Detecting Heterogeneous Treatment Effects of Spark Configuration Parameters using Causal Trees

Sadaf Sadeghian, Ghazal Sadeghian, Shadab Shaikh
University of British Columbia

ABSTRACT

Spark [6] is a distributed data analytic engine for speeding up applications processing big data. Multiple different configuration parameters in Spark can impact the execution time of an application. In this project, we use causal trees to investigate the heterogeneous effect of each Spark configuration on an application's execution time. Using these causal trees, we output a set of rules about the effect of each configuration. Both users and admins can benefit from these rules, as they can adjust the value of each configuration based on other configurations in order to get a maximum reduction in the execution time.

KEYWORDS

Spark, Causal Trees, Heterogeneous Treatment Effects, Configuration parameters

ACM Reference Format:
Sadaf Sadeghian, Ghazal Sadeghian, Shadab Shaikh. 2022. Detecting Heterogeneous Treatment Effects of Spark Configuration Parameters using Causal Trees. In *Causal ML, Winter Session Term 1, 2022.* , 9 pages.

1 BACKGROUND

1.1 Conditional Average Treatment Effect

Treatment effects can vary across different subpopulations of the data. This heterogeneity in treatment effects can be discovered by accurately estimating the Conditional Average Treatment Effect (CATE). More formally, CATE is defined as: $\tau(x) := \mathbb{E}[Y_i(1) - Y_i(0)|X_i = x]$, where $Y_i(1)$ and $Y_i(0)$ are the potential outcomes for when the unit i were treated and not treated respectively and X_i is the covariate vector. The goal then is to obtain accurate estimates $\hat{\tau}(x)$ for CATE $\tau(x)$.

1.2 Causal Trees

Overview. Causal trees are a popular approach for estimating heterogenous treatment effects. Imbens et al.[5] introduce a method to build trees by partitioning the covariate space by maximizing an objective function. A partitioning of the covariate space $\mathcal{X}$ into L partitions is defined as $\mathcal{X} = \mathcal{X}_1 \cup ..... \cup \mathcal{X}_L$ where $\mathcal{X}_l \cap \mathcal{X}_{l'} = \emptyset, \forall l \neq l'$. The set of examples corresponding to partition $\mathcal{X}_l$ is $\mathcal{S}_l = \{(X_i, Y_i, T_i) : X_i \in \mathcal{X}_l\}$.

CPSC 538L, Winter Session Term 1, 2022,
2022.

In each partition, the population is divided into treatment group and control group. The average treatment effect in a partition S_l is computed as:

$$\hat{\tau}(S_l) = \hat{\mu}_1(S_l) - \hat{\mu}_0(S_l) \tag{1}$$

where $\hat{\mu}_1$ and $\hat{\mu}_0$ are conditional means of treatment and control group's outcome in partition S_l, defined as:

$$\hat{\mu}_t(S_l) = \frac{1}{N_{l_t}} \sum_{T_i=t, i \in S_l} Y_i \tag{2}$$

here N_{l_1} and N_{l_0} are the number of units in the treatment and control groups in the partition.

Splitting. Given a node l that needs to be partitioned into two child nodes l_1, l_2, the algorithm finds the split that maximizes some partition measure F for the two children:

$$\max_{S_{l_1}, S_{l_2}} F(S_{l_1}) + F(S_{l_2}) \tag{3}$$

1.3 Removing Bias in Causal Trees

The traditional approach (also known as ***"adaptive"*** approach) Imbens et al. [5] use the same training sample for partitioning while creating the tree (eq. 3) as well as estimating the treatment effect for that partition (eq. 1). However, this could lead to estimation biases [4]. To counteract this effect, Athey et al. [4] come up with an ***"honest"*** estimation approach where the data sample S within a partition is splitted into training S^{tr} and estimation S^{est} sets where S^{tr} is used for constructing the partition and S^{est} is used to estimating the treatment effects within the partition. In the honest trees, splitting criteria rewards a partition for finding strong heterogeneity in treatment effects and penalizes a partition that creates variance in leaf estimates. Equation 4 shows the honest penalty term to penalize the variance in estimations across the partition S_l.

$$H(S_l) = \left(1 + \frac{N^{est}}{N}\right) \cdot \left(\frac{V^{(1)}(S_l)}{p} + \frac{V^{(0)}(S_l)}{1-p}\right) \tag{4}$$

where $p = \frac{N^{tr}}{N}$ is the fraction of treated units and $V^{(1)}(S_l)$, $V^{(0)}(S_l)$ are the variances of treated and control units.

1.4 Improving Generalizability in Causal Trees

In the previous approaches of building causal trees mentioned so far, there's no guarantee that they'll be able to find splits that generalize well to unseen data.

To fix this issue, Tran et al. [2] introduce **Validation Honest Causal Tree** approach which we make use of in our project. In this approach, the dataset S is explicitly split into training, validation, and estimation samples, denoted by S^{tr}, S^{val} and S^{est}. The tree is built on the training sample while penalizing generalizability on the validation set. This is done by introducing a generalization cost term shown in eq.5.

$$F_C(S_l) = \frac{(1-\lambda)\cdot F(S_l^{tr}) - \lambda \cdot C(S_l^{val})}{|N_l^{tr} - N_l^{val}| + 1} \tag{5}$$

In eq.5, $\lambda \in [0, 1]$ is a parameter that decides how strongly we want to penalize the splits that do not generalize well, and the cost term $C(S_l^{val}) = N_l^{val} \cdot |\hat{\tau}(S_l^{val}) - \tilde{\tau}(S_l^{val})|$, where $\hat{\tau}$ is the estimated ATE and $\tilde{\tau}$ is the true ATE.

