Infrastructure for Model Management and Model Diagnosis

by

Manasi Vartak

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Signature redacted

Author ....
Department of Electrical Engineering and Computer Science
May 25, 2018

Signature redacted

Certified by..................... ....
Samuel Madden
Professor
Thesis Supervisor

Signature redacted

Accepted by ....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science,
Chair, Department Committee on Graduate Students

Infrastructure for Model Management and Model Diagnosis

by

Manasi Vartak

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2018, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Building ML-based workflows in the real world is a trial-and-error, iterative process where an ML developer builds tens to hundreds of workflows before arriving at one that meets some task-specific acceptance criteria. This iterative process of workflow building is laborious for several reasons including the large variety of available ML models, the time required to train the workflow, difficulty keeping track of workflows built during the modeling process, and the time required for debugging trained workflows. In this thesis, we are primarily interested in two problems with the repetitive modeling process: first, how to manage ML-based workflows generated over multiple iterations of the modeling process, and second, how to efficiently debug or diagnose trained ML-based workflows. In this work, we study these questions from a systems perspective and propose novel software systems and techniques to address them. Specifically, our contributions are: 1. We propose ModelDB, a system to track provenance and performance of ML-based workflows. 2. We propose MISTIQUE, a system to store ML-based workflow intermediates in order to speed up model debugging tasks, and 3. We provide examples of new diagnostic techniques that can be designed using the data in MISTIQUE.

Thesis Supervisor: Samuel Madden
Title: Professor

Acknowledgments

My PhD work would not have been possible without the support of an incredible number of people who have personally or professionally taught me many things, given me unique opportunities, and believed in me.

First and foremost, this PhD wouldn't have been possible without the unwavering support of my PhD advisor, Samuel Madden. All through my PhD, Sam has given me freedom to take on and explore a wide variety of research problems ranging from systems research to visualization/HCI to machine learning and even computational biology. This freedom helped me clearly identify my interests and strengths. His unwavering support also enabled me to take on large, risky, often ill-defined problems without fear of failure. His metrics around "is it an interesting problem" and the importance of building a "real system" have shaped my view of the research world. His amazingly collaborative nature has also encouraged me to engage in many collaborations during my time at MIT. Sam's willingness to learn, enthusiam, humility, and kindness are always inspiring to me.

I would also like to thank Michael Stonebraker who I had the fortune to work with early on in my PhD and who has given me great feedback on research and career. Mike always asks the hard questions about any piece of work and has helped improve my research and this thesis document. I am also grateful to Tamara who, along with Mike, has been on my thesis committee and whose comments on my thesis encouraged me to view my work in a wider setting.

The work in this thesis wouldn't have been possible without Matei Zaharia who has been a collaborator both on ModelDB and MISTIQUE. Matei's help in shaping technical direction has been invaluable. I'm thankful to Aditya Parameswaran for introducing me to the wonderful area of data visualization and exposing me to problems that have since shaped my research tastes. I'm also incredibly grateful to Hugo Larochelle who at Twitter mentored me and opened up the whole world of deep learning practice and research. I have also have been privileged to have had many great mentors including Elke Rundensteiner who originally got me interested in databases, Irene Grief, and Tyreek Moore.

One of the highlights of my PhD tenure has been the opportunity to work with amazingly talented and motivated undergraduate and masters degree students at MIT. The work on ModelDB would truly have been impossible without them and I feel priviledged to have had the opportunity to be their fearless leader for a little while.

I am also grateful to have been supported by a Facebook PhD Fellowship from 2016 -2018.

I am thankful to the DB Group students past and present who have been on this journey with me and have played an important role in improving my research and communication of it. At MIT, I have also had the fortune of being part of many student groups including GWAMIT, GW6, GSW, and EECS REFS, where I was able to learn from a great number of amazingly driven students and I was given the opportunity to contribute to causes important to me. Morevoer, one of the best things about my time at MIT has been meeting like-minded friends who have been on this journey with me and this has been one of the reasons I have thought of MIT and Boston as "home."

Besides CSAIL, the MIT Trust Center for Entrepreneurship has most contributed to my growth at MIT. The Trust Center, through its staff and programming, introduced me to entrepreneurship and provided me with innumerable opportunities to test out ideas in a low-risk setting. I am grateful to have been part of such a supportive community.

None of my PhD (or anything else) would have been possible without the unflagging support of my parents, Jyoti and Pradeep. They have been rocks of strength, love, and clear-headed advice, three things that never fail to clear the fog. My husband, Ashutosh, has been with me every day of this journey and has many stories to tell! His encouragement, support, patience, wit, curiosity, kindness, and love have kept me going and have encouraged me to go farther. My PhD experience would have been so much less fun and enriching without him. My brother, Nimish, and his family have been yet another rock of support for me and a place I have turned to for advice so many times. My grandfather, Sadashiv, even in memory, has been an amazing inspiration for me through his lifelong qualities of resilience and hard work. I am grateful to the rest of my family for their love and encouragement.

Ultimately, for me, MIT was not one person, one department, or one degree. It was a land of opportunities, resources, and freedom of exploration. I am truly grateful to have been able to spend my PhD time here.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine Learning (ML) applications are becoming omni-present in a variety of domains. For example, ML powers object detection in self-driving cars [25, 29], voice-driven assistants through speech recognition [57, 9], and recommendation systems on many e-commerce sites [79, 67]. While the above applications are the most visible applications of machine learning, ML is also being used behind the scenes for tasks such as prioritizing and responding to customer service requests [65], determining where to drill for oil [66], and deciding whether to provide loans to applicants [92].

A key step in building an ML application is developing an ML-based data-processing workflow to perform the given task (e.g., pedestrian detection or loan default risk estimation). A typical ML-based workflow consists of multiple operations including data pre-processing, feature extraction, and training of an ML model. Building such a workflow is, in general, not a one-shot process and requires a great deal of experimentation. An ML developer must explore different data transformations, ML models, and hyperparameter[1] settings to identify an ML-based workflow that meets some task-specific performance or quality criteria. As a result, ML developers will often train and evaluate tens to hundreds of workflows. We refer to the process of developing an ML-based workflow for a particular application as the **modeling process**. Software systems to support the modeling process are the focus of this thesis.

[1]Informally, hyperparameters are model variables whose values affect the parameters the model can learn from data. We provide a detailed explanation in Chap. 2

We note that while the techniques proposed in this work are applicable to general data-processing workflows which require iterative experimentation, we are motivated by workflows where training an ML model is an important part of the workflow. Consequently, our motivating examples, implementations, and evaluation are oriented towards ML-based workflows. Moreover, our proposed techniques work best when applied to ML-based workflows because they are tailored to the unique requirements as well as properties of these workflows. We discuss the relationship of this work to general data-processing workflows in the Related Work chapter (Chap. 3).

1.1 The modeling process

Suppose our task is to predict home prices using historical data about homes and their sale value (see Chap. 2 for the full example). Given this task, a typical modeling process proceeds as follows. An ML developer starts with a simple workflow specification[2] consisting of some data pre-processing steps (e.g., removing missing values from the data), feature extraction operations (if any), and an ML model (e.g., linear regression) that captures assumptions about how the data was generated (e.g., that there is a linear dependence between the dependent and independent variables). The ML developer then trains this workflow. For steps without any data-dependent state, there is no explicit training required; however, for ML models, a learning algorithm is used to fit the model parameters to the training data. Once the workflow has been trained, the ML developer computes performance of the trained workflow using a metric specific to the task (e.g., error in predicted home price). Based on the performance of the trained workflow, the ML developer may decide to refine the workflow. For example, the ML developer may pre-process data in a different manner, build a different type of ML model (e.g., decision tree instead of linear regression), tune hyperparameters of the current ML model, and so on. *We call the process of specifying a workflow, training steps in the workflow,' evaluating the trained workflow, and finally refining the workflow definition based on its task performance as the*

[2]We delay formal definitions of ML-based workflows Chap. 2

modeling loop (see Fig. 1-1). Each sequence of these four actions constitutes one iteration through the modeling loop and an ML developer typically performs many tens, if not hundreds, of iterations through this loop.

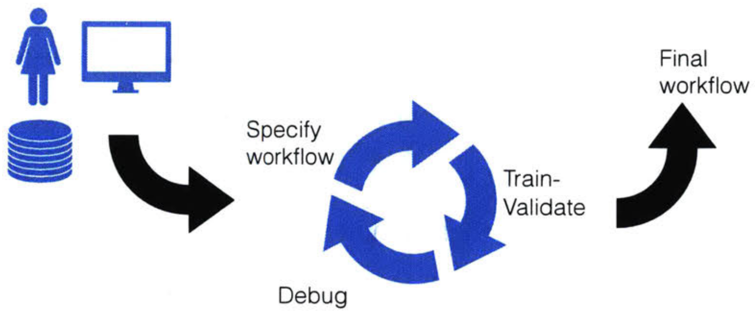


Figure 1-1: Modeling loop

The modeling process described above is repetitive and laborious for a variety of reasons including the large variety of available ML models, the time required to train models in the workflow, difficulty keeping track of workflows built during the modeling process, and the time required for debugging trained workflows. In this work, we are primarily interested in two problems with the repetitive modeling process: first, how to manage ML-based workflows generated over multiple iterations of the modeling process, and second, how to efficiently debug or diagnose trained ML-based workflows. In particular, we study these two questions from a *systems perspective* and propose novel software systems and techniques to address them.

1.1.1 Managing ML-based workflows over many iterations

ML developers often train and evaluate hundreds of ML-based workflows before arriving at one that meets the acceptance criteria on workflow performance or quality. However, developers currently have no means of tracking previously-trained ML-based workflows for insights from previous experimentation. For example, almost all ML developers interviewed as part of this work (employed at companies ranging from startups to large tech companies) either did not have any workflow tracking system in place, or at best, they manually tracked trained workflows in a text document or a spreadsheet. Remembering this information about each trained workflow or even manually logging them in a spreadsheet is challenging for more than a handful of models (because ML developers forget to record some workflows or make mistakes in recording). From our interviews with data scientists, we found that the lack of model management causes insights to be lost, produces workflows that are not reproducible or auditable, and leads to challenges in collaboration and in running meta-analyses (see Chap. 4). These challenges bring to light an important but little-studied problem in the practice of machine learning that we term **model workflow management** or simply **model management**. Model management in the context of this thesis deals with tracking ML-based workflows built during the modeling process, extracting steps (and parameters) for each workflow, and then making all of this information available for sharing, analysis, and reproducibility. **The first contribution of this thesis is to propose** MODELDB**, the first open-source system for model management.**

1.1.2 Debugging and Diagnosing Workflows

The second problem studied in this thesis is that of ML-based workflow debugging and diagnosis. Machine learning is the process of automatically extracting patterns from data. During this process, an ML developer does not explicitly define rules for extracting patterns from data; instead, these "rules" are learned automatically from the data by tuning model parameters. As a result, debugging an ML model (and its associated ML-based workflow) is as much about debugging data as it is about debugging code. Moreover, "bugs" in ML-based workflows can often result from patterns already present in the data as opposed to errors. For these two reasons, we refer to model debugging instead as *model diagnosis.* Just as a doctor diagnoses the underlying reason for a patient's symptoms, our goal with model diagnosis is to identify why certain outputs are produced by an ML-based workflow.

Concretely, *diagnosing* an ML-based workflow involves answering questions such as "why does the home price prediction workflow under-perform on old Victorian homes?" or "why does the image classification model classify a frog as a ship?" The results from model diagnosis such as "because there are very few old Victorian houses in the dataset" or "frog and ship images in the dataset have similar backgrounds" help identify failure modes for the workflow and devise remediations. A line of work related to model diagnosis studies model interpretability (e.g., [106, 13, 83, 41]). Model interpretability in turn seeks to answer different kinds of questions ranging from how to make results from ML models usable to non-experts (e.g., [138]) to understanding the abstract concepts learned by a neural network (e.g., [13]). For simplicity, we will refer to both kinds of analyses described above as "model diagnosis."

Many model diagnosis questions such as the ones described above can be answered by analyzing different data artifacts related to the ML-based workflow including input data, prediction values, and outputs of different workflow stages (e.g., the results of applying dimensionality reduction to an input dataset). We collectively refer to these datasets as model workflow intermediates or simply *model intermediates*. Currently, performing model diagnosis requires the ML developer to choose one of two solutions: either store all model intermediates (often hundreds of GBs in size) and incur a large storage cost; or repeatedly run workflows to obtain the relevant intermediates, an untenable solution for interactive diagnosis. Moreover, as shown by our empirical evaluation in Chap. 5.8, for some queries, the difference in execution time when re-running a workflow vs. reading an intermediate can be up to two orders of magnitude. Thus, choosing the right strategy for running model diagnosis queries can significantly impact time required for model diagnosis.

The bottleneck in supporting efficient and widely usable model diagnosis is therefore caused by two data management questions: (a) how to store large intermediates efficiently for storage and querying; and (b) how to trade-off intermediate storage vs. recreation? **To address this challenge of efficiently querying model intermediates, the second contribution of this thesis is MISTIQUE, a system designed to capture, store, and query model intermediates for model diagnosis.**

1.1.3 Techniques for Model Diagnosis

The MISTIQUE system described above provides ML developers with easy access to data that was previously expensive to obtain (computationally or storage-wise).

Access to this data enables the design and evaluation of new types of diagnostic techniques. Specifically, **the last contribution of this thesis is a novel user-interface for comparing and exploring predictions from multiple models**. Since current systems for ML infrastructure do not manage workflows across many iterations or store model intermediates, they cannot support comparison of predictions or intermediates across many ML-based workflows. Our novel interface, called the PREDICTIONVISUALIZER, uses data in MISTIQUE and visualizes predictions across any number of previously trained workflows. This novel visualization can enable the user to easily spot trends in specific model types while also analyzing individual model predictions.

Thus, this thesis takes first steps in defining infrastructure for managing models across many modeling iterations and performing model diagnosis. In the next section, we summarize the contributions made by our work and lay out the outline for the remainder of this document.

1.2 Summary of Contributions

The goal of this thesis is to propose infrastructure for managing ML-based workflows across many iterations and for supporting model debugging. Towards this goal, we make three contributions:

- First, we propose MODELDB, a system that serves as a centralized repository of ML-based workflows. MODELDB helps ML developers track what workflows they have trained, the different stages of each workflow along with their respective hyperparameters, and the performance of every trained workflow. This information avoids repeated work, provides auditability, and enables meta-analyses across models.

- Second, we propose MISTIQUE, a system to store data intermediates produced from workflows that can be used to perform a variety of diagnostic tasks. MISTIQUE can be used with traditional ML workflows in scikit-learn [98] as well as deep neural networks in Tensorflow [6]. Depending on the diagnostic query, MISTIQUE can run model diagnosis up to two orders of magnitude faster than current methods.

- And third, we propose to use the intermediates captured in MISTIQUE to develop new diagnostic techniques. Specifically, we present PREDICTIONVISUALIZER, a visual interface to compare instance-level predictions and summary performance metrics across many models.

1.3 Outline of Thesis

The rest of the thesis is organized as follows. We begin by providing a primer on machine learning and the specific types of ML-based workflows studied in this thesis (Chapter 2). We then present a survey of related work in Chapter 3. The MODELDB system including its design, implementation and evaluation is described in Chapter 4. In Chapter 5, we present the MISTIQUE system for model diagnosis. We present the Prediction Visualizer in Chapter 6. We finish with a discussion of future work in Chapter 7 and concluding thoughts in Chapter 8.

Chapter 2

Background

In this section, we provide a primer on machine learning, formally define the concept of ML-based workflows, and describe the types of ML-based workflows studied in this thesis. We also describe popular ML frameworks to motivate the implementation decisions in subsequent chapters and provide an overview of the Kaggle platform which we use as a source for real-world ML-based workflows.

2.1 A primer on Machine Learning

Basics. Machine learning is the process of finding patterns in data. We start with a set of input examples X=$\{x_1...x_n\}$ from which we want to extract patterns. This is called the **training set**. In many cases, the training set is processed to represent examples in a manner that is suitable for extracting patterns. This processing, involving the generation of new descriptors or attributes of data, is called feature extraction, and each descriptor of the data is called a **feature**. All the features of the training data together make the **data representation**. Broadly, a model is a set of assumptions about some process. A machine learning model captures our assumptions about the process that generated our training data, specifically, these are assumptions about the probability distribution of our data and are embodied in the **parameters** of our model [89]. There are many machine learning *models* (e.g., linear regression, multi-layer perceptron, support vector machines) as well as *methods* (e.g., decision tree classifier, principal component analysis) that have been proposed over the years[1].

A machine learning model has a set of parameters that can be adapted to the training data. A **machine learning algorithm** or a learning algorithm is used to learn these parameters from the training data. ML models also have additional variables associated with them called **hyperparameters** whose value is usually set before the learning algorithm is applied. Hyperparameters influence the distribution of parameters learned by the model and affect how well the model captures the underlying data generating process. For example, the number of clusters k is a hyperparameter in the k-means clustering algorithm.

Types of Learning Problems. In **supervised** learning problems, along with the data representation x_i, we are also provided with a target value y_i for each input example. Therefore, for these problems, we seek to model the probability distribution $p(y|x)$. When the target value y_i comes from a discrete set of values, the problem is said to be a **classification** problem; instead, if y_i consists of continuous values, then the problem is said to be a **regression** problem. If we are not provided a target variable and our goal is to find underlying structure in x, then the problem is said to be an **unsupervised** learning problem and we seek to model $p(x)$.

Model Performance. We can use different metrics to measure performance of our model. For regression settings this may be the mean squared error in the predicted and actual output whereas for classification this may be the misclassification rate. In ML, we are concerned with the performance of our model not on training data but with the expected error on unseen, future data (also called the **test data**). This error is called the **generalization error**. Since we do not have access to future data, we can create test data by partitioning the training data into two. We use one part to train the model (i.e., the **training set**) while we use the other part (called the **validation set**) to assess model performance on unseen data. When the original training dataset is small, instead of partitioning the data into two parts, we use **cross-validation**. In cross-validation, we divide the training data into a number of folds (i.e., parts that

[1]In this work, we do not distinguish between a machine learning model vs. method.

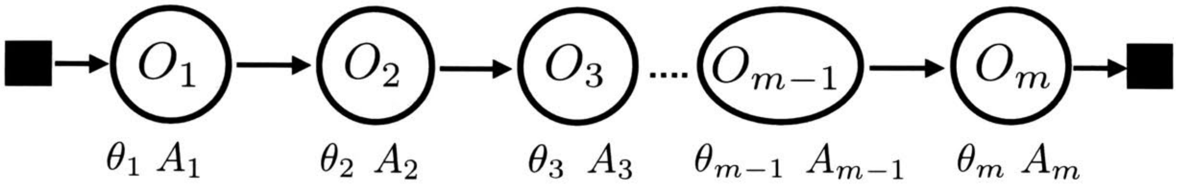


Figure 2-1: ML-based workflow Abstraction

may be constructed using different techniques), train a model on all folds but one and then test the said model on the left-out fold. We then average error across all folds and use it as a proxy for error on future unseen data.

2.2 ML-based workflows

As described (Chap. 1), the work in this thesis deals with the process by which ML developers build ML-based workflows for specific tasks. We now formally define an ML-based workflow and its constituent steps.

An ML-based workflow is a directed acyclic graph of **operations** (See Fig. 2-1 which for simplicity shows a linear workflow). Each node in the graph represents an operator and each edge represents the output of an operation. In most cases, a workflow takes as input a dataset X with N examples and transform it into a dataset Y with N examples such that the dataset Y closely matches a reference dataset T with the same shape and size as Y (where matching is measured with respect to a user defined function $\mathcal{P}$). We call the output of $\mathcal{P}(Y, T)$ as the performance of the workflow. In this work, we are most interested in tasks where the workflow that maximizes $\mathcal{P}$ is not known apriori and the workflow builder must experiment with multiple workflows to find one that produces the best performance.

An operator can perform a variety of things: it may read in data from some data source, it may process the data in different ways (e.g., scaling values in the dataset), it may train an ML model, and so on. Each operator O_i in the workflow can have parameters θ_i that depend on the data we seek to model and corresponding procedures (A_i) to compute these parameters (whether from the data or using user input).

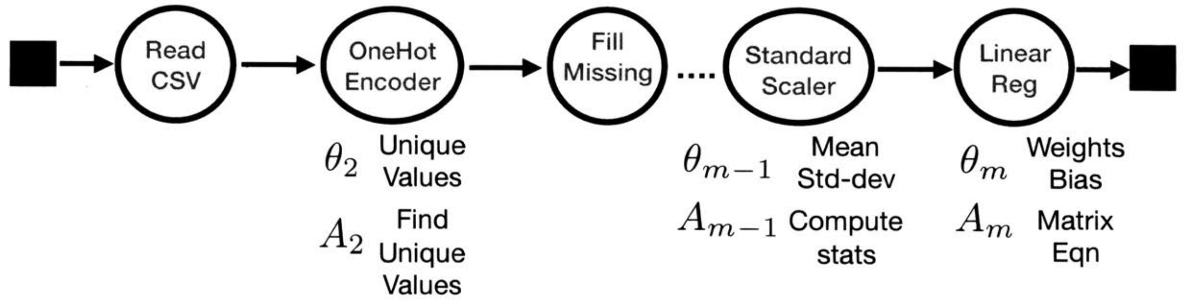


Figure 2-2: ML-based workflow Example

For example, Fig. 2-2 shows the ML-based workflow for the home price prediction task that will be discussed next. The workflow consists of an operator to read in the CSV file, then an operator to perform one-hot-encoding of categorical values[2], then an operator to fill in missing values, then scaling of all columns to be 0-mean and 1-variance, and finally a linear regression model to predict home prices. For each operation in the workflow, we show the parameters of the operator as well as the procedure used to learn these parameters. In case of the One-Hot-Encoding operation, learning parameters merely involves indentifying all unique values in a column. In the case of a linear regression model, the learning algorithm (in this case the learning algorithm involves solving matrix equations) would fit the weights and biases of the model.

In general, a workflow can be executed in three modes: (a) The *training* mode where the workflow is run on *training data*. In this mode, the workflow is provided both the input data X_{train} and the corresponding reference dataset T_{train}. Each operator of the workflow in turn gets the input data for that operator (i.e., results of applying all the Operators preceding this one to the input data) as well as the reference dataset T_{train}. An operator can use its input data and T_{train} to compute its parameters. (b) The *validation* mode where the entire workflow is run on the validation data $X_{validate}$ (which is usually different from the training data) and the result is compared with the reference dataset $T_{validate}$ to compute the performance of

[2]Given a column `col` with potential values A, B, and C, applying one-hot-encoding to this column generates three binary columns respectively indicating whether the column value is an A, B, or a C.

the workflow. (c) And third, the *test* mode where no reference dataset is available and the workflow is merely run from start to finish on *test data*, X_{test}.

A typical process of building such a workflow therefore consists of four stages: (i) specifying the workflow (i.e., stages, parameters types, and parameter computation procedures), (ii) running the workflow in training mode; (iii) running the workflow in validation mode to compute performance; and (iv) debugging the workflow to identify ways to improve workflow performance. A workflow developer will repeat these steps a number of times in order to find a workflow with high performance. Note that this work does not cover workflows where more data becomes available (or is collected) over time.

While our definition of workflows covers many types of data-processing workflows, the workflows that particularly motivate us in this thesis are machine learning-based workflows, i.e., workflows involving one or more machine learning methods. Specifically, ML-based workflows transform input data into predictions (for supervised learning methods) that accurately reflect some target variable. ML-based workflows often consist of a series of Operators responsible for data pre-processing, creation of new data descriptors (called feature engineering), and the training (and application) of ML methods. In this thesis, we limit ourselves to ML-based workflows where the last operator is a supervised machine learning method and none of the previous operators are supervised machine learning methods that are to be trained in the given workflow[3].

Thus, while the techniques proposed in this work are applicable to general data-processing workflows that fit the above criteria, our motivating examples, implementations, applications, and evaluation are oriented towards ML workflows.

2.3 Scope of this Work

Machine learning as a field is extremely wide and diverse. As a result, some ML models and techniques are clearly out of scope for this work. For example, our work

[3]We note that already trained supervised ML methods may be used in any operator

does not extend to reinforcement learning since our techniques expect all training data to be available when the workflow is run. Similarly, workflows involving active learning [112] fall outside the scope of this thesis. Along similar lines, for deep neural networks, this work does not cover topic of data augmentation [133] (the strategy of increasing the amount and diversity of training data, particularly for images, by applying various transformations to the data) during model training.

The general techniques described in this thesis are applicable to many supervised ML models (e.g., linear models, decision trees, random forests). However, the implementations of our systems do not support all types of models and workflows. For example, ModelDB does not (natively) support deep neural network models. In contrast, MISTIQUE supports both traditional ML models as well as deep neural networks. PredictionVisualizer once again is agnostic to the particular kind of model that produced the predictions that are being analyzed. Therefore, for every system and technique we propose, we clearly state the types of models, methods and workflows that can be used with the system. Furthermore, we assume that data cleaning and integration has already been performed before the modeling process starts. As a result, although cleaning and integration are important data pre-processing steps, these are also out of scope for this work. For active learning and data cleaning, we provide some pointers in the chapter on Future Work (Chap. 7) for how the proposed systems can be extended to support these processes.

2.4 Kaggle: a source for machine learning tasks and workflows

Before we discuss ML-based workflows, we highlight the source from which we have obtained many modeling workflows and tasks. Kaggle [3] is a platform for conducting and participating in data science and machine learning competitions. Kaggle hosts machine learning challenges organized by research and industrial sponsors (e.g., the Zillow competition described below, the ImageNet competition [2]). Competitors on

Kaggle participate in these challenges by building ML models for the competition task and submitting their predictions for grading by the Kaggle test server (Kaggle maintains a leaderboard for each competition). Some competitors also opt to share the full code used to generate their model via Kaggle *kernels* (i.e., runnable code notebooks). These kernels, along with leaderboard statistics (which team produced the best performing model), and interviews with top Kaggle competitors provide a rich source of data on machine learning and data science workflows. In this thesis, we use Kaggle kernels as a source of real-world, data scientist-generated machine learning workflows. These workflows motivate some of the proposed techniques and are also used as a test-bed for evaluating our proposed techniques, particularly in MISTIQUE.

2.5 Types of ML-based workflows

The systems proposed in this thesis broadly deal with two distinct kinds of modeling methods: **traditional machine learning** (TRAD) and machine learning using **deep neural networks** (DNN)[4]. In TRAD machine learning, the performance of models (e.g., logistic regression) is closely tied to the representation of data provided to the model (i.e., the features used to describe examples in the data). As a result, when training a traditional ML model, a data scientist or ML developer must identify the best representation of data for the particular task. Consequently, as noted in the literature [11, 142], a large fraction of the effort spent in traditional machine learning goes towards creating or crafting high quality features and data representations.

In DNN models, in contrast, the model itself *learns* representations of raw data which are optimized for the specific machine learning task. For example, if our task is to distinguish between cat and dog images, the model will learn features for this task that will (likely) be different from the features learned to distinguish between images of different brands of cars. A DNN consists of many different layers of computational

[4]Note that MODELDB mainly deals with traditional ML methods whereas MISTIQUE deals with both kinds of methods discussed here.

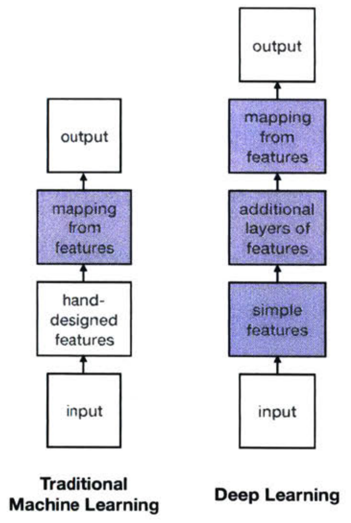


Figure 2-3: TRAD vs. DNN models (adapted from [51])

units (discussed in detail in subsequent sections) where each layer learns a different representation of the raw data. The final layer of the network (called the *output layer*) can be thought of as playing the role that a model plays in TRAD machine learning — it learns a function correlating the data representation with the expected label for each example. Since feature extraction takes place within the DNN model, as noted in the literature [146, 134], the focus of ML developers now shifts from feature engineering to engineering DNN architectures.

Fig. 2-3 shows a visual comparison of TRAD and DNN models (adapted from [51]). We now discuss each modeling method in detail along with a prototypical ML-based workflow for each method.

2.5.1 Traditional Modeling Methods (TRAD)

We begin with a motivating example modeled after the Zillow Home Value Prediction competition on Kaggle [34].

Suppose an online real estate company wants to build a machine learning model to predict home prices using historical data about home attributes (e.g., geographic location, number of rooms, architectural style, area of the home) and the price at which each home was sold[5]. Sample data for this *supervised* learning task is shown in Table. 2.1. The column we wish to predict (also called the *target* column or the *label*) is the **Price** column. Our goal is to use the remaining columns to the predict the value in the Price column.

Latitude	Longitude	Num Rooms	Area (sq. ft.)	...	Price ($)
34144440	-118654080	2	900	...	500K
34144440	-118654080	1	650	...	330K
33172410	-126657770	2	870	...	620K
34465048	-118568168	2	910	...	700K
34450042	-118555968	3	1010	...	660K
33192413	-126892370	2	790	...	490K

Table 2.1: Sample data for Home Price Prediction

As mentioned before, solving this modeling problem using traditional ML techniques requires the ML developer to perform a large amount of data pre-processing and feature engineering. ML-based workflows using traditional ML methods therefore have a number of operators dedicated to producing features and transforming data into a format that is best suited for the ML model that is used in the pipeline.

An example of such a modeling pipeline for the Zillow competition is shown in Fig. 2-4. This pipeline was obtained directly from a Kaggle kernel [40]. As shown in Fig. 2-4, this workflow consists of (a) reading data; (b) joining two tables; (c) performing data pre-processing (e.g., fill missing values); (d) feature engineering and selection (e.g., drop unimportant columns); (e) model training and prediction (e.g., LightGBM [72]). We find this pattern of pre-processing, feature engineering and training steps across a variety of `TRAD` workflows (e.g., workflows [40, 35] for the Zillow Home Value Prediction competition [34]; workflows [38, 39] for the Kaggle Titanic Prediction competition [33]; and workflows [36, 37] for the Avito Demand Prediction competition [32]). While these workflows vary in the complexity of their steps, they can all be captured in our ML-based workflow abstraction.

[5]the original Kaggle competition asks competitors to predict the *error* in Zillow's in-house price prediction model, but we use a slightly different problem for illustration

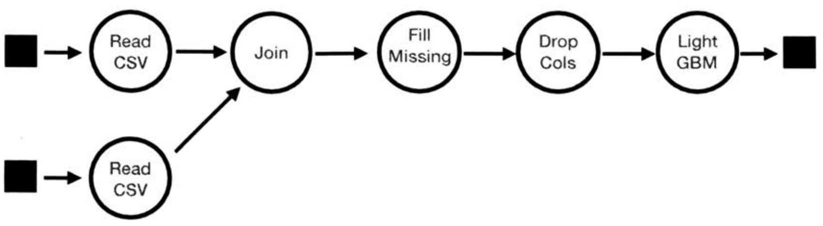


Figure 2-4: Sample TRAD pipeline for home price prediction

2.5.2 Deep Neural Networks

The second modeling method we study in this thesis is machine learning using Deep Neural Networks. Specifically, in this work, we focus on **Deep Feedforward Neural Networks** [51]. Unlike TRAD machine learning where the data scientist or ML developer must hand-craft features to create an appropriate representation for the model, a DNN model learns a representation from the raw data. A (feedforward) DNN consists of multiple layers of computational units (also called neurons) such that each computational unit learns functions of the input that can capture factors relevant for the machine learning task. Fig. 2-5 shows a schematic of a toy DNN containing a single layer. One of the hallmarks of DNNs is that they learn a hierarchy of functions such that functions in higher layers (i.e., layers away from the input) are composed of functions learned in the lower layers (i.e., layers close to the input). As a result, by increasing the number of units in each layer, we can learn more functions whereas by increasing the number of layers, we can learn more compositions of functions. The **architecture** of a DNN refers to the structure of the network, specifically, the different units in the network and how they are connected. In most architectures, units are organized into layers such that the output of layer k can be obtained by only using the outputs of layer $k-1$. In a neural network, the layer that reads in the training data is called the **input layer** while the layer that produces the model output is called the **output layer**. The layers between these two layers are called the **hidden layers**. Fig. 2-5 shows these layers for our toy DNN.

In a *feedforward* DNN or simply a multi-layer perceptron, information only flows from the input through the lower layers to the higher layers and to the output; there are no feedback connections that connect the output back to the input. The

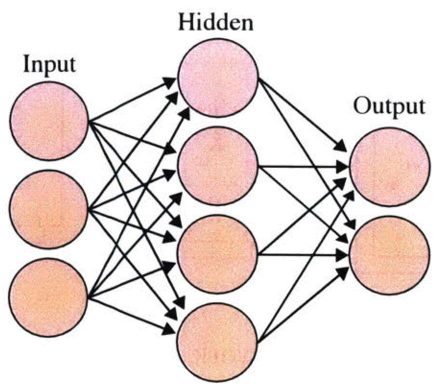


Figure 2-5: Toy neural network

class of feedforward neural networks includes fully connected neural networks and convolutional neural networks (CNNs). Convolutional neural networks are a specific type of feedforward DNNs optimized to process data with a grid-structure such as images and time series [51]. In this work, we limit our selves to CNNs that work with image data. These networks are called convolutional because they apply the convolution operation in at least one of their layers where a convolution operation (in the context of DNNs) involves applying a windowed element-wise matrix multiplication operation to the input data using another matrix called the kernel (see Chap. 9 of [51] for a detailed explanation). The second type of operation commonly found in CNNs is the *pooling* operation; this simply corresponds to applying a windowed spatial aggregation operation (e.g., with a MAX or AVERAGE operation) over a matrix.

