Discovering Process-Based Drivers for Case-Level Outcome Explanation

Peng Li[1*], Hantian Zhang[1⋆], Xu Chu[2], Alexander Seeliger[2], and Cong Yu[2]

[1] Georgia Tech, Atlanta, USA
{pengli, hantian.zhang}@gatech.edu
[2] Celonis, New York, USA
{x.chu, a.seeliger, c.yu}@celonis.com

Abstract. Process mining has shown great impact in improving business Key Performance Indicators (KPIs), which are typically measured as aggregations over case-level outcomes. A commonly encountered key question in achieving such impact is understanding the underlying reasons for why a certain outcome appears in some cases (e.g., why certain cases take long to finish). We use the term *drivers* to refer to explanations for case-level outcomes. We hypothesize that how process is run, in other words, process traces, directly influences case-level outcomes, and hence KPIs. In this paper, we propose a new method to automatically and efficiently discover process-based drivers that are effective, significant and interpretable. We formally define the problem of driver discovery as a constrained optimization problem. Given that the problem is NP-hard, we develop efficient greedy algorithms to solve the problem. We evaluate our method on real-world datasets to demonstrate the effectiveness and efficiency of our approach

Keywords: Process Mining · Explanation · Case outcome.

1 Introduction

Process mining is a discipline that aims to discover, monitor, and improve real-world processes. The growing interest in this discipline can be attributed to the increasing availability of recorded process execution data in businesses, as well as the strong desire to improve business outcomes, measured in various Key Performance Indicators (KPIs) [3]. A KPI for process operation is typically measured by taking a historical event log and calculating an aggregation over case-level *outcomes*. For example, on-time delivery rate as a KPI is calculated as the percentage of cases that have an outcome of being delivered on time; and average throughput time is calculated as the average time it takes to complete the execution of cases. Using process mining to discover process inefficiencies, execution gaps have shown significantly impact in terms of improving business KPIs.

⋆ equal contribution, work done in Celonis.

In our experience of delivering process mining solutions to customers for improving their KPIs, there are some important and common questions that are frequently encountered: *Why is a certain KPI so low (or high)? What causes certain cases to have a negative outcome, and hence affecting the overall KPIs?* To answer these real-world questions, we formally define and address the problem of *Driver Discovery* in this paper. *Drivers* refer to the explanations that are most likely to drive a particular case-level outcome, and hence have a direct impact on KPIs. In particular, we want to make use of the event log and discover process-based drivers that contain information about the events or activities in event log traces. The motivation is that process data which describes the execution trace of a case should reveal the fundamental root cause of case-level outcomes. For example, certain cases experiencing long throughput times can be attributed to the involvement of a particular manual activity in their event log traces.

For discovered drivers to be practically useful, they have to be *effective*, *significant*, and *interpretable*. Intuitively, a driver is effective if the probability of observing a certain outcome is highly likely given the driver. A driver should also be significant in that it should cover as many occurrences of a certain outcome as possible. In addition, a driver needs be interpretable so that users can understand the driver, and can then take corresponding actions.

This paper investigates methods to discover drivers on process data that meet those requirements. Concretely, the paper makes the following contributions:

- We formally define the problem of process-based driver discovery as a constrained optimization problem, and propose a discovery workflow (Section 2).
- We propose three categories of process-based features as well as efficient methods to enumerate and prune such features (Section 3).
- Given that the driver discovery problem is NP-hard, we develop an efficient algorithm based on beam search to solve the problem (Section 4).
- We evaluate our method on real-world datasets. The experiment results show the effectiveness and efficiency of our method (Section 5).

2 Preliminary

2.1 Input Data and Discovery Workflow

Data Model. We assume a standard case-centric event log data model. In particular, we assume as input a *case table* and an *event log* table. The case table has columns $\{case\_id, X_1, X_1, X_2, ...X_m, Y\}$. $\{X_1, X_2, ...X_m\}$ are case-level attributes, Y is the case-level outcome. We assume in this paper that the target variable Y is a binary outcome variable, while the attributes X_i can be either numerical or categorical. The event log table has three columns $\{case\_id, activity\_name, time\_stamp\}$.

Discovery Workflow. In process mining, the event log records the execution traces of individual cases. The execution trace of a case contains valuable source information and signals that might explain the outcome of a case. How do we leverage the execution traces of cases, together with case-level attributes? Figure 1 shows the overall workflow: we first flatten the event table by generating multiple process-based features $\{X_{m+1}, X_{m+2}, ...\}$ for each case, which are then joined with the case table on the $\{case\_id\}$ column. For example, for every unique activity A in the event log, we will create a process-based feature X_A to indicate the number of times the activity A appears in each case.

