The VLDB Journal (2024) 33:1129–1153
https://doi.org/10.1007/s00778-024-00835-2

SPECIAL ISSUE PAPER

Assisted design of data science pipelines

Sergey Redyuk[1,2] · **Zoi Kaoudi**[3] · **Sebastian Schelter**[4] · **Volker Markl**[1,2]

Received: 31 January 2023 / Revised: 25 December 2023 / Accepted: 2 January 2024 / Published online: 13 February 2024

Abstract
When designing data science (DS) pipelines, end-users can get overwhelmed by the large and growing set of available data preprocessing and modeling techniques. Intelligent discovery assistants (IDAs) and automated machine learning (AutoML) solutions aim to facilitate end-users by (semi-)automating the process. However, they are expensive to compute and yield limited applicability for a wide range of real-world use cases and application domains. This is due to (a) their need to execute thousands of pipelines to get the optimal one, (b) their limited support of DS tasks, e.g., supervised classification or regression only, and a small, static set of available data preprocessing and ML algorithms; and (c) their restriction to quantifiable evaluation processes and metrics, e.g., tenfold cross-validation using the ROC AUC score for classification. To overcome these limitations, we propose a human-in-the-loop approach for the *assisted design of data science pipelines* using previously executed pipelines. Based on a user query, i.e., data and a DS task, our framework outputs a ranked list of pipeline candidates from which the user can choose to execute or modify in real time. To recommend pipelines, it first identifies relevant datasets and pipelines utilizing efficient similarity search. It then ranks the candidate pipelines using multi-objective sorting and takes user interactions into account to improve suggestions over time. In our experimental evaluation, the proposed framework significantly outperforms the state-of-the-art IDA tool and achieves similar predictive performance with state-of-the-art long-running AutoML solutions while being real-time, generic to any evaluation processes and DS tasks, and extensible to new operators.

Keywords Intelligent discovery assistant · Data science pipelines · Automated pipeline generation

1 Introduction

Data science (DS) has unarguably contributed to advancements in science, industry, and society in general. This has been achieved via many libraries and tools that are available today. Yet, designing modern DS pipelines, i.e., compositions of data preprocessing and modeling operators (e.g., filters, transformers, estimators), can become overwhelming, especially for novice users and domain experts [10]. This is due to the large and growing toolkit for data analysis and machine learning (ML), exacerbated by the lack of explicit guidance on how these techniques should be applied (both separately and as part of an end-to-end pipeline). Even for ML experts, keeping up with the ever-increasing number of available techniques and algorithms is challenging.

To facilitate users with common tasks, such as supervised classification, automated ML (AutoML) solutions [23, 47, 55] aim at fully automating the synthesis of effective DS pipelines. They navigate an often large search space of available models and their hyperparameters, execute each explored pipeline to determine its performance, and output the optimal pipeline. Similarly, intelligent discovery assistant (IDA) is a family of tools that provide step-by-step guidance to end-users with a particular task (e.g., data clean-

Sergey Redyuk: Most of the work done while at TU Berlin.

✉ Sergey Redyuk
sergey.redyuk@dfki.de

Zoi Kaoudi
zoka@itu.dk

Sebastian Schelter
s.schelter@uva.nl

Volker Markl
volker.markl@tu-berlin.de

[1] German Research Center for Artificial Intelligence (DFKI), Berlin, Germany

[2] Technical University Berlin, Berlin, Germany

[3] IT University of Copenhagen, Copenhagen, Denmark

[4] University of Amsterdam, Amsterdam, The Netherlands

ing, preprocessing, and modeling) by utilizing a fixed set of meta-features, knowledge- or case- bases [45]. However, both AutoML and IDA solutions inherit a common set of drawbacks.

First, they support a limited set of operators and DS tasks. This makes them inapplicable in certain domains that require domain-specific techniques (e.g., marker gene identification in genomics). This is because such techniques are not readily available in existing tools, and incorporating them would require substantial code changes. Second, both solutions explore a predefined search space and require the execution of many pipeline candidates before finding the optimal one. This requires extensive time and resources. Should these solutions operate on a constantly growing search space without advancing their underlying search process, their execution time might grow exponentially. Third, both AutoML approaches and IDAs rely heavily on quantifiable evaluation processes (e.g., Tenfold cross-validation with ROC AUC score) to identify the optimal pipeline. This is problematic for many domain-specific applications, such as single-cell research for cancer studies (see Sect. 2), where domain experts cannot quantify the pipeline evaluation. Instead, they rely on literature reviews and manual comparison of pipeline outputs to validate the results and determine a satisfactory variant.

Devising a solution that mitigates these drawbacks is not straightforward. First, most existing approaches are very rigid and not sustainably extensible to new operators, DS tasks, evaluation processes and, thus, would require a complete code re-design. Second, constant extensibility translates to a continuously growing search space, which, consequently, may significantly hurt the efficiency of finding DS pipelines. Third, automating the design of DS pipelines in cases where quantifiable performance metrics cannot be formulated (very common in emerging application domains such as astrophysics or cell biology) is not feasible because of the lack of processes for (pairwise) pipeline comparison.

In this work, we propose DORIAN, a human-in-the-loop framework that is based on previously executed pipelines for the assisted design of DS pipelines. The human-in-the-loop aspect is crucial to tackle the challenge of non-existing quantifiable performance metrics. Using previously executed pipelines tackles the challenge of extensibility, as the design of the core components does not depend on the supported DS tasks or operators. Our goal is to provide real-time pipeline suggestions to users and support a broad range of arbitrary DS tasks, operators, and evaluation processes. Given a user query, i.e., data and a DS task (e.g., classification), DORIAN outputs in *real time* a ranked list of *relevant* pipeline candidates that the user can choose to execute or modify. DORIAN achieves this by (1) efficiently storing all previous experiments, i.e., executed pipelines together with the data and the evaluation processes used, (2) finding relevant pipelines and ranking them based on the user's past and current interactions, (3) converting the source code of DS scripts to a stringent graph-based pipeline representation that is crucial for storing and recommending pipelines. We have showcased DORIAN in Redyuk et al., 2022 [42].

After introducing a use case that shows a clear need for an extensible human-in-the-loop solution that can support a broad range of applications (Sect. 2), we make the following contributions:

(1) We formalize the generalized problem of the assisted design of DS pipelines in a way that subsumes a wide spectrum of specific problems in the areas of AutoML and IDA (Sect. 3);
(2) After presenting an overview of our solution (Sect. 4), we detail our recommendation engine that provides user-tailored pipeline suggestions in real-time, is *online* (i.e., with no delays in incorporating new user input) and *extensible* (i.e., users can fine-tune the ranking objectives used for the recommendation) (Sect. 5);
(3) We propose an efficient approach for storing and retrieving previously executed pipelines, used datasets, and user interactions. These form an Experiment Store that allows us to treat pipeline suggestion as a search problem in contrast to the expensive 'generate-evaluate' iterative process of AutoML solutions (Sect. 6);
(4) We devise a novel methodology for extracting a graph-based semantic representation of a DS pipeline from its source code based on rewrite rules and a knowledge graph. In this way, we can populate the Experiment Store with pipelines from multiple sources and improve recommendations (Sect. 7);
(5) We evaluate DORIAN against several IDA and AutoML baselines. As the baselines can only support quantifiable evaluation processes, we use three common DS tasks, namely classification, regression, and clustering, and the respective evaluation metrics. In addition, we analyze the predictive performance of real users via a preliminary user study. We show that DORIAN significantly outperforms the state-of-the-art IDA tool in both predictive and runtime performance. It achieves accuracy similar to long-running AutoML solutions while being real-time, generic, and extensible (Sect. 9).

2 Motivating use case

Consider the following scenario: a computational biologist (i.e., domain expert) at a research laboratory is working with single-cell gene expression data. Their task is to identify genes that are differentially expressed between two

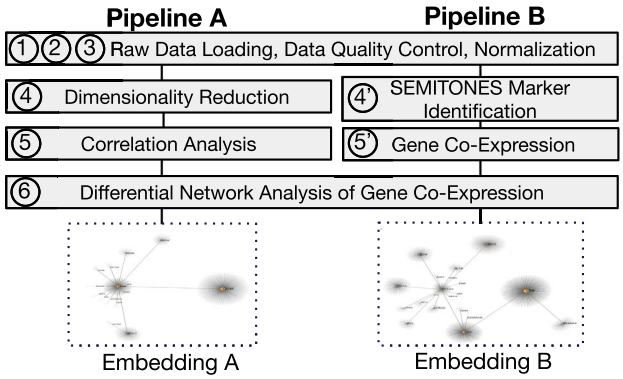


Fig. 1 Example of DS pipelines for RNAseq analysis

conditions—"healthy" and "exposed to a disease"—and play a role in disease development (e.g., cancer).

The researcher starts with pipeline A in Fig. 1: ① conversion of raw data from a sequencer into count matrices, ② frequency threshold filtering for quality control, ③ data normalization, ④ principal component analysis for dimensionality reduction, and ⑤ correlation analysis to calculate gene co-expression scores. ⑥ Then, they create graph embedding A that preserves relations between co-expressed genes [32]. By analyzing the difference in the embedding between the "healthy" and the "exposed to a disease" conditions, the domain expert is able to identify key genes that might be associated with that particular disease. During experimentation, the researcher decides to replace steps ④ and ⑤ with ④' and ⑤' (pipeline B in Fig. 1)—a new domain-specific technique, SEMITONES [53], that performs marker identification (Challenge **1**) and an algorithm for constructing gene co-expression networks. The researcher then compares the output of pipelines A and B in order to identify the most informative embedding and verify the findings. This ad hoc experimentation can be very complex (Challenge **2**) and continues until a suitable pipeline is found. Pipeline evaluation involves consultation with field experts and an exhaustive literature search for supportive evidence that particular genes are associated with a disease (Challenge **3**). In the following, we detail the challenges that make the process of pipeline design overwhelming:

(C1) **Keeping up with the advances in methods and tools**. Many research fields similar to that of single-cell RNA analysis are young, actively-evolving, and produce a variety of experimental domain-specific methods for data pre-processing and analysis at a rapid pace. For IDAs and AutoML tools to support a new technique, the developers (rarely, domain experts themselves) need to update the code base. Yet, this approach cannot sustain the pace of advancement in such research fields. *We, thus, require a solution that can be effortlessly extended to new operators and DS tasks.*

(C2) **Large search spaces**. A large and growing set of available techniques for data analysis and domain-specific tools leads to large search spaces. When domain experts design pipelines manually, a large search space complicates the decision-making process and becomes overwhelming, e.g., due to the high variety of potential changes to make in the pipeline and their order. In addition, large search spaces negatively affect the efficiency of IDAs and AutoML solutions, leading to a combinatorial explosion in the number of choices. *We, thus, require a solution that assists end-users in the process of pipeline design efficiently (ultimately, in real-time) under a growing set of choices.*

(C3) **Absence of quantifiable performance metrics**. In many application domains, the option to specify the training function or a quantifiable evaluation process might be unrealistic. For instance, in this use case, there exists no quantifiable metric to determine which embedding is "better". A single comparison of the results of two pipelines can be inherently manual (e.g., searching for scientific evidence). That renders IDAs and AutoML inapplicable due to the lack of quantifiable loss functions to optimize. Thus, *we require a user-aided solution that supports arbitrary evaluation processes.*

3 Problem statement

We start by defining DS operators and tasks, a DS pipeline and its evaluation process. Then, we formalize the problem: *assisting* end-users with the design of DS pipelines, supporting a large and growing set of DS operators, tasks, and arbitrary evaluation processes.

A DS operator $p \in \mathcal{P}$ is a specific black-box implementation of an algorithm that includes a set of hyperparameters $\{h_1, \ldots, h_q\}$ from the domain H (e.g., the `sklearn` implementation of a missing value imputation algorithm that is parameterized by the imputation strategy, e.g., using the median value of a numeric attribute). Each operator can perform one or more DS tasks $T \in \mathcal{T}$ (e.g., missing value imputation), although it performs exactly one task in the context of any given pipeline. Each DS task, in turn, is defined by a (potentially overlapping) set of operators $P_{T_k} = \{p \in \mathcal{P} \mid T_k \in \text{Tasks}(p)\}$.

A DS pipeline is a directed acyclic graph (DAG) $G(V, E)$. Nodes V comprise a finite set of DS operators $\{p_1, \ldots, p_m\}$. Edges E define the flow of data from one operator to the other. The sets of supported operators $\mathcal{P}$ and DS tasks $\mathcal{T}$ can be

extended, as new DS algorithms are constantly implemented, and domain-specific operators might be required.

The evaluation process $eval(\mathcal{C}, T, D)$ for a set of pipeline candidates $\mathcal{C}$ that solve task T on input data D is, in a broad sense, a sequence of actions—manual or automated—that are required to identify a partial order among candidates $c \in \mathcal{C}$ such that, for any two pipeline candidates $c_g, c_h \in \mathcal{C}$, c_g performs better, worse, or equally compared to c_h. The concept of pairwise comparison of pipeline outputs on arbitrary input data is important as it incorporates evaluation processes that cannot be quantified by a specific loss function. Input data D can be arbitrary (e.g., tabular, graph). A budget B is applied as a termination criterion to limit either the total time spent on comparing pipeline candidates in $eval(\mathcal{C}, T, D)$ or the number of performed comparisons.

Definition 1 (*Problem statement*) Given a DS task T, input data D, pipeline candidates C, evaluation process $eval(\mathcal{C}, T, D)$, and budget B, *assisted design of DS pipelines* is an interactive process of selecting pipeline $c^* \in \mathcal{C}$ that is "best" among all pipelines in $\mathcal{C}$ w.r.t. the evaluation process $eval(\mathcal{C}, T, D)$ under budget B.

The problem formulation does not specify the process of candidate generation, nor does it imply restrictions to the degree of its automation. Thus, it accommodates a wide spectrum of more specific problems in the design of DS pipelines, ranging from AutoML to human-in-the-loop IDA problems (e.g., suggesting the next operator to apply or hyperparameter tuning).