For DNNs, we work with a running example related to image classification. The task here is to classify an input image as belonging to a class such as cat, dog, ship, bird etc. Fig. 2-6 shows the schematic of a popular deep neural network used for image classification called VGG-16 [116]. VGG-16 contains 16 convolutional layers, 5 pooling layers, and three fully connected layers. It takes as input an RGB image and passes it through the various layers to ultimately produce a vector of size 1000, each index indicating the probability that the image belongs to one of the 1000 categories it was trained to predict (see the ImageNet competition [108]).

In this work, we focus solely on image networks, and leave for future work the study of deep neural networks for different data types such as text and time-series. Even

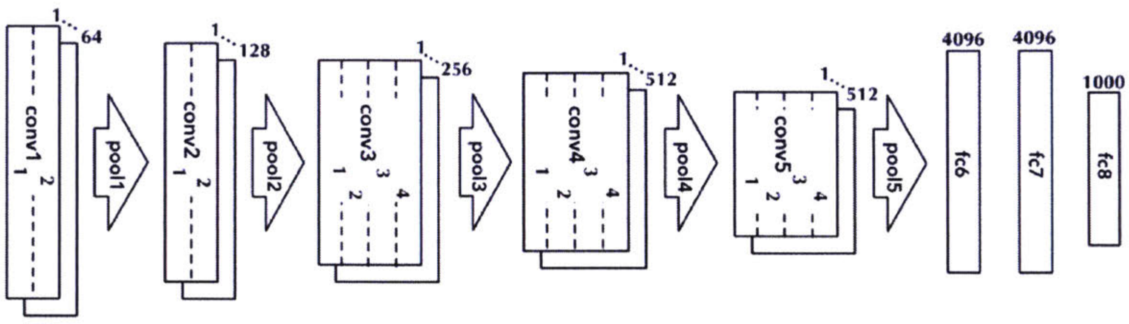


Figure 2-6: VGG16 network for image classification

within image networks, we focus primarily on networks with three types of layers: convolutional layers, pooling layers, and fully connected layers. While our techniques could support architectures such as ResNets [124], Recurrent Neural Networks [51], we do not study them at this time. Additionally, we do not cover neural network architectures that can change during training (e.g., as described in [91]).

We can represent DNNs via the ML-based workflow abstraction in several ways (each appropriate in different settings). At one extreme, we can abstract the entire DNN into a single operator where all layers and neurons in a network are grouped together; parameters for this operator are the parameters of the entire neural network. Applying this operator means running the entire neural network and an ML-based workflow with a DNN represented in this manner would have only two stages: one for reading data and another for applying the DNN model. At the other extreme, we can consider each neuron in the DNN as a separate operator and have as many operators in our workflow as neurons in the DNN. Parameters of each operator in this case are only the parameters corresponding to one neuron. Our workflow would thus have thousands of operators in a densely connected DAG. A middle ground between the two extreme approaches is to define workflow operators at the level of layers, i.e., groups of neurons. Since neurons in a given layer usually share the same inputs, we can obtain a workflow with fewer operators that are (largely) connected in a linear sequence. For example, for the VGG16 network described above, the workflow would have 24 operators corresponding to the 24 layers. Parameters for the workflow operators in this case would be the weights corresponding to an entire DNN layer.

We use different workflow representations of DNNs in this thesis. In MODELDB, we adopt the first (single operator) representation of DNNs for ease of capturing workflows; for MISTIQUE, however, we are concerned with analyzing internal state of models, and therefore we adopt a layer-level representation of DNNs.

2.6 Modeling Frameworks

A large number of frameworks are now available for machine learning, both using traditional ML techniques and deep neural networks. We briefly survey these frameworks with particular emphasis on the frameworks that we will study in subsequent chapters.

2.6.1 Traditional ML

A variety of frameworks for traditional machine learning are available in different languages. For example, scikit-learn [98] is the most popular open-source library for traditional machine learning in Python. It includes a large array of machine learning models for a variety of supervised and unsupervised tasks, and also provides functionality for data pre-processing and feature engineering. scikit-learn is often used is conjunction with pandas[6], a data processing library as well as numpy[7] and scipy[8] libraries for array-based numeric computation. spark.ml[9] is an open-source ML framework available in the Apache Spark distributed processing environment. To perform machine learning, it provides an interface very similar to scikit-learn and is available as a library in Scala as well as Python. R [103] is an extremely popular statistical computation environment that provides libraries with a variety of ML algorithms.

[6]https://pandas.pydata.org/
[7]http://www.numpy.org/
[8]https://www.scipy.org/
[9]https://spark.apache.org/docs/latest/ml-guide.html

Machine Learning API in scikit-learn

We briefly review the ML API available in scikit-learn because it mirrors the API in spark.ml and motivates our design choices in MODELDB.

The scikit-learn machine learning API consists of three main interfaces [20]: the *estimator* interface used for building and fitting models, the *predictor* interface used for making predictions based on trained models, and the *transformer* interface that is used to apply different transformations to data. In scikit-learn, a single entity (namely a model class) is used to represent the abstract notion of a model, the learning algorithm used to tune the parameters of the model, the hyperparameters of the algorithm, and trained model. The model class implements both the estimator and predictor interfaces defined above. A developer *initializes* an empty model (e.g., a RandomForest model) and optionally sets hyperparameters for this model. The developer can then supply training data and call `fit()` on the model to tune the model parameters to fit the training data. Once a model has been trained, the developer can use the model by calling the `predict()` function on the trained model.

Besides model definition and training, scikit-learn also provides various pre-processing modules that can be used for feature engineering. These pre-processing modules implement the transformer interface defined above (and occassionally even the estimator interface when modules must store state about the data). By calling `transform()`, these modules can be used to apply different pre-processing steps to the data. Examples of transformers include those that perform simple transformations such as 0-mean and 1-variance scaling and converting strings to integers to ones performing complex transformations like performing dimensionality reduction using principal component analysis.

The listing below shows a simple, end-to-end sample of how traditional machine learning can be performed in scikit-learn using the APIs described above.

Listing 2.1: Sample TRAD pipeline in scikit-learn

```
    # read data
    data = pd.read_csv(filename)

    # one hot encode column occupation
    data_ohe = pd.get_dummies(data, columns=['occupation'],
      drop_first=True)

    # scale the features
    scaler = preprocessing.StandardScaler()
    scaler.fit(data_ohe)
    data_ohe_scaled = scaler.transform(data_ohe)

    # build training set
    y_train = data_ohe_scaled['income']
    x_train = data_ohe_scaled.drop(columns=['income'])

    model = linear_model.LinearRegression()
    model.fit(x_train, y_train)

    return model
```

2.6.2 Deep Neural Networks

Part of the popularity of DNNs has resulted from the availability of a large number of frameworks for DNNs. These include Tensorflow [6] released by Google, PyTorch [96] from the open-source community (based on the previous Torch [31] framework), MXNet [28], and older libraries such as Caffe2 [118], Theano [130] and PyLearn2 [53]. There are also different wrapper libraries available on top of these frameworks that make it easy to define new models. One such library used in MISTIQUE is called Keras [30]. Keras provides an easy means to define and train DNN models in Tensorflow (as well as Theano).

Many of the above frameworks, prominently Tensorflow and Pytorch, use Python as the language of choice to define DNN models. This allows ML developers to use familiar libraries like pandas, numpy, and scipy described above. Unlike TRAD ML, where the training and testing of models is usually performed on the CPU, training and execution of DNN models is preferentially performed on the GPU. The reason is that operations commonly performed in DNNs can be easily vectorized and can therefore take advantage of the high throughput provided by GPUs.

DNN models are usually defined in terms of the *computation graph* of the neural network. Specifically, we define (a) what computational units are present in the network, (b) how they are connected, and (c) what operations are performed at each neuron. This computation graph can be specified at different levels of abstraction. For example, the computation graph may be defined in terms of (relatively low level) operations such as matrix multiplications and sigmoid functions, or simply in terms of layers of units and their associated operations (e.g., a fully connected layer with 100 units on the input and 10 units at the output).

To illustrate, a simple CNN architecture (and model) for image classification can be defined as shown below using the high-level APIs of the Keras library.

Listing 2.2: Keras CNN definition

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation,
   Flatten
from keras.layers import Conv2D, MaxPooling2D

input_shape = (32, 32, 3)
input_tensor = Input(shape=input_shape)

num_classes = 10
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
  input_shape=input_shape))
```

```
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))

return model
```

Chapter 3

Related Work

The work described in this thesis is related to and builds upon previous work in a variety of areas. We divide the work into the areas of (a) ML Model lifecycle management; (b) Workflow and Experiment Management; (c) Provenance capture and storage; (d) Data Versioning and Storage; and (e) ML model diagnosis and visualization techniques.

3.1 ML Model lifecycle management

As machine learning deployments are becoming ubiqutous, various systems have been developed in academia and in industry to manage the lifecycle of ML-based workflows. To the best of our knowledge, ModelDB was the first open-source system for model management. Several other systems have been proposed in academia and industry to solve similar problems. For example, the ModelHub [86] system was proposed to address the problem of managing the lifecycle of deep learning models. Specifically, this work proposed a domain specific language to allow easy exploration of a number of models, a (directory-based) model versioning system, and a storage system for DNN parameters. Some of the techniques proposed in MISTIQUE (e.g., quantization) have similarities to those used to compress DNN parameters in ModelHub. However, we note that the algorithms proposed in [86] for optimally determining which model versions to materialize have limited applicability in the MISTIQUE setting since the set of model intermediates we wish to store is not known ahead of time. The vision for *model selection management systems* which addresses problems similar to ModelDB was laid out by Kumar et. al. in [77]. As also noted in [77], providing a central repository and representation for ML-based workflows (as done in ModelDB) is a key requirement in speeding up the modeling process.

In [109], Sculley et. al. elegantly present the challenges with building and productionizing ML-based workflows based on their experience building ML-based products at Google. Two of the problems highlighted in [109] are the technical debt arising from "pipeline jungles" and large amount of configurations. ModelDB takes first steps in addressing these problems by providing a central repository where pipelines (i.e., workflows) and their configurations are stored in a standardized format.

Different companies have since published the architecture of their proprietary ML platforms such as the Michelangelo platform at Uber [47] and FBLearner at Facebook [46]. Similarly, in [131] authors describe a proprietary pipeline versioning system developed to track ML pipelines. Unlike ModelDB, this system performs versioning at the pipeline-stage granularity and versions not only the operators used at each stage but also the intermediate datasets produced by each stage.

Another important problem (not addressed in ModelDB or MISTIQUE) is efficiently training a large number of variations of ML-based workflows. Several papers explore various systems optimizations that can be leveraged when training ML-based workflows. The Columbus system [142], for example, studied the problem of optimizing the feature selection process by taking advantage of caching and sharing intermediate results in R programs. KeystoneML [119] tackled a similar problem by proposing a new framework to express ML pipelines using high-level primitives and optimizing their execution in a distributed setting. [137] presents early work in developing a similar system for speeding up ML workflow building.

3.2 Workflow and Experiment Management Systems

Our work on MODELDB is closely related to work on workflow and experiment management systems. Scientific workflow management is a rich area of research that has produced systems including Kepler [82] (built on top of Ptolemy II [44] and used to define workflows with different execution models), Taverna [135] workbench (used to design and execute a variety of life science workflows), Galaxy [129] (used for reproducible computational biomedical research), experiment workbench on Emulab [43] (used to replay network research) and many others. More broadly, the importance of *history* (i.e., providing a record of past operations) as an essential operation in data analysis has been highlighted by Shneiderman and others in [113, 60]. In line with this finding, [75] developed history tools for visual data mining, while [107] developed interactive visualizations of the visual analysis process in Re-Visualization. Graphical histories for visual analysis in the context of Tableau [5] were studied by Heer et. al. in [59].

The system closest in flavor and functionality to ModelDB is the VisTrails [14, 23, 22] system[1]. Building a visualization to answer a particular question is a trial-and-error exploratory process similar to building an ML model [132]. As a result, there is a need to track the provenance of visualizations as well. VisTrails provides users with the ability to create and maintain visualization pipelines along with optimizing their execution. A *vistrail* is a formal specification of a visualization pipeline. Users can build visualizations (and vistrails) using the VisTrails Builder GUI. The builder provides various modules wrapping key pieces of functionality such as data ingest, different plotting functions etc. Similar to MODELDB, VisTrails represents a visualization pipeline as a sequence of operations used to build a visualization. This specification is expressed in XML format. VisTrails uses this specification to both execute pipelines (a functionality not supported by MODELDB) and to optimize their execution.

[1]VisTrails was first introduced as a solution to track visual analysis workflows and has now evolved to support generalized scientific workflows, including support for creating workflows using scikit-learn.

A major drawback of VisTrails and most of the scientific workflow systems described above is that they require scientists to use a system-specific workflow definition interface (GUI or otherwise) that is separate from their scientific development environment. For example, VisTrails requires that every visualization be built in their system and Kelper requires the use of the Vergil interface introduced in Ptolemy. This is, in fact, the solution also adopted by some commercial ML systems including Microsoft Azure ML [1] and the SeaHorse product from DeepSense [4]. From our interviews with dozens of data scientists, however, we learned that data scientists found the use of a standalone workflow management systems (particularly GUI-based) extremely restrictive. Data scientists want the freedom to write workflows in their ML environment of choice and the ability to use new ML techniques without waiting for the workflow management system to reflect these updates. Moreover, we found that data scientists were unwilling to switch to new tools for one piece of functionality (however key the functionality may be). Thus we found that while a standalone workflow specification system would be much easier to build, it would have immense difficulty getting adoption with ML developers. As a result, a significant part of the effort in designing and implementing ModelDB was directed towards passively collecting ML-based workflows without requiring data scientists to change their modeling processes.

One system that embodies this philosophy for passively collecting provenance is the noWorkflow system built to collect provenance of Python scripts [90] (this functionality was extended to IPython notebooks by [100]). noWorkflow requires no instrumentation of user code nor a workflow management system. It uses a combination of static analysis, UDF instrumentation (at the language level) and environment profiling to capture provenance of Python scripts. Specifically, noWorkflow gathers data about definitions of functions, details about the environment in which the script was run (including libraries and versions) and details about function executions at run-time. In some ways, via static analysis and profiling APIs, noWorkflow can capture workflow data that is extremely similar to that captured in ModelDB. A drawback of automatically tracking all UDF calls, however, is the resulting performance overhead and the challenges in extending this approach to languages other than Python.

A hybrid approach is used by new workflow system such as Apache Airflow [49] and Luigi [120] where workflows are defined in code but the steps (or operators in ModelDB parlance) must follow a uniform interface. For example, workflows in Apache Airflow are DAGs of Tasks where a Task implements the common Operator interface. Every execution of the DAG creates a Run that is executed with the particular task instances. Defining workflows in code enables ML developers to use their preferred ML or data processing libraries while still making it easy to track and run workflows. The disadvantge of this solution however is that ML developers may end up making tasks large and monolithic, thus, making the workflow tracking ineffective. In ModelDB, instead of giving ML developers complete freedom to define Operators (and also to alleviate the need of each ML developer writing the same Operators), we leverage the fact that the ML libraries we target provide key interfaces (e.g., `Estimator`, `Transformer`) that we can track automatically. Thus, we get much of the instrumentation for free whereas ML developers writing custom functions can implement common interfaces (much as in Airflow) to enable logging of custom functions too. In the future, we imagine adding ModelDB clients for workflow systems such as Airflow and Luigi.

Workflow management is also related to the area of metadata management. For example, ProvDB [87] provides a system to manage metadata collected via collaborative data science workflows. Also, the Ground system [61] by Hellerstein et. al. provides a common framework for tracking data origin and use via generic abstractions that are applicable to data processing as well as ML workflows. In the future, ModelDB could be extended to incorporate data from Ground as well.

3.3 Provenance capture and storage

Lineage or provenance capture and storage has been a rich area of work in the database community. The questions commonly answered with provenance include *where* did the data come from, *why* was a particular value produced, and *how* a particular value was derived (see surveys, e.g., [21, 19]). Work in this area falls into two broad classes: tracking fine-grained lineage and tracking coarse-grained lineage. Fine-grained lineage focuses on storing "cell-level" lineage whereas coarse-grained lineage only captures transformations applied at the dataset-level.

There are many examples of coarse-grained provenance systems including the large number of scientific workflow systems described above (each of these systems capture transformations that are applied to data and seek to answer questions about how a dataset came to be). Some systems such as Spark also capture data lineage as part of their core abstractions (e.g., RDDs in Spark track their own lineage [140]). Similar to coarse-grained systems, many systems have been proposed to collect and query fine-grained lineage data (example-level or cell-level) for specific data types and computation models. An example of such a system is SubZero [136] which tracks fine-grained provenance for array data in SciDB and develops new techniques to track the input-output dependencies at the cell-level. Titian [68] is another fine-grained provenance system that provides the same functionality but for Apache Spark. As with SubZero, Titian deals with the challenge of storing cell-level input-output dependencies for large datasets. Other systems for different compute environments and frameworks include RAMP [95] for MapReduce systems and Lipstick [10] for PigLatin. In traditional database systems and data warehouses, the Trio [15] and Panda [64] projects have studied provenance in a variety of settings. Most recently, Zhang et. al. studied the problem of providing fine-grained provenance in KeystoneML to support diagnosis of ML pipelines [143].

Just like scientific workflow systems are examples of coarse-grained provenance systems for scientific workflows, ModelDB can also be thought of as a system to capture coarse-grained provenance for ML-based workflows. In contrast, MISTIQUE can be though of as a system to capture fine-grained provenance. However, there are a few key differences between MISTIQUE and fine-grained lineage systems. First, the questions answered by lineage systems are significantly different from those answered by a model diagnosis system. For instance, a lineage system seeks to answer queries of the form "what input record produced a particular prediction?" whereas MISTIQUE seeks to answer queries such as "find all the input examples that had high value for a given feature." Second, in many ML models, we do not require specialized systems for fine-grained (cell-level) lineage since this data can be obtained via the existing forward and backward propagation mechanisms that are used to train models such as DNNs [51]. And third, none of the fine-grained lineage systems address the problem of *storing* intermediate results like in MISTIQUE.

3.4 Data storage, versioning, and compression

Our goal in MISTIQUE is to store model workflow intermediates as efficiently as possible (i.e., minimizing storage footprint while keeping retrieval costs low). For ease of experimentation with different data layouts, in our implementation of MISTIQUE, we took a "relational" view of model intermediates. That is, we represented intermediates as tables. Further, for fast retrieval, we chose to represent our tables in a columnar format as in systems such as C-Store [122] and BigTable [24]. The concept of Partitions in MISTIQUE is similar in spirit to that of column family or projections in columnar database engines, although the allocation of columns to projections uses a different procedure than that used in MISTIQUE.

We also note that model workflow intermediates are often numeric arrays or tensors (e.g., for image models). As a result, they could alternatively be treated as multi-dimensional arrays and stored using an array-based storage system such a SciDB [121] or TileDB [94]. While these systems do not provide support for optimizations such as quantization, pooling or de-duplication, using array databases for intermediate storage is a promising avenue for future work.

The intermediate storage problem can also be viewed as a dataset versioning problem. Each intermediate within and across workflows can be considered to be a *version* of the input data and our goal is to store the data versions as efficiently as possible. The problem of dataset versioning has been well studied in relational as well as array-based storage systems. For example, the Decibel [84] system proposed efficient techniques to store versions of a relational dataset. In contrast, OrpheusDB [63] proposed a method to "bolt-on" versioning for traditional relational databases. Similarly, Bhattacherjee et. al. [17] studied the problem of recreating vs. storing relational datasets. While the techniques proposed in [17] are powerful, they have limited applicability in our setting because dataset "versions" (i.e., intermediates) in MISTIQUE may be quite different from one another and the complete set of data versions is not known apriori. On the side of array databases, [110] tackled the question of storing multiple versions of array data by taking advantage of delta encoding and compression, but did not consider quantization or summarization schemes. Finally, the DataHub [16] system proposed a vision for an architecture to perform collaborative data analysis by sharing datasets.

Besides dataset versioning, our proposed quantization strategies are similar to those used to compress neural network weights as in the paper on DeepCompression [56] and to store array data in PStore [18]. MISTIQUE currently uses off-the-shelf compression libraries to compress Partitions that are written to disk. However, the question of compressing floating point data has been studied in multiple papers including [105, 80] and some of these techniques could be incorporated into MISTIQUE in the future. Furthermore, we could also incorporate analysis techniques that can operate directly on compressed data as in [45]. Our strategy of identifying identical ColumnChunk and only storing unique ColumnChunks is similar to de-duplication strategies used for identifying identical blocks in the networked storage system Venti [102]. Similar techniques for file and data de-duplication are covered in [97].

3.5 Model Diagnosis and Visualization

As discussed in the introduction and referenced in Table 5.1, many techniques have recently been proposed for model diagnosis and interpretability. These include visualization tools such as ActiVis [70] and DeepVis Toolbox [139] that allow users to inspect data representations learned by DNNs. These visualizations enable ML developers to spot patterns in activations, identify errors, and also begin to understand what the model is learning. Similar tools for TRAD workflows such as VizML [26] and ModelTracker [8] provide ML developers the ability to prioritize errors, examine model workflow intermediates, and study, in a limited way, the evolution of consecutive models. As we describe in Chap. 6, the availability of intermediates at every workflow stage and across any number of models can help to significantly expand visual diagnosis functionality. Our proposed PREDICTIONVISUALIZER is an example of a tool that leverages data from many models to obtain a global view of trends in model predictions. Another visualization tool that serves to study the model structure and training progress is Tensorboard [2]. Unlike the previous tools, the focus of the Tensorboard visualizations (as of this writing) is to present statistics about training (e.g., loss curves) as opposed to aiding in model diagnosis.

Another line of work related to model diagnosis is on model interpretability. As discussed in [41], interpretability covers a variety of problems related to models. These range from making the results of ML models accessible to non-experts (e.g., as in [138]) to techniques that are mainly geared towards ML developers and explain model behavior and failures. For example, LIME [106] seeks to explain model predictions by building a local linear model. In contrast, PALM [76] seeks to explain model behavior by building a set of simpler surrogate models that apply to clusters of examples. Other interpretability techniques include Netdissect [13] and SVCCA [104] that examine hidden representations of models to answer questions such as 'what real-world concepts are encoded in each neuron' and 'whether the representations learned by two models are the same'.

There is a large class of techniques that provide model interpretability using gradient information, backpropagation or data perturbation. These include saliency maps [115] that indicate the sensitivity of a CNN model to each pixel in an image, GradCam [111] that provides a heatmap of the areas in an image that were respondible for a particular classification result, and models such as Deconvolutional Networks [141] that visualize what a network is learning. Recent work also uses the

[2]`https://www.tensorflow.org/programmers_guide/graph_viz`

theory of influence functions to compute how influential a particular training point is to a particular prediction [74], and game theory to assign importance to features in complex models such as deep neural networks (e.g., [114, 83]). While these last set of techniques cannot be supported by MISTIQUE, they represent important ways to introspect model behavior.

3.6 Other

The work in ModelDB also has relations to systems that automate the building of models (e.g., Automated Statistician [42] or [146]) and tuning of hyperparameters (e.g., [117]). The data captured in ModelDB could in the future be used to augment or improve automated modeling systems. Relatedly, the workflows captured in ModelDB could also be used to identify overfitting and errors due to multiple hypothesis testing as discussed in [50, 144].

Chapter 4

ModelDB

4.1 Introduction

Building an ML-based workflow for real-world applications is an iterative process. Data scientists and ML developers experiment with tens to hundreds of ML-based workflows before identifying one that meets some acceptance criteria on workflow performance. For example, as shown in Table 4.1, top competitors in Kaggle competitions typically submit predictions from hundreds of workflows to the leaderboard. The winning team in the Zillow Home Value Prediction competition made 253 submissions whereas the rank-3 team in the Toxic Comment Classification competition made 451 submissions[1]. Although data scientists (and data science teams) build many tens to hundreds of ML-based workflows when developing an ML application, they currently have no way to keep track of all the workflows they have built.

The above observation highlights a problem that we term **ML-workflow management**. ML-based workflow management involves recording ML-based workflows including their structure, the steps involved and their respective parameters, and any additional metadata about the workflow, and making all of this information available for querying.

[1]Each submission corresponds to the results on a test set that are generated by an ML-based workflow. There is generally a 1:1 relationship between an ML-based workflow and a submission. However, since teams do not submit results from a poorly-performing workflow, the actual number of workflows built is likely to be much larger

Competition	1	2	3	4	5	6	7	8	9	10
Toxic comment classification	171	129	451	164	299	182	247	59	240	397
Mercari Price Suggestion	99	57	62	119	57	394	7	347	147	152
Camera Model identification	109	124	93	133	77	31	65	17	228	173
Recruit Visitor Forecasting	114	81	39	103	47	208	65	87	93	124
Iceberg Classifier	118	24	66	106	114	60	60	15	5	55
Speech Recognition	276	119	126	195	206	170	183	270	174	123
Favorita Sales Forecasting	117	308	265	165	145	264	49	125	63	86
Zillow Home Value Prediction	253	53	111	180	10	63	449	352	251	95
Cdiscount's Image Classification	63	155	90	95	169	96	77	12	51	61
Porto Seguro's Safe Driver Prediction	83	231	12	92	47	58	65	252	107	24

Table 4.1: Number of submissions made by top-10 competitors in different Kaggle competitions. These numbers were obtained from the private leaderboard for competitions that finished in the last six months.

To understand the need to track ML-based workflows, we studied ML-based workflows across many companies (ranging from small startups to large tech companies working at the leading edge of machine learning). We found the the lack of ML-based workflow management impacts ML developers in a variety of ways:

- *Lost insights.* Data about previously built ML-based workflows is necessary to inform the next set of ML-based workflows and experiments. As studied in human computer interaction, data about previous experiments aids with the *sensemaking* [101] process, i.e., the process of finding and organizing information to build a mental model of the underlying phenomenon. As a result, the lack of ability to record experiments can lead insights to be lost.

- *Lack of reproducibility.* Similar to the difficulties in reproducing scientific experiments [12], lack of records about previously built ML-based workflows and lack of detailed provenance information about workflows leads to results that are not reproducible. Moreover, it causes significant modeling efforts to be duplicated and resources wasted on re-training expensive workflows. For example, one ML developer at a large tech company related how she had spent over a week just re-running a modeling experiment another employee had previously conducted since the experimental setup and results were not recorded.

- *Challenges in collaboration.* Lack of a centralized location of ML-based workflows makes it challenging for team members to collaborate and managers to get insight into the ML efforts. For example, one ML team manager we interviewed related how, without information about other ML-based workflows an ML developer had built, it was challenging to assess whether the workflow ultimately chosen by the ML developer was in fact the best one for the task.

- *Challenges in running meta-analyses.* Data scientists routinely test tens to hundreds of hyperparameter settings. However, there is currently no way to keep track of all the workflows that are built, in case the ML developer would like to revisit them or observe trends performance trends caused by hyperparameters. Similarly, an ML developer may want to find trends across workflows and identify gaps in experiementation, tasks that are challenging without a central repository ML-based workflows and their metadata.

- *Lack of auditability.* ML-based workflows are increasingly used in many key business processes in a variety of industries. As a result, government legislation is now being implemented (e.g., the GDPR regulations in the European Union [93]) which require that companies be able to explain any decisions made without human intervention. In these cases, it is essential to have access to all versions of a ML-based workflow that may have been used in a task along with full provenance information about the workflow.

Additionally, we note that having a central ML-based workflow repository will also enable new kinds of applications that are currently not possible. For example, the data collected in a ML-based workflow management system could be used to identify overfitting of a model and errors due to multiple hypothesis testing (e.g., [50]) or even to recommend "best" workflows for a new task.

4.1.1 Motivating Example

Consider data scientist Dave (a real Kaggle user, name changed for privacy) who is tasked with building an ML-based workflow for the the Home Value Prediction competition described in Chap. 2.5.1. Suppose that at some point in the modeling process,

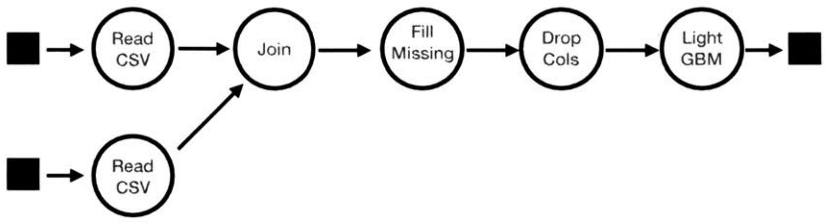


Figure 4-1: Sample TRAD pipeline for home price prediction

Dave builds the TRAD workflow shown in Fig. 4-1 to perform this task (reproduced from previous chapter).

After training and evaluating this workflow, Dave may decide that instead of using a LightGBM [72] model as shown in the figure, he wants to use an XGBoost [27] model. So he updates the workflow definition, re-trains the workflow and evaluates its performance. He finds that the XGBoost model gives him better results. He then decides that he would like to ensemble (i.e., combine) the predictions from the LightGMB and XGBoost models via a linear function in order to obtain even better performance. So he again updates his workflow to train both models and compute a weighted combination of their predictions. He then spends the next few iterations changing the weights assigned to predictions from the two models and observing the effect on workflow performance. This process continues until Dave finds a workflow that meets some acceptance criteria (e.g., error < 0.05).

The iterative modeling process decribed above is not unusual at all; in fact, it is modeled after a real modeling workflow documented in a Kaggle kernel [35]. Listing 4.1 shows actual code comments corresponding to a few workflow iterations constructed by Dave. Moreover, note that Dave is not a beginner, he is a proficient data scientist; he is ranked 501 (among all 82000 Kagglers) and has won seven medals in various Kaggle competitions. We present these comments to draw attention to three pertinent details: first, we notice that Dave has developed a large number of versions of his workflow (at least 61 based on this listing); second, we notice that many of the iterations are essentially trial-and-error; and third, we notice that the best option available to Dave to track evolution of his workflow is to manually make notes in his code comments.

Listing 4.1: Model versioning comments by a Kaggle competitor

```
# version 61
#   Drop fireplacecnt and fireplaceflag, following Jayaraman:
#     https://www.kaggle.com/valadi/xgb-w-o-outliers-lgb-with-
   outliers-combo-tune5

# version 60
#   Try BASELINE_PRED=0.0115, since that's the actual baseline
   from
#     https://www.kaggle.com/aharless/oleg-s-original-better-
   baseline

# version 59
#   Looks like 0.0056 is the optimum BASELINE_WEIGHT

# versions 57, 58
#   Playing with BASELINE_WEIGHT parameter:
#     3 values will determine quadratic approximation of optimum

# version 55
#   OK, it doesn't get the same result, but I also get a
   different result
#     if I fork the earlier version and run it again.
#   So something weird is going on (maybe software upgrade??)
#   I'm just going to submit this version and make it my new
   benchmark.

# version 53
#   Re-parameterize ensemble (should get same result).

# version 51
#   Quadratic approximation based on last 3 submissions gives
   0.3533
#     as optimal lgb_weight.  To be slightly conservative,
#     I'm rounding down to 0.35

# version 50
#   Quadratic approximation based on last 3 submissions gives
   0.3073
#     as optimal lgb_weight

# version 49
#   My latest quadratic approximation is concave, so I'm just
   taking
#     a shot in the dark with lgb_weight=.3

# version 45
#   Increase lgb_weight to 0.25 based on new quadratic
   approximation.
#   Based on scores for versions 41, 43, and 44, the optimum is
   0.261
#     if I've done the calculations right.
#   I'm being conservative and only going 2/3 of the way there.
#   (FWIW my best guess is that even this will get a worse score,
#    but you gotta pay some attention to the math.)

