Complaint Driven Training Data Debugging for Machine Learning Workflows

Lampros Flokas

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2023

Abstract

Complaint Driven Training Data Debugging

for Machine Learning Workflows

Lampros Flokas

As the need for machine learning (ML) increases rapidly across all industry sectors, so has the interest in building ML platforms that manage and automate parts of the ML life-cycle. This has enabled companies to use ML inference as a part of their downstream analytics or their applications. Unfortunately, debugging unexpected outcomes in the result of these ML workflows remains a necessary but difficult task of the ML life-cycle. The challenge of debugging ML workflows is that it requires reasoning about the correctness of the workflow logic, the datasets used for inference and training, the models, and interactions between them. Even if the workflow logic is correct, errors in the data used across the ML workflow can still lead to wrong outcomes. In short, developers *are not just debugging the code, but also the data*.

We advocate in favor of a *complaint driven* approach towards specifying and debugging data errors in ML workflows. The approach takes as input user specified *complaints* specified as constraints over the final or intermediate outputs of workflows that *use* trained ML models. The approach outputs explanations in the form of specific operator(s) or data subsets, and how they may be changed to address the constraint violations.

In this thesis we make the first steps towards our complaint driven approach to data debugging. As a stepping stone, we focus our attention on complaints specified on top of relational workflows that use ML model inference and whose errors are caused by errors in ML model's training data.

To the best of our knowledge, we contribute the first debugging system for this task, which we call Rain. In response to a user complaint, Rain ranks the ML model's training examples based on their ability to address the user's complaint if they were removed. Our experiments show that users can use Rain to debug training data errors by specifying complaints over aggregations of model predictions without having to specify the correct label for each individual prediction.

Unfortunately, Rain's latency may be prohibitive for use in interactive applications like analytical dashboards or business intelligence tools where users are likely to observe errors and complain. To address Rain's latency problem when scaling to large ML models and training sets, we propose Rain++. Rain++ pushes the majority of Rain's computation offline ahead of user interaction, achieving orders of magnitude online latency improvements compared to Rain.

To go beyond Rain's and Rain++'s approach that evaluates individual training example deletions independently we propose MetaRain, a framework for training classifiers that detect training data corruptions in response to user complaints. Thanks to the generality of MetaRain, users can adapt the classifiers chosen to the training corruptions and the complaints they seek to resolve. Our experiments indicate that making use of this ability results in improved debugging outcomes.

Last but not least, we study the problem of updating relational workflow results in response to changes to the inference ML model used. This can be leveraged by current or future complaint driven debugging systems that repeatedly change the model and reevaluate the relational workflow. We propose FaDE, a compiler that generates efficient code for the workflow update problem by casting it as view maintenance under input tuple deletions. Our experiments indicate that the code generated by FaDE has orders of magnitude lower latency than existing view maintenance systems.

Table of Contents

List of Figures

List of Tables

Acknowledgements

This thesis would not have been possible without the support of my collaborators: Eugene Wu, Jiannan Wang, Nakul Verma, Weiyuan Wu, Yejia Liu and Alexander Yao. I am forever grateful for their hard work, collaborative spirit, endless enthusiasm, relentless commitment and dedication, insightful advice and unwavering compassion. Taking this ambitious, arduous but also fulfilling research journey together was an honor of a lifetime for me.

I would also like to thank the members of my thesis committee: Kenneth Ross, Luis Gravano, Eugene Wu, Jiannan Wang and Nakul Verma. Their attention to detail, valuable feedback and thought-provoking questions played an important role in shaping this thesis. Their input across the whole process was instrumental in defining the positioning of this thesis.

Last but not least, I would like to express my gratitude to the members of the Columbia Database Group, Department of Computer Science and the Columbia community at large for creating an environment conducive to learning and research excellence. The friendships forged and memories made throughout the last 7 years were an integral part of my Columbia PhD experience.

Chapter 1: Introduction

1.1 Motivating Applications

Machine learning (ML) is increasingly a core part of a company's technical infrastructure, yet introduces considerable complexity. It requires creating, executing, and managing ML workflows that perform data extraction, labeling, model design and (re-)training, inference, and using the predictions as part of downstream analytics or applications. To facilitate this adoption, numerous ML platforms (e.g., TFX [1], Michelangelo [2], FBLearner Flow [3], Overton [4], MLFlow [5], and others [6]) have been proposed to manage and automate parts of the ML life-cycle.

As a running example, CompanyX (name anonymized) is a company that manages email campaigns for their enterprise customers. They train models to estimate user characteristics (e.g., churn likelihood, product affinities, etc). This empowers their customers to use user attributes and these models to define and monitor user cohorts used in email campaigns. Figure 1.1 shows the ML workflow that trains a churn rate prediction model and uses it to count the number of active users likely to churn, as well as the associated query that performs inference using the trained model.

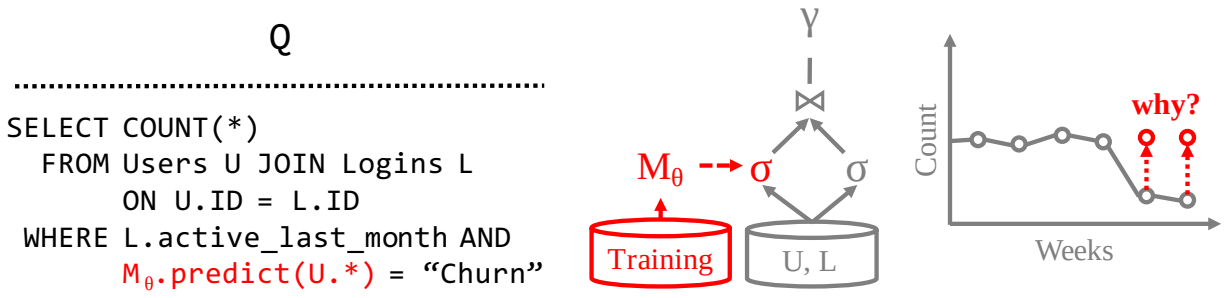


Figure 1.1: CompanyX workflow and output visualization where the user specifies surprising output values. Training and model inference steps in red.

Unfortunately, the power of ML—that it adapts the model to training data—also presents a

risk due to the increased difficulty of debugging when errors arise. Debugging ML workflows requires reasoning about the correctness of all ML workflow components including data loading and preprocessing, model training and inference, the downstream workflow logic and the interactions between them. In the CompanyX example, a customer is alarmed that the size of their user cohort dropped considerably and wants to understand why. Yet there can be a multitude of reasons: the query may be incorrect, there may be errors in the `Users` or `Logins` tables, the model may be misspecified, and there could be errors in the training data. In short, developers *are not just debugging the code, but also the data*.

1.2 Existing Approaches and their Limitations

Traditional workflow and query debugging tools are capable of helping debug query [7] and query data (e.g. the `Users` table in CompanyX example) [8, 9] errors, however, they do not typically address data errors that affect ML models. To solve this problem, modern ML platforms provide ways to check the correctness of the data. For instance, they can detect schema and data type mismatches [10, 11], and provide methods to define and detect dataset distribution shifts between training, test, and/or production [10]. Further, systems such as [11] support "data unit tests" where developers can provide user-defined functions that compose a list of primitive statistical data-checks. Unfortunately, these approaches both place considerable burden on the developer to specify checks, and are limited in their efficacy for several reasons.

First, it is a tall order to ask developers to specify the data constraints that will guarantee stable performance of their downstream ML workflow. Even if an ML application developer knows the key performance indicators for his application it may be difficult for him to specify which features are critical to maintain their stability. For example, CompanyX's engineers may have an insight that large cohort size drops in a single day are strong indications of data-related errors. In contrast, it's not clear how they can specify which feature distribution shifts are important for each of their clients. This is especially true because adding more checks is not necessarily helpful. Generic data checks may generate many false positive alarms that reduce the likelihood that true errors

are responded to [10]. This is because not all training data quality issues (e.g., corrupt values, distribution shifts, missing data) cause noticeable errors in the downstream workflow results. As a hypothetical example, let us assume that churning users are over-represented in the training data as compared to the serving data. This may not lead to a biased model in production and in fact it can be even necessary to get a high quality model if churning users are a minority.

Second, is that these approaches primarily focus on *detecting* that a constraint was violated, but do not provide direct support to debug the workflow and identify *why* the violation occurred. This latter functionality is important because in comparison to data constraint violations, workflow constraints involve ML inferences and are thus harder to debug. For example, once the CompanyX customer noticed the cohort drop, it is useful to understand that it was due to a corrupt subset of training data records that introduced bias in the model.

1.3 Advantages of Complaint Driven Debugging

To address the above limitations, this thesis advocates for a *complaint driven* approach towards specifying and debugging data errors in ML workflows. The approach takes as input *complaints* specified as constraints over the final or intermediate outputs of workflows that *use* trained ML models. The approach may then output explanations in the form of specific operator(s) or data subsets, and how they may be changed to address the constraint violations.

There are a number of benefits of this approach. As discussed above, a key advantage is that complaints can be easier to specify. Complaints are akin to traditional software bug reports that ask the user to describe the problem that they encountered. The only difference is that they are expressed as logical statements over the outputs similar to assertions in traditional software debugging. In the CompanyX example, the constraint is simply that the cohort size should > 150.

Further, complaints can help different stakeholders collaboratively debug an ML workflow. This is because different teams are often responsible for subsets of a ML workflow—a data modeling team may focus on model design, an ML engineering team implements and manages the training and deployment of the models, and end users execute queries that use the model predictions. Similarly,

a given model may be used in multiple analysis workflows—different customers make use the same ML models to define different user cohorts and performance metrics. Each of these people have visibility into a portion of the full ML workflow, yet can contribute their domain expertise to provide more accurate workflow debugging.

Complaints can also provide weak signals that can help debugging even in the absence of labels during serving. For example, label noise during training could be easily detected if we had access to the labels during serving. While exact serving labels might be unknown, violation of application specific constraints like the ones in CompanyX's case can be used to detect label noise phenomena. Another prominent case is debugging ML workflows involving multiple models. Once again, the correct intermediate predictions in a multi-model ML workflow may be unknown. However, violations of constraints on their aggregate outcome is a strong indication that at least one serving prediction is incorrect and thus they can be used to initiate debugging.

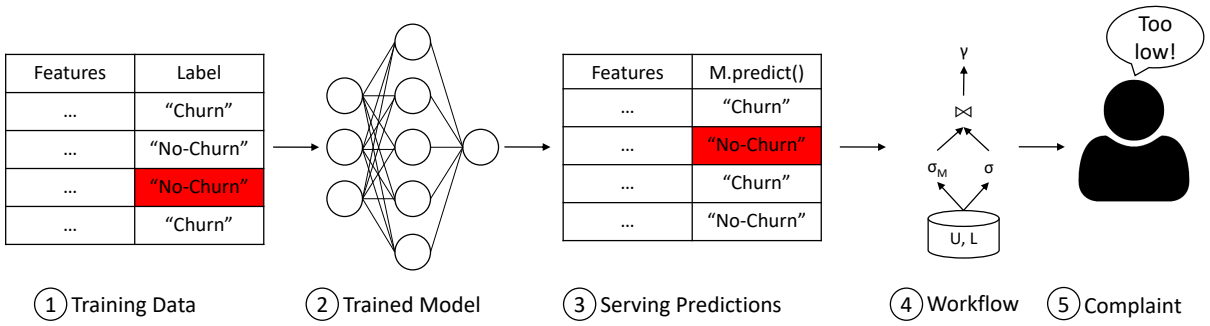


Figure 1.2: Label errors in the training data of CompanyX's workflow can lead to mispredictions in the serving data (both shown in red). Complaint driven training data debugging aims to leverage user complaints on surprising workflow outputs to deduce the training data errors.

1.4 Challenges of Complaint Driven Debugging

Despite the benefits complaint driven debugging can bring, it also carries new computational challenges that are not obviously solvable. The output of a complaint driven debugging system is an explanation in the form of proposed interventions to specific operator(s) or data subsets. Given a list of such candidate interventions, a complaint driven debugging system needs only to rank them based on how well they address the constraint violations. Unfortunately, evaluating the effect of

each intervention on the user's complaint may require re-executing the whole workflow, including model training for training data interventions, which is prohibitive.

Clearly, we need approaches that go beyond applying interventions, rerunning the workflow and finally checking if the user's complaint is resolved. White box approaches, knowing the types of models and operators used in the workflow, can leverage workflow properties to accelerate complaint driven debugging. However, developing white box complaint driven debugging systems that can debug large families of ML workflows remains challenging.

One challenge is that working step by step by debugging each workflow component in isolation is not always sufficient. For example, in an effort to debug Figure 1.2's training data, one might attempt to first identify which active user "No-Churn" predictions in (3) are mistaken. Then given these mispredictions, one can then identify the training data errors that caused them. The problem here is that in the first step there are several ways to choose these "No-Churn" predictions in order to at least partially resolve the complaint. Failing to choose the true mispredictions in the first step can lead to wrong intervention suggestions in the second. Picking a solution for the first step without consideration for the model and its training data in the second step can lead to suboptimal results.

We call complaints that can be resolved by changing the model's serving predictions in multiple ways *ambiguous*. Complaint ambiguity introduces a new level of complexity not shared by traditional ML debugging using validation sets where the labels of validation set examples are known. Handling ambiguous complaints allows us to make use of the weak signals complaints provide.

Another challenge pertains the variability of components in ML workflows. From training data preprocessing to model training to the usage of the model in serving, an ML workflow contains multiple components that may be written in a variety of libraries or even languages, and bear different properties like differentiability and continuity that can help debugging [12, 13]. Efficient and automated ML workflow debugging requires reasoning about all these components and their cascading interactions in a unified way that goes beyond step by step execution.

1.5 Outline and Contributions

This section outlines the rest of the thesis. Chapter 2 provides our complaint driven debugging vision, the scope of this thesis as well as the concrete problems we aim to solve in the following chapters. Chapter 3 presents the necessary background knowledge that our work relies on. Chapters 4 to 7 propose novel solutions to the problems introduced in Chapter 2, followed by a conclusion. The rest of this section summarizes Chapters 2 to 7 and our contributions.

1.5.1 Preliminaries

The space of all white box complaint driven debugging system designs is vast. Studying this space would involve analyzing all combinations of model types, workflow operators, candidate intervention types, types of user complaints, data quality issues. Instead, we restrict our attention to a small subset of this landscape as a stepping stone towards more general complaint driven debugging systems. Specifically, the focus of this thesis across the following dimensions is:

Error location and type This thesis studies training data debugging. Training data errors are particularly challenging for naive black box approaches. This is because they need to retrain the model for every suggested intervention on the training data, making them prohibitive. As a result the white box approaches studied here have the potential to bring big efficiency wins.

Regarding error types, we do not make assumptions about the error type. Instead, the underlying assumption of complaint driven debugging is that among available interventions, intervening on erroneous training data is the most effective way to resolve the complaint. To verify the usefulness of our complaint driven debugging systems, our experiments cover synthetically introduced errors of varying types and distributions as well as datasets with preexisting data quality problems.

In Section 2.1 we describe our vision for a general complaint driven training data debugging system. In Section 2.2 we explain how existing black box complaint debugging approaches are prohibitive when applied to training data debugging because they need to retrain the model for each intervention. In response to this limitation, we focus on white box approaches.

Workflow operators Following Figure 1.1, we focus on workflows that post-process the predictions of the trained model using SQL workflows. Given that SQL is in general Turing complete, we further restrict each SQL query in the workflow to be a Select-Project-Join-Aggregate (SPJA) query. The key property that makes this class tractable is that the corresponding operators have very simple control flow characteristics allowing us to efficiently capture how prediction changes affect the workflow output. For example, relational filters and joins require only a simple "if" condition for each candidate output tuple whereas group by aggregations like COUNT, SUM, or AVG do not require control flow at all. This allows us to capture and store execution traces that describe all potential execution paths of the query in an efficient way. Workflows with multiple models can be very complex as prediction errors can cascade through a model pipeline. For this reason, this thesis focuses on the simpler case of a single model.

At the same time, these workflows are expressive enough to capture commonly used queries in business intelligence and dashboard applications. These applications are particularly important for complaint driven debugging since users can visually identify anomalies in model behavior with greater ease. For example, CompanyX engineers may use SPJA queries to track prediction statistics for many serving data slices of interest. Visualizing the progression of these statistics over time makes it easier to detect sudden changes as is the case in the complaint of Figure 1.1.

In Section 2.3.1 we present the proliferation of SQL in the data life-cycle as motivation for the focus of this thesis. In Section 2.3.2 we provide additional motivation based on several applications of complaint driven training data debugging of SQL workflows. Section 2.4.1 describes the supported SQL queries and what makes these queries more tractable for debugging purposes.

Models We focus on differentiable probabilistic classifiers, which include commonly used models like logistic regression and neural network classifiers. This class of models very important as deep neural networks have been shown to achieve state of the art results in a wide variety of applications including computer vision [14, 15] and natural language processing [16]. The special properties of these models enable us to efficiently approximate the effects of training set changes to the optimal model parameters as well as the outcomes of the downstream operators.

Section 2.4.2 goes into more detail into the class of supported models, why this model class enables fast approximate debugging and its usefulness in practical applications.

Complaints Given the outputs and intermediates of a supported SQL workflow, we support two types of complaints. The first one allows users to identify violated constraints on individual attributes of each tuple. For example, in Figure 1.1, the users can complain that the aggregate count value is too low. The second one allows users to specify that a given tuple should not even exist in the output. More details can be found in Section 2.4.3.

Interventions One class of supported interventions are the ones that directly modify the training data by deleting or updating individual training examples. In this case, the complaint driven debugging system returns a ranking of the per training example interventions. Alternatively, we also propose a system that captures patterns of training example interventions in the form of a classifier. The task of this classifier is to decide whether each training example should be deleted or not in order to resolve the complaint. In Section 2.5 we state more formally the concrete complaint driven training data debugging problems that this thesis aims to address.

Section 2.6.1 is devoted to the limitations of our complaint driven debugging framework. The systems proposed are meant to assist users in their debugging process. Vetting of the suggested interventions remains necessary because the interventions suggested may not have lead to positive outcomes. One case is intervention suggestions based on incorrect user complaints. For example, in Figure 1.1 a user may incorrectly specify that the "Chrun" prediction count should be higher leading to interventions that make the model worse. Additionally, interventions may happen to resolve a complaint without fixing training data errors. Moreover, interventions may have side effects like introducing unintended biases. While we recognise that supporting users in their vetting process, we leave these problems for future work.

Section 2.6.2 analyzes the risks of blindly applying the suggested interventions. Users can degrade model performance by applying fixes for incorrect complaints. Training data errors may continue to fester because users may resolve their complaints with unrelated interventions or interventions that resolve the effects of the errors but not their cause. Users can also impose their

own biases on the model raising ethical concerns. There is a spectrum of risk associated with the extent of training data modifications used to resolve user complaints. In the most extreme case, one can manufacture a training dataset from scratch based on constraints on a target downstream distribution [17]. We compare and contrast with alternative approaches that propose training data modifications including data cleaning [18], data augmentation [19] and domain adaptation [20].

1.5.2 Background

In Chapter 3 we present existing work in ML optimization and relational provenance that will act as a basis for our work in Chapters 4 to 7. From the ML optimization community we will cover influence functions [21, 22], a tool that will allow us to estimate the sensitivity of the parameters and predictions of a differentiable ML model to changes in its training set. From the relational provenance community we will cover provenance metadata capture [23, 24, 25, 25, 26, 27], its connections to view maintenance under deletions [28, 29] and how this framework can be adapted to our complaint driven debugging problem.

1.5.3 Addressing Ambiguity Using Differentiable Query Relaxations

In Chapter 4 we propose Rain, the first system for training data debugging driven by complaints over relational workflows. The input to Rain is a relational workflow, a SQL query using ML inference, as well as the user-provided complaints expressed as violated constraints over the queries intermediate and/or final result sets. In order to identify potential bugs in the training data of the model, Rain aims to return a minimal subset of training examples which if removed address the user's complaints. Recognizing that a naive solution would require retraining an exponential number of ML models, we propose two heuristic solutions which use only linear retraining steps for the case of differentiable models like linear regression or neural networks.

The first solution operates in a two step fashion. First it aims to fix the serving predictions of (3) of Figure 1.2 by modifying the predictions so that the complaint of (5) is resolved. Then at the second step it tries to find training examples of (1) whose removal yields a trained model

making the aforementioned predictions in (3). Unfortunately for ambiguous complaints like the one in Figure 1.2 there is no unique way to fix the serving predictions of (3) in order to resolve the complaint. Even if we fix the predictions of (3) with the minimal amount of changes, this does not guarantee that the second step will require the minimal amount of training example deletions to be enforced. This is because not all serving predictions are equally sensitive to training example deletions. We empirically verify that that this two step approach struggles for ambiguous complaints.

Addressing complaint ambiguity requires jointly reasoning about the complaint, the model and its training set. To achieve this, our second approach relaxes the entire pipeline from (1) to (5) into a single differentiable function. This allows us to approximate the sensitivity of the complaint to training example deletions directly in an end to end fashion. Our relaxation of the non differentiable relational workflows relies on creating a symbolic representation of it based on the workflow's provenance. Our in depth experimental results consistently show that using this second approach, complaints can be a powerful tool. A single complaint over an aggregate result, despite it being ambiguous, can be as effective at recovering corrupted training records as specifying the labels of hundreds of individual serving predictions.

1.5.4 Precomputation Techniques for Interactive Debugging

In Chapter 5 we seek to reduce the high latency of Rain when scaling to large models or training sets. Similar to our example in Figure 1.1, we envision that users will issue complaints through a visual interface where anomalies in the query result can be easily detected. In this context, it is crucial that Rain responds at interactive timescales so as not to impede the user's analysis flow. Unfortunately Rain, despite its many optimizations, imposes high latency requirements since it needs to estimate the effect of each individual training example on the workflow. To remedy this we push a significant portion of Rain's computation offline ahead of the user interaction time.

As a first step we consider the setting where the training data of (1) and the trained model of (2) are known ahead of time. Without the complaint, we cannot compute the sensitivity of the pipeline in an end to end fashion. But we can still compute the sensitivity of the trained model's

parameters to training example deletions. Unfortunately for state of the art models like deep neural networks the space requirements for such an approach are onerous and the latency improves at most by constant factors. We leverage insights from the optimization literature to compress the sensitivity information of the model and at the same time reduce the online latency. Perhaps counter intuitively, our compression scheme encodes the sensitivity information of the least sensitive model parameters.

We extend our compression scheme to capture more precomputation settings. These include cases where the serving data of (3) are known in advance but the workflow of (4) is not, as well as cases where the workflow is fixed but the serving data arrive in a streaming fashion. We incorporate all these optimizations in a new system which we call Rain++. Our experimental results show up to 70000× latency improvements over Rain.

1.5.5 Identifying Training Data Corruption Patterns

In Chapter 6 we propose a complaint driven debugging system that returns patterns of training data errors instead of the training set subsets as in Rain and Rain++. Our motivations is that errors in the training set are usually introduced in batches rather than in isolation. For example, CompanyX may miss all new churning events of one of its clients due to a data integration error resulting in a batch of training examples being mislabelled. The common cause of these errors typically leads to all of them having a shared pattern. In CompanyX's case, all erroneous training example correspond to users of the same company. Not only do these shared patterns make the errors potentially easier to find but to fix as well. CompanyX's engineers can address the problem at its root by fixing the integration error instead of correcting the erroneous examples one by one.

Unfortunately, Rain and Rain++ cannot be easily modified to capture patterns of training set errors. Their approximation techniques are not accurate for estimating the effects of deleting several training examples in one go. As a result they are ill equipped to evaluate the effects of deleting a group of tuples that satisfy a pattern. Additionally, users do not control which training example attributes Rain and Rain++ should look into to identify the shared patterns.

To sidestep these challenges, we frame error pattern detection as a classification problem. A

classifier, termed the denoising model, is trained to detect which training examples are erroneous or not. We note that the denoising model and the model getting debugged, which we term the target model, are distinct. Despite this, we show how the denoising model can be trained by specifying complaints over the outputs of the target model. This framework, which we call MetaRain, enables users to infuse the debugging process with additional domain knowledge, by choosing the features and structure of the denoising model. Our experiments indicate that adapting the denoising architecture to the corruption and complaint at hand leads to improved denoising classifier performance, highlighting the importance of MetaRain's generality.

1.5.6 Accelerating Workflow Execution Using Provenance

Cust. Id	Purchases	M.predict()
100	$10	"Churn"
101	$200	"No-Churn"
102	$4000	"No-Churn"
103	$30	"Churn"

One-Hot Encoding

Cust. Id	Purchases	M.predict()	exists
100	$10	"Churn"	True
100	$10	"No-Churn"	False
101	$200	"Churn"	False
101	$200	"No-Churn"	True
102	$4000	"Churn"	False
102	$4000	"No-Churn"	True
103	$30	"Churn"	True
103	$30	"No-Churn"	False

Figure 1.3: The right table is a one-hot encoded representation of the left table. Each row not corresponding to a model prediction of the left table is marked as deleted by setting its "exists" annotation to "False". Thus updates to the model predictions in the left table can be equivalently translated to row deletions over the right table.

In Chapter 7 we shift our attention to accelerating the evaluation of the relational workflow for debugging purposes, an improvement that can be directly utilized by Rain and Rain++ as well as any future training set debugging system that relies on running the workflow several times. Our key observation in Figure 1.2 is that while the `Users` and `Login` tables as well as the serving model predictions of (3) can be nominally considered inputs of step (4), only the prediction column of (3) can change for the needs of training set debugging.

Inspired by the long line of work on partial evaluation in the programming language community, we seek to replace the original implementation of the workflow with a specialized and more efficient version that has the model predictions of (3) as its only input while treating the tables of `Users` and `Login` as well as the features column of (3) as constants known ahead of time.

Thinking of (3) as a database table, we can view evaluating the workflow for a new set of model predictions as a view maintenance problem. In Figure 1.3 the left table presents an instance of (3) of Figure 1.2 where we have two model features for each customer, her id and her purchases. Given a new model's prediction, we can update the values of the M.predict() column. View maintenance techniques [30] can then be used to update the workflow result in response to these updates.

In fact we observe that restricting our attention to view maintenance that only allows row deletions is sufficient for our use case. In Figure 1.3 the right table is generated by creating two copies of each row of the left table, one for the "Churn" prediction and one for the "No-Churn" prediction. To evaluate the workflow for a new model, we first delete all the rows except the ones representing the true model predictions. These rows are indicated by setting their "exists" annotation to "False". Updating the workflow in response to these deletions in the right table yields the same results as view maintenance under updates for the left table.

While existing approaches in the provenance literature [28, 29, 24] are applicable, naively applying them leads to specialized programs that suffer from poor data locality. This is because the specialized programs generated by these approaches are decoupled from the relational operators of the query's plan missing the data locality opportunities that they provide. To circumvent this issue, we propose an alternative code generator that annotates each operator in the relational workflow with its provenance information and generates specialized code on a per operator basis. Our techniques allow for small code size with improved data locality and easy parallelization and lead to several orders of magnitude latency improvements over existing systems.

Chapter 2: Preliminaries

In Section 2.1 we start with our vision for a complaint driven training data debugging system. We envision a tool that enables upstream developers and downstream domain experts to cooperate in the training data debugging process. We provide a high level definition of the complaint driven training data debugging problem as well as its inputs and outputs.

Section 2.2 discusses how existing black box approaches can in principle be used to rank interventions for any complaint. However, they incur a prohibitive computational cost as evaluating the effect of training data interventions requires model retraining and rerunning the workflow. To sidestep this issue, instead of black box one size fits all solutions we need specialized white box approaches that make use of workflow properties to accelerate debugging.

Section 2.3 explains the reasons why we chose to focus on relational workflows that use ML inference, a class of workflows which we term Query 2.0[1] workflows. Section 2.3.1 argues that SQL is becoming increasingly popular when it comes to expressing data pipelines and managing the data lifecyle of ML workflows. This includes the rising interest from both industry and academia to integrate ML inference in databases. Section 2.3.2 provides motivating use cases of complaint driven training data debugging for Query 2.0 workflows.

Section 2.4 defines the subset of Query 2.0 workflows that we will support in this thesis. In general, supporting Query 2.0 workflows that can use arbitrary SQL queries and train arbitrary ML models remains intractable. In Section 2.4.1 introduces the subset of downstream SQL queries we support. Section 2.4.2 does the same for the class of supported models, namely probabilistic differentiable classifiers. Section 2.4.3 presents the supported complaints. Where applicable, we explain how the constraints on the supported workflows makes debugging more tractable.

Section 2.5 motivates and defines the three complaint driven training data debugging problems

[1] In analogy to Machine Learning as "Software 2.0" [31, 32, 4]

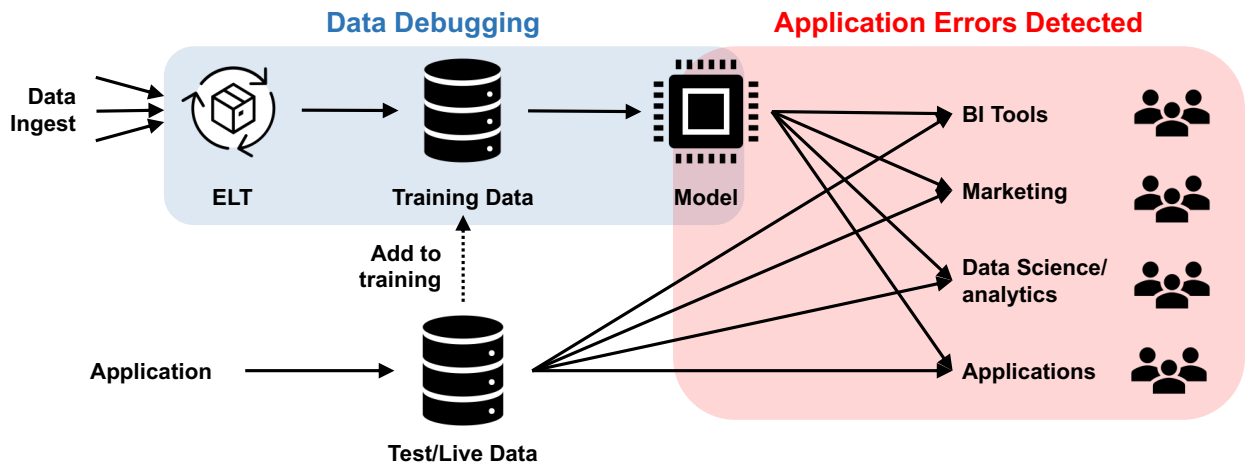


Figure 2.1: ML workflows produce models used by many teams and applications. Application-level data errors are necessarily detected downstream (red), however existing data debugging approaches (blue) ignore this downstream information. Complaint driven data debugging aims to bridge this gap by leveraging downstream knowledge expressed as complaints for upstream data debugging.

that Chapters 4 to 7 aim to solve. Section 2.5.1 focuses on debugging systems that return a set of per tuple interventions. Section 2.5.2 adds the additional constraint that the returned interventions need to form a pattern. An example of such a constraint is that all training examples chosen for deletion need to satisfy a binary conjunction over the training features. Section 2.5.3 introduces the batch view maintenance problem under deletions, a building block that can be used both by the systems proposed in this thesis as well as general complaint driven debugging systems.

Section 2.6 analyzes the limitations and risks of complaint driven training data debugging systems in Section 2.6.1 and Section 2.6.2 respectively. In light of these risks and limitations, we note that some form of vetting of the proposed interventions by the users is necessary. Thus the goal of the systems proposed in this thesis is to assist users in their debugging process, not replace them. Developing systems that can facilitate the intervention vetting process is left for future work.

2.1 Complaint Driven Training Data Debugging Vision

We consider all data-oriented processes parts of a workflow (Figure 2.1). Early parts ingest, prepare, clean, and extract data. Training data are then used to fit ML models, which are used in downstream applications (e.g., anomaly detection, monitoring, exploration, BI analytics, recom-

mendations, etc). Test data (e.g., user-generated content, forms, sensors) may be pre-processed and fed to models for inference and eventually gets labelled and added to the training corpus.

Different teams own different subsets of the worklow. For example, data engineering teams may manage different ingestion processes, a ML team is responsible for cleaning and model development, business analysts use BI tools that rely on trained ML models and the operations team are responsible for logging new test data and the model's prediction statistics.

Here, developers in charge of training data quality may not even communicate with downstream model users. Yet, downstream users have domain-expertise to best assess and annotate result errors as complaints. A complaint is a constraint over any derived data (e.g., the distribution of loan rejections should be more equal between genders). Informally, the core problem is as follows:

Problem 1 (Complaint Driven Debugging)**.** *Given a set of* ***complaints****, along with the* ***execution traces*** *for the complaints' data, to identify the minimal set of training data* ***interventions*** *needed to address the* ***errors that led to the complaints****.*

Each of the terms above is intentionally left open-ended, as different instantiations will lead to a wide variety of complaint driven data debugging problems and corresponding solutions.

Complaint: A complaint is a violated assertion over intermediate or final results, for instance whether a result value should instead be x. More complaints better aid automated data debugging, and different teams can specify complaints at the granularity/abstraction most natural to them: end users may complain over aggregated results while labellers identify individual model mispredictions.

Intervention: The relevant training data errors depend on the complaints and use case, and interventions differ by *type* and *granularity*. For types, interventions may remove (e.g. duplicates), modify (e.g. impute an outlier value) or even add training examples (e.g., label a record). For granularity, interventions can be per-tuple or per-group/predicate (e.g., remove all rejections of highly paid females from the training set). Intervention minimality helps avoid overfitting to the complaints, and tuple and group interventions should strive to insert, delete or modify the smallest number of training examples to best address the complaints.

Error: Model errors (and complaints) are due to different types of discrepancies between training and inference data. These include erroneous training examples (e.g. noisy labels, features or both) that do not appear in the inference data, and distributional differences between training and inference data. The latter can arise by data collection biases or different feature extraction procedures in training and inference. While all training examples might individually be correct, these differences can force the model to learn false associations.

Execution Trace: Workflow visibility, as encoded by the granularity of the execution traces, affects the computational efficiency of the debugging system. On one hand, the bare minimum is to treat the training pipeline as a black box and only log for each choice of interventions if it resolves the complaints. On the other hand, execution traces that encode the fine grained lineage allows for working backwards from the complaints to find the interventions that resolve them.

2.2 Black Box Debugging Approaches and their Limitations

As discussed above, execution traces are entirely optional when it comes to any form of complaint driven debugging. The bare minimum is to be able to evaluate the effect of interventions and see if it resolves the complaint or not. We call the approaches that do not use execution traces *black box* since they can be adapted to arbitrary workflows with little to no effort.

Black box approaches may differ in the interventions they consider, how they explore the space of available interventions as well as how they choose out of all the interventions that happen to resolve the complaint the ones that fix the actual causes of the complaint.

Many approaches focus on returning explanation for the complaint causes in the form of conjunctive predicates over the data. Scorpion [8] develops both bottom-up and top-down algorithms for finding deletion conjunctions that the complaint is most sensitive to. BOExplain [33] solves the same problem but formulates the search for influential conjunctions as a Bayesian optimization where the optimization variables correspond to the conjunction parameters. BugDoc [34] seeks minimal conjunctions over workflow configuration variables that are satisfied by at least one failing workflow run, but none of the succeeding ones.

Resolving the complaint alone does not always result in finding the root cause of the complaint. To this end, approaches impose additional restrictions on the error patterns they report. For example, DataPrism [35] has instead access to two runs of a workflow with two distinct datasets, one where the complaint constraint is violated and one where it is not. Any error pattern returned should not appear in the dataset of the successful run. At the same time, fixing the errors with the available interventions should resolve the complaint of the failed run. When there are causal dependencies between the variables used to form explanation predicates, they can be used by black box approaches to find predicates that are causing the complaints and not only correlated with them [36].

Black box approaches use the structure of the intervention space and/or use additional information or constraints to guide the search process towards an explanation of the complaint. These approaches are agnostic to the types of operations executed in the workflow. As a result, they aim to reduce the number of workflow evaluations instead of making each evaluation cheaper. Unfortunately for training data debugging even evaluating the effect of one intervention can take considerable time as it requires retraining the model.

Accelerating the evaluation of each individual intervention is also required to enable fast complaint driven training data debugging. White box approaches not only open up additional workflow specific ways to reduce intervention evaluations but also allow for ways of efficiently computing the effect of interventions or at least approximations thereof.

2.3 Why Relational Workflows?

In this thesis we will focus on workflows where the downstream analytics component can be expressed by SQL queries. We are motivated by recent trends in adoption of SQL in all parts of the data lifecycle including the integration of ML inference in databases as well several real world use cases. In addition, as we will see in the following chapters, this choice will allow us to accelerate the evaluation of intervention effects on the outputs of the downstream processes.

2.3.1 SQL-ization of the Data Life-cycle

In contrast to ad hoc Hadoop [37] or Spark jobs [38], arbitrary extraction scripts and black-box data science scripts, nearly every part of the data life-cycle is increasingly "relationalized". The lakehouse paradigm [39] advocates for declarative transformation from unstructured to structured data. This is powered by ETL/ELT tools like dbt [40] that allow organizations to specify data transformations and extractions as a huge collection of SQL queries integrated into the software engineering process. SQL can also be leveraged in ML preprocessing pipelines. Feature stores [41] allow organizations to centralize the management of ML preprocessing transformations. This avoids duplication of code and compute across teams and can eliminate discrepancies between training and ML inference preprocessing. Additionally, end-user analytics tools such as tableau [42] or powerBI [43] inherently express their logic as SQL.

Database researchers have long advocated the value of integrating model inference and training within the DBMS: data used for model inference and training is already in the DBMS, it brings the code (models) to the data, and it provides a familiar relational user interface. Early libraries such as MADLib [44] provide inference functionality by leveraging user-defined functions and type extensions in the DBMS. Model training inside the DBMS can also leverage factorized representations to accelerate learning of ML models trained over join outputs [45]. The recent and tremendous success of ML in recommendation, ranking, predictions, and structured extraction over the past decade have led commercial data management systems [46, 47, 44, 48] to increasingly providing first-class support for in-DBMS inference: Google's BigQuery ML [48] integrates native TensorFlow support including model training [49], and SQLServer supports ONNX [50] models. These developments point towards mainstream adoption of this new querying paradigm that we call Query 2.0. Many companies already leverage Query 2.0 in their core business with the case of CompanyX we discussed in Chapter 1 being just one example. Beyond CompanyX, both industry [51, 46] and research [52, 53, 54, 55, 56] are advocating for Query 2.0.

Together, these trends point to a future ecosystem where data is imported, loaded, cleaned, modeled, processed, and used in—conceptually—a large SQLized data pipeline that spans the

entire organization as in Figure 2.1. Different teams manage different portions of the workflow, and boundaries between steps may span individuals, teams, or companies. Individuals can interact with data at any stage of the pipeline. For instance, business analysts may use BI tools that process data near the end of the workflow, and end users may only see the final outputs; data engineers may manage the workflow near the source, where data is ingested, whereas ML modeling may be downstream from the data engineers. As this end to end workflow becomes more complex, allowing downstream complaints to be used to debug issues upstream becomes increasingly important.

2.3.2 Query 2.0 Complaint Use Cases

Training data errors can cause the trained models to mispredict leading to errors in the downstream analyses and thus to complaints. This section presents illustrative use cases of ML workflows with downstream SQL components that can benefit from complaint driven training data debugging.

E-commerce Marketing: CompanyX specializes in retail marketing. One of its core services manages email marketing campaigns for its customers. 10-20 ML models predict different user characteristics (e.g., will a user churn, product affinity). Customers see model predictions as attributes in views, and can use them, or raw user profile data, to create predicates to define user cohorts that are used in email campaigns and tracked over time as in Figure 1.1.

For development simplicity, CompanyX uses Google BigQuery for model training, cohort creation, and monitoring; the queries are instrumented at different points to be visualized or monitored purposes. For example, customer-facing metrics dashboards visualize user cohort sizes over time, and customers can set alerts for when the cohort's size drops or increases very rapidly, or exceeds some threshold.

CompanyX collects training data by scraping their customers' e-commerce websites. However, changes to the website—such as adding a new check-out step, or changing a product category—can introduce systematic training errors that degrade the re-trained models, and ultimately trigger customer monitoring alerts and lead to customer questions. Pipeline monitoring is not enough to

pinpoint the relevant training records, and their engineers are challenged to find and characterize the culprit training records.

Entity Resolution: A data scientist scrapes and trains a boolean classification model to use for entity resolution. Given two business records, the model determines whether they refer to the same real-world entity. However, when she uses it as the join condition over two business listings ($Listings_1 \bowtie_{M_\theta.predict(*)=1} Listings_2$), she finds that the dining business categories have zero matches. She is sure that should not be the case and suspects that training data quality is the reason that the classifier is incorrect.

Image Analysis: An engineer collects an image dataset and wants to train a hot-dog classifier. To create labels, she decides to use distant supervision [57], and writes a programmatic labelling function. She uses the classifier to label a hot-dog, and a non-hot-dog dataset, equi-joins the two datasets on the predicted label, and plots the resulting count. She is surprised that there are many join results when there should not have been any, and complains that the count should be 0.

Fairness: A bank develops a model to approve/reject loan applications. Past loan approvals rejected women at higher rates and are used as training data. The deployed model ends up over-rejecting women, and a ML engineer *complains* that women should be approved much more. This complaint can be seen as a complaint over the difference of the output of two SQL queries, one that computes the acceptance rate for men and one for women. New applications don't have ground-truth labels to help debug the training data. However, a complaint driven debugger can combine complaints with execution traces and train/inference data to identify training data interventions that would resolve the over-rejection. These interventions could remove/edit individual records or groups of records (e.g., data labelled by a biased annotator).

2.4 Supported Query 2.0 Workflows and Complaints

Even if we restrict our attention to Query 2.0 workflows, all ML workflows remain effectively in scope. Given that SQL is Turing complete, any downstream workflow can be rewritten in SQL making it a Query 2.0 workflow. Similarly all ML models remain in scope as we have not imposed so far any restrictions on the ML models used. Under such a general scope it is impossible to outperform the black box approaches of Section 2.2.

Revisiting our discussion of Section 2.1, we want to use traces of the workflow execution to accelerate the evaluation of training data interventions. Thus we need to restrict our attention to relational operators and ML models for which collecting traces is both tractable and useful for workflow acceleration. In this section we will discuss which SQL queries are supported for the downstream SQL component of the Query 2.0 workflow, which ML models and preprocessing pipelines as well as the format of the supported complaints.

2.4.1 Supported Downstream SQL Queries

As we argued above we need to identify a class of SQL queries for which tracing is both efficient and sufficient for workflow acceleration. The key concern for tracing is that in order to be able to rerun the workflow based on the traced information alone the traces need to capture information about all the execution paths of a program for all potential modifications we want to run. The challenge is that the number of control flow paths can grow exponentially with code size so tracing all paths may be much harder than merely executing a program.

Section 3.2 is devoted to covering the necessary background in relational provenance, the area of database literature that concerns itself with tracing database queries and its applications. Specifically in Section 3.2.1 we study the case of tracing SPJA queries and present how it can be naturally and efficiently incorporated in traditional query execution [23]. The races captured can be then used for executing the downstream workflow as we explain in Section 3.2.2. More general query classes like nested queries introduce difficulties as tracing during query execution may not be possible as we

discuss in Section 3.2.2. Quantifying the sensitivity of these queries can also be more difficult to do in a principled way as we argue in Section A.4.

Another aspect that controls our ability to trace SQL query execution is the complexity of its inputs. For our use case the only inputs to the downstream relational workflow that we care about are the ML model predictions. All other inputs including relational database tables that the query uses are considered fixed for complaint debugging purposes. It would be ideal if we could identify a small set of potential inputs that we can then trace the downstream workflow on. This is indeed the case for ML classification models where each prediction of the serving data can be assigned to one of the model's classes. In Section 3.2.4 we present a query plan rewrite that can be used to trace the database queries for all the potential input classifications. In contrast things are more complicated for regression models that can have a infinite number of potential outputs. Semantically understanding how the model predictions are used in a query's WHERE clause is required to trace the query. We leave the case of regression models for future work.

This work thus focuses on Select-Project-Join-Aggregate (SPJA) SQL queries which have zero or more inner joins and embed a single classification ML model. The query can use the same model in multiple expressions. Specifically, we support SP, SPJ, SPJA queries, such as:

SELECT `agg(·),` $\cdots$ **FROM** $R_1, R_2 \cdots R_n$

WHERE C_1 **AND** $\cdots$ **AND** C_m

GROUP BY $G_1, G_2 \cdots G_k$

where agg can be COUNT, SUM, or AVG, and each C_i is either a filter condition or a join condition. Conjunctive and disjunctive predicates are supported as well. A model M can appear in the SELECTION, WHERE, or GROUP BY clause (Table 2.1).

- SELECTION: model prediction appears in an aggregation function, denoted by agg(M.predict). For example, if M estimates customer salary, then Q_1 returns the average estimated salary.
- WHERE: model prediction appears in a filter condition or a join condition. For example, if M predicts if a customer will churn or not, then Q_2 returns the number of customers that may churn.

Q_1 :	SELECT AVG(M.predict(R)) FROM R
Q_2 :	SELECT COUNT(*) FROM R WHERE M.predict(R)
Q_3 :	SELECT * FROM R_1 and R_2 WHERE M.predict(R_1) = M.predict(R_2)
Q_4 :	SELECT * FROM R_1 and R_2 WHERE M.predict(R_1+R_2)
Q_5 :	SELECT COUNT(*) FROM R GROUP BY M.predict(R)

Table 2.1: Query 2.0 examples. Model prediction can be embedded in an aggregation/projections (Q_1), filters (Q_2), join conditions (Q_3, Q_4), and group bys (Q_5).

If M extracts the user type, then Q_3 returns pairs of customers from two datasets that are the same user type (note that Q_3 is a SPJ query). Finally, if M estimates if two records are the same entity, then Q_4 finds pairs of records that are the same entity.

- GROUP BY: model prediction appears in the GROUP BY clause. For example, if M predicts the sentiment of a customer comment, then Q_5 returns the number of comments for each sentiment class (positive, neutral, or negative).

We note that the single ML model restriction is not related to tracing. Tracing of multiple model usages is just as easy if predictions come from multiple models instead of one. Identifying which model caused the downstream errors and error cascades across model pipelines make debugging more difficult in this case so it is left as promising future work.

Despite the restrictions imposed on the downstream workflow, the class of supported queries can still capture commonly used queries in business intelligence and dashboard applications as well as all the complaint driven training data debugging use cases of Section 2.3.2.

2.4.2 Supported ML Models

ML models are not making predictions directly on raw training data. Preprocessing steps are typically required, including complex transformations of the raw available data to a featurized training examples ready to be used by the ML model [58]. Debugging via interventions on the raw training data as compared to the featurized representations is not always equivalent. Preprocessing pipelines may allow interactions between raw training examples in cases like feature normalization, feature selection and data cleaning [58, 59, 60, 61, 18]. The problem now becomes more challenging

as changes to even a relatively small part of the raw data can lead to global changes of all the featurized examples. Clearly if the training data are allowed to change so drastically, avoiding model retraining in the worst case may be impossible. Indeed many approaches for tuning or debugging preprocessing pipelines either focus on models that do not require training like nearest neighbors [60] or need to tune the pipelines by training new models [59, 61].

Here we will follow the approach of DataScope [60] and allow per example training data interventions on the raw data ignoring the effect the intervention has on other training examples. For a large enough dataset, a deletion of a small number of examples may have limited effects on global training data statistics and thus the other training examples. At the same time, these training examples may have a large impact on the predictions of the model on similar serving examples. As a result the interventions can still have a large effect on the complaint even if they do not have large effect on the preprocessing pipeline. We empirically find that for relatively simple preprocessing pipelines our approach is effective. Extending the tracing analysis of downstream workflows to data preprocessing in order to capture its effects more accurately and efficiently is left for future work.

Regarding the ML model itself, we will focus on probabilistic classifiers with twice differentiable loss functions. This class of models includes logistic regression models as well as neural networks that produce probabilistic outputs e.g., using a softmax layer. This class of models is very broad and popular especially in application domains like natural language processing [16] and computer vision [14, 15] where neural networks typically outperform competing model classes.

Section 3.1 analyzes how the model's first and second derivatives can act as traces and help approximate the effect of deletions, updates and insertions of training examples to the ML model parameters and its predictions. Focusing on probabilistic classifiers will be helpful in Section 4.2.3 for quantifying the sensitivity of the downstream query to changes to the model's predictions.

Let T be the training set for model M and $\mathcal{D} = \{R_1, R_2, \cdots, R_n\}$ denotes a database containing queried relations. The trained model M will make predictions using data from $\mathcal{D}$. Given a downstream Query 2.0 query Q, we denote its output result over $\mathcal{D}$ by $Q(\mathcal{D}; M(T))$. If the context is clear, the notation is simplified as $Q(\mathcal{D}; T)$.

