O'REILLY®

Early Release

RAW & UNEDITED

Compliments of HOPSWORKS

Building Machine Learning Systems with a Feature Store

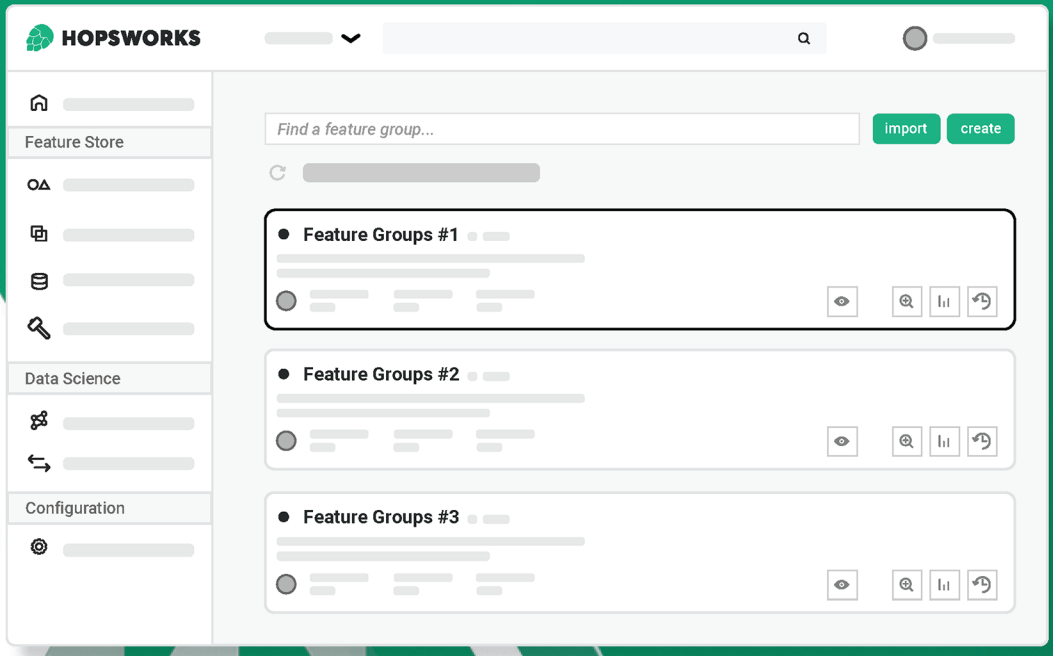
Batch, Real-Time, and LLM Systems

Jim Dowling

HOPSWORKS

Machine Learning Platform & Feature Store

Empowering scale, speed, & real-time AI. With robust data layers designed for extreme performance requirements.



hopsworks.ai

Building Machine Learning Systems with a Feature Store

Batch, Real-Time, and LLM Systems

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Jim Dowling

O'REILLY®

Building Machine Learning Systems with a Feature Store
by Jim Dowling

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisition Editor: Nicole Butterfield
Development Editor: Gary O'Brien
Production Editor: Clare Laylock
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

July 2025: First Edition

Revision History for the Early Release

2024-02-08: First Release
2024-04-09: Second Release
2024-06-24: Third Release
2024-07-25: Fourth Release
2024-10-28: Fifth Release
2025-04-14: Sixth Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098165239* for release details.

978-1-098-16517-8

[TO COME]

Table of Contents

Brief Table of Contents (*Not Yet Final*)

Preface

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the preface of the final book. The GitHub repo can be found at *https://github.com/featurestorebook/mlfs-book*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

This book is the coursebook I would like to have had for ID2223, "Scalable Machine Learning and Deep Learning", a course I developed and taught at KTH Stockholm. The course was, I believe, the first university course that taught students to build complete machine learning (ML) systems using non-static data sources. By the end of the course, the students built their own ML system they developed (around 2 weeks work, in groups of 2) that included:

1. A unique data source that generated new data at some cadence,
2. A prediction problem they would solve with ML using the data source, and
3. A ML system that creates features from the data source, trains a model, makes predictions on new data, visualizes the ML system output with a user interface (interactive or dashboard), and a UI to monitor the performance of their ML system.

Charles Fyre, developer of the Full Stack Deep Learning course, said the following of ID2223:

> In 2017, having a shared pipeline for training and prediction data that updated automatically and made models available as a UI and an API was a groundbreaking stack at Uber. Now it's stanard part of a well-done (ID2223) project.

Some of the examples of ML systems built in ID2223 are shown in Table P-1 below. The ML systems built were a mix of ML systems built with deep learning and LLMs, and more classical ML systems built with decision trees, such as XGBoost.

Table P-1. Example Machine Learning Systems

Prediction Problem	Data Source(s)
Air Quality Prediction	Air quality data, scraped from public sensors and public weather data
Water Height Prediction	Water height data published from sensor readings along with weather data
Football Score Prediction	Football score history and fantasy football data about players and teams
Electricity Demand Prediction	Public electricity demand data, projected demand data, and weather data
Electricity Price Prediction	Public electricity price data, projected price data, and weather data
Game of Thrones Tours Review Response Generator	Tripadvisor reviews and responses
Bitcoin price prediction	Twitter bitcoin sentiment using a Twitter API and a list of the 10,000 top crypto accounts on Twitter

Overview of this book's mission

The goal of this book is to introduce ML systems built with feature stores, and how to build the pipelines (programs with well-defined inputs and outputs) for ML systems while following MLOps best practices for the incremental development and improvement of your ML systems. We will deep dive into feature stores to help you understand how they can help manage your ML data for training models and making predictions. You will acquire some practical skills on how to create and update reusable features with model-independent transformations, as well as how to select, join, and filter features to create point-in-time correct training data for models. You will learn how to implement model-dependent feature transformations that are applied consistently in both training and serving (such as text encoding for large language models (LLMs)). You will learn how to build real-time ML systems with the help of the feature store that provides history and context to (stateless) online applications. You will also learn how to automate, test, and orchestrate ML pipelines. We will apply the skills you acquire to build three different types of ML system: batch ML systems (that make predictions on a schedule), real-time ML systems (that run 24x7 and respond to requests with predictions), and LLM systems (that are personalized using fined-tuning and retrieval augmented generation (RAG)). Finally, you will learn how to govern and manage your ML assets to provide transparency and maintain compliance for your ML system.

Target Reader of this Book

The ideal reader has a role in implementing a data science process and is interested in operationalizing data science. Data engineers, data scientists, and machine learning engineers will enjoy the exercises that will enable them to build the basic components of a feature store.

Chief Digital Officers, Chief Digital Transformation Officers, and CTO's will learn how ML infrastructure, including feature stores, model registries, and model serving infrastructure, enables the transition of machine learning models out of the lab and into the enterprise. Readers should have a basic understanding of Python, databases, and machine learning. Those intending to understand and perform the lab exercises must have Python skills and basic Jupyter notebook familiarity.

The architectural skills you will learn in this book include:

- How to structure a ML system (batch, real-time, or LLM) as modular ML pipelines that can be independently developed, tested, and operated;
- How to ensure the consistency of feature data between offline training and online operations;
- How to govern data in a feature store and promote collaboration between teams with a feature store;
- How to follow MLOps principles of automated testing, versioning, and monitoring of features and models.

The modeling skills you will learn in this book include:

- How to train ML models from (time-series) tabular data in a feature store;
- How to personalize LLMs using fine-tuning and RAG;
- How to validate models using evaluation data from a feature store.

The ML engineering skills you will learn in this book include:

- How to identify and develop reusable model-independent features;
- How to identify and develop model-dependent features;
- How to identify and develop on-demand (real-time) features;
- How to validate feature data;
- How to test feature functions;
- And how to test ML pipelines.

The operational skills you will acquire in this book include:

- How to schedule feature pipelines and batch inference pipelines;
- How to deploy real-time models, connected to a feature store;
- How to log and monitor features and models with a feature store;
- How to develop user-interfaces to ML systems.

Introduction

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the introduction of the final book. The GitHub repo can be found at *https://github.com/featurestorebook/mlfs-book*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

Companies of all stages of maturity, size, and risk adversity are adopting machine learning (ML). However, many of these companies are not actually generating value from ML. In order to generate value from ML, you need to make the leap from training ML models to building and operating ML systems. Training a ML model and building a ML system are two very different activities. If training a model was akin to building a one-off airplane, then building a ML system is more like building the aircraft factory, the airports, the airline, and attendant infrastructure needed to provide an efficient air travel service. The Wright brothers may have built the first heavier-than air airplane in 1903, but it wasn't until 1922 that the first commercial airport was opened. And it took until the 1930s until airports started to be built out around the world.

In the early 2010s, when both machine learning and deep learning exploded in popularity, many companies became what are now known as "hyper-scale AI companies", as they built the first ML systems using massive computational and data storage infrastructure. ML systems such as Google translate, TikTok's video recommendation engine, Uber's taxi service, and ChatGPT were trained using vast amounts of data

(petabytes using thousands of hard drives) on compute clusters with 1000s of servers. Deep learning models additionally need hardware accelerators (often graphical processing units (GPUs)) to train models, further increasing the barrier to entry for most organizations. After the models are trained, vast operational systems (including GPUs) are needed to manage the data and users so that the models can make predictions for hundreds or thousands of simultaneous users.

These ML systems, built by the hyperscale AI companies, continue to generate enormous amounts of value for both their customers and owners. Fortunately, the AI community has developed a culture of openness, and many of these companies have shared the details about how they built and operated these systems. The first company to do so in detail was Uber, who in September 2017, presented its platform for building and operating ML systems, Michelangelo. Michelangelo was a new kind of platform that managed the data and models for ML as well as the feature engineering programs that create the data for both training and predictions. They called Michelangelo's data platform a *feature store* - a data platform that manages the feature data (the input data to ML models) throughout the ML lifecycle—from training models to making predictions with models. Now, in 2024, it is no exaggeration to say that all Enterprises that build and run operational ML applications at scale use a feature store to manage their data for AI. Michelangelo was more than a feature store, though, as it also includes support for storing and serving models using a model registry and model serving platform, respectively.

Naturally, many organizations have not had the same resources that were available to Uber to build equivalent ML infrastructure. Many of them have been stuck at the model training stage. Now, however, in 2024, the equivalent ML infrastructure has become accessible, in the form of open-source and serverless feature stores, vector databases, model registries, and model serving platforms. In this book, we will leverage open-source and serverless ML infrastructure platforms to build ML systems. We will learn the inner workings of the underlying ML infrastructure, but we will not build that ML infrastructure—we will not start with learning Docker, Kubernetes, and equivalent cloud infrastructure. You no longer need to build ML infrastructure to start building ML systems. Instead, we will focus on building the software programs that make up the ML system—the ML pipelines. We will work primarily in Python, making this book widely accessible for Data Scientists, ML Engineers, Architects, and Data Engineers.

From ML Models to MLOps to ML Systems

The value of a ML model is derived from the predictions it makes on new input data. In most ML courses, books, and online tutorials, you are given a static dataset and asked to train a model on some of the data and evaluate its performance using the rest of the data (the holdout data). That is, you only make a prediction once on the

holdout data—your model only generates value once. Many ML educators will say something like: "we leave it as an exercise to the reader to productionize your ML model", without defining what is involved in model productionalization. The new discipline of Machine learning operations (MLOps) attempts to fill in the gaps to productionization by defining processes for how to automate model (re-)training and deployment, and automating testing to increase your confidence in the quality of your data and models. This book fills in the gaps by making the leap from MLOps to building ML systems. We will define the principles of MLOps (automated testing, versioning, and monitoring), and apply those principles in many examples throughout the book. In contrast to much existing literature on MLOps, we will not cover low-level technologies for building ML infrastructure, such as Docker and Terraform. Instead, what we will coverthe programs that make up ML systems, the ML pipelines, and the ML infrastructure they will run on in detail.

Supervised learning primer and what is a feature anyway?

In this book, we will frequently refer to concepts from supervised learning. This section is a brief introduction to those concepts that you may safely skip if you already know them.

Machine learning is concerned with making accurate predictions. *Features* are measurable properties of entities that we can use to make predictions. For example, if we want to predict if a piece of fruit is an apple or an orange (apple or orange is the fruit's *label*), we could use the fruit's color as a feature to help us predict the correct class of fruit, see figure 1. This is a classification problem: given examples of fruit along with their color and label, we want to classify a fruit as either an apple or orange using the color feature. As we are only considering 2 classes of fruit, we can call this a *binary classification problem*.

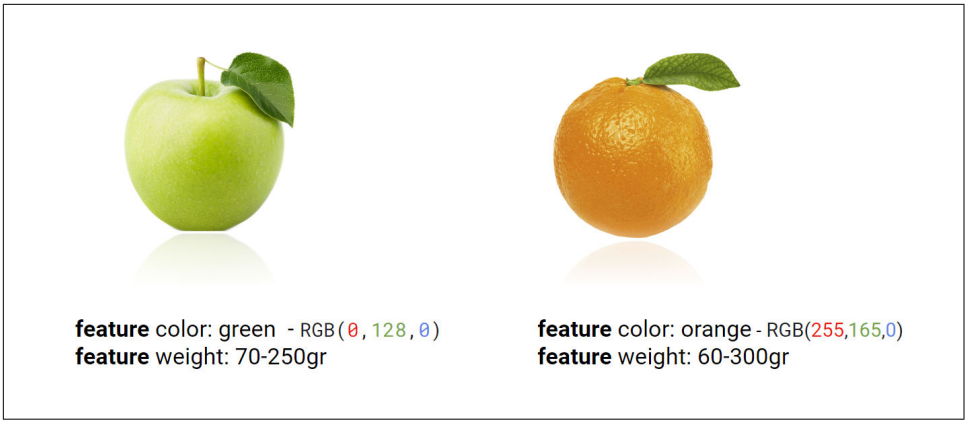


Figure I-1. A feature is a measurable property of an entity that has predictive power for the machine learning task. Here, the fruit's color has predictive power of whether the fruit is an apple or an orange.

The fruit's color is a good feature to distinguish apples from oranges, because oranges do not tend to be green and apples do not tend to be orange in color. Weight, in contrast, is not a good feature as it is not predictive of whether the fruit is an apple or an orange. "Roundness" of the fruit could be a good feature, but it is not easy to measure —a feature should be a measurable property.

A supervised learning algorithm trains a machine learning model (often abbreviated to just 'model'), using lots of examples of apples and oranges along with the color of each apple and orange, to predict the label "Apple" or "Orange" for new pieces of fruit using only the new fruit's color. However, color is a single value but rather measured as 3 separate values, one value for each of the red, green, and blue (RGB) channels. As our apples and oranges typically have '0' for the blue channel, we can ignore the blue channel, leaving us with two features: the red and green channel values. In figure 2, we can see some examples of apples (green circles) and oranges (orange crosses), with the red channel value plotted on the x-axis and the green channel value plotted on the y-axis. We can see that an almost straight line can separate most of the apples from the oranges. This line is called the decision boundary and we can compute it with a linear model that minimizes the distance between the straight line and all of the circles and crosses plotted in the diagram. The decision boundary that we learnt from the data is most commonly called the (trained) model.

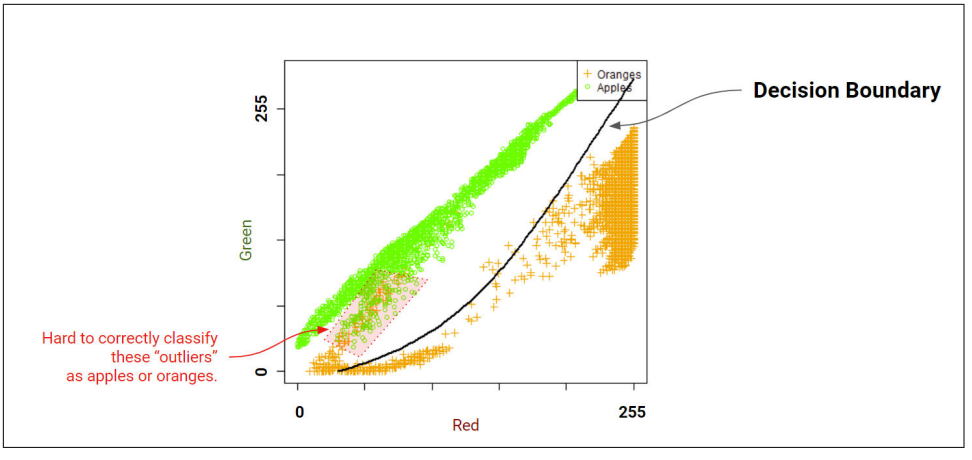


Figure I-2. When we plot all of our example apples and oranges using the observed values for the red and green color channels, we can see that most apples are on the left of the decision boundary, and most oranges are on the right. Some apples and oranges are, however, difficult to differentiate based only on their red and green channel colors.

The model can then be used to classify a new piece of a fruit as either an apple or orange using its red and green channel values. If the fruit's red and green channel values place it on the left of the line, then it is an apple, otherwise it is an orange.

In figure 2, you can also see there are a small number of oranges that are not correctly classified by the decision boundary. Our model could wrongly predict that an orange is an apple. However, if the model predicts the fruit is an orange, it will be correct - the fruit will be an orange. We can say that the model's *precision* is 100% for oranges, but is less than 100% for apples.

Another way to look at the model's performance is to consider if the model predicts it is an apple, and it is an apple - it will not be wrong. However, the model will not always predict the fruit is an orange if the fruit is an orange. That is, the model's *recall* is 100% for apples. But if the model predicts an orange, it's recall is less than 100%. In machine learning, we often combine precision and recall in a single value called the *F1 Score*, that can be used as one measure of the model's performance. The F1 score is the harmonic mean of precision and recall, and a value of 1.0 indicates perfect precision and recall for the model. Precision, recall, and F1 scores are model performance measures for classification problems.

Let's complicate this simple model. What if we add red apples into the mix? Now, we want our model to classify whether the fruit is an apple or orange - but we will have both red and green apples, see figure 3.

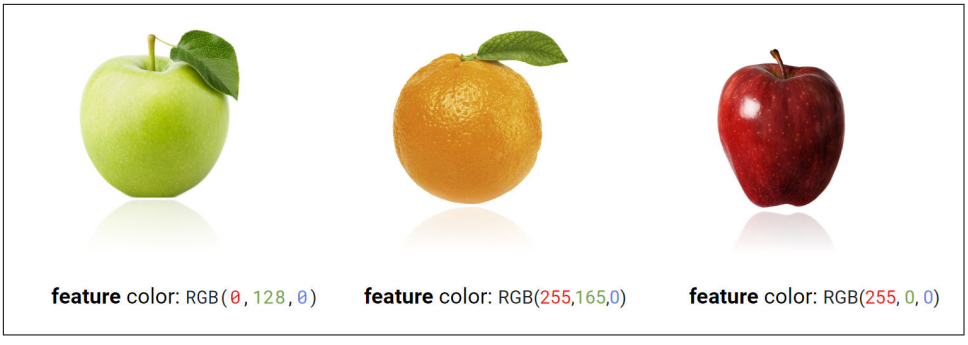


Figure I-3. The red apple complicates our prediction problem because there is no longer a linear decision boundary between the apples and oranges using only color as a feature.

We can see that red apples also have zero for the blue channel, so we can ignore that feature. However, in figure 4, we can see that the red examples are located in the bottom right hand corner of our chart, and our model (a linear decision boundary) is broken—it would predict that red apples are oranges. Our model's precision and recall is now much worse.

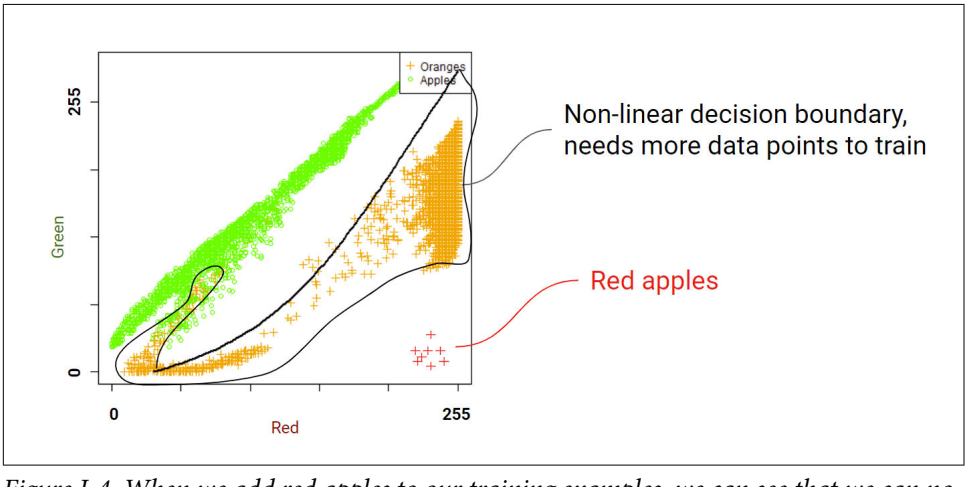


Figure I-4. When we add red apples to our training examples, we can see that we can no longer use a straight line to classify fruit as orange or apple. We now need a non-linear decision boundary to separate apples from oranges, and in order to learn the decision boundary, we need a more complex model (with more parameters), more training examples, and m.

Our fruit classifier used examples of features and labels (apples or oranges) to train a model (as a decision boundary). However, machine learning is not just used for classification problems. It is also used to predict numerical values—*regression problems*. An example of a regression problem would be to estimate the weight of an apple. For

the regression problem of predicting the weight of an apple, two useful features could be its diameter, and its green color channel value—dark green apples are heavier than light green and red apples. The apple's weight is called the target variable (we typically use the term label for classification problems and target in regression problems).

For this regression problem, a supervised learning model could be trained using examples of apples along with their green color channel value, diameter, and weight. For new apples (not seen during training), our model, see figure 5, can predict the fruit's weight using its type, red channel value, green channel value, and diameter.

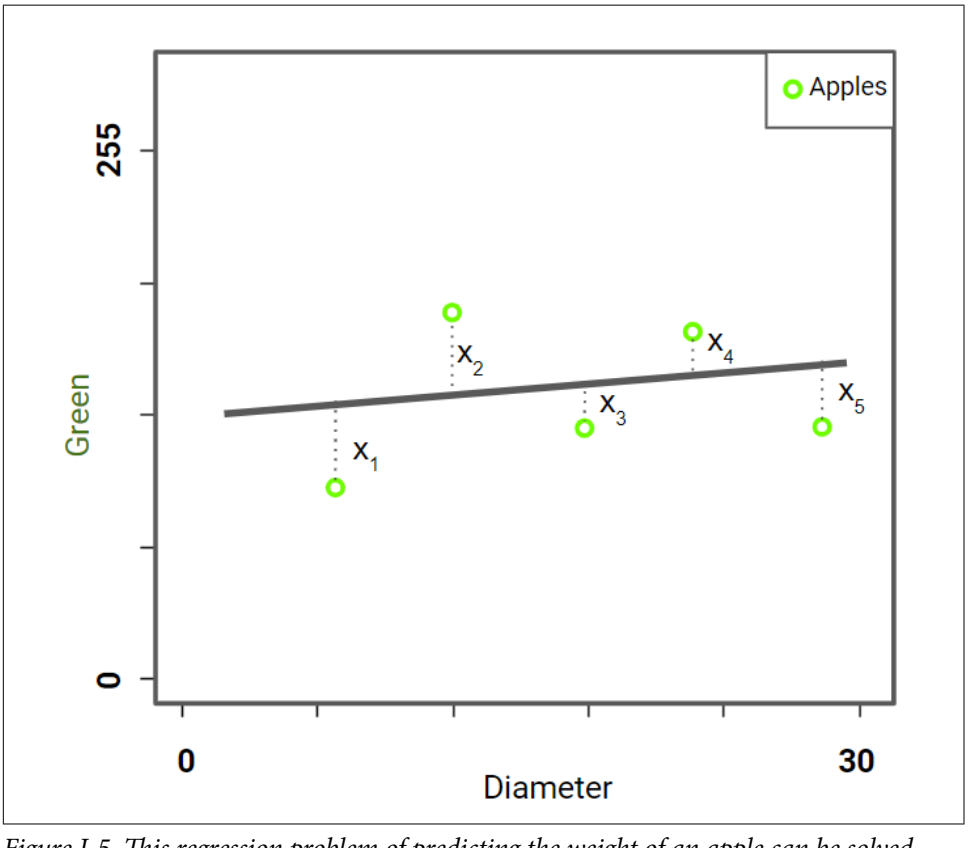


Figure I-5. This regression problem of predicting the weight of an apple can be solved using a linear model that minimizes the mean-squared error

In this regression example, we don't technically need the full power of supervised learning yet—a simple linear model will work well. We can *fit* a straight line (that predicts an apple's weight using its green channel value and diameter) to the data points by drawing the line on the chart such that it minimizes the distance between the line and the data points (X_1, X_2, X_3, X_4, X_5). For example, a common technique is to sum together the distance between all the data points and the line in the *mean absolute*

error (MAE). We take the absolute value of the distance of the data points to the line, because if we didn't take the absolute value then the distance for X_1 would be negative and the distance for X_2 would be positive, canceling each other out. Sometimes, we have data points that are very far from the line, and we want the model to have a larger error for those outliers—we want the model to perform better for outliers. For this, we can sum the square of distances and then take the square root of the total. This is called the *root mean-squared error* (RMSE). The MAE and RMSE are both metrics used to help fit our linear regression model, but also to evaluate the performance of our regression model. Similar to our earlier classification example, if we introduce more features to improve the performance of our regression model, we will have to upgrade from our linear regression model to use a supervised learning regression model that can perform better by learning non-linear relationships between the features and the target.

Now that we have introduced supervised learning to solve classification and regression problems, we can claim that supervised learning is concerned with extracting a pattern from data (features and labels/targets) to a model, where the model's value is that it can used to perform *inference* (make predictions) on new (unlabeled) data points (using only feature values). If the model performs well on the new data points (that were not seen during training), we say the model has good *generalization* performance. We will later see that we always hold back some example data points during model training (a *test set* of examples that we don't train the model on), so that we can evaluate the model's performance on unseen data.

Now we have introduced the core concepts in supervised learning[1], let's look at where the data used to train our models comes from as well as the data that the model will make predictions with.

The source code for the supervised training of our fruit classifier is available on the book's github repository in chapter one. If you are new to machine learning it is a good exercise to run and understand this code.

1 The source code for the supervised training of our fruit classifier is available on the book's github repository in chapter one. If you are new to machine learning it is a good exercise to run and understand this code.

CHAPTER 1
Building Machine Learning Systems

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. The GitHub repo can be found at *https://github.com/featurestorebook/mlfs-book*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

Imagine you have been tasked with producing a financial forecast for the upcoming financial year. You decide to use machine learning as there is a lot of available data, but, not unexpectedly, the data is spread across many different places—in spreadsheets and many different tables in the data warehouse. You have been working for several years at the same organization, and this is not the first time you have been given this task. Every year to date, the final output of your model has been a PowerPoint presentation showing the financial projections. Each year, you trained a new model, and your model made one prediction and you were finished with it. Each year, you started effectively from scratch. You had to find the data sources (again), re-request access to the data to create the features for your model, and then dig out the Jupyter notebook from last year and update it with new data and improvements to your model.

This year, however, you realize that it may be worth investing the time in building the scaffolding for this project so that you have less work to do next year. So, instead of

delivering a powerpoint, you decide to build a dashboard. Instead of requesting one-off access to the data, you build feature pipelines that extract the historical data from its source(s) and compute the features (and labels) used in your model. You have an insight that the feature pipelines can be used to do two things: compute both the historical features used to train your model and compute the features that will be used to make predictions with your trained model. Now, after training your model, you can connect it to the feature pipelines to make predictions that power your dashboard. You thank yourself one year later when you only have to tweak this *ML system* by adding/updating/removing features, and training a new model. The time you saved in grunt data source, cleaning, and feature engineering, you now use to investigate new ML frameworks and model architectures, resulting in a much improved financial model, much to the delight of your boss.

The above example shows the difference between training a model to make a one-off prediction on a static dataset versus building a batch ML system - a system that automates reading from data sources, transforming data into features, training models, performing inference on new data with the model, and updating a dashboard with the model's predictions. The dashboard is the value delivered by the model to stakeholders.

If you want a model to generate repeated value, the model should make predictions more than once. That means, you are not finished when you have evaluated the model's performance on a test set drawn from your static dataset. Instead you will have to build ML pipelines, programs that transform raw data into features, and feed features to your model for easy retraining, and feed new features to your model so that it can make predictions, generating more value with every prediction it makes.

You have embarked on the same journey from training models on static datasets to building *ML systems*. The most important part of that journey is working with dynamic data, see figure 1. This means moving from static data, such as the hand curated datasets used in ML competitions found on Kaggle.com, to batch data, datasets that are updated at some interval (hourly, daily, weekly, yearly), to real-time data.

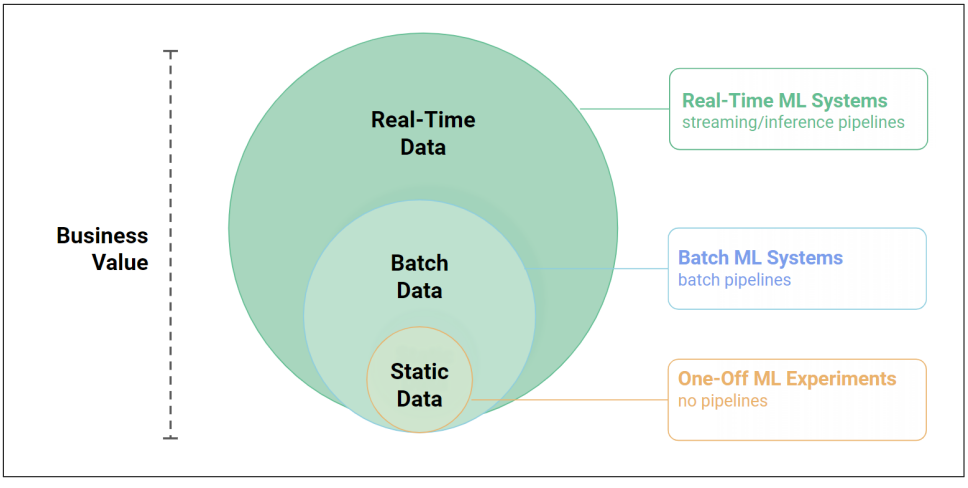


Figure 1-1. A ML system that only generates a one-off prediction on a static dataset generates less business value than a ML system that can make predictions on a schedule with batches of input data. ML systems that can make predictions with real-time data are more technically challenging, but can create even more business value.

A ML system is a software system that manages the two main life cycles for a model: training and inference (making predictions).

The Evolution of Machine Learning Systems

In the mid 2010s, revolutionary ML Systems started appearing in consumer Internet applications, such as image tagging in Facebook and Google Translate. The first generation of ML systems were either batch ML systems that make predictions on a schedule, see figure 2, or interactive online ML systems that make predictions in response to user actions, see figure 3.

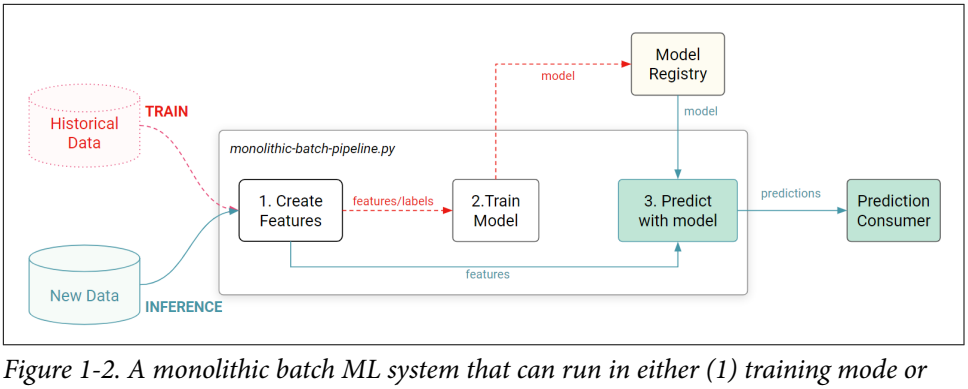


Figure 1-2. A monolithic batch ML system that can run in either (1) training mode or (2) inference mode.

Batch ML systems have to ensure that the features created for training data and the features created for batch inference are consistent. This can be achieved by building a monolith batch pipeline program that is run in either training mode or inference mode. The architecture ensures the same "Create Features" code is run in training and inference.

In figure 3, you can see an interactive ML system that receives requests from clients and responds with predictions in real-time. In this architecture, you need two separate systems - an offline training pipeline, and an online model serving service. You can no longer ensure consistent features between training and serving by having a single monolithic program. Early solutions to this problem involved versioning the feature creation source code and ensuring both training and serving use the same version, as in this Twitter presentation.