4 Overview

The goal of DORIAN is to provide a solution for the assisted design of DS pipelines that can be easily extended by end-users (C1, e.g., by adding custom DS operators), maintain its efficiency under constantly expanding search spaces (C2), and support arbitrary evaluation processes (C3). With DORIAN, end-users receive interactive recommendations for DS pipelines to solve a user-defined DS task on particular data.

Figure 2 illustrates our proposed framework, comprised of three main components: (1) the *Recommendation Engine* that provides a ranked list of pipeline candidates to solve the DS task at hand, (2) the *Experiment Store* that persists all previous experiments, i.e., DS pipelines that were executed with a particular set of hyperparameters, along with the accompanying artifacts (e.g., data, source code, pipeline specification, meta information), and (3) the *DS Pipeline Extractor* that captures a stringent graph-based pipeline representation directly from source code with little to no overhead from the end-user in order to populate the Experiment

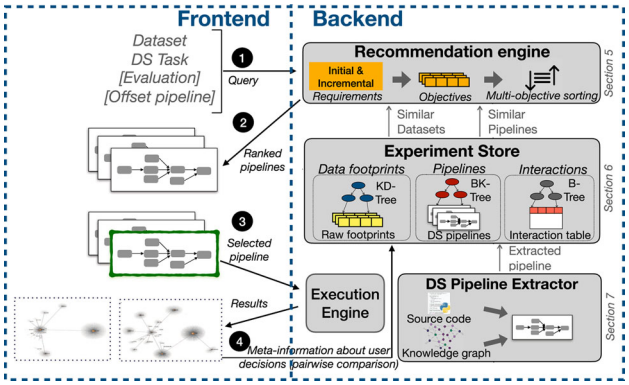


Fig. 2 DORIAN overview: The end-user specifies a query, chooses a preferred DS pipeline, alters it, submits it for execution, and evaluates the results. Based on the user interaction and previous experiments, DORIAN suggests new pipelines

Store and, consequently, improve the quality of recommendation.

In the following, we describe the user interaction flow and outline how it is reflected in the current implementation of DORIAN's user interface (UI). [1] Initially, the end-user specifies a query, i.e., data D and a DS task T ❶. Optionally, they provide (1) the specification of a quantifiable evaluation process (e.g., tenfold cross-validation and AUC ROC score for classification) and (2) the initial DS pipeline to work with if they need suggestions for improvements. This constitutes the beginning of the *User Interaction Cycle*. In the UI, the query form includes file uploads for data and two selectors with auto-completion for the specification of DS tasks and evaluation processes, respectively. The "DS Task" selector allows adding new tasks as text tags, whereas the "Evaluation Process" selector opens a code editor for new evaluation processes. DS pipelines can be created via the visual pipeline composition tool or imported as source code to the *Pipeline Extractor*. In our example from Sect. 2, the biologist uploads data from the RNA sequencer and selects "graph embedding" from the list of supported DS tasks.

Once DORIAN receives the query (D, T), the *Recommendation Engine* (Sect. 5) retrieves pipelines from the Experiment Store that were previously executed on data similar to D (in our example, pipelines that were previously applied on the RNAseq datasets), filters out pipelines that do not perform a specified DS task T and ranks the candidates based on relevance to the user query, giving preference to a wide variety of suggested candidates. The notion of relevance is defined by a list of ranking objectives that identify the order of pipeline candidates. This ranked list of suggestions is then presented to the end-user ❷ (i.e., initial recommendation), who, in turn, can alter the ranking objectives to tailor rec-

[1] For a detailed description of the UI, we refer the reader to the demonstration paper [42] and video (https://dorian.studio/vldb22demo/).

ommendations to their needs. The end-user then can (1) edit candidates by adding, removing, altering operators, or adjusting the corresponding hyperparameters; (2) discard them as irrelevant based on the domain knowledge or personal preference; (3) execute ❸. In the UI, suggested pipelines are visualized as graphs and can be scrolled. Pipeline editing is performed via the visual pipeline composition tool. Respective buttons discard and execute a pipeline. The ranking objectives are presented as an ordered list of objectives (i.e., their textual description) that end-users can re-order or remove, and a selector with auto-completion to add new objectives. Users can implement new objectives in the code editor (with a fixed function signature that defines the interface) if desired ranking objectives do not exist.

Pipeline performance is computed w.r.t. the user-defined evaluation process. In case no quantifiable evaluation process exists, the end-user can compare the pipeline structure and the output results for any given pair of executed pipelines and identify a "better" candidate in accordance with the domain expertise and personal preferences of the end-user. In the UI, the pairwise comparison functionality is implemented as a dedicated window, where two pipeline candidates are visualized as graphs side-by-side together with the visual representation of the results (operators for user-defined textual output and data visualization are supported as part of the pipeline structure). In our example, the computational biologist directly compares two embedding graphs against one another and identifies the most informative graph by conducting an exhaustive literature search and consultations with other field experts. When the end-user specifies a quantifiable evaluation process, the pairwise pipeline comparison is automated. We record the end-user decisions to take user preferences into account and improve suggestions over time ❹.

DORIAN iteratively repeats steps 2-4 until the end-user finds a suitable pipeline (i.e., the recommendation for incremental improvements). In the UI, iterations of suggestions are visualized with pagination, where end-users can revise suggestions across iterations at will, navigating manually through the pages of suggested pipelines. Once the suitable pipeline is found, the end-user can export it back to a DS script if a need for further integration with other downstream applications exists. The export procedure concatenates the source code of every DS operator that the pipeline contains, utilizes the nodes of the pipeline graph to generate function calls and the edges—to generate variables[2]

Note that the end-user can interact with DORIAN to populate the Experiment Store with past experiments, either from local repositories and code bases or from publicly available experiment databases such as OpenML and kaggle (not shown in the figure for simplicity). As the experiments are usually defined with general-purpose programming languages or workflow specification languages, the *DS Pipeline Extractor* is responsible for converting the source code to a semantic graph representation of the underlying DS pipeline.

Algorithm 1: DS pipeline recommendation.

Input: **userQuery** - input data; DS task; optionally, the offset pipeline and the evaluation process;
objectives - the list of ranking objectives; default=empty list;
n - the number of candidates to suggest, default=20;
m - the number of datasets to query, default=3;
Output: **suggestions** - the ranked list of pipeline candidates to suggest

```
1  Initialize rankingScores = empty list; ExperimentStore as store;
2  if userQuery.pipeline is empty then
3      if objectives is empty list then objectives = initialObjectives;
4      similarData, distances = store.kdtree.getDatasets(userQuery.data, m);
           /* (ID, distance) to NN datasets                    */
5      suggestions = store.btree.getPipelines(similarData);
           /* Pipelines previously applied to similarData      */
6  else
7      if objectives is empty list then objectives = incrementalObjectives;
8      suggestions, pdist = store.bktree.getPipelines(userQuery.pipeline, n);
           /* (ID, distance) to NN pipelines                   */
9      foreach candidate ∈ suggestions do
           /* .getDiscarded retrieves pipeline candidates that were previously
              discarded during the user query                  */
10         if candidate ∈ store.interactionTable.getDiscarded(userQuery) or
           candidate.Task != userQuery.Task then
11             suggestions.remove(candidate);
12         else
13             rankingScores.append([objective(candidate, userQuery, store) foreach objective ∈ objectives]);
14     return non-dominated-sorting(suggestions, rankingScores)[:n]
           /* sort candidates based on computed objectives     */
```

5 User-tailored recommendation

The *Recommendation Engine* is at the heart of DORIAN and is responsible for suggesting new relevant pipelines or alterations to existing ones. It addresses the challenges of *selecting* and *ranking* pipeline candidates for a given user query in real-time.

[2] In the cases where DS operators are implemented with different programming languages, the edges of the graph, combined with the information about the data types of operators' inputs and outputs, are used to implement a serialization protocol and pass data across runtimes.

5.1 Recommendation process

The general idea of our proposed recommendation process is first to select a subset of DS pipelines that are *relevant* to the user and then rank them. This way, DORIAN avoids unnecessary computation for ranking DS pipelines that do not match the user's input. DORIAN supports two types of suggestions—*initial recommendation* when the end-user does not specify any initial pipeline at the beginning of the *User Interaction Cycle* and *recommendations for incremental improvements* when the end-user has already selected an offset pipeline as part of the query. The pseudocode of the entire recommendation process is shown in Algorithm 1. Lines 3–6 concern the initial recommendation and line 8—suggestions for incremental improvements. The rest fits both.

Pipeline Candidate Selection. We distinguish two processes for selecting a pool of pipeline candidates: one for the initial recommendation and one for the recommendations for incremental improvements. Given a query (D, T), the *Recommendation Engine* provides an initial recommendation by finding pipelines that were previously executed on *datasets similar to D* and filtering out the ones that do not solve task T (line 11). This selection ensures the exploration element in the recommendation process. For incremental suggestions, the Recommendation Engine selects candidates that are *similar to the pipeline chosen* in the last user interaction cycle (line 9). This ensures exploitation in the recommendation. In both cases, the Recommendation Engine efficiently retrieves similar datasets and pipelines from the Experiment Store (Sect. 6).

User-tailored Pipeline Ranking. Once we have a pool of pipeline candidates, we need to rank them. Initial recommendations exhibit different requirements for ranking than incremental suggestions. This is due to the difference in the amount of information available and the exploration-exploitation trade-off. Despite the different requirements, we frame the ranking of pipeline candidates in both cases as a problem of multi-objective sorting where the user requirements for ranking are mapped to a list of quantifiable ranking objectives, i.e., numerical values. This design decision allows for a user-defined specification of "pipeline relevance" that can be altered between user queries and within a single query over time (to help the end-user better control the exploration-exploitation trade-off). In the following, we formulate this problem and discuss the different ranking objectives in detail.

5.2 Ranking as multi-objective sorting

We formulate the problem of candidate ranking as the problem of multi-objective sorting. We map every pipeline candidate to a small numeric vector that contains the values of the identified objectives (described in the following subsection) in a specified order. A pipeline candidate A is said to dominate over pipeline B if and only if, for every objective, the numerical values of A are greater or equal to the values of B, and there exists at least one value (i.e., objective) where its value is strictly greater. Formally, given a set of pipeline candidates $\mathcal{C}$ and function $f_o(p)$ that returns an array of objectives for pipeline $p \in \mathcal{C}$:

$$\forall p_i, p_j \in \mathcal{C},\ p_i \ dominates\ p_j \iff$$
$$\forall k\ f_o^k(p_i) \geq f_o^k(p_j) \text{ and } \exists k'\ \ f_o^{k'}(p_i) > f_o^k(p_j)$$

where f_o^k denotes the k-th objective returned by f_o.

When comparing two pipeline candidates, not all objectives will be greater than or equal for one candidate to declare domination over another. For this reason, we resort to the problem of non-dominated sorting. Specifically, we utilize the modified Generalized Jensen algorithm [12, 25]. Candidates have rank 0 if they are not dominated by any other candidate and rank i if they are dominated by at least one candidate of rank $i - 1$. Solutions with equal ranks are considered "equally good". The algorithm splits candidates into a sequence of Pareto fronts of equal rank. Ranking within one front is based on domination in the first k objectives.

This approach provides a flexible solution where end-users can alter, tune, and extend the list of objectives in a straightforward way. For example: (1) A user wants to "pin" an operator and prioritize suggestions where this exact operator exists in the pipeline. Then, they can simply add the first objective to be a binary metric that denotes the presence of an operator; (2) A user wants to take the concepts of explainability and fairness of the suggested pipeline candidates into account. The corresponding metrics can be added to the list of objectives as well. Such extensions can be easily implemented with DORIAN. The end-user can adjust the ranking objectives by adding or removing an objective, changing their order, or creating a custom one (e.g., prioritizing candidates that demonstrated low average execution time on similar data or contained a particular set of operators), thus tuning the recommendations. The only requirement is that any objective has to be computed with the available data and used in the specified order. Other solutions, such as learning-to-rank algorithms [31], would require re-training of the classifier and preparation of a completely new training set that represents the ground truth in accordance with newly established objectives. They are, therefore, not suitable for our problem.

5.3 Ranking objectives

To provide a flexible *user-tailored* recommendation engine and support multi-objective ranking, we propose the follow-

Algorithm 2: Ranking objectives.

```
   /* Implements the Objective interface     */
1  objective1 (candidate, userQuery, store):
2      similarData, dist = store.kdtree.getDatasets(userQuery.data,
       m);
3      performance = [getPerformance(candidate,
       store.interactionTable.query(D)) for D ∈ similarData];
4      return weightedAverage(performance, weights=dist);
5  objective2 (candidate, userQuery, store):
6      return getPerformance(candidate, store)
7  objective3 (candidate, userQuery, store):
       /* .getPrevious retrieves the count on
          how often the candidate was suggested
          for the given userQuery                */
8      return store.interactionTable.getPrevious(candidate,
       userQuery)
9  objective4 (candidate, userQuery, store):
10     suggestions, pdist =
       store.bktree.getPipelines(userQuery.pipeline, m);
11     return pdist.query(candidate);
12
13 initialObjectives = [objective1, objective2];
14 incrementalObjectives = [−1×objective3, objective1,
   objective2, −1×objective4]
15
16 getPerformance (candidate, storeSubset):
17     perf = storeSub-
       set.interactionTable.query('preferred'==candidate.id).
18     performance;
       /* compute the percentile rank of a
          performance score relative to a list
          of all performance scores              */
19     percentile = percentile_score(perf,
       storeSubset.interactionTable.performance);
20     preferred = storeSub-
       set.interactionTable.query('preferred'==candidate.id)).
21     length();
22     compared = storeSub-
       set.interactionTable.query('compared'==candidate.id)).
23     length();
24     if preferred + compared == 0 then return percentile
25     else return 0.5 * (percentile + preferred / (preferred +
       compared))
```

ing interface, which can be used to define a set of numerical objectives:

```
rank(candidate, query, store)
 → score
```

where `candidate` is a pipeline candidate to be ranked, `query` is a user query (i.e., data, DS task, offset pipeline, and evaluation process), `store` is a reference object that enables access to the content of the Experiment Store, and `score` is a numerical value returned by the user-defined objective. In the following, we describe the default objectives that `DORIAN` supports and how we represent them as numerical values for both initial and incremental recommendations (see Algorithm 2). Users can easily add new ranking objectives if they implement the above interface and return a numerical value.

