Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)

Exploring Context-Free Languages via Planning: The Case for Automating Machine Learning

Michael Katz, Parikshit Ram, Shirin Sohrabi, Octavian Udrea
IBM T.J. Watson Research Center
1101 Kitchawan Rd, Yorktown Heights, NY 10598, USA

Abstract

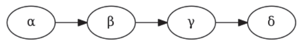
While Machine Learning has achieved considerable success in recent years, this success crucially relies on human experts to select appropriate features, workflows, algorithms with their hyper-parameters, etc. Automating the role of the human expert has seen some attention from the Machine Learning community, with a dedicated workshop running since 2014 at one of the top Machine Learning conferences. In this work, we propose to exploit multiple AI Planning tools for automating the human expert, generating Machine Learning pipelines automatically. We start from a knowledge about possible valid pipelines encoded as Context-Free Grammar, translate the problem of generating the corresponding language into Hierarchical Task Network (HTN) Planning model, further translate the HTN Planning model into a classical planning model. We use existing planners to produce multiple plans for the classical planning task, translate these plans into Machine Learning pipelines, train and evaluate these pipelines. Based on pipelines' accuracy feedback we update the classical planning model to improve the quality of pipelines obtained in next iterations. Using planning tools allows us to exploit the flexibility of model update instead of solution modification. We present an application that helps users to focus pipelines' exploration process by allowing to encode additional constraints on desired pipelines. Our experimental evaluation shows the feasibility of using planning techniques in this context.

Introduction

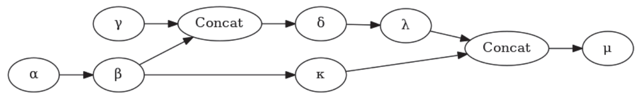
There is no doubt that the field of Machine Learning has achieved considerable successes in recent years. This success, however, crucially relies on human Machine Learning experts, that spend countless hours on selecting appropriate features, workflows, algorithms with their hyper-parameters, etc. Automating the role of the human expert is thus extremely beneficial and has received considerable attention from the Machine Learning community. There is a dedicated workshop, AutoML, running since 2014 at one of the top Machine Learning conferences (Hutter et al. 2014). As Machine Learning methods (or operators) often come with configuration parameters or hyper-parameters such as the depth of the tree for decision trees or regularization penalty for linear models or number of layers and activation functions for neural networks, finding the best configuration has been the initial focus of the research in the area (Shahriari et al. 2016). While hyper-parameter optimization is a hard problem in general, for a small number of hyper-parameters, grid search often suffices to produce good predictive performance. In other cases, sequential model-based optimization, e. g. Gaussian Process Regression (Snoek, Larochelle, and Adams. 2012) or Random Forests (Hutter, Hoos, and Leyton-Brown 2011) has been widely used.

When additionally considering the problem of choosing complete Machine Learning pipelines, selecting a good performing configuration becomes even harder. Focusing on four data preparation steps: data cleaning $\rightarrow$ preprocessing $\rightarrow$ transformation $\rightarrow$ feature selection, the combined algorithm selection and hyper-parameter optimization (CASH) problem (Thornton et al. 2012; Feurer et al. 2015) selects the Machine Learning methods for each of these pipeline steps and the corresponding hyper-parameters of these methods for a pipeline of fixed shape (fixed number of pipeline steps, fixed data-flow graph, see Figure 1a), effectively limiting the number of possible pipelines.

In an attempt to go beyond the small bounded number of pipelines, Mohr, Wever, and Hüllermeier (2018) suggest using Hierarchical Task Network (HTN) Planning for generating valid pipelines. However, the paper develops a domain-specific planner for the produced HTN task and the description of the HTN model is incomplete/missing. Further, the model is limited to chain-shaped data-flow graphs (Figure 1a). Other work suggests a Reinforcement Learning approach, applying Monte Carlo Tree Search to explore the space of possible (not necessarily valid) pipelines (Rakotoarison, Schoenauer, and Sebag 2019; Drori et al. 2018). The follow-up work, suggests additionally using grammar for implicitly defining the space of possible valid pipelines (Drori et al. 2019). Although the work explores a regular grammar, limited again to chain-shaped data-flow graphs, it is not clear what is the reason for such a restriction. Reinforcement Learning requires a large amount of available data, as well as means of specifying the reward function for strings that consist of both terminal and non-terminal sym-



(a)



(b)

Figure 1: Data-flow graph shapes, (a) chain and (b) DAG.

bols. It is not clear how the authors overcome these restrictions. Similarly, yet another work exploits Genetic Programming to explore the space of (not necessarily valid) pipelines with data-flow graphs of complex shapes (Olson and Moore 2016) and then additionally uses a Context-Free Grammar to prune the many invalid pipelines generated by genetic mutations and crossovers (de Sá et al. 2017). As in the previous work the use of grammar is also limited to chain-shaped data-flow graphs.

In this work, we show how to exploit the power of AI Planning to generate pipelines of complex Directed Acyclic Graph (DAG) shapes. As Context-Free Grammars (CFG) have been shown to be a convenient tool for modeling Machine Learning pipelines generation, we use CFG as our input. There is a great deal of similarity between CFG and HTN Planning, with the complexity of plan existence for HTN Planning proved through reduction from the question of whether the languages of two CFGs have a non-empty intersection (Erol, Hendler, and Nau 1994). The planning tools for HTN, however, are not as well developed as for classical planning. In particular, there are no tools that can obtain multiple plans for HTN, similarly to top-k, top-quality, and diverse planners in classical planning (Nguyen et al. 2012; Katz et al. 2018; Katz, Sohrabi, and Udrea 2020). However, a restricted fragment of HTN Planning can be converted into classical planning (Alford, Kuter, and Nau 2009; Alford et al. 2016). Then, classical planning tools can be used to derive multiple plans, that can then be translated all the way back to strings in the grammar. Such translations to classical planning allow us to perform a goal-oriented exploration of strings in a grammar, a task of a particular interest for Machine Learning pipelines generation. Further, the goal of the classical planning model can be extended to include user-specified constraints, focusing on specific elements that the user is interested to obtain in her pipelines. Some of these specified goals can be seen as soft goals, which is especially useful if not all constraints can be satisfied simultaneously. These soft goals extend the classical planning formalism, but can be then compiled away (Keyder and Geffner 2009). Another benefit of classical planning translation is the ability to modify the model through action costs modification, preserving the solution space exactly, while modifying individual plans costs. This slight modification, effectively, allows us to obtain different solutions when using cost-sensitive techniques.

In this paper, we present an application that demonstrates how the aforementioned techniques are exploited for helping users to focus pipelines' exploration process. Our experimental data shows that we are able to generate structurally complex pipelines. Further, we demonstrate how pipelines' accuracy improves with each exploration. Note that while our application receives Machine Learning grammar as an input, our overall approach can work with any CFG and applied to other problems, such as natural language generation. Further, to the best of our knowledge, the grammar we utilize in this work is by far the most complex among those explored for automating Machine Learning to this day. As an additional contribution, we create a new classical planning domain and will make it openly available.

The rest of the paper is structured as follows. First, we present the preliminaries, then we discuss Machine Learning pipelines and grammar in detail. Next, we describe our solution and the user-guided exploration of the space of pipelines, present our experimental evaluation, and conclude with related work and discussion.

Preliminaries

While our application of interest in Automated Machine Learning, our idea can be applied broadly to other applications and therefore we start by describing the problem of exploring the space of strings for a Context-Free Grammar.