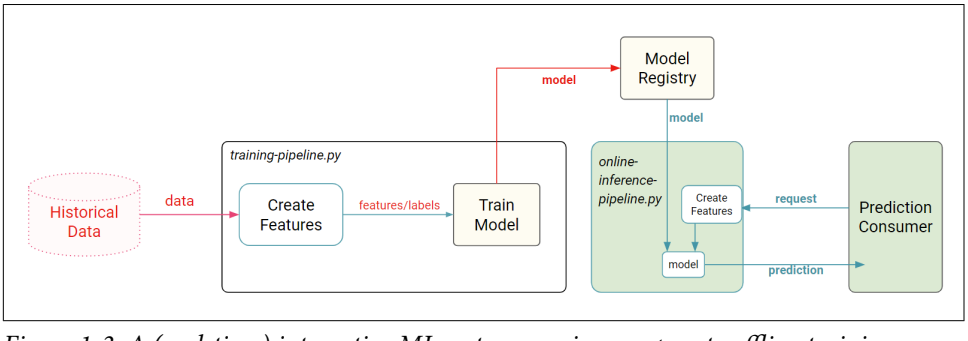


Figure 1-3. A (real-time) interactive ML system requires a separate offline training system from the online inference systems.

Notice that the online inference pipeline is stateless. We will see later than stateful online inference pipelines require adding a feature store to this architecture.

Stateless online ML systems were, and still are, acceptable for some use cases. For example, you can download a pre-trained large language model (LLM) and implement a chatbot using only the online inference pipeline - you don't need to implement the training pipeline - which probably cost millions of dollars to run on 100s or 1000s of GPUs. The online inference pipeline can be as simple as a Python program run on a web application server. The program will load the LLM into memory on startup and make predictions with the LLM on user input data in response to prediction requests. You will need to tokenize the user input prompt before calling predict on the model, but otherwise, you need almost no knowledge of ML to build the online inference service using an LLM.

However, a personalized LLM (or any ML system with personalized predictions) needs to integrate external data, in a process called retrieval augmentation generation (RAG). RAG enables the LLM to enrich its input prompt with historical data or contextual data. In addition to RAG, you can also collect the LLM responses and user

responses (the prediction logs), and with them you will be able to generate more training data to improve your LLM.

So, the general problem here is one of re–integration of the offline training system and the online inference system to build a stateful integrated ML system. That general problem has been addressed earlier by feature stores, introduced as a platform by Uber in 2018. The feature store for machine learning has been the key ML infrastructure platform in connecting the independent training and inference pipelines. One of the main motivations for the adoption of feature stores by organizations has been that they make state available to online inference programs, see figure 4. The feature store enables input to an online model to be augmented with historical and context data by low latency retrieval of precomputed feature data from the feature store. In general, feature stores enable richer, personalized online models compared to stateless online models. You can read more about feature stores in Chapters 4 and 5.

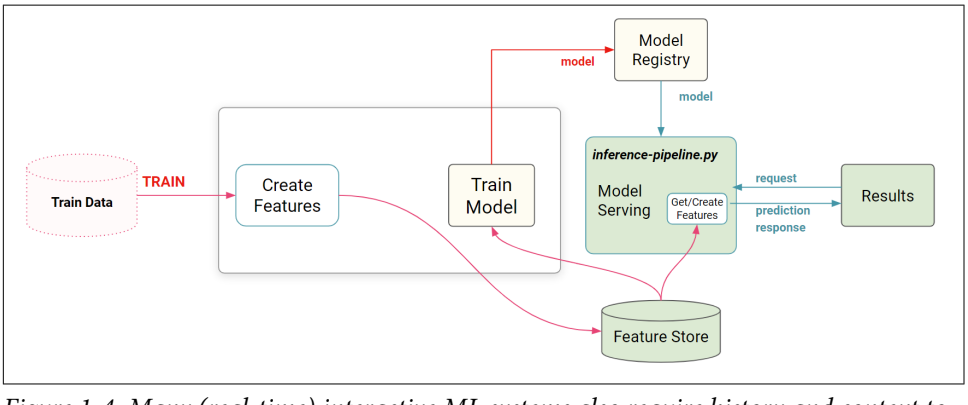


Figure 1-4. Many (real-time) interactive ML systems also require history and context to make personalized predictions. The feature store enables personalized history and context to be retrieved at low latency as precomputed features for online models.

The evolution of the ML system architectures described here, from batch to stateless real-time to real-time systems with a feature store, did not happen in a vacuum. It happened within a new field of machine learning engineering called machine learning operations (MLOps) that can be dated back to 2015, when authors at Google published a canonical paper entitled Hidden Technical Debt in Machine Learning Systems. The paper cemented in ML developers minds the adage that only a small percentage of the work in building ML systems was training models. Most of the work is in data management and building and operating the ML system infrastructure.

Inspired by the DevOps[1] movement in software engineering, MLOps is a set of practices and processes for building reliable and scalable ML systems that can be quickly and incrementally developed, tested, and rolled out to production using automation where possible. Some of the problems considered part of MLOps were addressed already in this section, such as how to ensure consistent feature data between training and inference. An O'Reilly book entitled "Machine Learning Design Patterns" published 30 patterns for building ML systems in 2020, and many problems related to testing, versioning, and monitoring features, models, and data have been identified by the MLOps community.

However, to date, there is no canonical MLOps architecture for ML systems. As of early 2024, Google and Databricks have competing MLOps architectures containing 26 and 28 components, respectively. These MLOps architectures more closely resemble the outdated enterprise waterfall lifecycle development model that DevOps helped replace, rather than the test-driven, start-small development culture of DevOps, which promotes getting to a working system as fast as possible.

MLOps is currently in a phase similar to the early years of databases, where developers were expected to understand the inner workings of magnetic disk drives in order to retrieve data with high performance. Instead of saying what data to retrieve with SQL, early database users had to tell databases how to read the data from disk. Similarly, most MLOps courses today assume that you need to build or deploy the ML infrastructure needed to run ML systems. That is, you start by setting up continuous integration systems, how to containerize your ML pipelines, how to automate the deployment of your ML infrastructure with Terraform, and how Kubernetes works. Then you only have to cover the remaining 20 other components identified for building reliable ML systems, before you can build your first ML system.

In this book we will build on existing widely deployed ML infrastructure, including a feature store to manage feature and label data for both training and inference, a model registry as a store for trained models, and a model serving platform to deploy online models behind a REST or gRPC API. In the examples covered in this book, we will work with (free) serverless versions of these platforms, so you will not have to learn infrastructure-as-code or Kubernetes to get started. Similarly, we will use serverless compute platforms so that you don't even have to containerize your code, meaning knowledge of Python is enough to be able to build the ML pipelines that will make up the ML systems you build that will run on (free) serverless ML infrastructure.

1 Wikipedia states that "DevOps integrates and automates the work of software development (Dev) and IT operations (Ops) as a means for improving and shortening the systems development life cycle."

The Anatomy of a Machine Learning System

One of the main challenges you will face in building ML systems is managing the data that is used to train models and the data that models make predictions with. We can categorize ML systems by how they process the new data that is used to make predictions with. Does the ML system make predictions on a schedule, for example, once per day, or does it run 24x7, making predictions in response to user requests?

For example, Spotify weekly is a *batch ML system*, a recommendation engine, that, once per week, predicts which songs you might want to listen to and updates them in your playlist. In a batch ML system, the ML system reads a batch of data (all 575m+ users in the case of Spotify), and makes predictions using the trained recommender ML model for all rows in the batch of data. The model takes all of the input features (such as how often you listen to music and the genres of music you listen to) and, for each user, makes a prediction of the 30 "best" songs for you for the upcoming week. The predictions are then stored in a database (Cassandra) and when the user logs on, the Spotify weekly recommendation list is downloaded from the database and shown as recommendations in the user interfaces.

Tiktok's recommendation engine, on the other hand, is famous for adapting its recommendations in near real-time as you click and watch their short-form videos. This is known as a *real-time ML system*. It predicts which videos to show you as you scroll and watch videos. Andrej Karpathy, ex head of AI at Tesla, said Tiktoks' recommendation engine "is scary good. It's digital crack". Tiktok described in its Monolith research paper how it both retrains models very frequently and also how it updates historical feature values used as input to models (what genre of video you viewed last, how long you watched it for, etc) in near real-time with stream-processing (Apache Flink). When Tiktok recommends videos to you, it uses a wealth of real-time data as well as any query your enter. Iyour recent viewing behavior (clicks, swipes, likes), your historical preferences, as well as recent context information (such as what videos are trending right now for users like you). Managing all of this user data in real-time and at scale is a significant engineering challenge. However, this engineering effort was rewarded as Tiktok were the first online video platform to include real-time recommendations, which gave them a competitive advantage over incumbents, enabling them to build the world's second most popular online video platform.

We will address head-on the data challenge in building ML systems. Your ML system may need different types of data to operate - including user input data, historical data, and context data. For example, a real-time ML system that predicts the validity of an insurance claim will take as input the details of the claim, but will augment this with the claimant's history and policy details, and further enrich this with context information about the current rate of claims for this particular policy. This ML system is a long way from the starting point where a Data Scientist received a static data dump and was asked if she could improve the detection of bogus insurance claims.

Types of Machine Learning

The main types of machine learning used in ML systems are supervised learning, unsupervised learning, self-supervised learning, semi-supervised learning, reinforcement learning, and in-context learning.

Supervised Learning
: In *supervised learning*, you train a model with data containing features and labels. Each row in a training dataset contains a set of input feature values and a label (the outcome, given the input feature values). Supervised ML algorithms learn relationships between the labels (also called the target variable) and the input feature values. Supervised ML is used to solve classification problems, where the ML system will answer yes-or-no questions (is there a hotdog in this photo?) or make a multiclass classification (what type of hotdog is this?). Supervised ML is also used to solve regression problems, where the model predicts a numeric value using the input feature values (estimate the price of this apartment, given input features such as its area, condition, and location). Finally, supervised ML is also used to fine-tune chatbots using open-source large language models (LLMs). For example, if you train a chatbot with questions (features) and answers (labels) from the legal profession, your chatbot can be fine-tuned so that it talks like a lawyer.

Unsupervised Learning
: In contrast, *unsupervised learning* algorithms learn from input features without any labels. For example, you could train an anomaly detection system with credit-card transactions, and if an anomalous credit-card transaction arrives, you could flag it as suspected for fraud.

Semi-supervised Learning
: In *semi-supervised learning*, you train a model with a dataset that includes both labeled and unlabeled data, usually mostly unlabeled. Semi-supervised ML combines supervised and unsupervised machine learning methods. Continuing our credit-card fraud detection example, if we had a small number of examples of fraudulent credit card transactions, we could use semi-supervised methods to improve our anomaly detection algorithm with examples of bad transactions. In credit-card fraud, there is typically an extreme imbalance between "good" and "bad" transactions (<0.001%), making it impractical to train a fraud detection model with only supervised ML.

Self-supervised Learning
: *Self-supervised learning* involves generating a labeled dataset from a fully unlabeled one. The main method to generate the labeled dataset is *masking*. For natural language processing (NLP), you can provide a piece of text and mask out individual words (Masked-Language Modeling) and train a model to predict the

missing word. Here, we know the label (the missing word), so we can train the model using any supervised learning algorithm. In NLP, you can also mask out entire sentences with next sentence prediction that can teach a model to understand longer-term dependencies across sentences. The language model BERT uses both masked-language modeling and next sentence prediction for training. Similarly, with image classification, you can mask out a (randomly chosen) small part of each image and then train a model to reproduce the original image with as high fidelity as possible.

Reinforcement Learning
: *Reinforcement learning* (RL) is another type of ML algorithm (not covered in this book). RL is concerned with learning how to make optimal decisions. In RL, an agent learns the best actions to take in an environment, by the environment giving the agent a reward after each action the agent executes. The agent then adapts its behavior to either maximize the rewards it receives (or minimizes the costs) for each action.

In-context Learning
: There is also a very recent type of ML found in large language models (LLMs) called *in-context learning*. Supervised ML, unsupervised ML, semi-supervised ML, and reinforcement learning can only learn with data they are trained on. That is, they can only solve tasks that they are trained to solve. However, LLMs that are large enough exhibit a different type of machine learning - *in-context learning* (ICL) - the ability to learn to solve new tasks by providing "training" examples in the prompt (input) to the LLM. LLMs can exhibit ICL even though they are trained only with the objective of next token prediction. The newly learnt skill is forgotten directly after the LLM sends its response - its model weights are not updated as they would be during training.

ChatGPT is a good example of a ML system that uses a combination of different types of ML. ChatGPT includes a LLM trained use self-supervised learning to train the foundation model, supervised learning to fine-tune the foundation model to create a task-specific model (such as a chatbot), and reinforcement learning (with human feedback) to align the task-specific model with human values (e.g., to remove bias and vulgarity in a chatbot). Finally, LLMs can learn from examples in the input prompt using in-context learning.

Data Sources

Data for ML systems can, in principle, come from any available data source. That said, some data sources and data formats are more popular as input to ML systems. In

this section, we introduce the data sources most commonly encountered in Enterprise computing.[2]

Tabular data

Tabular data is data stored as tables containing columns and rows, typically in a database. There are two main types of databases that are sources for data for machine learning:

- Relational databases or NoSQL databases, collectively known as *row-oriented data stores* as their storage layout is optimized for reading and writing rows of data;
- Analytical databases such as data warehouses and data lakehouses, collectively known as column-oriented data stores as their storage layout is optimized for reading and processing columns of data (such as computing the min/max/average/sum for a column).

Row-oriented databases are operational data stores that power a wide variety of applications that store their records (or rows) row-wise on disk or in-memory. Relational databases (such as MySQL or Postgres) store their data as rows as pages of data along with indexes (such as B-Trees and hash indexes) to efficiently find data. NoSQL data stores (such as Cassandra, and RocksDB) typically use log-structured merge trees (LSM Trees) to store their data along with indexes (such as Bloom filters) to efficiently find data. Some data stores (such as MongoDB) combine both B-Trees and LSM Trees. Some row-oriented databases are distributed, scaling out to run on many servers, some as servers on a single host, and some are embedded databases that are a library that can be included with your application.

From a developer perspective, the most important property of row-oriented databases is the data format you use to read and write data. Popular data formats include SQL and Object-Relational Mappers (ORM) for SQL (MySQL, Postgres), key-value pairs (Cassandra, RockDB), or JSON documents (MongoDB).

Analytical (or columnar) data stores are historical stores of record used for analysis of potentially large volumes of data. In Enterprises, data warehouses collect all the data stored in all operational data stores. Programs called data pipelines extract data from the operational data stores, transform the data into a format suitable for analysis and machine learning, and load the transformed data into the data warehouse or lakehouse. If the transformations are performed in the data pipeline (for example, a Spark or Airflow program) itself, then the data pipeline is called an *ETL pipeline* (extract,

2 Enterprise computing refers to the information storage and processing platforms that businesses use for operations, analytics, and data science.

transform, load). If the data pipeline first loads the data in the Data Warehouse and then performs the transformations in the Data Warehouse itself (using SQL), then it is called an *ELT pipeline* (extract, load, transform). Spark is a popular framework for writing ETL pipelines and DBT is a popular framework for writing ELT pipelines.

Columnar data stores are the most common data source for historical data for ML systems in Enterprises. Many data transformations for creating features, such as aggregations and feature extraction, can be efficiently and scalably implemented in DBT/SQL or Spark on data stored in data warehouses. Python frameworks for data transformations, such as Pandas 2+ and Polars, are also popular platforms for feature engineering with data of more reasonable scale (GBs, not TBs or more).

> A Lakehouse is a combination of (1) tables stored as columnar files in a data lake (object store or distributed file system) and (2) data processing that ensures ACID operations on the table for reading and writing that store columnar data. They are collectively known as Table File Formats. There are 3 popular open-source table formats: Apache Iceberg, Apache Hudi, and Delta Lake. All 3 provide similar functionality, enabling you to update the tabular data, delete rows from tables, and incrementally add data to tables. You no longer need to read up the old data, update it, and write back your new version of the table. Instead you can just append or upsert (insert or update) data into your tables.

Unstructured Data

Tabular data and graph data, stored in graph databases, are often referred to as structured data. Every other type of data is typically thrown into the antonymous bucket called *unstructured data*—text (pdfs, docs, html, etc), image, video, audio, and sensor-generated data are all considered unstructured data. The main characteristic of unstructured data is that it is typically stored in files, sometimes very large files of GBs or more, in low cost data stores, such as object stores or distributed file systems. The one type of data that can be either structured or unstructured is text data. If the text data is stored in files, such as markdown files, it is considered unstructured data. However, if the text is stored as columns in tables, it is considered structured data. Most text data in the Enterprise is unstructured and stored in files.

Deep learning has made huge strides in solving prediction problems with unstructured data. Image tagging services, self-driving cars, voice transcription systems, and many other ML systems are all trained with vast amounts of unstructured data. Apart from text data, this book, however, focuses on ML systems built with structured data that comes from feature stores.

Event Data

An event bus is a data platform that has become popular as (1) a store for real-time event data and (2) a data bus for storing data that is being moved or copied between different data stores. In this book, we will mostly consider event buses as the former, a data source for real-time ML systems. For example, at the consumer tech giants, every click you make on their website or mobile app, and every piece of data you enter is typically first sent to a massively scalable distributed event bus, such as Apache Kafka, from where real-time ML systems can use that data to create *fresh features* for models powering their ML-enabled applications.

API-Provided Data

More and more data is being stored and processed in Software-as-a-Service (SaaS) systems, and it is, therefore, becoming more important to be able to retrieve or scrape data from such services using their public application programming interfaces (APIs). Similarly, as society is becoming increasingly digitized, more data is becoming available on websites that can be scraped and used as a data source for ML systems. There are low-code software systems that know about the APIs to popular SaaS platforms (like Salesforce and Hubspot) and can pull data from those platforms into data warehouses, such as Airbyte. But sometimes, external APIs or websites will not have data integration support, and you will need to scrape the data. In Chapter 2, we will build an Air Quality Prediction ML System that scrapes data from the closest public Air Quality Sensor data source to where you live (there are tens of thousands of these available on the Internet today - probably one closer to you than you imagine).

Ethics and Laws for Data Sources

In addition to understanding how to collect data from your data sources, you also have to understand the laws, ethics, and organizational policies that govern this data. Does the data contain personally identifiable information (PII data)? Is use of the data for machine learning restricted by laws, such as GDPR or CCAP or the EU AI act? What are your organization's policies for the use of this data? It is also your responsibility as an individual to understand if the ML system you are building is ethical and that you personally follow a code of ethics for AI.

Incremental Datasets

Most of the challenges in building and operating ML systems are in managing the data. Despite this, data scientists have traditionally been taught machine learning with the simplest form of data: *immutable datasets*. Most machine learning courses and books point you to a dataset as a static file. If the file is small (a few GBs at most), the

file often contains comma-separated values (csv), and if the data is large (GBs to TBs), a more efficient file format, such as Parquet[3] is used.

For example, the well-known titanic passenger dataset[4] consists of the following files:

train.csv
: the training set you should use to train your model;

test.csv
: the test set you should use to evaluate the performance of your trained model.

The dataset is static, but you need to perform some basic feature engineering. There are some missing values, and some columns have no predictive power for the problem of predicting whether a given passenger survives the Titanic or not (such as the passenger ID and the passenger name). The Titanic dataset is popular as you can learn the basics of data cleaning, transforming data into features, and fitting a model to the data.

Immutable files are not suitable as the data layer of record in an enterprise environment where GDPR (the EU's General Data Protection Regulation) and CCPA (California Consumer Privacy Act) require that users are allowed to have their data deleted, updated, and its usage and provenance tracked. In recent years, open-source table formats for data lakes have appeared, such as Apache Iceberg, Apache Hudi, and Delta Laker, that support mutable datasets (that work with GDPR and CCPA) that are designed to work at massive scale (PBs in size) on low cost storage (object stores and distributed file systems).

In introductory ML courses, you do not typically learn about *incremental datasets*. An incremental dataset is a dataset that supports efficient appends, updates, and deletions. ML systems continually produce new data - whether once per year, day, hour, minute, or even second. ML systems need to support incremental datasets. In ML systems built with time-series data (for example, online consumer data), that data may also have *freshness* constraints, such that you need to periodically retrain your model so that it does not degrade in performance. So, we need to accumulate historical data in incremental datasets so that, over time, more training data becomes avail-

3 Parquet files store tabular data in a columnar format - the values for each column are stored together, enabling faster aggregate operations at the column level (such as the average value for a numerical column) and better compression, with both dictionary and run-length encoding.

4 The titanic dataset is a well-known example of a binary classification problem in machine learning, where you have to train a model to predict if a given passenger will survive or not.

able for re-training models to ensure high performance for our ML systems - models degrade over time if they are not periodically retrained using recent (fresh) data.

Incremental datasets introduce challenges for feature engineering. Some of the data transformations used to create features are parametrized by all of the feature data, such as feature encoding and scaling. This means that if we want to store encoded feature data in an incremental dataset, every time we write new feature data, we will have to re-encode all the feature data for that feature, causing massive *write amplification*. Write amplification is when writes (appends or updates) take increasingly longer as the dataset increases in size - it is not a good system property. That said, there are many data transformations in machine learning, traditionally called "data preparation steps", that are compatible with incremental datasets, such as aggregations, binning, and dimensionality reduction. In Chapters 6 and 7, we categorize data transformations for feature engineering as either (1) data transformations that create features stored in incremental datasets that are reusable across many models, and (2) data transformations that are not stored in incremental datasets and create features that are specific to one model.

What is an incremental dataset? In this book, we will not use the tried and tested and failed method of creating incremental datasets by storing the new data as a separate immutable file (*titanic_passengers_v1.csv,..., titanic_passengers_vN.csv*). Nor will we introduce write amplification by reading up the existing dataset, updating the dataset, and saving it back (for example, as parquet files). Instead, we will use a *feature store* and we append, update, and delete data in tables called *feature group*s. A detailed introduction to feature stores can be found in Chapters 4 and 5, but we will start using them already in Chapter 2.

The key technology for maintaining incremental datasets for ML is the pipeline. Pipelines collect and process the data that will be used to train our ML models. The pipeline is also what we will use to periodically retrain models. And we even use pipelines to automate the predictions produced by the batch ML systems that run on a schedule, for example, daily or hourly.

What is a ML Pipeline ?

A pipeline is a program that has well-defined inputs and outputs and is run either on a schedule or 24x7. ML Pipelines is a widely used term in ML engineering that loosely refers to the pipelines that are used to build and operate ML systems. However, a problem with the term ML pipeline is that it is not clear what the input and output to a ML pipeline is. Is the input raw data or training data? Is the model part of input or the output? In this book, we will use the term ML pipeline to refer collectively to any pipeline in a ML system. We will not use the term ML pipeline to refer to a specific stage in a ML system, such as feature engineering, model training, or inference.

An important property of ML systems is *modularity*. Modularity involves structuring your ML system such that its functionality is separated into independent components that can be independently run and tested. Modules should be kept small and easy to understand/document. Modules should enable reuse of functionality in ML systems, clear separation of work between teams, and better communication between those teams through shared understanding of the concepts and interfaces in the ML system.

In figure 5, we can see an example of a modular ML system that has factored its functionality into three independent ML pipelines: a feature pipeline, a training pipeline, and an inference pipeline.

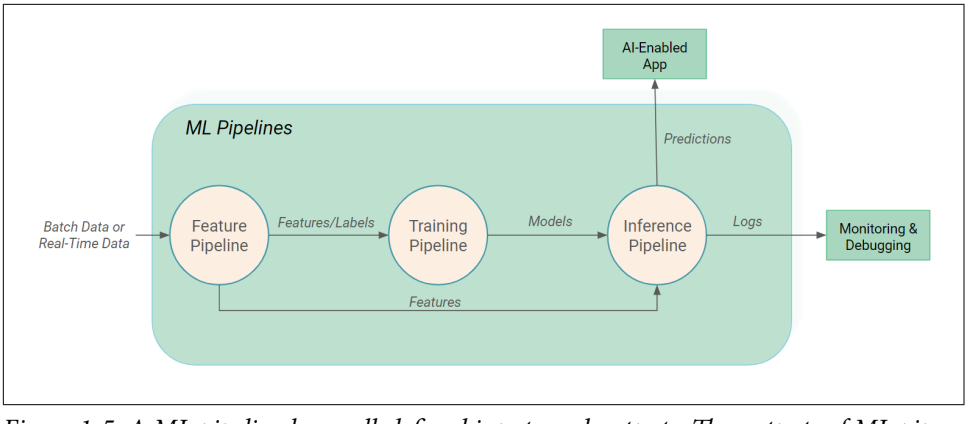


Figure 1-5. A ML pipeline has well-defined inputs and outputs. The outputs of ML pipelines can be inputs to other ML pipelines or to external ML Systems that use the predictions and prediction logs to make them "AI-enabled".

The three different pipelines have clear inputs and outputs and can be developed and operated independently:

- A *feature pipeline* takes data as input and produces reusable features as output.
- A *training pipeline* takes features as input trains a model and outputs the trained model.
- An *inference pipeline* takes features and a model as input and outputs predictions and prediction logs.

The feature pipeline is similar to an ETL or ELT data pipeline, except that its data transformation steps produce output data in a format that is suitable for training models. There are many common data transformation steps between data pipelines and feature pipelines, such as computing aggregations, but many transformations are specific to ML, such as dimensionality reduction and data validation checks specific to ML. Feature pipelines typically do not need GPUs, but run instead on commodity

CPUs. They are often written in frameworks such as DBT/SQL, Apache Spark, Apache Flink, Pandas, and Polars, and they are scheduled to run at defined intervals by some orchestration platform (such as Apache Airflow, Dagster, Modal, or Mage). Feature pipelines can also be streaming applications that run 24x7 and create fresh features for use in real-time ML systems. The output of feature pipelines are features that can be reused in one or model models. To ensure features are reusable, we do not encode or scale feature values in feature pipelines. Instead these transformations (called *model-dependent transformations* as they are parameterized by the training dataset), are performed consistently in the training and inference pipelines.

The training pipeline is typically a Python program that takes features (and labels for supervised learning) as input, trains a model (using GPUs for deep learning), and saves the model in a model registry. Before saving the model in the model registry, it is important to additionally validate that the model has good performance, is not biased against potential groups of users, and, in general, does nothing bad.

The inference pipeline is either a batch program or an online service, depending on whether the ML system is a batch system or a real-time system. For batch ML systems, the inference pipeline typically reads features computed by the feature pipeline and the model produced by the training pipeline, and then outputs the model's predictions for the input feature values. Batch inference pipelines are typically implemented in Python using either PySpark or Pandas/Polars, depending on the size of input data expected (PySpark is used when the input data is too large to fit on a single server). For real-time ML systems, the online inference pipeline is a program hosted as a service in *model serving infrastructure*. The model serving infrastructure receives user requests and invokes the online inference pipeline that can compute features using on user input data and enrich using pre-computed features and even features computed from external APIs. Online inference pipelines produce predictions that are sent as responses to client requests as well as *prediction log* entries containing the input feature values and the output prediction. Prediction logs are used to monitor the performance of ML systems and to provide logs for debugging ML systems. Another less common type of real-time ML system is a stream-processing system that uses a trained model to make predictions on features computed from streaming input data.

Building our first minimal viable ML system using feature, training, and inference pipelines is only the first step. You now need to iteratively improve this system to make it a production ML system. This means you should follow best practices in how to shorten your development loop while having high confidence that your changes will not break your ML system or clients of your ML system. For this, we will follow best practices from MLOps.

Notebooks as ML Pipelines?

Many software engineering problems arise with Jupyter/Colaboratory notebooks when you write ML pipelines as notebooks, including:

- There is a huge temptation to build a monolithic ML pipeline that does feature engineering, model training, and inference in one single notebook;
- Features are computed in cells making it impossible to write unit tests for the feature logic;
- Many orchestration engines do not support scheduling notebooks as jobs.

These problems can be overcome by following good software engineering practices, such as refactoring feature computation code into modules that are invoked by the notebook—the feature logic can then be unit tested with PyTest. Even if your notebook cannot be scheduled by an orchestrator, a common solution is convert the notebook to a Python program, for example, using *nbconvert*, and then run the cells in order from top to bottom.

Principles of MLOps

MLOps is a set of development and operational processes that enables ML Systems to be developed faster that results in more reliable software. MLOps should help you tighten the development loop between the time you make changes to software or data, test your changes, and then deploy those changes to production. Many developers with a data science background are intimidated by the systems focus of MLOps on automation, testing, and operations. In contrast, DevOps' northstar is to get to a minimal viable product as fast as possible - you shouldn't need to build the 26 or 28 MLOps components identified by Google and Databricks, respectively, to get started. This section is technology agnostic and discusses the MLOps principles to follow when building a ML system. You will ultimately need infrastructure support for the automated testing, versioning, and monitoring of ML artifacts, including features, models, and predictions, but here, we will first introduce the principles that transcend specific technologies.

The starting point for building reliable ML systems, by following MLOps principles, is testing. An important observation about ML systems is that they require more levels of testing than traditional software systems. Small bugs in data or code can easily cause a ML model to make incorrect predictions. ML systems require significant engineering effort to test and validate to make sure they produce high quality predictions and are free from bias. The testing pyramid shown in figure 6 shows that testing is needed throughout the ML system lifecycle from feature development to model training to model deployment.

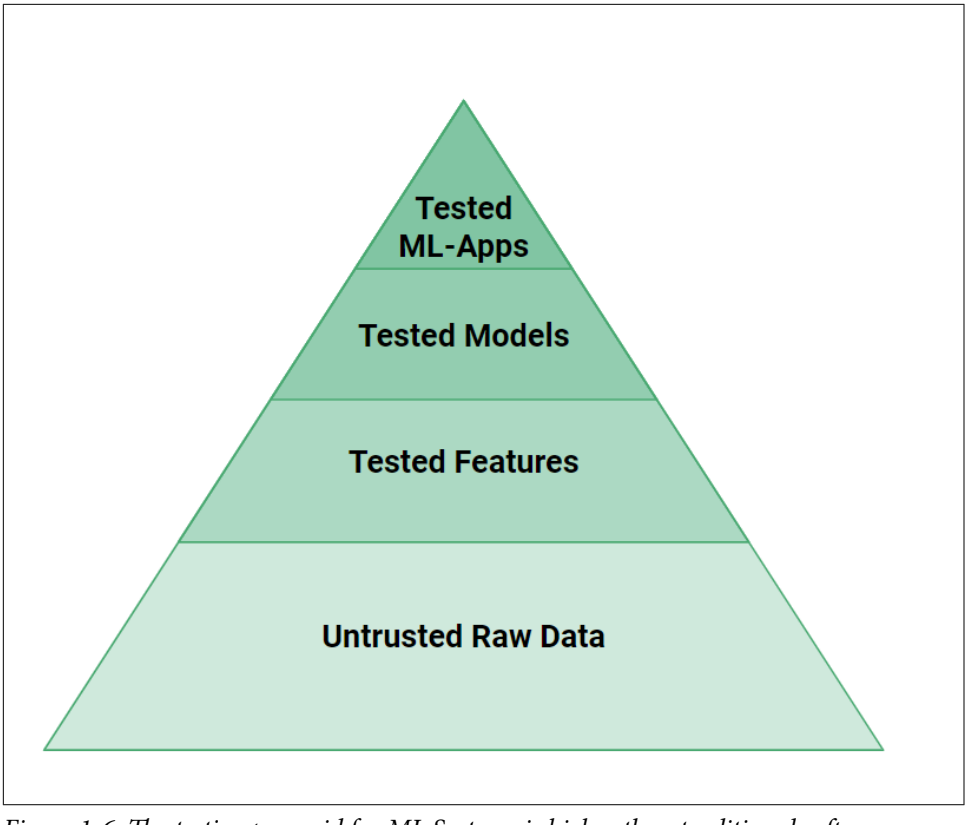


Figure 1-6. The testing pyramid for ML Systems is higher than traditional software systems, as both code and data need to be tested, not just code.

It is often said that the main difference between testing traditional software systems and ML systems is that in ML systems we need to test both the source-code and data - not just the source-code. The features created by feature pipelines can have their logic tested with unit tests and their input data checked with data validation tests, see Chapter 5. The models need to be tested for performance, but also for a lack of bias against known groups of vulnerable users, see Chapter 6. Finally, at the top of the pyramid, ML-Systems need to test their performance with A/B tests before they can switch to use a new model, see Chapter 7.

Given this background on testing and validating ML systems and the need for automated testing and deployment, and ignoring specific technologies, we can tease out the main principles for MLOps. We can express it as MLOps folks believe in:

- Automated testing of changes to your source code;
- Automated deployment of ML artifacts (features, training data, models);

- Validation of data ingested into your ML system;
- Versioning of ML artifacts;
- A/B testing ML artifacts;
- Monitoring the predictions, prediction quality, and SLAs (service-level agreements) for ML systems.

MLOps folks believe in testing their ML systems and that running those tests should have minimal friction on your development speed. That means automating the execution of your tests, with the tests helping ensure that changes to your code:

1. Do not introduce errors (it is important to catch errors early in a dynamically typed language like Python),
2. Do not break any client contracts (for example, changes to feature logic can break consumers of the feature data as can breaking schema changes for feature data or even SLA violations due to changes that result in slower code),
3. Integrates as expected with data sources and sinks (feature store, model registry, inference store), and
4. Do not introduce model bias or degrade model performance.