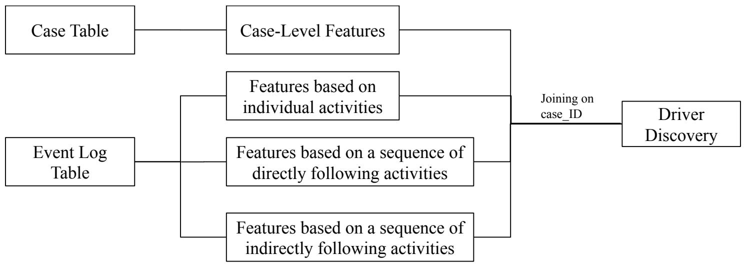
Case Table
Case-Level Features
Features based on individual activities
Event Log Table
Features based on a sequence of directly following activities
Features based on a sequence of indirectly following activities
Joining on case_ID
Driver Discovery

Fig. 1. Driver Discovery Workflow.

2.2 KPI Driver Formal Definition and Goodness

Given a binary outcome $Y = y \in \{0, 1\}$ to explain, we formally define a driver d as a conjunction of constraints on a subset of attribute values:

$$d := (X_1 \odot v_1) \wedge (X_2 \odot v_2) ... \wedge (X_k \odot v_k) \tag{1}$$

where $X_j \in \boldsymbol{X}$ is a feature that could be a case-level feature or a flattened process-based feature, x_j is a feature value and $\odot \in \{=, \neq, <, >, \leq, \geq\}$ is a comparison operator.

Given that there are exponential number of driver with respect to the number of features considered, we define the following goodness metrics for drivers based on various conversations with customers. Let $D = \{t_i | i \in [n]\}$ denote the joined dataset (c.f. Figure 1) with n cases, where t_i is the i^{th} case. Let $D_d \subseteq D$ denote the subset of cases that fulfill the attribute constraints specified in the driver d:

- **Effectiveness (high precision).** Given a particular outcome $Y = y$, good drivers should effectively drive that outcome rather than preventing the occurrence of that outcome. Therefore, a good driver d should be *effective*, or has high precision, if for cases that satisfy the driver conditions, the probability of the occurrence of the given outcome is high. Formally, this is written as $P_d(D, y) = \frac{\sum_{t_i \in D_d} \mathbb{1}(t_i[Y]=y)}{|D_d|}$.
- **Significance (high recall).** Good drivers are supposed to cover as many cases with the interested outcome as possible. This is especially important if the driver is used to find the underlying reasons for some issues, such as identifying the reasons for low KPI. If the driver only covers a small number (e.g., 1%) of low-KPI cases, it cannot be used to significantly improve the KPI even if the driver is effective. Hence, a good driver should be *significant*, or has high recall, if it covers as many occurrences of a given outcome as possible. This property can be formally written as $R_d = \frac{\sum_{t_i \in D_d} \mathbb{1}(t_i[Y]=y)}{\sum_{t_i \in D} \mathbb{1}(t_i[Y]=y)}$.

– **Interpretability.** Good drivers should be easily interpretable, and hence users can take actions to fix the problem. Intuitively, the simpler the driver is, the easier it can be understood. Let $|d|$ denote the number of attributes in the driver. Then, we use $|d|$ as the proxy metric for interpretabilty.

Our goal is to find a good driver that is *effective*, *significant* and easily *interpretable*. However, it may not be possible to optimize all three properties simultaneously. To balance the trade-off between effectiveness and significance, inspired by using F-score [11] to balance precision-recall trade-off, we combine the effectiveness and significance into one metric denoted by F_d: $F_d(D, y) = \frac{2P_dR_d}{P_d+R_d}$. This metric will be high as both effectiveness and significance are high. Either a low effectiveness or low significance will result in a low score. Therefore, optimizing F_d will yield a driver with a good balance on effectiveness and significance. For interpretability, in practice, if the number of attributes involved in the driver is not too large (e.g., $|d| \leq 3$), its interpretability is generally acceptable. Therefore, we only need to ensure $|d|$ under a threshold θ (e.g., $\theta = 3$) to make it easily interpretable rather than optimize it.

