an optimal tree of arbitrary depth is NP-hard. However, if we restrict our attention to trees of depth k, then the problem can be solved in polynomial time. Here, we implement the global optimization via an exhaustive (unconstrained) tree search that runs in $O(P^k N^k (\log N + D) + PN \log N)$ time, where N is the number of individuals, P the number of characteristics observed for each individual and D is the number of available treatment choices (see details below). If an individual's characteristics only takes on a few discrete values, the runtime can be reduced by a factor of N^k. Additionally, an optional approximation parameter lets the user control how many splits to consider.

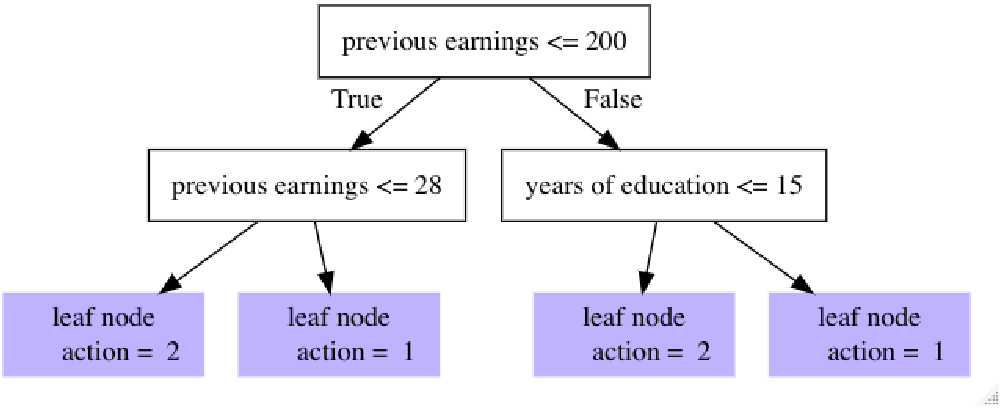

Figure 1: A depth 2 tree fit on data from the National Job Training Partnership Act Study (Bloom et al., 1997). The reward matrix contains two outcomes: not assigning treatment (action 1), and assigning treatment, a job training program (action 2). The covariate matrix contains two variables: a candidate's previous annual earnings in $1,000 and years of education. Note: the optional package `DiagrammeR` is needed to plot trees.

Our package is integrated with the `R` package `grf` of Athey, Tibshirani, & Wager (2019), allowing for a simple workflow that uses random forests to estimate the nuisance components required to automatically form the doubly robust scores. We also generalize the `causal_forest` function from `grf` to multiple treatment effects with a one vs all encoding described in Zhou et al. (2018). The following simulation example illustrates this workflow in a setting with $D = 3$ actions; here, we write covariates with `X`, outcomes as `Y`, and actions as `W`. Figure 1 shows a tree similarly grown on a dataset considered by Kitagawa & Tetenov (2018).

```r
library(policytree)
X <- matrix(rnorm(2000 * 10), 2000, 10)
W <- sample(c("A", "B", "C"), 2000, replace = TRUE)
Y <- X[,1] + X[,2] * (W == "B") + X[,3] * (W == "C") + runif(2000)
multi.forest <- multi_causal_forest(X = X, Y = Y, W = W)
DR.scores <- double_robust_scores(multi.forest)
tr <- policy_tree(X, DR.scores, depth = 2)
plot(tr)
```

The core tree search functionality is built in C++ using the `Rcpp` interface (Eddelbuettel et al., 2011). This approach to tree search is discussed further by Zhou et al. (2018), who find it to scale better to large sample size problems than an alternative based on mixed-integer programming. We also note the `R` package `evtree` by Grubinger, Zeileis, & Pfeiffer (2014), which can be used to heuristically optimize over decision trees via evolutionary search, and `tmle3mopttx` by Malenica, Coyle, & van der Laan (2020), for estimating optimal individual treatments.