Before defining the objectives, it is crucial to devise a metric of pipeline performance that can be used in cases where quantifiable evaluation metrics are absent, as well as across different evaluation metrics, if available. For this, we define the metric of pipeline preference ratio (PPR) as follows (Algorithm 2, lines 16–22). For a given dataset d and pipeline p, PPR(p, d) is a ratio between the number of times p was preferred by any end-user as a superior candidate during pairwise comparison on d and the number of times p was used with d in total. When the evaluation process specifies a quantifiable performance metric (e.g., ROC AUC score), we compute PPR as a percentile of the candidate's performance, given the performance distribution that other pipelines exhibit on the same data (Algorithm 2, lines 17–18). In the majority of cases, two ways to define PPR (with or without quantifiable performance metric) are mutually exclusive—we do not need to rely on pairwise comparison when a quantifiable performance metric exists. To support all use cases, we compute the average between the leaderboard percentile (scaled from 0 to 1) and the fractional notation of PPR (each defaulting to 0 when no signals exist to compute the metrics).

Initial Recommendations. Initial recommendations constitute the beginning of the *User Interaction Cycle* where no offset pipeline is specified in the user query. This case provides the least amount of information to determine candidate ranking. Algorithm 1 (lines 3–6, 10–14) depicts the procedure for initial candidate recommendation. We define two objectives of the initial recommendation:

(1) We first choose to prioritize pipeline candidates that were previously applied to similar data. Among those pipelines, we prioritize the ones that performed relatively better w.r.t. the chosen evaluation process. Suggesting candidates that performed well in the past on similar data is a common strategy that has been demonstrated to work well in practice [50], even if no theoretical guarantees exist: pipelines executed on similar data tend to yield similar predictive performance. Based on this requirement, the first objective is the weighted average of the pipeline preference ratio (PPR) normalized by the dataset similarity score. The weights are the negative Euclidean distances between a fixed-sized numeric representation of the given dataset and each similar dataset (Algorithm 2, lines 1–4):

$$O_1(p, d) = \sum_{d_{\text{sim}}}^{D_{\text{sim}}} \text{PPR}(p, d_{\text{sim}}) * \frac{1 - \text{MSE}(d, d_{\text{sim}})}{\sum_{d'_{\text{sim}}}^{D_{\text{sim}}} \text{MSE}(d, d'_{sim})}$$

where p is a pipeline candidate, d is a footprint of the query data, D_{sim} is a set of footprints of similar datasets

found in the Experiment Store, and MSE is a mean-squared error.

(2) Furthermore, if the number of candidates that performed well on similar data is low, we prioritize pipeline candidates with better performance in general as a proxy metric for robustness and potential generalization of a particular DS pipeline to a wide variety of tasks. This leads to the following objective, which calculates the average PPR ratio for a given pipeline on any datasets that it was previously applied (Algorithm 2, lines 5–6):

$$O_2(p) = \sum_{d \in D} \frac{\text{PPR}(p, d)}{\|D\|}$$

where p is a pipeline candidate, D is a set of data footprints for the datasets that p was previously applied to, $\|D\|$ is the number of those datasets. This objective ensures that highly similar pipelines can still be suggested to a query with not-so-similar data.

Incremental Recommendations. The procedure for recommending incremental improvements is depicted in Algorithm 1 (lines 8–18). Note that pipeline candidates that were previously discarded by the end-user (within a single user query, line 11) or the *Execution Engine* (in case of raised exceptions) are removed from the set of candidates and do not get any rank assigned. When choosing a pipeline candidate from the pool of suggestions, more user input becomes available—pipeline candidates chosen, altered, discarded, executed, and compared to one another. Thus, the ranking objectives shift from exploration to exploitation, prioritizing pipeline candidates that are relevant to the immediate past of the user interaction rather than the "good on average" ones. In that case, additional requirements and, thus, objectives become necessary:

(1) We prioritize pipeline candidates that were not previously suggested to the end-user. Consequently, all the suggested and never chosen candidates have to be moved lower in the ranking. This requirement leads to the following objective (Algorithm 2, lines 7–8):

$$O_3(p, d, t, e) = -1 * \| \, i \in \text{it} \mid \text{i}_{data} = d \wedge \text{i}_{task} = t \wedge \text{i}_{eval} = e \wedge \text{i}_s = p \|$$

where p is a pipeline candidate, $\|\|$ denotes the set size, d, t, e are the query data, task, and the evaluation process respectively, it is the Interaction Table (Experiment Store), i_s is a suggested pipeline.

(2) We prioritize suggestions that constitute smaller, incremental updates to the offset pipeline candidate—ultimately, one atomic change at a time: addition, replacement, or removal of an operator— to streamline the pipeline comparison process. Suggesting pipeline candidates with significant changes would increase the complexity of consequent root-cause analysis and overwhelm the users. To achieve this, we use the graph edit distance between the pipeline candidate c and the offset candidate off_c (Algorithm 2, lines 9–11):

$$O_4 = -\text{graph\_edit\_distance}(c, \text{off\_}c)$$

As we must ensure a wide variety of suggestions that go beyond the most "popular" pipelines, we use O_4 to promote exploration. When a solution (i.e., pipeline candidate) exists that none of the aforementioned ranking objectives helped discover, a wide variety of recommendations increases the chances of finding it. O_4, in turn, allows suggesting alterations to the offset pipeline regardless of the observed predictive performance of the altered variant.

6 Storage and search of DS experiments

To enable a real-time Recommendation Engine that makes useful suggestions, we store and index information from all previous DS experiments, i.e., DS pipelines together with the datasets used, evaluation metrics (if available), and user interactions into an *Experiment Store*. This component tackles the following main challenges: (1) how to store a large set of datasets in a space-efficient manner, which at the same time allows for efficiently performing similarity search, (2) how to efficiently perform similarity search on the graph-based representation of DS pipelines so that online suggestions (i.e., < 2 seconds to render [35]) are possible, and, (3) which information from the user interaction to store so that comparison of DS pipelines without quantifiable metrics is also possible. To make things harder, these challenges must be tackled on the premise of a constantly growing set of DS experiments. In the following, we detail our solution to each challenge.

6.1 Storing and finding similar datasets

Storing all the datasets used in the experiments would require significant storage capacities. For this reason, we choose to store only a numeric vector of meta-features that serves as a lightweight *data footprint* for previously used datasets. The benefit of doing so is twofold: data footprints are space-efficient and can be used to efficiently search for similar datasets that past experiments applied to. We compute a small set of meta-features, which are a subset of the ones used in the `auto-sklearn` library [21]:

- *Generic* features include the number of instances, columns, numeric and categorical features, the inverse

dataset ratio (i.e., the number of instances divided by the number of columns), in ordinal and logarithmic scales;
- *Statistical* features include aggregated skewness and kurtosis for numeric attributes, represented as the mean, minimum, maximum, and standard deviation of the vector over all numeric attributes of the dataset;
- *Target-specific*[3] features include the number of classes, the maximal class probability—the number of occurrences of the most frequent class divided by the number of instances;
- *Landmark-based** features are performance metrics of relatively simple ML algorithms that are run over the dataset under analysis and used as baselines. We use Linear Discriminant Analysis, Naive Bayes, Decision Tree, Decision Stump (i.e., decision tree with the depth 1), and 1NN (kNN classifier with $k = 1$) as landmarks.
- *PCA-based* features that are based on Principal Component Analysis include skewness and kurtosis of the first principal component and the fraction of principal components that "explain" 95% of the dataset variance.

The rationale behind the selection of these meta-features is twofold. First, they are quantifiable characteristics of the task: They represent the signals of pipeline scalability (number of instances), data imbalance (ratio of the least frequent class to the most frequent class), feature normality and informativeness (Skewness, Kurtosis, Entropy) [50]. Second, their compute time is suitable for real-time user interaction, i.e., < 2s.

For large-scale datasets, even Landmark- and PCA-based meta-features can take longer time to compute. In this case, we use a partial dataset footprint and update pipeline recommendations when the computation is completed. To get a partial footprint in real time, we compute independent meta-features in parallel and sort them in the order of faster compute times. Upon any updates in a data footprint, we repeat Algorithm 1 in an incremental manner: We take the computed list of pipeline candidates, identify new candidates that were not suggested in the previous version of recommendations, and update the ranking by fitting new pipeline candidates into the existing Pareto fronts instead of re-running from scratch.

Simply storing dataset footprints as numeric vectors is not enough because similarity search would be very inefficient: For every new dataset, we would need to (a) compute n Euclidean distances from this dataset to the n other datasets already in the Experiment Store, (b) sort the distances, and (c) select the first k datasets. Such a strategy has an average complexity of $O(nlogn)$, which can be very impractical for real-time searches. We thus choose to utilize KD-trees [5] as an index structure for dataset similarity search that utilizes the Euclidean distance between a pair of dataset footprints as a similarity score. When the end-user comes with a new dataset, we compute its dataset footprint and insert it into the KD-tree. To retrieve datasets similar to the input data, which is required for the initial recommendations, the Experiment Store queries the KD-tree for k nearest-neighbor (kNN) dataset footprints.

This data structure is optimized for kNN search in multidimensional spaces, making it attractive for similarity search of fixed-length numeric vectors. For k-dimensional data, each tree node of depth d acts as a hyperplane that splits a set of data points into two equal-sized disjoint subsets. That is done by selecting a median data point by means of projection onto the axis $d \mod k$.

KD-Trees enable us to provide efficient similarity search[4] without adding storage overhead (i.e., $O(n)$ space complexity) thanks to its $O(logn)$ average complexity for search, insertion, and deletion. Heterogeneity of features (i.e., values of varying magnitudes) can lead to the phenomenon where low-magnitude values have a negligible effect on the distance compared to the high-magnitude values. In other words, when searching for similar datasets, the similarity of high-magnitude values might become "more important" than the similarity of low-magnitude values. To mitigate this, we use Min-Max scaling before computing the Euclidean distance.

6.2 Storing and finding similar DS pipelines

To store previously executed DS pipelines and enable efficient search among them, we make the following design decisions. First, we represent each pipeline as a directed acyclic graph (see Sect. 3) where any DS operator consists of three elements: the specification of DS tasks that this operator is designed to perform (e.g., supervised classification), the reference to the physical, black-box implementation of the operator (e.g., the `RandomForest`), and the values of hyperparameters that the operator can be configured with (e.g., the number of estimators `n_estimators=40`). Figure 3 illustrates four DS pipelines as graphs. Each operator node can have multiple inputs, parameters, and outputs. Edges between the operator inputs and outputs represent data flow. Edges between the operator and the parameter map the hyperparameter value. Note that the graphs are simplified for illustration purposes. They are conceptually similar to dataflow and task graphs, and depict functions, their dependencies on data, parameters, and semantic anno-

[3] This feature requires a target column, e.g., the labels for supervised classification. When the target column is not provided, e.g., in dimensionality reduction, the value of this meta-feature is initialized with zero to keep the length of the vector fixed.

[4] In our preliminary experiments, our KD-Tree-based solution performed up to 3 orders of magnitude faster compared to the naive approach.

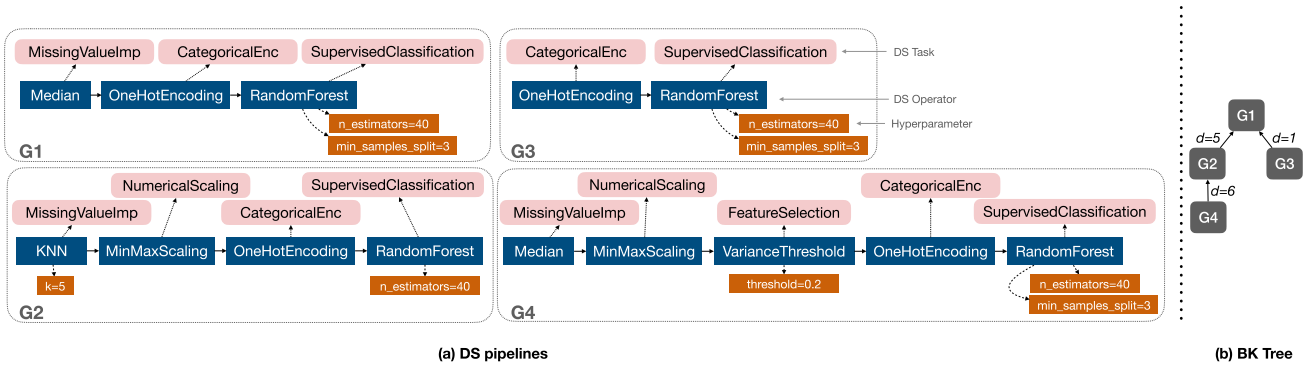


Fig. 3 **a** Pipelines $G1 - G4$ consisting of logical operators (pink), physical operators (blue), and hyperparameters (orange). **b** BK-Tree populated with $G1 - G4$ in the direct order

tations. Each function is defined with a code snippet written in a given programming language and, thus, is as expressive as the grammar of the language allows. DORIAN's *Execution Engine* then executes operators by forwarding their code snippets, data, and parameters to an execution runtime of that programming language. Therefore, the expressiveness of DORIAN's pipeline specification can accommodate pipelines of varying complexity, including production machine learning applications.

Second, to support efficient similarity search over DS pipelines, we utilize the discrete graph edit distance [2] as a measure of similarity between pipelines and the Burkhard–Keller tree (BK-Tree) [11] as an index to accelerate the search process. The graph edit distance encodes the minimal changes required to transform graph A to graph B. Given a set of DS pipelines, every node of the BK-Tree corresponds to a pipeline, and an edge in the tree depicts the discrete graph edit distance between two pipelines. The root node of the tree can be chosen arbitrarily. Every newly added pipeline is recursively moved down the tree by following the path of matching distances (i.e., edges with the distance value equal to the distance between the newly inserted pipeline and the parent node). A new subtree is created if no edge with a particular distance value exists. Pipelines with distance 0 are equal.

