Towards Interactively Improving ML Data Preparation Code via “Shadow Pipelines”

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

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

Sebastian Schelter
schelter@tu-berlin.de
BIFOLD & TU Berlin
Germany

ABSTRACT

Data scientists develop ML pipelines in an iterative manner: they repeatedly screen a pipeline for potential issues, debug it, and then revise and improve its code according to their findings. However, this manual process is tedious and error-prone. Therefore, we propose to support data scientists during this development cycle with automatically derived *interactive suggestions for pipeline improvements*. We discuss our vision to generate these suggestions with so-called *shadow pipelines*, hidden variants of the original pipeline that modify it to auto-detect potential issues, try out modifications for improvements, and suggest and explain these modifications to the user. We envision to apply incremental view maintenance-based optimisations to ensure low-latency computation and maintenance of the shadow pipelines. We conduct preliminary experiments to showcase the feasibility of our envisioned approach and the potential benefits of our proposed optimisations.

ACM Reference Format:
Stefan Grafberger, Paul Groth, and Sebastian Schelter. 2024. Towards Interactively Improving ML Data Preparation Code via “Shadow Pipelines”. In *Workshop on Data Management for End-to-End Machine Learning (DEEM 24), June 9, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3650203.3663327

1 INTRODUCTION

Software systems that learn from user data with machine learning (ML) have become ubiquitous over the last years and participate in many 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, especially with respect to issues in the data they process [2, 18, 21, 26]. As a consequence, several data-centric techniques are being developed to detect, quantify, and improve ML applications with respect to reliability, fairness, and prediction quality [1, 4, 13, 15, 24, 25, 27]. Applying these techniques to ML pipelines currently requires a high level of expertise, as existing approaches such as mlinspect [7, 9], DataScope [14], mlwhatif [6, 8], Rain [28], Gopher [19] or ArgusEyes [22, 23] assume that data scientists know in advance what kind of errors they are looking for.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
DEEM 24, June 9, 2024, Santiago, AA, Chile
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0611-0/24/06
https://doi.org/10.1145/3650203.3663327

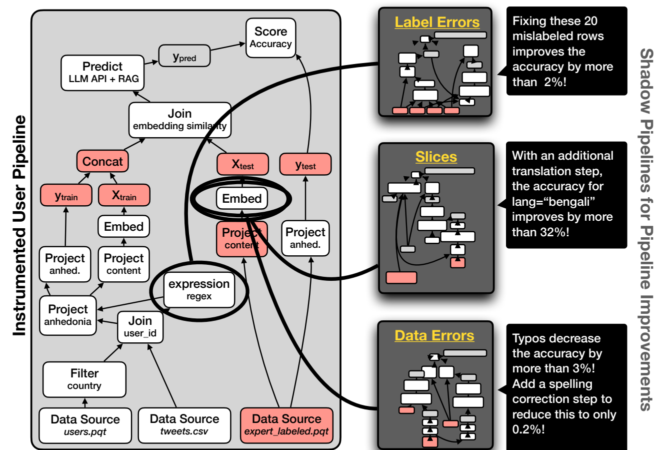

Figure 1: Our vision – several automatically maintained "shadow pipelines" give actionable suggestions on how to improve a user's ML pipeline code at development time.

The need for interactively improving ML pipelines. In reality, data scientists typically do not know in advance what pipeline issues to look for, and often “discover serious issues only after deploying their systems in the real world” [11]. At development time, data scientists currently have to iteratively screen their pipeline for potential issues, debug these issues, and then revise and improve the pipelines according to their findings. This process is tedious, as it requires repeated manual code re-organisation and re-execution in an environment like a Jupyter notebook.

We argue that ML pipeline development should be accompanied by *interactive suggestions* to improve the pipeline code, similar to code inspections in modern IDEs like IntelliJ [12] or text corrections in writing assistants like Grammarly [10]. For that, we can re-use some techniques from previous work, but are still faced with a set of challenges: (*i*) We need *low-latency auto-detection* of pipeline improvement opportunities, to seamlessly integrate into the development workflow; (*ii*) we should identify *pipeline problems spanning several operators*, instead of being artificially limited to screening individual operators one-at-a-time as in e.g., mlinspect [7, 9]; (*iii*) users should receive *provenance-enabled explanations* [3] for detected problems and suggested improvements.