2.4.3 Supported Complaints

A user may have a *complaint* about the query output $Q(\mathcal{D};T)$. We consider two types: *value complaints* and *tuple complaints*. A value complaint lets the user ask why an output attribute value in $Q(\mathcal{D};T)$ is not equal to (larger than or smaller than) another value. In Figure 1.1, the user can specify why the two right-most low points in the visualization are not equal to (or larger than) the corresponding red points. A tuple complaint lets the user ask why an output tuple in $Q(\mathcal{D};T)$ appears in the output. This can be because a tuple should have been filtered by a predicate that compares with a model prediction, or because an aggregated group exists when it should not. For example, the user may ask why a pair of loyal customers are in the join output of Q_3 in Table 2.1.

Definition 1 presents a formal definition of complaints.

Definition 1 (Complaint)**.** *A complaint $c(t)$ is expressed as a boolean constraint over a tuple t in the output relation $Q(\mathcal{D};T)$. The complaint can take two forms. The first is a Value Complaint over an attribute value $t[a]$, where $op \in \{=, \leq, \geq\}$ and v may take any value in the attribute's domain (if $t[a]$ is discrete, then $\leq, \geq$ do not apply):*

$$c_{value}\big(t, Q(\mathcal{D};T)\big) = \begin{cases} True, & if\ t[a]\ op\ v \\ False, & otherwise \end{cases} \tag{2.1}$$

The second is a Tuple Complaint over the tuple t which states that t should not be in the output relation:

$$c_{tuple}\big(t, Q(\mathcal{D};T)\big) = \begin{cases} True, & if\ t \notin Q(\mathcal{D};T) \\ False, & otherwise \end{cases} \tag{2.2}$$

Multiple Complaints: The user may express multiple complaints against the result of $Q(\mathcal{D};T)$ or even against intermediate results of the query. In addition, if the user executed other queries using the same model $M(T)$, then complaints against those queries may also be used to identify training set errors. For ease of presentation, the text will focus on the single complaint case. However, the

proposed approaches support multiple complaints, and we evaluate them in the experiments.

2.5 Problem Statement

2.5.1 Unconstrained Training Data Interventions

Given a Query 2.0 query, there can be several ways to account for a user's complaint c by making changes to the training set T. For example, one might modify training examples in T, augment it with new training examples or even delete training examples. While all the above interventions make sense in different scenarios, for simplicity we will pose the problem definitions in terms of deletions. The problem definitions can be similarly extended to the other intervention types as well and we experiment with some of those in Chapter 5. We are ready to define the Query 2.0 debugging problem when we are allowed arbitrary deletions from the training set.

Problem 2 (Query 2.0 Debugging Problem)**.** *Given a training dataset T, a database $\mathcal{D}$ containing queried relations, a query Q, and a complaint c on tuple t, the goal is to identify the minimum set of training records such that if they were deleted, the complaint would be resolved:*

$$\begin{aligned} &\underset{\Delta \subseteq T}{minimize} && |\Delta| \\ &subject\ to && c\big(t, Q(\mathcal{D}; T \setminus \Delta)\big) = True \end{aligned}$$

The minimality requirement in Problem 2 is necessary because training examples superfluously deleted from the training set may be unrelated to the complaint. For example, for CompanyX's complaint in Figure 1.1 that the count of "Chrun" predictions is too low, a naive solution might have been to delete all "No-Chrun" training examples and force the model to always predict "Churn". Clearly such non minimal solutions are not helpful for debugging. In Chapters 4 and 5 we will develop computationally efficient approximate solutions for Problem 2.

2.5.2 Pattern Aware Training Data Interventions

As we will discuss in Chapter 6, Problem 2 may lead to suboptimal results because its solution is not constrained to detect systematic training data error patterns. This is especially important as errors may not be introduced one by one independently but in batches as a result of their shared cause. For example, a change in the checkout process of one of CompanyX's clients may lead to all the client's new training records to be mislabelled. Constraining the deletions to follow systematic patterns may lead to better quality suggestions. At the same time, actually returning the patterns found to the user instead of unstructured training example lists may be more actionable because users can use them to resolve the true source of the errors.

Let us assume that we want to enforce that the deletion interventions on the training data follows a pattern. For example, we may want to enforce that all of the tuples deleted satisfy a conjunction of few terms over the training data features. For each member of the family of allowed deletion interventions we associate a set of parameters ρ that uniquely identify it. For conjunctions this would be the binary vector over the available conjunction terms that indicates which terms are included. Let $N_\rho(T)$ be the subset of training examples that is deleted if we choose configuration ρ. The equivalent problem to Problem 2 using this pattern constraint becomes

Problem 3 (Query 2.0 Pattern Aware Debugging Problem)**.** *Given a training dataset T, a database $\mathcal{D}$ containing queried relations, a query Q, a complaint c on tuple t, the goal is to identify the parameters ρ of N that resolve the complaint with minimum deletions:*

$$\begin{aligned} &\underset{\rho}{\textit{minimize}} && |N_\rho(T)| \\ &\textit{subject to} && c\big(t, Q(\mathcal{D}; T \setminus N_\rho(T))\big) = True \end{aligned}$$

Problem 3 is a generalization of Problem 2 as the parametrization that chooses independently which training example should be deleted is a special case. Once again, a black box that tries to

solve this problem for all pattern families is required to evaluate all potential configurations ρ. In Chapter 6 we will develop a computationally efficient approach to approximately solve Problem 3 when the deletion pattern constraint is based on a differentiable classifier, e.g. a neural network, that decides if each training example should be deleted. Differentiable models can then be used as proxies for non differentiable pattern class like conjunctions.

2.5.3 View Maintenance for SPJA Queries under Deletion

Complaint driven debugging systems need to consider how $Q(\mathcal{D}; M(T))$ is affected indirectly by the changes in T through the changes to M. View maintenance in response to changes to the database of queried relations $\mathcal{D}$ is a well established problem in databases [30]. At first glance though, changes to the ML model modify user defined functions inside the query Q, a modification that traditional view maintenance techniques are not equipped to handle.

In Section 1.5.6 we discussed how this problem can be reduced to view maintenance under deletions. We observed that for each ML model prediction that may take place during query execution the output must be one of the classes of the classification problem that the model solves. Under this perspective, we can first collect all the inputs to model M that may appear during query execution. Then we can replicate each input collected for each potential class output and create a new relation V_M. Now given a new model we can simply delete all the rows of V_M that do not agree with the new model's predictions, resulting in one row for each model input. Query Q can then be rewritten to replace the calls to M with joins with the remaining tuples of V_M. Traditional view maintenance techniques are then directly applicable for this problem.

A complaint driven debugging system that supports Query 2.0 with arbitrary black box models may make use of this reduction to evaluate the effect of several training data interventions. As we shall see in Chapters 4 to 6, variations of these reductions can be used to efficiently support debugging for differentiable models as well. A question that arises is how can we execute view maintenance under deletions for SPJUA queries, SPJA that additionally allow unions, especially when we have batches of distinct deletions like in complaint driven debugging. Formally we want

to efficiently solve the following problem

Problem 4. *(Batch Counterfactual Intervention Evaluation) A deletion intervention I is a function that takes the database D and returns $I(D) \subseteq D$, the remaining database after the deletions are performed. Let D and Q be an apriori known database and SPJUA query respectively. Given interventions $\{I_1, \ldots, I_k\}$, evaluate $\{Q(I_1(D)), \ldots, Q(I_k(D))\}$.*

2.6 Complaint Driven Training Data Debugging Limitations and Risks

2.6.1 Limitations

Efficiently solving Problems 2 and 3 or even broader subsets of our complaint driven debugging vision would accelerate human in the loop debugging processes. Unfortunately, even if these problems were to be solved perfectly, applying the interventions suggested by the resulting systems should be done after vetting from a user. A system that ranks the available interventions has 3 limitations when it comes to automatically debugging complaints without further user assistance.

The first limitation is that a complaint driven debugging system cannot detect inaccurate complaints on its own. This can be problematic because an inaccurate complaint may lead to suggested interventions that actually make the resulting model worse. For example, in the case of Figure 1.1 someone may complain that the count of "Churn" predictions should be even lower instead of higher. If someone does not vet the suggested interventions before applying, the resulting model may start underpredicting the "Churn" class consistently on the downstream data. This limitation is important because complaint driven debugging seeks to enable users to complain on the workflow outputs directly without first identifying actual model mispredictions.

The second limitation is that a complaint driven debugging system cannot verify if suggested interventions fix true errors of the training data. An intervention being highly influential for the user's complaint is neither a sufficient nor a necessary condition for identifying errors in the training set. This is the case even if the complaint issued is accurate. This is because the system lacks a definition of what is a true error in the training data. This limitation is also important because as

discussed in Chapter 1, one of the motivations of complaint driven debugging is that users have difficulties defining error detection rules on the training data.

The third limitation is that a complaint driven debugging system cannot reason if a complaint, even a correct one, should be resolved in the first place. Even though a system may strive for minimal number of tuple interventions, these interventions may still have side effects. For example, resolving a complaint that the test accuracy of the model is below 100% may lead to overfitting with performance being suboptimal in other data distributions. Similarly, fixing a complaint that a classifier is unfair to a minority may degrade classifier accuracy. This limitation is important because even if we fix training true data errors using a complaint over one distribution of data, it is not necessary that the performance of the model will improve in another distribution of interest. In fact, the authors [62] identify cases where removing errors degrades test time accuracy.

Users can to some extent mitigate the above limitations by providing either multiple complaints or constraining the interventions to follow a pattern as in Problem 3. Both approaches limit the selection of potential interventions that address all complaints. The reduced selection makes manual vetting of interventions easier. Additional complaints also allow users to manage potential side effects of interventions. Of course the additional complaints and pattern constraints need to be carefully chosen to get the desired outcome. Complaint driven debugging effectively enables downstream users to save time during intervention vetting by defining accurate complaints and candidate patterns based on their domain expertise .

2.6.2 Risks

Without vetting of the suggested interventions, ML workflow users and developers may be exposed to significant risk. Users may end up degrading model performance by applying interventions suggested based on incorrect complaints. The ease of providing complaints without specifying model mispredictions increases this risk of incorrect complaints even further. Additionally, users may be eager to accept any intervention that resolves their complaint. This gives a false sense of security that the underlying problem is resolved when in reality the training data errors persist. In

fact with complaints being seemingly resolved, the true errors and their causes maybe harder to find because developers are not incentivized to look for them. Moreover, users may impose their own beliefs on the model by modifying its training set. Training data interventions based on complaints may end up reinforcing existing training data biases or even introduce entirely new ones.

The risk in each of the scenarios above critically depends on how much we allow training data to be modified in response to complaints. In the most extreme case we can allow complaint driven debugging to create a training set from scratch so that the downstream complaints are satisfied. While this has practical applications including dataset distillation [17], it is clear that such extreme dataset modifications carry a huge risk because there might be unforeseen consequences.

It is important to note that some amount of risk may be inevitable for approaches that modify the model's training data. To contextualize the risk of complaint driven training data debugging, we compare and contrast it with other approaches that modify the training data of a model.

Domain adaptation [20] aims to close the performance gap between training and testing by using information from the downstream domain of interest. Modifying the training data distribution is one of the many techniques used for domain adaptation. This is very similar to complaint driven training data debugging where downstream complaints can be used to modify the model's training distribution. Just like complaint driven debugging, unsupervised domain adaptation techniques can make use of downstream data even if they are not labelled. The core differentiation between the two is that complaint driven debugging accounts also about how the model used not just the downstream domain. Domain adaptation may face similar risks as complaint driven debugging.

Another common technique that modifies training data is data augmentation [19]. Data augmentation applies one or more transformations to the available training examples to generate new ones that can then be used for training. Unlike complaint driven debugging, data augmentation does not require downstream signals to choose its transformations. The risk in this case is that the transformations used do not lead to downstream improvements. In fact extreme augmentation policies, like adding too much noise to a training example's features, may in fact make the model worse. Thus choosing augmentation policies without taking the downstream distribution into account leads to an

additional risk factor not shared with complaint driven debugging.

Data cleaning [18] can also be used to modify a model's training data. Traditional data cleaning techniques detect and correct errors in a dataset to derive a clean dataset that can be used in place of the original erroneous dataset. An obvious risk is that the definition of an error is application dependent so incorrectly configured data cleaning approaches may introduce new errors in the effort of fixing new ones. Additionally, there is a risk that fixing even true training data errors can still degrade the model's downstream performance [63, 62]. Complaint driven debugging avoids this risk by being aware of both the downstream data distribution and the use of the model in the workflow.

Clearly none of the approaches outlined above are risk-free. To the extent that they are not properly configured, the changes that they propose may have unintended consequences downstream. While complaint driven training data debugging is not immune to this, the advantage of complaints is that they allow users to specify the intended downstream workflow behavior directly, giving users the opportunity to control these unintended consequences.

Chapter 3: Background

In Chapter 2 we specified that the focus of this thesis is building efficient systems for complaint driven training data debugging for Query 2.0 workflows that use differentiable probabilistic classifiers for ML inference. There are two factors that motivated the choice of both the downstream relational analytics and differentiable classifiers. The first factor, that we have already extensively discussed in Chapter 2, is the popularity of these components in real life ML workflow deployments. The second factor is that for these families of components we can develop efficient approximate techniques for complaint driven training data debugging.

In this chapter we will present existing work in ML optimization and relational provenance that will be the basis of our techniques in the following chapters. Specifically, we will start with influence functions, a tool from the ML optimization literature that will allow us to estimate the sensitivity of the a differentiable model to changes in its training set. Then we will discuss techniques from relational provenance that allow us to capture traces of its execution and use them to estimate the sensitivity of a query to changes in the input database. In Chapters 4 to 7 we will combine these techniques to estimate the end to end effect of training set interventions to the Query 2.0 output.

3.1 Influence Functions

In Chapter 2 we focused our attention on differentiable models. But even in the context of differentiable models, solving exactly Problem 2 remains intractable. If we have T training examples in our training set, a brute force solution to Problem 2 is to enumerate every possible set of deletions, and for each set, to retrain the model, update the query result, and evaluate the complaint. However, this needs to retrain up to $2^{|T|}$ models. The key is to reduce the number of models retrained.

A greedy approach would be to construct deletion sets based on individual training examples

whose removal is highly influential towards resolving the user's complaint. Influence functions [21, 22] provide a powerful way to estimate how the model parameters and functions thereof change by adding/deleting/updating a training point *without retraining the model*.

3.1.1 Influence Functions as Taylor Approximation

Given a differentiable model like a neural network, there is no closed form solution for the optimal parameters of the training loss functions and thus we cannot just incrementally update them in response to a deletion of a training example. The influence functions framework [21, 22] works around this limitation by constructing a surrogate loss function for which a closed form solution exists and then uses it to derive an approximate solution. In particular, quadratic functions $h(\theta)$ are useful surrogate functions because they have closed form solutions to compute the minimizers θ_h^*

$$h(\theta) = a\theta^2 + b\theta + \gamma \qquad \theta_h^* = -\frac{b}{2a}.$$

Modifications to h like adding a linear function $g(\theta) = r\theta + s$ can be easily handled as well with an incremental formula

$$\theta_{h+g}^* = \theta_h^* - \frac{g'(\theta_h^*)}{h''(\theta_h^*)} = \theta_h^* - \frac{r}{2a} = -\frac{b+r}{2a}. \tag{3.1}$$

Now we turn to reducing complex optimization problems to the easy cases above. Let $z_i = (x_i, y_i)$ be the i-th training example of T, composed of pair of a feature vector x_i and its corresponding label y_i. Let $\ell(\theta, z)$ return the loss of a training example z for a model with parameters θ. We define our loss function and its minimizer

$$L(\theta) = \sum_{i=1}^{|T|} \ell(\theta, z_i) \qquad \theta^* = \arg\min_{\theta} L(\theta).$$

Let us suppose that we want to estimate the effects of adding a training example $z = (x, y)$. We need to compute

$$\theta^*_{new} = \arg\min_{\theta}\{L(\theta) + \ell(\theta, z)\}.$$

The influence function framework reduces $L(\theta)$ and $\ell(\theta, z)$ to the quadratic surrogate function above by computing their Taylor series approximation

$$\begin{aligned} L(\theta) \approx & L(\theta^*) + \langle \nabla_\theta L(\theta^*), \theta - \theta^* \rangle + \frac{1}{2}(\theta - \theta^*)^T H_{\theta^*}(\theta - \theta^*) \\ \ell(\theta, z) \approx & \ell(\theta^*, z) + \langle \nabla_\theta \ell(\theta^*, z), \theta - \theta^* \rangle \end{aligned}$$

where the Hessian matrix H_{θ^*} is the second derivative of $L(\theta)$. Now applying the multivariate version of Equation (3.1) we get

$$\theta^*_{new} \approx \theta^* - H^{-1}_{\theta^*} \nabla_\theta \ell(\theta^*, z). \tag{3.2}$$

For complaint driven debugging we are not interested in θ^*_{new} itself but in functions computed over its output predictions (such as the output of Q that the complaint is specified over). If the function of interest is a differentiable function of the model parameters $q(\theta)$, we can approximate the effects of the training example addition using the first order Taylor approximation of q

$$q(\theta^*_{new}) \approx q(\theta^*) - \nabla_\theta q(\theta^*) H^{-1}_{\theta^*} \nabla_\theta \ell(\theta^*, z). \tag{3.3}$$

Approximating the effect of training example deletions is entirely symmetrical with the minus sign being replaced with a plus sign. Influence functions allow us to identify which training examples are the most influential to a target function q if they were to be added/removed. For an appropriate choice of q, this will be very useful in Chapters 4 to 6.

3.1.2 Influence Functions as Derivative Calculation

So far we have presented influence functions non rigorously from the lens of a Taylor approximation. To gain some intuition about what influence functions compute, we will seek to analyze them from a more rigorous perspective. For this more rigorous analysis we will assume that L is a strongly convex function of the model parameters. Adding a new training sample z with weight ϵ to the training loss leads to new set of optimal parameters θ^*_ϵ:

$$\theta^*_\epsilon = \arg\min_\theta \{L(\theta) + \epsilon \cdot \ell(\theta, z_i)\} \tag{3.4}$$

Of course there is no closed-form of θ^*_ϵ for $\epsilon = 1$, which would give us the new parameters after adding a training point without retraining. Influence functions quantify the case when $\epsilon \approx 0$. By the first order optimality condition, since θ^*_ϵ minimizes the objective of Equation (3.4)

$$\nabla_\theta L(\theta^*_\epsilon) + \epsilon \cdot \nabla_\theta \ell(z, \theta^*_\epsilon) = 0$$

Taking into account that θ^*_ϵ is a function of ϵ, we differentiate with respect to ϵ

$$\nabla^2_\theta L(\theta^*_\epsilon)\frac{d\theta^*_\epsilon}{d\epsilon} + \epsilon \cdot \nabla^2_\theta \ell(z, \theta^*_\epsilon)\frac{d\theta^*_\epsilon}{d\epsilon} + \nabla_\theta \ell(z, \theta^*_\epsilon) = 0$$

Substituting $H_{\theta^*} = \nabla^2_\theta L(\theta^*)$, where H_{θ^*} is the Hessian of the loss function $L(\theta)$, and simple algebra derives the following when $\epsilon = 0$:

$$\left.\frac{d\theta^*_\epsilon}{d\epsilon}\right|_{\epsilon=0} = -H^{-1}_{\theta^*}\nabla_\theta \ell(z, \theta^*).$$

Note ϵ is dropped from θ^*_ϵ because it is set to 0. Recent work has shown that using the derivative where $\epsilon = 0$ is a good approximation of the change in model parameters for (θ^*_{-1} or θ^*_1) [64, 65].

Similar to our non rigorous analysis, if we have a differentiable $q(\theta)$ of the model parameters we can estimate the effects of a training example addition directly. $q(\theta^*_\epsilon)$ is a differentiable function

of θ for $\epsilon \approx 0$. By the chain rule we have that

$$\left.\frac{dq(\theta^*_\epsilon)}{d\epsilon}\right|_{\epsilon=0} = -\nabla_\theta q(\theta^*)\, H^{-1}_{\theta^*} \nabla_\theta \ell(z, \theta^*). \tag{3.5}$$

Comparing and contrasting Equation (3.5) and Equation (3.3), we get the intuition that influence functions estimate the new value of q based on how sensitive it is to increasing/decreasing the weight of an example that is to be inserted/deleted.

3.1.3 Training Example Updates

Influence functions also support updating a training example z to z'. Such interventions can change the features, the label, or both. Modeling the update as deleting z and adding z' we get

$$\theta^*_{new} = \arg\min_\theta \{L(\theta) - \ell(\theta, z) + \ell(\theta, z')\}.$$

Following the analysis of Section 3.1.1 or Section 3.1.2 we get the same approximation

$$q(\theta^*_{new}) \approx q(\theta^*) - \nabla_\theta q(\theta^*) H^{-1}_{\theta^*} \left(\nabla_\theta \ell(\theta^*, z) - \nabla_\theta \ell(\theta^*, z')\right). \tag{3.6}$$

3.1.4 Inverse Hessian Vector Products

Naively evaluating Equation (3.3) is still infeasible. Given a model with d parameters ($\theta \in \mathbb{R}^d$), simply inverting the Hessian already takes $O(d^3)$ time and $O(d^2)$ space where d can be 10^6 or more for state of the art neural network architectures.

The good news is that even though we need to evaluate Equation (3.3) or its variants for deletions and updates once for every training example intervention, we can share the computation of $\nabla_\theta q(\theta^*) H^{-1}_{\theta^*}$ across all interventions. The problem thus boils down to solving a single linear equations system over the unknown vector $w \in \mathbb{R}^d$

$$H_{\theta^*} w = \nabla_\theta q(\theta^*). \tag{3.7}$$

Yet again, this linear system is still impractical to solve exactly. [66] observes that approximately solving Equation (3.7) amounts to find an approximate minimizer of the following function

$$\mu(w) = w^T H_{\theta^*} w - \langle \nabla_\theta q(\theta^*), w \rangle.$$

The benefit of this formulation is that $\nabla_w \mu(w) = H_{\theta^*} w - \nabla_\theta q(\theta^*)$ can be easily computed without materializing H_{θ^*}. Automatic differentiation frameworks (e.g. Tensorflow) can compute $H_{\theta^*} w$ given a vector w in $O(|T|d)$ time. The Conjugate Gradient (CG) algorithm [67] can then solve the problem exactly using d calls to $\nabla_w \mu(w)$. In practice [12] finds that a constant number of evaluations yields an empirically sufficient approximation.

3.1.5 Training Data Debugging Using Nested Optimization

Problems 2 and 3 are instances of nested optimization problems. Inside the top level problem of minimizing the number of deletions required to resolve the complaint, there is another optimization problem, namely the problem of deriving the optimal ML model parameters for the training set after the training example deletions. Problems 2 and 3 follow other existing work [13, 68, 69] to pose training data debugging as a nested optimization problem.

The common thread among the three existing approaches [13, 68, 69] is to view the training example weights as hyperparameters that we can tune in order to optimize a given objective. The objective is to minimize the validation loss of the trained model with [13] including some additional regularization terms. In summary, all these approaches seek to find the best choice of training example weights to minimize the validation loss of trained model. This is similar to Problem 2 if restrict the weights to be binary so that an example is either deleted or not deleted and replace validation loss in the objective with the user's complaint.

In this section we will present the formulation of the above nested optimization problem and how influence functions can be used to solve it. We start with the free variables of this nested optimization problem, namely the training example weights. Unlike in Section 3.1.2 where we

focused on one training example at a time, here each training example of T gets its own example weight $\epsilon_i \in [0, 1]$ and we concatenate those in a $|T|$ dimensional vector ϵ.

Given the vector of weights ϵ we can then form our weighted training loss function $L(\theta, \epsilon)$. Here we will use the the weighted sum of the per training example losses or divided by the sum of ϵ_i. This normalization does not change the optimal model parameters but helps maintain the scale of the loss function improving training stability. Existing work [13, 68] control the scale of the loss in different waysbut the idea remains the same. The formula of the loss function $L(\theta, \epsilon)$ is

$$L(\theta, \epsilon) = \frac{\sum_{i=1}^{|T|} \epsilon_i \ell(\theta, z_i)}{\sum_{i=1}^{|T|} \epsilon_i}.$$

Similar to our analysis in Section 3.1.2, we assume that $\ell(\cdot, z)$ is strongly convex with respect to the first argument θ for all training examples $z \in T$. We then have that for each vector ϵ there is a unique set of optimal parameters θ^*_ϵ computed as follows

$$\theta^*_\epsilon = \arg\min_\theta L(\theta, \epsilon). \tag{3.8}$$

Now we can use θ^*_ϵ in our top level objective q. We extend the function q to depend on ϵ both through θ^*_ϵ and on ϵ directly. This direct dependence is useful for additional regularization terms like in DUTI [13]. We can then write $q(\epsilon) = f(\theta_\epsilon, \epsilon)$ for a differentiable function f and solve

$$\min_{\epsilon_i \in [0,1]} q(\epsilon). \tag{3.9}$$

To reduce the problem to a classical optimization problem we simple want to be able to compute the derivatives of q with respect to each ϵ_i. The derivatives for the direct dependence of q on ϵ can be computed without complications. For the indirect dependence through θ^*_ϵ we need to make use of Equation (3.5). The derivative of q with respect to ϵ_i becomes

$$\frac{\partial q}{\partial \epsilon_i} = -\nabla_\theta f(\theta^*_\epsilon, \epsilon) \nabla^2_\theta L(\theta^*_\epsilon, \epsilon)^{-1} \nabla_\theta \ell(\theta^*_\epsilon, z_i) + \frac{\partial f}{\partial \epsilon_i} \tag{3.10}$$

where $\nabla_\theta f(\theta^*_\epsilon, \epsilon)$ is the derivative of f with respect to its first argument θ and $\frac{\partial f}{\partial \epsilon_i}$ is the corresponding derivative for the direct dependence on ϵ_i.

Given Equation (3.10), we are ready to attack the optimization problem of Equation (3.9). We can proceed in two steps. Given the set of current weights ϵ we compute the new optimal model parameters θ^*_ϵ. In practice, we compute an approximation of the parameters using standard optimization algorithms like stochastic gradient descent. We can then apply Equation (3.10) for all the weights ϵ_i and once again standard optimization algorithms to update the weights and project them to $[0, 1]$ to respect the constraints. In our evaluation of Equation (3.10) we can still use the same techniques as the ones we used in Section 3.1.4 for influence functions. Additionally, for the computation of θ^*_ϵ using Equation (3.8), we can use the previous θ^*_ϵ iterate as an initialization.

We will now discuss two variations to the vanilla nested optimization approach introduced by Meta-Weight-Net [69]. The first variation changes the free variables of the problem. In the formulation above ϵ_i are the free variables of the problem that can be updated independently. Meta-Weight-Net proposes for ϵ_i to be outputs of a new differentiable ML classification model themselves. This ML model has its own parameters, which we call ρ. This new ML model is a function w that can have its own inputs including the training example z_i and even the parameters θ. This turns ϵ_i from a free variable to an intermediate $\epsilon_i = w(\rho, z_i, \theta)$. The free variables now become the parameters of the new ML model ρ. The objective now becomes $q(\rho)$ and taking the derivatives with respect to ρ requires backpropagating through w for each $z_i \in T$.

The second variation has to do with computing derivatives without influence functions. The key idea is that the optimization iterates of an algorithm like gradient descent have a closed form by definition. Given an initialization θ_0, Meta-Weight-Net computes the one step gradient descent iterate $\theta_1(\epsilon)$ on an unnormalized version of $L(\theta, \epsilon)$. It then replaces θ^*_ϵ with θ_1 in q. After each update to ϵ, θ_1 becomes the new θ_0. This reduces the need for training in each iteration and also avoids the inverse Hessian vector product of influence functions.

3.2 Relational Provenance

In this section, we provide the necessary background in fine grained provenance capture, representation and its application to view maintenance under deletions. Relational provenance is the key information we extract from the downstream query of the Query 2.0 workflow to estimate its sensitivity to changes in the ML model predictions in Chapters 4 to 6. The discussion here will inform our approach for solving Problem 4 in Chapter 7.

3.2.1 Provenance Capture Metadata

Fine-grained provenance capture records for each query operator's output tuple which input tuples contributed to it. While there are many ways to capture and represent provenance [23, 70, 24, 25, 26, 27], we will follow the approach of Smoke [23] as Chapter 7 builds upon its representation. In the rest of the subsection, we will analyze each operator type individually.

Selection Selections filter input relations, rejecting input tuples that do not match a given condition. Each output tuple is contributed by exactly one input tuple. We can describe the mapping between input and output tuples in terms of sequential ids assigned to each tuple. Provenance capture for selections can be implemented using a Python-like syntax:

```
def select(table_a):
    P_σ = []
    for i, record_a in enumerate(table_a):
        if condition(record_a):
            P_σ.append(i)
            yield record_a
```

$P_\sigma[k]$ contains for the k-th output tuple, the index of the input tuple that contributed to its creation.

Join Here we focus on the case of the binary join between two relations. For joins each tuple is contributed by exactly one pair of input tuples, each coming from one of the input relations. Thus, we need to record two ids for each output tuple. A nested loop join with capture would look like:

```
def join(table_a, table_b):
    P⋈ = []
    for i, record_a in enumerate(table_a):
        for j, record_b in enumerate(table_b):
            if condition(record_a, record_b):
                P⋈.append((i,j))
                yield (record_a, record_b)
```

Here $P_{\bowtie}[k]$ contains for the k-th output tuple, the pair of indices from `table_a` and `table_b` that contributed to its creation.

Aggregation For aggregations without group by clauses, effectively all input tuples contribute to the singular output tuple so no recording of provenance is necessary. For group by aggregations, we need to record for each group the ids of the tuples that belong to it. It is equivalent and more convenient to store the inverse one to many mapping from input ids to output group ids. This is achieved by the following pseudocode:

```
def groupby(table_a):
    agg_data, groupno, Pγ = dict(), dict(), []
    for record_a in table_a:
        groupkey = groupby_key(record_a)
        if groupkey not in agg_data:
            agg_data[groupkey] = init_agg()
            groupno[groupkey] = len(groupno)
        agg_data[groupkey].update_agg(record_a)
        Pγ.append(groupno[groupkey])
```

Here $P_{\gamma}[k]$ contains for the k-th input tuple, the group id of the tuple it belongs to. `groupno` assigns to each group key a sequentially increasing number by order of appearance in `table_a`.

The above information records which tuples contributed to which groups. This is not sufficient though neither for complaint driven debugging nor for re-executing the aggregation under tuple deletion as required by Problem 4. Both applications need to know how each tuple contributed to

the aggregate. Beyond the aggregation function, we need to know aggregation values as well. The above pseudocode can be extended to materialize the columns that participated for each aggregate function. This information is necessary both for group by and simple aggregations.

Projection Projections under bag semantics do not require provenance capture. The i-th tuple of the output is contributed by the i-th input tuple. Projections with set semantics are equivalent to a group by on the projection columns and can be treated as above.

Union For union under bag semantics, each tuple in the output is contributed by exactly one tuple from the input tables. For each output tuple, we need to record the input tuple id and a corresponding table identifier. Here we will assume for simplicity that union merely concatenates tables instead of interleaving their tuples. Only the input table tuple counts are required in this case.

3.2.2 View Maintenance under Deletions

Fine-grained provenance is known to sufficient for view maintenance under deletions for SPJUA queries [28, 29]. This result is based on two observations. The first one is that a deletion intervention cannot create completely new tuples in the output. It can only change the output aggregate values or delete an output tuple. Thus, view maintenance under deletion only needs to figure out the new aggregation values and the deletion status of each output tuple. The second observation is that the provenance metadata above are sufficient to do just that. A selection output tuple is deleted when its contributing tuple is deleted. A join output tuple is deleted when either contributing tuple is deleted. A group by output tuple is deleted when all its contributing tuples are deleted. A group by or simple aggregation can be updated by evaluating it over the non-deleted contributing tuples.

The view maintenance approach sketched above does not need to be incremental like IVM in order to be faster than evaluating the view from scratch. When using provenance metadata, one does not need to re-execute filter and join conditions, or build hash tables for joins and group bys. No time is spent on inputs that have not contributed to an intermediate operator's output. While one can easily derive incremental versions of the above view maintenance algorithms for additional

wins for small interventions, we leave those for future work.

View maintenance using provenance metadata can be extended beyond SPJUA queries. The key difficulty here is that deletions can lead to new tuples in the query output. This means that simply tracing the query execution over the full input database is not sufficient because potential tuples might be missed. For example, prior work [29] extends view maintenance to queries with aggregate comparisons like the ones in `HAVING` clauses. We leave support for these operators both for complaint driven debugging and view maintenance for future work.

3.2.3 Provenance Circuits

Based on the discussion in Section 3.2.2, a natural representation of the provenance metadata for the whole query is a directed acyclic graph (DAG). Each node in the DAG corresponds to either a tuple or an aggregation value. The node's descendants in the DAG are the contributing nodes recorded during provenance capture. To enable view maintenance, each node is augmented with an arithmetic or logical operation. Join output tuples have an `AND` as the tuple remains only if both contributing tuples remain. Group by tuples are similarly annotated with an `OR` because the tuple remains if even one tuple in the group remains. Aggregation tuples are annotated with their aggregation function and the associated column values. Leaf nodes, which correspond to input database tuples, are labelled with 0 for deleted or 1 otherwise.

ProvSQL [24] evaluates the arithmetic circuits in a top down traversal using user defined types and functions and recursive SQL. Here is a simplified version supporting only count aggregations:

```
def evaluate(node):
    if node.operation == "COUNT":
        return sum([evaluate(child) for child in node.children])
    elif node.operation == "OR":
        return evaluate(node.left) or evaluate(node.right)
    elif node.operation == "AND":
        return evaluate(node.left) and evaluate(node.right)
    elif node.operation == "LEAF":
        return node.exists
```

The main advantages of this representation and execution are simplicity and versatility. The

provenance representation is independent of the query plan and how provenance was tracked. This makes the circuit evaluation code easier to write. This representation is also very general and it can be consumed by other provenance enabled applications like our complaint driven debugging use case. We will revisit the choice of provenance representation in Chapter 7.

3.2.4 Provenance Capture for Query 2.0

In our discussion so far we have discussed how provenance capture can help us with track tuple dependencies across a query as well as how to do view maintenance under input database tuple deletions. All these however are not directly helpful for tracking the dependencies of Query 2.0 workflow on the model's predictions. However, as we have discussed both in Chapter 1 and Section 2.5.3 we can reduce the view maintenance of Query 2.0 workflows under model updates to the tuple deletion version. Here we will describe this reduction in more detail.

The part of a query plan we need to attend to is the ML model calls. Without loss of generality, we can always break down a `WHERE` clause or aggregation involving ML model prediction into one or more projections that add a single ML prediction as an output column combined with a `WHERE` clause or aggregation involving no ML model predictions respectively. Having done this rewrite, we need to alter how the newly introduced projections behave under provenance capture.

Let us have a table A, which can be a database table or a plan intermediate, which uses column x as its ML model input to make a prediction creating a new intermediate table I. The projection operator that we need to modify under provenance capture is equivalent to the following SQL query

```
SELECT A.*, M.predict(A.x) FROM A
```

where M.predict outputs the corresponding classification for each row.

During provenance capture we are not interested in analysing the output for any single specific model but for all possible models. So it does not make sense to execute the operator based on the current model and propagate its result to the next operator. We need to consider all possible classifications that may be appear in I. Let C be a table with one column id having one row for each corresponding output class of M. We then create a view V_M as follows

```
SELECT DISTINCT A.x, C.id FROM A, C
```

containing the unique model inputs and classification outputs pairs.

We now replace the ML model projection SQL query over A with a new SQL query

```
SELECT A.*, V_M.id FROM A, V_M
WHERE V_M.x = A.x
```

whose output then replaces I in the query plan for the provenance capture needs. All of the operators that used to operate over the projection on A can now consider all potential classifications.

Observe that after this rewrite the query output that the provenance capture process produces does not correspond to any one model. But by deleting some tuples in the view V_M we can get the query output for any model of interest. Given a model M, we use the following statement

```
DELETE FROM V_M WHERE id != M.predict(x)
```

and then we can execute the rewritten query using the updated view to get the output. The process is the same if there are more ML model calls, thus more projections and more created views. Each view needs to be updated before running the rewritten query.

Of course, the point of this rewrite is not just to generate an equivalent way to execute the same Query 2.0. Given the metadata of the initial provenance capture run we can execute this view maintenance under deletions on V_M more efficiently. Complaint driven debugging can also use this rewrite to quantify the sensitivity of the query output to model changes.

Chapter 4: Training Data Debugging for Query 2.0

Revisiting the example in Chapter 1 , CompanyX tracks users on e-commerce websites and scrapes the pages for data to estimate user retention. They regularly retrain their model M_θ. However, systematic errors, such as changing the name of a product category or adding a new check-out step, can cause M_θ to suddenly underestimate user churn likelihoods. Customers will see a surprising cohort size drop in the monitoring chart (Figure 1.1) and complain. Despite assertions and error checking in their workflow systems, CompanyX engineers still spend considerable time to find the training errors. Ideally, a debugging system can help them quickly identify examples of the training records that were responsible for the customer complaint.

Query debugging is not new, and there are existing explanation and debugging approaches for relational queries or machine learning models. On the one hand, SQL explanation [8, 9, 71, 72] uses user complaints of query results to identify queried records or predicates, and can fix the complaint through intervention (deleting those records). However in the context of Query 2.0, these methods would only identify errors in the queried data (e.g., `U, L` in Figure 1.1), rather than in M_θ's training data. On the other hand, case-based ML explanation algorithms [73, 13] use labeled mispredictions to identify training points that, if removed, would fix the mispredictions. This is akin to specifying complaints over the intermediate outputs of the query (specifically, the outputs of the M_θ`.predict()` predicate). Unfortunately, finding and labeling the mispredictions can take considerable effort. Further, users such as CompanyX's customers only see the final chart.

To this end, we present Rain, a system to facilitate *complaint driven data debugging for Query 2.0*. Given that the query and the queried data are correct, Rain detects errors in the training data. Users simply report errors in intermediate or final query results as complaints, which specify whether an output value should be higher, lower, or equal to an alternative value, or if an output tuple should not exist. Rain returns a subset of training records that, if the models are retrained

without those records, would most likely address the complaints. This problem combines aspects of integer programming, bi-level optimization, and combinatorial search over all subsets of training records—each is challenging in isolation, and together poses novel challenges faced neither by SQL nor ML explanation approaches.

To address these challenges, we describe and evaluate two techniques that bring together SQL and ML explanation techniques. Both iteratively identify training records that, if removed, are most likely to fix user complaints. TwoStep uses a two-step approach: it models the output of model inference as a view, and uses an existing SQL explanation method to identify records in the view that are responsible for user complaints. Those records are marked as mispredictions and then used as input to a case-based ML explanation algorithm. This method works well when SQL explanation can correctly identify the model mispredictions (or the user directly labels them). However, it can work poorly when there are many satisfying solutions for the complaints in the SQL explanation step; we call this complaint **ambiguity**, and provide theoretical intuition and empirical evidence that it causes TwoStep to incorrectly identify erroneous training points.

To address these limitations, the Holistic approach models the entire pipeline—the query plan, model training, and user complaints—as a single optimization problem. This directly measures the effect of each training record on the user complaints, without needing to guess mispredictions correctly. We also provide theoretical intuition for when and why Holistic should be more effective than existing approaches that do not account for SQL queries nor user complaints. To summarize, our contributions include:

- The design and implementation of Rain, a solution framework that integrates elements of existing SQL and case-based ML explanation algorithms.
- TwoStep, which sequentially combines existing ILP-based SQL explanation approaches and ML influence analysis techniques. Our theoretical analysis shows that TwoStep is sensitive to the ILP's solution space, and we empirically validate this in the experiments.
- Holistic, which combine user complaints, the query, and model training in a single problem that avoids the ambiguity issues in TwoStep.

- An extensive evaluation of Rain against existing explanation baselines. We use a range of datasets containing relational, textual, and image data. We validate our theoretical analyses: TwoStep is susceptible to performance degradation when ambiguity is high, and that approaches that do not use complaints are misled when there are considerable systematic training set errors. We find that Holistic's accuracy dominates the other approaches—including settings where alternative approaches cannot find *any* erroneous training records.

4.1 Main Ideas of Our Approaches

Unfortunately, influence functions cannot be directly applied to solve the Query 2.0 Debugging Problem since we need to calculate the impact of deletions of training points on a Query 2.0 query output, and SQL queries are not naturally differentiable.

We use two novel ideas to address this challenge. TwoStep first finds a minimal set of model predictions flips that resolve the complaint. Then it estimates the effect that training example deletions have on these predictions. Holistic encodes a Query 2.0 query (both SQL and model parts) into a single differentiable function and then directly calculates the impact of deletions of training points on the query result. We developed Rain, a Query 2.0 debugging system that implements TwoStep and Holistic approaches. The next section will describe the system details.

4.2 The Rain System

This section describes the overall architecture of Rain, which uses either TwoStep (Section 4.2.2) or Holistic (Section 4.2.3) to solve the Query 2.0 Debugging Problem.

4.2.1 Architecture Overview

Rain (Figure 4.1) consists of a query processor that supports training machine learning models (step ⓪), performing model inference (step ①) and executing SQL queries based on the model outputs (step ②). The user examines the output or intermediate result set of a query Q, and specifies a set of complaints C (step ③ complaints that the result should be 2 instead of 1). The

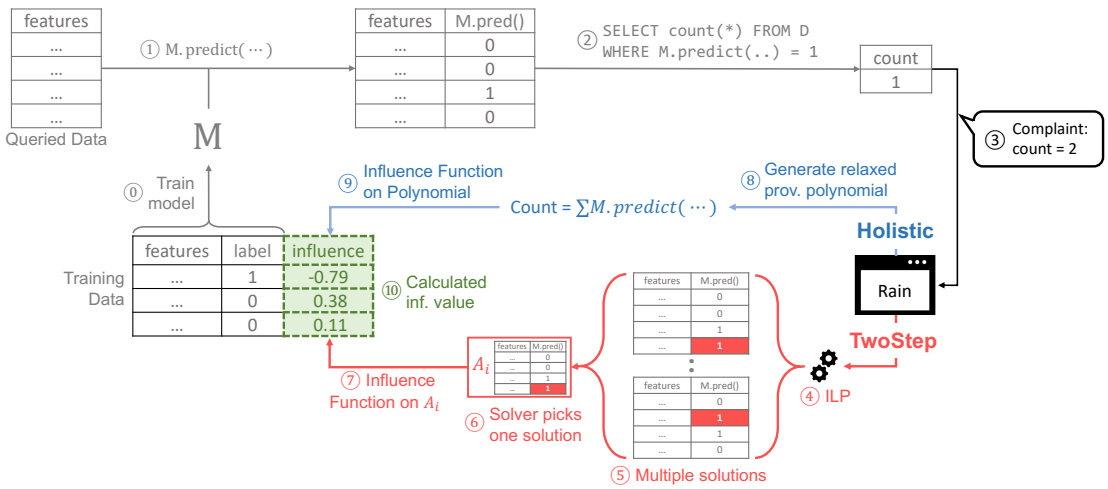


Figure 4.1: Rain architecture.

optimizer uses a simple heuristic to choose between the two methods. As we will discuss in Section 4.2.2, TwoStep is preferable when there is a unique way to fix the querying set predictions that resolves C. For all other cases, Holistic is used.

TwoStep turns the complaints into a discrete ILP problem and uses an off-the-shelf solver (step ④) to label a subset of the model inferences with their (estimated) correct predictions. If multiple satisfying solutions exist (step ⑤), solvers will opaquely output one of the solutions dependent on the specific implementation (step ⑥). The solution is encoded as an influence function to estimate how each training record "fixes" the mispredictions (step ⑦). Holistic encodes the query and model training as a single relaxed provenance polynomial (step ⑧) that serves as an influence function to estimate how much each training record "fixes" the complaints (step ⑨). For both approaches, Rain finds the top k training records by influence (step ⑩).

Both approaches will first rerun Q (step ②) in a "debug mode" to generate fine-grained lineage metadata that encodes the optimization problem. Rain then runs a train-rank-fix scheme, where each iteration (re)trains the model (step ⓪), reruns the query (step ①-②), finds and deletes the top training records by influence (step ④-⑩), and repeats. The result is a sequence of training records that comprise the output explanation Δ. Assuming each iteration selects the top-k training records, then Rain executes $\frac{|\Delta|}{k}$ iterations.

4.2.2 TwoStep Approach

Query 2.0 plans consist of relational pipelines and model inference. Since there are existing solutions to address each in isolation, the naive approach combines them into a TwoStep solution. This section describes this approach, and provides intuition on its strengths and limitations. We now describe the SQL and Influence Analysis steps of TwoStep.

SQL Step: TwoStep uses the approach of Section 3.2.4 to replace each model inference expression, such as M_θ`.predict(U.*)` in Figure 1.1, with a materialized *prediction view* containing the input's primary key and the prediction result. Let V_M be the prediction view for model M, and D_T be the database containing the views. $Q(D;T)$ can be rewritten as $Q_m(D_T)$ to instead refer to the model views rather than perform model inference directly. For instance, the query in Figure 1.1 would be rewritten in terms of the view V_M as follows:

```
SELECT COUNT(*) FROM Users U, Logins L, V_M
 WHERE U.ID = L.ID AND U.ID = V_M.ID
       AND L.active_last_month
       AND V_M.prediction = "Churn"
```

We build on Tiresias [9], which takes as input a set of complaints, along with attributes in queried relations that can be changed to fix those complaints. It translates the complaints and query into an ILP, where marked attributes are replaced with free variables that the solver (e.g., Gurobi [74], CPLEX [75]) assigns. We mark the predicted attribute in the prediction views, and the objective minimizes the number of prediction changes.

The translation to an ILP relies on database provenance concepts. Each potential output of Q_m defines a function over the prediction view that evaluates to 1 if the tuple exists in the query output for the given prediction view or 0 if not. In addition, each aggregation output value of Q_m defines a function over the prediction view that returns the aggregation value. Prior provenance work [28, 29] shows how to translate the supported queries into symbolic representations of these functions also

known as provenance polynomials, which Tiresias encodes as ILP constraints. We illustrate the reduction for the example in Figure 1.1:

Example 1. *Let the query plan for Figure 1.1 first filter and join L with U, and then apply the churn filter before the aggregation. Let K be the number of the remaining rows after the join and filter on L, and* $\boldsymbol{r} \in \{0,1\}^K$ *be the binary model predictions over these rows.* $r_i = 1$ *means the user is predicted to churn, and the query result is* $\sum_{i=1}^{K} r_i$*. If the user complains that the query output should be X, then the generated ILP is as follows, where* $t_i \neq r_i$ *means that record i should be labeled as a misprediction:*

$$\begin{aligned} &\underset{\boldsymbol{t}\in\{0,1\}^K}{\textit{minimize}} && \sum_{i=1}^{K} |t_i - r_i| \\ &\textit{subject to} && \sum_{i=1}^{K} t_i = X \end{aligned} \tag{4.1}$$

Influence Analysis Step: The previous step assigns each record x_i a (possibly "corrected") label t_i: $\{x_i, t_i\}_{i=1}^{K}$. Let $p_{t_i}(x_i, \theta)$ be the probability that model M_θ predicts x_i to be class t_i, where θ is the vector of the ML model parameters. We construct function $q(\theta) = -\sum_{i=1}^{K} p_{t_i}(x_i, \theta)$ that is used as input to an influence analysis framework [12, 76, 13, 73]. These frameworks return a ranking of training points that, if removed, are most likely to change the predictions of x_i to t_i; this indirectly addresses the user's complaint.

For example, suppose we use the influence analysis framework of [12]. TwoStep uses Equation (3.5) to score every training record. The initially trained model has optimal parameters θ^*. The training loss Hessian H_{θ^*} and the training loss gradient of each training record $\nabla_\theta \ell(z, \theta^*)$ are evaluated at θ^*. The function q constructed by TwoStep is then substituted to encode the user's complaint. Training records with large positive scores imply that their removal would decrease q the most, implicitly addressing the complaint. TwoStep ranks these records at the top.

In most settings, the number of records not marked as a misprediction ($t_i = r_i$) is considerably larger than those marked as mispredictions ($t_i \neq r_i$). In our experiments, we only encode the marked mispredictions into $q(\cdot)$ in Equation (3.5), and empirically find that they result in comparable

rankings as when encoding all records.

Although TwoStep is simple, there are several limitations due to the nature of the ILP formulation of the SQL step. First, the ILP problem can be *ambiguous* and is not guaranteed to identify the correct solution. Second, TwoStep depends on the user submitting a correct complaint.

