A System for Storage and Analysis of Machine Learning Operations

by

Harihar G. Subramanyam

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2017

Author ....
Department of Electrical Engineering and Computer Science
December 16, 2016

Certified by..
Samuel Madden
Professor, EECS
Thesis Supervisor

Accepted by ....
Christopher Terman
Chairman, Department Committee on Graduate Theses

A System for Storage and Analysis of Machine Learning Operations

by

Harihar G. Subramanyam

Submitted to the Department of Electrical Engineering and Computer Science
on December 16, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Data scientists go through an iterative process when building machine learning models. This process includes operations like feature selection, cross validation, model fitting, and evaluation, which are repeated until a sufficiently accurate model is produced. This thesis describes ModelDB Server and the ModelDB Spark Client, which record operations as the data scientist performs them, stores the operations and models in a central database, and exposes an API for gleaning information from the operations and models. ModelDB Server is library agnostic and it can serve as a foundation for other applications. ModelDB Spark Client is a library for the Apache Spark ML machine learning library that lets the data scientist log their operations and models with minimal code changes. ModelDB Server and Spark Client have low time and space overhead for large training dataset sizes and the overhead is independent of the dataset size.

Thesis Supervisor: Samuel Madden
Title: Professor, EECS

Acknowledgments

I would like to thank Prof. Madden and Manasi Vartak, whose advice and guidance have been extremely valuable as I worked on this thesis. I would also like to thank the rest of the ModelDB team; it was a pleasure to work together.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Schema

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

1.1 The Model Building Process

Machine learning models have been used to solve problems in a large number of domains. They can be used to recommend movies [22], classify tumors as benign or malignant [23], recognize faces [29], and more. Building these highly predictive machine learning models, however, is not easy.

A data scientist engaged in the model building process goes through many cycles of experimentation. For example, in a single cycle, a data scientist may perform the following steps:

1. Apply preprocessing operations to the data.

2. Split the data into train and test sets.

3. Select a model type and optimization criterion for training.

4. Define a hyperparameter search grid for the model.

5. Use cross validation to evaluate hyperparameter configurations and select a promising model.

6. Evaluate the model on the test set.

Over many cycles, the data scientist will create and evaluate many models, apply various preprocessing pipelines, and try numerous feature sets and hyperparameter configurations. Currently, recording the operations performed in these cycles requires a lot of manual effort. The data scientist must write their intermediate data files, models, evaluation metrics, and preprocessing code to different files. They may invent some ad-hoc directory layout and naming conventions to organize this information [20]. Even if the data scientist diligently records all their cycles, it is not easy for them to glean information from their recorded data without writing some custom scripts.

Recording information about the model building process can help the data scientist make better decisions on improving model performance. Comparing hyperparameters and feature sets may provide insight into why two models differ in performance. Examining the most important features may highlight what parts of the dataset are most useful for prediction. Tracing a model's lineage can show what preprocessing steps were most effective. Storing evaluation metrics for models can help the data scientist focus on only the highest quality models.

Recording information about the model building process can answer some useful questions, but currently it is difficult and time-consuming to do.

1.2 ModelDB Server and ModelDB Spark Client

This challenge of recording, storing, and learning from the model building process is precisely the problem that ModelDB Server and ModelDB Spark Client aim to solve.

ModelDB Spark Client is a lightweight library for Apache Spark's machine learning library (Spark.ML) that allows the user to collect information about their machine learning operations with minimal changes to their code.

The Spark Client sends the information it collects to ModelDB Server, which stores the operations and models in a centralized database and the serialized models in its filesystem.

The abstractions used in ModelDB Server are library agnostic. This means ModelDB Server and its database schema are not tied to any particular machine learning library, making it possible to develop clients for other popular machine learning libraries like Scikit-learn for Python.

ModelDB Server also exposes an API that allows the user to glean information about their operations and models. This allows users to compare models, find similar models, rank models, fetch the steps that produced a model, and more.

While ModelDB Server has functionality for storing and extracting information from all kinds of models, it stores extra information and provides richer APIs for logistic regression, linear regression, decision tree, random forest, and gradient boosted tree models.

Concretely, this thesis makes the following contributions:

1. A set of library-agnostic abstractions that can be used to represent a wide range of machine learning operations and models.

2. A number of algorithms that can be applied on the above abstractions to gain insight into the model building process.

3. A working system that can both store data about machine learning operations and models (especially linear and tree models) as well as extract useful information from this data.

4. A working client library for Spark ML that can collect information about various machine learning operations and models with minimal required changes to a data scientist's code.

For the rest of this paper, ModelDB Server and the Spark Client will be referred to collectively as ModelDB S+C.

1.3 Usage Scenario

To motivate why ModelDB S+C would be useful, consider the usage scenario described below.

Data scientist Alice is trying to build a model that can predict the click through rate of an advertisement based on various features of the advertisement (e.g. target demographic, color scheme). She uses Spark and ModelDB S+C. Alice first applies various preprocessing steps to her data, which are recorded and stored in ModelDB Server. At one point, she realizes that there is a bug in one of her dataset's columns, so she uses ModelDB Server's API to determine all the operations she has applied to her dataset. With this information, she realizes she forgot a preprocessing step, so she makes sure to apply it. Next, Alice trains a few models on the dataset and computes various evaluation metrics on them. Using ModelDB Server's API, she can rank these models along various metrics and identify which one performed best overall. Alice selects the best model and uses ModelDB Server's API to rank its features by importance. She sees that it is one of the few models she created that uses the "color scheme" feature, so she suspects that this feature may be important. So, Alice uses ModelDB S+C's annotation API and annotates the model with the text "Model works really well, I think the color scheme feature is important for accurate prediction" so that her colleagues will be informed if they later train similar models. Alice trains a new model and finds that its performance is worse. To debug the issue, she uses ModelDB Server's API to compare the model to the best one she has trained so far. She sees that there's a difference in the number of iterations the models were trained for. Alice trains the new model for a greater number of iterations and finds that the performance of the model has greatly improved. Happy with the results, she uses ModelDB's visualization tool (part of the overall ModelDB system, it is built on ModelDB Server) to graphically view all the operations she performed so that she can refer to it as she writes about her process and shares her results with her colleagues.

Later, data scientist Bob comes along and is also looking for a system to predict ad click through rate. He uses ModelDB Server's API to find all the models that descended from the ads dataset. He uses ModelDB Server's API to group these models based on their problem type, and he focuses on the ones that were used for predicting click through rate. Then, he uses ModelDB Server's API to rank these models and find the best performing model. The best performing model turns out to be one that would be difficult to scale for Bob's use case, so he uses ModelDB Server's API to find a similar model to the best one. This yields Alice's model, which is similar in performance, but much easier to scale. ModelDB S+C stores serialized models, so Bob loads Alice's model and uses it to make a few predictions. Happy with the results, Bob decides to evaluate the model on the latest installment of the ads dataset. However, he doesn't know how to preprocess the data. Fortunately, Bob is able to use ModelDB Server's API to extract the preprocessing pipeline that was applied to the model's original dataset, and he loads each of the data preprocessors and preprocesses his data. Then, he feeds in his dataset and evaluates the model. Happy with its performance, he decides to use it in his system.

1.4 Outline of this Thesis

Chapter 2 covers related work. It first discusses machine learning in general, exposing the key concepts that are relevant in the model building process. The discussion then shifts to linear and tree models in particular, because ModelDB S+C has special support for these kinds of models. Next, Spark and Spark ML are briefly described. The chapter then proceeds to discuss popular machine learning libraries and their abstractions and concludes by discussing other machine learning support systems.

Chapter 3 covers the abstractions used in ModelDB S+C. It begins with a discussion of Transformer, TransformerSpec, and DataFrame, which are the primitives that are used to create the other abstractions. Next, the chapter introduces the Syncable Event abstraction, which represents a machine learning operation that can be logged. Then, some of the simple Syncable Events, like FitEvent, which represents the fitting of a model, are described. Next, more complicated Syncable Events, like PipelineEvent are discussed. The chapter then moves on to discuss the abstractions for representing linear and tree models. Finally, the ModelDB Syncer, a client side abstraction for sending events to ModelDB Server, is discussed.

Chapter 4 covers algorithms used in ModelDB S+C. It begins with storage algorithms that capture the complexities involved in storing and connecting the Syncable

Events. The next topic is ancestry algorithms, which operate on the ancestry forest of DataFrames. The chapter then talks about algorithms that operate on the feature sets of models and algorithms that compare, rank, and group models. Finally, the chapter discusses algorithms for linear models and tree models.

Chapter 5 covers implementation. It starts with an overall view of the system, including ModelDB Server, the database, the model filesystem, Spark, and more. Then, attention shifts to an outline of ModelDB Server's implementation. Next, the chapter discusses the Spark Client's implementation. The chapter concludes with a discussion of how applications can be built on top of ModelDB Server.

Chapter 6 is the evaluation. It first describes an experiment run to measure the time and space overhead of collecting operations and models from Spark and recording them in ModelDB Server. The chapter then describes an experiment to measure the time taken to run some of the more complicated API methods exposed by ModelDB Server. The chapter then considers real machine learning workflows, and examines how well ModelDB S+C is able to capture the operations performed. Finally, the chapter discusses potential performance improvements.

Chapter 7 describes the role of ModelDB S+C in ModelDB, a larger system that includes other components like a client library for Scikit-learn, a command line toolkit, a prediction store, and a visualization application.

Chapter 8 covers potential future work that could be done on ModelDB S+C.

Chapter 9 concludes the thesis.

Chapter 2

Related Work

2.1 Machine Learning

There are many classes of problems that machine learning can be used to solve, such as the following [3]:

- **Supervised Learning**: Predict a real valued output (regression) or select one of C predefined categorical outputs (classification) for a given input vector. This also includes problems like anomaly detection, ranking, and regression.
- **Unsupervised Learning**: Find structure or regularity in a given set of input vectors. This includes problems like density estimation and clustering.
- **Reinforcement Learning**: Develop a policy that allows an agent to observe the state of its environment and take the actions that will achieve a large cumulative reward in the long run [28].

ModelDB S+C can store operations and models for supervised and unsupervised learning problems. Reinforcement learning, however, is out of scope for this thesis.

For a supervised learning problem, suppose there is a dataset $\mathcal{D}$ that consists of n pairs of $(\mathbf{x}^{(i)}, y^{(i)})$. There is a pair for each $i \in \{1...n\}$. $\mathbf{x}^{(i)}$ is a p-dimensional feature vector that decribes the i^{th} example. So, $\mathbf{x}^{(i)} \in \mathbb{R}^p$. $y^{(i)}$ is either the class label associated with the i^{th} example or the regression target for that example. That is, $y^{(i)} \in \{1...C\}$ for classification and $y^{(i)} \in \mathbb{R}$ for regression. The $\mathbf{x}^{(i)}$ vectors can also be expressed as a $n \times p$ matrix $\mathbf{X}$.

A machine learning model is a function $f(\mathbf{x}; \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ is a vector of model parameters (e.g. the weights of a linear regression model or the encoded splits of a decision tree model). This function accepts an input vector $\mathbf{x} \in \mathbb{R}^p$ and produces a real output (for regression) or probability of being in a specific class (for classification).

When training a model, it is useful to split the dataset $\mathcal{D}$ into a training set $\mathcal{D}_{train}$ and testing set $\mathcal{D}_{test}$. This ensures that the model is evaluated on data it has not seen before. An objective function $J(\boldsymbol{\theta}, \mathcal{D}_{train})$ is defined and $\boldsymbol{\theta}$ is chosen to maximize (or minimize, depending on the formulation) the objective function.

Machine learning also requires some hyperparameters (e.g. maximum depth of decision tree, regularization constant) to be set in order to guide the training of the model. One way to pick values for the hyperparameters is to use a validation set. That is, the dataset $\mathcal{D}$ is broken into three pieces, $\mathcal{D}_{train}$, $\mathcal{D}_{validation}$, and $\mathcal{D}_{test}$. For each hyperparameter configuration, a model is trained on $\mathcal{D}_{train}$. Then, the model is evaluated on $\mathcal{D}_{validation}$. Then, the chosen hyperparameter configuration is the one that yielded the model that performed best on the validation set. Using a technique called cross validation can make better use of the data. In k-fold cross validation, $\mathcal{D}_{train}$ is broken into k pieces (or folds) of roughly equal size. Then, for a given hyperparameter configuration, a total of k models are trained, each trained on all but one of the folds. Each model is evaluated on the fold that was left out of its training, and the resulting evaluation metrics are aggregated (e.g. averaged) to produce the evaluation metric for the hyperparameter configuration. The hyperparameter configuration with the largest evaluation metric is chosen as the best hyperparameter configuration. Then, a final model is trained on the entire $\mathcal{D}_{train}$ and is then evaluated on $\mathcal{D}_{test}$ [12].

While the above concepts are not exhaustive, they provide an overview of machine learning that is sufficient to understand the model building process and ModelDB S+C.

2.2 Linear and Tree Models

ModelDB S+C has special support for linear regression, logistic regression, decision tree, random forest, and gradient boosted tree models. Therefore, it is worth discussing these models briefly.

A linear regression model is a function $f(\mathbf{x}; \boldsymbol{\theta}) = \boldsymbol{\theta}^T \mathbf{x}$, where $\boldsymbol{\theta} \in \mathbb{R}^p$. Requiring that all feature vectors have an first entry of 1 allows the model to incorporate an intercept term.

A logistic regression model is not actually used for regression, it is used for binary $(C = 2)$ classification. Under appropriate assumptions, the logistic regression model aims to predict the likelihood that a given input feature vector belongs to class 1. Concretely, the logistic regression function is: $f(\mathbf{x}; \boldsymbol{\theta}) = \frac{\exp \boldsymbol{\theta}^T \mathbf{x}}{1 + \exp \boldsymbol{\theta}^T \mathbf{x}}$. Its output lies between 0 and 1.

A decision tree model partitions the input space into non-overlapping regions, where all points in the same region are assigned the same value. This value could be a class label in the case of classification or a real value in the case of regression. For a given input vector, the decision tree considers a different feature at each internal node. The input vector is forwarded to a child node based on the value of the feature. This child node, if it is an internal node, then considers another feature and performs the same process. Eventually, the input vector arrives at a leaf node. Each leaf node corresponds to a region of the input space, and thus has an associated value (i.e. class label or real number), which is taken to be the prediction for the input vector. An illustration of this is shown in Figure 2-1.

Random forest models and gradient boosted tree models are both ensembles of decision trees. They just differ in how they are trained and applied. A random forest model trains many trees in parallel, where each tree is trained on a different bootstrapped sample of the original dataset and where each split is only allowed to consider a subset of the features. A gradient boosted tree model trains decision trees iteratively, where each successive decision tree learns to correct the mistakes of the previous trees. Like decision trees, random forest and gradient boosted tree models

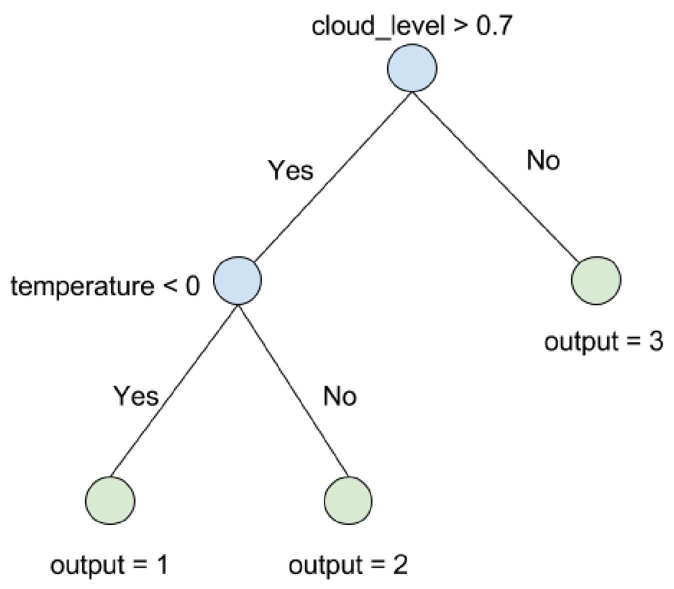


Figure 2-1: This is a decision tree model that could be used to predict whether the weather will be snowy (output 1), rainy (output 2), or neither (output 3) by looking at the cloud cover and temperature. The tree first checks if the cloud cover is high. If not, then it predicts there will be neither rain nor snow. Otherwise, the tree checks if the temperature is below 0. If so, then it predicts snow. Otherwise, it predicts rain.

can be used for both classification and regression. [17]

These are not the only linear and tree models that are used in practice, but they are models that are supported in Spark.ML. Therefore, ModelDB S+C provides additional functionalty for the aforementioned models.

2.3 Spark.ML

Apache Spark [32] is a cluster computing engine that efficiently performs distributed computation on data stored in-memory on many different machines. It is built around an abstraction called the Resilient Distributed Dataset (RDD). An RDD represents a dataset, its lineage, and its partitioning. By tracking a dataset's lineage, an RDD allows the dataset to be recreated if it is lost, thus providing fault tolerance. By tracking the partitioning and lazily performing operations, Spark uses RDDs to schedule operations intelligently and pipeline them for efficiency.

Spark includes a number of libraries, two of which are relevant for this thesis.