Definition 1 (Driver Discovery Problem). *Given a dataset D and an outcome y, we would like to select a driver d that maximizes the metric F_d, while keeping the number of constraints (denoted by $|d|$) smaller than θ. This can be written as a constrained optimization problem as follows.*

$$\arg\max_{d} \quad F_d(D, y) \quad s.t. \ |d| \leq \theta$$

Problem Complexity. A naive way to solve this constrained optimization problem is to enumerate all drivers with $|d| \leq \theta$ and return the one with the highest score. For datasets with categorical attributes only, the number of possible drivers with at most θ constraints is $O((mc)^{\theta})$, where m is the number of attributes, c is the number of possible attribute values. Consider $m = 100$ and $c = 10$. Even for $\theta = 3$, there would be 1 billion possible drivers, let alone that there are infinite drivers for datasets with numerical attributes. Therefore, this naive approach is not feasible in practice. In fact, we can prove the problem is NP-hard (formal proof omitted due to space).

3 Process-Based Feature Engineering

We include three categories of process-based features: (1) features based on individual activities; (2) features based on a sequence of activities that directly follow one another; and (3) features based on a sequence of activities that indirectly follows one another.

Features based on individual activities. Let $\mathcal{A}$ denote all unique activities. We will create a feature X_A for each unique activity $A \in \mathcal{A}$ that appears in the event log of all cases. For each case t_i, the feature value $t_i[A]$ is the number of times the activity A appears in the trace of the case t_i. The complexity for generating these features is $O(|\mathcal{A}|)$.

Features based on a sequence of directly following activities. A feature in this category consists of a sequence of directly following activities, such as $X^D_{\{A_1,A_2\}}$ or $X^D_{\{A_2,A_4,A_5,A_1\}}$, where the superscript D means directly following and the subscript is the activity sequence of the feature. For each case t_i, a directly following sequence-based feature value is 1 if the feature's sequence appears at least once as a sub-sequence in the case's event log, and 0 otherwise.

We have two potential ways to generate features in this category. First, we could enumerate all possible sequences using $\mathcal{A}$, which is exponential $O(2^{|\mathcal{A}|})$. However, a feature is only useful if at least one case's feature value is 1. Therefore, we can alternatively generate candidate features by enumerating all cases. For each case t_i, we enumerate the start index s and the end index e of the case's trace, and the sub-sequence between s and e would be a candidate feature. The number of features generated this way is $O(L^2 * n)$, where n is the number of cases and L is the maximum trace length of any case. We first compare the two numbers $2^{|\mathcal{A}|}$ and $L^2 * n$, and pick the enumeration method with the lower complexity. In practice, we took the second enumeration option for all datasets we experimented with. We then prune features that do not meet the minimum precision and recall thresholds.

Features based on a sequence of indirectly following activities. A feature here consists of a sequence of indirectly following activities, such as $X^I_{\{A_1,A_2\}}$ or $X^I_{\{A_2,A_4,A_5,A_1\}}$, where the superscript I means indirectly following and the subscript is the activity sequence of the feature. For each case t_i, in indirectly following sequence-based feature value is 1 if the feature's sequence's activities appear in the right order at least once in the case's trace, and 0 otherwise.

Example 1. Let us consider two features $X^D_{\{A_1,A_2\}}$ and $X^I_{\{A_1,A_2\}}$. For a case t_i with trace $\{A1, A4, A2, A5\}$, $t_i[X^D_{\{A_1,A_2\}}] = 0$ and $t_i[X^I_{\{A_1,A_2\}}] = 1$.

For this category of features, we will have to enumerate all possible sequences using $\mathcal{A}$, which is exponential $O(2^{|\mathcal{A}|})$. This is because, the enumeration method starting from cases would have a similar, and sometimes even higher given big n, exponential complexity of $O(n \times 2^{|L|})$.

We followed the famous *Apriori* algorithm [1] for enumerating and pruning features in this category using a lattice data structure. For the i^{th} level of the lattice, we generate and prune all candidate sequences of length i. The candidates in level i are generated using frequent candidates in the previous level $i-1$. Specifically, a candidate of length i can only be frequent if all of its sub-sequences of length $i-1$ are frequent. For example, the indirectly following sequence A_1, A_2, A_3 can only be frequent if A_1, A_2, A_1, A_3, and A_2, A_3 are frequent. Given that we only want to retain candidate features that are above a certain recall threshold θ_r, the minimum support, i.e., number of cases where the feature is 1, for a candidate to be frequent is $|\mathbb{1}(t_i[Y] = y)| \times \theta_r$. Concretely, we generate the indirectly following sequence features in the following steps:

1. Generate $|\mathcal{A}|$ length 1 candidate using $\mathcal{A}$. Calculate the support for them and retain only those with support bigger than or equal to $|\mathbb{1}(t_i[Y] = y)| \times \theta_r$.
2. Generate length i candidate using the frequent length $i-1$ candidates. Note that a length i candidate is generated only if all its sub-sequences of length $i-1$ are frequent.

