Automating and Optimizing Data-Centric What-If Analyses on Native Machine Learning Pipelines

Stefan Grafberger
s.grafberger@uva.nl
AIRLab, University of Amsterdam

Paul Groth
p.groth@uva.nl
University of Amsterdam

Sebastian Schelter
s.schelter@uva.nl
University of Amsterdam

ABSTRACT

Software systems that learn from data with machine learning (ML) are used in critical decision-making processes. Unfortunately, real-world experience shows that the pipelines for data preparation, feature encoding and model training in ML systems are often brittle with respect to their input data. As a consequence, data scientists have to run different kinds of *data centric what-if analyses* to evaluate the robustness and reliability of such pipelines, e.g., with respect to data errors or preprocessing techniques. These what-if analyses follow a common pattern: they take an existing ML pipeline, create a pipeline variant by introducing a small change, and execute this pipeline variant to see how the change impacts the pipeline's output score. The application of existing analysis techniques to ML pipelines is technically challenging as they are hard to integrate into existing pipeline code and their execution introduces large overheads due to repeated work.

We propose **mlwhatif** to address these integration and efficiency challenges for data-centric what-if analyses on ML pipelines. **mlwhatif** *enables data scientists to declaratively specify what-if analyses for an ML pipeline, and to automatically generate, optimize and execute the required pipeline variants.* Our approach employs "pipeline patches" to specify changes to the data, operators and models of a pipeline. Based on these patches, we define a multi-query optimizer for efficiently executing the resulting pipeline variants jointly, with four subsumption-based optimization rules. Subsequently, we detail how to implement the pipeline variant generation and optimizer of mlwhatif. For that, we instrument "native" ML pipelines written in Python to extract dataflow plans with re-executable operators.

We experimentally evaluate mlwhatif, and find that its speedup scales linearly with the number of pipeline variants in applicable cases, and is invariant to the input data size. In end-to-end experiments with four analyses on more than 60 pipelines, we show speedups of up to 13x compared to sequential execution, and find that the speedup is invariant to the model and featurization in the pipeline. Furthermore, we confirm the low instrumentation overhead of mlwhatif.

SIGMOD, 2023, Seattle, WA, USA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Stefan Grafberger, Paul Groth, and Sebastian Schelter. 2023. Automating and Optimizing Data-Centric What-If Analyses on Native Machine Learning Pipelines. In *Proceedings of ACM SIGMOD (SIGMOD).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Software systems that learn from data with machine learning (ML) are used in critical decision-making processes [51]. Unfortunately, real-world experience shows that the pipelines for data preparation, feature encoding and model training in ML systems are often brittle with respect to issues in the data they process [4, 39, 44, 51].

Data centric what-if analyses on ML pipelines. During the development of such pipelines, an important task of data scientists is to understand the sensitivity of their pipeline by performing *data-centric what-if analyses* [15] on it. Such analyses, for example, focus on (*i*) the robustness against data errors [47], asking *what-if the input data to a pipeline had certain errors like missing values or outliers?* (*ii*) feature importance [5], asking *what-if the pipeline did not have access to a particular feature?* (*iii*) the impact of preprocessing operators on the pipeline's fairness [2], asking *what-if the pipeline filtered or featurized the training data differently?*, and (*iv*) the impact of data cleaning operations [25], asking *what-if the pipeline applied a particular error detection and cleaning technique?* These what-if analyses follow a common pattern: they take an existing ML pipeline, create a *pipeline variant* by introducing a small change, and *execute this pipeline variant* to see how the change impacts the pipeline's output score, e.g., its accuracy or a fairness metric. Currently, the application of these analysis techniques to ML pipelines poses several technical challenges.

Integration challenge. Data-centric what-if analyses are difficult to integrate with existing pipeline code. Many analysis techniques are designed for single input datasets in matrix form [19, 32] and not for pipelines with multiple heterogeneous input datasets. Additionally, they are often implemented as stand-alone software packages [2, 25, 54] with hardcoded data preparation steps which are difficult to adjust. As a result, the integration of such analyses requires significant and costly manual development efforts. Reducing this development time is crucial, as data scientists already spent more than 60% of their time on data preparation tasks [61]. This leads to our first research question RQ1: *How can we automatically apply data-centric what-if analyses to existing ML pipeline code?*

Efficiency challenge. A major part of the computation time in ML pipelines is spent on data preparation and validation [56]. This overhead grows when we run what-if analyses on a pipeline, as the repeated execution of the pipeline variants incurs a lot of redundant work. It is important to reduce this runtime overhead to enable data scientists to iterate faster during development, and reduce the

time-to-deployment for pipelines [39]. This leads to our second research question RQ2: *How can we optimize the joint execution of several variants of an ML pipeline for a what-if analysis?*

Automated and optimized what-if analyses with mlwhatif. In this paper, we address these challenges with **mlwhatif**, which *enables data scientists to declaratively specify what-if analyses for an ML pipeline, and to automatically generate, optimize and execute the required pipeline variants.*
Our approach is based on an existing dataflow representation for ML pipelines (Section 3), and addresses our research questions with the following contributions.

Contribution 1: "Pipeline patches" as a formal framework to generate different variants of an ML pipeline (Section 3). We address RQ1 by proposing "pipeline patches" to generate different variants of a pipeline required for a what-if analysis. These patches specify changes to the data, operators and models of ML pipelines. We detail how to express four complex existing what-if analyses [2, 5, 25, 47] with these patches.

Contribution 2: Multi-query optimization for the joint execution of several pipeline variants. (Section 5). We address RQ2 by proposing four subsumption-based [43, 49] multi-query optimization rules to generate a joint, optimized execution plan from several pipeline variants, which re-uses shared intermediates to drastically reduce the runtime of a what-if analysis.

Contribution 3: A reference implementation to generate and optimize pipeline variants from existing code. (Section 7). We further address RQ1 and RQ2 on an implementation level. We detail how to automatically and efficiently execute what-if analyses on ML pipelines. For that, we extend instrumentation techniques from "mlinspect" [16] to extract dataflow plans for pipelines with re-executable operators. Next, we modify these plans according to our patches to automatically generate and optimize variants of an ML pipeline from existing Python code. Our implementation works on natively written ML pipelines, e.g., pipelines that use code from popular data science libraries such as pandas [29], scikit-learn [35] and keras [8]. This enables data scientists to apply our techniques without the need to rewrite their code to a new language or library.

Contribution 4: Experimental evaluation. (Section 9). We analyze the speedup provided by our optimization rules in different scenarios, and find that it scales linearly with the number of pipeline variants in applicable cases, and is invariant to the input data size (Section 9.1). Next, we measure the overall speedup and runtime reduction for the end-to-end execution of more than 60 different pipelines in Section 9.2. We show speed-ups of up to 13x compared to sequential execution, and find that the speedup is invariant to the model and featurization in the pipeline. Finally, we confirm the low instrumentation overhead of mlwhatif, and advantages over manual optimizations. In addition, we make the code for mlwhatif publicly available at https://github.com/stefan-grafberger/mlwhatif.

2 WHAT-IF ANALYSES ON ML PIPELINES

Data-centric what-if analyses. We detail four data-centric what-if analyses from existing work, which are difficult to apply to existing pipelines to motivate our problem direction.

Robustness against errors (`robustness`*).* A common problem in production ML is that the serving data at inference time might contain errors and data shifts [4]. The *Jenga* [47] project provides a library to test how robust a trained model is to such errors at inference time (e.g., missing values, typos, distribution shifts) by repeatedly corrupting its test data and computing the prediction quality of the model on the corrupted data versions. This requires data scientists to conduct a custom rewrite of pipelines into a task abstraction [60].

Preprocessing operator impact on fairness (`preproc`*).* Another data-centric analysis is to evaluate the impact of preprocessing operations on a pipeline's output score. A recent experimental study [2] analyzed how the score of a pipeline (in particular a fairness metric) varies under different preprocessing and featurization operations. The study required the researchers to manually implement each custom pipeline variant [58].

Feature importance (`importance`*).* A common data-centric analysis task is to determine the importance of a particular feature for a trained classifier. A classical approach is permutation feature importance [5]. The idea is to shuffle the values of a particular feature in the test data, which does not change the marginal distribution of the feature, but makes the actual feature value per sample random. The subsequent change in the prediction quality of the model denotes the importance of the particular feature. Current implementations require custom code [59] and are only applicable to featurized data in matrix form.

Joint data cleaning and machine learning (`cleanlearn`*).* An important complex what-if analysis is to understand the impact of data cleaning on a model's prediction quality. This was studied comprehensively by the *CleanML* [25] benchmark, which evaluates the impact of different ways to detect and fix missing values, outliers, duplicates and label errors on the performance of an ML model. The study required a custom benchmarking framework [57] to apply and evaluate such cleaning techniques to different pipelines and models.