Towards low-latency suggestions for pipeline improvement. We envision a system that instruments a data scientist's ML pipeline code and creates and maintains so-called “*shadow pipelines*” with low-latency to generate suggestions for improvements, as illustrated in Figure 1. Such a shadow pipeline is a hidden variant of

the original pipeline, which modifies it to auto-detect potential issues and tries out different pipeline modifications for improvement opportunities. Subsequently, each shadow pipeline provides the user with code suggestions to improve the pipeline, accompanied by a provenance-based explanation and a quantification of the expected impact on the pipeline outputs. From a technical perspective, the main challenge is to conduct the required computations with low latency by reusing and updating intermediates via incremental view maintenance. We introduce the general problem using a running example (Section 2), present our envisioned approach based on shadow pipelines (Section 3), and finally conduct preliminary experiments for the feasibility of our approach (Section 4). In summary, this paper provides the following contributions:

- We introduce the problem of interactively generating suggestions during ML pipeline development (Section 2).
- We propose *shadow pipelines* as an effective approach to computing such suggestions and discuss ideas for efficiently maintaining them based on incremental view maintenance (Section 3).
- We conduct preliminary experiments with several shadow pipelines to show the feasibility of computing low-latency suggestions, where we find that our proposed optimisations potentially reduce the runtime by up to two orders of magnitude (Section 4).
- We provide the source code for our experiments at https://github.com/stefan-grafberger/shadow-pipeline-experiments

2 PROBLEM STATEMENT

Running example. Imagine a data scientist who is developing an ML pipeline for a social media platform to detect user posts that show signs of the anhedonia symptom for depression.

```
1  def load_data(countries_to_include):
2    users = pd.read_csv('users.csv')
3    posts = pd.read_parquet('posts.parquet')
4    users = users[users.country.isin(include_countries)]
5    return users.merge(posts, on='user_id')
6
7  # Weak labeling with expert rules
8  def signs_of_anhedonia(post):
9    return (post.contains('(0|no|zero)_(motivation)')
10       | post.contains('lost_(interest|motivation)'))
11     & ~(posts.contains('recover_from_(0|no|zero)_interest'))
12
13 # Vectorstore retriever for RAG
14 def setup_retrieval_corpus(train_data, weak_labels):
15   vector_store = Chroma.from_texts(texts=train_data['posts'],
16     metadatas=weak_labels, embedding=...))
17   return vectorstore.as_retriever()
18
19 # LLM + RAG pipeline using Langchain
20 def setup_llm_chain():
21   return ({"context": retriever | format_docs,
22     "question": RunnablePassthrough()}
23     | ChatPromptTemplate.from_template(...)
24     | MyCustomLLM()| JsonOutputParser() | format_response)
25
26 countries_to_include = ['US', 'CAN']
27 train_data = load_train_data(countries_to_include)
28 weak_labels = signs_of_anhedonia(train_data['posts'])
29 retriever = setup_retrieval_corpus(train_data, weak_labels)
30 test_data = pd.read_parquet('expert_labeled_data.parquet')
31 rag_chain = setup_llm_chain(retriever)
32
33 y_predicted = rag_chain.batch(test['posts'])
34 y_test_binarized = label_binarize(test['signs_of_anhedonia'])
35 accuracy = accuracy_score(y_predicted, y_test_binarized)
```

Listing 1: Example pipeline for an NLP use case.

Their pipeline (Listing 1) loads and integrates data (Lines 1-5) and performs weak labeling of the training data, based on regular expressions designed by experts (Lines 7-11). Next, the pipeline computes embeddings of the weakly labeled samples, inserts them into a vector store (Lines 13-17), and leverages these embeddings to classify samples using an LLM with retrieval augmentation (Lines 19-24). In lines 26-35, the final pipeline is then evaluated on a test set (which was manually labeled by experts). Note that this example is inspired by an existing pipeline from NLP research [17].