The first is Spark SQL, which builds an abstraction called a DataFrame on top of the RDD abstraction. A DataFrame is simply a table of data with named, typed columns. The second library is Spark.ML, which lets users build machine learning models using Spark. Spark.ML lets the user create Transformers that take an input DataFrame and produce an output DataFrame. It lets users use Estimators, which accept hyperparameters and a DataFrame and produce a resulting model, which is simply a Transformer. Finally, Spark.ML provides a number of other classes to support common model building tasks like cross validation and making preprocessing pipelines.

ModelDB S+C is built on three primitives: Transformer, DataFrame, and TransformerSpec, which are all inspired by Spark SQL and Spark.ML. Transformer and DataFrame match their Spark counterparts, but unlike a Spark Estimator, a TransformerSpec simply describes how to create a model, rather than containing the logic for actually training the model. ModelDB S+C does not train models, so there is no need for it to have model training logic.

The three primitives above, when coupled with ModelDB S+C's Syncable Event abstraction, can express a wide range of model building operations.

The ModelDB Spark Client is a library designed for Spark.ML. It allows the user to use their existing Spark.ML code, and with a few minor changes, store all their operations and models in ModelDB Server.

2.4 Machine Learning Libraries

Since ModelDB Server aims to be library agnostic, it is worth looking at some other popular machine learning libraries and understanding their abstractions for the model building process.

2.4.1 MLI

MLI [27] is an API for distributed machine learning. It includes an MLTable abstraction that is very similar to the DataFrame abstraction in ModelDB S+C. MLI includes two other tabular abstractions, the LocalMatrix and MLNumericTable, which are more restrictive forms of the MLTable. MLNumericTable and LocalMatrix make computation easier, but they do not add anything to the expressive power of MLI.

MLI's Optimizer and Algorithm abstractions help specify the logic for training a model. However, as stated before, ModelDB S+C does not perform any training of models and thus these abstractions are not important in the context of ModelDB S+C. That being said, the user can store information about the algorithms and optimizer used to train a model in the hyperparameters of the model's associated TransformerSpec.

MLI's Model abstraction maps to the Transformer abstraction in ModelDB S+C. There are two key differences between these abstractions. First, a Transformer is more general than a Model; it can represent data preprocessors as well as models. Second, a Model stores the logic for making predictions while a Transformer does not. This is because ModelDB S+C does not aim to make predictions, just to store and analyze the model building process. However, the user is able to store serialized models in ModelDB S+C, which can later be deserialized and used to make predictions.

2.4.2 Scikit-learn

Scikit-learn [7] is a machine learning library for Python. It expects data to be given as matrices (two dimensional arrays in the numpy Python library) and DataFrames (from the Python Pandas library). Both of these abstractions map nicely to ModelDB S+C's DataFrame. A matrix can be thought of as a DataFrame where all columns are numeric and where the column names are ignored.

Scikit-learn includes the concept of an Estimator, which contains logic for training a model and stores the hyperparameters. This maps well to ModelDB S+C's TransformerSpec. The logic for training the model is not necessary for ModelDB S+C. Scikit-learn also includes Transformer and Predictor concepts. The former is almost identical to ModelDB S+C's Transformer abstraction while the latter is simply a more restrictive kind of Transformer.

Finally, Scikit-learn includes abstractions for cross validation and pipelines, much like Spark.ML does. Both of these model building tasks are represented in ModelDB S+C as Syncable Events.

2.4.3 Weka

Weka [16] is a machine learning toolkit that includes a user interface targeted for non-expert users. It represents data as tables, which is very similar to ModelDB S+C's DataFrame. However, Weka does not include separate abstractions for describing a model and describing the fitting of a model. Instead, it combines them together in an abstraction called a Scheme. Thus, ModelDB S+C's TransformerSpec and Transformer abstraction together reflect the same information as a Weka Scheme. Finally, Weka includes pre-processing and post-processing utilities which map well to ModelDB S+C's Transformer abstraction.

2.4.4 Pylearn 2

Pylearn 2 [13] is a machine learning library geared towards researchers. It includes the concept of a Dataset, Model, and TrainingAlgorithm, which are analogous to ModelDB S+C's concepts of a DataFrame, Transformer, and TransformerSpec. ModelDB S+C's Transformer abstraction is more general than a Model because it can capture both models as well as data preprocessors. Pylearn 2 also includes abstractions such as Cost and TerminationCriterion. Since ModelDB S+C does not concern itself with the details of training models, it has no corresponding abstractions and instead allows the data scientist to specify cost and termination criterion as hyperparameters in a TransformerSpec.

2.4.5 Tensorflow

Tensorflow [1] is machine learning library that allows users to specify computation graphs, which is especially useful for training neural network models. It represents data in the form of a Tensor. This is actually more general than the DataFrame concept in ModelDB S+C. However, using a tensor abstraction in ModelDB S+C may add a good deal of complexity to the other abstractions without enabling a great deal of new functionality.

Tensorflow represents data in the form of a computation graph, where nodes called variables use operations along edges to send tensors through the graph. ModelDB S+C stores a graph, specified by the TransformEvents, where the nodes are DataFrames and the edges are Transformers. Tensorflow's computation graph is useful for performing efficient training of models. However, since ModelDB S+C does not actually do any training of models and instead just captures operations like transformations and model-fitting, its graph is different than TensorFlow's. Additionally, Tensorflow requires the user to specifiy their computation graph up-front in order for the library to determine how to train the model. ModelDB S+C, on the other hand, builds up its graph over time as the user performs operations.

2.4.6 Summary

Thus, ModelDB S+C's abstractions seem to be general enough that there are analogs in a number of other machine learning libraries. Of the examples above, Tensorflow is the only one with a significantly different set of abstractions than ModelDB S+C.

2.5 Model Building Support Systems

ModelDB S+C does not actually train any machine learning models and it does not make any predictions. Rather, it collects, stores, and analyzes data about the model building process. In this way, ModelDB S+C supports the process of model building rather than performing the process itself. This section describes other systems that support the model building process.

2.5.1 ModelHub

ModelHub [26] is similar to the overall ModelDB system in that it also aims to provide a suite of systems like model version control, a command line toolkit, and a model exploration API, to support the model building process. ModelHub focuses on deep learning and provides specialized tools for that domain, however, while ModelDB is aimed at machine learning models in general. Additionally, while ModelHub tracks changes made to models, it does not track transformations made to datasets like ModelDB S+C does. Finally, ModelHub does not describe any client libraries or abstractions that allow collecting data about the data scientist's operations as they are performed. ModelDB Spark Client, on the other hand, records a wide range of machine learning operations, like random splitting of datasets, annotating datasets, and creating preprocessing pipelines, as the user does them and stores them in ModelDB Server.

2.5.2 MLDB

MLDB [11] is a system that allows users to create and store machine learning models as well as datasets in a central database and access them through a REST API. While it supports the storage of models, it does not store operations (e.g. random splitting, pipeline creation) that the user performs. Additionally, there are no client libraries that can record these operations behind the scenes and send them to the server with minimal user involvement. Finally, MLDB does not offer an API for querying operations and gleaning information from them.

2.5.3 Azure ML

Microsoft's Azure Machine Learning [5] is a cloud service that allows users to create machine learning models with a web interface, store them in the cloud, and access them via web service. Azure Machine Learning requires users to construct a dataflow graph indicating their full pre-processing and model training operations before model building happens. ModelDB S+C, on the other hand, does not ask the user to indicate their full model building workflow beforehand, and will instead record the operations and models as they appear. ModelDB Server focuses on storing and analyzing the model building process and allows the user to use another library (Spark.ML in this thesis) for training models. Azure Machine Learning, on the other hand, requires that the user use its model training system. Finally ModelDB Spark Client works with a user's existing Spark.ML code, allowing them to log their operations and models with few code changes. Azure Machine Learning, on the other hand, requires users to convert significant parts of their code into dataflow graphs using its proprietary user interface.

2.5.4 Velox

Velox [10] is a system for low latency serving and management of machine learning models. While it does support storage of models, it does not store operations performed in model building and it does not expose APIs for analyzing the model building process.

2.5.5 MLBase

MLBase [24] is a database system with the ability to store and serve machine learning models. It includes an optimizer that figures out a plan for training a model. Unlike ModelDB S+C, it does not track and store the operations in the model building process. It also requires users to specify their model building process using a declarative language with hints for the optimizer. While this can be useful for non-experts, it reduces the control that a user has over their model building process. ModelDB S+C on the other hand, only requires a few minor changes to the user's Spark.ML code in order to store the data associated with the model building process.

2.5.6 PMML

PMML [15] is an XML-like markup language for representing machine learning models. While it can describe a wide range of machine learning models, it cannot represent general model building operations like ModelDB Server's database does. It is not possible to run queries on PMML either. ModelDB Server, on the other hand, exposes an API for querying model building operations and stores operation data in a SQL database. PMML can, however, complement ModelDB Server because a user can choose to store their serialized models in PMML form.

2.5.7 Longview

Longview is a predictive DBMS. It allows users to store machine learning models in a relational database and access them via a user defined function in their queries. Unlike ModelDB S+C, it cannot work with a user's existing code, and instead requires them to use Longview for all their model training. Finally, Longview does not store machine learning operations like ModelDB Server does.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

Abstractions

While Spark.ML and other machine learning libraries provide abstractions for various machine learning concepts like cost functions and optimizers, they do not have any abstractions that are suitable for representing the model building process.

Therefore, ModelDB S+C must develop its own set of abstractions that can represent the different operations for model building. These abstractions must be simple so that they are easily understood and not too difficult to implement. Since ModelDB Server aims to be library agnostic, they must be general enough to be applied to a number of different machine learning libraries. Finally, since it is impossible to anticipate all the model building operations that a user may be interested in, the abstractions must be composable and flexible so that new abstractions can be built from them that can represent other pieces of the model building process.

This chapter describes the abstractions in ModelDB S+C, building up from the simplest abstractions to the higher level ones.

ModelDB S+C stores data in a SQL database (currently, a SQLite database is used). Many of the abstractions below are represented in the SQL tables in Appendix A. Each abstraction will be discussed with a reference to its corresponding tables, if any. The tables are summarized in Table 3.1.

Table	Schema
DataFrame	numRows, dataSource
DataFrameColumn	dfId, name, type, columnIndex
Transformer	transformerType, filepath
Feature	name, featureIndex, importance, transformer
TransformerSpec	transformerType
HyperParameter	spec, paramName paramValue, paramMinValue, paramMaxValue
Event	eventType, eventId
TransformEvent	oldDf, newDf, transformer, inputColumns, outputColumns
FitEvent	transformerSpec, transformer, df, predictionColumns, labelColumns, problemType
MetricEvent	transformer, df, metricType, metricValue
CrossValidationEvent	df, spec
CrossValidationFold	metric, event
GridSearchCrossValidationEvent	best
GridCellCrossValidation	gridSearch, crossValidation
PipelineStage	pipelineFitEvent, transformOrFitEvent, isFit, stageNumber
PipelineStage (modified)	pipelineFitEvent, transformStage, fitStage, stageNumber
Annotation	posted
AnnotationFragment	annotation, fragmentIndex, transformer, dataFrame, spec, message
LinearModelTerm	model, termIndex, coefficient, tStat, stdErr, pValue
TreeNode	isLeaf, prediction, impurity, gain, splitIndex, rootNode
TreeLink	parent, child, isLeft
TreeModel	model, modelType
TreeModelComponent	model, componentIndex, componentWeight, rootNode

Table 3.1: Datasets used in evaluation

3.1 Primitives

ModelDB S+C is built on three primitives: the DataFrame, Transformer, and TransformerSpec.

3.1.1 DataFrame

A DataFrame represents a table of data. It has a number of named, typed columns and it stores a count of rows. The schema is given in Listing 1.

The DataFrame table stores the number of rows, which is useful for understanding the sizes of various DataFrames used in the model building process. It also indicates its data sources, which could be a CSV in the local filesystem, a JSON file stored on HDFS, or something else altogether. Since users may store data in a variety of formats and locations, a DataFrame cannot make any assumptions about what the data looks like or where it lives. The columns of a DataFrame are stored in another table called DataFrameColumn. Each DataFrame column simply points to its associated DataFrame and provides the name, type, and numerical index (e.g. 0 for first column, 1 for second column) of the column. Observe that ModelDB S+C does not place any restrictions on the allowed types in a DataFrame, because this can vary across machine learning libraries. One notable aspect of ModelDB S+C is that it only stores metadata about a DataFrame, rather than the actual rows. The reasoning here is that a user's dataset could be very large, and importing it into ModelDB S+C would be intractable. Additionally, storing the data in ModelDB S+C would limit the user's freedom to store data in the locations and file formats that best suit their application. However, by storing the dataSource column for each DataFrame, the user is given a pointer to the underlying data, which they can load with the tool of their choice.

3.1.2 Transformer

The DataFrame abstraction describes the data involved in the model building process. The Transformer abstraction, on the other hand, describes the operations that can be performed on that data. A Transformer is an object that takes an input DataFrame and produces an output DataFrame. Its tables are shown in Listing 2.

The Transformer table is quite simple, just storing a type (e.g. OneHotEncoder, LinearRegressionModel) and a path to the file containing the serialized Transformer. This is again done to maximize flexibility. ModelDB S+C does not impose any restrictions on the kinds of Transformers that can be stored, as different machine learning libraries may store different kinds of transformers. By storing the filepath to the serialized Transformer, the user is able to load the Transformer into their machine learning library of choice and use it to transform data or make predictions. Many machine learning libraries, including Spark.ML and Scikit-learn offer facilities for serializing and deserializing Transformers. Transformers include features, which are the columns they expect as inputs. Each feature has a name, index (in the feature vector), and importance. Notice that a Transformer does not have to be a machine learning model, like a logistic regression model. It can also be a data preprocessor, like a one-hot encoder. One reasonable criticism of the Transformer table is that its simplicity prevents the user from storing useful model parameters, like the weights of a linear regression model, that they may want to query. This is not an issue, however, because it is possible to create additional tables on top of Transformer which support this information (as in the case of LinearModel and TreeModel, which will be described later in this chapter).

3.1.3 TransformerSpec

The final primitive in ModelDB S+C is the TransformerSpec. Some Transformers, which ModelDB S+C calls models, are created by fitting a DataFrame where the fitting is guided by a set of hyperparameters. These hyperparameters are stored in a TransformerSpec, whose tables are shown in Listing 3.

The TransformerSpec table is very simple, it just indicates the kind of Transformer being specified. Each HyperParameter points to its associated TransformerSpec and indicates the name, type, value, and bounds for the hyperparameter. The bounds may be ignored for non-numerical hyperparameters. Notice that this allows expressing a wide range of hyperparameters such as the regularization parameter of a linear regression model, the number of trees in a random forest, or the optimization algorithm (e.g. stochastic gradient descent) used to train a logistic regression model.

It is worth noting that the Transformer table does not reference the TransformerSpec table and that the TransformerSpec table does not reference the Transformer table. This is for two reasons. First, some Transformers can be created without specifying any hyperparameters (e.g. a Transformer that removes all rows containing null values). Second, a user may want to create a TransformerSpec that lives independently of any Transformer. For example, suppose that a user has defined a set of hyperparameters to train a random forest model, and that they receive new datasets every week. They may choose to re-use the same TransformerSpec for each week of data. Consequently, the TransformerSpec is not tied to just one Transformer and can even be tied to zero Transformers when it is first created. With all this being said, there does exist a mechanism by which ModelDB S+C indicates that a Transformer was produced from a given TransformerSpec, and that is the FitEvent, which is introduced later in this chapter.

DataFrame, Transformer, and TransformerSpec are very simple abstractions, but are powerful in that they can express a huge variety of datasets, models/data preprocessors, and model training configurations. Equally important is the fact that they are also present, in some capacity, in many other machine learning libraries.

3.2 Syncable Events

ModelDB S+C is designed with the assumption that most of the interesting operations in model building can be represented as combinations of the three primitives described in the previous section. A specific operation that occurs in the user's model building process is called a Syncable Event ("Syncable" because it is shared between server and client, "Event" because it is a specific operation performed at a specific time). The following sections describe several of these Syncable Events. Each Syncable Event receives its own table (or few tables) in the database and Syncable Events can be composed together to create other events. All of the Syncable Events that ModelDB S+C has stored are referenced in the Event table, which is shown in Listing 4.

An Event includes a type (e.g. "fit", "transform") and the ID of the event in its corresponding table.

3.3 Core Events

The three foundational Syncable Events are TransformEvent, FitEvent, and MetricEvent.

3.3.1 TransformEvent

A TransformEvent represents the creation of a new DataFrame from an old DataFrame by applying a Transformer. This is a very general idea and can be used to represent both data preprocessing steps and prediction-making with a model.

The TransformEvent table has the schema shown in Listing 5.

The event indicates both the old and new DataFrames, the Transformer performing the transformation, and the input and output columns. The input and output columns could also be represented in their own tables, rather than as strings. This may be worth doing in the future because it would more easily enable queries such as finding the TransformEvent that produced a given column in a DataFrame.