Illustrative example. We present an illustrative, simplified example to introduce the multi-query optimization problem at the core of this paper: Imagine a data scientist in healthcare, who developed the ML pipeline shown in Listing 1, to create a classifier that predicts potential complications for patients based on historical treatment data. The pipeline integrates patient and treatment data (Lines 1-5 & 23-28). Afterwards, it featurizes the `weight`, `smokes` and `notes` attributes in Lines 14-20. Finally, the ML pipeline trains a neural network model to detect potential complications for patients, and computes the F1-score of its predictions as output score (Lines 30-35).

```
1  def combine(patients, patient_histories, consent_required):
2   if consent_required:
3     patients = patients[patients.gave_consent==True]
4   with_history = patients.merge(patient_histories, on="ssn")
5   return with_history
6
7  def create_neural_net():                          # Model definition
8   net = Sequential()
9     .add(Dense(8, activation='relu')
10    .add(Dense(4, activation='relu'))
11    .add(Dense(2, activation='softmax'))
12  return KerasClassifier(net.compile(loss='crossentropy'))
13
```

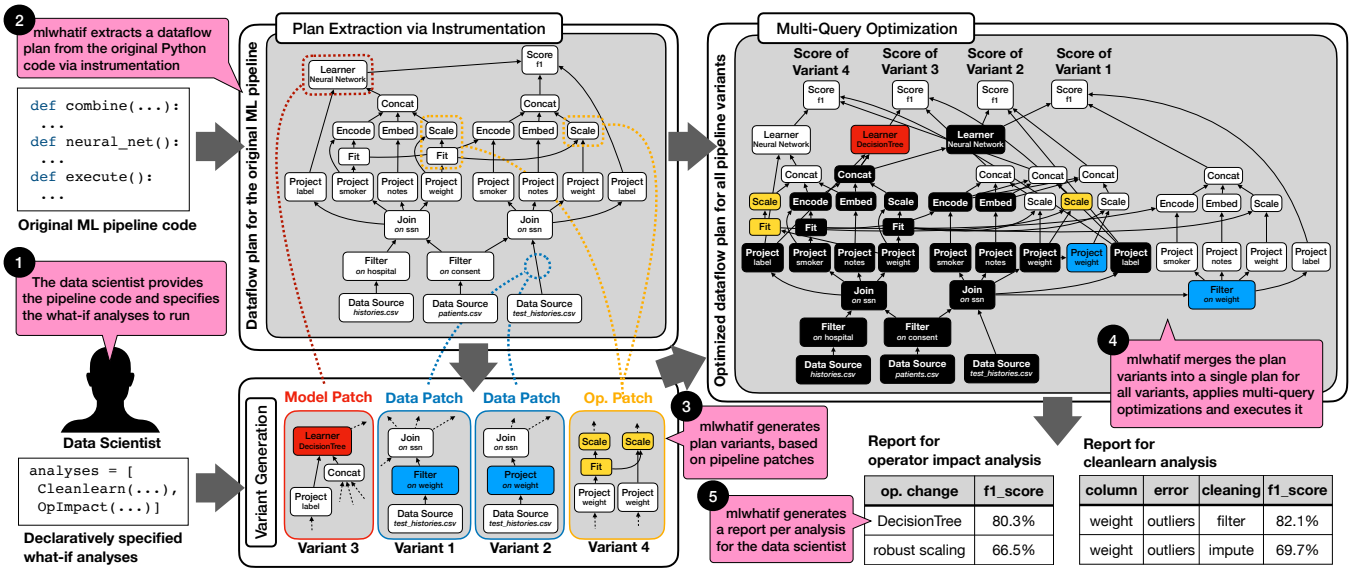


Figure 1: Overview of mlwhatif: The data scientist provides the code for an ML pipeline (Listing 1) and declaratively specifies the what-if analyses to run on that pipeline ❶. Next, mlwhatif extracts a dataflow plan from the original pipeline via code instrumentation ❷. Based on the specified what-if analyses and the extracted plan, mlwhatif generates different pipeline variants with "pipeline patches" ❸, and merges the variants into a single joint dataflow plan. Next, mlwhatif applies various multi-query optimization rules to re-use shared intermediates (illustrated as black operators) between variants ❹. Finally, mlwhatif executes the optimized plan, and generates a report per analysis, which details the variants and their output scores ❺.

```
14  def featurization():                                  # Featurization
15    sent_bert = SentenceTransformer.load('...')
16    embed = lambda txt: sent_bert.encode(txt)
17    return ColumnTransformer([
18      (StandardScaler(), "weight"),
19      (OneHotEncoder(), "smokes"),
20      (FunctionTransformer(embed, "notes")]))
21
22  def execute_pipeline():
23    patients = pd.read_csv("s3://...")     # Relational preprocessing
24    histories = pd.read_csv("s3://...")
25    test_histories = pd.read_csv("s3://...")
26    histories = histories[histories.hospital.isin(...)]
27    train = combine(patients, histories, consent_required=True)
28    test = combine(patients, test_histories, consent_required=True)
29
30    encode_and_learn = Pipeline([        # Model training and scoring
31      ('features', featurization(),
32      ('learner', KerasClassifier(create_neural_net(...))])
33    model = encode_and_learn.fit(train, train.had_complications)
34    pred = model.predict(test)
35    return f1_score(pred, test.had_complications)
```

Listing 1: Exemplary toy pipeline for a healthcare use case.

Imagine now that the data scientist realizes that there are problems with respect to the `weight` attribute in the test data of the pipeline. Sometimes the weight is zero, and in other cases the weight is unnaturally high. She notifies a data engineering team to fix these issues, but also wants to understand how such errors might impact the output scores of her pipeline.

For that, she decides to run what-if analyses with four different variants `V1`-`V4` on her pipeline. She is interested in the impact of data cleaning to determine what would happen (`V1`) if she filtered out the erroneous samples from the test data or (`V2`) if she replaced the erroneous weight values in the test data with a default weight. Furthermore, she wants to understand the impact of certain operators in her pipeline, by answering what would happen (`V3`) if she used a different model than the neural network or (`V4`) if she scaled the weight attribute in a different way.

Challenges in manually implementing the what-if analysis. Imagine that our data scientist now starts to manually implement and execute the four variants `V1`-`V4` of the what-if analysis for the pipeline. She downloads the *CleanML* [25] and *Jenga* [47] packages for such analyses, but realizes that these are difficult to integrate as they are designed for a single dataset only and contain hardcoded preprocessing steps. As a result, she has to manually integrate and copy code from these packages into her pipeline.

Runtime overheads. During the implementation, it would become clear that there are several ways to accelerate the execution of all these variants. For example, `V1` and `V2` do not impact model training, and only require different versions of the test data, so they could re-use the trained model. Furthermore, `V3` consumes the same train data as `V1` and `V2`, and only applies a different model, so this variant could re-use the train data if it was cached. `V4` only changes a single attribute of the train and test data, so it would also not be necessary to completely re-compute the other attributes from scratch. Finally, the data scientist realizes that all four variants could have re-used the join results from Line 4 if the filter from `V1` would be applied after the join.

Automated and optimized what-if analyses with mlwhatif. Our library mlwhatif enables the data scientist to declaratively specify such what-if analyses, and to automatically generate, optimize and execute the required pipeline variants. For that, she chooses from a variety of off-the-shelf what-if analyses, which mlwhatif can automatically run on her pipeline:

```
from mlwhatif.analysis import Cleanlearn, OperatorImpact
analyses = [ # Declarative definition of what-if analyses to run
  Cleanlearn(column='weight', error=Error.OUTLIER,
             cleanings=[Clean.FILTER, Clean.IMPUTE]),
  OperatorImpact(robust_scaling=True, alternative_model=...)]
# Execution of what-if analyses on a given pipeline
report_with_scores = mlwhatif.execute_whatif('healthcare.py', analyses)
print(report_with_scores)
```

Each analysis produces a report as output, which shows the output scores resulting from changes applied by the what-if analysis. The report for the cleaning analysis would, for example, describe how a particular error detection and cleaning approach impacts the F1-score of her pipeline's classifier:

```
analysis   | column   | error_detector | cleaning     | f1_score
-----------+----------+----------------+--------------+----------
cleanlearn | weight   | outliers       | filter       | 0.821
cleanlearn | weight   | outliers       | impute_const | 0.697
```

We provide a demonstration notebook to further detail the user API at https://github.com/stefan-grafberger/mlwhatif/blob/main/demo/feature_overview/feature_overview.ipynb.

Under the covers. Figure 1 illustrates how mlwhatif automatically conducts the desired what-if analysis and optimizes the execution. The data scientist provides the code for an ML pipeline (Listing 1) and declaratively specifies the what-if analyses to run on that pipeline ❶. Next, mlwhatif extracts a dataflow plan from the original pipeline via code instrumentation ❷. Based on the specified what-if analyses and the extracted plan, mlwhatif generates different pipeline variants to run with the help of "pipeline patches" ❸. mlwhatif merges the pipeline variants into a single joint dataflow plan, and applies various multi-query optimization rules to re-use shared intermediates (illustrated as black operators) between variants ❹. Finally, mlwhatif executes the optimized plan, and generates a report per analysis for the data scientist, which details the variants and their corresponding output scores ❺.

3 PRELIMINARIES

Machine learning pipelines. In general, ML applications for supervised learning in real-world scenarios do not start from a single source of input data in matrix form. Instead they work with several input data sources $\mathcal{D}_1, \ldots, \mathcal{D}_n$, such as log files, database tables, or files in a data lake, often in the form of relational data accompanied by unstructured data such as text and images [39, 44, 64]. Therefore, the model training and evaluation is typically conducted by a *machine learning pipeline*, which consists of three subsequent high-level stages:

(1) *Relational preprocessing*. This stage integrates the input datasets $\mathcal{D}_1, \ldots, \mathcal{D}_n$ into a single training relation $\mathcal{D}_{\text{train}}$ and a single test relation $\mathcal{D}_{\text{test}}$. This stage requires the execution of relational joins to combine data, and includes data integration, cleaning and filtering steps, with selections and (extended) projections to filter out tuples and remove or compute attributes.

(2) *Featurization*. This stages encodes the relations $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$ into matrix form, producing the train set $(\mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}})$ and test set $(\mathbf{X}_{\text{test}}, \mathbf{y}_{\text{test}})$ for the ML model. This stage typically applies ML-specific feature encoding operations based on linear algebra, which produce matrix outputs (e.g., one-hot-encoding, embedding, feature hashing).

(3) *Model training and scoring*. The final stage conducts the model training based on the train set $(\mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}})$, producing the model f_θ, and computes the output score $U(f_\theta(\mathbf{X}_{\text{test}}), \mathbf{y}_{\text{test}})$ denoting its prediction quality on the unseen test set.

Estimator/transformers. Furthermore, ML pipelines contain so-called estimator/transformer operations, which are popular in common ML libraries such as scikit-learn [35], SparkML [30], Tensorflow Transform [1] or Ray [62]. These operations are a composable and nestable way to hide the complexity of featurization operations. The estimator part is typically applied to training data, where it conducts a global aggregation to compute statistics, which are then used by a subsequent transformer (a tuple-at-a-time operation) to transform tuples in both train and test data.