BK-trees require $O(n)$ space complexity and $O(logn)$ average complexity for search, insertion, and deletion. In practice, the time complexity depends significantly on the complexity of the distance measure and the choice of the tolerance value. The tolerance value is applied to each node to limit tree traversal to children with distances within the range $[d - tol, d + tol]$, where d is the distance between the query pipeline and the node pipeline, tol is the tolerance value. This step prunes children nodes that lead to sub-trees with "dissimilar" pipelines. As computing the exact graph edit distance is proven NP-hard [57], we utilize the DF-GED [2] algorithm for computing the exact graph edit distance that dynamically generates a tree-based space of all possible mappings between two given graphs and applies the depth-first search algorithm to find a mapping with the least number of edits. One property of this algorithm is that, depending on the input graphs, it can dynamically yield sub-optimal mappings with non-increasing edit costs. That is, a mapping that corresponds to the exact but not necessarily minimal edit distance. We turn this property to our advantage and return the first, sometimes sub-optimal, edit path with the "not-necessarily minimal" edit distance. This design decision allows us to improve search performance without sacrificing the quality of the search results, as computing the exact minimum graph edit distance is not a critical requirement for our approach.

Figure 3(b) depicts an example BK-tree of the 4 pipelines shown in Fig. 3(a). When $G4$ is added to the tree $G1 - G3$, it is first compared against the root node $G1$. Graphs $G1$ and $G4$ have graph edit distance $d = 5$ because $G4$ contains two new operators (min–max scaling and variance threshold feature selection, one new hyperparameter node ($threshold$ is set to 0.2). As an edge with $d = 5$ already exists for $G2$, the insertion algorithm creates a new node $G4$ with the depth of 2 and parent $G2$. The edge $G2 - G4$ gets the graph edit distance $d = 6$ (change to the missing value imputation step, addition of the feature selection step, addition of the $min\_samples\_split = 3$ hyperparameter to the Random Forest estimator).

6.3 Storing user interactions

DORIAN stores user interactions in the *Interaction Table* (IT). This is used by the Recommendation Engine to compute the ranking objectives for any pipeline candidate that is potentially relevant to the user query. Importantly, the interaction table stores the results of *pairwise* comparisons between two pipeline candidates that the end-user executes. This is crucial for cases where quantifiable performance metrics are non-existent and for personalizing recommendations. Below, we illustrate the information kept in IT. Importantly, for each pairwise comparison, it records the pipeline candidates being

suggested, compared, as well as the decision of which candidate is preferred by the user.

```
IT schema:
  dataset_id    | References Data
  task_id       | References Tasks
  discarded_id  | References Pipelines
  compared_id   | References Pipelines
  preferred_id  | References Pipelines
  user_id       | References Users
  eval_id       | References Eval
                     Processes
  performance   | Real (metric type
                      in eval)
```

7 DS pipeline extraction

The Experiment Store gets continuously populated with DS pipelines that end-users choose to execute. Yet, we need to find a solution for the *cold-start problem*, where the Experiment Store is initially empty and, thus, `DORIAN` cannot recommend pipelines. One way to tackle this is to incorporate AutoML operators to `DORIAN` itself so that multiple pipelines can be generated, executed, and stored. Although this is possible in `DORIAN`, it is not enough because it will lead to a very low variety in the pipelines and, thus, affect recommendations.

We, thus, propose an additional solution: The general idea is to allow end-users to populate the Experiment Store with their own initial pipelines or with pipelines that are publicly available in sites such as `OpenML` [51], `kaggle`, or `Kipoi` [4]. However, DS pipelines are usually expressed by means of a general-purpose programming language, such as `python`, `R`, `julia`, or a specific workflow specification language (e.g., `snakemake`). To populate these pipelines in the Experiment Store and make meaningful recommendations, we need to extract their graph-based representation from the source code. This extraction process should take into account not only the syntax of the pipelines but also their semantics (e.g., type of DS task a function performs). To achieve this goal, `DORIAN` provides a *Pipeline Extractor* which proceeds in three steps (see Fig. 4). The first two steps aim at extracting a syntactic language-agnostic representation of the pipeline using a language parser and graph rewrite rules (Sect. 7.1). The third step semantically enriches the previously constructed graph based on a knowledge graph (Sect. 7.2). Figure 4 illustrates an example of the extraction process that we will use in the following.

7.1 From source code to pipeline graph

Given the source code of a DS pipeline, we first use a language-specific parser to extract the Abstract Syntax Tree (AST) ❶. Then, we leverage graph rewrite rules to convert a language-specific AST to a language-agnostic graph that represents the data flow of the DS-related computations and their corresponding hyperparameters (as defined in Sect. 3). Such rewrite rules are, for instance, deleting any language-specific control flow nodes from the tree. Each rewrite rule comprises two parts: the left-hand side that depicts a pattern represented as a DAG structure (being as expressive as the grammar of the underlying programming language), and the right-hand side that depicts a set of procedural commands to alter the graph, e.g., add, remove, update a node, edge, or attribute (i.e., a complete set of operations required to construct any DAG structure). We also implement a regex-based comparator for nodes and edges, enabling a more succinct and reusable rule definition. However, this functionality is used merely for convenience and does not expand nor restrict the generic method of pattern matching and the rewrite rules. The rewrite rules are applied in direct order, as applying one rule changes the graph and, hence, affects the matches of downstream rules. We specify the initial order of the rewrite rules based on their inter-dependencies but enable the end-user to change this order by adding new custom rewrite rules. To support complex multi-line DS scripts, we use a set of "dependency management" rewrite rules that generate edges between variable assignment nodes and variable references. That allows the *Pipeline Extractor* to parse code snippets without conceptual restriction. For code snippets with repetitive variable assignments (i.e., not in the Static Single-assignment Form) where different content is assigned to variables of the same symbolic representation throughout the source code, we provide a rewrite rule that keeps edges to the most recent assignment in code (based on the order of graph traversal and sequential application of rewrite rules) to prevent the "reference before assignment" edges. Currently, we have 10 python-specific rewrite rules and 4 `sklearn`-specific rules that process compositional interfaces for pipeline declaration. `DORIAN` parses data science scripts that consist of up to 25 expressions in under 2 seconds, and more complicated scripts of 100 expressions—under 5 seconds. Note that most rules (apart from "dependency management" and alias resolution for imported functions) are confined to a single expression and executed concurrently.

The DAG structure embraces the *Composite* design pattern that allows to treat a single DS operator `X` as a pipeline with one step (i.e., an operator can be treated as a pipeline), as

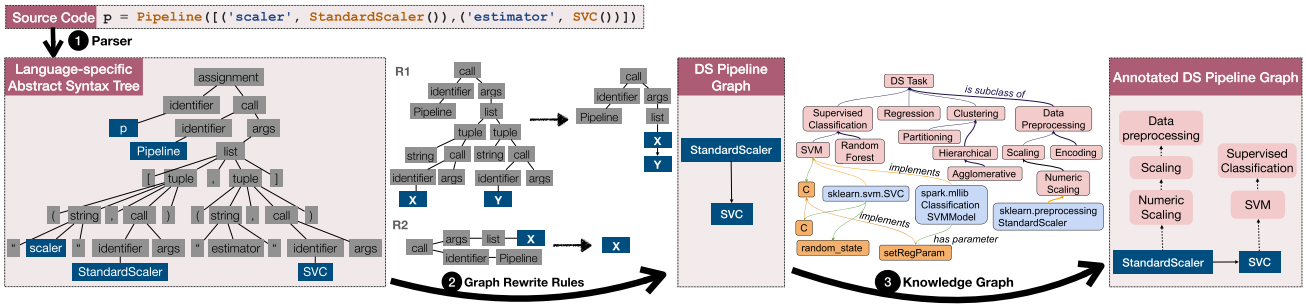


Fig. 4 Extraction of an example DS pipeline from source code. DORIAN parses the textual representation with a language-specific parser, transforms it to a graph with rewrite rules, and semantically enriches the DS graph with a knowledge graph

well as a multi-step pipeline can be treated as a complex operator. That allows end-users to compose (i.e., group together) multiple DS operators that, in the context of a pipeline, belong to one atomic DS task, e.g., a definition of a feed-forward neural network where each layer of the network is declared separately (i.e., with a dedicated "Add Layer" operator). In the UI, the *Composite* principle is represented as a drill-down visualization where a high-level operator is shown, and its detailed representation (i.e., dependent operators) can be shown or hidden on demand.

To support generic pattern matching, these rules also utilize wildcards to "ignore" parts of the subgraph (e.g., variable names or syntactic sugar of a particular programming language). Based on the rewrite rules and a graph pattern-matching mechanism, DORIAN generates the DS pipeline graph ❷. Specifically, we utilize Sesqui-pushout rewriting [17] as the *deterministic* approach to graph transformation that supports "precedence of [node] deletion over preservation"—a valuable property when it comes to graph transformation by deletion; and VF2 [16]—a *deterministic* mechanism for pattern matching and (sub)graph isomorphism testing that supports the graph-subgraph isomorphism problem (in our work, patterns mostly represent subgraphs of a pipeline and rarely involve the pipeline as a whole), is efficient even for large-scale graphs and has reduced memory requirements compared to the related work [15, 33].

Figure 4 (step ❷) depicts two example rewrite rules for the `scikit-learn` ML library. Here, the `Pipeline` class simply denotes pipeline composition – chaining multiple DS operators together in a sequence. Hence, when the full name `sklearn.composition.Pipeline` is matched in the AST, rewrite rule `R1` simplifies the graph by removing purely syntactic nodes and chaining two operator functions together. When the pipeline has more than two operators, the rewrite is applied recursively, one per match, until all operators are chained. Rewrite rule `R2` simplifies the graph by removing the `Pipeline` node and other purely syntactic nodes. The final pipeline contains two operators: the standard scaling mechanism for normalization of numeric data and the SVM classifier.

7.2 Semantically annotating pipeline graphs

Once we get the syntactic DS pipeline graph composed of physical operators and their hyperparameters, we need to annotate each operator with its corresponding DS tasks. This annotation provides semantic information about the type and, thus, capabilities of each operator (e.g., whether it performs a scaling transformation or builds an SVM model). This information is crucial when searching for similar DS pipelines and leads to better recommendations. In contrast to Patterson et al., 2018 [40] who propose an approach for semantically enriching DS programs using dynamic code analysis, we focus on static code analysis only, as it does not require executing a DS script. As most DS pipelines take a significant amount of time to produce the results, it is very inefficient to extract the corresponding graph. Given the scale of thousands of DS programs that one would like to process to populate the Experiment Store from publicly available sites (`OpenML`, `kaggle`), using an approach of Patterson et al., 2018 [40] is impractical.

To enable this semantic enrichment, we construct a knowledge graph that provides information on DS operators. The knowledge graph contains a hierarchical categorization of known DS tasks. An excerpt of our constructed knowledge graph is shown in Fig. 4. Each node in the hierarchy (pink nodes) represents a concept of a DS task, while an edge denotes a generalization/specialization relationship. At the leaves of the hierarchy, we place specific classes of algorithms, e.g., `SVM`. These can be seen as logical operators in database terminology. We then populate the hierarchy with functions of publicly available DS frameworks[5] (blue nodes in Fig. 4). For instance, `sklearn.svm.svc` is a specific implementation of an SVM algorithm provided by the `scikit-learn` library. These operators can be seen as physical operators in database terms. In addition, the knowl-

[5] Our preliminary experiments show that 93% of all function calls in real-world DS scripts belong to libraries and frameworks, whereas only 7% correspond to user-defined functions, lambda expressions, and other objects.

edge graph contains the implementation-specific information and hyperparameters of the operators' physical implementations (orange nodes in Fig. 4). By matching the nodes of the intermediate language-agnostic DS graph with the physical operators on the knowledge graph (blue nodes), we get a semantically annotated DS pipeline graph ❸. The flexibility provided by knowledge graphs allows us to handle cases where certain hyperparameters belong only to a specific implementation and not to the algorithm. We manually curate the knowledge graph, leveraging the involved contributors' domain expertise and scientific literature. We believe full automation of this task would be impractical and would still require curation via feedback loops or voting. The user interface allows end-users to add new DS tasks locally (on a per-account basis) and extend the centralized ontology globally, pending a review.

Extension of known implementations of DS operators, on the other hand, is done semi-automatically based on web crawling of documentation websites. We implement crawlers per website and reuse existing "building blocks" when possible, as the documentation of data science frameworks and libraries is oftentimes generated automatically by code annotations and comments.

At the time of writing, we crawled *sklearn* and *pandas* as two core libraries that handled ML and data management tasks, respectively. When a new DS operator is introduced in the Experiment Store, the minimal necessary input from the end-user is the mapping between the full name of the operator's implementation (e.g., `sklearn[v0.24].preprocessing.MinMaxScaler`) and its corresponding DS task (e.g., data preprocessing, normalization of numeric data). This meta-information immediately becomes available to other users, keeping the manual overhead low thanks to the collaborative environment of the Experiment Store.

8 Extensibility

In this section, we discuss DORIAN's extensibility. Namely, we convey how end-users can add new ranking objectives and functionality to support new DS tasks, operators, and evaluation processes. In addition, we discuss the ability of supporting multiple programming languages used in DS scripts for converting them to DORIAN's graph representation.

Ranking objectives. Although we propose a particular list of ranking objectives for the initial recommendations and the recommendations for incremental improvements, new user-tailored ranking objectives can be added locally and shared across other end-users using the interface we introduced in Sect. 5. Example use cases for new ranking objectives are as follows:

- An end-user wants to "pin" a particular DS operator or a part of the pipeline in order to prioritize suggestions where this exact part exists in the candidate. The user can simply add the first objective to be a binary metric that implements sub-graph matching to depict the presence of the operator in a pipeline candidate;
- An end-user wants to incorporate a cost-based model for estimating the pipeline execution time and prioritize pipeline candidates that are faster to compute;
- An end-user wants to explore pipelines that are frequently selected by other end-users when querying similar data or in general. This can be done by applying an SQL-like query to the relational Interaction Table and computing the count of a given pipeline identifier in the attribute `selected`.