Appendix:

Details on tree search

The pseudocode for the tree search is outlined in Algorithm 1 and Algorithm 2. At a high level, in the main recursive case for $k >= 2$, the algorithm maintains the data structure $sorted\_sets$ to quickly obtain the sort order of points along all dimensions P for a given split. For each of the $P \times (N-1)$ possible splits, for each dimension j all points on the right side are stored in $set_R(j)$. All points on the left side are stored in $set_L(j)$. For each split candidate, the point is moved from the right set to the left set for all dimensions. This proceeds recursively to enumerate the reward in all possible splits.

The $O(PN \log N)$ term arises from the fixed amortized cost of creating the global sort order once for every sample along all P dimensions. The remaining $O(P^k N^k(\log N + D))$ term is obtained by inductively calculating the runtime for increasing depths k.

Algorithm 1: Exact tree search.
In the implementation, parents with identical actions in both leaves are *pruned*. It also features an optional approximation parameter than controls the number of splits to consider.
The recursion base case is both at a leaf node ($k = 0$) as well as at the parent of a leaf ($k = 1$) where one can jointly compute the best action in each leaf in $O(NPD)$ by a dynamic programming style algorithm). Peripheral functions are outlined at the end

```
function tree_search ($sorted\_sets, \Gamma, k$);
Input  : $P$-vector $sorted\_sets$, $N \times D$ score matrix $\Gamma$, tree depth $k$
Output: The optimal tree, a structure with (left node, right node, total reward, action)
if $k$ = 0 then
    $tree.reward, tree.action \leftarrow \{\text{max, argmax}\}_{j \in 1,\ldots,d} \sum_{i \in 1,\ldots,N} \Gamma_{ij}$;
    $tree.left = \emptyset, tree.right = \emptyset$ ;
    return $tree$;
end
if $k$ = 1 then
    return $tree\_search\_single\_split(sorted\_sets, \Gamma)$;
end
$best\_tree_L \leftarrow \emptyset$;
$best\_tree_R \leftarrow \emptyset$;
$best\_reward \leftarrow -\infty$;
for $p$=1:P do
    $sets_R \leftarrow copy(sorted\_sets)$;
    $sets_L \leftarrow create\_empty\_sorted\_sets()$;
    for $n$=(1: N-1) do
        $sample_n \leftarrow sets_R(p).begin()$;
        $sets_L(p).insert(sample_n)$;
        $sets_R(p).erase(sample_n)$;
        for $j \neq p$ do
            $set_R(j).erase(sample_n)$;
            $set_L(j).insert(sample_n)$;
        end
        $tree_L \leftarrow$ tree_search$(sets_L, \Gamma, k-1)$;
        $tree_R \leftarrow$ tree_search$(sets_R, \Gamma, k-1)$;
        $reward = tree_L.reward + tree_R.reward$;
        if $best\_tree\_L = \emptyset \mid\mid reward > best\_reward$ then
            $best\_tree_L \leftarrow tree_L$;
            $best\_tree_R \leftarrow tree_R$;
            $best\_reward = reward$;
        end
    end
end
return $tree(best\_tree_L, best\_tree_R, best\_reward, \emptyset)$;
```