Ambiguity: The generated ILP may not always have a unique solution. For example, Figure 4.1 shows how the ILP of a complaint on a `COUNT` aggregation can have multiple solutions $A_1, A_2, \ldots, A_n$ (step (4)). We call such complaints *ambiguous*. Picking a solution A_i in step (5) that makes incorrect prediction fixes can negatively affect the influence step (6). Intuitively, a complaint with more ILP solutions should lead to worse rankings because, among all solutions that minimize the ILP problem, only a few minimize Problem 2. We identify two sources of ambiguity.

The first are aggregations. In Figure 4.1, flipping any single prediction 0 is a valid and minimal solution, but only one solution is correct. The same argument extends to all the aggregates supported by Rain as all of them are symmetric with respect to their inputs.

The second are join and selection predicates. Consider a join $A \bowtie_{A.a=B.b} B$, where $A.a$ and $B.b$ are both estimated by a model M. If the user specifies that a join result should not exist, then one has to choose between changing $A.a$ or $B.b$. More generally, selection predicates that involve two or more model predictions can also be ambiguous.

Section A.1 lists specific settings where ambiguity provably causes TwoStep to rank the true training errors arbitrarily low, thus forbidding us sampling multiple solutions from ILP to avoid bad results for the whole problem. Unfortunately, formally quantifying its effect in the general case is challenging because partially correct solutions A_i can still yield high quality rankings depending on the model and the corrupted training records.

Our experiments vary the level of ambiguity and empirically suggest that TwoStep performs better when the number of solutions of the SQL step is smaller. This agrees with our theoretical results in Section A.2 where even an unambiguous complaint on a single prediction can rank training errors at the top whereas loss based baselines rank them at the bottom.

Complaint Sensitivity: The second limitation is due to the discrete formulation of the ILP: identifying correct assignments depends on the correctness of the complaint. For example, if the user selected a slightly incorrect X in Equation (4.1), the satisfying assignments can be considerably different than the true mispredictions. Unfortunately, if the user finds surprising points in a visualization, she may have an intuition that the point should be higher or lower, but is unlikely to know its exact correct value. We see this sensitivity in our experiments.

4.2.3 Holistic Approach

In this section, we present the Holistic approach that addresses many of the limitations of TwoStep. The key insight is to connect training records with the user complaints by modeling the query probabilistically and interpreting the confidence of model predictions as probabilities. This lets us leverage prior work in probabilistic databases [77, 78] to represent Query 2.0 statements as a differentiable function that is amendable to influence analysis. Note that although provenance and influence analysis alone build on prior work, integrating them for the purpose of complaint driven training data debugging is the key novelty.

4.2.4 Relaxation Approach

As noted above, the symbolic SQL query representations are not naturally differentiable due to discrete inputs (values in the prediction views), and thus are incompatible with an influence analysis framework. In contrast to TwoStep, Holistic leverages techniques from probabilistic databases [78, 77] to relax these functions of discrete inputs into continuous variable functions.

Revisiting Equation (4.1), Holistic substitutes the count of churn predictions with the expectation of the count. For example, let $r_i(\theta)$ be the boolean churn prediction and $p_i(\theta)$ be the churn probability assigned by M_θ, Holistic substitutes:

$$\sum_{i=1}^{K} r_i(\theta) \rightarrow \sum_{i=1}^{K} p_i(\theta).$$

Unfortunately, expectations of provenance polynomials are not always straightforward to compute. Even calculating the expectation of a k-DNF formula is #P-complete [78]. To sidestep the computational difficulty of exact probabilistic relaxation, we propose a tractable alternative under the simplifying assumption that variables and sub-expressions are independent. We first replace discrete predictions in the provenance polynomial with their corresponding probabilities (similar to $r_i(\theta) \rightarrow p_i(\theta)$ above). We then replace boolean operators (`AND`, `OR`, `NOT`) with continuous alternatives

$$\begin{aligned} x \ \texttt{AND} \quad y \quad &\rightarrow x \cdot y \\ x \ \texttt{OR} \quad y \quad &\rightarrow 1 - (1 - x) \cdot (1 - y) \\ \texttt{NOT} \quad x \quad &\rightarrow (1 - x). \end{aligned}$$

Observe that the first two formulas above can be mapped to the probability formulas for the `AND` and `OR` of two independent random variables. Our relaxation applies this rule even when x and y are complex expressions that share random variables and thus may not be independent. When each variable appears only once in the provenance polynomial as discussed in [78], our approach yields the actual expectation.

Our relaxation focuses on tractability. Alternative differentiable relaxations of logical constraints based on probabilistic interpretations are axiomatically principled [79] albeit generally intractable. Comparing relaxation approaches is a promising direction for future work.

4.2.5 Translating Complaints to Influence Functions

To adapt the above into an influence analysis framework, we translate user complaints over relaxed provenance polynomials into a differentiable function $q(\theta)$ that we want to minimize. We will first assume one equality complaint $t_i[a] = X$ on a single value, and then relax these assumptions to support multiple, more general complaints.

Let $r_q(\theta)$ be the relaxed provenance polynomial for $t_i[a]$. We adapt it to the complaint by defining $q(\theta) = \big(r_q(\theta) - X\big)^2$. Minimizing $q(\theta)$ forces $t_i[a]$ to be close to X. Akin to Section 4.2.2, this function is now compatible with modern influence analysis frameworks [12, 76, 13].

We support tuple complaints by taking the relaxed tuple polynomial $r_q(\theta)$ for tuple t, and

Q_1	SELECT COUNT(*) FROM DBLP WHERE predict(*)='match'
Q_2	SELECT COUNT(*) FROM Enron WHERE predict(*)='spam' AND text LIKE '%**word**%'
Q_3	SELECT * FROM MNIST L, MNIST R WHERE predict(L) = predict(R)
Q_4	SELECT COUNT(*) FROM MNIST L, MNIST R WHERE predict(L) = predict(R)
Q_5	SELECT COUNT(*) FROM MNIST WHERE predict(*)=1
Q_6	SELECT AVG(predict(*)) FROM Adult GROUP BY gender
Q_7	SELECT AVG(predict(*)) FROM Adult GROUP BY agedecade

Table 4.1: Summary of queries used in the experiments. $predict(\cdot)$ is shorthand for $M_\theta.predict(\cdot)$.

defining $q(\theta) = (r_q(\theta) - 0)^2$. Inequality value complaints like $t[a] >= X$ are supported within the train-rank-fix scheme of the system. While the complaint is false, we model it as an equality complaint; iterations where the inequality is satisfied can ignore the complaint until it is once again violated. Finally, to support multiple complaints, we sum their q functions.

4.3 Experiments

Our experiments seek to understand the trade-offs of Rain as compared to existing SQL-only and ML-only explanation methods, and to understand when complaint driven data debugging can be effective. We then study how ambiguity, increasing the number of complaints, and errors in the complaints affect Rain and the baselines. The majority of our experiments are performed with linear models but we also experiment with convolutional neural networks.

4.3.1 Experimental Settings

We now describe the experimental settings. Across our experiments we use a range of SPJA queries that are summarized in Table 4.1.

Approaches

We evaluate 3 baselines and the two approaches. Each approach returns a ranked list of training points using a train-rank-fix scheme. Each iteration trains the model, and then selects and removes the top-10 ranked training records. Thus, removed records affect future iterations and potentially

improves the results.

For the baselines, **Loss** ranks from the highest training loss to lowest, it is the most convenient approach because it is naturally computed during training; **InfLoss** uses the model-based influence analysis [12] to rank a training point higher if removing it increases its individual training loss the most. This is the state of the art approach of using the influence analysis framework for training set debugging without requiring additional labels. We compare these against **TwoStep** (Section 4.2.2) and **Holistic** (Section 4.2.3).

Datasets

We use record, text, and image-based datasets. In each experiment, we will systematically corrupt the labels of $\mathcal{K}$ training records.

DBLP-GOOG publication entity resolution dataset used in [80]. Each publication entry contains four attributes: title, author list, venue, and year. It contains two bibliographical sources—DBLP and Google Scholar—and the logistic regression model classifies a pair of DBLP, Scholar entries as same or not. We represent each pair using 17 features from [81]. The dataset is split in a training and querying set and a logistic regression model is trained.

ADULT income dataset [82], also known as the "Census Income" dataset. The task of this dataset is to predict based on census data whether a person makes more than 50K$ per year. Following the code of the author's of [83], we take three features of the dataset, namely age, education and gender and turn them in 18 binary variables. This process creates a lot of training examples with identical features (but not necessarily identical labels). Creating large groups of training examples with identical features is a necessary preprocessing step for many approaches of countering bias in learning [84] which complicates training bug detection as we shall see in Section 4.3.5.

ENRON spam classification dataset [85]. It contains 5172 emails received and sent by ENRON employees. The logistic regression model classifies each email as spam or not spam. Each email is

represented as a bag of words.

MNIST digits recognition dataset [86] contains 70000 hand-written images of 0-9 digits, each consisting of a 28×28 grid of pixels. The task is given an input image to output the digit depicted. We will experiment on this dataset using both logistic regression and neural architectures trained on 10000 training examples.

Training Errors:

Our experiments generate systematic training set errors by corrupting training labels. To do so, we choose records that match a predicate, and change the labels for a subset of the matching records. For example, for some of the MNIST image experiments, we select images of the digit 1, and change varying subsets of those images to be labeled 7. We describe the predicate and subset size in the corresponding experiments.

Complaints:

Most of our experiments specify equality value complaints for outputs of aggregation queries, tuple deletion complaints for outputs of join and non-aggregation queries. The complaints are generated from the ground truth. In Section 4.3.6, we execute two queries on the same query dataset submitting complaints for both queries; we also simulate misspecified equality value complaints that overestimate or underestimate the correct value, or where the value is completely incorrect.

Metrics:

We report recall r_k as the percentage of correctly identified training records in the top-k returned records, where $k \in [0, \mathcal{K}]$ increases to the number of actual corruptions $\mathcal{K}$. Unlike ML model evaluation, we note that for a given k, precision can be derived from recall.

Comparing curves across experiments can be challenging, thus we take inspiration from the area under the curve measure for precision-recall curves (AUC_{PR}) to introduce an area under the curve

measure for our corruption-recall curves. We call it AUC_{CR}, and compute it as the normalized average of the recalls across all k values: $AUC = \frac{2}{\mathcal{K}} \sum_{k=1}^{\mathcal{K}} r_k$ where r_k are the recall percentages. We also report running time when appropriate.

Implementation:

All our experiments are implemented in Tensorflow [87] and run on a google cloud n1-highmem-32 machine (32 vCPU, 208GB memory) with 4 NVIDIA V100 GPU. All models are implemented in Keras, and trained using the L-BFGS algorithm in Tensorflow. As noted in Section 3.1.4, we use the Conjugate Gradient algorithm to efficiently calculate $\nabla_\theta q(\theta^*) H_{\theta^*}^{-1}$.

4.3.2 Baseline Comparison: SPA Queries

We first evaluate the efficacy of complaint-based methods as compared to the baselines for detecting systematic errors in training records. We use a `COUNT(*)` query, and a single value complaint with the correct equality value. We first report detailed results for systematic corruptions of the DBLP dataset, where we flip a percentage of the `match` training labels to be `notmatch`. The percentage varies from 30% to 70% of the `match` training records, affecting 7% to 17% of the training labels accordingly. We run Q_1 from Table 4.1, and complain that the count is incorrect.

Figure 4.2 shows the recall curves for low (30%), medium (50%), and high (70%) corruption rates, where the grey line is a reference for perfect recall. Both loss-based approaches (Loss, InfLoss) degrade substantially as the corruption rate increases because the model begins to overfit to the training corruptions instead. This is corroborated by Figure 4.3. There we observe the F1 score of the model, the geometric mean of the model precision and recall, on the querying set as the corruption rate increases. For small corruption rates, the model treats the few corruptions as outliers and it does not fit them leading to robust performance. However, this changes for corruption rates larger than 50% where performance starts to drop drastically indicating that the model has started fitting to the corrupted data. TwoStep initially performs poorly, but improves as the systematic errors dominate the training set (70%) and reduce the complaint ambiguity. In contrast, Holistic

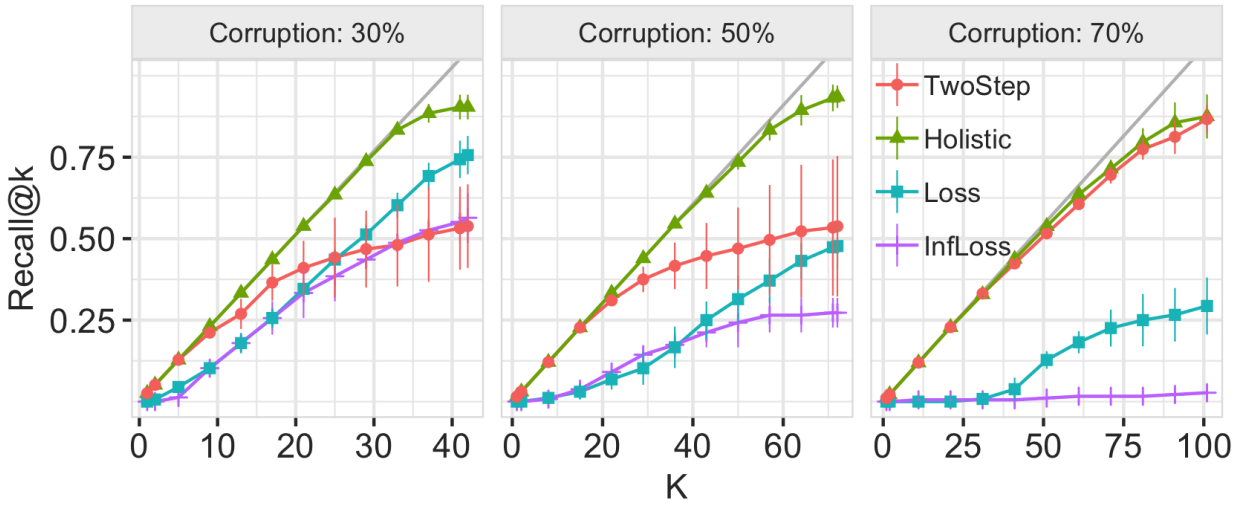


Figure 4.2: Recall curves when varying corruption rate for DBLP (grey line is perfect recall). Loss-based approaches perform poorly as corruption rate increases, while TwoStep improves at very high corruption rates (70%). Holistic dominates the other approaches.

is nearly perfect, and is robust to the different corruption rates. For reference, the AUC_{CR} of the approaches for medium corruption are shown as the first row in Table 4.2.

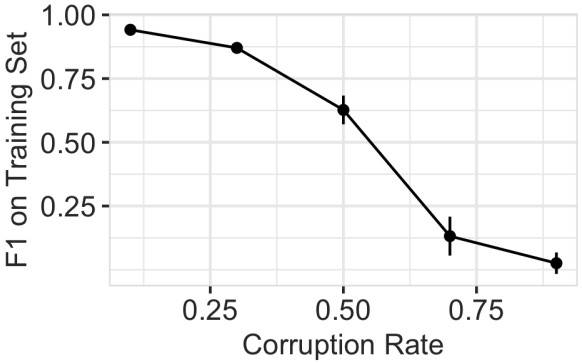


Figure 4.3: F1 vs corruption rate on DBLP

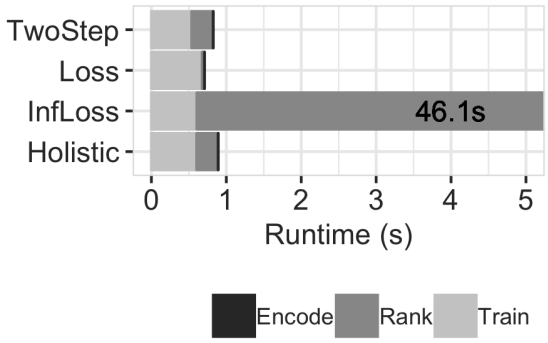


Figure 4.4: Per-iteration runtime on DBLP, 50% corruption. InfLoss takes 46.1s.

Figure 4.4 shows the runtime for each train-rank-fix iteration. We report three values, based on the terms in Equation (3.5). Train refers to model retraining to compute the model parameters θ^*; Encode refers to the cost of computing the influence function $-\nabla_\theta q(\theta^*)$; Rank refers to evaluating $\nabla_\theta q(\theta^*) H_{\theta^*}^{-1}$, which is dominated by calculating the Hessian vector products required by the Conjugate Gradient algorithm. Loss is the fastest because it simply uses the training loss and avoids

costly influence estimation; InfLoss has similar or worse recall curves than Loss, but is by far the slowest because it computes a unique influence function for each training record. Holistic and TwoStep are comparable, and dominated by the ranking cost.

We next evaluate the ENRON dataset using Q_2, where the search word in the LIKE predicate is either 'http' or 'deal'. The corruptions simulate rule-based labeling functions. For the 'http' query, we label all training emails containing 'http' as spam (13% of emails, of which 76% already labeled spam). The label corruption method is similar for the 'deal' query (18% of emails, 2.7% labeled spam). Table 4.2 summarizes the results: InfLoss, Loss and TwoStep perform poorly. It is worth pointing out that InfLoss takes 2 days to produce the results. Holistic performs much better for 'deal' because 17.5% more training labels were flipped, in contrast to only 3.14% for 'http'.

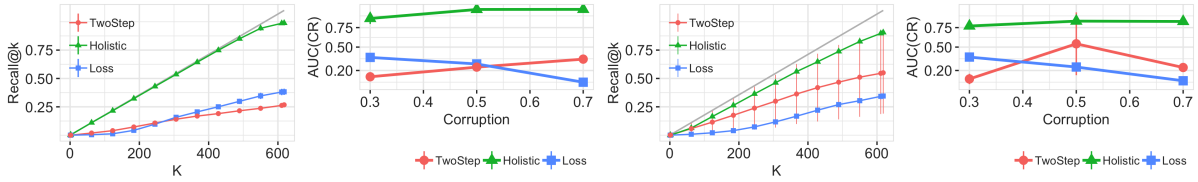
Dataset		InfLoss	Loss	TwoStep	Holistic
DBLP		0.30	0.35	0.71	**0.99**
ENRON	'%http%'	0.05	0.02	0.04	**0.12**
ENRON	'%deal%'	0.17	0.02	0.07	**0.40**

Table 4.2: AUC for DBLP with medium corruption, and ENRON with different search words.

Takeaways: Loss-based approaches are sensitive to the number of systematic errors in the training set—at large corruption rates, the model can overfit to the errors and lead to poor debugging quality. In contrast, complaints help ensure training records are ranked according to their effects on the complaints. We find that InfLoss takes over 40s per iteration, yet performs poorly under systematic errors. For these reasons, we do not evaluate InfLoss in subsequent experiments, but keep Loss to serve as a comparison point.

4.3.3 Baseline Comparison: SPJA Queries

This section uses the MNIST dataset to evaluate complaint-based debugging against the baselines for SPJA queries containing joins. The first two experiments join two image subsets that do not overlap in their digits, and thus expect no results of the join operation. We introduce corruptions by flipping a random subset of digit 1 images to be labeled 7 instead. We corrupt 30% (low), 50% (medium), and 70% (high) of the labels, impacting 3%, 5% and 7% of the total training labels



(a) Recall for point complaints (50% corruption).

(b) AUC_{CR} for point complaints.

(c) Recall for COUNT complaint (50% corruption).

(d) AUC_{CR} for COUNT complaint.

Figure 4.5: MNIST complaints on individual join rows (a-b), or COUNT of join results (c-d).

accordingly. We chose MNIST to make the problem more ambiguous: the model is a 10-digit classifier, thus there are 10 ways (1 = 1, 2 = 2, e.t.c.) to incorrectly satisfy the join condition, but 90 ways to incorrectly fix it (all other label combinations). We thus expect TwoStep to perform poorly due to a large number of satisfying, but incorrect, ILP solutions.

We first use Q_3, which joins images of 1 with images of 7. We generate tuple complaints for join results where the left (or right) side of the join was correctly predicted, but the right (left) side was incorrect. This results in 121, 550, and 931 complaints for the low, medium, and high corruption rates. Figure 4.5a shows that TwoStep and Loss perform poorly compared to Holistic, despite 550 complaints. When varying the corruption rate in Figure 4.5b, TwoStep improves slightly, but is still dominated by Holistic.

Our second experiment runs a COUNT aggregation (Q_4) on Q_3's results. The left relation contains images with digits 1 through 5; the right relation contains digits $6-9, 0$. The complaint says that the result should be 0—this is the same as a delete complaint on all join tuples, and states that all left tuples should not have the same prediction as any in the right relation. As expected, the lower ambiguity improves the likelihood that TwoStep's ILP picks a good satisfying solution, but the large standard deviation shows that it is unstable (Figure 4.5a). Figure 4.5d shows both Loss and TwoStep perform poorly across corruption rates; note that TwoStep is erratic between runs and doesn't show a clear trend.

Our third experiment joins two image datasets that overlap. We use the same relations as the previous experiment, and set the corruption rate to 50%. However, we move a subset of the 1 digit

images from the left relation to the right, which we call the *mix rate*. For example, a mix rate of 25% means that we move 25% of the 1 images (296 out of 1125) from the left relation to the right—the true output of Q_4 should be $829 \times 296 = 245384$, whereas the incorrect output was 1044470. As noted in Section 4.2.2, this is far more ambiguous than the previous experiment. As we vary the mix rate between 5%, 25%, 35%, the AUC_{CR} for Loss is stable at ≈ 0.24, whereas Holistic is initially high then decreases slightly ($AUC_{CR} = 0.78, 0.57, 0.48$, respectively). TwoStep does not solve the ILP within 30 minutes, thus we cannot report its results.

Takeaways: Overall, Holistic achieves the highest recall on SPA and SPJA queries as compared to the baselines as well as TwoStep. TwoStep is sensitive to the ILP solver as well as the level of ambiguity, which we will evaluate in the next subsection.

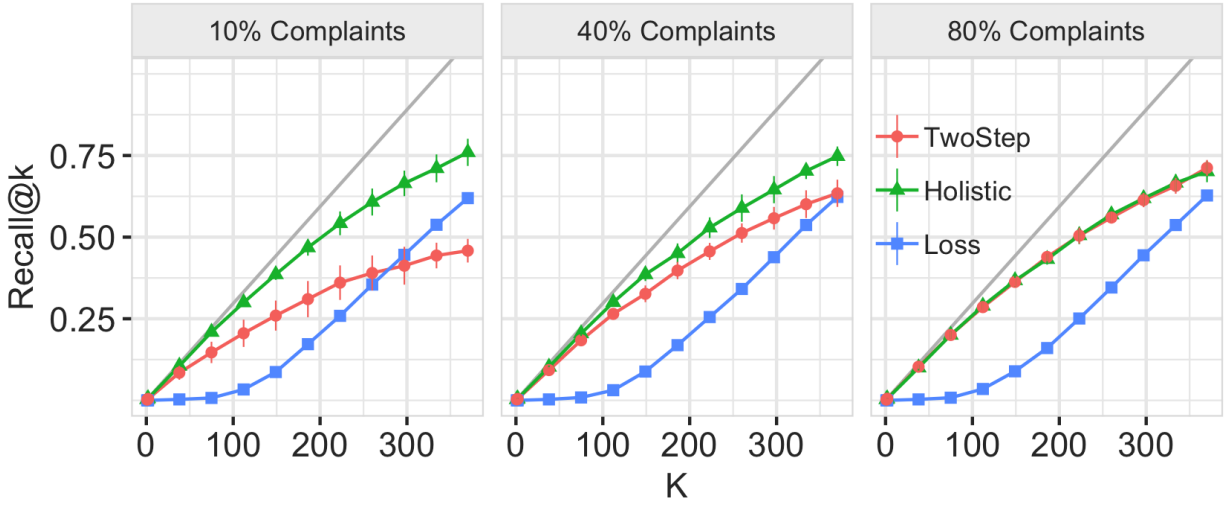


Figure 4.6: Varying ambiguity of the MNIST point complaints experiment. Each facet varies the percentage of join result complaints that are replaced with direct complaints over the model mispredictions.

4.3.4 Effects of Ambiguity

The previous experiments suggested the effects of high ambiguity on the different approaches. In this experiment, we use the same setup as Q_3 in the SPJ experiment, and carefully vary the amount of complaint ambiguity. In the previous experiment, the complaint only specifies that the join output record should not exist, but does not prescribe how to fix it. Here, we will replace a subset α of

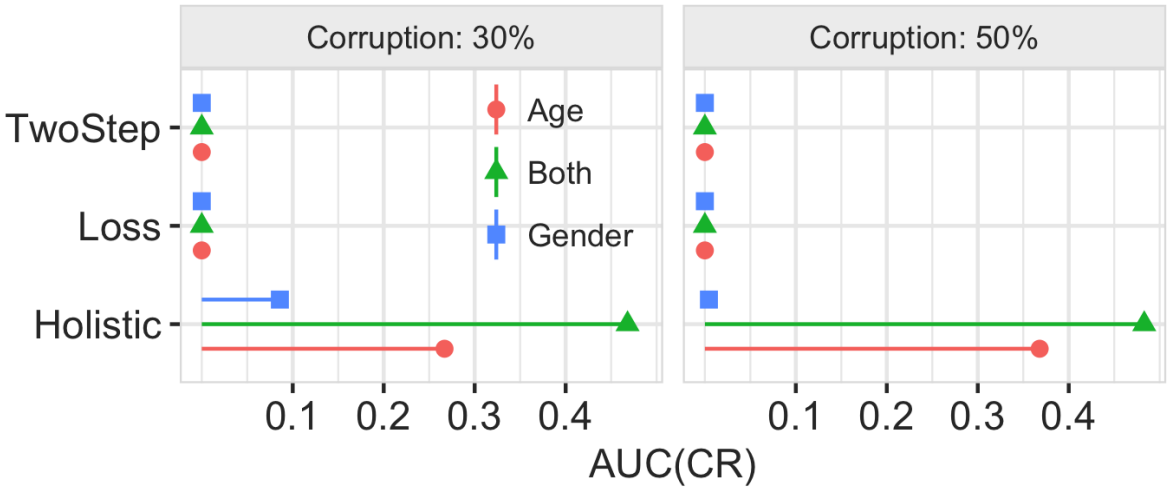


Figure 4.7: Holistic can benefit from combining complaints of multiple queries.

those complaints with unambiguous complaints. Specifically, for a complaint over a join output record $(l \in L, r \in R)$, we replace it with value complaints on the output of the model predictions $predict(l)$ and $predict(r)$. We corrupt 30% of the 1 digits as in the previous experiment.

Figure 4.6 shows that Holistic dominates the approaches at high ambiguity (10% complaints), however at low ambiguity (80%), TwoStep is competitive with Holistic. In addition, this experiment illustrates how Rain can make use of complaints from different parts of the query plan. Specifically, we can view the join record complaints as complaints on the output of Q_3 and the unambiguous complaints on the predictions of L and R as complaints that target the provenance of Q_3.

Takeaways: TwoStep is sensitive to ambiguity. TwoStep converges to Holistic when ambiguity is reduced by, for example, directly labelling many model mispredictions.

4.3.5 Multi-Query Complaints

So far, we have evaluated Rain using a single query and on a single attribute. In this experiment, we use the multi-attribute Adult dataset, and illustrate that complaints over *different* queries (that use the same model) can be combined to more effectively identify training set errors. We execute Q_6 and Q_7 from Table 4.1. Q_6 groups the dataset by *Gender* and creates a value complaint for the male average value. Q_7 aggregates the dataset by *Age* (bucketed into decades), and creates a value

complaint for the 40-50 age group's average value. To corrupt the training set, we select records that satisfy the conjunction of low income, male, and 40-50 years old, and flip α% of their labels from low income (y=0) to high income (y=1). 8.2% of the training set matches this predicate. We set $\alpha \in \{30\%, 50\%\}$ thus affecting the labels of 2.4% and 4.1% training points respectively.

Figure 4.7 shows that TwoStep, Loss, and Holistic when given each complaint in isolation, and when given both. TwoStep and Loss are unable to find any erroneous training records. One of the reasons is that the preprocessing step borrowed from [83] only uses three attributes to construct their features. This results in many duplicate training points (118/6512 points are unique). Thus, considerably more iterations for TwoStep and Loss are spent proposing and removing duplicates. Further, TwoStep's SQL step is agnostic to the model and training set, and fails to leverage this information when solving the ILP.

Holistic is, to a lesser degree, affected by the duplicates for the *Gender* complaint. This is because *Gender* is less selective than *Age*: in the training set, only 23.1% of males are between 40 and 50 but 71.3% of people between 40 and 50 are males. Holistic benefits considerably from using both complaints because they serve to narrow the possible training errors to those within the corrupted subspace.

Takeaways: Users often run multiple queries over the same dataset. We find that Holistic is able to leverage complaints across multiple queries. In contrast, techniques that are oblivious to the complaints (Loss) or oblivious to the model and training (TwoStep) perform poorly.

4.3.6 Do Complaints Reduce Debugging Effort?

One of the potential benefits of a complaint-based debugging approach is that users can specify a few aggregate but potentially ambiguous complaints, rather than label many individual, unambiguous, model predictions. In addition, it is desirable that complaints are robust to mis-specifications. For example, if a result value is 20 but should be 49, then a value complaint that is 50, or 60, or 45 should not greatly affect the returned training records. We now evaluate both of these questions in sequence. We use the MNIST dataset with corruptions that flip 10% of the training images with the

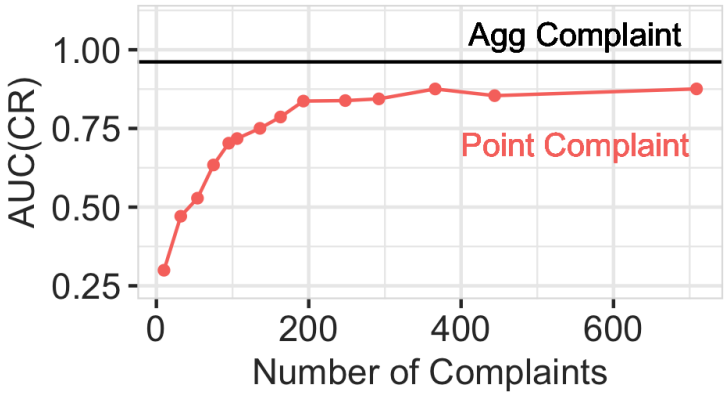


Figure 4.8: Comparison of one aggregate complaint (black) and increasing the number of point complaints (red) selected randomly among points predicted by the model to be 1.

digit 1 to be labeled 7.

First, we compare aggregate-level and prediction-level complaints. *Agg Complaint* is a single value complaint over Q_5, which counts the number of 1 digits; *Point Complaints* varies the number of complaints of model mispredictions from 1 to 709, and is equivalent to state-of-the-art influence analysis [12]). As complaining about random model predictions is unlikely to work, we pick the point complaints from the examples predicted to be 1, which in our experiments was the most successful option. Figure 4.8 shows that the aggregate complaint is enough to achieve $AUC_{CR} \approx 1$, whereas TwoStep requires over 200 point complaints to reach $AUC_{CR} \approx 0.87$. This suggests that, from an user perspective, aggregate-level complaints can require less effort.

A potential drawback of aggregate-level complaints is that they may be sensitive to mis-specification. To evaluate this, we introduce three types of errors to the user's value complaint. The errors vary in the user-specified X in the equality complaint $t[a] = X$, as compared to the ground truth X^*. *Overshoot* overcompensates for the error by setting $X = 1.2 \times X^*$, meaning if the query result was 10 and the ground truth was 100, then X is set to 120. *Partial* under-estimates the error but correctly identifies the direction the query result should move—X is set to the average of the query result and the ground truth (e.g., 55 in the preceding example). *Wrong* overcompensates in the incorrect direction, and sets $X = 0.8 \times t[a]$.

Figure 4.9 shows that Holistic is relatively robust to misspecified complaints, as long as they

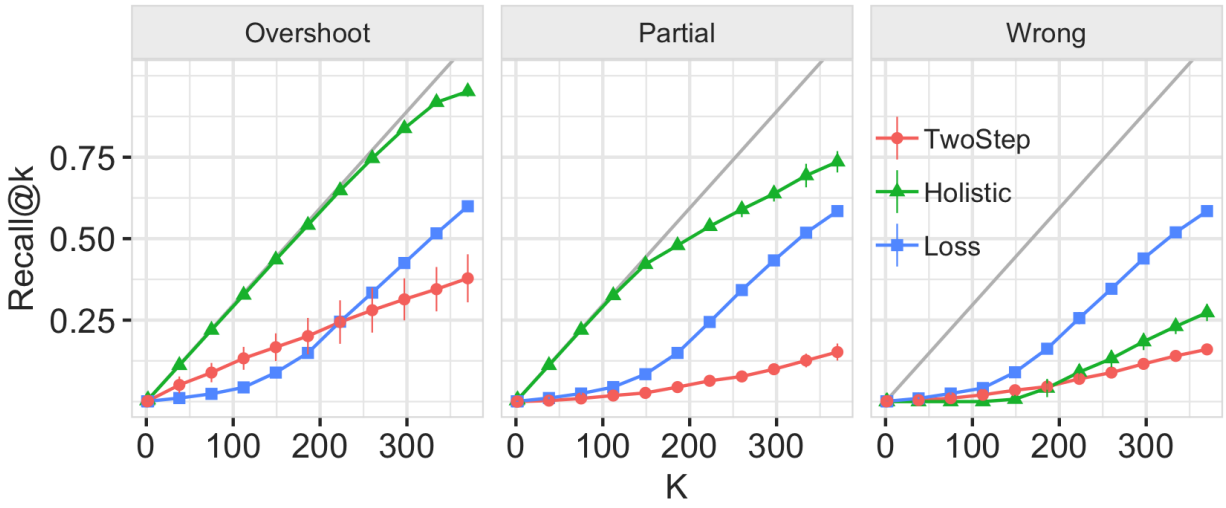


Figure 4.9: How errors in complaints affect each approach.

point in the correct direction of error. Specifically, the Holistic*Partial* curve degrades around $K = 150$ because the complaint has been satisfied. Holistic performs poorly when the complaint direction is *Wrong* because it tries to identify training records that if removed reduce the count whereas the true corruptions do the opposite. TwoStep similarly degrades, whereas Loss is insensitive because it does not rely on complaints at all.

Takeaways: Complaint-based approaches allow users to provide few ambiguous complaints over aggregated results, and still accurately identify training set errors. Holistic is robust to misspecifications as long as the direction of the complaint is correct.

4.3.7 Debugging Neural Networks

In this section, we evaluate TwoStep, Loss, and Holistic on a convolutional neural network (CNN) model. We execute the COUNT query Q_5 on the MNIST dataset, and we corrupt the training set by flipping 50% of the 1 digit images to be labeled 7. The CNN model consists of 3-layers (convolution, max pooling, dense with RELU activation). [12] showed empirically that the influence function analysis works even for neural networks, which are non-convex, including CNN architectures. We also include the logistic regression model for comparison.

This section reports the AUC_{CR} for the three approaches. We find that Holistic degrades slightly

when debugging the CNN model. This agrees with [12], where influence analysis scores were shown to be less accurate for non-convex than convex models. Recent work on influence analysis [76] has provided improved approaches that are more accurate on non-convex models. Rain is compatible with [76] and can continue to leverage any improvements on influence analysis by the ML community.

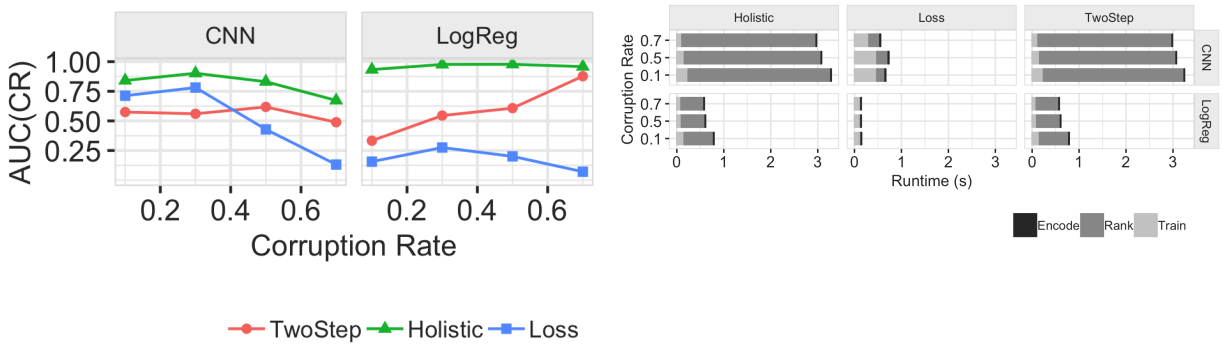


Figure 4.10: a) AUC_{CR} when using CNN and logistic regression models. b) Per-iteration runtimes for debugging CNN and logistic regression models for different corruption rates.

Section 4.3.7 shows that per-iteration runtimes for Loss is dominated by retraining costs. We note that the models are trained incrementally in each iteration i.e. the previous values of the weights are used as initializations for the next debugging iteration. Thus, retraining costs can vary across methods depending on which points are removed. Intuitively removing points with high loss may result in significant model changes and thus can lead to higher retraining costs, explaining the higher retraining time for Loss in Section 4.3.7 when compared to TwoStep and Holistic.

In contrast, TwoStep and Holistic are dominated by calculating the Hessian vector products required by the Conjugate Gradient algorithm. Even if it is much faster than naively computing the inverse Hessian, its cost grows linearly with the number of parameters of the model.

Takeaways: Holistic supports queries that use neural network models. It performs well under low and moderate corruption rates and dominates TwoStep and Loss. However, each iteration takes ≈ 3 *seconds due to the cost of the ranking (hessian inverse) step, which is particularly costly for neural network models.*

Chapter 5: Training Data Debugging at Interactive Speeds

In Chapter 4 we proposed Rain, the first step towards *complaint driven training data debugging*. Unfortunately, it has performance and scalability challenges that limit its use to small models and training datasets. Complaint driven debugging is typically initiated from a data visualization, where the user can easily see anomalies and interactively annotate and specify them as complaints. In this context, it is crucial that the debugging system responds at interactive timescales to not impede the user's analysis flow [88]. However, as shown in Figure 5.1, Rain's responsiveness quickly degrades beyond models containing a few thousand parameters. Unfortunately, modern deep neural network (DNN) models, such as Wide Residual Networks (WRN) used in image classification, can have millions of parameters. At such scales, Rain takes minutes to identify and rank erroneous training records for a complaint, whereas our goal is to debug at interactive time scales (< 100ms [89, 90]).

Rain's poor scalability arises from the cost of estimating two types of model sensitivities. The first type is the sensitivity of the model parameters to removing examples from the training set. Instead of analyzing the sensitivity of the loss function directly, which requires retraining the model and rerunning the query, Rain uses a quadratic Taylor approximation of the loss function that is faster. However, it still requires computing the second order derivative (the Hessian) which is expensive. The second type is the sensitivity of model predictions (and the inference query) to changes in the model parameters, whose computation cost increases with the model size. Rain approximates the effects of training set deletions on the relaxation of the query without materializing the quadratic approximation of the loss function. Unfortunately, for a training set of size $|T|$ and d model parameters, Rain still costs $O(d|T|)$, which is untenable for non-trivial models.

Our work builds on three insights. First, a significant amount of computation can be pushed offline by precomputing the quadratic approximation of the loss function (Section 5.1). However, a naive approach requires considerable space and only reduces latency by a constant factor. Thus,

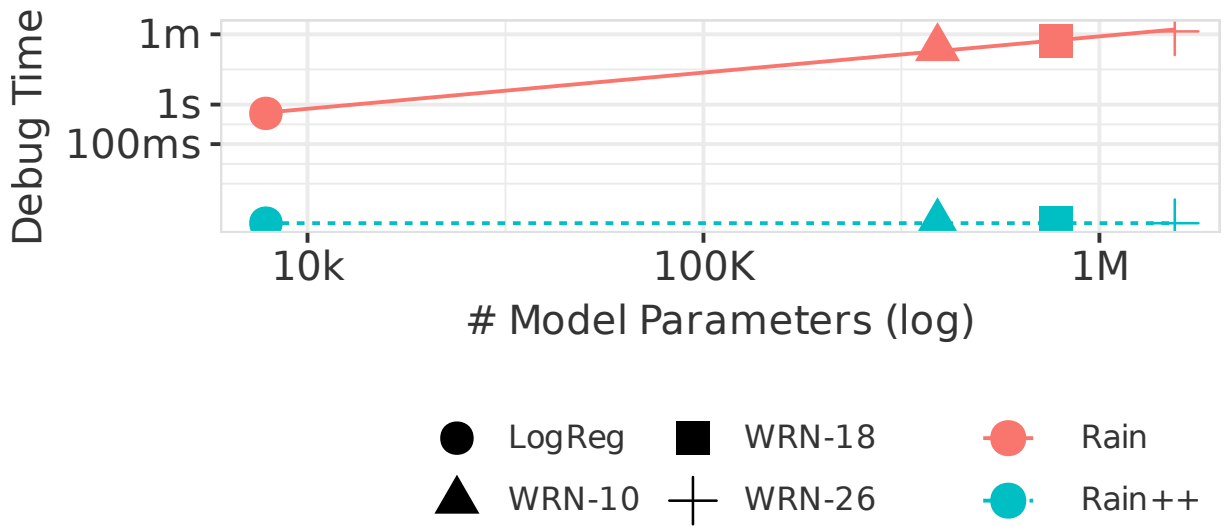


Figure 5.1: Rain++ (this chapter) reduces the time for complaint driven data debugging by over 70000× to support ~1ms interactive debugging. This complaint is over a join-count query that where the join condition is $M.predict(left) = M.predict(right)$.

our second insight is to build space-efficient approximations of the model loss function that only rely on a tiny subset of the model parameters (Section 5.2). Although conventional wisdom towards loss function approximation is to choose parameters that are most sensitive to training set perturbations, we show that the exact opposite is true for influence-based complaint debugging. Namely, that the more sensitive a model parameter is, the less the model relies on it when making predictions—in other words, the less it contributes to debugging! In fact, including sensitive parameters introduces numerical instabilities that *degrade* debugging quality. Finally, we observe that, rather than compress the model parameters directly, it is even more effective to directly compress the quadraic approximation of Rain (Section 5.2).

Rain++ uses these insights to precompute a small number of eigenvalues and eigenvectors of the loss function's inverse Hessian. While matrix compression traditionally computes the largest eigenvalues, we show counter-intuitively that the *smallest* eigenvalues are most appropriate for training data debugging (Section 5.2). We further develop optimizations when the inference database or the inference query is known apriori. The former precomputes gradients for inference DB tuples that accelerate *any* future inference query, while the latter incrementally maintains the query gradient

as new batches of records are inserted.

These optimizations reduce complaint debugging latency by >70000× from over 1 minute to ~1ms (Figure 5.1). The precomputation costs are modest: less than 30 minutes for a WRN-26 neural network model with 1.5M parameters. Beyond scalability, Rain++ addresses two additional limitations in Rain. First, Rain relies on access to the model training infrastructure (preprocessing pipeline, model definition, and parameters) in order to compute the above derivatives and gradients. However, model *users* rarely have access to this infrastructure. Rain++ can debug complaints solely using a set of precomputed data structures, obviating the need for this access. Users with access to the raw training data can analyze Rain++'s output and make suggestions to upstream model developers. Otherwise, Rain++ can compress their complaint into a vector which can be handed off to the model development team. Second, Rain assumes that deleting errors is always appropriate. However, this is both undesirable when training records are sparse, and incorrect if the errors cannot be fixed by deletions. We propose extensions to support updated-based interventions and illustrate in the experiments how the correct intervention choice is crucial for training data debugging.
To summarize, our contributions include:

- Offline precomputation techniques that both speed up complaint driven debugging and improve numerical stability.
- Offline precomputation techniques when the inference database is know a priori (e.g., a published dashboard).
- Maintenance-based optimizations for streaming queries where new batches of data are inserted into the inference database.
- Extension of the Rain problem formulation to support interventions that fix, rather than delete, erroneous training examples.
- Extensive evaluations using image (MNIST, FASHION-MNIST, CIFAR10), text (SST2), and tabular (ADULT) datasets, and a variety of linear and neural network models (CNNs,

Feed Forward Nets, Logistic Regression, WRNs, LSTMs). Rain++ has comparable or better debugging accuracy, >70000× lower latency, and supports non-deletion interventions.

- In-depth analysis of the conditions when complaint-based debugging can be expected to be effective. We find evidence that complaints based on queries whose outputs significantly incorrect are more likely to accurately identify training errors, which matches the settings when an end-user will identify and submit a complaint. **Critically, we show that knowing how a model is used in the downstream application is critical to accurate and efficient data debugging.** This is leveraged in other problem areas such as domain adaptation [20] in the ML literature, but rarely applied in the data cleaning literature.

5.1 Limitations of Rain

The primary bottleneck of Rain is computing the first and second order model derivatives, especially when d is large, since they are needed when calculating $\nabla_\theta q(\theta^*)$, the $H_{\theta^*}w$ computations for the Conjugate Gradient algorithm, as well as $\nabla_\theta l(\theta^*, z)$ for all training examples in T. The resulting complexity of the algorithm, even if we ignore the calculation of $\nabla_\theta q(\theta^*)$, is $O(|T|d)$. Thus, Rain will not remain interactive when wither d or $|T|$ is large.

Fundamentally, a complexity of $O(|T|d)$ should be expected for any solution– a solution must, at minimum, evaluate M over all training examples in order to return a ranking. To go beyond constant factor improvements over Rain, a significant portion of the computation needs to be made *complaint independent*, meaning it is independent of Q, so that it can be pushed offline.

Unfortunately naively doing so brings constant factor improvements at best. Computing H_{θ^*} or its inverse offline makes things worse: Even reading H_{θ^*} or its inverse takes $O(d^2)$ which is slower than $O(|T|d)$ for most state of the art neural net architectures. Another approach would be to calculate offline and store $H^{-1}_{\theta^*}\nabla_\theta \ell(\theta^*, z)$ for every training example z in T. While we avoid a significant amount of first and second order derivatives at online time, the online complexity remains $O(|T|d)$ yielding only a constant factor improvement . On top of that, the offline cost can be prohibitive, requiring $O(d|T|^2)$ time and $O(|T|d)$ space. Clearly neither approach is practical.

5.2 Insights from Optimization

The critical challenge is computing $H_{\theta^*}^{-1}\nabla_\theta \ell(\theta^*, z)$ for all z in T using less than $O(d|T|^2)$ time and $O(d|T|)$ space. Given it is unrealistic to solve one linear system at a time using less than $O(d|T|^2)$ time, our goal is to compress $H_{\theta^*}^{-1}$ and quickly process each z.

Given a matrix operation $A \cdot b$ where A is a square matrix, the predominant way to compress A is low rank factorization [91]. This keeps the top eigenvalues and eigenvectors of A, which captures the set of vectors b where $A \cdot b$ most sensitive. This ensures that the accuracy along those sensitive directions will be high. Despite these guarantees, low rank factorization is *not* appropriate due to interactions between $A = H_{\theta^*}^{-1}$ and $b = \nabla_\theta \ell(\theta^*, z)$ unique to our problem. This section provides intuition for why Rain++ instead compresses $H_{\theta^*}^{-1}$ using its *smallest* eigenvalues.

First, the smallest eigenvalues are most accurate for representing the effects of training set changes on model predictions. In fact, recent work in deep learning optimization suggests that state-of-the-art neural networks training loss gradients are concentrated on the subspace spanned by the smallest eigenvectors of $H_{\theta^*}^{-1}$ [92]. We illustrate this using a simple example based on a linear regression model with four training examples with features X and targets y

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 10^{-3} \\ 0 & -10^{-3} \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 1 \\ 3 \end{bmatrix}.$$

Learning the two feature weights $\theta = (\theta_1, \theta_2)$ amounts to minimizing the squared loss

$$\begin{aligned} L(\theta) &= \sum_{i=1}^{4} \ell(\theta, z_i) = \sum_{i=1}^{4} (\langle x_i, \theta \rangle - y_i)^2 \\ &= (\theta_1 - 1)^2 + (-\theta_1 - 3)^2 + (10^{-3}\theta_2 - 1)^2 + (-10^{-3}\theta_2 - 3)^2 \\ &= 2\theta_1^2 + 4\theta_1 + 10 + 2 \cdot 10^{-6}\theta_2^2 + 4 \cdot 10^{-3}\theta_1 + 10 \end{aligned}$$

The optimal parameters are $\theta^* = (-1, -10^3)$, and inverse Hessian is

$$H_{\theta^*}^{-1} = \begin{bmatrix} \frac{\partial^2 L(\theta)}{\partial^2 \theta_1} & \frac{\partial^2 L(\theta)}{\partial \theta_1 \partial \theta_2} \\ \frac{\partial^2 L(\theta)}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 L(\theta)}{\partial^2 \theta_2} \end{bmatrix}^{-1} = \begin{bmatrix} 4 & 0 \\ 0 & 4 \cdot 10^{-6} \end{bmatrix}^{-1} = \frac{1}{4} \begin{bmatrix} 1 & 0 \\ 0 & 10^6 \end{bmatrix}.$$

The top eigenvector of $H_{\theta^*}^{-1}$ is $(0, 1)$ with eigenvalue $0.25 \cdot 10^6$. For training example $z_i = (x_i, y_i)$, its training gradient is $\nabla_\theta \ell(\theta^*, z_i) = 2(\langle x_i, \theta^* \rangle - y_i) x_i$. Thus for z_1 and z_3 we have

$$\nabla_\theta \ell(\theta^*, z_1) = \begin{bmatrix} -4 \\ 0 \end{bmatrix} \quad \nabla_\theta \ell(\theta^*, z_3) = \begin{bmatrix} 0 \\ -4 \cdot 10^{-3} \end{bmatrix}.$$

Examples z_2 and z_4 have the same gradients as z_1 and z_3 albeit with opposite signs. Note that the gradients of all of these examples are concentrated around the inverse Hessian's bottom eigenvector $(1, 0)$ direction, rather than the top eigenvector. Thus, multiplying the smallest eigenvalues of $H_{\theta^*}^{-1}$ with the loss gradients will more accurately approximate $H_{\theta^*}^{-1} \nabla_\theta \ell(\theta^*, z)$.