DS tasks. As DORIAN treats the specification of data science tasks as a tag annotation of a DS operator, the end-user can add a new DS task in two ways: (a) to add a new DS operator and annotate it with the string-based tag that depicts a task that did not exist in the Experiment Store before, or (b) to extend the knowledge graph directly via the user interface. The latter option is preferred because the hierarchical relationships between the sub-tasks can be curated (i.e., a new DS task can be connected to the existing tasks by one of the supported relationship types), which, in turn, improves the quality of the *DS Pipeline Extractor*.

DS operators. DORIAN represents a physical DS operator as a black-box implementation of an algorithm that is parametrized by a set of hyperparameters. Therefore, each DS operator has the following specification:

- Full function name and signature, i.e., the number of data inputs and outputs, their corresponding names and data types, and the corresponding hyperparameters with their name, data type, default value, and, optionally, the domain of values that this parameter accepts. If the domain is not specified, DORIAN curates it automatically by querying the Experiment Store and searching for other pipeline candidates that previously applied this operator. The record of each physical operator exists in the knowledge graph and is connected by the relationships "implements", "has parameter", and "is equivalent to" to the specification of logical algorithms that these physical operators implement;
- The annotation of corresponding DS tasks that the DS operator performs which is linked within the knowledge graph either directly (the "task-operator" link) with the relationship "implements", or via the corresponding algorithm (the "task-algorithm-operator" link);

- The operator's implementation as a function with the signature that matches the specification stored in the knowledge graph.

Note that, in the cases when DORIAN uses the *DS Pipeline Extractor* to retrieve the graph-based pipeline representation, new operators can be added automatically by means of static code analysis, where the only remaining step for the end-user is to provide the annotation of DS tasks that the operator performs as it cannot be detected automatically.

Evaluation processes. Evaluation process for a set of pipeline candidates is a particular case of a DS operator with a distinctive interface—it takes a set of pipeline candidates, specification of the DS task, and the dataset under evaluation. In the majority of cases, the evaluation process assigns a numeric score to each pipeline candidate that corresponds to its predictive performance w.r.t. the given evaluation logic (e.g., k-fold cross-validation for supervised classification and the ROC AUC score as the performance metric). In some cases, however, there are no quantifiable measures of predictive performance (e.g., as in our running example). Then, by receiving a set of pipeline candidates, the evaluation process returns a partially ordered list that depicts relative predictive performance without assigning objective scores. Implementation of evaluation processes, apart from following the specified programming interface, is analogous to the implementation of DS operators—it is a black-box function with a restricted function signature.

DS scripts languages. In practice, end-users build DS pipelines as a composition of tools from various libraries, frameworks, and languages. For this reason, the DS Pipeline Extractor allows end-users to add new languages by simply specifying their grammar and extending a set of graph rewrite rules for additional language constructs (e.g., syntactic sugar, loops, etc.). Note that the only restriction on a language that DORIAN imposes is that it has a context-free grammar that can be represented in the Extended Backus–Naur form [19]. We consider this condition reasonable as context-free grammars have sufficient expressiveness to describe the recursive syntactic structure of many languages, including workflow specification languages, such as `snakemake` [36] or infrastructure automation tools, such as `ansible` [24].

9 Evaluation

The goal of DORIAN is to assist users with the design of DS pipelines, especially in cases of unspecified evaluation metrics and domain-specific operators, as discussed in Sect. 2. Evaluating DORIAN in such cases is very challenging because it would require extensive user studies based on a large pool of domain experts. In addition, a comparison with AutoML and IDA approaches under non-quantifiable use cases would not be possible. As we do not have the resources for such a user study and would like to compare our solution to existing AutoML and IDA tools, we choose to evaluate both the effectiveness (i.e., predictive performance) and efficiency (i.e., execution time) of DORIAN on three common DS tasks using their corresponding evaluation metrics and compare it against both automated (Sect. 9.2) and the human-in-the-loop baselines (Sect. 9.3). We conducted a preliminary user study with a few data scientists and a qualitative interview and reported our findings (Sect. 9.4). Moreover, we evaluate DORIAN's extensibility, the diversity of rendered recommendations, and the effects of the cold-start problem (Sect. 9.5).

> In summary, we found that DORIAN performs better in predictive performance than human-in-the-loop baselines and is similar to long-running automated approaches while it requires shorter execution time. At the same time, DORIAN is generic to fit arbitrary evaluation processes, adapt to new operators, and provide user-tailored recommendations.

9.1 Experimental setup

Hardware, DS tasks, and datasets. We used an Ubuntu workstation with 8 Intel i7-8550U CPU cores (1.80GHz) and 24Gb RAM. As we cannot evaluate DORIAN with domain-specific tasks, operators, or evaluation processes, we evaluate it using three common DS tasks, namely supervised classification, regression, and clustering, and their common evaluation metrics. For supervised classification, we use 63 datasets and 3900 pipelines of the `OpenML-CC18` benchmark suite [8] and choose ROC AUC score as the performance metric. For regression, we use 33 datasets from the `OpenML` AutoML Regression Benchmark (OpenML ID: 269). As `OpenML` contains only 300 regression pipelines available, we stored the intermediate DS pipelines that the AutoML baselines generated in the Experiment Store resulting in 1000 pipelines in total. We choose the mean absolute error as the performance metric for regression. For clustering, we use 40 datasets from `OpenML` and 1000 pipelines (also selected from the intermediate results of the AutoML baselines, as no clustering pipelines existed on `OpenML` at the time of writing). We choose the silhouette score as the performance metric.

For all of the collected pipelines, we utilized the DS Pipeline Extractor to convert DS scripts from different users and sources to the graph-based pipeline representation. For human-in-the-loop baselines, we needed to select a small number of datasets (10) that would make the preliminary user study tangible. To ensure a fair and realistic evaluation, we chose all datasets to be as diverse as possible. We achieved

this by running the incremental farthest neighbor search algorithm [41] over the data footprints of all available datasets. As the outcome of this algorithm depends on the initial seed (i.e., the first randomly selected dataset), we iterate over all available datasets and select a sample with the largest mean Euclidean distance. This procedure aims at selecting datasets with the most distinct data footprints and, hence, the highest variety.

Baselines. We use (1) the `DABL`[6] library for baseline pipeline generation that applies a static set of pipeline synthesis rules to generate a pipeline for a given task, (2) the popular `auto-sklearn` [21] AutoML solution using two variants: with the total execution budget of 5min and 1h respectively, (3) the `Ray Tune` [28] library as a state-of-the-art hyperparameter tuning solution, (4) the `RapidMiner Auto Model` as the state-of-the-art IDA solution, (5) a synthetic "oracle" baseline that imitates the end-user that knows the optimal path to the best-performing pipeline candidate, and (6) `FLAML` [7]—a fast AutoML solution that optimizes the cost of training as well as predictive performance. As `Ray Tune` comprises a framework with hyperparameter optimization algorithms and not a full-fledged AutoML solution, one has to define the set of algorithms and the domain of hyperparameters that the framework will consider during the search. Therefore, for supervised classification and regression, we define `Ray`'s search spaces equivalent to that of `auto-sklearn`. For clustering, we use the common algorithms supported by `sklearn`[8] with the values of the hyperparameters suggested in the documentation. Note that not all baselines support all three DS tasks, e.g., `DABL`, `auto-sklearn`, and `FlaML` do not support clustering.

`DORIAN`. We implemented `DORIAN` in Python and showcased it in [42]. We run three variants of `DORIAN`: with simulated user behavior, with a simulated oracle user, and with real users as part of a preliminary user study. Simulated variants are necessary for comparison against baselines on a large set of queries. To simulate user behavior, we apply an exhaustive (i.e., does not "skip" suggestions) breadth-first search strategy: We first execute pipeline candidates from one iteration and then recursively choose the suggestions for incremental improvements in the direct order. We choose breadth-first search because it fits a common decision-making pattern of evaluating all available options first and then selecting an attractive candidate to drill down. Note that the breadth-first search algorithm is exhaustive—e.g., n suggestions in the first iteration lead to n new iterations, n suggestions each, totaling n^2 suggestions in the second iteration, and so on. Parameters k for kNN similarity search are system constants and set to 3 and 20 for the datasets and pipelines, respectively. We study these values in Sect. 9.6.

Evaluation strategy. As `DORIAN` bases suggestions on the information about past user queries and accumulated DS experiments, we need to synthetically exclude all the meta-information about the user query under evaluation from the Experiment Store to ensure the validity. For each task, we apply a leave-one-out scheme where the experiments related to all but one dataset are kept in the Experiment Store, and the one dataset with all corresponding pipelines specifies a previously unseen user query (i.e., benchmark dataset). We use all pipelines executed on the benchmark dataset and their corresponding predictive performance metrics as ground truth. We choose fivefold cross-validation as the evaluation process $eval(C, T, D)$ for AutoML solutions on benchmark data. When evaluating the quality of `DORIAN`'s suggestions, we take the maximal predictive performance across suggested pipelines as the performance of the best-found candidate.

9.2 Comparison against AutoML baselines

Evaluation scenario. We use the evaluation strategy on all the datasets as specified in the setup.

Results. Figure 5 depicts the average predictive performance and total execution time of the baselines on all the datasets for supervised classification, regression, and clustering. `DORIAN`, on average, performs on the level of automated baselines, outperforming `DABL` and `auto-sklearn 5min`. Although the state-of-the-art AutoML baselines can show slightly better predictive performance to `DORIAN`, they take more time to execute (1 h compared to 0.5−2.5 min that `DORIAN` needs). For example, for classification `DORIAN` achieves an average ROC AUC score of 97% in few seconds while `auto-sklearn` requires 1 hour to arrive to an average ROC AUC score of 99%. In addition, AutoML solution require more resources while `DORIAN` can provide suggestions to the end-user asynchronously to (or even without requiring) the execution of pipeline candidates. `DORIAN simulated user` by design cannot outperform the `oracle user` baseline, as both baselines draw suggestions from the same set of pipelines, but `DORIAN oracle user` simulates the end-user who knows how to find the best performing DS pipeline. However, `DORIAN simulated user` reaches predictive performance that is comparable to the `oracle user` baseline (for regression and clustering), which suggests DS pipelines with predictive performance similar or equal to the best performing DS pipeline even under the limited search budget.

Moreover, the AutoML baselines require the end-user to specify quantifiable loss functions in order to find the best-performing pipeline candidate, as well as the bounded search space to draw candidate configurations. We point out that setting the baselines for this experiment took a matter of

[6] https://amueller.github.io/dabl/dev/, A.: 2023-01-26.

[7] https://microsoft.github.io/FLAML/, A.: 2023-10-19.

[8] https://scikit-learn.org/stable/modules/clustering.html, Accessed: 2023-01-26.

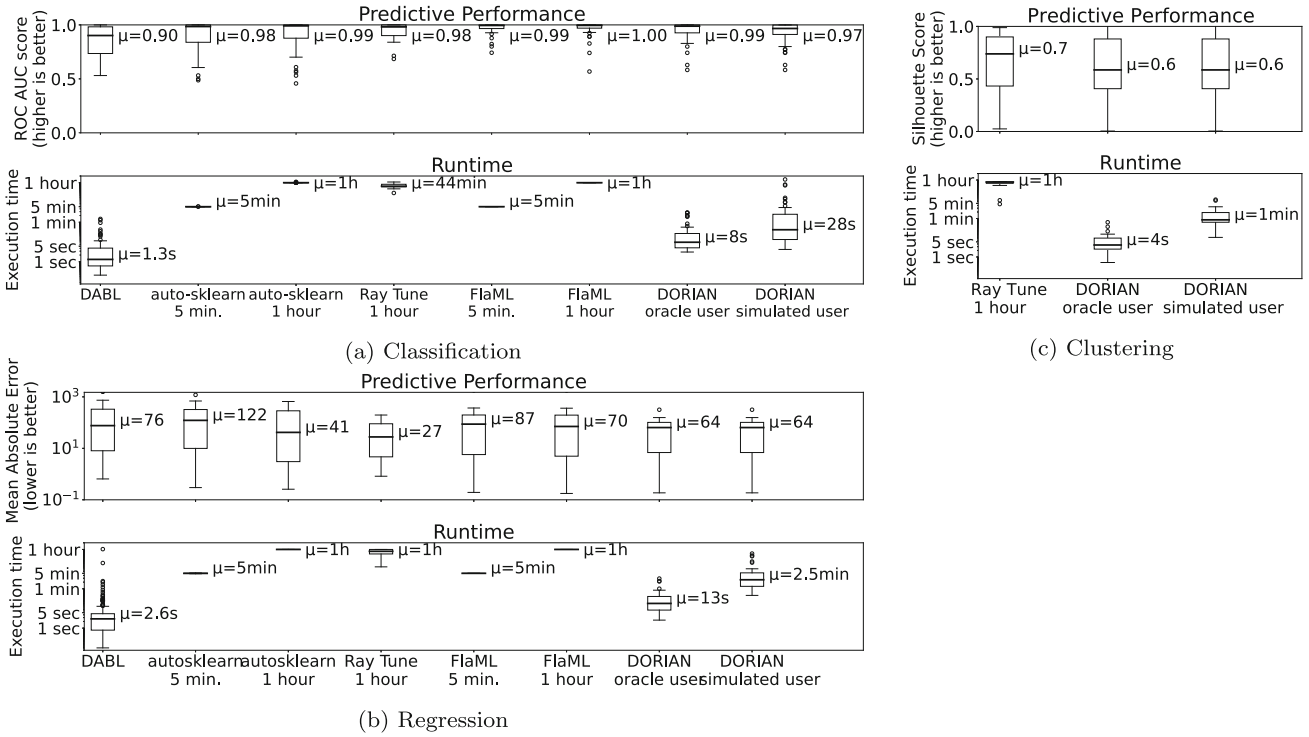


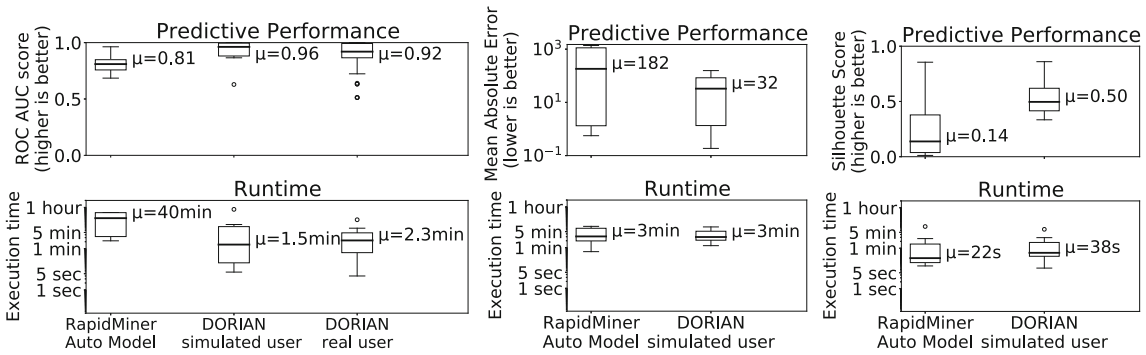
Fig. 5 Comparison against six AutoML baselines: DORIAN performs similarly to its automated counterparts in predictive performance while being significantly faster

hours and a reasonable level of familiarity with various ML models and data preprocessing techniques in order to specify the search space (the set of algorithms and domains of acceptable hyperparameters) and successfully execute the baselines. When assuming an audience with limited proficiency in DS and ML (novice users and domain experts) or the complexity of the underlying domain-specific task, these AutoML solutions might be impractical to utilize. DORIAN with a simulated user represents a trade-off that performs better than simpler baselines and faster than the full-fledged AutoML solutions.