Finally, the **Validation Honest Causal Tree** approach makes use of both honest penalty and the validation set to provide both generalizability and unbiased estimation. The partition measure in these trees is calculated as shown in the eq.6.

$$F_{HL}(S_l) = F_C(S_l) - H(S_l) \tag{6}$$

1.5 Handling Continuous Treatments

In many scenarios, the treatment is continuous and the effect is only triggered if the treatment is greater than or equal to a certain threshold. In such scenarios, the partitioning will depend not only on the covariate space but also on the optimal triggers. Since in our project we also have some continuous treatments, we decided to use this trigger-based approach.

Trigger: Considering T to be the set of all treatment values, the trigger $\theta_l \in T$ of a sample S_l is defined as the threshold that splits S_l into two sub-populations S_{l_1} and S_{l_0} in a way that maximizes the treatment effect for S_l. As shown in eq.7

$$\underset{\theta_l}{\operatorname{argmax}}\, \hat{\mu}_1(S_{l_1}) - \hat{\mu}_0(S_{l_0}) \tag{7}$$

where $S_l = S_{l_1} \cup S_{l_0}$ and $T_i = 1$ if $t_i \geq \theta_l$ and $T_i = 0$ otherwise. Since the treatment effect now also depends on the choice of trigger θ_l for partition l, it becomes a joint optimization problem where the objective eq. 6 is optimized over both the covariate to split on as well as the trigger value. Equation 8 shows the partition measure for the trigger-based treatment effect.

$$F^{\mathcal{T}} = \max_{\theta_l} F_{HL}(S_l) \tag{8}$$

Note that, the optimal trigger value can be different for each node in the tree.

Pruned Trees: As proposed by Tran et al. [2], we report results for trees that are pruned based on the statistical significance value (p) of ATE estimation in each node. To illustrate, the nodes with a p-value higher than a threshold are pruned from the tree as their estimation is not significant. The p value measures whether the assumption that the treated and control population in each node have a similar underlying distribution which is one of the assumptions that this paper relies on for having a good estimation of treatment effect.

2 PROBLEM

In this project, we consider five Spark configurations, and we aim to investigate the treatment effect of each configuration on the execution time of an application running on Spark. Since it is favorable to configure the Spark configurations to values that result in faster execution of the application, we aim to find ranges of each configuration that result in a decrease in the application execution time using the estimated treatment effects.

2.1 Potential Outcomes Framework

To explain our project formally, we use the potential outcomes framework:

Unit: One execution of the application

Treatment: Assigning a value to the Spark configuration under investigation (For example if the configuration under investigation is Serializer, we can set it to KyroSrialilzer or JavaSerializer)

Outcome: The execution time of the configured application

In our setup, we have SUTVA assumptions since when we set the configuration to a specific value, the treatment will be similar. Also, there is no interference between units since they are running individually and don't share resources with other applications.

2.2 Causal Graph

Fig.1 shows the causal graph of our setup that is built using domain knowledge. Let

- $\mathcal{T}$ be the set of treatments (Spark configurations)
- $\mathcal{M}$ be the set of mediators (runtime system measurements)
- Y be the outcome variable (the execution time of an application)

Suppose we are analyzing the effect of a particular treatment $t_i \in \mathcal{T}$. Let $\mathcal{M}_i^{cp}$ be the set of mediators on the causal path between t_i and Y.

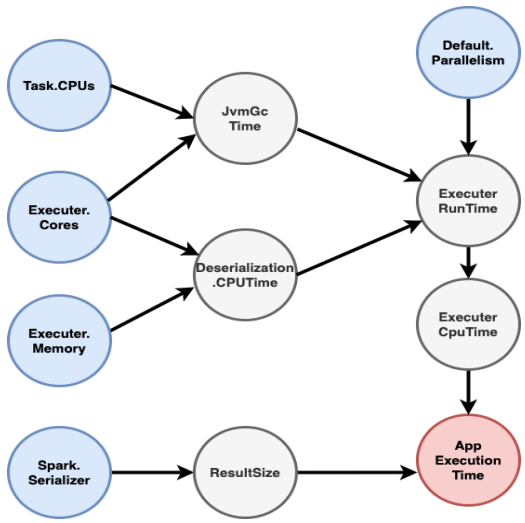

Figure 1: Causal graph showing the treatments (blue), mediators (grey), and the outcome (red) in our setup.

We run our analysis in 2 modes of observability:

(1) ***User-level:*** A user can only observe the configurations present in the treatment set $\mathcal{T}$. Hence we consider $\left(\mathcal{T} \setminus t_i\right)$ as the covariate vector X_i

(2) ***Admin-level:*** An administrator can observe all the treatments in $\mathcal{T}$ as well as well as runtime metrics present in $\mathcal{M}$. Hence we consider $\left(\mathcal{T} \setminus t_i\right) \cup \left(\mathcal{M} \setminus \mathcal{M}_i^{cp}\right)$ as the covariate vector X_i

2.3 Dataset and Setting

The dataset we're using in this project is collected by running the Hibench [3] K-means application with a large input size on a cluster consisting of 5 nodes using Spark. The data is collected from a randomized experiment in which the value of each configuration is picked randomly in a specified range. After each execution of the application with the chosen configurations, the application execution time and runtime metrics are measured.