There are many DevOps platforms that can be used to implement continuous integration (CI) and continuous training (CT). Popular platforms for CI are Github Actions, Jenkins, and Azure DevOps. An important point is that support for CI and CT are not a prerequisite to start building ML systems. If you have a data science background, comprehensive testing is something you may not have experience with, and it is ok to take time to incrementally add testing to both your arsenal and to the ML systems you build. You can start with unit tests for functions (such as how to compute features), model performance and bias testing your training pipeline, and add integration tests for ML pipelines. You can automate your tests by adding CI support to run your tests whenever you push code to your source code repository. Support for testing and automated testing can come after you have built your first minimal viable ML System to validate that what you built is worth maintaining.

MLOps folks love that feeling when you push changes in your source code, and your ML artifact or system is automatically deployed. Deployments are often associated with the concept of development (dev), pre-production (preprod), and production (prod) environments. ML assets are developed in the dev environment, tested in preprod, and tested again before for deployment in the prod environment. Although a human may ultimately have to sign off on deploying a ML artifact to production, the steps should be automated in a process known as continuous deployment (CD). In this book, we work with the philosophy that you can build, test, and run your whole ML system in dev, preprod, or prod environments. The data your ML system can access will be dependent on which environment you deploy in (only prod has access

to production data). We will start by first learning to build and operate a ML system, then look at CD in Chapter 12.

MLOps folks generally live by the database community maxim of "garbage-in, garbage-out". Many ML systems use data that has few or no guarantees on its quality, and blindly ingesting garbage data will lead to trained models that predict garbage. The MLOps philosophy deems that rather requiring users or clients to clean the data after it has arrived, you should validate all input data before it is made accessible to users or clients of your system. In Chapter 5, we will dive into how to design and write data validation tests and run them in feature and inference pipelines (these are the pipelines that feed external data to your ML system). We will look at what mitigating actions we can take if we identify data as incorrect, missing, or corrupt.

MLOps is also concerned with operating ML systems - running, maintaining, and updating systems. In particular, updating ML systems has historically been a very complex, manual procedure where new models are rolled out in stages, checking for errors and model performance at each stage. MLOps folks dream of a ML system with a big green button and a big red button. The big green button upgrades your system, and the big red button rolls back the most recent upgrade, see figure 7. Versioning of ML artifacts is a necessary prerequisite for the big green and red buttons. Versioning enables ML systems to be upgraded without downtime, to support rollback after failed upgrades, and to support A/B testing.

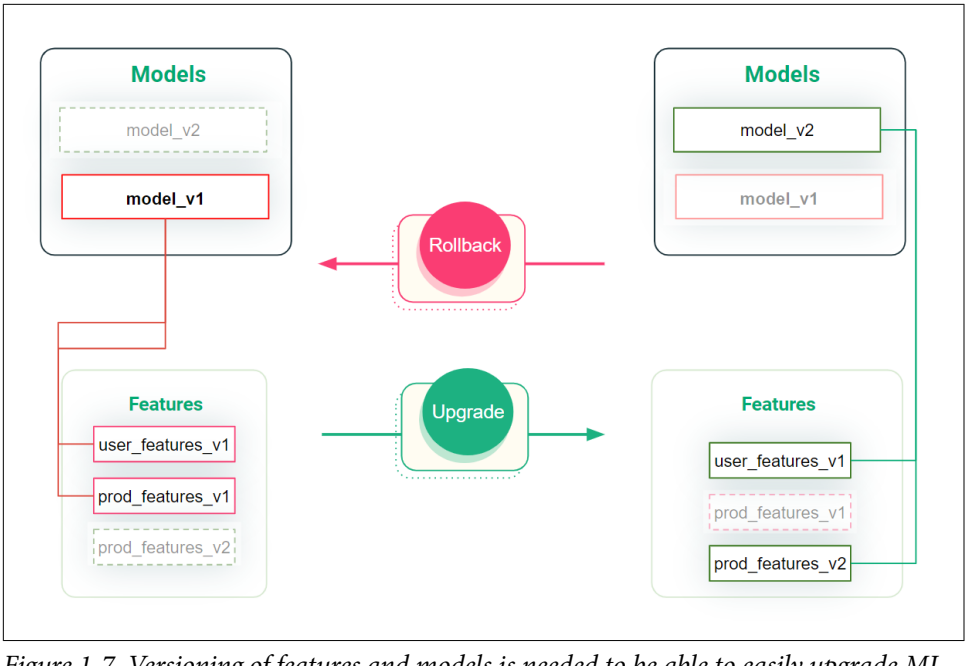


Figure 1-7. Versioning of features and models is needed to be able to easily upgrade ML systems and rollback upgrades in case of failure.

Versioning enables you to simultaneously support multiple versions of the same feature or model, enabling you to develop a new version, while supporting an older version in production. Versioning also enables you to be confident if problems arise after deploying your changes to production, that you can quickly rollback your changes to a working earlier version (of the model and features that feed it).

MLOps folks love to experiment, especially in production. A/B testing is important for ensuring continual delivery of service for a ML system that supports upgrades. A/B testing requires versioning of ML artifacts, so that you can run two versions in parallel. Models are connected to features, so we need to version both features and models as well as training data.

Finally, MLOps folks love to know how their ML systems are performing and to be able to quickly troubleshoot by inspecting logs. Operations teams refer to this as observability for your ML system. A production ML system should collect metrics to build dashboards and alerts for:

1. Monitoring the quality of your models' predictions with respect to some business key performance indicator (KPI),
2. Monitoring the quality/distribution of new data arriving in the ML system,
3. Measuring the performance of your ML system's components (model serving, feature store, ML pipelines)

Your ML system should provide service-level agreements (SLAs) for its performance, such as responding to a prediction request within 100ms or to retrieve 100 precomputed features from the feature store in less than 10ms. Observability is also about logging, not just metrics. Can Data Scientists quickly inspect model prediction logs to debug errors and understand model behavior in production - and, in particular, any anomalous predictions made by models? Prediction logs can also be collected for the goal of creating new training data for models.

In chapters 12 and 13, we go into detail of the different methods and frameworks that can help implement MLOps processes for ML systems with a feature store.

Machine Learning Systems with a Feature Store

A machine learning system is a platform that includes both the ML pipelines and the data infrastructure needed to manage the ML assets (reusable features, training data, and models) produced and consumed by feature engineering, model training, and inference pipelines, see figure 8. When a feature store is used with a ML system, it stores both the historical data used to train models as well as the latest feature data used to make predictions (model inference). It provides two different APIs for reading feature data - a batch API to efficiently read large volumes of feature data and an realtime API to read the latest feature data at low latency.

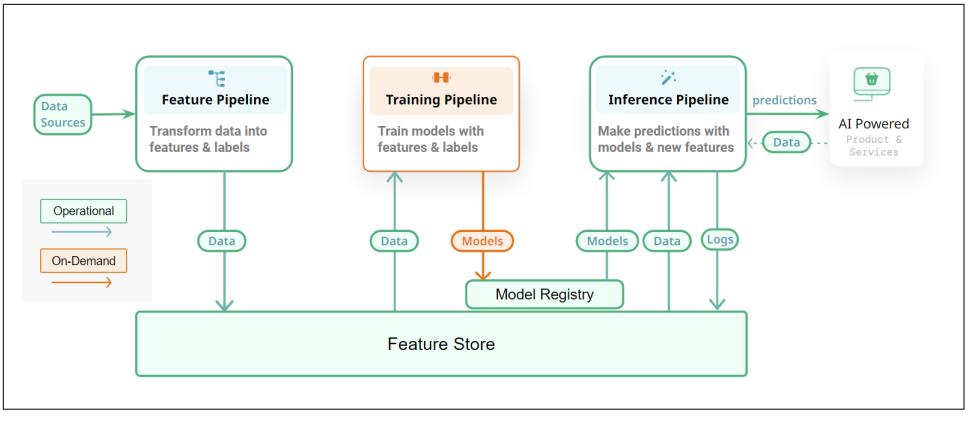


Figure 1-8. A ML system with a feature store supports 3 different types of ML pipeline: a feature pipeline, a training pipeline, and inference pipeline. Logging pipelines help implement observability for ML systems.

While the feature store stores feature data for ML pipelines, the model registry is the storage layer for trained models. The ML pipelines in a ML system can be run on potentially any compute platform. Many different compute engines are used for feature pipelines - including SQL, Spark, Flink, and Python - and whether they are batch or streaming pipelines, they typically are operational services that need to either run on a schedule (batch) or 24x7 (streaming). Training pipelines are most commonly implemented in Python, as are online inference pipelines. Batch inference pipelines can be Python, PySpark, or even a streaming compute engine or SQL database.

Given that this is the canonical architecture for ML systems with a feature store, we can identify four main types of ML systems with this architecture.

Three Types of ML System with a Feature Store

A ML system is defined by how it computes its predictions, not by the type of application that consumes the predictions. Given that, Machine learning (ML) systems that use a feature store can be categorized into three different types:

1. *Real-time interactive* ML systems make predictions in response to user requests using fresh feature data (at most a few seconds old). They ensure fresh features either by computing features on-demand from request input data or by updating precomputed features in an online feature store using stream processing;
2. Batch ML systems run on a schedule, running batch inference pipelines that take new feature data and a model to make predictions that are typically stored in some downstream database (called an inference store), to be later consumed by some ML-enabled application;

3. Stream processing ML systems use an embedded model to make predictions on streaming data. They may also enrich their stream data with historical or contextual precomputed features retrieved from a feature store;

Real-time, interactive applications differ from the other systems as they can use models as network hosted request/response services on model serving infrastructure. The other systems use an embedded model, downloaded from the model registry, that they invoke via a function call or an inter-process call. Real-time, interactive applications can also use an embedded model, if model-serving infrastructure is not available or if very low latency predictions are needed.

Embedded/Edge ML Systems

The other type of ML system, not covered in this book, is *embedded/edge* applications. They typically use an embedded model and compute features from their rich input data (often sensor data, such as images), typically without a feature store. For example, Tesla Autopilot is a driver assist system that uses sensors from cameras and other systems to help the ML models to make predictions about what driving actions to take (steering direction, acceleration, braking, etc). Edge ML Systems are real-time ML systems that run on resource-constrained network detached devices. For example, Tetra Pak has an image classification system that runs on the factory floor, identifying anomalies in cartons.

The following are some examples for the three different types of ML systems that use a feature store:

Real-Time ML Systems

ChatGPT is an example of an interactive system that takes user input (a prompt) and uses a LLM to generate a response, sent as an answer in text.

A credit-card fraud prevention system that takes a credit card transaction, and then retrieves precomputed features about recent use of the credit card from a feature store, then predicts whether the transaction is suspected of fraud or not, letting the transaction proceed if it is not suspected of fraud.

Batch ML Systems

An air quality prediction dashboard shows air quality forecasts for a location. It is built from predictions made by a batch ML system that uses observations of air quality from sensors and weather data as features. A trained model can predict air quality by using a weather forecast (input features) to predict air quality. This will be the first example ML system that we build in Chapter 3.

Google Photos Search is an interactive system that uses predictions made by a batch ML system. When your photos are uploaded to Google Photos, a classifica-

tion model is used to tag parts of the photo. Those tags (things/people/places) are indexed against the photo, so that you can later search in free-text on Google Photos to find photos that match your search query. For example, if you type in "bike", it will show you your photos that have one or more bicycles in them.

Stream Processing ML Systems
: Network intrusion detection is a real-time pattern matching problem that does not require user input. You can use stream processing to extract features about all traffic in a network, and then in your stream processing code, you can use a model to predict anomalies such as network intrusion.

ML Frameworks and ML Infrastructure used in this book

In this book, we will build ML systems using programs written in Python. Given that we aim to build ML systems, not the ML infrastructure underpinning it, we have to make decisions about what platforms to cover in this book. Given space restrictions in this book, we have to restrict ourselves to a set of well-motivated choices.

For programming, we chose Python as it is accessible to developers, the dominant language of Data Science, and increasingly important in data engineering. We will use open-source frameworks in Python, including Pandas and Polars for feature engineering, Scikit-Learn and PyTorch for machine learning, and KServe for model serving. Python can be used for everything from creating features from raw data, to model training, to developing user interfaces for our ML systems. We will also use pre-trained LLMs - open-source foundation models. When appropriate, we will also provide examples using other programming frameworks or languages widely used in the Enterprise, such as Spark and DBT/SQL for scalable data processing, and stream processing frameworks for real-time ML systems. That said, the example ML Systems presented in this book were developed such that only knowledge of Python is a prerequisite.

To run our Python programs as pipelines in the cloud, we will use serverless platforms, such as Modal and Github Actions. Both Github and Modal offer a free tier (Model requires credit card registration, though) that will enable you to run the ML pipelines introduced in this book. Again, the ML pipeline examples could easily be ported to run on containerized runtimes such as Kubernetes or serverless runtimes, such as AWS Lambda. Another free alternative is Github Actions. Currently, I think that Modal has the best developer experience of available platforms, hence its inclusion here.

For exploratory data analysis, model training, and other non-operational services, we will use open-source Jupyter notebooks. Finally, for (serverless) user interfaces hosted in the cloud, we will use Streamlit which also provides a free cloud tier. An alternative would be Hugging Face Spaces and Gradio.

For ML infrastructure, we will use Hopsworks as serverless ML infrastructure, using its feature store, model registry, and model serving platform to manage features and models. Hopsworks is open-source, was the first open-source and enterprise feature store, and has a free tier for its serverless platform. The other reason for using Hopsworks is that I am one of the developers of Hopsworks, so I can provide deeper insights into its inner workings as a representative ML infrastructure platform. With Hopsworks free serverless tier, that you can use to deploy and operate your ML systems without cost or the need to install or operate ML infrastructure platforms. That said, given all of the examples are in common open-source Python frameworks, you can easily modify the provided examples to replace Hopsworks with any combination of an existing feature store, such as FEAST, model registry and model serving platform, such as MLFlow.

Summary

In this chapter, we introduced ML systems with a feature store. We introduced the main properties of ML systems, their architecture, and the ML pipelines that power them. We introduced MLOps and its historical evolution as a set of best practices for developing and evolving ML systems, and we presented a new architecture for ML systems as feature, training, and inference (FTI) pipelines connected with a feature store. In the next chapter, we will look closer at this new FTI architecture for building ML systems, and how you can build ML systems faster and more reliably as connected FTI pipelines.

CHAPTER 2
Machine Learning Pipelines

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. The GitHub repo can be found at *https://github.com/featurestorebook/mlfs-book*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

In 1968, Edsger Dijkstra published an influential letter in the Communications of the ACM entitled "Go To Statement Considered Harmful" to highlight the excessive use of the GOTO statement in programming languages.[1] In 2024, the term "machine learning pipeline" is often used as a catch-all term to describe how to productionize ML models. However, there is currently widespread confusion about what a ML pipeline is and what it is not. What are the inputs and outputs to a ML pipeline? If somebody says they built their ML system using a ML pipeline what information can you glean from that? As such, the term ML pipelines, as it is currently used, could be "considered harmful" when communicating about building ML systems. Instead, we will strive to describe ML systems in terms of the actual pipelines used to build it. We provide a rigorous definition of different ML pipelines and describe how to modula-

1 Edsger Dijkstra (March 1968). "Go To Statement Considered Harmful" (PDF). Communications of the ACM. 11 (3): 147–148. doi:10.1145/362929.362947. S2CID 17469809

rize your ML system using ML pipelines that communicate via the feature store, model registry, and model-serving infrastructure.

Let's begin with pipelines. A pipeline is a computer program that has clearly defined inputs and outputs (that is, it has a well-defined interface) and it either runs on a schedule or continuously. A machine learning pipeline is any pipeline that outputs ML artifacts used in a ML system. You can modularize a ML system by connecting independent ML pipelines together - a feature pipeline to create feature data, a training data pipeline to create training data from feature data and labels, a model training pipeline to read training data and create a model, and a batch inference pipeline that reads feature (inference) data and a model and outputs predictions to some sink for use by an AI-enabled application.

When we talk about ML pipelines, we talk abstractly about the pipelines that create ML artifacts. We typically name a concrete ML pipeline after the ML artifact(s) they create - a feature pipeline, a (model) training pipeline or an inference (predictions) pipeline. Occasionally, you may name a ML pipeline based on how they modify a ML artifact - such as a model or feature validation pipeline that asynchronously validates a model or feature data, respectively. In this chapter, we cover many of the different possible ML pipelines, but we will double click on the most important ML pipelines for building a ML system - feature pipelines, training pipelines, and inference pipelines. Three pipelines and the truth.

Building ML Systems with ML Pipelines

Before we develop our first ML pipelines, we will look at how we build ML systems. ML systems are software systems, and software engineering methodologies help guide you when building software systems. For example, DevOps is a software engineering methodology that integrates software development and operations to build, test, and release software faster using automation, versioning, source code control, and separate development and production environments.

The first generation of software development processes for machine learning, such as Microsoft's Team Data Science Process, concentrated primarily on data collection and modeling, but did not address how to build ML systems. As such, they were quickly superseded by MLOps, which focuses on automation, versioning, and collaboration between developers and operations to build ML systems. As discussed in Chapter 1, modular ML systems are also key for MLOps.

Minimal Viable Prediction Service (MVPS)

We introduce here a minimal MLOps development methodology based on getting as quickly as possible to a minimal viable ML system, or MVPS (minimal viable prediction service). I followed this MVPS process in my course on building ML systems at

KTH, and it has enabled students to get to a working ML system (that uses a novel data source to solve a novel prediction problem) within a few days, at most.

ML artifacts include models, features, training data, experiment tracking data, model deployments, predictions, prediction logs. ML artifacts are stateful objects that are produced by ML pipelines and are managed by your ML infrastructure services. All ML artifacts are immutable, except for feature data, which is mutable as it is updated over time, and model deployments that can be A/B tested and upgraded. ML artifacts can be used by other internal ML pipelines or by external clients of the ML system. For example, features in a feature store are used in training pipelines and online inference pipelines by interactive applications.

MVPS Process

The MVPS development process, illustrated in Figure 2-1, starts with

- Identifying the prediction problem you want to solve
- The KPIs (key performance indicators) you want to improve
- The data sources you have available for use.

Once you have identified these three pillars that make up your ML system, you will need to map your prediction problem to a ML proxy metric - a target you will optimize in your ML system. This is often the most challenging step.

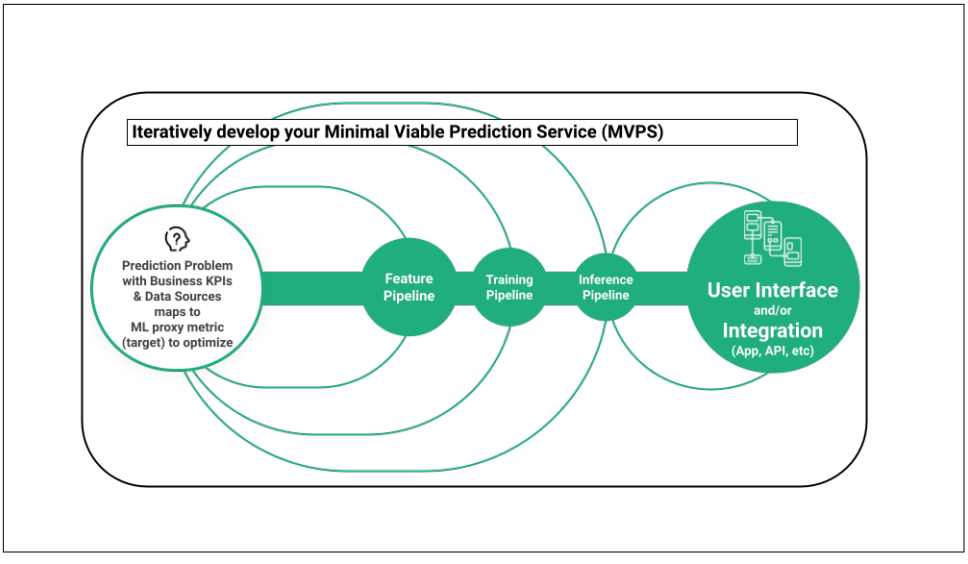


Figure 2-1. The MVPS process for developing machine learning systems starts in the leftmost circle by identifying a prediction problem, how to measure its success using KPIs, and how to map it onto a ML proxy metric. Based on the identified prediction problem and data sources, you implement the feature/training/inference pipeline, as well as either a user interface or integration with an external system that consumes the prediction. The arcs connecting the circles represent the iterative nature of the development process, where you often revise your pipelines based on user feedback and changes to requirements.

For example, you might want to predict items or content that a user is interested in. For recommending items in an e-commerce store, the KPI could be increased conversion as measured by users placing items in their shopping cart. For content, a measurable business KPI could be to maximize user engagement, as measured by the time a user spends on the service. Your goal as a data scientist or ML engineer is to take the prediction problem and business KPIs and translate them into a ML system that optimizes some ML metric (or *target*). The ML metric might be a direct match to business KPI - the probability that a user places an item in a shopping cart, or the ML metric might be proxy metric for the business KPI - the expected time a user will engage with a recommended piece of content (a proxy for increasing user engagement on the platform).

Once you have your prediction problem, KPIs, and ML target, you need to think about how to create training data with features that have predictive power for your target, based on your available data. You should start by enumerating and obtaining access to the data sources that feed your ML system. You then need to understand the data, so that you can effectively create features from that data. Exploratory data analy-

sis (EDA) is a first step you often take to gain an understanding of your data, its quality, and if there is a dependency between any features and the target variable. EDA typically helps develop domain knowledge of the data, if you are not yet familiar with the domain. It can help you identify which variables could or should be used or created for a model and their predictive power for the model. You can start EDA by examining your data and its distributions in a feature store (or Kaggle), and move on performing EDA in notebooks if needed, visually analyzing the data.

Once you have a reasonable understanding of your data and the features you need, you have to extract both the target observations (or labels) and features from your data sources. This involves building feature pipelines from your data sources. The output of your feature pipelines will be the features (and observations/labels) that are stored in a feature store. If you are fortunate enough that your feature store already contains the target(s) and/or features you need for your prediction problem, you can skip implementing the feature pipelines.

From the feature store, you can create your training data, and then implement a training pipeline to train your model that you save to a model registry. Finally, you implement an inference pipeline that uses your model and new feature data to make predictions, and add a UI or dashboard to create your minimal viable prediction service. This MVPS development process is iterative, as you incrementally improve the feature, training, and inference pipelines. You add testing, validation, and automation. You can later add different environments for development, staging, and production.

The next (unavoidable) step is to identify the different technologies you will use to build the feature, training, and inference pipelines, see Figure 2-2. We recommend using a Kanban board for this. A Kanban board is a visual tool that will track work as it moves through the MVPS process, featuring columns for different stages and cards for individual tasks. Atlassian JIRA and Github projects are examples of Kanban boards, widely used by developers.

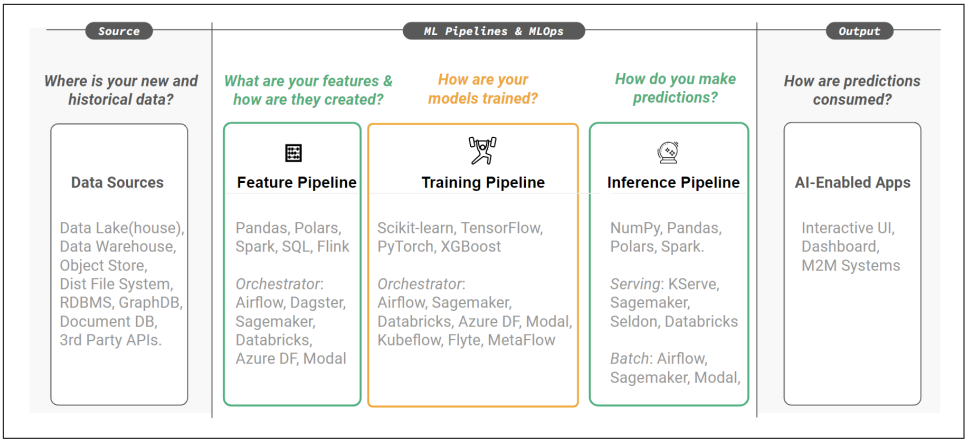


Figure 2-2. The Kanban board for our MVPS identifies the potential data sources, technologies used for ML pipelines, and types of consumers of predictions produced by ML systems. Here, we show some of the possible data sources, frameworks and orchestrators used in ML pipelines, and AI apps that consume predictions.

It is a good activity to fill in the MVPS Kanban board before starting your project to get an overview of the ML system you are building. You should entitle the Kanban board with the name of the prediction problem your ML system solves, then fill in the data sources, the AI applications that will consume the predictions, and the technologies you will use to implement the feature/training/inference pipelines. You can also annotate the different Kanban lanes with non-functional requirements, such as the volume, velocity, and freshness requirements for the feature pipelines, or the SLO (service-level objective) for the response times for an online inference pipeline. After we have captured the requirements for our ML system, we move on to writing code.

Wanted: Modular Code for Machine Learning Pipelines

A successful ML system will need to be updated and maintained over time. That means you will need to make any changes to your source code, such as:

1. The set of features computed or the data they are computed from;
2. How you train the model (its model architecture or hyperparameters) to improve its performance or reduce any bias;
3. For batch ML systems, make predictions more (or less) frequently or change the sink where you save your predictions;
4. For online ML systems, changes in the request latency or feature freshness requirements.

Now, imagine you had developed your system as a monolithic batch ML pipeline or a couple of separated programs with non DRY (do not repeat yourself) source code. How are you going to make sure the changes you make work correctly before you deploy the changed code? How are you going to on-board a new developer to work on the codebase?

The solution is to have a modular architecture and codebase. Modularity enables a software system to have its components separated and recombined. For example, source code can be factored into functions that each encapsulate a piece of work, and those functions can then be reused in different parts of a codebase. You hide the piece of code in the function (with all of its complexity) behind an interface. In Python, the interface to a function is the function's signature - its name, parameters, and return type(s). This interface provides a contract to clients that use the function - you will not change the function such that you break the expectations of clients. Modularity and encapsulation enable you to reduce complexity in a software system by decomposing a system into more manageable parts and hiding the complexity of each part behind an interface.

At the system architecture level, we can modularize the ML system into our 3 (or more) pipelines - feature pipeline, training pipeline, and inference pipelines. The pipeline is our abstraction and the interface is the input and output of each pipeline. But that is not enough modularization to build a maintainable, understandable software system.

Imagine we write a feature pipeline, computing data transformations in Pandas, in Example 2-1.

Example 2-1. Example of non-modular feature engineering code in Pandas. The method compute_features creates five different features that are not independently testable or documented.

```
import pandas as pd
def compute_features(df: pd.Dataframe): -> pd.Dataframe
if config["region"] == "UK":
df["holidays"] = is_uk_holiday (df["year"], df[" week"])
else:
df["holidays"] = is_holiday (df["year"], df ["week"])
df["avg_3wk_spend"] = df["spend"].rolling (3).mean()
df["acquisition_cost"] = df["spend"]/df["signups"]
df["spend_shift_3weeks"] = df["spend"].shift(3)
df["special_feature1"] = compute_bespoke_feature(df)
return df

df = pd.read_parquet("my_table.parquet")
df = compute_features(df)
```

This code snippet is not modular, as one function computes five features. It is difficult to test the individual features computed in the above code. It is challenging to independently update the individual features computed in the above code. It is difficult to understand the features the function `compute_features` computes. It is difficult to debug individual feature computations.

The team at DAGster behind the open-source Hamilton framework proposed a solution to refactor your Python source code as *feature functions* that update a DataFrame containing the features. For each feature computed, you define a new feature function. The features are created in a DataFrame (Pandas, PySpark, or Polars) by applying the feature functions in the correct order, and that featurized DataFrame is then used for training and inference.

We will follow the feature functions approach to build featurized DataFrames, but our feature pipelines will store the DataFrame in a feature group in the feature store, so that they can later be used for training and inference. Our approach to write modular feature engineering is to build a DataFrame containing feature data using feature functions (featurized DataFrame), see Figure 2-3. Each featurized DataFrame is written to a feature group in the feature store as a "commit" (append/update/delete). The feature group stores the mutable set of features created over time. Training and Inference steps can later use a feature query service to read a consistent snapshot of feature data from one or more feature groups to train a model or to make predictions, respectively.

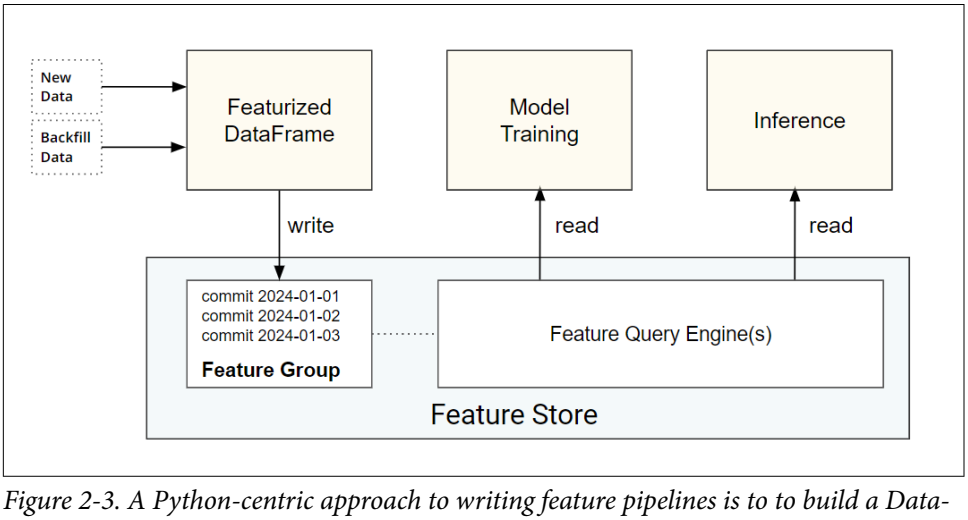


Figure 2-3. A Python-centric approach to writing feature pipelines is to to build a DataFrame and write it to a feature group in the feature store. The data can later be read from feature groups by training and inference pipelines using a feature query engine or service.

The approach to modularize your feature logic is as follows. For every feature computed as a column in the Pandas DataFrame, we have some feature logic. For example, here, we compute the column `aquisition_cost` as the *spend* divided by the number of users who sign up to our service (`signups`):

```
df['aquisition_cost'] = df['spend'] / df['signups']
```

We refactor the logic used to compute the `aquisition_cost` into a *feature function* as follows:

```
def aquisition_cost(spend: pd.Series, signups: pd.Series) -> pd.Series:
    """Acquisition cost per user is total spend divided by number of signups."""
    return spend / signups
```

At first glance, this increases the number of lines of code we have to write. However, now we have a documented function that can potentially be reused by different programs. We can now write a unit test for our `aquisition_cost` feature, as follows:

```
@pytest.fixture
def get_spends(self) -> pd.DataFrame:
    return pd.DataFrame([[20, 40], [5, 4], [4, 10],
        columns=["spends", "signups", "aquisition_cost"])
def test_spend_per_signup (get_spends : Callable):
    df=get_spends()
    df["res"] = aquisition_cost(df["spends"), df["signups"])
    pd.testing.assert_series_equal(df["res"], df["aquisition_cost"])
```

This unit test enforces a contract for how the `acquisition_cost` feature is computed - if you or another developer changes how to compute the `acquisition_cost`, the unit test below would fail, indicating its contract is broken for downstream clients that use the feature. You can, of course, update the feature logic for `acquisition_cost`, but that should typically be performed by creating a new version of the feature, and the new version would require a new unit test. We will cover versioning features in Chapter 4 on feature stores.

We will apply this method for modularizing feature logic code into feature functions for all data transformations performed using Python in this book. In the next section, we will see that building modular ML systems also requires you to know the type of feature you are creating with a data transformation - a reusable feature, a model-specific feature, or an on-demand feature.

Normally, I would advocate using Google Colaboratory to run notebooks, but in its current state in early 2024, you cannot easily import Python modules from files external to your notebook. For example, you can't store your *.ipynb* notebook in the same directory as a *my_functions.py* file in a Github repository, and then checkout your Colaboratory notebook and call '*import my_functions*' in your notebook. However, this works fine with Jupyter notebooks, so we will use Jupyter instead - it is best practice to store feature functions in Python modules, so they can be independently unit-tested and reused in different ML pipelines.

A Taxonomy for Data Transformations in ML Pipelines

Data transformations are key to ML systems. ML systems read in data and progressively perform transformations on the data (cleaning, mapping, reformatting, compressing) until the data is fitted to a model. ML systems also perform inference, reading in new data to make predictions with, and apply the same transformations that were used in training to create the features, and then making predictions on the new data with the trained model.

In monolithic ML pipelines, exactly the same data transformations are executed in the feature engineering, training, and inference phases, as they are performed in the same program with the same code. In other words, in a monolithic ML pipeline, all data transformations are essentially equivalent. However, when you break up your monolithic ML pipeline by adding a feature store to the mix, you quickly see that not all data transformations are equivalent - you can't just refactor your monolith to put all data transformations in feature pipelines. Let's examine why.