# version 44
#   Increase lgb_weight to 0.23, per Nikunj's suggestion, even
   though
#     my quadratic approximation said I was already at the
   optimum
```

To address the problem of ML-based workflow management, we propose a novel system called ModelDB. ModelDB automatically tracks ML-based workflows as they are built, records provenance information for each step in the workflow, stores this data in a standard format, makes it available for querying via an API and a visual interface.

Building a system for ML-based workflow management presents three key challenges. First, ML-based workflows are extremely diverse. They can be built using different ML libraries and in different programming languages. They may also use a variety of different data pre-processing steps as well as ML models. Therefore, it is unclear how to capture these workflows via a single system. Second, different workflows and their associated steps may have different metadata and provenance information associated with them. Consequently, it is challenging to capture all this metadata using a single unified schema. And third, since we want to build a system that we can deploy in the real world, we must minimize changes that an ML developer would need to make to their current modeling process.

In the next sections, we describe the design decisions we made to address each of the above challenges. We also outline the limitations of the current system and provide pointers for potential remediations.

4.2 Preliminaries

As defined in the Background in Chap. 2, in this work, we represent ML-based workflows as a DAG of operations. Each operation in turn can perform diverse functions; for example it may pre-process data (e.g., remove some examples), create new descriptors for the data (i.e., feature engineering), or apply a machine learning model. Further, each operator has zero or more parameters associated with it that can be tuned to fit the training data.

There are two key ways to capture ML-based workflows built during experimentation: either require ML developers to use a workflow management system (distinct from their ML development environment) in order to specify and run their workflows, or instrument ML code to capture workflows passively. The first solution is the one adopted by GUI-based machine learning workflow definition software such as Microsoft Azure ML [1] and Weka [55]. However, after interviewing many ML developers we found that data scientists usually have a preferred programming environment (i.e., language, software libraries) for constructing ML-based workflows and they are unwilling to switch to standalone workflow management systems. We also found that workflow management software restricts the operators that can be used in the workflow to those operators available in the software. However, ML developers want the ability to use the lastest ML models without waiting for the workflow software to incorporate these updates. Therefore, in ModelDB, we decided to adopt the second route — to instrument ML code in order to automatically extract ML-based workflows. Specifically, we allow ML developers to build workflows in their favorite ML environments and augment these environments with libraries to extract and log ML-based workflows.

We accomplish the task of passively capturing workflows via two key design elements: (a) we develop a set of common abstractions for representing different types of ML-based workflows (described next), and (b) we instrument key functions in different ML libraries that capture a large variety of commonly performed steps in ML-based workflows (described in Chap. 4.4.2).

4.2.1 Key Abstractions

Our generic representation of an ML-based workflow as a DAG of operators affords a lot of flexibility to capture diverse workflows. Our implementation builds upon this definition to capture ML-based workflows defined in different ML libraries. Specifically, we use the following abstractions. Note that these abstractions in ModelDB are currently tailored for *traditional* ML-based workflows.

- DataFrame: The key ingredient of any ML-based workflow is the dataset. ModelDB currently only deals with TRAD workflows that operate on tables or arrays of data. We represent these datasets via an abstraction called the **DataFrame**. A DataFrame consists of rows (examples) and columns (features). ModelDB does not store the actual contents of a DataFrame but stores metadata about a dataframe such as where it came from (e.g., filename) and schema information about its columns (e.g., name and type of value).

- Workflow: This is a container abstraction that contains an ordered set of Operators (currently ModelDB only supports linear sequences of operators).
- Operator: An Operator[2] represents any operation that takes in a DataFrame and produces a DataFrame. As discussed before, an Operator may be a simple operation such as scaling the data or a complex operation like applying a trained model. Operators have state that can be learned from the data and we can use algorithms to learn parameters of that state.
- OperatorSpec: This abstraction holds all the state required to completely reproduce the Operator including parameters of the operator that may be learned from the data and also any user-specified parameters (i.e., often hyperparameters for an ML model or method)[3].

Note that the abstractions presented above are used to represent the actual ML-based workflow. An ML developer will perform different procedures for model evaluation (e.g., cross-validation) and model selection (e.g., random search in the space of hyperparameter values) that will not be captured through these abstractions. ModelDB chooses to capture these *workflow building steps* separate from the workflows through the use of an asbtraction called Events (See ModelDB implementation in Chap. 4.4).

4.3 ModelDB **Architecture**

Figure 4-2 shows the high-level architecture of our system. ModelDB consists of three key components: native client libraries for different machine learning environments, a backend that stores workflow data, and a web-based visualization interface.

[2]In the ModelDB code, this abstraction is named the Transformer

[3]In the implementation, the OperatorSpec is named the TransformerSpec and it only stores the hyperparameters for a trained model. The model object is responsible for storing the parameters

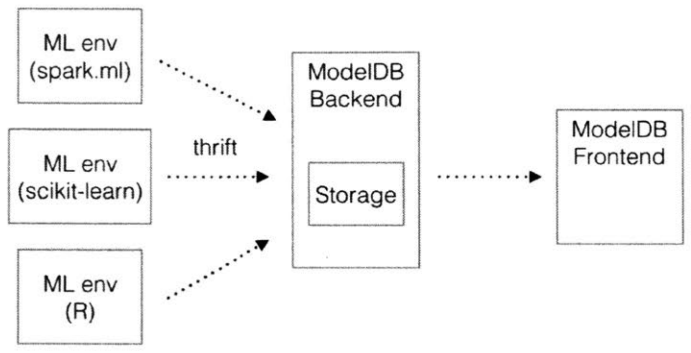


Figure 4-2: ModelDB Architecture

Client libraries are responsible for automatically extracting ML-based workflows from code and passing them to the ModelDB backend. ModelDB client libraries are currently available for scikit-learn and spark.ml. This means that ML developers can continue to build workflows and perform experimentation these environments while the native libraries passively capture workflow information. The ModelDB backend exposes a thrift[4] interface to allow clients in different languages to communicate with the ModelDB backend. The backend can use a variety of storage systems to store the workflow metadata. For ease of implementation, we chose to store ModelDB generated workflow data in a relational database whereas for workflow metadata that can have a flexible schema (e.g., user-specified key-value pairs or Light API specifications as defined later in this chapter), we chose to use a key-value store. The third component of ModelDB, the visual interface, provides an easy-to-navigate layer on top of the backend storage system that permits visual exploration and analyses of ML-based workflow data.

4.4 Implementation

Chap. 4.2 described the abstractions used by ModelDB to represent ML-based workflows. As mentioned before, in order to allow ML developers to continue building ML-based workflows in their preferred environments and minimize changes to their

[4]https://thrift.apache.org/

current modeling process or code, we chose to instrument key functions in popular ML libraries.

ModelDB client libraries capture ML-based workflow data at two levels: first, ModelDB uses an **Event** abstraction to track function calls in user code that are relevant to workflow or model building activities. For example, an event may record a transformation of data, the fitting of a model, splitting of a dataset into train-vs-test, cross-validation and so on. Once the ModelDB backend receives an event, a second level of data extraction occurs. Information about each event is processed and is represented in terms of the underlying workflow that is being built. For example, suppose the client library fires a GridSearchCrossValidation *event*. The GridSearchCrossValidation operation builds a number of ML models with different hyperparameter settings and evaluates them via cross-validation. Therefore, when a GridSearchCrossValidation event fires, a few things happen: first, ModelDB backend logs the GridSearchCrossValidation event. It then extracts information about all the models that were built as part of the operation and the corresponding cross-validation operations. It then fires GridCellCrossValidation events for each model that was built (notice *Cell* vs. *Grid* in the two event names). Each GridCellCross-Validation event then logs a CrossValidation event and stores the workflow that was built and evaluated.

In order to capture the key operations that take place in an ML environment, we examined the ML APIs for popular ML libraries and identified the most frequently used operations. As described in Chap. 2, the scikit-learn machine learning API offers three key interfaces: the estimator interface, the transformer interface, and the predict interface. Within these interfaces, the most frequently used functions are `fit` (fit a model to data), `transform` (apply an operation to some data), and `predict` (apply a trained model to data). Along with other model building functions such as such as `train_test_split`, these functions cover a significant portion of functions that are relevant for workflow-building.

The key workflow building events tracked by ModelDB are shown below; the full list is available in our GitHub repository [54]. The firing of an event in ModelDB can result in other events being fired and in a variety of data being added to ModelDB including new workflows, their metrics etc.

- Event: This is the core abstraction from which different types of events are derived.
- RandomSplitEvent: This event captures the splitting of a Dataframe into a train and test frame, and stores enough data to replicate the split.
- FitEvent: FitEvent captures the fitting of a model to a Dataframe in order to produce a trained model (i.e., a Transformer). This event stores the TransformerSpec, the input DataFrame, features, and the resulting Transformer.
- TransformEvent: This captures any transformation that is applied to a DataFrame (other than fitting of models). The event captures the input and output dataframes (only ids are stored in the database), features (i.e., columns) on which the transformer operates, and any transformer hyperparameters.
- MetricEvent: This event captures the fact that a model was evaluated against some test data to compute a metric. The event logs the evaluation dataframe, the model (transformer) tested, type of metric, and metric value.
- CrossValidationEvent: This event captures the k-fold cross-validation operation run on any model (see definition in Chap. 2). Specifically, models and metrics for each fold are captured via the CrossValidationFold abstraction.
- GridSearchCrossValidationEvent: As described before, this event corresponds to the operation of evaluating the performance (via cross-validation) of models built using different hyperparameter settings. The GridSearchCrossValidationEvent is associated with multiple events of type GridCellCrossValidation where each sub-event corresponds to cross-validation of a model using a single set of hyperparameter values.

Note that these events are not an exhaustive list of all events that are relevant to workflow building activities and in the future we can expand these events to record more functions.

Lastly, in addition to the common abstractions defined before, ModelDB also introduces three organizational abstractions to group workflows together.

- Project: All workflows related to one application are assumed to belong to the same project. A Project, in turn, is made up of one or more Experiments.
- Experiment: Workflows semantically grouped together by the user (e.g., an experiment to explore different neural network architectures) form an Experiment. An Experiment is composed of many ExperimentRuns.
- ExperimentRun: An ExperimentRun represents one execution of script or code. A re-run of the same script produces a different ExperimentRun.

DB schema

Each of the abstractions above is represented as a table in a relational database. The full schema is available at `https://github.com/mitdbg/modeldb/blob/master/server/codegen/sqlite/createDb.sql`. In addition to the abstractions defined above, we also provide users the ability to associate arbitrary key-values with any part of the workflow or ExperimentRun.

4.4.1 Native Libraries

As described above, ModelDB logs key events in ML environments so it can extract the ML-based workflow that is being built. Further, as discussed for scikit-learn, there are three functions that cover a large portion of the operations that are of interest: the `fit`, `transform`, and `predict` functions. Instead of overriding the implementations of these functions directly, ModelDB client libraries provide `sync` variants of these functions (e.g., `fit` becomes `fit_sync`). The `sync` variants not only perform the particular operation but also log the corresponding event to the ModelDB backend. While ModelDB logs a small number of function calls, these are the key functions that all ML models in scikit-learn are expected to implement.

Therefore, so long as implementers of ML models implement these interfaces, we can capture workflows containing these models.

As expected, a limitation of this technique for capturing events and workflows is that ML developers may perform the operations usually performed in the `fit` or `transform` functions using a different set of APIs. In this case, our client library would not be able to identify that operation correctly. This is a significant drawback in scikit-learn since ML-based workflows in scikit-learn often use other libraries such as numpy, scipy and pandas along with scikit-learn. However, in spark.ml, most operations follow standard interfaces (Estimator, Transformer) and ModelDB can therefore capture most workflow operations of interest. As ML developers see the advantages offered by workflow management and the need to produce re-producible workflows becomes paramount, we expect to see more standardization in how ML-based workflows are built and therefore the coverage of our functions is likely to grow. For now, if the operation being performed is not currently captured in ModelDB, then the ML developer can wrap the operation in the appropriate interface (Estimator or Transformer) or alternatively use the Light API to log the entire workflow.

Usage Example

Next, we provide an example of how ModelDB can be used to captured ML-based workflows in **spark.ml**.

To use ModelDB, a developer simply needs to import the ModelDB native client library and perform a few lines of setup to connect to the ModelDB back-end. Once that is done, the ML developer only has to use the `sync`-variants of key functions. In spark.ml, the functions `fit`, `transform`, and `predict` respectively become `fitSsync`, `transformSync`, and `predictSsync`.

The Listing 4.2 shows an example of spark.ml code without ModelDB logging whereas Listing 4.3 below shows the same spark.ml code using ModelDB.

Listing 4.2: Sample spark.ml workflow without ModelDB

```
def main(args: Array[String]) {
```

```
val sc = new SparkContext(...)

// set up spark
val spark = SparkSession
  .builder()
  .appName("Simple␣Sample")
  .getOrCreate()

// read data
val df = spark
  .read
  .option("header", true)
  .option("inferSchema", true)
  .csv(path)

// select features
val assembler = new VectorAssembler()
  .setInputCols(Array("LIMIT_BAL", "SEX",
    "EDUCATION", "MARRIAGE", "AGE"))
  .setOutputCol("features")

val transformedDf = assembler.transform(df)

// define ML model
val logReg = new LogisticRegression()
  .setLabelCol("DEFAULT")

// split data into train and test
val Array(trainDf, testDf) = transformedDf
  .randomSplit(Array(0.7, 0.3))

// train ML model
```

```
  val logRegModel = logReg.fit(trainDf)
  System.out.println(s"Coefficients:${logRegModel.coefficients}")

  // predict on test data
  val predictions = logRegModel.transform(testDf)
  predictions.printSchema()

  // evaluate performance of model
  val evaluator = new BinaryClassificationEvaluator()
    .setLabelCol("DEFAULT")

  val metric = evaluator.evaluateSync(predictions, logRegModel)
  System.out.println(s"Metric:${metric}")
}
```

Listing 4.3: Sample spark.ml workflow with ModelDB

```
def main(args: Array[String]) {
  // set up spark
  val sc = new SparkContext(...)
  val spark = SparkSession
    .builder()
    .appName("Simple␣Sample")
    .getOrCreate()

  // set up modeldb
  val MODELDB_ROOT = ""
  ModelDbSyncer.setSyncer(
     new ModelDbSyncer(projectConfig = NewOrExistingProject("Demo
        ",
       "modeldbuser",
       "Test-project"
    ),
    experimentConfig = new DefaultExperiment,
    experimentRunConfig = new NewExperimentRun)
)

ModelDbSyncer.setSyncer(new ModelDbSyncer(SyncerConfig(
    MODELDB_ROOT + "/client/syncer.json")))

// read data
val path = MODELDB_ROOT + "/data/credit-default.csv"
val df = spark
  .read
  .option("header", true)
  .option("inferSchema", true)
  .csvSync(path)

// select features
val assembler = new VectorAssembler()
  .setInputCols(Array("LIMIT_BAL", "SEX",
    "EDUCATION", "MARRIAGE", "AGE"))
  .setOutputCol("features")

val transformedDf = assembler.transformSync(df)

// define model
val logReg = new LogisticRegression()
  .setLabelCol("DEFAULT")

// split data into train and test
val Array(trainDf, testDf) = transformedDf
  .randomSplitSync(Array(0.7, 0.3))
  // fit model
  val logRegModel = logReg.fitSync(trainDf)
  System.out.println(s"Coefficients:${logRegModel.coefficients}")

  // predict on test data
  val predictions = logRegModel.transformSync(testDf)
  predictions.printSchema()

  // evaluate performance of model
  val evaluator = new BinaryClassificationEvaluator()
    .setLabelCol("DEFAULT")

  val metric = evaluator.evaluateSync(predictions, logRegModel)
  System.out.println(s"Metric:${metric}")
}
```

4.4.2 ModelDB **Light API**

Since ModelDB can only log workflows from scikit-learn and spark.ml, for the remaining ML environments, we provide a *Light API*. In the Light API, a workflow built in any language or ML environment may be described using the YAML [5] or JSON [6] formats. The specification requires a small set of mandatory fields (e.g., workflow name, type, configuration, and metrics). Other than that, the ML developer has the flexiblity to include arbitary metadata via key-value pairs. For example, users often include metadata such as model authors, maintainers, and metrics over time. An example of a workflow specification using YAML is shown in Listing 4.4. Unlike the ModelDB native libraries, logging models via the light API does not (by default) include information about the workflow or model architecture; the ML developer must explicitly supply this information in the specification.

Currently, the Light API is being used to log models from environments like R

[5]http://yaml.org/
[6]https://www.json.org/

and proprietary ML frameworks. It is also being used in cases where the model has already been developed and therefore ML developers cannot retroactively use the `sync` API defined above. The code listing below shows an example of a YAML file that can be used to log arbitrary models.

Listing 4.4: Sample YAML config file

```
# define datasets used
DATASETS:
    - FILENAME: /path/to/train_file.csv
      METADATA:
        num_cols: 1
      TAG: train
    - FILENAME: /path/to/test_file.csv
      METADATA:
        num_cols: 1
      TAG: test

# define the model workflow
MODEL:
    # required key-value pairs
    NAME: Sample Model
    TYPE: Normal distributions
    PATH: path/to/model.R
    TAG: train
    CONFIG:
        l1: 20
    METRICS:
        - TYPE: accuracy
          VALUE: 0.9
    # optional kv pairs
    owner:       bob
    date-created:    2016-02-23
```

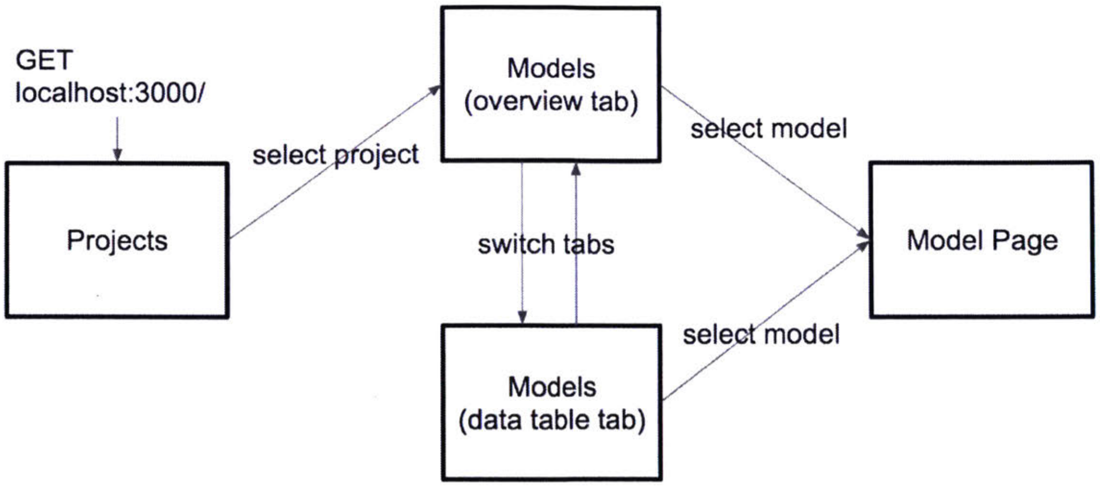


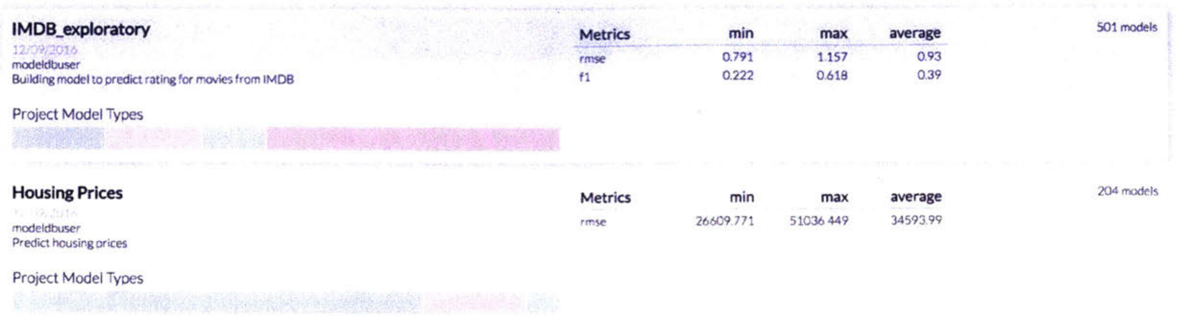
Figure 4-3: Overview of ModelDB Frontend Architecture (reproduced from [78])

4.5 Data Visualization for ModelDB

ModelDB captures a large amount of metadata about ML-based workflows. To support easy access to this data, ModelDB provides a visual interface. The interface has been implemented using nodejs and jquery. The data visualizations for ModelDB were developed in collaboration with Wei-En Lee as part of his Masters Thesis at MIT [78]. An overview of the frontend architecture is shown in Fig. 4-3.

We provide three key views for exploring the data stored in ModelDB. A user starts with the projects summary page (Fig. 4-4) that provides a high level overview of all the projects in the system including the name, description for each project, the number of ML-based workflows built for the project, accuracy metrics for each workflow, and a breakdown of the workflows by type. This view can be easily extended to add other summary fields as well.

The user can then click on a particular project to see the workflows for that project. We present models via two key visualizations, first, we present a workflow timeline page (Fig. 4-5) where the ML developer can see all the workflows built for the project, along with their quality metrics, arranged as a timeline. This view helps the ML developer see the evolution of the workflow over time. The ML developer can click on any particular workflow to view attributes of the workflow including its type



IMDB_exploratory
12/09/2016
modeldbuser
Building model to predict rating for movies from IMDB

Metrics	min	max	average
rmse	0.791	1.157	0.93
f1	0.222	0.618	0.39

501 models

Project Model Types

Housing Prices
12/09/2016
modeldbuser
Predict housing prices

Metrics	min	max	average
rmse	26609.771	51036.449	34593.99

204 models

Project Model Types

Figure 4-4: Projects Summary View (reproduced from [78])

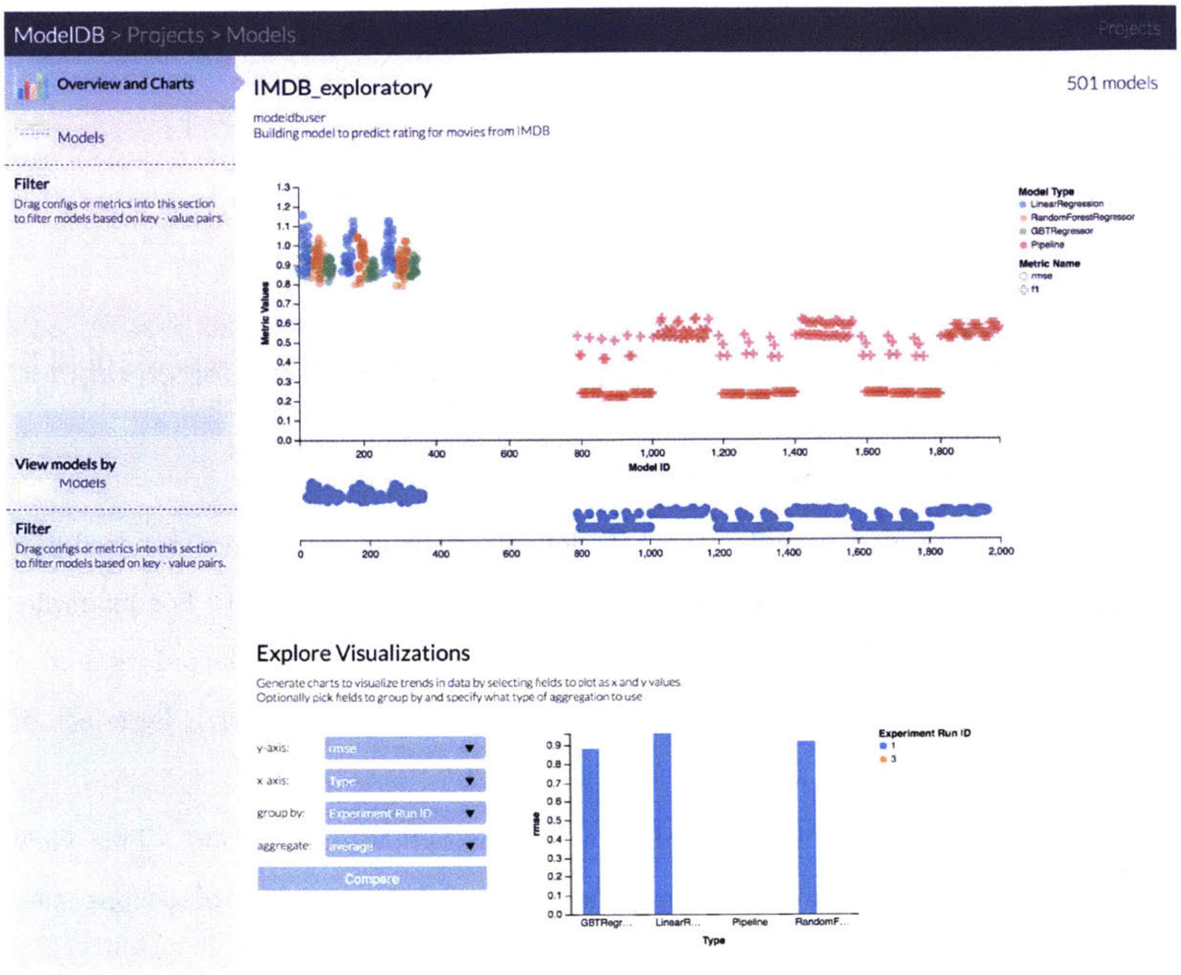
and hyperparameter settings (Fig. 4-6).

The same page also provides the ML developer an easy interface for building custom visualizations for analyzing workflow performance (Fig. 4-7). For example, any attribute of a workflow (e.g., hyperparameter values) can be plotted against a metric (e.g., accuracy) to easily see the impact of that attribute for a large set of workflows.

We also provide a tabular view (Fig. 4-8) of workflow information. This view presents all workflow information at a glance and supports a variety of actions such as filtering, sorting, grouping, and search (Fig. 4-9).

From any of the above interfaces, the ML developer can drill-down into a single workflow. Fig. 4-10 shows the single workflow page that lists various attributes of the workflow such as ML model used and hyperparameters. It also shows the workflow that was captured automatically. We can easily see the inputs to the workflow, transformations, and ML model used. We also provide the ability for the ML developer to annotate different workflows for collaboration. Last, we provide the functionality to associate arbitrary metadata with a workflow in the form of a JSON or YAML specification. This functionality is particularly useful for the Light API described earlier. Fig. 4-12 shows the visualization of this metadata.

Thus, as we can see from the visualizations above, the metadata logged in ModelDB is rich and can be explored in a variety of ways. There are certainly other, more sophisticated visualizations and operations that we can support with ModelDB. For example, various questions related to workflow analysis (e.g., comparing



IMDB_exploratory

501 models

modeldbuser

Building model to predict rating for movies from IMDB

Filter

Drag configs or metrics into this section to filter models based on key - value pairs.

View models by Models

Filter

Drag configs or metrics into this section to filter models based on key - value pairs.

Explore Visualizations

Generate charts to visualize trends in data by selecting fields to plot as x and y values

Optionally pick fields to group by and specify what type of aggregation to use

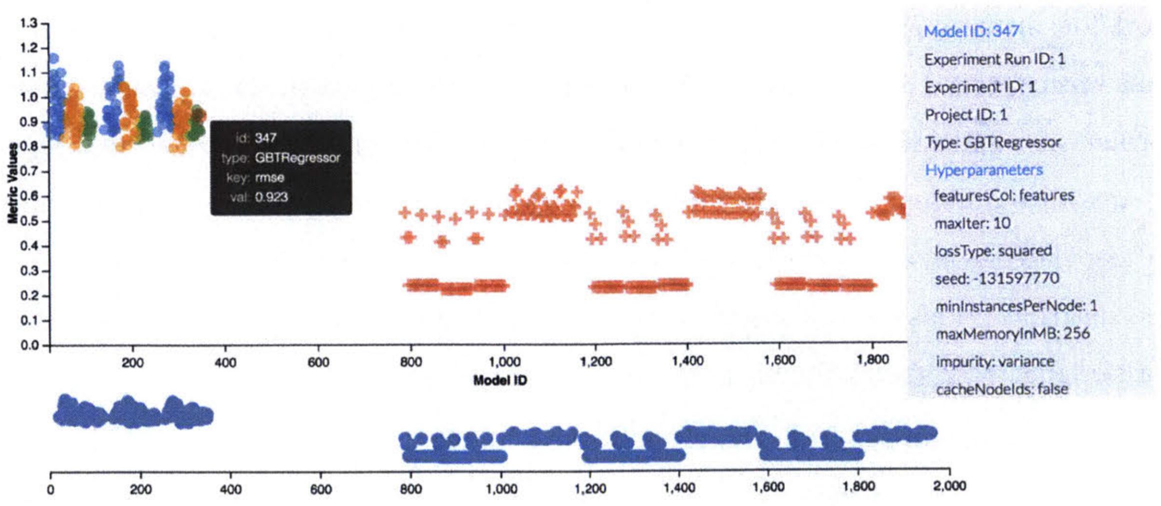
y-axis: rmse

x axis: Type

group by: Experiment Run ID

aggregate: average

Figure 4-5: Timeline Visualization (reproduced from [78])



id: 347
type: GBTRegressor
key: rmse
val: 0.923

Model ID: 347

Experiment Run ID: 1

Experiment ID: 1

Project ID: 1

Type: GBTRegressor

Hyperparameters

featuresCol: features

maxIter: 10

lossType: squared

seed: -131597770

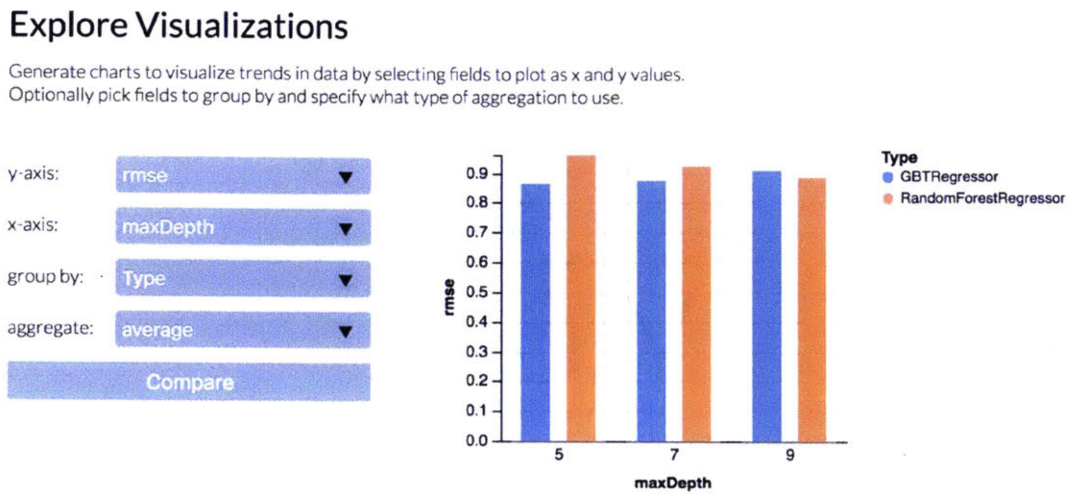
minInstancesPerNode: 1

maxMemoryInMB: 256

impurity: variance

cacheNodeIds: false

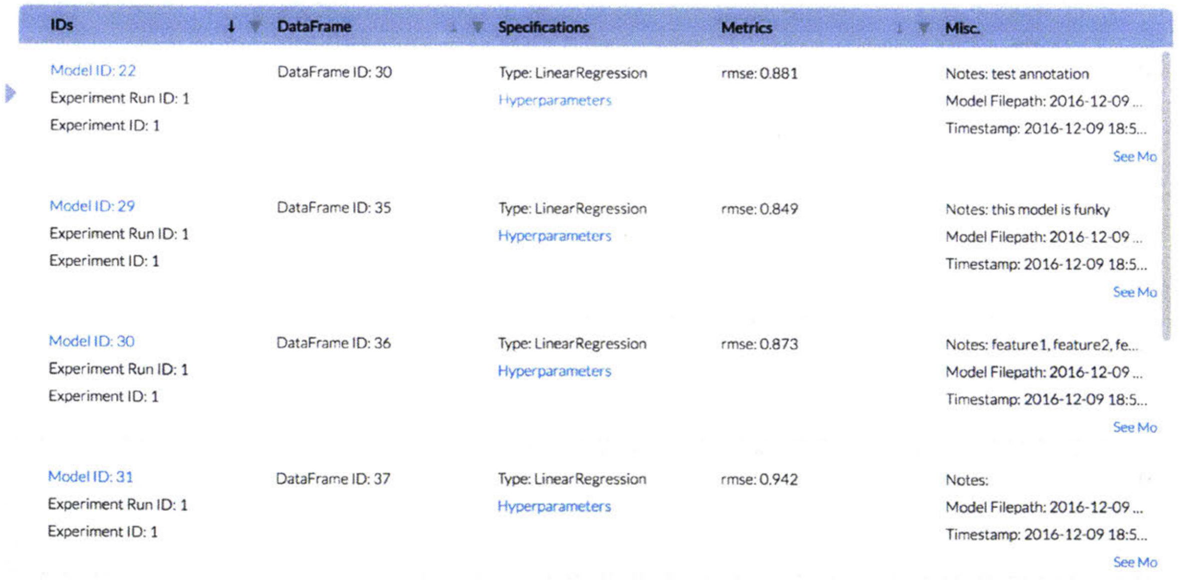
Figure 4-6: Model Timeline Drilldown (reproduced from [78])


Explore Visualizations

Generate charts to visualize trends in data by selecting fields to plot as x and y values.
Optionally pick fields to group by and specify what type of aggregation to use.

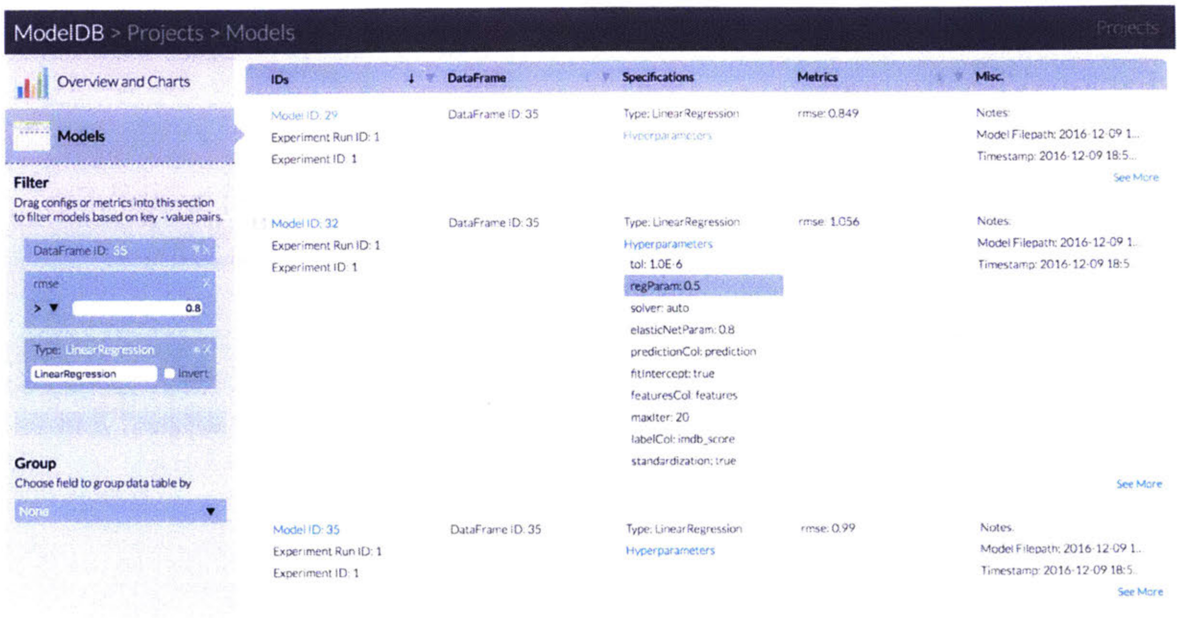
y-axis: rmse
x-axis: maxDepth
group by: Type
aggregate: average

Figure 4-7: Custom Visualizations (reproduced from [78])



IDs	DataFrame	Specifications	Metrics	Misc.
Model ID: 22 Experiment Run ID: 1 Experiment ID: 1	DataFrame ID: 30	Type: LinearRegression Hyperparameters	rmse: 0.881	Notes: test annotation Model Filepath: 2016-12-09... Timestamp: 2016-12-09 18:5... See Mo
Model ID: 29 Experiment Run ID: 1 Experiment ID: 1	DataFrame ID: 35	Type: LinearRegression Hyperparameters	rmse: 0.849	Notes: this model is funky Model Filepath: 2016-12-09... Timestamp: 2016-12-09 18:5... See Mo
Model ID: 30 Experiment Run ID: 1 Experiment ID: 1	DataFrame ID: 36	Type: LinearRegression Hyperparameters	rmse: 0.873	Notes: feature1, feature2, fe... Model Filepath: 2016-12-09... Timestamp: 2016-12-09 18:5... See Mo
Model ID: 31 Experiment Run ID: 1 Experiment ID: 1	DataFrame ID: 37	Type: LinearRegression Hyperparameters	rmse: 0.942	Notes: Model Filepath: 2016-12-09... Timestamp: 2016-12-09 18:5... See Mo

Figure 4-8: Models View (reproduced from [78])


ModelDB > Projects > Models
Projects
Overview and Charts
Models
Group
Choose field to group data table by
None

Filter

Drag configs or metrics into this section to filter models based on key - value pairs.

DataFrame ID: 35

rmse

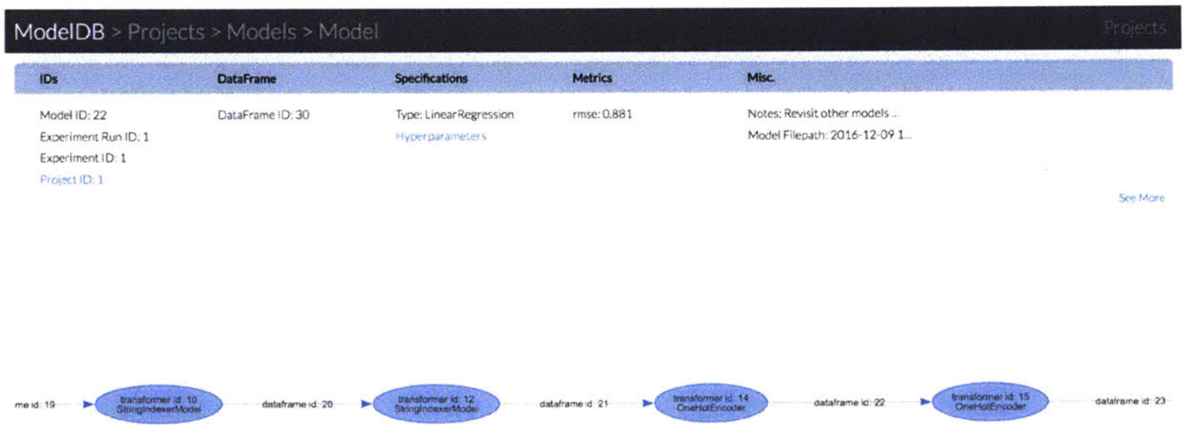
> ▼ 0.8

Type: LinearRegression

LinearRegression Invert

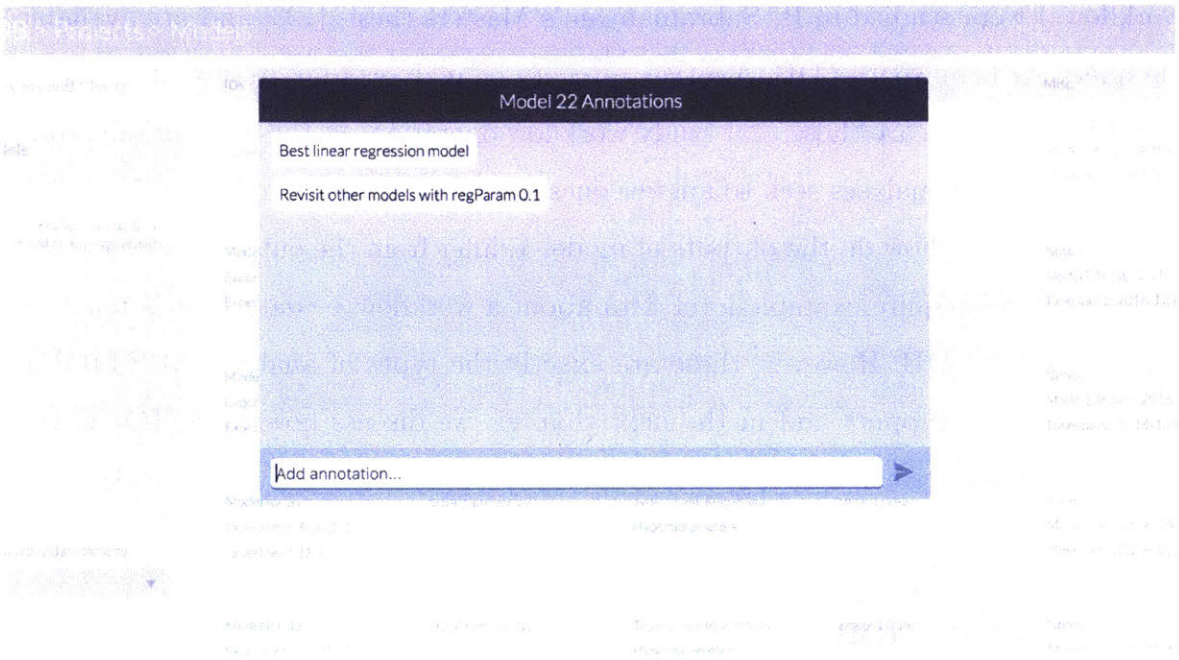
IDs	DataFrame	Specifications	Metrics	Misc.
Model ID: 29 Experiment Run ID: 1 Experiment ID: 1	DataFrame ID: 35	Type: LinearRegression Hyperparameters	rmse: 0.849	Notes: Model Filepath: 2016-12-09 1... Timestamp: 2016-12-09 18:5... See More
Model ID: 32 Experiment Run ID: 1 Experiment ID: 1	DataFrame ID: 35	Type: LinearRegression Hyperparameters tol: 1.0E-6 regParam: 0.5 solver: auto elasticNetParam: 0.8 predictionCol: prediction fitIntercept: true featuresCol: features maxIter: 20 labelCol: imdb_score standardization: true	rmse: 1.056	Notes: Model Filepath: 2016-12-09 1... Timestamp: 2016-12-09 18:5 See More
Model ID: 35 Experiment Run ID: 1 Experiment ID: 1	DataFrame ID: 35	Type: LinearRegression Hyperparameters	rmse: 0.99	Notes: Model Filepath: 2016-12-09 1... Timestamp: 2016-12-09 18:5... See More

Figure 4-9: Model Filtering (reproduced from [78])


ModelDB > Projects > Models > Model
Projects
me id: 19
transformer id: 10 StringIndexerModel
dataframe id: 20
transformer id: 12 StringIndexerModel
dataframe id: 21
transformer id: 14 OneHotEncoder
dataframe id: 22
transformer id: 15 OneHotEncoder
dataframe id: 23

IDs	DataFrame	Specifications	Metrics	Misc.
Model ID: 22 Experiment Run ID: 1 Experiment ID: 1 Project ID: 1	DataFrame ID: 30	Type: LinearRegression Hyperparameters	rmse: 0.881	Notes: Revisit other models ... Model Filepath: 2016-12-09 1... See More

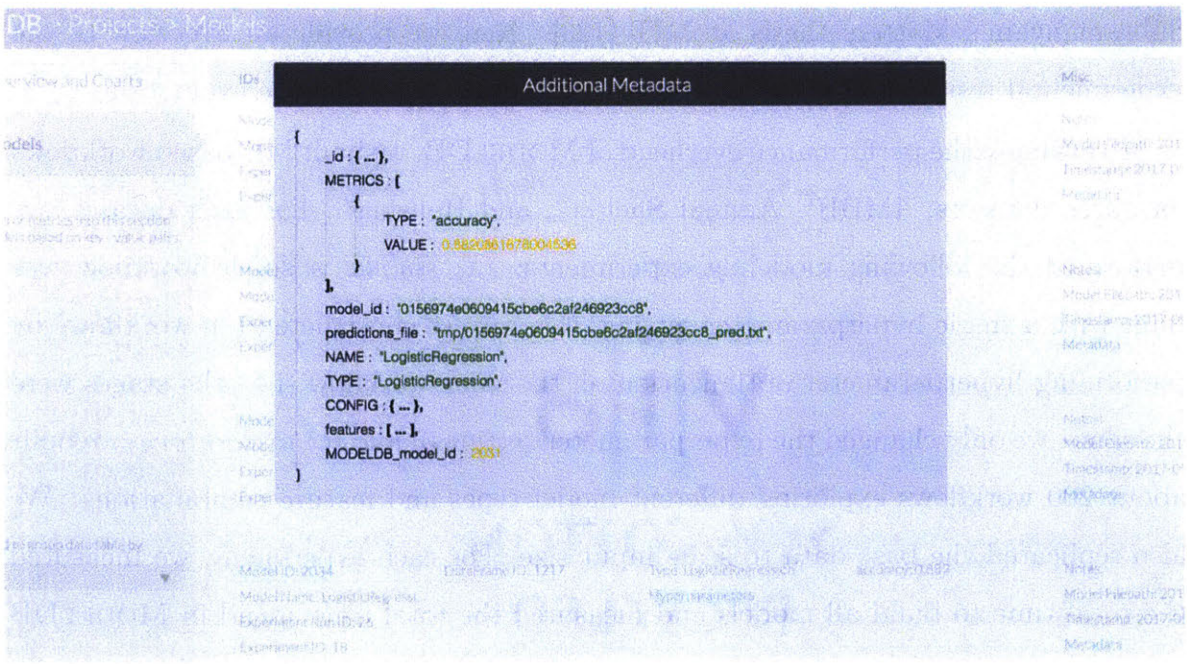
Figure 4-10: Model Pipeline View (reproduced from [78])



Best linear regression model

Revisit other models with regParam 0.1

Figure 4-11: Model Annotation (reproduced from [78])