We consider 5 Spark configurations mentioned below:

- Executor Cores: The number of cores to use on each executor, range = [1, 7]
- Executor Memory: Amount of memory to use per executor process, range = [1, 12]
- Parallelism: Default number of partitions in RDDs returned by transformations, range = [8, 50]
- Serializer: Class to use for serializing objects that will be sent over the network or need to be cached in serialized form, range = [KryoSerializer, JavaSerializer]
- Task CPUs: Number of cores to allocate for each task, range = [1, 2]

We consider 5 runtime metrics that are aggregated from the task level for each application. They are mentioned below:

- JvmGcTime: Elapsed time the JVM spent in garbage collection while executing this task, unit = milliseconds
- ExecutorDeserializationCpuTime: CPU time taken on the executor to deserialize this task, unit = nanoseconds
- ExecutorRunTime: Elapsed time the executor spent running this task, unit = milliseconds
- ExecutorCpuTime: CPU time the executor spent running this task, unit = nanoseconds
- ResultSize: The number of bytes this task transmitted back to the driver as the TaskResult, unit = byte

Finally, the execution time is measured in milliseconds and we consider it as our outcome in this project.

3 EXPERIMENT RESULTS

As discussed in the background section, the goal is to derive heterogeneous subgroups and estimate the treatment effect for each configuration parameter using causal trees. To create the causal trees, we used CTL tool [1], which Tran et al. [2] proposed.

By traversing the tree starting from the root node and going through the path to reach a specific leaf, we can find the decision rules for that subpopulation. These decision rules show a range of possible values for the covariates. These rules can be seen as a look-up table, which helps both the users and admins in setting proper values for the configurations.

3.1 User-level Analysis

For each configuration parameter, that specific parameter is considered as the treatment, and the subgroups are differed based on the other configurations' values (covariates). The goal here is to find the treatment effect (defined as the differences in execution times) for each subpopulation.

Based on each tree, we defined a set of rules as a look-up table. Therefore, by knowing other configurations' values, the user can simply find an acceptable range to pick a value for the treatment configuration. The best choice would be a value that can give her the maximum reduction in the execution time.

In this section, the rules derived from the trees for each configuration are provided.

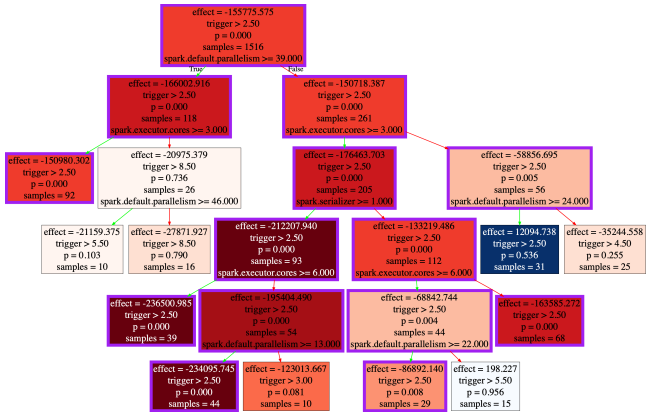

Figure 2: A tree illustrating the effect of the executor memory as the treatment on the execution time based on the other configurations as the covariates(User-level).

3.1.1 *Executor Memory:*

Figure2 shows the tree where the executor memory is chosen as the treatment. As an example of how these rules are derived, the right-most path in the tree reads as: "If parallelism is less than 24, and the number of executor cores is less than 3, the average effect of setting the value of the memory to more than 4.5 GB on the execution time is a decrease of around 35.2 seconds."

The rest of the rules are listed in Table 1. The negative effect(shown in red color) indicates a decrease in the execution time which is desirable, and a positive effect(shown in blue color) shows an increase, which is undesirable in our setting.

Treatment	taskCPUs	ExeCores	Serializer	Parallelism	Effect (sec)
ExecMemory > 2.5 GB		≥ 3		≥ 39	-150.9
ExecMemory > 5.5 GB		< 3		≥ 46	-21.1
ExecMemory > 8.5 GB		< 3		≥ 39∧ <46	-27.8
ExecMemory > 2.5 GB		≥ 3∧ < 6	Java	<39 ∧ ≥ 13	-234
ExecMemory > 3 GB		≥ 3∧ < 6	Java	<13	-123
ExecMemory > 2.5 GB		≥ 6	Java	<39	-236.5
ExecMemory > 2.5 GB		≥ 3∧ < 6	Kyro	<39	-163.5
ExecMemory > 2.5 GB		≥ 6	Kyro	≥ 22∧ < 22	-86.8
ExecMemory > 4.5 GB		< 3		< 24	-35.2
ExecMemory > 5.5 GB		≥ 3	Kyro	< 22	+0.198
ExecMemory > 2.5 GB		< 3		≥ 24∧ < 39	+12

Table 1: User-level look-up table showing the effect of the executor memory based on other configurations

3.1.2 *Executor Cores:*

The rules for the executor cores as the treatment are provided in Table2, and the causal tree is provided in Figure 5 in Appendix A.1 section.