Iterative improvements for the pipeline. Next, imagine that the data scientist finds that this initial pipeline version provides unsatisfactory accuracy and starts to manually debug and revise the pipeline to figure out how to improve it. They might, for example, write debugging code to look for mislabeled training samples via Shapley Values [14] and detect that some wrongly labeled samples are fed to the LLM from the retrieval corpus. After mapping the embedded samples back to the original posts, the data scientist would pinpoint the problem to the regular expressions used for weak labeling. They would inspect the problematic posts and extend the regular expressions to cover the cases that were not handled well. Next, the data scientist would re-run the updated pipeline and observe that the accuracy improved a little but that there are still instances of likely mislabeled samples. They again isolate the samples to which the label error detection points and realise that some users' posts contain too many typos to be reliably matched. As a result, the data scientist would once again rewrite the pipeline to integrate a spelling correction step. Once no more likely mislabeled samples are detected, the data scientist would try other evaluation techniques, e.g., to find small slices in the test data where the pipeline does not perform well [4, 20] and start another cycle of iterative debugging and rewriting.

Towards automated, low-latency suggestions for improvements. As already mentioned in the introduction, data scientists spend a lot of time in this iterative process of "guessing" what a potential problem might be, rewriting the pipeline to determine if that is indeed the case and then testing and integrating improvements. Furthermore, this tedious process requires a high level of expertise, as the data scientists need to identify the actual improvement opportunities and concrete steps to fix the detected problems.

We argue that we should automate the underlying steps in the outlined iterative improvement cycle as follows. First, common techniques for identifying problematic portions of the data or the data preparation operations (like label error detection or the detection of problematic data slices) should be *automatically run on the pipeline in the background*. Once potential issues are detected, data scientists should be presented with *provenance-based explanations* for the locations in the data or pipeline code that cause the issues. Furthermore, the data scientist should also be presented with a list of *suggestions on how to improve their pipeline*, which ideally already *quantify the expected impact* on a metric of interest such as accuracy or fairness.

Existing systems such as mlwhatif [8] or ArgusEyes [23] fall short to address this problem, as they assume that the data scientist already knows exactly what issues to look for. They are designed as one-off approaches with no consideration of the iterative development cycle, and in general incur rather high execution times.

3 ENVISIONED APPROACH

In the following, we describe our ideas for automatically maintained shadow pipelines to assist the user with ML pipeline development.

Shadow pipelines. Figure 1 gives a high-level overview of our envisioned approach, in reference to the running example from Section 2. On the left, we see an automatically extracted "logical plan" of a user's ML pipeline, consisting of both relational and ML-specific operators. We showed in previous work [6, 7] how to extract such plans by instrumenting Python code. For the user pipeline, we generate so-called shadow pipelines (shown on the right). A shadow pipeline is a hidden variant of the original pipeline, which modifies it to auto-detect potential issues and tries out different pipeline modifications to detect improvement opportunities. Each shadow pipeline consists of several general steps. The first step is issue detection, for which the shadow pipeline introduces operators that take intermediates from the original pipeline as input to screen for potential problems. For detected issues, a shadow pipeline continues with root cause determination, aiming to localise specific pipeline operators and/or input tuples responsible for the identified issue. Finally, the shadow pipeline integrates and evaluates potential fixes, generates the corresponding provenance-based change explanations, and provides a quantification of the expected impact on the pipeline outputs. To prevent information overload, shadow pipelines will have to use various techniques to carefully select tuples for user explanations, e.g., via stratified sampling, slice-finding [4, 20], based on Shapley values [13, 14], and by focusing on test samples with label changes.

Low-latency computation of shadow pipelines. A key technical challenge is ensuring low-latency shadow pipeline computations for an interactive user experience. For that, we envision to reuse intermediates from the original pipeline, sometimes even on a tuple-level, via incremental view maintenance (IVM) techniques [16]. Every shadow pipeline must only compute the minimum required difference compared to the original user pipeline and should ideally avoid costly operations like model re-training. In cases where such re-training is unavoidable, we plan to explore cheap proxy models [13] to estimate the impact of a change on expensive models like neural networks. Moreover, certain issue detection techniques and potential fixes can be evaluated on selected subsets of the data only, when faced with costly operations like LLM inference APIs or embedding computations for text or image data. Some shadow pipelines may themselves involve expensive operators like slice finding, Shapley value computation, spelling correction, or text translation. Efficient implementations of these operators will be crucial, and we may even need to work with samples only in extreme cases. Furthermore, the parallel execution of different shadow pipelines and the prioritisation of different analyses will be crucial for performance.