9.3 Comparison against IDA baselines

Evaluation scenario. As we cannot evaluate human-in-the-loop baselines for all the available datasets due to the manual overhead, we selected six datasets for each DS task. To ensure a fair and realistic evaluation, we chose the datasets to be as diverse as possible by running the incremental farthest neighbor search algorithm [41] over the data footprints of all available datasets. For RapidMiner, we use each of the datasets and the corresponding DS task as a query for the Auto Model component that provides partial automation of the pipeline candidate generation and search process yet requires input from the end-user. For DORIAN, we conducted a preliminary user study with 6 PhD students with varying expertise in DS. We asked them to use DORIAN in order to design a DS pipeline that satisfies a pre-selected user query (i.e., each of the 6 datasets, supervised classification). We report the results of the user study as DORIAN real user, while we also included results for the simulated user. We apply the leave-one-out scheme as described in Sect. 9.1.

Results. Figure 6 depicts the average predictive performance for classification, regression, and clustering, as well as the total execution time of the three baselines—RapidMiner Auto Model, DORIAN simulated user (automated), and DORIAN real user. In this experiment, the total execution time includes delays for user interaction (i.e., real end-user). We observe that, for supervised classification, both DORIAN baselines are one order of magnitude faster than RapidMinder and provide superior performance (92 − 96% against 81%). DORIAN real user reaches the ROC AUC score of 92% that is comparable to DORIAN simulated user, requiring, on average, 1 minute longer to compute. The difference in the total time between DORIAN simulated and real is due to the time spent on user interaction and the fact that fewer pipelines were submitted for execution by real end-users compared to the automated DORIAN simulated user baseline. For regression and clustering, the human-in-the-loop baselines require comparable total time, while DORIAN simulated user systematically outperforms RapidMiner in terms of accuracy metrics. This is due to the amount of information that DORIAN stores for decision making and the



(a) Classification (b) Regression (c) Clustering

Fig. 6 Comparison with state-of-the-art IDA tool: DORIAN with both simulated and real users significantly outperforms the baseline in predictive performance while requiring one order of magnitude less time in the best case (classification) and slightly more time in the worst case (clustering)

variety of pipeline configurations that it can recommend. RapidMiner, in turn, has a small fixed number of pipeline configurations that can be reduced for efficiency purposes based on the data features.

9.4 Preliminary user study and qualitative interview

Evaluation Scenario. We conducted a preliminary user study with 6 PhD students with varying expertise in DS—they were tasked to use DORIAN to design a DS pipeline that satisfies a pre-selected user query (i.e., 10 datasets and the corresponding DS task). When the user interacted with DORIAN, we recorded the user interaction, such as the rank of selected pipeline candidates in the list of suggestions, the number of iterations that the user chose, the best trailing predictive performance among all the pipelines that the user has chosen for each user query, the total time that the user spent working on each query (including the time for browsing and idle time). We also selected one user query for which the execution engine was disabled. In that case, the end-user could only rely on the visual inspection of the pipeline structure to make decisions about pipeline comparison and not the predictive performance. This baseline task served two purposes: (a) to estimate the level of user expertise in DS, and (b) to answer the question of whether it is reasonable to assume that the users are capable of assessing the pipeline predictive potential by visual inspection only.

Results. For all the users and queries that we evaluate in this study, we observe that: (a) in 90% of the cases, the users stopped after 4 iterations of recommendations; (b) in 90% of the cases, the users selected one of the top-5 suggested pipelines, with the lowest selected candidate in the list of recommendations having the rank 9; (c) suggested pipeline candidates follow the power law distribution where 5 candidates were chosen in the majority of cases (account for 20% of all execution requests) and over 70 pipeline candidates were chosen for execution with the frequency under 2% (the long tail of infrequent pipeline candidates accounts for 60% of all execution requests); (d) the users applied a mixture of browsing patterns that resemble the logic of traversal algorithms for search trees, such as breadth-first search and A-star search, and less frequently—random, depth-first or always-top search patterns; (e) more experienced users (the ones that correctly compared a higher fraction of pipelines in the baseline task with the disabled execution engine) tend to be more selective in their search, varying the browsing pattern based on the given pipeline suggestions and making more decisions to select or discard a pipeline candidate without executing it; (f) less experienced users oftentimes employed the execute-discard pattern where they submit a pipeline for execution and immediately discard it from the list of suggestions in case they perceive the reported predictive performance as not satisfiable; (g) the users perceive the tool as useful in scenarios where they want to explore new DS operators or pipeline candidates that might be relevant to their query and where they want to receive a pool of potentially relevant pipeline candidates fast, without writing code.

Interview Protocol. We base the protocol of the qualitative interview on the setup of the preliminary user study and expose another control group of end-users (domain experts and software engineers) with varying expertise in data science to DORIAN's user interface with several pre-specified queries. For the qualitative interview, we provide four tasks: (a) an introductory task where end-users explore DORIAN and its UI while selecting a pipeline for the supervised classification task with the execution engine being disabled and the evaluation process set to pairwise comparison—in this scenario, end-users cannot get quantifiable performance metrics and have to make decisions based on their experience and intuition, comparing pipeline only by their structure; and three scenarios where the end-users need to add (b) a new DS task, (c) a DS operator, and (d) a ranking objective. While

performing the tasks, the control group can ask questions and provide suggestions that are later aggregated.

Insights. We present the aggregated observations and suggestions from both control groups, marking the commentary of domain experts as **[D]** and software engineers as **[S]**. When evaluating an action in terms of intuitiveness and complexity, the individuals identified three categories: "easy" (i.e., "makes sense", "reasonable", "fast", "straightforward"), "requires effort" (i.e., "learning curve", "commitment"), and "complicated" (i.e., "do not know", "skip", "would not spend time"). In summary, the respondents appreciated the practical relevance and timeliness of our approach. Moreover, when discussing the use case of non-quantifiable evaluation processes, most respondents characterized DORIAN's fallback pairwise comparison as intuitive. They also identified two challenges to DORIAN's practical adoption and extensibility—the learning curve of the UI and the extension of the knowledge graph.

(1) **[D, S]** Adding new data science tasks is straightforward because they are represented as string tags (i.e., annotations); adding relations to the new task entity (i.e., to position the task in the existing data science ontology) is an extra effort yet not necessary—the relation "X is a subclass of Data Science Task" is generated by default. Several individuals resorted to adding new tasks with the default relation.
(2) **[D, S]** adding new ranking objectives and data science operators is straightforward as the function signatures are specified, but it takes effort to write code. Individuals noted that they would instead implement and test the functions in their development environment and copy-paste it into the DORIAN's code editor.
(3) **[D, S]** collaborative user interactions, e.g., curating the data science ontology, suggesting changes, and requesting confirmation of the original authors;
(4) **[S]** inexperienced users are believed to be "less forgiving" in adopting DORIAN. They would require explicit guidance for user interaction and feedback loops, implemented primarily as *sequences* of automatically-triggered pop-up dialogues and explanatory visual cues (text, sound, and video tutorials). Individuals reported automation of user interaction in "teach me by guiding me through" scenarios as particularly important for driving DORIAN's extensibility.

9.5 Quality of ranked recommendations

We evaluate the quality of our recommendations by measuring (1) the success rate of getting a promising pipeline in the suggestions, (2) the diversity of the recommendations, and (3) the quality of recommendation under the cold-start problem, i.e., when operators or DS tasks of a user query are not present in the Experiment Store. In all cases, we consider the variant of DORIAN with the simulated user to be able to scale to all datasets. We report the metrics for the supervised classification task only, as the metrics for regression and clustering are similar across the tasks.

9.5.1 Success rate: NDCG@k, Precision@k, Recall@k

We answer the questions on how many iterations of pipeline suggestions are required to yield a "high-quality" pipeline and how many candidates per iteration are necessary, and measure DORIAN's ranking quality with the ndcg@k, precision@k and recall@k metrics.

Evaluation Scenario. We provide a drill-down analysis of the ranking quality of DORIAN under the varying number of iterations k of the *User Interaction Cycle* and the varying number of pipeline suggestions n per iteration. We find the best-performing pipeline candidate under the constrained budget (i.e., k and n) and report the normalized discounted cumulative gain (NDCG[9]) as a common measure of ranking quality [54].

In addition, we evaluate the ranking of suggestions that DORIAN provides against the ground-truth leaderboard—an ordered list of pipeline candidates and their corresponding predictive performance (evaluation metric in accordance with the DS task). We report the precision@5 and recall@5 metrics for top-5 suggested pipelines that DORIAN provides as recommendations "from scratch", i.e., without user interaction. When evaluating recommender systems, precision@k is defined as the ratio of relevant recommendations to the number of recommendations (i.e., k), and recall@k is defined as the ratio of relevant recommendations to the total number of relevant pipelines. As datasets under evaluation differ in complexity, we call a pipeline 'relevant' if its predictive performance exceeds the 90th percentile of the leaderboard. We limit the number of iterations k and the number of suggestions n to 5 as a heuristic of the human capacity for processing information efficiently [34].

Results. Figure 7 shows a heatmap with the average NDCG score aggregated across 63 datasets for supervised classification. The NDCG metric increases consistently with the growing budget for suggestions, ranging from a one-shot suggestion (top-left) to 3905 suggestions (bottom-right) within one session. Note that, as this number also corresponds to the number of pipelines available in the Experiment Store in the context of this experiment, suggestions with the budget $[k = 5, n = 5]$ likely rank the complete list of avail-

[9] Note that, as $NDCG$ is a discounted metric (it penalizes high-performing pipelines that are suggested lower in the list), its values cannot be interpreted as aggregated predictive performance but serve as a comparative ranking quality metric.

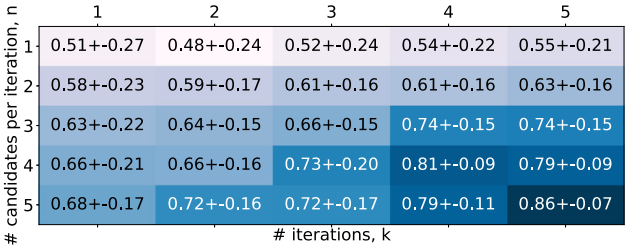


Fig. 7 Ranking quality (NDCG metric—the higher, the better) with varying numbers of iterations and candidates per iteration. DORIAN suggests reasonable candidates early on and improves its suggestions over time

able pipelines. The consistent increase of the NDCG score with the growing budget corresponds to (a) the comparative improvements of the ranking quality (i.e., high-performing candidates have high rank and low-performing candidates have low rank) and (b) the fact that the top performance score grows with the increasing budget (i.e., the maximum ROC AUC score among the suggested candidates is higher with suggestions made).

We report the aggregated precision@5 of 0.42 ± 0.25. In other words, on average, DORIAN recommends 1 to 3 (within 1 standard deviation) top-performing pipelines out of 5 suggested from scratch. We report the aggregated recall@5 of 0.54 ± 0.24. On average, for 5 pipeline candidates that DORIAN suggests, it accounts for half of the top-performing pipelines in the leaderboard. Please note, however, that the value of recall@5 is skewed, as the public leaderboard does not fully cover the pool of known pipeline candidates. In other words, not all of the known pipelines were executed on any given dataset. Thus, the actual recall@5 might be lower, as the denominator is likely higher.

9.5.2 Diversity

We measure the diversity of pipeline suggestions for a single user query as well as the adaptivity of recommendations to new user queries for all baselines for supervised classification.

Evaluation Scenario. We compute (1) how many unique candidates were suggested among all user queries and, in the case of similar candidates, their overlap, (2) the number of operators involved in these pipelines excluding their hyperparameters, (3) the number of pipelines that appear in the list of suggestions for more than 80% of the datasets (frequent pipelines), less than 20% of the datasets (rare pipelines), and (4) the average pairwise Jaccard distance between lists of suggestions for different user queries.

Results. Table 1 presents the results. For 63 datasets, DORIAN used 54 pipeline candidates for suggestions (column #Unique). These pipelines consisted of 34 operators.