Treatment	taskCPUs	ExecMemory	Serializer	Parallelism	Effect (sec)
ExeCores > 3.5		≥ 2∧ < 3		≥ 19∧ < 39	-12
ExeCores > 5.5		≥ 2∧ < 3		≥ 44	-19
ExeCores > 2.5	1	≥ 3		≥ 44	-29
ExeCores > 3.5	2	≥ 3∧ < 4			-22
ExeCores > 5.5	2	≥ 4∧ < 11		< 50	-29
ExeCores > 5.5	2	≥ 4∧ < 11		≥ 50	-43
ExeCores > 5.5	2	≥ 11		≥ 12∧ < 16	-31
ExeCores > 3.5	2	≥ 11		≥ 16∧ < 29	-47
ExeCores > 3.5	2	≥ 11		≥ 29	-25
ExeCores > 3.5		<2		< 19	+62
ExeCores > 4.5		≥ 2∧ < 3		< 19	+82
ExeCores > 2.5		<2		≥ 19∧ < 39	+182
ExeCores > 5.5		<3		≥ 39∧ < 44	+192
ExeCores > 2.5		<2		≥ 44	+233
ExeCores > 4.5	2	≥ 11		<12	+4

Table 2: User-level look-up table showing the effect of executor cores based on other configurations

3.1.3 *Parallelism:*

Based on the resulting tree, we got a single node tree, so we found that the effect of parallelism is not heterogeneous. The causal tree is provided in Figure 6 in Appendix A.1 section. It seems like setting the parallelism to a value above 10 always decreases the execution time by 26 seconds.

3.1.4 *Task CPUs:*

We got task CPUs as a binary configuration in our dataset, which can have a value of 1 or 2. Based on the resulting tree, we got a single node tree, so the effect of the number of task CPUs (1 or 2) is not heterogeneous. The causal tree is provided in Figure 7 in Appendix A.1 section. Apparently setting the task CPUs to 2 always increases the execution time by 10.5 seconds. (compared to having 1 task CPU)

3.1.5 *Serializer:*

The method resulted in a single node tree for this treatment, so we found that the effect of the type of serializer is also not heterogeneous. The causal tree is provided in Figure 8 in Appendix A.1 section. The Java Serializer always results in a higher execution time (by 5.5 seconds) compared to the Kyro Serializer.

3.2 Admin-level Analysis

In the admin-level analysis, the goal is to divide the subgroups based on the runtime metrics besides other configurations. The user cannot use these look-up tables as she won't have access to the runtime metrics, but the generated look-up table would be helpful for the admins in order to analyze the execution times in different scenarios.

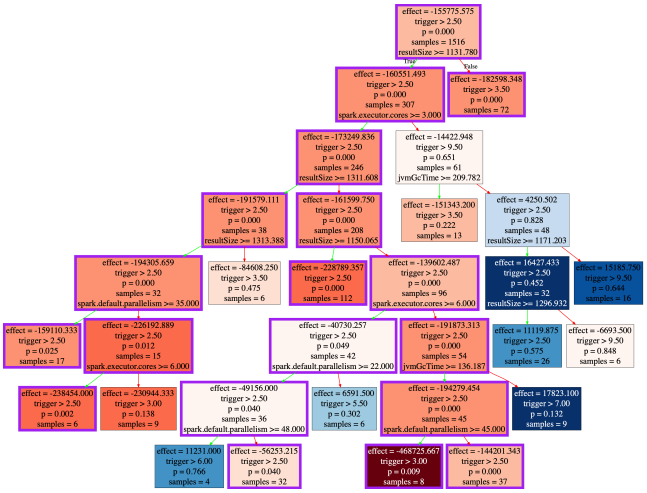

Figure 3: A tree illustrating the effect of the executor memory as the treatment on the execution time based on the runtime metrics besides other configurations as the covariates (Admin-level).

Figure3 illustrates the effect of the memory on the execution time, with the subgroups partitioned by the runtime metrics besides other configurations. Table3 shows the corresponding look-up table.

Treatment	taskCPUs	ExeCores	Serializer	Parallelism	ResultSize	JvmGCTime	Effect (sec)
Mem > 3.5					< 1131.7		-182.5
Mem > 2.5		≥ 3		≥ 35	≥ 1313.3		-159.1
Mem > 2.5		≥ 6		<35	≥ 1313.3		-238.4
Mem > 3		≥ 3∧ < 6		<35	≥ 1313.3		-230.9
Mem > 3.5		≥ 3			≥ 1311.6∧ <1313.3		-84.6
Mem > 2.5		≥ 3			≥ 1150∧ <1311.6		-228.7
Mem > 3.5		< 3			≥ 1131.7	≥ 209.7	-151.3
Mem > 2.5		≥ 6		≥ 22∧ <48	≥ 1131.7∧ <1150		-56.2
Mem > 9.5		< 3			≥ 1171.2∧ <1296.9	< 209.7	-6.6
Mem > 3		≥ 3∧ <6		≥ 45	≥ 1131.7∧ < 1150	< ≥ 136.1	-468.7
Mem > 2.5		≥ 3∧ <6		< 45	≥ 1131.7∧ <1150	≥ 136.1	-144.2
Mem > 6		≥ 6		≥ 48	≥ 1131.7∧ <1150		+11.2
Mem > 5.5		≥ 6		< 22	≥ 1131.7∧ <1150		+6.5
Mem > 9.5		< 6		< 22	≥ 1131.7∧ <1171.2	<209.7	+15.1
Mem > 2.5		< 3		< 22	≥ 1296.9	<209.7	+11.1
Mem > 7		≥ 3∧ <6		< 22	≥ 1131.7∧ <1150	<136.1	+17.8

Table 3: Admin-level look-up table showing the effect of the executor memory based on runtime metrics besides other configurations

The remaining look-up tables and their corresponding trees for other configurations are provided in Appendix A.2.