Following the notation of Segovia-Aguas, Jiménez, and Jonsson (2017), we define a *Context-Free Grammar* (CFG) as a tuple $\mathcal{G} = \langle V, v_0, \Sigma, R\rangle$ where V is the finite set of non-terminal symbols, $v_0 \in V$ is the start non-terminal symbol, Σ is the finite set of terminal symbols, and $R = \{\alpha \to \beta \mid \alpha \in V, \beta \in (V \cup \Sigma)^*\}$ is the finite set of production rules in the grammar. The semantics of CFG is as follows: each $v \in V$ represents a sub-language of the language defined by the grammar and Σ is the alphabet of the language defined by $\mathcal{G}$ and can contain the empty string, which we denote by $\epsilon \in \Sigma$. By e_0 we denote the string that contains only the initial non-terminal symbol v_0. For a string $e_1 = u_1 \alpha u_2 \in (V \cup \Sigma)^*$ and a rule $r = \alpha \to \beta \in R$, we say that e_1 directly yields $e_2 = u_1 \beta u_2$, denoted by $e_1 \xrightarrow{r} e_2$. We say that e_1 yields e_n (denoted by $e_1 \to^* e_n$) iff there exist strings $e_1, \dots, e_n \in (V \cup \Sigma)^*$ and rules $r_1, \dots, r_{n-1} \in R$, such that for all $1 \leq i < n$ we have $e_i \xrightarrow{r_i} e_{i+1}$. In such cases, we say that $r_1 \cdot \ldots \cdot r_{n-1}$ is an *inducing sequence of rules* for the pair of strings $\langle e_1, e_n\rangle$. The language of a CFG, $L(\mathcal{G}) = \{e \in \Sigma^* : v_0 \to^* e\}$, is the set of all strings that contain only terminal symbols and that can be yielded from the string e_0. For a set of symbols $c \subseteq V \cup \Sigma$, in what follows referred to as a *constraint*, we define a *constrained* language $L^c(\mathcal{G})$ as follows. For a string e we have $e \in L^c(\mathcal{G})$ iff (i) e contains all the terminal symbols in c, and (ii) there exists an inducing sequence of rules $r_1 \cdot \ldots \cdot r_n$ for the pair of strings $\langle e_0, e\rangle$ such that $c \cap V \subseteq \bigcup_{i=1}^{n} \{v \mid r_i = v \to \alpha\}$. In words, the constrained language consists of strings that can be yielded from the string e_0 through all constrained symbols.

Definition 1 *Given a CFG $\mathcal{G}$ and a constraint c, a constrained string existence is a problem of deciding whether there exists a string $s \in L^c(\mathcal{G})$.*

```
⟨start⟩  → ⟨dag⟩≫⟨est⟩
⟨dag⟩    → NoOp()|⟨est⟩|⟨tfm⟩|⟨dag⟩≫⟨dag⟩
⟨dag⟩    → ((⟨dag⟩){ & (⟨dag⟩)}⁺) ≫Concat()
⟨est⟩    → ⟨glm⟩|⟨mlpc⟩|⟨dtc⟩|⟨ebm⟩|⟨gnb⟩|⟨knc⟩|⟨qda⟩
⟨glm⟩    → LogisticRegression(solver=⟨glmslv⟩, penalty=⟨glmpen⟩)
⟨glmpen⟩ → 'l1'|'l2'
⟨glmslv⟩ → 'newton-cg'|'sag'|'saga'|'lbfgs'|'liblinear'
⟨mlpc⟩   → MLPClassifier(activation=⟨mlpca⟩, solver=⟨mlpcs⟩,
                                        learning_rate=⟨mlpcl⟩)
⟨mlpca⟩  → 'identity'|'relu'|'tanh'|'logistic'
⟨mlpcs⟩  → 'lbfgs'|'sgd'|'adam'
⟨mlpcl⟩  → 'constant'|'invscaling'|'adaptive'
⟨dtc⟩    → DecisionTreeClassifier(criterion=⟨dtcc⟩, splitter=⟨dtcs⟩)
⟨dtcc⟩   → 'gini'|'entropy'
⟨dtcs⟩   → 'best'|'random'
⟨ebm⟩    → ⟨rfc⟩|⟨gbc⟩|⟨etc⟩
⟨rfc⟩    → RandomForestClassifier(criterion=⟨rfcc⟩)
⟨rfcc⟩   → 'gini'|'entropy'
⟨gbc⟩    → GradientBoostingClassifier(loss=⟨gbcl⟩)
⟨gbcl⟩   → 'deviance'|'exponential'
⟨etc⟩    → ExtraTreesClassifier(criterion=⟨rfcc⟩)
⟨gnb⟩    → GaussianNB()
⟨knc⟩    → KNeighborsClassifier(weights=⟨kncw⟩, metric=⟨kncm⟩)
⟨kncw⟩   → 'uniform'|'distance'
⟨kncm⟩   → 'euclidean'|'manhattan'|'minkowski'
⟨qda⟩    → QuadraticDiscriminantAnalysis()
⟨tfm⟩    → ⟨utfm⟩|⟨wtfm⟩
⟨utfm⟩   → ⟨urfnc⟩|⟨ucfnc⟩
⟨urfnc⟩  → ⟨rimp⟩|⟨scale⟩
⟨ucfnc⟩  → (KeepNumbers & (KeepNonNumbers≫OneHotEncoder()))
                                                    ≫Concat()
⟨rimp⟩   → SimpleImputer(strategy=⟨rimps⟩)
⟨rimps⟩  → 'mean'|'median'|'most_frequent'
⟨scale⟩  → Normalizer()|MinMaxScaler()|RobustScaler()|StandardScaler()
⟨wtfm⟩   → ⟨pca⟩|⟨nys⟩
⟨pca⟩    → PCA(svd_solver=⟨pcas⟩)
⟨pcas⟩   → 'auto'|'full'|'arpack'|'randomized'
⟨nys⟩    → Nystroem(kernel=⟨nysk⟩)
⟨nysk⟩   → 'rbf'|'cosine'|'poly'|'linear'|'laplacian'|'sigmoid'
```

Figure 2: Grammar rules fragment.

Definition 2 *Given a CFG $\mathcal{G}$ and a constraint c, a constrained string generation is a problem of generating a string $s \in L^c(\mathcal{G})$.*

For HTN Planning, we follow the notation of Alford et al. (2016) and Bercher, Alford, and Höller (2019). An HTN problem is a tuple $\mathcal{P} = (X_p, X_n, \mathcal{O}, \mathcal{M}, s_I, tn_I)$, where:

- X_p and X_n are a finite set of primitive and non-primitive task names respectively,
- $\mathcal{O}$ is a set of HTN operators, where each $o \in \mathcal{O}$ is a triple (n, χ, e), with $n \in X_p$ being a primitive task name, χ the precondition, and e the effect of the planning operator,
- $\mathcal{M}$ is a set of HTN methods, where each $m \in \mathcal{M}$ is a triple (c, χ, tn), with $c \in X_n$ being a non-primitive task name, χ being the precondition of m, and tn being a task network,
- s_I is the initial state and tn_I is the initial task network.