Firstly, the feature store should store features that can be reused across many models. That means feature pipelines should create reusable features. This leads many Data Scientists to the reasonable question - "should I store encoded feature data in the feature store?". The answer, as we will examine in detail in the next section, is that we should not, in general, store encoded feature data in the feature store. Feature encoding is a data transformation that is parameterized by a model's training dataset and the output feature data is, therefore, not reusable across many models - it is specific to that model (and its training data).

Another data transformation that needs to be performed outside of a feature pipeline is a real-time data transformation on input only available at request-time. These on-demand transformations are performed in online inference pipelines (for example, with a Python user-defined function or a SQL query). But, what if we want to reuse the same feature logic from the online inference pipeline to compute (or *backfill*) feature data in our feature pipeline using historical data?

To address both of these challenges, we now introduce a taxonomy for data transformations in ML pipelines that use a feature store. The taxonomy organizes data transformations into 3 different groups (model-dependent, model-independent, and on-demand transformations), informing you in which ML pipeline(s) to implement the data transformation. But, before looking at the taxonomy, we will first introduce data transformations from data science that are parameterized by training data - the encoding, scaling, and normalizing of feature data.

Feature Types and Model-Dependent Transformations

A data type for a variable in a programming language defines the set of valid operations on that variable - invalid operations will cause an error, either at compile time or runtime. Feature types are a useful extension to data types for understanding the set of valid operations on a variable in machine learning. For example, we can encode a categorical variable (convert it from a string to a numerical representation), but we cannot encode a numerical feature. Similarly, we can tokenize a string (categorical) input to a LLM, but not a numerical feature. We can normalize a numerical variable, but not a categorical variable. In Figure 2-4, you can see that in addition to the conventional categorical variables (strings, enums, booleans) and numerical variables (int, float, double), I included arrays (lists, vector embeddings) as feature types. A vector embedding is a fixed-size array of either floating point numbers or integers, and they are used to store a compressed representation of some higher dimensional data. Lists and vector embeddings are now widely stored as features in feature stores - and they have well defined sets of valid operations. For example, taking the 3 most recent entries in a list is a valid operation on a list, as is indexing/querying a vector embedding.

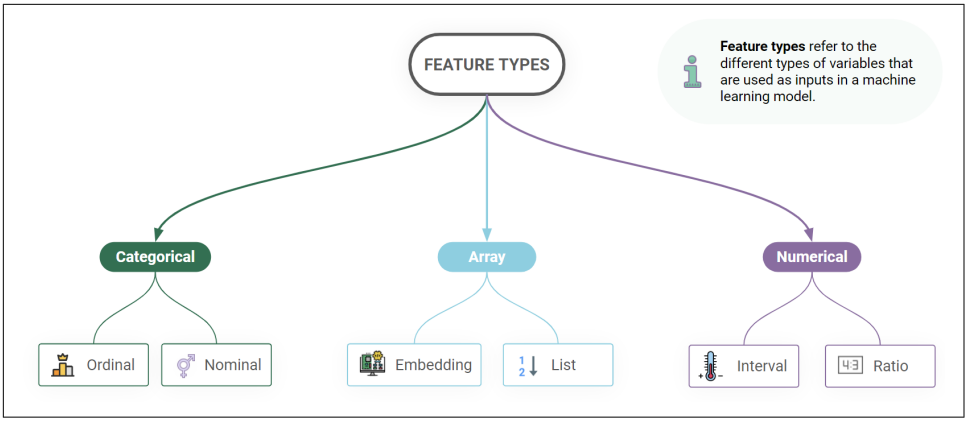


Figure 2-4. Data types in machine learning can be categorized into one of three different feature types - categorical, numerical or an array. Within those categories, there are further subclasses. Ordinal variables have a natural order (e.g., low/med/high), while nominal variables do not. Ratio variables have a defined zero-point, while interval variables do not. Arrays can be a list of values or an embedding vector.

Feature types lack programming language support, instead they are supported in ML frameworks and libraries. For example, in Python, you may use a ML framework such as Scikit-Learn, TensorFlow, XGBoost, or PyTorch, and each framework has its own implementation of the encoding/scaling/normalization transformations for their own feature types.

As discussed earlier, the main challenge in structuring ML systems with feature encoding is that they produce features that can be reused across multiple models. For example, if I want to fine-tune a LLM on a dataset, and I have two candidate LLM models (such as Llama 2 and Mistral), each LLM will have its own tokenizer. If I tokenize the text in my dataset for Mistral, I can't use the tokenized text to fine-tune a model in Llama2, and vice versa. Similarly, although different models might want to reuse the same numerical feature, they might want to encode or scale the same feature differently. For example, gradient-descent models (deep learning) often work better when numerical features have been normalized, but decision trees do not benefit from normalization.

Another problem with these transformations on feature types is that if you were to store encoded, centered, or scaled feature data in the feature store, it would not be amenable to EDA. For example, if you normalized the annual income for citizens from census data, you make the data impossible to understand - it is easier for a data scientist to understand and visualize an income of $74,580 compared to its normalized value of 0.5. Even worse, every time you write new encoded feature data to a feature store, you would have to recompute all of the data for that feature - as the mean/standard deviation/set-of-categories may have changed with the new data. This could

make even very small writes to the feature store very expensive (in what is called *write amplification* - not a good thing).

The reason why encoding/scaling/normalization creates features that are not reusable across other models is that they are parameterized by a training dataset. For example, when we use min-max scaling to normalize a numerical feature, we need the min and max values for that numerical feature in the training dataset. When we one-hot encode a categorical feature (convert it into an array of bytes, with each category represented by a bit in the array, with a binary one for the variable's category and binary zeros for all the other categories) it is parameterized, by the set of all categories in the training dataset. For this reason, we call these types of transformations *model-dependent transformations*, the transformations are dependent on the model and its training data. And we should not perform these transformations in feature pipelines, before the feature store. So, we need to apply model-dependent transformations in both the training and inference pipelines, and we need to make sure there is no *skew* between the model-dependent transformations if the training and inference pipelines are separate programs.

Reusable Features with Model-Independent Transformations

Data engineers are typically not very familiar with the model-dependent transformations introduced in the last section. Those transformations are specific to machine learning and the goals of model-dependent transformations is to make feature data compatible with a particular machine learning library or to improve model performance (such as normalization of numerical features for gradient-descent based ML).

The types of transformations that data engineers are very familiar with that are widely used in feature engineering are (windowed) aggregations (such as the max/min of some numerical variable), windowed counts (for example, number of clicks per day), and any transformations to create RFM (recency, frequency, monetary) features. Transformations that create features that can be reused across many models are called model-independent transformations. Model-independent transformations are applied once in batch or streaming feature pipelines, and the reusable feature data produced by them is stored in the feature store, to be later used by downstream training and inference pipelines.

Real-Time Features with On-Demand Transformations

What if I have a real-time ML system and the data required to compute my feature is only available as part of a user request? In that case, we will have to compute the feature in the online inference pipeline in what is called an on-demand transformation that produces an on-demand (or real-time) feature. Ideally, we would like to also use the same on-demand transformation in a feature pipeline to compute the same feature from historical data logged from your real-time ML system. We will see later in

Chapter 9 how we implement on-demand feature functions as user-defined functions (UDFs) as either Python functions or Pandas UDFs.

The ML Transformation Taxonomy and ML Pipelines

Now that we have introduced the three different types of features produced by ML pipelines, we can present a taxonomy for the data transformations that create reusable, model-specific, and real-time features in machine learning, see Figure 2-5. Our taxonomy includes:

- Model-independent transformations that produce reusable features that are stored in a feature store;
- Model-dependent transformations that produce features specific to a single model;
- On-demand transformations that require request-time data to be computed, but can also be computed on historical data to backfill features to a feature store.

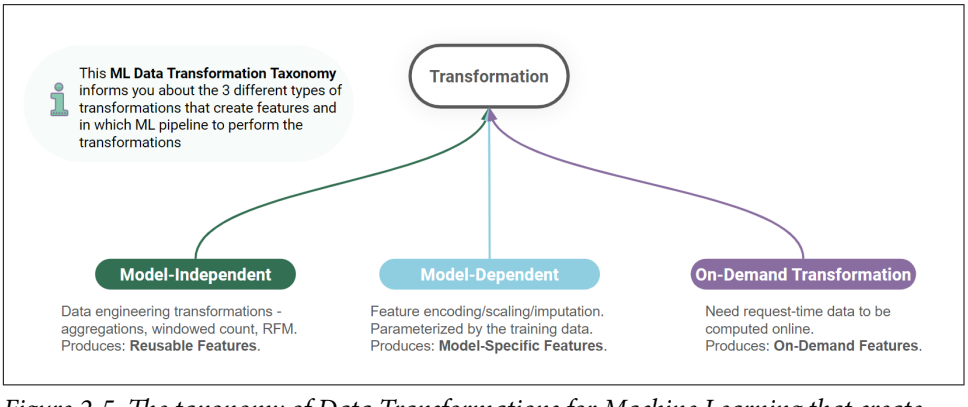


Figure 2-5. The taxonomy of Data Transformations for Machine Learning that create reusable features, model-specific features, and real-time features.

In Figure 2-6, we can see how the different data transformations in our taxonomy map onto our three ML pipelines.

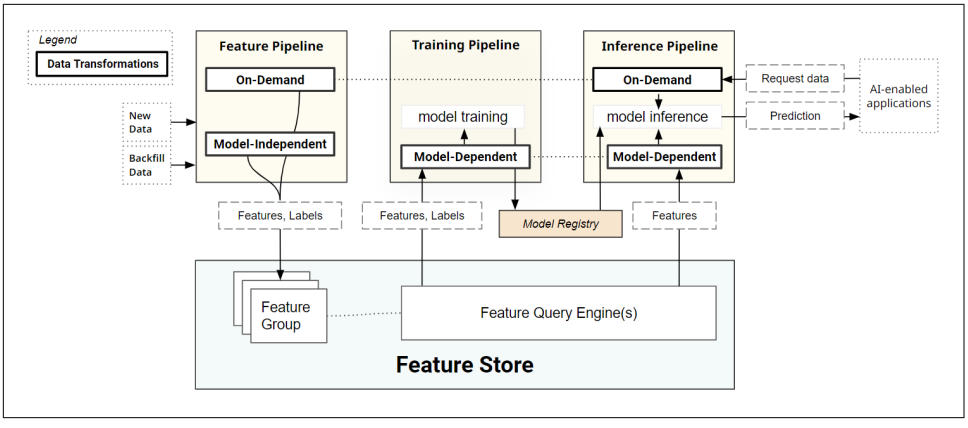


Figure 2-6. Data Transformations for Machine Learning and the ML Pipelines they are performed in.

Notice that model-independent transformations are only performed in feature pipelines. However, model-dependent transformations are performed in both the training and inference pipelines. On-demand transformations are also performed in two different pipelines - the (online) inference pipeline and the feature pipeline. As these different pipelines are separate programs, you need to ensure that exactly the same data transformation is applied in both ML pipelines - that is, there should be no skew between the two different implementations. Any skew between transformations in two different ML pipelines is very difficult to diagnose and can negatively affect your model performance.

Now that we have introduced our classification of data transformations, we can dive into more details on our three ML pipelines, starting with the feature pipeline.

Feature Pipelines

A feature pipeline is a program that orchestrates the execution of a dataflow graph of model-independent and on-demand data transformations. These transformations include extracting data from a source, data validation and cleaning, feature extraction, aggregation, dimensionality reduction (such as creating vector embeddings), binning, feature crossing, and other feature engineering steps on input data to create and/or update feature data, see Figure 2-7.

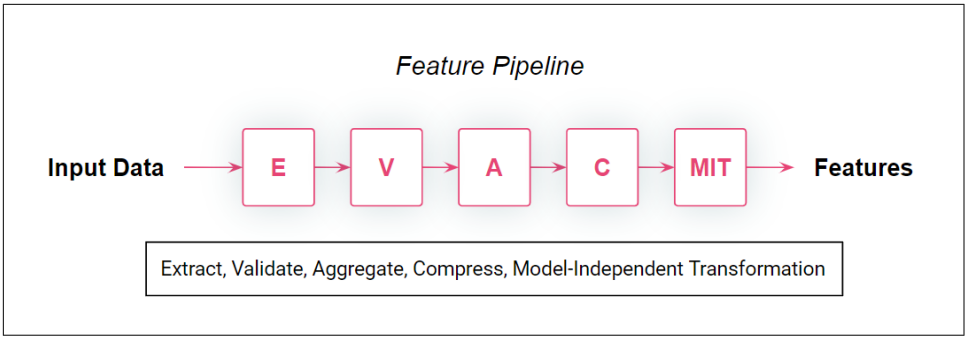


Figure 2-7. A feature pipeline performs data transformations on input data to create reusable features that are stored in the feature store. It can be run against historical data (backfilling) or new data that arrives in batches or as a stream of incoming data.

A feature pipeline is, however, more than just a program that executes data transformations. It has to be able to connect and read data from the data sources, it needs to save its feature data to a feature store, and it also has non-functional requirements, such as:

Backfilling or operational data
: The same feature pipeline (or at least the same transformations) should be able to create feature data using historical data and newly arrived data.

Scalability
: Ensure the feature pipeline is provisioned with enough resources to process the expected data volume.

Feature freshness
: What is the maximum permissible age of precomputed feature data used by clients? Do feature freshness requirements mean you have to implement the feature pipeline as a stream processing program or can it be a batch program?

Governance and security requirements
: Where can the data be processed, who can process the data, will processing create a tamper-proof audit log, will the features be organized and tagged for discoverability?

Data quality guarantees
: Does your feature pipeline minimize the amount of corrupt data that is written to the feature store?

Let's start with the source data for your feature pipeline - where does it come from? Imagine developing a new feature pipeline and getting data from a source you've never parsed before (for example, an existing table in a data warehouse). The table may have been gathering data for a while, so you could run your data transformations

against the historical data in the table to *backfill* feature data into your feature store. It may also happen that you change the data transformations in your feature pipeline, so you, again, want to backfill feature data from the source table (with your new feature transformations). Your data warehouse table will also probably have new data available at some cadence (for example, hourly or daily). In this case, your feature pipeline should be able to extract the new data from the table, compute the new feature data, and append or update the feature data in the feature store.

What does the feature data look like that is created by your feature pipeline? The output feature data is typically in tabular format (one or more DataFrame(s) or table(s)) and it is typically stored in a feature group(s) in the feature store. Feature groups store feature data as tables that are used by clients for both training and inference (both online applications and batch programs).

Scalability and feature freshness requirements can be addressed by implementing a feature pipeline in one of a number of different frameworks and languages. You have to select the best technology based on your feature freshness requirements, your data input sizes, and the skills available in your team. In Figure 2-8, we can see some of the most popular frameworks used to feature pipelines. Batch programs are run on a schedule (or in response to upstream events like data arrival), while stream processing programs are run 24x7.

Figure 2-8. Popular data processing options for implementing your feature pipelines, showing which technologies can process which data sizes and whether the programs are batch or streaming pipelines.

Different data processing engines have different capabilities for (1) efficient processing, (2) scalable processing, and (3) ease of development and operation. For example, if your batch feature pipeline processes less than 1 GB per execution, Pandas is often the easiest framework to start with - the code example from earlier in this chapter, Example 2-1, creates features in Pandas. But for TB-scale workloads, Spark and SQL are popular choices. dbt is a popular framework for executing feature pipelines defined in SQL. dbt adds some modularity to SQL by enabling transformations to be defined in separate files (dbt calls them models) as a form of pipeline. The pipelines can then be chained together to implement a feature pipeline, with the final output a table in a feature store.

When your ML system needs fresh feature data, you may need to use stream processing to compute features. For stream processing feature pipelines, Bytewax or Quix Streams are Python-native choices that are easy to get started with, but for large scale Flink will give you the freshest features, as it processes events one-at-time as they arrive, while Spark Streaming which is also scalable, and supports Python, has higher latency than Flink due to it processing events in batches. We will cover more on batch feature pipelines in Chapter 8, and streaming feature pipelines in Chapter 9.

Finally, feature pipelines tend not to have a very large number of parameters (compared to training pipelines). They can be parameterized with the connection details for the source data, by a `start_time` and `end_time` for backfilling feature data or the `latest_missing_data` for operational model, with parameters for the feature engineering steps (for example, a window size or the number of bins), with parameters for optimizing feature data layout (partitioning or bucketing the feature data for faster querying), and parameters for the pipeline program (number of CPUs, amount of memory, number of workers, when and how to trigger the pipeline).

Training Pipelines

A training pipeline is a program that reads in training data (that is, feature data and labels for supervised learning), applies model-dependent transformations to the training data, trains a machine learning model using a ML framework, validates the model for performance and absence of bias, see Figure 2-9. Training pipelines are either run on-demand, when needed, or on a schedule (for example, new models are re-deployed once per day or week).

Training pipelines can often have a large number of parameters, in particular for deep-learning models. Examples of training parameters for fine-tuning a LLM include the base LLM model, text encoding parameters, hyperparameters for the fine-tuning method (such as LoRA or QLoRA) including quantization, batch size, gradient accumulation, resource estimation and limits (for both GPU and CPU availability), and supervised fine-tuning dataset parameters (url or path, the type of dataset (instruction, conversation, completion).

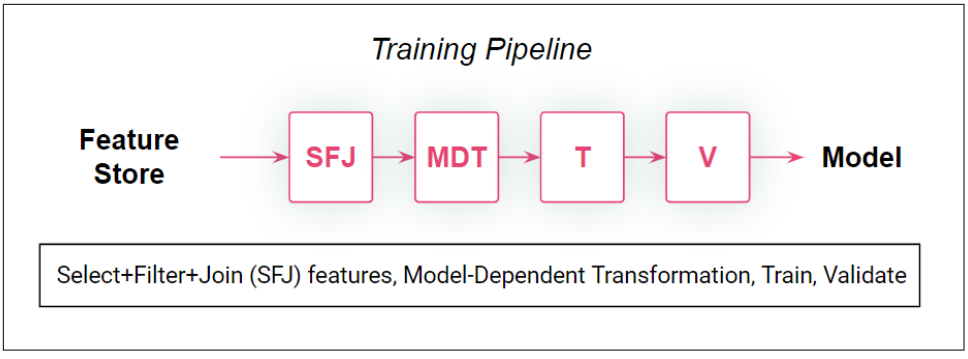


Figure 2-9. A training pipeline consists of a number of steps, from selecting the feature data from the feature store (select, filter, join), to performing model-dependent transformations, to training the model, and to validating the model before it is saved to a model registry.

The output of the training pipeline is the trained, validated model, and it is typically saved to a model registry. For online models, the model can also be deployed directly to model serving infrastructure.

For larger models managed by larger teams, the training pipeline can be further decomposed into a *training data pipeline*, where you select, filter, and join feature data from a feature store to create training data that you then apply model-dependent transformations on, see Figure 2-10.

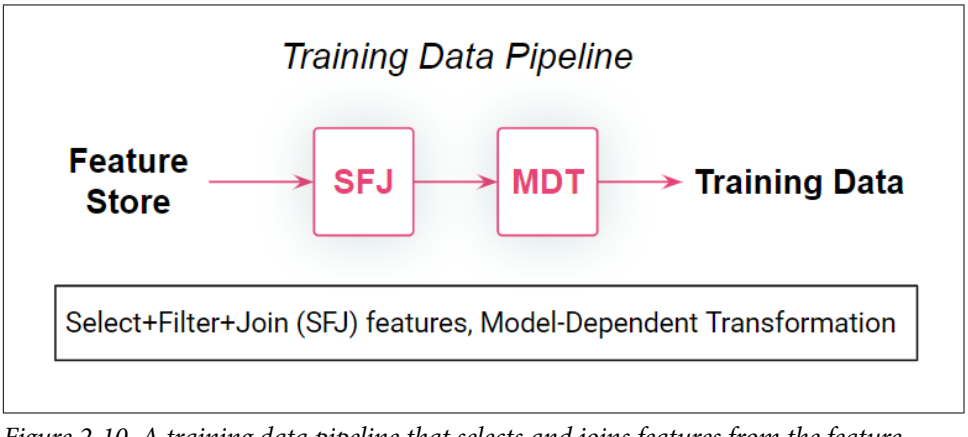


Figure 2-10. A training data pipeline that selects and joins features from the feature store, outputting training data to a file system or object store for later use in a model training pipeline.

The training data is then typically stored to a file system or an object store (such as S3) or a high performance file system backed by NVMe (nonvolatile memory

express) drives, such as HopsFS. For example, when fine-tuning a LLM, even with the high performance PyTorch data loader, they are often I/O bound - the training pipeline cannot read data fast enough from object store, so expensive GPUs are not fully utilized. In this case, we often have a training data pipeline that stores training data to high performance NVMe drives (currently ~8 GB/s throughput for modern NVMes versus ~200 MB/s for AWS S3), which have high enough throughput to keep up with the GPUs.

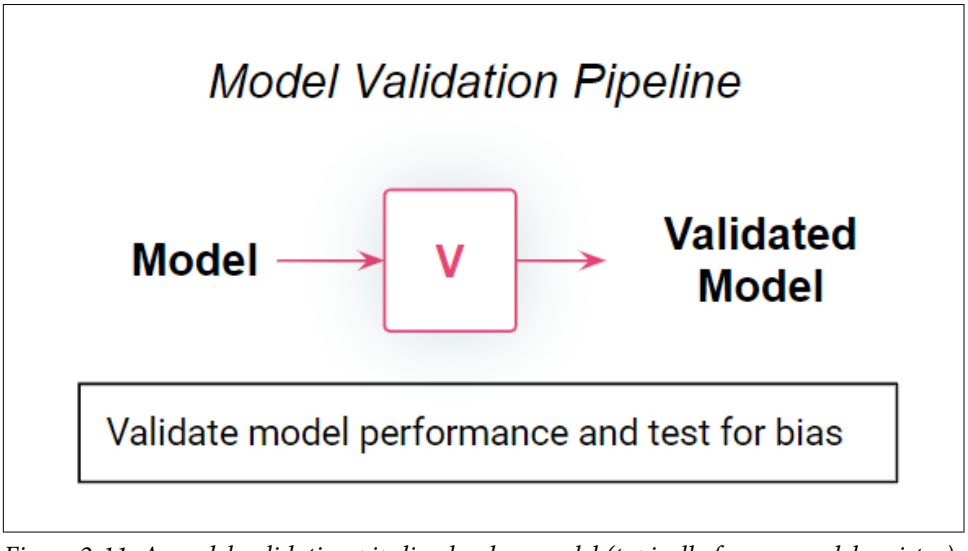


Figure 2-11. A model validation pipeline loads a model (typically from a model registry) and validates that the model has both satisfactory performance and is free from bias, before saving the validated model back to the model registry, annotating that the model has passed all tests.

You can also perform model validation in its own *model validation pipeline*, where the model is asynchronously evaluated after it has been saved to the model registry. This is useful when model validation is a computationally intensive step, and the model training pipeline uses GPUs, such as in LLMs.

Once our model is trained, validated, and stored, it will also need to be deployed if it is an online model (batch models are typically downloaded from a model registry when the batch inference pipeline is run). Model deployment can be performed as part of the training run, but often a model needs approval from a human before deployment. In this case, you would have a separate model deployment pipeline, as shown in Figure 2-12, where a model is copied from a model registry, along with the online inference pipeline program and any other deployment artifacts, to model serving infrastructure.

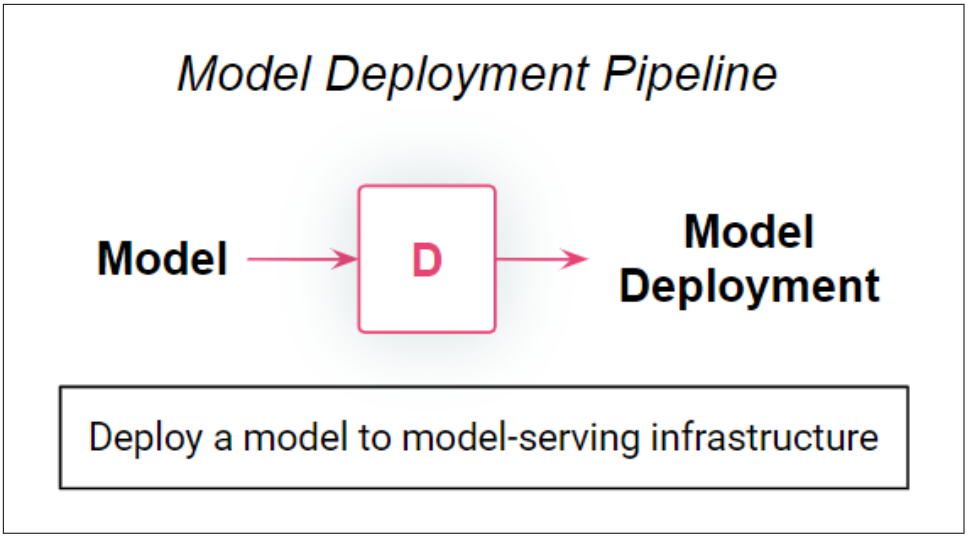


Figure 2-12. A model deployment pipeline deploys a model from a model registry to model serving infrastructure.

The model deployment pipeline is typically run after the model has been approved, but it can also be run on a schedule (for example, after daily or weekly retraining). Model deployment often involves A/B tests, where the model is first deployed as a shadow version and later promoted to the active version if it demonstrates good enough performance and behavior.

Inference Pipelines

An inference pipeline is a program that reads in new feature data, applies model-dependent transformations to the feature data, and makes predictions with the trained model. Depending on whether the ML system is a real-time (interactive) ML system or a batch ML system, your inference pipeline will be either a batch program or a (Python) program invoked by a prediction request on the model serving infrastructure.

In Figure 2-13, we can see a batch inference pipeline, which reads inference data from the feature store, downloads the model from the model registry, and makes predictions. Batch inference pipelines are typically implemented with DataFrames in either Pandas, Polars, or Spark (although some data warehouses have recently added support for batch inference with UDFs).

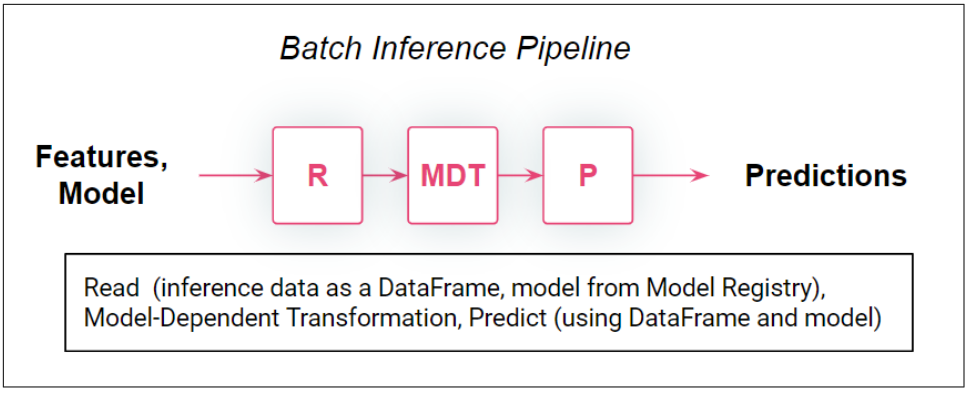


Figure 2-13. A batch inference pipeline reads the inference data from the feature store into a DataFrame (Pandas or PySpark, typically) and downloads the model from the model registry.

Batch inference pipelines are run on a schedule and make predictions for all the rows in the DataFrame (or SQL table) using the model, and the predictions are typically stored in a table in a database (sometimes called an inference store) from where consumers use those predictions. An example of a batch inference ML system was a daily surf height prediction service I wrote for a beach in Ireland (Lahinch), where I have surfed a lot. It scrapes data from websites and publishes a dashboard on Github pages every day.

Batch inference pipelines tend not to have a large number of parameters. Maybe they will be parameterized by a `start_time` and `end_time` or the `latest_missing_data` for inference data. Or maybe the inference data will be all the users or a subset of users, in which case we identify the IDs of the users as a parameter. The details of the sink for predictions may require user-supplied parameters.

Online inference pipelines are run in response to prediction requests. The prediction requests typically contain ID(s) for the entities the prediction is being made for as well as any runtime data required to compute features for the model. For example, in an online retailer, the entity could be a customer and the ID could be their account number, or an order reference number, or a session identity (if they are browsing the website without an account). The online model is typically hosted on model serving infrastructure or embedded in an online application. Online inference pipelines, see Figure 2-14, merge precomputed features from the feature store with any on-demand features to build a feature vector. Model-dependent transformations are then applied to feature data before the transformed feature vector is passed to the model for prediction.

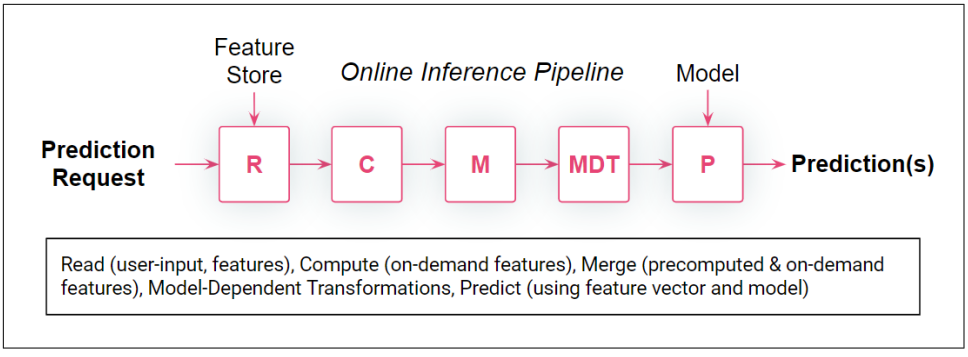


Figure 2-14. An online inference pipeline takes the request parameters and uses them to read any precomputed features from the feature store, compute any on-demand features, and merge them together into a feature vector that the model makes the prediction with.

The output of an online inference pipeline is a prediction (or a batch of predictions) and that is returned to the requesting client and also logged for model monitoring. Typically, you log the untransformed feature values along with the prediction.

Titanic survival as a ML System built with ML pipelines

We now introduce our first example ML system, built with our three ML pipelines, using one of the best known ML problems - predicting the probability of a passenger surviving the Titanic. The Titanic passenger survival data is a static dataset. An ML model is trained and evaluated on the static dataset. That makes it a good introductory dataset for learning ML, as you skip the step of creating the training data. But we want to move beyond the idea of just training models with a static data dump.

In Figure 2-15, we see the outline of our ML system in a Kanban board, including its data sources, its final output (a dashboard), and the technologies used to implement our ML system.

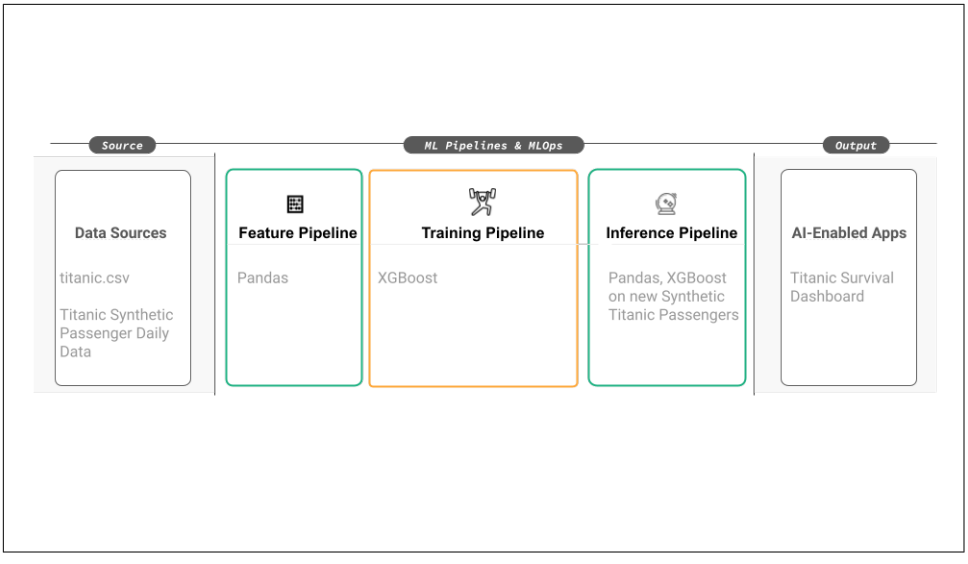
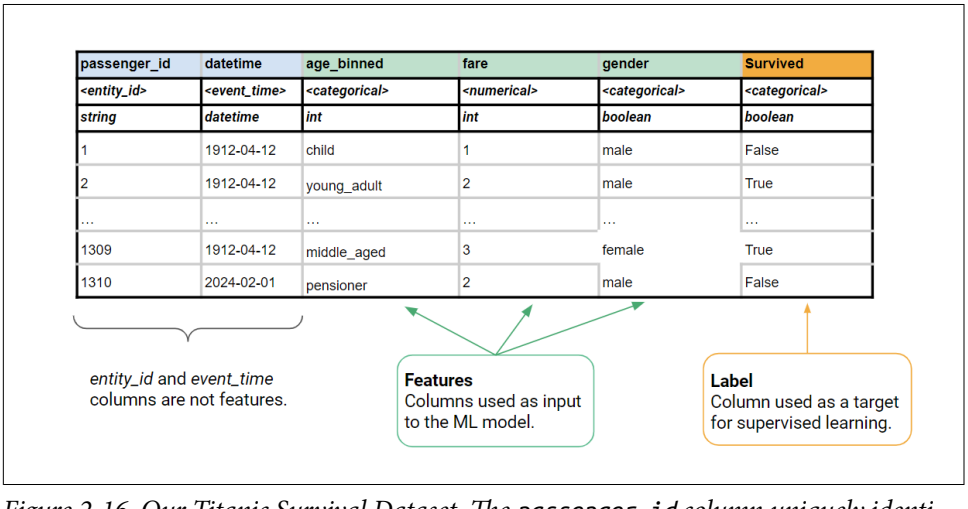


Figure 2-15. The MVPS Kanban board for our Titanic Passenger Survival ML system.