In general, it's a problem if model predictions rely heavily on model parameters that are overly sensitive to small training set changes, which are precisely captured by the directions of the largest eigenvectors of $H_{\theta^*}^{-1}$.

Second, the largest eigenvalues greatly *reduce*, rather than improve, the accuracy of influence function approximations. Suppose we add a new training example z_5 with $x_5 = (0, 1)$ and $y_5 = 1$. The new loss term $(\theta_2 - 1)^2$ in $L(\theta)$ dominates all existing θ_2 terms. The new optimal parameters should be $\approx (-1, 1)$, yet our approximation using Equation (3.2) is wildly off

$$\theta_{new}^* \approx \theta^* - H_{\theta^*}^{-1} \nabla_\theta \ell(\theta^*, z_5) \approx \begin{bmatrix} -1 \\ 5 \cdot 10^8 \end{bmatrix}.$$

The reason is that influence functions employ a Taylor approximation that is accurate only in a small neighborhood of the original solution θ^*. Sensitive parameters like θ_2 can change considerably when the training set is perturbed, and render the Taylor approximations highly inaccurate.

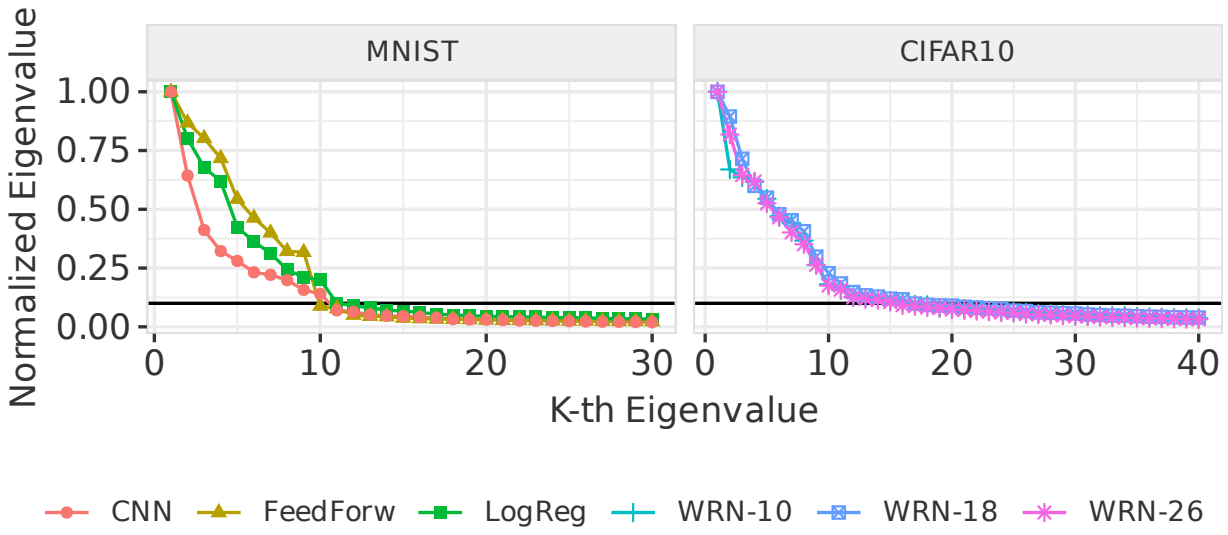


Figure 5.2: Caching the top-k Hessian eigenvalues for MNIST and CIFAR10 (with $K \approx 15$) is sufficient for influence functions.

Third, accurately estimating the top eigenvalues themselves suffers from numerical stability issues. Hessians of modern NN architectures, like the ones used in computer vision, are low rank with 99.99% of the Hessian eigenvalues being near-zero [93]. Since the eigenvalues of the inverse Hessian are reciprocal to the ones of the Hessian, calculating the top eigenvalues of the inverse suffers from numerical stability issues.

Looking ahead: The above points highlight that large eigenvalues of the $H_{\theta^*}^{-1}$ should not be used for influence analysis, and that the smallest eigenvalues should instead be used. Further, Figure 5.2 shows that 10-20 eigenvalues are sufficient to store for even million-parameter models. Surprisingly, in addition to its performance benefits, this also improves the approximation accuracy and robustness compared to computing and using the full inverse Hessian matrix.

This significant amount of compression is enabled by the parameter redundancy of modern NN architectures. NNs learn to recognize patterns common to each predicted class. These common patterns are encoded in the weights of the neurons so that examples of the same class exhibit the same neuron activation patterns leading to the same classification. These co-activating neuron groups correspond to the dominant eigenvectors of the Hessian. A sufficiently powerful model may need only few such groups for each class. This agrees with our empirical observations where for 10

class models of **MNIST** and **CIFAR-10** in Figure 5.2 have close to 10 dominant eigenvectors. In contrast, neurons that do not belong to these dominant groups are rarely activated during training. Thus they are highly sensitive to small training set changes but also less likely to be critical for model predictions.

5.3 Our Approach

As we discussed in Section 5.2, Rain++ computes $H_{\theta^*}^{-1} \nabla_\theta \ell(\theta^*, z)$ for all z in T offline to accelerate the online evaluation of Equation (3.3). The basis of our implementation is the approximation of $H_{\theta^*}^{-1}$ through the top eigenevectors of H_{θ^*}. Notice that because the eigenvalues of H_{θ^*} and $H_{\theta^*}^{-1}$ are reciprocal, the top eigenvectors of H_{θ^*} correspond to the bottom eigenvectors of $H_{\theta^*}^{-1}$. Let v_i and λ_i be the eigenvectors of H_{θ^*} in descending order. Rain++ computes only the top-k v_i and λ_i to replace H_{θ^*} in Equation (3.3) with the following surrogate

$$\tilde{H}_{\theta^*} = \sum_{i=1}^{k} \lambda_i v_i v_i^T.$$

The matrix above coincides with the exact Hessian for $k = d$. The crux of our implementation is to approximate $H_{\theta^*}^{-1} \nabla_\theta \ell(\theta^*, z)$ while materializing as few of its intermediates of the computation as possible. Rain++ computes the top-k v_i and λ_i without first materializing the uncompressed H_{θ^*}. $H_{\theta^*}^{-1} \nabla_\theta \ell(\theta^*, z)$ is then computed directly in a compressed format without needing to materialize $\nabla_\theta \ell(\theta^*, z)$ first. Further, we will remark on heuristics to estimate k – in our experiments, k is on the order of 10 or 15, as compared to the $>10^6$ model parameters.

5.3.1 The Lanczos Algorithm

Given H_{θ^*}, computing its top-k eigenvalues and eigenvectors would be a straightforward task using any linear algebra library. As we have noted before though, even computing the full Hessian would take $O(|T|d^2)$ time which in practice would be prohibitive.

The Lanczos Algorithm [94] is a generalization of the CG algorithm that allows us to compute

eigenvalues and eigenvectors of H_{θ^*} without requiring access to H_{θ^*} (recall that CG was used to simply compute $\nabla_\theta q(\theta^*) H_{\theta^*}^{-1}$). Similar to CG, the Lanczos Algorithm only requires access to an oracle that, given a v, computes $H_{\theta^*} v$. This again can be done using backpropogation in an auto-differentiation framework such as TensorFlow. For $k \ll d, |T|$ as we discussed, the complexity of this algorithm is $O(k|T|d)$.

5.3.2 Gradient Compression

Replacing H_{θ^*} with $\tilde{H}_{\theta^*}$ in Equation (3.3) we get

$$q(\theta^*_{new}) \approx q(\theta^*) - \nabla_\theta q(\theta^*) \sum_{i=1}^{k} \frac{1}{\lambda_i} v_i v_i^T \nabla_\theta \ell(\theta^*, z).$$

We can reorganize the expression using vector notation to highlight the opportunity to compress $H_{\theta^*}^{-1} \nabla_\theta \ell(\theta^*, z)$:

$$q(\theta^*_{new}) \approx q(\theta^*) - \sum_{i=1}^{k} \langle \nabla_\theta q(\theta^*), v_i \rangle \frac{1}{\lambda_i} \langle v_i, \nabla_\theta \ell(\theta^*, z) \rangle. \tag{5.1}$$

The compressed version of $H_{\theta^*}^{-1} \nabla_\theta \ell(\theta^*, z)$ that Rain++ stores is thus

$$b_z \longleftarrow \left[\frac{1}{\lambda_1} \langle v_1, \nabla_\theta \ell(\theta^*, z) \rangle, \cdots, \frac{1}{\lambda_k} \langle v_k, \nabla_\theta \ell(\theta^*, z) \rangle \right].$$

Observe that b_z has size k much smaller than the uncompressed d.

Computing all the b_z exhibits a time-space trade-off. Computing large batches of $\nabla_\theta \ell(\theta^*, z)$ to leverage GPU parallelism requires large amounts of GPU memory, a limited resource. The situation is more dire compared to model training since we compute one d dimensional gradient per training example in the batch and not one gradient per batch as in stochastic gradient descent.

Rain++ computes the $\langle v_i, \nabla_\theta \ell(\theta^*, z) \rangle$ directly without materializing $\nabla_\theta \ell(\theta^*, z)$. Rain++ views the calculation of each of the k projections as the derivative of a function of one scalar variable h

$$\langle v_i, \nabla_\theta \ell(\theta^*, z) \rangle = \frac{\partial \ell(\theta^* + h v_i, z)}{\partial h} \bigg|_{h=0}.$$

Backpropagation here ends up calculating $\nabla_\theta \ell(\theta^*, z)$ and projecting it to v_i which does not help. Forward mode differentiation [95], available in frameworks like Tensorflow, can calculate $\langle v_i, \nabla_\theta \ell(\theta^*, z)\rangle$ on the fly while evaluating $\ell(\theta^*, z)$. As a result $\nabla_\theta \ell(\theta^*, z)$ is never materialized. The dramatic memory reduction allows for significantly larger batch sizes that more than make up the cost of the $k \ll d$ passes needed, one for each v_i.

Storing the k vectors v_i, and b_z for each training example z, reduces space complexity from $O(|T|d)$ to $O(kd+k|T|)$. Given $\nabla_\theta q(\theta^*)$, they also reduce the online computation of Equation (5.1) for all training examples from $O(|T|d)$ in Rain to $O(kd+k|T|)$. In our experiments, for a WRN-26 model of 1.5M parameters and 50K training examples, setting $k = 20$ our technique reduces the cost by one order of magnitude. For this setting, our forward mode based gradient compression is 4 times faster than computing the uncompressed gradients with backpropagation.

5.3.3 Choosing the Number of Eigenvalues

It is clear that the choosing the right number of eigenvalues k is critical for Rain++. Given that no uniform choice of k works for all models and datasets, it is important to design heuristics to identify an appropriate setting. As we can see in Figure 5.2, at the beginning of the spectrum of the Hessian eigenvalues decrease rapidly with λ_{i+1}/λ_i being significantly lower than one. This behaviour continuous until we reach an inflection point after which the drop-off stops and λ_{i+1}/λ_i spikes to a value close to one. Our heuristic initializes k to the number of classes, which is 10 for Figure 5.2, and then scans the spectrum to identify the first inflection point. In our experiments, we observe that the chosen k consistently achieves performance that is close to the optimal choice.

To allow for quicker calibration of k, one could use a model with fewer parameters as a cheap proxy. Increasing d tends to have diminishing returns in terms of model capacity. Models with far smaller d may have similar capacity to learn the unifying patterns of the task resulting in a similar number of dominant eigenvectors as discussed in Section 5.2. In Section 5.5.4 we find that for overparametrized models the optimal k does not vary significantly with d.

5.4 Optimizations and Extensions

The previous sections focused on pushing the computations of first and second order derivatives of M over the training set offline. In our discussions we have ignored the cost of computing $\nabla_\theta q(\theta^*)$ which can become the new bottleneck given our optimizations. In this section we discuss two optimizations to address this.

5.4.1 Known inference database

One of the key cases where Rain++ can accelerate the computation of $\nabla_\theta q(\theta^*)$ is when the inference database $\mathcal{D}$ is known offline. Let V be the total number of model predictions for $\mathcal{D}$ and R is the number of model classes. Rain relaxes the complaints C in differentiable functions q that operate on top of the $V \times R$ matrix of prediction probabilities $P(\theta)$ of each inference of M over $\mathcal{D}$. An element $p_{ij}(\theta)$ of $P(\theta)$ corresponds to the probability of class j assigned to the i-th inference example of $\mathcal{D}$. As a result, q can be written as a function f applied on $P(\theta)$. Using the multi-variate chain rule on the equality on $q(\theta) = f(P(\theta))$ we get

$$\nabla_\theta q(\theta^*) = \sum_{i=1}^{V} \sum_{j=1}^{S} \frac{\partial f(P(\theta^*))}{\partial p_{ij}} \nabla_\theta p_{ij}(\theta^*). \tag{5.2}$$

Equation (5.2) decomposes the sensitivity of the relaxed complaint q in two distinct factors. The first one is the sensitivity of q to the changes of the probabilities in $P(\theta)$ expressed by $\partial f(P(\theta^*))/\partial p_{ij}$. The second is the sensitivity of the prediction probabilities to model parameter changes $\nabla_\theta p_{ij}(\theta^*)$. Despite the fact that there is an infinite number of potential complaints, using Equation (5.2) all complaint gradients can be expressed as a linear combination of the VR gradients $\nabla_\theta p_{ij}(\theta^*)$.

This gives a concrete approach towards accelerating the computation of $\nabla_\theta q(\theta^*)$. We can compute offline all the VR gradients $\nabla_\theta p_{ij}(\theta^*)$ as well as the matrix $P(\theta^*)$. During the online computation, we can construct the function f that corresponds to the user's complaint. Since $\partial f(P(\theta^*))/\partial p_{ij}$ depends only on $P(\theta^*)$ and the complaint and $\nabla_\theta p_{ij}(\theta^*)$ is already computed, we can compute $\nabla_\theta q(\theta^*)$ without requiring any additional model inference or derivative.

Unfortunately it is prohibitive to store $\nabla_\theta p_{ij}(\theta^*)$ for large models as it takes $O(VRd)$ space. However the computation of Equation (5.1) requires only the k projections $\langle \nabla_\theta q(\theta^*), v_i \rangle$. Applying this projection to Equation (5.2) we get

$$\langle \nabla_\theta q(\theta^*), v_i \rangle = \sum_{i=1}^{V} \sum_{j=1}^{S} \frac{\partial f(P(\theta^*))}{\partial p_{ij}} \langle \nabla_\theta p_{ij}(\theta^*), v_i \rangle. \tag{5.3}$$

Thus storing only $\langle \nabla_\theta p_{ij}(\theta^*), v_i \rangle$ is sufficient, reducing space to $O(VRk)$. The eigenvectors v_i are also no longer required for the online computation so only $O(VRk + k|T|)$ space is needed which is independent of d. This applies to the online time complexity as well where given $\partial f(P(\theta^*))/\partial p_{ij}$, we only require $O(VRk + k|T|)$. In Figure 5.7 we show that this optimization can reduce the cost of computing $\nabla_\theta q(\theta^*)$ by three orders of magnitude. For WRN-26 model of 1.5M parameters and $V = 10$K, $R = 10$ and $k = 20$ using forward mode gradient compression is 12 times faster than just calculating the gradients with backpropagation. The increased speed up compared to Section 5.3.2 is because for each example i backpropation does a single forward pass to compute all class probabilities $p_{ij}(\theta^*)$ but needs to do one backward pass for each class to compute all $\nabla_\theta p_{ij}(\theta^*)$. In contrast, forward mode calculates $\langle \nabla_\theta p_{ij}(\theta^*), v_i \rangle$ for all j in a single forward pass.

5.4.2 Streaming queries

Another important setting where interactive response times are critical is the case where there is an incoming stream of inference examples. Although Equation (5.3) is in theory always applicable, as the stream increases in size, computing the projections of $\nabla_\theta q(\theta^*)$ from scratch becomes increasingly more costly. In this section we will discuss how we can incrementally update $\nabla_\theta q(\theta^*)$ for complaints over streaming queries.

For simplicity, we focus on a streaming aggregation query Q. Let us assume that we start with an inference database $\mathcal{D}$ and after a single tuple insert we get a database $\mathcal{D}'$. Since SPJA queries are incrementally maintainable, we know that there is a query ΔQ, a delta query as it is usually

called, that efficiently computes the difference in the value of Q

$$\Delta Q = Q(\mathcal{D}'_M) - Q(\mathcal{D}_M).$$

ΔQ is itself an SPJA query that Rain can analyze and relax just like it would do for the original query Q. Let $h(\theta)$ be the relaxation of $Q(\mathcal{D}_M)$, $\Delta h(\theta)$ be the relaxation of ΔQ and $h'(\theta)$ be the relaxation of $Q(\mathcal{D}'_M)$. The relaxation of Rain preserves addition so we have

$$\nabla_\theta h'(\theta^*) = \nabla_\theta h(\theta^*) + \nabla_\theta \Delta h(\theta^*).$$

Given $\nabla_\theta h'(\theta^*)$ and $h'(\theta^*)$, which we can compute by a similar addition rule, we can compute any complaint gradient on top of $Q(\mathcal{D}'_M)$ via the chain rule. Thus to the extent that ΔQ depends only on a small number of model inferences, we can incrementally compute the complaint gradient $\nabla_\theta q(\theta^*)$ efficiently. As a canonical example, in Figure 5.9 we will study the case of streaming class frequency counts. For this case, the update cost depends only on the size of the incremental update which allows Rain++ to scale to very large databases $\mathcal{D}$.

5.4.3 Non-Deletion Interventions

The above discussion is focused on the context where query complaints can be fixed by deleting corrupted training examples. However, there may be other valid interventions. For instance, the user may wish to apply a low-pass filter to fix images with random or salt-and-pepper noise, or to set erroneous numerical attributes to a default or median value. In addition, when the set of relevant training records is limited (e.g., there are few examples for a given class), deleting the corrupted records is undesirable as it reduces the effective number of samples that are available for training.

Rain can use Equation (3.6) and the optimizations described in this chapter to approximate the effects of any per-record intervention, and rank them based on how well they address the query complaint. In our experiments, we show the importance of corruption-relevant interventions on query complaints. We implement this by precomputing the interventions on all training records,

along with their corresponding offline data structures.

5.5 Experiments

Our experiments seek to understand how the choice of k affects Rain++'s debugging quality, offline, and online runtimes. Comparing with the baseline Rain system we find that Rain++ maintains or improves debugging quality and reduces online runtimes by orders of magnitude, while requiring modest amounts of offline precomputation times. We further study the characteristics of complaints, as well as types of intervention, that affect debugging quality.

5.5.1 Experimental Settings

The optimal number of eigenvectors k depends on the hessian's spectral properties, which varies based on datasets, tasks and model architectures. Thus we vary these three dimensions.

Datasets & Models

Scalability becomes a major factor as the number of model parameters increases. Thus we focus on settings that use deep neural networks (DNNs). We use 3 object classification image datasets, and a sentiment analysis NLP dataset. We also use a tabular dataset to show that Rain++ is competitive even on models with fewer parameters that are not overparameterized.

- **MNIST** [86] contains 70k gray scale 28×28 pixel images of handwritten digits 0-9. 60k are used for training and 10k for testing. The model classifies each image with the depicted diget. We trained three models: a Logistic Regression model with 7850 parameters, a two layer Feed Forward network with 1.8M parameters, and a three layer CNN with 1.2M parameters.

- **Fashion-MNIST** [96] is a harder version of **MNIST**. It has the same number of image dimensions, but the images are of clothing from 10 clothing classes. We use the same models as MNIST.

- **CIFAR-10** [97] contains 60k 32×32 color images of 10 different object classes; 50k are used for training. For this classification task, we use three Wide Residual Network (WRN) models [98]

with 10, 18 and 26 layers (390K, 778K and 1.55M parameters respectively). This is a harder task than **MNIST** and **Fashion-MNIST**.

- **SST-2** [99], or the Stanford Sentiment Treebank v2, is a binary sentiment analysis dataset. The training set contains 67349 sentence fragments labelled as positive or negative sentiment. For this binary classification task, we use a LSTM based classifier with 3.4M parameters.

- **ADULT** [82] is a tabular dataset that predicts whether a person makes more or less than $50K per year, given their census information. DNNs often fail to offer competitive performance on tabular datasets as compared to simpler alternatives like linear models or decision trees [100]. We thus use a Logistic Regression classifier with 50 parameters.

Training Set Errors

Training examples can have errors in the features, labels, or both; the errors can be random or systematic over the training set. We generate systematic corruptions by choosing a subset of the training set that satisfies a feature or label-based predicate, and adding errors to a random subset of those examples. Random corruptions are uniformly distributed in the dataset. Tables 5.2 and 5.3 and summarize the corruption and rates we use for label and feature corruptions.

- **Class-conditional Label Error** chooses a class from the training set, and flips a percentage (the corruption rate) of those labels to another class. For example, we flip 40% of **MNIST** '1' labels to '7' ultimately corrupts 4% of the total training set. We vary the corruption rates in 10% increments. For **MNIST** we include a generalization where training examples of two classes are flipped to two other classes at the same corruption rate per class.

- **Feature Noise** adds Salt & Pepper and gaussian blur to a random or systematic subset of the image training examples. Salt & Pepper randomly sets 30% of the image pixels to either 0 or 1 with equal probability. Gaussian blur convolves the image using a Gaussian kernel of $\sigma = 2px$, resulting in a blurred image. Systematic corruption is done by corrupting a subset of examples from one class.

- **Feature-conditional Label Error** chooses a predicate defined over the features of the model and corrupts the label of a percentage of the examples that satisfy it. For **ADULT** we set the label of training examples representing males who work in the private sector and get less then \$50K to ≥\$50K.

Note that the random corruption rates are over the *full training set*, whereas class-conditional rates are with respect to the *subset* of the training set with the corrupted label value. We also use Salt & Pepper noise to evaluate non-deletion interventions in Section 5.5.9

Complaints

We evaluate complaints over three types of aggregation queries shown in Table 5.1. The complaint specifies that the aggregation output is either too high or too low, depending on its value as compared to the ground truth.

Measures

Following Rain, we summarize the quality of the top-k results using AUC_R. Let r_i be the percentage of correctly identified corrupted training examples in the top-i ranked points. AUC_R averages the r_i up to the true number of corruptions N, i.e. $\frac{1}{N}\sum_{i=1}^{N} r_i$. The result is divided by its maximum value to derive a normalized score in $[0, 1]$. We also evaluate offline and online runtimes.

Implementation

Rain and Rain++ are implemented in JAX [101], an automatic differentiation framework on top of XLA [102]. All experiments are run on a Google Cloud **n1-standard-8** machine with one NVIDIA V100 GPU. Runtimes assume that the all code required for the GPU acceleration is precompiled. This is possible for the gradients needed for hessian vector products, gradients for each training example, and for streaming queries, because they are known in advance. In general however, query gradients depend on the user complaint and add additional overhead (8sec on

Q_1	SELECT COUNT(*) FROM LEFT L, RIGHT R WHERE predict(L) = predict(R)
Q_2	SELECT COUNT(*) FROM D WHERE predict(*)= IN {class-list}
Q_3	SELECT COUNT(*) FROM D WHERE predict(*)= {class}
Q_4	SELECT AVG(predict(*)) FROM D WHERE workclass = "Private" AND gender = "Male"
Q_5	SELECT AVG(predict(*)) FROM D WHERE workclass = "Private"
Q_6	SELECT AVG(predict(*)) FROM D WHERE gender = "Male"
Q_7	SELECT AVG(predict(*)) FROM D

Table 5.1: Summary of query templates used in the experiments.

Dataset	**Corruption**	**Rate**
MNIST	$1 \rightarrow 7$, $1 \rightarrow 7 \wedge 4 \rightarrow 9$	10 - 40%
Fashion-MNIST	pants $\rightarrow$ sneakers	10 - 40%
CIFAR-10	automobile $\rightarrow$ horse	10 - 40%
SST2	negative $\rightarrow$ positive	10 - 40%
ADULT	<\$50K $\rightarrow$ $\geq$\$50K	10 - 40%

Table 5.2: Summary of label corruptions.

unoptimized code)—deeper integration between Rain++ and NN compilers like XLA to reduce this overhead is promising for future work.

5.5.2 Effects of small eigenvalues

Our first experiments illustrate how small eigenvalues of H_{θ^*} degrade influence analysis. We use the join-count query Q_1 on MNIST, where digits $0-4$ (LEFT) are joined with digits $5-9$ (RIGHT). The ground truth query should return 0. We evaluate Logistic Regression and CNN models over class conditional label noise. The compaint specifies that the output should be lower.

Figure 5.3 shows how the number of CG iterations (for Rain, solid lines) and eigenvalues (for Rain++, dashed lines) affect debugging quality; line colors depict corruption rate. The location of the peak matches the eigenvalue spectra in Figure 5.2(left), where the normalized eigenvalues with respect to the maximum eigenvalue is near-zero after 10 eigenvalues. Rain converges to its peak more quickly because CG solves for any orthogonal vectors to minimize the objective function, whereas Lanczos used in Rain++ is restricted to eigenvectors of the Hessian. Increasing the number of iterations/eigenvalues beyond the peak ultimately degrades AUC_{CR} due to numerical instability, as small eigenvalues dominate the gradient analysis. As the number of iterations/eigenvalues

Error Type	Affected Subsets	Rate
Gauss. Blur $\sigma = 2$	1 (MNIST), pants (Fashion)	10 - 100%
Salt & Pepper 30%	auto (CIFAR)	70 - 100%
Gauss. Blur $\sigma = 2$	All classes (MNIST, Fashion, CIFAR)	10 - 40%
Salt & Pepper 30%	All classes (MNIST, Fashion, CIFAR)	10 - 40%
<\$50k $\rightarrow$ $\geq$\$50k	male $\wedge$ private-sector (ADULT)	50-70%

Table 5.3: Summary of feature and feature conditional corruptions.

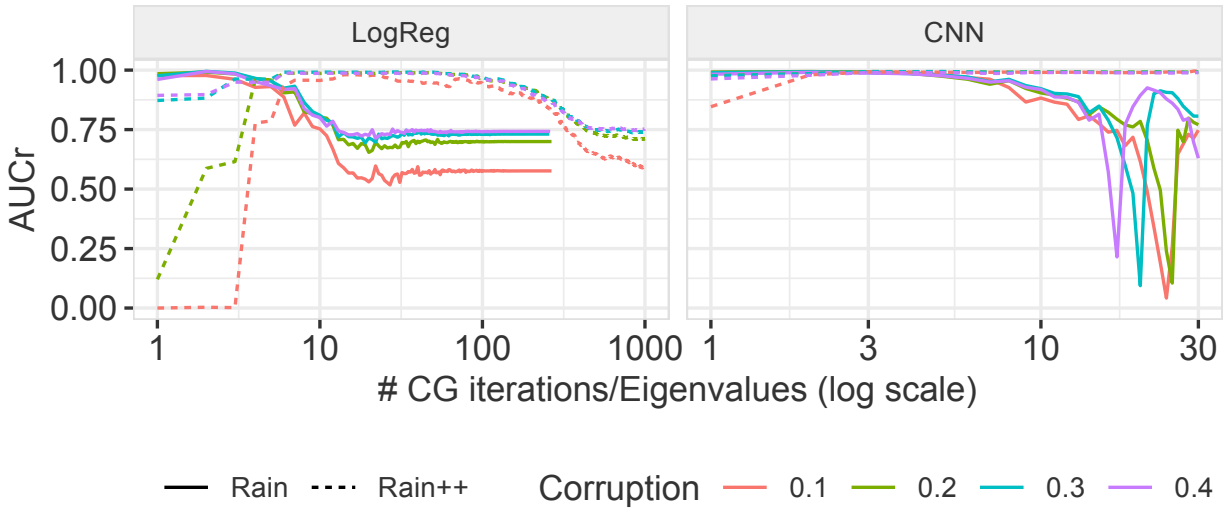


Figure 5.3: Q_1 AUC$_R$ varying the number of CG iterations and eigenvectors for corruption rates 0.1-0.4.

converges to the total number of model parameters d, we expect both approaches to be equivalent.

Non-convex models such as the CNN are typically trained to reach an approximate local minima because it is faster to compute and reduces the risk of overfitting. As a result, the Hessian can potentially have small negative eigenvalues whose eigenvectors correspond to directions that increase the loss. This is why Rain's AUC_{CR} fluctuates widely beyond the peak. In contrast, Rain++ uses positive eigenvalues, and does not suffer from this instability.

Takeaway: Small and negative eigenvalues of the Hessian degrade AUC_{CR}. Rain++ avoids these issues by only using the top k eigenvalues.

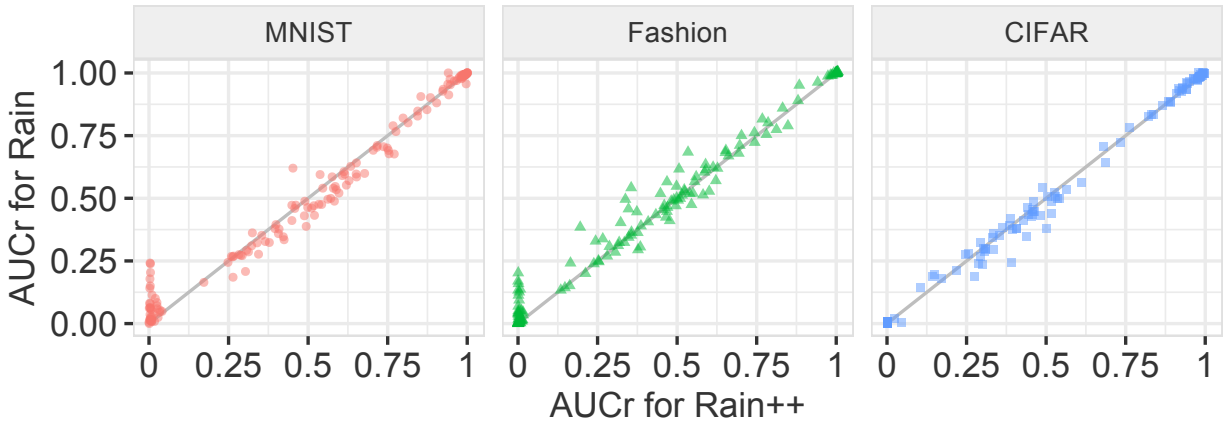


Figure 5.4: Peak AUC_R Rain vs Rain++

5.5.3 Baseline Comparison: Debugging Quality

Figure 5.4 compares the peak AUC_{CR} for Rain and Rain++ across all models, image datasets, and queries Q_1 and Q_3. We use all corruptions and rates of Tables 5.2 and 5.3 except for the multiple corruption cases of **MNIST** and the feature conditional ones of **ADULT** which will be studied in more detail in Section 5.5.8. For Q_1, the join condition is over two disjoint subsets of the inference dataset, so the aggregation is expected to be 0. For Q_3, we filter on '1' digit, pants, and automobiles for the three datasets. We vary the number of CG iterations/eigenvalues and report peak AUC_{CR}. Each point compares the peak AUC_{CR} for both approaches.

The vast majority of points are near the gray $y = x$ line, and shows that the debugging qualities are comparable. Additionally, the peak AUC_{CR} for both approaches is interspersed across $[0, 1]$ indicating that not all complaints are effective for all settings. We study the conditions when a complaint can be expected to be effective for debugging in Sections 5.5.8 and 5.5.9. We find that the relative error in the query output and the choice of intervention are key factors in complaint effectiveness.

Takeaway: Rain and Rain++ report comparable peak AUC_{CR}.

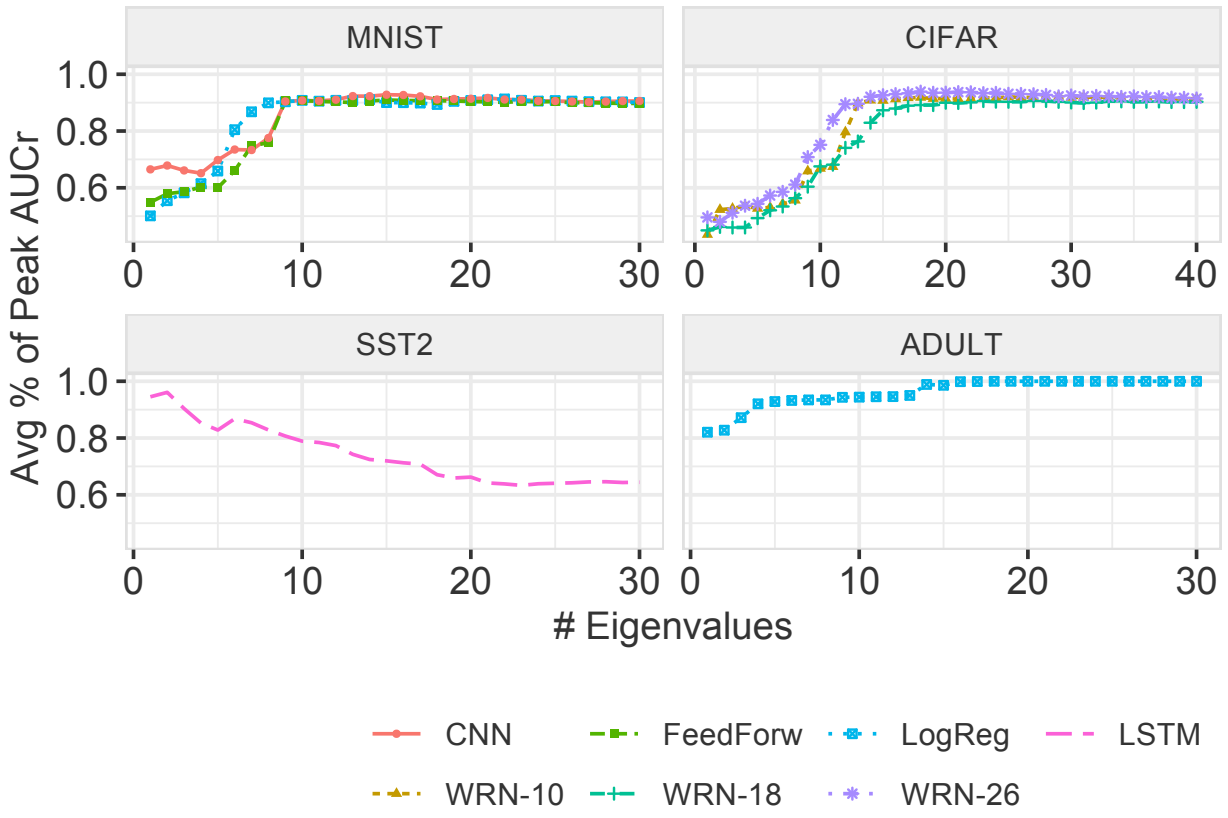


Figure 5.5: Percentage of peak AUC_R for varying eigenvalues, averaged over all corruption types and rates.

5.5.4 Number of Eigenvalues

Figure 5.5 focuses on Rain++ and studies how AUC_{CR} varies with the number of eigenvalues used k. We report the percentage of the peak AUC_{CR}, averaged over all corruptions and queries Q_1 and Q_3. We exclude results when Rain++ is ineffective (peak $AUC_{CR} \leq 0.1$) but the results do not change if they are included; for **SST2**, we report results for Q_3 and for the **ADULT** on Q_7.

Across different models for each dataset, the best choice for k does not change significantly. Furthermore, our heuristic for choosing k in Section 5.3.3 would select 10, 13, 2 and 6 for the four datasets, which are near optimal. Interestingly, even though the LSTM model for SST2 has the most parameters (3.4M), only two eigenvalues are needed for complaint-based debugging.

Takeaway: Number of eigenvalues for peak AUC_R is empirically robust to model size for the

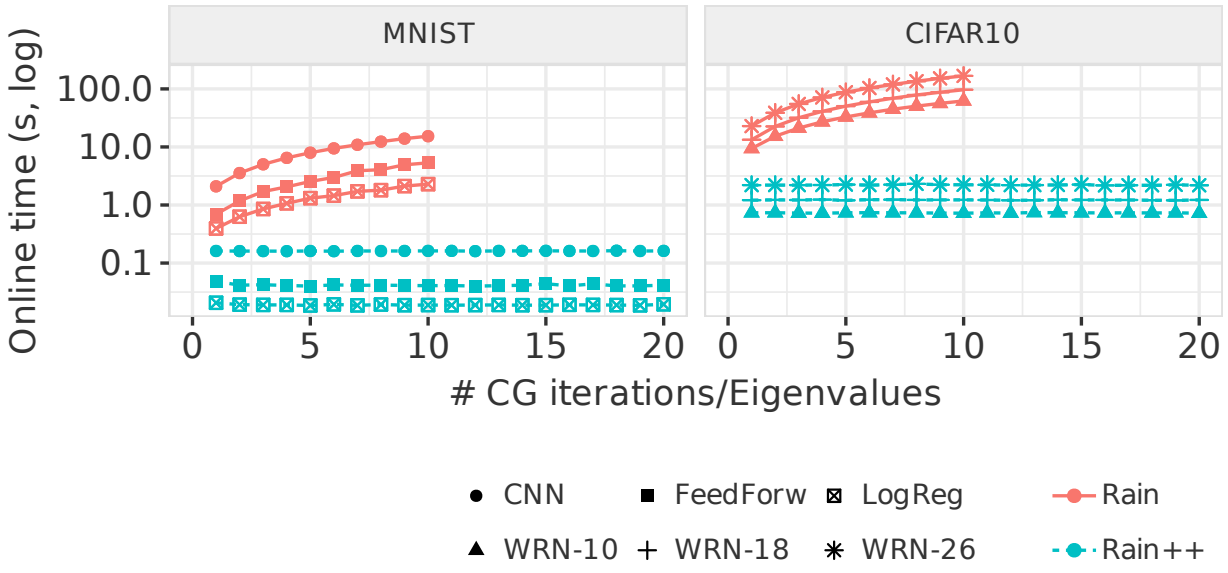


Figure 5.6: Online complexity varying eigenvalues.

same dataset, and is close to the number of classes even for overparametrized models.

5.5.5 Baseline Comparison: Online Runtime

Figure 5.6 reports the end-to-end online runtimes to compute influence scores for all training examples in the MNIST and CIFAR10 datasets, for a Q_1 complaint. We run Rain, and run Rain++ without the query-gradient optimizations in Section 7.4. We use a single corruption setting, since it does not affect runtime performance.

Rain++ reduces runtimes by over an order of magnitude even when compared to a single CG iteration. Interestingly, runtimes for **CIFAR-10** are longer than for **MNIST** despite fewer model parameters. CNNs and WRNs reuse the parameters for many operations in their convolutional layers and thus their gradient computations is more costly. Further, sequential layer operations in deeper models like WRNs are more expensive because they are not parallelizable.

In this section we will compare the online time required by Rain and Rain++ to return the scores for all training set interventions given a complaint on the query output . This includes computing $\nabla_\theta q(\theta^*)$ and using it as a part of the influence calculations for Rain and Rain++. Here we will focus

		Rain-only			
Model	$\nabla_\theta q(\theta^*)$	**CG iter**	**Rain score**	**Rain**	**Rain++**
LogReg	0.02	0.23	0.16	0.61	0.02
CNN	0.16	1.46	0.48	3.50	0.16
FeedForw	0.04	0.46	0.16	1.14	0.04
WRN-10	0.73	5.90	2.92	26.91	0.73
WRN-18	1.21	9.26	2.91	41.02	1.22
WRN-26	2.20	15.86	4.63	71.08	2.20

Table 5.4: Breakdown of online runtime (sec) for Rain and Rain++ ($\nabla_\theta q(\theta^*)$ is shared). Rows $1-3$ are for MNIST, $4-6$ for CIFAR-10.

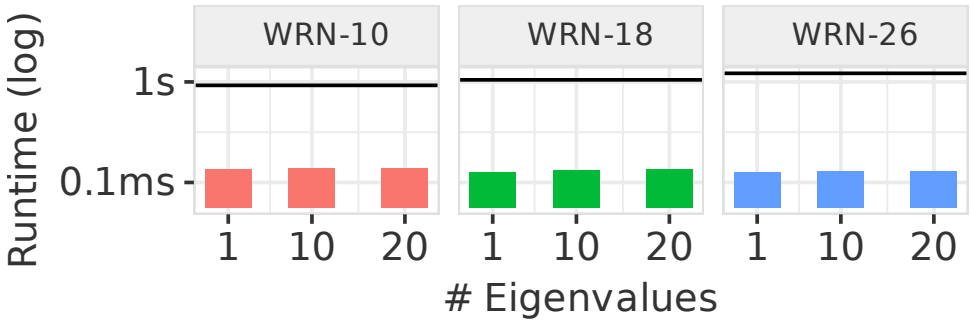


Figure 5.7: Query gradient optimization for **CIFAR-10**. Horizontal line is unoptimized time.

on the performance improvements of Rain++ without the use of the optimizations of Section 7.4.

Table 5.4 breaks down the runtimes into individual steps. Computing $\nabla_\theta q(\theta^*)$ is common to both approaches, however Rain must also computer CG, multiply $\nabla_\theta q(\theta^*) H_{\theta^*}^{-1}$ with each $\nabla_\theta \ell(\theta^*, z)$ to compute each training example's score (Rain score). We report Rain end-to-end time for 2 CG iterations for **MNIST** and 4 iterations for **CIFAR-10**, which typically achieves close to peak AUC_R in our experiments, and Rain++ using 20 eigenvalues. CG is the bottleneck for Rain, whereas computing query gradients ($\nabla_\theta q(\theta^*)$) is the bottleneck for Rain++ on complex models like WRNs. We will evaluate query gradient optimizations next.

Takeaway: Rain++ reduces runtimes by orders of magnitude, but is bottlenecked by computing the query gradient $\nabla_\theta q(\theta^*)$.

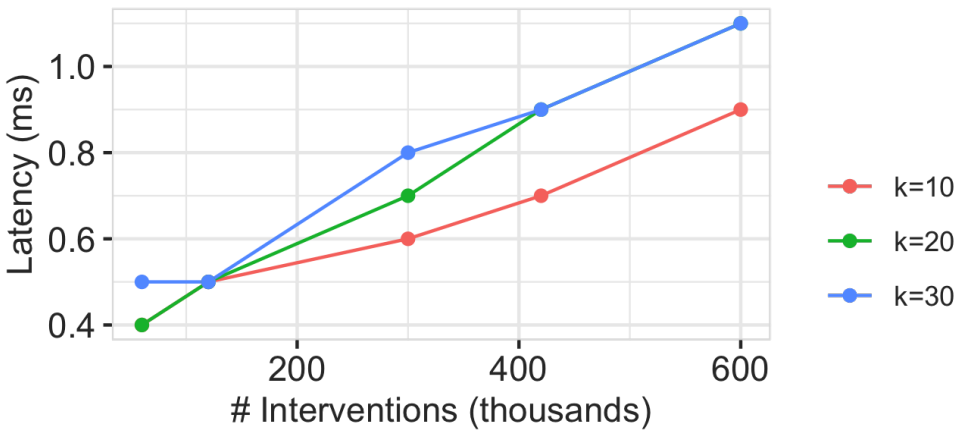


Figure 5.8: End-to-end debugging time using gradient query optimization for varying k and # of interventions I, on **MNIST** dataset.

5.5.6 Query Gradient Optimizations

Figure 5.7 reports the runtime optimization benefits when the inference database is known apriori (Section 5.4.1). We focus on **CIFAR-10** since its results are representative. The horizontal line corresponds to the unoptimized cost to compute the query gradient $\nabla_\theta q(\theta^*)$. Computing the gradient for each test example dominates the query gradient runtime, so precomputing them reduces the runtime by over an order of magnitude, and in effect, eliminates the computational bottleneck. As a result, Rain++ can compute influence scores for all experimental settings in interactive time. **For WRN-26 on CIFAR-10, our suite of optimizations reduces the end-to-end debugging time from over 1.18 minutes using Rain, to less than** 1**ms using Rain++: a** 70000× **reduction.**

Multiple Interventions: What if there are many ways to clean a training record? Multiple interventions may slow down complaint debugging. We now create 10 interventions for each **MNIST** training record—delete the record or change the label to one of the other 9 classes—for a pool of 600K potential interventions. Using the previous settings, Figure 5.8 reports end-to-end debugging latency as the number of potential interventions I increases, for different numbers of eigenvectors k. Thanks to precomputation, latency is independent of d, so we report results for a Logistic Regression model. The maximum latency is still $\approx 1ms$, and translates to **evaluating** ∼28**M interventions per second**.

Streaming: Figure 5.9 reports the incremental maintenance cost of ΔQ_2 is run over a streaming database that updates in varying update sizes. We set $k = 20$. This is akin to the fashion monitoring use case described in the introduction. We see that the incremental update cost varies with the update size and is independent of the test database size. In fact, updates sizes of up to one thousand records can update in under 500ms because the gradient deltas can be computed on the GPU in one batch. Larger update sizes must be split and run on the GPU in serial order. Smaller update sizes underutilize the GPU, which is why the curve is flat.

Takeaway: the query gradient optimizations leverage the known inference database or query to ensure interactive debugging times.

5.5.7 Offline Precomputation Time

Figure 5.10 reports the offline costs to precompute the gradients for the training set (Section 7.3) as well as the query gradients when the inference database is known (Section 5.4.1). We vary the number of eigenvalues to precompute, and mark $k = 10$ with a vertical line. The overall costs are quite reasonable—for instance, at $k = 20$, it takes 20 minutes to precompute gradients for the 26 layer WRN model. This corresponds to the time for Rain to answer 15 Q_1 complaints using 2 CG iterations (see Table 5.4).

Takeaway: Offline preprocessing times are comparable to running Rain for a dozen complaints.

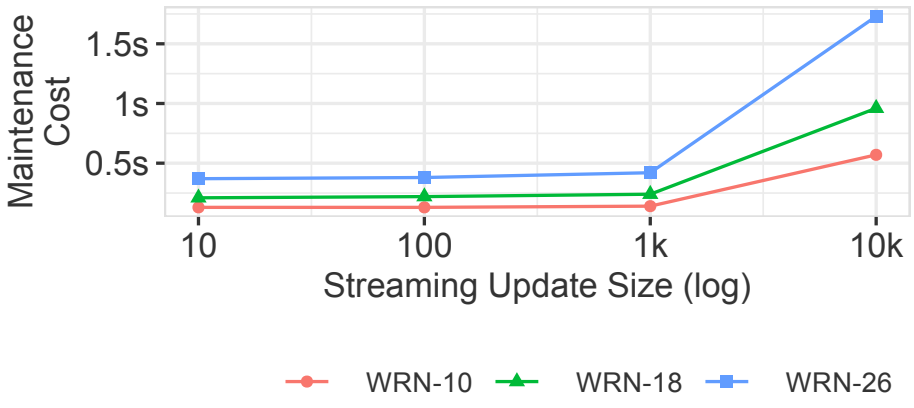


Figure 5.9: Maintenance cost for varying stream update sizes.