Modeling ML pipelines as dataflow computations. For this work, we treat ML pipelines for supervised learning as dataflow computations, in accordance with recent research [16, 21]. This allows us to reason about the operations and intermediate results in a pipeline. The dataflow computation consumes the input datasets $\mathcal{D}_1, \ldots, \mathcal{D}_n$ and produces the score $U(f_\theta(\mathbf{X}_{\text{test}}), \mathbf{y}_{\text{test}})$ as output. The operators in the dataflow computation correspond to relational operations, e.g., to selections, projections and joins in the relational preprocessing stage. We additionally treat the ML-specific operations in the featurization stage as global aggregations and extended projections which output arrays, model training as a black-box aggregation, and model prediction again as extended projection. Formally, the pipeline is thereby treated as a directed-acyclic graph whose vertices V correspond to operators for the discussed relational operations, and whose edges correspond to data exchange between the operators. We will refer to this dataflow graph as *plan* of the pipeline in the following.

4 PIPELINE VARIANT GENERATION

In this section, we address RQ1: *How can we automatically apply data-centric what-if analyses to existing ML pipeline code?* As outlined in Section 2, mlwhatif starts from the original ML pipeline and a declaratively specified what-if analysis, both provided by a data scientist.

Dataflow plan extraction. As a first step, we extract a dataflow plan representation (as outlined in Section 3) from the original ML pipeline provided by the data scientist. For that, we build upon recent research under the umbrella of "mlinspect" [16, 17]. We apply code instrumentation to map declarative operations in the source code to the corresponding logical operators. This allows us to work with pipelines consisting of natively written code leveraging declarative abstractions from popular ML libraries such as pandas [29] or scikit-learn [35].

Variant generation with pipeline patches. We now discuss how to generate different pipeline variants for what-if analyses based

on the dataflow plan of the original ML pipeline. We introduce a formal way to generate such a pipeline variant from the dataflow plan of the original pipeline with an abstraction called a *pipeline patch*. Such a patch defines how to change the plan of the original pipeline to generate a custom variant of the plan. We introduce three different categories of pipeline patches.

Model patch. This patch denotes that a pipeline variant should use a different model. Recall that the training of a model f is represented as a black-box aggregation operator over the inputs $(\mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}})$ in our dataflow plan. A model patch replaces the aggregation operator for the original model f with an aggregation operator for training an alternative model f'. An example is variant `V3` of our running example, where we replace the `Learner` operator from the original plan with another model.

Operator patch. This kind of patch specifies that a particular operator v in the plan should be replaced or removed. An *operator replacement* patch replaces an operator v with an alternative operator v', and makes v' the endpoint of all edges incident to v, and the starting point of all edges adjacent to v. Note that this requires the input and output types of v and v' to be compatible. An *operator removal* patch removes v and its connected edges from the plan, and adds new edges to connect all incident operators with all adjacent operators. This requires that the output types of incident operator are compatible with the input types of adjacent operators (e.g., one cannot remove a join). Variant `V4` in our running example requires such an operator replacement to change the way in which a particular attribute is scaled.

Data patches. The third type of patch specifies that a particular operation should be applied to a column of an input data source. In contrast to the other patches, data patches are declarative as they only specify the semantics of the operation to apply to the input column, but no plan location to change. We distinguish between three different kinds of data patches; (i) a *data filter* patch defines a filter on an input and specifies the predicate or user-defined function (UDF) to apply for filtering, the column to filter, and an indication whether to filter the train and/or test data; (ii) a *data projection* patch defines an extended projection, and specifies the read/write set (columns to read and modify as part of the projection) and optionally a selection function in case the projection should only be applied to a subset of the data; (iii) a *data estimator* patch declares that an estimator/transformer (Section 3) should be applied to a particular input column. In Section 2, we use data patches for the variants `V1` and `V2`, which define different ways to modify the test data of the pipeline.

What-if analyses as pipeline patches. Next, we detail how to concretely leverage these patches to express the four data-centric what-if analyses from existing work introduced in Section 2. We can model the data corruptions for `robustness` as *data projection* patches, such that the corresponding extended projection introduces errors, and the corresponding filter function selects random proportions of the test data to apply these errors to. For `preproc`, we define variations for the preprocessing options via *operator replacement* and *operator removal* patches (e.g., to evaluate different ways to normalize numerical attributes or to completely remove normalization). The shuffling of input columns at test time required for `importance` can be expressed with a *data estimator* patch for the test branch. The `cleanlearn` analysis requires a variety of patches: filtering out missing values at training time for example requires a *data filter* patch, while the detection and removal of outliers and duplicates is conducted with *data estimator* patches. Additionally, this analysis requires *model* patches to automatically apply costly label error detection techniques [19, 32].

5 MULTI-QUERY OPTIMIZATION ON PIPELINE VARIANTS

We address our second research question RQ2: *How can we optimize the joint execution of several variants of an ML pipeline for a what-if analysis?* In general, multi-query optimization (MQO) is a very difficult problem in database management systems. This is because efficiently identifying re-use potential between complex incoming queries is already a hard problem, and estimating costs of alternative execution plans is difficult for unseen queries [43]. By design, mlwhatif tackles a simpler multi-query optimization problem for several reasons: (i) our pipeline variants all originate from a single dataflow plan (coming from the original pipeline), and contain only small differences specified by the pipeline patches; (ii) The initial instrumented execution of the original pipeline already gives us valuable information such as the overall runtime of the original pipeline, the output cardinalities and runtimes of individual operators. Based on the characteristics, we design specialized optimization rules tailored to our pipeline patches from Section 3. Our optimization rules are inspired by existing subsumption rules, where the result from one subexpression can be obtained by evaluating an additional operation on another more general subexpression [28, 43, 49].

5.1 Optimization Rules

We detail four optimization rules tailored to our pipeline patches. The goal of these rules is to change individual pipeline variants to increase the shared work between all variants. These optimizations focus on the patches for the original plan. Because all subexpressions until the first patch location are already shared between variants, we only need to consider rewrites to move the patches further up in the plan to potentially re-use more subexpressions.

Projection push-up (PP). Data projection and data estimator patches introduce operations that (in general) modify the values in a set of columns based on another set of input columns. In cases where we have pipeline variants with and without such a patch, we push up the projections in the final plan to re-use more results from the lower parts of the plan, as illustrated in Figure 2. This is always beneficial, as we need to execute the lower pipeline part without the early projection anyway in a different variant, and thereby never introduce additional work. We investigate the read/write set of the extended projection and insert the projection according to the target of the data patch (train or test branch). Next, we push it up as as high as possible by looking for the first operation in the branch that modifies one of the read/write columns of the projection; we push the project directly below it.

If a patched projection only reads from and writes a single column, which is (without further modifications) the input to a estimator/transformer featurizer, then the projection can be pushed up

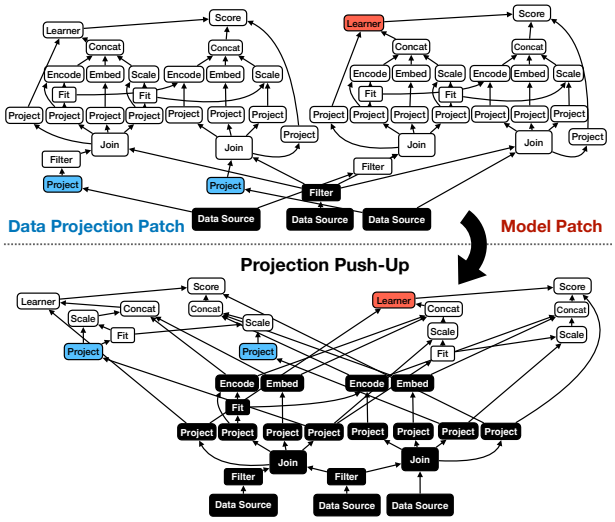


Figure 2: Projection push-up optimization rule (re-usable operators marked in black).

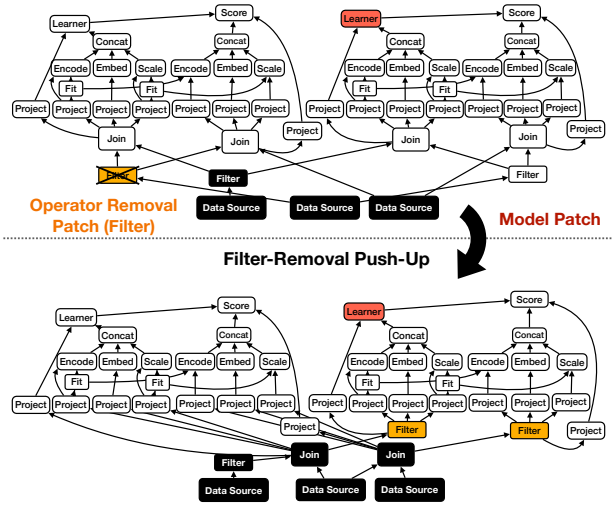


Figure 3: Filter removal push-up optimization rule (re-usable operators marked in black).

just below the aggregation originating from the estimator. Thereby, featurizers for other columns no longer consume data from the patched projection. In absence of other patches, we can then fully re-use their featurization results between variants. This can be especially beneficial for costly featurizers that for example need to compute embeddings.

Filter addition push-up (FAP). Data filter patches introduce additional filters for the input tuples of the train or test branch of a pipeline. In single-query optimization, filters are pushed down to reduce the number of tuples as early as possible. In our MQO setup, if we encounter variants with and without a particular filter, we want to push that filter up as high as possible to re-use results from the variant without the patched filter for the filtering variant (analogous to the projection push-up shown in Figure 2). However we cannot push these filters up as high as projections, due to the aggregations introduced by estimator/transformers to featurize columns. In order to retain the correctness of the pipeline outputs, the filter needs to be applied before every aggregation operator but cannot be pushed over them.