Maintenance of shadow pipelines. During the iterative development cycle of data scientists, they continuously rewrite and re-run their pipelines. In light of such rewrites, we again plan to update the original pipeline and its shadow pipelines with IVM techniques, based on detected changes in operators or input tuples. If the user followed a concrete code suggestion from our shadow pipeline, we may even be able to update the original pipeline with results previously computed in the shadow pipeline. However, in general, updating ML pipelines presents some tough challenges, as IVM approaches [16] typically assume a fixed query with changing input data, while in our scenario, both data and operators can change between iterations. Therefore, we plan to apply a best-effort approach, where we only apply IVM if certain simplifying assumptions are met, e.g. that only a single operator changes.

Application to the running example. Consider the exemplary "label errors" shadow pipeline shown in Figure 1, which uses Shapley values [13, 14] to identify likely label errors in the user pipeline's weakly labeled samples. Next, the shadow pipeline simulates fixes in the regexes by flipping the labels of samples with negative Shapley values in the train data, and re-evaluates the pipeline to observe the impact on its accuracy. In the initial shadow pipeline run, we can directly apply the Shapley value algorithm [14] on intermediates captured from the original pipeline. Next, we can update the labels of these problematic samples directly in the vector store metadata, and thereby reuse the previously computed train embeddings, which do not change. Finally, the shadow pipeline only needs to rerun the expensive inference for test set predictions that relied on train samples with flipped labels. Similar optimisations can quickly update the shadow pipeline with low latency in case of user changes, and an incremental version of the Shapley value algorithm could even decrease the latency further.

4 PRELIMINARY EXPERIMENTS

We present preliminary experiments on computing and maintaining shadow pipelines with hardcoded optimisations. Our goal is to validate the feasibility of continuously managing these pipelines in the background as a user works on their ML pipeline.

User pipelines. We experiment with two versions of the NLP pipeline from our running example in Section 2. The first version, called `rag`, uses retrieval augmentation and a trained LLM as detailed in Listing 1. The second version, called `train`, does not use an LLM but trains a simple neural network classifier on the embeddings.

Shadow pipelines. We evaluate three shadow pipelines inspired by the issues outlined in Section 2: `slices` uses SliceLine [20] to detect subsets of the test set where the user pipeline does not work well. These samples correspond to posts in non-English languages; therefore, `slices` integrates an additional translation step for posts from these languages as a fix. The `data-errors` shadow pipeline determines how robust the user pipeline is against data quality issues such as typos by evaluating synthetic errors, and integrates an additional spell-checking step to improve the robustness. `Label-errors` uses Shapley values [13, 14] to identify potentially mislabeled training samples from weak labeling, subsequently flips the label of the most likely mislabeled samples and re-evaluates the pipeline. All shadow pipelines also compute provenance-based explanations.

Optimisations. For `slices`, SliceLine [4, 20] is applied directly on the original pipeline's intermediates (parts of the unfeaturised test data and the corresponding labels) to identify the most problematic slice. In `data-errors`, we randomly corrupt 10% of the data with typos, and recompute embeddings and run inference only for corrupted rows to measure the impact. For potential fixes, we analogously restrict expensive translation and spelling correction

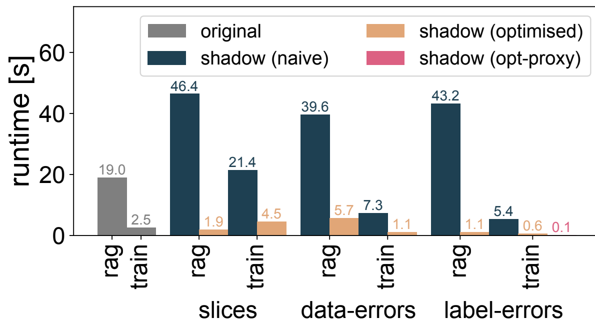

Figure 2: Benefits of our proposed optimisations for computing shadow pipelines. Our shadow pipeline optimisations decrease the runtime by up to a factor of 38.