```
{
    _id : { ... },
    METRICS : [
        {
            TYPE : "accuracy",
            VALUE : 0.8820861678004536
        }
    ],
    model_id : "0156974e0609415cbe6c2af246923cc8",
    predictions_file : "tmp/0156974e0609415cbe6c2af246923cc8_pred.txt",
    NAME : "LogisticRegression",
    TYPE : "LogisticRegression",
    CONFIG : { ... },
    features : [ ... ],
    MODELDB_model_id : 2031
}
```

Figure 4-12: Visualizing Model Metadata (reproduced from [78])

workflows, finding common ancestors for different results in a workflow, finding similar workflows) were studied in H. Subramanyam's Masters thesis [123] and are available via query APIs in ModelDB. Another category of analyses that are useful in understanding workflows are those that study workflow operations at the level of individual examples. These analyses seek to answer questions such as "what models mis-predict example-50?" or "how do the outputs of model-1 differ from the outputs of model-2?" These analyses require example-level data about a workflow — data that is not captured in ModelDB. However, these are exactly the types of analyses MISTIQUE was designed to support and in the next chapter, we discuss how MISTIQUE can answer such queries efficiently.

4.6 Evaluation

We evaluated ModelDB performance on pipelines in spark.ml with a particular focus on the overhead of ModelDB both in terms of execution time and storage. Experimental evaluation of the spark.ml ModelDB client was done as part of H. Subramanyam's Masters thesis at MIT [123]. Here we provide an overview of the experimental results and refer the reader to [123] for details on the evaluation.

To evaluate the performance overhead of ModelDB, we built ML-based workflows for three datasets: IMDB[7], Animal Shelter[8], and Housing[9]. For each dataset, we performed the following modeling experiments: (a) *simple*: a single workflow was built with a single hyperparameter setting; (b) *full*: we constructed ten workflows by performing hyperparameter optimization on the same workflow (i.e., the stages were the same, we only changed the hyperparameter settings); and (c) *exploratory*: we built about 200 workflows exploring different model types and feature combinations. We also replicated the base data to scale input size. For each experiment, we measured the total time to build all models and measured the total time spend in ModelDB

[7]New data link, old data no longer available: `https://www.kaggle.com/tmdb/tmdb-movie-metadata`

[8]`https://www.kaggle.com/c/shelter-animal-outcomes/data`

[9]`https://www.kaggle.com/c/house-prices-advanced-regression-techniques`

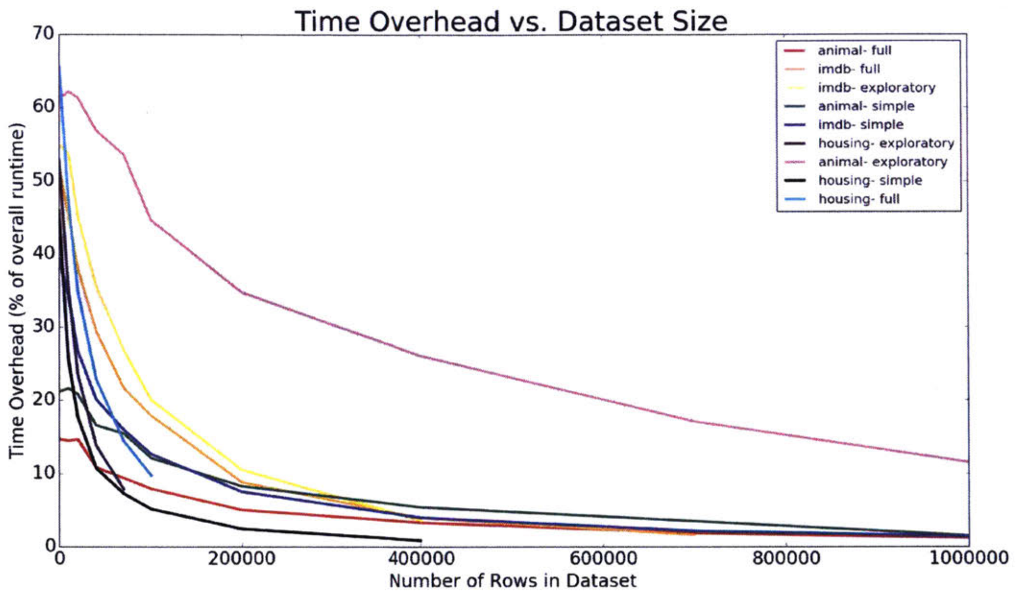


Figure 4-13: ModelDB Execution Overhead (reproduced from [123])

operations as a percentage of the total time. The results are shown in Fig. 4-13. On the x axis, we show the dataset size measured as number of rows in the table and on the y axis, we show the time overhead of using ModelDB. First, we note that the overhead depends on the number of models that are built; therefore, the overhead for simple is always lowest followed by full and then exploratory. As we can see, ModelDB can have large overhead for small datasets (<100K rows) but this overhead reduces to <10% for most datasets over 300K rows.

Next, we evaluated the cost of storing the metadata collected by ModelDB. Fig. 4-14 shows the total storage required for all the data captured by ModelDB (including model storage). Note that this experiment stores only the best model from hyperparamter optimization. As with execution overhead, we find that storage scales linearly with the number of models built and results in a neglibible storage overhead (<2 MB). If we store every model produced during hyperparamter optimization, the storage increases proportional to the number of trained models.

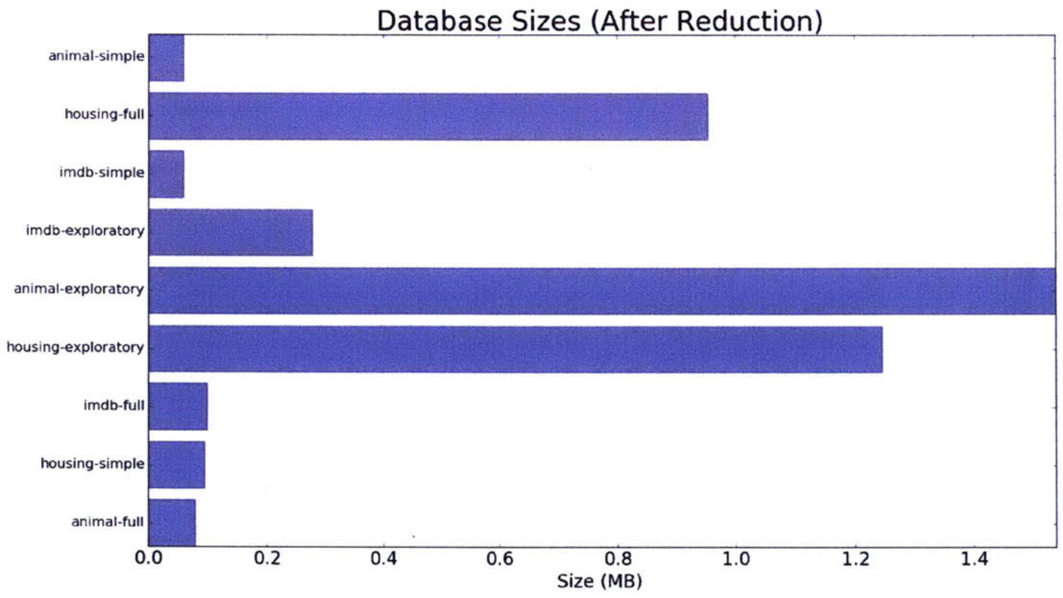


Figure 4-14: MODELDB Storage Overhead (reproduced from [123])

4.7 Adoption and Release

We officially released MODELDB as an open-source model management system in Feb. 2017 and since then there has been a large amount of interest and adoption of MODELDB. Specifically, over the last year, our GitHub repository [54] has garnered > 400 stars, has been cloned over a thousand times, and has been forked >70 times. MODELDB has been tested at multiple small and large companies, and is deployed in various settings.

Here we describe three sample deployments of MODELDB.

- **Model Management at Large Tech Company**: A large tech company X[10] has built a machine learning platform built on top of the Spark distributed computing environment. This platform supports hundreds of data scientists and ML developers at Company X who build thousands of models for customers and internal users. While thousands of models are ultimately deployed in production, an even larger number of models are built internally for research and experimentation. Before MODELDB, Company X did not have a good

[10]anonymized for privacy

solution of keeping model histories and this was reducing data scientist productivity as well as managers' visibility into their team's machine learning efforts. After ModelDB was released, Company X found that because of ModelDB's native support for spark.ml, ModelDB could be easily used to track models on their platform. As a result, Company X tested and integrated ModelDB into their ML platform. Company X currently uses the native ModelDB libraries for spark.ml to track models built on their platform and store model metadata in a central model repository. ModelDB's model tracking functionality is being used to help data scientists maintain model histories, ensure that their models are reproducible, version models, and perform meta-analyses to understand model performance. As of this writing, Company X is also making independent improvements to ModelDB to better adapt it to their internal modeling procedures.

- **Model Management at International Bank**: Bank Y[11] was one of the earliest adopters of ModelDB. Machine learning models are often used in banks for applications such as risk modeling. As with regular data science projects, a lot of experimentation is involved in developing robust risk models. Additionally, financial services is also a highly regulated industry. Therefore, model management and auditability have become essential from a regulatory perspective. As mentioned in the introduction to this chapter, many governments are introducing legislation that require financial institutions to keep track of all models used to make financial decisions and make them available for audits. Information that must be made available for audits includes workflow metadata such as the data used to build the model, the pre-processing steps used on the data, metrics about workflow performance and so on. A model management system like ModelDB is ideally suited to track this information. Therefore, Bank Y decided to integrate ModelDB into their workflow and build a central repository for workflow metadata. Unlike the tech company mentioned above, this bank uses R as their ML environment of choice. ModelDB only provides

[11]anonymized for privacy

native support for scikit-learn and spark.ml frameworks. Therefore, to enable the bank to use ModelDB in their ML workflow, we developed the Light API described before. The Light API enabled this bank to log models developed in R along with all the expected metadata. Although they were unable to log fine-grained information about the ML workflows via this API, the Light API enabled the bank to retroactively log workflows without updating any model code. The bank currently uses ModelDB to store every new version of a ML-based workflow along with metadata such as the model owners, the last time the model was trained, back-testing metrics and so on.

- **Model Management in Open-source ML Infrastructure Project**: KubeFlow [127] is an open, community driven project to make it easy to deploy and manage an ML stack on Kubernetes [128]. As described previously, an important part of ML-based workflow development is searching for hyperparameters of a model. The hyperparameter tuning module of KubeFlow, called Katib [126] uses ModelDB to track models built during hyperparameter tuning and visually explore the results as shown in Fig. 4-15. Since KubeFlow mainly works with Tensorflow models, this project is also using the Light API for ModelDB, indicating the need to provide native support for Tensorflow models.

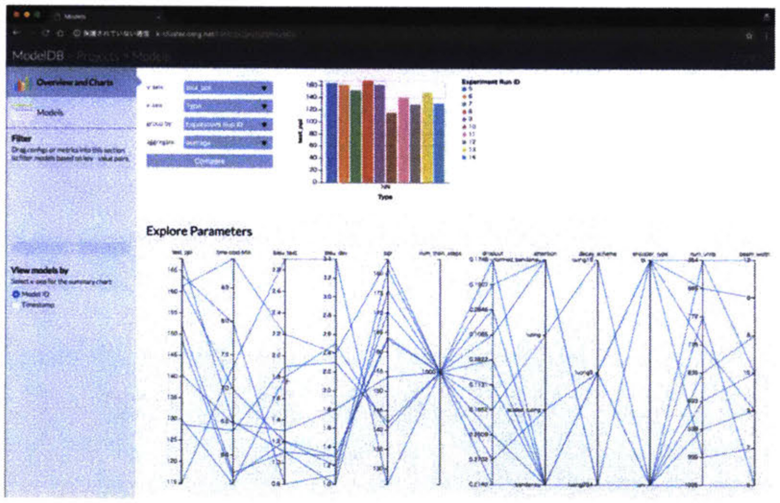


Figure 4-15: MODELDB use in Katib for managing hyperparameter tuning experiments (screenshot from [126])

Chapter 5

MISTIQUE

5.1 Introduction

Machine learning is the process of automatically extracting patterns from data. Consequently, debugging an ML-based workflow is as much about debugging data as it is about debugging code. Additionally, "bugs" in ML-based workflows can often result from patterns already present in the data as opposed to errors. As a result, we refer to model debugging instead as model diagnosis. Just as a doctor diagnoses the underlying reason for a patient's symptoms, our goal with model diagnosis is to identify the reasons why an ML-based workflow produces a particular output.

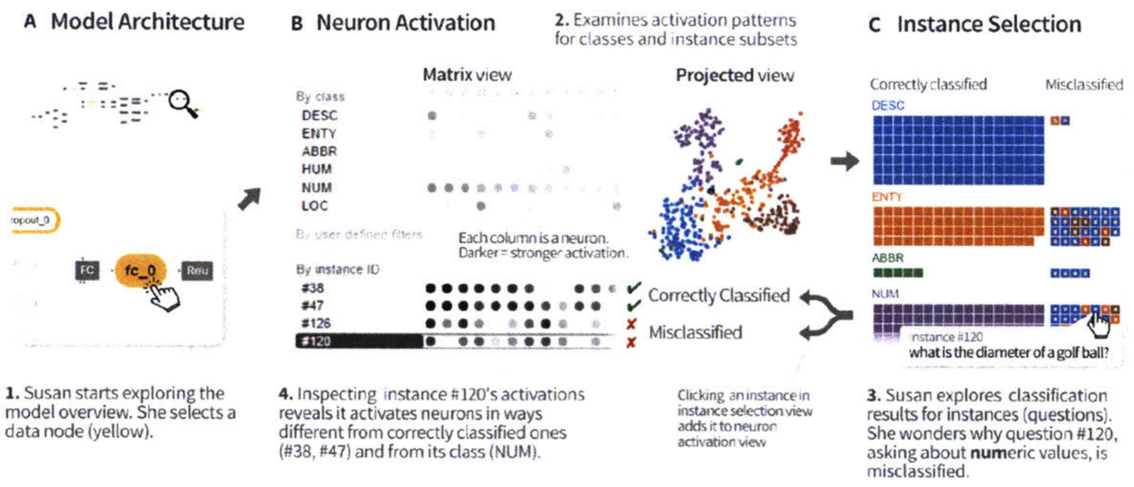
Many diagnostic queries can be answered by analyzing different data artifacts related to ML-based workflows including input data, prediction values, and data representations produced by the model or workflow operators (e.g., high-dimensional representations of homes or images learned by the model). We collectively refer to these datasets as *model workflow intermediates* (formal definition in Sec. 5.2). **Given the importance of model workflow intermediates for diagnosis, in this work, we explore the question of how to efficiently store and query intermediates to support efficient model diagnosis.** We propose MISTIQUE (**M**odel **I**ntermediate **ST**ore and **QU**ery **E**ngine), a system designed to capture, store, and query model workflow intermediates to support diagnostic queries.

5.1.1 Motivating Examples

We begin by highlighting three diagnostic techniques that have been proposed in the literature and describe the role that model workflow intermediates intermediates play in each of them. A more extensive list of techniques is presented in Chap. 5.2.2.

Visualizations

A popular means to understand the working of any model is via *visualization*. For example, the ActiVis tool from Facebook [70] (screenshot in Fig. 5-1) provides developers of neural networks an interactive visualization of neuron activations. This information can help ML developers identify activation patterns, compare activations between classes, and find potential sources of error. Similar tools have also been built for traditional modeling workflows. For example, VizML [26] (Fig. 5-2) provides an interface where ML developers can prioritize errors, examine feature distribution, and debug model results.



1. Susan starts exploring the model overview. She selects a data node (yellow).

2. Examines activation patterns for classes and instance subsets

3. Susan explores classification results for instances (questions). She wonders why question #120, asking about **numeric** values, is misclassified.

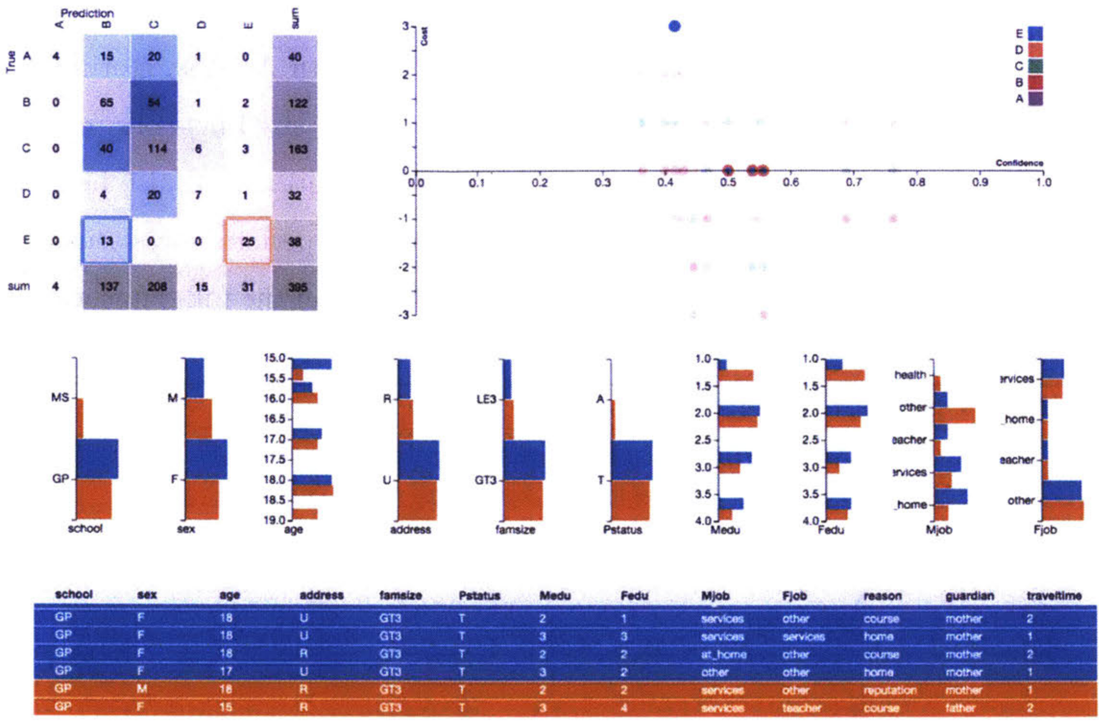
4. Inspecting instance #120's activations reveals it activates neurons in ways different from correctly classified ones (#38, #47) and from its class (NUM).

Each column is a neuron. Darker = stronger activation.

Clicking an instance in instance selection view adds it to neuron activation view

Figure 5-1: ActiVis Tool screenshot from [70]

Intermediates. In order to visualize arbitrary model intermediates, the relevant intermediates must first be generated and stored (re-running the model each time is too expensive for an interactive setting). For ActiVis, this means that data representations at each model layer must be stored. As expected, the total cost to store



	A	B	C	D	E	sum
A	4	15	20	1	0	40
B	0	65	54	1	2	122
C	0	40	114	6	3	163
D	0	4	20	7	1	32
E	0	13	0	0	25	38
sum	4	137	208	15	31	395

school	sex	age	address	famsize	Pstatus	Medu	Fedu	Mjob	Fjob	reason	guardian	traveltime
GP	F	18	U	GT3	T	2	1	services	other	course	mother	2
GP	F	18	U	GT3	T	3	3	services	services	home	mother	1
GP	F	18	R	GT3	T	2	2	at_home	other	course	mother	2
GP	F	17	U	GT3	T	3	2	other	other	home	mother	1
GP	M	18	R	GT3	T	2	2	services	other	reputation	mother	1
GP	F	15	R	GT3	T	3	4	services	teacher	course	father	2

Figure 5-2: VizML Tool screenshot from [26]

all intermediates is tremendous. E.g., storing intermediates for ten variants of the popular VGG16 network [116] on a dataset with 50K examples requires 350GB of compressed storage. As a result, ActiVis requires users to restrict the layers and number of examples for which intermediates (and aggregates of intermediates) will be logged.

SVCCA

Raghu et. al. recently proposed SVCCA [104], a technique to compare representations learned by different layers in one or more neural networks. In brief, SVCCA takes as input the activations of neurons in two layers l_1 and l_2, performs SVD on the two sets of activations, projects activations into the subspace identified by SVD, and computes canonical correlation to find directions in these subspaces that are most aligned (see Alg. 1).

Intermediates. To perform class sensitivity analyses across the whole network as described [104], activations for *all* examples at *all* layers must be available. Furthermore, if one wants to study training procedure dynamics as described in the paper, this data must be collected at every training epoch. As with ActiVis, storing data for ten training epochs would take 350GB for a moderately sized network, creating a major bottleneck in using this technique. These intermediates could also be generated anew each time the analysis was to be run; however, to perform class sensitivity analysis, this would require running the model >200 times on the full dataset.

Algorithm 1 SVCCA [104]

```
1: procedure SVCCA($A_{l_1}$, $A_{l_2}$) // activations from layers $l_1$ and $l_2$
2:     $A'_{l_1} \leftarrow$ SVD($A_{l_1}$, 0.99) // directions explaining 99% variance
3:     $A'_{l_2} \leftarrow$ SVD($A_{l_2}$, 0.99) // as above
4:     $\{\rho_i, z^{l_1}, z^{l_2}\} \leftarrow$ CCA($A'_{l_1}$, $A'_{l_2}$) // canonical correlations analysis (CCA)
5:     return $\frac{\sum_i \rho_i}{min(size(l_1), size(l_2))}$
```

Network Dissection

Bau et. al. recently proposed Netdissect [13] to learn interpretable concepts for filters in a convolutional neural network (CNN). For every convolutional filter k, Netdissect calculates the distribution of values for the the activation map (i.e., activations produced at neuron-k) $A_k(x)$ and computes a threshold T_k such that $p(A_k(x) > T_k) = \alpha$ where α is a small constant like 0.005. T_k is then used to convert A_k into a 0-1 representation (we will call this the *binarized* representation) indicating whether the activation is above the threshold or not. Finally, the correlation between the binarized representation and the original concept label is computed.

Intermediates. Netdissect requires that the activation maps for every image and every convolutional unit be available. If Netdissect is to be run for a single unit or layer, it is conceivable that the computation can be done in memory. However, when performing this computation for all units or tuning the threshold T_k (e.g., for a new dataset), then it may be more efficient to store the intermediates vs. re-running the model repeatedly.

Algorithm 2 Netdissect [13]