To understand the flexibility of TransformEvent, it is worth considering a few sample operations that it can represent. First, consider one-hot encoding. One-hot encoding is a technique for converting categorical variables into binary vectors. If a categorical variable with k levels, then level l can be indicated with a vector with k entries where the l^{th} entry is 1 and the other entries are 0. To represent one-hot encoding of a column, "col", we can create a Transformer of type OneHotEncoder. Then, we can create a TransformEvent where the input column is marked as "col" and the output column is marked as "oneHotEncodedCol". Notice, that the old and new DataFrames in the TransformEvent only need to be logically, rather than physically, distinct. Therefore, the underlying machine learning library may actually use the same chunk of memory to represent data that is common to the two DataFrames. Consider another example - making a prediction with a model. In this case, the model can be the Transformer, the input columns are the feature names, and the output column can contain the prediction (and perhaps there can be another column that contains a confidence level in that prediction). As a final example, consider a transformation that removes rows from a DataFrame that contain null values. The "null remover" can be a Transformer, the input and output columns can be empty strings, and the old and new DataFrames can have identical columns, but a differing number of rows. Thus, TransformEvent is a very flexible abstraction that can represent a wide range of operations.

3.3.2 FitEvent

The second core Syncable Event is the FitEvent, which represents the fitting of a DataFrame with a TransformerSpec to produce a Transformer. It is given in Listing 6.

The FitEvent is the main abstraction that represents the building of a model. It indicates the data used to produce the model, the specification that guided training, and the produced model. It indicates the label columns that were used to teach the model, the columns in which the model will output its prediction, and the type of problem (e.g. regression, binary classification) that the model solves. Indeed, it is possible at this point to clearly define what a model is in ModelDB S+C. A model is a Transformer with an associated FitEvent. Notice that the feature columns are not included because they are already stored in the Feature table.

3.3.3 MetricEvent

Thus far, the chapter has explained how ModelDB S+C represents the operations that create data (TransformEvent) and operations that create models (FitEvent). Another potential output of an operation is a statistic, metric, or other number that describes or evaluates a dataset or model. For this purpose, ModelDB S+C includes a Syncable Event called MetricEvent. It is shown in Listing 7.

The representation is quite flexible, in that it allows metrics on datasets as well as metrics on models. For example, consider the problem of finding a the standard deviation of a column. This could be represented by a MetricEvent where metricType is "standard deviation", the metricValue is the actual numeric value for the standard deviation, the transformer is a StandardDeviationComputer (with one feature column - the column for which the standard deviation is being computed), and the df is the DataFrame with the column for which the standard deviation is being computed. As another example, consider the problem of evaluating a model's prediction accuracy. In this case, the transformer could be the model being evaluated, the df could be the DataFrame for which the model made predictions, the metricType could be "accuracy", and the metricValue could be the actual accuracy score.

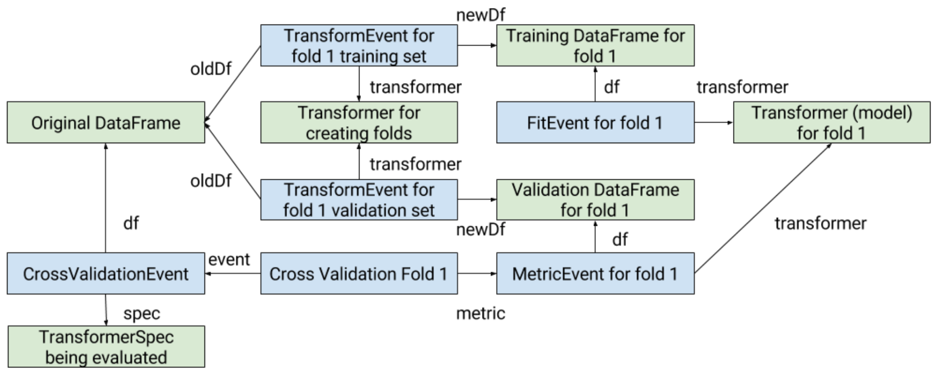
TransformEvent, FitEvent, and MetricEvent are the fundamental Syncable Events in ModelDB S+C. They build on ModelDB S+C's three primitives: DataFrame, Transformer, and TransformerSpec. While these Syncable Events are powerful on their own, they can also be combined together to represent higher level events.

3.4 Composite Events

ModelDB S+C includes a number of other events which combine the three core Syncable Events and the three primitives. For brevity, only the most interesting events will be discussed in this paper.

3.4.1 CrossValidationEvent

Cross validation is a model selection operation that is an integral part of the model building process. Recall that cross validation involves considering a hyperparameter configuration, breaking a DataFrame into pieces (folds), training a model for each fold (it is trained on all folds except one), and evaluating a model for each fold (it is evaluated on the fold that was left out of training). This operation is represented using the tables shown in Listing 8.



TransformEvent for fold 1 training set
newDf
Training DataFrame for fold 1
oldDf
transformer
df
transformer
Original DataFrame
Transformer for creating folds
FitEvent for fold 1
Transformer (model) for fold 1
oldDf
transformer
TransformEvent for fold 1 validation set
Validation DataFrame for fold 1
newDf
transformer
df
df
CrossValidationEvent
event
Cross Validation Fold 1
MetricEvent for fold 1
spec
metric
TransformerSpec being evaluated

Figure 3-1: Diagrammatic representation of cross validation in ModelDB S+C. Green boxes represent primitives and blue boxes represent Syncable Events. Arrows represent foreign key relationships.

Two simple tables are all that is needed to allow ModelDB S+C to store cross validation events. They are deceptively simple because ModelDB S+C leverages the three primitives and three core Syncable Events heavily when representing cross validation. To illustrate, consider the diagrammatic representation of cross validation in Figure 3-1 (only one fold has been shown to avoid making the diagram too large).

To understand Figure 3-1, it is best to start at the CrossValidationEvent. The CrossValidationEvent points to the TransformerSpec (i.e. hyperparameter configuration) that is being evaluated as well as to the full DataFrame that is being used to train models. There is one CrossValidationFold that points to the CrossValidationEvent. The fold points to the MetricEvent that contains its evaluation metric. The MetricEvent points to the model (Transformer) that was trained when the fold was excluded and to the validation DataFrame on which the metric was computed. The validation DataFrame was produced from the original DataFrame using a special Transformer that creates folds. The model for the fold was created via a FitEvent that points to the training DataFrame for the fold. The training DataFrame for the fold was created from the original DataFrame using the same special Transformer that creates folds.

Thus, the complex process of cross validation can be represented with ModelDB S+C by adding two simple tables and cleverly combining the primitives and core Syncable Events.

3.4.2 GridSearchCrossValidationEvent

The cross validation operation is often combined with a grid search, in which several hyperparameter configurations are evaluated and cross validation is performed for each one. Then, the best hyperparameter configuration (i.e. the one with the largest average evaluation metric) is selected and used to train a model on the entire dataset. This is a complex process, but can be easily supported in ModelDB S+C by adding the tables shown in Listing 9.

The GridSearchCrossValidationEvent table simply stores a reference to a FitEvent that produced the final model (a Transformer) by using the best TransformerSpec to train on the entire original DataFrame. Then, GridCellCrossValidations are created (one for each hyperparameter configuration that was considered) and each points to the CrossValidationEvent for its corresponding hyperparameter configuration.

Looking back, the prefix "GridSearch" is actually a misnomer, because there is nothing about the above abstraction that requires a grid search to be performed. The abstraction above can represent grid search, random search, and various other model selection strategies that consider a number of hyperparameter configurations.

3.4.3 PipelineEvent

One common practice in Spark.ML, Python Scikit-learn, and perhaps other machine learning libraries is that of building a preprocessing pipeline. For example, consider the simple preprocessing pipeline in Figure 3-2. This pipeline assumes that the machine learning library includes Transformer objects that produce an output DataFrame from an input DataFrame and Estimator objects that apply a TransformerSpec on a DataFrame to create a Transformer. Spark.ML and Python Scikit-learn both include Transformers and Estimators. The user arranges Estimators and Transformers into a chain, called a Pipeline, feeds in a DataFrame, and the result is a chain of Transformers. This chain of Transformers, or Pipeline Model, can be used in other parts of the program. Notice that if the final piece of the Pipeline is an Estimator that produces a model, then the user may be able to represent their entire machine

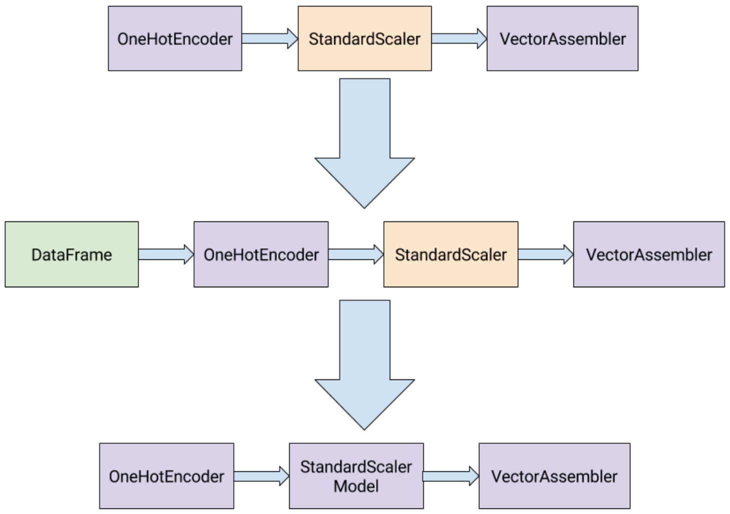


Figure 3-2: Creation of a preprocessing pipeline. First, the user chains together some Transformers and Estimators (objects that can apply their TransformerSpec to create a Transformer from a DataFrame). Next, the user feeds a DataFrame into the pipeline. The Transformers transform their input DataFrame, produce an output DataFrame, and pass the output on to the next step in the pipeline. The Estimators use the input DataFrame to create a Transformer, and then they replace themselves with this Transformer - the Transformer then transforms the input DataFrame and passes the output to the next step.

learning system using a Pipeline Model. They can also add post-processing steps by adding more Transformers at the end.

ModelDB S+C represents the creation of Pipelines using the table in Listing 10.

First, ModelDB S+C represents the creation of a Pipeline Model (which is a Transformer) from a DataFrame using a FitEvent. The TransformerSpec associated with this FitEvent can have an empty set of HyperParameters. Then, the pipeline creation is broken into "stages". A stage is either a "transform stage" or a "fit stage". A transform stage is a TransformEvent where a Transformer in the pipeline transforms its input and forwards the output to the next step. A fit stage is when an Estimator creates a Transformer by applying its TransformerSpec to its input DataFrame. Notice that each Estimator in the user's pipeline produces one fit stage and one transform stage. The stages are ordered using the stageNumber, which should be higher for later stages.

The transformOrFitEvent and isFit fields are used to distinguish the stage as a transform stage or fit stage. Admittedly, this is a bit inelegant. A cleaner solution would be the table shown in Listing 11.

In this case, every Estimator and every Transformer in the Pipeline would produce a PipelineStage. For the Estimators, the fitStage would point to their creation of a Transformer by applying their TransformerSpec on their input DataFrame. For the Estimators, the transformStage would point to the transformation of their input DataFrame into an output DataFrame using the Transformer that was created. For a Transformer, the transformStage would be its application to the input DataFrame to produce an output DataFrame and its fitStage would be NULL.

3.4.4 Annotation

The above events demonstrate how ModelDB S+C is able to express complex model building operations by cleverly combining its primitives and core events. There are more such events (e.g. RandomSplitEvent) in ModelDB S+C which are similar in nature, but which will not be discussed here for brevity.

Another composite event of interest, however, is the Annotation, which does not simply represent an existing model building operation, but actually enables new functionality. ModelDB Spark Client allows the user to make notes for themselves and link these notes to DataFrames, Transformers, and TransformerSpecs. This can be done with Scala code like the following:

```
ModelDbSyncer.get.annotate(
  "I'm getting poor performance on",
  testDf1,
  "with model",
  model1,
  "Is there a bug with this model? Should investigate tomorrow."
)
```

The Scala code sample above stores an Annotation in ModelDB S+C, which involves the tables in Listing 12.

An Annotation indicates the time it was created and is associated with a number of AnnotationFragments. The AnnotationFragments are ordered using the fragmentIndex column. A fragment can either contain text, a reference to a Transformer, a reference to a DataFrame, or a reference to a TransformerSpec (in Spark.ML code, this would correspond to an Estimator). The Scala code above would result in the creation of five AnnotationFragments, where the first, third, and fifth store text, the second refers to a DataFrame, and the fourth refers to a Transformer.

In hindsight, Annotation should be called AnnotationEvent in order to make it consistent with the "Event" suffix associated with other Syncable Events.

3.5 Linear and Tree Models

While ModelDB S+C does allow the user to serialize their models to a filesystem, the description of the abstractions so far does not allow querying of the models because the Transformer table is so simple. To make querying possible, ModelDB S+C includes additional tables that build off the Transformer table to support storage (and querying) of additional model data. Specifically, ModelDB S+C has tables for linear models and tree models, which are designed to support the logistic regression, linear regression, decision tree, gradient boosted tree, and random forest models in Spark.ML.

3.5.1 Linear Models

Linear models are defined by their vector of weights. These weight vectors are stored with the table in Listing 13.

A linear model is represented by a Transformer, and its weights are represented by the associated LinearModelTerms. Each term indicates the weight (the coefficient) and some statistics about the weight (t-statistic, standard error, and p-value). The position of the weight in the weight vector is given by termIndex (which is 0 for an intercept term).

This table makes it possible to run some useful queries (e.g. build confidence intervals aroudn the weights) and also makes it possible to develop tools to reconstruct linear models based on the weights and the transformerType.

The actual implementation of ModelDB S+C includes a table called LinearModel, but this is there for legacy purposes and is not actually needed (its rmse, r2, and explainedVariance columns can be stored in MetricEvents rather than as columns).

3.5.2 Tree Models

A decision tree consists of nodes and edges (or links) between them. ModelDB S+C represents this using the the tables in Listing 14.

Nodes indicate whether they are leaves. Leaf nodes indicate their prediction, impurity, and the root node of the tree (unless they are the root). Internal nodes indicate their gain, impurity, the feature they are splitting, and their root (unless they are the root). This table could be augmented so that internal nodes also store the splitting criterion (i.e. how to decide which child to forward an input example to), but this is not utilized in ModelDB S+C so it is currently omitted.

The TreeLink table simply indicates a parent child relationship. Spark.ML's decision trees are binary trees, so ModelDB S+C also assumes decision trees are binary. However, for generalization to non-binary trees, the isLeft column could be replaced by a childIndex column that indicates the index (0 being leftmost) of the child.

Random forests and gradient boosted trees are ensembles of decision trees, where each tree is given some weight. In fact, a decision tree can be thought of as an ensemble consisting of a single tree that gets all the weight. With this in mind, ModelDB S+C defines the tables in Listing 15.

A TreeModel simply indicates the associated Transformer and the type of the model. The model is associated with a number of trees through the TreeModelComponent, which indicates the tree's numerical order in the overall ensemble (componentIndex), the tree's weight, and the root node of the tree. A decision tree can be thought of as TreeModel with a single TreeModelComponent.

While the above tables are designed for tree ensembles specifically, they could be extended to support ensemble models in general.

3.6 ModelDB Syncer

ModelDB Syncer is a client side abstraction for recording events and sending them to ModelDB Server. When operations (e.g. creation of a model, splitting of a DataFrame) are detected by the ModelDB Spark Client, a SyncableEvent object is created and sent to a global ModelDB Syncer object. This ModelDB Syncer maintains an ordered buffer of Syncable Events that it periodically flushes to ModelDB Server. Each Syncable Event is imbued with logic for converting the Spark objects it represents into the corresponding ModelDB S+C abstraction and sending that information to ModelDB Server. The ModelDB Syncer performs other roles as well, such as maintaining a mapping between Spark objects and their IDs in ModelDB S+C. ModelDB Syncer is described in greater detail in the Implementation chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

Algorithms

The abstractions discussed in the previous chapter are stored in SQL tables in ModelDB Server's database. This database is useful on its own because the user can query it or build applications on top of it. Nevertheless, ModelDB Server provides a number of algorithms that can be run on the database to glean useful information about the model building process.

This chapter covers some of the algorithms used in ModelDB Server.

The first section discusses storage algorithms, which describe how operations are stored in the database while respecting the constraints between events and primitives. It begins by discussing the storage algorithms for the three primitives and core Syncable Events. Then, the storage algorithms for some of the more complicated Syncable Events, such as RandomSplitEvent, GridSearchCrossValidationEvent, and PipelineEvent are discussed. Finally, the storage algorithms for linear and tree models are covered.

The second section covers algorithms that operate on the DataFrame ancestry forest, which consists of the DataFrames and the links between them, as reflected in the TransformEvents. The section explains how the forest can be used to find the ancestry of a given DataFrame or model, the common ancestor of two DataFrames, and the models that derived from a given DataFrame. The section concludes by showing how the forest can be used to extract machine learning pipelines from ModelDB Server.

The third section focuses on algorithms that operate on the feature sets of models. It first discusses how it is possible to find the original DataFrame columns that were used to generate the features used by a given model. Next, it explains how to find models that use all the features in a given feature set.

The fourth section discusses algorithms that operate on multiple models. It begins by discussing how the feature sets of two models can be compared. Next, it explains how the DataFrame ancestry forest can be used to find a common DataFrame from which two models are derived. Afterwards, it shows how the hyperparameters of two models can be compared. The section concludes by explaining how models can be ranked and how ModelDB S+C can find models similar to a given model.