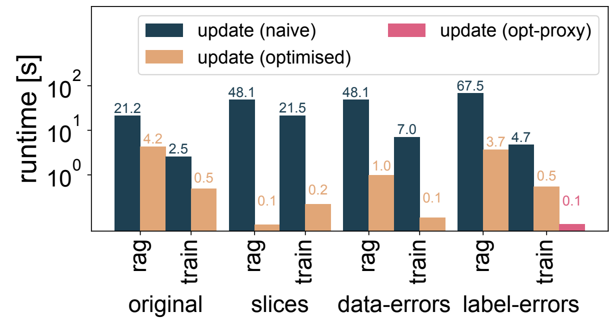

Figure 3: Benefits of our proposed optimisations for incrementally updating the pipelines. The runtime (shown on a log scale) is less than one second in all but one scenarios.

steps, embedding computation, and inference to relevant test subsets. `Label-errors` also changes the train data. For `rag`, we ensure that the expensive inference is re-run only for test set predictions based on train samples with flipped labels. The `train` variant of the user pipeline, unfortunately, requires neural network retraining; for that, we evaluate a special optimisation referred to as `opt-proxy`, which substitutes the neural network with a cheaper model and uses its performance difference as a proxy for the label update's impact. For subsequent shadow pipeline updates after the user's pipeline change, we re-use intermediates from the original shadow pipeline run wherever possible. In particular, we partially re-use translation, typo corruption, spell correction, and inference results.

Data and APIs. We use a synthetic dataset with 1,000 rows, inspired by real depression detection data [17], since the original dataset cannot be shared due to its sensitive nature. For reproducibility, we fix random seeds and replay cached answers from external APIs for the LLM and translation step with artificial wait times. We repeat each experiment seven times, preceded by one warm-up run, and report the median runtimes.

Computing shadow pipelines. We measure the execution time for computing the shadow pipelines after first executing the original pipeline. In each case, we compare the runtime of the `naive` baseline (repeated execution of manually rewritten versions of the original pipeline) to the runtime of the `optimised` execution, where we implement hardcoded shadow pipelines with our proposed optimisations.

Results and discussion. Figure 2 illustrates that the optimised shadow pipelines decrease the runtime by up to a factor of 38 (55 with the proxy model). The optimised variant consistently runs at least 4.7 times faster in all scenarios. We find that the runtime reduction primarily depends on three key factors: inference time, costly operations within shadow pipelines (e.g., spell correction, translation), and the need for retraining (not applicable for `rag`). Substituting neural network retraining with a proxy model results in a noticeable runtime improvement of 55, whereas the variant without a proxy model only achieves a speedup of nine. Notably, for the scenarios with the highest runtime (`slices` on `train` and `data-errors` on `rag`), the initial shadow pipeline execution accounts for the majority of the runtime due to expensive operators, where we can reuse intermediate results for shadow pipeline maintenance. It is noteworthy that code inspections in modern IDEs like IntelliJ can also incur several seconds of latency, which seems generally acceptable for users, with the option to disable specific inspections.

Maintaining shadow pipelines. In the second experiment, we simulate that a user makes a small change to their pipeline code (the update of a regex), and then measure how long it takes to update the results of both the original pipeline and its shadow pipelines.

Results and discussion. The incremental shadow pipeline maintenance is up to 626 times faster than the full naive re-execution, with original pipeline runtimes decreasing up to fivefold. The additional re-use of intermediates from previous shadow pipeline executions leads to bigger runtime decreases than their initial optimised execution in the previous experiment (where we only saw decreases by up to a factor of 55). Additionally, shadow pipeline maintenance takes less than one second in all but one scenarios. The label flipping for the most likely label errors in `label-errors` on `rag` affects train rows retrieved for many test predictions, which leads to high inference costs, which we cannot avoid. Notably, we now observe particularly low latencies for the scenarios where we observed the highest runtimes in the previous experiment (`slices` on `train` and `data-errors` on `rag`), as we can reuse expensive intermediates from the initial execution of the respective shadow pipelines.

5 NEXT STEPS & OPEN QUESTIONS

We plan to explore how to create and efficiently maintain shadow pipelines for user-defined pipelines, by automatically applying our outlined optimisations. In particular, we aim to develop a runtime for incremental view maintenance and shadow pipeline parallelisation (e.g., by building on DuckDB). Additionally, we plan to conduct a user study to evaluate the usefulness of our envisioned system and its provenance-based explanations for data scientists. Several questions remain open, including whether to directly build certain optimisations into the shadow pipelines or build a custom optimiser, the extent to which we will need custom incremental operators, the reliance on sampling, and approximations such as proxy models. Furthermore, we aim to explore the integration of LLM-based code suggestions [5] for detected data-related problems into our system.