Filter removal push-up (FRP). A case that needs special care are operator removal patches for existing filters. As such a filter is already present in the original pipeline, its removal in a particular variant impacts the shared part with all other variants, if we want to re-use as many intermediate results as possible and push-up the filter. Therefore, we need to adjust all pipeline variants jointly to handle filter removals. If we have a single filter in the plan only, then the optimization works analogously to the filter addition push-up, with the difference that the filter to remove may be located before the train/test split in the original plan, so we need to duplicate it to the train and test branches in the other variants and move it up there, as illustrated in Figure 3. However, the optimization becomes more difficult if we have multiple filters to remove in different variants, as it is not necessarily beneficial to push-up filters in such cases. To decide which subsets of filters to move up in these cases, we use cost-based heuristics, which we will discuss in detail in Section 5.2.

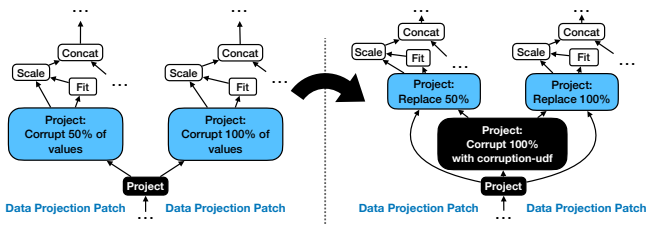


Figure 4: UDF-split-re-use optimization (re-usable operators marked in black).

UDF split-reuse (USR). The extended projections introduced by data projection patches can be costly (e.g., for string operations), and in many cases, we might see patches with the same projection operation, but different selection functions to repeatedly apply the projection on different proportions of the data (e.g., when we want to introduce errors in various randomly sampled fractions of the data). As an example, assume we have two patches with the same projection, but randomly applied to 50%, and 100% of values in a column. Naive execution would require us to apply the projection to 150% (the sum of the previous fractions) of the values in total. Figure 4 illustrates an optimization for such cases, where we apply the projection to 100% of the data once, and then repeatedly sample from this result in the plan variants according to two different required fractions, to save the time for applying the projection to the additional 50% of the values.

Optimization in phases. Our optimization is based on the outlined patches and optimization rules and proceeds in three phases: (*i*) first, each rule can rewrite the original plan. FRP for example affects the original pipeline on which all pipeline variants depend, and has to be applied in this phase. (*ii*) in the second phase, each optimization rule can rewrite individual plan variants, which originate

from pipeline patches applied to the rewritten original plan from the previous phase; We apply FAP, PP and USR here. As USR breaks up projections, it comes after PP which pushes the projections into optimal places. (*iii*) in the third and final phase, the optimized plan variants are merged into a single joint plan and we remove shared work by applying common subexpression elimination.

5.2 Cost-Based Heuristics for Filter Push-ups

Our rule-based optimisations are not always beneficial: FAP and FRP can lead to performance regressions when applied carelessly. To address this, we augment our rule-based approach with simple cost-based heuristics to decide whether to perform rewrites for FAP and FRP.

Failure cases.For FAP, the optimisation is always beneficial if there is at least one pipeline variant without an added filter, because the pipeline operations after the filter addition need to be executed with the full data anyway in at least one other variant. However, if there are filter additions in all variants, the push-up may not help in cases where the selectivity of the added filters is very low. Analogously, for FRP, indiscriminately pushing up all filters can cause performance regressions. Instead, it may be beneficial to only push-up subsets of the to-be-removed filters or to not push filters up at all. An extreme example for this would be a case with two independent filters that each remove 99% of the data, leading to two executions of the following pipeline operations with only 1% of the data without optimisation. If we carelessly pushed both filters up, we would need to execute the operators between the new and old location only once, but with 100% of the data, which could have a strongly negative impact on the runtime.

Heuristics. To prevent such failures for both FAP and FRP, we apply cost-based heuristics based on estimated selectivities. We estimate the selectivities of individual filters for FRP based on the initial run of the original pipeline,[1] by counting the number of input and output tuples of filter operators, and for FAP, what-if analyses can provide bounds on the selectivities for filter additions as an optional argument.[2]

For FAP, mlwhatif estimates the amount of data to be processed with and without the optimisation as follows: for each filter patch in a location, it multiplies the estimated selectivity of the added filters, sums the overall selectivities of all variants, and only performs the push-up if the result is above one. Thus, the optimisation is only applied if executing the operator once with 100% of data is expected to cost less than repeatedly executing the operators with smaller fractions of data.

The heuristic for FRP is again based on multiplying selectivities and counting the number of variants where common subexpressions cannot be re-used. In adversarial cases, where only a subset of filters in one location is worth pushing up, the respective filters will always be the ones removing the least rows. Therefore, mlwhatif sorts the filters by their estimated selectivity, and computes how expensive the execution would be for each growing subset of them. Based on these cost estimates, we can determine the subset of filters whose push-up leads to the lowest expected number of rows to process in case of a push-up. Note that these estimates assume independence between the filters, which might not be true.

In adversarially constructed cases, this strategy could fail, e.g., with two identical filters behind each other, each removing 99% of the tuples. We prevent such edge cases with another heuristic that checks whether the minimum observed selectivity times the number of filters to remove is greater than one. This formula corresponds to the scenario where the selectivities of all filters but the first one are masked by the first filter, so performing the optimisation would not be beneficial.

[1] https://github.com/stefan-grafberger/mlwhatif/blob/9b66ebd7a71d772b96d54513308425e0aec68fc2/mlwhatif/optimization/_operator_deletion_filter_push_up.py#L54

[2] https://github.com/stefan-grafberger/mlwhatif/blob/9b66ebd7a71d772b96d54513308425e0aec68fc2/mlwhatif/optimization/_simple_filter_addition_push_up.py#L28

6 DISCUSSION & LIMITATIONS

We compare our approach to manual implementation of what-if analyses in a Jupyter notebook and discuss its limitations.

Comparison to manual what-if analyses implementation in Jupyter. A competent data scientist could also manually implement the what-if analyses and optimizations applied by mlwhatif. However, this has several disadvantages, as we discuss in the following:

(*i*) Most importantly, the manual implementation work is tedious and time-consuming, especially for external analysis packages such as *CleanML*, which are not designed for integration (as detailed in Section 2). Given the worrying state of data science, where data scientists already spend 60% of their time on data wrangling and not on modeling [61], we consider it to be an important task of data management research to automate what-if analyses and reduce the required implementation time.

(*ii*) As we will experimentally show in Section 9.3, a multitude of optimizations need to be implemented to obtain on-par performance with mlwhatif.

(*iii*) Experimental pipeline code from Jupyter notebooks must be refactored into deployable code for productionization afterwards, which introduces additional implementation effort and potential for bugs. This problem is currently a focus of industry [63, 64]. It is especially tedious in cases where the requirements for what-if analyses on the pipeline are not fully known in advance, and the data scientist has to go through multiple iterations of dissecting the pipeline code into a notebook and refactoring it again afterwards. An advantage of mlwhatif is that it can directly work on the clean refactored code.

Limitations. A major limitation of our approach is that it requires the pipeline code to leverage operations with known semantics (e.g., the pandas `merge` function for relational joins) in order for mlwhatif to extract and instrument dataflow plans. While this limits the applicability of mlwhatif in principle, there are many popular ML frameworks which provide the required relational preprocessing and feature encoding abstractions such as SparkML [30] or Tensorflow Transform [1], to which our approach could be applied in the future.

A second limitation is that a pipeline must contain enough sharable preprocessing work for our optimizations to have a beneficial impact on the runtime of what-if analyses. This should be the case for many real-world pipelines, which spend large parts of their computation time on data preparation and validation [56].

7 IMPLEMENTATION

We now detail how mlwhatif implements the execution and optimization of a what-if analysis for a given pipeline. Each such analysis is written by an expert (and provided to other data scientists later) by implementing two methods in the `WhatIfAnalysis` interface. The `generate_patches` method reads a pipeline plan and outputs a set of `PipelinePatch` instances to apply for the analysis. The `generate_final_report` method receives a dictionary with the extracted intermediate results and output scores after execution, and produces a final report (in the form of a pandas `Dataframe`) as output of the analysis. The actual execution and optimization of a what-if analysis for a given pipeline proceeeds in four stages, which we describe in the following.

Stage 1 – Plan extraction via instrumentation. This stage instruments the function calls in the AST of the original pipeline, and extracts the corresponding dataflow plan with re-executable `processing_funcs` for the operators. For the instrumentation, we extend mlinspect's [16] instrumentation mechanism, already introduced in Section 3. mlwhatif extends the monkey patching and AST rewriting from mlinspect in two ways: (*i*) as part of the monkey patching, when new plan nodes are created, we additionally extract a `processing_func`, which can be used to replay the operator on different input data. We capture these re-executable functions by making the monkey-patched functions callable with different arguments by treating them as partial functions. This way, we get a fully re-executable plan, that we can rewrite and re-execute as needed. (*ii*) We additionally implement runtime tracking via monkey-patching: for each dataflow operation, we track the execution duration and the output shape, to be used for optimization later.

Stage 2 – Pipeline variant generation with pipeline patches. In this stage, the specified what-if analyses create patches to apply. For that, mlwhatif invokes the `generate_patches` function on each analysis with the extracted plan and collects the generated pipeline patches.

Stage 3 – Multi-query optimization on pipeline variants. This stage applies our optimization rules to rewrite and optimally position patches for the pipeline variants. Each such rule implements the `optimize_original_plan` function for plan rewriting and a function `optimize_patches` for patch rewriting and positioning in the `QueryOptimizationRule` interface. Because we work on patches, which already capture the differences between the pipeline variants, implementing this optimizer is much easier than implementing optimizers for typical multi-query optimizers, where efficiently identifying re-use potential between complex incoming queries is already a hard problem. The optimizer proceeds in three phases as outlined in Section 5.

Phase 1. The first phase allows each optimization rule to rewrite the underlying original plan, by calling the `optimize_original_plan` method of each rule.

Phase 2. The second phase rewrites patches and optimally positions them. For that, we invoke the `optimize_patches` method on each optimization rule. This allows our rules to (*i*) rewrite declarative data patches into operator patches which exactly specify their location in the plan, and (*ii*) to optimally position the patches in the plan (e.g., by pushing them up as much possible). Note that this requires the ability to identify the train/test branches of the plan for rewriting data patches. We identify train/test split operators by searching for the lowest common ancestor of the operators conducting model training (on the training data) and inference (on the test data).