The fifth section focuses on algorithms for linear and tree models. It explains how features can be ordered by importance, how models can be compared by convergence time, and how feature importance can be compared between models.

The final section briefly mentions some other algorithms that ModelDB S+C provides, but which, for brevity, are not covered at length in this thesis document.

4.1 Storage Algorithms

Before discussing the algorithms that glean information about model building, it is important to first consider the algorithms that store data in ModelDB Server's database. These algorithms are straightforward, but it is illustrative to see how they store data and what tables they affect.

4.1.1 DataFrame, Transformer, and TransformerSpec

To begin, consider the algorithms for storing the three primitives: DataFrame, Transformer, and TransformerSpec.

When ModelDB Server is asked to store a primitive, it first checks if there is already a primitive in the database with the same ID. If not, then ModelDB Server stores the primitive in the corresponding table. Next, it stores any auxiliary data (e.g. DataFrameColumns for a DataFrame, HyperParameters for a TransformerSpec) associated with the primitive.

4.1.2 TransformEvent, FitEvent, MetricEvent

The core Syncable Events are TransformEvent, FitEvent, and MetricEvent. They are defined as compositions of the three primitives. Similarly, the algorithms that store the core Syncable Events to the database utilize the storage algorithms for the primitives.

Consider the TransformEvent. When ModelDB Server is asked to store a TransformEvent, it first stores the input DataFrame, output DataFrame, and Transforer using the corresponding storage algorithm for the primitive. Then, it performs some processing specific to TransformEvent (e.g. sorting and de-duplicating the input and output columns). Next, it stores an entry in TransformEvent. Finally, it stores an entry in the Event table.

FitEvent and MetricEvent have storage algorithms that are similar in structure to TransformEvent's storage algorithm.

4.1.3 Annotation

In order to store an Annotation, the client sends ModelDB Server a list containing the annotation fragments (each is a Transformer, TransformerSpec, DataFrame, or a String). ModelDB Server first stores the primitives that the fragments refer to. Then, it stores an entry in Annotation. Next, it stores an AnnotationFragment for each fragment. Finally, it makes an entry in the Event table. ModelDB Server also verifies that each fragment refers to exactly one primitive or contain a String. This check could also potentially be done at the database level.

4.1.4 RandomSplitEvent

This is a Syncable Event which represents the random splitting of a DataFrame to create smaller DataFrames according to a weight vector. For example, doing a random split of a 100-row DataFrame with weights of 0.7 and 0.3 would produce DataFrames with row-counts of 70 and 30.

To store a RandomSplitEvent, ModelDB Server first makes an entry in a table called RandomSplitEvent and an entry in Event. Next, it stores the input DataFrame as well as all the output DataFrames. Then, it indicates the weight of each output DataFrame in a table called DataFrameSplit. Finally, it stores TransformEvents to indicate that the output DataFrames were derived from the input DataFrame (it uses a synthetic Transformer called "RandomSplitTransformer" to reflect this operation).

Note that RandomSplitEvent demonstrates how to overcome TransformEvent's single-input, single-output limitation. A RandomSplitEvent has a single input and multiple outputs.

4.1.5 PipelineEvent

Recall that PipelineEvent represents the creation of a Pipeline Model from a Pipeline. A Pipeline is a chain of Transformers and Estimators (i.e. an object that applies a TransformerSpec to a DataFrame in order to create a Transformer). When a DataFrame is fed into the Pipeline, a Pipeline Model (a chain of Transformers) is produced. The storage algorithm is shown below:

```
Data: A PipelineEvent, denoted $pe$
Store $pe.pipelineFitEvent$, which stores the actual FitEvent that created the
  PipelineModel from the input DataFrame using a dummy Pipeline
  TransformerSpec.
Sort $pe.transformStages$ and $pe.fitStages$ by stageNumber.
Apply the merge procedure (i.e. from merge sort) to order all the stages by
  increasing stageNumber. Also require that a fit stage precede a transform
  stage if they have the same stageNumber.
$currentDataFrame \leftarrow pe.inputDataFrame$
for each stage, taken in increasing stageNumber do
    if stage is a fit stage then
        Store a FitEvent for the stage where $currentDataFrame$ is the the
          input DataFrame for the FitEvent.
        $currentDataFrame \leftarrow stage.inputDataFrame$
    else
        Store a TransformEvent for the stage where $currentDataFrame$ is the
          input DataFrame for the TransformEvent.
        $currentDataFrame \leftarrow stage.outputDataFrame$
    end
end
for $fitStage$ in $pe.fitStages$ do
    Store an entry in the PipelineStage table.
end
for $transformStage$ in $pe.transformStages$ do
    Store an entry in the PipelineStage table.
end
```

Algorithm 1: Storage of a PipelineEvent

The key challenge in the PipelineEvent algorithm is enforcing the constraint that the output of one pipeline stage is the input to the next pipeline stage. This is done by processing the stages in increasing stageNumber (with fit stages put before transform stages if they have equal stageNumbers). The *currentDataFrame* variable marks the input into the next pipeline stage.

4.1.6 CrossValidationEvent

Recall that a CrossValidationEvent corresponds to a single hyperparameter configuration.

To store a CrossValidationEvent, ModelDB Server first stores the input DataFrame and the TransformerSpec (i.e. the hyperparameter configuration under consideration). Then, it stores an entry in the CrossValidationEvent table and an entry in the Event table. Next, ModelDB Server iterates through each fold in cross validation and, for each fold, it stores a FitEvent, MetricEvent, and an entry in CrossValidationFold. ModelDB Server also enforces that the same input DataFrame is used for all the folds.

4.1.7 GridSearchCrossValidationEvent

Recall that grid search cross validation considers multiple hyperparameter configurations and performs cross validation event for each one. GridSearchCrossValidationEvent's storage algorithm is the following. First, a FitEvent is stored that represents the creation of the final model over all the data. Next, an entry is made into the GridSearchCrossValidationEvent table and an entry is made into the Event table. Then, all the DataFrames used for validation and all the DataFrames used for training are stored, and a TransformEvent is logged for each to indicate that they originated from the original input DataFrame. Next, the CrossValidationEvent storage algorithm is run for each cross validation that was performed. ModelDB Server includes some additional logic to ensure that the validation DataFrames and training DataFrames are shared across the different cross validations.

4.1.8 LinearModel

Recall that ModelDB S+C can augment the Transformer table with tables to store the weights of a linear model. To store a linear model, ModelDB Server requires that the user indicate (via ID) an existing Transformer to augment. Next, ModelDB Server makes an entry in the LinearModel table for the linear model. It then stores a LinearModelTerm for each coefficient of the linear model. Next, it updates the Feature rows associated with the Transformer with the feature importance scores computed by the linear model. Finally, ModelDB Server logs the history of the objective function used for training into a table called ObjectiveFunctionHistory.

4.1.9 TreeModel

Recall that ModelDB S+C can augment the Transformer table with tables to store ensembles of trees. In addition, recall that a decision tree model can be thought of as an ensemble of one tree.

The storage algorithm for a tree model goes as follows. First, ModelDB Server requires the user to indicate (via ID) an existing Transformer to augment. ModelDB makes an entry in the TreeModel table for the tree model. Then, for each tree in the model, ModelDB Server stores an entry in the TreeModelComponent table, then does a breadth first search starting at the root for the tree, and for each encountered node, it stores an entry in TreeNode and (if the node is not the root node) an entry in TreeLink. Finally, after storing all the trees in the tree model, ModelDB Server updates the Feature rows of the associated Transformer to indicate their feature importance scores as computed by the tree model.

4.2 Ancestry Algorithms

Recall that a TransformEvent indicates an input DataFrame and an output DataFrame. If each DataFrame is a node, and there is a directed edge from input DataFrame to output DataFrame, then the TransformEvent table defines a forest over all the DataFrames. This forest is hereafter referred to as the "DataFrame Ancestry Forest", or $\mathcal{F}$. Suppose that each DataFrame, df, in $\mathcal{F}$ has a field $df.parent$ that points to the DataFrame that created it and suppose that it also has a field $df.children$ that points to a list of the DataFrames that it produced.

There are a number of interesting algorithms that can be run on $\mathcal{F}$.

4.2.1 Ancestry of a DataFrame

The simplest algorithm is to find the ancestry of a DataFrame. That is, given a DataFrame *df* in $\mathcal{F}$, we follow parent pointers all the way up to a root DataFrame (i.e. a DataFrame with an empty parent pointer). The reverse of this path is the ancestry of a given DataFrame. This operation allows a data scientist to see how a particular DataFrame was created and identify where potential bugs in the data may have been introduced.

4.2.2 Ancestry of a Model

It is also possible to find the ancestry of a model. Recall that a model is simply a Transformer with an associated FitEvent. Finding the ancestry of a model involves looking up the DataFrame in its associated FitEvent and finding the ancestry of that DataFrame. This allows the user to see what DataFrames contributed towards the data used to train a model.

4.2.3 Common Ancestor of Two DataFrames

Given two DataFrames $df1$ and $df2$ in $\mathcal{F}$, it is possible to find the common ancestor DataFrame (if there is one) that led to their creation. This can be useful in finding commonalities between intermediate DataFrames (e.g. $df1$ is producing good results and $df2$ is producing bad results - where in their lineage did they start to differ?). Finding the common ancestor is implemented by simplying finding the ancestry of $df1$ and the ancestry of $df2$, and finding youngest DataFrame common to both the ancestries.

4.2.4 Descendent Models of DataFrame

In addition to following parent pointers in $\mathcal{F}$, it is also useful to follow child pointers. A data scientist may be interested in determining all the models that used data that originated in a DataFrame df. To compute this, a breadth-first search is performed starting at *df* and following child pointers. All the DataFrames encountered in the search are added to a list $descendentDfs$. Then, a scan is made of the FitEvent table to find all Transformers that were created by fitting a DataFrame in $descendentDfs$. These are the models that descended from *df*.

4.2.5 Pipeline Extraction

The user may explicitly create Pipeline Models via ModelDB S+C's PipelineEvent. However, since Pipeline Models are simply chains of Transformers (some of which were produced by fitting a TransformerSpec to a DataFrame), the user implictly creates Pipeline Models as they make FitEvents and TransformEvents. ModelDB Server offers a mechanism for extracting pipelines from $\mathcal{F}$. The algorithm for extracting a pipeline is specified below.

First the user specifies the ID, mId, of a model (i.e. Transformer with associated FitEvent) that they'd like to extract a pipeline for.

Second, the FitEvent table is scanned to find the DataFrame *df* that is associated with the Transformer with ID mId.

Third, the ancestry of *df* is computed, and all the TransformEvents along the chain are noted down in a list (call it $transformEvents$).

Fourth, the Transformer for each TransformEvent in $transformEvents$ is extracted and put into a list (call it $transformers$).

Fifth, the FitEvent table is scanned and the TransformerSpec is extracted for all the FitEvents who have a Transformer appearing in $transformers$ (call the resulting list of TransformerSpecs *specs*).

Finally, $transformers$ and *specs* are returned to the user, sorted by the position of their corresponding TransformEvent in the ancestry chain.

4.3 Feature Algorithms

The Feature table lists the features that are used by a model (i.e. Transformer with associated FitEvent). There are a number of useful operations we can perform using this table.

4.3.1 Original Feature Set of Model

Suppose that there is a DataFrame *df*1 with the columns $labScore$ and $homeworkScore$. Then, suppose that *df*1 is transformed to produce *df*2, which contains a column called $assignmentScore$ that is computed by combining $labScore$ and $homeworkScore$ in some way. Finally, suppose that a model, m, is created to predict a new column, called $finalExamScore$, based on the value of $assignmentScore$.

In the above example, m has a single feature column, $assignmentScore$. However, the data scientist may find it useful to find the original features that produced the $assignmentScore$ column (i.e. $labScore$ and $homeworkScore$).

It is possible to use the Feature table and the DataFrame ancestry forest ($\mathcal{F}$) to compute the original features. The algorithm for doing so is presented below.

First, the user must specify the ID (call it mId) of a model.

Second, the corresponding FitEvent for model mId is found, and its DataFrame (call it *df*) is extracted.

Third, a set called $originalFeatures$ is initialized to contain the features (from the Features table) of model mId.

Fourth, the ancestry of *df* is computed, with the TransformEvents noted down in the list $transformEvents$.

Fifth, the $transformEvents$ list is scanned such that the TransformEvent that directly produced *df* is first and the TransformEvent that operated on a root DataFrame is last. For each such TransformEvent $transformEvent$, the following steps are performed:

1. If none of the columns listed in $originalFeatures$ appears in $transformEvent.outputColumns$, then $transformEvent$ is skipped and the next TransformEvent is considered.

2. Otherwise, all the columns in $originalFeatures$ that are also

in $transformEvent.outputColumns$ are removed from $originalFeatures$.

3. All columns in $transformEvent.inputColumns$ are added to $originalFeatures$.

Finally, $originalFeatures$ is returned.

The above algorithm simply starts with the model's described feature columns and walks up the ancestry chain trying to find the columns that originated these columns.

4.3.2 Models using a Given Feature Set

ModelDB Server can also find models that use any of the features listed in a given set of features, $featureSet$. Basically, the Feature table is scanned for rows whose feature name is included in $featureSet$. The Transformer IDs for these Feature rows are extracted, de-duplicated, and returned to the user.

Note that a data scientist may find it useful to find all the models that indirectly OR directly use any features in a given feature set. The above algorithm handles the "directly" case, but a variation of the Original Features algorithm would be needed to handle the "indirectly" case. This has not been implemented, but may be a useful algorithm to implement in the future.

4.4 Multi-model Algorithms

A data scientist creates multiple models in the model building process, and therefore may want to compare models in some way to help select the best ones. The algorithms described in this section offer some useful ways of doing this.

4.4.1 Compare Features of Two Models

One simple operation is to find the features shared by two models. Given the IDs of two models, $mId1$ and $mId2$, ModelDB Server finds their corresponding features (by scanning the Feature table) and returns the features they have in common, as well as the features that appear in only one of them.

While it has not been implemented, it would not be hard to utilize the Original Features algorithm here so that the original, rather than direct, features of the models are compared.

4.4.2 Common Ancestor DataFrame of Two Models

Identifying the DataFrame from which two models are derived can be useful in understanding their commonalities. Given the IDs, call them $mId1$ and $mid2$, of two models, ModelDB Server finds the common ancestor DataFrame as follows.

First, the FitEvent table is scanned to yield the DataFrames $df1$ and $df2$ that produced the models with IDs $mId1$ and $mId2$.

Then, the common ancestor DataFrame (using the algorithm described earlier in this chapter) of $df1$ and $df2$ is computed and returned.

4.4.3 Compare Hyperparameters of Two Models

Models of the same type (e.g. random forest) can have wildly different performance outcomes on the same training DataFrame because they have different hyperparameters. ModelDB Server makes it possible to compare the hyperparameters of two models given their IDs (call them $mId1$ and $mId2$).

First, the FitEvents for $mId1$ and $mid2$ are read and their TransformerSpecs $spec1$ and $spec2$, respectively, are extracted.

Second, the hyperparameters for $spec1$ and $spec2$ are read from the HyperParameter table and put into lists $hyperparameters1$ and $hyperparameters2$, respectively,

Third, hyperparameters in $hyperparameters1$ and $hyperparameters2$ with the same name are extracted and put into a list called $commonHyperparameters$. Hyperparameters with names unique to just one of lists are put into lists called

$model1Hyperparameters$ and $model2Hyperparameters$.

Finally, the three resulting hyperparameter lists are returned to the user.

4.4.4 Rank Models

Sometimes, a user may want to find the model that performs best in a given set of models. Since ModelDB Server stores MetricEvents for these models, it can sort the models according to a given metric type (e.g. accuracy) when evaluated on a given DataFrame.

4.4.5 Find Similar Models

A user may want to find other models similar to a given model. ModelDB Server allows the user to search for models that match the given model's type (e.g. Random Forest), problem type (e.g. regression, binary classification), performance, and more.

4.5 Linear Model and Tree Model Algorithms

ModelDB S+C stores extra data about linear and tree models, and this section summarizes a few of the algorithms that can be run on these models.

4.5.1 Features ordered by Importance

One operation of great value in model building is computing the most important features in a model. ModelDB Server stores an importance column in the Feature table for its linear and tree models. Thus, it simply sorts the features in descending importance and returns them to the user.

For linear models, ModelDB Server only computes feature importance if the linear model is standardized (i.e. all feature columns were scaled to have zero mean and unit variance). In this case, the feature importance is taken to be the absolute value of the weight associated with the feature.

For tree models, ModelDB Server defers to Spark.ML's implementation of feature importance. Thus, the Spark Client uses Spark.ML's algorithm to compute feature importances and then notifies ModelDB Server. However, since ModelDB

Server stores the entire tree model in its database, this algorithm could also just be implemented in ModelDB Server directly.

4.5.2 Iterations Until Convergence

ModelDB Server has a table called ObjectiveFunctionHistory, which stores, for a given model, the value of the objective function over several iterations of training. With this table in hand, it is easy for ModelDB Server to simply return the number of iterations taken for a model to converge (convergence occurs when the value of the objective function changes by less than ϵ in a single iteration).