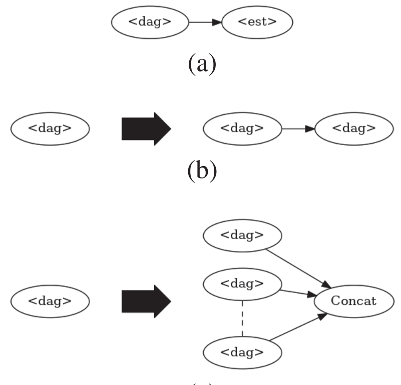


Figure 3: Visual depiction of grammar rules, (a) $\langle\texttt{dag}\rangle \gg \langle\texttt{est}\rangle$, (b) $\langle\texttt{dag}\rangle \rightarrow \langle\texttt{dag}\rangle \gg \langle\texttt{dag}\rangle$, and (c) $\langle\texttt{dag}\rangle \rightarrow ((\langle\texttt{dag}\rangle)\{$ & $(\langle\texttt{dag}\rangle)\}^+) \gg$Concat().

A task network is defined as a tuple $tn = (T, \prec, \tau)$, where T is a finite set of task instance symbols, $\prec$ is a partial order over T, and $\tau : T \mapsto (X_p \cup X_n)$ is a mapping from the task instance symbols to task names.

The full semantics of HTN Planning is described in Alford et al. (2016). A task network tn_S is a solution to an HTN problem $\mathcal{P}$ if and only if tn_S can be obtained from the initial task network tn_I by a sequence of method or operator applications (progression), does not contain non-primitive tasks, and is executable (i.e., it contains a linearization of its primitive tasks that is executable from the initial state s_I).

Machine Learning Pipelines & Grammar

We consider a CFG to define a search space that allows us to not only search over ML operators but also over complex pipeline shapes/structures. We define pipelines with complex structures utilizing the following "combinators" in the LALE python library (Hirzel et al. 2019)[1]:

- The $\gg$ combinator performs the 'pipe' operation, where $\alpha \gg \beta$ is a pipeline where the data goes into the α operator and the output of α is piped into the β operator.
- The & combinator performs parallel independent executions, where α & β is a (partial) pipeline with the operators α and β applied to the data independently in parallel. The output of this (partial) pipeline could be piped ($\gg$) into the LALE Concat operator to concatenate (or horizontally stack) the features – the pipeline would be defined as $(\alpha$ & $\beta) \gg$ Concat.
- Note that in the above examples, the operators α, β can be ML operators as well as themselves be (partial) pipelines.

AutoML Grammar

Equipped with LALE we can define a CFG for pipelines. While our approach is not restricted to any particular gram-

[1]The ability to define complex pipelines is not unique to LALE. However, LALE makes the definition of complex pipelines relatively succinct, allowing us to consider a concise CFG with strings in its language corresponding to executable LALE code.

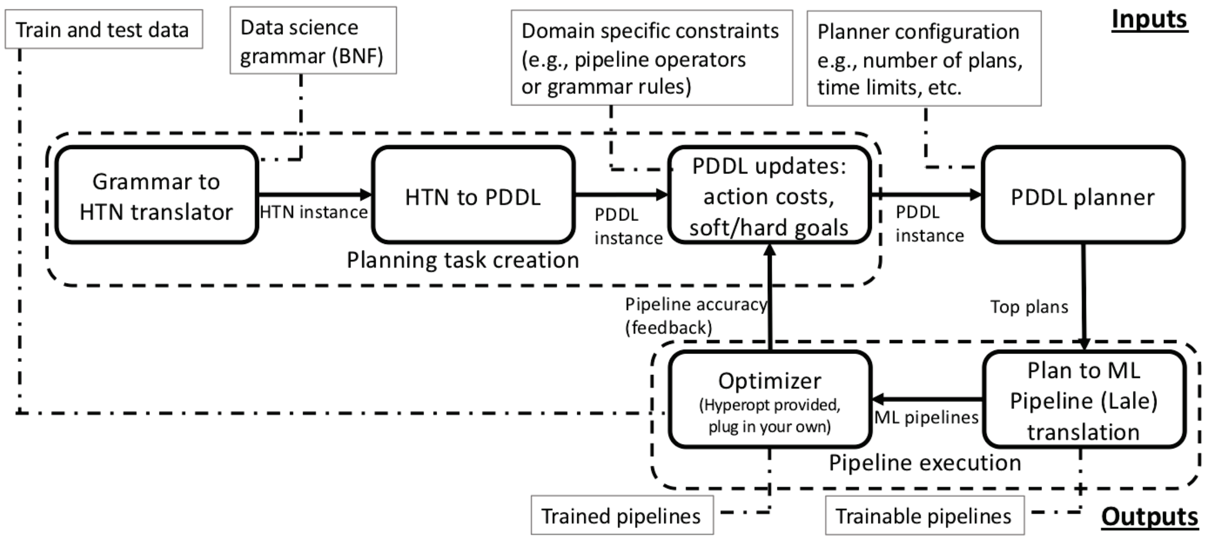

Inputs
Train and test data
Data science grammar (BNF)
Domain specific constraints (e.g., pipeline operators or grammar rules)
Planner configuration e.g., number of plans, time limits, etc.
Grammar to HTN translator
HTN instance
HTN to PDDL
PDDL instance
PDDL updates: action costs, soft/hard goals
PDDL instance
PDDL planner
Planning task creation
Pipeline accuracy (feedback)
Top plans
Optimizer (Hyperopt provided, plug in your own)
ML pipelines
Plan to ML Pipeline (Lale) translation
Pipeline execution
Trained pipelines
Trainable pipelines
Outputs

Figure 4: Solution architecture

mar, Figure 2 provides an example CFG which encodes the search space for both the shape of the data-flow graph of the pipeline and the ML operators we wish to use in the pipelines. Note that a pipeline that is valid under this CFG is an *executable* LALE *pipeline* – the grammar directly generates executable code. To save space, the notation $\alpha \to \beta|\gamma$ is used to denote two rules, $\alpha \to \beta$ and $\alpha \to \gamma$. Further, $\{\alpha\}^+$ is used to denote at least one appearance of the (terminal or non-terminal) symbol α, which can be encoded by an additional rule. The non-terminal symbols are denoted by $\langle\alpha\rangle$ and the terminal symbols are the other strings. For instance, the right hand side of the rule

$$\langle\texttt{rfc}\rangle \to \textsf{RandomForestClassifier(criterion=}\langle\texttt{rfcc}\rangle\textsf{)}$$

has one non-terminal symbol $\langle\texttt{rfcc}\rangle$ and two terminal symbols "RandomForestClassifier(criterion=" and ")".

The first three production rules in Figure 2 encode the shape of the pipelines. The first rule for the non-terminal $\langle\texttt{start}\rangle$ symbol indicates that the pipeline contains a data flow graph (denoted by the non-terminal $\langle\texttt{dag}\rangle$) piped into a ML modeling step (or estimator $\langle\texttt{est}\rangle$), depicted in Figure 3a. The production rule for $\langle\texttt{dag}\rangle$ allows it to be :

i. A NoOp implying the data is passed as is,
ii. a $\langle\texttt{est}\rangle$ which allows us to encode the practice of using predictions from one ML modeling step as features for downstream data processing and modeling,
iii. a $\langle\texttt{tfm}\rangle$ corresponding to ML data preprocessing and transformation operators,
iv. an extension of the pipeline via the recursive $\langle\texttt{dag}\rangle \gg \langle\texttt{dag}\rangle$ containing the LALE pipe combinator $\gg$, allowing the pipeline to be of arbitrary length (Figure 3b), and
v. another form of extending the pipeline data-flow graph via the recursive $((\langle\texttt{dag}\rangle)\{\&(\langle\texttt{dag}\rangle)\}^+) \gg$ Concat() containing the LALE & combinator, allowing the pipeline data-flow graph to contain parallel data-processing paths followed by a concatenation of features (Figure 3c).