4 CONSISTENCY ANALYSIS

As discussed in the experiment results section, we have extracted decision rules from trees for different treatments. In this section, we want to check whether the estimation of the treatment effect given by these trees is consistent with the behavior of the real system (Spark running the K-means algorithm). We do this by simulating the condition of a rule (values of covariates) and then assigning values to the treatment configuration representing the treated and control group. After running the application with these configurations, we compare the execution time of the treated and control group.

To explain our consistency experiment in more detail, we investigate one rule from each configuration's user-level look-up table. After choosing the rule, we set values of the covariates (other configurations) to match the sub-population described by that rule. Also, for other covariates that haven't been mentioned in the rule, we set them to an arbitrary value. To sample from the control and the treated groups, we choose one value per group. For the control group, a value less than the trigger specified by the rule is chosen for the configuration. And for the treated group, a value more than the trigger is chosen for the treatment. Finally, we run the application with these two sets of configurations and measure the execution time.

To select the rule to investigate from each treatment tree, we choose the rule corresponding to a leaf node with a p-value close to zero. The reason is that the p-value close to zero specifies that the CATE estimation in this leaf node is statistically significant and reliable. This p-value implies whether the assumption made by the trigger method (the sub-population that has received treatment above the threshold represents the same underlying distribution as the subpopulation below the threshold) holds or not.

Furthermore, to account for randomness in Spark execution, we run each experiment with a specific set of configurations 5 times and report the average execution time. In the rest of this section, we will discuss the results.

4.1 Executor Cores Experiment

We consider the following rule for this configuration.

Rule: If

3 GB <= Executor Memory, task.cpus = 1,

having Executor Cores > 2.5 decreases execution time by 29 seconds.

Treatment	ExecMemory	Parallelism	Serializer	TaskCPUs	Avg. ExeTime
ExecCores = 6	4	27	Java	1	36.5
ExecCores = 2	4	27	Java	1	59

We can observe that the rule is consistent with the system's behavior and by increasing the executor cores, the execution time has been decreased (by 22.5 seconds).

4.2 Serializer Experiment

Since the treatment effect is not heterogeneous, we consider the only rule the tree gives us.

Rule: Having KyroSerializer decreases execution time by 5.5 seconds.

Treatment	ExecCores	ExecMemory	Parallelism	TaskCPUs	Avg. ExeTime
Serializer = Java	2	4	27	1	59
Serializer = Kryo	2	4	27	1	52.75

Based on Spark documentation, the ground truth is that KryoSerializer is a faster and more optimized serializer in comparison to JavaSeriailizer. The rule extracted from the tree is also consistent with this behavior and using KryoSerializer, reduces the execution time (by 6.25 seconds).

4.3 Executor Memory Experiment

We consider the following rule for this configuration.
Rule: If

$$13 \geq \text{parallelism} < 39,\ 3 \geq \text{executor cores} < 6,\ \text{Serializer} = \text{JavaSerializer},$$

having Executer Memory > 2.5 GB decreases execution time by 234 seconds.

Treatment	ExecCores	Parallelism	Serializer	TaskCPUs	Avg. ExeTime
ExecMem = 4	5	20	Java	1	38.5
ExecMem = 2	5	20	Java	1	84.25

While the impact of increasing executor memory is not as significant as mentioned in the rule, we can observe that the rule is consistent with the system's behavior and by increasing the executor memory, the execution time has been reduced by 45.75 seconds.

4.4 Parallelism Experiment

Since the treatment effect is not heterogeneous, we consider the only rule the tree gives us.
Rule: Having parallelism > 10.5 decreases execution time by 26 seconds.

Treatment	ExecCores	ExecMem	Serializer	TaskCPUs	Avg. ExeTime
Parallelism = 20	5	4	Java	1	38.5
Parallelism = 8	5	4	Java	1	52.75

This experiment also verifies the rule extracted from the tree considering parallelism as the treatment and it shows that with parallelism higher than the trigger we could reduce the execution time (by 14.25 seconds).

4.5 Task CPUs Experiment

Since the treatment effect is not heterogeneous, we consider the only rule the tree gives us.

Rule: Having Task CPUs = 2 increases execution time by 10.5 seconds compared to when Task CPUs = 1.

Treatment	ExecCores	ExecMem	Serializer	Parallelism	Avg. ExeTime
Task CPUs = 1	5	4	Kyro	8	40.75
Task CPUs = 2	5	4	Kyro	8	50.75

The system behaved based on the rule, as setting task CPUs to 2 increased the execution time by 10 seconds compared to having a similar system with task CPUs set to 1. The intuition behind this behavior is that since the number of cores available in the executor is limited if we need to assign more cores to a task, we would eventually have fewer tasks running in parallel which causes an increase in the execution time.

5 ROBUSTNESS ANALYSIS

As mentioned in the background, the Validation Honest trees improve generalizability using validation sets, and we believe making use of an estimation set improves the robustness of the tree. In this section, we want to empirically analyze the robustness of the Val-Honest trigger trees to see how different the resulting trees would have been if we train them on different random samples of our training set.