4.5.3 Compare Feature Importances

ModelDB Server also supports comparing the importance of features between two models. First, it gets the list of features for each model, ordered by importance. Using the order, it then computes the percentile ranking of the feature's importance. Then, ModelDB Server finds features common to both features and returns the percentile ranking of the feature in each model. It also returns the percentile rankings for the features that appear in only one of the models.

4.6 Other Algorithms

There are a number other algorithms that ModelDB Server supports (e.g. building confidence intervals for linear model coefficients) and even more that can be implemented without much code (e.g. for a given model, see if there is another model that has scored higher than it for every MetricEvent). For brevity, however, these algorithms will not be discussed.

Chapter 5

Implementation

This chapter discusses some important aspects of ModelDB S+C's implementation.

5.1 System Architecture

The overall architecture for ModelDB S+C is shown below in Figure 5-1.

In the figure, the operations occur and the models first appear on the Spark worker nodes. These nodes send their data to the Spark driver node. The driver node runs the Spark.ML library, which stores objects representing the models and which has functions to trigger the operations. The ModelDB Spark Client sits on top of the Spark.ML library, and receives the operations and model data. The Spark Client runs an Apache Thrift client, which it uses to send data to ModelDB Server. ModelDB Server receives the data, performs the appropriate computations, and stores it in the SQLite Database. Notice that ModelDB Server does not speak directly to the model filesystem, which contains the serialized model files. Instead, the Spark Client speaks directly to the model filesystem. The reasoning for this decision will be discussed later in this chapter.

Recall that ModelDB Server also exposes an API for gleaning information about the model building process. ModelDB Spark Client includes convenience functions that the user can call to get this information. In this case, the arrows would be reversed. When the Spark Client makes a request (via Apache Thrift) to ModelDB

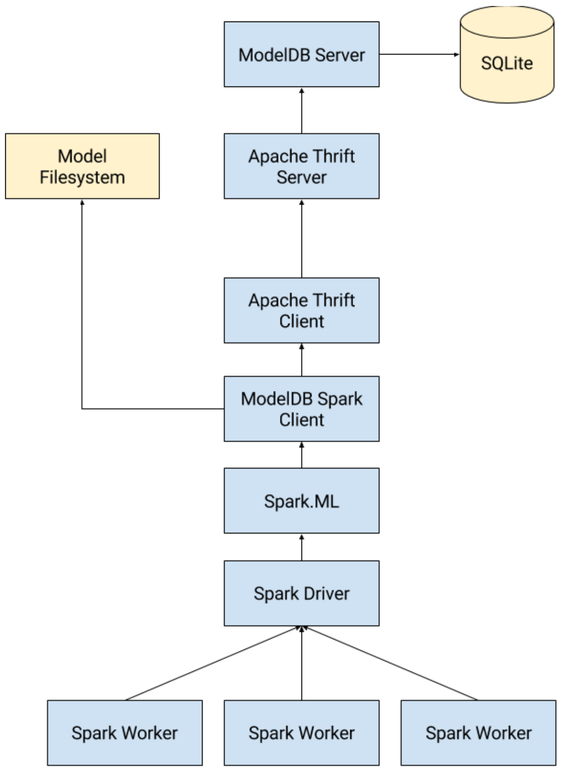


Figure 5-1: The overall architecture of ModelDB S+C. The arrows indicates the flow of operation + model data. Notice that the data originates at the Spark workers and finds its way into the SQLite database or model filesystem.

Server to run an API method, ModelDB Server reads the database, performs the appropriate computations, and responds to the Spark Client. The Spark Client then extracts the desired information from the response, and returns it to be used in the user's driver node program.

The two key pieces of this diagram are the ModelDB Server and Spark Client. Both will be discussed below.

5.2 Server

While there is a good deal of code (written in Java) in ModelDB Server, this section will focus only on interesting pieces of implementation that were not described in the previous chapters.

5.2.1 Database

Currently, ModelDB Server stores its data in a SQLite database. However, it can be configured to use a number of other SQL databases, like PostgreSQL. This is primarily achieved by using the JOOQ library. This library asks the user to provide a configuration file that specifies the database and runs a code generator to create Java classes that ModelDB Server can use to interact with the database. Changing this configuration file (and making appropriate changes to the database schema, in order to correct for slight differences in the SQL language) can allow ModelDB Server to use a different SQL database than SQLite. In addition, ModelDB Server is designed to avoid using any esoteric SQLite data types, functions, or features. Some problems which could be solved by appealing to a special database function are instead solved in ModelDB Server. This places minimal requirements on the SQL database that ModelDB Server uses.

5.2.2 Model Filesystem

ModelDB Server allows the user to store their serialized model files in a filesystem. However, as shown in Figure 5-1, the ModelDB Server does not ever communicate with this filesystem. Instead, the flow is as follows.

First, the user calls a function in ModelDB Spark Client to indicate that they'd like to save a model and optionally provides a filename that they would like to use.

Second, the Spark Client indicates to the ModelDB Server that it would like to store the given model under the given filename.

Third, ModelDB Server generates a filepath (incorporating the user's requested filename, if one is provied) for the model, updates the model's entry in Transformer table to reflect this filepath, and sends the filepath to the Spark Client.

Finally, the Spark Client serializes the model and writes the result to the designated filepath.

There are a number of advantages that the above approach has over an approach in which the client sends the serialized model to the server and the server stores the model in the filesystem. First, the serialized model file may be large, so storing it directly to the filesystem is cheaper than first sending it to the server and having the server store the model. Second, some machine learning libraries (e.g. Spark.ML) store models in a directory structure rather than as a file, and having the client store the model directly to the given filepath makes this possible. Finally, by pushing the storage logic to the client, the client could allow the user to provide logic that stores a serialized model to a brand new or esoteric filesystem (perhaps their own experimental filesystem).

5.2.3 Configuration

All the configuration for ModelDB Server (e.g. port to launch server on, prefix for filepath generation) is defined in a single configuration file, and some sensible defaults are provided so that ModelDB Server is usable right out of the box rather than requiring configuration before use.

5.2.4 Stateless, Separated, Logic

ModelDB Server exposes all its functionality to the client as Thrift endpoints. The client can make remote procedure calls to this endpoint to store operations, store models, compute ancestries, and more. The actual implementation of these Thrift endpoints are extremely short, most of them being just a single function call. This is done for three reasons. First, it pushes all of ModelDB Server's logic into other classes, called Data Access Objects (DAOs), for which unit tests can be easily written without having to worry about complexities that come with using Apache Thrift. Second, it allows ModelDB Server to be used as a library, so that other programs could import and use it without having to launch a Thrift server. Finally, it makes the server stateless, which can be useful if future work adds support to scale ModelDB Server to multiple machines.

The ModelDB Server algorithms described in Chapter 4 are each implemented as a single function or group of functions. They do not persist state between executions and they are expected to have all their dependencies injected as arguments. The abstractions described in Chapter 3 are implemented as SQL tables and also have corresponding Thrift structures to allow them to be sent and received via Thrift calls.

5.3 Spark Client

The Spark Client is written in Scala, primarily because Spark is written in Scala. Before discussing the implementation details, it is worth seeing a sample usage of the Spark Client.

5.3.1 Sample Usage

Consider the following sample code, written without ModelDB Spark Client, in which a model is trained and evaluated.

```
val Array(train, test) = data.randomSplit(Array(0.7, 0.3))
val lr = new LogisticRegression()
      .setMaxIter(20)
      .setLabelCol(FeatureVectorizer.indexed(labelCol))
      .setPredictionCol(predictionCol)
      .setFeaturesCol(featuresCol)
val ovr = new OneVsRest()
      .setClassifier(lr)
      .setLabelCol(FeatureVectorizer.indexed(labelCol))
      .setPredictionCol(predictionCol)
      .setFeaturesCol(featuresCol)
val model = ovr.fit(train)
val predictions = model.transform(test)
val eval = new MulticlassClassificationEvaluator()
      .setLabelCol(FeatureVectorizer.indexed(labelCol))
      .setPredictionCol(predictionCol)
      .setMetricName("f1")
val score = eval.evaluate(predictions, model)
```

Below, consider the same code WITH ModelDB Spark Client. The changed or new lines are commented.

```
// New line.
import edu.mit.csail.db.ml.modeldb.client.ModelDbSyncer._

// New line.
val syncer = ModelDbSyncer.setSyncer(new ModelDbSyncer())

// Use randomSplitSync instead of randomSplit.
val Array(train, test) = data.randomSplitSync(Array(0.7, 0.3))
val lr = new LogisticRegression()
```

```
      .setMaxIter(20)
      .setLabelCol(FeatureVectorizer.indexed(labelCol))
      .setPredictionCol(predictionCol)
      .setFeaturesCol(featuresCol)
val ovr = new OneVsRest()
      .setClassifier(lr)
      .setLabelCol(FeatureVectorizer.indexed(labelCol))
      .setPredictionCol(predictionCol)
      .setFeaturesCol(featuresCol)

// Use fitSync instead of fit.
val model = ovr.fitSync(train)

// Use transformSync instead of transform.
val predictions = model.transformSync(test)
val eval = new MulticlassClassificationEvaluator()
      .setLabelCol(FeatureVectorizer.indexed(labelCol))
      .setPredictionCol(predictionCol)
      .setMetricName("f1")

// Use evaluateSync instead of evaluate.
val score = eval.evaluateSync(predictions, model)
```

As displayed above, the key changes required to use ModelDB Spark Client are to import and set up the ModelDB Syncer (the first two lines) and append the suffix "Sync" to the operations that the user would like to store in ModelDB Server. ModelDB Spark Client provides "Sync" variants of many Spark.ML methods such as evaluate(), transform(), fit(), save(), and randomSplit(). Additionally, the ModelDBSyncer object (called "syncer" in the code sample above) exposes a number of methods for interacting with ModelDB Server as well (e.g. deserialize a model stored in the ModelDB S+C filesystem, compute ancestry of a DataFrame).

ModelDB Spark Client uses the same types as Spark.ML, so it should interact nicely with other Spark.ML code and with other libraries that build on Spark.ML. Additionally, it does not re-implement the functionality of Spark.ML, it only adds functionality. To understand how this is done, it is worth looking at some aspects of the implementation.

5.3.2 Syncable Event

Recall that a Syncable Event represents an operation that can be stored on ModelDB Server. The ModelDB Spark Client has a SyncableEvent class that is responsible for converting Spark objects into their corresponding Thrift structures and then making the appropriate Thrift endpoint call to store the operation and its primitives on ModelDB Server. Specifically, SyncableEvent is a class from which many other subclasses derive. These subclasses implement a few methods:

1. **constructor**: The constructor accepts a number of Spark.ML objects (e.g. a DataFrame, Estimator, and Transformer, in the case of FitEvent) and sets them as state variables.

2. **makeEvent**: This method uses the state variables described above and creates a Thrift structure representing the event.

3. **sync**: This method calls makeEvent and stores the result on ModelDB Server using the appropriate API endpoint. It then passes the result of the API call to the associate method.

4. **associate**: This method uses the result of the API call to update the ModelDBSyncer as appropriate. This includes, for example, updating the ID mappings. The ModelDBSyncer is described later in this chapter.

There are SyncableEvent subclasses for all of the Syncable Events that can be stored on ModelDB Server. These include FitEvent, TransformEvent, MetricEvent, GridSearchCrossValidationEvent, and more.

5.3.3 ModelDBSyncer

The Spark Client includes a global object called the ModelDBSyncer. This object has a few responsibilities:

1. It maintains a buffer of SyncableEvents, which it periodically flushes (by calling sync() on each entry of the buffer). The user can configure the syncing strategy (e.g. sync immediately when the buffer contains a SyncableEvent, sync only when the ModelDBSyncer is explicitly told to sync).

2. It exposes convenience functions for interacting with ModelDB Server's API endpoints for gleaning information about the model building process (e.g load a deserialized model, find the original features that were used to create a model).

3. It maintains some state about the Spark objects. For example, it stores a mapping between Spark objects (e.g. Transformer) to their corresponding IDs in ModelDB Server. This makes it possible to easily get the Spark object with a given ID and to get the corresponding ID of a Spark object. Another example, it stores, for each DataFrame, the filepath containing the DataFrame's data.

4. It implements all the traits in the ModelDB Syncer, making it possible to bring all the implicit classes into the user's program with just a single import statement.

5.3.4 Implicit Classes and Traits

ModelDB Spark Client aims to augment methods in Spark.ML without having to reimplement or modify any existing code in Spark.ML. For example, consider the following Spark.ML method:

```
val model = estimator.fit(dataframe)
```

The method invocation above uses an Estimator to train a model (a Transformer) on a DataFrame.

ModelDB Spark Client includes the following method, which involves the same estimator, dataframe, and model objects indicated above and each one is the same type as it is in the code sample above.

```
val model = estimator.fitSync(dataframe)
```

This is almost identical in function to Spark.ML's fit method, except that it logs a FitEvent (or a GridSearchCrossValidationEvent or PipelineEvent if estimator is a CrossValidator or Pipeline, respectively) to ModelDB Server.

To make the above code possible, ModelDB Spark Client uses a combination of Scala implicit classes and Scala traits. A Scala trait is similar to an interface in Java or C#, but it also can provide default implementations for interface methods. A Scala implicit class serves a purpose similar to Java's autoboxing and unboxing features.

Implementing the fitSync above requires the following:

1. A Scala trait called SyncableEstimator is defined.
2. Inside SyncableEstimator, a Scala implicit class for Estimator, called EstimatorSync, is defined. This EstimatorSync implicit class contains an implementation for fitSync(), which calls fit() on its corresponding Estimator object, creates a FitEvent (a SyncableEvent in Spark Client), and buffers it onto the global ModelDBSyncer.
3. ModelDBSyncer is marked as implementing the SyncableEstimator trait, so that the EstimatorSync implicit class is automatically imported into the user's main program.

Thus, when the user calls fitSync, the following occurs.

1. Scala looks for an implementation of fitSync() in the Estimator class, and cannot find one.
2. Scala notices there is an implicit class (imported when ModelDBSyncer was imported) for Estimator called EstimatorSync that does implement fitSync().

3. Scala creates an EstimatorSync object and passes the Estimator object as a constructor argument.

4. Scala calls the fitSync() method on the EstimatorSync object.

5. Scala returns the result of the fitSync() call, which is the trained model.

This makes it possible augment the Estimator class's fit() function so that it buffers a FitEvent in the ModelDBSyncer without having to rewrite or edit existing Spark.ML code.

This technique is used many times in ModelDB Spark Client (e.g. for transformSync(), evaluateSync()).

5.3.5 FeatureVectorizer

ModelDB Spark Client also includes a class called FeatureVectorizer which is a convenience class for building pre-processing pipelines. The user specifies a DataFrame and indicates the categorical and numerical columns they would like to use. Then, FeatureVectorizer builds a PipelineModel that performs common preprocessing steps (e.g. string indexing, one-hot encoding, standardization of numerical columns) and creates an output DataFrame with a "features" column that can be used by a machine learning Estimator or Model.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Evaluation

By importing ModelDBSyncer and appending "Sync" to the method names, the user can use ModelDB Spark Client to record and send their operations and models on ModelDB Server, which stores the data in its database and model filesystem. Naturally, this takes extra running time and storage space than if the user did not use ModelDB S+C at all. Additionally, the API methods exposed by ModelDB Server take some time to run too. The goal of this chapter is to measure whether the time and space requirements of ModelDB S+C are reasonable.

6.1 Overview

Experiments were conducted to answer the following questions:

1. Is the time overhead of storing operations and models on ModelDB Server reasonably small (compared to the overall running time of the Spark.ML program)?

2. Is the space taken up by ModelDB S+C's database and model files reasonably small (compared to the size of the dataset)?

3. Are ModelDB S+C's tree model and linear model representations reasonably small (compared to the size of corresponding PMML files)?

Dataset Name	Problem Type	Number of Rows	Number of Features
IMDB	Regression	5,043	28
Animal Shelter	Multiclass Classification	26,729	10
Housing Prices	Regression	1,460	81
Iris	Multiclass Classification	150	5
Titanic	Binary Classification	1,309	14
SMS Spam	Binary Classification	5,574	N/A (text)
Flight Delays	Binary Classification	7,453,215	29

Table 6.1: Datasets used in evaluation

4. Is the time take to execute the ModelDB Server API methods reasonably small (compared to the training time of the model)?

5. Can ModelDB S+C record most of the machine learning operations in real Spark.ML programs?

The chapter also discusses performance improvements that can further improve ModelDB S+C with regard to the above questions.

6.2 Datasets

ModelDB S+C was evaluated on real datasets, which are described below and outlined in Table 6.1.

The **IMDB** dataset [19] includes features like genre, number of reviews, language, and more for over 5000 movies in the IMDB Movie Database. The dataset also includes the IMDB score (i.e. a 1 to 10 rating of how good the movie is considered) for each movie. A machine learning model could solve a regression problem on this dataset in which it predicts the IMDB score for a given movie. Such a model could be used to identify the best movies and recommend them to users. This dataset was used when measuring the time/space overhead of ModelDB S+C and was also used when measuring the execution time for ModelDB Server API methods.