3. Calculate the support for length i candidates, and retain only the frequent ones. Iterate step (2) and step (3) until we have no more frequent candidates.
4. Calculate the precision of all frequent candidates from all levels and retain those meeting the precision threshold θ_p.

Feature Selection. Given the exponential number of sequence-based features, we perform feature selection based on their effectiveness (precision) and significance (recall). Our directly follow and indirectly follow features are binary features. We define the precision and recall of a binary feature X with respect to a target outcome y as follows:

$$P_X = \frac{\sum_{t_i \in D} \mathbb{1}(t_i[Y] = y) \ \& \ t_i[X] = 1}{\sum_{t_i \in D} t_i[X] = 1} \qquad R_X = \frac{\sum_{t_i \in D} \mathbb{1}(t_i[Y] = y) \ \& \ t_i[X] = 1}{\sum_{t_i \in D} \mathbb{1}(t_i[Y] = y)}$$

We keep a process-based sequence feature X if its precision and recall are greater than some thresholds $P_X \geq \theta_p$ and $P_X \geq \theta_r$. Note that we set θ_p and θ_r to low numbers ($\theta_p = 0.5$ and $\theta_r = 0.2$ in our experiments) so as not to lose any features might contribute to useful drivers when combined with other features.

4 Driver Discovery Algorithm

Given the hardness of the problem, this section introduces our greedy algorithm to solve the aforementioned issue. Specifically, we present an efficient greedy searching algorithm based on beam search.

A straightforward greedy algorithm starts from a driver with no constraint and iteratively adds one constraint with the highest benefit into the driver until the number of constraints in the driver exceeds the given limit. The benefit of a constraint can be computed as the F_d score after adding the constraint into the current driver. Note that it is possible that the F_d score decreases with any additional constraint. Therefore, we keep track of the driver with the highest F_d score during exploration and return it as the final result.

However, this greedy algorithm has a drawback: adding one optimal constraint at each step may not lead to the global optimal driver. For example, assume the global optimal driver is $X_1 = a \wedge X_2 = b$. However, for drivers with one constraint, the optimal driver is $X_3 = c$. Then the greedy algorithm will select $X_3 = c$ at the first step and it is no longer possible to reach global optimal driver by adding more constraints.

To increase the probability of finding the best driver, we use beam search [12], which is a heuristic search algorithm. Instead of only keeping and developing one best driver at each step, we keep the top K (e.g, K = 10) drivers with the highest F_d scores in a "beam". At each step, we will extend each driver in the beam with one additional constraint, and keep the top K resulting drivers for the next step. We will still keep track of the best driver that we have seen during beam search and return it as the final result. The term "beam search" refers to the way the algorithm explores the search space by considering a limited number of candidates at each step, forming a "beam" of possible solutions.

Algorithm. The pseudocode of the beam search algorithm is shown in Algorithm 1. We start from an empty driver (Line 1 - 2). At each iteration, we extend each driver in the beam by one additional constraint (Line 4 - 7). The candidate set of constraints can be generated by enumerating all possible combination of attributes, operators, and attributes values. For numerical attributes, we can choose c splitting points (e.g., 10-percentile, 20-percentile, etc) and for each splitting point v, we can generate two candidate constraints as $X \leq v$ and $X > v$. We compute the F_d score of each driver (Line 8), and we keep the top K drivers with the highest score for the next step (Line 12 - 13). This process will be repeated for θ iterations such that the number of constraints in the driver will not be greater than θ (Line 3). We keep track of the best driver d^* that we have seen (Line 10 - 11) and return it at the end (Line 14).