Phase 3. The final phase creates the actual pipeline variants, using the rewritten plan from the first phase and the patches from the second phase, and merges them into a single execution plan. We copy the original plan for each variant and modify it by calling the `apply` method of all corresponding patches. Next, we merge the plans for the variants into a single execution plan and apply common subexpression elimination.

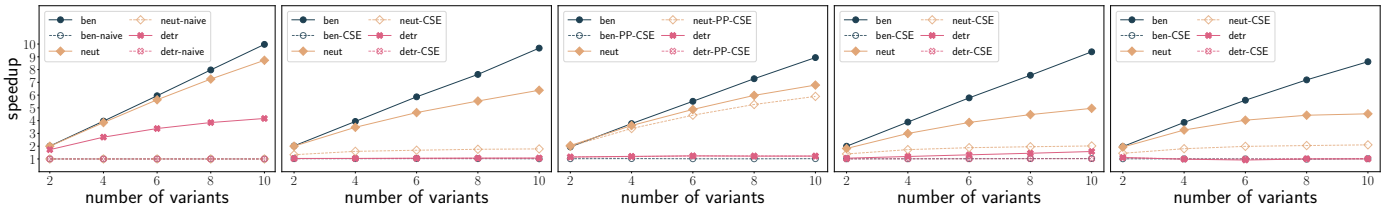
Stage 4 – Plan execution and report generation. Finally, we execute the single optimized plan by invoking the `processing_funcs` of the operators. After the execution of an operator, we replace it with its result in the plan, and have the final consuming child operator clean up the allocated memory by setting the variable to `None`. Finally, we invoke the `generate_final_report` method of the what-if analysis with the output scores computed during plan execution to generate the final report for the user.

8 RELATED WORK

Improving the correctness, efficiency and fairness of ML pipelines at development and deployment time is in the focus of the data management and ML community in recent years [1, 20, 44, 51, 56]. A wide variety of data-centric model and prediction analyses techniques, whose application to pipelines we aim to automate, have been proposed in recent years. Examples include methods for data validation [4, 12, 18, 22, 25–27, 36, 47], data valuation [19, 21] and fairness evaluation [2, 40, 46].

Multi-query optimization [28], based on the subsumption principle [42, 43, 49, 52], is a well-studied area for relational data processing. We are, to the best of our knowledge, the first to propose its application to data-centric what-if analysis on ML pipelines. Closely related to our techniques is work from Xin et al. [55], who optimize a single ML pipeline at different development stages. In contrast to our approach, they do not know the full workload at optimization time, and can therefore only apply a limited set of local optimizations. Derakhshan et al. [9] focus on multi-query optimisation for production ML pipelines with repeated re-training on sliding windows of data, but do not reorder operators to increase re-use potential. Furthermore, both of these approaches cannot work with native ML pipelines, but require users to rewrite their pipelines to custom domain specific languages. mlinspect [16, 17, 45] introduced the instrumentation machinery to work with native ML pipelines, which we build upon, but does only inspect a given pipeline at runtime, and cannot re-execute plan operators or rewrite the plans to perform more advanced analyses.

Our work draws inspiration from research on accelerating the execution of programs written with data science libraries like pandas [33, 37, 41, 48] and on defining a unified representation for relational and ML operators [3, 23, 24, 34]. Further related areas are what-if analyses for relational queries [6, 10, 14, 31] and the outputs of a single ML model [54], as well as the acceleration of a single ML pipeline [50] or featurization workload [38].



(a) Common sub. elim. (CSE). (b) Projection push-up (PP). (c) UDF split-and-re-use (USR). (d) Filter-addition push-up (FAP). (e) Filter-removal push-up (FRP).

Figure 5: Speedup of our optimizations in relation to the number of variants for beneficial (ben), neutral (neutr) and detrimental (detr) scenarios. The speedup scales linearly with the number of pipeline variants in beneficial and neutral scenarios.

9 EVALUATION

To the best of our knowledge, mlwhatif is the only approach proposed for automating and optimizing data-centric what-if analyses for ML pipelines. Therefore, we compare mlwhatif to the sequential execution of the generated pipeline variants as a baseline. We validate that the speedups over this sequential baseline are comparable to the speedups over a manual baseline. For that, we replicate two analysis use cases by manually implementing them with existing stand-alone packages [47, 57]. Note that, we focus on evaluating our approach and optimizations only, and did not built a custom runtime (which is orthogonal to our approach and an interesting avenue for future work). Therefore, we run each experiment with single-threaded execution in Python 3.9 on a machine with AMD EPYC 7H12 2.6 GHz cores and AlmaLinux 8.6. One benefit of mlwhatif, which we do not explicitly address is the automatic generation of the pipeline variants, which potentially saves the data scientists significant development time. We plan to evaluate this with a user study in future work. We provide the source code for our experiments at https://github.com/stefan-grafberger/mlwhatif/blob/9b66ebd7a71d772b96d54513308425e0aec68fc2/experiments.

9.1 Optimization Benefits

The goal of our first experiment is to investigate the benefits of our optimizations in isolation based on different pipeline characteristics. We evaluate how the speedup scales with the number of pipeline variants and the size of the input data.

Experimental setup. We ignore the model training time in this experiment (as it is not affected by the optimizations) by leveraging a no-op model. We apply the optimizations on pipelines consuming two tables with numerical and categorical attributes and randomly generated data. We pre-generate this data and read it from the filesystem in CSV format. In order to account for different pipeline characteristics, we design three scenarios per optimization, which contain pipelines that are beneficial (ben), neutral (neutr) and detrimental (detr) for a given optimization. We aim to showcase how much we can improve the runtime in beneficial and neutral scenarios, but also aim to validate that the optimization does not negatively impact the runtime in detrimental cases. We repeat each experiment seven times with different random seeds, and measure the runtime of the generated optimized execution plan. We compare this runtime against a baseline execution which leverages a subset of the optimizations (e.g., common subexpression elimination), but not the optimization rule under investigation.

Detailed setup per optimization. We discuss the setup per optimization in the following, and refer to the source code of this experiment for further details on the operations used in the beneficial, neutral and detrimental scenarios.

Common subexpression elimination (CSE). We evaluate this optimization with model patches against a baseline which does not apply any optimizations at all. Beneficial scenarios have costly relational preprocessing operations (e.g., a fuzzy join operation that occurs before the model patch) whose results can be re-used, while detrimental scenarios have no relational preprocessing at all before the model patch. Neutral scenarios have preprocessing with low to medium costs (e.g., a regular join and filter, standard scaling, and one-hot encoding).

Projection push-up (PP). We evaluate this optimization with data projection patches and compare it against a baseline applying CSE. Beneficial scenarios for PP have costly relational preprocessing operations (e.g., a fuzzy join and a projection patch that can be pushed above it) or costly featurizers on columns not affected by the projection patch, while detrimental scenarios have no relational preprocessing operations to conduct optimizations on and costly featurizers on affected columns, such as a PCA estimator that has to occur after the projection patch. Neutral scenarios have operations that can occur before the projection patch, and another set of operations that must occur after the patch, but do not dominate the overall runtime (e.g., robust scaling).

UDF split-and-re-use (USR). We evaluate this optimization on data projection patches, and compare it to a baseline applying CSE and PP. Beneficial scenarios for USR have a costly UDF applied to more than 100% of the data (e.g., a costly corruption function on strings, repeatedly applied to large fractions of the data), which dominates the execution time. In contrast, detrimental scenarios have UDFs that get applied to only small proportions of the data (e.g., a string corruption function on only 10% of data across all variants). Neutral scenarios have UDFs, which do not dominate the overall runtime (e.g., adding gaussian noise) but get applied to more than 100% of the data.

Filter addition push-up (FAP). We evaluate this optimization on data filter patches with CSE as baseline. Beneficial scenarios have costly preprocessing operations and filters with a high selectivity, such

as a filter addition patch that removes only 5% of the rows and can be pushed above a costly fuzzy merge. Detrimental cases have filters that remove the majority of tuples early (e.g., a filter addition patch that removes 95% of the rows) before costly preprocessing operations such as a fuzzy join, over which it should not be pushed. Neutral scenarios have filters with a medium selectivity (e.g., removing 20% of rows), before costly operations, which do not dominate the overall runtime (e.g., a regular join).

*Filter removal push-up (*FRP*).* We evaluate this optimization on operator removal patches, deleting filters, with CSE as baseline. Beneficial scenarios have filters that only remove low number of tuples (e.g., filters in a pipeline that each remove only 1% of the rows, followed by a fuzzy join), while the detrimental scenario applies highly selective filters before costly operations like fuzzy joins, which, e.g., remove up to 96% of rows. Neutral scenarios have filters with a medium selectivity followed by before other operations (e.g., multiple filters that together remove 15% of rows before a regular join).

9.1.1 Speedup in relation to the number of pipeline variants. Our first goal is to show that our optimizations result in a speedup that scales linearly with the number of variants. Moreover, we validate that a particular optimization enables us to re-use additional intermediate results between variants, compared to the baseline optimization(s). In this experiment, we only vary the number of pipeline variants generated per scenario from two to ten with a constant data scale factor of one.

Results and discussion. Figures 5(a)-5(e) plot the speedup with and without a particular optimization rule compared to naive execution without any optimizations at all. We find that the speedup widely varies depending on the scenario, the pipeline and the patches. We observe that for the beneficial scenarios (ben), the speedup is close to the number of variants, e.g., between 8.5 and 10 for 10 variants. This means that the optimized runtime is close to the runtime of the execution of a single variant, e.g., that the optimizer amortizes the additional work introduced by the variants. In detrimental scenarios (detr), where, by construction, the optimizations do not save us from repeating work, the speedup is close to one, as expected. For the neutral scenarios (neut), the speedup with optimizations is lower then for ben but still significant with factors of 4 to 8.5 for 10 variants. Importantly, we observe for each scenario, that our optimization rule never underperforms compared to the baseline which does not apply it. This validates that our optimizations do not decrease performance in cases where they do not provide additional speedup.