We will use the Titanic Survival dataset for historical data, shown in Figure 2-16.



passenger_id	datetime	age_binned	fare	gender	Survived
<entity_id>	*<event_time>*	*<categorical>*	*<numerical>*	*<categorical>*	*<categorical>*
string	*datetime*	*int*	*int*	*boolean*	*boolean*
1	1912-04-12	child	1	male	False
2	1912-04-12	young_adult	2	male	True
...	...	...	...	...	...
1309	1912-04-12	middle_aged	3	female	True
1310	2024-02-01	pensioner	2	male	False

Figure 2-16. Our Titanic Survival Dataset. The `passenger_id` column uniquely identifies each row - it is not a feature. We augmented the dataset with the `datetime` column - the original dataset has 1309 rows with the date of the Titanic disaster, while each new (simulated) row has the datetime of its creation.

We will then write a synthetic data creation function that creates new passengers for the Titanic. The simulated passenger feature values are drawn from the same distribution as the original dataset, so we will not have any problems with feature drift and

any need to retrain our model. It's an overly simplified example, but still a useful one for getting started with dynamic data.

We will write both the historic and new feature data to a single feature group the feature store with a feature pipeline written in Python using Pandas, see Figure 2-16. We will then schedule the feature pipeline to run once per day, creating one new passenger for the Titanic for that day.

```
import pandas as pd
import hopsworks
BACKFILL=True
def get_new_synthetic_passenger():
    # see github repo for details
if BACKFILL==True:
    df = pd.read_csv("titantic.csv")
    # Remove columns that are not predictive of passenger survival
else:
    df = get_new_synthetic_passenger()
fs = hopsworks.login().get_feature_store()
fg = fs.get_or_create_feature_group(name="titanic", version=1,
    primary_keys=['id'], description="Titanic passengers")
fg.insert(df)
```

We will select the features we want to use in our model and create a *feature view* to represent the input features and output labels/targets for our model:

```
def get_feature_view():
    fs = hopsworks.login().get_feature_store()
    fg = fs.get_feature_group(name="titanic", version=1)
    selected_features = fg.select_all()
    return fv.get_or_create_feature_view(name="titanic", version=1,
label=['survived'], description="Titanic passenger survival")
```

We will use the feature view to create training data (the feature view will query the feature data from the feature store) from the historical Titanic passenger survival data. We will then train the model with XGBoost, a gradient-boosted decision tree library in Python. We will store our trained model in a model registry.

```
fv = get_feature_view()
training_data = fv.training_data()
# perform EDA
# see github repo for details
```

You can discover important features obtained by joining data from a secondary dataset.

```
import XGBoost
import pandas as pd
fv = get_feature_view()
X_train, X_test, y_train, y_test = fv.train_test_split(test_ratio=0.2)
model = XGBoost()
```

```
model.fit(X_train, y_train)
# save model to model registry
```

We will write a batch inference pipeline that will be scheduled to run once per day. It will read any new simulated passengers from the feature store, download our trained model from the model registry, and use the model to predict if the simulated passengers survived or not, outputting its predictions to a new table (a feature group in the feature store in this example) and logging predictions to a logging feature group in the feature store. Finally, we will write a Dashboard in Python using Gradio to show the model's prediction for the most recent synthetic passenger - did they survive, and also showing historical model prediction performance.

```
# Model inference - make predictions on new data
y_preds = model.predict(X_test)
accuracy = classification_report(y_test, y_preds)
mr = hopsworks.login().get_model_registry()
mr.register_model(name="titanic", accuracy)
mr.save_model(joblib.save(model))
```

This ML system solves what is called a counterfactual (what-if) prediction problem. What if there were a passenger who was male, aged 49, and traveled third class on the Titanic - what's the probability he would have survived? We will finally also add an interactive UI - making it also an interactive ML system using Python and Gradio. This enables you to directly ask the model what-if questions about hypothetical passenger survival probabilities.

```
# Model inference - make predictions on new data
y_preds = model.predict(X_test)
accuracy = classification_report(y_test, y_preds)
mr = hopsworks.login().get_model_registry()
mr.register_model(name="titanic", accuracy)
mr.save_model(joblib.save(model))
```

The full source code for this "Titanic passenger survival as a ML system" example is found in the book's source code repository in Github. To get started with this example you will need to install the Hopsworks Python library. On Linux and Apple, this involves calling:

```
pip install hopsworks
```

In Windows, you first need to install the `twofish` library, before you install the Hopsworks library. You will also need to create an account on *app.hopsworks.ai* and you will also need a Hopsworks API key so that you can securely read from and write to Hopsworks. You can either run the first notebook, and it will prompt you to create a Hopsworks API key or you can follow the docs. Hopsworks offers a free-forever serverless tier, with 35GB of free storage, more than enough to complete the projects in this book.

Summary

When building ML systems, we start with the ML pipelines and the data transformations performed in the feature, training, and inference pipelines. We introduced a taxonomy for data transformations for ML pipelines based around reusable features (created by model-independent transformations in feature pipelines), model-specific features (created by model-dependent transformations in training/inference pipelines), and real-time features (created by on-demand transformations in online inference pipelines, that can also be applied to historical data to create features in feature pipelines). We closed out the chapter with our first ML system - a dynamic data version of the Titanic passenger survival prediction problem. We showed how to build both batch and interactive ML systems for Titanic passenger survival. In the next chapter, we will go one step further and you will build a ML system for your neighborhood or region. You will build an air quality prediction service for the neighborhood you live in, and we will use the same frameworks used in the Titanic example - Python, Pandas, XGBoost, and Gradio.

CHAPTER 3

Your Friendly Neighborhood Air Quality Forecasting Service

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. The GitHub repo can be found at *https://github.com/featurestorebook/mlfs-book*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

The first ML project we will build is an air quality forecasting service for a neighborhood you care about. We will follow the minimal viable prediction service (MVPS) process from Chapter 2 - *divide et impera* (divide and conquer). Your work will be a public service built to survive, so please put some time and care into it, and your community will love you for it. I have a personal interest in this project as I have two boys with cystic fibrosis, a genetic disorder that primarily affects the lungs. They were born on the same day, two years apart, and diagnosed the same day. Anyway, I think I speak for the whole cystic fibrosis community in saying this would be a fantastic service for us and many others[1]!

1 You can support cystic fibrosis research via the Cystic Fibrosis Foundation, https://www.cff.org

The prediction problem our ML system will solve is to predict the air quality for a public air quality sensor close to your home or work, or wherever. A worldwide community of Internet of Things (IoT) hobbyists place sensors in their gardens and balconies and publish air quality measurements on the Internet. Where I live in Stockholm, there are over 30 public sensors, and in my home city of Dublin, there are over 40. There is a world air quality index website where you can find a sensor on the map to build your ML system on. Pick one that has both (1) historical data - we will train a ML model on the historical data, so if you have a few years of data that is great, and (2) produces reliable measurements (some sensors are turned off for periods of time or malfunction). A reliable sensor will enable your ML system to continue to collect measurement data, enabling you to retrain and improve the model as more data becomes available. Even though you will provide a free public service to your community, it won't cost you a penny - we will run the system on free serverless services (GitHub and Hopsworks).

Air quality prediction is a pretty straightforward ML problem. We will model the prediction problem as a regression problem - we predict the value of PM2.5. PM2.5 is a fine particulate measure for particles that are 2.5 micrometers or less in diameter, and high levels increase the risk of health problems like low birth weight, heart disease, and lung disease. High levels of PM2.5 also reduces visibility, causing the air to appear hazy. What are the features we will use to predict the level of PM2.5? PM2.5 is correlated with wind speed/direction, temperature and precipitation, so we will use weather forecast data to predict air quality as measured in PM2.5. This makes sense because air quality is generally better when the wind blows in a particular direction - if you live beside a busy road, wind direction is crucial. Air quality is often worse in colder weather as cold air is denser and moves slower than warm air, and in cities where more people may drive than bike when commuting. Even parts of India that don't experience cold winter weather have worse air quality in winter months.

But wait. You may have read that air quality forecasting is a solved problem. In 2024, Microsoft AI built Aurora, a deep learning model that predicts air pollution for the whole world. Microsoft's use of AI was championed as a huge step forward compared to the physical models of air quality, computed on high-performance computing infrastructure by the European Union's Copernicus project. However, as of mid-2024, if you examine the performance of Aurora in a city, such as Stockholm, you will see its predictions are not very accurate compared to the actual air quality sensor readings you can find on https://waqi.info. Your challenge is to build a ML system that produces better air quality predictions than Aurora for the location of your chosen air quality sensor at a fraction of its cost. In this project, better quality data and a decision tree ML model will outperform deep learning.

Finally, every project benefits from a wow factor. We will sprinkle some GenAI dust on the project by making your air quality "friendly" by giving it a voice-driven UI powered by an open-source LLM.

ML System Overview

In my course at KTH, students built a unique ML system that solved a prediction problem using a dynamic data source. But before they started their project, they had to get it approved, and I found that the simplest way to do so was with a prediction service card, see Table 3-1. The card is a slimmed down version of the Kanban board from Chapter 2, omitting the implementation details.

Table 3-1. ML System Card for our Air Quality Forecasting Service

Dynamic Data Sources	Prediction Problem	UI or API	Monitoring
Air Quality Sensor Data: https://*aqicn*.info Weather Forecasts: https://open-meteo.com/	Daily forecast of the level of PM2.5 for the next 7 days at the position of an existing air quality sensor.	A web page with graphs and a LLM-powered UI in Python.	Hindcast graphs show prediction performance of our model.

The ML system card succinctly summarizes its key properties, including the data sources and the prediction problem it solves. For example, with air quality, there are many possible air quality prediction problems, such as the predicting PM10 levels (larger particles that include dust from roads and construction sites), and NO2 (nitrogen dioxide) levels (pollution mostly from internal combustion engine vehicles). The prediction service card also includes the data sources, useful as a feasibility test that the data exists and is accessible for your prediction problem. You should also define how the predictions produced by our ML system will be consumed - by a UI or API. A UI is a very powerful tool to communicate the value of your model with stakeholders, and it is now straightforward to build functional UIs in Python. In our ML system, we will use LLMs to improve the accessibility of our service - you should be able to ask the Air Quality Forecasting Service questions in natural language. And, finally, you should outline how you will monitor the performance of your running ML system to ensure it is performing as expected.

We will use open-source and free serverless services to build our ML system - GitHub Actions/Pages and Hopsworks. We will write the following four Jupyter notebooks in Python:

1. create feature groups to store our data and backfill them with historical data,
2. a daily feature pipeline to retrieve new data and store it in the feature store,
3. a training pipeline to train a XGBoost regression model and save it in the model registry,
4. a batch inference pipeline to download the model and make predictions on new feature data, read from the feature store, producing air quality forecast/hindcast graphs.

We will also use a number of libraries in Python and other technologies to build the system, including:

- REST APIs to read data from our data sources,
- Pandas for processing the data,
- Hopsworks to store feature data and models,
- GitHub Actions to schedule our notebooks to run daily, and
- GitHub Pages as a dashboard web page containing the forecasts/hindcast graphs.

We will also write a Streamlit Python application with a voice and text-powered UI, backed by the open-source Whisper transformer model that translates voice to text and a fine-tuned version of the open-source Llama-3-8B LLM that translates from text to function calls on our ML system.

That is a lot of technologies for our first project, but don't be overawed. Just like much great music can be made with three chords, many great ML systems can be made from a feature pipeline, a training pipeline, and an inference pipeline.

Air Quality Data

Thousands of hobbyists around the world have installed air quality sensors and made their measurements publicly and freely available. You can locate many of these air quality sensors with both historical and live data using the map on the World Air Quality Index project, see Figure 3-1. The website is an aggregator of sensor data from many sources, but as a community service it provides no guarantees on the data quality.

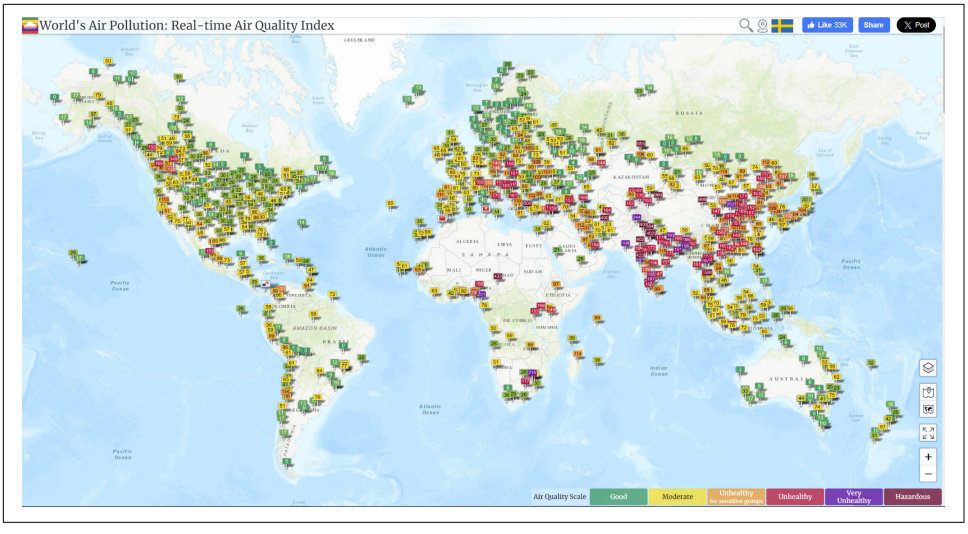


Figure 3-1. On waqi.info, you can navigate on the map to the location of the air quality sensor you will use for this project. You will be redirected to https://aqicn.org where you find the sensor API details and historical data for the sensor.

In Figure 3-2, you can see that I have selected a sensor in Stockholm that has both live and historical data available. I chose it because it is very close to the Hopsworks office. You should pick a sensor either close to you or somewhere special to you. When you click on the link to your sensor/location of choice, it will redirect you to another website, https://aqicn.org - the website that provides real-time air pollution index and API for 100+ countries.

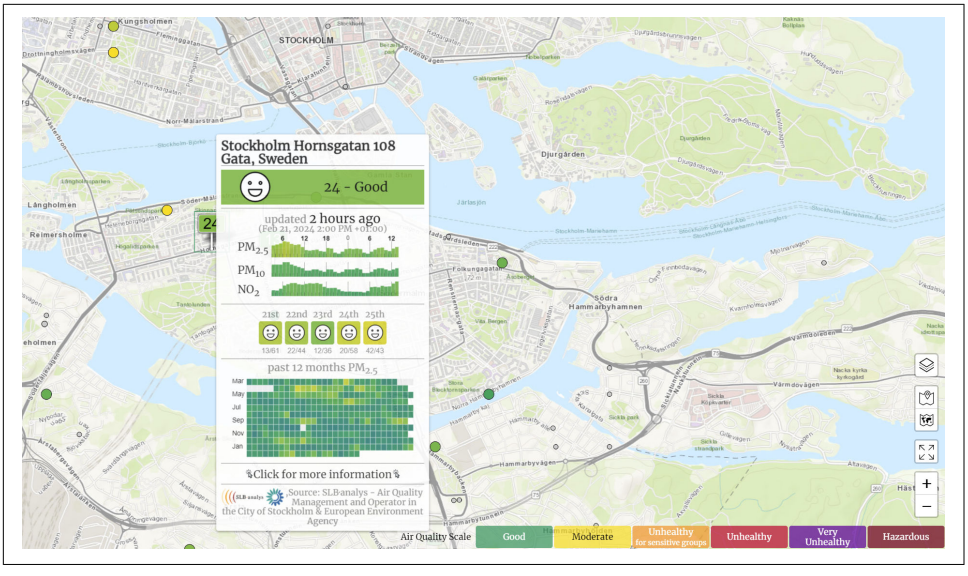


Figure 3-2. We can see here that there is available historical data for "past 12 months PM2.5" for this sensor. A few small gaps in sensor readings like I have here is generally ok.

In my case, for Stockholm Södermalm, it redirected me to https://aqicn.org/station/sweden/stockholm-hornsgatan-108-gata. Scroll down the page and you will find a button to download the historical data for that sensor, see Figure 3-4. If you can't find the download link for the historical measurements on your sensor's webpage, you can probably find them from here https://aqicn.org/historical. If you still can't find the download link, pick another sensor. Unfortunately, as of mid 2024, there is no API call available to download historical data, so you have to perform this step manually.

Figure 3-3. On the URL with our sensor's data at aqicn.org, we can export the historical data by clicking on the "Download this data (CSV format)" button.

Download the CSV (comma separated values) file. I renamed mine to *stockholm-hornsgatan-108.csv*. For your sensor, you should rename the CSV file you downloaded if it has spaces or unusual characters. You should open the CSV file in a text editor to check if its column names are as expected. Our backfilling Python program will read the CSV file into a Pandas DataFrame and expect that the CSV file has a header line and 2 of the columns are *pm25* and *date*. If there are more columns, that is ok. However, some files do not have a *pm25* column - instead they have *min/max/median/stdev* daily measurements for PM2.5. The easiest way to fix this is to just rename the *median* column to *pm25* in the header in your CSV file.

You can now create the GitHub repository for the project by forking the book's GitHub repository at https://github.com/featurestorebook/mlfs-book to your GitHub account. If you don't have a GitHub account, you should create one - they are currently free. You should move your CSV file to the *data/* directory in your forked repository and then commit and push it to GitHub. I ran the following commands to achieve this:
git clone git@GitHub.com:jimdowling/fsbook.git

```
cd fsbook/data
mv ~/Downloads/stockholm-hornsgatan-108.csv .
git add stockholm-hornsgatan-108.csv
git commit -am 'Adding my historical sensor data'
git push
```

The CSV files are quite small (mine is 7.3KB), so there is no problem adding them to GitHub. Files of GBs or larger are not suitable for storage in source-code repositories

like GitHub[2]. When working in Python, we strongly recommend that you create a virtual environment for the book, using a Python dependency management framework such as conda, poetry, virtualenv, or pipenv. The dependencies introduced for our project can be installed in your virtual environment. See the book's source code repository for details on setting up a virtual environment and installing your Python dependencies for this project.

Working with Hopsworks

You will need to create an account on https://app.hopsworks.ai, as we will use Hopsworks as the data layer that stores the outputs of our ML pipelines. You can create a free account on Hopsworks with a Gmail account, a GitHub account, or an email address. You will receive 50GB of free-forever storage and a single project (your project name needs to be unique).. You need to create an API key in Hopsworks, see Figure 3-1. I recommend that you save it to *data/hopsworks-api-key.txt*.

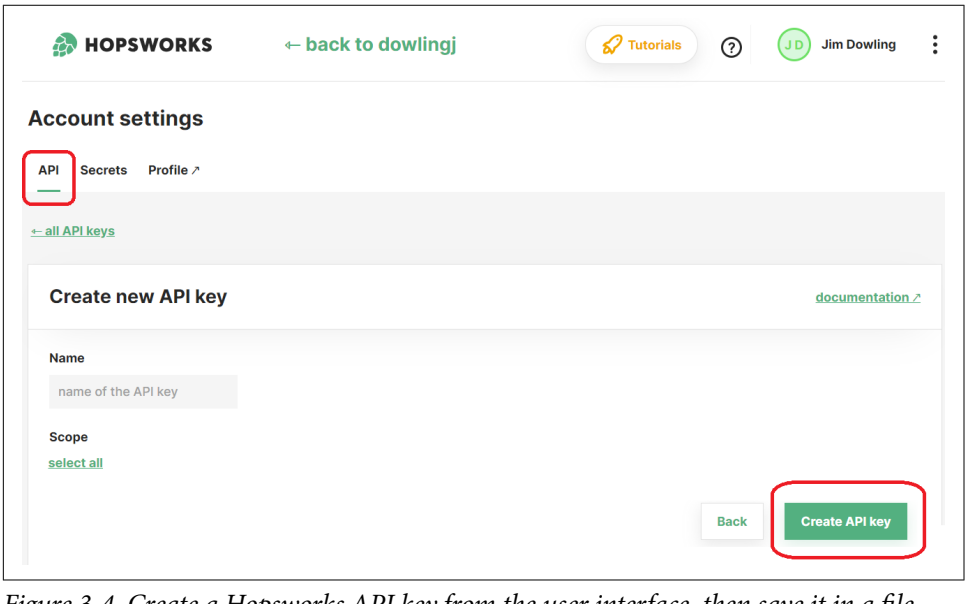


Figure 3-4. Create a Hopsworks API key from the user interface, then save it in a file (data/hopsworks-api-key.txt - you should not commit this file to GitHub). Select all scopes when you are testing, then restrict the scopes for a production HOPSWORKS_API_KEY.

2 Large files should be stored in highly available, scalable distributed storage, such as an S3 compatible object store. These are also currently the cheapest place to store large files.

Your ML pipeline programs can then read the API key securely from the file and use it to login to Hopsworks. You will also need to install the Hopsworks library, used by your ML pipelines:

```
pip install hopsworks
```

Exploratory Dataset Analysis

Before we jump in and start building, we should take some time to understand the data we will work with. In general, there are six properties or dimensions of any data source that you should understand before using it to solve a prediction problem:

1. Validity
2. Accuracy
3. Consistency
4. Uniqueness
5. Update Frequency
6. Completeness

Let's now examine our air quality and weather data sources through this lens.

We primarily use Jupyter notebooks, instead of Google Colabatory (Colab) in this book (although many notebooks will run fine on Colab. Some notebooks won't run because they import Python modules that we write and Colab currently does not support cloning a full GitHub repository and importing those modules. Python modules are inevitable when you modularize your ML system into FTI pipelines, because there will often be code that is common to more than one pipeline. For example, in this chapter the matplotlib graphing code is used in both training and batch inference pipelines. If you were to duplicate the code in the both pipelines, you end up with non-DRY (Don't-Repeat-Yourself) code, which is bad software engineering practice. When you have non-DRY code, you could update code in one notebook, but forget to update it in another notebook, at best, leading to frustration and at worst, introducing bugs. That all said, we will use Colab when we need a free GPU.

Air Quality Data

How does our air quality data source rank along these six properties of dataset quality? We will start with *data validity*, a measure of how accurately the data reflects what it is intended to measure. We focus on measuring PM2.5 rather than PM10 or NO2,

as, according to the UN - "PM2.5 … poses the greatest health threat ", according to current knowledge.

Next up is *data accuracy* that refers to how close the measurements are to the true value. The aqicn.org website tells me that my sensor's data in Stockholm comes from "SLB·analys - Air Quality Management and Operator in the City of Stockholm" and the "European Environment Agency". Therefore, I am inclined to trust the data accuracy. In contrast, in Figure 3-6, you can see a different sensor in Stockholm maintained by the "Citizen Science project sensor community" that started malfunctioning producing the maximum possible PM2.5 readings some time around late 2022 – I can attest that the air quality is not actually that bad in the Solna district near Stockholm.

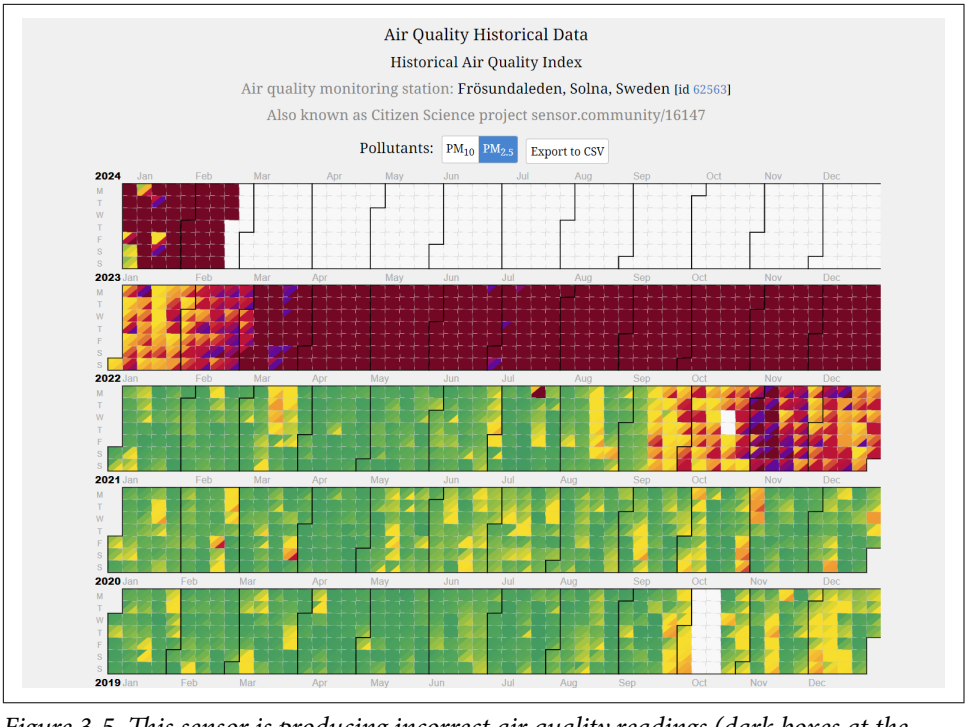


Figure 3-5. This sensor is producing incorrect air quality readings (dark boxes at the top). It appeared to work correctly until it started malfunctioning in late 2022 / early 2023.

Returning to the *stockholm-hornsgatan-108* dataset, we claim that the data is *unique*. After a web search, I am not aware of any other public air quality sensor on that street. Looking at Figure 3-4, I can see that the data is mostly complete, quite *consistent* (the colors indicating air quality follow an expected pattern, unlike those in Figure 3-6), and the data is timely - it arrives hourly. One note of warning: the names of the columns in the CSV file are not always consistent. In some locations, I have seen

min/max/median PM2.5 values instead of a single value - in this case, I would rename the *median* column to *pm25*.

In general, you should also examine the data in a notebook to check its *completeness*. In the following code snippet, we read the CSV file as a Pandas DataFrame and then we keep only those columns we need from our air quality dataset (the *date*, and our target, *pm25*):

```
  # you may need to rename columns in your CSV file to 'pm25' and 'date'
df = pd.read_csv(
"../../data/stockholm-hornsgatan-108.csv",
parse_dates=['date'], skipinitialspace=True)
air_quality_df = df[["date", "pm25"]]
air_quality_df["country"] = "sweden"
air_quality_df["city"] = "stockholm"
air_quality_df["street"] = "stockholm-hornsgatan-108-gata"
air_quality_df["url"] = "https://api.waqi.info/feed/@10009"
```

We also store the *country, city*, *street*, and *url* for the sensor. We will use the *city* column to join our air quality data with the weather features for the same *date*. The *country, city*, *street*, and *url*

columns are *helper columns* that are used when we create a dashboard with air quality forecasts. Although their values will be duplicated over all rows for the same sensor, they will not consume much storage space on disk, as our feature store will use columnar compression.

The second part of dataset completeness is to check for missing data. You can call the `isna()` function on the DataFrame to list any missing values. However, that may produce a huge number of rows as output, so instead we will apply a `sum()` to the result of isna(), summarizing how many values are missing for each column in *df*:

```
df.isna().sum()
```

You can then remove any rows with any missing columns by calling:

```
df.dropna(inplace=True)
```

Removing missing observations is reasonable at this point, as there will be no point in later collecting data where either the date or target is missing.

Often, at this point, we would typically dive deeper into identifying data sources and candidate features for our model. We would try to identify features that have predictive power for the target (PM2.5). If there are not enough samples for deep learning models to be performant, we might try to engineer features that capture domain knowledge about our prediction problem. However, we will skip those steps in this case, to make it a simpler prediction problem. We will use weather features for our model as they have good predictive power for PM2.5 levels. There will be room for improvement in the model we will train, but right now our goal is to build an MVPS for our air quality forecasting problem.

Weather Data

We will use OpenMeteo (https://open-meteo.com) to download both historical weather data and weather forecast data for the same location as your chosen air quality sensor. The weather data from OpenMeteo ranks very high along all our six axes of dataset quality. OpenMeteo provides two different free APIs: one to download historical weather data, and one for weather forecasts. If you are not sure of the best city to use for your weather data, you can search for available weather locations at https://open-meteo.com/en/docs/historical-weather-api. In contrast to air quality data, which is very localized (two neighboring streets could have very different air quality conditions), weather data at the city or even region level is probably good enough for your model.

We will restrict ourselves to those weather conditions that are universally available at weather stations and have the highest predictive power for air quality - precipitation, wind speed, wind direction, and temperature. The OpenMeteo APIs expect longitude and latitude as parameters for your weather location. We use the *geopy* library to resolve the longitude and latitude for a city name that you need to specify (you may need to longitude and latitude manually, if they block your IP).

In the following code snippet using the historical API, we need to provide the location and time range as *longitude*, *latitude*, *start_date*, and *end_date* parameters:

```
        url = "https://archive-api.open-meteo.com/v1/archive"
        params = {
            "latitude": latitude,
            "longitude": longitude,
            "start_date": start_date,
            "end_date": end_date,
            "daily": ["temperature_2m_mean", "precipitation_sum",
    "wind_speed_10m_max", "wind_direction_10m_dominant"]
        }
        responses = openmeteo.weather_api(url, params=params)
```

The weather forecast data will be retrieved by a similar REST call:

```
        url = "https://api.open-meteo.com/v1/ecmwf"
        params = {
            "latitude": latitude,
            "longitude": longitude,
            "daily": ["temperature_2m", "precipitation", "wind_speed_10m",
    "wind_direction_10m"]
        }
        responses = openmeteo.weather_api(url, params=params)
```

However, you should note that our forecast API call receives hourly forecasts but our historical API call retrieves aggregate data over a day (i.e., mean temperature, sum of precipitation, max wind speed). This is not ideal, but is good enough for our purposes (we did say the model could be improved!).

There are two utility functions, `get_historical_weather()` and `get_weather_forecast()`, defined in *weather-util.py* that return the weather data as Pandas DataFrames:

```
    import util
    historical_weather_df = util.get_historical_weather("Stockholm",
"2019-01-01",  "2024-03-01")
    weather_forecast_df = util.get_weather_forecast("Stockholm")
```

Note that these functions make network calls, so the code may fail if the program does not have Internet connectivity. The same holds for the function we will use to retrieve real-time air quality data.

Creating and Backfilling Feature Groups

We will store our featurized DataFrames in feature groups in the Hopsworks Feature Store. To run the code in this section, you will need to create an account on app.hopsworks.ai, as described in Chapter 2. We will have two feature groups, one for air quality data, containing the observations of PM2.5 values and the timestamps for those observations, and another feature group to store both the historical weather observations as well as the weather forecast data. The feature group stores the incremental set of features created over time. Training and inference steps can later use a feature query service to read a consistent snapshot of feature data from one or more feature groups to create a model or to make predictions, respectively.

```
air_quality_fg = fs.get_or_create_feature_group(
    name='air_quality',
    description='Air Quality observations daily',
    version=1,
    primary_key=['city', ‘street’],
    expectation_suite = aq_expectation_suite,
    event_time="date",
)
air_quality_fg.insert(df_air_quality)
```

We call `get_or_create_feature_group`, instead of just `create_feature_group`, as we want the notebook to be idempotent (`create_feature_group` fails if the feature group already exists).

```
weather_fg = fs.get_or_create_feature_group(
    name='weather',
    description='Historical daily weather observations and weather forecasts',
    version=1,
    primary_key=['city'],
    event_time="date",
    expectation_suite = weather_expectation_suite
)
weather_fg.insert(df_weather)
```

Notice that both feature groups define an `expectation_suite` parameter. This is a set of data validation rules that we declaratively attach once to the feature group, but are applied every time we write a DataFrame to the feature group. Let's double click on data validation now.

Data Validation

We want to build a ML system we can trust, so we will validate data retrieved from the air quality and weather data sources. This simple test would have helped identify faults in the sensor from Figure 3-6 from the moment they started happening. Great Expectations is a popular open-source library for declaratively specifying data validation rules. In the following code snippet, we define an expectation in Great Expectations that checks all the values in the *pm25* column in our DataFrame, *df*, to make sure that the.scraped values are neither negative nor greater than 500 (a reasonable upper limit for the expected PM2.5 values for my location):

```
import great_expectations as ge
aq_expectation_suite = ge.core.ExpectationSuite(
    expectation_suite_name="aq_expectation_suite"
)
aq_expectation_suite.add_expectation(
    ge.core.ExpectationConfiguration(
        expectation_type="expect_column_min_to_be_between"
        kwargs={
            "column":"pm25",
            "min_value":0.0,
            "max_value":500.0,
            "strict_min":True
        }
    )
)
```

We will see later in Chapter 6, how you can easily add a notification (like Slack or email) if a data validation rule fails and what to do about it. In the book's source code repository, there are also similar expectations defined for the weather data on the *temperature_2m* and *precipitation* columns.