Table 1 Diversity of suggestions across user queries

Baseline	#Unique	#Op.	80/20	Jaccard
DORIAN	54	34	11/26	.37 +-.16
DABL	8	11	8/0	1.0 +- 0.0
RapidMiner	9	19	9/0	1.0 +- 0.0
auto-sklearn	41*	15	41/0	1.0 +- 0.0
Ray Tune	41*	15	41/0	1.0 +- 0.0

DORIAN suggests a larger number of distinctive DS pipelines and operators with higher variety (i.e., lower pairwise Jaccard similarity metric), and 'tunes' recommendations to user queries, as opposed to baselines

11 pipeline candidates were suggested for more than 80% of the datasets, whereas 26 pipelines were suggested in less than 20% of the cases (column 80/20). On average, the pairwise Jaccard similarity measure is in the range of [0.3, 0.6] (column Jaccard), accounting for approximately half of the generated suggestions as common or popular and another half—rare and more specialized to a particular dataset. In contrast, the baselines exhibit no variation in the structure of their pipeline candidates. DABL tries only 8 pipelines, while RapidMiner AutoModel considers only 9 pipelines with different hyperparameters, leading to 205 models, with no variation from one dataset to another. auto-sklearn uses 15 operators and 41 pipelines (without considering the choice of hyperparameters). As we base the search space of Ray Tune on that of auto-sklearn, we report equivalent metrics. We, thus, conclude that DORIAN achieves a much wider variety of suggestions compared to the baselines, involving more DS operators and, hence, applying more unique pipelines. DORIAN's 80/20 ratio of 11/26 demonstrates its ability to account for the "fat-tail distribution" in recommendation [9]. We also show that DORIAN is able to make a trade-off between generic suggestions that are based on the common, well-performing pipelines (*initial recommendations*) and the specialized suggestions that are based on prior user actions (*incremental improvements*).

9.5.3 Cold-start problem

We measure the difference in the predictive performance of the best-found pipeline candidate that DORIAN recommends under two conditions—(a) where the whole Experiment Store is available to render suggestions and (b) where part of the Experiment Store is held out (e.g., a subset of DS operators, all the pipelines that contain this operator, and logs of user interaction that involves these pipelines). That allows us to evaluate the effects of the cold-start problem on DORIAN's recommendations.

Evaluation Scenario. We compute the difference in percentile scores of the leaderboard for two conditions and evaluate the performance drop as a quantifiable negative

impact that the absence (i.e., not knowing) of an operator carries. We parameterize the DS operators to be held out from the Experiment Store together with all the corresponding meta-information that relates to these pipelines. This is a retrospective simulation of the cold-start problem that estimates the gain in predictive performance when new pipeline candidates that utilize a new operator or a new family of operators are introduced to the Experiment Store.

Results. We report that, depending on the operator (e.g., missing value imputer or a data transformer), the aggregate ROC AUC score drops from 1 to 17 points in the leaderboard percentiles. In other words, if the best-performing ground-truth pipeline candidate for a given user query corresponds to the 100-th percentile in the distribution of ROC AUC scores, the best-performing pipeline candidate for the same query on the reduced Experiment Store occupies just the 83-rd percentile.

In the case when estimators are excluded from the Experiment Store, we reach the error rate of 100%, as DORIAN cannot suggest a single pipeline candidate that would perform the DS task from the user query (supervised classification and regression, respectively). The absence of data transformation techniques, such as missing value imputation or categorical encoding, also leads to a fraction of queries where none of the suggested pipelines could be executed without runtime exceptions. In this case, even though DS operators that perform the DS task at hand exist, they cannot process the raw dataset from the user query without additional data transformation.

9.6 Varying k for the nearest-neighbor search

We measure DORIAN's relative predictive performance under varying k for the nearest neighbor search for datasets and pipelines, respectively, as well as the execution time that is required to render suggestions.

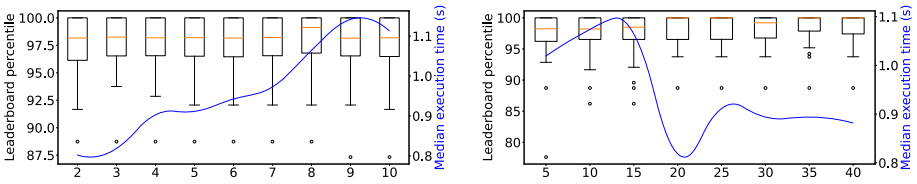
Evaluation Scenario. For every query in the experimental setup, we compute the leaderboard percentile of the best-found pipeline candidate given the search budget, the value of parameter k for the nearest neighbor search on datasets (from 2 to 10 included), and pipelines (from 10 to 40 included, with the increment step of 5). We also measure the total execution time (in seconds) for rendering the suggestions and evaluate the trade-off between the number of pipeline candidates to evaluate and the time it takes.

Results. Figure 8a depicts the steady relative predictive performance of best-found pipeline candidates across the queries. For $k \in \{3, 4\}$, the standard deviation of the percentile is relatively smaller, and for $k = 8$, the median value is relatively higher than the average across different k. However, there is no systematic evidence to demonstrate that the recommendations with parameter $k \in \{3, 4, 8\}$ are inherently better. When it comes to the execution time, on the other hand, we observe a high correlation between the parameter k and the required execution time. Note that the median execution time is under 2 seconds regardless of k, which qualifies as a real-time response. Based on this experiment, we made the decision to use $k = 3$ by default, as it does not lead to significant improvements in the quality of recommendations but is faster.

Figure 8b depicts improvements in the quality of recommendation across different k for the nearest-neighbor search on pipelines, as the median value of the percentiles increases slightly with the increase of k, and the standard deviation of percentile distribution decreases. The execution timeline chart (blue) consists of three distinctive parts—(a) steady growth for $k \in \{5, 10, 15\}$, (b) significant decrease for $k = 20$, and (c) a constant-like pattern for $k \leq 25$ with minor variation. The breakdown analysis of the execution time to render suggestions showed the following: for pattern (a), the choice of k was insufficient to render the required number of suggestions after filtering out pipeline candidates that were either previously suggested or explicitly discarded by the end-user. To compensate for the insufficient number of candidates to suggest, we repeat Algorithm 1 with the higher value of k. The fact of re-running the kNN query twice leads to increased overhead in execution time. For pattern (c), the choice of k was high enough to lead to the convergence in the number of distinctive pipeline candidates that were retrieved as a result of the kNN query. For $k = 20$, we reached a "sweet spot" given the state of the Experiment Store during the evaluation. Thus, for this evaluation, we chose to use $k = 20$ by default, which yielded a sufficient amount of pipeline candidates, even after filtering out the irrelevant ones. Note that, in practice, the optimal value of the parameter k can vary depending on the number of distinctive pipelines that are persisted in the Experiment Store and their pairwise similarity. In the case when DORIAN starts producing an insufficient number of suggestions or re-runs extended kNN queries (with $2 \times k$) frequently, we can tune k by repeating this experiment on the new state of the Experiment Store.

9.7 DORIAN as a warm-starting scheme for AutoML

We now evaluate whether DORIAN's suggestions can be successfully used as warm-starting other AutoML solutions. We



(a) k for the kNN search on datasets (b) k for the kNN search on pipelines

Fig. 8 The trade-off between the relative predictive performance of suggested pipelines (boxplot) and the time to render suggestions (blue), depending on the parameter k for the nearest neighbor search on datasets (**a**) and pipelines (**b**)

answer two questions: whether the warm-started alternative outperforms the original baseline in terms of predictive performance and whether an equally good pipeline is suggested faster.

Evaluation strategy. We use the evaluation strategy on all the datasets for supervised classification and test three baselines: `Ray Tune 5 min`, `Ray Tune 1 h`, and a variation of `Ray Tune 5 min` that is warm-started by using 5 initial suggestions that `DORIAN` renders for the query. We then compare the predictive performance (ROC AUC) of best-found pipeline candidates and claim that the warm-started variation outperforms the original `Ray Tune 5 min` baseline when its average score during cross-validation is higher than that of `Ray Tune 5 min` plus 2 standard deviations of the ROC AUC score recorded during k-fold cross-validation.

Results. We report that, for the search budget of 5 min, the warm-started variation outperformed the original baseline in 38% of the cases. In 3 datasets (i.e., user queries), the warm-started `Ray Tune 5 min` also outperformed the `Ray Tune 1 h` baseline. In absolute terms, the average difference in predictive performance between the original baseline and its warm-started variation is 0.025 (ROC AUC scores are measured from 0 to 1). In the following, we outline several observations regarding `DORIAN`'s potential to act as a warm-starting scheme for other AutoML solutions.

First, not all of the AutoML solutions allow for user-defined warm-starting—across the baselines under analysis, only `Ray Tune` is flexible enough to support warm-starting. Secondly, such a scheme necessitates the implementation of an adaptor that converts `DORIAN`'s suggestions (i.e., pipeline DAGs) into an instantiation of the search space that the AutoML solution operates. Furthermore, it requires an assumption that such a conversion is possible, i.e., there are no operators that `DORIAN` suggests and the AutoML solution "does not know about". In this experiment, we curated the search space of the `Ray Tune` baselines and could guarantee one-to-one mapping. However, in practice, the implementation of the adaptor should be responsible for accurate conversion and fallback options for exception handling.

10 Limitations and future work

In this section, we discuss the main limitations of our framework, namely recommendations under the cold-start problem and error-prone decisions of end-users.

Cold-Start Problem and learning from the past. The quality of recommendations that `DORIAN` makes is as good as the curated past experiments persisted in the Experiment Store and cannot provide high-quality recommendations for unknown or insufficiently represented DS operators and tasks. This is a well-known problem in recommendation systems in general [29]. However, `DORIAN` is designed to render suggestions regardless of the extent of available evidence and incorporate human feedback to improve. In case little information is available, we rely on human creativity that constitutes the exploration part of the exploration-exploitation trade-off. To keep the Experiment Store up-to-date, we allow for (1) populating the Experiment Store from other publicly available experiment databases using the *Pipeline extractor* (Sect. 7), (2) persisting intermediate results when AutoML operators are used in a pipeline, and (3) incorporating feedback loops where end-users report identified blank spots in `DORIAN`'s knowledge and, when appropriate, fill these blank spots themselves. As one direction for future work, we consider extending the knowledge graph and adding extra logic to generate pipelines that include unseen DS operators. That allows `DORIAN` to render suggestions where all the baselines would not be able to function, providing an environment for productive exploration of available options.

Imperfect End-Users. `DORIAN` is uniquely positioned among other existing solutions by treating user interaction, past and present, as a first-class modality for rendering recommendations. In this case, however, we have to operate under the assumption of trust in (and the authority of) the end-user as the domain expert, especially in cases when no quantifiable evaluation processes exist. Systematic mistakes or random actions might lead to suboptimal recommendations. We attempt to counteract by incorporating ranking objectives that aggregate across end-users, queries, datasets, and thus promote the wisdom of the crowd—a theory that assumes groups to be collectively smarter than individual experts [49]. In this case, even a fraction of mistakes made by one end-user

is unlikely to bear significant effects on the overall quality of recommendations.

Future work. We aim to continue curating the knowledge graph, expanding to new data science tasks, operators (guided by the input of end-users), and libraries (by introducing more crawlers). For streamlining the incremental creation of rewrite rules and adapting DORIAN to new programming languages, we plan to introduce the "generate by example" functionality where the rule $i+1$ is suggested automatically by comparing the pipeline graph after applying rules $0:i$ with the graph corrected by the end-user (i.e., ground truth).

Furthermore, as end-users are prone to make mistakes, we plan to utilize the *User Interaction Table* in order to identify potential contradictions in user actions, when the user actions are compared either to the past actions, or to actions of other end-users.

11 Related work

We outline related work for both AutoML and IDA solutions but also for every component of DORIAN.

Intelligent Discovery Assistants. In the area of IDAs [20], existing solutions vary in the spectrum of supported use cases and the means for knowledge representation. Expert systems [18] store expert-curated rules for the design of DS pipelines. They use multiple-choice questions as a trigger to return a ranked list of recommended techniques. Compared to the rule-based expert systems, DORIAN supports growing evidence (i.e., updates in the experiment database). Case-based reasoning systems [1], in turn, search for previously executed pipelines that performed well in similar cases that are maintained by the experts. As opposed to case-based systems, DORIAN takes into account all stored pipelines that are automatically updated and does not require the involvement of the experts. Meta-learning systems [7, 38] compute meta-features for the data of interest and predict DS pipelines that are likely to perform well on a given user query. These solutions train meta-models to recommend pipeline structure and reduce the search space for hyperparameter tuning. DORIAN does not apply any ML models for pipeline recommendation or require to curate training data or re-train meta-classifiers when new evidence arrives. Fusi et al. [22] propose a collaborative filtering approach to the recommendation of pipeline candidates that utilizes probabilistic matrix factorization on the dataset-pipeline-performance matrix, where the goal is to utilize prior knowledge and identify high-performing candidates quickly. In contrast, DORIAN allows for varying tasks, evaluation processes, and performance metrics.

AutoML. Related AutoML solutions [6, 21, 23, 26, 27, 39, 46, 47] consider the problem at hand an optimization task by iteratively generating and evaluating pipelines. To evaluate the generated candidate, these solutions require defining an objective loss function. In contrast, our approach is generic to provide support for arbitrary evaluation processes, including the ones with no objective loss functions available. We search for previously executed pipelines instead of generating new candidates. Interactive AutoML solutions [7, 47] combine design decisions from AutoML and IDA and highlight the necessity of keeping the end-user in the loop. Unfortunately, these solutions focus only on a particular DS task (e.g., classification or data preprocessing) and utilize a fixed set of supported operators and pipelines to choose from. Existing IDA solutions also vary in terms of the scope of supported functionality, e.g., data preparation [14, 43] or end-to-end pipeline synthesis [47]. DORIAN, in turn, is extensible by design.

Recommendation Engine. Existing solutions in the area of non-dominated sorting (i.e., multi-objective ranking) have an average running complexity of $O(Nlog^{K-1}(N))$, where N is the number of candidates to rank and K—the number of objectives. DORIAN utilizes the implementation of the non-dominated sorting algorithm proposed by Buzdalov et al. [12] as its *worst-case* running complexity is asymptotically proven to be of $O(Nlog^{K-1}(N))$. This property is critical as DORIAN is explicitly required to generate suggestions in real time. Learning-to-rank [3, 30] is another family of algorithms that uses Machine Learning to fit a ranking model. However, it requires training data for ranking, ground truth that we do not have in our context, and which would lead to constant model re-training in order to adapt to scenarios where the set of supported operators expands.

Experiment Store. There exist several open-source and commercial solutions of experiment databases [4, 37, 44, 51, 52, 56] whose goal is reproducibility and tracking of provenance and metadata (e.g., for root-cause analysis) as well as management of digital artifacts that correspond to a data science experiment (e.g., persisted ML models and performance metrics). These solutions store metadata with the identifier of the experiment session and the trial. They persist the underlying DS pipeline either as a source code text file or by outsourcing the task altogether to a version-control system (e.g., `git`). Commercial solutions like Comet,[10] KNIME,[11] Rapid Miner,[12] SAS Enterprize Miner[13] implement their internal workflow management component to represent DS pipelines as graphs and persist them in a proprietary data format. DORIAN, on the other hand, represents DS pipelines in an open-source format directly in the Experiment Store. That enables operations on graphs as part of the programming interface for ranking objectives.