The remaining production rules in Figure 2 provide the different options for the non-terminal $\langle\texttt{est}\rangle$ corresponding to the ML modeling operators and the non-terminal $\langle\texttt{tfm}\rangle$ corresponding to the ML transformation operators. The production rules also demonstrate how we can handle categorical (non-numeric) hyper-parameters within the CFG. For example, $\langle\texttt{est}\rangle \to \langle\texttt{mlpc}\rangle$ and

$$\langle\texttt{mlpc}\rangle \to \textsf{MLPClassifier(activation=}\langle\texttt{mlpca}\rangle\textsf{,solver=}\langle\texttt{mlpcs}\rangle\textsf{,}\\ \textsf{learning\_rate=}\langle\texttt{mlpcl}\rangle\textsf{)}$$

specifies the neural network modeling operator MLPClassifier from `scikit-learn` (Pedregosa et al. 2011) along with its categorical hyper-parameters for the neuron activation function $\langle\texttt{mlpca}\rangle$, optimization solver $\langle\texttt{mlpcs}\rangle$ and the learning rate schedule $\langle\texttt{mlpcl}\rangle$ for the optimization solver, defined appropriately in the subsequent production rules.

Given a CFG for ML pipelines, in the next section we detail our proposed approach to automatically generating high performing Machine Learning pipelines.

Constrained AutoML

While automation provides a lot of ease to the users, there are many cases, where the user wishes to provide certain constraints on outcome pipelines. Since the users specify the input as a grammar, these constraints should also be specified in terms of the grammar. Fortunately, as we have seen in the previous section, CFG makes it very easy for the user to specify such constraints.

For example, the user can select specific non-terminal or terminal symbols in the grammar – they can select the

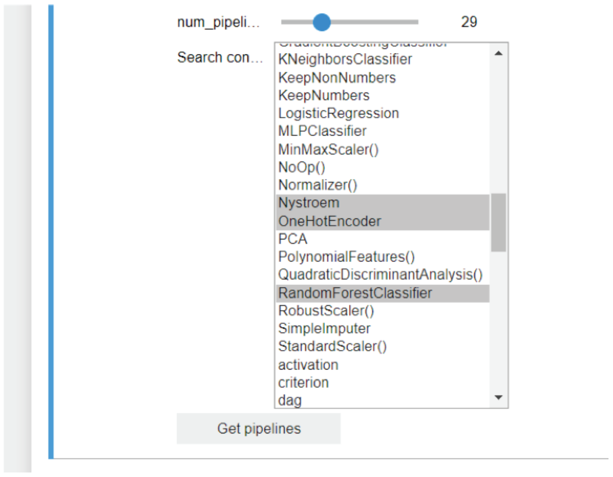


Figure 5: Example of a CFG constraint chosen.

$\langle\texttt{dtc}\rangle$ and $\langle\texttt{ebm}\rangle$ non-terminals to constrain the automated pipeline configuration to a decision tree based ML model for the modeling step or just select $\langle\texttt{ebm}\rangle$ if they prefer only tree-based ML ensembles. As discussed in the next section, the CFG makes it simple to specify multiple such constraints, allowing the user to potentially incorporate any domain knowledge or requirements. Furthermore, the CFG itself can be modified in a manner that further facilitates the constraint specification without changing the language defined by the grammar. For example, we could modify the following production rule

$$\langle\texttt{est}\rangle \to \langle\texttt{glm}\rangle|\langle\texttt{mlpc}\rangle|\langle\texttt{dtc}\rangle|\langle\texttt{ebm}\rangle|\langle\texttt{gnb}\rangle|\langle\texttt{knc}\rangle|\langle\texttt{qda}\rangle$$

into a different organization as follows:

$$\begin{aligned}\langle\texttt{est}\rangle &\to \langle\texttt{glm}\rangle|\langle\texttt{mlpc}\rangle|\langle\texttt{tree}\rangle|\langle\texttt{others}\rangle\\ \langle\texttt{tree}\rangle &\to \langle\texttt{dtc}\rangle|\langle\texttt{ebm}\rangle\\ \langle\texttt{others}\rangle &\to \langle\texttt{gnb}\rangle|\langle\texttt{knc}\rangle|\langle\texttt{qda}\rangle.\end{aligned}$$

This modified grammar makes it even easier for the user to specify a constraint such as only choosing tree based methods by only selecting a single non-terminal $\langle\texttt{tree}\rangle$.

Solution Scheme

While there could be multiple possible ways of generating strings for (constrained) Context-Free Grammars, in this work we chose to employ out-of-the-box planning techniques. While Figure 4 shows the overall architecture, in what follows, we describe the steps of our solution shown in the figure in detail.

(I) *Grammar → HTN translator.* The (unconstrained) Context-Free Grammar (the 'Data Science BNF' input in Figure 4) is translated into an HTN Planning model (Alford et al. 2016), as detailed below.

(II) *HTN → PDDL.* The HTN model is translated into classical planning (Alford, Kuter, and Nau 2009). We use the available off-the-shelf tool for STRIPS-compatible translation for totally-ordered problems. The algorithm requires to specify a parameter that roughly corresponds to the non-tail recursion depth of the HTN. We set that parameter to 20.

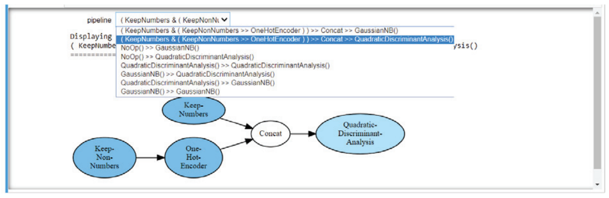

pipeline
(KeepNumbers & (KeepNonNu
(KeepNumbers & (KeepNonNumbers >> OneHotEncoder)) >> Concat >> GaussianNB()
(KeepNumbers & (KeepNonNumbers >> OneHotEncoder)) >> Concat >> QuadraticDiscriminantAnalysis()
NoOp() >> GaussianNB()
NoOp() >> QuadraticDiscriminantAnalysis()
QuadraticDiscriminantAnalysis() >> QuadraticDiscriminantAnalysis()
GaussianNB() >> QuadraticDiscriminantAnalysis()
QuadraticDiscriminantAnalysis() >> GaussianNB()
GaussianNB() >> GaussianNB()
Displaying
(KeepNumbe
ysis()
Keep-Numbers
Keep-Non-Numbers
One-Hot-Encoder
Concat
Quadratic-Discriminant-Analysis

Figure 6: Pipeline visualization in LALE.

(III) *PDDL updates: action costs, soft/hard goals.* The classical planning model is updated to incorporate the constraints as hard goals. This is done by extending the goal of the translated classical planning task with atoms that correspond to the constraints (c.f. Figure 5). Further, we allow tweaking the planning model by modifying the costs of individual actions. This forces the quality aware planners to produce solutions different from previously found ones.

(IV) *PDDL planner.* We exploit planners that produce multiple solutions (Katz et al. 2018; Katz, Sohrabi, and Udrea 2020; Katz and Sohrabi 2020) to derive multiple plans, translating these plans to strings in the constrained CFG. By using quality or diversity focused planners we can somewhat control the exploration through the space of strings in the constrained CFG. Further, these constraints can be easily relaxed by turning the corresponding hard goals into soft goals. Such soft goals can be compiled away, producing again a classical planning model (Keyder and Geffner 2009). In our implementation, we are using a top-k planner (Katz et al. 2018).