Feature Pipeline

We just presented the program that creates the feature groups and backfills them with historical data. But we also need to process new data daily. We could extend our previous program and parameterize it to either run in backfill mode or in normal mode. But, instead we will write the daily feature pipeline as a separate program - this separates the concerns of creating the feature groups and backfilling them from daily updates to the feature groups.

The feature pipeline will be scheduled to run once per day, and it performs the following tasks:

- reads today's PM2.5 measurement,
- reads the today's weather data measurements,
- reads the weather forecast data for the next seven days,
- inserts all of this data into the air quality and weather feature groups, respectively.

There is no feature engineering required in this example. We will read all of the data as numerical feature data, and will not encode that data before it is written to feature groups. The code shown for downloading the sensor readings and weather forecasts is found in the *functions*/util.py module:

```
url = f"{aqicn_url}/?token={AQI_API_KEY}"
data = trigger_request(url)
aq_today_df = pd.DataFrame()
aq_today_df['pm25'] = [data['data']['iaqi'].get('pm25', {}).get('v', None)]
aq_today_df['city'] = city
..
aq_today_df['date'] = datetime.date.today()
air_quality_fg.insert(df_air_quality)
url = "https://api.open-meteo.com/v1/ecmwf"
params = {
        "latitude": latitude,
        "longitude": longitude,
        "hourly": ["temperature_2m", "precipitation",
"wind_speed_10m", "wind_direction_10m"]
}
responses = openmeteo.weather_api(url, params=params)
hourly_df = # populate with responses data
daily_df = hourly_df.between_time('11:59', '12:01')
weather_fg.insert(daily_df)
```

Our remote API calls to aqicn and openmeteo return the air quality and weather forecast data, respectively, and we put the returned data in Pandas DataFrames that are then inserted into their respective feature groups. When you insert the DataFrame to the feature group, its data validation rules will be executed, as shown in Figure 3-7. You can set a policy on whether to fail the ingestion or allow the ingestion and trigger an alert, see Chapter 5 for more details.

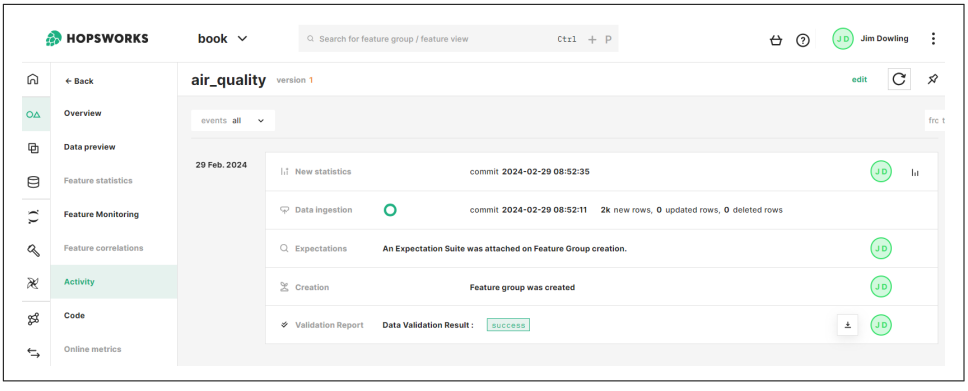


Figure 3-6. You can inspect the result of your feature pipeline run in the Hopsworks UI, seeing the number of new/updated/deleted rows and whether data validation rules passed or not.

You can now see the results of your feature pipeline in the Hopsworks UI. Log in to https://app.hopsworks.ai, using the account you created earlier. In the Hopsworks UI , navigate to feature groups and inspect the data you inserted. Try out *Data preview* to see the sample rows of data ingested. Have a look at the *Feature statistics* computed over the data inserted and the data validation results.

Training Pipeline

We decided that we would model PM2.5 as a regression problem, and we know we will only have a few hundred or possibly a thousand rows or so. This is decidedly in the realm of small data, so we will not use deep learning. Instead, we will use the go-to ML framework for small data - XGBoost (eXtreme Gradient Boosting), an open-source gradient-boosted decision tree framework. XGBoost works well out of the box, and we won't do any hyperparameter tuning here - we will leave it as an exercise for you to squeeze more performance out of the model.

We will start by selecting the features we are going to use in our model. For this, we will use the Feature View in Hopsworks. A Feature View defines the schema for a model - its input features and output targets (or labels). Hopsworks provides a Pandas-like API for selecting features from different feature groups and then joining the selected features together using a *query* object. The *select()* and *select_*all() methods on a feature group returns a *query* object that provides a *join()* method (more details in Chapter 5). When you create the feature view, you also specify which of the selected features are the label columns. The code for selecting the features from the feature groups, joining them together using the common 'city' column, and creating the feature view looks as follows:

```
selected_features = air_qual-
ity_fg.select(['pm2_5']).join(weather_fg.select_all( on=['city'])
feature_view = fs.create_feature_view(
    name='air_quality_fv',
    version=version,
    labels=['pm2_5'],
    query=selected_features
)
```

With a feature view object, you can now create training data:

```
X_train, X_test, y_train, y_test = feature_view.train_test_split(test_size=0.2)
```

Here we read training data as Pandas Dataframes, randomly split (80/20) into training set features (*X_train*), training set labels (*y_train*), and test set features (*X_test*) and test set labels (*y_test*). In a single call, *train_test_split* reads the data, performing a point-in-time correct JOIN of the air quality and weather data, and then performs a scikit-learn random split of the data into features and labels for both training and test sets. The reason I chose a random split over a time-series split is that our chosen features are not time-dependent. A useful exercise would be to improve this air quality model by adding features related to air quality (historical air quality, seasonality factors, and so on) and change to a time-series split.

We can now train our model using *XGBoostRegressor*. We simply fit our model to our features and labels from the training set, using the default hyperparameters for *XGBoostRegressor*:

```
xgb_regressor = XGBRegressor()
clf.fit(X_train, y_train)
```

Training should only take a few milliseconds. Now you can evaluate the trained model, *clf*, using the features from our test set to produce predictions, *y_pred*:

```
y_pred = clf.predict(X_test)
mse = mean_squared_error(y_test, y_pred, squared=False)
r2 = r2_score(y_test, y_pred)
plot_importance(clf, max_num_features=4)
```

As we are modeling PM2.5 prediction as a regression problem, we are using mean-squared error (MSE) and R squared error as metrics to evaluate model performance. An alternative to MSE is the mean absolute error (MAE), but MSE punishes a model more if its predictions are wildly off from the outcome compared to MAE. With the scikit-learn library, it is just a method call to compute one of many different model performance metrics when you have your outcomes (*y_test*) and your predictions (*y_pred*) readily available. We also calculate feature importance, that we later save as a .png file.

Now, we need to save the output of this training pipeline, our trained model, *clf*, to a model registry. We will use the Hopsworks model registry. This process involves first saving the model to a local directory, and then registering the model to the model

registry, including its name (*air_quality_xgboost_model*) and description, its evaluation metrics, and the feature view used to create the training data for the model:

```
model_dir = "air_quality_model"
os.mkdirs(model_dir + "/images")
clf.save_model(model_dir + "/model.json")
plt.savefig(model_dir + "/images/feature_importance.png")
mr = project.get_model_registry()
mr.python.register_model(
    name="air_quality_xgboost_model",
    description="Air Quality (PM2.5) predictor."
    metrics={ "MSE": mse, "r2": r2 },
    feature_view = feature_view,
    model_dir=model_dir
)
```

The model registry client extracts the schema and lineage for the model using the feature view object. Any other files in the local directory containing the model will also be uploaded, and any .png/.jpeg files in the *images* subdirectory (*feature_importance.png*) will be shown in the *model evaluation images* section, see Figure 3-8.

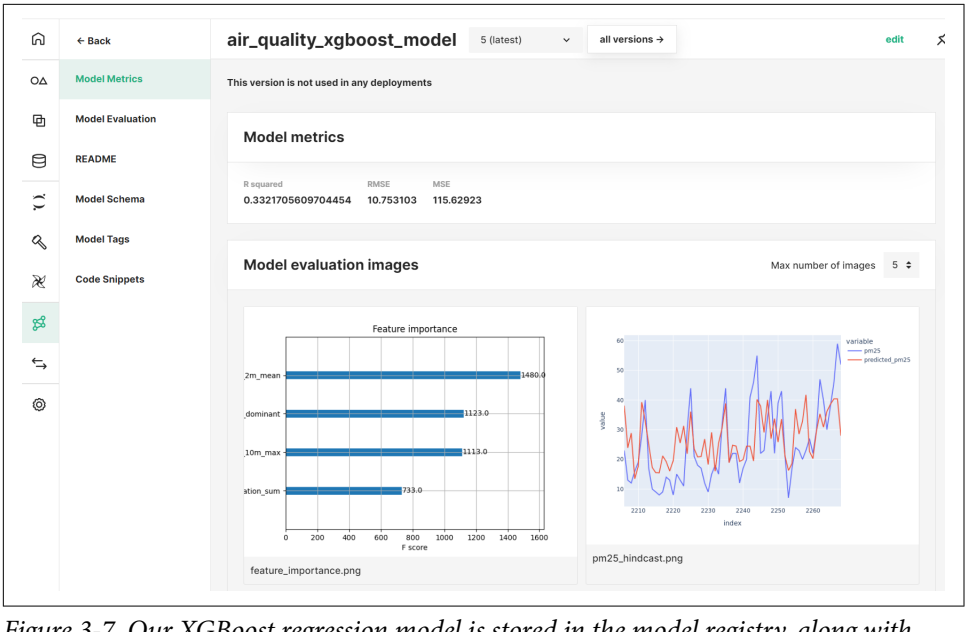


Figure 3-7. Our XGBoost regression model is stored in the model registry, along with model metrics, and two model evaluation images.

Notice that every time we register a model, we will get a new version of the model. Unlike feature groups and feature views, we don't need to provide the version for the model when creating it - a monotonically increasing version number will be assigned

to the newly registered model. With our trained model in the model registry, we can now write our batch inference pipeline that will generate our air quality dashboard.

Batch Inference Pipeline

The batch inference pipeline is a Python program that downloads the trained model from the model registry, fetches the weather forecast feature data, and uses the model and the weather forecast data to predict air quality for the next seven days. We will make seven different predictions, one for each of the seven days. We will create a graph of the air quality forecasts using *plotly*, save that graph as a PNG file, and push that PNG file to a GitHub repository that contains a public website with GitHub pages. GitHub pages has a free tier that allows you to build webpages, dashboards, personal blogs, and you get a dedicated domain name for your website.

First, we need to download our model from the model registry and load it using the *XGBRegressor* object:

```
model_ref = mr.get_model(
    name="air_quality_xgboost_model",
    version=1,
)
saved_model_dir = model_ref.download()
retrieved_xgboost_model = XGBRegressor()
retrieved_xgboost_model.load_model(saved_model_dir + "/model.json")
```

Then, we retrieve a batch of inference data (our weather forecast data for the next seven days) using the feature view.

```
batch_df = feature_view.get_batch_data(start_time=today)
```

The *batch_df* DataFrame now contains the weather forecast features for the next seven days. With these features, we can now make the predictions using the model:

```
for index, row in batch_df.iterrows():
batch_df["pm25_predicted"] = model.predict(row)
batch_df["days_before_forecast"] = index
```

We store the predictions in the `pm25_predicted column of` *batch_df* along with the number of days before the forecast. There are ten forecasts for each day.. The first one is seven days beforehand and the last forecast is 1 day beforehand. This `days_before_forecast` column will help us evaluate the performance of our model depending on how many days in advance it is forecasting. We are going to save `batch_df` to the feature store, to be used to monitor the features/predictions, as `batch_df` includes the predictions, feature values, and helper columns.

```
monitoring_fg = fs.get_or_create_feature_group(
    name='monitoring_aq',
    description='Monitor Air Quality predictions’,
    version=1,
```

```
    primary_key=['city', 'street']
)
monitoring_fg.insert(batch_df)
```

We also have to plot our air quality prediction dashboard. We will use the plotly library:

```
import plotly.express as px
fig = px.line(batch_df , x = "date", y = "pm25_predicted", title = "..")
….
fig.write_image(file="forecast.png", format="png", width=1920, height=1280)
```

We will use a GitHub Action to publish the forecast.png file on a webpage, as described in the next section, see Figure 3-7.

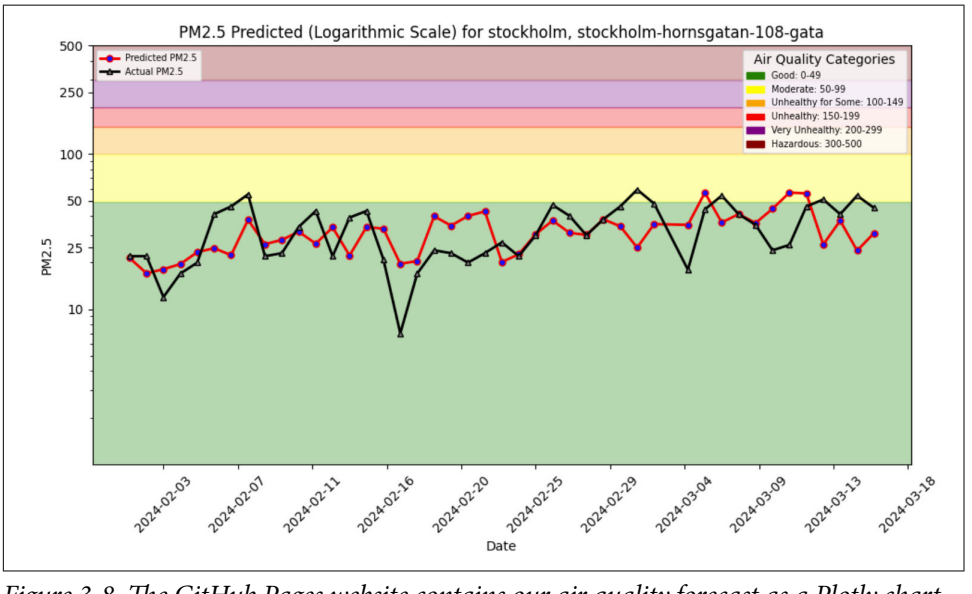


Figure 3-8. The GitHub Pages website contains our air quality forecast as a Plotly chart and the hindcast (shown here) that shows both the predicted PM2.5 and actual PM2.5 values.

Finally, we create some hindcast PNG files that compare our model's predictions, from the monitoring feature group data, and the outcomes, from the air quality feature group data. See the batch inference pipeline notebook in the book's source code repository for details.

Running the Pipelines

To get started, you should run the Jupyter notebooks on your laptop to ensure they work as expected. Run them from the first cell to the last cell. You should switch to the Hopsworks UI after running each notebook to see the changes made - such as

creating a feature group, writing to a feature group, creating a feature view, and saving a trained model to the model registry.

First, run the feature backfill notebook (*1_air_quality_feature_backfill.ipynb*). This will create the *air_quality* and *weather* feature groups. You should then run the feature pipeline (*2_air_quality_feature_pipeline.ipynb*) and check the feature groups to see if new rows have been added to them as expected. Then you can train your model by running the model training pipeline (*3_air_quality_training_pipeline.ipynb*) - check the feature view (*air_quality_fv*) was created and the trained model is in the model registry. Finally, test that your batch inference pipeline (*4_air_quality_batch_inference.ipynb*) works as expected - it should have created a *aq_predictions* feature group. If you find a bug, please post a Github issue. If you can improve the code, please file a PR (pull request). If you need help, please ask questions on the serverless-ml discord channel linked in the book's GitHub repository.

Scheduling the Pipelines as a GitHub Action

We will use GitHub Actions to schedule the feature and batch inference pipelines, and build our dashboard using GitHub Pages. As of 2024, GitHub's free tier gives you 2,000 free minutes of compute every month. That is more than enough to run our feature and batch inference pipelines. You can run the training pipeline on a Jupyter notebook on your laptop - we won't run it on a schedule for now. For our UI, we will use GitHub Pages (that hosts webpages for your GitHub repository), and in their free tier, as of 2024, webpages cannot be larger than 1GB and pages have a soft bandwidth limit of 100 GB per month. This should be more than enough for this project.

There are many different platforms that can be used to schedule our pipelines. In my ID2223 course, students could choose between modal.com and GitHub Actions. Modal's free tier is generous and its developer experience is great, but Modal requires a credit card for access and can't schedule notebooks (only Python programs). There are many other serverless compute platforms that offer orchestration capabilities that you could use instead to run the Python programs including Google Cloud Run, Azure Logic Apps, AWS Step Functions, fly.io, any managed Airflow platform, Dagster, and Mage.ai.

So, what is GitHub Actions? It is a continuous integration and continuous deployment (CI/CD) platform that allows you to automate your build, test, and deployment pipelines. GitHub Actions is typically used to schedule tests (unit tests or integration tests) and deploy artifacts. In our case, our feature and batch inference pipelines can be considered deployment pipelines that create features in the feature store and build our dashboard artifacts for GitHub Pages.

For your GitHub Action to run successfully, you need to set the HOPSWORKS_API_KEY as a repository secret, so that your pipelines can authenticate with Hopsworks, see Figure 3-10.

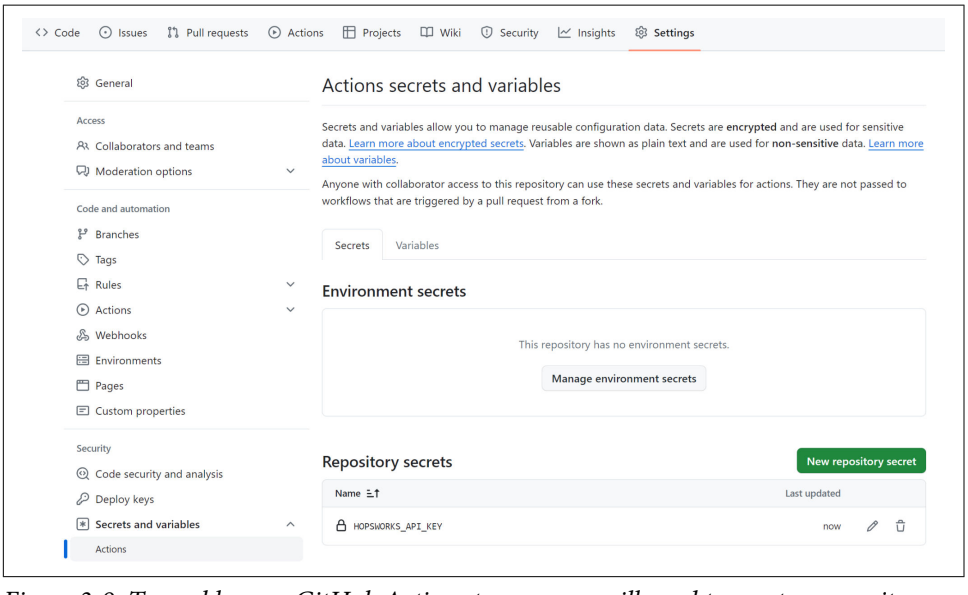


Figure 3-9. To enable your GitHub Actions to run, you will need to create a repository secret for the HOPSWORKS_API_KEY.

We can then proceed to define the YAML file containing the GitHub Actions, found in the GitHub repository at *.github/workflows/air-quality-daily.yml*. You can run the workflow in the GitHub Actions UI for your repository by clicking on *Run workflow*, see Figure 3-10.

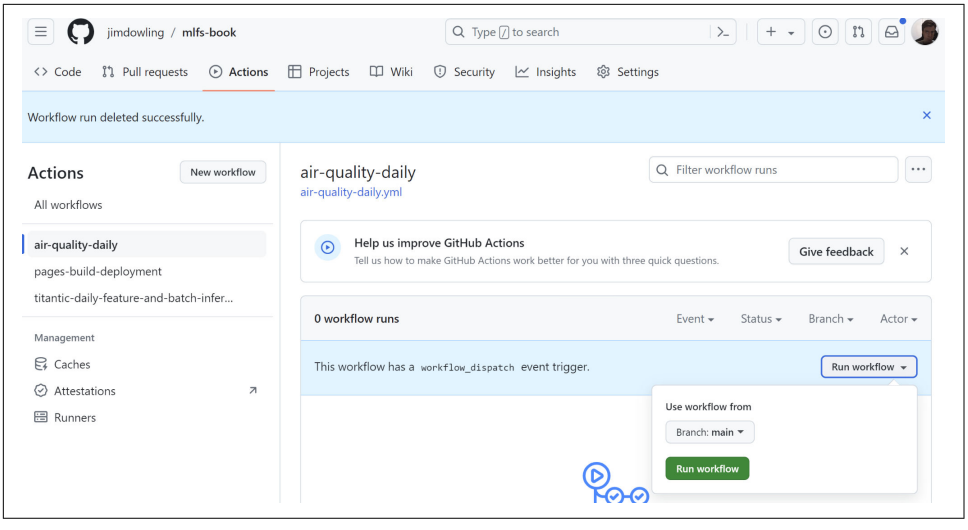


Figure 3-10. You can test running your GitHub Action manually by clicking on the "Run workflow" button. To get there, navigate to the "Actions" tab in your repository, and select the air-quality-daily action.

The workflow code shows the actions taken by the workflow. Firstly, the scheduled execution of this action has been commented out. When you have successfully run this GitHub action without errors, you can uncomment the "*schedule*" and "*- cron*" lines near the beginning of the file and this GitHub Action will then run daily at 6:11 am.

The steps taken by the workflow are as follows. First, the workflow will run the steps on a container that uses the latest version of Ubuntu. Second, it will checkout the code in this GitHub repository to a local directory in the container and change the current working directory to the root directory of the repository. Third, it will install Python. Fourth, it will install all the Python dependencies in the *requirements.txt* file using pip (after upgrading pip to the latest version). Finally, it will run the feature pipeline followed by the batch inference pipeline, after having set the HOPSWORKS_API_KEY as an environment variable. Our GitHub actions execute our feature pipeline and batch inference notebooks with the help of the nbconvert utility that first transforms the notebook into a Python program and then runs the program from the first cell to the last cell. The HOPSWORKS_API_KEY environment variable is set so that these pipelines can authenticate with Hopsworks.

```
on:
  workflow_dispatch:
  #schedule:
  #  - cron: '11 6 * * *'
jobs:
  test_schedule:
```

```
    runs-on: ubuntu-latest
    steps:
      - name: checkout repo content
        uses: actions/checkout@v4
      - name: setup python
        uses: actions/setup-python@v4
        with:
         python-version: '3.10.13'
      - name: install python packages
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
      - name: execute pipelines
        env:
          HOPSWORKS_API_KEY: ${{ secrets.HOPSWORKS_API_KEY }}
        run: |
          cd notebooks/ch03
jupyter nbconvert --to notebook --execute 2_air_quality_feature_pipeline.ipynb
jupyter nbconvert --to notebook --execute 4_air_quality_batch_inference.ipynb
```

Building the Dashboard as a GitHub Page

Our GitHub Action also includes steps to commit and push the PNG files created by the batch inference pipeline to our GitHub repository, and then to build and publish a GitHub Page containing the Air Quality Forecasting Dashboard (with our PNG charts). The GitHub action YAML file contains a step called *git-auto-commit-action* that pushes the new PNG files to our GitHub repository and rebuilds the GitHub Pages. You shouldn't need to change this code.

```
      - name: publish GitHub pages
        uses: stefanzweifel/git-auto-commit-action@v4
        [ … ]
```

Note, that every time the action runs, in your GitHub history it will be shown as a commit by you to the repository.

In order for the *git-auto-commit-action* step to be able to run successfully, you first have to enable GitHub pages in your repository.

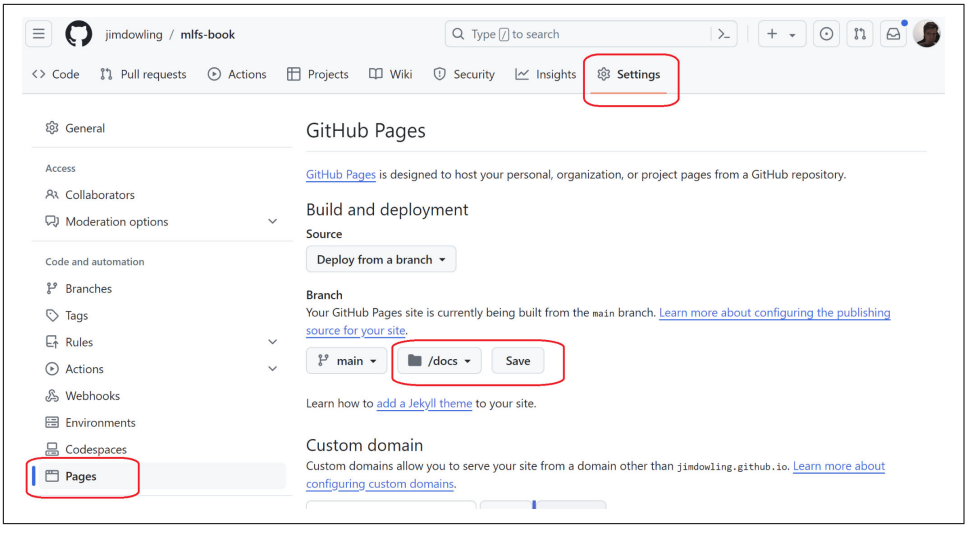


Figure 3-11. You have to enable GitHub Pages for your forked repository. Select the /docs directory and click on the save button to create the GitHub Page for your repository.

And that's it. Once you have the GitHub Page enabled, and your GitHub Action runs your workflow every day, your Dashboard will be updated daily with the latest air quality forecasts (as shown earlier in Figure 3-8)!

Function Calling with LLMs

You now should have a working air quality forecasting system powered by ML. But, we want to make it even more accessible by adding a voice-activated UI. For this, we are going to use two different open-source transformer models (see Figure 3-12 and the notebook *5_function_calling.ipynb* in the repository):

- Whisper transcribes audio into text - users speak and ask a question to our application and the model will output what the user said as text,
- the transcribed text will then be fed into a fine-tuned Llama-3-8B LLM that will return one function (from a set of four available functions) including the parameter values to that function.
- The chosen function will be executed returning either historical air quality measurements or a forecast for air quality and that output will be fed back into the LLM as part of the prompt along with your original voice-issued question to the same Llama-3-8B LLM.
- The LLM will return a human-understandable answer about the air quality (is it safe or healthy) that is not just about the PM2.5 levels.

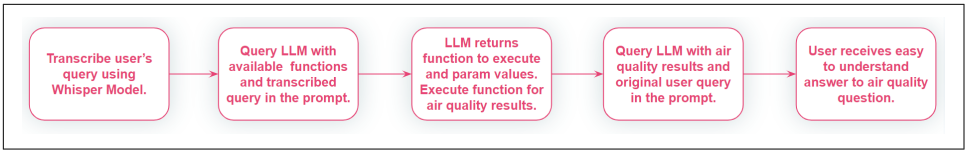


Figure 3-12. Our Voice-Activated UI uses Whisper to transcribe a user query that triggers a function to be executed that will either return historical air quality measurements from the feature group or forecasts from the model. Those results are passed again to the LLM that answers the original question but the prompt now also includes the external context information provided by our Air Quality ML system. This RAG without a vector database.

We are building our voice-activated UI using the paradigm of RAG (retrieval augmented generation) using function calling with LLMs. With LLMs, the user enters some text, called the *prompt*, and the LLM returns with a response. For chat-based LLMs, like OpenAI's ChatGPT, the response is usually a conversational style response. Function calling with LLMs involves the user entering a prompt, but now the LLM will respond with a JSON object containing the function to execute (from a set of available functions) along with the parameters to pass to that function. We will use a LLM that is fine-tuned to return JSON objects describing the functions. The returned JSON can then be parsed and used to execute one of our predefined functions:

- *get_future_data_for_date*
- *get_future_data_in_date_range*
- *get_historical_air_quality_for_date*
- *and get_historical_data_in_date_range*

That is, users will not be able to get answers to arbitrary questions about air quality - only historical readings and air quality forecasts. You can ask questions like "what was the air quality like last month?" or "what will the air quality be like on Tuesday?".

After you pass the list of function declarations in a query to the function-calling LLM, it tries to answer the user query with one of the provided functions. The LLM understands the purpose of a function by analyzing its function declaration. The model doesn't actually call the function. Instead, you parse the response to call the function that the model returns.

Here are the two forecast functions that we provide in the prompt. The other two historical functions are not shown here as they have similar definitions. Notice that they are quite verbose with human understandable parameter names, a description, and descriptions of all arguments and return values.

```
def get_future_data_for_date(date: str, city_name: str, feature_view, model) ->
pd.DataFrame:
    """
    Predicts future PM2.5 data for a specified date and city using a given fea-
ture view and model.

    Args:
        date (str): The target future date in the format 'YYYY-MM-DD'.
        city_name (str): The name of the city for which the prediction is made.
        feature_view: The feature view used to retrieve batch data.
        model: The machine learning model used for prediction.

    Returns:
        pd.DataFrame: A DataFrame containing predicted PM2.5 values for each
day starting from the target date.

    """

def get_future_data_in_date_range(date_start: str, date_end: str, city_name:
str, feature_view, model) -> pd.DataFrame:
    """
    Retrieve data for a specific date range and city from a feature view.

    Args:
        date_start (str): The start date in the format "%Y-%m-%d".
        date_end (str): The end date in the format "%Y-%m-%d".
        city_name (str): The name of the city to retrieve data for.
        feature_view: The feature view object.
        model: The machine learning model used for prediction.

    Returns:
        pd.DataFrame: A DataFrame containing data for the specified date range
and city.
    """
```

We designed the following prompt template for the function calling query to our LLM as follows. Firstly, we define the available functions, then include the JSON representation of those functions including their parameters, types, and descriptions. The fine-tuned LLM should also be hinted about which function to choose and not to return a function unless it is confident one of them matches the user query.

```
prompt = f"""<|im_start|>system
You are a helpful assistant with access to the following functions:

get_future_data_for_date
get_future_data_in_date_range
get_historical_air_quality_for_date
get_historical_data_in_date_range

{serialize_function_to_json(get_future_data_for_date)}
{serialize_function_to_json(get_future_data_in_date_range)}
{serialize_function_to_json(get_historical_air_quality_for_date)}
```

```
{serialize_function_to_json(get_historical_data_in_date_range)}

You need to choose what function to use and retrieve parameters for this func-
tion from the user input.
IMPORTANT: Today is {datetime.date.today().strftime("%A")}, {date-
time.date.today()}.
IMPORTANT: If the user query contains 'will', it is very likely that you will
need to use the get_future_data function
NOTE: Ignore the Feature View and Model parameters.
NOTE: Dates should be provided in the format YYYY-MM-DD.

To use these functions respond with:
<multiplefunctions>
    <functioncall> {fn} </functioncall>
    <functioncall> {fn} </functioncall>
    ...
</multiplefunctions>

Edge cases you must handle:
- If there are no functions that match the user request, you will respond
politely that you cannot help.<|im_end|>
<|im_start|>user
{prompt}<|im_end|>
<|im_start|>assistant"""
```

The prompt for the second LLM query can be found in the source code repository. It is not shown here as it is straightforward - it includes the results of the function call, the original user query, some domain knowledge about air quality questions, and today's date.

Running the Function Calling Notebook

The *5_function_calling.ipynb* notebook needs a GPU to run efficiently. It also has its own set of Python requirements that you need to install:

```
pip install -r requirements-llm.txt
```

If you do not have one on your laptop, you can use Google Colabatory with a T4 GPU at no cost (you will need a Google account, though). You need to uncomment and run the first two cells in the notebook to install the LLM Python requirements and download some Python modules. The notebook quantizes the weights in the Llama-3-8B to 4-bits, reducing its size in memory so that the LLM will run on a T4 GPU (which has 16GB of RAM). Weight quantization does not appear to negatively affect LLM performance for our system.

There is also a Streamlit program (*streamlit_app.py*) that wraps the same LLM program in a UI. Streamlit is a framework for building a UI as an imperative program written in Python. You can host them in a free serverless service such as streamlit.io or Huggingface Spaces.

Summary

In this chapter, we built our first ML system together - an air quality forecasting service. We decomposed the problem into 5 Python programs in total - a program to create and backfill feature groups, an operational feature pipeline that downloads air quality readings and weather forecasts, a model training pipeline that we run on-demand, a batch-inference pipeline that outputs an air quality forecast chart as well as a hindcast as PNG files, and a LLM-powered program with a voice-driven UI for our service. We also defined a GitHub Action workflow as a YAML file to schedule the feature pipeline and batch inference pipeline to run daily. That was a good chunk of work, but now you have a ML system that you, and your community, can be proud of.