Algorithm 1: Beam Search Algorithm

```
   input   : Input dataset $D$, an outcome $y$, threshold $\theta$, beam width $K$
   output  : A driver $d$
1  $d^* \leftarrow d_{empty}$
2  $B_{cur} \leftarrow [d_{empty}]$
3  for $i$ = 1, 2, ... $\theta$ do
4      $B_{next} \leftarrow []$
5      foreach $d$ in $B_{cur}$ do
6          foreach $(X_j \odot v_j)$ in candidates do
7              $d' \leftarrow d \wedge (X_j \odot v_j)$
8              $d'.score = F_{d'}(D, y)$
9              $B_{next}.append(d_{next})$
10             if $d'.score > d^*.score$ then
11                 $d^* \leftarrow d'$
12     sort $B_{next}$ by $d.score$
13     $B_{cur} \leftarrow B_{next}[: K]$
14 return $d^*$
```

Complexity. Consider a dataset with n examples and m features, and each attribute can take c values (assuming numerical attributes are split into c intervals). The number of candidate constraint will be $O(mc)$. Let K be the beam size. For each iteration, we compute the F_d score $O(Kmc)$ times, which takes $O(nmc)$ time. Therefore, the total time complexity for our algorithm is $O(Knmc\theta)$.

5 Experiment

We evaluate the effectiveness and efficiency of both the process-based feature engineering and the beam search algorithm for driver discovery.

5.1 Experimental Setup

Datasets. We use three popular datasets from BPI Challenge, which are widely used in the literature, and a synthetic dataset to test the scalability of the driver discovery algorithm. The stats of the datasets are listed in Table 1. BPIC 2017 [7] contains 31,509 loan application cases of a Dutch financial institute; BPIC 2018 [9] contains 43,809 applications for EU direct payments for German farmers from the European Agricultural Guarantee Fund; and BPIC 2019 [8] contains 251,734 purchase orders for a dutch company. For all three real datasets, our target is a binary outcome that indicates whether each case is taking longer than 75 percentile throughput time of all cases. We use the synthetic dataset to evaluate the scalability of the driver discovery algorithm and the features are already engineered.

Table 1. Dataset Statistics (AAPC: average activities per case; DF: directly follows features; IDF: indirectly follows features)

Dataset	**#Cases**	**#Case Features**	**#Activities**	**AAPC**	**DF**	**IDF**
BPIC 2017	31,509	4	26	38.2	7	20
BPIC 2018	43,809	61	41	57.4	111	404
BPIC 2019	251,734	16	42	6.3	112	258
Synthetic	1,000,000	100	N/A	N/A	N/A	N/A

Methods Compared. We compare the following driver discovery methods. Note that, all methods use the same feature set generated by Section 3. We will evaluate the effectiveness of process-based feature engineering in Section 5.2 and compare driver discovery algorithms in Section 5.3.

- Beam Search (Beam): This is our proposed algorithm in Section 4. By default we set the maximum number of constraints in a driver θ to be 3, and we set the beam search width K to be 8.
- Decision Tree (DT): We run the decision tree algorithm to find the drivers. The decision tree is trained to classify the KPI. Then we down the tree from root to leaf, each path would be a driver and all the points in the leaf would be the cases that satisfies the given driver.
- Exhaustive Search (ES): This approach enumerates all possible drivers and find the best one with the highest score.

Evaluation Metrics. We use F1 score to measure the quality of the drivers, as defined in Section 2.2. We use running time to evaluate algorithm efficiency.

5.2 Evaluating Process-Based Feature Engineering

We evaluate the effectiveness of the process-based feature engineering described in Section 3. The number of each type of features is shown in Table 1. Specifically, we compare the following feature set in a cumulative manner:

- Feature Set A: case-level features only
- Feature Set B: features based on activity counts and features in A
- Feature Set C: directly following activities and all features in B
- Feature Set D: indirectly following activities and all features in C

Qualitative Assessment. We show the Top-1 driver discovered by each feature set in Table 2. For example, for the BPIC 2017 dataset, an important driver is "Count of W_Call after offers $>$ 4", the repetition is likely to be the cause of long throughput time, and is highlighted by the driver discovery algorithm. The directly following feature "A_Cancelled directly followed by O_Cancelled" and the indirectly following feature "A_Submitted indirectly followed by A_Cancelled" show that cancelled cases takes longer than normal cases, which is counter-intuitive. After we looked at the data more deeply, we noticed that this is caused by automatic cancellation, which takes 30 days and causes long throughput time. For BPIC 2018 dataset, we find that the indirectly following feature "finish payment indirectly followed by save" has a high relevance with long throughput time Overall, our feature generation algorithm can generate features related to the process and provide insightful patterns that are worth looking into more carefully.