To perform the analysis, we use bootstrapping which is a random sampling technique with replacement. Using this approach, we generate different datasets of the same size as our training dataset. Later, we fit our Val-Honest trees on these sampled datasets and compare the resulting trees. Ideally, the trees resulting from these datasets should be similar to prove the robustness of the approach.

In the rest of this section, we explain the result of our robustness experiment when considering the user-level setup of executor cores as the treatment.

First, we compare the features used in the trees. We observed the trees have either used 3 or 4 features. Some have used only executor memory, task CPUs, and parallelism, while the rest use these three features in addition to the serializer. Although the used features sets are not completely similar, one of them is the subset of the other and the sets are not disjoint. As a result, we cannot reject the robustness of the approach, however, we also cannot conclude that this approach is completely robust. As an example of these trees, Fig.5 which is the tree trained on the whole training dataset, uses 3 features while Fig.4 trained on a bootstrap sample of the training dataset uses 4 features.

To investigate further, we considered the sign of the estimated CATE for similar cases in two different trees. In order to do this, we traversed the equivalent paths in two different trees based on the set of conditions on the covariates. We observed that while in most cases the sign of CATEs is similar, there exist some paths that result in inconsistent signs, one

implying a positive treatment effect and the other implying a negative treatment effect. We mention one such example path below. In the table below, the first row corresponds to the tree trained on the whole training dataset (Fig.5), and the second row corresponds to a tree trained on a bootstrap sample of the dataset shown in (Fig.4). As you can observe, the second tree was not able to detect the sub-population having a positive treatment effect like the first tree.

Treatment	ExecMemory	Parallelism	Serializer	TaskCPUs	Effect (sec)
ExecCores > 4.5	≥ 11	< 12		2	+4
ExecCores > 3.5	≥ 2			2	−27

Overall, based on the results obtained, while we could find cases where the tree was not entirely robust, we could not reject the robustness of their approach.

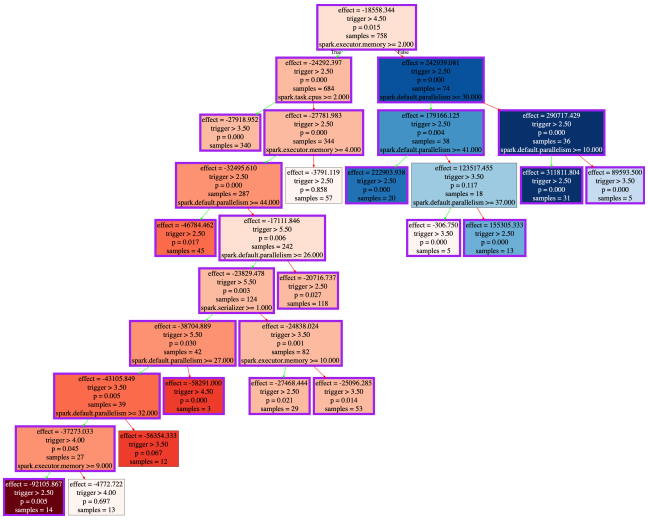

Figure 4: User-level tree for executor cores as the treatment trained on a bootstrap sample of dataset

6 CONCLUSION

In this project, we used Validation Honest Trigger-based causal trees to find the heterogeneous effect of Spark configuration parameters on the application execution time. Using these causal trees, we generated some guidelines to help with tuning configuration parameters in the format of look-up tables in two levels of observability: user-level and admin-level. Later, we investigated the consistency of the generated look-up tables with the behavior of the system. Finally, we did experiments to analyze the robustness of this causal tree approach and reported our findings.

REFERENCES

[1] https://github.com/edgeslab/ctl.

[2] Tran C. and Zheleva E. Learning triggers for heterogeneous treatment effects. *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence*, 2019.

[3] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *2010 IEEE 26th International conference on data engineering workshops (ICDEW 2010)*, pages 41–51. IEEE, 2010.

[4] Athey S. and Imbens G. W. Recursive partitioning for heterogeneous causal effects. 2016.

[5] Guido Imbens Susan Athey. Machine learning methods for estimating heterogeneous causal effects. 2015.

[6] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

A APPENDIX

A.1 User-level Trees

The figures below show the user-level trees, illustrating the effect of each configuration on the execution time.

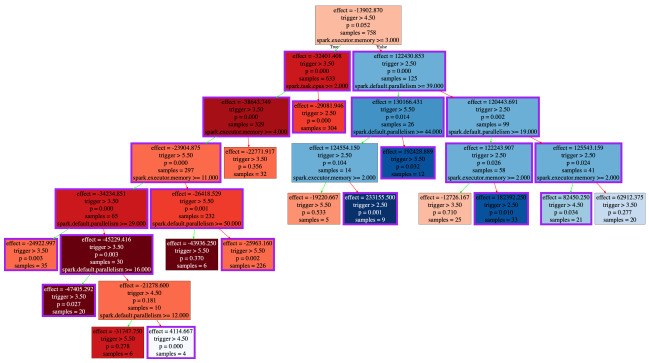

Figure 5: User-level tree for executor cores as the treatment.