CHAPTER 4
Feature Stores

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. The GitHub repo can be found at *https://github.com/featurestorebook/mlfs-book*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

As we have seen in the first three chapters, data management is one of the most challenging aspects of building and operating AI systems. In the last chapter, we used a feature store to build our air quality forecasting system. The feature store stored the output of the feature pipelines, provided training data for the training pipeline, and inference data for the batch inference pipeline. The feature store is a central data platform that stores, manages, and serves features for both training and inference. It also ensures consistency between features used in training and inference, and enables the construction of modular AI systems by providing a shared data layer and well-defined APIs to connect feature, training, and inference pipelines.

In this chapter, we will dive deeper into feature stores and answer the following questions:

- What problems does the feature store solve and when do I need one?
- What is a feature group, how does it store data, and how do I write to one?

- How do I design a data model for feature groups?
- How do I read feature data spread over many feature groups for training or inference?

We will look at how feature stores are built from a columnar store, a row-oriented store, and a vector index. We describe how feature stores solve challenges related to feature reuse, how to manage time-series data, and how to prevent skew between feature, training, and inference pipelines. We will also weave a motivating example of a real-time AI system that predicts credit card fraud throughout the chapter.

A Feature Store for Fraud Prediction

We start by presenting the problem of how to design a feature store for an AI system that makes real-time fraud predictions for credit card transactions. The ML System Card for the system is shown in Table 4-1.

Table 4-1. ML System Card for our Real-Time Credit Card Fraud Prediction Service.

Dynamic Data Sources	Prediction Problem	UI or API	Monitoring
Credit Card Transactions arrive in an Event Bus. Credit card, issuer, and merchant details in tables are in a Data Warehouse.	Whether a credit card transaction is suspected of fraud or not.	Real-time API that rejects suspected fraud transactions.	Offline investigations of suspected vs actual reported fraud.

The source data for our AI system comes from a Data Mart (consisting of a data warehouse and an event bus (such as Apache Kafka or AWS Kinesis), see Figure 4-1.

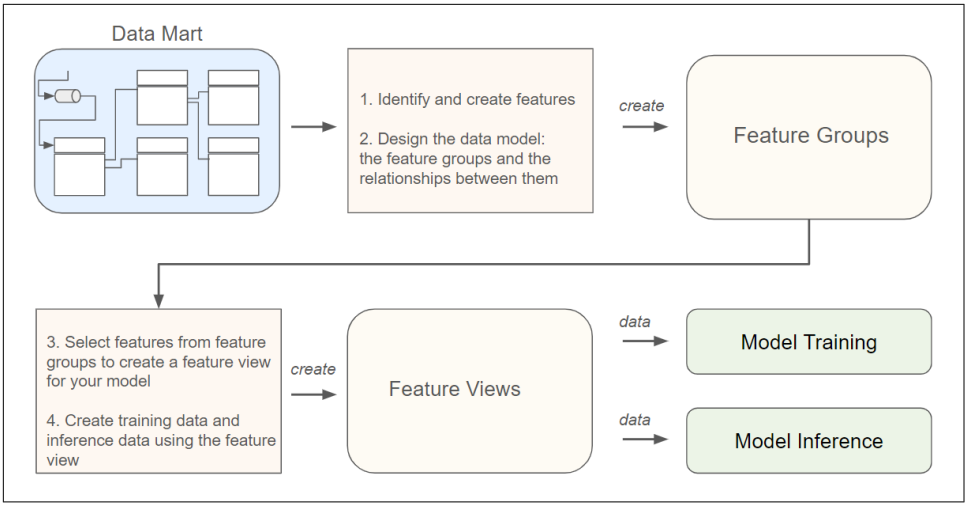


Figure 4-1. We design our feature store by first identifying and creating features from the data sources, organizing the features into tables called feature groups, selecting features from different feature groups for use in a model by creating a feature view, and creating training/inference data with the feature view.

Starting from our data sources, we will learn how to build a feature store with four main steps:

1. identify entities and features for those entities,
2. organize entities into tables of features (feature groups), and identify relationships between feature groups,
3. select the features for a model, from potentially different feature groups, in a feature view
4. retrieve data for model training, batch/online inference with the feature view.

This chapter will provide more details on what feature groups and feature views are, but before that, we will look at the history of feature stores, what makes up a feature store (its anatomy), and when you may need a feature store.

Brief History of Feature Stores

As mentioned in Chapter 1, Uber introduced the first feature store for machine learning in 2017 as part of its Michelangelo platform. Michelangelo includes a feature store (called Palette), a model registry, and model serving capabilities. Michelangelo also introduced a domain-specific language (DSL) to define feature pipelines. In the DSL, you define what type of feature to compute on what data source (such as count the number of user clicks in the last 7 days using a *clicks table*), and Michelangelo

transpiles your feature definition into a Spark program and runs it on a schedule (for example, hourly or daily).

In late 2018, Hopsworks was the first open-source feature store, introducing an API-based feature store, where external pipelines read and write feature data using a DataFrame API and there is no built-in pipeline orchestration. The API-based feature store enables you to write pipelines in different frameworks/languages (for example, Flink, PySpark or Pandas). In late 2019, the open-source Feast feature store adopted the same API-based architecture (*DataSets*) for reading/writing feature data. Now, feature stores from GCP, AWS, and Databricks follow the API-based architecture, while the most popular DSL-based feature store is Tecton. In the rest of this Chapter, we describe the common functionality offered by both API-based and DSL-based feature stores, while in the next chapter, we will look at the Hopsworks Feature Store, which is representative of API-based feature stores.

The term *feature platform* has been used to describe feature stores that support managed feature pipelines (but not managed training or inference pipelines). The *virtual feature store* is a moniker for a feature store that has pluggable offline and online stores. Finally, the *AI Lakehouse* describes a feature store that uses Lakehouse tables as its offline store and has an integrated online store for building real-time AI systems.

The Anatomy of a Feature Store

The feature store is a factory that produces and stores feature data. It enables the faster production of higher quality features by managing the storage and transformation of data for training and inference, and allows you to reuse features in any model. In Figure 4-2, we can see the main inputs, outputs, and the data transformations managed by the feature store.

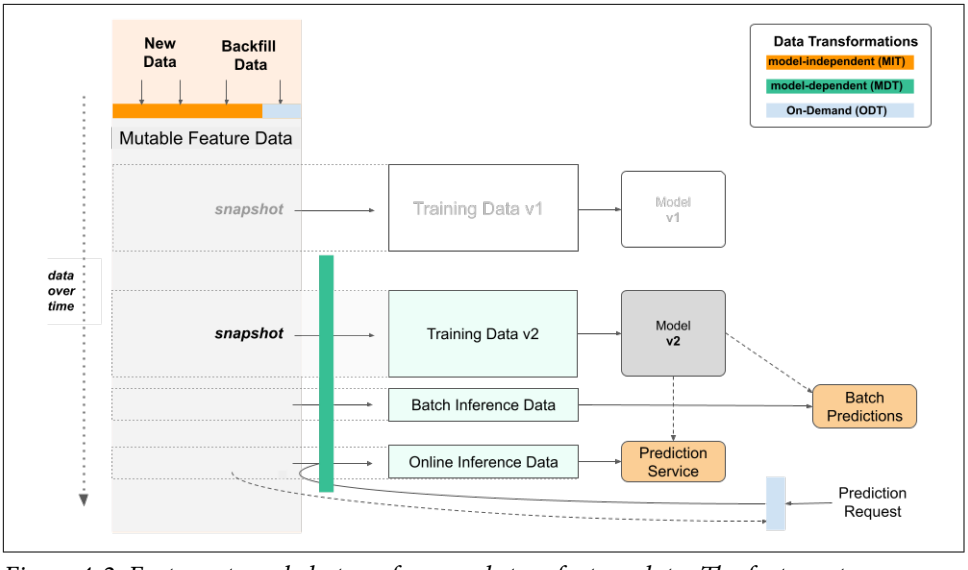


Figure 4-2. Feature stores help transform and store feature data. The feature store organizes the data transformations to create consistent snapshots of training data for models, as well as the batches of inference data for batch AI systems, and the online inference data for real-time AI systems.

Feature pipelines feed the feature store with feature data. They take new data or historical data (backfilling) as input and transform it into reusable feature data, primarily using model-independent transformations. On-demand transformations can also be applied to historical data to create reusable feature data. The programs that execute the model-independent (and on-demand transformations) are called feature pipelines. Feature pipelines can be batch or streaming programs and they update the feature data over time - the feature store stores mutable feature data. For supervised machine learning, labels can also be stored in the feature store and are treated as feature data until they are used to create training or inference data, in which case, the feature store is aware of which columns are features and which columns are labels.

Feature stores enable the creation of versioned training datasets by taking a point-in-time consistent snapshot of feature data and then applying model-dependent transformations to the features (and labels). Training datasets are used to train models, and the feature store should store the lineage of the training dataset for models. The feature store also creates point-in-time consistent snapshots of feature data for batch inference, that should have the same model-dependent transformations applied to them as were applied when creating the training data for the model used in batch inference.

The feature store also provides low latency feature data to online applications or services. They issue prediction requests, and parameters from the prediction request can

be used to compute on-demand features and retrieve precomputed rows of feature data from the feature store. Any on-demand and precomputed features are merged into a feature vector that can have further model-dependent transformations applied to it (the same as those applied in training) before the model makes a prediction with the transformed feature vector.

Feature stores support and organize the data transformations in the taxonomy from Chapter 2. Model-independent transformations (MITs) are applied only in feature pipelines on new or historical data to produce reusable feature data. On-demand transformations (ODTs) are applied in both feature pipelines and online inference pipelines, and feature stores should ensure that exactly the same ODT is executed in the feature and online inference pipelines, otherwise there is a risk of skew. Model-dependent transformations (MDTs) are applied in training pipelines, batch inference pipelines, and online inference pipelines. Again, the feature store should ensure that the same MDT is executed in the training and inference pipelines, preventing skew. In Figure 4-3, you can see examples of directed acyclic graphs (DAGs) of valid and invalid combinations of MITs, ODTs, and MDTs.

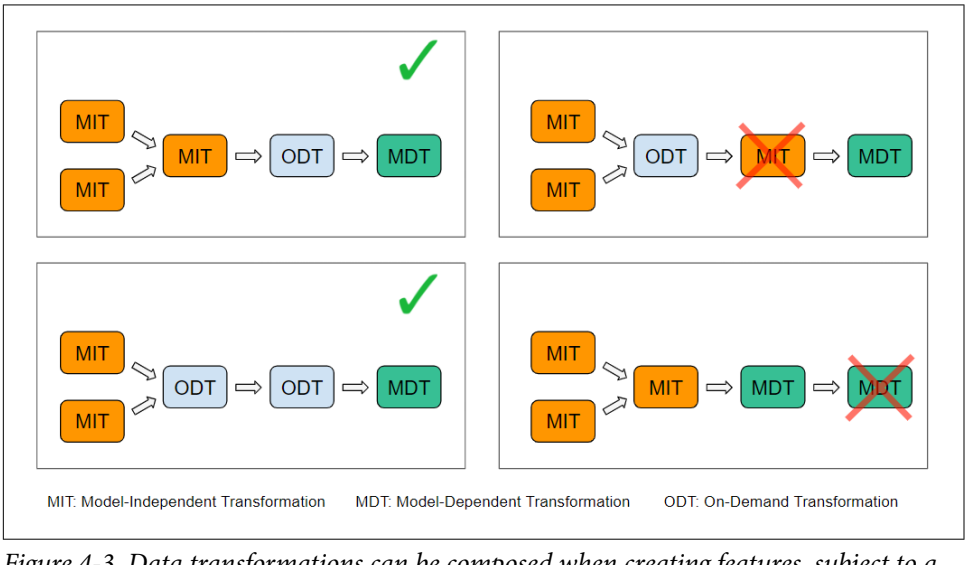


Figure 4-3. Data transformations can be composed when creating features, subject to a couple of constraints: MITs cannot come after ODTs in the DAG, and if there is a MDT, it has to be the last transformation in the DAG and there can be only a single MDT.

Feature stores can support the composition of model-independent transformations (MITs), model-dependent transformations (MDTs), and on-demand transformations (ODTs) in pipelines, subject to the following constraints:

- MDTs are always the last transformation in a DAG (just before the model is called),
- MDTs are not normally composed (for example, you don't encode a categorical feature twice or normalize and then standardize a numerical feature),
- You can build a DAG of MITs and ODTs, but ODTs should not come before MITs in the DAG - in an online inference pipeline, there is no way to execute a MIT after the ODT. If you could run the MIT after the ODT, then, by definition, the MIT would then be an ODT.

This chapter, however, is concerned primarily with the storage, modeling, and querying of the feature data. Chapters 6, 7, and 8 will address the MITs, MDTs, and ODTs.

When Do You Need a Feature Store?

When is it appropriate for you to use a feature store? Many organizations already have operational databases, an object store, and a data warehouse or lakehouse. Why would they need a new data platform? The following are scenarios where a feature store can help.

For Context and History in Real-Time AI Systems

We saw in chapter 1 how real-time AI systems need history and context to make personalized predictions. In general, when you have a real-time prediction problem but the prediction request has low information content, you can benefit from a feature store to provide context and history to enrich the prediction request. For example, a credit card payment has limited information in the prediction request - only the credit card number, the merchant ID (unique identifier), the timestamp and location for the payment, the category of goods purchased, and the amount of money. Building an accurate credit card fraud prediction service with AI using only that input data is almost impossible, as we are missing historical information about credit card payments. With a feature store, you can enrich the prediction request at runtime with history and context information about the credit card's recent usage, the customer details, the issuing bank's details, and the merchant's details, enabling a powerful model for predicting fraud.

For Time-Series Data

Many retail, telecommunications, and financial AI systems are built on time-series data. The air quality and weather data from Chapter 3 is time-series data that we update once per day and store in tables along with the timestamps for each observation or forecast. Time-series data is a sequence of data points for successive points in time. A major challenge in using time-series data for machine learning is how to read

(query) feature data that spread over many tables - you want to read point-in-time correct training data from the different tables without introducing future data leakage or including any stale feature values, see Figure 4-4.

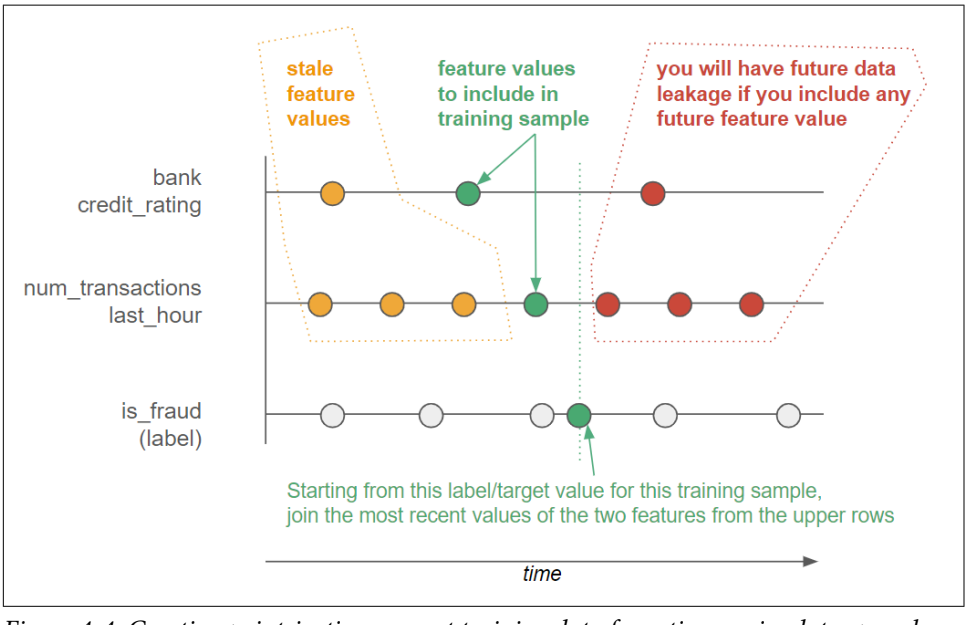


Figure 4-4. Creating point-in-time correct training data from time-series data spread over different relational tables is hard. The solution starts from the table containing the labels/targets (Fraud Label), pulling in columns (features) from the tables containing the features (Transactions and Bank). If you include feature values from the future, you have future data leakage. If you include a feature value that is stale, you also have data leakage.

Feature stores provide support for reading point-in-time correct training data from different tables containing time-series feature data. The solution, described later in this chapter, is to query data with temporal joins. Writing correct temporal joins is hard, but feature stores make it easier by providing APIs for reading consistent snapshots of feature data using temporal joins.

You have probably encountered data leakage in the context of training models - if you leak data from your test set or any external dataset into your training dataset, your model may perform better during testing than when it is used in production on unseen data. Future data leakage is when you build training datasets from time-series data and incorrectly introduce one or more feature data points from the future.

For Improved Collaboration with the FTI Pipeline Architecture

An important reason many models do not reach production is that organizations have silos around the teams that collaborate to develop and operate AI systems. In Figure 4-5, you can see a siloed organization where the data engineering team has a metaphorical wall between them and the data science team, and there is a similar wall between the data science team and the ML engineering team. In this siloed organization, collaboration involves data and models being thrown over the wall from one team to another.

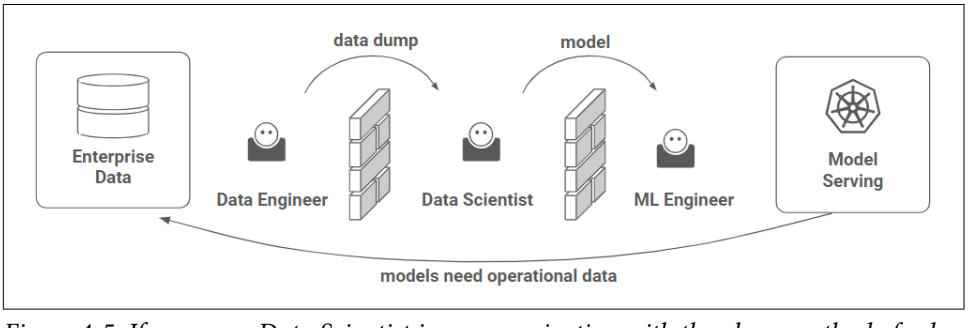


Figure 4-5. If you are a Data Scientist in an organization with the above method of collaboration (where you receive dumps of data and you throw models over the wall to production), Conway's Law implies you will only ever train models and not contribute to production systems.

The system for collaboration at this organization is an example of Conway's Law, where the process for collaboration (throwing assets over walls) mirrors the siloed communication structure between teams. The feature store solves the organizational challenges of collaboration across teams by providing a shared platform for collaboration when building and operating AI systems. The feature, training, and inference (FTI) pipelines from Chapter 2 also help with collaboration. They decompose an AI system into modular pipelines that use the feature store acting as the shared data layer connecting the pipelines. The responsibilities for the FTI pipelines map cleanly onto the teams that develop and operate production AI systems:

- data engineers and data scientists collaborate to build and operate feature pipelines;
- data scientists train and evaluate the models;
- ML engineers write inference pipelines and integrate models with external systems.

For Governance of AI Systems

Feature stores help ensure that an organization's governance processes keep feature data secure and accountable throughout its lifecycle. That means auditing actions taken in your feature store for accountability and tracking lineage from source data to features to models. Feature stores manage mutable data that needs to comply with regulatory requirements, such as the European Union's AI Act that categorizes AI systems into four different risk levels: unacceptable, high, limited, and minimal risk.

Beyond data storage, the feature store also needs support for *lineage* for compliance with other legal and regulatory requirements involving tracking the origin, history, and use of data sources, features, training data, and models in AI systems. Lineage enables the reproducibility of features, training data, and models, improved debugging through quicker root cause analysis, and usage analysis for features. Lineage tells us where AI assets are used. Lineage does not, however, tell you whether a particular feature is allowed to be used in a particular model - for example, a high risk AI system. Access control, while necessary, also does not help here either as it only informs you whether you have the right to read/write the data, not whether your model will be compliant if you use a certain feature. For compliance, feature stores support custom metadata to describe the scope and context under which a feature can be used. For example, you might tag features that have personally identifiable information (PII). With lineage (from data sources to features to training data to models) and PII metadata tags for features, you can easily identify which models use features containing PII data.

For Discovery and Reuse of AI Assets

Feature reuse is a much advertised benefit of feature stores. Meta reported that "most features are used by many models" in their feature store, and the most popular 100 features are reused in over 100 different models each. The benefits of feature reuse include: higher quality features through increased usage and scrutiny, reduced storage cost, and reduced feature development and operational costs, as models that reuse features do not need new feature pipelines. Computed features are stored in the feature store and published to a *feature registry*, enabling users to easily discover and understand features. The feature registry is a component in a feature store that has an API and user interface (UI) to browse and search for available features, feature definitions, statistics on feature data, and metadata describing features.

For Elimination of Offline-Online Feature Skew

Feature skew is when significant differences exist between the data transformation code in either an ODT or MDT in an offline pipeline (a feature or training pipeline, respectively), and the data transformation code for the ODT or MDT in the corresponding inference pipeline. Feature skew can result in silently degraded model per-

formance that is difficult to discover. It may show up as the model not generalizing well to the new data during inference due to the discrepancies in the data transformations. Without a feature store, it is easy to write different implementations for an ODT or MDT - one implementation for the feature or training pipeline and a different one for the inference pipeline. In software engineering, we say that such data transformation code is not DRY (Do not Repeat Yourself). Feature stores support the definition and management of ODTs and MDTs, and ensure the same function is applied in the offline and inference pipelines.

For Centralizing your Data for AI in a single Platform

Feature stores aspire to be a central platform that manages all data needed to train and operate AI systems. Existing feature stores have a dual-database architecture, including an *offline store* and an *online store*. However, feature stores are increasingly adding support for vector database capabilities - vector indexes to store vector embeddings and support similarity search.

The online store is used by online applications to retrieve feature vectors for entities. It is a row-oriented data store, where data is stored in relational tables or in a NoSQL data structure (like key-value pairs or JSON objects). The key properties of row-oriented data stores are:

- low latency and high throughput CRUD (create, read, update, delete) operations using either SQL or NoSQL,
- support for primary keys to retrieve features for specific entities,
- support for time-to-live (TTL) for tables and/or rows to expire stale feature data,
- high availability through replication and data integrity through ACID transactions.

The offline store is a columnar store. Column-oriented data stores are:

- central data platforms that store historical data for analytics,
- low cost storage for large volumes of data (including columnar compression of data) at the cost of high latency for row-based retrieval of data,
- faster complex queries than row-oriented stores through more efficient data pruning and data movement, aided by data models designed to support complex queries.

The offline stores for existing feature stores are lakehouses. The lakehouse is a combination of a data lake for storage and a data warehouse for querying the data. In contrast to a data warehouse, the lakehouse is an open platform that separates the storage of columnar data from the query engines that use it. Lakehouse tables can be queried

by many different query engines. The main open-source standards for the lakehouse are the table formats for data storage (Apache Iceberg, Delta Lake, Apache Hudi). A table format consists of data files (Parquet files) and metadata that enables ACID (atomic, consistent, isolation, durable) updates to the Parquet files - a commit for every batch append/update/delete operation. The commit history is stored as metadata and enables time-travel support for lakehouse tables, where you can query historical versions of tables (using a commit ID or timestamp). Lakehouse tables also support schema evolution (you can add columns to your table without breaking clients), as well as partitioning, indexing, and data skipping for faster queries.

The offline and/or online store may also support storing vector embeddings in a vector index that supports approximate nearest neighbor (ANN) search for feature data. Feature stores either include a separate standalone vector database (such as Weaviate, Pinecone), or an existing row-oriented database that supports a vector index and ANN search (such as Postgres PgVector, OpenSearch, and MongoDB). Now that we have covered why and when you may need a feature store, we will look into storing data in feature stores in feature groups.

Feature Groups

Feature stores use feature groups to hide the complexity of writing and reading data to/from the different offline and online data stores. We encountered feature groups in Chapters 2 and 3, but we haven't formally defined them. Feature groups are tables, where the features are columns and the feature data is stored in offline and online stores. Not all feature stores use the term feature groups - some vendors call them *feature sets* or *feature tables*, but they refer to the same concept. We prefer the term feature group as the data is potentially stored in a group of tables - more than one store. We will cover the most salient and fundamental properties of feature groups employed by existing feature stores, but note that your feature store might have some differences, so consult its documentation before building your feature pipelines.

A feature group consists of a schema, metadata, a table in an offline store, an optional table in an online store, and an optional vector index. The metadata typically contains the feature group's:

- *name*
- *version* (a number)
- *entity_id* (a primary key, defined over one or more columns)
- *onlined_enabled* - whether the feature group's online table is used or not
- *event_time* column (optional)
- tags to help with discovery and governance.

The *entity_id* is needed to retrieve rows of online feature data and prevent duplicate data, while the *version* number enables support for A/B tests of features by different models and enables schema breaking changes to feature groups. The *event_time* column is used by the feature store to create point-in-time consistent training data from time-series feature data. Depending on your feature store, a feature group may support some or all of the following:

- *foreign_key* columns (references to a primary key in another feature group)
- a *partition_key* column (used for faster queries through partition pruning)
- *vector embedding* features that are indexed for similarity search
- *feature definitions* that define the data transformations used to create the features stored in the feature group.

In Figure 4-6, we can see a feature group containing different columns related to credit-card payments. You will notice that most columns are not feature columns.

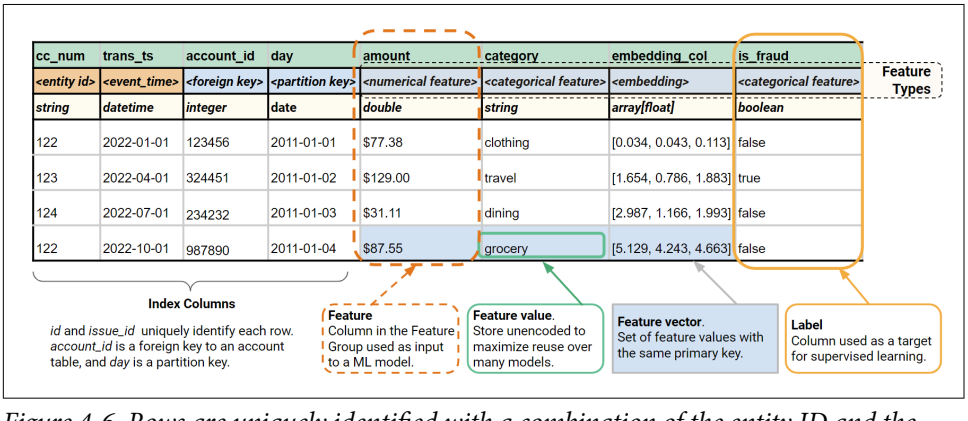


Figure 4-6. Rows are uniquely identified with a combination of the entity ID and the event_time. You can have a foreign key that points to a row in a different feature group, and a partition key, used for push-down filters for faster queries. The index columns are not features. Any feature could be used as a label when creating training data from the feature group.

The first four columns are collectively known as *index columns* - the `cc_num` (*entity id*), `trans_ts` is the *event_time*, `account_id` is a *foreign key* to an `account_details` feature group (not shown), and `day` is a *partition_key* column enabling queries that filter by *day* to be faster by only reading the needed data (for example, read yesterday's feature data will not read all rows, only the rows where *day* value is yesterday). The next 3 columns (*amount*, *category*, and *embedding_col*) are features - the *embedding_col* is a vector embedding that is indexed for similarity search in the vector index. Finally, the *is_fraud* column is also a feature column but is identified as a 'label'

in the figure. That is because features can also be labels - the *is_fraud* column could be a label in one model, but a feature in another model. For this reason, labels are not defined in feature groups, but only defined when you select the features and labels for your model.

You can perform inserts, updates, and deletes on feature groups, either via a batch (DataFrame) API or a streaming API (for real-time AI systems). As a feature group has a schema, your feature store defines the set of supported data types for features - strings, integers, arrays, and so on. In most features, you can either explicitly define the schema for a feature group or the feature store will infer its schema using the first DataFrame written to it. If a feature group contains time-series data, the *event_time* column value should reflect the time the feature values in that row were created (not when the row of data was ingested). If the feature group contains non time-series data, you can omit the *event_time* column.

The entity ID is a unique identifier for an entity that has features in the modeled world. The entity ID can be either a natural key or a surrogate key**.** An example of a natural key is an email address or social security number for a user, while an example of a surrogate key is a sequential number, such as an auto increment number, representing a user.

Feature Groups store untransformed feature data

Feature pipelines write untransformed feature data to feature groups. The MDTs, such as encoding a categorical feature, are performed in training and inference pipelines after reading feature data from the feature store. In general, feature groups should not store transformed feature values (that is, MDTs should not have been applied) as:

- The feature data is not reusable across models (model-specific transformations transform the data for use by a single model or set of related models).
- It can introduce *write amplification*. If the MDT is parameterized by training data, such as standardizing a numerical feature, the time taken to perform a write becomes proportional to the number of rows in the feature group, not the number of rows being written. For standardization, this is because updates first require reading all existing rows, recomputing the mean and standard deviation, then updating the values of all rows with the new mean and standard deviation.
- Exploratory data analysis works best with unencoded feature data - it is hard for a data scientist to understand descriptive statistics for a numerical feature that has been scaled.

Feature Definitions and Feature Groups

A feature definition is the source code that defines the data transformations used to create one or more features in a feature group. In API-based feature stores, this is the source code for your MITs (and ODTs) in your feature pipelines. For example, this could be a Pandas, Polars, or Spark program for a batch feature pipeline. In DSL-based feature stores, a feature definition is not just the declarative transformations that create the features, but also the specification for the feature pipeline (batch, streaming, or on-demand).

Writing to Feature Groups

Feature stores provide an API to *ingest* feature data. The feature store manages the complexity of then updating the feature data in the offline store, online store, and vector index on your behalf - the updates in the background are *transparent* to you as a developer. Figure 4-7 shows two different types of APIs for ingesting feature data. In Figure 4-7(a), you have a single batch API for clients to write feature data to the offline store. The offline store is normally a lakehouse table and they provide change data capture (CDC) APIs where you can read the data changes for the latest commit. A background process either runs periodically or continually and reads any new commits since the last time it ran and copies them to the online store and/or vector index. For feature groups storing time-series data, the online store only stores the latest feature data for each entity (the row with the most recent *event_time* key value for each primary key).

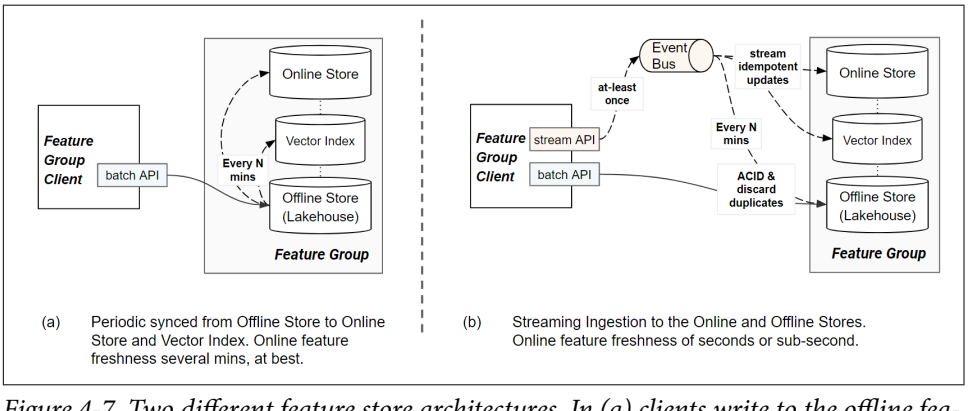


Figure 4-7. Two different feature store architectures. In (a) clients write to the offline feature store and updates are periodically synchronized to the online store and vector index. In (b) clients can also write via a stream API to an event bus, after which writes are streamed to the online store and vector index, then periodically synchronized to the offline store.

In Figure 4-7(b), there are two APIs - a *batch API* and a *stream API*. Clients can use the batch API to write to only the offline store. If a feature group is *online_enabled*, clients write to the steam API. Clients that write to the stream API can be either batch programs (Spark, Pandas, Polars) or stream processing programs (Flink, Spark Streaming). The difference with the stream API is that updates are written first to the online store and vector index (here via an event bus), and then synchronized periodically with the offline store. Feature data is available at lower latency in the online store via the stream API - that is, the stream API enables *fresher* features. For feature groups storing time-series data, the online store can again store either the latest feature data for each entity (the row with the most recent *event_time* key value for each primary key). Some online feature stores with a stream API also support computing aggregations as ODTs (for example, max amount for a credit card transaction in the last 15 minutes), and, in this case, a TTL can be specified for each row or table so that feature data is removed when its TTL has expired.

Feature Freshness

The *freshness* of feature data in feature groups is defined as the total time taken from when an event is first read by a feature pipeline to when the computed feature becomes available for use in an inference pipeline, see Figure 4-8. It includes the time taken for feature data to land in the online feature store and the time taken to read from the online store.