Table 2. Top 1 driver with different set of features on three datasets.

Features	Top 1 Driver
BPIC 2017 A	"Application Type==New credit" & "Requested Amount > 0"
BPIC 2017 B	"Count of A_Cancelled > 0" & "Count of A_Submitted > 0" & "Count of W_Call after offers > 4"
BPIC 2017 C	"A_Cancelled directly followed by O_Cancelled" & "A_Concept directly followed by W_Complete application" & "Count of W_Call after offers > 4"
BPIC 2017 D	"A_Submitted indirectly followed by A_Cancelled" & "A_Cancelled directly followed by O_Cancelled" & "Count of W_Call after offers > 4"
BPIC 2018 A	"Year==2015" & "number parcels > 3" & "amount applied0 > 1135.24"
BPIC 2018 B	"Count of initialize > 4" & " Count of begin preparations <= 0 " & "Count of save > 0"
BPIC 2018 C	"Count of initialize > 4" & " amount applied0 > 4025.81 " & "finish payment directly followed by save"
BPIC 2018 D	"finish payment indirectly followed by save" & "case year != 2017"
BPIC 2019 A	"Item Category != Consignment" & "Item Type != Third-party"
BPIC 2019 B	"Count of Record Invoice Receipt > 0" & "Item Type != Third-party"
BPIC 2019 C	"Clear Invoice directly followed by Record Invoice Receipt" & "Item Type != Third-party" & "Count of Record Invoice Receipt>=2"
BPIC 2019 D	"Change Quantity indirectly followed by Record Goods Receipt" & "Item Type != Third-party" & "Count of Record Invoice Receipt>=2"

Table 3. F1 Score of the driver with feature set.

Dataset and Feature Set	Top 1	Top 5	Top 10
BPIC 2017 A	0.429	0.429	0.428
BPIC 2017 B	0.650	0.648	0.643
BPIC 2017 C	0.653	0.652	0.649
BPIC 2017 D	0.655	0.653	0.650
BPIC 2018 A	0.647	0.646	0.646
BPIC 2018 B	0.714	0.713	0.708
BPIC 2018 C	0.718	0.716	0.710
BPIC 2018 D	0.906	0.905	0.903
BPIC 2019 A	0.506	0.506	0.505
BPIC 2019 B	0.522	0.521	0.521
BPIC 2019 C	0.532	0.530	0.523
BPIC 2019 D	0.535	0.532	0.527

Table 4. F1 Score of the Top 1 driver with driver discovery algorithm, NA means that the algorithm cannot finish in 1 day.

Dataset	Beam	DT	ES
BPIC 2017 D	0.655	0.608	0.664
BPIC 2018 D	0.906	0.904	NA
BPIC 2019 D	0.535	0.493	NA

Quantitative Assessment. Table 3 shows the F1 score of top drivers found by our algorithm. Here top 5 is the average F1 score of top 5 drivers and Top 10 is the average F1 score of the top 10 drivers. We observe that adding the count features can already increase the F1 score significantly compared with only using case-level features, because we are focusing on long throughput time as the target and usually long throughput time is closely related to rework, which is captured by the count features. After adding the directly following and indirectly following features, the F1 score also increases, indicating that new features are used in the top drivers, revealing more causes that are related to the processes. Especially, we see that after adding the indirectly following features, the F1 score of the top-1 driver for the BPIC 2018 dataset (BPIC 2018 D) becomes 0.906, which shows that the driver has very high relevance with the long throughput time.

Feature Selection Knob Analysis. We evaluate the how tuning the precision and recall threshold would have an impact on the number of features generated, and the F1 score of the drivers found using the BPIC 2017 dataset. In Figure 2(a), we fix the recall threshold, $\theta_r = 0.2$ and tune the precision threshold θ_p, we see that as θ_p becomes higher, there are less features, but the impact on the top 1 F1 is pretty insignificant. Similarly, In Figure 2(b), we fix the precision threshold, $\theta_p = 0.5$ and tune the recall threshold θ_r, we see that the top 1 F1 does not change at all. The reason is that after the features are generated, we still need to run the driver discovery algorithm, which would select the high quality features. The selected features in the top drivers usually have high precision and recall, which means that the result is not sensitive on the knob for feature selection.