```
1: procedure NETDISSECT($I$, $c$, $k$, $\alpha$) // images $I$, Concept $c$, unit $k$, activation threshold $\alpha$
2:     $D_k \leftarrow A_k(I)$ // get activation maps for unit $k$
3:     $T_k \leftarrow percentile(D_k, 1-\alpha)$ // get threshold
4:     for Image x in $I$ do
5:         $A(x) \leftarrow A_k(x)$
6:         for [i,j] in $A(x)$.cells do // binarize
7:             if $A(x)[i,j] \geq T_k$ then
8:                 $A(x)[i,j] = 1$
9:             else
10:                $A(x)[i,j] = 0$
11:        $L(x) \leftarrow Labels_c(x)$ // pixel-wise label for concept
12:    return $\frac{\sum_x |L(x) \bigcap A(x)|}{\sum_x |L(x) \bigcup A(x)|}$ // intersection over union metric
```

5.1.2 MISTIQUE: storing model workflow intermediates

As demonstrated by the diagnostic techniques above, model workflow intermediates form the substrate on which a variety of diagnostic and interpretability techniques are based. However, model workflow intermediates require many tens to hundreds of GBs in storage, making it challenging to use existing diagnostic techniques as well as develop new ones. In addition, computing intermediates by re-running the model for each analytic query not only slows down the process of model diagnosis but can also be unacceptable in interactive query workloads. Thus, the bottleneck in supporting efficient and widely usable model diagnosis is caused by two data management questions: (a) how to store large amounts of data efficiently (for storage as well as querying); and (b) how to trade-off intermediate storage vs. recreation?

To address these questions, we propose MISTIQUE, a system designed to capture, store, and query model workflow intermediates for model diagnosis. MISTIQUE can work with traditional ML workflows as well as deep neural networks. MISTIQUE leverages unique properties of intermediates in both kinds of modeling approaches to drastically reduce storage costs while giving up little accuracy in most analytic techniques. Specifically, MISTIQUE is based on three key ideas: 1. Activation quantization and summarization: we take inspiration from existing diagnostic techniques

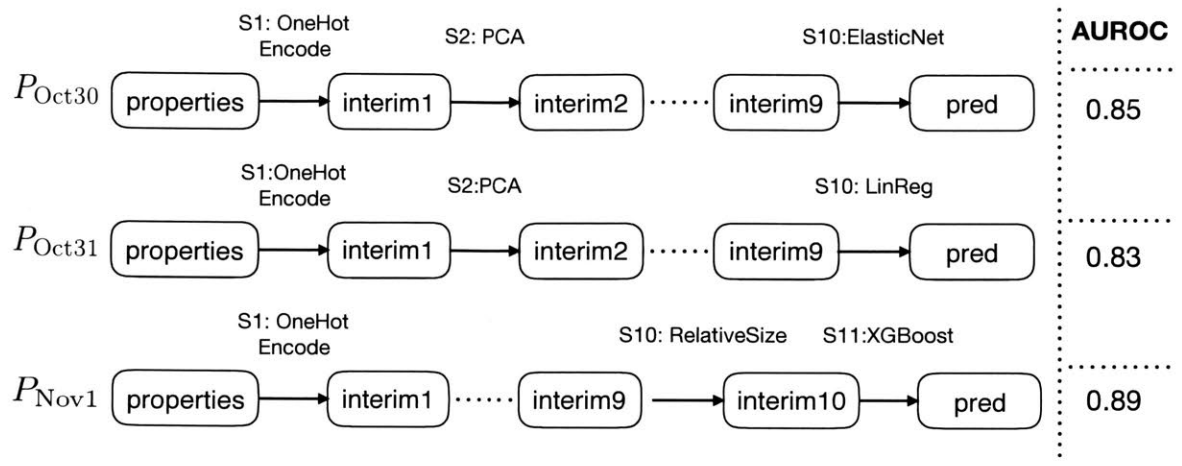


Figure 5-3: Zillow pipelines

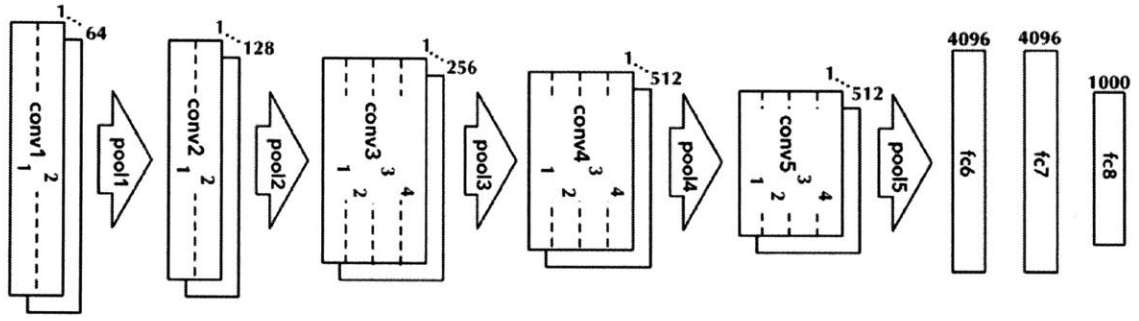


Figure 5-4: VGG16 architecture (reproduced from [116])

to encode neuron activations based on data distributions, thus getting drastic storage reductions without trading off accuracy; 2. Similarity-based compression: we leverage data similarity in traditional ML-based workflows as well as DNNs to remove redundancy between intermediates and obtain large compression ratios. 3. Adaptive querying and materialization: we propose a cost model to determine when a query for intermediates should be answered by re-running the model vs. reading a materialized intermediate. A similar cost model determines when an intermediate should be materialized. Together, these techniques can reduce storage for intermediates by up to 110X for traditional ML-based workflows and 6X for deep neural networks, and provide a query speed-up of up to two orders of magnitude.

5.2 Preliminaries

In this section, we describe the models supported by MISTIQUE and our problem formulation. We also present a list of commonly used diagnostic techniques along with a categorization based on the amount of data used by each technique.

5.2.1 Model Workflows and Intermediates

As described in the Background chapter (Chap. 2), in this work we consider two kinds of workflows: (a) traditional ML workflows (`TRAD`) and (b) workflows with deep neural networks (`DNN`).

For `TRAD`, as beforc, we consider the running example of Zillow Home Value Prediction. Fig. 5-3 shows examples of `TRAD` workflows built for this task. As noted before, the ML-based workflows contain a variety of operators performing a mixture of data pre-processing (e.g., OneHotEncoding, Scaling), feature engineering (e.g., computing the RelativeSize feature), and model training and prediction (e.g., gradient boosted trees in XGBoost [27]).

Similarly, for `DNN`s, we consider the running example of `DNN` models for classifying images. The models we work with in MISTIQUE have been trained to perform classification on the `CIFAR10`[1] dataset. This dataset consists of images drawn from ten classes such as frog, ship, deer, etc. Fig. 5-4 shows `VGG16` [116] network that was previously discussed in Chap. 2. As described before, each layer in the DNN learns different features of the input data and therefore the effect of each layer is to represent data in a high-dimensional sub-space different from the other layers. In MISTIQUE, we take a layer-level view of a `DNN` and represent it as a workflow: specifically, every layer of the `DNN` is cast as an operator in the ML-based workflow.

The process of training an ML-based workflow (`TRAD` or `DNN`) can produce different artifacts: the learned model parameters, log files, gradient information, intermediate datasets produced by different operators in the workflow, etc. In this work, we focus on model workflow intermediates that are the datasets produced at different

[1]https://www.cs.toronto.edu/ kriz/cifar.html

stages of the ML-based workflow. For traditional ML workflows, these are the results of different operators (labeled "intermX" in Fig. 5-3) whereas for DNNs, these are the hidden representations (i.e., neuron activations) produced by different layers of the neural network.

The techniques used in MISTIQUE can be used to store intermediates from many different models and workflows; these could be runs of different workflows on the same input data or runs of the same workflow on different data. As will be described in the data model (Chap 5.3.1), MISTIQUE associates a unique `row_id` with every example. Consequently, while MISTIQUE could be used in either of these settings, the optimizations described next are most effective when applied to intermediates derived from the same input data. Therefore, in our current implementation, we assume that MISTIQUE is used to store intermediates from many workflows applied to *the same input data.* With extra book-keeping, the system can be extended to support the more general case of varying input data.

5.2.2 Characterization of Diagnostic Queries

In Sec. 5.1, we highlighted three techniques used for model diagnosis. In Table 5.1 we provide a survey of diagnostic and interpretability techniques drawn from the literature. For each diagnostic technique, we show an example of the question answered by that technique (or query) in terms of our running examples. For completeness, we also include analyses that cannot be handled solely with MISTIQUE either because they require access to data other than model workflow intermediates (e.g., gradients) or because they require the ability to perturb data or models. Furthermore, to characterize the query performance of our system, we categorize diagnostic techniques based on amount of data required by each technique. Specifically, based on the number of *Rows*, i.e., input examples, and *Columns*, i.e., features, used by each technique, we define four categories: *Few Columns, Few Rows* (FCFR), *Few Columns, Many Rows* (FCMR), *Many Columns, Few Rows* (MCFR), and *Many Columns, Many Rows* (MCMR). In this work, *few* denotes <100. A typical diagnostic workload contains queries belonging to different categories; for example, one workload might be:

(i) plot the prediction error for workflow P_{Nov1} (FCMR); (ii) for the house with highest prediction error, H^*, examine its raw features (MCFR); (iii) find the performance of houses P_{Nov1} on houses "most similar" to H^* (MCFR). (iv) plot the features of H^* vs. the average features of all houses (MCMR). A system for model diagnosis must therefore be able to support queries in all four categories. In Table 5.1, we identify by name (e.g., POINTQ) the queries that will be used in our experimental evaluation.

5.2.3 Problem Formulation

Each of the queries discussed above requires access to different intermediates from an ML-based workflow, e.g., predictions or hidden representations. For a given intermediate, there are two ways of computing it: (a) either we can re-run the workflow up to the particular intermediate (denoted `RERUN`) or (b) we can read the intermediate that has previously been materialized to disk (denoted `READ`). For instance, the Net-dissect implementation from [13] re-runs the full model any time an analysis is to be performed. While this solution may suffice when computing intermediate results for a small number of examples, running the model over tens of thousands of examples is slow (e.g., up to two orders of magnitude slower than reading as shown in Chap. 5.8) and wastes computation. In contrast, systems like ActiVis [70] and VizML [26] store intermediates to disk and read them to answer queries. While materializing intermediates is essential for providing interactive query times, this can come at a large storage cost. As mentioned before, storing intermediates for ten epochs of the `VGG16` network on `CIFAR10` takes about 350GB (gzip compressed), a storage cost most developers are unwilling to pay. Similarly, storing fifty traditional ML workflows with 9 — 19 stages takes 67 GB of storage (gzip compressed). Thus, the strategies of `RERUN` and `READ` are optimal for some intermediates while they may be expensive (with respect to time or storage) for others. **In this work, we seek to address the question of speeding up diagnostic queries by intelligently choosing when to re-run a ML-based workflow vs. when to (store and) read an intermediate and in turn minimize the cost of storing intermediates.**

Query Category	Specific instantiation	Intermediates Queried
	Queries Using Intermediates	
Few Columns, Few Rows (FCFR)	(POINTQ) Find the activation map for neuron-35 in layer-4 for image-345.png [139]	X, I
	(POINTQ) Find average lot size feature for for the Home-135 in P_{Oct31}	X, I
	(TOPK) Find the top-10 images that produce the highest activations for Neuron-35 in layer-13 [139]	X, I
	(TOPK) Find prediction error on the 10 homes that were most recently built	X, I
	Get the predicted price error for Home-150 [8]	X, P
	Get accuracy of P_{Oct31} on the top-50 most expensive homes in LA [70]	Y, X, P
Few Columns, Many Rows (FCMR)	(COL_DIFF) Compare model performance for P_{Oct31} and P_{Nov1} grouped by type of house [85, 88]	X, Y, P
	(COL_DIFF) Find the examples whose predictions differed between `CIFAR10_CNN` and `CIFAR10_VGG16` [88]	X, $Y_{\texttt{cnn}}$, $Y_{\texttt{vgg}}$
	(COL_DIST) Plot the error rates for all homes [26]	X, I, Y, P
	(COL_DIST) Plot the confidence score for all images predicted as cats [26]	X, I, Y, P
	Find number of images that were predicted as a frog but were in fact a ship [26]	Y, P
	Compute the confusion matrix for the training dataset [8]	$Y_{\texttt{train}}$, $P_{\texttt{train}}$
Many Columns, Few Rows (MCFR)	(KNN) Find performance of `CIFAR10_CNN` for images similar to image-51 [8]	X, $x_{\text{img-51}}$, Y, P
	(KNN) Find predictions for the 10 homes most similar to Home-50	X, x_o, Y, P
	(ROW_DIFF) Compare features for Home-50 and Home-55 that are known to be in the same neighborhood but have very different prices [70]	I, Y
	(ROW_DIFF) Compare the activations of neurons in layer-6 between an adversarial image and it's equivalent non-adversarial image	I, Y
	Determine whether this test point is an adversarial example [48]	X, x_{test}, Y, y_{test}
	Find training examples that contributed to the prediction of this test example [76]	X, I, x_{test}, i_{test}

Table 5.1: A Categorization of Diagnostic Queries. Last column, X=input, Y=target, I=intermediate dataset, P=predictions.

Query Category (Contd.)	Specific instantiation (Contd.)	Intermediates Queried (Contd.)
	Queries Using Intermediates	
Many Columns, Many Rows (MCMR)	(SVCCA) Compute similarity between the logits of class ship and the representation learned by the last convolutional layer [104]	I
	(SVCCA) Find the features from interm-8 most correlated with the residual errors of P_{Oct31}	X, Y, P
	(VIS) Plot the average activations for all neurons in layer-5 across all classes [70]	I
	(VIS) Plot the average feature values for Victorian homes in Boston vs. Victorian homes in Seattle	
	Compare the representations learned in layer-5 by AlexNet and by VGG16 in Layer-8 [104]	I_{AlexNet}, I_{VGG}
	Find correlation between the activation of each neuron and pixels corresponding to concept lamps [13]	X, I
	Queries Not Using Intermediates	
Gradient-based	Find the salient pixels in Image-250 [115, 111, 145]	
Feature importance methods	Find importance of pixel-50 in this model [106, 114]	
Perturbing examples	Find the minimal change that must be made to mispredict Image-51 [52, 99]	
Training New Models	Find a smaller model that performs similarly to a larger model [62, 76]	

Table 5.2: (Contd.) A Categorization of Diagnostic Queries. Last column, X=input, Y=target, I=intermediate dataset, P=predictions.

5.3 MISTIQUE Overview

In this section, we give a high-level overview of the system architecture and how it can be used to run diagnostic queries.

5.3.1 Architecture

The system architecture for MISTIQUE is shown in Fig. 5-5. MISTIQUE consists of three primary components: the PipelineExecutor, the DataStore, and the ChunkReader. These three components are tied together by a central repository called the MetadataDB that is used to track metadata about intermediates and workflows. The PipelineExecutor is responsible for running ML models and workflows in logging mode. This means that the executor runs the workflow forward, finds all interme-

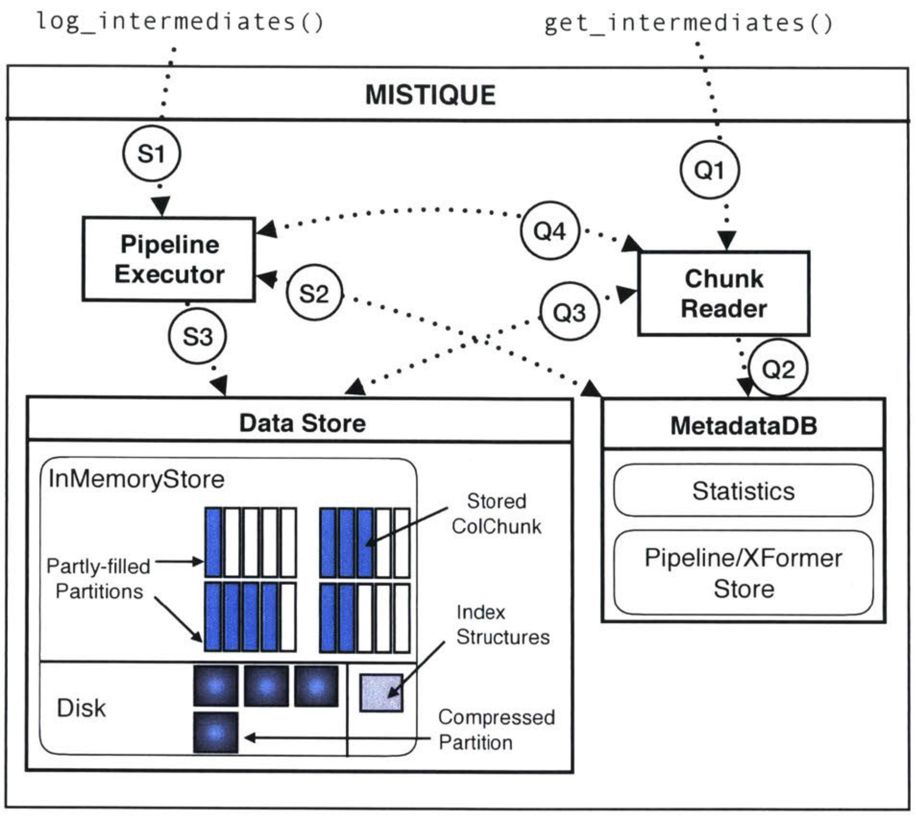


Figure 5-5: MISTIQUE Architecture with data flow during storage (S1-3) and querying (Q1-4)

diates produced by the workflow and registers information about the workflow and intermediates in MetadataDB. The PipelineExecutor does not make decisions about data storage or placement; that falls under the purview of the DataStore. Along with logging intermediates, the PipelineExecutor is also responsible for storing operators trained in a workflow (e.g., a trained XGBoost model) so that the operator may be re-run in the future without retraining.

Intermediates produced by the PipelineExecutor are passed on to the DataStore (Sec. 5.4) for decisions about whether and how to store the intermediate. The DataStore is made up of an InMemoryStore and a persistent store (on-disk in our implementation). MISTIQUE adopts a column-oriented scheme (much like [122, 7]) to store intermediates. Specifically, MISTIQUE represents each intermediate (including the source data and the final predictions) as a *DataFrame*[2] that is divided horizontally

[2]We choose to call it a dataframe because of the familiarity of the concept and not due to any

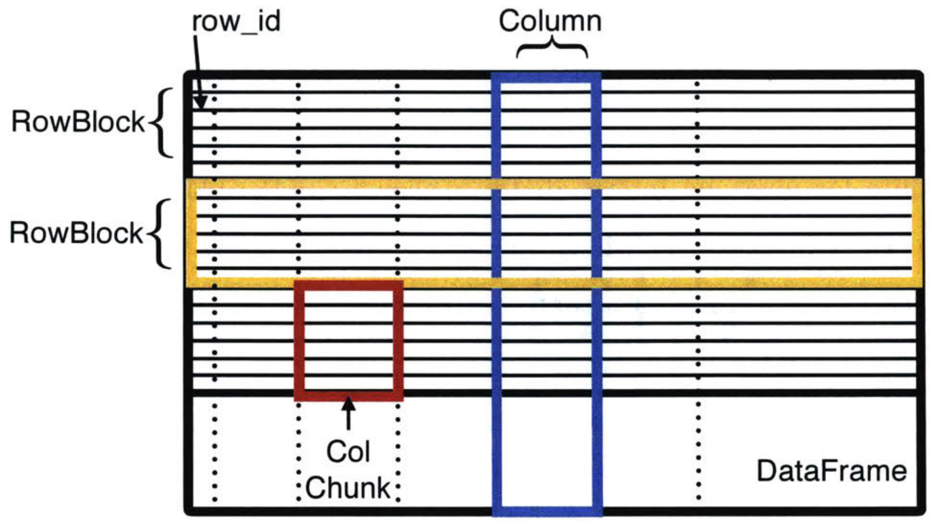


Figure 5-6: MISTIQUE Data Model

into a set of *RowBlocks*. Every row is associated with a unique `row_id` that is preserved across all intermediates and is used as a primary index. A DataFrame is also associated with a set of *Columns* that make up the DataFrame. The part of a Column that falls into a particular RowBlock is called a *ColumnChunk*. The data model is shown in Fig. 5-6.

The unit of data storage in the DataStore is a Partition. A Partition is a collection of ColumnChunks from *one or more* DataFrames that are to be stored together. The InMemoryStore serves as a bufferpool and keeps a number of (uncompressed) Partitions in memory. When a Partition is evicted from the InMemoryStore, the Partition is compressed and written out to disk. As described in subsequent sections, storage decisions in MISTIQUE are made at the level of ColumnChunks, giving the system fine-grained control over data placement. The DataStore also stores any indexes built on the data (by default, a primary index on `row_id`). The process of storing intermediates is indicated by S1—S3 in Fig. 5-5: the request to log intermediates is sent to the PipelineExecutor which sends each intermediate to the DataStore. The DataStore in turn queries the MetadataDB to determine whether the intermediate (or some columns) should be stored and, if so, stores the data using optimizations described below.

parallels with R, pandas or Spark dataframes. They can equally be considered as relational tables.

The final component of MISTIQUE is the ChunkReader (Sec. 5.6) which is responsible for servicing query requests. Query execution in MISTIQUE may involve fetching data from the DataStore or re-running the workflow to re-create intermediates. The procedure for querying intermediates is shown as Q1—Q4 in Fig. 5-5: the query is sent to the ChunkReader which queries the MetadataDB to determine whether to re-run the model or read data that was previously stored. Depending on the response, the ChunkReader either invokes the PipelineExecutor or queries the DataStore. The decision between these two alternatives is made by the cost model (Sec. 5.5). Regardless of how the data is obtained, a query to MISTIQUE produces a numpy array[3] that can be used as input to analytic functions.

5.3.2 Usage Example

As mentioned in the Introduction (Chap. 1), the current implementation of MISTIQUE can be used to log intermediates from DNNs defined using the Keras library (see Background in Chap. 2) on top of Tensorflow, or TRAD workflows using scikit-learn.

To log intermediates from Keras and Tensorflow, the ML developer must only provide MISTIQUE with the path to the model checkpoint and as well as an input loading function. Calling `log_intermediates(checkpoint, input_func)` causes MISTIQUE to run the DNN model forward by calling the input function and logging intermediates for every model layer.

For scikit-learn, we have defined a YAML specification (modeled after Apache Airflow[4])[5] that is used to express scikit-learn pipelines in a standard format. We wrapped a number of common scikit-learn functions for use in the YAML specification (e.g. models like XGBoost, LinearRegression as well as preprocessing steps like Scaling and LabelIndexing). `log_intermediates(yaml_file, input_func)` similarly logs all the pipeline intermediates. Once intermediates have been logged in

[3]http://www.numpy.org/

[4]https://airflow.incubator.apache.org

[5]This YAML specification is different from the Light API in ModelDB. This specification has all the information required to instantiate relevant scikit-learn objects and run each step of the ML workflow. The YAML specification for the ModelDB Light API has minimal requirements and, in general, is insufficient to instantiate a scikit-learn pipeline.

MISTIQUE, the ML developer can use a set of query APIs to access the data in MISTIQUE.

The key query API exposed by MISTIQUE is `get_intermediates([keys])`. This API can be used to retrieve any column, of any intermediate, belonging to any model that has been logged with MISTIQUE. Keys take the form of `project.model.intermediate.column`. The API returns a numpy array with the required columns as well as the `row_id` column. For ease of use, MISTIQUE provides implementations of common analytic functions that can be applied on top of the numpy array result (although this is not the focus of our contribution). Since MISTIQUE returns numpy arrays, it is also easy to add other analytic functions.

5.4 Data Store

Once MISTIQUE has used the cost model (Sec. 5.5) to determine that an intermediate is to be stored, the DataStore is responsbile for determining how to most efficiently store the intermediate. The naïve strategy when storing intermediates is to fully store every intermediate from any workflow that is run. While simple, this strategy requires a great deal of storage, e.g., it logs 350 GB of compressed data across ten epochs of the moderately sized `VGG16` model and requires 67 GB to store fifty traditional ML pipelines with <20 stages. Therefore, we explore different storage strategies to reduce footprint of intermediates without compromising query time or accuracy. Specifically, we propose three key optimizations: (a) for DNNs, we propose multiple quantization and summarization schemes to reduce the size of intermediates; (b) for DNNs as well as traditional workflows, we perform exact and approximate de-duplication between ColumnChunks within and across models; (c) we perform adaptive materialization of intermediates by trading off the increased storage cost with reduction in query time. We now expand upon each of these optimizations.

5.4.1 Quantization and Summarization

A key insight from diagnostic techniques proposed for DNNs is that ML developers are much more interested in *relative values* of neuron activations than they are in the exact values. For example, the visualizations in ActiVis are used to compare activations of neurons in different classes (see Fig. 5-1). Since the visualization cannot display >256 shades of the same color, at most 256 distinct activation values may be shown in the visualization. Along similar lines, the Netdissect technique only examines neuron activations in the top 99.5th percentile, i.e., the technique only needs to know if the activation is "very high" or "not very high"—regardless of the actual activation value. This indicates that we can *quantize* or discretize neuron activations into a much smaller number of values without affecting the accuracy of many diagnostic techniques. Previous work on model compression and model storage [56, 86] explored the use of quantization of *model weights* to reduce model size for inference as well as storage. In this work, we propose to extend those techniques to aggressively quantize neuron activations. Note that since these activations are only used for diagnostic purposes, we can perform drastic quantization. MISTIQUE supports three quantization schemes:

- *Lower precision float representation* (`LP_QT`): Storing a double precision float value as a single precision (float32) or half point precision (float16) value can lead respectively to a 2X and 4X reduction in storage with no effect on diagnostic accuracy.
- *k-bit quantization* (`KBIT_QT`): Since many diagnostic techniques are based on relative activations, we can reduce storage costs by representing values using quantiles (similar strategies are used to quantize weights in [86]). Given the maximum number of bits b to be allocated for storing each activation, we can compute 2^b bins using quantiles and assign each value to the corresponding bin. Quantization of activation values from o to b bits reduces storage by a factor of $\frac{o}{b}$.
- *Threshold-based quantization* (`THRESHOLD_QT`): To support queries such as Netdissect that use an explicit activation threshold, we can directly encode the data

as 0-1s depending on whether values are above or below the threshold. Once a threshold has been picked, however, we cannot encode the data with respect to another threshold. This scheme reduces storage cost by o, the number of bits used by the original values.

The quantizations above reduce the storage required for each value, but do not reduce the *number* of values stored. This is particularly important in CNNs where the size of an activation map can significantly increase storage costs. Therefore, to reduce the number of activations, we support summarization via *pooling* (similar to the max-pooling operator in DNNs). In pooling quantization (POOL_QT), we apply an aggregation operation such as average (default) or max to adjacent cells in an activation map to obtain a lower resolution representation of the map. Assuming a 2-D activation map per channel (as in CNNs), pooling quantization reduces storage by $\frac{S^2}{\sigma^2}$ where we assume that the size of an activation map is SxS and size of the aggregation window is σxσ. We support two levels of pooling quantization: σ=2 (default, also denoted pool(2)) and σ=S, denoted pool(S), e.g., pool(32) for CIFAR10. $\sigma = S$ is the most extreme version of pool-based quantization where we compute a single average value to represent each activation map.

Implementation.

Both KBIT_QT and THRESHOLD_QT require the system to first collect samples of activations to build a distribution and subsequently use this distribution to perform quantization. By default, for KBIT_QT, we set $k = 8$, which means that we compute $2^8 = 256$ quantiles for the activation distribution and assign each activation value to the appropriate quantile. 8BIT_QT reduces storage by 4X when raw activations are single precision floats and 8X for double precision. Note that when fetching an 8BIT_QT intermediate, we must also pay a *reconstruction cost* to go from the quantized values (0 - 255) to floating points. For POOL_QT, we conservatively use $\sigma = 2$. However, the user can choose to set a more aggressive pooling level depending on the application. In Sec. 5.8, we study the trade-offs involved in setting different σ values.

5.4.2 Exact and Approximate De-duplication

This optimization is based on two observations. First, intermediates in traditional ML workflows often have many identical columns. For example, in the TRAD workflows of Fig. 5-3, consecutive intermediates often only differ in a handful of features (e.g., RelativeSize between interm9 and interm10 in P_{Nov1}) and workflows share many stages (e.g., in an extreme case like P_{Oct30} and P_{Oct31}, all intermediates are identical except for `pred`). Second, TRAD and DNN intermediates often have similar columns (e.g., predictions from multiple models for the same task such as P_{Oct30}, P_{Oct31}; intermediates from different epochs for the same DNN; and quantized versions of intermediates). We can leverage these insights to avoid storing redundant data and to compress similar data to obtain higher compression ratios.

Implementation

Implementing de-duplication (exact and approximate) requires two steps: first, we must identify identical or similar ColumnChunks, and second, we must compress these ColumnChunks when writing to storage (MISTIQUE does not currently compress ColumnChunks when in memory). We can identify identical ColumnChunks simply by computing the hash of the ColumnChunks. If an identical ColumnChunk has previously been stored, then the current ColumnChunk can be skipped. For detecting similar columns, we use the MinHash. For every new ColumnChunk, the DataStore computes the MinHash for the ColumnChunk (after discretizing the values) and queries the LSH index for Partitions with Jaccard similarity above a threshold τ. If an existing ColumnChunk is found to have similarity above τ, the new ColumnChunk is stored in the same partition as the existing ColumnChunk. Otherwise, a new Partition is created.

For DNNs, we perform two simplifications: (a) we only perform exact de-duplication because DNN columns seldom have similar values; (b) we assign columns from the same intermediate to the same Partition because DNN intermediates have many more columns than TRAD intermediates and Partition assignments based on data-similarity disperse columns over a large number of Partitions.

The previous procedure performs a rough clustering of ColumnChunks based on similarity and assigns ColumnChunks to Partitions. When a Partition is to be written to disk (e.g., because it is full or gets evicted from the InMemoryStore), the Partition is compressed and written out. MISTIQUE supports a variety of off-the-shelf compression schemes including gzip, HDF5, and Parquet.

5.4.3 Adaptive Materialization

Adaptive materialization is motivated by the observation that traditional ML workflows and DNN models are often many stages long but not all intermediates or columns are accessed with equal frequency. Some intermediates (e.g., predictions of a model or image embeddings from the last convolutional block) may be accessed very often while others (e.g., activations from the first convolutional layer) may be accessed less frequently. Therefore, we trade-off the increase in storage cost due to materialization against the resulting speedup in query time. We capture this trade-off in parameter γ formally described in our storage cost model (Sec. 5.5.2). If γ for an intermediate is larger than a threshold, the intermediate is materialized. An intermediate that is queried a large number of times has a large γ value and is more likely to be materialized. Similarly, an intermediate with a small storage cost leads to a larger γ and is more likely to be materialized. The full algorithm for logging intermediates is shown in Alg. 3.

5.5 Cost Model

In order to make the decision about (a) whether to store an intermediate, and (b) whether to execute a query by re-running a workflow or reading an intermediate, we respectively develop a storage cost model and a query cost model. We begin with the query cost model which provides the building blocks for the storage cost model.

Algorithm 3 Storing intermediates

```
1: procedure STORE_INTERMEDIATE(row_block, $\gamma_{min}$)
2:     for col_chunk ∈ row_block.columns do
3:         stats ← get_stats(col_chunk)
4:         if $\gamma < \gamma_{min}$ then // don't store
5:             update_stats(stats)
6:             return
7:         col_chunk ← quantize(col_chunk)
8:         identical_col ← get_identical_cols(col)
9:         if identical_col == nil then
10:            partition ← get_closest_partition(stats, sim)
11:            partition.add(col_chunk)
12:            evicted_partition ← bufferpool.add(partition)
13:            compress_and_store_partition(evicted_partition)
14:        else
15:            update(identical_col)
```

5.5.1 Query Cost Model

The total time to execute a diagnostic technique (t_{diag}) can be computed as the sum of the time to fetch the required intermediates (t_{fetch}) and the time to perform computation on the data (t_{compute}).

$$t_{\text{diag}} = t_{\text{fetch}} + t_{\text{compute}} \tag{5.1}$$

The time to fetch the data, in turn, is equal to either the time to re-run the workflow or to read a previously stored intermediate. Since the compute time is the same in both cases, we only model t_{fetch}. Suppose we want to fetch the intermediate at stage i in workflow M (e.g., i-th layer in a DNN or stage-i in a traditional ML workflow) and we seek to compute the intermediate for n_ex examples (where n_ex is between 1 and TOTAL_EXAMPLES). Then, if $t_{\text{i,re-run}}$ denotes the time to re-run the workflow to intermediate i, we can compute this quantity as the sum over each stage s $\leq$ i of: (a) time to read the transformer for s ($t_{\text{read\_xformer}}$), (b) time to load the input for s

($t_{\text{read\_xformer\_input}}$), and (c) time to execute s ($t_{\text{exec\_xformer}}$).

$$\begin{aligned} t_{\text{i,re-run}} &= \Sigma_{s=0}^{i}\{t_{\text{read\_xformer}}(s) + t_{\text{read\_xformer\_input}}(s) \\ &\quad + t_{\text{exec\_xformer}}(s)\} \end{aligned} \tag{5.2}$$

For a DNN, we can rewrite the model as follows: (a) we usually load the entire model at once, so we can rewrite the first term as $t_{\text{model\_load}}$; (b) explicit input to the DNN is only provided at layer-0, so we read input once; and (c) since the model is usually used to predict on *batches* of examples, we factor batch size into the time to execute a stage. If $t_{\text{model\_load}}$ denotes the time to read the model, sizeof denotes the size of an object in bytes, bt_size denotes the batch size, ρ denotes the read speed of storage, and $t_{\text{fwd}}(\text{s}, \text{bt\_size})$ denotes the time to run a single batch of examples through the DNN up to layer s, the cost model for re-running DNNs can be written as in Eq. 5.3.

$$\begin{aligned} t_{\text{i,re-run,NN}} &= t_{\text{model\_load}} + \frac{\text{n\_ex} \cdot \text{sizeof(ex)}}{\rho} \\ &\quad + \frac{\text{n\_ex}}{\text{bt\_size}} \cdot \Sigma_{s=0}^{i} t_{\text{fwd}}(s, \text{bt\_size}) \end{aligned} \tag{5.3}$$

$$t_{\text{i,read}} = \frac{\text{n\_ex} \cdot \text{sizeof(ex)}}{\rho_d} \tag{5.4}$$

Instead of re-running a model to obtain an intermediate, we can also read a previously-stored intermediate. The time to read an intermediate (denoted t_{read}) is simply proportional to the size of the intermediate, i.e., the number of examples multiplied by the size of one example. We assume that the size of the example accounts for the precision of the value (e.g., float16, uint8). We fold the time to read, decompress, and reconstruct the data into the constant ρ_d.

If $t_{\text{re-run}} \geq t_{\text{read}}$, we run the query by reading a previously stored intermediate.

5.5.2 Storage Cost Model

Using the query cost model, we can also determine when to store (i.e., materialize) an intermediate. The decision to materialize can be made at the level of an entire intermediate (i) or a particular column in an intermediate (i, c). In either case, we compute $t_{\text{re-run}}$ and t_{read} as above by setting n_ex to TOTAL_EXAMPLES. If $t_{\text{re-run}} \geq t_{\text{read}}$, we can trade-off the speedup from storing the intermediate against the additional cost of storing the intermediate.

$$\gamma = \frac{(t_{\text{i,re-run}} - t_{\text{i,read}}) \cdot \text{n\_query(i)}}{\text{S(i)}} \tag{5.5}$$

This trade-off is captured in γ shown in Eq. 5.5 where S(i) is the (amortized) storage required for intermediate i, n_query(i) is the number of queries made to intermediate i that *gets updated with each query to the system.* γ is computed every time the DataStore has to make a decision on whether to re-run a model or read its intermediates. The units of γ *are seconds per GB* and it captures the amount of query time saved per GB of data stored. For example, a γ of 1000 sec/GB indicates that the ML developer is willing to use 1GB of storage in exchange for a total saving of 1000s in query time. Note that the numerator of Eq. 5.5 increases with the number of times intermediate i is queried. S(i), in turn, is affected by storage of other intermediates; intermediates with similar data will lead to lower S(i) because they are compressed together.

Note that the only unknown quantities in Eqs. 5.3 and 5.4 are the time to load the model or workflow and time to execute each stage of the workflow. So long as MISTIQUE has executed a workflow at least once, it will have measurements for both these quantities and can estimate the trade-off between re-running vs. reading. If the workflow has never been executed before, MISTIQUE must execute the workflow in order to collect statistics, irrespective of whether the intermediates will eventually be stored.

5.6 Fetching data from MISTIQUE

Diagnostic techniques like those discussed in Sec. 5.2.2 are executed by first fetching the data from MISTIQUE and then running analyses on it. We currently have a simple query execution model inside MISTIQUE. The ChunkReader is responsible for fetching intermediates either by reading them from the DataStore or re-running the workflow and returning them to the user. When a query for an intermediate arrives, the ChunkReader first queries the MetadataDB to check if the intermediate has been stored and if so, verifies that the time to read the intermediate is less than the time to re-run the workflow (as computed by the cost model in Sec. 5.5). If the time to read is smaller, the ChunkReader queries the DataStore for the intermediate. The DataStore in turn identifies the Partitions containing ColumnChunks for this intermediate. For particular kinds of queries (e.g., fetch results by row_id), MISTIQUE can use the primary index to speed up retrieval of relevant Partitions. In the future, we can incorporate specialized indexes for particular types of queries (e.g., nearest neighbor index). Once the relevant Partitions have been read, the ColumnChunks for this intermediate are stitched together and returned. On the other hand, if it is faster to re-run the workflow, the ChunkReader invokes the PipelineExecutor to obtain the intermediate. The PipelineExecutor in turn executes previously stored operators for this model or workflow. In either case, the result of the query to MISTIQUE is a numpy array which is then used as input for further analysis (e.g., SVCCA, visualization). Pseudocode for querying MISTIQUE is shown in Alg. 4.

5.7 Experimental Setup

In the previous sections, we described the data model, architecture and optimizations implemented in MISTIQUE. In this section, we present an experimental evaluation of our system on multiple real-world models and analytical techniques. We begin with a description of the experiment setup, including the generation of ML-based workflows, and then describe our results.

5.7.1 Workflows

We evaluated our storage techniques on different traditional ML workflows from scikit-learn and on deep neural network models built in Tensorflow.

Algorithm 4 Reading Data from ModelDB