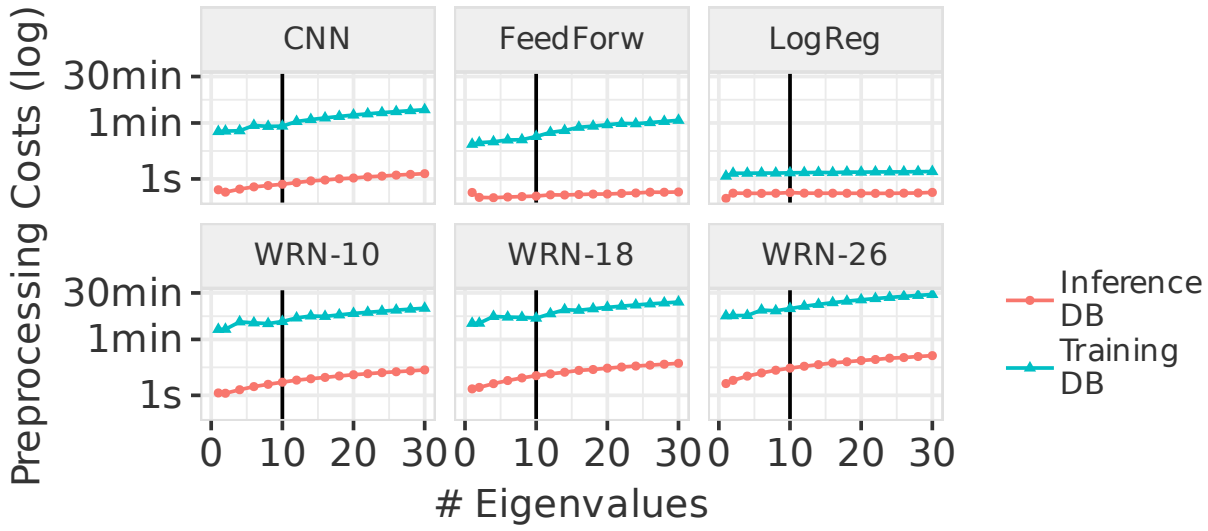


Figure 5.10: Offline precomputation costs.

We believe it is reasonable enough to perform as a preprocessing step before releasing a model.

5.5.8 When Are Complaints Useful?

Section 5.5.3 showed that correctly expressed complaints can still be ineffective at training set debugging. In response, we seek to understand the properties of a query complaint that affect AUC_{CR}. Intuitively, we should expect that it depends on the relationship between the corruption and how it affects the query. At the extreme, if the training corruptions only cause an $\epsilon \approx 0$ to the query's result value, then we should not expect it to be effective.

Initial Point Complaint Analysis

Our analysis will be based on Equation (5.2), which decomposes $q(\theta)$ into a linear combination of individual prediction probabilities $p_{ij}(\theta)$ for each inference record i and class j. We can view p_{ij} as a *point complaint* that record i should have label j. Intuitively a point complaint is likely to be effective for debugging if the model mispredicts i *and* if removing the training errors would lead to a correct prediction. To check this intuition, we add class conditional label noise in **CIFAR-10** (40% rate), and point complaints where the corrupted model prediction differs from the clean model's

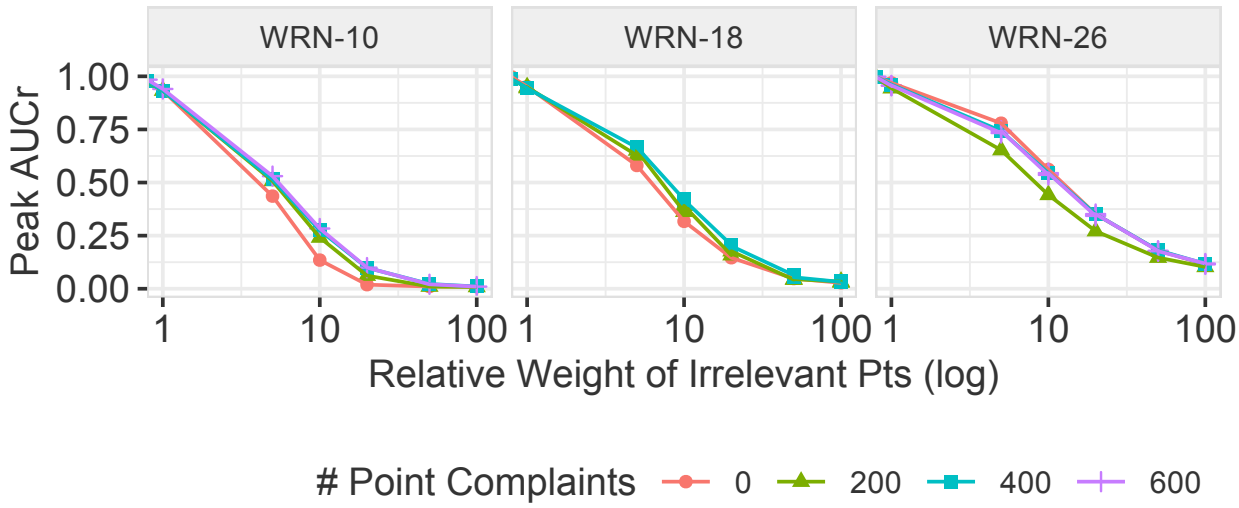


Figure 5.11: Effects of varying the number of relevant point complaints, and the overall weight of irrelevant point complaints.

prediction. These complaints indeed have a high peak AUC_{CR} of 86%, agreeing with our intuition.

Relevant and Adversarial Point Complaints

We now use point complaints to study query complaints. Query complaints are modeled as linear combinations of point complaints that differ in terms of their weights $\partial f(P(\theta^*))/\partial p_{ij}$. We expect that a query complaint that assigns high weights to the "relevant" point complaints and low weights to the "adversarial" point complaints should be effective at debugging training errors.

Our experiment varies the weights that we assign to relevant and adversarial point complaints. This is akin to a SUM aggregation with a predicate where tuples that satisfy the predicate increase the sum by an amount based on their attributes. Join aggregations have this property as well. We define relevant complaints test points as those whose true label is *automobile*, are mislabeled by the model but are correctly predicted when the training errors are removed. We define adversarial point complaints as test examples whose true predicted labels are *horse*. The two types of point complaints push the model in different directions. We use an equal number of relevant and adversarial points, but give adversarial points $Y\times$ the weight as relevant points, where $Y \in [0, 100]$.

Figure 5.11 shows that debugging quality is insensitive to the number of point complaints,

but is highly sensitive to the *ratio* of weights. When the ratio is $\leq 5\times$, Rain++ remains effective, however the quality quickly degrades. This suggests that query complaints are most effective when adversarial complaints do not dominate. Note that an irrelevant point complaint p_{ij}—for instance, a test point predicted with high confidence as *bird* for a query that filters on *horse*—has negligible effects on debugging quality since their contribution to the query gradient is 0.

Multiple Errors

We now show when complaints can identify multiple error types at once. To this end, we use the **MNIST** dataset and CNN model, and introduce two sets of label flips—we flip 10%-40% of all 1s to 7s and all 4s to 9s. We use the queries $Q_3 - X$ and $Q_2 - Y$, which respectively count the number of predictions of class X and within a set of classes Y, where $X \in \{1, 4, 7, 9\}$ and $Y = (1, 4)$ and $(7, 9)$. All Q_3 complaints have $\text{AUC}_R \in [.69 - .80]$ since the complaints only target half of the relevant errors. In contrast, Q_2 complaints are $\sim$1. We find that Rain++ identifies errors relevant to the complaint, irrespective of the number of error types.

Feature Conditional Errors

So far, we have studied class-conditional errors, however errors can be concentrated in feature space as well. We now evaluate feature-conditional label errors using the **ADULT** dataset—for males working in the private sector (`gender=male ∧ workclass=private-sector`), we flip 50%-70% of <\$50K labels to ≥\$50K. We use complaints that compute the proportion of $\geq$ \$50K predictions for increasingly less precise subsets of the inference dataset: men in private sector (Q_4), private sector only (Q_5), all men (Q_6), and all records (Q_7). Figure 5.12(right) shows that overly general complaints (Q_6, Q_7) are not very useful, whereas more precise predicates are very effective (Q_4, Q_5), irrespective of the degree of corruption. Figure 5.12(left) shows that the AUC tends to be high when the relative difference between the corrupted query result and the clean result is large. From a practical standpoint, this is a promising result, as it connects debugging effectiveness with the degree that training data errors actually affect query results in undesirable

Rate	Q_3-1	Q_3-4	Q_3-7	Q_3-9	Q_2-(1,4)	Q_2-(7,9)
10%	0.69	0.70	0.78	0.70	0.98	0.98
20%	0.69	0.73	0.78	0.70	0.99	0.99
30%	0.80	0.72	0.78	0.70	0.99	0.99
40%	0.85	0.72	0.80	0.70	0.98	0.98

Table 5.5: Peak Rain++ AUC_R varying complaints for multiple label corruptions of varying rates on **MNIST** CNN.

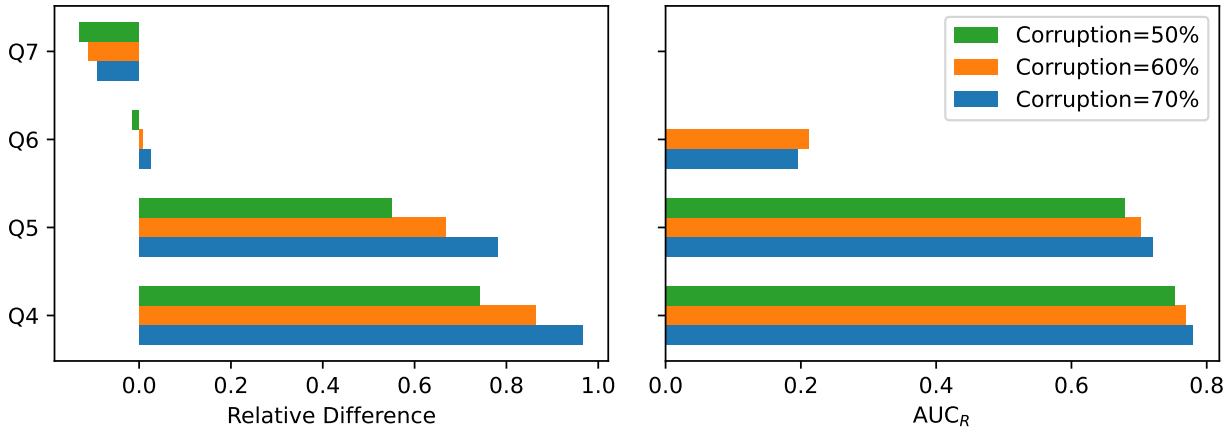


Figure 5.12: Relative difference of query output and peak Rain++ AUC_R for **ADULT**'s feature conditional label noise.

ways that manifest in the downstream application. We explicitly study this connection next.

Magnitude of Query Errors

Although the above analysis sheds light on when Rain++ can be effective, it relies on apriori knowledge of relevant and irrelevant point complaints. However, assuming this puts the cart before the horse, as the user only has visibility of the query results. Thus, this experiment studies the relationship between the magnitude of the query's output error wrt the correct query output, and debugging effectiveness. The intuition is that larger query errors may be more likely to be due to training example corruptions, rather than for spurrious reasons.

We use all models, the **CIFAR-10** and **MNIST** datasets, and vary the rate of class conditional label noise from 10% to 40%. We sweep the possible queries that can be generated using Q_1 and Q_3 templates. For Q_1, we set the left and right sides of the join to subsets of the test database with

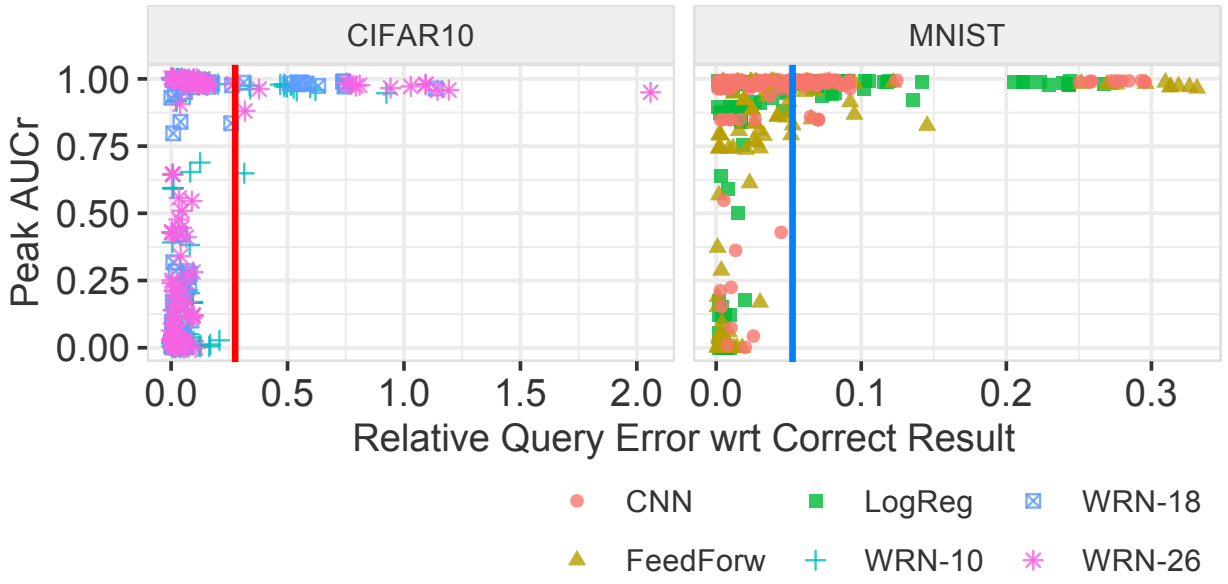


Figure 5.13: Relationship between the query output's relative error and debugging quality. Vertical lines at 25% (left) and 5% (right).

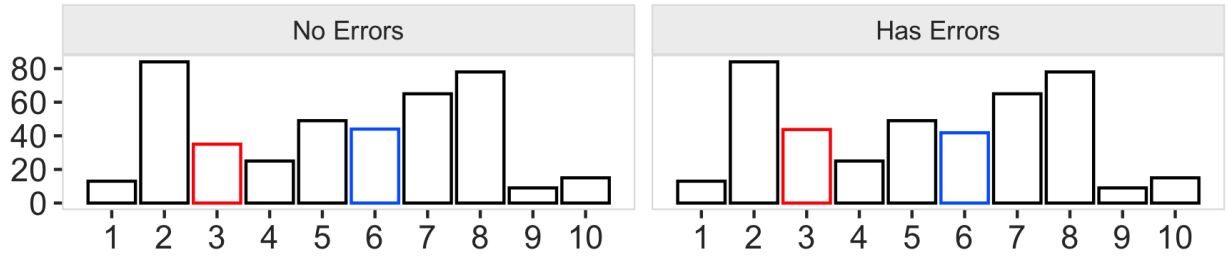


Figure 5.14: Small relative errors are difficult to visually detect. The height of bars 3 and 6 are changed by 25% and 5%, respectively.

different true labels (e.g., LEFT is digit '5', RIGHT is digit '9' for **MNIST**). We use this procedure to generate 20 random query complaints. For Q_3, we vary the filter condition over all 10 classes for each dataset, resulting in 10 complaints per dataset. This results in 300 complaints per dataset.

Figure 5.13 shows that, irrespective of the model architecture, the peak AUC_{CR} improves as the relative query error increases. When the query result increases beyond a threshold (25% in red, 5% in blue vertical lines), the peak AUC_{CR} tends to be near-1. This is an encouraging result, because small relative differences are difficult to see [103, 104, 105], and users are most likely to submit complaints when errors are noticable. Figure 5.14 illustrates that it is difficult to tell, even side by side, that the heights of bars 3 and 6 have been respectively changed by 25% and 5%.

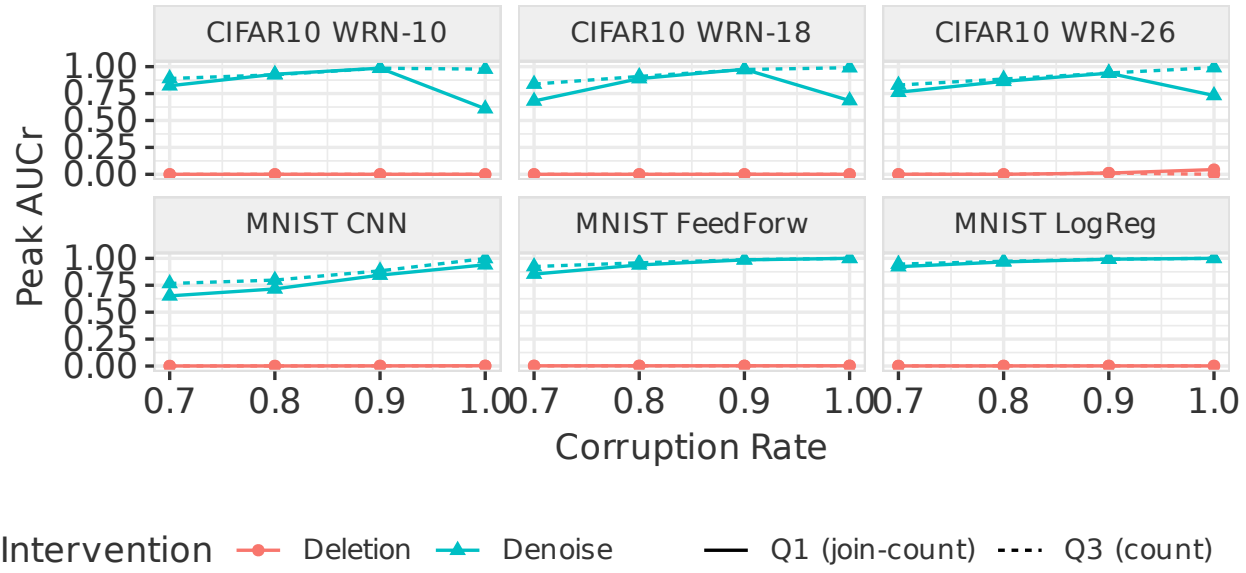


Figure 5.15: Deletion is an ineffective intervention for addressing Salt & Pepper noise; denoising via median filter is effective.

Among the corruptions that did not affect the query results, we found cases where heavy corruptions did not have a significant effect on model accuracy. For example, Salt & Pepper noise on even 50% of 1 digits of **MNIST** reduced test set accuracy by less than 1%. This strongly indicates that data debugging approaches that are unaware of how the model is used downstream may spend significant amounts of time cleaning training examples that do not end up affecting model accuracy. The complaint driven approach of Rain and Rain++ clearly avoids that.

Takeaway: Debugging quality is related to the weights of relevant point complaints as compared to adversarial point complaints. Further, larger query output differences that are more likely to be detected downstream directly correlate with higher peak AUC_{CR}*. Complaint driven debugging can reduce debugging effort by prioritizing training set errors (even of different types) that affect model predictions in a way that manifests downstream.*

5.5.9 Intervention Effectiveness

So far we have assumed that deleting the corrupted training examples is sufficient to resolve the user's complaint. However, this is not always the case. For example, when we corrupt 90%

of the automobile training examples with Salt & Pepper on **CIFAR-10**, WRN-26 predicts 847 automobiles in the test database when the true count is 1000. Deleting these corrupted training examples will simply worsen the query output. If deleting corrupted training examples does not resolve the complaint, then Rain and Rain++ will not rank those examples highly. Using poor interventions affects the debugging effectiveness of influence analysis.

To illustrate this, we corrupt **CIFAR-10** and **MNIST** training sets with Salt & Pepper, and evaluate Rain++ using the deletion intervention that we have used so far, and a denoise intervention. The latter assigns each pixel the median value of its neighboring pixels; this is effective for Salt & Pepper noise. We run Q_1 and Q_3 using their default configurations. Figure 5.15 shows that across all models, datasets, and queries, the deletion intervention is completely ineffective. Although the query complaint specifies that the query result should be higher, deleting the corrupted training examples actually *reduces* the query results further. In contrast, denoise has a peak AUC_{CR} consistently above 0.65 and converges to 1 as the corruption rate (of the corrupted class) increases to 100%.

Takeaway: Effective training example debugging relies on using the appropriate intervention, and deletion is not always the most effective. Further studies are needed to better understand the interaction between data corruption, interventions, and complaints.

5.5.10 Application to Interactive Debugging Interfaces

Figure 5.16 presents an example complaint driven training data debugging application that relies on Rain++ to provide responses at interactive time scales. Users run ML inference queries and visualize the result. If they identify any anomalies in the model's predictions, they can specify a complaint using the interface. Then Rain++ ranks the interventions accordingly.

We now describe the specific setting of Figure 5.16. In advance of user interaction, the ML model is trained and its Rain++ data structures are computed. Here we train an **MNIST** Logistic Regression model on a dataset where 60% of '1' labels are randomly flipped to a '7'. At interaction time the user specifies a query in the text box and runs it. In our case the user chooses to visualize the predicted class frequencies with the result shown in the bar chart. As expected based on the

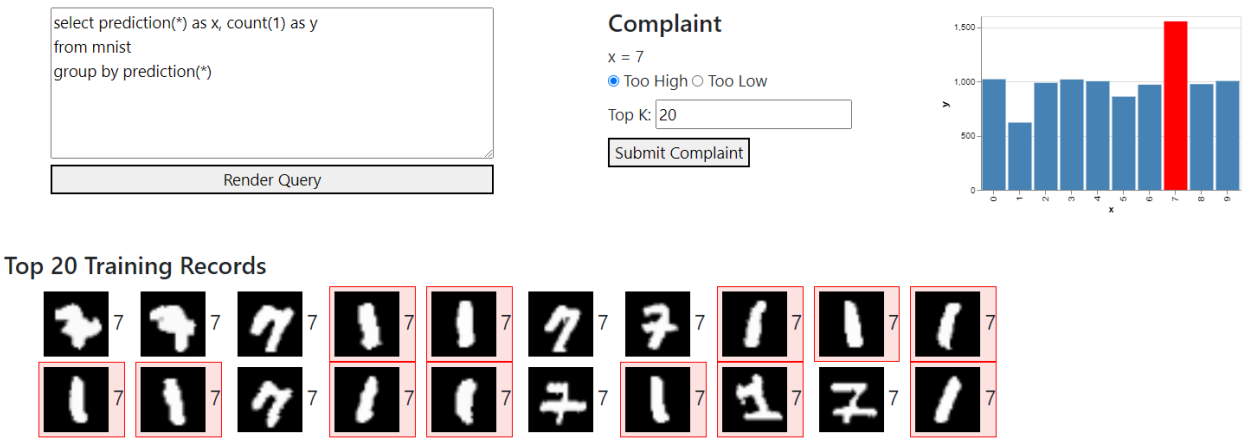


Figure 5.16: Interactive complaint driven training data debugging application powered by Rain++.

corruption, although the true labels of the test set is spread uniformly across classes the predicted counts of '1' is lower and those of '7' are higher than the others. The user, being interested in the anomaly of class '7' highlights the corresponding bar, which is shown in red in the figure. Using the radial button she picks the direction of the complaint, namely that the count is too high.

Rain++ ranks the deletion interventions, each represented by the corresponding image and its given label. We can visually confirm that all the top 20 examples ranked are either of class '1' or '7'. In fact more than half of the examples have corrupted labels. We annotate them here with a red background for the purposes of the example but the true labels and thus the annotation are not available in practice. Even among true '7' examples, some of them could be confused for being of class '1' especially by a classifier trained on the corrupted training set at hand. Thus the ranking of Rain++ agrees with our intuition about which examples are influential for the given complaint.

Thanks to Rain++ the user gets the ranking interactively without disturbing her analysis flow. In addition, neither running the query nor ranking the interventions requires ML model inference or access to training infrastructure, making deployment of this application very convenient.

Chapter 6: Learning to Debug Training Data Using Complaints

In Chapter 4 and Chapter 5 we focused our attention to complaint driven debugging approaches that rank per training example interventions. Unfortunately this comes with a few disadvantages.

The first disadvantage is that Rain and Rain++ rank interventions at the top only if they are estimated to be individually effective at addressing the user's complaint. A group of interventions that are jointly effective at addressing the user's complaint but not each one individually may be overlooked. This is important because in large training sets many per training example interventions may need to be applied to resolve the user's complaint. The second disadvantage is that Rain and Rain++ do not take into account that errors in the training data are typically systematic and thus form patterns. Let us revisit the example of CompanyX from Chapter 1, a company specializing in email campaign management for its enterprise customers. A change in one of CompanyX's customers internal process, e.g. a new check-out step, may result in systematically incorrect training data for this client. A complaint driven debugging system could make use of the systematic nature of errors to prioritize groups of similar interventions over interventions that do not follow a pattern. The case of systematic errors is particularly important. ML models may be robust to training data errors that are isolated or random making a user complaint less likely.

The third disadvantage is that returning rankings of per-training example interventions may not always be actionable. If the errors in the training data follow patterns as discussed above, it may be more convenient to return the error pattern and the corresponding intervention instead of listing all the per training example interventions. This is especially true when training data is a result of a pipeline itself and fixing the training data in place is not an admissible solution. For example, if CompanyX is facing an integration issue with one its clients, simply fixing the affected training examples is not a permanent solution because new errors will be continuously be introduced until the integration issue is resolved. Knowing that all errors are associated with a single customer

may tip-off CompanyX's engineers that they are facing an issue in their data integration pipeline. Thus knowing the error pattern may be advantageous over a ranked list of per training example interventions to verify. In this section, we frame finding the pattern of training set errors in response to complaints as a classification problem. Just like with Rain and Rain++, the input is a complaint and a per training example intervention like deleting the example. Instead of returning a ranked list of training examples to delete, the debugging system now returns a new model, which we term the denoising model. For each training example, the denosing model decides whether to apply the given intervention or not. The goal of the system is return a denoising model that best addresses the user's complaint. We note that the denoising model is different from the model getting debugged, which for simplicity we term the target model.

Unfortunately, Rain and Rain++ cannot be trivially extended to solve this version of the complaint driven debugging problem. The first reason is that the local sensitivity approximations that Rain and Rain++ employ to approximate the effects of training example interventions are not accurate for large modifications of training set. The second reason is that Rain and Rain++ rely on enumerating and evaluating all the available interventions. Enumerating the space of all groups of training example interventions takes exponential time in the training set size, which is prohibitive.

Prior work Gopher [106] and BOExplain [33] tackle this challenge by constraining the denoising model to be a conjunctive predicate of the training example features. This not only reduces the combinations to enumerate but it also leads to more succinct and explainable results reported to end users. To accelerate searching for the optimal predicate, Gopher employs additional heuristics to prune conjunctive patterns that are not influential for the user's complaint whereas BOExplain formulates this a Bayesian Optimization problem. Both Gopher and BOExplain need to evaluate the effect of a candidate denoising model to the user's complaint. Gopher uses a second order influence function approach and BOExplain retrains the target model and evaluates the complaint.

The class of conjunctive predicates is not expressive enough to capture errors in domains like computer vision and natural language processing. For these domains, one may need more complex models to detect the training data errors like linear models or even neural networks. The enumeration

approach of Gopher and the Bayesian Optimization one of BOExplain do not scale to problems of thousands ore more parameters needed to train this type of models.

Our key observation in this section is that just like the non-differentiability of SQL precluded us from directly using influence analysis for Rain, it is the non-differentiability of conjunctive predicates that forces Gopher and BOExplain to use black box approaches when searching for the optimal denoising predicate. Both approaches are black box meaning that for each candidate denoising model they only compute its effect on the complaint without measuring the sensitivity of the complaint to the denoising model parameters.

We leverage this observation to pose training the denoising model as a differentiable nested optimization problem. By nested optimization problem we mean here that the objective of the denoising model contains references to solutions of other optimization problems. In our case this is the optimal target model parameters when optimized on the training set as modified by the denoising model. To ensure end to end differentiability of the denoising model's objective we use the same complaint relaxation techniques as in Rain and Rain++.

Framing training set debugging as a nested optimization problem is not new [68, 13, 69]. However, our work differs from prior art in two ways. First, prior art does not consider general complaints as an input to the debugging process but instead assumes a labelled validation set. Ambiguous complaints pose unique difficulties because the denoising model may overfit the complaint. Second, they either do not leverage error patterns [13, 68] similar to Rain and Rain++ or consider very simple denoising models that use the target model's loss as the only feature [69].

We contribute MetaRain, a nested optimization framework for training denoising models based on user complaints. The advantage of MetaRain is that it is very general. It can natively train differentiable denoising models like logistic regression models and neural networks. It can also be used to train denoising models that are not naturally differentiable, like binary conjunctions, using a differentiable relaxation. The key challenge in using MetaRain is designing denoising model architectures that are both capable of addressing the user's complaints without overfitting to them. In response, we experimentally study the tradeoffs between model architecture and regularization

choices and complaint overfitting to provide concrete recommendations to users.

6.1 Use Cases

MetaRain aims to return a denoising model that captures the error patterns in the training data. Here we outline 3 concrete applications where returning the denoising model is advantageous over the ranked training example list approach of Rain and Rain++.

Filtering noisy crowd-sourced labels George, an, ML Engineer at a e-commerce clothing retailer, is tasked with creating a model that classifies images based on their clothing type. He turns to crowd sourcing to create an initial small training dataset. He then trains a model and applies it on a sample of production data. Observing that the count of socks predicted is too high, he is concerned that some of the training images might be mislabeled. Just finding the errors in the initial sample is not a satisfying solution because as new training data become available the process must be repeated. Ideally George would be able to train a second model that filters noisy training examples as they become available through crowd-sourcing. George can use MetaRain to train such a model using only his complaint on socks as supervision.

The following use case was first explored by Gopher [106].

Explaining model unfairness Mary, a data analyst at a bank, is tasked with creating a model predicting whether a client will default on a loan. Unfortunately, Mary observes that the the statistical parity gap between men and women is too large, meaning that the model is unfair to women. In order to understand why the model behaves this way, she wants to identify small and interpretable subsets of training data, which if removed, can significantly reduce the statistical parity gap. Mary can use MetaRain and her complaint on the statistical parity gap to find conjunctive predicates whose deletion best addresses the complaint.

Debugging weak supervision rules Taylor, a data scientist working for a review aggregation website, is tasked with creating a sentiment classification model for restaurant reviews. Lacking a budget for manual labels, Taylor devises weak supervision rules in order to train a model on the unlabelled data available. When Taylor applies the model on some held out data, she finds that the

positive sentiment reviews are surprisingly high for reviews having two stars or less. She suspects that some of the rules that she provided might have low precision. Taylor can use MetaRain and her complaint on the low star reviews to supervise a denoising model that decides whether to remove a vote of a weak labelling rule or not. For rules whose votes are frequently removed, Taylor can try to refine them or even remove them and retrain the target model.

6.2 A Relaxation Approach to Pattern Aware Debugging

In this chapter we seek to solve Problem 3, the pattern aware version of the complaint driven training data debugging problem. Given a family of allowable training example deletion patterns we seek to identify the member of the family that does the minimum amount of deletions while addressing the user's complaint. If we are only allowed to enumerate the members of the pattern family in a black box fashion, then it is impossible to improve upon a brute force solution that enumerates all the candidate patterns. This is completely analogous to the problem we would have if we wanted to support debugging of black box models and black box workflows in general. To sidestep this issue, we seek to once again construct a differentiable relaxation of Problem 3.

Problem 3 is very similar to Equation (3.9), the basis of several training data debugging approaches based on nested optimization [13, 68, 69]. These approaches frame ML debugging as the problem of adjusting the weights of training examples so that the trained model has low validation loss in a distribution of interest. Similarly, in Problem 3 we want to set the training example weights to either one or zero, corresponding to a training example that remains or gets deleted respectively, so as to resolve the user's complaint.

There are two key points where Problem 3 breaks end to end differentiability. The first one is the user complaint, which can be directly handled similar to Rain and Rain++. The second one is that deletions over the training set are constrained to be generated by one of the members of the input family of patterns of Problem 3. In general, the hard decissions of the deletion pattern are not differentiable with respect to the pattern parameters ρ.

The relaxation approach now becomes straightforward. We restrict our attention to patterns

based on differentiable denoising classification models. The denoising model assigns to each training example z_i a probability of being deleted $\epsilon_i = w(\rho, z_i)$ that is a differentiable function of the denoising model parameters ρ. This is once again similar to Equation (3.9) where the training example weights can also be interpreted as probabilities.

Beyond complaint relaxation, the main difference between the relaxation of Problem 3 and the nested problems of existing approaches [13, 68, 69] is how each one generates the training example weights ϵ_i. Two of the approaches [13, 68] do not leverage training example error patterns similar to Rain and Rain++. The weights ϵ_i are completely decoupled. In contrast Meta-Weight-Net [69] studies a special class of denoising models that take the loss of the target model $\ell(\theta, z_i)$ as its only input to produce an ϵ_i for each $z_i \in T$. The intuition is that z_i with high $\ell(\theta, z_i)$ are the ones causing unsatisfactory downstream performance and should be downweighted.

Problem 3 asks for a solution that goes further than just validation loss complaints and only one denoising model architecture. Ideally we would be able to plug in our complaint relaxation and a denoising model in one of the existing approaches and just have it work. However, as we will see in Section 6.3 there are tradeoffs involved that make the problem non trivial.

6.3 Challenges

We identify two challenges regarding plugging in any denoising model in the relaxation of Problem 3. The first one is actually choosing the right denoising model for the task in order to capture the training data error patterns. The second one is not being able to use a denoising model of interest because it is not differentiable. Given that black box solutons are typically inefficient, we focus on binary conjunctions following existing work [33, 106].

6.3.1 Denoising Model Capacity

The key concern with designing the denoising model is giving it high enough learning capacity to capture the true error patterns but not too high capacity so that it can capture spurious patterns. This particularly important because a denoising model can resolve a complaint without identifying

the true training data error pattern especially if the complaint is ambiguous. Drawing a similarity to supervised learning models that achieve low training loss but fail to generalize, we will say that a denoising model is overfitting to the complaint if it resolves it without learning the true error pattern.

Similar to supervised learning, there is no free lunch when it comes to avoiding complaint overfitting. Users need to be able to encode what they know about the error patterns in the denoising model design. However, users do not have many options to pick from. On the one hand, some approaches encode no prior information in their denoising approach by using fully decoupled weights ϵ_i [13, 68, 69]. On the other hand, Meta-Weight-Net is designed to prioritize removing training examples with high loss under the target model. Users need guidance on how to translate their domain knowledge about the errors to denoising model designs.

6.3.2 Binary Conjunctions

Another concern with MetaRain is that it can natively handle only differentiable denoising models. While this family of models is pretty expansive, simpler models like the conjunctions of [33, 106] are more concise and thus more interpretable to end users compared to linear models and of course more complex differentiable model families.

6.4 Our Approach

In this section we will analyze our approach towards addressing the two challenges outlined above. Regarding controlling the capacity of the denosing model, we will provide recommendations on how to design the denoising model and how to regularize it. Regarding the case of non differentiable denoising models, we will focus our attention on binary conjunctions and propose strategies on how to relax them into differentiable models.

6.4.1 Controlling Denoising Model Capacity

The denoising model takes as input a training tuple $z_i = (x_i, y_i)$, where x_i are the training example features and y_i the training example label, and decides whether to intervene or not on the

training example. In this section we will provide suggestions to MetaRain users on how to design such a denoising model. We note that regarding the use of the x_i input of the denoising model, this is highly domain specific. In general, the best practices as applied to the target model are also applicable to the denoising model. Our focus is going to be on handling the y_i input since this is not an input for the target model and thus requires special treatment. Whether and how to use y_i for the denoising model depends on how the training set is corrupted. We now examine four cases.

General corruptions In the general case, whether a training example is corrupted can heavily depend on the given label y_i. This is clearly the case when the label y_i itself might be noisy. But even if the labels are not corrupted, what is a nominal or an outlier value for a feature in x_i may differ from class to class so providing y_i can be helpful to the denoising model.

A straightforward approach to design a denoising model is to simply copy the architecture of the target model. Given an (x_i, y_i), the denoising model can return the probability of the y_i class. While we could train a separate model for each class, this approach has two advantages. The first one is that denoising models for different classes can share parameters. The second one is that the model learns that in general the model classes are mutually exclusive which also constrains the model. Both of these properties can lead to reduced overfitting.

Label noise with a single source class Here we focus on cases of corrupted training labels with the following restriction: All the corrupted training examples are instances of the same true class j and j is known to us. This class of errors is of particular interest for complaint driven debugging because users can easily detect the effects of such errors with a class count histogram and they can use a complaint that the relevant class count is too low to debug them.

Knowing this property of the label noise simplifies the denoising model design. The only thing that the denoising model needs to detect is if a given training example is compatible with the j class or not. This allows us to frame the denoising model as a one vs rest classifier for the class j, meaning that a single output is only required. Training examples for which $y_i = j$ are also known to be clean under this assumption, so the denoising model can output a 1 for them directly.

Label noise with a single target class Here we focus on cases of corrupted training labels with the

symmetrical restriction: All the corrupted training examples have the same $y_i = j$ and j is known to us. Similar to the single source case, users can easily detect and debug these errors by finding classes where the class count is too high. The solution is then symmetrical. We frame the denoising model as a one vs rest classifier for class j. All examples having $y_i \neq j$ are known to be clean, so the denoising model can output a 1 for them directly.

Corrupted feature based slices One common case is training errors concentrated in feature-based slice of the training set. For example, the training examples coming from one client of CompanyX may be corrupted due to an integration error. Another example is outlier feature values in the training set. In all of these cases y_i is redundant because x_i is sufficient to capture the error pattern regardless if the y_i, the x_i or both parts is the one that is actually corrupted. Ignoring y_i in this case may lead to improved denoising model predictions.

Another way to control denoising model capacity is to use regularization. Here we will study regularizers that encourage the denoising model to do the minimal number of deletions as dictated by Problem 3. To encourage the model to delete as few training examples as possible we can regularize the objective with the number of deletions, a practice also followed by BOExplain. Of course the number of hard deletions proposed by the denoising model is non differentiable. DUTI [13] proposes the following relaxation instead

$$q(\rho) = q_c(\theta^*_\epsilon(\rho)) + \lambda \sum_{i=1}^{n} [1 - w(\rho, z_i)] \tag{6.1}$$

where q_c is the relaxed complaint component that only depends on the trained model parameters θ^*_ϵ. MetaRain uses an alternative relaxation [107] that converges to the hard count with decision threshold 0 as the hyperparamter β goes to infinity

$$q(\rho) = q_c(\theta^*_\epsilon(\rho)) + \lambda \sum_{i=1}^{n} \left[e^{-\beta w(z_i, \rho)} \right]. \tag{6.2}$$

Of course the downside of increasing β is that it makes the objective less smooth making the optimization problem of Equation (6.2) harder to solve.

6.4.2 Learning Binary Conjunctions

Differentiable models are already powerful enough to capture binary conjunctions, conjunctions over binary variables. For example, linear models can readily capture binary conjunctions. Unfortunately, even simple model classes like linear models are very expressive and can capture much more than just conjunctions. As a result, training a linear denoising model over a set of binary features with MetaRain does not guarantee that the returned model is a conjunction. This is particularly problematic if we want to return a conjunction as a concrete summary of what is the cause of the users complaint. Here we will propose two different relaxation approaches that restrict the class of expressible models while including all binary conjunctions.

Linear model Our first observation is that when expressing a conjunction as a linear model, all non-zero feature weights have the same sign. Given our convention that the denoising model returns 0 when a training example is deleted and 1 when it is not, all the feature weights of the denoising model should be negative or zero. In addition we need a positive bias parameter. Our second observation is that in order to return concise conjunctions to users, we should encourage the denoising model feature weights to be sparse. Applying the sparsity regularization of [107], we introduce two new hyperparameters $\mu, \gamma > 0$ and augment the default objective

$$\rho^* \in \underset{\substack{0 \le i \le n : \rho_i \le 0 \\ \rho_{n+1} \ge 0}}{\arg\min}\ q_c(\theta^*_\epsilon(\rho)) + \lambda \sum_{i=1}^{n} \left[e^{-\beta w(z_i, \rho)} \right] - \mu \sum_{i=1}^{n} e^{\gamma \rho_i}. \tag{6.3}$$

In the case that the returned model is not equivalent to a conjunction, we can still use the features corresponding to the top-k negative weights, for some user chosen k, to form a conjunction.

Conjunction enumeration Another solution is to enumerate the available conjunctions following the approach of Gopher. By applying the generated conjunctions on each training example we can then generate a new set of binary features. We can then reduce the problem to learning a 1 term conjunction. Despite the fact that the denoising model may have a very high number of parameters, one for each candidate conjunction, the fact that we are looking for 1 sparse solutions enables us to

aggressively regularize the denosing model objective and converge in fewer iterations.

6.5 Experiments

In this section we seek to experimentally study the performance of MetaRain. This will involve comparing its accuracy with nested optimization baselines DUTI [13] and Meta-Weight-Net [69] and comparing its latency on conjunction based explanations with Gopher [106]. Our experimental analysis will also provide insights on the trade-offs between model expressivity and complaint overfitting as well as tuning of regularization parameters. Both are important for practitioners seeking to apply the techniques of MetaRain on their problems.

6.5.1 Experimental Settings

Methods

DUTI [13] is a nested optimization approach for training data debugging. Similar to Rain and Rain++, the training example weights are model free so the output is a ranking of training examples. In the trade-off between denoising model expressivity and overfitting to the complaint, DUTI aims for maximum expressivity by leaving the training example weights unconstrained.

Meta-Weight-Net [69] is also a nested optimization approach for training data debugging. In the trade-off between denoising model expressivity and overfitting to the complaint, Meta-Weight-Net aims for very low expressivity by constraining the training example weights to be functions of a single feature, the target model loss.

Rain++ is based on our work in Chapter 5. Unlike the approaches above it does not rely on nested optimization and relies exclusively on an influence function based sensitivity analysis. Comparisons with Rain++ will help us understand when computationally expensive nested optimization approaches have an advantage over the computationally cheaper local ones like Rain++.

Gopher [106] specializes in finding influential conjunction predicates over the training set whose deletion addresses user provided fairness complaints. Gopher relies on efficiently enumerating the potential conjunctions and approximating the effect of their deletions with a second order influence

function approach [108], a more accurate but more computationally expensive version compared to the one used by Rain and Rain++. We treat the output of Gopher as ground truth for the problems where it is applicable and comparing the output of MetaRain with it. We will also compare the scalability of the two approaches as the number of parameters of the target model increases.

MetaRain is our nested optimization framework for training denoising models in order to address user complaints. For each complaint and training corruption combination, we will be evaluating the debugging performance of our denoising model architecture proposal as discussed in Section 6.4.1 and comparing with the applicable baseline systems.

Datasets

Our evaluations will focus on datasets that are known to either have label errors or to exhibit fairness biases. This allows us to evaluate to what extent the above approaches can detect realistic patterns of errors and biases that are not synthetically engineered to be simple or to exactly match the expressiveness of the denoising models evaluated.

CIFAR-10N [109] is a variation of the standard CIFAR-10 [97] image classification benchmark dataset. The difference is that CIFAR-10N replaces the officially provided labels of CIFAR-10, which are typically assumed to be perfectly accurate, with labels collected using Amazon Mechanical Turk. Each image of the training set of CIFAR-10 was annotated by 3 workers after blocking and rejecting workers "who submit answers with fixed/regular distribution patterns". The three labels per training examples are then used to create 5 different singly labelled datasets. The first setting, termed the aggregate setting, the label is chosen to be the majority vote of labels or a randomly chosen answer in the case of a three-way tie. The next three settings, termed the random settings, pick the i-th worker vote for $i \in \{1, 2, 3\}$ for each of the images. The last setting, termed the worst setting, for each image it chooses randomly one of the wrong label annotations if one exists. Otherwise it chooses the correct label. The label error rates are 9.03%, 17.23%, 18.12%, 17.64% and 40.21% in the aggregate, random-1, random-2, random-3 and worst settings respectively.

The **German Credit Dataset** [82] contains the personal financial information of 1000 bank

account holders. The predictive task is to classify whether an account holder has a good or bad credit risk. The problem is that models trained on the dataset exhibit group bias against young account holders. Gopher identified that deleting a small percentage of training examples and retraining the target model can result in significantly reduced bias. We will thus compare MetaRain's and Gopher's abilities in detecting these tuples in response to fairness complaints. We use the same preprocessing and train test splits as Gopher [106].

Target Models

Given that nested optimization is computationally expensive, our primary focus will be on simple target model architectures that can be efficiently optimized. Within the context of simple architectures, we will also evaluate scalability with respect to the number of target model parameters.

For the CIFAR-10N dataset will use Logistic Regression as our target model architecture. Instead of training on the raw pixel data of CIFAR-10, we will embed the images using a SIMCLRv2 [110] model trained on Imagenet-1K. The dimension of the embeddings is 4096. The embedding model was chosen based on the following considerations. On the one hand, the Logistic Regression model trained on top of the embeddings should not be robust to the label noise of the CIFAR-10N. Otherwise it will be difficult to specify valid complaints on the model's behavior that target these label errors. On the other hand, the embeddings should be chosen such that model prediction errors not caused by the training data label errors do not dominate the downstream behavior of the model. We empirically find that the SIMCLRv2 embeddings satisfy both criteria.

For the German Credit Dataset, we will use Logistic Regression using the 31 features designed by [106]. We also test the scalability of the applicable approaches as the number of target model parameters increases. To avoid introducing new features, we instead duplicate the existing 31 features several times. We then measure and compare the latency of the applicable methods as the number of target model parameters grows.

Q_1	SELECT -AVG(LOG(predict_proba(*, D.label))) FROM D
Q_2	SELECT COUNT(*) FROM D WHERE predict(*) = "deer"
Q_3	SELECT AVG(predict(*)) FROM D WHERE D.age < 45
Q_4	SELECT AVG(predict(*)) FROM D WHERE D.age ≥ 45

Table 6.1: Summary of queries used in the experiments.

Complaints

We will use 3 distinct complaints. To separate the effects of complaint ambiguity on overfitting to the complaint, we will first compare the applicable methods on the unambiguous complaint that the cross entropy loss over the test data of the target model on CIFAR-10N is not 0. The loss is computed by Q_1 of Table 6.1, where predict_proba(*, D.label) computes the probability of the ground truth label D.label. This complaint is not ambiguous because it can only be addressed in a unique way, by changing the target model predictions to match the true labels exactly. For the second complaint on CIFAR-10N, we turn to the ambiguous complaint that the count of deer class predictions, computed by Q_2 of Table 6.1, is too low. We choose the deer class for our complaint as its count is heavily affected by the CIFAR-10N label errors, especially for the worst noise level setting. The third complaint targets the German Credit Dataset. Inspired by Gopher's [106] application on debugging fairness complaints, we use the same complaint, namely that the statistical parity difference between the minority group of young people and the majority group of old people should be 0. The statistical parity difference between young and old peopleusing 45 as an age threshold can be computed as the difference between Q_3 and Q_4 of Table 6.1.

Measures

Even if MetaRain and Meta-Weight-Net can in principle return classifiers as a result of their analysis, in general the only way to evaluate the quality of these classifiers is by measuring the debugging quality of their predictions. As a result, in the majority of the experiments we will follow the evaluation procedure of Rain and Rain++ and use AUC_R. As a reminder, let r_i be the percentage of correctly identified corrupted training examples in the top-i ranked points. AUC_R computes

the average r_i up to the true number of corruptions N, i.e. $\frac{1}{N}\sum_{i=1}^{N} r_i$. The result is divided by its maximum value to derive a normalized score in $[0, 1]$.

As a special case, when MetaRain returns a binary conjunction based classifier we can also evaluate the correctness of the output directly, without going through its predictions. Treating the return of Gopher as a ground truth, we will evaluate the Jaccard similarity between the conjunction terms returned by MetaRain and Gopher. For the purposes of evaluating the latency of MetaRain and Gopher, we will exclude the cost of initially training the target model.

Implementation

Similar to Rain++, MetaRain is implemented in JAX [101], an automatic differentiation framework on top of XLA [102]. All experiments are run on a Google Cloud **n1-standard-8** machine with one NVIDIA V100 GPU. For a fair comparison, we also use our own JAX implementations of DUTI, Meta-Weight-Net and Gopher. The differentiation of $\theta^*_{\text{soft}}(\rho)$ for all approaches that require it was implemented using the library JAXopt [111].

6.5.2 Loss Complaints

In this experiment we seek to understand how the expressiveness of the denosing model affects its ability to overfit to the user's complaint. The approach of DUTI corresponds to the most expressive denoising model because the weights of each training example can be chosen independently to best address the user's complaint. In contrast, the denoising model of Meta-Weight-Net is not very expressive as weights are constrained to be functions of the model's target loss. MetaRain also limits the expressiveness of the denoising model compared to DUTI, but the denoising model can make use of all the training example features instead of just the target model loss.

One problem that may arise with this evaluation is that the use of regularization may also contribute to the ability of the denoising model to overfit to the user's complaint. In order to measure the effect of denoising model choice in isolation, it would be preferable to evaluate the different choices without regularization. In general this runs the risk that all some or all approaches converge

to trivial solutions leading to no meaningful insights. For example, for the count complaint of $Q2$ denoising models may choose to remove all non-deer examples to maximize the count of deer predictions. Similarly, a denoising model may choose to delete all bad credit examples to trivially resolve the fairness complaints of $Q3$ and $Q4$. Instead, we are going to use the loss complaint of $Q1$ where there is no trivial solution that minimizes the loss of the target model on the test set.