[10] https://www.comet.com/site/, Accessed: 2023-01-26.

[11] https://www.knime.com/, Accessed: 2023-01-26.

[12] https://rapidminer.com/, Accessed: 2023-01-26.

[13] https://www.sas.com/, Accessed: 2023-01-26.

DS Pipeline Extractor. The ML Bazaar project [48] allows treating DS operators as the "building block" primitives for pipeline composition. As the representation of operators contains the specification of interfaces to fit a predictor or perform data transformation, this tool automatically handles the "glue code" between primitives that come from different libraries and frameworks. IBM's Data Science Ontology project[14] aims to catalog data science concepts and semantically annotate various frameworks and libraries of the data science toolbox. Extending the knowledge bases of these projects, however, incurs a significant learning curve and manual overhead for the end-user. AL [13] is a tool for the automated generation of supervised learning programs that analyzes dynamic program traces and learns a conditional probability model to generate pipeline candidates for new data. DORIAN, in turn, extracts pipelines based on static code analysis (i.e., without the necessity to execute a DS script) and does not require model re-training when new DS pipelines, tasks, or operators become available.

12 Conclusion

We proposed DORIAN, a human-in-the-loop framework for the *assisted design of data science pipelines* that is based on previously executed pipelines. Based on a user query (i.e., dataset and DS task), DORIAN outputs in *real-time* a ranked list of pipeline candidates that the user can choose from to execute or modify. To achieve this, DORIAN stores in an efficient manner all previous DS experiments and utilizes similarity search to identify relevant datasets and pipelines. Its recommendation engine can easily be extended with user-defined objective functions using a simple interface. It uses multi-objective sorting to rank the retrieved candidate pipelines and takes user interactions into account to improve suggestions over time. We showed that DORIAN significantly outperformed the state-of-the-art IDA tool and achieved similar predictive performance with state-of-the-art long-running AutoML solutions while being real-time, generic to any evaluation processes and DS tasks, and extensible to new DS operators.

Acknowledgements This work was funded by the HEIBRiDS graduate school, with the support of the German Ministry for Education and Research as BIFOLD, BBDC 2 (01IS18025A), BZML (01IS18037A), the Software Campus Program (01IS17052), Ahold Delhaize, and the research department 'Data Science and its Applications' of the German Research Center for Artificial Intelligence. All content represents the opinion of the authors, which is not necessarily shared or endorsed by their respective employers and/or sponsors.

Funding Open Access funding enabled and organized by Projekt DEAL.

[14] https://www.datascienceontology.org/, A.: 2023-01-26.

References

1. Aamodt, A., Plaza, E.: Case-based reasoning: foundational issues, methodological variations, and system approaches. AI Commun. **7**(1), 39–59 (1994)
2. Abu-Aisheh, Z., Raveaux, R., Ramel, J., Martineau, P.: An exact graph edit distance algorithm for solving pattern recognition problems. In: ICPRAM'15. Lisbon, Portugal (2015). https://doi.org/10.5220/0005209202710278, https://hal.archives-ouvertes.fr/hal-01168816
3. Amashukeli, S., Elshawi, R., Sakr, S.: ismartml: an interactive and user-guided framework for automated machine learning. In: Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA'20 (2020)
4. Avsec, Ž, et al.: The kipoi repository accelerates community exchange and reuse of predictive models for genomics. Nat. Biotechnol. **37**(6), 592–600 (2019)
5. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975). https://doi.org/10.1145/361002.361007
6. Bergstra, J., Yamins, D., Cox, D.: Hyperopt: a python library for optimizing the hyperparameters of machine learning algorithms. In: SciPy'13, vol. 13, p. 20. Citeseer (2013)
7. Bilalli, B., Abelló, A., Aluja-Banet, T., Wrembel, R.: Intelligent assistance for data pre-processing. Comput. Stand. Interfaces **57**, 101–109 (2018)
8. Bischl, B., et al.: Openml benchmarking suites and the openml100. stat **1050**, 11 (2017)
9. Borges, R., Stefanidis, K.: On measuring popularity bias in collaborative filtering data. In: EDBT/ICDT Workshops (2020)
10. Brazdil, P., van Rijn, J., Soares, C., Vanschoren, J.: Automating workflow/pipeline design, pp. 123–140. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-67024-5_7
11. Burkhard, W.A., Keller, R.M.: Some approaches to best-match file searching. Commun. ACM **16**(4), 230–236 (1973). https://doi.org/10.1145/362003.362025
12. Buzdalov, M., Shalyto, A.: A provably asymptotically fast version of the generalized jensen algorithm for non-dominated sorting. In: PPSN'14, pp. 528–537. Springer (2014)
13. Cambronero, J.P., Rinard, M.C.: Al: Autogenerating supervised learning programs. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). https://doi.org/10.1145/3360601
14. Chen, C., Golshan, B., Halevy, A.Y., Tan, W.C., Doan, A.: Biggorilla: an open-source ecosystem for data preparation and integration. IEEE Data Eng. Bull. **41**(2), 10–22 (2018)
15. Cordella, L., Foggia, P., Sansone, C., Vento, M.: Performance evaluation of the vf graph matching algorithm. In: Proceedings 10th International Conference on Image Analysis and Processing, pp. 1172–1177 (1999). https://doi.org/10.1109/ICIAP.1999.797762

16. Cordella, L., Foggia, P., Sansone, C., Vento, M.: A (sub) graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. **26**(10), 1367–1372 (2004)
17. Corradini, A., Heindel, T., Hermann, F., König, B.: Sesqui-pushout rewriting. In: International Conference on Graph Transformation, pp. 30–45. Springer (2006)
18. Craw, S., Sleeman, D., Graner, N., Rissakis, M., Sharma, S.: Consultant: providing advice for the machine learning toolbox. In: Proceedings of the Research and Development in Expert Systems IX, pp. 5–23 (1992)
19. Cremers, A., Ginsburg, S.: Context-free grammar forms. J. Comput. Syst. Sci. **11**(1), 86–117 (1975)
20. Fayyad, U., Piatetsky-Shapiro, G., Smyth, P.: From data mining to knowledge discovery in databases. AI Mag. **17**(3), 37–37 (1996)
21. Feurer, M., et al.: Auto-sklearn: efficient and robust automated machine learning. In: Automated Machine Learning, pp. 113–134. Springer, Cham (2019)
22. Fusi, N., Sheth, R., Elibol, M.: Probabilistic matrix factorization for automated machine learning. In: Advances in Neural Information Processing Systems, vol. 31 (2018)
23. He, X., Zhao, K., Chu, X.: Automl: a survey of the state-of-the-art. Knowledge-Based Systems **212**, 106,622 (2021). https://doi.org/10.1016/j.knosys.2020.106622, https://www.sciencedirect.com/science/article/pii/S0950705120307516
24. Hochstein, L., Moser, R.: Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way. O'Reilly Media Inc, New York (2017)
25. Jensen, M.: Reducing the run-time complexity of multiobjective eas: The nsga-ii and other algorithms. IEEE Trans. Evol. Comput. **7**(5), 503–515 (2003). https://doi.org/10.1109/TEVC.2003.817234
26. Kotthoff, L., Thornton, C., Hoos, H., Hutter, F., Leyton-Brown, K.: Auto-weka 2.0: automatic model selection and hyperparameter optimization in weka. J. Mach. Learn. Res. **18**(1), 826–830 (2017)
27. Le, T.T., Fu, W., Moore, J.H.: Scaling tree-based automated machine learning to biomedical big data with a feature set selector. Bioinformatics **36**(1), 250–256 (2020)
28. Liaw, R., et al.: Tune: a research platform for distributed model selection and training. arXiv:1807.05118 (2018)
29. Lika, B., Kolomvatsos, K., Hadjiefthymiades, S.: Facing the cold start problem in recommender systems. Expert Systems with Applications **41**(4, Part 2), 2065–2073 (2014). https://doi.org/10.1016/j.eswa.2013.09.005, https://www.sciencedirect.com/science/article/pii/S0957417413007240
30. Liu, T.Y.: Learning to Rank for Information Retrieval. Springer, Berlin (2011)
31. Liu, T.Y., et al.: Learning to rank for information retrieval. Found. Trends Inf. Retriev. **3**(3), 225–331 (2009)
32. Luecken, M., Theis, F.: Current best practices in single-cell rna-seq analysis: a tutorial. Mol. Syst. Biol **15**(6), e8746 (2019)
33. McKay, B.: Practical graph isomorphism. Congr. Numerantium **87**, 30–45 (1981)
34. Miller, G.A.: The magical number seven, plus or minus two: some limits on our capacity for processing information. Psychol. Rev. **63**(2), 81 (1956)
35. Miller, R.B.: Response time in man-computer conversational transactions. In: Proceedings of the December 9–11, 1968, Fall Joint Computer Conference, Part I, AFIPS '68 (Fall, part I), p. 267-277. ACM, New York (1968). https://doi.org/10.1145/1476589.1476628
36. Mölder, F., Jablonski, K., Letcher, B., Hall, M., Tomkins-Tinch, C., Sochat, V., Forster, J., Lee, S., Twardziok, S., Kanitz, A., Wilm, A., Holtgrewe, M., Rahmann, S., Nahnsen, S., Köster, J.: Sustainable data analysis with snakemake [version 2; peer review: 2 approved]. F1000Research **10**(33) (2021). https://doi.org/10.12688/f1000research.29032.2
37. Namaki, M.H., Floratou, A., Psallidas, F., Krishnan, S., Agrawal, A., Wu, Y., Zhu, Y., Weimer, M.: Vamsa: automated provenance tracking in data science scripts. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '20, pp. 1542–1551. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3394486.3403205
38. Nguyen, P., Hilario, M., Kalousis, A.: Using meta-mining to support data mining workflow planning and optimization. J. Artif. Int. Res. **51**(1), 605–644 (2014)
39. Olson, R., Moore, J.: Tpot: a tree-based pipeline optimization tool for automating machine learning. In: ICML'16 AutoML Workshop, pp. 66–74. JMLR (2016)
40. Patterson, E., Baldini, I., Mojsilovic, A., Varshney, K.R.: Semantic representation of data science programs. In: IJCAI, pp. 5847–5849 (2018)
41. Rahman, S., Rochan, M.: A fast farthest neighbor search algorithm for very high dimensional data. In: 19th International Conference on Computer and Information Technology (ICCIT), pp. 351–356 (2016). https://doi.org/10.1109/ICCITECHN.2016.7860222
42. Redyuk, S., Kaoudi, Z., Schelter, S., Markl, V.: DORIAN in action: assisted design of data science pipelines. Proc. VLDB Endow. **15**(12), 3714–3717 (2022). https://doi.org/10.14778/3554821.3554882
43. Rezig, E.K., Cao, L., Stonebraker, M., Simonini, G., Tao, W., Madden, S., Ouzzani, M., Tang, N., Elmagarmid, A.K.: Data civilizer 2.0: a holistic framework for data preparation and analytics. Proc. VLDB Endow. **12**(12), 1954–1957 (2019)
44. Schelter, S., Böse, J.H., Kirschnick, J., Klein, T., Seufert, S., Amazon: declarative metadata management: a missing piece in end-to-end machine learning. SysML (2018). https://api.semanticscholar.org/CorpusID:52841157
45. Serban, F., Vanschoren, J., Kietz, J.U., Bernstein, A.: A survey of intelligent assistants for data analysis. ACM Comput. Surv. CSUR **45**(3), 1–35 (2013)
46. Shahriari, B., et al.: Taking the human out of the loop: a review of Bayesian optimization. Proc. IEEE **104**(1), 148–175 (2015)
47. Shang, Z., et al.: Democratizing data science through interactive curation of ml pipelines. In: SIGMOD'19, pp. 1171–1188. ACM (2019). https://doi.org/10.1145/3299869.3319863, https://doi.org/10.1145/3299869.3319863
48. Smith, M.J., Sala, C., Kanter, J.M., Veeramachaneni, K.: The machine learning bazaar: harnessing the ml ecosystem for effective system development. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 785–800 (2020)
49. Surowiecki, J.: The Wisdom of Crowds. Knopf Doubleday Publishing Group (2005). https://books.google.de/books?id=hHUsHOHqVzEC
50. Vanschoren, J.: Meta-learning. In: Automated Machine Learning, pp. 35–61. Springer, Cham (2019)
51. Vanschoren, J., Van Rijn, J.N., Bischl, B., Torgo, L.: Openml: networked science in machine learning. ACM SIGKDD Explor. Newsl **15**(2), 49–60 (2014)
52. Vartak, M., Subramanyam, H., Lee, W.E., Viswanathan, S., Husnoo, S., Madden, S., Zaharia, M.: Modeldb: a system for machine learning model management. In: Proceedings of the Workshop on Human-In-the-Loop Data Analytics, pp. 1–3 (2016)
53. Vlot, A., Maghsudi, S., Ohler, U.: Semitones: single-cell marker identification by enrichment scoring. Cold Spring Harbor Laboratory (2020)
54. Wang, Y., Wang, L., Li, Y., He, D., Chen, W., Liu, T.: A theoretical analysis of ndcg ranking measures. In: Proceedings of the 26th Annual Conference on Learning Theory (COLT 2013), vol. 8, p. 6 (2013)

55. Yao, Q., et al.: Taking human out of learning applications: a survey on automated machine learning. arXiv:1810.13306 (2018)
56. Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S.A., Konwinski, A., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., et al.: Accelerating the machine learning lifecycle with mlflow. IEEE Data Eng. Bull. **41**(4), 39–45 (2018)
57. Zeng, Z., Tung, A.K.H., Wang, J., Feng, J., Zhou, L.: Comparing stars: on approximating graph edit distance. Proc. VLDB Endow. **2**(1), 25–36 (2009). https://doi.org/10.14778/1687627.1687631