```
procedure GET_DATA(columns)
    dfs ← columns.parent_dataframe()
    tmp1 ← []
    for df ∈ dfs do
        tmp2 ← []
        for row_block ∈ df.row_blocks() do
            tmp3 ← []
            for column ∈ columns do
                if column.is_materialized() then
                    partition ← column.read_partition()
                    tmp.append(partition[column])
                else
                    rerun_workflow(column, row_block)
            row_block_data ← h_concat(tmp3)
            tmp2.append(row_block_data)
        df_data = v_concat(tmp2)
        tmp1.append(df_data)
    res ← h_concat(tmp1)
    return res
```

Traditional ML Workflows (TRAD)

To replicate a real-world machine learning scenario for traditional models, we use the dataset and task from the Zillow Home Value Prediction competition discussed in Chap. 2. The goal of this competition is to use data about homes (e.g., number of rooms, average square footage) to build a model that can predict the *error* of Zillow's in-house price prediction model.

For this task, we are given three `csv` files: (i) `properties` containing attributes of homes in the dataset; (ii) `training` data listing the property id, date of the property sale, and the error between the Zillow price estimate and the actual sale price; (iii) `test` data containing the property id and three dates of potential property sale. To obtain ML-based workflows for this task, we took ten scikit-learn scripts uploaded by Kagglers for this task and turned them into workflows in MISTIQUE. Table 5.3 shows all the workflow template used in the Zillow workload. The evaluation workflows contained between 9 - 19 different stages including data preprocessing, feature engineering, feature selection, and training as well as prediction using the model. Since Kaggle competitiors often submit scripts with the best hyperparameter settings, we also defined 5 variations for each workflow by varying the hyperparameter settings. While we could have generated a much larger number of variations for each pipeline, we believe that the resulting set of 50 pipelines are sufficient to illustrate the advantages of MISTIQUE.

DNN Models (DNN)

To illustrate the efficacy of our techniques on deep neural networks, we use with the CIFAR10 image classification task. CIFAR10 contains 50K training images from 10 classes where each image has dimensions 64x64x3. We evaluate on two models trained on CIFAR10: the VGG16 model fine-tuned on CIFAR10, denoted as CIFAR10_VGG16 (the original model has been trained on the IMAGENET [108] dataset) and a well-accepted, simple CNN model trained from scratch, denoted as CIFAR10_CNN [6]. The original VGG16 model consists of 13 convolutional layers, 5 pooling layers, and 3 fully connected layers. During fine-tuning of VGG16, we take the first 13 pre-trained convolutional layers, freeze their weights, replace the original fully connected layers with two smaller, fully connected layers (because CIFAR10 does not require these layers to be wide) and train these layers. In contrast, CIFAR10_CNN has 4 convolutional layers and 2 fully connected layers, and is trained from scratch. We checkpoint model weights after every 10% of the total number of epochs (i.e., 10 checkpoints each).

Choice of ML-based workflows. We chose the Zillow Home Value Prediction Challenge as our task for TRAD models because many hundreds of Kagglers have participated in this competition and thus we can obtain real-world workflows that have been submitted to the competition. Moreover, operators used in these Kaggle workflows are also used in many other ML-based workflows that model tabular data. Thus, while these workflows don't capture all possible ML-based workflows, they capture key operators and enable us to evaluate our techniques on workflows with

[6] https://github.com/keras-team/keras/blob/master/examples/cifar10_cnn.py

ID	Pipeline Template	Hyperparameters
P1	ReadCSV (3) → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainLightGBM → Predict (2)	learning_rate,sub_feature, min_data
P2	ReadCSV (3) → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainXGBoost → Predict (2)	eta,lambda,alpha,max_depth
P3	ReadCSV (3) → OneHotEncoding → FillNA (2) → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainEasticNet → Predict (2)	l1_ratio,tol
P4	ReadCSV (3) → Avg → OneHotEncoding → FillNA (2) → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainEasticNet → Predict (2)	l1_ratio,tol, normalize
P5	ReadCSV (3) → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainXGBoost,TrainElasticNet → Predict (4) → CombinePredictions (2)	eta,lambda,alpha,max_depth, xgb_weight, lgbm_weight
P6	ReadCSV (3) → Avg → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainLightGBM → Predict (2)	eta,lambda,alpha,max_depth, bagging_fraction
P7	ReadCSV (3) → Avg → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainXGBoost → Predict (2)	eta,lambda,alpha,max_depth, bagging_fraction
P8	ReadCSV (3) → Avg → GetConstructionRecency → OneHotEncoding → FillNA (2) → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainEasticNet → Predict (2)	l1_ratio,tol, normalize
P9	ReadCSV (3) → Avg → GetConstructionRecency → OneHotEncoding → FillNA (2) → ComputeNeighborhood → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainEasticNet → Predict (2)	ComputeNeighborhood_params, l1_ratio,tol, normalize
P10	ReadCSV (3) → Avg → GetConstructionRecency → OneHotEncoding → FillNA (2) → ComputeNeighborhood → IsResidential → Join (2) → SelectColumn → DropColumns (2) → TrainTestSplit → TrainEasticNet → Predict (2)	IsResidential_params, l1_ratio,tol, normalize

Table 5.3: Workflow Templates for Zillow workload. The numbers in params indicate the number of times a transformation is applied (typically once on the training set and then again on test set)

these operators.

On the DNN-side, VGG16 is one of the most popular models used for image classification. It also represents a moderately sized network that is ideal for evaluation in a research setting. Moreover, the key building blocks of convolution, pooling, and fully connected layers are the hallmarks of image networks and therefore evaluating MISTIQUE on networks with these layers provides a view of its applicability to other image networks. Note that we have not tested our system on networks such as ResNets that have skip connections and dropout.

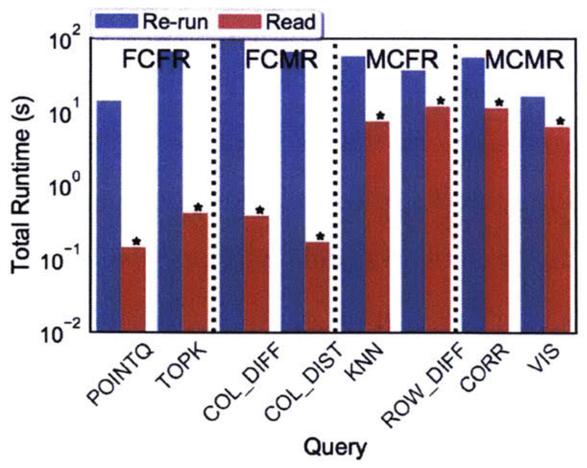
MISTIQUE has been entirely implemented in Python. All experiments were run on an Intel Core i7-6900K machine running at 3.20 Ghz. 32 core machine (16 CPUs) with 64 GB RAM, and 2 GM200 GeForce GTX Titan X GPU. GPU support was enabled when running DNN models.

5.8 Experimental Results

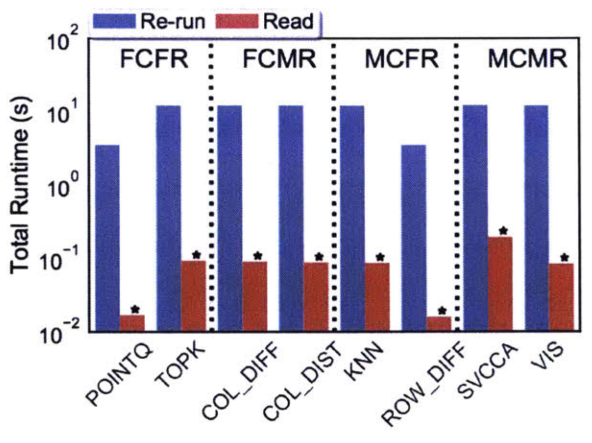
The overall gains offered by a system like MISTIQUE will vary depending on the details of the ML workflows that are logged with MISTIQUE. The set of experiments described next evaluate MISTIQUE in a controlled setting with a limited number and type of workflows. As a result, the performance tradeoffs may change in other settings and we discuss the generalizability of the results in Chap. 5.9.

Our goals in the experimental evaluation are to answer the following key questions:

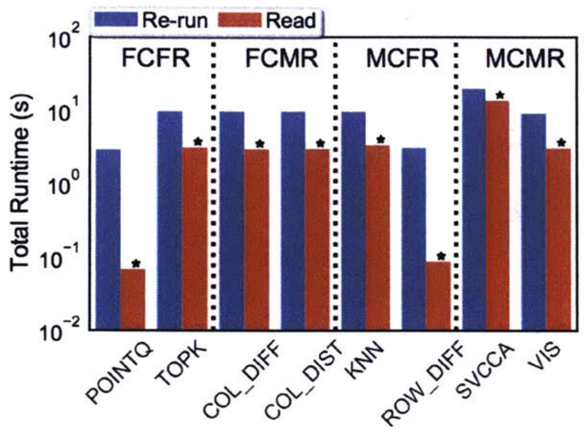
- What is the speedup in execution time from using MISTIQUE to run diagnostic queries?
- What overall storage gains can we achieve by using MISTIQUE and our proposed optimizations?
- Does our cost model accurately capture the re-run vs. read trade-off?
- For DNNs, how do the proposed quantization schemes affect accuracy of diagnostic techniques?



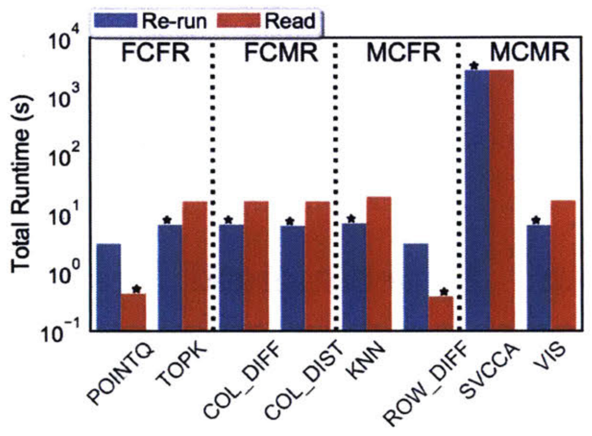
(a) Zillow pipelines



(b) CIFAR10_VGG16 (Layer 21)



(c) CIFAR10_VGG16 (Layer 11)



(d) CIFAR10_VGG16 (Layer 1)

Figure 5-7: End-to-end query runtimes. Asterisk indicates strategy picked by cost model

- What is the overhead of using MISTIQUE vs. baselines?
- What is the impact of adaptive materialization on storage and query time?

5.8.1 End-to-End Query Execution Times

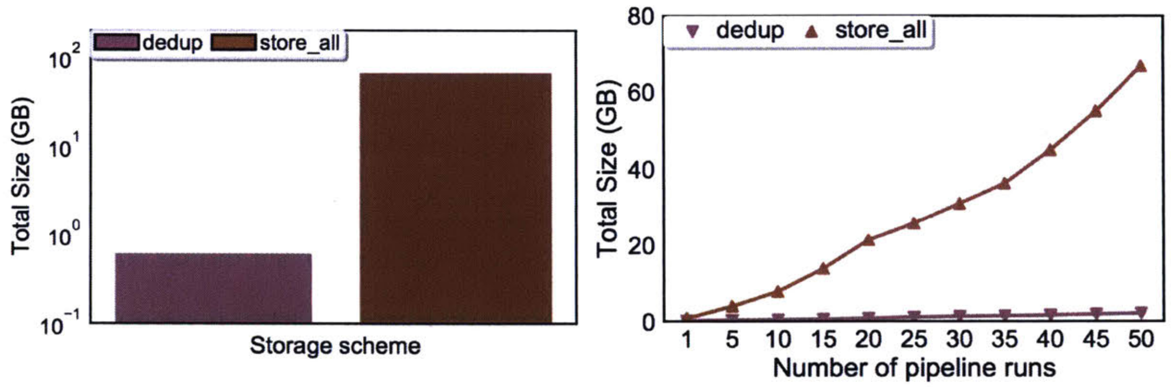
In this set of experiments, we evaluate the end-to-end execution times for a representative set of diagnostic techniques from Table. 5.1. For each query category (FCFR, FCMR, MCFR, MCMR), we evaluate on two queries for the TRAD and DNN workflows. For the DNN queries, we run the same query at multiple layers to show how the trade-off between reading and re-running changes across layers. In this experimental setup, when re-running DNN models, we pre-fetch the entire input into memory. Batch size for the DNN queries was set to 1000 and RowBlocks in MISTIQUE were set to be 1K rows. Fig. 5-7 shows results of this experiment with an *asterisk indicating the strategy chosen by the cost model.* Note the log scale on the y-axes.

For TRAD models, we find that running a query by reading an intermediate is always faster than re-running the workflow. For example, in Fig. 5-7a, we see a speedup of between 2.5X — 390X. In contrast, for DNN models (here CIFAR10_VGG16), the decision between whether to re-run or read depends on the model layer and number of examples fetched by the query. For queries on Layer21 (Fig.5-7b), the last layer, reading intermediates is 60X — 210X faster that than re-running the model. For Layer11 (Fig.5-7c), we see that reading the intermediate is again faster by 2X — 42X. In contrast, we find that at Layer1 (Fig.5-7d), *re-running* the model can be up to 2.5X faster for some queries. This is because Layer1 is very close to the input and is the largest layer by size (see Table. 5.4). Since we pre-fetched input data for DNN queries, we expect to observe even larger speedups when the input must be read from disk. For all queries we evaluated, with the exception of SVCCA, compute time is a small fraction of the total query time. For SVCCA on Layer1, in contrast, compute time accounts for about 99% of the total time.

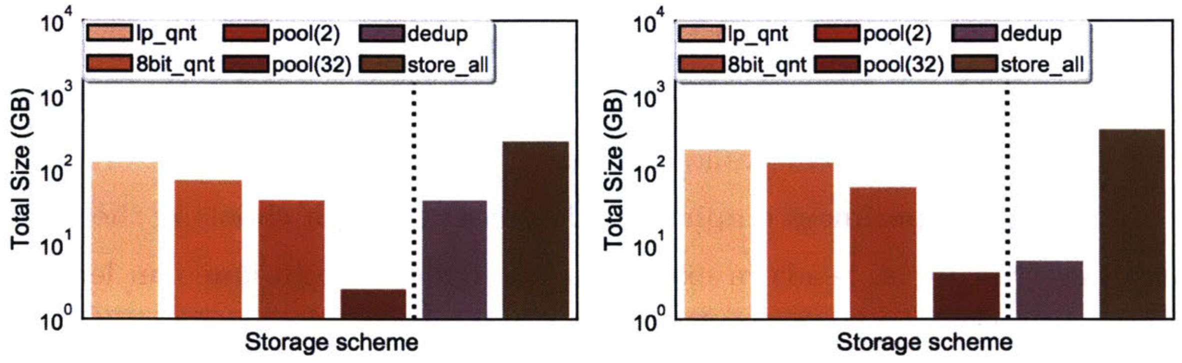
The above experiment empirically demonstrates that choosing the right strategy (re-run vs. read) in executing a diagnostic technique can lead to speedups of between 2X— 390X.

5.8.2 Intermediate Storage Cost

Next we examine the storage gains obtained by the different optimizations proposed in MISTIQUE. Fig. 5-8 shows the storage cost, i.e., number of bytes used, for Zillow, CIFAR10_CNN and CIFAR10_VGG16 models. For every set of models, we store intermediates from *all* stages. For TRAD, we evaluate the basic strategies of STORE_ALL and DEDUP. For DNN models, we also compare the storage cost when applying different quantization schemes: LP_QT, 8BIT_QT, and POOL_QT ($\sigma = 2, 32$) described in Sec. 5.4.



(a) `Zillow` workflows



(b) `CIFAR10_CNN` (left) and `CIFAR10_VGG16` (right)

Figure 5-8: Storage sizes for different strategies

Layer num	Name	Shape
0	input	(32, 32, 3)
1, 2	block1_conv1, block1_conv2	(32, 32, 64)
3	block1_pool	(16, 16, 64)
4, 5	block2_conv1, block2_conv2	(16, 16, 128)
6	block2_pool	(8, 8, 128)
7, 8, 9	block3_conv1	(8, 8, 256)
10	block3_pool	(4, 4, 256)
11, 12, 13	block4_conv1	(4, 4, 512)
14, 15, 16, 17	block4_pool	(2, 2, 512)
18	block5_pool	(1, 1, 512)
19	flatten	(512)
20	dense	(256)
21	logits	(10)

Table 5.4: Size of Layers in CIFAR10_VGG16

For Zillow (Fig. 5-8a, left), we find that the raw dataset is 168 MB compressed but STORE_ALL requires 67GB to store all the intermediates across 50 workflows. The 400X larger storage footprint indicates that the naïve STORE_ALL strategy cannot scale to large input data, long workflows or large number of models. In contrast, DEDUP, which applies approximate and exact de-duplication as discussed in Chap. 5.4.2, drastically reduces storage cost by 110X to 611 MB. On the right side of Fig. 5-8a, we see that the cumulative storage cost for Zillow increases linearly for STORE_ALL, while it stays relatively constant for the DEDUP strategy. This is because, for Zillow (and most TRAD workflows), many columns are shared between pipelines. Consequently, most of the storage cost is due to the first workflow whereas deltas are stored for the rest.

In Fig. 5-8b we show the cost in bytes of storing intermediates for CIFAR10_CNN and CIFAR10_VGG16. For both models, we store intermediates for ten epochs. The raw size of the input data (CIFAR10) is 170MB (compressed) in both cases. For CIFAR10_CNN we find that STORE_ALL with no quantization requires 242 GB, while STORE_ALL consumes 350 GB for CIFAR10_VGG16. The storage cost per DNN model is much larger than that of a traditional ML workflow. As a result, reducing storage footprint is even more essential for DNN models. For each model, we present storage sizes for LP_QT, 8BIT_QT and POOL_QT (σ=2,32). We apply DEDUP on top of the default scheme of POOL_QT (σ=2) to obtain the final size.

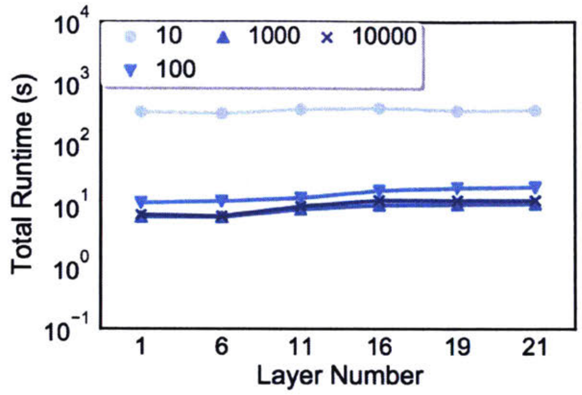
For CIFAR10_CNN , we see that LP_QT reduces storage from 242GB to 128 GB for and 8BIT_QT further reduces it to 72.4 GB. The biggest storage gains can be achieved by applying different levels of POOL_QT which can reduce storage to 39 GB for $\sigma = 2$ (6.2X reduction) and 2.53 GB for $\sigma = 32$ (95X reduction). Applying DEDUP does not produce significant gains because, unlike Zillow workflows, CIFAR10_CNN has few or no repeated columns. CIFAR10_VGG16 shows the same trends as that of CIFAR10_CNN except for the impact of DEDUP. Applying POOL_QT to CIFAR10_VGG16 reduces storage by 6X for $\sigma = 2$ (58 GB) and by 83X for $\sigma = 32$ (4.19 GB). As discussed in Sec. 5.7, CIFAR10_VGG16 is trained such that the bottom 13 convolutional layers of the network are frozen while only the top fully connected layers are trained. Thus, intermediates from the 13 layers are the same across all models and therefore applying DEDUP in addition to POOL_QT ($\sigma = 2$) reduces storage footprint by 60X to 5.997 GB.

Thus, MISTIQUE **can reduce storage footprint by up to 6X (DNN)—110X (TRAD), and upto 60X for fine-tuned models.**

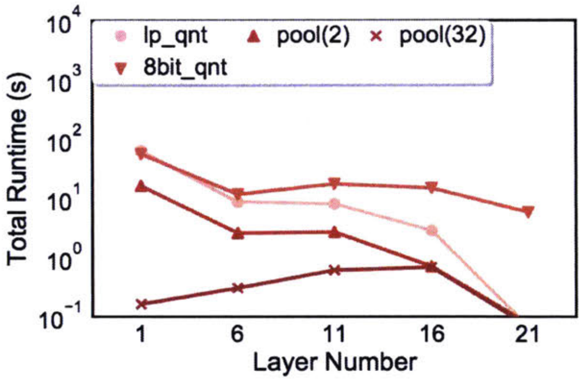
5.8.3 Validating the Cost Model

In Sec. 5.5, we proposed models to quantify the cost of re-running a model and the cost of reading an intermediate. In this section, we present experiments verifying our cost models and the resulting trade-off, focusing particularly on DNNs. For this evaluation, we used the CIFAR10_VGG16 model and stored intermediates for all layers to disk. Next, we ran an experiment where we fetched each intermediate either by reading the intermediate from disk or by re-running the model. When re-running the DNN model, we pre-fetched the entire input data into memory to avoid disk access. We repeated this process for different number of examples (n_ex in the cost model). The results are shown in Fig. 5-9 and 5-10.

Fig. 5-9a shows the time required to compute intermediates by re-running the model (n_ex=TOTAL_EXAMPLES) with different batch size settings. We see that

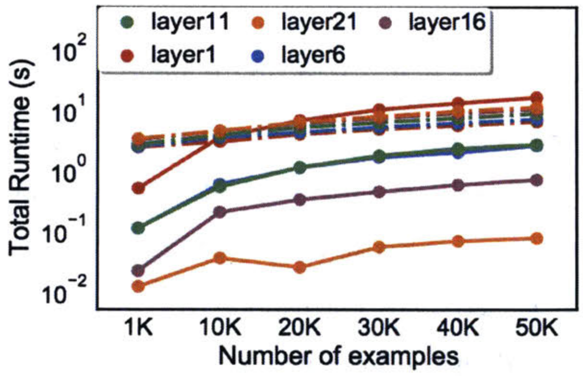


(a) Cost of re-running model

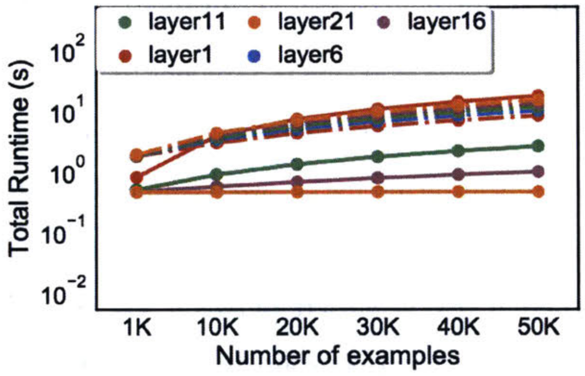


(b) Cost of reading intermediate

Figure 5-9: Verifying the cost model



(a) Empirical



(b) Predicted

Figure 5-10: Read (solid) vs. Exec (dashed) Trade-off

batch size (number of examples that are run through the model at a time) has a significant impact on execution time. This is because batch size affects the number of times the model is run forward. Computing intermediates for batch size=10 is 30X slower than for batch size=1000. Performance degrades slightly for batch size of 10000, whereas larger batchsizes overflow the GPU memory (11 GB). (We therefore use batch size of 1000 in all experiments.) We also find that the time to re-run increases proportionately to the layer number and we pay a fixed cost of 1.2s for model load. As shown in Fig. 5-10a, execution time also increases linearly with n_ex (i.e., the total number of examples that are run through the model).

Fig. 5-9b shows the time to read an intermediate from disk for different layers and quantization schemes. RowBlock size=1K rows and n_ex=TOTAL_EXAMPLES. As captured in Eq. 5.4, the time to read an intermediate depends on the number of examples in the intermediate and size of each example. We find that 8BIT_QT has the largest read time (due to high reconstruction cost), followed by LP_QT (but with 2X as much storage as 8BIT_QT), followed by pool(2), and finishing with pool(32). Although pool(32) produces the best query time, the drastic summarization makes it impossible to run certain queries on it (e.g., SVCCA). **pool(2) therefore presents a good trade-off with respect to query time and storage, and we use it as the default storage scheme in** MISTIQUE.

Next, we examine the impact of the cost difference between re-running and reading an intermediate when querying different layers of the CIFAR10_VGG16 model. When reading intermediates, we assume that the intermediates have been stored with the default pool(2) scheme. Fig. 5-10a shows the retrieval cost for five different layers when n_ex is varied between 1K — 50K. Dashed lines correspond to re-running the model whereas solid lines correspond to reading the intermediate. We find that (as captured in our cost model), the time to read or re-run intermediates scales linearly with the number of examples. Similar to the trend in end-to-end runtimes, reading intermediates is cheaper than re-running the model for all layers except Layer1 (>10K examples). Layer1 is anomalous because the intermediate is of large size (and therefore takes long to read) but is close to the input (and therefore is fast to re-

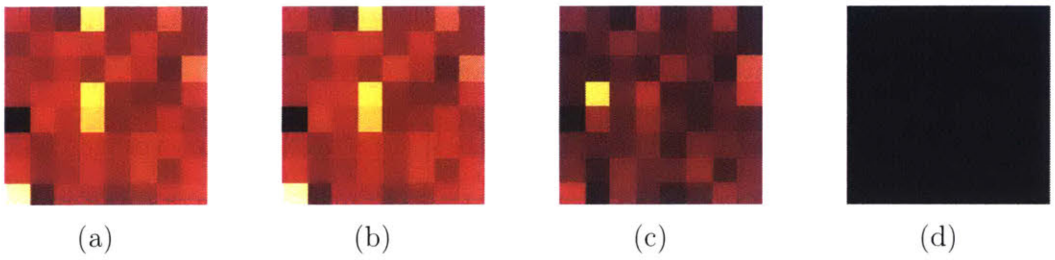


Figure 5-11: Visualizing average activations for different storage schemes: (a) full precision (float32), (b) LP_QT (float16), KBIT_QT ($k = 8$), POOL_QT ($\sigma = 32$) (all produce the same heatmap); (c) KBIT_QT ($k = 3$); (d) THRESHOLD_QT

run). Fig. 5-10b shows the same trade-off from Fig. 5-10a, except as predicted by our cost model. We see that the cost model accurately predicts the trade-off between re-running vs. reading and can be used to determine the right query execution strategy (as in Fig. 5-7).

One constraint when querying an intermediate via reading is that the number of RowBlocks read depends on the whether the examples queried are scattered and whether there is an appropriate index available on the RowBlocks. However, since the dotted and solid lines in Fig. 5-10 do not intersect, we see that even if MISTIQUE has to read the entire intermediate (50K) examples, it is faster to read the intermediate vs. re-run the model. In addition, while RERUN can only benefit from indexes on the input (e.g., find predictions for examples 36), MISTIQUE can index any intermediate and speed up queries in different layers (e.g., find predictions for examples with neuron-50 activation > 0.5).

5.8.4 Effect of Quantization on Accuracy

Next, we discuss the effect of our quantization strategies on diagnostic techniques. We highlight results from three queries, namely, VIS, SVCCA and KNN from Table 5.1.

VIS: Similar to [70], suppose we want to visualize the average activation of 256 neurons in layer-9 of the CIFAR10_VGG16 network. Fig. 5-11 shows heatmaps of these activations for full precision values (float32), LP_QT (float16), KBIT_QT (k=8), POOL_QT ($\sigma = 32$), KBIT_QT (k=3) and THRESHOLD_QT (99.5%). We see that there is no visual difference between full precision, LP_QT (float16), KBIT_QT (k=8) and POOL_QT (σ=32 or equivalently σ=2). However, KBIT_QT (k=3) and THRESHOLD_QT show obvious visual discrepancies.

SVCCA: The results of performing CCA [104] (captured in the average cca coefficient) between the logits produced by the CIFAR10_VGG16 network and representations of four different layers are shown in Table 5.5. We see that the cca coefficient for the 8BIT_QT intermediate is extremely similar to the full precision intermediate. In contrast, POOL_QT ($\sigma = 2$) introduces a discrepancy in the coefficient that reduces as the layer number increases. While 8BIT_QT is more accurate, reading 8BIT_QT is 6X slower and takes 1.5X more storage than pool(2).

K-nearest neighbors (KNN): In KNN, our goal is to find the k most similar examples to a given example similar to [8]. Table 5.6 shows the accuracy of KNN on different layers when using 8BIT_QT and POOL_QT ($\sigma = 2$). Here, we set k=50 and measure accuracy as the fraction of nearest neighbors that overlap with the true nearest neighbors computed on the full precision data. As with SVCCA, we find that 8BIT_QT produces almost the same neighbors as the full precision intermediates whereas POOL_QT usually captures 75% of the neighbors.

Thus, we find that 8BIT_QT is more accurate than pool(2) for some diagnostic queries; however, the increased accuracy comes at the cost of 1.5X more storage and 6X slower queries. In MISTIQUE, we choose to accept this lower accuracy of pool(2) but provide the user the option of using 8BIT_QT as the default storage scheme.

Layer	Full precision	8BIT_QT	pool(2)
SVCCA (value of average cca coefficient)			
6	0.8886	0.8868	0.6098
11	0.9185	0.9176	0.7085
16	0.7891	0.787	0.7464
19	0.8182	0.8182	0.8086

Table 5.5: SVCCA accuracy: Comparison of CCA coefficient across different storage schemes

Layer	Full precision	8BIT_QT	POOL_QT ($\sigma = 2$)
11	1.0	0.94	0.74
16	1.0	0.96	0.84
19	1.0	1.0	1.0

Table 5.6: KNN accuracy: Fraction of overlap between true KNN and KNN computed across different storage schemes.

5.8.5 Adaptive Materialization

In Sec. 5.5.2, we proposed a simple cost model to trade-off storage for an intermediate vs. the resulting decrease in query time. The impact of adaptive materialization is *highly workload dependent.* We present results from applying this optimization on a small synthetic query workload as a preliminary proof of efficacy for this optimization. For the Zillow task, we constructed a synthetic query workload by selecting at random 25 queries (with repetition) from Table 5.1. These queries where drawn from all the query categories — FCFR, FCMR, MCFR, and MCMR. We then used MISTIQUE to log intermediates with adaptive materialization turned on. We set γ to 0.5s/KB (i.e., trade-off 1 KB of storage for a 0.5s speedup in query time). Fig. 5-12 shows the impact of adaptive materialization on storage size and runtime of queries in this workload. On the left, we see that adaptive materialization (ADAPTIVE) has a small storage footprint compared to both STORE_ALL and DEDUP: intermediates are materialized only once an intermediate has been queried a large number of times. On the right of Fig. 5-12 we see the query times for three different queries (chosen to demonstrate three different behaviors). When no columns have been materialized, queries in the adaptive strategy take as long as RERUN. As more queries get executed (and therefore columns are materialized), the response time for queries reduces. In this example, we see reduction in response times for the VIS query but not for the COL_DIST query.

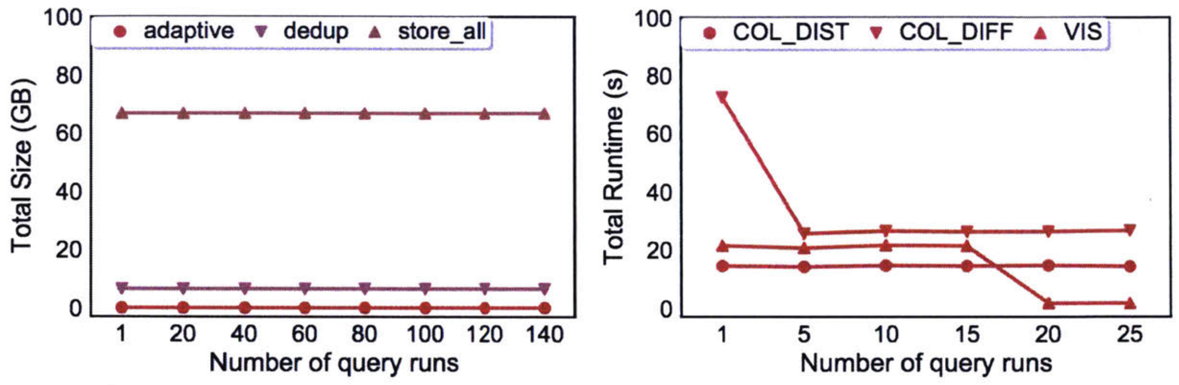
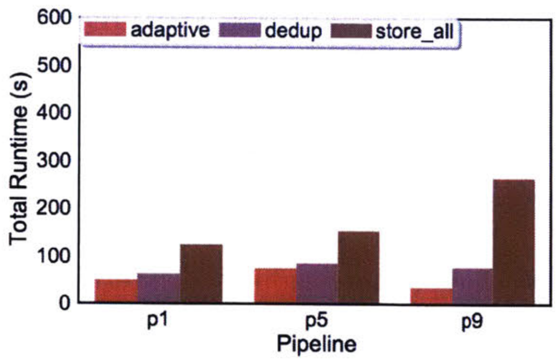


Figure 5-12: Adaptive Materialization: effect on storage and query time for synthetic `Zillow` workload

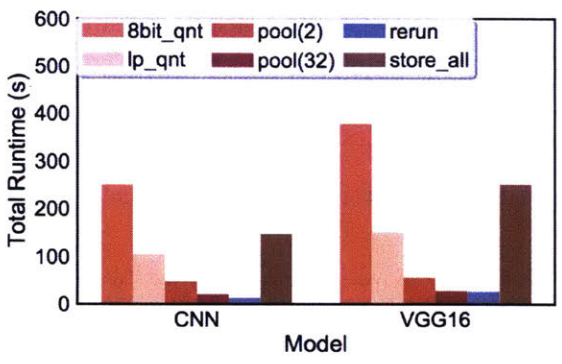
5.8.6 Workflow Overhead

As shown in the architecture diagram in Fig. 5-5, a new intermediate that is to be logged in MISTIQUE is first added to the InMemoryStore. Partitions from the InMemoryStore are written to disk only if the Partition is full or the Partition gets evicted from the InMemoryStore. Therefore, the exact overhead of logging depends on whether the relevant Partitions are full and if the InMemoryStore is already saturated: if the InMemoryStore is saturated, then logging an intermediate will result in a write to disk; however, if the InMemoryStore is not saturated, then there is no overhead associated with logging. Since InMemoryStore and Partition saturation depend closely on the workload, it is challenging to accurately estimate logging overhead in general. Instead, to provide an **upperbound** on logging overhead, we measure workflow execution time when each intermediate is written to disk synchronously.

Fig. 5-13 shows the total runtimes (including logging overhead) for three `TRAD` workflows, P1, P5 and P9 (see Table. 5.3). These were picked as representative workflows because of varying lengths and use of diverse models (they contain 12, 17, and 18 stages respectively). We find that workflow runtime is directly correlated with the amount of data written to storage. The `STORE_ALL` strategy consistently produces the largest workflow execution time since it writes the largest amount of data (see Fig. 5-8). `ADAPTIVE`, in contrast, has low but non-zero overhead (because



(a) Zillow workflows



(b) DNN models

Figure 5-13: Logging Overhead

it stores transformations used in the workflow). The DEDUP strategy produces modest overhead that is close to ADAPTIVE because it stores little data for each workflow.

For DNNs, we find that running the CIFAR10_VGG16 model without logging takes 19s. Storing all intermediates (without compression) takes 252s for single precision floats and 151s for half precision floats. When using 8BIT_QT, we pay an extra cost for computing quantiles and binning the data, resulting in workflow execution time of 379s. While 8BIT_QT takes 13X longer than running the model with no logging, this overhead is small compared to the time taken to train a model (often >30 mins for this model). Finally, using POOL_QT ($\sigma = 32$) on CIFAR10_VGG16 results in execution time of 20s — comparable to the time to run the model while $\sigma = 2$ requires 56s and $\sigma = 4$ requires 38s. Since CIFAR10_CNN shows similar trends, we do not discuss its logging overhead separately.

5.9 Summary and Discussion of Experimental Results

As mentioned before, the overall gains offered by a system like MISTIQUE vary depending on the details of ML-based workflows logged with the system. This includes the types of data consumed by ML-based workflows; the type, number and diversity of ML-based workflows logged; and properties of the operators used in workflows.

The set of experiments described in the previous section evaluated MISTIQUE in a controlled setting with a limited number of workflows. In this section, we summarize our results and comment on how we expect the results to generalize to other experimental settings and ML-based workflows.