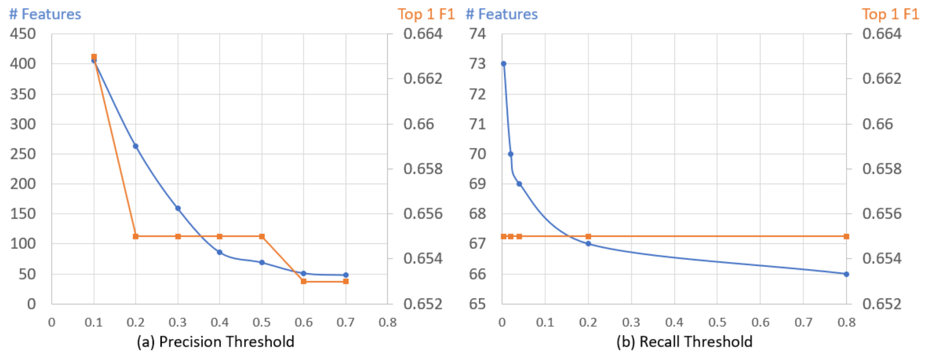

Fig. 2. Knob Analysis.

5.3 Evaluating the Driver Discovery Algorithm

Quality of the driver We evaluate the F1 score of the found driver using the three algorithms: beam search, decision tree (DT) and exhaustive search (ES), and the result is shown in Table 4. For exhaustive search, it takes a long time to run and cannot finish in 1 day for the BPIC 2018 and BPIC 2019 dataset. For BPIC 2017, exhaustive search achieved the highest F1 score at 0.664, which is understandable because it searches the entire search space. However, our beam search achieved a F1 score of 0.655, and the difference is less than one percentage point. Compared to the decision tree algorithms, our beam search can consistently outperform the decision tree algorithm significantly, by up to 4.7 percentage points.

Running Time Comparison. We use the synthetic dataset to verify the scalability of driver discovery algorithm. Figure 3 shows the running time comparison of different methods with varying size of datasets. As we can see, in Figure 3 (a), as the number of rows increases, all three grows linearly, and decision tree and beam search have similar running time, and they are 4x faster than exhaustive search. In Figure 3 (b), as the number of columns grows, the running of the exhaustive search method grows polynomial and takes over 1 hour to finish when the number of columns is greater than 100. However, both decision tree and beam search algorithm grow almost linearly and thus have a better scalability, and decision tree and beam search have similar running times.

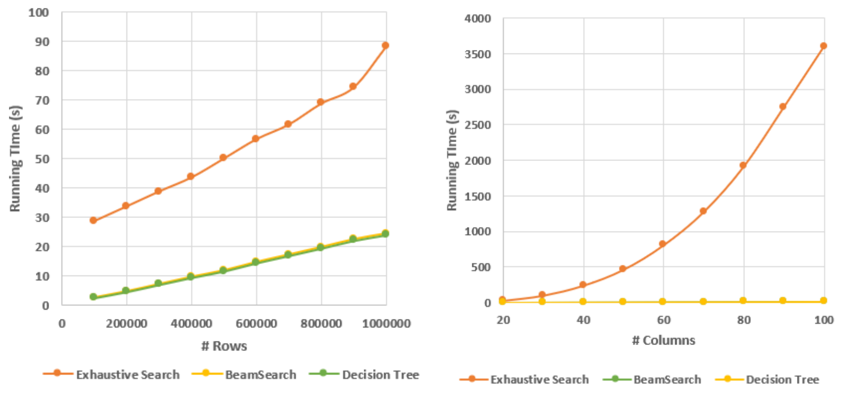

Fig. 3. Running Time Comparison (left: Varying # rows; right: Varying # columns

6 Related Work

Feature Importance. Our work is related to estimating feature importance in ML, which estimates the impact of each feature on model predictions [10]. Some interpretable ML models provides feature importance by themselves. For example, in logistic regression [6], the weight of a feature indicates its importance. In tree-based methods, such as decision tree [15] and random forest [4], the feature importance is computed as the reduction of the prediction error brought by that feature. Permutation importance [2] defines the importance of feature by the decrease of the model performance when that feature value is randomly shuffled. Chung et al. [5] propose an automated data slicing method to validate ML models to find potential performance issues. Also, SHAP values [17] are used to determine the importance of a feature in ML models to explain KPI changes.

Drivers discovered by our method also refer the most important features in a dataset, but different from feature importance in ML models, our method does not tie to a specific ML model and focuses on the impact of features on outcomes.