For CSE in Figure 5(a), we observe that we can save a lot of computation time by reusing the already featurized model inputs. However, in the unrealistic case of pipelines with extremely fast preprocessing operations (as in detr), CSE does not help much, as the test set predictions and score aggregations in every variant dominate the total pipeline runtime. Figure 5(b) shows that PP successfully re-uses results from costly relational preprocessing operations by pushing the projection above them. However, for expensive featurization operations on the columns changed by the patched projection (as in detr), one cannot optimize away this featurization cost. Figure 5(c) illustrates that USR potentially provides large speedups, depending on the relationship between the cost of the UDF and the remaining operations in the pipeline. For FAP in Figure 5(d), we see that pushing filters over expensive relational preprocessing operations potentially helps a lot (as in ben), yet the the speedup is more limited for filters with a low selectivity that remove a large number of rows (as in neutr). FRP in Figure 5(e) again successfully re-uses results from costly relational preprocessing. The speedup depends on the filter selectivities, the costs of the relational preprocessing in the pipeline, and the cost of the operations between the old and new filter location.

9.1.2 Invariance of the speedup to the input data size. Our next experiment aims to validate that the speedup provided by our optimizations is invariant to the size of the input data, and that our optimizations rules always outperform sequential execution and CSE only. For that, we fix the number of pipeline variants to four for all scenarios and scale the data size from a factor of one to four.

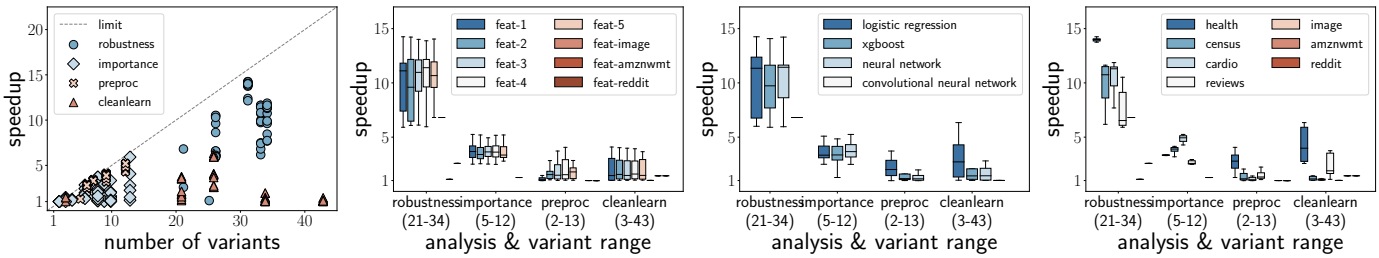
Results and discussion. The results match our expectations: the speedup of our optimizations stays almost constant when varying the datasize; thus, our optimisations work irrespective of the input data size. The beneficial (ben) scenarios keep a nearly perfect speedup of four, while the speedup for the neutral (neutr) scenarios varies between three and four. In the detrimental (detr) scenarios, which do not benefit from the optimization, the speed stays above one, as expected. For CSE, the speedup for the model patches is relatively high even in the detr case, because it can still re-use all intermediates below the model operator. We observe minor variations, which we attribute to operations in the pipelines that do not scale linearly with the number of rows such as fuzzy joins or scikit-learns robust scaling featurizer. As in the previous experiment, our optimizations do not underperform their corresponding baselines, indicating that they add re-use potential to the execution.

9.2 End-to-End Runtime Benefits

The goal of this experiment is to investigate the end-to-end benefits of mlwhatif for various analyses on a wide variety of ML pipelines.

Experimental setup. We evaluate the benefits of mlwhatif for the four different what-if analyses outlined in Section 2: robustness (with 21-34 variants), importance (with 5-12 variants), preproc (with 2-13 variants) and cleanlearn (with 3-43 variants). We compare the end-to-end runtime of the optimized execution of each analysis to the runtime of a baseline execution, which executes all generated variants without applying any optimization.

In order to cover a wide variety of pipelines, we create a "construction kit" of "pipeline pieces" consisting of five different datasets with custom relational preprocessing, six different ways to featurize the resulting train and test data, and four different models. Table 1 lists the five datasets with preprocessing (which are available in our experiment repository), detailing their size, the corresponding data types (tabular, text, image) and the number of joins and filters conducted. The pipelines load one or more CSV files, and implement relational preprocessing with pandas, applying operations such as joining input tables, using filters, fuzzy joins, groupbys, aggregations, deriving new columns, and performing train/test splits in different ways. We include code snippets and normalised variants of established datasets from related research [15, 16, 45, 66, 67]. We aim to include relational preprocessing operations in our benchmarks,



(a) Speedup vs. number of variants. (b) Speedup per featurization. (c) Speedup per model. (d) Speedup per dataset.

Figure 6: Speedups with respect to analyses, datasets, models and featurizations, as well as best case runtimes for our end-to-end experiments on more than 60 pipelines. We encounter speedups of up to 13x compared to sequential execution, and find that the speedup is invariant to the model and featurization in the pipeline.

name	data	data types	#tuples	#joins	#filters
`health`	medical (synth.)	tabular, text	1k	1 (fuzzy)	1
`image`	product images	image, tabular	6k	1	1
`amznwmt`	data integration	text, tabular	25k	4	-
`census`	census	tabular	30k	-	-
`reddit`	mental health	text	40k	-	2
`cardio`	medical	tabular	70k	2	-
`reviews`	product reviews	tabular, text	100k	3	4

Table 1: Datasets with relational preprocessing for the end-to-end experiments (ordered by size).

which represent pipelines of different complexities, and which are challenging to process automatically. The relational data resulting from these preprocessing operations is featurized with six different featurization pipeline pieces `feat1`-`feat5` and `feat-image`, which are inspired by featurizations from the OpenML [53], DSPipes [11], arguseyes [45] and keras [13] projects. The featurizations `feat1` to `feat5` encode numerical, categorical and textual data, using estimator/transformers from scikit-learn [35]. We design the featurizations to become increasingly more expensive (e.g., applying principal component analysis for numerical features in `feat4` instead of simple scaling in `feat3`, or applying UDFs to clean textual data before computing word embeddings in `feat5`). As models, we use logistic regression from scikit-learn [35], gradient-boosted decision trees implemented with xgboost [7], as well as a feed-forward network and a convolutional neural network implemented via keras [8]. In addition, we experiment with two custom pipelines, which we design to be challenging for mlwhatif: `reddit`, a pipeline for a mental health use cases, which consumes JSON data from the reddit API, and learns to distinguish sentences denoting mental health discussions from non-mental health topics by analyzing their embeddings from a neural language model. The second pipeline `amznwmt` is about data integration between product data from Amazon and Walmart [68], and learns to match entities from two tables of electronic products based on several expensive distance measures between their textual descriptions such as the Levenshtein distance and the distance in the embedding space of a BERT model [69]. We run each analysis with every featurization and model (except for the image, reddit and data integration pipelines, which require a custom featurization and model, and where not every analysis is applicable), and repeat each run seven times. In total, we experiment with 63 pipelines and execute 1,732 runs.

Results and discussion. We plot the results in Figure 6. We confirm mlwhatif's' potential to accelerate what-if analysis for data scientists, as we find that it can reduce the runtime by a factor of up to 13x. As expected, the speedup depends on the analysis to run. Figure 6(a) illustrates the speedup achieved for a particular what-if analysis in relation to the the number of variants executed (including the original pipeline execution).

In an end-to-end setup, we always need to execute the original pipeline once to extract its plan, before generating the pipeline variants. Therefore, we always need to pay at least the cost of two executions of the original pipeline (assuming that the variants have a comparable cost). Thus, the maximum possible speedup is the number of pipeline variants divided by two, marked as dashed line in Figure 6(a). We observe that the speedups, which mlwhatif provides for `importance` and `robustness` are close to this theoretical limit. This is because mlwhatif succesfully re-uses a majority of the featurization and model training computation: as these analyses only change the test branch of the pipeline, mlwhatif can re-use the trained model. The speedups for the expensive analyses `preproc` and `cleanlearn` are lower, as the model needs to be retrained in each variant, but mlwhatif still has a decisive practical impact. This is becomes evident, once we investigate the best observed runtime savings per analysis and dataset. For example, mlwhatif reduces the runtime of `cleanlearn` on `reviews` from 104 to 24 seconds, and the runtime of `robustness` on `image` from 95 to 10 seconds. Another noteworthy case is the challenging `reddit` pipeline, which contains very expensive featurizations, where the baseline runtime of 80 minutes is reduced to 29 minutes via our optimizations.

Next, we discuss how mlwhatif's' speedup is influenced by the model, featurization and dataset. Figures 6(c) & 6(b) illustrate that the speedup is relatively independent of the model and featurization applied, and varies mostly with the analysis run. This is an indication that mlwhatif is applicable to a large variety of pipelines. The influence of a particular dataset is more noticeable, as illustrated in Figure 6(d). This is due to the fact that each dataset has different relational preprocessing operations and may or may not use costly text or image features. For `health`, we observe the highest speedups for all analyses but `importance`. This is related to the fuzzy join in the relational preprocessing, and an expensive featurization for a text column. mlwhatif exploits this by not repeatedly executing

the join, which is expensive compared to the other operations in the pipeline, and by re-using featurization results where possible, especially for the text featurization.

For `census` and `cardio`, which both have cheap relational preprocessing and no text column, we observe a higher speedup than for `reviews` with the `robustness` and `importance` what-if analyses. We attribute this to the fact that the test branch for `census` and `cardio` is fast to execute compared to the train branch (which only needs to be executed once as we can re-use the model for these analyses), while the text preprocessing in `reviews` still has high costs even for the test branch. For the `preproc` and `cleanlearn` analyses, which require model retraining in every pipeline variant, `reviews` has higher speedups then `cardio` and `census`. We attribute this, again, to mlwhatif exploiting the expensive relational preprocessing and, more importantly, not having to repeat the expensive featurization of the text column in every variant. For `image`, we observe lower speedups, because this pipeline has a single image column with an expensive image-specific featurization, which we have to completely re-execute for any change.