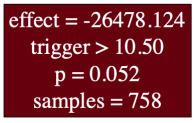

Figure 6: User-level tree for Parallelism as the treatment. (Not heterogeneous effect)

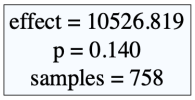

Figure 7: User-level tree for Task CPUs (binary in our dataset, treatment if CPU = 2, and control if CPU = 1) as the treatment. (Not heterogeneous effect)

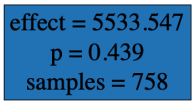

Figure 8: User-level tree for Serializer (binary in our dataset, treatment if JavaSerializer, and control if KyroSerializer is used) as the treatment. (Not heterogeneous effect)

Treatment	Memory	ExeCores	Serializer	Parallelism	ResultSize	DeserializationCPUTime (sec)	Effect (sec)
CPU = 2				< 16	< 1314.997	< 0.131165	-0.7
CPU = 2	≥ 8	<3		≥ 16	< 1314.997	≥ 0.044283∧ <0.131165	-1.9
CPU = 2	≥ 8	≥ 3		≥ 16	<1127.885	≥ 0.044283∧ <0.131165	-72
CPU = 2	≥ 8	≥ 3		≥ 16	≥ 1127.885∧ <1129.84	≥ 0.044283∧ <0.131165	-11
CPU = 2					≥ 1314.9977		+88
CPU = 2					< 1314.9977	≥ 0.131165	+129
CPU = 2	<8			≥ 16	< 1314.9977	< 0.131165	+18
CPU = 2	≥ 8			≥ 16	< 1314.9977	< 0.044283	+17
CPU = 2	≥ 12	≥ 3		≥ 16	≥ 1130.7∧ <1314.997	≥ 0.044283∧ <0.131165	+21.6
CPU = 2	≥ 8∧ < 12	≥ 3		≥ 16	≥ 1130.7∧ <1314.997	≥ 0.044283∧ <0.131165	+15.8
CPU = 2	≥ 8	≥ 3		≥ 16	≥ 1129.8∧ <1130.7	≥ 0.044283∧ <0.131165	+7.1

Table 4: Admin-level look-up table showing the effect of Task CPUs based on runtime metrics besides other configurations (In our dataset, this configuration is binary, CPU = 2 is assumed to be the treatment, and CPU = 1, as a control).

A.2 Admin-level Experiments

The admin-level experiments, including both the tree and their corresponding look-up table, for the remaining configurations, are provided in this section.

Task CPUs:

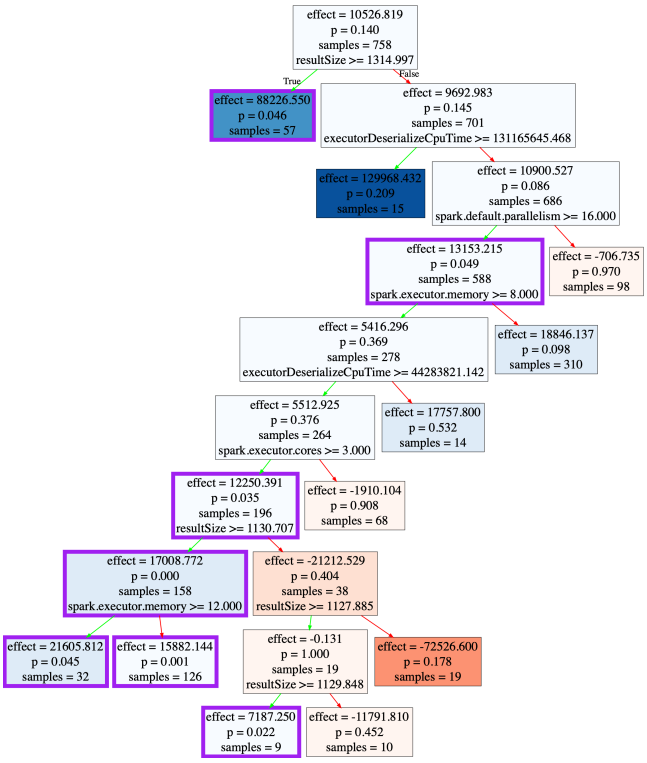

Figure 9: Admin-level tree illustrating the effect of Task CPUs as the treatment on the execution time.

Serializer:

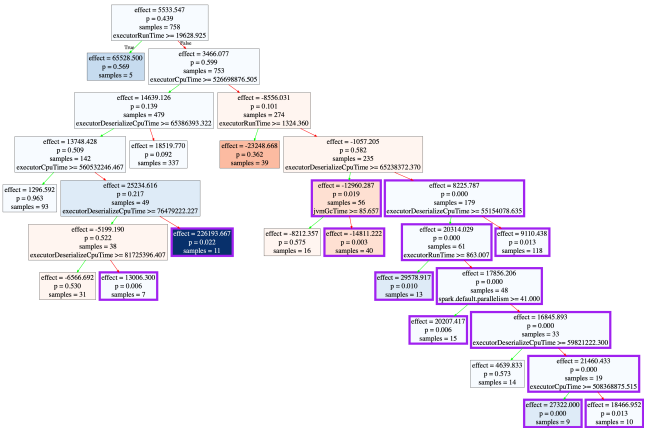

Figure 10: Admin-level tree illustrating the effect of Serializer as the treatment on the execution time.