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] Sarah Bird, Miro Dudík, Richard Edgar, Brandon Horn, Roman Lutz, Vanessa Milan, Mehrnoosh Sameki, Hanna Wallach, and Kathleen Walker. 2020. Fairlearn: A toolkit for assessing and improving fairness in AI. *Microsoft, Tech. Rep. MSR-TR-2020-32* (2020).

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

[3] Zaheer Chothia, John Liagouris, Frank McSherry, and Timothy Roscoe. 2016. *Explaining outputs in modern data analytics.* Technical Report. ETH Zurich.

[4] Yeounoh Chung, Tim Kraska, Neoklis Polyzotis, Ki Hyun Tae, and Steven Euijong Whang. 2019. Slice finder: Automated data slicing for model validation. In *2019 IEEE 35th International Conference on Data Engineering (ICDE).* IEEE, 1550–1553.

[5] GitHub. 2021. GitHub Copilot · Your AI pair programmer. https://copilot.github.com/.

[6] Stefan Grafberger, Paul Groth, and Sebastian Schelter. 2023. Automating and Optimizing Data-Centric What-If Analyses on Native Machine Learning Pipelines. *SIGMOD* (2023).

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

[8] Stefan Grafberger, Shubha Guha, Paul Groth, and Sebastian Schelter. 2023. mlwhatif: What If You Could Stop Re-Implementing Your Machine Learning Pipeline Analyses over and over? *Proc. VLDB Endow.* 16, 12 (aug 2023), 4002–4005. https://doi.org/10.14778/3611540.3611606

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

[10] Grammarly. [n. d.]. Demo. https://demo.grammarly.com/.

[11] Kenneth Holstein, Jennifer Wortman Vaughan, Hal Daumé III, Miro Dudik, and Hanna Wallach. 2019. Improving fairness in machine learning systems: What do industry practitioners need?. In *Proceedings of the 2019 CHI conference on human factors in computing systems*. 1–16.

[12] Jetbrains. [n. d.]. Code inspections. https://www.jetbrains.com/help/idea/code-inspection.html#access-inspections-and-settings.

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

[14] Bojan Karlaš, David Dao, Matteo Interlandi, Sebastian Schelter, Wentao Wu, and Ce Zhang. 2023. Data Debugging with Shapley Importance over Machine Learning Pipelines. In *The Twelfth International Conference on Learning Representations.*

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

[16] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow.. In *CIDR*.

[17] Thong Nguyen, Andrew Yates, Ayah Zirikly, Bart Desmet, and Arman Cohan. 2022. Improving the Generalizability of Depression Detection by Leveraging Clinical Questionnaires. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, Dublin, Ireland, 8446–8459. https://doi.org/10.18653/v1/2022.acl-long.578

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

[19] Romila Pradhan, Jiongli Zhu, Boris Glavic, and Babak Salimi. 2022. Interpretable data-based explanations for fairness debugging. In *Proceedings of the 2022 International Conference on Management of Data*. 247–261.

[20] Svetlana Sagadeeva and Matthias Boehm. 2021. Sliceline: Fast, linear-algebra-based slice finding for ml model debugging. In *Proceedings of the 2021 International Conference on Management of Data*. 2290–2299.

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

[22] Sebastian Schelter, Stefan Grafberger, Shubha Guha, Bojan Karlaš, and Ce Zhang. 2023. Proactively Screening Machine Learning Pipelines with ArgusEyes. *SIGMOD* (2023).

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

[24] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating Large-Scale Data Quality Verification. *Proc. VLDB Endow.* 11, 12 (aug 2018), 1781–1794. https://doi.org/10.14778/3229863.3229867

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

[26] Julia Stoyanovich, Bill Howe, Serge Abiteboul, H.V. Jagadish, and Sebastian Schelter. 2022. Responsible Data Management. In *Communications of the ACM*.

[27] Florian Tramer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, Jean-Pierre Hubaux, Mathias Humbert, Ari Juels, and Huang Lin. 2017. Fairtest: Discovering unwarranted associations in data-driven applications. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 401–416.

[28] Weiyuan Wu, Lampros Flokas, Eugene Wu, and Jiannan Wang. 2020. Complaint-driven training data debugging for query 2.0. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1317–1334.