- Query time: In our TRAD workflows, we found that reading an intermediate is always faster than re-running the model and can produce speedups between 2.5X — 390X. Large gains are seen for FCMR queries whereas MCMR shows small gains. For DNNs, the query speedup depends on what layer of the DNN that is queried; for Layer-21 (last layer), reading an intermediate is 60X — 210X faster, whereas for Layer-1 (first layer, closet to input), re-running can be upto 2.5X faster. In general, reading an intermediate is generally faster but the cost models can identify the right trade-offs depending on the intermediate and number of examples queried.

  In other settings, we expect that the trade-off between re-run and execute will depend on the workflow stages and sizes of intermediates. This highlights the need for a cost model such as the one we have developed in Chap. 5.5.

- Storage Cost: For TRAD models, we can take advantage of the large redundancy in intermediate columns and use the DEDUP strategy to reduce storage by 110X for the Zillow workflows. In contrast, for DNN models, we find that 8BIT_QT reduces storage footprint by 4X while POOL_QT ($\sigma = 2$) reduces footprint by up to 6X and POOL_QT ($\sigma = 32$) by up to 95X. DEDUP does not have a significant impact for DNNs except for cases where some weights are frozen (e.g., during fine-tuning for CIFAR10_VGG16).

  For TRAD models, a significant fraction of the storage reduction comes from consecutive intermediates and multiple workflows sharing columns. In experimental settings where workflows do not have the same input data (e.g., with many ML developers working on different problems or input data changing over time) or intermediates have few shared columns, we expect these gains to be lower. For DNN models, most of the gains we see are from quantization and summarization techniques that are applicable to the most common CNN layers. Moreover, these techniques and the resulting gains are independent of other DNNs that are trained. Consequently, we expect the DNN storage reductions to carry over to other experimental settings.

- Cost Model: Our experiments validate that the cost model proposed in Chap. 5.5 holds across different layers of a DNN and varying numbers of examples. The cost model can also accurately predict when it is faster to read an intermediate vs. re-run a model.

  Our proposed cost model is fairly general and does not make strong assumptions about the workflow or its input data. We expect it to be valid in a variety of experimental settings where the model is run on a single node. What the model currently does not capture, however, are any overheads that may be involved in materializing intermediates that are being generated in a distributed fashion (e.g., with Spark) and adding these considerations to the cost model is an important avenue for future work.

- Effect on Accuracy: The different storage strategies proposed by MISTIQUE represent intermediates in lossy fashion. As a result, not all storage schemes can support all diagnostic queries. For visualization queries, all storage schemes except for aggressive quantization (e.g., KBIT_QT (k=3) and THRESHOLD_QT) can produce visualizationst that are largely indistinguishable visually. For SVCCA and KNN, 8BIT_QT produces results that are highly accurate (but at a higher query cost) whereas POOL_QT ($\sigma = 2$) introduces some errors (but has low query cost). MISTIQUE adopts POOL_QT ($\sigma = 2$) with LP_QT as the default intermediate representation but depending on the diagnostic query workload, the ML developer can choose more aggressive or conservative quantization.

  For other experimental settings, we expect to see similar trends where the quality of results depends on the precision required by diagnostic techniques.

- Adaptive Materialization: This strategy can reduce the time required to run diagnostic queries (e.g., COL_DIFF in this workload) while maintaining a small storage footprint. However, the results of adaptive materialization are highly workload dependent. For example, columns will be materialized only if many queries in the workload query a column repeatedly. Therefore, if the queries are very diverse, then we will not see significant benefits of this optimization. Moreover, we note that adaptive materialization also requires the materialization threshold to be tuned appropriately.

  The impact of adaptive materialization in general depends significantly on the query workload. Consequently, we expect this optimization to produce large speedups in query time for workloads with similar queries and negligible speedups for other workloads.

- Overhead: Any type of intermediate logging adds an overhead that is directly proportional to the amount of data logged. In `TRAD` workflows, the `STORE_ALL` strategy can take upto 300s for logging large workflows whereas the `ADAPTIVE` strategy introduces negligible overhead beyond the raw workflow runtime of 30-50s. Similarly, for `DNN`s, `STORE_ALL` takes about 250s (compared to 19s for only running the model) while `POOL_QT` ($\sigma = 32$) takes 20s. However, `POOL_QT` ($\sigma = 32$) can be used to perform very few diagnostic queries. `POOL_QT` ($\sigma = 2$) which takes 56s to run presents a good trade-off between overhead, storage size, and supported diagnostic queries.

  In other experimental settings, we expect to see similar trends — we expect the overhead to scale with the amount of data logged.

Finally, all the workflow we tested were ones where the intermediate data could fit on a single machine. When the input data is so large that it cannot be stored on one machine (or exceeds storage limits), we may need to explore alternatives to fully materializing intermediates. For example, it may suffice to sample intermediates at the example level and only analyze a subset of examples. Similarly, we could develop techniques to estimate how much data is required to correctly perform a particular diagnosis task (similar to the work in visualization [73]). We believe this to be a ripe area for future work and expand on it in Chap. 7 on Future Work.

As suggested by the discussion above, the performance of MISTIQUE is ML workload dependent, and therefore, performing a user study with many ML applications in a real enterprise setting is an important direction for future work.

5.10 Conclusion

Model diagnosis is an essential part of the model building process. Analyses performed during model diagnosis often require access to model workflow intermediates such as features generated via feature engineering and embeddings learned by deep neural networks. Querying these intermediates for diagnosis requires either the intermediate to have been pre-computed and stored or to be re-created on the fly. As we demonstrate in this work, making an incorrect decision regarding reading vs. re-running can slow down diagnostic techniques by up to two orders of magnitude. In this work, we proposed a system called MISTIQUE tailored to capture, store, and query model intermediates generated from ML-based workflows. MISTIQUE uses a cost model to determine when to re-run a workflow vs. read an intermediate from storage. When storing intermediates, MISTIQUE uses unique properties of traditional machine learning workflows and deep neural networks to reduce the storage footprint of intermediates by 6X— 110X while reducing query execution time by up to 210X for DNNs and 390X for TRAD workflows.

Chapter 6

Applications of MISTIQUE

The MISTIQUE system described in the previous chapter provides ML developers easy access to data that was previously expensive to obtain (computationally or storage-wise). Efficient and easy access to arbitrary model intermediates can enable the development of new types of diagnostic techniques. As an example of new diagnostic techniques that can be enabled by MISTIQUE, the last contribution of this thesis is PREDICTIONVISUALIZER, a novel user-interface to visualize predictions (and other intermediate data) across many models. This work was done in collaboration with Wei-En Lee as part of his Masters thesis [78]. We provide an overview of the work here and point the reader to [78] for a detailed discussion.

6.1 Motivation

Visualization tools such as ActiVis and VizML described in the previous chapter provide ML developers the ability to visualize model workflow intermediates (or aggregates of model workflow intermediates) for a single workflow. These tools, however, are unable to extend to many workflows or facilitate comparison between workflows. Since ML developers in fact build tens to hundreds of ML-based workflows, providing ML developers the ability to quickly identify trends in workflow performance or among examples can speed up the modeling loop. Tools such as ModelTracker [8] took the first steps in providing such type of functionality by allowing the ML developer to compare predictions between consecutively built workflows. However, it is challenging to extend such a tool to hundreds of workflow iterations without a storage system such as MISTIQUE. Therefore, with PredictionVisualizer, we build upon the functionality of ModelTracker and provide ML developers the ability to run example-level comparisons across any number of workflows. For this work, we focus on visualizing **predictions** across many workflows. However, the same ideas could be used to visualize other intermediates such as embeddings from a CNN or results of different operators in a `TRAD` workflow.

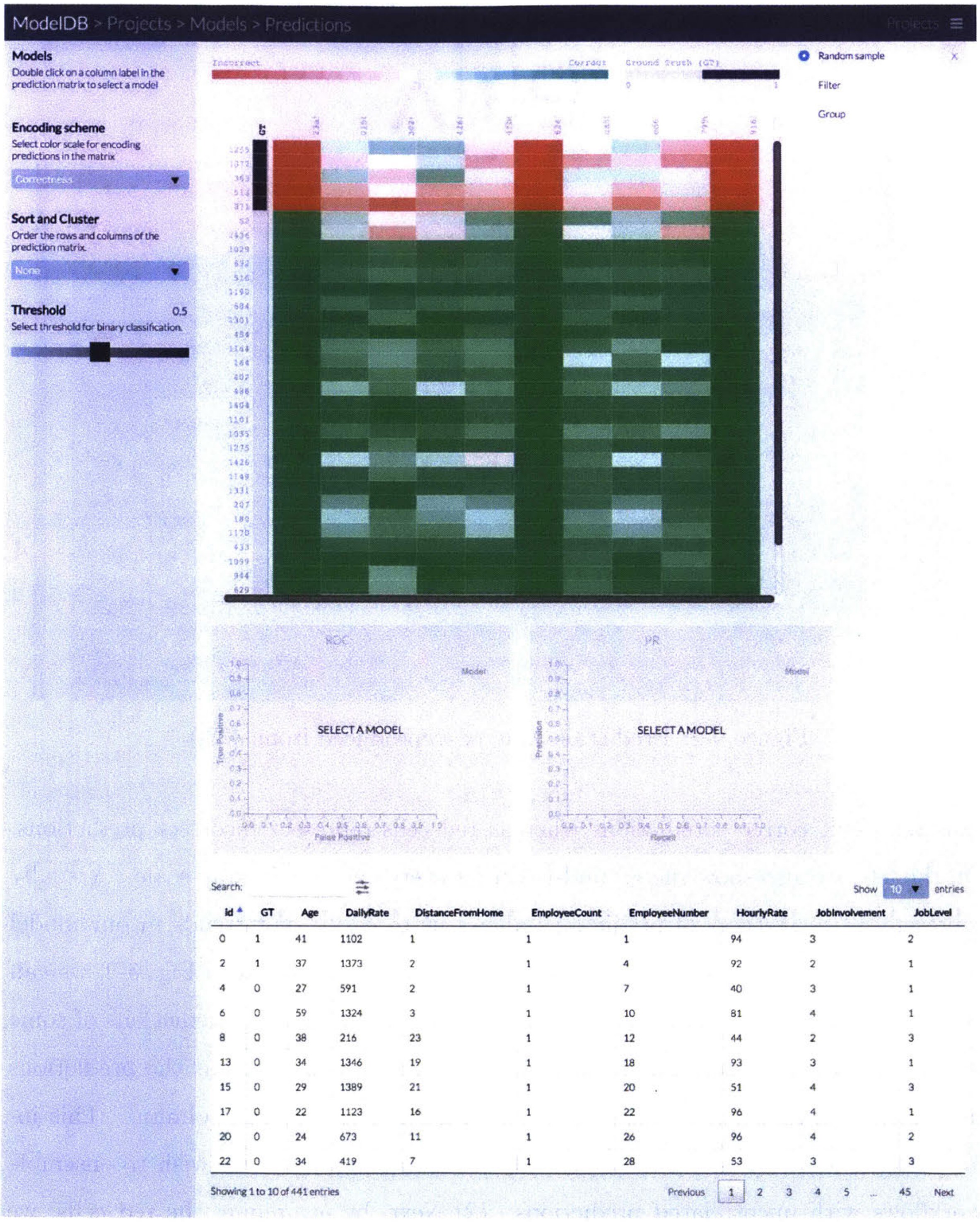
Note that our goal with this work was to explore new visual interfaces that can be constructed with intermediate data available in a system such as MISTIQUE. Building these interfaces to support large scale datasets brings up several scalability challenges that are out of scope for this work and are ripe directions for further exploration (see Future Work in Chap. 7). Similarly, the initial observations collected in our pilot study require further exploration via a large-scale user study.

6.2 PredictionVisualizer Interface

Fig. 6-1 shows an overview of the PredictionVisualizer interface. The visualizer is composed of three parts: the *Prediction Matrix* (top), the *Summary Pane* (middle), and *Input Data Explorer* (bottom). The Prediction Matrix presents example-level predictions across many examples and models, the Summary Pane provides summary statistics about model performance (e.g., accuracy, AUROC), and the Input Data Explorer provides the means to contextualize trends we see in predictions. We now examine each of these views in turn.

6.2.1 Prediction Matrix

Fig. 6-2 shows a detailed view of the Prediction Matrix. Each column in the Prediction Matrix corresponds to a ML-based workflow whereas each row of the matrix correspond to an example in the data. The color of each cell represents the correctness of the prediction captured as `1 - predicted_probability` for positive exam-



Models
Double click on a column label in the prediction matrix to select a model

Encoding scheme
Select color scale for encoding predictions in the matrix
Correctness

Sort and Cluster
Order the rows and columns of the prediction matrix.
None

Threshold 0.5
Select threshold for binary classification.

Search:

Show 10 entries

id	GT	Age	DailyRate	DistanceFromHome	EmployeeCount	EmployeeNumber	HourlyRate	JobInvolvement	JobLevel
0	1	41	1102	1	1	1	94	3	2
2	1	37	1373	2	1	4	92	2	1
4	0	27	591	2	1	7	40	3	1
6	0	59	1324	3	1	10	81	4	1
8	0	38	216	23	1	12	44	2	3
13	0	34	1346	19	1	18	93	3	1
15	0	29	1389	21	1	20	51	4	3
17	0	22	1123	16	1	22	96	4	1
20	0	24	673	11	1	26	96	4	2
22	0	34	419	7	1	28	53	3	3

Showing 1 to 10 of 441 entries

Previous 1 2 3 4 5 ... 45 Next

Figure 6-1: Prediction Visualization (reproduced from [78])

ples and `predicted_probability - 0` for negative examples. For the purpose of this work, we limit ourselves to binary classification problems. The cells in green

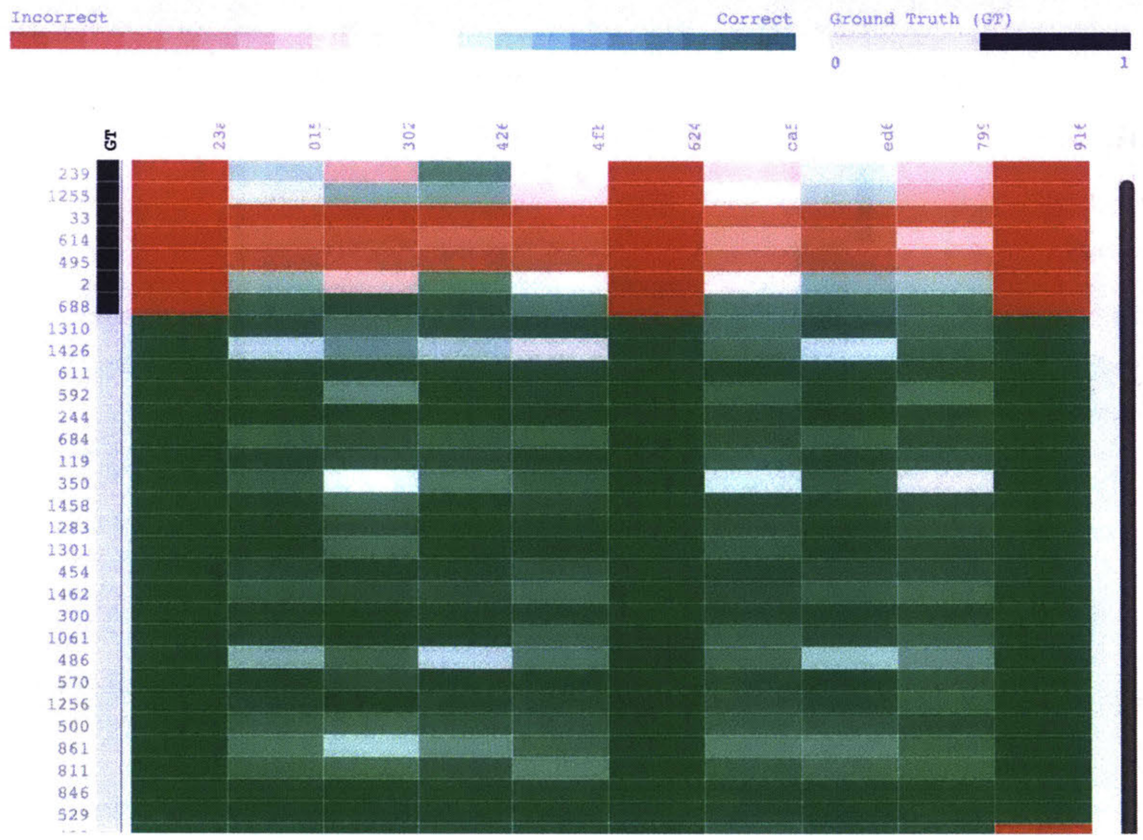


Figure 6-2: Prediction Matrix (reproduced from [78])

thus represent correct predictions whereas red cells represent incorrect predictions. On the left, we also show the ground-truth for every example in gray-scale.[1] Visually representing correctness of predictions allows us to easily spot trends in our model performance. For example, by examinging the Prediction Matrix in Fig. 6-2, we can spot a few trends right away: (1) As indicated by the colors, the predictions of some workflows are highly correlated (e.g., first and last column) whereas the predictions from some workflows have less correlation (e.g., first and second column). This information is directly useful in tasks such as ensembling where we wish to ensemble workflows with uncorrelated predictions. (2) Next, by examining the red cells, we can easily see where the errors made by these models are clustered. In this particular example, we can see that the workflows are mainly mis-predicting examples with ground-truth=1 (possibly because the dataset is imbalanced). This informa-

[1]Note that we can apply different color schemes for encoding prediction correctness.

tion can aid in improving the workflows or recommend the collection of more data with ground-truth=1. (3) Last, by examining changes in row color across multiple columns, we can easily identify prediction churn as described in Chap. 5).

The Prediction Matrix also supports other functions to support model diagnosis. For example, as shown in Fig. 6-3, it supports clustering of examples (or models) based on rows (or columns) to easily identify trends in the data. This operation can aid the ML developer in identifying examples (or models) with similar underlying characteristics. Similar to ModelTracker [8], we also allow the ML developer to easily find "nearest neighbor" or most similar examples (Fig. 6-4). This information about similar examples can help with finding errors such as mislabelled examples or suggest new ideas for feature engineering.

Last, the Prediction Matrix allows ML developers to define example groups and compare model performance across groups (similar to [71]). For example, Fig. 6-5 shows aggregate performance for examples with different values of the feature "age." Similar to [70], these aggregate statistics can be easier to interpret than individual instances and can therefore aid in identifying trends.

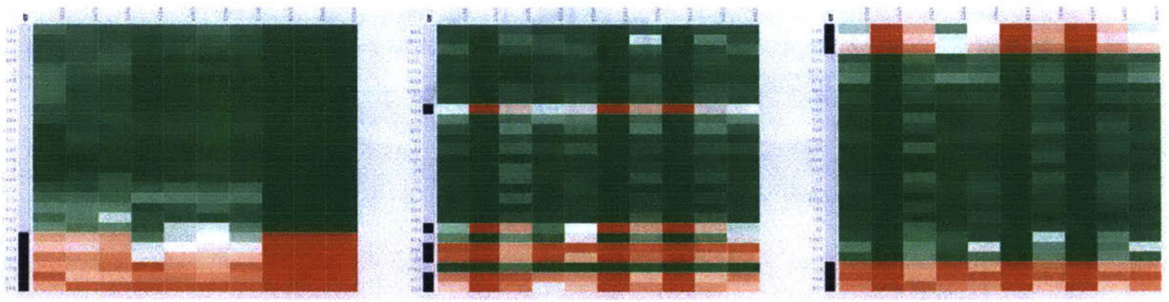


Figure 6-3: Clustering the Prediction Matrix (reproduced from [78])

6.2.2 Summary Pane

The second part of the PREDICTIONVISUALIZER interface is the Summary Pane. The user can select models in the Prediction Matrix to view summary statistics about model performance and make comparisons. Figs. 6-6—6-7 show the summary charts produced for different models. For every model selected in the Prediction Matrix, the

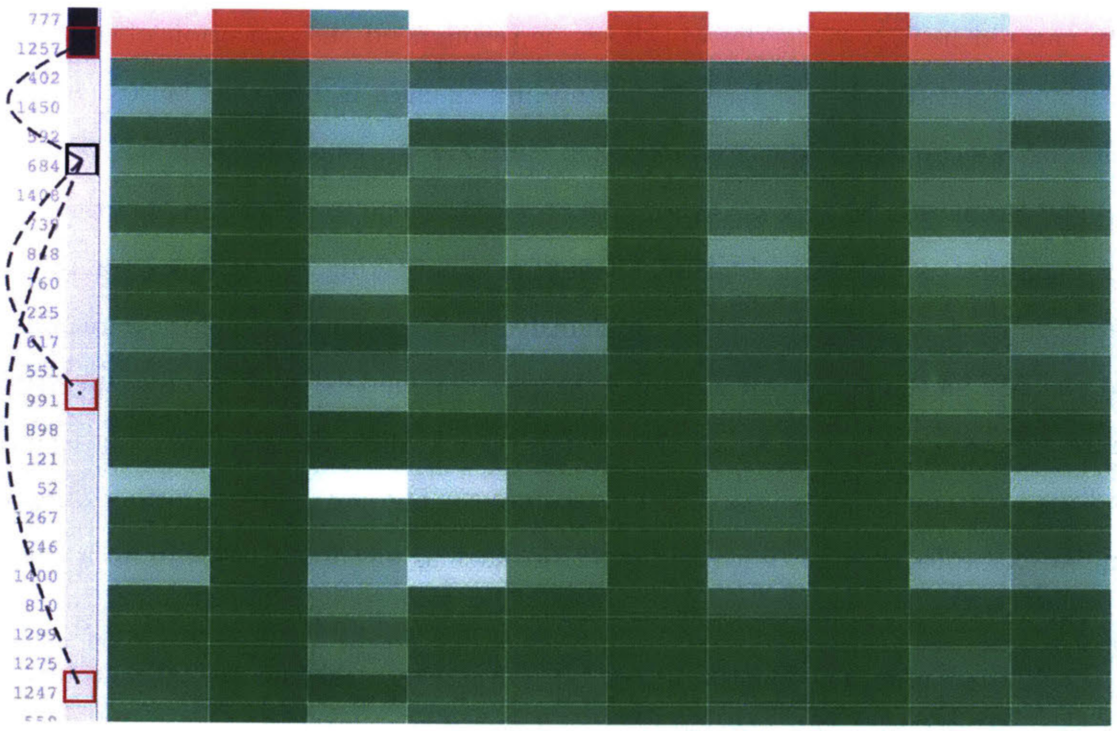


Figure 6-4: Nearest Neighbors in the Prediction Matrix (reproduced from [78])

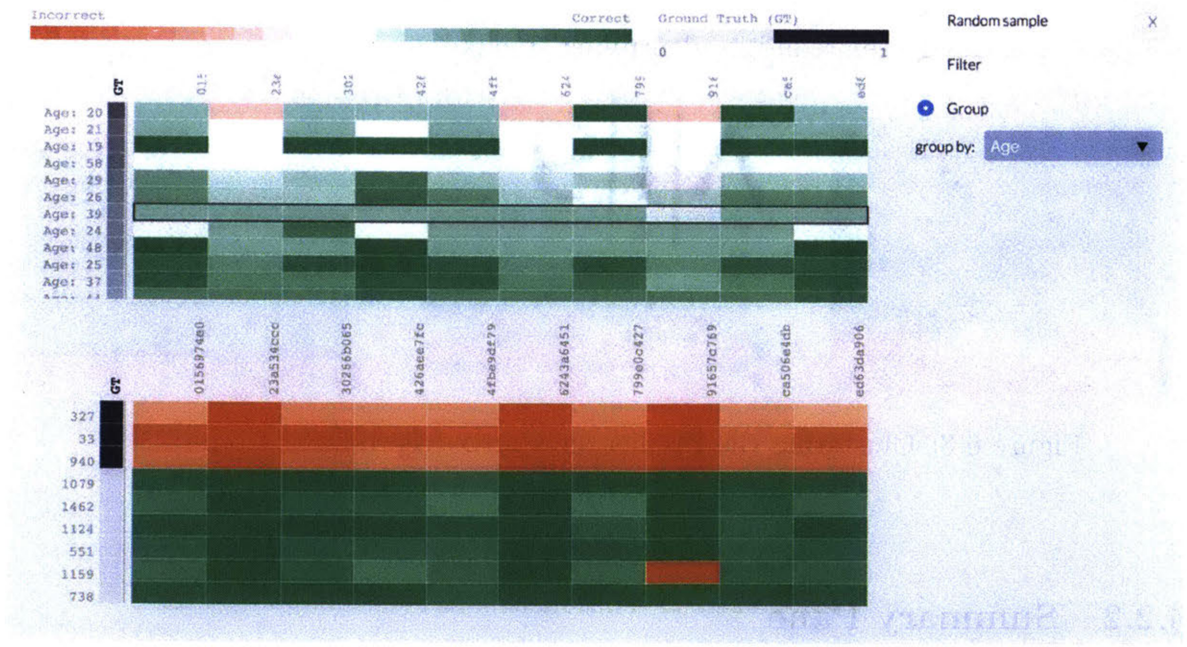


Figure 6-5: Grouping Predictions (reproduced from [78])

PREDICTIONVISUALIZER computes the ROC and PR curves as well as the confusion matrix. The Summary Pane then presents this data together in order to enable the

ML developer to quickly compare model performance.

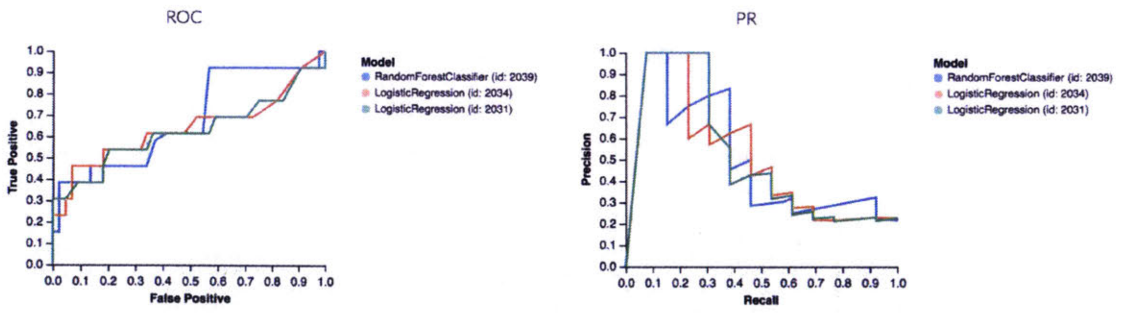
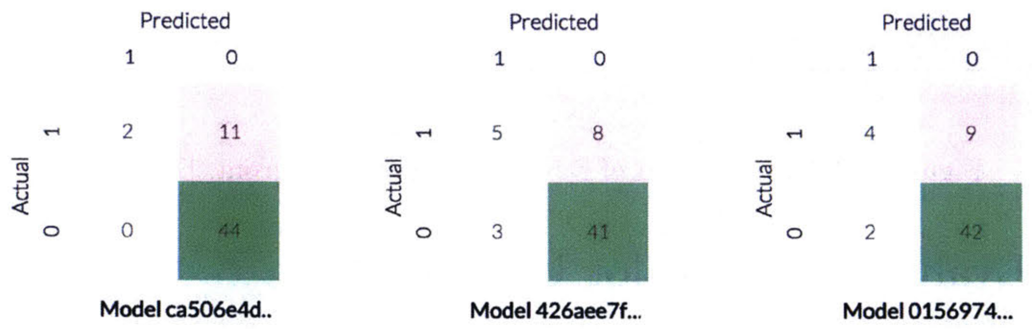


Figure 6-6: Comparing ROC and PR curves across models (reproduced from [78])



	Predicted 1	Predicted 0
Actual 1	2	11
Actual 0	0	44

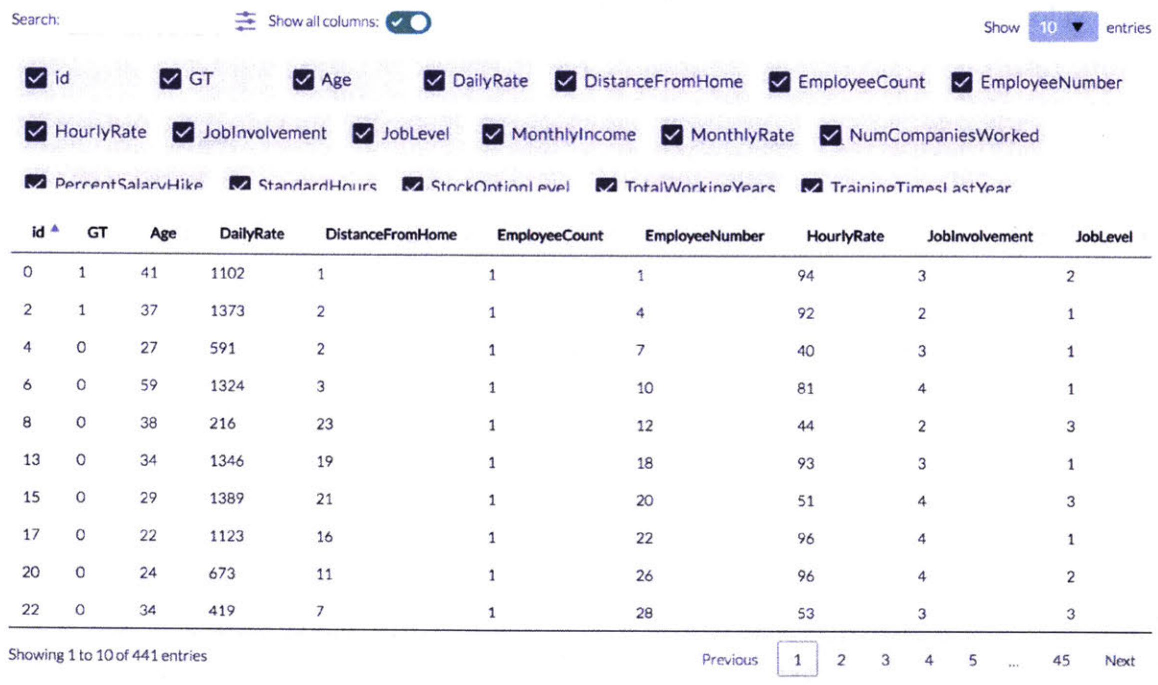
	Predicted 1	Predicted 0
Actual 1	5	8
Actual 0	3	41

	Predicted 1	Predicted 0
Actual 1	4	9
Actual 0	2	42

Figure 6-7: Comparing confusion matrices across models (reproduced from [78])

6.2.3 Input Data Explorer

The final part of the PREDICTIONVISUALIZER interface is the Input Data Explorer. In order to identify trends in example-level model performance, an ML developer needs the ability to contextualize results. Specifically, in order to (say) investigate why some examples are mispredicted by a model, the ML developer needs the ability to examine the examples themselves including their input representations. Therefore, the Input Data Explorer shows raw input data in tabular form (Fig. 6-8). The ML developer can sort and select examples, as well as search through the data to quickly drill-down to the relevant information. This part of the visualization is inspired by the input data visualization in VizML [26]. In the future, this visualization can be extended to visualize intermediates other than the input data.


Search:

Show all columns: ✓

Show 10 entries

✓ id ✓ GT ✓ Age ✓ DailyRate ✓ DistanceFromHome ✓ EmployeeCount ✓ EmployeeNumber

✓ HourlyRate ✓ JobInvolvement ✓ JobLevel ✓ MonthlyIncome ✓ MonthlyRate ✓ NumCompaniesWorked

✓ PercentSalaryHike ✓ StandardHours ✓ StockOptionLevel ✓ TotalWorkingYears ✓ TrainingTimesLastYear

id ▲	GT	Age	DailyRate	DistanceFromHome	EmployeeCount	EmployeeNumber	HourlyRate	JobInvolvement	JobLevel
0	1	41	1102	1	1	1	94	3	2
2	1	37	1373	2	1	4	92	2	1
4	0	27	591	2	1	7	40	3	1
6	0	59	1324	3	1	10	81	4	1
8	0	38	216	23	1	12	44	2	3
13	0	34	1346	19	1	18	93	3	1
15	0	29	1389	21	1	20	51	4	3
17	0	22	1123	16	1	22	96	4	1
20	0	24	673	11	1	26	96	4	2
22	0	34	419	7	1	28	53	3	3

Showing 1 to 10 of 441 entries

Previous 1 2 3 4 5 ... 45 Next

Figure 6-8: Visualization of raw data (reproduced from [78])

6.3 Evaluation via Pilot User Study

We evaluated the PREDICTIONVISUALIZER interface through a pilot user study. Our goal in this small-scale study was to examine (a) whether comparing across multiple workflow (vs. two workflow comparison) is important in debugging; and (b) whether visualizing predictions aids in workflow debugging. The pilot study was conducted with four participants and two variations of the interface. The first variation allowed prediction comparisons across any number of workflows (as above). The second, however, only showed data for two consecutive workflows: the current workflow and the one trained immediately prior to the current one. The second variation was designed to mimic the functionality provided by the ModelTracker interface [8]. In the pilot study, two participants were assigned to each interface.

The study was designed as follows. Study staff constructed ten workflows for the IBM HR Analytics Employee Attrition and Performance competition on Kaggle[2]. The task in this competition is to use employee records (e.g., years in current

[2]https://www.kaggle.com/pavansubhasht/ibm-hr-analytics-attrition-dataset

postition, job satisfaction rating) to predict whether an employee will churn. These workflows generated by the staff used different types of pre-processing steps as well as ML models and had varying error rates (i.e., no workflow was significantly better than the others). The resulting workflows were logged and predictions were loaded into the PREDICTIONVISUALIZER interface. Study participants had access to data about the workflow as well as predictions via the PREDICTIONVISUALIZER interface. Participants were asked to review both sources of data and suggest improvements to the workflow. They were also asked to talk aloud while exploring the interface. We collected participants' suggestions for improving the workflows as well as their feedback on the interface.

Our pilot study made the following *qualitative* observations. A large-scale study is required to draw quantitative conclusions about the efficacy of our interface.

- The Prediction Matrix visualization provides an easy means to identify problems with models and examples. For example, based on the visualizations, participants immediately identified that certain kinds of models performed poorly on the data.

- While example-level data is useful in debugging, summary data provides a better overview of workflow performance. This feedback suggests that the interface could be reordered to present the Summary Pane first followed by the Prediction Matrix.

- Multi-workflow comparisons are useful compared to two-workflow comparisons. Participants who used the two-workflow interface expressed interest in the ability to compare data across more than just two workflows.

- Tighter integration with workflow training is required. All study participants wanted the ability to build workflows from the same interface as the PREDICTIONVISUALIZER and examine prediction changes in real time.

Our pilot study thus provided some validation of our hypothesis that comparing results from multiple workflows is more useful than pairwise workflow comparisons and suggested ways of extending the functionality of our interface. Further development of the interface and a full-scale user study is necessary to quantify the efficacy of the PREDICTIONVISUALIZER.

Chapter 7

Future Work

This thesis work has made inroads into the problems of building systems for managing models and supporting model diagnosis. Much work remains to be done in each of these areas both to make the proposed systems applicable across diverse machine learning tasks and to address new problems brought up by each of these proposed systems.

7.1 ModelDB

We see three key directions for future work building on ModelDB. First, one of the main limitation with ModelDB (as of this writing) is that ModelDB provides native clients only for two libraries, namely scikit-learn and spark.ml. Other frameworks must use the Light API to log ML-based workflows to ModelDB. Consequently, the immediate next steps for ModelDB are to extend native support to other ML environments and libraries. In particular, providing ModelDB support for ML models built in R would benefit the large R user community. Similarly, given the increasing popularity of deep learning, extending ModelDB to treat deep learning models as first-class citizens would make model management available to a large number of deep learning researchers. Projects such as KubeFlow have already incorporated ModelDB into their system via the Light API but we could provide much deeper instrumentation including the logging of architectures, metrics, and training progress, among other things.

The second fruitful direction for future work is to explore alternative means of logging modeling pipelines. Currently, ModelDB relies on instrumenting key functions in various libraries to capture transformations. However, this approach has inherent limitations since new functions are constantly added to ML libraries and ML developers may compose functions in unusual ways leading to ModelDB being unable to log their workflow. Alternative ways of logging may include compiler/interpreter-level hooks or even intercepting calls at the operating system level. Some strides have been made in this direction through systems such as noWorkflow [90] and Ground [61]. For languages such as Scala and Java, it may even be possible to apply LLVM transformations to capture modeling pipelines with no change to user code (e.g., as in [125]).

The third direction for future work is in the area of model representation and storage. Often, ML developers prototype models in a research environment (e.g., in R or Python scikit-learn) but then need to deploy the model using a system such as Spark. In that case, there is no easy way to convert an R or Python model to Spark. Therefore, developing a common intermediate representation (such as ONNX [1] or PMML[2]) for inter-operability between frameworks can significantly speed up the process of ML deployment. A related direction for research is model storage. ModelDB currently routes model serialization to various ML frameworks and doesn't provide special support for optimized model storage. However, similarly to [86], we find that optimizing model storage is important for ML developers who want to build and version hundreds of models.

Last, our work assumes that the data that we use for modeling is adequate for the task and that it has been cleaned and integrated. While this is often true, there are many cases where more labeled data must be obtained, potentially through user input (e.g., via active learning [112]). We believe that tracking active learning in terms of new examples obtained or labeled is crucial to tracking how the performance of the ML model evolved with additional data. Including data gathering as another kind of

[1]https://github.com/onnx/onnx
[2]http://dmg.org/pmml/v4-1/GeneralStructure.html

operation in ModelDB is a rich avenue for future work and may produce extremely interesting insights about the utility of each training example.

Similarly, in data cleaning and integration, tracing how a record wound up in the ultimate training dataset is crucial for debugging potential errors in the model outputs. As discussed in the Related Work chapter (Chap. 3), there is a rich body of literature studying provenance or lineage of data. It would be worth exploring whether there are common operations or interfaces (in the software sense) that could be used to track cleaning operations. In the simplest case, if the key cleaning or integration operations were derived from the basic "transformer" interface described in ModelDB, then they could be tracked in the current system. However, it is likely that these operations will not follow a common interface and therefore there is room for a lot of work to increase coverage of these operations.

7.2 MISTIQUE

While our work on MISTIQUE made inroads into the problem of storing and querying model intermediates, we see multiple avenues for future work.

First, our work focuses on efficient storage techniques as a way to speed up diagnostic queries. A parallel means of achieving this goal is to speed up query execution via techniques such as indexing, sampling, and approximation. For example, deciding which examples are most "useful" in answering a diagnostic query might enable us to re-run the model on very small amount of data and thus produce a different performance trade-off. Similarly, MISTIQUE currently assumes that there are no special indexes present on the intermediates. However, one of the advantages of materializing intermediates is that we can build highly optimized index structures to speed up specific queries. For example, we could speedup nearest neighbor queries using techniques such as those described in [69]. Similarly, our query execution currently separates intermediate fetching from analysis; however, using knowledge about the analysis may allow us to make better decisions about fetching intermediates (e.g., whether we need to reconstruct quantized unit8 data to float32).

Next, as the size of the model scales, e.g., to ResNet-150 [58] with 150 layers, or the data scales to sizes comparable to ImageNet [108], it is possible that the trade-offs with respect to reading intermediates vs. re-running would change, and that we might need completely new methods for storing or encoding intermediates. On similar lines, extending our work to other types of models and data types, e.g., recurrent neural networks or time series data, may identify new opportunities for optimizations that are based on specific data properties or access patterns.