Treatment	Memory	ExeCores	Parallelism	ResultSize	DeserializationCPUTime	executorRunTime	executorCpuTime	jvmGCtime	Effect (sec)
Serializer = 1					≥ 0.081	< 19.6	≥ 0.52 ∧ < 0.56		-6.5
Serializer = 1					≥ 0.065	< 1.34	< 0.52	≥ 0.085	-8.2
Serializer = 1					≥ 0.065	< 1.34	< 0.52	< 0.085	-14.8
Serializer = 1					< 0.065	1.34 ≥ ∧ < 19.6	< 0.52		-23.24
Serializer = 1					≥ 0.065	< 19.6	≥ 0.56		+1.29
Serializer = 1			< 41		< ≥ 0.059 ∧ < 0.065	< 0.863	< 0.52		+4.6
Serializer = 1					< 0.055	< 1.34	< 0.52		+9.11
Serializer = 1					≥ 0.065 ∧ < 0.081	< 19.6	≥ 0.52 ∧ < 0.56		+13.0
Serializer = 1			< 41		< ≥ 0.055 ∧ < 0.059	< 0.863	< 0.50		+18.46
Serializer = 1					< 0.065	< 19.6	≥ 0.52		+18.5
Serializer = 1			≥ 41		< ≥ 0.055 ∧ < 0.065	< 0.863	< 0.52		+20.2
Serializer = 1			< 41		< ≥ 0.055 ∧ < 0.059	< 0.863	≥ 0.50 ∧ < 0.52		+27.32
Serializer = 1					≥ 0.055 ∧ < 0.065	≥ 0.863 ∧ < 1.34	< 0.52		+29.578
Serializer = 1						≥ 19.6			+65.5
Serializer = 1					≥ 0.065 ∧ < 0.076	< 19.6	≥ 0.52 ∧ < 0.56		+226.19

Table 5: Admin-level look-up table showing the effect of Serializer based on runtime metrics besides other configurations (In our dataset, this configuration is binary, Serializer = 1 (Kyro/java?) is assumed to be the treatment, and Serializer = 0 (Kyro/java?), as a control).

Parallelism:

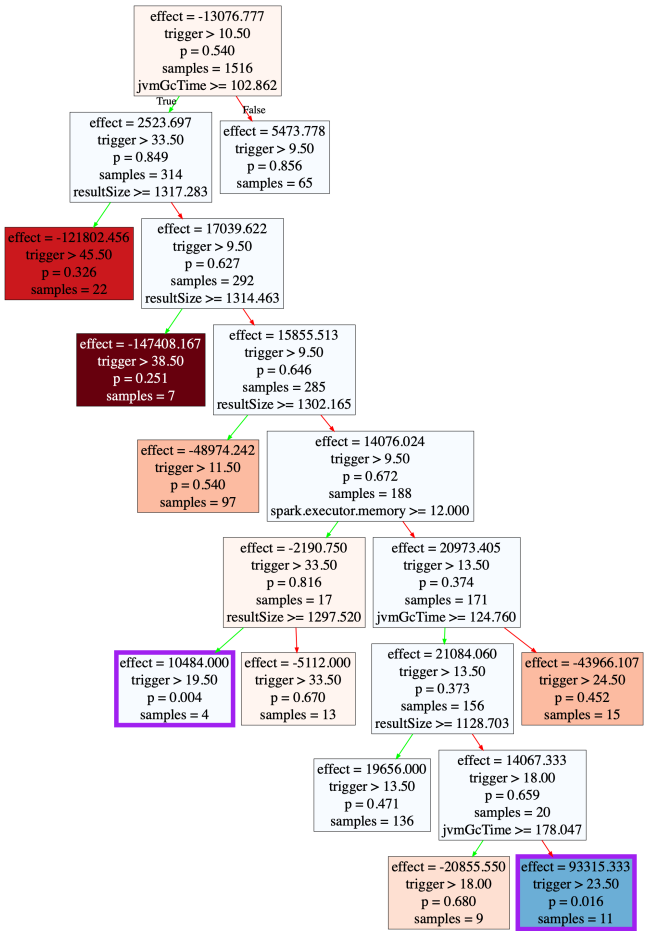

Figure 11: Admin-level tree illustrating the effect of Parallelism as the treatment on the execution time.

Treatment	ExecMemory	ResultSize	jvmGCtime	Effect (sec)
Parallelism > 33.5	≥ 12	< 1297.52	≥ 102.86	-5.11
Parallelism > 18	< 12	< 1128.7	≥ 178.04	-20.85
Parallelism > 24.5	< 12	< 1302.16	≥ 102.86 ∧ < 124.76	-43.9
Parallelism > 11.5		≥ 1302.16 ∧ < 1314.46	≥ 102.86	-48.97
Parallelism > 45.5		≥ 1317.2	≥ 102.86	-121.8
Parallelism > 38.5		≥ 1314.4 ∧ < 1317.2	≥ 102.86	-147.4
Parallelism > 9.5			< 102.86	+5.47
Parallelism > 19.5	≥ 12	≥ 1297.52 ∧ < 1302.16	≥ 102.86	+10.48
Parallelism > 13.5	< 12	≥ 1128.7 ∧ < 1302.16	≥ 124.76	+19.56
Parallelism > 23.5	< 12	< 1128.7	≥ 124.76 ∧ < 178.04	+93.31

Table 6: Admin-level look-up table showing the effect of Parallelism based on runtime metrics besides other configurations (In our dataset, this configuration is continuous with values ranging between 8 and 50)