(V) *Plan → ML pipeline (LALE) translation.* A set of plans is translated into a set of Machine Learning pipelines. In our implementation, we focus on LALE pipelines (Hirzel et al. 2019). An example pipeline visualization in LALE is shown in Figure 6.

(VI) *Optimizer.* The pipelines are trained (with unspecified hyper-parameters configured with an off-the-shelf hyper-parameter optimizer such as HyperOpt or SMAC) on the training data and their performance is tested on heldout data. The example of such a training is shown in Figure 7. The accuracy of trained pipelines is translated into a feedback on a quality of the pipeline, expressed in action costs (see Figure 8). The feedback is then used as an input to step III, to update the classical planning model. The computation is described in what follows.

In the rest of this section, we describe the translation of an unconstrained CFG to HTN Planning model as well as the action cost modification from the feedback.

Given a CFG $\mathcal{G} = (V, v_0, \Sigma, R)$ as defined earlier, we define the induced HTN Planning problem $\mathcal{P}^{\mathcal{G}} = (\Sigma, V, \mathcal{O}, \mathcal{M}, s_I, tn_I)$ as follows.

- $\mathcal{O} = \{(n, \emptyset, \emptyset) \mid n \in \Sigma\}$ [2]

[2]It is possible to introduce a predicate for each operator, de-

- $\mathcal{M} = \{m_r = (\alpha, \emptyset, (T_r, \prec_r, \tau_r)) \mid r = \alpha \to \beta \in R\}$, where
  $\beta = e_1 \cdot \ldots \cdot e_n$,
  $T_r = \{t_1, \ldots, t_n\}$,
  $t_i \prec_r t_j$ iff $i < j$, and
  $\tau_r(t_i) = e_i$ for $1 \leq i \leq n$,
- $s_I = \emptyset$, and
- $tn_I = (\{t_I\}, \emptyset, \tau_I)$, where $\tau_I(t_I) = v_0$.

In words, we define an operator with empty precondition and effects for each terminal node in the grammar and a method for each production rule. Note, multiple production rules with the same left hand side symbols will result in multiple methods for the same task. The initial task network comprises of one task that represents the initial non-terminal symbol in the grammar. It is worth noting that similar constructions have previously been suggested (Erol, Hendler, and Nau 1996; Höller et al. 2014).

Theorem 1 *For a Context-Free Grammar $\mathcal{G}$, there is a bijective mapping between its language $L(\mathcal{G})$ and the set of solutions to its induced HTN Planning problem $\mathcal{P}^{\mathcal{G}}$.*

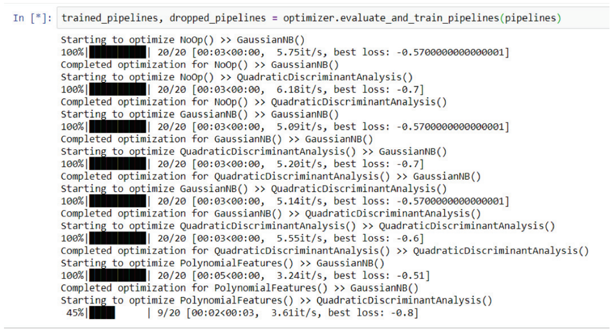
Proof sketch: Let $tn_S = (T, \prec, \tau)$ be a solution to the HTN problem $\mathcal{P}^{\mathcal{G}}$. Then $T = \{t_1, \ldots, t_n\}$ consists of primitive tasks only. Further, $\prec$ is a total order over T, w.l.o.g., $t_1 \cdot \ldots \cdot t_n$, since each method's task network corresponds to a total order over its tasks. Let $e = \tau(t_1) \cdot \ldots \cdot \tau(t_n)$ be the corresponding sequence of terminal symbols, ρ be the sequence of method and operator applications that produced tn_S, and $m_1, \ldots, m_k$ be the sub-sequence of ρ of methods. Then, applying the corresponding sequence of rules $r_1, \ldots, r_k$ to the initial string e_0 would result in e and therefore $e \in L(\mathcal{G})$.

For the other direction, let $e = e_1 \cdot \ldots \cdot e_n \in L(\mathcal{G})$. Among the possible sequences of rules that yield e from e_0, let $r_1, \ldots, r_k$ be one such that at each step i the rule r_i is applied to the left-most non-terminal symbol. Then, a sequence of method and operator applications that produces a solution to the HTN problem $\mathcal{P}^{\mathcal{G}}$ can be obtained by merging the sequences $m_1, \ldots, m_k$ of methods and $e_1, \ldots, e_n$ of operators.

□

We now describe the cost allocation, as derived from the feedback mechanism. Action costs are updated automatically after each optimization round as follows. First, we record all feedback from the optimizer from the beginning of the process. The feedback takes the form of scores in $[0, 1]$ per pipeline, based on the optimized metric, be it accuracy, area under the curve, or fairness, with higher scores being better. Given all such scores for pipelines, we assign each action an integer cost between 1 and highest possible action cost (100 in our current implementation), inversely proportional to the average score of the pipelines in which it appears. Actions that have not yet appeared in any pipeline receive a default score (30 in our implementation).

scribing whether the operator was applied. In our implementation, we add such predicates to the classical planning model.



```
In [*]: trained_pipelines, dropped_pipelines = optimizer.evaluate_and_train_pipelines(pipelines)

        Starting to optimize NoOp() >> GaussianNB()
        100%|██████████| 20/20 [00:03<00:00,  5.75it/s, best loss: -0.5700000000000001]
        Completed optimization for NoOp() >> GaussianNB()
        Starting to optimize NoOp() >> QuadraticDiscriminantAnalysis()
        100%|██████████| 20/20 [00:03<00:00,  6.18it/s, best loss: -0.7]
        Completed optimization for NoOp() >> QuadraticDiscriminantAnalysis()
        Starting to optimize GaussianNB() >> GaussianNB()
        100%|██████████| 20/20 [00:03<00:00,  5.09it/s, best loss: -0.5700000000000001]
        Completed optimization for GaussianNB() >> GaussianNB()
        Starting to optimize QuadraticDiscriminantAnalysis() >> GaussianNB()
        100%|██████████| 20/20 [00:03<00:00,  5.20it/s, best loss: -0.7]
        Completed optimization for QuadraticDiscriminantAnalysis() >> GaussianNB()
        Starting to optimize GaussianNB() >> QuadraticDiscriminantAnalysis()
        100%|██████████| 20/20 [00:03<00:00,  5.14it/s, best loss: -0.5700000000000001]
        Completed optimization for GaussianNB() >> QuadraticDiscriminantAnalysis()
        Starting to optimize QuadraticDiscriminantAnalysis() >> QuadraticDiscriminantAnalysis()
        100%|██████████| 20/20 [00:03<00:00,  5.55it/s, best loss: -0.6]
        Completed optimization for QuadraticDiscriminantAnalysis() >> QuadraticDiscriminantAnalysis()
        Starting to optimize PolynomialFeatures() >> GaussianNB()
        100%|██████████| 20/20 [00:05<00:00,  3.24it/s, best loss: -0.51]
        Completed optimization for PolynomialFeatures() >> GaussianNB()
        Starting to optimize PolynomialFeatures() >> QuadraticDiscriminantAnalysis()
         45%|████      | 9/20 [00:02<00:03,  3.61it/s, best loss: -0.8]
```

Figure 7: Pipeline training phase.