Third, MISTIQUE currently optimizes access to intermediates on a per-query basis. However, real-world diagnosis session often involves many queries, and therefore there may be opportunities to further reduce execution time via caching and pre-fetching. Similarly, batching queries could introduce new opportunities to optimize both reads and model re-runs.

Finally, the two systems MODELDB and MISTIQUE are closely related: MODELDB can be thought of as performing logical logging of ML-based workflows, whereas MISTIQUE performs physical logging. While our focus in this thesis was on developing the two sub-systems, integrating them is also a fruitful direction for future work.

7.3 Diagnostic Techniques

PREDICTIONVISUALIZER scratched the surface in terms of the new diagnostic techniques can be developed using the data in MISTIQUE. Other interesting applications of this data include detecting discrepancies between training and test datasets on the fly, summarization techniques to characterize mis-predicted examples, and new differencing techniques for comparing models and model pipelines, among many other.

Our small-scale pilot study with the PREDICTIONVISUALIZER also indicated that users would benefit from a unified workflow for building models and debugging, leaving lots of room to develop interfaces that support both of these "modes". We also see room for many improvements in scaling visualizations to large model intermediates with thousands of rows and columns (e.g., possibly by using techniques such as those proposed in imMens [81]). Finally, there are several interesting questions to be answered about the prediction visualizer interface with a full-scale user study: two key questions are (a) whether the prediction visualizer reduces the time to diagnose models and (b) which model data is most useful to surface in the debugger.

Chapter 8

Conclusion

Machine learning has become ubiqitous in a variety of applications. The key step in using ML in any application is to build a machine-learning based workflow to perform the given task. The process of building this ML-based workflow, i.e., the modeling process, is an iterative process with the ML developer experimenting with hundreds of workflows before obtaining one that meets some acceptance criteria. In this thesis, we explore the problem of making the modeling processes faster and more efficient. Specifically, we develop solutions to two problems: first, how can we manage ML workflows developed across many iterations in the modeling process; and second, how can we provide common infrastructure to support the diagnosis of ML workflows.

To address the first question, we propose a system called ModelDB that can automatically track ML workflows as the ML developer is developing them and make workflow metadata available in a common format. ModelDB currently provides client libraries for the popular scikit-learn and spark.ml libraries that can automatically track ML workflows. ModelDB also provides an intuitive and easy-to-use web-based interface to visualize the workflow data captured by the client libraries. ModelDB was the first open-source model management system and has been adopted at banks, large tech companies, and has been used in other open-source machine learning tools.

To address the second question, we propose a system called MISTIQUE. Our observation is that many model diagnosis techniques are based on data artifacts that are generated at different stages of a trained ML workflow. ML developers wanting to use these techniques must therefore either store large amounts of model intermediate data or continually re-run models in order to answer diagnostic queries. To address this problem and speed-up the process of model diagnosis, we built MISTIQUE—the **M**odel **I**ntermediate **ST**orage and **Q**uery **E**ngine. MISTIQUE provides a way to easily log intermediates from a variety of ML workflows (traditional and using DNNs) and query these intermediates efficiently. MISTIQUE implements different storage strategies to reduce the footprint of model intermediates and also develops a cost model to determine when to re-run a model for a diagnostic query vs. when to read a stored intermediate. The same cost model is used to decide when an intermediate should be materialized. We evaluate MISTIQUE on traditional ML workflows and DNNs and demonstrate that MISTIQUE can speed up certain kinds of diagnostic queries by up to two orders of magnitude. We also show that the storage strategies proposed in MISTIQUE can reduce storage footprint of traditional models by up to 110X and of deep neural network models by up to 6X.

The final contribution of this thesis is the PREDICTIONVISUALIZER, a new visualization interface that is an example of the new types of model diagnosis techniques (and tools) that can be enabled by MISTIQUE. Existing model performance debugging tools only support debugging predictions from one model or at most two models built consecutively. However, with MISTIQUE, an ML developer can log arbitrary intermediates from a large number of models. In PREDICTIONVISUALIZER, we surface prediction data from a large number of models to the ML developer and provide the developer with a visual interface for performing meta-analyses across models as well as identifying instance-level trends. Our pilot user study shows that ML developers can benefit from such a tool and identifies directions for further development.

The two systems proposed in this thesis, MODELDB and MISTIQUE, together take the first steps in building a common infrastructure for supporting model management and diagnosis. As machine learning continues to become a key part of business processes across many industries, we predict that the importance and need for such common infrastructure will continue to grow.

Bibliography

[1] Azure machine learning studio. `https://azure.microsoft.com/en-us/services/machine-learning-studio/`.

[2] Imagenet object detection challenge. `https://www.kaggle.com/c/imagenet-object-detection-challenge`.

[3] Kaggle.com. `https://kaggle.com`.

[4] Seahorse: Visual spark. `https://seahorse.deepsense.ai/`.

[5] Tableau software: Business intelligence and analytics. `https://www.tableau.com/`.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[7] Anastassia Ailamaki, David J DeWitt, and Mark D Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal - The International Journal on Very Large Data Bases*, 11(3):198–215, 2002.

[8] Saleema Amershi, Max Chickering, Steven M. Drucker, Bongshin Lee, Patrice Simard, and Jina Suh. Modeltracker: Redesigning performance analysis tools for machine learning. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 337–346, New York, NY, USA, 2015. ACM.

[9] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International Conference on Machine Learning*, pages 173–182, 2016.

[10] Yael Amsterdamer, Susan B. Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. *Proc. VLDB Endow.*, 5(4):346–357, December 2011.

[11] Michael R Anderson, Dolan Antenucci, Victor Bittorf, Matthew Burgess, Michael J Cafarella, Arun Kumar, Feng Niu, Yongjoo Park, Christopher Ré, and Ce Zhang. Brainwash: A data system for feature engineering.

[12] Monya Baker. 1,500 scientists lift the lid on reproducibility. *Nature News*, 533(7604):452, 2016.

[13] David Bau, Bolei Zhou, Aditya Khosla, Aude Oliva, and Antonio Torralba. Network dissection: Quantifying interpretability of deep visual representations. In *Computer Vision and Pattern Recognition*, 2017.

[14] Louis Bavoil, Steven P Callahan, Patricia J Crossno, Juliana Freire, Carlos E Scheidegger, Cláudio T Silva, and Huy T Vo. Vistrails: Enabling interactive multiple-view visualizations. In *Visualization, 2005. VIS 05. IEEE*, pages 135–142. IEEE, 2005.

[15] Omar Benjelloun, Anish Das Sarma, Alon Halevy, and Jennifer Widom. Uldbs: Databases with uncertainty and lineage. In *Proceedings of the 32nd international conference on Very large data bases*, pages 953–964. VLDB Endowment, 2006.

[16] Anant Bhardwaj, Amol Deshpande, Aaron J. Elmore, David Karger, Sam Madden, Aditya Parameswaran, Harihar Subramanyam, Eugene Wu, and Rebecca Zhang. Collaborative data analytics with datahub. *Proc. VLDB Endow.*, 8(12):1916–1919, August 2015.

[17] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *Proc. VLDB Endow.*, 8(12):1346–1357, August 2015.

[18] Souvik Bhattacherjee, Amol Deshpande, and Alan Sussman. Pstore: an efficient storage framework for managing scientific data. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, page 25. ACM, 2014.

[19] Rajendra Bose and James Frew. Lineage retrieval for scientific data processing: a survey. *ACM Computing Surveys (CSUR)*, 37(1):1–28, 2005.

[20] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, et al. Api design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238*, 2013.

[21] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *Proceedings of the 8th International Conference on Database Theory*, ICDT '01, pages 316–330, London, UK, UK, 2001. Springer-Verlag.

[22] Steven P Callahan, Juliana Freire, Emanuele Santos, Carlos E Scheidegger, Claudio T Silva, and Huy T Vo. Managing the evolution of dataflows with vistrails. In *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*, pages 71–71. IEEE, 2006.

[23] Steven P Callahan, Juliana Freire, Emanuele Santos, Carlos E Scheidegger, Cláudio T Silva, and Huy T Vo. Vistrails: visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 745–747. ACM, 2006.

[24] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[25] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Computer Vision (ICCV), 2015 IEEE International Conference on*, pages 2722–2730. IEEE, 2015.

[26] Dong Chen, Rachel KE Bellamy, Peter K Malkin, and Thomas Erickson. Diagnostic visualization for non-expert machine learning practitioners: A design study. In *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on*, pages 87–95. IEEE, 2016.

[27] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[28] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[29] Xiaozhi Chen, Kaustav Kundu, Ziyu Zhang, Huimin Ma, Sanja Fidler, and Raquel Urtasun. Monocular 3d object detection for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2147–2156, 2016.

[30] FranÃğois Chollet. keras. `https://github.com/fchollet/keras`, 2015.

[31] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[32] Kaggle Competition. Avito demand prediction challenge. `https://www.kaggle.com/c/avito-demand-prediction/`.

[33] Kaggle Competition. Titanic: Machine learning from disaster. `https://www.kaggle.com/c/titanic/`.

[34] Kaggle Competition. Zillow prize: ZillowâĂŹs home value prediction (zestimate). `https://www.kaggle.com/c/zillow-prize-1`.

[35] Kaggle Competitor. Kaggle kernel. `https://www.kaggle.com/aharless/xgboost-lightgbm-and-ols`.

[36] Kaggle Competitor. Kaggle kernel: Avito demand prediction competition. `https://www.kaggle.com/sudalairajkumar/simple-exploration-baseline-notebook-avito/code`.

[37] Kaggle Competitor. Kaggle kernel: Avito demand prediction competition. `https://www.kaggle.com/bminixhofer/aggregated-features-lightgbm`.

[38] Kaggle Competitor. Kaggle kernel: Titanic predition challenge. `https://www.kaggle.com/arthurtok/introduction-to-ensembling-stacking-in-python/code`.

[39] Kaggle Competitor. Kaggle kernel: Titanic predition challenge. `https://www.kaggle.com/poonaml/titanic-survival-prediction-end-to-end-ml-pipeline/code`.

[40] Kaggle Competitor. Kaggle kernel: Zillow home value prediction competition. `https://www.kaggle.com/aharless/xgb-w-o-outliers-lgb-with-outliers-combined`.

[41] Been Doshi-Velez, Finale; Kim. Towards a rigorous science of interpretable machine learning. In *eprint arXiv:1702.08608*, 2017.

[42] David Duvenaud, James Robert Lloyd, Roger Grosse, Joshua B. Tenenbaum, and Zoubin Ghahramani. Structure discovery in nonparametric regression through compositional kernel search. In *Proceedings of the 30th International Conference on Machine Learning*, June 2013.

[43] Eric Eide, Leigh Stoller, and Jay Lepreau. An experimentation workbench for replayable networking research. In *Proceedings of the 4th USENIX Conference on Networked Systems Design &#38; Implementation*, NSDI'07, pages 16–16, Berkeley, CA, USA, 2007. USENIX Association.

[44] Johan Eker, Jörn W Janneck, Edward A Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity-the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[45] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. Compressed linear algebra for large-scale machine learning. *The VLDB Journal*, Sep 2017.

[46] Facebook Engineering. Introducing fblearner flow: Facebook's ai backbone. `https://code.facebook.com/posts/1072626246134461/introducing-fblearner-flow-facebook-s-ai-backbone/`.

[47] Uber Engineering. Meet michelangelo: Uber's machine learning platform. `https://eng.uber.com/michelangelo/`.

[48] Reuben Feinman, Ryan R Curtin, Saurabh Shintre, and Andrew B Gardner. Detecting adversarial samples from artifacts. *arXiv preprint*, 2017.

[49] Apache Software Foundation. Airflow.

[50] Andrew Gelman and Eric Loken. The garden of forking paths: Why multiple comparisons can be a problem, even when there is no âĂIJfishing expeditionâĂİ or âĂIJp-hackingâĂİ and the research hypothesis was posited ahead of time. 2013.

[51] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[52] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *ICLR*, 2015.

[53] Ian J Goodfellow, David Warde-Farley, Pascal Lamblin, Vincent Dumoulin, Mehdi Mirza, Razvan Pascanu, James Bergstra, Frédéric Bastien, and Yoshua Bengio. Pylearn2: a machine learning research library. *arXiv preprint arXiv:1308.4214*, 2013.

[54] MIT DB Group. Modeldb. `https://github.com/mitdbg/modeldb`, 2017.

[55] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[56] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015.

[57] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[58] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[59] Jeffrey Heer, Jock D Mackinlay, Chris Stolte, and Maneesh Agrawala. Graphical histories for visualization: Supporting analysis, communication, and evaluation. *Visualization and Computer Graphics, IEEE Transactions on*, 14(6):1189–1196, 2008.

[60] Jeffrey Heer and Ben Shneiderman. Interactive dynamics for visual analysis. *Commun. ACM*, 55(4):45–54, April 2012.

[61] Joseph M Hellerstein, Vikram Sreekanti, Joseph E Gonzalez, James Dalton, Akon Dey, Sreyashi Nag, Krishna Ramachandran, Sudhanshu Arora, Arka Bhattacharyya, Shirshanka Das, et al. Ground: A data context service.

[62] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[63] Silu Huang, Liqi Xu, Jialin Liu, Aaron J Elmore, and Aditya Parameswaran. O rpheus db: bolt-on versioning for relational databases. *Proceedings of the VLDB Endowment*, 10(10):1130–1141, 2017.

[64] Robert Ikeda and Jennifer Widom. Panda: A system for provenance and data. In *Proceedings of the 2Nd Conference on Theory and Practice of Provenance*, TAPP'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[65] IBM Inc. How chatbots can help reduce customer service costs by 30%. `https://www.ibm.com/blogs/watson/2017/10/how-chatbots-reduce-customer-service-costs-by-30-percent/`, 2017.

[66] IBM Inc. How gpus are transforming the oil and gas industry. `https://blogs.nvidia.com/blog/2017/03/15/transforming-oil-and-gas-industry/`, 2017.

[67] StitchFix Inc. Stichfix algorithms tour. `http://algorithms-tour.stitchfix.com/`, 2017.

[68] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. Titian: Data provenance support in spark. *Proceedings of the VLDB Endowment*, 9(3):216–227, 2015.

[69] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

[70] Minsuk Kahng, Pierre Andrews, Aditya Kalro, and Duen Horng Chau. Activis: Visual exploration of industry-scale deep neural network models. *CoRR*, abs/1704.01942, 2017.

[71] Minsuk Kahng, Dezhi Fang, and Duen Horng (Polo) Chau. Visual exploration of machine learning results using data cube analysis. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, HILDA '16, pages 1:1–1:6, New York, NY, USA, 2016. ACM.

[72] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3149–3157, 2017.

[73] Albert Kim, Eric Blais, Aditya Parameswaran, Piotr Indyk, Sam Madden, and Ronitt Rubinfeld. Rapid sampling for visualizations with ordering guarantees. *Proceedings of the VLDB Endowment*, 8(5):521–532, 2015.

[74] Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1885–1894, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[75] M. Kreuseler, T. Nocke, and H. Schumann. A history mechanism for visual data mining. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS '04, pages 49–56, Washington, DC, USA, 2004. IEEE Computer Society.

[76] Sanjay Krishnan and Eugene Wu. Palm: Machine learning explanations for iterative debugging. In *Proceedings of the 2Nd Workshop on Human-In-the-Loop Data Analytics*, HILDA'17, pages 4:1–4:6, New York, NY, USA, 2017. ACM.

[77] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M. Patel. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Rec.*, 44(4):17–22, May 2016.

[78] Wei-En Lee. *Visualizations for model tracking and predictions in machine learning*. Master's thesis, 2017.

[79] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing*, 7(1):76–80, 2003.

[80] Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics*, 20(12):2674–2683, 2014.

[81] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. immens: Real-time visual querying of big data. *Computer Graphics Forum (Proc. EuroVis)*, 32, 2013.

[82] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[83] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems 30*, pages 4768–4777. Curran Associates, Inc., 2017.

[84] Michael Maddox, David Goehring, Aaron J Elmore, Samuel Madden, Aditya Parameswaran, and Amol Deshpande. Decibel: The relational dataset branching system. *Proceedings of the VLDB Endowment*, 9(9):624–635, 2016.

[85] H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, and Jeremy Kubica. Ad click prediction: A view from the trenches. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1222–1230, New York, NY, USA, 2013. ACM.

[86] H. Miao, A. Li, L. S. Davis, and A. Deshpande. Modelhub: Deep learning lifecycle management. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1393–1394, April 2017.

[87] Hui Miao, Amit Chavan, and Amol Deshpande. Provdb: Lifecycle management of collaborative analysis workflows. 2017.

[88] Mahdi Milani Fard, Quentin Cormier, Kevin Canini, and Maya Gupta. Launch and iterate: Reducing prediction churn. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 3179–3187. Curran Associates, Inc., 2016.

[89] Kevin P Murphy. *Machine learning: a probabilistic perspective.* MIT Press, Cambridge, MA, 2012.

[90] Leonardo Murta, Vanessa Braganholo, Fernando Chirigati, David Koop, and Juliana Freire. noworkflow: capturing and analyzing provenance of scripts. In *International Provenance and Annotation Workshop*, pages 71–83. Springer, 2014.

[91] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

[92] NPR. Will using artificial intelligence to make loans trade one kind of bias for another? https://www.npr.org/sections/alltechconsidered/2017/03/31/521946210/will-using-artificial-intelligence-to-make-loans-trade-one-kind-of 2017.

[93] Official Journal of the European Union. General data protection regulation. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex%3A32016R0679.

[94] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. The tiledb array data storage manager. *Proceedings of the VLDB Endowment*, 10(4):349–360, 2016.

[95] Hyunjung Park, Robert Ikeda, and Jennifer Widom. Ramp: A system for capturing and tracing provenance in mapreduce workflows. 2011.

[96] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[97] João Paulo and José Pereira. A survey and classification of storage deduplication systems. *ACM Computing Surveys (CSUR)*, 47(1):11, 2014.

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

[99] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18. ACM, 2017.

[100] João Felipe Nicolaci Pimentel, Vanessa Braganholo, Leonardo Murta, and Juliana Freire. Collecting and analyzing provenance on interactive notebooks: When ipython meets no workflow. In *Proceedings of the 7th USENIX Conference on Theory and Practice of Provenance*, TaPP'15, pages 10–10, Berkeley, CA, USA, 2015. USENIX Association.

[101] Peter Pirolli and Stuart Card. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of international conference on intelligence analysis*, volume 5, pages 2–4, 2005.

[102] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage.

[103] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria.

[104] Maithra Raghu, Justin Gilmer, Jason Yosinski, and Jascha Sohl-Dickstein. Svcca: Singular vector canonical correlation analysis for deep learning dynamics and interpretability. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 6078–6087. Curran Associates, Inc., 2017.

[105] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. Fast lossless compression of scientific floating-point data. In *Data Compression Conference, 2006. DCC 2006. Proceedings*, pages 133–142. IEEE, 2006.

[106] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 1135–1144, New York, NY, USA, 2016. ACM.

[107] Anthony C Robinson and Chris Weaver. Re-visualization: Interactive visualization of the process of visual analysis. In *Proceedings of the GIScience Workshop on Visual Analytics and Spatial Decision Support*, 2006.

[108] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[109] D Sculley, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high-interest credit card of technical debt.

[110] Adam Seering, Philippe Cudre-Mauroux, Samuel Madden, and Michael Stonebraker. Efficient versioning for scientific array databases. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 1013–1024, Washington, DC, USA, 2012. IEEE Computer Society.

[111] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. 7(8), 2016.

[112] Burr Settles. Active learning literature survey. Technical report, 2010.

[113] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, VL '96, pages 336–, Washington, DC, USA, 1996. IEEE Computer Society.

[114] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. *CoRR*, abs/1704.02685, 2017.

[115] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *CoRR*, abs/1312.6034, 2013.

[116] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[117] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[118] Facebook Open Source. Caffe2. `https://github.com/caffe2/caffe2`, 2015.

[119] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 535–546, April 2017.

[120] Spotify. Luigi.

[121] Michael Stonebraker, Paul Brown, Jacek Becla, and Donghui Zhang. Scidb: A database management system for applications with complex analytics. *Computing in Science and Engg.*, 15(3):54–62, May 2013.

[122] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[123] Harihar Subramanyam. *A system for storage and analysis of machine learning operations*. Master's thesis, 2017.

[124] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[125] Dawood Tariq, Maisem Ali, and Ashish Gehani. Towards automated collection of application-level data provenance. In *TaPP*, 2012.

[126] KubeFlow Team. Katib: Repository for hyperparameter tuning. `https://github.com/kubeflow/katib`.

[127] KubeFlow Team. Kubeflow: open, community driven project to make it easy to deploy and manage an ml stack on kubernetes. `https://github.com/kubeflow`.

[128] Kubernetes Team. Kubernetes: Production-grade container orchestration. `https://kubernetes.io/`.

[129] The Galaxy Team and Community. The galaxy project.

[130] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[131] Tom van der Weide, Dimitris Papadopoulos, Oleg Smirnov, Michal Zielinski, and Tim van Kasteren. Versioning for end-to-end machine learning pipelines. In *Proceedings of the 1st Workshop on Data Management for End-to-End Machine Learning*, page 2. ACM, 2017.

[132] Manasi Vartak, Sajjadur Rahman, Samuel Madden, Aditya Parameswaran, and Neoklis Polyzotis. S ee db: efficient data-driven visualization recommendations to support visual analytics. *Proceedings of the VLDB Endowment*, 8(13):2182–2193, 2015.

[133] Jason Wang and Luis Perez. The effectiveness of data augmentation in image classification using deep learning. Technical report.

[134] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2074–2082, 2016.

[135] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic acids research*, 41(W1):W557–W561, 2013.

[136] Eugene Wu, Samuel Madden, and Michael Stonebraker. Subzero: A fine-grained lineage system for scientific databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 865–876, Washington, DC, USA, 2013. IEEE Computer Society.

[137] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Rong Ma, Shuchen Song, and Aditya Parameswaran. Helix: Holistic optimization for accelerating iterative machine learning. Technical report.

[138] Hongyu Yang, Cynthia Rudin, and Margo Seltzer. Scalable bayesian rule lists. *arXiv preprint arXiv:1602.08610*, 2016.

[139] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. In *Deep Learning Workshop, International Conference on Machine Learning (ICML)*, 2015.

[140] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.

[141] Matthew D. Zeiler and Rob Fergus. *Visualizing and Understanding Convolutional Networks*, pages 818–833. Springer International Publishing, Cham, 2014.

[142] Ce Zhang, Arun Kumar, and Christopher Ré. Materialization optimizations for feature selection workloads. *ACM Transactions on Database Systems (TODS)*, 41(1):2, 2016.

[143] Zhao Zhang, Evan R. Sparks, and Michael J. Franklin. Diagnosing machine learning pipelines with fine-grained lineage. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, pages 143–153, New York, NY, USA, 2017. ACM.

[144] Zheguang Zhao, Lorenzo De Stefani, Emanuel Zgraggen, Carsten Binnig, Eli Upfal, and Tim Kraska. Controlling false discoveries during interactive data exploration. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 527–540. ACM, 2017.

[145] B. Zhou, A. Khosla, Lapedriza. A., A. Oliva, and A. Torralba. Learning Deep Features for Discriminative Localization. *CVPR*, 2016.

[146] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.