Algorithm 2: O(NPD) implementation for a single split

```
function tree_search_single_split ($sorted\_sets, \Gamma$);
Input  : $P$-vector $sorted\_sets$, $N \times D$ score matrix $\Gamma$
Output: The optimal tree, a structure with (left node, right node, total reward, action)
$cum\_rewards \leftarrow array(D)(P)(N)$ ;
for $d = (1 : D)$ do
    for $p = (1 : P)$ do
        $iter = sorted\_sets(p).first()$;
        $index = iter.index()$;
        $iter.next()$;
        $cum\_rewards(d)(p)(1) \leftarrow \Gamma_{1index}$;
        for $n = (2 : N)$ do
            $index = iter.index()$;
            $iter.next()$;
            $cum\_rewards(d)(p)(n) \leftarrow \Gamma_{indexd} + cum\_rewards(d)(p)(n-1)$;
        end
    end
end
$best\_reward_L, best\_reward_R \leftarrow \emptyset, \emptyset$;
$best\_action_L, best\_action_R \leftarrow \emptyset, \emptyset$;
for $p = (1 : P)$ do
    for $n = (1 : N)$ do
        $reward_L, action_L \leftarrow$
          $\{\text{max, argmax}\}_{d \in 1,..,D} cum\_rewards(d)(p)(n)$;
        $reward_R, action_R \leftarrow$
          $\{\text{max, argmax}\}_{d \in 1,..,D} cum\_rewards(D)(p)(N) - cum\_rewards(d)(p)(n)$;
    end
    if $reward_L + reward_R > best\_reward_L + best\_reward_R$ then
        $best\_reward_L, best\_action_L \leftarrow reward_L, action_L$;
        $best\_reward_R, best\_action_R \leftarrow reward_R, action_R$;
    end
end
$best\_tree_L \leftarrow tree(\emptyset, \emptyset, best\_reward_L, best\_action_L)$;
$best\_tree_R \leftarrow tree(\emptyset, \emptyset, best\_reward_R, best\_action_R)$;
return $tree(best\_tree_L, best\_tree_R, best\_reward_L + best\_reward_R, \emptyset)$;
```

Deriving the running time

Base Case 1: k = 0 (no splits): In this case, all we need to do is calculate the sum of rewards over each of the available treatment choices D for the N users. Hence, the time complexity is $O(ND)$.

Base Case 2: k = 1 (1 split): In this case, as we show in Algorithm 2, the time complexity is $O(NPD + NP \log N)$. We first sort all N points along all P dimensions. This accounts for the $NP \log N$ term. Along each dimension, first we keep a cumulative sum of rewards for each treatment on both sides of every possible split. This takes time $O(ND)$ given the sorted order on points along that dimension. We can then calculate the best split point given this sort order, along with the best policy in both splits in time $O(ND)$, as in the pseudocode. Doing this over all dimensions, we get $O(NPD)$. Combining this with the initial sort, we get $O(NPD + NP \log N) = O(NP(\log N + D))$.

Recursive Case We propose the time complexity for $k >= 1$ (1 or more splits) to be $O(P^k N^k (\log N + D))$. This is satisfied for base case 2 above. For the recursive case, there are PN possible split points. For every single split along along every dimension we remove a sample from a Binary Search Tree and add to another; this takes $O(\log N)$ time, and we do this for each of the P dimensions, leading to time $(P \log N)$. Further, for each split, we recursively call $tree\_search$ for depth $k-1$, in general there are m_1 and m_2 points in each split at the top level such that $N = m_1 + m_2$. Assuming the recursive expression, the amount of work done for each split is then

$$O(P \log N + m_1^{k-1} P^{k-1}(\log m_1 + D) + m_2^{k-1} P^{k-1}(\log m_2 + D))$$

Note that,

$m_1^{k-1} P^{k-1}(\log m_1 + D) < m_1^{k-1} P^{k-1}(\log N + D)$ since $m_1 < N$.

Similarly,

$m_2^{k-1} P^{k-1}(\log m_2 + D) < m_2^{k-1} P^{k-1}(\log N + D)$ since $m_2 < N$.

Further,

$$m_1^{k-1} P(\log N + D) + m_2^{k-1} P(\log N + D) < N^{k-1} P^{k-1}(\log N + D)$$

since $m_1 + m_2 = N, m_1, m_2, N > 0$.