Our way of detecting evidence of overfiting is inspired by the analogous phenomenon in traditional supervised learning. In traditional supervised learning a key indication of overfitting is that the training loss of the supervised model decreases but the performance of the model in the validation or the test data degrades. Mapping this intuition to our experiment, our evidence of overfiting will be that the loss of the denoising model decreases but the quality of the training example rankings as measured by the AUC_R degrades. For the complaint on $Q1$ the loss of the denoising model coincides with the target model test loss when using $\theta^*_{\text{soft}}(\rho)$ as its parameters.

Before analyzing the performance of the three approaches, we briefly discuss the configuration of MetaRain for this experiment. Given that the label errors of CIFAR-10N are distributed across all classes and our complaint on $Q1$ seeks to resolve all of them, we design the denoising model to match exactly the architecture of the target model as discussed in Section 6.4.1. Regrading the initialization of the denoising model, instead of opting for a randomized initialization we use the parameters of the target model as initialization. The target model for all three approaches is initialized by training on the noisy training data for 800 epochs.

In Figures 6.1 to 6.3 we compare the target model loss and denoiging models AUC_R for DUTI, Meta-Weigh-Net and MetaRain for the aggregate, random-1 and worst noise settings respectively. We note that the number of epochs for Meta-Weight-Net are not comparable with the ones of DUTI and MetaRain. In each denoising model epoch Meta-Weight-Net trains the target model for a single epoch following the implementation of [69]. In contrast, in each denoising model epoch DUTI and MetaRain train the target model over several epochs.

Overall the experimental results agree with our intuition. DUTI, being the most expressive, achieves the lowest target losses out of all the three approaches and exhibits significant overfitting

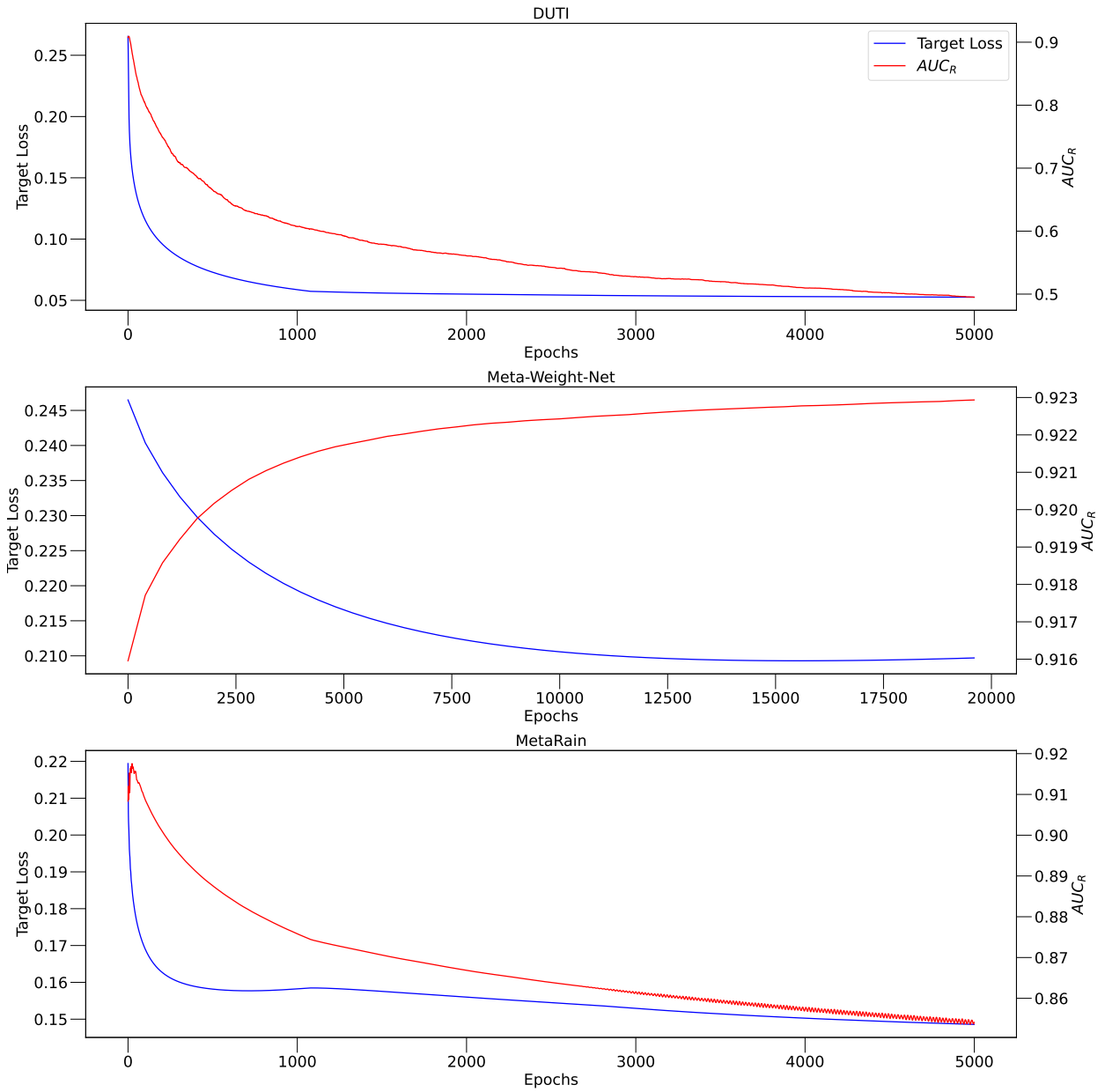


Figure 6.1: Comparison of target model loss and AUC_R for DUTI, Meta-Weight-Net and MetaRain for the loss complaint of $Q1$ on CIFAR-10N aggregate noise setting over denoising model epochs.

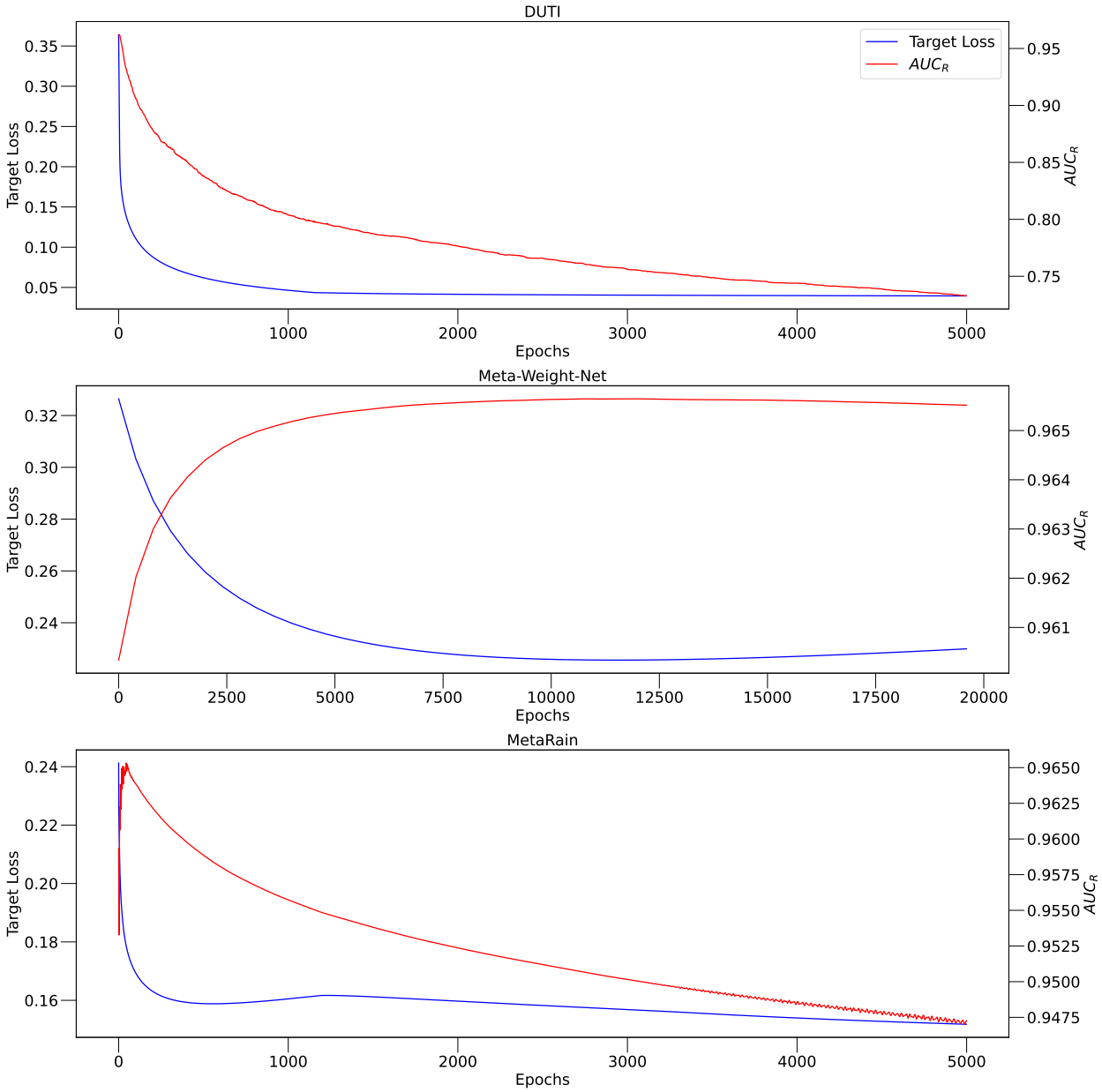


Figure 6.2: Comparison of target model loss and AUC_R for DUTI, Meta-Weight-Net and MetaRain for the loss complaint of $Q1$ on CIFAR-10N random-1 noise setting over denoising model epochs.

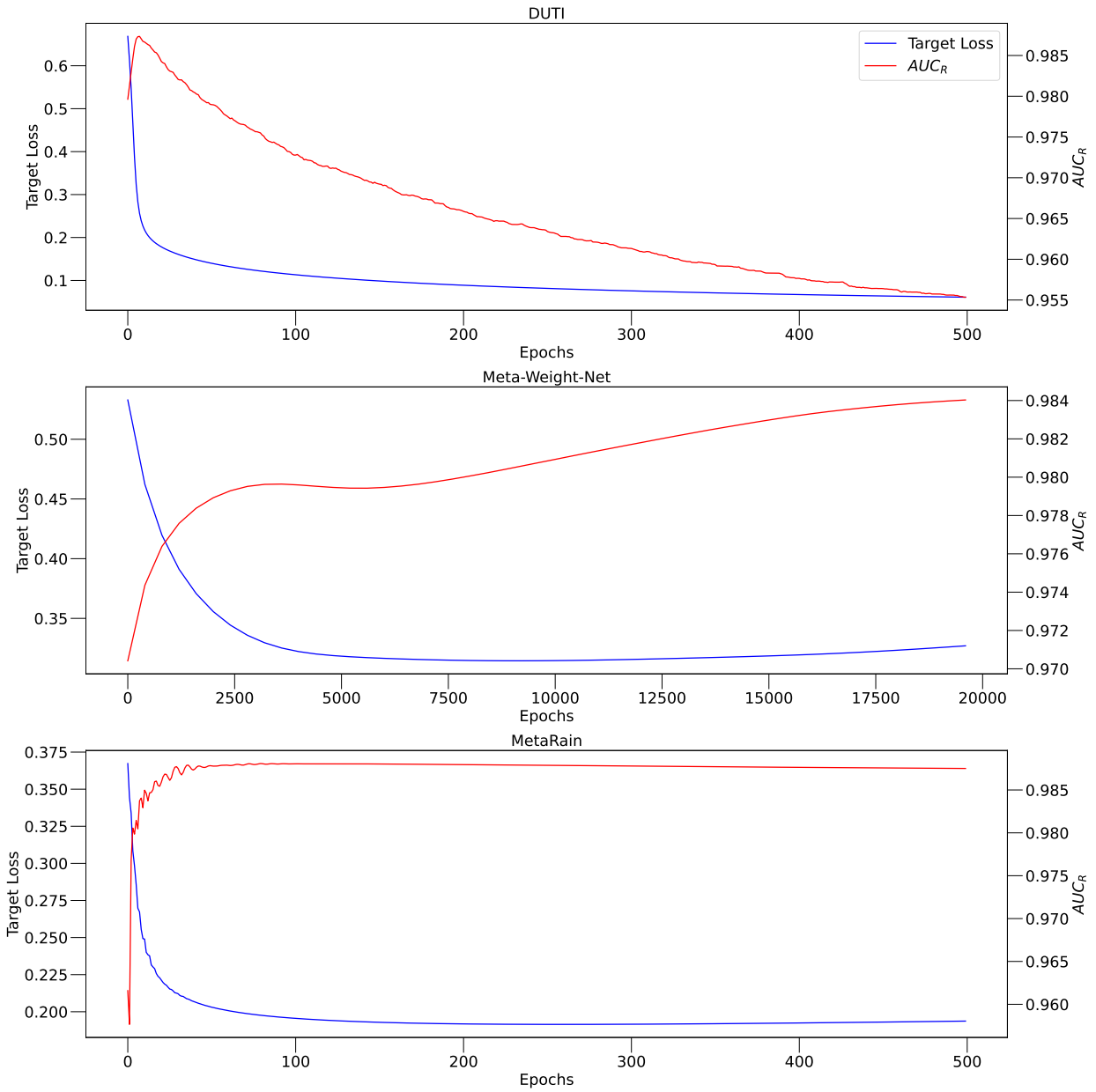


Figure 6.3: Comparison of target model loss and AUC_R for DUTI, Meta-Weight-Net and MetaRain for the loss complaint of $Q1$ on CIFAR-10N worst noise setting over denoising model epochs.

in the aggregate and random-1 noise settings. Specifically, in these settings AUC_R drops from above 90% to around 50% and 75% respectively. MetaRain, which is less expressive than DUTI, also exhibits significantly smaller degradations of AUC_R due to overfitting in the aggregate and random-1. Finally, Meta-Weigh-Net, the least expressive approach, exhibits no indications of overfitting across all three settings as AUC_R does not degrade as the target loss decreases.

Of course DUTI's and MetaRain's overfitting behavior can be corrected with regularization. This has been already demonstrated by the work that proposed DUTI in the first place [13]. We will similarly demonstrate this in our ambiguous and thus more challenging count complaints on $Q2$ for both MetaRain and DUTI. Still, not requiring to use regularization and tune its hyperparameters as Meta-Weigh-Net does in this experiment is in advantage.

Takeaway: Less expressive denoising model architectures like Meta-Weigh-Net are more robust to overfitting issues. In contrast, the more expressive denoising archiectures of DUTI and MetaRain may need the help of regularization to avoid overfitting.

6.5.3 Count Complaints

We now move on to the more challenging count complaint on $Q2$. We will focus on the worst noise setting of CIFAR-10N where the discrepancy of the computed count and the true count is the greatest. Because the count complaint targets only the label errors where a true deer training example is labelled as an example of another class, in our AUC_R calculation we will consider only these label errors as true errors and we will treat all other errors as non-errors. We fix the weights of all training examples with a deer label to 1 for DUTI, Meta-Weigh-Net and MetaRain. We note however, that we did not remove or correct any of the worst setting labels for any of the approaches.

We briefly discuss the configuration of MetaRain for this count complaint. Based on our discussion in Section 6.4.1, we chose a one vs rest classifier design using the same features as the target model. The parameters of the denoising model are randomly initialized. Because the weights chosen by the denoising model are initially random, the target model learned in the first few epochs can be far from optimal. This problem can be propagated to the next epochs if these suboptimal

target models are then used as initialization for the next epoch. To avoid this we reset the target model parameters to their initial value for the first 100 epochs of denoising model training.

In this experiment we seek to understand how model expressiveness when combined with regularization affects the debugging output quality of the trained denoising model. We use for all approaches the regularizer of Equation (6.2) with $\beta = 3$ and vary the regularization weight λ. Intuitively, higher values of λ force the denoising model to downweight less examples which is in agreement with the minimality condition in Problem 3. To monitor these tradeoffs we will measure both the AUC_R as well as the average weight of all training examples across the optimization trajectory of the denoising model.

Unfortunately, we found that with or without regularization Meta-Weight-Net was not able to get rankings that were significantly better than random in terms of AUC_R. We attribute this to the fact that the target model loss alone is not sufficient to determine if the removal of the training example addresses the $Q2$ complaint. An example having a high or low target model loss for its given label does not determine on its own if its true label should be deer or not.

In Figures 6.4 and 6.5 we initialize the target model by training for 800 epochs and run DUTI and MetaRain varying λ from 0.4 to 0.7 in 0.025 increments. The results are once again in agreement with our intuition. Small values of λ allow the denoising model of both approaches o reach the trivial solution of deleting all non-dear examples leading to very low AUC_R. Increasing the value of λ leads to both higher AUC_R and average weights. Too high values for λ lead to further increases to the average weights but without the corresponding AUC_R increases. Specifically, MetaRain may also converge to denosing models that do almost no deletions which leads to very low AUC_R.

A user having an estimate on how many training example tuples should be deleted to address the complaint can tune the regularization value λ with a binary search strategy. Requiring the user to supply some information on how many tuples should be deleted is fairly common. BOExplain [33] allows users to tune a regularization weight similar to our approach. Gopher [106] limits their search to deletion predicates that have support greater than a minimum threshold. The same support constraint can be found in query explanation in general [112].

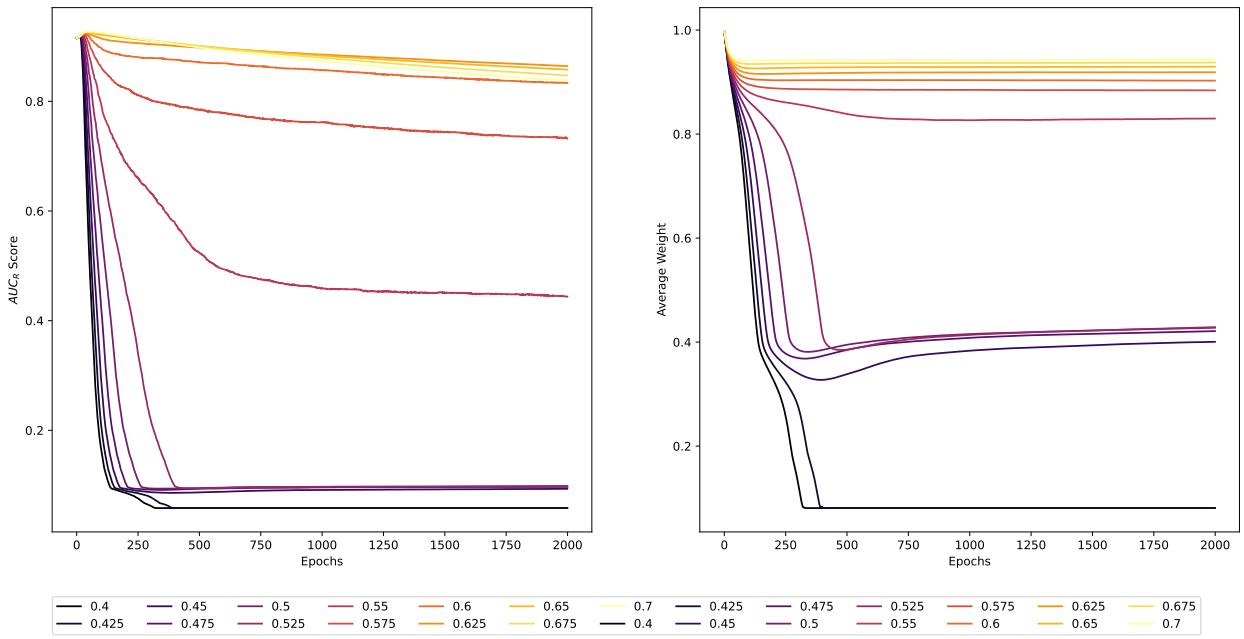


Figure 6.4: AUC_R and average training example weight for DUTI for the count complaint of $Q2$ on CIFAR-10N worst noise setting over denoising model epochs varying the regularization weight λ. The target model is initialized by training for 800 epochs on the noisy training data.

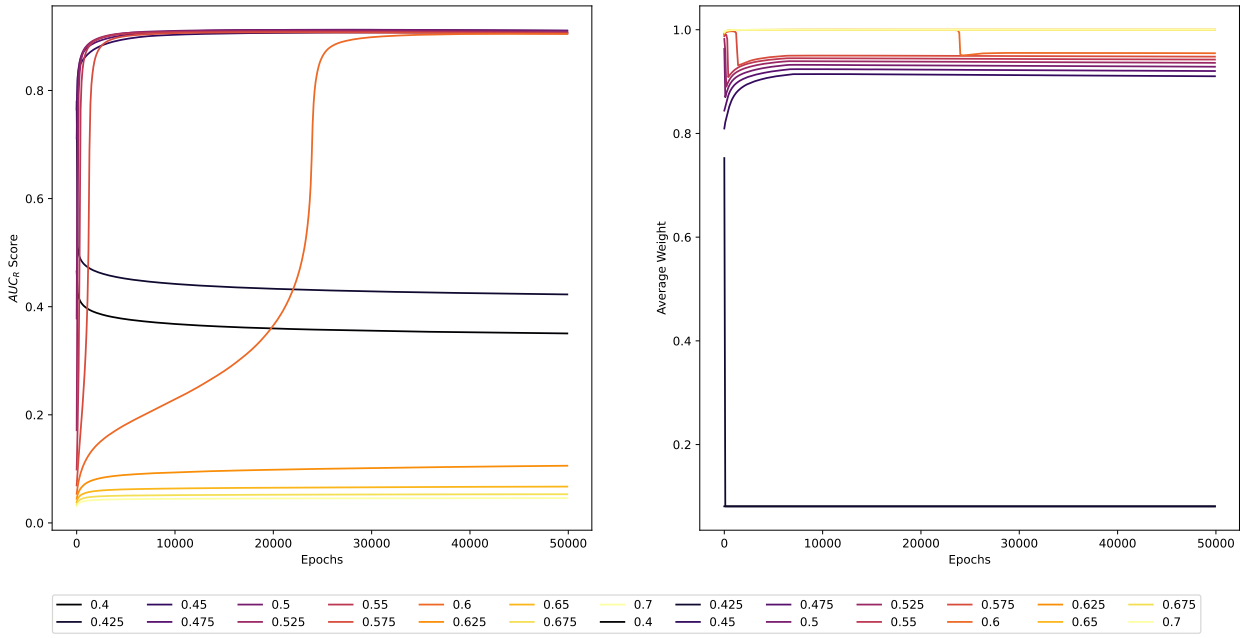


Figure 6.5: AUC_R and average training example weight for MetaRain for the count complaint of $Q2$ on CIFAR-10N worst noise setting over denoising model epochs varying the regularization weight λ. The target model is initialized by training for 800 epochs on the noisy training data.

Method	AUC_R (800 epochs)	AUC_R (80000 epochs)
DUTI	0.864	0.845
MetaRain	0.91	0.909
Rain++	0.905	0.745

Table 6.2: Summary of $Q2$ AUC_R performance for DUTI, MetaRain and Rain++ when initializing the target model with 800 and 80000 epochs of training respectively.

We seek now to verify how robust are our findings to the target model initialization. We change the initialization of the target model from 800 epochs of training to 80000 epochs to allow the target model to fit more to the noise in the training labels of the CIFAR-10N worst noise setting. In Figures 6.6 and 6.7 we observe that the same overall trends in the trade-offs between the values of λ, average weights and AUC_R for both DUTI and MetaRain.

Finally we want to conduct a comparison between DUTI, MetaRain and Rain++ across both target model initialization settings. For DUTI and MetaRain we take the AUC_R of the last epoch for all regularization settings and pick the highest score. For Rain++, we pick the highest AUC_R varying the number of eigenvectors up to $k = 30$. We summarize our results in Table 6.2. We find that MetaRain has higher AUC_R than DUTI in both initialization settings. Compared to Rain++, we find that at the 800 epochs initialization performance is comparable. However, because Rain++ is a local method the bad initialization of 80000 epochs reduces its AUC_R significantly whereas the effect on MetaRain's AUC_R is comparatively negligible.

Takeaway: Less expressive denoising model architectures like Meta-Weigh-Net may fail to resolve the user's complaint if they are not equipped with appropriate features to identify the complaint's cause. DUTI and MetaRain can leverage regularization to debug even ambiguous complaints. MetaRain outperforming DUTI on the $Q2$ count complaint, indicates that limiting model expressiveness is still useful even in the presence of regularization.

6.5.4 Fairness Complaints

In this experiment we seek to evaluate the ability of MetaRain to return binary conjunctions that describe which training examples should be deleted in order to resolve the user's complaint. Our

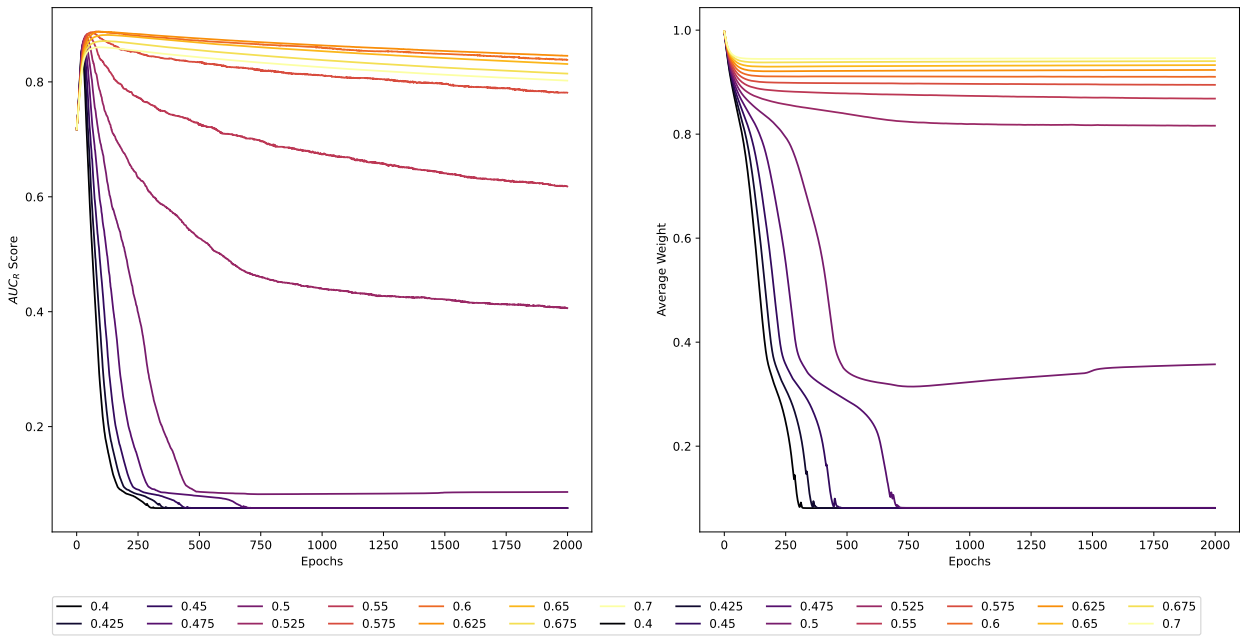


Figure 6.6: AUC_R and average training example weight for DUTI for the count complaint of $Q2$ on CIFAR-10N worst noise setting over denoising model epochs varying the regularization weight λ. The target model is initialized by training for 80000 epochs on the noisy training data.

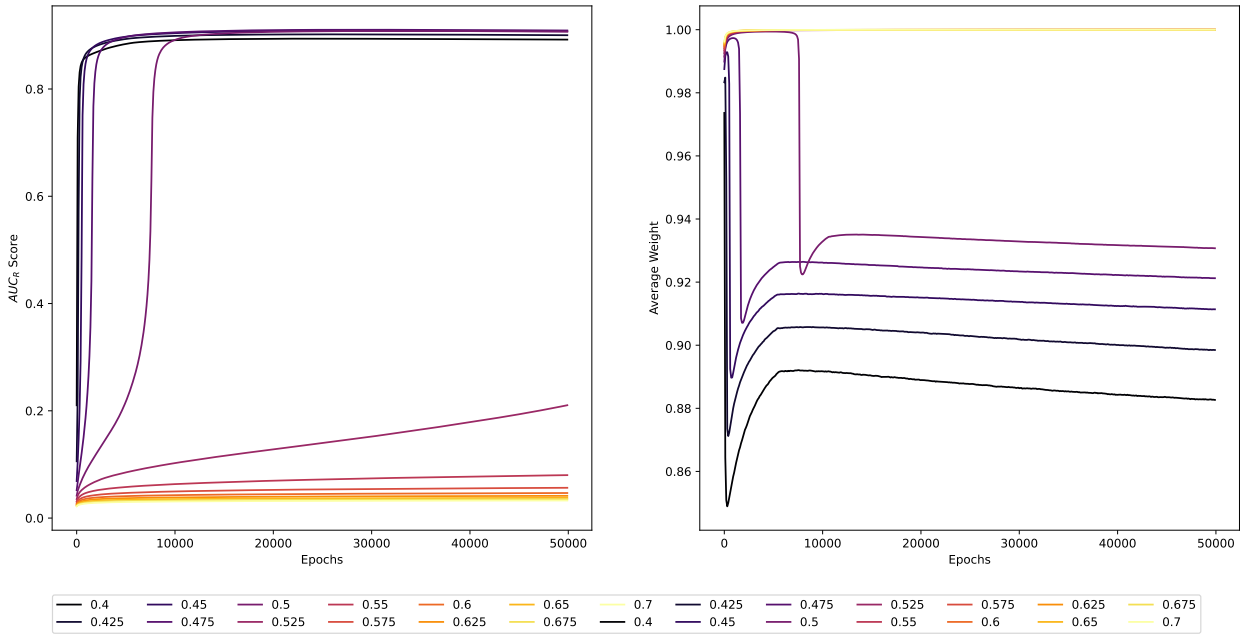


Figure 6.7: AUC_R and average training example weight for MetaRain for the count complaint of $Q2$ on CIFAR-10N worst noise setting over denoising model epochs varying the regularization weight λ. The target model is initialized by training for 80000 epochs on the noisy training data.

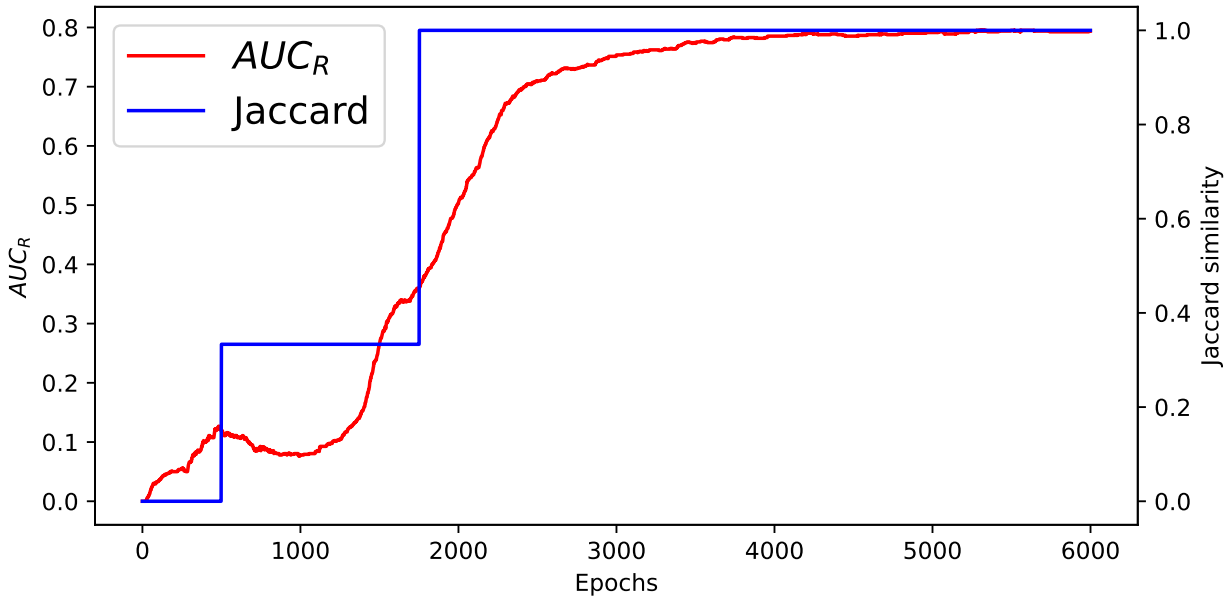


Figure 6.8: MetaRain's training example AUC_R and term Jaccard similarity of its top two terms with Gopher's conjunction varying the denoising model epochs.

evaluation will be based on the fairness application of Gopher [106] on the German Credit Dataset. The fairness complaint is that the statistical parity difference between old and young people as expressed by the difference of $Q3$ and $Q4$ is too high.

One approach to return a binary conjunction as discussed in Section 6.4.2 is to enumerate the candidate conjunctions and turn them into binary features of the denoising model. We follow the approach of [106] and consider conjunctions of up to 4 terms that have support from 5% to 20% over the training set. Ignoring the Gopher's pruning heuristics we identify around 23K conjunctions.

We use the regularized objective of Equation (6.3) to train the denoising model. After tuning the available hyperparameters we find that 800 epochs are enough for MetaRain to identify the same conjunction as Gopher. This takes 2 minutes including the time to generate the 23K binary conjunctions. Assuming all the required hessians and gradients are precomputed, generating and scoring the 23K conjunctions with Gopher takes around 30s. While MetaRain in this experiment evaluated fewer candidate denoising models than Gopher without pruning heuristics (800 vs 23K), each evaluation was slower for MetaRain because it retrains the target model whereas Gopher uses second order influence analysis.

The other alternative is to learn the conjunctions from the base predicates directly. We use once again Equation (6.3) and tune the hyper parameters. In Figure 6.8 we evaluate the MetaRain's denoising model in two ways. We first compute the AUC_R of MetaRain's ranking treating the training examples deleted by Gopher as the ground truth corruption. We also compute the Jaccard similarity of the top two terms found by MetaRain with the two terms of Gopher's conjunction. While the denoising does not converge to Gopher's conjunction exactly, as evidenced by the fact that its AUC_R even at 6K epochs is less than 1, it manages to identify the importance of the terms chosen by Gopher at around 2K epochs.

We finally want to compare the latency of MetaRain using the base predicates with that of Gopher as the number of features of the target model increases. For Gopher we will once again measure the cost of generating and scoring all the 23K conjunctions. We will not include the time to required to precompute the gradients and Hessians of all training examples that is required by Gopher's second order approach as these costs can be pushed offline similar to Rain++. In Table 6.3 we scale the number of parameters by replicating the available features 5, 10 and 20 times.

We find that MetaRain's runtime actually decreases despite the increase of target model features. There are several reasons for that. First, duplicating a feature increases the effective learning rate of the target model allowing it to train faster. This indicates that the scale 1 learning rate could have been tuned to a higher value to achieve a faster convergence. Second, MetaRain does not have quadratic time complexity in the number of target model parameters. MetaRain does not materialize any Hessians or their inverses. It instead uses the conjugates gradient algorithm with a small number of steps for its influence analysis step. Third, even at scale 20x the target model has 621 features which is relatively small and does not fully utilize the available parallelism of our GPU.

In contrast Gopher's runtime continues to increase as the scale increases as a result of its quadratic time complexity in the number of target model parameters. At scale 20x we find that the available GPU memory is not enough to fit all the training example Hessians that Gopher precomputes to accelerate second order influence analysis.

It is important to note that the runtime evaluations of MetaRain do not include the time required

Scale	MetaRain (600 epochs)	Gopher (23K conjunctions)
1x	1m27s	30.1s
5x	47.4s	45.7s
10x	46.9s	1m43s
20x	46.9s	-

Table 6.3: Runtime comparisons between MetaRain and Gopher as the number of parameters is multiplied by a factor of 1x, 5x, 10x and 20x. Gopher runs out of GPU memory at 20x scale.

to tune the hyperparameters used by the objective of Equation (6.3). Still, the approach of MetaRain may be preferable if the memory footprint of Gopher is prohibitive.

Takeaway: Leveraging the relaxation techniques of Section 6.4.2, we can use MetaRain to extract binary conjunctive predicates that succinctly describe the proposed deletions. While Gopher may require less hyperparameter tuning than MetaRain, we find that quadratic space complexity in terms of target model parameters can be prohibitive even for datasets with less than 1K training examples and 1K features. In contrast MetaRain does not exhibit this quadratic behavior.

Chapter 7: Accelerating Workflow Execution Using Provenance

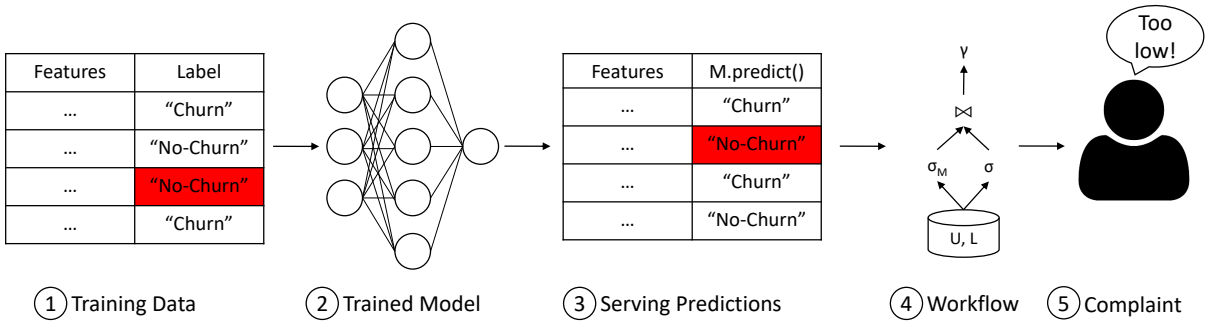
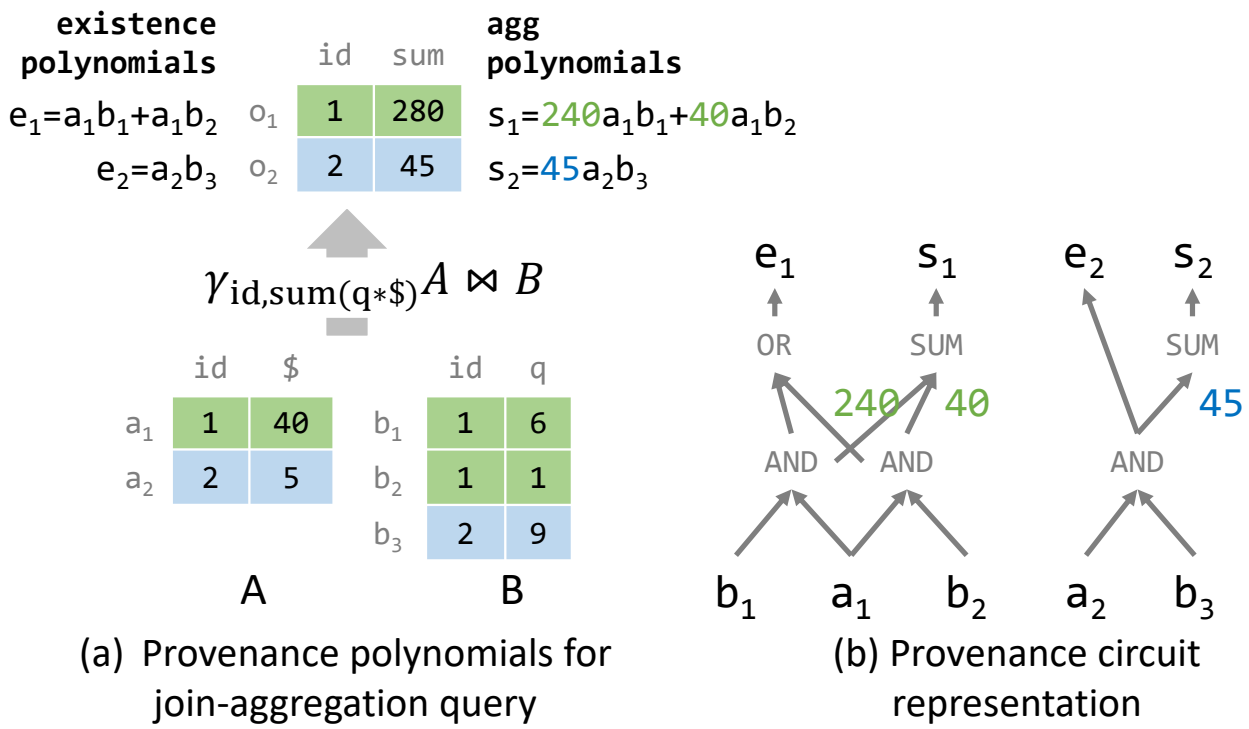


Figure 7.1: Overview of complaint driven training data debugging. This chapter focuses on accelerating the evaluation of (4) when model predictions of (3) change.

In Figure 7.1 we revisit the propagation of errors from training data to workflow outputs and the complaint of the user. A key step of evaluating the effect of training data interventions is refreshing the output of (4) given the modified ML model predictions of (3). In Section 1.5.6 we discussed how this can be cast as a view maintenance problem under input tuple deletions. Indeed, for each tuple in our serving database where we want to apply our model, we can replicate it once for every possible classification. Then when we seek to evaluate the workflow on a new model we can delete all the tuples except the ones representing the true model predictions and refresh the workflow result view. This observation can be readily used by any future or existing complaint driven debugging that need to evaluate the workflow output several times.

This observation is already leveraged by Rain and Rain++. Their provenance polynomials directly map to the above view maintenance problem. Rain's polynomial inputs with value 1 correspond to input tuples the remain in the input whereas 0 valued inputs correspond to deleted input tuples. Computing and combining the values of the provenance polynomials of each output row and aggregate result, we can reconstruct the refreshed view after the deletions.

The reduction of view maintenance under input tuple deletions to provenance polynomial



(a) Provenance polynomials for join-aggregation query

(b) Provenance circuit representation

Figure 7.2: Join-aggregation query provenance and its circuit representation.

evaluation has been long established for a wide class of queries that subsumes SPJUA [28, 29]. Here we give a concrete example about how the reduction between these to problems works in practice.

Example 2. *Consider the query $\gamma_{id,sum(q*\$)}A \bowtie B$ in Figure 7.2, where each input tuple is annotated with an existence bit a_i or b_i. Each output tuple in Figure 7.2(a) is annotated with an existence provenance polynomial. Evaluating this polynomial determines if the tuple is deleted in response to input tuple deletions. Deleting A's second tuple by setting $a_2 = 0$ leads to $e_1 = 1$ and $e_2 = 0$. This indicates that only o_1 remains after the deletion. Each aggregate result is annotated with an aggregation polynomial that computes the updated the aggregation value. Deleting B's second tuple by setting $b_2 = 0$ leads to $s_1 = 240$ and $s_2 = 45$, the aggregation values for o_1 and o_2 respectively.*

Provenance polynomials can be represented as a circuit [70], a DAG of logical and arithmetic operations. In Figure 7.2(b) computing the s_i and o_i outputs is equivalent to evaluating the symbolic formulas of Figure 7.2(a). Figure 7.3(a) is the pseudocode that evaluates a circuit by recursively traversing it in a top down fashion.

```
def e(n, args):
  match n.operation:
    case "SUM":
      tmp = map(e, n.children)
      return sum(tmp * args)
    case "OR":
      return e(n.left) | e(n.right)
    case "AND":
      return e(n.left) & e(n.right)
    case "LEAF":
      return n.exists
```

(a) Recursive circuit evaluation pseudocode

```
def intervene(a, b):
  '''interventions for A and B'''
  join = [a[1]b[1], a[1]b[2], a[2]b[3]]
  vals = [240, 40, 45]  // $*qty (cached)
     e = [join[0:1].sum(), join[2:2].sum()]
     s = [vals[0:1]*join[0:1], vals[2]*join[2]]
```

(b) This work generates code for fast provenance-based interventions

Figure 7.3: Join-aggregation circuit-based evaluation.

It is clear that reducing the latency of provenance polynomial evaluation would be to the benefit of complaint driven debugging systems that evaluate the relational component of the workflow using different models as inputs. Rain and Rain++ can also stand to benefit from such advances as the relaxed provenance polynomials they evaluate and differentiate differ only in terms of the logical and arithmetic operations evaluated in each circuit node.

When it comes to view maintenance under input tuple deletions, the main alternative to provenance polynomial evaluation is incremental view maintenance (IVM). IVM is a more general view maintenance technique compared to provenance polynomial evaluation as it can handle arbitrary modifications to the input relations and not just deletions. Unfortunately, the generality of IVM comes with some drawbacks. First, it is well established that IVM can be slower than full re-executing the view's query when the number of input tuple modifications is large. Second, IVM relies on materializing and indexing intermediate query relations, penalizing memory-bound queries. In general, IVM incurs unnecessary costs to support arbitrary database changes.

In theory, fine-grained provenance execution addresses the drawbacks of IVM mentioned above. First, it is faster than evaluating the query from scratch because it has strictly less work to do. The recursive evaluation of Figure 7.3(a) does not need to recompute which tuples matched in the join between A and B or the expression values in aggregates. Unlike IVM, this performance advantage is irrespective of intervention size. Additionally, provenance execution does not need to maintain

intermediate relations. The provenance circuit of Figure 7.2(b) does not materialize the tuples of $A \bowtie B$. This reduces the memory pressure that IVM can require. In practice, the situation is quite stark. To the best of our knowledge, ProvSQL [24] is the only system that implements provenance-based deletion view maintenance. Unfortunately, it is slower than general IVM systems like DBToaster [30] and even re-running the query from scratch.

The key reason is that the circuit representation is a poor fit for high performance. While algorithmically efficient, this representation suffers from poor data and instruction locality. The recursive evaluation leads to poor use of the memory bandwidth and CPU cache due to non-sequential memory accesses. Additionally, because each circuit node executes a different operation, there is little code locality and potential for out of order execution.

Our insight is that every output query shares the same circuit structure—all circuit nodes that correspond to the same logical query operator apply the same logical/arithmetic operations and access the same data. We can improve instruction and data locality by executing all node operations of a query operator together, and storing the nodes they access contiguously.

In this work we propose **FaDE**, a compilation based engine that generates efficient code for **Fa**st **D**eletion **E**valuation on counterfactual interventions. As noted above, FaDE and its techniques can be used by complaint driven debugging systems that evaluate the relational component of the workflow using different models as inputs and by Rain and Rain++. Beyond complaint driven training data debugging, FaDE can also be used by a variety of query explanation approaches [8, 113, 114]. FaDE leverages the state of the art fine-grained provenance capture engine Smoke [23]. In contrast to ProvSQL [24], Smoke records and exposes provenance metadata on a per operator basis in a compact pointer-free format. This is a perfect fit for FaDE, which uses the metadata of each operator to produce efficient compiled code. For example, in Figure 7.3(b) the inputs `a` and `b`, intermediates `join` and `vals`, and outputs `s` and `e` are all stored as arrays for improved data locality. Code is generated on a per operator level with $A \bowtie B$ yielding the first line and the group by aggregation the rest. To further decrease latency, FaDE generates efficient multi-threaded code. Additionally, when multiple independent interventions need to be evaluated, FaDE generates SIMD-

vectorized code that can evaluate thousands of interventions at a time. Moreover, FaDE prunes the provenance metadata captured by Smoke, resulting in computational and space reductions. Our full suite of optimizations enable up to 600× and 10,000× speed ups over DBToaster [30] and ProvSQL [24] respectively across several TPC-H queries. In summary, we contribute

- A compilation engine named FaDE capable of generating efficient code for counterfactual intervention evaluation.
- Extensions of FaDE to multi-threaded execution as well as batched intervention execution using SIMD-vectorized instructions.
- Provenance pruning optimizations that reduce both the space and runtime complexity of the generated executables.
- Extensive experimental comparisons between FaDE, IVM engine DBToaster and provenance based engine ProvSQL across several queries, being up to 600× and 10,000× times faster respectively. Our experiments showcase that FaDE can enable interactive query explanations by evaluating hundreds of thousands of interventions on multi-join queries in less than 100ms.

The rest of the chapter is organized as follows: Section 7.1 discusses additional applications of FaDE, Section 7.2 outlines our insights on the drawbacks of the standard arithmetic circuit model for view maintenance. Section 7.3 analyzes the approach of FaDE and Section 7.4 describes additional optimizations. Experimental results are presented in Section 7.5.

7.1 Additional Use Cases

In this section, we discuss additional applications of FaDE beyond complaint driven debugging, across query explanation, interactive visualization and probabilistic databases.

Query Explanation Scorpion [8] aims to explain anomalous aggregate query results. For example, a user of a sensor dataset may want to know why the temperature measurements for a group of sensors exhibits high variance. Similar to Rain but without any ML model component, this is

formulated as a *complaint* that states an output attribute value is too high (or low). In response, Scorpion returns predicates over the sensor measurements table, e.g. an id of a malfunctioning sensor, sensor measurements during specific parts of the day, etc. The predicates are chosen such that if the measurements satisfying them are deleted, the variance of temperature measurements decreases the most. Scorpion explores the search space of possible predicates by evaluating a batch of predicates at a time and then drilling down on the the most promising ones. Scorpion can make use of an efficient algorithm for Problem 4 to accelerate its batch evaluation step. Similar improvements can be applied to other query explanation works [113, 112].