We find that the speedup for the custom pipelines `reddit` and `amznwmt`, which we designed to be particularly challenging for mlwhatif, is lower than in other cases. This is due to the expensive featurizations `feat-reddit` and `feat-amznwmt`, which dominate the runtime, as they have to compute embeddings for textual data with complex neural networks like BERT. We cannot push operators like data corruption projection patches above these operators, so for every variant, these expensive operators need to be re-executed. However, while the relative speedup is low, the difference in absolute numbers (in one case 29 minutes for the optimized execution compared to 80 minutes for the baseline execution, as discussed previously), is still significant, and can increase the number of variants that a data scientist can try out in a single work day.

In summary, we find that mlwhatif accelerates the execution of what-if analyses in the vast majority of cases, with speedups of up to 13x compared to sequential execution, and that this speedup is invariant to the model and featurization in the pipeline.

9.3 Speedup over Manual Baselines

We validate the results from the previous experiment, in order to showcase the observed speedups also in comparison to manually implemented what-if analyses (which a data scientist working on the pipeline with a Jupyter notebook would implement).

Experimental setup. We choose one of the pipelines from the end-to-end experiment in Section 9.2, and manually implement the what-if analysis variants for the `robustness` analysis based on the *Jenga* [47] package, and for the `cleanlearn` analysis, based on the *CleanML* package. We use the `reviews` dataset with featurization `feat-2` and a logistic regression model. For `robustness`, we execute the original pipeline and 25 variants with five different data corruptions and five corruption fractions. For the `cleanlearn` anlysis, we clean five errors with different strategies, resulting in 20 variants.

We evaluate four different manual baselines, which become increasingly sophisticated and mimic a competent data scientist working on the pipeline code with a Jupyter notebook. The first baseline `naive` leverages a simple for-loop to execute the different variants. The second baseline `load-once` additionally moves the initial data loading before the loop. This mimics the data scientist moving the data loading to a separate notebook cell which is only executed once. The baseline `relational-once` mimics the aforementioned data scientist realizing that the different variants share relational data preprocessing operations like joins. As a result, the data scientist would move the shared relational preprocessing operations into a separate cell, to be only executed once. The fourth baseline `train-once` additionally re-uses the trained model for the `robustness` analysis, which does not require retraining. We compare these manual baselines to mlwhatif, and an additional baseline `auto-gen`, which runs the autogenerated variants from mlwhatif without applying optimizations. We repeat each run 30 times and report its median runtime.

Results and discussion. We plot the results in Figure 7. We find that the runtime of the `naive` baseline (plotted in blue) is on par with the autogenerated variants (plotted in gray) without any optimizations. The `load-once` and `relational-once` baselines, which mimic smart re-use of results in a Jupyter notebook reduce the runtime for both what-if analyses, by a factor of up to 1.91. The `train-once` optimization, which is only applicable to the `robustness` analysis even reduces the runtime by a factor of 3.24. However, in both cases, the automatic optimizations applied by mlwhatif provide a significantly stronger speedup of up to 10.86 for `robustness` (compared to 3.24 for the best manual optimization) and 5.17 for `cleanlearn` (compared to 1.91 for the best manual optimization). The additional speedup comes from pushing up pipeline changes as high as possible, within the featurization of individual columns. Manually implementing this optimization is very tricky, as it would prevent users from leveraging high-level interfaces like scikit-learn's `ColumnTransformer`, and they would have to completely disassemble their pipeline to manually trigger the featurization for each column. Additionally, they would have to keep track of which variant requires which version of which column, manually perform a large number of operator push-up optimisations to apply various data transformations on feature columns copies, and manually implement optimizations like `USR`.

In summary, our findings confirm that mlwhatif not only saves implementation time for data scientists by automatically generating pipeline variants, which would otherwise have to be manually implemented, but also provides significantly lower execution times compared to manual optimisations. Thereby, it potentially allows the data scientists to try out more analyses and variants in the same amount of time, compared to manual experimentation.

9.4 Instrumentation Overhead

Next, we measure the overhead for instrumenting an ML pipeline during execution. Ideally, we want the instrumentation overhead to be so small that users can always execute their pipelines in an instrumented manner during development. This would allow us to use mlwhatif immediately for what-if analysis without first having to execute the original pipeline one additional time to extract the plan. We execute the pipelines from the end-to-end experiment (Section 9.2), and measure the end-to-end instrumentation overhead of mlwhatif, compared to the execution without instrumentation via a function call in Python. Our instrumentation optionally supports

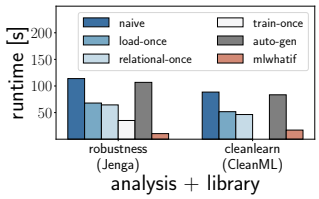


Figure 7: Speedup of mlwhatif over manual and auto-generated implementations.

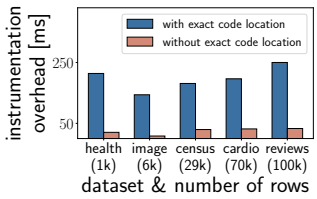


Figure 8: Median pipeline instrumentation overhead per dataset (in ms).

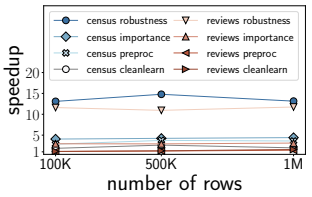


Figure 9: End-to-end scaling of mlwhatif's speedup with the input data size.

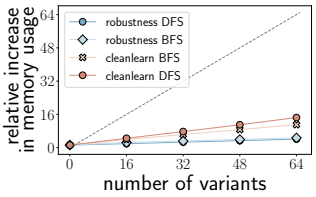


Figure 10: Increase of memory usage for a growing number of variants.

tracking the exact location of operators in the original code. We run the instrumentation with and without code location tracking enabled, and repeat each run 30 times. Figure 8 shows the resulting median instrumentation overheads per dataset. The median overhead without exact code location tracking is below 34 milliseconds for all pipelines, and below 251 milliseconds if exact code location tracking is enabled. This confirms that our instrumentation will not noticeably impact the developer experience, as ML pipelines that train models have to run for several minutes or seconds anyway.

9.5 Scalability & Memory Consumption

Scalability. We investigate whether mlwhatif retains its end-to-end speedups when we scale up the input data.

Experimental setup. We create scaled up versions of the `reviews` and `census` datasets from the previous experiments in Section 9.2 with 100,000, 500,000, and 1,000,000 rows (by including additional historical data from the original data repositories). We execute all four what-if analyses with a pipeline leveraging the featurization `feat-3` and `xgboost` as model on these scaled-up datasets, and compare mlwhatif's runtime to a baseline, which executes the variants sequentially. Due to the high runtime of the sequential baseline (up to 43 hours for a single analysis run), we repeat each run only three times.

Results and discussion. The results in Figure 9 confirm that mlwhatif retains its speedups under growing data sizes. While we see some minor variations (e.g., a slight increase in the speedup for `robustness` on `census` for 500K rows), we do not encounter a single case where the speedup for the pipeline on 1,000,000 rows is smaller than for the pipeline on 100,000 rows.

Memory consumption. We additionally investigate the memory consumption of mlwhatif in relation to the number of variants generated by the what-if analyses.

Experimental setup. We use the `reviews` pipeline with 1,000,000 rows from the previous scalability experiment, and execute the `robustness` and `cleanlearn` analyses with a growing number of pipeline variants between 0 and 64. We do not include `preproc` and `importance` because they generate a fixed number of variants for a given pipeline, and `robustness` and `cleanlearn` already represent cases of analyses with and without model retraining. We again repeat each run three times. We measure the maximum memory consumption during each execution with `psutil`. Additionally, we compare two different execution strategies (traversal in a depth-first or breadth-first order) for the plan generated by mlwhatif.

Results and discussion. We plot the relative increase in mlwhatif's memory usage compared to the memory usage of the original pipeline in Figure 10. We find that the memory consumption scales linearly with the number of pipeline variants, with a low constant factor between 0.06 and 0.23, which implies that we can run a large number of variants. We attribute this positive result to the fact (i) that the intermediate data of the pipeline variants often only differs by a single feature column, which means that many intermediates are completely shared between all variants, and (ii) that we immediately trigger the garbage collection for intermediate results once all dependant nodes are processed (as detailed in Section 7). We also see that the choice of execution strategy (which determines when intermediate nodes can be garbage collected) matters, and that no strategy strictly outperforms the other in all scenarios. We consider optimizing the execution under memory pressure as an interesting direction for future work.

10 CONCLUSION & FUTURE WORK

We proposed mlwhatif to address the need for the automation and optimization of data-centric what-if analyses on ML pipelines, based on our proposed "pipeline patches" and multi-query optimization rules. We find that the speedup provided by our optimization rules scales linearly with the number of pipeline variants in applicable cases, and is invariant to the input data size. In end-to-end experiments with more than 60 pipelines, we encountered speedups of up to 13x compared to sequential execution.

In future work, we plan to include more what-if analyses, and execute our optimized plans with a different runtime (e.g., an analytical database [41, 48] or an ML system such as SparkML [30], Ray [62] or SystemDS [3]. We expect the execution on SparkML to be relatively straighforward, as we could (i) extract our dataflow plans based on its lazily built up lineage graph of Spark's RDD transformations, and (ii) leverage SparkML's existing abstractions (DataFrames, estimator/transformers), which are equivalent to the already supported abstractions from pandas/sklearn. In a distributed setting, we plan to extend our optimizer to exploit the available parallelism in the cluster, e.g., to concurrently execute the independent, non-sharable operations of all variants.

Acknowledgements. This work was supported in part by Ahold Delhaize. All content represents the opinion of the authors, which is not necessarily shared or endorsed by their respective employers and/or sponsors.

REFERENCES

[1] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. Tfx: A tensorflow-based production-scale machine learning platform. *KDD* (2017).

[2] Sumon Biswas and Hridesh Rajan. Fair preprocessing: towards understanding compositional fairness of data transformers in machine learning pipeline. *ESEC/FSE* (2021).

[3] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, et al. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. *CIDR* (2020).