The **Animal Shelter** dataset [21] includes features like animal type (cat or dog), breed, color, age, and more for over 25,000 animals. The dataset also includes the animal's outcome (e.g. adopted, returned to owner, transferred) for each animal. A machine learning model could solve a multi-class classification problem on this dataset in which it predicts the most likely outcome for a given animal. Such a model could be used to identify the most likely outcome for an animal newly admitted into the animal shelter. This dataset was used when measuring the time/space overhead of ModelDB S+C.

The **Housing Prices** dataset [18] includes features like square footage, number of bathrooms, neighborhood type, and more for about 1,500 houses. The dataset also includes the sale price of each house. A machine learning model could solve a regression problem on this dataset in which it predicts the sale price of a given home. Such a model could be used by realtors who are trying to find the best price for a home when putting it up on the market. This dataset was used when measuring the time/space overhead of ModelDB S+C.

The **Iris** dataset [25] is a famous dataset that includes features like petal length, sepal width, and more for 150 Iris plans. The dataset also includes the species of the Iris plant. A machine learning model could solve a multi-class classification problem on this dataset in which it predicts the species of a given Iris plant. Such a model may be useful for botanists looking to classify their plants. This dataset was used when comparing the size of ModelDB S+C models to PMML models (the PMML website includes model files for the Iris dataset).

The **Titanic** dataset [9] is a famous dataset includes features like passenger sex, passenger class, and more for over 1300 passengers of the Titanic. The dataset also includes whether each passenger survived or died. A machine learning model could solve a binary classification problem on this dataset in which it predicts whether a given passenger survived. Such a model may be useful for historians. This dataset was used when evaluating ModelDB S+C on an existing machine learning workflow.

The **SMS Spam** dataset [2] includes over 5500 text messages and a label indicating whether the message is spam or not. A machine learning model could solve a binary classification problem on this dataset in which it predicts whether a given text message is spam. Such a model could be used as a text messaging app's spam filter. This dataset was used when evaluating ModelDB S+C on an existing machine learning workflow.

The **Flight Delays** dataset [4] includes features like carrier, departure time, and more for over 7 million airplane departures. It also includes a field indicating whether the plane was delayed. A machine learning model could solve a binary classification problem on this dataset in which it predicts whether a given airplane is delayed. Such a model could be used by airports and airlines to anticipate delays for particular airplanes. This dataset was used when evaluating ModelDB S+C on an existing machine learning workflow.

6.3 Methodology

6.3.1 Machine

These experiments were run on a DigitalOcean machine with a 160 GB SSD disk, an 8 core processor, and 16GB of memory.

6.3.2 Time and Space Overhead

The first experiment focused on evaluating the time and space overhead of ModelDB S+C. The IMDB, Housing Prices, and Animal Shelter datasets were used for this experiment.

First, for each of the datasets listed above, three programs (called workflows) were created. The **simple** workflow simply trains and evaluates one machine learning model. The **full** workflow creates a preprocessing pipeline for the data, trains some models with grid search cross validation (thus trying many hyperparameter configurations), and evaluates the best model. The **exploratory** worklow executes many full workflows, trying different model types (e.g. random forest, linear regression) and different feature sets.

Second, a program was written to artificially increase the size of the dataset by duplicating rows. Dataset sizes were varied from the dataset's original number of rows to one million rows.

Third, ModelDB Spark Client was instrumented so that it recorded the time spent running code related to ModelDB S+C.

Fourth, each (dataset, dataset size, workflow) triple's program was executed with ModelDB S+C enabled. The time that ModelDB S+C spent recording and storing each operation and model was recorded. The database size, number of rows in each table, and size of the model files was also recorded. Finally, the overall running time of the program and the size of the (artificially enlarged) dataset were also noted. Due to Spark's large memory requirements, it was not possible to run the exploratory workflow for the housing dataset when the number of rows was large because Spark.ML consumed too much memory. This is because Spark is designed to be run on machines with large amounts of RAM, and the machine used for these experiments had only 16 GB of RAM.

6.3.3 Computation Time of API Methods

The second experiment focused on evaluating the running time of ModelDB Server's API methods. The IMDB dataset was used here.

First, the IMDB exploratory workflow was run to populate ModelDB Server's database.

Second, the database was duplicated N times (i.e. N additional rows were created for every row in every table of the database), where N was varied from 0 to 400. This was done to simulate many exploratory workflows being run on ModelDB S+C.

Third, various API methods were executed on the (artificially enlarged) database and the running time for each method was recorded.

6.3.4 Compare ModelDB S+C Models to PMML

The PMML website [14] includes sample files for a 200 tree random forest model and logistic regression model that were trained on the Iris dataset. So, this experiment trained a 200 tree random forest model and a one vs. rest logistic regression model in Spark.ML (with ModelDB S+C enabled). The sizes of the model files and database were recorded.

6.3.5 Evaluating Existing Workflows

Three machine learning workflows were collected from the Internet. The first [31], from a Hortonworks article, uses the Flight Delays dataset to build a classification model to predict delays. The second [8] and third [33] are from ZeppelinHub, a website that allows users to share their machine learning workflows as Zeppelin notebooks. The second workflow trains two models to predict passenger survival status for the Titanic dataset and the third builds a spam classifier using the SMS Spam dataset. These workflows were cleaned up, ported to Spark v2.0.0, and augmented with ModelDB Spark Client (i.e. import ModelDbSyncer and add *Sync to the methods).

6.4 Time Overhead Results

To measure the time overhead, the time spent running ModelDB S+C code is divided by the total running time of the program and the result is taken as a percentage. Plotting this time overhead percentage, for each (dataset, workflow) pair, as a function of the number of rows in the dataset yields Figure 6-1. For this experiment, ModelDB S+C did NOT count the number of rows in each DataFrame.

Figure 6-1 shows that, as the dataset size grows, the time overhead percentage for ModelDB S+C goes down. For a one million row dataset, most of the workflows spend less than 5% of their time running ModelDB S+C code.

For the results here, ModelDB S+C is configured not to count the number of rows in the DataFrames. Counting the number of rows in a Spark DataFrame requires a sequential scan of the DataFrame. Doing this would cause the absolute time spent running ModelDB S+C code to grow linearly with the dataset size. So, ModelDB Spark Client disables row-counting by default.

In Figure 6-1, the dataset size is measured in number of rows, rather than in the actual size of the data file. This is done because the file format of the data has a big impact on the size of the data file. Nevertheless, for reference, the largest dataset size

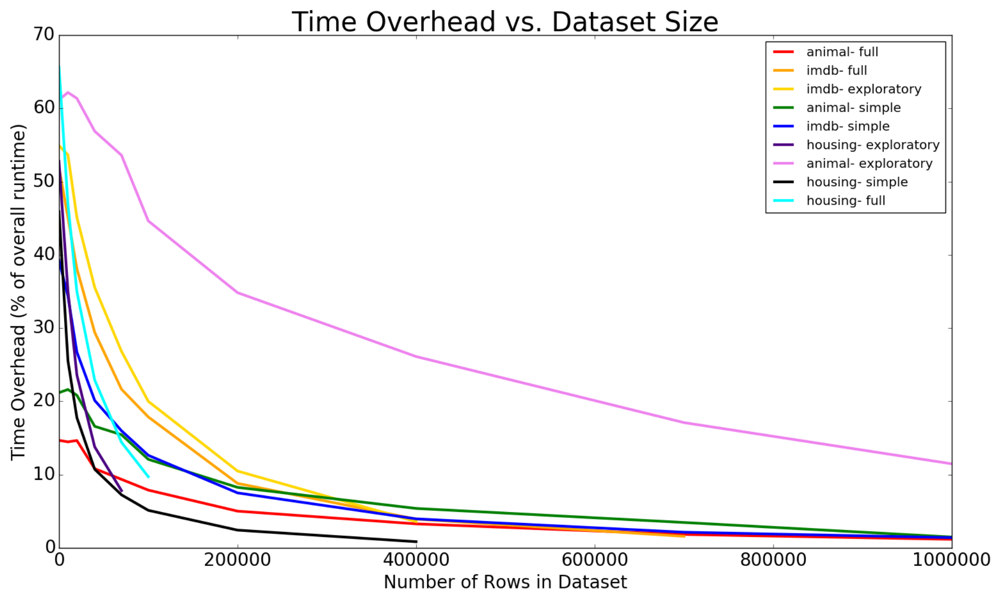


Figure 6-1: Time overhead of ModelDB S+C.

(1 million rows) was under 300 MB in size. All data was in CSV format.

Thus, Figure 6-1 shows that the time overhead of ModelDB S+C becomes insignificant as the dataset grows beyond 300 MB.

The slowest step in recording and storing a model or operation, as determined by instrumenting various lines of code, is writing to the SQLite database. This is not a surprise because the SQLite file lives on disk and because SQLite is not a very performant database. Replacing SQLite with a more performant database may not only reduce the time overhead, it may also reduce the storage requirements as well if data is appropriately compressed.

Focusing on the individual operations, the operation that consumed the most time, by far, was GridSearchCrossValidationEvent. The average time overhead percentages for the most time consuming operations are shown in Figure 6-2.

Figure 6-2 shows that roughly 3% of the time in the overall program is spent storing GridSearchCrossValidationEvents. This occurs because a GridSearchCrossValidationEvent has many other smaller events contained inside it, such as TransformEvents, FitEvents, MetricEvents, and CrossValidationEvents. Consequently, ModelDB Server has to write many rows to many database tables when a GridSearchCrossValidation-

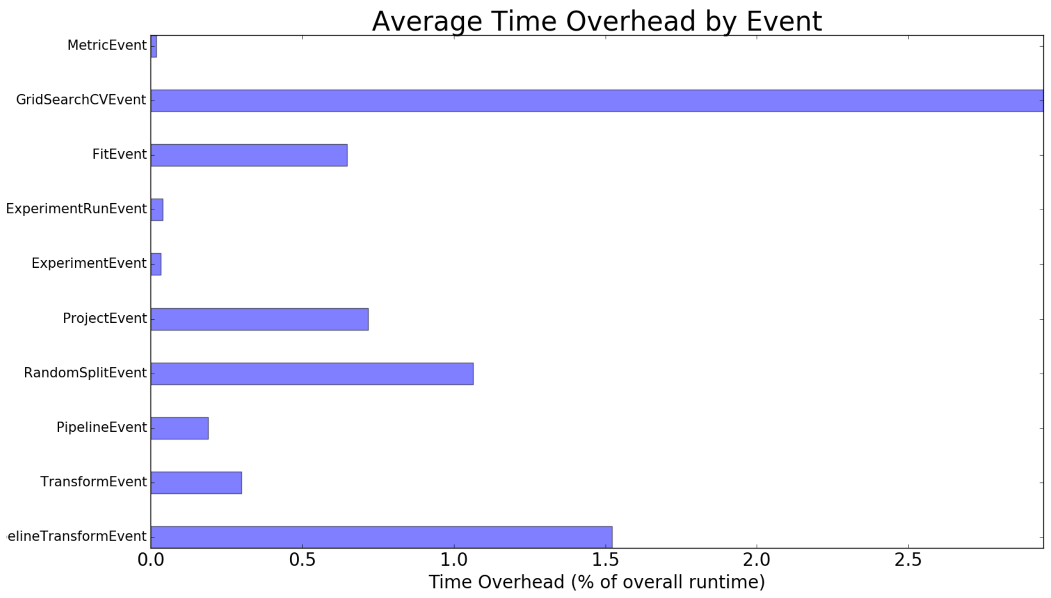


Figure 6-2: Average time overhead percentage by event.

Event occurs. If the user is not interested in storing all the intermediate FitEvents, TransformEvents, etc. for a GridSearchCrossValidation, then much of this overhead is a waste. Therefore, ModelDB S+C allows the user to configure whether they would like to actually store all the data associated with a GridSearchCrossValidationEvent, or whether they would like to simply store the FitEvent associated with the produced model. This makes it possible to cut out much of the overhead caused by GridSearchCrossValidationEvents.

Thus, to summarize, ModelDB S+C's time overhead is small when the dataset is large (i.e. 300 MB or more). A large fraction of the time is spent storing GridSearchCrossValidationEvents, and since the user may not care about storing all the intermediate TransformEvents, MetricEvents, etc., the user is allowed to indicate whether they would like to store the full event or or just the FitEvent for the produced model.

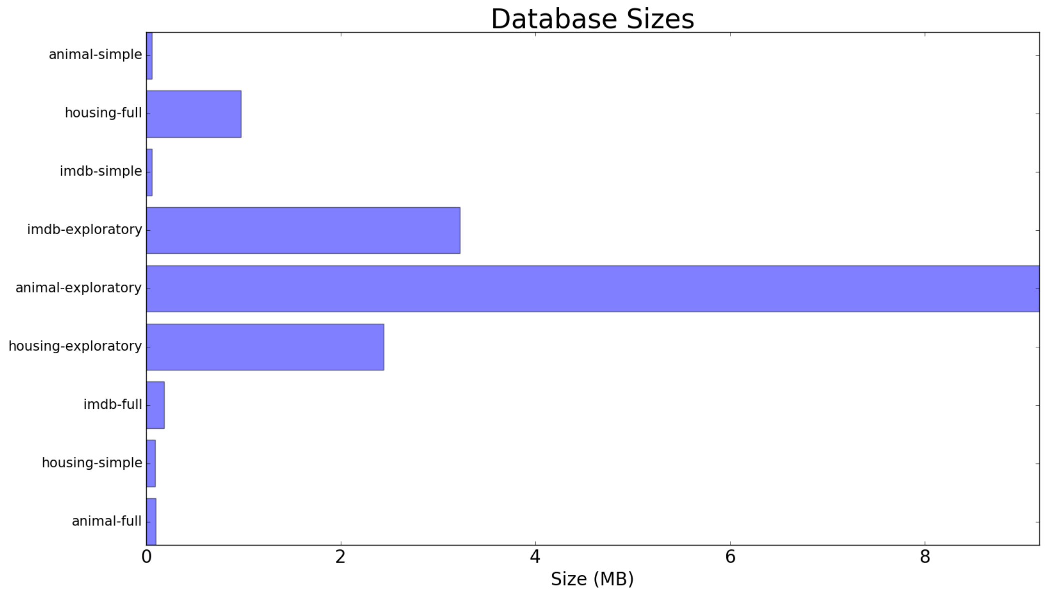


Figure 6-3: Database size (in MB) for each (dataset, workflow) pair.

6.5 Storage Overhead Results

For a fixed (dataset, workflow) pair, the size of ModelDB S+C's database and the size of the model files do not change even as the number of rows in the dataset changes. This makes sense because, except for counting the number of rows, ModelDB S+C does not access any data from the DataFrame's rows. The size of ModelDB S+C's SQLite database is shown for each of the (dataset, workflow) pairs in Figure 6-3.

For all the (dataset, workflow) pairs, the database size remains under 10MB, and is just a few MB for most of the (dataset, workflow) pairs. Since machine learning datasets tend to be large (e.g. several hundred GB), 10MB is quite small. That being said, inspecting the number of rows in each table for the Animal Shelter exploratory workflow can provide some insight into reducing the database size even further. This is shown in Figure 6-4.

Overwhelmingly, the TreeModel and TreeLink tables have the most rows. A little thought, however, shows that this is not surprising. The Animal Shelter exploratory workflow trains a number of random forest models. Consider a random forest with 20 decision trees where each tree has depth 7. Further assume that each decision tree is a

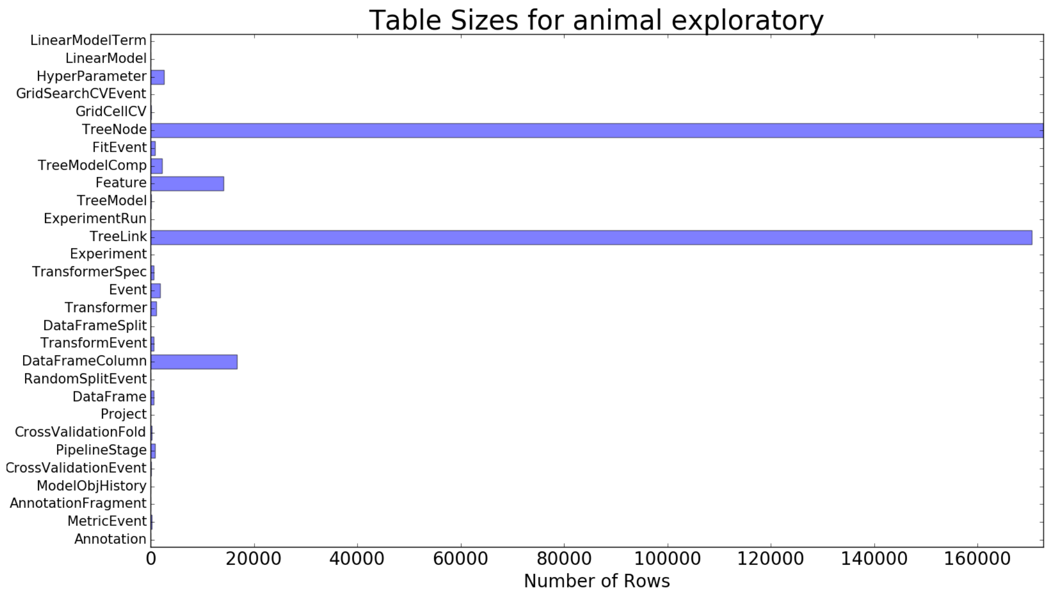


Figure 6-4: Size of tables (in number of rows) for exploratory workflow for Animal Shelter dataset.

binary tree. This means that each tree has on the order of $2^7 = 128$ nodes (and roughly the same number of links). Thus, the random forest has $20 \times 128 = 2560$ nodes (and roughly the same number of links). If 3-fold grid search cross validation is conducted with 8 hyperparameter configurations (a 4×2 search grid), and a total of 3 such grid search cross validations are performed, this becomes $2560 \times 8 \times 3 \times 3 = 184,320$ nodes (and roughly the same number of links). Therefore, it is no surprise that the TreeNode and TreeLink tables are so large.