Even though FaDE cannot accelerate query explanations for arbitrary data pipelines, it can be used to accelerate the evaluation of the its most expensive parts. Revisiting the sensors application, a user may compare the yearly temperature measurement histograms and observe they differ significantly. If she complaints that the KL divergence of the last two histograms should be smaller, FaDE can accelerate the repeated histogram updates and then pass the histograms to a UDF that calculates their KL divergence.

Interactive Cross-Filtering Prior work [115] aims to interactively update cross filtering based visualizations. In a typical cross filtering setup, users highlight data of interest in one view and the results of another view update to consider only the selected subset. While [115] already makes use of provenance metadata to identify the selected data in one view, it could also use Problem 4 instead of IVM to update the results of the other view. Gains from using Problem 4 can be even greater if multiple selection evaluations are batched for an visualization application using prefetching [116].

Probabilistic Databases Probabilistic databases [117] can evaluate queries over databases whose tuples may or may not exist with some probability. One way to provide answers to these queries is to sample database instances based on the uncertainty information of each tuple and evaluate the average query result. Probabilistic databases can use Problem 4 to quickly evaluate the queries over the sampled databases since they are all subsets of the full database.

The acceleration techniques and performance of FaDE are independent of how the input tuple deletions are chosen or their statistics. As a result, the considerations of FaDE are decoupled

from the upstream ML model components that control the provenance polynomial inputs. Thus to simplify our discussion in the following sections, we will focus on the application of FaDE on the relatively simpler problem of query explanation [8, 112, 113] where no ML models are involved.

7.2 Limitations and Opportunities

In Section 3.2.3 we discussed the arithmetic circuit representation of provenance metadata used by ProvSQL [24] and other works [70]. In this section, we will present the limitations of using the arithmetic circuit representation for the purposes of view maintenance. This analysis will directly motivate the design of FaDE in Section 7.3.

The arithmetic circuit representation and evaluation exhibits poor instruction and data locality. In terms of instruction locality, evaluating the circuit relies heavily on branching and pointer based accesses to figure out the next instruction to execute. This slows down modern hardware, limiting their potential for out of order execution. In terms of data locality, pointer based accesses lead to non-sequential memory accesses. These access patterns make poor use of CPU caches and the available memory bandwidth. This is because circuits nodes that are accessed together are not necessarily collocated in memory.

While optimizing the evaluation of an arbitrary arithmetic circuit is challenging, arithmetic circuits generated from SQL queries have access patterns that make them significantly easier to optimize. All of the nodes that correspond to the same query operator are annotated with the same operation type and access the same nodes. Thus, we can reorganize the execution of the circuit to evaluate these node operations together and store their corresponding inputs in contiguous locations. Instead of augmenting circuits with query operator information and then extracting the access patterns from the circuit, we instead design FaDE on top of the provenance representation of Smoke [23], which directly records per-operator provenance metadata as discussed in Section 3.2.

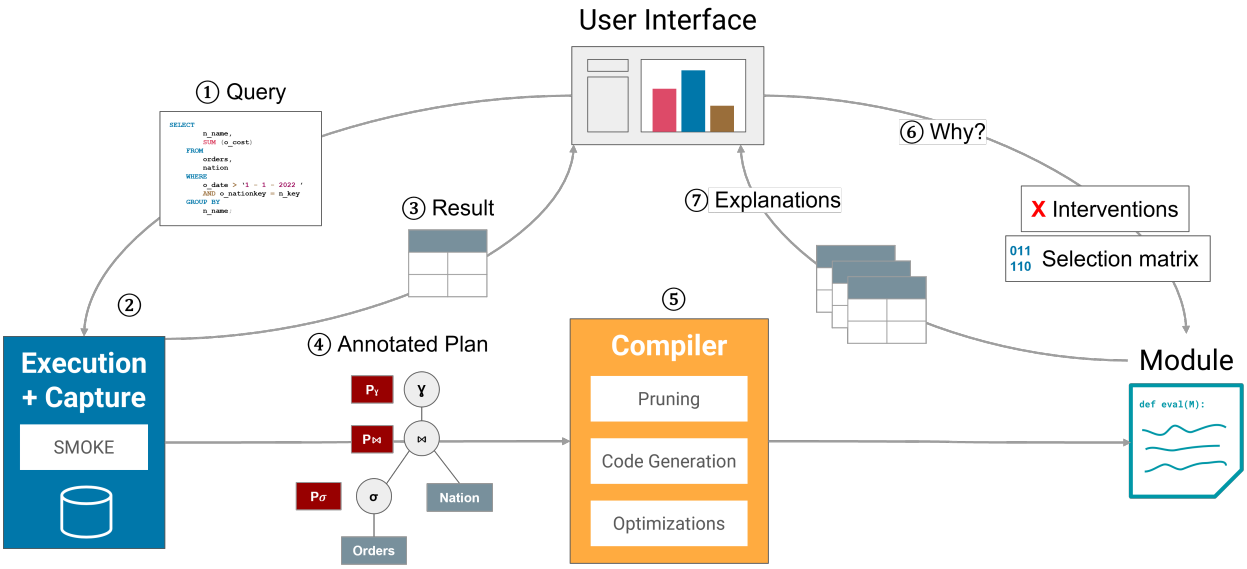


Figure 7.4: FaDE system architecture.

7.3 Approach

In this section, we describe the design of FaDE, our code generation engine which enables interactive time counterfactual evaluation. We discuss the code generation pipeline, intermediate data format, and evaluation logic for individual query operators.

7.3.1 System Architecture

As shown in Figure 7.4, FaDE generates code modules based on input queries to provide interactive time explanations for deletion interventions. Beginning with (1), a user executes a query which may require explanations. FaDE uses Smoke to execute the query under provenance capture (2). This process annotates the query plan with provenance tables, each containing the provenance information for one operator. The unmodified query result is returned back to the user interface for display (3), while the annotated query plan is passed to the code generation module (4).

During the code generation step (5), FaDE walks the annotated query plan and produces a linkable module which can perform fast intervention evaluations. For each query plan operator, FaDE generates an associated code snippet which computes the effect of deleting input tuples to that operator. Operators at leaf nodes are converted to accept interventions over the base tables as

input, and the root operator emits updated views. The updated views consist of either binary vectors corresponding to the deletion status of each output in the initial view or value vectors containing newly computed aggregate values. During this stage, we also apply optimizations on the generated code to increase performance. These optimizations include parallelization and provenance pruning, which we describe further in Section 7.4.

As the user explores the results of their query, they may encounter unexpected results and seek to answer the question "Why?". To interact with FaDE, query explanation systems convert these questions into possible input database interventions (6). Internally, we represent these interventions as binary selection vectors or matrices representing the deletion status of each row. These objects are dense representations of the existence bits from Chapter 1. Intermediate operators pass data around using similarly formatted vectors and matrices, which we further describe in Section 7.3.2. Finally, the output values of the generated module (7) can be directly linked into explanation systems [8, 112, 113]. The efficient loop of "Why?" questions to intervention generation to fast module evaluation enables interactivity in explanation pipelines.

Scope: FaDE currently accelerates view maintenance for SPJUA queries but it can be combined with an IVM system to handle unsupported queries. The unsupported query can be broken down in two steps: A collection of SPJUA views and then a query that operates on top of them. FaDE can then do view maintenance on the views of the first step and then propagate the deletions or updates to its outputs to an IVM engine that incrementally updates the query of the second step.

7.3.2 Selection Vectors and Matrices

The code FaDE generates for individual operators exchanges binary vectors and matrices representing the deletion status of individual tuples. For a single intervention, operators exchange binary vectors. Each selection vector S consists of binary annotations where

$$S[i] = \begin{cases} 0, & \text{if } i\text{-th row is deleted} \\ 1, & \text{otherwise.} \end{cases}$$

For each relation in its input, a FaDE operator accepts a selection vector as input, and produces a single selection vector as output. Users provide the initial selection vectors based on counterfactual interventions over the base tables participating in a query.

To encode multiple interventions, selection vectors can be combined into binary matrices, each consisting of R rows and C columns where R = # of relation rows and C = # of counterfactual interventions. The value of each element

$$S[i, j] = \begin{cases} 0, & \text{if } i\text{-th row for intervention } j \text{ is deleted} \\ 1, & \text{otherwise.} \end{cases}$$

We note that binary selection vectors and matrices are amenable to efficient memory representations. Each element takes a single bit and the underlying storage can be packed together. This operation saves space while also improving the computational performance of batch evaluation of multiple interventions.

7.3.3 Single Operator Design

Here we describe the set of composable relational operators. Each operator is parameterized by its provenance P, encoding exactly the provenance information gathered during query execution.

Select As discussed in Section 3.2.1, for the case of selections only one tuple is contributing to each output tuple. The generated code has to propagate the deletion status from the input to the corresponding output. Recall that $P_\sigma[i]$ contains the input tuple id that contributed to the i-th output. Using S as the selection input and R as the output, the generated code becomes

```
for i, p in enumerate(P_σ):
    R[i,:] = S[p,:]
```

where we use numpy-style notation to extend the operators to the intervention direction. In terms of data locality, all writes to R are sequential. Unfortunately for highly selective selections, accesses to S may not be sequential leading to suboptimal performance. Section 7.4.3 will alleviate this

problem by removing all unused tuples from operator inputs during compilation. As a result, no code needs to be generated for selections.

Join Each output row in a binary join is present if both contributing input rows are not deleted, so joins correspond to logical ANDs. Recall that $P_{\bowtie}[i]$ contains the ids of the tuples that contributed to the i-th output. Let A, B be the selection matrices corresponding to the two input relations for the join, the operator computes the output matrix R as:

```
for i, p_left, p_right in enumerate(P_⋈):
    R[i,:] = A[p_left,:] & B[p_right,:]
```

Observe that the logical AND required for computing $R[i,:]$ over 64 interventions takes a single instruction in a 64-bit platform, leading to significant throughput wins for batched executions. In terms of data locality, the access patterns over A and B follow exactly the ones of the physical operator that was traced during provenance capture. For example, nested loop joins scan A and B several times whereas a hash join using A as an outer table scans A once performing random accesses to B. FaDE can directly leverage improved physical operator designs with better data localities. In fact FaDE's code generation policies do not need to change in response to physical operator changes. The improved data localities are embedded through the values of $P_{\bowtie}$, not the generated code. Once again, Section 7.4.3 will further improve data locality of A and B accesses by removing unused tuples.

Aggregation We first focus on group by aggregation. Recall that $P_{\gamma}[i]$ contains for the i-th input tuple its group id. Provenance also records the columns participating in the aggregation. Here we will present an example for a sum aggregation over a column $P_{\gamma\ \text{values}}$. In addition to the aggregation values, aggregations compute the deletion status for each group tuple. This operation corresponds to a binary OR over the selection matrices of each contributing tuple. Let S be the input selection matrix. R is the output selection matrix having one row for each group and one bit per row for each intervention which is 0 initialized. Agg contains the aggregation values for each group also 0 initialized. The generated code for this group by sum is

```
for i, p in enumerate(P_γ):
    R[p,:]    |= S[i,:]
    Agg[p,:] += P_γ values[i] * S[i,:]
```

Similar to joins, the boolean OR instructions have much higher throughput for many interventions. Regarding the aggregation update, one could achieve an equivalent effect with branching on $S[i,:]$ one by one. The branchless version proposed here leads to simpler control flow and can be faster than the branching version on modern hardware despite doing more work. As we will see in Section 7.4.2, vectorization can lead to further improvements.

In terms of the access patterns, while the reads of S are sequential, the writes to Agg and R are not necessarily. For small number of groups this is not problematic because all the group data may fit in cache. Alternatively, one can generate a partition merge plan. In fact given P_γ, one can compute a partition that evenly splits the input data based on their group at code generation time. An advantage over traditional query execution is that the group counts are known at code generation time so one can pick the most appropriate plan. We leave applying this optimization to FaDE for future work.

Simple aggregations can be derived as special edge cases of groups bys with a single group. Similar to traditional query execution, when computing several aggregations we can fuse them in a single loop to avoid repeated scans of S.

Project For projections under bag semantics the input selection matrix is identical to the output one so no code generation is needed. Set semantics can be handled with a simple group by aggregation as above.

Union Following Section 3.2.1, we will focus on union implementations that do not interleave tuples and just concatenate tables. In terms of code generation, this translates to concatenating the corresponding selection matrices. One can avoid the redundant copies by fusing the union with its descendants. The children of the union operator can write to the union output directly.

7.4 Optimizations

In this section, we discuss properties about our system that make it particularly amenable to high performance and throughput. Specifically, we describe optimizations for multi-dimensional parallelism using both multi-threading and vectorization, as well as a technique we refer to as provenance pruning which provides upper bounds for execution SPJA queries that are linear in the join output size.

7.4.1 Multi-threading

For selections and joins in Section 7.3.3, we discussed how FaDE can generate code that computes the deletion status of each output tuple independently. This is a prime target for thread-level parallelization. Output tuples can be evenly split across threads and their deletion status can be computed independently, achieving perfect scaling. Observe that the simple paralleization of joins is in stark contrast to traditional query execution where multi threaded joins is a complex and extensively studied topic [118].

Parallelizing group by aggregations is more challenging. Following Section 7.3.3, FaDE partitions the inputs evenly across threads and then merges the resulting aggregations. An alternative is to use a partition merge design to improve data access locality. Once again, choosing between these approaches can be done in a principled way given that the groups statistics are known at code generation time. This optimization is left for future work.

7.4.2 Vectorization

One of the key properties about our selection matrices is that computation across interventions, i.e. over the columns, can be vectorized. As discussed in Section 7.3.3, each operator iterates over all interventions in its input matrix and performs computation. Since selection matrices place interventions in adjacent columns, operators are performing sequential scans across the columns of each accessed row. Instead of computing each intervention individually, we can load batches of

interventions and operate on them simultaneously with vectorized instruction sets built into most modern chips. While compilers can automatically vectorize simple programs, following prior work on vectorized operator execution [119], we manually design vectorized operators for FaDE. We observe that vectorization accounts for significant performance improvements in practice.

As an example, let us consider group by aggregation. Without vectorization, aggregates for each intervention need to be updated one by one. With vectorization, we can load the W aggregates chunks of $S[i, j : j + W]$, where W is a hardware supported width for vectorized instructions, into a vectorized register. Then, we call vector instructions to perform operations over the loaded values.

```
for i, p in enumerate(P_γ):
    for j in range(0, k, W):
        R[p,j:j+W] = vect_OR(R[p,j:j+W], S[i,j:j+W])
        _agg_val = vect_MASKLOAD(P_γ values[i],S[i,j:j+W])
        Agg[p,j:j+W] = vect_ADD(Agg[p,j:j+W],_agg_val)
```

For updating the R value matrix, we can perform a vectorized logical OR over blocks of interventions. For updating the `Agg` value matrix, we can load the value into a register using the input selection matrix as a bitmask. In this case, the intermediate value `_agg_val` is $P_{\gamma\ \text{values}}[i]$ wherever the selection matrix is 1 and 0 otherwise. Finally, we perform a vectorized addition function to update all the results in the `Agg` array.

7.4.3 Provenance Pruning

In contrast to arithmetic circuits of Section 3.2.3, FaDE evaluates the query plan in a bottom up fashion. As a result, FaDE may evaluate the deletion status of tuples that may not end up contributing to the final query result. A simple example is a selection filter, where the deletion status of tuples that fail the filter condition remains unused. The same applies to joins. This leads to wasted computation and suboptimal data locality, as CPU cache and memory bandwith is wasted on fetching data that is not used.

Pruning these tuples away can be done by traversing the query plan top down. Each operator receives from its parent a set of tuple ids that the parent uses. Then the operator removes the unused

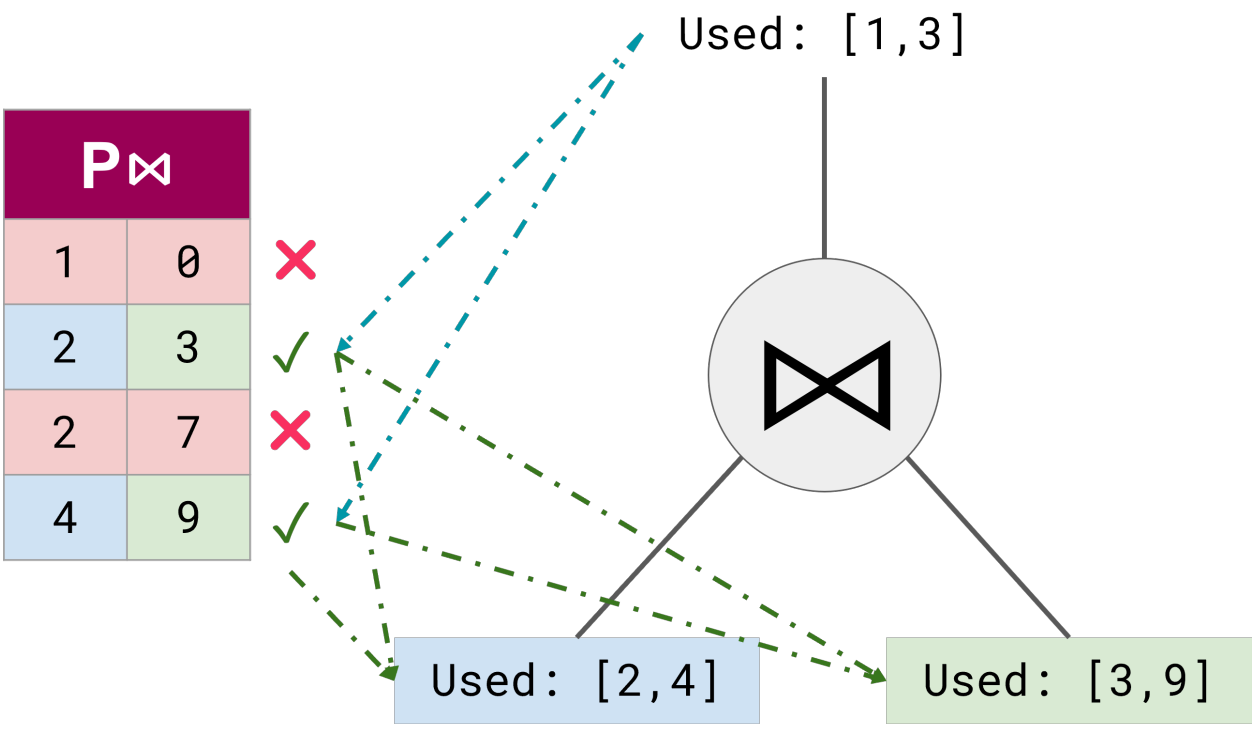


Figure 7.5: Provenance pruning propagates which tuples are used by the parent operator of a join to each of its children using the provenance metadata $P_{\bowtie}$.

tuple ids and computes among the remaining output tuples the set of input tuple ids used by its children. Then traversal continues recursively. For example, consider the code generation for the join operator in Figure 7.5. The parent of this join passes a used set which states that future operators only use tuples 1 and 3 from the join output. The join re-indexes to only consider these two outputs by removing tuples 0 and 2, and then informs the left and right children operators which tuples it needs from them. In this case, it needs tuples 2 and 4 from the left child (blue), and tuples 3 and 9 from the right child (green).

This pruning optimization can lead to significant performance gains. To quantify these gains, we analyze the worst case complexity of view maintenance using the pruned provenance data. For an SPJUA query executed only using binary joins, the computational complexity of executing the query can be polynomially higher than the final join output [120]. This is because intermediate joins can be polynomially larger than the final join output for an unfortunate join order. Here we show that after provenance pruning, each intermediate join cannot have size larger than the final

join output. Thus for all SPJU queries, provenance pruning guarantees linear complexity in terms of the final join output.

Theorem 1. *Given a fixed SPJU query with M tuples in the join output, the complexity of view maintenance under deletions using provenance pruning is $O(M)$.*

Proof. For selections, no work needs to be done. After pruning unused tuples, selections become identity mappings. The same applies to bag semantics projections. Bag unions can be eliminated as discussed in Section 7.3.3 as well. Considering the final binary join, since it has M tuples, at most M tuples of each of its descendants are actually needed and thus computed under provenance pruning. As a result all joins require only $O(M)$ binary ANDs which can be done in $O(M)$ time. All in all, the execution takes $O(M)$ for each join, which is $O(M)$ in total for a fixed query. □

7.5 Experiments

In this section, we experimentally evaluate the performance of FaDE against state of the art IVM and circuit based based view maintenance systems. We also perform ablation studies for the optimizations of Section 7.4. Our experimental results show that:

- When evaluating one deletion intervention at a time, FaDE is up to 600× faster than IVM, and 10,000× faster than circuit based provenance evaluation methods which can be even slower than running queries from scratch.
- FaDE can evaluate batches of interventions at a time increasing throughput up to 8× and up to an additional 5× using vectorization
- FaDE can efficiently distribute work allowing up to linear scaling with thread count.
- FaDE's performance scales with database size, enabling interactivity on query explanation workloads where it was not previously possible.

7.5.1 Experimental Settings

We now describe the experimental settings, including the systems compared, the workload and intervention parameters as well as implementation settings.

Systems

We compare FaDE's view maintenance latency on a single intervention of varying sizes against DBToaster [30]. We also include evaluations of ProvSQL [24] which is to the best of our knowledge, the only fine grained provenance based view maintenance system prior to our work. Given that both FaDE and DBToaster are main memory compilation based systems and that ProvSQL is based on PostgreSQL, for a fair comparison we compare ProvSQL to running the query on PostgreSQL from scratch. Specifically, we evaluate the following 5 systems:

DBT-full: We configure DBToaster to generate C++ code and to evaluate batches of individual tuple deletion events at a time. For each table used by the query, we apply all deletions of a single table in a single batch. In or latency measurements, we exclude code compilation as well as the loading of the initial database and measure only the time taken for deletions. We use the latest available version in the author's repository.

DBT-pruned: Same as DBT-full but we restrict the input database to the target query's why provenance, the subset of input database tuples that have contributed to the end result of the query. Results of IVM on this smaller database are guaranteed to be equivalent to the original one when applying deletion interventions. We first use Smoke [23] to derive each query's why provenance and filter the input tables before loading them to DBToaster.

ProvSQL: We run the ProvSQL extension on top of PostgreSQL 13. We only measure the arithmetic circuit evaluation time, skipping query execution and provenance capture costs. ProvSQL's top down evaluation already skips non-contributing input tuples and intermediates, so a pruned variant like DBToaster is not required.

PostreSQL: As a baseline for view maintenance systems, we measure the time taken to run the query on PostgreSQL 13 from scratch without applying any deletion to the input database.

View maintenance systems need to consistently beat from scratch execution to be practically useful. Surprisingly, we find that ProvSQL fails to do so in many instances.

FaDE: For our proposed system, we measure the runtime of the generated code, not provenance capture time of Smoke [23], code generation and compilation or the generation of the selection matrices given the interventions. We ablate the optimizations of Section 7.4 on a per experiment basis. For experiments involving multiple independent interventions, FaDE's vectorization optimizations rely on the AVX-512 instruction set.

The work of [113] is also relevant here as it can transform the incremental evaluation of several independent interventions into a single SQL query to achieve increased throughput. Unfortunately, the implementation of [113] is not openly available. While FaDE supports intervention batching, unlike [113] FaDE does not require batching to beat traditional IVM systems. We leave comparisons of throughput gains thanks to intervention batching to future work.

Query and Data Workload

To test the systems under a variety of analytical aggregation queries, we evaluate them across a variety of TPC-H queries. We restrict our attention to SPJUA queries that both IVM and provenance based view maintenance support. We leave extensions of FaDE to nested queries and the corresponding comparisons for future work. Regarding ORDER BY and LIMIT clauses, while they can be evaluated both by IVM and FaDE they do not benefit significantly by incremental evaluation and provenance metadata respectively. We vary the scale factor of the TPC-H database between 1, 5 and 10.

Interventions

End users, in their effort to understand what really affects query results, may want to perform complex interventions across many database tables. To simulate such a complexity, we choose interventions that randomly delete tuples from multiple tables at a time. To represent interventions of varying granularity, we vary the deletion probability of each tuple being deleted, always keeping the

same probability across tables. We note that the performance of FaDE is independent of the deletion patterns of the inputs so this choice of interventions does not affect our evaluation. Regarding IVM systems, random interventions represent an average case analysis. Each heavy hitter tuple, whose deletion leads to many deletions in the query's intermediates, has the same probability of getting picked as the rest of the tuples. Tuples to be deleted for DBToaster-pruned are randomly chosen over the remaining tuples in the database after pruning. For FaDE, our experiments also vary the number of interventions tested, up to 2560 simultaneously evaluated interventions.

Implementation

All our experiments are executed on Google Cloud `c2-standard-16` machines (16 vCPU, 64GB memory). Vectorized operations use AVX-512, a 512-bit SIMD instruction set on Intel processors. We generate and compile C++ programs using `GCC 11.2.0`. Our implementation makes use of threading primitives of the `C++20` standard library and compiler intrinsics for vectorization.

7.5.2 IVM and Intervention Size

In this first experiment, we compare the performance of FaDE and IVM baselines DBT-full and DBT-pruned. As noted in Chapter 1, while incremental evaluation is preferable for small interventions over the input database, view maintenance latency gets progressively higher as the intervention size increases. To experimentally verify this claim, we evaluate FaDE and the two IVM baselines using deletion probability 0.1. For both IVM baselines, we sample 10 random deletions and report averages. Figure 7.6 presents our findings for various TPC-H queries at scale factor 1. FaDE is configured to use a single thread and no vectorization to allow for a fair comparison with the IVM baselines.

In Figure 7.6, we find that FaDE is outperforming both baselines systems despite the fact that FaDE does not incrementally apply deletions. Since we observe that FaDE's performance advantage varies across queries, we perform a case by case analysis.

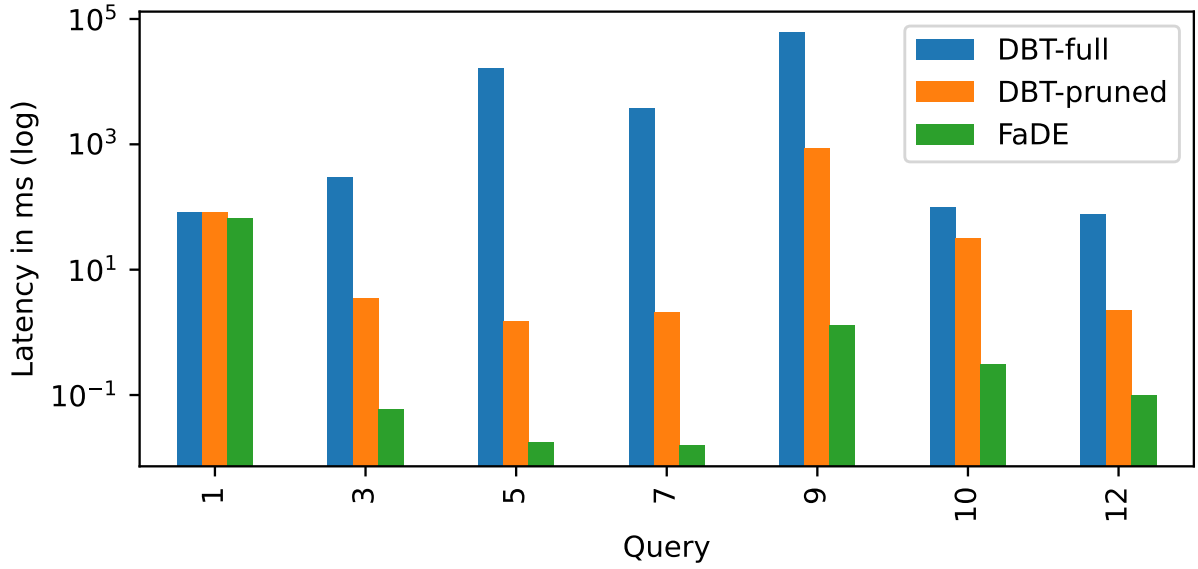


Figure 7.6: Latency of FaDE, DBT-full and DBT-pruned for deletion probability 0.1 for various TPC-H queries at scale factor 1.

The performance of all three systems is only comparable for Query 1, a group by aggregation with a single selection and without joins. The filter selectivity is low with 99% of the input tuples matching it. As a result, filtering on the why provenance gives limited gains with DBT-Full and DBT-pruned having the same latency. For a 10% deletion probability, both baselines have to process 10% of the input tuples while FaDE processes 100% of the input table to compute the same result. Despite this, FaDE is faster because it does not have to evaluate expensive filter conditions like date comparisons or the expressions that participate in the final aggregations. Both of them are computed once during provenance capture and are leveraged during counterfactual intervention evaluation.

All other queries evaluated here involve joins. We observe that FaDE has 20 to 660 times lower latency than DBT-pruned with performance gains being significantly more pronounced for queries with several joins like Query 5. Performance gains over DBT-full are even larger, ranging from hundreds of times lower latency for Query 10 up to a million times for Query 5.

Latency of IVM systems varies with intervention size so it is important to quantify the crossover point where deletions using IVM has lower latency than FaDE. Since DBT-pruned always has superior performance to DBT-full, we focus our comparisons between it and FaDE. For each of

the queries, we vary the deletion probability from 0.0005 to 0.1 and measure the deletion latency of DBT-pruned. In Figure 7.7 we plot the latency of DBT-pruned normalized by the latency of FaDE for the corresponding query. As expected for join-free queries, the latency for Query 1 of DBT-pruned grows linearly with deletion probability so DBT-pruned outperforms FaDE for deletion probabilities just below 0.1. For all other queries, we were not able to find a crossover point in this probability range with FaDE being at least 10× faster.

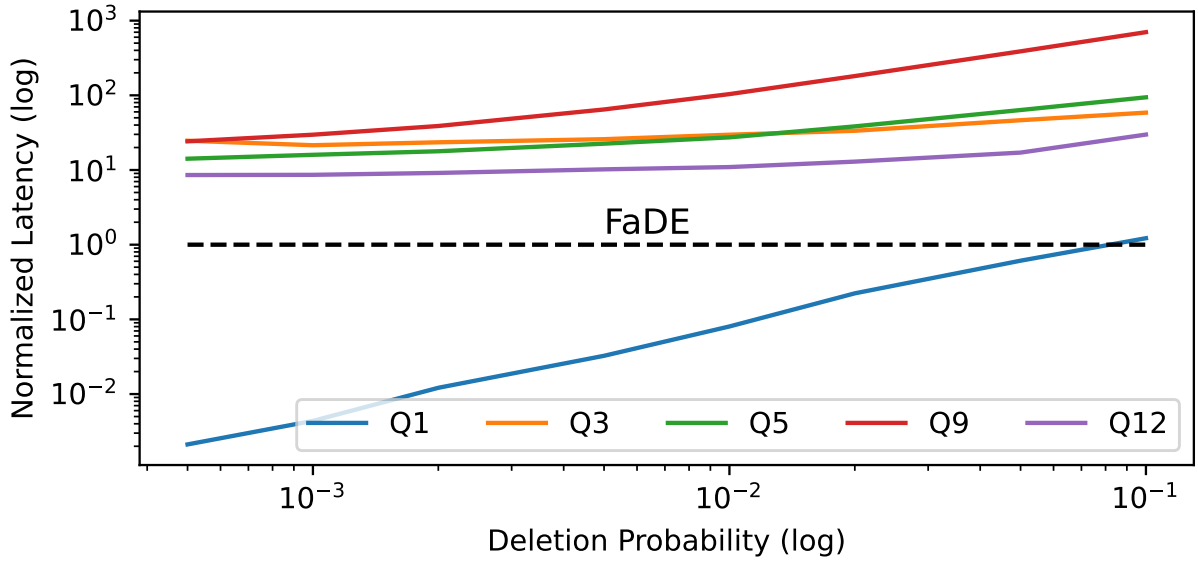


Figure 7.7: Latency of DBT-pruned normalized against FaDE with varying deletion probability for various TPC-H queries at scale factor 1.

Takeaway: Existing query explanation works use IVM in a naive way similar to DBT-full. Thus integrating FaDE can bring up to a million times latency improvements. FaDE's approach even without optimizations goes far beyond evaluating the query on its why provenance yielding up to 600× latency improvements over DBT-pruned.

7.5.3 Efficiency of Circuit Evaluation

In this section, we compare FaDE to the circuit approach of ProvSQL. In Figure 7.8, we evaluate the performance of ProvSQL without any deletions. Across all evaluated TPCH queries at scale factor 1, FaDE's latency improvements ranged from 900× for Query 7 up to 90000× for Query 5.

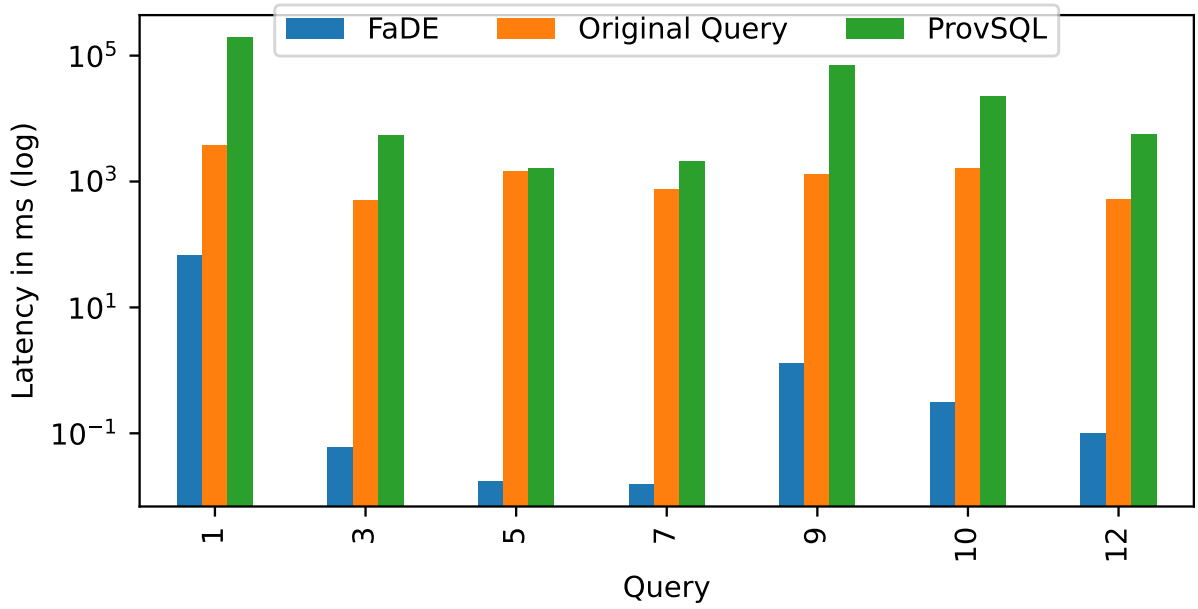


Figure 7.8: Latency of FaDE, the original query and ProvSQL without deletions for various TPC-H queries at scale factor 1.

The designs of FaDE and ProvSQL differ significantly. FaDE is main memory compilation based engine whereas ProvSQL is implemented using user defined functions and types and recursive SQL query on top of PostgreSQL. For a more fair comparison, we also include the run times of PostgreSQL on the same queries with no input deletions. Unfortunately, in Figure 7.8 we find that PostgreSQL's latency is always comparable or even up to 50× lower.

Takeaway: The circuit evaluation approach of ProvSQL does not outperform running the query from scratch on PostgreSQL and is orders of magnitude slower compared to FaDE.

7.5.4 Scaling with Multiple Interventions

Query explanation systems rely on running multiple counterfactual interventions. It is thus important for FaDE to provide high intervention evaluation throughput via intervention batching. The gains from intervention batching are twofold. First, the packed representation of selection vectors allows FaDE to use a single instruction for the binary AND and OR operations of independent interventions. Similar gains can be achieved for aggregation maintenance using vectorization, which is studied separately in Section 7.5.5. Second, for all operators the packed representation of selection

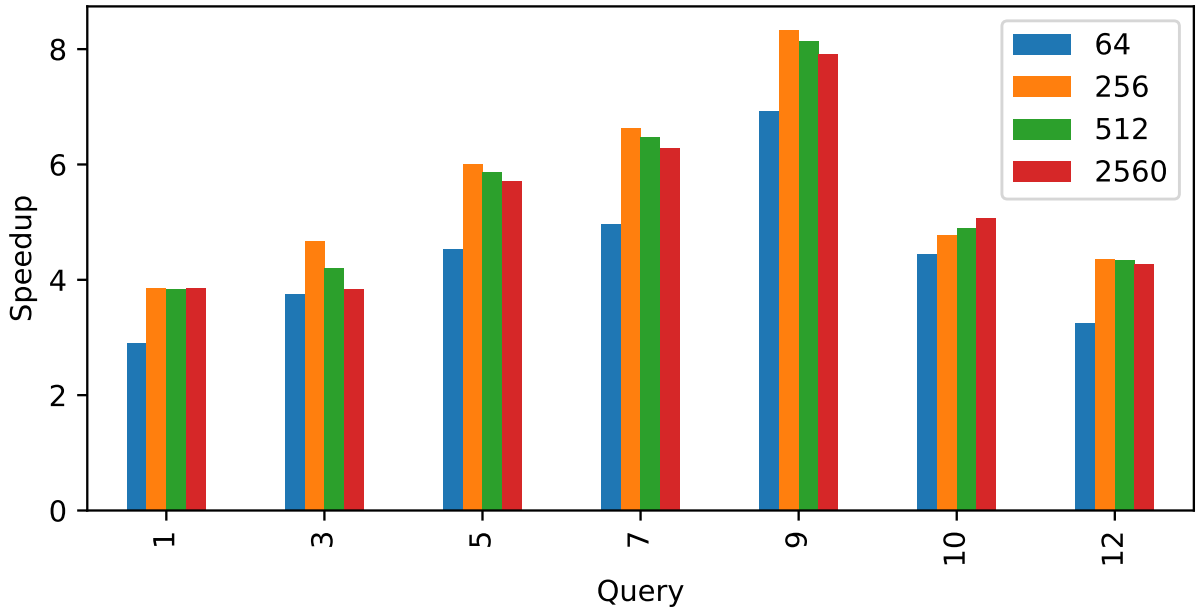


Figure 7.9: Batching speedup over single intervention performance of Figure 7.6 varying numbers of interventions.

vectors can also improve data locality and thus performance.

We evaluate the effects of intervention batching in Figure 7.9 varying the intervention count per batch without the use of FaDE's vectorization capabilities. For each batch size and query we measure the throughput gains as the intervention evaluation throughput normalized against the single intervention throughput.

Overall we observe that across all queries throughput gains range from 3 to 8 times. In absolute terms, these significant throughput gains can allow FaDE to evaluate tens of thousands of interventions for Queries 5 and 7 in less than 100ms using a single thread.

As shown in Figure 7.9, throughput gains vary among queries. As expected, queries with many joins like 5, 7 and 9 exhibit the highest throughput gains as they benefit both from instruction count and data locality improvements. While FaDE can evaluate up to 64 interventions with a single instruction for joins, throughput gains are less than 64 because aggregations become the bottleneck. For example, join free queries like Query 1 exhibit up to 4× gains. The vectorization improvements of Section 7.5.5 will allow FaDE to unlock further throughput gains.

We also observe in Figure 7.9 that throughput gains vary with batch size. The throughput

gains are significant for batch size 64 and 128 but further increases may even lead to throughput degradation. This is a result of two opposing phenomena. On the one hand, increasing the intervention count leads to more sequential accesses that improve the CPU's cache hit rate. After the cache hit rate reaches a saturation point, increasing interventions leads to diminishing returns. On the other hand, larger intervention counts means that less selection matrix rows can fit in cache. This increases the likelihood that selection rows that were previously in cache can be evicted leading to a cache miss when accessed again.

Takeaway: Batching interventions allows FaDE to improve its throughput up to 8× with gains being more significant for joins. This allows FaDE to evaluate tens of thousands of interventions in less 100ms enabling interactive debugging. Further gains, especially for aggregations, can be unlocked by using vectorization.

7.5.5 Vectorization Optimizations

In the previous section we discussed that further throughput gains were possible by employing vectorization especially for the case of aggregations. To test this claim we implemented vectorized join and aggregation operators in FaDE using the AVX-512 instruction set. In AVX-512, computing the AND of two 512 bit numbers, as required by joins, can be implemented with a single instruction. Similarly, adding two 16 element arrays of 32 bit floats or integers takes one instruction as well. In Figure 7.10 we measure the speed ups gained by vectorization for varying intervention counts over the performance of the unvectorized code of Section 7.5.4.

We see in Figure 7.10 that vectorization produces relative speedups in all queries ranging from 1.8× to over 5× speedup for Query 12. The gains are more pronounced for aggregations given that we could evaluate several interventions for joins with a single instruction even without AVX-512. While all queries have aggregations, some aggregations benefit more than others. Aggregation queries with small group counts benefit the most. For example, Queries 1, 5, 7, and 12 that have the highest gains have only 4, 5, 4, 2 output groups respectively. In contrast, Queries 3, 9, 10, which have the lowest speedups, have 10K, 175 and 30K groups respectively. This is expected as increased

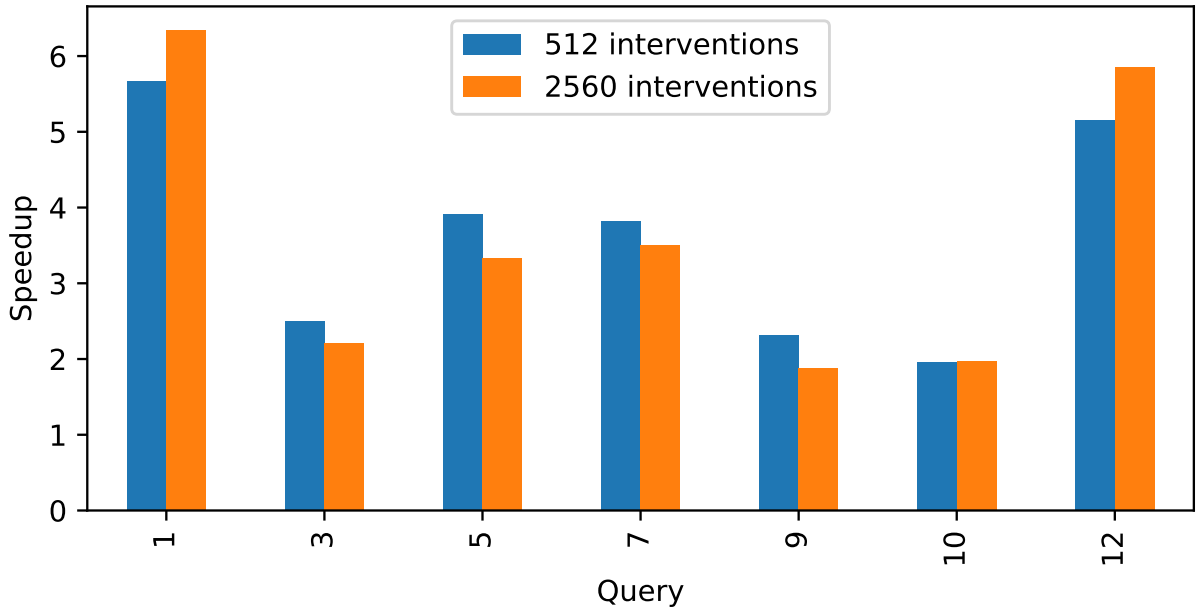


Figure 7.10: Additional vectorization speedup over batched execution of Figure 7.9 varying numbers of interventions.

group counts can lead to more cache misses unless the input data are not processed on a group by group basis. Partition/sort based group by operators can improve locality for large group counts and are left as promising future work.

Speedups for different intervention counts also differ based on the phenomena discussed in Section 7.5.4. Unlike in Section 7.5.4, we observe that increasing the batch size to 2560 can still give significant gains for some queries. Since the performance of FaDE is independent of the deletion patterns of the input, the batch size can be tuned in advance of the user interaction using synthetic inputs. For intervention counts smaller than 512, we cannot make full use of AVX-512 for joins. Instead we can opt to use AVX-512 only for the aggregations which requires only 16 interventions at a time. This gives more fine grained control over the throughput and latency tradeoff since joins can joins can still be executed 64 at a time without using any vectorized instruction.

Takeaway: Batching interventions using AVX-512 vectorization gives FaDE additional throughput improvements ranging from 1.8× to 5×. Vectorization is particularly useful for aggregations especially the ones with small group counts.

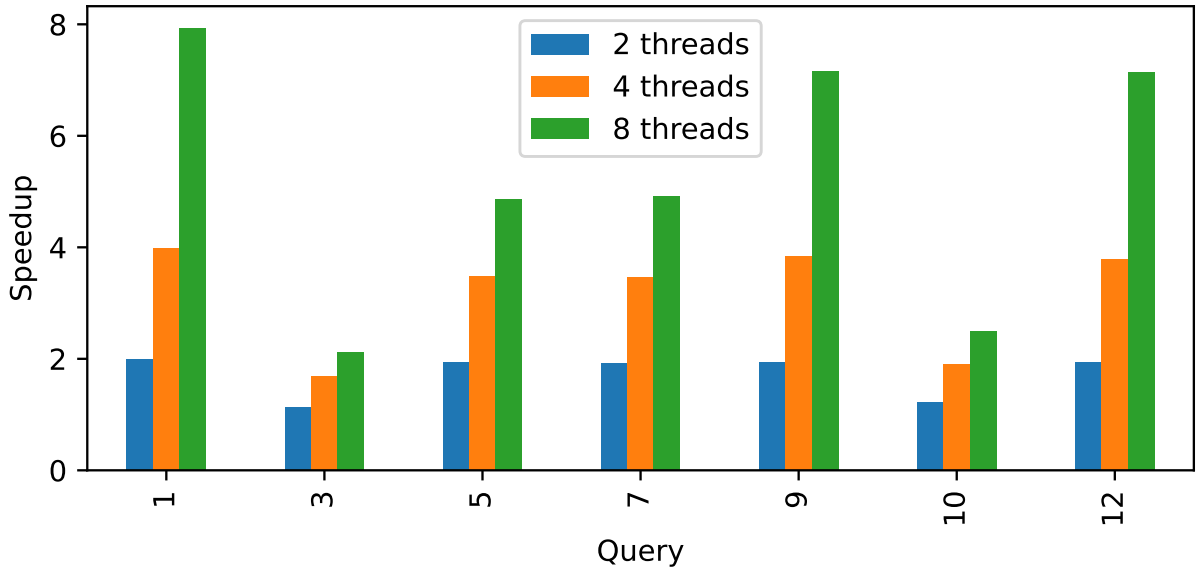


Figure 7.11: Additional threading speedup for 2, 4 and 8 threads over batched execution of Figure 7.9 for 2560 interventions.

7.5.6 Multithreading Optimizations

FaDE can always use thread level parallelism to evaluate multiple intervention batches at the same time. While this increases throughput, latency remains constant. In practice, query explanation systems like those in [8, 112] do not unconditionally evaluate a set of interventions in a batch fashion. Instead they use the results of initial interventions to prune the search space and converge faster to the desired explanation. It is thus critical that FaDE can use the available threads to reduce latency as well.

In this section, we empirically evaluate the performance gains from the threading optimizations of Section 7.4. To do this we configure FaDE to use 2, 4 and 8 threads and 2560 interventions without vectorization optimizations. We then measure the latency improvements over the single threaded case for each setting as seen in Figure 7.11. We choose 2560 interventions because we find that for fewer interventions fast queries exhibit more limited speedups from threading due to synchronization overhead.

For up to 4 threads all queries except Query 3 and 10 exhibit close to linear scaling with thread

count. This is because both queries have large a large output group count to input tuple ratio for their aggregations. Our parallelization strategy first splits the aggregation input evenly across threads and then assigns each thread a subset of all output groups to merge the contributions from all threads. Unless the ratio of input tuples to group counts is above the thread count, the merge step needs to aggregate more data than the original aggregation. We leave partition/sort merge group by implementation for future work.

Regarding 8 threads, we find that close to linear scaling persists for the rest of the queries except Query 5 and 7. Among the queries evaluated, these two queries had the lowest latency in Figure 7.6 and showed further significant throughput improvements in Figure 7.9. Their particularly low runtimes make them vulnerable to synchronization overhead that increases with thread count. In our experiments we have identified that both larger intervention counts and TPC-H scale factors lead to larger speed ups for 8 threads for these queries.

Takeaway: FaDE's execution model allows to evenly distribute the work of each operator across threads enabling close to linear scaling for many settings. Introducing partition/sort merge group by operators can lead to further gains from multithreading. These are well suited for FaDE since partitions/sorts can be computed once during code generation and reused during execution.