[4] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Whang, and Martin Zinkevich. Data Validation for Machine Learning. *MLSys* (2019).

[5] Leo Breiman. Random forests. *JMLR* 45, 1 (2001).

[6] Felix S. Campbell, Bahareh Sadat Arab, and Boris Glavic. Efficient Answering of Historical What-If Queries. *SIGMOD* (2022).

[7] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *KDD* (2016).

[8] François Chollet et al. Keras. https://github.com/fchollet/keras.

[9] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Zoi Kaoudi, Tilmann Rabl, and Volker Markl. Materialization and Reuse Optimizations for Production Data Science Pipelines. *SIGMOD* (2021).

[10] Daniel Deutch, Zachary G Ives, Tova Milo, and Val Tannen. Caravan: Provisioning for What-If Analysis. *CIDR* (2013).

[11] DS3Lab ETH Zuerich. DSPipes. https://github.com/DS3Lab/datascope-pipelines/tree/main/dspipes.

[12] Lampros Flokas, Weiyuan Wu, Yejia Liu, Jiannan Wang, Nakul Verma, and Eugene Wu. Complaint-Driven Training Data Debugging at Interactive Speeds. *SIGMOD* (2022).

[13] François Chollet. Simple MNIST convnet. https://keras.io/examples/vision/mnist_convnet/.

[14] Sainyam Galhotra, Amir Gilad, Sudeepa Roy, and Babak Salimi. HypeR: Hypothetical Reasoning With What-If and How-To Queries Using a Probabilistic Causal Approach. *SIGMOD* (2022).

[15] Stefan Grafberger, Paul Groth, and Sebastian Schelter. Towards data-centric what-if analysis for native machine learning pipelines. *DEEM workshop @ SIGMOD* (2022).

[16] Stefan Grafberger, Paul Groth, Julia Stoyanovich, and Sebastian Schelter. Data distribution debugging in machine learning pipelines. *VLDBJ* (2022).

[17] Stefan Grafberger, Shubha Guha, Julia Stoyanovich, and Sebastian Schelter. MLINSPECT: A Data Distribution Debugger for Machine Learning Pipelines. *SIGMOD* (2021).

[18] Alireza Heidari, Joshua McGrath, Ihab F Ilyas, and Theodoros Rekatsinas. Holodetect: Few-shot learning for error detection. *SIGMOD* (2019).

[19] Ruoxi Jia, David Dao, Boxin Wang, Frances Ann Hubis, Nezihe Merve Gurel, Bo Li, Ce Zhang, Costas J Spanos, and Dawn Song. Efficient task-specific data valuation for nearest neighbor algorithms. *VLDB* (2019).

[20] Sayash Kapoor and Arvind Narayanan. Leakage and the Reproducibility Crisis in ML-based Science. https://arxiv.org/abs/2207.07048

[21] Bojan Karlaš, David Dao, Matteo Interlandi, Bo Li, Sebastian Schelter, Wentao Wu, and Ce Zhang. 2022. Data Debugging with Shapley Importance over End-to-End Machine Learning Pipelines. https://arxiv.org/abs/2204.11131

[22] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J Franklin, and Ken Goldberg. Activeclean: Interactive data cleaning for statistical modeling. *PVLDB* 9, 12 (2016).

[23] Andreas Kunft, Alexander Alexandrov, Asterios Katsifodimos, and Volker Markl. Bridging the gap: towards optimization across linear and relational algebra. *BeyondMR workshop @ SIGMOD* (2016).

[24] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. An Intermediate Representation for Optimizing Machine Learning Pipelines. *PVLDB* (2019).

[25] Peng Li, Xi Rao, Jennifer Blase, Yue Zhang, Xu Chu, and Ce Zhang. Cleanml: A benchmark for joint data cleaning and machine learning [experiments and analysis]. *ICDE* (2019).

[26] Mohammad Mahdavi and Ziawasch Abedjan. Baran: Effective Error Correction via a Unified Context Representation and Transfer Learning. *PVLDB* 13, 12 (2020).

[27] Mohammad Mahdavi and Ziawasch Abedjan. Semi-Supervised Data Cleaning with Raha and Baran. *CIDR* (2021).

[28] Stefan Manegold, Arjan Pellenkoft, and Martin Kersten. A Multi-Query Optimizer for Monet. *Advances in Databases* (2000).

[29] Wes McKinney et al. pandas: a foundational Python library for data analysis and statistics. *Python for high performance and scientific computing* 14, 9 (2011).

[30] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *JMLR* 17, 1 (2016).

[31] Yuval Moskovitch, Jinyang Li, and H. V. Jagadish. Bias Analysis and Mitigation in Data-Driven Tools Using Provenance *(TaPP '22)*.

[32] Curtis Northcutt, Lu Jiang, and Isaac Chuang. Confident learning: Estimating uncertainty in dataset labels. *JAIR* 70 (2021).

[33] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, et al. Evaluating end-to-end optimization for data analytics applications in weld. *PVLDB* 11, 9 (2018).

[34] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. End-to-End Optimization of Machine Learning Prediction Queries. *SIGMOD* (2022).

[35] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *JMLR* 12 (2011).

[36] Eduardo HM Pena, Edson R Lucas Filho, Eduardo C de Almeida, and Felix Naumann. Efficient detection of data dependency violations. *CIKM* (2020).

[37] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. Towards Scalable Dataframe Systems. *PVLDB* 13, 12 (2020).

[38] Arnab Phani, Lukas Erlbacher, and Matthias Boehm. UPLIFT: Parallelization Strategies for Feature Transformations in Machine Learning Workloads. *VLDB* (2022).

[39] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data lifecycle challenges in production machine learning: a survey. *SIGMOD Record* 47, 2 (2018).

[40] Romila Pradhan, Jiongli Zhu, Boris Glavic, and Babak Salimi. Interpretable data-based explanations for fairness debugging. *SIGMOD* (2021).

[41] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. *SIGMOD* (2019).

[42] Nicholas Roussopoulos. View Indexing in Relational Databases. *ACM Transactions on Database Systems* 7, 2 (1982).

[43] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. *SIGMOD* (2000).

[44] Sebastian Schelter, Felix Biessmann, Tim Januschowski, David Salinas, Stephan Seufert, and Gyuri Szarvas. On challenges in machine learning model management. *IEEE Data Engineering Bulletin* (2018).

[45] Sebastian Schelter, Stefan Grafberger, Shubha Guha, Olivier Sprangers, Bojan Karlaš, and Ce Zhang. Screening Native ML Pipelines with "ArgusEyes". *CIDR* (2022).

[46] Sebastian Schelter, Yuxuan He, Jatin Khilnani, and Julia Stoyanovich. Fairprep: Promoting data to a first-class citizen in studies on fairness-enhancing interventions. *EDBT* (2019).

[47] Sebastian Schelter, Tammo Rukat, and Felix Biessmann. JENGA - A Framework to Study the Impact of Data Errors on the Predictions of Machine Learning Models. *EDBT* (2021).

[48] Maximilian E. Schüle, Luca Scalerandi, Alfons Kemper, and Thomas Neumann. Blue Elephants Inspecting Pandas: Inspection and Execution of Machine Learning Pipelines in SQL. (2023).

[49] Timos K. Sellis. Multiple-Query Optimization. *ACM Transactions on Database Systems* 13, 1 (1988).

[50] Evan R Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J Franklin, and Benjamin Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. *ICDE* (2017).

[51] Julia Stoyanovich, Bill Howe, Serge Abiteboul, H.V. Jagadish, and Sebastian Schelter. Responsible Data Management. *Commun. ACM* (2022).

[52] Subbu N. Subramanian and Shivakumar Venkataraman. Cost-Based Optimization of Decision Support Queries Using Transient-Views. *SIGMOD Record* 27, 2 (1998).

[53] Joaquin Vanschoren, Jan N Van Rijn, Bernd Bischl, and Luis Torgo. OpenML: networked science in machine learning. *KDD* (2014).

[54] James Wexler, Mahima Pushkarna, Tolga Bolukbasi, Martin Wattenberg, Fernanda Viegas, and Jimbo Wilson. The What-If Tool: Interactive Probing of Machine Learning Models. *IEEE Transactions on Visualization and Computer Graphics* (2019).

[55] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. HELIX: Holistic Optimization for Accelerating Iterative Machine Learning. *VLDB* (2018).

[56] Doris Xin, Hui Miao, Aditya Parameswaran, and Neoklis Polyzotis. Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. *SIGMOD* (2021).

[57] CleanML benchmark. https://github.com/chu-data-lab/CleanML.

[58] FairPreprocessing benchmark. https://github.com/sumonbis/FairPreprocessing/tree/c644dd38615f34dba39320397fb00d5509602864/benchmark.

[59] Permutation Feature Importance. https://scikit-learn.org/stable/modules/permutation_importance.html.

[60] Task abstraction in Jenga. https://github.com/schelterlabs/jenga/blob/c219c645c664d2e81b7dfab2c51262e64e20f4ab/src/jenga/basis.py#L25.

[61] Anaconda.com. 2020. The State of Data Science. https://www.anaconda.com/state-of-data-science-2020.

[62] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Pual, and Michael Jordan. Ray: A distributed framework for emerging {AI} applications. *OSDI* (2018).

[63] Linea.py. 2022. Move fast from data science prototype to pipeline. https://lineapy.org.
[64] Databricks. 2022. Mlflow recipes. https://www.mlflow.org/docs/latest/recipes.html.
[65] Ray. 2022. Ray Dataset API. https://docs.ray.io/en/latest/data/api/dataset.html.
[66] Frances Ding, Moritz Hardt, John Miller, and Ludwig Schmidt. Retiring Adult: New Datasets for Fair Machine Learning. *NeurIPS* (2018).
[67] Cardio Data Set. 2022. Cardio Kaggle Dataset. https://www.kaggle.com/datasets/mdshamimrahman/cardio-data-set.
[68] Avanika Narayan, Ines Chami, Laurel Orr, and Christopher Ré. Can Foundation Models Wrangle Your Data?. *VLDB* (2022).
[69] Nils Reimers, and Iryna Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *EMNLP* (2019).