Decision Mining in Process Mining. In process mining, decision mining is first introduced in [18] to identify the set of features as rules that define the choices made in a process. An extension to support conditions with disjunctions and inequalities is introduced by de Leoni et al. [14]. A typical limitation of decision tree learning is the assumption of full deterministic process executions and, therefore, non-overlapping rules. Mannhardt et al. [16] introduce an additional step to learn new decision trees for each leaf node in the first iteration for wrongly classified instances. An alignment based approach was introduced in [13] which additionally allows to discover rules associated with XOR-splits/joins and certain types of loops. Our work focuses on the factors that influenced the outcome and not only a particular choice. Consequently, our approach aims to find factors on a case-level end to end instead of a local decision point.

7 Conclusion

We formally define the problem of driver discovery, which is a technique to explain case-level outcomes in process mining. We form three categories of process-based features, and we propose a beam search method to automatically and efficiently discover effective, significant and interpretable drivers. We show the effectiveness and efficiency of our approach.

References

1. Agrawal, R., Srikant, R., et al.: Fast algorithms for mining association rules. In: Proc. 20th int. conf. very large data bases, VLDB. vol. 1215, pp. 487–499. Santiago, Chile (1994)
2. Altmann, A., Toloşi, L., Sander, O., Lengauer, T.: Permutation importance: a corrected feature importance measure. Bioinformatics **26**(10), 1340–1347 (2010)
3. Badakhshan, P., Wurm, B., Grisold, T., Geyer-Klingeberg, J., Mendling, J., vom Brocke, J.: Creating business value with process mining. The Journal of Strategic Information Systems **31**(4) (2022)
4. Breiman, L.: Random forests. Machine learning **45**, 5–32 (2001)
5. Chung, Y., Kraska, T., Polyzotis, N., Tae, K.H., Whang, S.E.: Slice finder: Automated data slicing for model validation. In: IEEE 35th International Conference on Data Engineering (ICDE). pp. 1550–1553. IEEE (2019)
6. Cox, D.R.: The regression analysis of binary sequences. Journal of the Royal Statistical Society: Series B (Methodological) **20**(2), 215–232 (1958)
7. van Dongen, B.: Bpi challenge 2017 (2017). https://doi.org/10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b
8. van Dongen, B.: Bpi challenge 2019 (2019). https://doi.org/10.4121/uuid:d06aff4b-79f0-45e6-8ec8-e19730c248f1
9. van Dongen, B., Borchert, F.F.: Bpi challenge 2018 (2018). https://doi.org/10.4121/uuid:3301445f-95e8-4ff0-98a4-901f1f204972
10. Hooker, S., Erhan, D., Kindermans, P.j., Kim, B.: Evaluating Feature Importance Estimates. arXiv (2018), `https://arxiv.org/pdf/1806.10758.pdf`
11. Hossin, M., Sulaiman, M.N.: A review on evaluation metrics for data classification evaluations. International journal of data mining & knowledge management process **5**(2), 1 (2015)
12. Kumar, A., Vembu, S., Menon, A.K., Elkan, C.: Beam search algorithms for multilabel learning. Machine learning **92**, 65–89 (2013)
13. de Leoni, M., van der Aalst, W.M.P.: Data-Aware Process Mining: Discovering Decisions in Processes Using Alignments. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. p. 1454–1461. SAC '13, Association for Computing Machinery, New York, NY, USA (2013)
14. de Leoni, M., Dumas, M., García-Bañuelos, L.: Discovering Branching Conditions from Business Process Execution Logs. In: Fundamental Approaches to Software Engineering, pp. 114–129. Springer Berlin Heidelberg (2013)
15. Lewis, R.J.: An introduction to classification and regression tree (CART) analysis. In: Annual meeting of the society for academic emergency medicine in San Francisco, California. vol. 14. Citeseer (2000)
16. Mannhardt, F., de Leoni, M., Reijers, H.A., van der Aalst, W.M.P.: Decision Mining Revisited - Discovering Overlapping Rules. In: Advanced Information Systems Engineering, pp. 377–392. Springer International Publishing (2016)
17. Padella, A., de Leoni, M., Dogan, O., Galanti, R.: Explainable process prescriptive analytics. In: 2022 4th International Conference on Process Mining (ICPM). IEEE (oct 2022)
18. Rozinat, A., van der Aalst, W.M.P.: Decision Mining in ProM. In: Lecture Notes in Computer Science, pp. 420–425. Springer Berlin Heidelberg (2006)