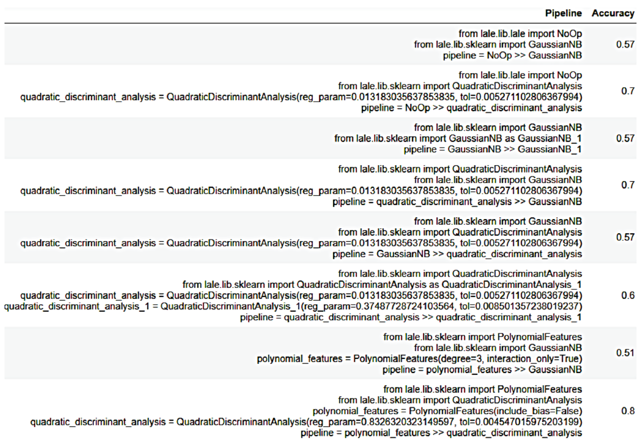
User Guided Exploration

In this section, we put all the pieces together and describe our implementation of the proposed approach with a view of user's interaction with the system.

We have implemented the proposed approach as outlined in Figure 4 in a python notebook in a docker container. The user has an option to select a data science grammar from a list of possible grammars (see for example the grammar in Figure 2). We then create the HTN domain from the data science grammar followed by translating the HTN model to classical planning (Alford et al. 2016). We then allow the user to select a set of constraints as well as the number of pipelines they like to generate as shown in Figure 5. This step is optional, but selecting constraints allows the user to obtain pipelines that better match what they are interested in. Recall that the constraints are treated as soft goals; a subset of them will be met if it is not possible to obtain solutions that would meet the selected constraints.

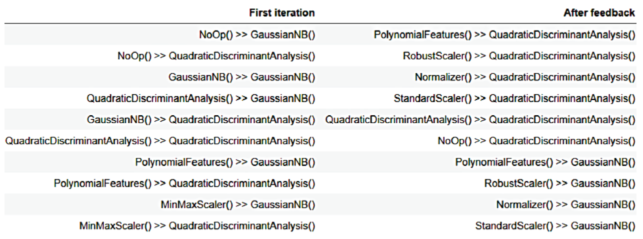
Next, the user has the ability to explore the generated pipelines further by visualizing them. The set of generated pipelines will be shown in a drop-down menu and the user has the ability to select a desired pipeline and visualize them. An example is shown in Figure 6.

The generated pipelines then go to a training phase in which we use a number of different optimizers. An example of a corresponding cell in the python notebook is shown in Figure 7. Once training is done, we obtain the accuracy of each generated pipeline and show that to the user, as exemplified in Figure 8 (a). The figure shows pipelines, with their accuracy varying from 0.51 to 0.8. These accuracy numbers are then used to update the cost of associated actions. For example, the pipelines that include GaussianNB() seem to have a low accuracy which results in associating a high cost with the GaussianNB() action. Figure 8 (b) shows the 10 generated pipelines sorted based on their cost, and you can see that just after one iteration, the "after feedback" column takes into account the updated costs to re-sort. For example, the pipelines that include GaussianNB() either drop down from the list or move down on the list illustrating the effect of the feedback. On the other hand, given that the pipeline that included QuadraticDiscriminantAnalysis() has a high accuracy of 0.8, the pipelines that include QuadraticDiscriminantAnalysis() either move up the list



Pipeline	Accuracy
from lale.lib.lale import NoOp from lale.lib.sklearn import GaussianNB pipeline = NoOp >> GaussianNB	0.57
from lale.lib.lale import NoOp from lale.lib.sklearn import QuadraticDiscriminantAnalysis quadratic_discriminant_analysis = QuadraticDiscriminantAnalysis(reg_param=0.013183035637853835, tol=0.005271102806367994) pipeline = NoOp >> quadratic_discriminant_analysis	0.7
from lale.lib.sklearn import GaussianNB from lale.lib.sklearn import GaussianNB as GaussianNB_1 pipeline = GaussianNB >> GaussianNB_1	0.57
from lale.lib.sklearn import QuadraticDiscriminantAnalysis from lale.lib.sklearn import GaussianNB quadratic_discriminant_analysis = QuadraticDiscriminantAnalysis(reg_param=0.013183035637853835, tol=0.005271102806367994) pipeline = quadratic_discriminant_analysis >> GaussianNB	0.7
from lale.lib.sklearn import GaussianNB from lale.lib.sklearn import QuadraticDiscriminantAnalysis quadratic_discriminant_analysis = QuadraticDiscriminantAnalysis(reg_param=0.013183035637853835, tol=0.005271102806367994) pipeline = GaussianNB >> quadratic_discriminant_analysis	0.57
from lale.lib.sklearn import QuadraticDiscriminantAnalysis from lale.lib.sklearn import QuadraticDiscriminantAnalysis as QuadraticDiscriminantAnalysis_1 quadratic_discriminant_analysis = QuadraticDiscriminantAnalysis(reg_param=0.013183035637853835, tol=0.005271102806367994) quadratic_discriminant_analysis_1 = QuadraticDiscriminantAnalysis_1(reg_param=0.37487728724103564, tol=0.008501357238019237) pipeline = quadratic_discriminant_analysis >> quadratic_discriminant_analysis_1	0.6
from lale.lib.sklearn import PolynomialFeatures from lale.lib.sklearn import GaussianNB polynomial_features = PolynomialFeatures(degree=3, interaction_only=True) pipeline = polynomial_features >> GaussianNB	0.51
from lale.lib.sklearn import PolynomialFeatures from lale.lib.sklearn import QuadraticDiscriminantAnalysis polynomial_features = PolynomialFeatures(include_bias=False) quadratic_discriminant_analysis = QuadraticDiscriminantAnalysis(reg_param=0.8326320323149597, tol=0.004547015975203199) pipeline = polynomial_features >> quadratic_discriminant_analysis	0.8

(a)



First iteration	After feedback
NoOp() >> GaussianNB()	PolynomialFeatures() >> QuadraticDiscriminantAnalysis()
NoOp() >> QuadraticDiscriminantAnalysis()	RobustScaler() >> QuadraticDiscriminantAnalysis()
GaussianNB() >> GaussianNB()	Normalizer() >> QuadraticDiscriminantAnalysis()
QuadraticDiscriminantAnalysis() >> GaussianNB()	StandardScaler() >> QuadraticDiscriminantAnalysis()
GaussianNB() >> QuadraticDiscriminantAnalysis()	QuadraticDiscriminantAnalysis() >> QuadraticDiscriminantAnalysis()
QuadraticDiscriminantAnalysis() >> QuadraticDiscriminantAnalysis()	NoOp() >> QuadraticDiscriminantAnalysis()
PolynomialFeatures() >> GaussianNB()	PolynomialFeatures() >> GaussianNB()
PolynomialFeatures() >> QuadraticDiscriminantAnalysis()	RobustScaler() >> GaussianNB()
MinMaxScaler() >> GaussianNB()	Normalizer() >> GaussianNB()
MinMaxScaler() >> QuadraticDiscriminantAnalysis()	StandardScaler() >> GaussianNB()

(b)

Figure 8: Feedback mechanism, (a) pipelines accuracy and (b) comparison between the first and the second iteration.

or new pipelines with QuadraticDiscriminantAnalysis() are added to the list of generated pipelines.