Combining, the amount of work in each split is upper bounded by

$$O(P^{k-1} N^{k-1}(\log N + d)).$$

Since we have PN splits, this leads to a running time of

$$O(PN(P^{k-1} N^{k-1}(\log N + d)) = O(P^k N^k(\log N + d)).$$

Algorithm: Peripheral functions for Algorithm 1

```
function create_sorted_sets ($X$);
Input   : $N \times P$ covariate matrix $X$
Output: A length $P$ vector, the jth vector containing all $N$ samples sorted along
        dimension $j$
result ← vector(P);
for $j$=1:$P$ do
    $result(j)$ ← binary_search_tree(j);
    for $i$=1:$N$ do
        $result(j)$.insert($x_i$);
    end
end
return result;
function create_empty_sorted_sets ();
Input   : $P$ Number of dimensions
Output: A length $P$ vector, the jth vector is empty, but to be sorted along dimension $j$
result ← vector(P);
for $j$=1:$P$ do
    $result(j)$ ← binary_search_tree(j);
end
return result;
```

References

Athey, S., Tibshirani, J., & Wager, S. (2019). Generalized random forests. *The Annals of Statistics*, *47*(2), 1148–1178.

Athey, S., & Wager, S. (2017). Efficient policy learning. *arXiv preprint arXiv:1702.02896*.

Bloom, H. S., Orr, L. L., Bell, S. H., Cave, G., Doolittle, F., Lin, W., Bos, J. M., et al. (1997). The benefits and costs of jtpa title ii-a programs: Key findings from the national job training partnership act study. *Journal of human resources*, *32*(3). doi:10.2307/146183

Eddelbuettel, D., François, R., Allaire, J., Ushey, K., Kou, Q., Russel, N., Chambers, J., et al. (2011). Rcpp: Seamless r and c++ integration. *Journal of Statistical Software*, *40*(8), 1–18.

Grubinger, T., Zeileis, A., & Pfeiffer, K.-P. (2014). Evtree: Evolutionary learning of globally optimal classification and regression trees in r. *Journal of Statistical Software*, *61*(1), 1–29. doi:10.18637/jss.v061.i01

Kallus, N., & Zhou, A. (2018). Confounding-robust policy improvement. In *Advances in neural information processing systems* (pp. 9269–9279).

Kitagawa, T., & Tetenov, A. (2018). Who should be treated? Empirical welfare maximization methods for treatment choice. *Econometrica*, *86*(2), 591–616. doi:10.3982/ECTA13288

Luedtke, A. R., & van der Laan, M. (2016). Super-learning of an optimal dynamic treatment rule. *The international journal of biostatistics*, *12*(1), 305–332. doi:10.1515/ijb-2015-0052

Malenica, I., Coyle, J., & van der Laan, M. (2020). *Tmle3mopttx: Targeted maximum likelihood estimation of the mean under optimal individualized treatment*. Retrieved from https://tlverse.org/tmle3mopttx

Manski, C. F. (2004). Statistical treatment rules for heterogeneous populations. *Econometrica*, *72*(4), 1221–1246. doi:10.1111/j.1468-0262.2004.00530.x

Mbakop, E., & Tabord-Meehan, M. (2016). Model selection for treatment choice: Penalized welfare maximization. *arXiv preprint arXiv:1609.03167*.

R Core Team. (2020). *R: A language and environment for statistical computing*. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from https://www.R-project.org/

Swaminathan, A., & Joachims, T. (2015). Batch learning from logged bandit feedback through counterfactual risk minimization. *The Journal of Machine Learning Research*, *16*(1), 1731–1755.

van der Laan, M., & Luedtke, A. R. (2015). Targeted learning of the mean outcome under an optimal dynamic treatment rule. *Journal of causal inference*, *3*(1), 61–95.

Zhao, Y., Zeng, D., Rush, A. J., & Kosorok, M. R. (2012). Estimating individualized treatment rules using outcome weighted learning. *Journal of the American Statistical Association*, *107*(499), 1106–1118.

Zhou, Z., Athey, S., & Wager, S. (2018). Offline multi-action policy learning: Generalization and optimization. *arXiv preprint arXiv:1810.04778*.