7.5.7 Combined Optimizations

Multihreading and vectrorization parallelize the execution of FaDE across different axes. FaDE splits the computation of the outputs of a single operator across threads. Then each thread can employ vectorization to compute the outputs assigned to it across different interventions. Thus FaDE can combine the speedups from both vectroization and multithreading to get the products of the individual speedups in the best case.

In Figure 7.12 we evaluate the speedups enabled by vectorization, as discussed in Section 7.5.5, multithreading for 8 threads, as dicusssed in Section 7.5.6 and combining both. We observe that combining optimizations gives improvements that are bigger than using vectorization or multi-threading alone. Query 1 exhibits the highest combined improvements. Vectorization is offering

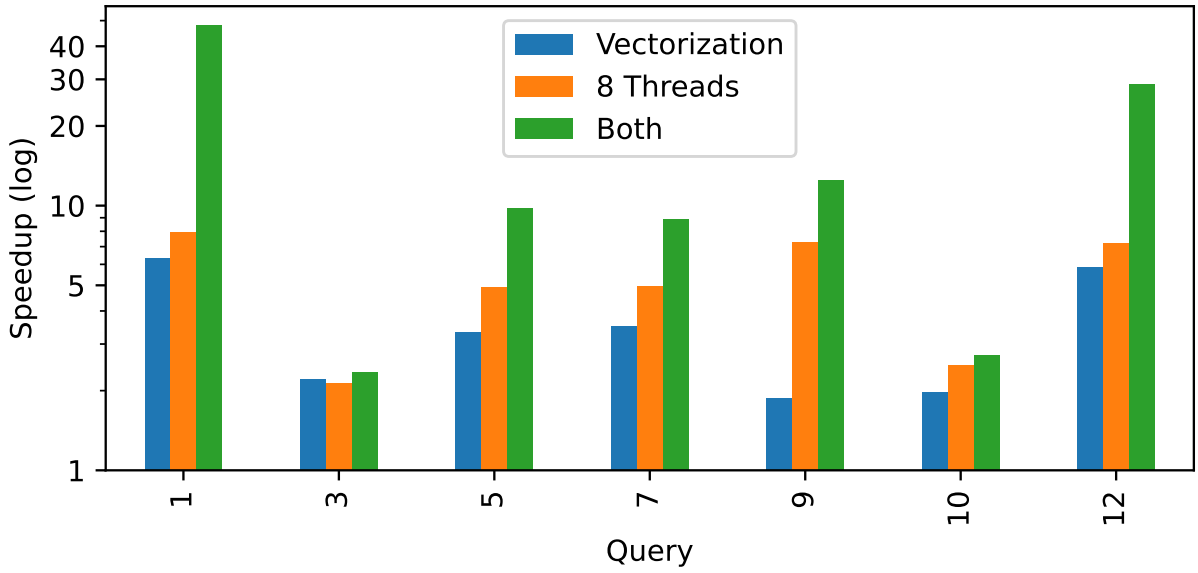


Figure 7.12: Additional vectrorization, multithreading and combined speedups over batched execution of Figure 7.9 for 2560 interventions.

6.3× improvements, multithreading 7.8× and combining both 48.2× close to the product of 49.1×. Query 12 is exhibiting 28× combined gains improving on the 5.8× and 7.2× improvements from vectorization . Queries 5 and 7, as discussed in Section 7.5.6 are bottlenecked by synchronization overhead due to their already low latency. Vectorization reduces the per thread latency even further but synchronization overhead prevents further gains. As discussed in Section 7.5.5, Queries 3, 9 and 10 exhibit low data locality due to their large output group counts. Further acceleration with multithreading is limited as the last level CPU cache and memory bandwidth are shared across threads.

Takeaway: FaDE can leverage multithreading and vectorization to get combined improvements that are bigger than using one of them alone. High degrees of data locality can enable FaDE to get even the product of individual improvements.

7.5.8 Scaling with Data Size

As the scale of the data grows, accounting for data locality becomes more important as the fraction of the data that can be cache resident at any point in time shrinks. To verify FaDE's

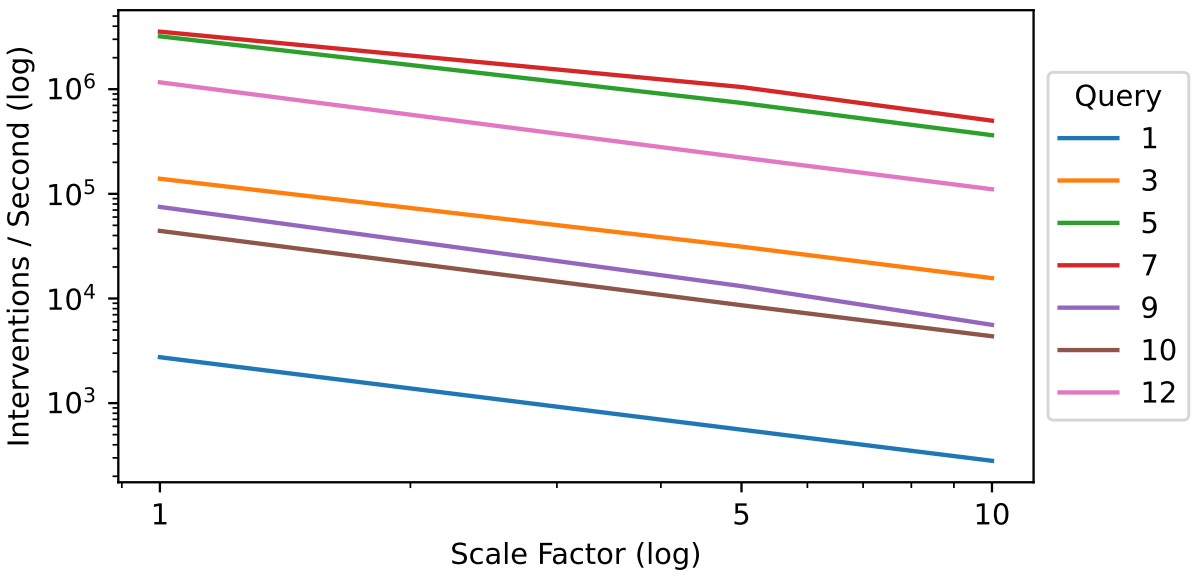


Figure 7.13: Throughput of intervention evaluation using 8 threads and vectorization for various TPC-H queries at scale factors 1, 5 and 10.

scalability, we vary the scale factor of the TPC-H database between 1, 5, and 10 and configure FaDE to use 8 threads and vectorization for 2560 interventions. In Figure 7.13, we plot the throughput of each query against the database scale. We observe that for all queries, throughput decreases linearly with increasing scale factor. Even at scale factor 10, FaDE can evaluate from hundreds up to half a million of interventions per second for Queries 1 and 5 respectively.

FaDE for Query Explanation We now contextualize our performance results using the query explanation application from Roy et al.[113]. In their NSF award application, the user runs two queries over the Awards and Institution tables to compute the Computer Science NSF funding for UIUC and CMU. Each table has 400K tuples. Then the users asks why UIUC has received more Computer Science funding from NSF than CMU. To provide explanations, [113] evaluates counterfactual interventions over the two queries. Each query filters the Award and Institution tables joins them and computes an aggregate. Then 188K counterfactual interventions are generated that collectively delete 1.7M tuples from the Award table. Their batched IVM approach takes 1.6s to evaluate the interventions on both queries despite each explanation deleting less than 10 tuples on average.

In contrast, our SF=1 results show that FaDE can evaluate millions of interventions per second for complex join queries over tables of similar size. Thus, FaDE evaluates all interventions and generates explanations in an interactive latency [90, 89] of $<100ms$.

Takeaway: Throughput gains from FaDE scale linearly with data size, allowing it to perform interactive time explanations over larger datasets than previously possible.

7.5.9 Code Generation Overhead

In FaDE's architecture, code generation is a one time cost incurred after initial query execution and before the user asks for an arbitrary number of query explanations. We profile FaDE's code generation times for all tested queries. We measure time taken to traverse the query plan, process provenance tables, and generate C++ code. For scale factor 1, generation takes between 0.5 and 4 seconds. For scale factor 10, generation takes between 3 and 40 seconds. We do not observe any significant increases in generation times between different optimization settings.

Takeaway: Although the current implementation of FaDE is not optimized for code generation performance, it performs code generation for tested queries in a few seconds.

Conclusion

In Chapter 1 we set out to make the first steps towards addressing the challenges of complaint driven training data debugging. Throughout this thesis we restricted our attention to relational workflows as a stepping stone towards debugging more general workflows.

In Chapter 4 we introduced Rain the first system for complaint driven training data debugging for relational workflows. Instead of debugging the relational workflow and ML model components independently, Rain jointly reasons about both components in a holistic fashion. To achieve this, Rain relaxes the whole pipeline, from model training to the user's complaint, in a single differentiable function. This end to end approach allows Rain to debug complaints on aggregate results, which despite being ambiguous, can be as effective at recovering corrupted training records as specifying the labels of hundreds of individual serving predictions.

In Chapter 5 we proposed Rain++, a system seeking to reduce the high latency of Rain when scaling to large models or training sets. Rain++ achieves this by pushing a significant portion of Rain's computation offline ahead of user interaction time. Rain++ leverages insights from the optimization literature to precompute the sensitivity information of the model parameters and the serving data predictions in a compressed format. This not only reduces the onerous space requirements when targeting modern neural networks of millions of parameters but also significantly reduces Rain++'s online latency, with up to 70000× latency improvements over Rain.

In Chapter 6 we proposed MetaRain, a framework for training classifiers that detect training data corruptions in response to user downstream complaints. This is in contrast to Rain and Rain++ that rank interventions on individual training examples without reasoning about training data corruption

patterns or the joint effect of the individual interventions. Our experiments indicate that adapting the design of corruption detection classifiers, including their architecture and input features, to the corruption and complaint at hand leads to improved debugging performance. This highlights the importance of MetaRain's generality compared to the one size fits all approaches in prior work.

In Chapter 7 we proposed FaDE, a compiler that generates efficient code for relational view maintenance under input tuple deletions. FaDE can be leveraged by any complaint driven training data debugging technique that relies on running the relational workflow several times in response to model prediction changes. Rain and Rain++ can also leverage the techniques of FaDE to generate efficient code for computing the sensitivity of the relaxed workflow. The key advantage of FaDE's approach is that it allows for small code size with improved data locality and easy parallelization leading to orders of magnitude latency improvements over existing view maintenance systems.

Future Work

Undoubtedly there is room for improvement across all problems that we tackled in each of the chapters above. Beyond these problems, here we highlight additional directions for future work.

Sensitivities for General Pipelines In this thesis, the key to data debugging is a way to assess the sensitivity of pipeline results to interventions over the training data. While black-box approaches like BOExplain [33] that repeatedly run the pipeline for each intervention are always applicable, they are also inefficient. Rain showed how fine-grained SQL lineage can be used to efficiently estimate end-to-end sensitivity. However, given that applications inevitably combine custom code and relational operators, extending Rain's framework beyond SQL remains critical.

Symbolic execution is promising because it generalizes lineage to arbitrary programs and is widely used for constraint-based program debugging. For example, given a Spark pipeline Acorn [121] can extract UDFs and symbolically execute them. The challenge is that the number of execution paths can increase exponentially with the program size. Alternative solutions are a) building a library of differentiable primitive algorithms similar to what Rain did for relational

algebra and b) replacing UDFs with differentiable proxies learned from input, output pairs.

Error Localization In this thesis we focused on training data debugging, and we have also assumed that downstream errors are the direct and *sole* result of training data errors. In practice, errors can arise in the inference data, the pipeline code, or in the training data pre-processing logic. Even assuming all interventions resolving the complaint are known, ranking incomparable interventions like modifying code versus data remains challenging. This becomes even more cumbersome when synergies between intervention types are needed to resolve the complaint.

Usability Data debugging is a highly iterative process, and one of the major challenges is how to design the appropriate interface that 1) matches the user's expertise, and 2) minimizes the amount of user effort. Although traditional cleaning approaches were overwhelmingly designed for programmers, there is increasingly a diversity of roles, technical backgrounds, and use cases in today's data pipelines that demand greater emphasis on usability. Below, we list a number of considerations that we feel are important.

Visualization and User Interfaces. Visualization tools such as TensorBoard and TSNE have shown the immense power of data visualization as a tool for exploring, understanding, and identifying errors in ML models and data. What are the appropriate visualizations or user interfaces to aid complaint driven data debugging? How can data debugging be integrated into existing end-user interfaces, such as radiology rendering software or BI tools?

Explainable Debugging. Complaint driven data debugging is not an exact science, and users must reason about how proposed interventions were chosen and be presented with enough information to decide whether to accept an intervention or not. Further, how can the user know that addressing a complaint will not introduce new data errors or affect a different complaint? For instance, a model may learn to compensate for errors in an upstream model, thus fixing any individual model may degrade the application quality.

Integration into Software Engineering and MLOps. Data debugging is performed within the broader software ecosystem within an organization, and should ideally integrate into existing (or future) software engineering workflows as well as ML operations. For instance, how does data debugging fit into continuous testing and integration processes? Systems like MLTrace [122] and ModelDB [123] track coarse-grained lineage and versioning; where does data debugging infrastructure fit in?

References

[1] D. Baylor *et al.*, "TFX: A tensorflow-based production-scale machine learning platform," in *SIGKDD*, 2017 (cit. on p. 1).

[2] Uber, *Meet michelangelo: Uber's machine learning platform*, https://eng.uber.com/michelangelo/, 2019 (cit. on p. 1).

[3] Facebook, *Introducing fblearner flow: Facebook's ai backbone*, https://engineering.fb.com/core-data/introducing-fblearner-flow-facebook-s-ai-backbone/, 2016 (cit. on p. 1).

[4] C. Ré, F. Niu, P. Gudipati, and C. Srisuwananukorn, "Overton: A data system for monitoring and improving machine-learned products," *arxiv*, 2019 (cit. on pp. 1, 14).

[5] MLflow, *Mlflow - a platform for the machine learning lifecycle*, https://mlflow.org/, 2019 (cit. on p. 1).

[6] S. Schelter, J.-H. Böse, J. Kirschnick, T. Klein, and S. Seufert, "Automatically tracking metadata and provenance of machine learning experiments," in *ML Systems@NeurIPS*, 2017 (cit. on p. 1).

[7] A. Chapman and H. Jagadish, "Why not?" In *SIGMOD*, 2009 (cit. on p. 2).

[8] E. Wu and S. Madden, "Scorpion: Explaining away outliers in aggregate queries," *VLDB*, 2013 (cit. on pp. 2, 17, 48, 132, 133, 135, 137, 154).

[9] A. Meliou and D. Suciu, "Tiresias: The database oracle for how-to queries," in *SIGMOD*, 2012 (cit. on pp. 2, 48, 52).

[10] E. Breck, N. Polyzotis, S. Roy, S. Whang, and M. Zinkevich, "Data validation for machine learning.," in *MLSys*, 2019 (cit. on pp. 2, 3).

[11] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Bießmann, and A. Grafberger, "Automating large-scale data quality verification," *VLDB*, 2018 (cit. on p. 2).

[12] P. W. Koh and P. Liang, "Understanding black-box predictions via influence functions," in *ICML*, 2017 (cit. on pp. 5, 39, 53, 56, 58, 67–69, 172, 175).

[13] X. Zhang, X. Zhu, and S. J. Wright, "Training set debugging using trusted items," in *AAAI*, 2018 (cit. on pp. 5, 39, 40, 48, 53, 56, 104, 106–108, 110, 112, 121).

[14] Z.-Q. Zhao, P. Zheng, S.-T. Xu, and X. Wu, "Object detection with deep learning: A review," *IEEE Trans. Neural Networks Learn. Syst*, 2019 (cit. on pp. 7, 25).

[15] S. Minaee, Y. Boykov, F. Porikli, A. Plaza, N. Kehtarnavaz, and D. Terzopoulos, "Image segmentation using deep learning: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, 2022 (cit. on pp. 7, 25).

[16] D. W. Otter, J. R. Medina, and J. K. Kalita, "A survey of the usages of deep learning for natural language processing," *IEEE Trans. Neural Networks Learn. Syst.*, 2021 (cit. on pp. 7, 25).

[17] D. Maclaurin, D. Duvenaud, and R. P. Adams, "Gradient-based hyperparameter optimization through reversible learning," in *ICML*, ser. JMLR Workshop and Conference Proceedings, 2015 (cit. on pp. 9, 32).

[18] I. F. Ilyas and X. Chu, *Data cleaning*. Morgan & Claypool, 2019 (cit. on pp. 9, 24, 33).

[19] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *J. Big Data*, 2019 (cit. on pp. 9, 32).

[20] F. Zhuang *et al.*, "A comprehensive survey on transfer learning," *Proc. IEEE*, 2020 (cit. on pp. 9, 32, 73).

[21] R. D. Cook and S. Weisberg, "Characterizations of an empirical influence function for detecting influential cases in regression," *Technometrics*, 1980 (cit. on pp. 9, 35).

[22] A. W. Van der Vaart, *Asymptotic statistics*. Cambridge university press, 2000 (cit. on pp. 9, 35).

[23] F. Psallidas and E. Wu, "Smoke: Fine-grained lineage at interactive speed," *VLDB*, 2018 (cit. on pp. 9, 22, 42, 132, 135, 145, 146).

[24] P. Senellart, L. Jachiet, S. Maniu, and Y. Ramusat, "Provsql: Provenance and probability management in postgresql," *VLDB*, 2018 (cit. on pp. 9, 13, 42, 45, 132, 133, 135, 145).

[25] B. S. Arab, S. Feng, B. Glavic, S. Lee, X. Niu, and Q. Zeng, "Gprom - A swiss army knife for your provenance needs," *IEEE Data Eng. Bull.*, 2018 (cit. on pp. 9, 42).

[26] B. Glavic and G. Alonso, "Perm: Processing provenance and data on the same data model through query rewriting," in *ICDE*, 2009 (cit. on pp. 9, 42).

[27] T. Müller, B. Dietrich, and T. Grust, "You say 'what', I hear 'where' and 'why'? (mis-)interpreting SQL to derive fine-grained provenance," *VLDB*, 2018 (cit. on pp. 9, 42).

[28] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *PODS*, 2007 (cit. on pp. 9, 13, 44, 52, 130).

[29] Y. Amsterdamer, D. Deutch, and V. Tannen, "Provenance for aggregate queries," in *PODS*, M. Lenzerini and T. Schwentick, Eds., 2011 (cit. on pp. 9, 13, 44, 45, 52, 130).

[30] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic, "Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views," *VLDB*, 2012 (cit. on pp. 13, 29, 132, 133, 145).

[31] P. Varma, B. Hancock, S. Suri, J. Dunnman, and C. Ré, *Debugging training data for software 2.0*, https://dawn.cs.stanford.edu/2018/08/30/debugging2/, 2018 (cit. on p. 14).

[32] A. Karpathy, *Software 2.0*, https://medium.com/@karpathy/software-2-0-a64152b37c35, 2017 (cit. on p. 14).

[33] B. Lockhart, J. Peng, W. Wu, J. Wang, and E. Wu, "Explaining inference queries with bayesian optimization," *VLDB*, 2021 (cit. on pp. 17, 103, 107, 108, 122, 160).

[34] R. Lourenço, J. Freire, and D. E. Shasha, "Bugdoc: Algorithms to debug computational processes," in *SIGMOD*, 2020 (cit. on p. 17).

[35] S. Galhotra, A. Fariha, R. Lourenço, J. Freire, A. Meliou, and D. Srivastava, "Dataprism: Exposing disconnect between data and systems," in *SIGMOD*, 2022 (cit. on p. 18).

[36] A. Fariha, S. Nath, and A. Meliou, "Causality-guided adaptive interventional debugging," in *SIGMOD*, 2020 (cit. on p. 18).

[37] *Apache Hadoop*, https://hadoop.apache.org/, Accessed: 2023-01-30 (cit. on p. 19).

[38] *Apache Spark™ - unified engine for large-scale data analytics*, https://hadoop.apache.org/, Accessed: 2023-01-30 (cit. on p. 19).

[39] M. Zaharia, A. Ghodsi, R. Xin, and M. Armbrust, "Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics," in *CIDR*, 2021 (cit. on p. 19).

[40] *dbt - transform data in your warehouse*, https://www.getdbt.com/, Accessed: 2021-08-26 (cit. on p. 19).

[41] *Amazon SageMaker feature store for machine learning (ml) – Amazon Web Services*, https://aws.amazon.com/sagemaker/feature-store/, Accessed: 2023-01-3 (cit. on p. 19).

[42] *Business intelligence and analytics software*, https://www.tableau.com/, Accessed: 2021-08-31 (cit. on p. 19).

[43] *Data visualization | Microsoft power bi*, https://powerbi.microsoft.com/en-us/, Accessed: 2021-08-26 (cit. on p. 19).

[44] J. M. Hellerstein *et al.*, "The madlib analytics library or MAD skills, the SQL," *VLDB*, 2012 (cit. on p. 19).

[45] M. Nikolic, H. Zhang, A. Kara, and D. Olteanu, "F-IVM: learning over fast-evolving relational data," in *SIGMOD*, 2020 (cit. on p. 19).

[46] SQLFlow, *Sqlflow: Bridging data and ai*, https://sqlflow.org, 2019 (cit. on p. 19).

[47] A. Agrawal *et al.*, "Cloudy with high chance of DBMS: a 10-year prediction for enterprise-grade ML," in *CIDR*, 2020 (cit. on p. 19).

[48] G. LLC, *Introduction to bigquery ml*, https://cloud.google.com/bigquery-ml/docs/bigqueryml-intro, 2019 (cit. on p. 19).

[49] *The create model statement | bigquery ml | google cloud*, https://cloud.google.com/bigquery-ml/docs/reference/standard-sql/bigqueryml-syntax-create, Accessed: 2023-01-30 (cit. on p. 19).

[50] O. N. N. Exchange, *Onnx*, https://onnx.ai/, 2019 (cit. on p. 19).

[51] Logicblox, *Logicblox – next generation analytics applications*, https://logicblox.com, 2019 (cit. on p. 19).

[52] Y. Li *et al.*, "Subjective databases," *VLDB*, 2019 (cit. on p. 19).

[53] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan, "Mlbase: A distributed machine-learning system," in *CIDR*, 2013 (cit. on p. 19).

[54] M. Boehm *et al.*, "SystemML: Declarative machine learning on spark," *VLDB*, 2016 (cit. on p. 19).

[55] Y. Lu, A. Chowdhery, S. Kandula, and S. Chaudhuri, "Accelerating machine learning inference with probabilistic predicates," in *SIGMOD*, 2018 (cit. on p. 19).

[56] D. Jankov *et al.*, "Declarative recursive computation on an rdbms, or, why you should use a database for distributed machine learning," *arXiv*, 2019. eprint: 1904.11121 (cit. on p. 19).

[57] D. Zeng, K. Liu, Y. Chen, and J. Zhao, "Distant supervision for relation extraction via piecewise convolutional neural networks," in *EMNLP*, 2015 (cit. on p. 21).

[58] F. Psallidas *et al.*, "Data science through the looking glass: Analysis of millions of github notebooks and ML.NET pipelines," *SIGMOD Rec.*, 2022 (cit. on p. 24).

[59] B. Hilprecht, C. Hammacher, E. Reis, M. Abdelaal, and C. Binnig, "Diffml: End-to-end differentiable ML pipelines," *arxiv*, 2022. eprint: 2207.01269 (cit. on pp. 24, 25).

[60] B. Karlas *et al.*, "Data debugging with shapley importance over end-to-end machine learning pipelines," *arxiv*, 2022. eprint: 2204.11131 (cit. on pp. 24, 25).

[61] G. Yu *et al.*, "Windtunnel: Towards differentiable ML pipelines beyond a single model," *VLDB*, 2021 (cit. on pp. 24, 25).

[62] F. Neutatz, B. Chen, Z. Abedjan, and E. Wu, "From cleaning before ML to cleaning for ML," *IEEE Data Eng. Bull.*, 2021 (cit. on pp. 31, 33).

[63] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg, "Activeclean: Interactive data cleaning for statistical modeling," *VLDB*, 2016 (cit. on p. 33).

[64] R. Giordano, W. T. Stephenson, R. Liu, M. I. Jordan, and T. Broderick, "A swiss army infinitesimal jackknife," in *AISTATS*, 2019 (cit. on p. 37).

[65] P. W. Koh, K. Ang, H. H. K. Teo, and P. Liang, "On the accuracy of influence functions for measuring group effects," in *NeurIPS*, 2019 (cit. on p. 37).

[66] J. Martens, "Deep learning via hessian-free optimization," in *ICML*, 2010 (cit. on p. 39).

[67] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems1," *Journal of Research of the National Bureau of Standards*, 1952 (cit. on p. 39).

[68] M. Ren, W. Zeng, B. Yang, and R. Urtasun, "Learning to reweight examples for robust deep learning," in *ICML*, 2018 (cit. on pp. 39, 40, 104, 106–108).

[69] J. Shu *et al.*, "Meta-weight-net: Learning an explicit mapping for sample weighting," in *NeurIPS*, 2019 (cit. on pp. 39, 41, 104, 106–108, 112, 117).

[70] D. Deutch, T. Milo, S. Roy, and V. Tannen, "Circuits for datalog provenance," in *ICDT*, 2014 (cit. on pp. 42, 130, 135).

[71] S. Roy, L. Orr, and D. Suciu, "Explaining query answers with explanation-ready databases," *VLDB*, 2015 (cit. on p. 48).

[72] F. Abuzaid *et al.*, "DIFF: A relational interface for large-scale data explanation," *VLDB*, 2018 (cit. on p. 48).

[73] R. Khanna, B. Kim, J. Ghosh, and S. Koyejo, "Interpreting black box predictions using fisher kernels," in *AISTATS*, 2019 (cit. on pp. 48, 53).

[74] L. Gurobi Optimization, *Gurobi optimizer reference manual*, http://www.gurobi.com, 2019 (cit. on p. 52).

[75] I. ILOG, *Cplex optimization studio*, http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer, 2014 (cit. on p. 52).

[76] S. Hara, A. Nitanda, and T. Maehara, "Data cleansing for models trained with SGD," in *NeurIPS*, 2019 (cit. on pp. 53, 56, 69).

[77] N. N. Dalvi and D. Suciu, "Efficient query evaluation on probabilistic databases," in *VLDB*, 2004 (cit. on p. 55).

[78] B. Kanagal, J. Li, and A. Deshpande, "Sensitivity analysis and explanations for robust query evaluation in probabilistic databases," in *SIGMOD*, 2011 (cit. on pp. 55, 56).

[79] J. Xu, Z. Zhang, T. Friedman, Y. Liang, and G. Van den Broeck, "A semantic loss function for deep learning with symbolic knowledge," in *ICML*, 2018 (cit. on p. 56).

[80] S. Das *et al.*, *The magellan data repository*, https://sites.google.com/site/anhaidgroup/projects/data (cit. on p. 58).

[81] P. Konda *et al.*, "Magellan: Toward building entity matching management systems," *VLDB*, 2016 (cit. on p. 58).

[82] D. Dua and C. Graff, *UCI machine learning repository*, http://archive.ics.uci.edu/ml, 2017 (cit. on pp. 58, 84, 113).

[83] F. du Pin Calmon, D. Wei, B. Vinzamuri, K. N. Ramamurthy, and K. R. Varshney, "Optimized pre-processing for discrimination prevention," in *NeurIPS*, 2017 (cit. on pp. 58, 66).

[84] B. Salimi, L. Rodriguez, B. Howe, and D. Suciu, "Interventional fairness: Causal database repair for algorithmic fairness," in *SIGMOD*, 2019 (cit. on p. 58).

[85] V. Metsis, I. Androutsopoulos, and G. Paliouras, "Spam filtering with naive bayes - which naive bayes?" In *CEAS*, 2006 (cit. on p. 58).

[86] Y. LeCun, C. Cortes, and C. J.C. Burges, *MNIST handwritten digit database*, http://yann.lecun.com/exdb/mnist/, 2010 (cit. on pp. 59, 83).

[87] Martín Abadi *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015 (cit. on p. 60).

[88] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," *IEEE Trans Vis Comput Graph*, 2014 (cit. on p. 70).

[89] W. D. Gray and D. Boehm-Davis, "Milliseconds matter: An introduction to microstrategies and to their use in describing and predicting interactive behavior.," *Journal of experimental psychology. Applied*, 2000 (cit. on pp. 70, 158).

[90] Z. Liu and J. Stasko, "Mental models, visual reasoning and interaction in information visualization: A top-down perspective," *IEEE Trans Vis Comput Graph*, 2010 (cit. on pp. 70, 158).

[91] C. Eckart and G. Young, "The approximation of one matrix by another of lower rank," *Psychometrika*, 1936 (cit. on p. 74).

[92] G. Gur-Ari, D. A. Roberts, and E. Dyer, "Gradient descent happens in a tiny subspace," *arxiv*, 2018. eprint: 1812.04754 (cit. on p. 74).

[93] B. Ghorbani, S. Krishnan, and Y. Xiao, "An investigation into neural net optimization via hessian eigenvalue density," in *ICML*, 2019 (cit. on p. 76).

[94] C. Lanczos, *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office Los Angeles, CA, 1950 (cit. on p. 77).

[95] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, Eds., *Differentiation Methods for Industrial Strength Problems*. Springer, 2002 (cit. on p. 79).

[96] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms," *arxiv*, 2017. eprint: 1708.07747 (cit. on p. 83).

[97] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," *Master's thesis, Department of Computer Science, University of Toronto*, 2009 (cit. on pp. 83, 113).

[98] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arxiv*, 2016. eprint: 1605.07146 (cit. on p. 83).

[99] R. Socher *et al.*, "Recursive deep models for semantic compositionality over a sentiment treebank," in *EMNLP*, 2013 (cit. on p. 84).

[100] OpenML, *Openml supervised classification on adult*, https://www.openml.org/t/7592, 2020 (cit. on p. 84).

[101] JAX, *Jax reference documentation — jax documentation*, https://jax.readthedocs.io/en/latest/, 2020 (cit. on pp. 85, 116).

[102] Tensorflow, *Xla: Optimizing compiler for machine learning*, https://www.tensorflow.org/xla, 2020 (cit. on pp. 85, 116).

[103] J. Heer and M. Bostock, "Crowdsourcing graphical perception: Using mechanical turk to assess visualization design," *SIGCHI*, 2010 (cit. on p. 98).

[104] W. Cleveland and R. McGill, "Graphical perception: Theory, experimentation, and application to the development of graphical methods," *J. Am. Stat. Assoc.*, 1984 (cit. on p. 98).

[105] J. Talbot, V. Setlur, and A. Anand, "Four experiments on the perception of bar charts," *IEEE Trans Vis Comput Graph*, 2014 (cit. on p. 98).

[106] R. Pradhan, J. Zhu, B. Glavic, and B. Salimi, "Interpretable data-based explanations for fairness debugging," in *SIGMOD*, 2022 (cit. on pp. 103, 105, 107, 108, 112, 114, 115, 122, 126).

[107] Y. Gu, J. Jin, and S. Mei, "L_0 norm constraint LMS algorithm for sparse system identification," *IEEE Signal Process. Lett.*, 2009 (cit. on pp. 110, 111).

[108] S. Basu, X. You, and S. Feizi, "On second-order group influence functions for black-box predictions," in *ICML*, 2020 (cit. on p. 113).

[109] J. Wei, Z. Zhu, H. Cheng, T. Liu, G. Niu, and Y. Liu, "Learning with noisy labels revisited: A study using real-world human annotations," in *ICLR*, 2022 (cit. on p. 113).

[110] T. Chen, S. Kornblith, K. Swersky, M. Norouzi, and G. E. Hinton, "Big self-supervised models are strong semi-supervised learners," in *NeurIPS*, 2020 (cit. on p. 114).

[111] M. Blondel *et al.*, "Efficient and modular implicit differentiation," *arxiv*, 2021. eprint: 2105.15183 (cit. on p. 116).

[112] F. Abuzaid *et al.*, "DIFF: A relational interface for large-scale data explanation," *VLDB*, 2018 (cit. on pp. 122, 134, 135, 137, 154).

[113] S. Roy, L. J. Orr, and D. Suciu, "Explaining query answers with explanation-ready databases," *VLDB*, 2015 (cit. on pp. 132, 134, 135, 137, 146, 157).

[114] F. Abuzaid *et al.*, "DIFF: a relational interface for large-scale data explanation," *VLDB J.*, 2021 (cit. on p. 132).

[115] F. Psallidas and E. Wu, "Demonstration of smoke: A deep breath of data-intensive lineage applications," in *SIGMOD*, 2018 (cit. on p. 134).

[116] H. Mohammed, Z. Wei, R. Netravali, and E. Wu, "Continuous prefetch for interactive data applications," *VLDB*, 2020 (cit. on p. 134).

[117] N. N. Dalvi and D. Suciu, "Efficient query evaluation on probabilistic databases," in *VLDB*, 2004 (cit. on p. 134).

[118] M. Albutiu, A. Kemper, and T. Neumann, "Massively parallel sort-merge joins in main memory multi-core database systems," *VLDB*, 2012 (cit. on p. 141).

[119] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking SIMD vectorization for in-memory databases," in *SIGMOD*, 2015 (cit. on p. 142).

[120] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case optimal join algorithms: [extended abstract]," in *PODS*, 2012 (cit. on p. 143).

[121] L. Ramjit, M. Interlandi, E. Wu, and R. Netravali, "Acorn: Aggressive result caching in distributed data processing frameworks," ser. SoCC, 2019 (cit. on p. 160).

[122] *Home - mltrace 0.16 documentation*, https://mltrace.readthedocs.io/, Accessed: 2021-08-31 (cit. on p. 162).

[123] M. Vartak, "MODELDB: A system for machine learning model management," in *CIDR*, 2017 (cit. on p. 162).

Appendix A: Theory and Extensions of Rain

A.1 Ambiguity & TwoStep

In this section, we will describe a setting where TwoStep is unlikely to identify the correct training errors because the complaint is ambiguous. Specifically, we will formally prove that very few solutions to the SQL step of TwoStep can lead to a training error discovery in the influence step.

For the needs of our setting, we will focus on debugging a binary logistic regression model M. Its training set T consists of data that is drawn from a clean distribution as well as a single noisily labeled example t. Let $l' \in \{0, 1\}$ be the noisy label of t. For convenience, let the feature vector of t be orthogonal to the other records in T, i.e. its inner product with all other feature vectors is zero. Symmetrically, the queried records distribution contains n records with all but m being orthogonal to t, just like the clean distribution. The remaining m records can be arbitrary.

Let the query Q count the number of records in the queried dataset where M.predict(r)=$1 - l'$, the opposite of the t's incorrect label. The user complaint is that the query result should be k when the current result is 0. We use TwoStep with an influence analysis step based on [12].

Theorem 2. *Assuming that the ILP solver picks uniformly at random from the satisfying solution space, then for fixed m, k the probability that TwoStep assigns t a non-zero score in the influence analysis step converges to* 0 *as* $n \to \infty$.

The intuition of the proof is straightforward. Given the orthogonality condition, the predictions of the $n - m$ queried records coming from the clean distribution would be the same regardless if t existed or not. Unless the ILP assignment picks at least one of the remaining m records, influence analysis at the second step will always assign a zero score for t. As n increases with k and m fixed the probability of picking even one of the m records decreases to 0.

For each of the ILP solutions that do not favor the recovery of t, we can always construct clean training records with positive scores that are ranked above t. Injecting as many as we want for each ILP solution, we can guarantee that TwoStep ranks t arbitrarily low. We will conclude this section with the proof of the main theorem.

Proof. Let θ be the parameters of the logistic regression problem. We can write

$$\theta = \theta_{\text{noise}} + \theta_{\text{clean}}$$

where θ_{noise} is the projection of θ on the direction of the feature vector of t and the second term is the orthogonal residue. We will call v_{noise} the feature vector of t and v_i and y_i the feature vectors and labels of the clean data. Let ℓ be the sample loss function of M

$$L(\theta) = \sum_{i=1}^{T_c} \ell(v_i, y_i, \theta) + \ell(v_{\text{noise}}, l', \theta) + \lambda \|\theta\|^2$$

where the first term corresponds to the loss of the clean data and the second term corresponds to the loss of t. ℓ takes the feature vector and projects it to θ and then applies the sigmoid function and then the log loss. Let f denote the function implementing the steps after the projection

$$\ell(\theta, v_i, y_i) = f(\langle\theta, vi\rangle, y_i) = f(\langle\theta_{\text{clean}}, v_i\rangle, y_i)$$
$$\ell(\theta, v_{\text{noise}}, l') = f(\langle\theta, v_{\text{noise}}\rangle, l') = f(\langle\theta_{\text{noise}}, v_{\text{noise}}\rangle, l')$$

That is the clean distribution loss depends only on θ_{clean} and the loss on t depends only on θ_{noise}. Thus we essentially have two loss functions that depend on disjoint variables

$$\begin{aligned} L(\theta) &= \sum_{i=1}^{T_c} f(\langle\theta_{\text{clean}}, v_i\rangle, y_i) + \lambda \|\theta_{\text{clean}}\|^2 \\ &\quad + f(\langle\theta_{\text{noise}}, v_{\text{noise}}\rangle, l') + \lambda \|\theta_{\text{noise}}\|^2 \\ &= L_1(\theta_{\text{clean}}) + L_2(\theta_{\text{noise}}) \end{aligned}$$

Essentially we have two distinct optimization problems

$$\theta^*_{\text{clean}} = \arg\min_{\theta_{\text{clean}}} L_1(\theta_{\text{clean}}) \qquad\qquad \theta^*_{\text{noise}} = \arg\min_{\theta_{\text{noise}}} L_2(\theta_{\text{noise}})$$

Observe that the existence of t does not affect the value of θ^*_{clean}. Additionally, predictions on queried records that have feature vectors that are orthogonal to t depend only on θ^*_{clean}. Thus complaints on these queried records cannot be resolved by deleting t. For these complaints, t would be assigned a zero score by the influence step of TwoStep.

Only the m queried records could have feature vectors with non-zero inner product with t. Thus, out of all the satisfying solutions of the ILP, only ones that assign the label of $1 - l'$ to at least one of the m queried records has any hope of giving t a non-zero score in the influence analysis step. Observe that there are $\binom{n}{k}$ solutions to the ILP and there are $\binom{n-m}{k}$ assignments that do not pick any of the m points. The probability of picking such an assignment is converging to 1.

$$\lim_{n\to\infty} \frac{\binom{n-m}{k}}{\binom{n}{k}} = \lim_{n\to\infty} \frac{(n-m)!(n-k)!}{(n-m-k)!n!} = \lim_{n\to\infty} \prod_{i=0}^{k-1} \frac{n-i-m}{n+i} = 1$$

The probability of assigning t a non-zero score goes to 0. □

A.2 The Value of Complaints

In this section, we will describe a setting where ordering training records based on loss or loss sensitivity ranks training corruptions at the bottom. At the same time, an appropriately selected complaint is sufficient to rank all corrupted training records at the top.

For the needs of our setting, we will focus on debugging a binary logistic regression model M. Our training set T is a mixture of clean and corrupted training records. Clean records have been perfectly labelled whereas the corrupted ones have had theirs inverted. Corrupted training records labelled as being in class 1 are truly in class 0 and vice versa.

For simplicity, we are going the two following assumptions. First, the feature vectors of the clean training records are all orthogonal to the ones in the corrupted distribution. That is for each

pair of records from the two distributions, the corresponding feature vectors have zero inner product. Second, the feature vectors for all corrupted training records are parallel. That is for each pair of feature vectors v_i and v_j from the corrupted training records, there is a $\kappa_{ij} \in \mathbb{R}$ such that $v_i = \kappa_{ij} v_j$. Third, we are going to assume that the corrupted training records are linearly separable, i.e. there is a linear classifier that can correctly specify the labels of the corrupted training records.

Let us discuss the two ways we can use the model loss to rank training records. The first one is to use the loss value of each training record. The Loss baseline discussed in the experiments ranks training records with higher loss at the top. The second one ranks training records based on loss sensitivity. Specifically, [12] considers the effect of the removal of each training record on its own loss. This is the InfLoss baseline. For each training record z it computes a loss sensitivity

$$-\nabla_\theta \ell(\theta^*, z) H_{\theta^*}^{-1} \nabla_\theta \ell(\theta^*, z).$$

These scores are negative or zero since the Hessian of logistic regression, and thus its inverse, is positive definite. Large negative values indicate that when the training record is removed, its own loss tends to increase rapidly. These training records are ranked at the top by InfLoss.

Theorem 3. *As the number of corrupted training records goes to infinity, the loss and loss sensitivity of corrupted training records goes to zero.*

Observe that 0 is the minimum value of the loss and maximum value of the loss sensitivity. Thus both approaches rank corrupted training records at the bottom.

Proof. Let θ be the parameters of the logistic regression problem. We can once again write

$$\theta = \theta_{\text{noise}} + \theta_{\text{clean}}.$$

θ_{noise} is the projection of θ on the direction of the feature vector of a corrupted training record. It does not matter which one since all are parallel. θ_{clean} is the orthogonal residue.

We can apply the same techniques as in the proof of Theorem 2 to get two independent

optimization problems. Let K be the number of corrupted points, v_i the feature vectors and y_i the corresponding corrupted labels. Using the f function from the proof of Theorem 2

$$\theta^*_{\text{noise}} = \arg\min_{\theta_{\text{noise}}} \left(\sum_{i=1}^{K} f(\langle v_i, \theta_{\text{noise}} \rangle, y_i) + \lambda \left\| \theta_{\text{noise}} \right\|^2 \right)$$

As a first step we want to prove that

$$\lim_{K \to \infty} f(\langle v_i, \theta^*_{\text{noise}} \rangle, y_i) = 0$$

for all pairs of v_i and y_i from the corrupted training records. This states that the loss of the corrupted records goes to 0, our first claim. Let σ be the sigmoid function. By first order optimality conditions on the optimization problem of θ^*_{noise} we have

$$\sum_{i=1}^{K} \left(\sigma(\langle v_i, \theta^*_{\text{noise}} \rangle) - y_i \right) v_i + 2\lambda \theta^*_{\text{noise}} = \mathbf{0}.$$

We have assumed that the corrupted training records are linearly separable. Thus there exists a vector u that linearly separates the data with margin 1. Multiplying by u we have

$$\sum_{i=1}^{K} \left(\sigma(\langle v_i, \theta^*_{\text{noise}} \rangle) - y_i \right) \langle v_i, u \rangle + 2\lambda \langle \theta^*_{\text{noise}}, u \rangle = 0.$$

In turn we have that

$$\sum_{i=1}^{K} \frac{\left(\sigma(\langle v_i, \theta^*_{\text{noise}} \rangle) - y_i \right) \langle v_i, u \rangle}{\langle \theta^*_{\text{noise}}, u \rangle} = -2\lambda.$$

Given that u has margin 1, we have that

$$(2y_i - 1)\langle v_i, u \rangle \geq 1$$

and thus for both y_i equal to 0 and 1, all summation terms of the previous equation need to have the same sign. As the number of terms K increases, each term individually cannot be bounded away

from 0 because the sum remains finite. Thus θ^*_{noise} is such that either the numerator goes to zero or the denominator goes to infinity or both. For the numerator to go to zero, θ^*_{noise} must perfectly fit its data with an infinite margin making the denominator go to infinity as well. Thus we have

$$\lim_{K\to\infty} \langle \theta^*_{\text{noise}}, u \rangle = \infty$$

Given that all v_i are parallel to u and θ^*_{noise}, the loss of the samples going to 0 follows immediately.

Similarly, the gradients of the losses go to a 0 norm. For the loss sensitivity scores, we have

$$0 \geq -\nabla_\theta \ell(\theta^*, z) H^{-1}_{\theta^*} \nabla_\theta \ell(\theta^*, z) \geq - \left\|\nabla_\theta \ell(z, \theta^*)\right\|^2 \lambda_{\max}(H^{-1}_{\theta^*})$$

where $\lambda_{\max}(H^{-1}_{\theta^*})$ the biggest eigenvalue of the inverse Hessian. The minimum eigenvalue of the Hessian is at least 2λ. Thus the inverse Hessian has bounded eigenvalues by $\frac{1}{2\lambda}$.

$$0 \geq -\nabla_\theta \ell(\theta^*, z) H^{-1}_{\theta^*} \nabla_\theta \ell(\theta^*, z) \geq - \left\|\nabla_\theta \ell(z, \theta^*)\right\|^2 \frac{1}{2\lambda}$$

With the gradient norms going to 0, the loss sensitivity scores of all corrupted training records converge to 0 as well. □

Thus for a large enough number of corrupted training records and the clean ones fixed, we can force the corrupted training records to the bottom of the rankings for both Loss and InfLoss. What remains to discuss is how an appropriate complaint can bring these records to the top of the ranks.

Complaints on queried records with feature vectors parallel to the ones of the corrupted training records are particularly interesting. For these complaints, the clean training records receive 0 influence scores following the same discussion as in Theorem 2. It thus remains to find one such complaint that assigns positive scores to all corrupted training records.

Even identifying one of the mispredicted queried records that have the property above will do. Let $z_q = (v_q, y_q)$ be the identified record with its correct label. The influence score of each

corrupted training record $z_i = (v_i, y_i)$ is

$$-\nabla_\theta \ell(\theta^*, z_q) H_{\theta^*}^{-1} \nabla_\theta \ell(\theta^*, z_i).$$

We have that

$$\nabla_\theta \ell(\theta^*, z_q) = \left(\sigma(\langle v_q, \theta^*_{\text{noise}} \rangle) - y_q\right) v_q$$
$$\nabla_\theta \ell(\theta^*, z_i) = \left(\sigma(\langle v_i, \theta^*_{\text{noise}} \rangle) - y_i\right) v_i.$$

v_q and v_i are parallel but z_q is mispredicted while z_i is correctly predicted. Simple algebra shows that the gradients have opposite directions. That is there is a $k > 0$ such that

$$\nabla_\theta \ell(\theta^*, z_q) = -k \nabla_\theta \ell(\theta^*, z_i)$$

Since the Hessian of logistic regression and thus its inverse is positive definite, we have

$$-\nabla_\theta \ell(\theta^*, z_q) H_{\theta^*}^{-1} \nabla_\theta \ell(\theta^*, z_i) =$$
$$k \nabla_\theta \ell(\theta^*, z_i) H_{\theta^*}^{-1} \nabla_\theta \ell(\theta^*, z_i) > 0.$$

Thus the influence scores of all corrupted training records are positive and the complaint ranks all of them at the top.

A.3 Supporting Multiple Classes with Holistic

Relaxing SQL queries that use multi-class classification models is also supported. As an example, let us consider the MNIST dataset that we describe in Section 4.3. We can design a classifier that takes one image, represented by a 28×28 grid of pixels, and yields a number from 0 to 9 corresponding to the digit displayed. We may want to use this model in an optical character recognition application that takes a handwritten multi-digit number, segments it into small images

each containing a single digit and uses the classifier to figure out the numerical value of the whole number. Let us assume that the segmentation has occurred and that we have the sequence of N images stored in a table `DIGITS` in an attribute called `image`. Along with each image we have a field `position` indicating the digit position from the right. The numeric value of the number is represented in SQL by the following query

```
SELECT SUM(POWER(10, position)* M.predict(image)) FROM DIGITS
```

Let θ be the parameters of our model and $p_{ij}(\theta)$ be the probability assigned by the classifier that the image at position i is digit j. Then the relaxation of the query output is the following quantity

$$\sum_{i=1}^{N} 10^{i-1} \sum_{j=0}^{9} j \cdot p_{ij}(\theta)$$

A.4 Aggregate Comparisons with Holistic

SQL queries are allowed to use comparisons in their selection and join predicates. Unfortunately, the relaxation rules for Holistic as described in Section 4.2.3 do not directly support comparison operators. Regardless of its complexity, every comparison has an equivalent logical formula involving only `AND`, `OR`, `NOT` that Holistic supports. Finding such a logical formula can be non-trivial when aggregate values are compared. SQL can express aggregate comparisons through a `HAVING` clause or by using filters on nested queries.

For example, let us revisit the optical character recognition application of the previous subsection. We want to express a predicate selecting numbers that are greater or equal than 95. Let $x_{i,j}$ be the boolean value expressing that the digit at position i from the right is classified as being j. For simplicity we focus on the case of two-digit numbers. Then the aggregate comparison is equivalent to the following formula

$$x_{2,9} \ \texttt{AND} \ \left(\bigvee_{9 \geq \ell \geq 5} x_{1,\ell} \right)$$

In general, finding the equivalent formula can be a computationally expensive procedure and the resulting formula can have a large amount of terms slowing down the influence analysis step.

Identifying ways to relax comparisons directly is a promising direction for future work. It is important to note that comparisons that are part of the complaint itself do not require special care. They can be handled directly based on the techniques described in Section 4.2.3.