Experimental Evaluation

In order to empirically evaluate our suggested approach, we have performed two experiments. First, to evaluate the feasibility of using planning tools in a way we suggest in this paper, we generated a benchmark set from the grammar presented in this paper and a randomly chosen subset of constraints. By applying the sequence of reformulations as described in the paper, we obtain a classical planning task for each such grammar and a set of constraint. We have created a set of 100 tasks, 10 for each constraint size, from 1 to 10 and made the benchmark set publicly available[3] (Katz et al. 2020). The feasibility of our approach is evaluated by attempting to produce 1000 plans with a K^* planner (Katz et al. 2018). The experiments were performed on Intel(R) Xeon(R) CPU E7-8837 @2.67GHz machines, using a time and memory bounds of 30 minutes and 4GB, respectively. Figure 9 shows the total time until all 1000 plans were found. The planner was able to find all 1000 plans for all 100 tasks in our benchmark set. For constraint sets of up to size 4, for all tasks the planner was able to produce all plans in under 6 seconds. For relatively large constraint sets of up to size 7, the planner always finishes in less than 50 seconds. As expected, the total time grows exponentially with constraint set size, but even for very large constraint sets of up to size 10, the planner always finishes in less than 500 seconds. Note that while we have scaled the constraint set size to 10, we do not expect the human experts that work with our application to select such a large set of constraints.

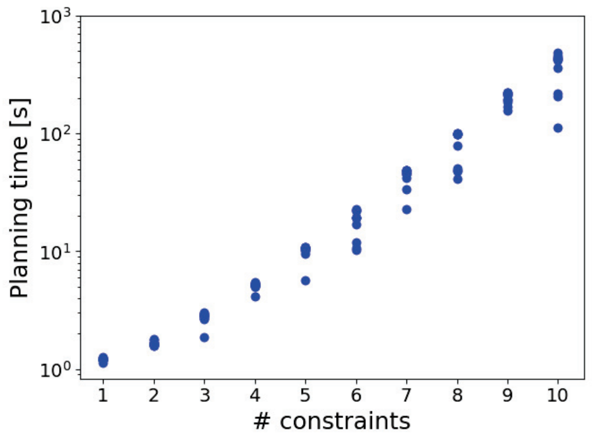


Figure 9: Total time until 1000 plans are found.

Second, we compare our proposed methods to the well established CASH approach for automated Machine Learning pipeline generation (Thornton et al. 2012; Feurer et al. 2015). For computational reasons, the algorithm selection in CASH is restricted to 6 data transformers and 8 estimators. The configurations we compare to correspond to two-step (data transformer step, then modeling step) and three-step (data scaling step, then data transformer step, then modeling step) chain-shaped pipelines, HandCrafted 2 ops and 3 ops, respectively. The CASH problem is solved using Hyperopt[4] (Bergstra et al. 2011), that optimizes both algorithm and hyper-parameter selection. Since Machine Learning algorithms have non-categorical hyper-parameters, our approach also requires the use of hyper-parameter optimization, at least for the non-categorical hyper-parameters. Therefore, in our experiments, we have decided to defer the entire hyper-parameter selection to Hyperopt, removing hyper-parameters from our input grammar. We evaluate two suggested approaches. Our first approach corresponds to a single planner run generating 1000 plans and then optimizing the corresponding pipelines in the order of plan costs (One-shot). Our second approach exploits the feedback mechanism, generating a number of pipelines per round and modifying the action costs based on the feedback obtained from the planner (Feedback). We experiment with 20, 40, 60, 80, and 100 pipelines per round. For parity, we restrict our approach to the same set of Machine Learning algorithms and optimizer (in this case Hyperopt) as the baseline approaches. Each configuration was executed 10 times for approximately the same amount of time and we report the aggregate performance.

We consider 27 datasets from OpenML (Van Rijn et al.

[3]https://github.com/IBM/PDDL-benchmark-ds-grammar

[4]https://pypi.org/project/hyperopt/

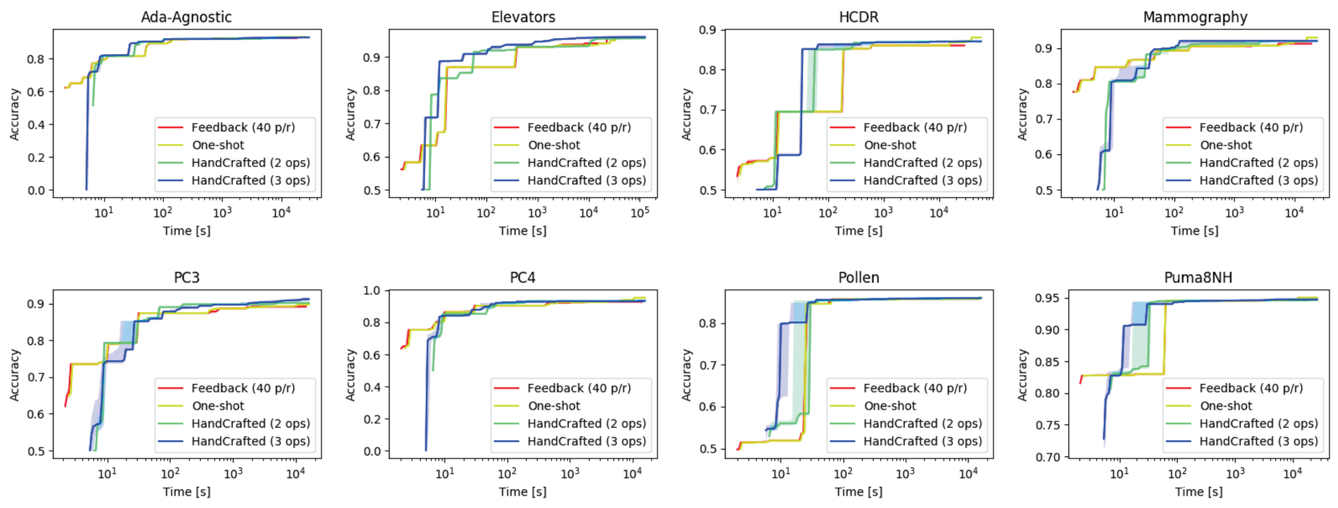


Figure 10: Accuracy comparison of our approaches to the state-of-the-art on various datasets. The x axis is in log scale.

2013) and compare the balanced-accuracy of all evaluated methods. We observed that all the tested methods achieve an extremely high accuracy ($> 95\%$) very early on many of the datasets we experimented with. Therefore, we limit our presentation to 8 datasets where the maximal achieved accuracy was $\leq 95\%$. Figure 10 depicts the (balanced) accuracy (and the inter-quartile range across 10 runs) comparison of our techniques to the hand crafted baseline approaches. For the Feedback scheme, we utilize 40 pipelines per round. Our experiments show that changing the number of pipelines per round has limited effect on accuracy and hence these results are omitted from the paper. First, note that Feedback accuracy is very close to One-shot. We conjecture that our approach that penalizes algorithms that participate in pipelines of low accuracy is too simplistic to be able to guide the search towards pipelines of higher accuracy. Second, in 5 out of 8 datasets One-shot achieves the highest *final accuracy* among the tested approaches. There is only one case when the HandCrafted 3 ops eventually achieves a better accuracy and two domains where the final accuracy is eventually the same.

Related Work

As mentioned before, there is a great deal of similarity between Context-Free Grammars and HTN Planning (Erol, Hendler, and Nau 1994; Geier and Bercher 2011), but also between regular grammars and classical planning (Erol, Hendler, and Nau 1994; Lekavỳ and Návrat 2007). Another previously mentioned work goes the opposite direction from taken by this work, exploiting planning techniques for generating CFGs (Segovia-Aguas, Jiménez, and Jonsson 2017). Going beyond CFGs, Höller et al. (2014) show that HTN Planning produces context-sensitive languages. They also identify subclasses of HTN planning that generate regular or context-free languages. It was also shown that Combinatory Categorial Grammars (CCGs) can be used for plan recognition or planning (Geib and Goldman 2011; Geib 2015). Yet another work exploits attribute grammars to validate HTN plans (Barták, Maillard, and Cardoso 2018).