ModelDB S+C provides two mechanisms to reduce database size. First, as described before, it allows the user to forgo storing ALL the data associated with a GridSearchCrossValidationEvent and instead store only the most important parts. Second, it allows the user to forgo storing entries in the TreeLink and TreeNode table (the node and link data are already stored in the serialized model file) except for specified models. This can reduce the database size greatly, and allow the user to only store TreeNode and TreeLink rows when it is necessary. In the exploratory workflow for the Animal Shelter workflow, only one (i.e. the best) model's TreeLink and TreeNode data is really needed, but the workflow stores data for dozens of random

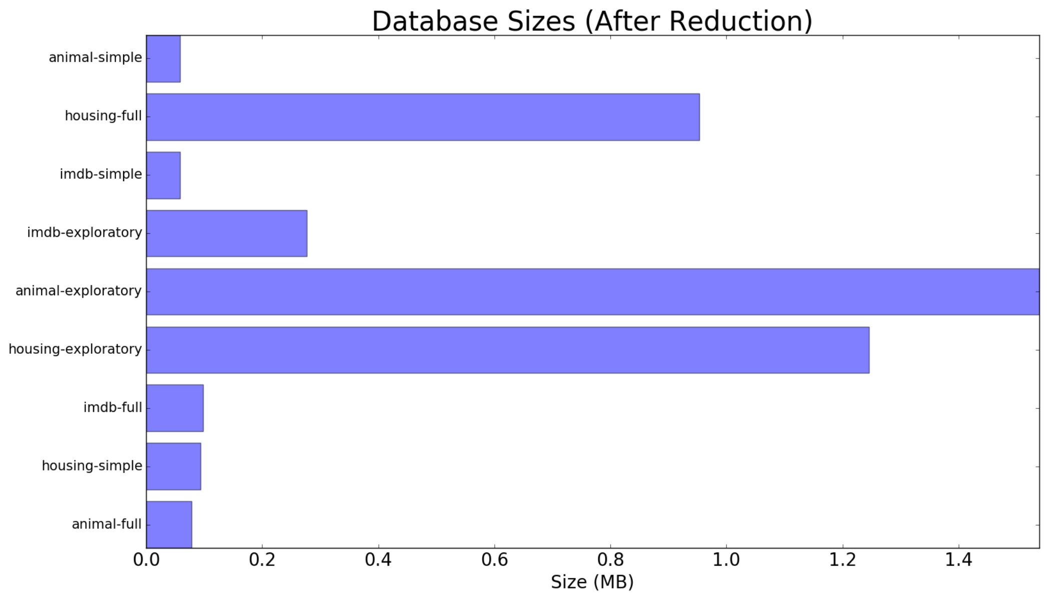


Figure 6-5: Size of database for each (dataset, workflow) pair when GridSearchCross-ValidationEvent intermediates are not stored and when the TreeLink and TreeNode tables are only populated for the last model.

forest models. When the above two mechanisms are applied, the size of the database falls from about 9 MB to 1.6 MB, as shown in figure 6-5.

Finally, it is worth looking at the size of the model files produced by each (dataset, workflow) pair. This is shown in Figure 6-6.

Again, while these sizes are small (< 5 MB) compared to the size of typical machine learning datasets, they can still be reduced. Rather than serializing and storing every single model produced in the workflow, ModelDB S+C allows the user to indicate which models they would actually like to serialize and store (by calling saveSync() on the model). When the final models are explicitly serialized and stored, rather than serializing and storing all the produced Transformers, the model sizes in Figure 6-7 occur.

The models in Figure 6-7 take up much less space.

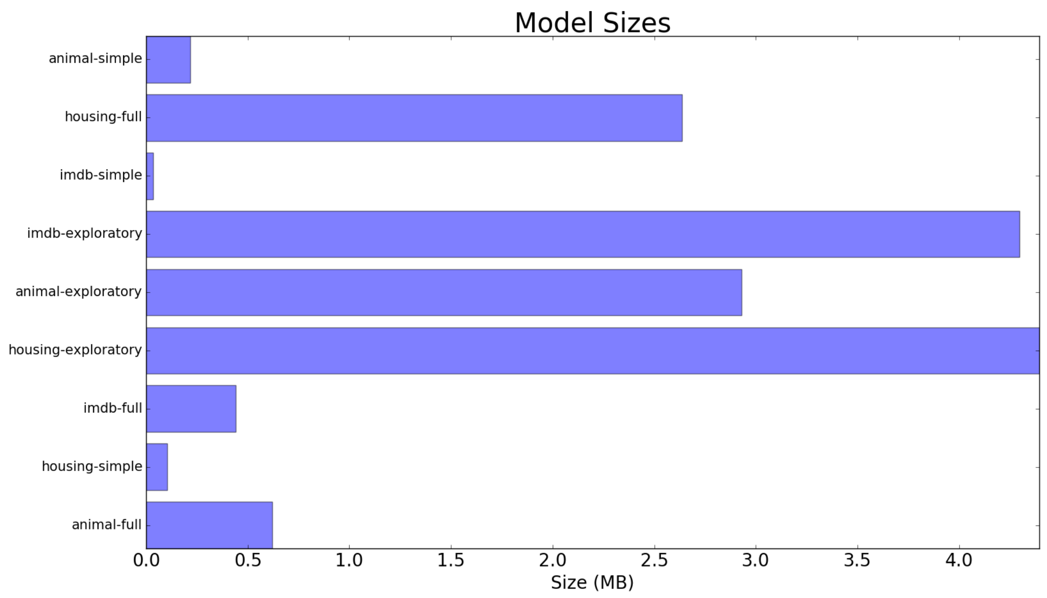


Figure 6-6: Size of model files for each (dataset, workflow) pair.

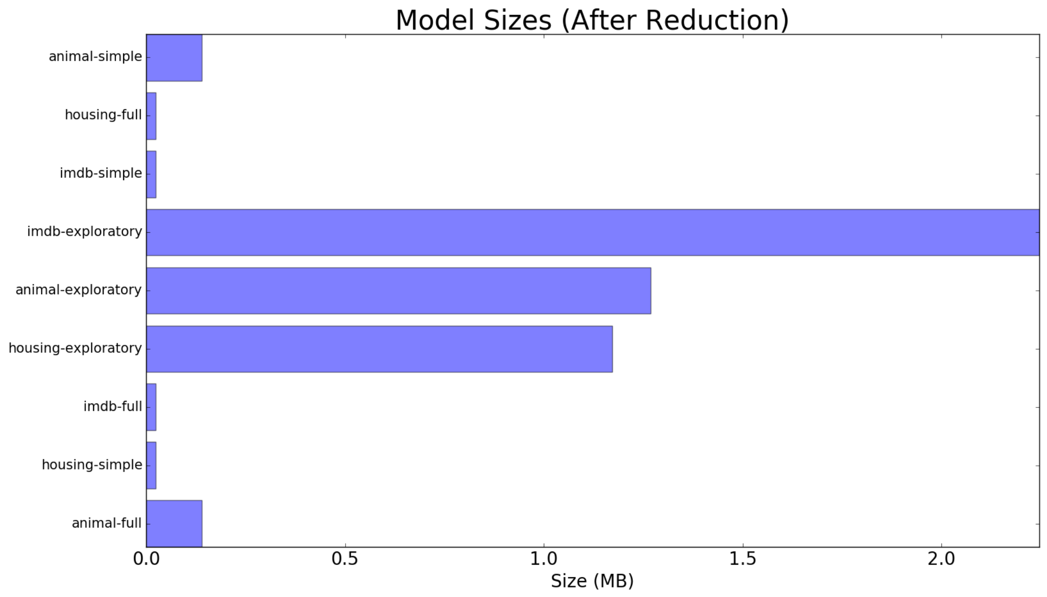


Figure 6-7: Size of model files for each (dataset, workflow) pair when only the final models are serialized and stored, rather than serializing and storing all produced Transformers.

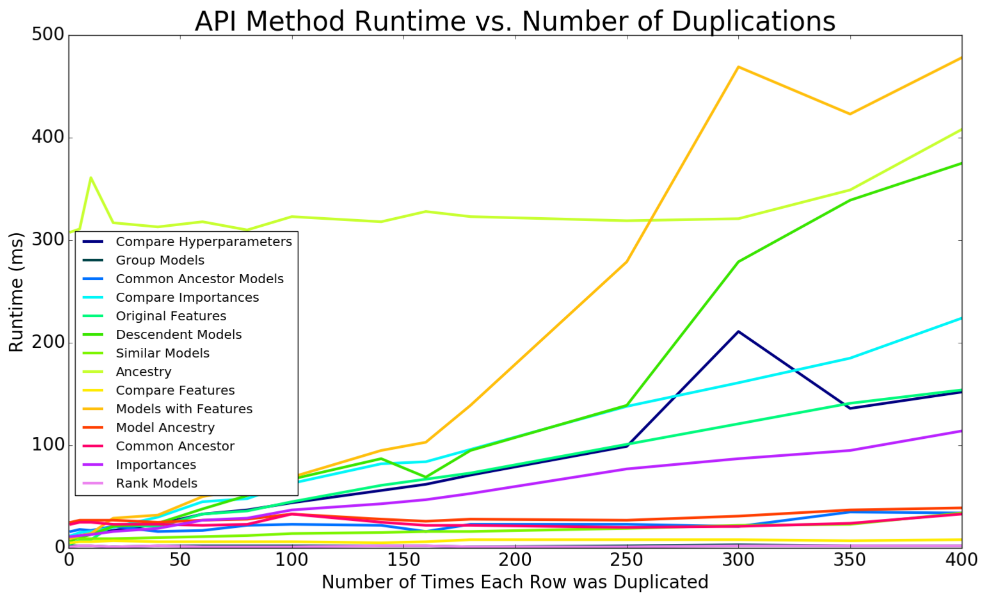


Figure 6-8: Time taken for each method, plotted as a function of the number of times that each row was duplicated.

6.6 API Method Time Results

The IMDB exploratory workflow took about 490 seconds (a little over 8 minutes) to execute, and the majority of this time was spent training models. Figure 6-8 shows the time taken by various API method, as a function of the number of times each row was duplicated (recall that the duplication was done to simulate multiple workflows).

Figure 6-8 shows that the running times of the API methods are very small, with not a single API method taking more than 500 milliseconds. The fast running times are due to the indices in ModelDB Server's database, which speed up the slow queries and due to the fact that all of ModelDB Server's API methods run in $O(n)$ time or better, where n is the size of the database. The running time grows slowly with the number of workflows, and should remain small (under a few seconds) even for a large number of workflows. That being said, there are still a few potential indices that could be created in order to speed up the API methods. Second, some API methods (e.g. descendent models) could theoretically be parallelized. Third, the overall ModelDB system includes abstractions for grouping together Syncable Events

Model Type	Database Size (KB)	Models Size (KB)	PMML Size (KB)
Logistic Regression	16	12	4
Random Forest	116	224	420

Table 6.2: Comparing ModelDB database and model files to PMML file

and primitives (e.g. ExperimentRun, Project), and queries could be restricted to run within a specific ExperimentRun or Project, because these tend to be small. Fourth, using a more performant database than SQLite could speed up the API methods. Overall, though, the running time of the API methods is very small compared to the time taken to run the IMDB exploratory workflow.

6.7 Model Files Compared to PMML Results

The PMML Website includes model files for a random forest model and logistic regression model training on the Iris dataset. To compare ModelDB S+C's storage to PMML, a a random forest model and one vs. rest logistic regression model were trained on the Iris dataset. The results are shown in Table 6.2. Note that the one vs. rest classifier actually trains multiple logistic regression models. A one vs. rest classifier was used because Spark.ML does not currently support the multi-class logistic regression model. To compensate, only one logistic regression model is counted for the sizing, rather than all of them.

So, for the logistic regression model, ModelDB S+C uses $16 + 12 = 28$ KB, which is higher than the 4 KB required for PMML. This is because the logistic regression model is quite simple, which allows the PMML file to be small. ModelDB S+C must store information about the DataFrame, TransformerSpec, DataFrameColumns, and Hyperparameters in its database and it stores some metadata for the deserializer in the model file. These two factors cause ModelDB S+C to use more storage space than the PMML file does.

The situation is reversed for the random forest, however. ModelDB S+C only uses $116 + 224 = 340$ KB, while the PMML file uses 420 KB. The random forest is large (200 trees), so the overhead of storing the DataFrame, TransformerSpec, DataFrameColumn, and Hyperparameters and the overhead of storing deserialization metadata are now small. ModelDB is able to leverage the (somewhat) efficient storage format of SQLite and efficient storage format of Parquet (for the model file), which keeps the overall storage space small. PMML, on the other hand, is an XML-like language, which becomes quite verbose for a large model like the random forest.

Overall, it seems that ModelDB S+C does not require significantly more storage space than PMML for storing models. While it may require more storage space in some cases, it is worth it because PMML cannot be queried like the SQLite database can and Spark.ML cannot deserialize PMML models and use them in the program. One advantage that PMML has is that it is a human readable file (The Parquet file and SQLite file are not human readable) and it is more well-known. However, since ModelDB S+C is not tied to a particular storage format, the user could store their serialized model in PMML, if they so choose.

6.8 Evaluating Existing Workflows Results

Three real machine learning workflows (Flight Delays, Titanic, and SMS Spam) were considered and modified to use ModelDB S+C. This was done to see how many operations in the machine learning workflow ModelDB S+C is able to record. Note in the following paragraphs that "captures" means that ModelDB S+C is able to automatically record and store the operations and machine learning models associated with a step of the workflow.

The Flight Delays workflow begins by reading the CSV file and parsing the data. These parsing steps (e.g. convert timestamp into Java Date) are not captured by ModelDB S+C. While it is possible for the user to indicate this parsing step by manually buffering a TransformEvent in the ModelDBSyncer, it is not done automatically. Then, the workflow creates a pre-processing pipeline, which can be captured completely by ModelDB S+C. Next, a decision tree model and logistic regression model are trained and used to make predictions on the test data - this too can be captured completely by ModelDB S+C.

The Titanic workflow begins by reading the data and splitting the data into training and testing sets, which is captured by ModelDB S+C. Then, it creates a number of visualizations - this is completely ignored by ModelDB S+C. Next, some null values are are replaced with the median value of the column - this is not automatically captured by ModelDB S+C, but the user could manually log a TransformEvent to indicate this cleaning step. Then, grid search cross validation is used to train a logistic regression model - ModelDB S+C completely captures this step. Next, a random forest model is trained using grid search cross validation, and this too is completely captured by ModelDB S+C. Then, the feature importances of the models are listed - this too is captured by ModelDB S+C (it stores feature importances for linear and tree models).

The SMS Spam workflow begins by reading the data and applying a HashingTF transformer to it - ModelDB S+C can capture this step. Then, a logistic regression mdoel is trained, which ModelDB S+C captures. Finally, the model is used to make some predictions, which is also captured by ModelDB S+C.

The above examples illustrate that ModelDB S+C is able to log many of the machine learning operations and models in actual machine learning workflows. It is not able to log any created visualizations, but that is out of scope for ModelDB S+C. There are some data parsing and cleaning steps that it is not able to log automatically, but it allows the user to manually buffer a TransformEvent for this purpose. In the case of the workflows above, it may actually be possible to log some of the data parsing and cleaning steps automatically (create implicit classes for Spark's RDD methods and log a TransformEvent when they are executed), but that has not been implemented yet in ModelDB Spark Client.

6.9 Improvements

While a number of potential improvements have been discussed in the previous sections, it is worth focusing on a few of them below.

First, using a better database than SQLite may improve ModelDB S+C's storage and time overhead. SQLite is not designed for performance. It was used because it is simple and allows ModelDB S+C to be used out of the box.

Second, ModelDB S+C misses a number of opportunities to parallelize queries. For example, when storing CrossValidationEvents in a GridSearchCrossValidation-Event, each CrossValidationEvent is stored one at a time. This is not strictly necessary, as they could be stored in parallel. Writing ModelDB's storage algorithms so that they can store multiple Syncable Events or primitives in parallel may speed up the runtime significantly.

Third, rather than creating a TreeLink table and TreeNode table, it may be possible to store tree models in a JSON column. Since JSON columns are not supported in SQLite, however, this was not done.