Discussion and Future Work

We have introduced a novel application of planning approaches to automating Machine Learning. Our application integrates multiple existing planning techniques, including HTN Planning, classical planning, soft goals, translations from HTN Planning and from soft goals to classical planning, solvers that produce multiple solutions, as well as introduces a novel translation from grammars to HTN Planning. Our application creates a planning model, generates plans and then iteratively modifies the planning model based on previously produced plans, to produce plans that better reflect user preferences. This allows, for the first time, to easily explore the space of all possible valid pipelines (as defined by the grammar), without posing severe restrictions on the set of such pipelines.

Note that this work did not explore various possible model modifications, given a feedback on the previously found pipelines. We restricted ourselves to action cost modifications. Further, our current method is rather straightforward: we used a very simple linear function to assign scores to individual pipeline components based on the accuracy of the entire pipeline. Our experimental evaluation shows that such an approach does not provide any meaningful benefit over the base One-shot one. For future work, one possible idea is to consider action interaction. Another possible future direction is applying our approach to other domains, such as natural language generation. A connection between natural language generation and planning has been previously established (e.g., Koller and Hoffmann 2010). Grammars are explored widely for generating natural language and thus our approach should work almost out of the box, allowing for focused exploration of possible sentences.

References

Alford, R. W.; Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; and Aha, D. W. 2016. Bound to plan: Exploiting classical heuristics via automatic translations of tail-recursive HTN problems. In *Proc. ICAPS 2016*, 20–28.

Alford, R. W.; Kuter, U.; and Nau, D. 2009. Translating HTNs to PDDL: A small amount of domain knowledge can go a long way. In *Proc. IJCAI 2009*, 1629–1634.

Barták, R.; Maillard, A.; and Cardoso, R. C. 2018. Validation of hierarchical plans via parsing of attribute grammars. In *Proc. ICAPS 2018*, 11–19.

Bercher, P.; Alford, R.; and Höller, D. 2019. A survey on hierarchical planning – one abstract idea, many concrete realizations. In *Proc. IJCAI 2019*, 6267–6275.

Bergstra, J. S.; Bardenet, R.; Bengio, Y.; and Kégl, B. 2011. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, 2546–2554.

de Sá, A. G.; Pinto, W. J. G.; Oliveira, L. O. V.; and Pappa, G. L. 2017. Recipe: a grammar-based framework for automatically evolving classification pipelines. In *European Conference on Genetic Programming*, 246–261. Springer.

Drori, I.; Krishnamurthy, Y.; Rampin, R.; Lourenco, R.; One, J.; Cho, K.; Silva, C.; and Freire, J. 2018. AlphaD3M: Machine learning pipeline synthesis.

Drori, I.; Krishnamurthy, Y.; Lourenco, R.; Rampin, R.; Cho, K.; Silva, C.; and Freire, J. 2019. Automatic machine learning by pipeline synthesis using model-based reinforcement learning and a grammar. arXiv:1905.10345 [cs.LG].

Erol, K.; Hendler, J. A.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *Proc. AAAI 1994*, 1123–1128.

Erol, K.; Hendler, J.; and Nau, D. S. 1996. Complexity results for HTN planning. *Annals of Math. and AI* 18(1):69–93.

Feurer, M.; Klein, A.; Eggensperger, K.; Springenberg, J.; Blum, M.; and Hutter, F. 2015. Efficient and robust automated machine learning. In *Proc. NIPS 2015*, 2962–2970.

Geib, C., and Goldman, R. 2011. Recognizing plans with loops represented in a lexicalized grammar. In *Proc. AAAI 2011*, 958–963.

Geib, C. 2015. Lexicalized reasoning. In *Proc. ACS 2015*.

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In *Proc. IJCAI 2011*.

Hirzel, M.; Kate, K.; Shinnar, A.; Roy, S.; and Ram, P. 2019. Type-driver automated learning with lale. arXiv:1906.03957 [cs.PL, cs.LG, cs.SE].

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language classification of hierarchical planning problems. In *Proc. ECAI 2014*, 447–452.

Hutter, F.; Caruana, R.; Bardenet, R.; Bilenko, M.; Guyon, I.; Kégl, B.; and Larochelle, H., eds. 2014. *ICML 2014 AutoML Workshop*.

Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2011. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, 507–523. Berlin, Heidelberg: Springer-Verlag.

Katz, M., and Sohrabi, S. 2020. Reshaping diverse planning. In *Proc. AAAI 2020*.

Katz, M.; Sohrabi, S.; Udrea, O.; and Winterer, D. 2018. A novel iterative approach to top-k planning. In *Proc. ICAPS 2018*.

Katz, M.; Ram, P.; Sohrabi, S.; and Udrea, O. 2020. Grammar2PDDL: PDDL benchmark generated from data science grammar. https://doi.org/10.5281/zenodo.3694326.

Katz, M.; Sohrabi, S.; and Udrea, O. 2020. Top-quality planning: Finding practically useful sets of best plans. In *Proc. AAAI 2020*.

Keyder, E., and Geffner, H. 2009. Soft goals can be compiled away. *JAIR* 36:547–556.

Koller, A., and Hoffmann, J. 2010. Waking up a sleeping rabbit: On natural-language sentence generation with FF. In *Proc. ICAPS 2010*, 238–241.

Lekavỳ, M., and Návrat, P. 2007. Expressivity of STRIPS-like and HTN-like planning. In *KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*, 121–130. Springer.

Mohr, F.; Wever, M.; and Hüllermeier, E. 2018. ML-Plan: Automated machine learning via hierarchical planning. *Machine Learning* 107(8):1495–1515.

Nguyen, T. A.; Do, M. B.; Gerevini, A.; Serina, I.; Srivastava, B.; and Kambhampati, S. 2012. Generating diverse plans to handle unknown and partially known user preferences. *AIJ* 190:1–31.

Olson, R. S., and Moore, J. H. 2016. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *ICML 2016 AutoML Workshop*, 66–74.

Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; and Duchesnay, É. 2011. Scikit-learn: Machine learning in Python. *JMLR* 12:2825–2830.

Rakotoarison, H.; Schoenauer, M.; and Sebag, M. 2019. Automated machine learning with monte-carlo tree search. In *Proc. IJCAI 2019*, 3296–3303.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2017. Generating context-free grammars using classical planning. In *Proc. IJCAI 2017*, 4391–4397.

Shahriari, B.; Swersky, K.; Wang, Z.; Adams, R. P.; and de Freitas, N. 2016. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE* 104(1):148–175.

Snoek, J.; Larochelle, H.; and Adams., R. P. 2012. Practical bayesian optimization of machine learning algorithms. In *Proc. NIPS 2012*, 2951–2959.

Thornton, C.; Hoos, H. H.; Hutter, F.; and Leyton-Brown, K. 2012. Auto-weka: Automated selection and hyper-parameter optimization of classification algorithms. *arXiv*.

Van Rijn, J. N.; Bischl, B.; Torgo, L.; Gao, B.; Umaashankar, V.; Fischer, S.; Winter, P.; Wiswedel, B.; Berthold, M. R.; and Vanschoren, J. 2013. OpenML: A collaborative science platform. In *Joint European conference on Machine Learning and Knowledge Discovery in Databases*, 645–649. Springer.