Fourth, ModelDB does not apply any compression of its model files. Doing this can reduce their storage requirements.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 7

Role in Overall ModelDB System

ModelDB S+C was not built in isolation. It interacts with a number of other systems in an overall system called ModelDB [30]. ModelDB is a system for machine learning model management. It includes:

1. a web application for visualizing and examining data about models and operations

2. a client, like the Spark Client, designed for Python's Scikit-learn machine learning library

3. a command line toolkit that includes a versioning system for code

4. a prediction store for the predictions made by models

Figure 5-1 showed the system architecture of ModelDB S+C. Figure 7-1 augments Figure 5-1 to show how some of the other pieces of ModelDB interact with ModelDB S+C. The prediction store is excluded from the figure because it currently has not been integrated into the rest of the ModelDB system.

The other systems in ModelDB can be seen as use cases that demonstrate how ModelDB S+C can serve as a foundation for other applications.

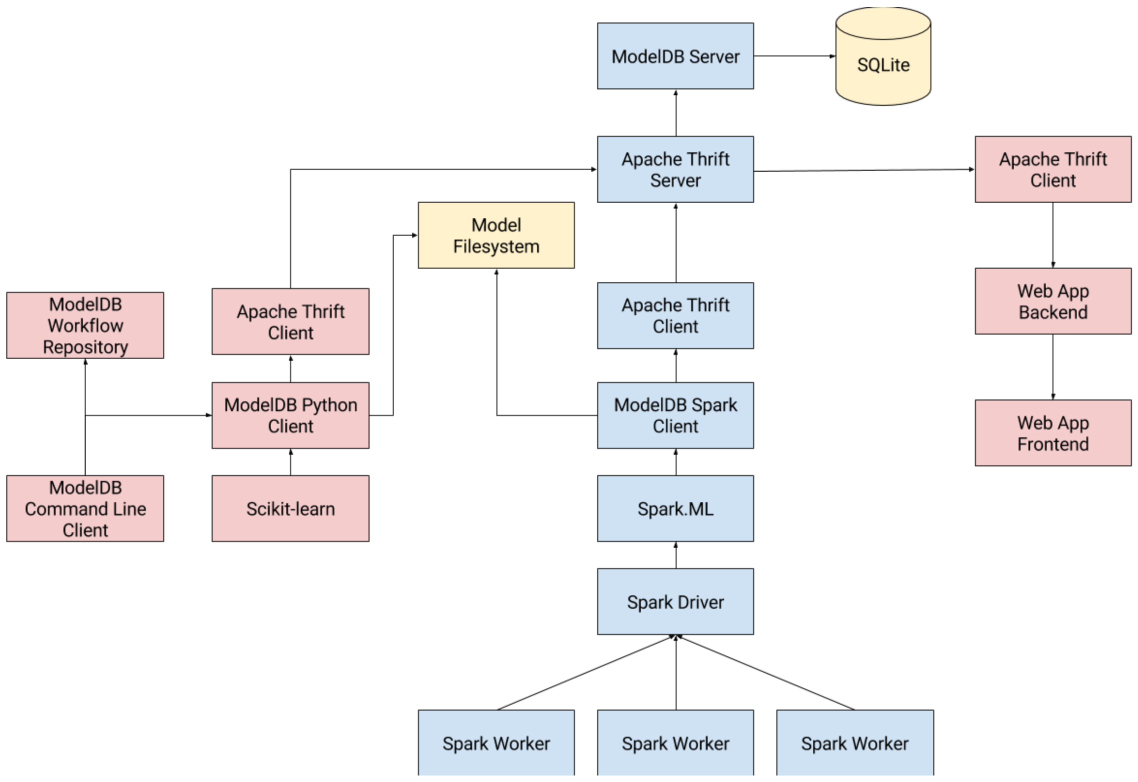


Figure 7-1: The overall architecture of ModelDB that shows the components that interact with ModelDB S+C. The arrows indicates the flow of operations + models data.

7.1 Scikit-learn Client

ModelDB includes a client for Python's Scikit-learn machine learning library. Like the ModelDB Spark Client, the Scikit-learn Client also utilizes the ModelDB Syncer abstraction, the Syncable Event abstraction, and the "Sync" API (e.g. fitSync, transformSync). This Scikit-learn Client was inspired by the abstractions in ModelDB Spark Client, and demonstrates that the ModelDB Server abstractions (e.g. FitEvent, TransformerSpec) can generalize to other machine learning libraries and that ModelDB Spark Client abstractions (e.g. ModelDBSyncer) can also generalize to other machine learning libraries.

7.2 Command Line Toolkit

ModelDB includes a command line toolkit that allows users to record various operations (e.g. evaluation of a model) without using any kind of machine learning library. This command line toolkit still uses the underlying abstractions in ModelDB Server, showing that they can generalize even if there is no machine learning library.

7.3 Web Application

ModelDB includes a web application that can display information about models and visualizations of the model building process. This web application is built directly on the API methods exposed by the ModelDB Server, and does not actually use the database directly. This shows how an application can be built on top of ModelDB Server's data and API.

7.4 Possible Database Applications

Although no applications in ModelDB access the database directly, the database, on its own, could serve as a building block for other applications. Additionally, the database could simply be used in other data management systems, like DataHub [6].

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 8

Future Work

ModelDB S+C is a usable end-to-end system with other systems being built on top of it. However, there are a number of areas for improvement.

8.1 Neural Networks

ModelDB S+C supports special storage and algorithms for linear and tree models. It may be interesting to also add support for neural network models. This was omitted for this thesis because Spark.ML supports only simple multi-layer perceptron models, which is one kind of neural network. Adding support for more general neural network models, especially deep neural networks, could be valuable.

8.2 Library Agnostic Model Format

ModelDB S+C stores detailed data about linear and tree models that could be used to reconstruct them. Consequently, ModelDB Server's database tables could serve as a library agnostic storage format for machine learning models. It may be interesting to implement classes in Spark Client and Scikit-learn Client to read ModelDB Server's tables and reconstruct a Spark Transformer or Scikit-learn Predictor from the contents of the table. This would make it possible, for example, to create a model in Scikit-learn, store it in ModelDB Server's database, and then read it and use it in Spark.ML.

Currently, ModelDB supports storage of serialized model files, so a model created in Spark.ML can later be read and used in Spark.ML.

8.3 Scalability

ModelDB Server currently runs on a single node, but its design is stateless. Therefore, it may be interesting to see how ModelDB Server scales to multiple nodes.

8.4 New clients, API methods, columns

It would be useful to create client libraries for other machine learning libraries, such as those in R. There are a number of interesting operations that could be performed on the data stored in ModelDB Server, so it may be useful to implement more API methods. Finally, storing additional data in the columns of ModelDB Server's tables (e.g. descriptive statistics for each DataFrameColumn), may enable a whole new set of API methods.

Chapter 9

Conclusion

This thesis describes ModelDB Server and ModelDB Spark Client, which together constitute an end-to-end system for logging operations and models created in the model building process.

The primitives and Syncable Event abstractions in ModelDB S+C were presented as building blocks for representing a wide range of operations, such as random splitting of a DataFrame, grid search cross validation, and creation of pre-processing pipelines.

ModelDB Server provides a number of algorithms, exposed as Thrift API endpoints, which leverage the aforementioned abstractions to glean useful information, like model rankings and comparisons, about the model building process.

ModelDB S+C stores detailed data about logistic regression, linear regression, decision tree, random forest, and gradient boosted tree models. It also supports storing serialized models in a filesystem.

ModelDB Spark Client allows automatic logging of machine learning operations performed in Spark.ML, requiring only minor changes to code. Its client side abstractions, such as ModelDB Syncer and SyncableEvent, are general and can be applied to other machine learning libraries (e.g. Scikit-learn).

ModelDB S+C serves as a foundation for the overall ModelDB system, and other systems have been built on top of it.

Recording the model building process is currently difficult and time-consuming to do. ModelDB S+C aims to make this process much easier and utilize data about the model building process to provide insights for the data scientist. ModelDB S+C hopes to be a step towards an era of applications that use data about the model building process to make the model building process easier, more efficient, and less time consuming.

Appendix A

Schema

This appendix contains the schema for the various tables in ModelDB Server's database. Some details (e.g. primary key, non-nullity constraints, indices, non-essential columns) have been omitted for brevity and ease of understanding.

```sql
CREATE TABLE DataFrame (
  numRows INTEGER,
  dataSource TEXT
);

CREATE TABLE DataFrameColumn (
  dfId INTEGER REFERENCES DataFrame,
  name TEXT,
  type TEXT,
  columnIndex INTEGER
);
```

Listing 1: DataFrame Schema

```
CREATE TABLE Transformer (
  transformerType TEXT,
  filepath TEXT
);

CREATE TABLE Feature (
  name TEXT,
  featureIndex INTEGER,
  importance DOUBLE,
  transformer INTEGER REFERENCES TRANSFORMER
);
```

Listing 2: Transformer Schema

```
CREATE TABLE TransformerSpec (
  transformerType TEXT
);

CREATE TABLE HyperParameter (
  spec INTEGER REFERENCES TransformerSpec,
  paramName TEXT,
  paramType TEXT,
  paramValue TEXT,
  paramMinValue REAL,
  paramMaxValue REAL
);
```

Listing 3: TransformerSpec Schema

```
CREATE TABLE Event (
  eventType TEXT,
  eventId INTEGER
);
```

Listing 4: Event Schema

```sql
CREATE TABLE TransformEvent (
  oldDf INTEGER REFERENCES DataFrame,
  newDf INTEGER REFERENCES DataFrame,
  transformer INTEGER REFERENCES Transformer,
  -- Should be comma-separated, no spaces, alphabetical.
  inputColumns TEXT,
  -- Should be comma-separated, no spaces, alphabetical.
  outputColumns TEXT
);
```

Listing 5: TransformEvent Schema

```sql
CREATE TABLE FitEvent (
  transformerSpec INTEGER REFERENCES TransformerSpec,
  transformer INTEGER REFERENCES Transformer,
  df INTEGER REFERENCES DataFrame,
  -- Should be comma-separated, no spaces, alphabetical.
  predictionColumns TEXT,
  -- Should be comma-separated, no spaces, alphabetical.
  labelColumns TEXT,
  problemType TEXT
);
```

Listing 6: FitEvent Schema

```sql
CREATE TABLE MetricEvent (
  transformer INTEGER REFERENCES Transformer,
  df INTEGER REFERENCES DataFrame,
  metricType TEXT,
  metricValue REAL
);
```

Listing 7: MetricEvent Schema

```sql
CREATE TABLE CrossValidationEvent (
  df INTEGER REFERENCES DataFrame,
  spec INTEGER REFERENCES TransformerSpec
);

CREATE TABLE CrossValidationFold (
  metric INTEGER REFERENCES MetricEvent NOT NULL,
  event INTEGER REFERENCES CrossValidationEvent NOT NULL,
);
```

Listing 8: CrossValidationEvent Schema

```
CREATE TABLE GridSearchCrossValidationEvent (
  best INTEGER REFERENCES FitEvent,
);

CREATE TABLE GridCellCrossValidation (
  gridSearch INTEGER REFERENCES GridSearchCrossValidationEvent,
  crossValidation INTEGER REFERENCES CrossValidationEvent
);
```

Listing 9: GridSearchCrossValidationEvent Schema

```
CREATE TABLE PipelineStage (
  pipelineFitEvent INTEGER REFERENCES FitEvent,
  transformOrFitEvent INTEGER REFERENCES Event,
  isFit INTEGER, -- 0 if transform stage and 1 if this is a fit stage.
  stageNumber INTEGER
);
```

Listing 10: PipelineStage Schema

```
CREATE TABLE PipelineStage (
  pipelineFitEvent INTEGER REFERENCES FitEvent,
  transformStage INTEGER REFERENCES TransformEvent,
  fitStage INTEGER REFERENCES FitEvent,
  stageNumber INTEGER
);
```

Listing 11: PipelineStage Modified Schema

```
CREATE TABLE Annotation (
  posted TIMESTAMP
);

CREATE TABLE AnnotationFragment (
  annotation INTEGER REFERENCES Annotation,
  fragmentIndex INTEGER,
  transformer INTEGER REFERENCES Transformer,
  dataFrame INTEGER REFERENCES DataFrame,
  spec INTEGER REFERENCES TransformerSpec,
  message TEXT,
);
```

Listing 12: AnnotationEvent Schema

```
CREATE TABLE LinearModelTerm (
    model INTEGER REFERENCES Transformer,
    termIndex INTEGER,
    coefficient DOUBLE,
    tStat DOUBLE,
    stdErr DOUBLE,
    pValue DOUBLE
);
```

Listing 13: LinearModelTerm Schema

```
CREATE TABLE TreeNode (
  -- 1 if node is leaf, 0 if node is internal
  isLeaf INTEGER,
  -- Internal nodes obviously do not use their predictions
  prediction DOUBLE,
  -- Impurity of node.
  impurity DOUBLE,
  -- Information gain at node. NULL for leaf nodes.
  gain DOUBLE,
  -- Index of feature that the internal node is splitting.
  -- NULL if this is a leaf node.
  splitIndex INTEGER,
  -- NULL for the root node
  rootNode INTEGER REFERENCES TreeNode
);

DROP TABLE IF EXISTS TreeLink;
CREATE TABLE TreeLink (
  parent INTEGER REFERENCES TreeNode,
  child INTEGER REFERENCES TreeNode,
  -- 1 if the child is a left child
  -- 0 if the child is a right child.
  isLeft INTEGER NOT NULL
);
```

Listing 14: TreeNode and TreeLink Schema

```
CREATE TABLE TreeModel (
  model INTEGER REFERENCES Transformer,
  -- Should be "Decision Tree", "GBT", or "Random Forest"
  modelType TEXT NOT NULL
);

CREATE TABLE TreeModelComponent (
  model INTEGER REFERENCES Transformer,
  componentIndex INTEGER,
  componentWeight DOUBLE,
  rootNode INTEGER REFERENCES TreeNode
);
```

Listing 15: TreeModel Schema

Bibliography

[1] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow. org*, 1, 2015.

[2] Tiago A. Almeida and JosÃľ MarÃŋa GÃşmez Hidalgo. Sms spam collection v. 1. Accessed: 2016-12-09.

[3] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2014.

[4] American Statistical Association. Airline on-time performance. Accessed: 2016-12-09.

[5] Roger Barga, Valentine Fontama, Wee Hyong Tok, and Luis Cabrera-Cordon. *Predictive analytics with Microsoft Azure machine learning*. Springer, 2015.

[6] Anant Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J Elmore, Samuel Madden, and Aditya G Parameswaran. Datahub: Collaborative data science & dataset version management at scale. *arXiv preprint arXiv:1409.0798*, 2014.

[7] 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.

[8] P. Bustios. Machine learning on titanic disaster. Accessed: 2016-12-09.

[9] T. Cason. Titanic data. Accessed: 2016-12-09.

[10] Daniel Crankshaw, Peter Bailis, Joseph E Gonzalez, Haoyuan Li, Zhao Zhang, Michael J Franklin, Ali Ghodsi, and Michael I Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *arXiv preprint arXiv:1409.3809*, 2014.

[11] Datacratic. The machine learning database. 2016.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[13] 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.

[14] Data Mining Group. Pmml sample files.

[15] Alex Guazzelli, Michael Zeller, Wen-Ching Lin, Graham Williams, et al. Pmml: An open standard for sharing models. *The R Journal*, 1(1):60–65, 2009.

[16] Geoffrey Holmes, Andrew Donkin, and Ian H Witten. Weka: A machine learning workbench. In *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*, pages 357–361. IEEE, 1994.

[17] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 6. Springer, 2013.

[18] Kaggle. House prices: Advanced regression techniques. Accessed: 2016-12-09.

[19] Kaggle. Imdb 5000 movie dataset. Accessed: 2016-12-09.

[20] Kaggle. Kaggle: How to manage machine learning projects. `https://www.kaggle.com/forums/f/15/kaggle-forum/t/4815/how-to-manage-machine-learning-projects?forumMessageId=25484#post25484`. Accessed 2016-12-03.

[21] Kaggle. Shelter animal outcomes. Accessed: 2016-12-09.

[22] Yehuda Koren. The bellkor solution to the netflix grand prize. *Netflix prize documentation*, 81:1–10, 2009.

[23] Konstantina Kourou, Themis P Exarchos, Konstantinos P Exarchos, Michalis V Karamouzis, and Dimitrios I Fotiadis. Machine learning applications in cancer prognosis and prediction. *Computational and structural biotechnology journal*, 13:8–17, 2015.

[24] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, volume 1, pages 2–1, 2013.

[25] M. Lichman. UCI machine learning repository, 2013.

[26] Hui Miao, Ang Li, Larry S Davis, and Amol Deshpande. Modelhub: Towards unified data and lifecycle management for deep learning. *arXiv preprint arXiv:1611.06224*, 2016.

[27] Evan R Sparks, Ameet Talwalkar, Virginia Smith, Jey Kottalam, Xinghao Pan, Joseph Gonzalez, Michael J Franklin, Michael I Jordan, and Tim Kraska. Mli: An api for distributed machine learning. In *2013 IEEE 13th International Conference on Data Mining*, pages 1187–1192. IEEE, 2013.

[28] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT Press Cambridge, 1998.

[29] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014.

[30] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. Modeldb: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, page 14. ACM, 2016.

[31] Q. Wang. Spark machine learning pipeline by example. Accessed: 2016-12-09.

[32] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[33] ZeppelinHub. Mllib spam classification example. Accessed: 2016-12-09.