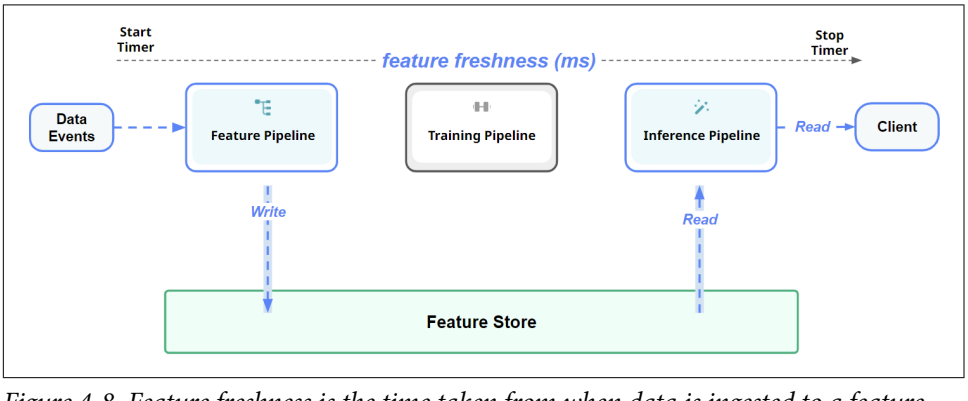


Figure 4-8. Feature freshness is the time taken from when data is ingested to a feature pipeline to when the feature(s) computed is available for reading by clients.

Fresh features for real-time AI systems typically require streaming feature pipelines that update the feature store via a stream API. In Chapter 1, we described how TikTok is a real-time AI system - when you swipe or click, features are created using information about your viewing activity using streaming feature pipelines, and within a couple of seconds they are available as precomputed features in feature groups for predictions. If it took minutes, instead of seconds, TikTok's recommender would not

feel as if it tracks your intent in real-time - it's AI would be too laggy to be useful as a recommender.

Data Validation

Some feature stores support *data validation* when writing feature data to feature groups. For each feature group, you specify constraints for valid feature data values. For example, if the feature is an adult user's age, you might specify that the age should be greater than 17 and less than 125. Data validation helps avoid problems with data quality in feature groups. Note that there are some exceptions to the general "garbage-in, garbage-out" principle. For example, it is often ok to have missing feature values in a feature group, as you can impute those missing values later in your training and inference pipelines.

Now that we've covered what a feature group is, what it stores, and how you write to one, let's now look at how to design a data model for feature groups.

Data Models for Feature Groups

If the feature store is to be the source of our data for AI, we need to understand how to model the data stored in its feature groups. Data modeling for feature stores is the process of deciding:

- what features to create for which entities and what features to include in feature groups,
- what relationships between the feature groups look like,
- what the freshness requirements for feature data is,
- and what type of queries will be performed on the feature groups.

Data modeling includes the design of a data model. A data model is a term from database theory, that refers to how we decompose our data into different feature groups (tables), with the goals of:

- ensuring the integrity of the data,
- improving the performance of writing the data,
- improving the performance of reading (querying) the data,
- improving the scalability of the system as data volumes and/or throughput increases

You may have heard of Entity-Relationship diagrams (see Figure 4-8, for example) from relational databases. It is a way of identifying *entities* (such as credit card transactions, user accounts, bank details, and merchant details) and the relationships

between those entities. For example, a credit card transaction could have a reference (*foreign key*) to the credit card owner's account, the bank that issued the card, and the merchant that performed the transaction. In the relational data model, entities typically map to tables and relationships to foreign keys. Similarly, in feature stores an entity maps to a feature group and relationships map to foreign keys in a feature group.

What is the process to go from requirements and data sources to a data model for feature groups, such as an Entity Relationship Diagram? There are 2 basic techniques we can use:

Normalization
: Reduce data redundancy and improve data integrity,

Denormalization
: Improve query performance by increasing data redundancy.

These two techniques produce data models that can be categorized into one of two types: denormalized data models that include redundant (duplicated) data and normalized data models that eliminate redundant data. The benefits and drawbacks of both approaches are shown in Table 4-2.

Table 4-2. Comparison of denormalized data models versus normalized data models.

	Denormalized Data Model	Normalized Data Model
Data Storage Costs	Higher due to redundant data in the (row-oriented) online store	Lower due to no redundant data
Query Complexity	Lower, due to less need for JOINs when reading from the online store	Higher, due to more JOINs needed when querying data

In general, denormalized data models are more prevalent in columnar data stores (lakehouses and data warehouses) as they can often efficiently compress redundant data in columns with columnar compression techniques like run-length encoding, while row-oriented data stores cannot compress redundant data, and, therefore, favor normalized data models.

Before we start identifying entities, features, and feature groups for entities/features, we should consider the types of AI systems that will use the feature data:

1. batch AI systems
2. real-time AI systems

For (1), feature groups only need to store data in their offline store. As such, we could consider existing data models for columnar stores, such as the star schema, snowflake schema that are widely used in analytical and business intelligence environments. For (2), we have feature groups with tables in both the offline and online store. For this,

we should use a general purpose data model that works equally well for both batch and real-time queries. We will see in the next section that the snowflake schema (a normalized data model) is our preferred methodology for data modeling in feature stores, although some feature stores only support the star schema, so we will introduce both data models. The star schema and snowflake schema are data models that organize data into a fact table that connects to dimension tables. In the star schema, columns in the dimension tables can be redundant (duplicated), but the snowflake schema extends the star schema to enable dimension tables to be connected to other dimension tables, enabling a normalized data model with no redundant data. We will now look at how to design a star schema or snowflake schema data model with fact and dimension tables using *dimension modeling*.

Other popular data models used in columnar stores include the *data vault model* (used to efficiently handle data ingestion, where data can arrive late and schema changes happen frequently), and the *one big table* (OBT) data model (which simplifies data modeling by storing as much data as possible in a single wide table). OBT is not suitable for AI systems, as it would store all the labels and features in a single denormalized table which would explode storage requirements in the (row-oriented) online store, and it is not suited for storing feature values that change over time. You can learn more on data modeling in the book *"Fundamentals of Data Engineering"*.

Dimension modeling with a Credit Card Data Mart

The most popular data modeling technique in data warehousing is dimension modeling that categorizes data as facts and dimensions. Facts are usually measured quantities, but can also be qualitative. Dimensions are attributes of facts. Some dimensions change value over time and are called *slowly changing dimensions* (SCD). Let's look at an example of facts and dimensions in a credit card transactions *data mart*. A data mart is a subset of a data warehouse (or lakehouse) that contains data focused on a specific business line, team, or product.

In our example, the credit card transactions are the facts and the dimensions are data about the credit card transactions, such as the card holder, their account details, the bank details, and the merchant details. We will use this data mart to power a real-time AI system for predicting credit card fraud. But first, let's look at our data mart, illustrated in an Entity-Relationship Diagram in Figure 4-9 using a snowflake schema data model.

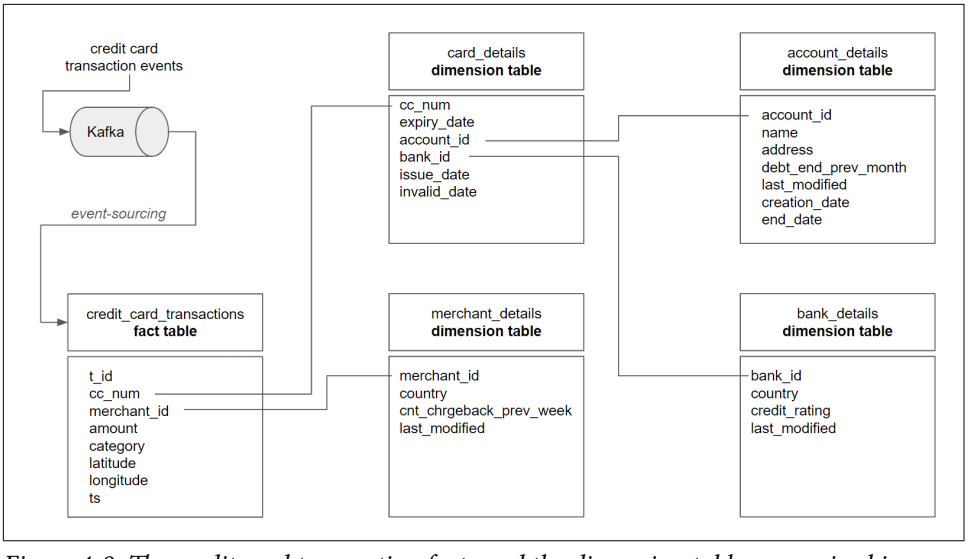


Figure 4-9. The credit card transaction facts and the dimension tables, organized in a snowflake schema data model. The lines between the tables represent the foreign keys that link the tables to one another. For example, card_details includes a reference to the account that owns the card (account_details) and the bank that issued the card (bank_details).

The *fact table* stores:

credit_card_transactions
: A unique ID for the transaction (*t_id*), the credit card number (*cc_num*), a timestamp for the transaction (*event_time*), the amount of money spent (*amount*), the location (*longitude* and *latitude*), and the *category* of the item purchased.

The dimension tables for the credit card transactions are:

card_details
: Its expiry date, issue date, the date if the card has been invalidated, and foreign keys to account and bank details tables (the foreign keys make this a snowflake schema data model).

account_details
: Name, address, debt at the end of the previous month, date when the account was created and closed (*end_date*), and date when a row was *last_modified*.

bank_details
: Credit rating, country, the date when a row was *last_modified*.

merchant_details

Count of chargebacks for the merchant in the previous week (*cnt_chrgeback_pre_week*), its country, and date when a row was *last_modified*.

The credit card transactions table is populated using the *event sourcing* pattern, whereby once per hour, an ETL Spark job reads all the credit card transactions that arrived in Kafka during the previous hour, and persists the events as rows in the *credit_card_transactions* table. The dimension tables are updated by ETL or ELT pipelines that read changes to dimensions for operational databases (not shown). We will now see how we can use the credit card transaction events in Kafka and the dimension tables to build our real-time fraud detection AI system.

Labels are Facts and Features are Dimensions

In a feature store, the facts are the labels (or targets/observations) for our models, while the features are dimensions for the labels. Like facts, the labels are immutable events that often have a timestamp associated with them. For example, in our credit card fraud model, we will have a label *is_fraud* for a given credit card transaction and a timestamp for when the credit card transaction took place. The features for that model will be the card usage statistics, details about the card itself (expiry date), the cardholder, the bank, and the merchant. These features are dimensions for the labels, and they are often mutable data. Sometimes they are SCDs, but in real-time machine learning systems, they might be fast changing dimensions. Irrespective of whether the feature values change slowly or quickly, if we want to use a feature as training data for a model, it is crucial to save all values for features at all points in time. If you don't know when and how a feature changes its value over time, then training data created using that feature could have future data leakage or include stale feature values.

Dimension modeling in data warehousing introduced SCD types to store changing values of dimensions (features). There are at least 5 well-known ways to implement SCDs (SCD Types), each optimized for different ways a dimension could change. Implementing different SCD Types in a data mart is a skilled and challenging job. However, we can massively simplify managing SCDs for feature stores for two reasons. Firstly, as feature values are observations of measurable quantities, each new feature value replaces the old feature value (a feature cannot have multiple alternative values at the same time). Secondly, there are a limited number of query patterns for reading feature data - you read training data and batch inference data from the offline store and rows of feature vectors from the online store. That is, feature stores do not need to support all 5 SCD Types, instead they need a very specific set of SCD Types (0, 2, and 4), and support for those types can be unobtrusively added to feature groups by simply specifying the *event_time* column in your feature group. This way, feature stores simplify support for SCDs compared to general purpose data warehouses.

Table 4-3 shows how feature stores implement SCD Types 0, 2, and 4 with the relatively straightforward approach of specifying the feature group column that stores the *event_time*.

Table 4-3. Feature stores implement variants of SCD Types 0, 2, and 4.

SCD Type	Usage	Description	Feature Store
Type 0	Immutable feature data	No history is kept for feature data, suitable for features that are immutable.	Feature Group, no *event_time*
Type 2	Mutable feature data used by batch AI systems	When a feature value is updated for an entity ID, a new row is created with a new *event_time* (but the same entity ID). Each new row is a new version of the feature data.	Offline Feature Group with *event_time*
Type 4	Online features for real-time AI Systems. Offline data for training.	Features are stored as records in two different tables - a table in the online store with the latest feature values and a table in the offline store with historical feature values.	Online/Offline Feature Group with *event_time*

Type 0 SCD is a feature group that stores immutable feature data. If you do not define the *event_time* column for your feature group, you have a feature group with Type 0 SCD. Type 2 SCD is an offline-only feature group (for batch AI systems), where we have the historical records for the time-series data. In classical Type 2 SCD, it is assumed that rows need both an *end_date* and an *effective_date* (as multiple dimension values may be valid at any point-in-time). However, in the feature store, we don't need an *end_date*, only the *effective_date*, called the *event_time*, as only a single feature value is valid at any given point-in-time. Type 4 SCD is implemented as a feature group, backed by tables in both the online and offline stores. A table in the online store stores the latest feature data values, and a table with the same name and schema in the offline store stores all of the historical feature data values. In traditional Type 4 SCD, the historical table does not store the latest values, but in feature stores, the offline store stores both the latest feature values and the historical values.

Feature stores hide the complexity of designing a data model that implements these 3 different SCD Types by implementing the data models in their read/write APIs. For example, in the AWS Sagemaker feature store (an API-based feature store), you only need to specify the event time column when defining a feature group:

```
feature_group.create(
    description = "Some info about the feature group",
    feature_group_name = feature_group_name,
    event_time_feature_name = event_time_feature_name,
    enable_online_store = True,
    ...
    tags = ["tag1","tag2"]
)
```

Writes to this feature group will create Type 4 SCD features, with the latest feature data in a key-value store, ElastiCache or DynamoDB, and historical feature data in a columnar store (Apache Iceberg).

Real-Time Credit Card Fraud Detection AI System

Let's now start designing our real-time AI system to predict if a credit card transaction is fraudulent. This operational AI system (online inference pipeline) has a service level objective (SLO) of 50ms latency or lower to make the decision on suspicion of fraud or not. It receives a prediction request with the credit card transaction details, retrieves precomputed features from the feature store, computes any on-demand features, merges the precomputed and on-demand features in a single feature vector, applies any model-dependent transformations, makes the prediction, logs the prediction and the untransformed features, and returns the prediction (fraud or not-fraud) to the client.

To build this system and meet our SLO, we will need to write a streaming feature pipeline to create features directly from the events from Kafka, as shown in Figure 4-8. Stream processing enables us to compute aggregations on recent historical activity on credit cards, such as how often a card has been used in the last 5 minutes, 15 minutes, or hour. These features are called *windowed aggregations*, as they compute an aggregation over events that happen in a window of time. It would not be possible to compute these features within our SLO if we only use the *credit_card_transaction* table in our data mart, as it is only updated hourly. We can, however, compute other features from the data mart, such as the credit rating of the bank that issued the credit card, and the number of chargebacks for the merchant that processed the credit card transaction.

We will also create on-demand features from the input request data. A feature with good predictive power for geographic fraud attacks is the distance and time between consecutive credit card transactions. If the distance is large and the time is short, that is often indicative of fraud. For this, we compute *haversine_distance* and *time_since_last_transaction* features. These on-demand features are computed at runtime with an on-demand transformation function that takes one or more parameters passed as part of the prediction request. On-demand features can additionally take precomputed features from feature groups as parameters.

We have described here an AI system that contains a mix of features computed using stream processing, batch processing, and on-demand transformations. However, when we want to train models with these features, the training data will be stored in feature groups in the feature store. So, we need to identify the features and then design a data model for the feature groups.

Data Model for our Real-Time Fraud Detection AI System

We are using a supervised ML model for predicting fraud, so we will need to have some labeled observations of fraud. For this, there is a new `cc_fraud` table, not in the data mart, with a *t_id* column (the unique identity for credit card transactions) that contains the credit card transactions identified as fraudulent, along with columns for the person who reported the fraud and an explanation for why the transaction is marked as fraudulent. The fraud team updates the `cc_fraud` table weekly in a Postgres database they manage. Using the data mart and the event bus, we can create features that have predictive power for fraud and the labels, as shown in Table 4-4.

Table 4-4. Features we can create from our data mart and event bus for credit card fraud.

Data Sources	Simple Features	Engineered Features
credit_card_transactions	amount category	{num}/{sum}_trans_last_10_mins {num}/{sum}_trans_last_hour {num}/{sum}_trans_last_day {num}/{sum}_trans_last_week prev_ts_transaction prev_loc_transaction haversine_distance time_since_last_transaction
credit_card_transactions cc_fraud		is_fraud
credit_card_transactions card_details table		days_until_expired max_debt_last_12_months
account_details table	debt_end_prev_month	max_debt_last_12_months
mechant_details table	cnt_chrgeback_prev_week	cnt_chrgeback_prev_month
bank_details table	credit_rating	days_since_bank_cr_changed

There are many frameworks and programming languages that we could use to create these features, and we will look at source code for them in the next few chapters. For now, we are interested in the data model for our feature groups that we will design to store and query these features, as well as the fraud labels. The feature groups will need to be stored in both online and offline stores, as we will, respectively, use these features in our real-time AI system for inference and in our offline training pipeline. We will now design two different data models, first using the star schema and then using the snowflake schema.

Star Schema Data Model

The star schema data model is supported by all major feature stores. In Figure 4-10, we can see that the feature group containing the fraud labels is called a *label feature group*.

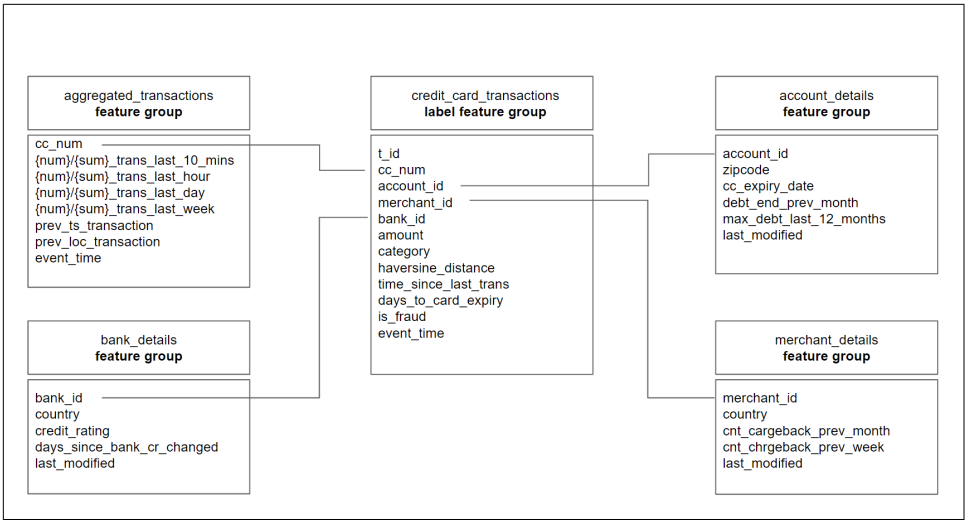


Figure 4-10. Star schema data model for our credit card fraud prediction AI system. Labels (and on-demand features) are the facts, while feature groups are the dimension tables.

The feature group that contains the labels for our credit card transaction (fraud or not-fraud) is known as the *label feature group*. In practice, a label feature group is just a normal feature group. As we will see later, it is only when we select the features and labels for our model that we need to identify the columns in feature groups as either a feature or a label.

Some feature stores do not support storing labels in feature groups. Instead, for these feature stores, clients provide the labels, label timestamps (*event_time*), and entity IDs for feature groups (containing features they want to include) when creating training data and inference data. In the Feast feature store, clients provide the labels, label timestamps, and entity IDs in a DataFrame called the *Spine DataFrame*. The Spine DataFrame contains the same data as our label feature group, but it is not persisted to the feature store. The Spine DataFrame approach can be more flexible for prototyping, as it can also contain additional columns (features) for creating training data. Instead of having to first create your features and write them to a feature group, you can include them as columns in your Spine DataFrame. However, be warned as additional columns can result in skew - it is your responsibility to ensure that any additional columns provided when creating training data are also included (in the same order, with the same data types) when reading inference data.

In Figure 4-10, you can see that the label feature group contains foreign keys to the 4 feature groups that contain features computed from the data mart tables and the event bus. These feature groups are all updated independently in separate feature pipelines that run on their own schedule. For example, the *aggregated_transactions* feature group is computed by a streaming feature pipeline, while the *account_details*, *bank_details*, and *merchant_details* feature groups are computed by batch jobs that run daily.

Snowflake Schema Data Model

The snowflake schema is a data model that, like the star schema, consists of tables containing labels and features. In contrast to the star schema, however, the feature data is normalized, making it suitable as a data model for both online and offline tables. Each feature is split until it is normalized, see Figure 4-11. That is, there is no redundancy in the feature tables, no repetition of values (except for foreign keys that point to primary keys).

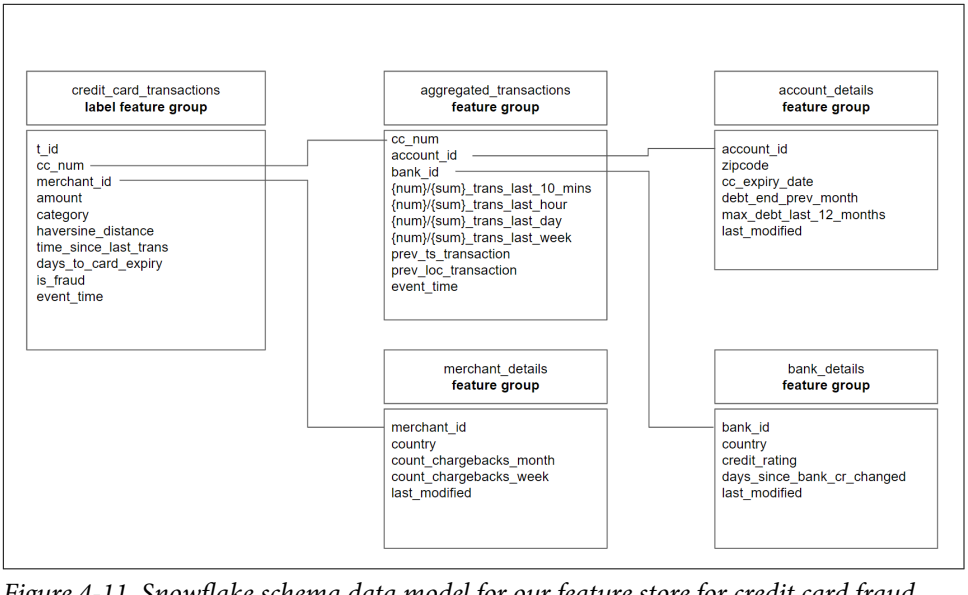


Figure 4-11. Snowflake schema data model for our feature store for credit card fraud prediction.

In the snowflake schema, you can see that the label feature group now only has 2 foreign keys, compared to 4 foreign keys in the star schema data model. As we will see in the next section, the advantage of the snowflake schema here over the star schema is clearest when building a real-time AI system. In a real-time AI system, the foreign keys in the label feature groups need to be provided as part of prediction requests by clients. With a snowflake schema, clients only need to provide the *cc_num* and *mer-*

chant_id as request parameters in order to retrieve all of the features - features from the nested tables are retrieved with a subquery. In the star schema, however, our real-time AI system needs to additionally provide the *bank_id* and *account_id* as request parameters. This makes the real-time AI system more complex, as the client has to also provide the values for *bank_id* and *account_id*.

Feature Store Data Model for Inference

Labels are obviously not available during inference - our model predicts them. Similarly, none of the values of the features in our label feature group (*credit_card_transactions*) are available as precomputed features at online inference time (either for the star schema or snowflake data model). They are all either passed as feature values in the prediction request (the foreign keys to the feature groups and the *amount* and *category* features) or computed as on-demand feature values using request parameters (*time_since_last_trans*, *haversine_distance, days_to_card_expiry*). For this reason, the label feature group is offline only. Its rows can be computed using historical data to create offline training data, but for online inference all of its columns are either passed as parameters, computed, or predicted (the label(s)).

Online Inference

For online inference, a prediction request includes as parameters the foreign keys, any passed features from the label feature group, and any parameters needed to compute on-demand features, see Figure 4-12. The online inference pipeline uses the foreign keys to retrieve all the precomputed features from online feature groups. Feature stores provide either language level APIs (such as Python) or a REST API to retrieve the precomputed features.

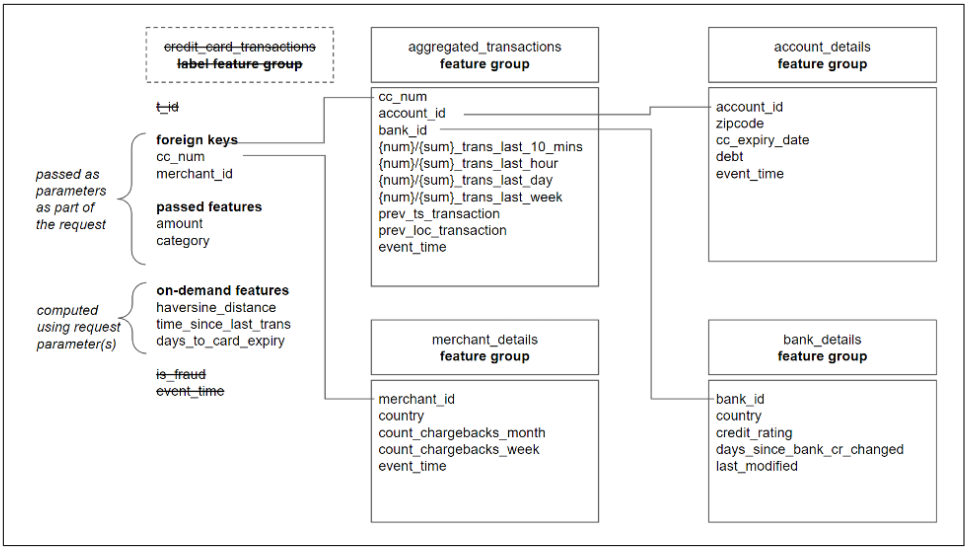


Figure 4-12. During online inference, the rows in the label feature group are not available as precomputed values. Instead, the parameters in a prediction request provide the foreign keys to the feature groups (cc_num and merchant_id), some features are provided as parameters (amount, category), and some features are computed as on-demand features (haversine_distance, time_since_last_trans, days_to_card_expiry).

Batch Inference

Batch inference has similar data modeling challenges to online inference. In Chapter 11, we will re-imagine our credit card fraud prediction problem as a daily batch job that, for all of yesterday's credit card transactions, predicts whether they were fraudulent or not. In this case, the labels are not available, of course, but all features in the label feature group can be populated by a feature pipeline ahead of time. This includes computing the on-demand features using historical data. In this case, the on-demand and passed features are updated at a different cadence from the labels, and, as such, it is often beneficial to move labels into their own feature group, separate from the on-demand features.

Feature stores often support batch inference data APIs, such as:

1. read all feature data that have arrived in the last 24 hours and return them as a DataFrame, or
2. read all the latest feature data for a batch of entities (such as all users or all users who live in Sweden).

An alternative API is to allow batch inference clients to provide a Spine DataFrame containing the foreign keys (and timestamps) for features. The feature store takes the

Spine DataFrame and adds columns containing the feature values from the feature groups (using the foreign keys and timestamps to retrieve the correct feature values). The Spine DataFrame approach does not work well for case (1), but works well for case (2) above. You have to do the work of adding all foreign keys to the Spine DataFrame, which is easy if we want to read the latest feature values for all users, and we pass a Spine DataFrame containing all user IDs. However, reading all feature data since yesterday requires a more complex query over feature groups, and, here, dedicated batch inference APIs to support such queries are helpful.

Reading Feature Data with a Feature View

After you have designed a data model for your feature store, you need to be able to query it to read training and inference data. Feature stores do not provide full SQL query support for reading feature data. Instead, they provide language level APIs (Python, Java, etc) and/or a REST API for retrieving training data, batch inference data, and online inference data. But, reading precomputed feature data is not the only task for a feature store. The feature store should also apply any model-dependent transformations and on-demand transformations before returning feature data to clients.

Feature stores provide an abstraction that hides the complexity of retrieving/computing features for training and inference for a specific model (or group of related models) called a *feature view*.

The feature view is a selection of features and, optionally, labels to be used by one or more models for training and inference. The features in a feature view may come from one or more feature groups.

When you have defined a feature view, you can typically use it to:

- retrieve point-in-time correct training data,
- retrieve point-in-time correct batch inference data,
- build feature vectors for online inference by reading precomputed features and merging them with on-demand and passed features,
- apply model-dependent transformations to features when reading feature data for training and inference without introducing offline-online skew,
- apply on-demand transformations in online inference pipelines.

The feature view prevents skew between training and inference by ensuring that the same ordered sequence of features is returned when reading training and inference data, and that the same model-dependent transformations are applied to the training and inference data read from the feature store. Feature views also apply on-demand

transformations in online inference pipelines and ensure they are consistent with the feature pipeline.

For training and batch inference data, feature stores support reading data as either DataFrames or files. For small data volumes, Pandas DataFrames are popular, but when data volumes exceed a few GBs, some feature stores support reading to Polars and/or Spark DataFrames. Spark DataFrames are, however, not that widely used for training pipelines (Python ML frameworks are used to train the vast majority of models). For large amounts of data (that don't fit in a Polars or Pandas DataFrame), feature stores support creating training data as files in an external file system or object store, in file formats such as Parquet, CSV, and TFRecord (TensorFlow's row-oriented file format that is also supported by PyTorch).

Different feature stores use different names for feature views, including *FeatureLookup (Databricks)* and *FeatureService (Feast, Tecton)*. I prefer the term feature view due to its close relationship to views from relational databases - a feature view is a selection of columns from different feature groups and it is metadata-only (feature views do not store data). A feature view is also not a service when it is used in training or batch inference pipelines, and it is not just a selection of features (as implied by a *FeatureLookup*). For these reasons, we use the term feature view.

All major feature stores support on-demand transformations, but, at the time of writing, Hopsworks is the only feature store to support model-dependent transformations in feature views. Not all feature stores implement on-demand transformations according to the data transformation taxonomy presented in this book, either. For example, Databricks' on-demand transformations are applied in training and online inference pipelines. That is, Databricks' on-demand transformations produce model-specific features (as they cannot be used in feature pipelines to create reusable features).

Point-in-Time Correct Training Data with Feature Views

Now we look at how feature views create point-in-time correct training data using temporal joins (see Figure 4-4 earlier). A temporal join processes each row from the table containing the labels, and uses each row's *event_time* value to join columns from other feature tables, where the joined rows from the feature tables are the rows that have their own *event_time* value that is closest to, but not greater than the label's *event_time* value. If there are no matching rows in the feature tables, the temporal join should return null values.

This temporal join is implemented as an ASOF LEFT (OUTER) JOIN, where the query starts from the table containing the labels, pulling in columns (features) from the tables containing the features, with the ASOF condition ensuring there is no future data leakage for the joined feature values and the LEFT OUTER JOIN condition ensuring rows are returned even if feature values are missing in the resultant

training data. The number of rows in the training data should be the same as the number of rows in the table containing the labels.

In Figure 4-13, we can see how the ASOF LEFT JOIN creates the training data from 4 different feature groups (we omitted the *account_details* feature group for brevity). Starting from the label feature group (*credit_card_transactions*), it joins in features from the other 3 feature groups (*aggregated_transactions*, *bank_details*, *merchant_details*), as of the *event_time* in *credit_card_transactions*.

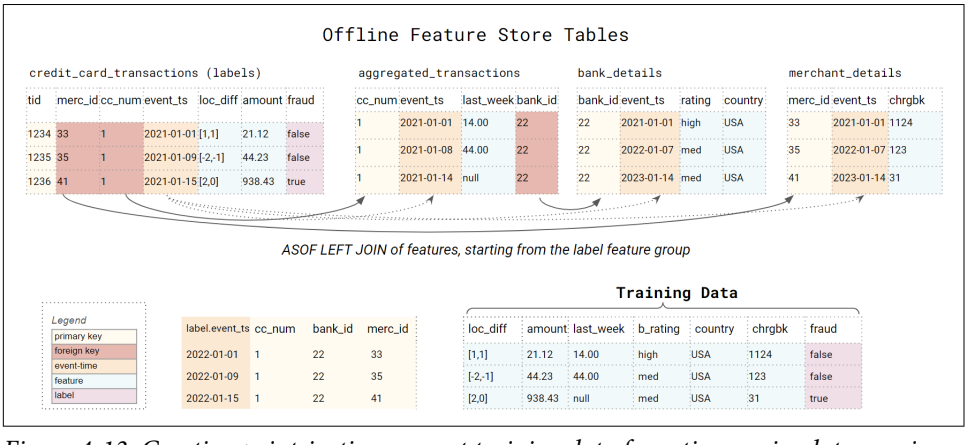


Figure 4-13. Creating point-in-time correct training data from time-series data requires an ASOF LEFT (OUTER) JOIN query that starts from the table containing the labels, pulling in columns (features) from the tables containing the features, with the ASOF condition ensuring there is no future data leakage for the feature values.

For example, in our credit card fraud data model, if we want to create training data from the 1st January 2022, we could execute the following nested ASOF LEFT JOIN on our label table and feature tables:

```
SELECT label.loc_diff, label.amount, aggs.last_week, bank.country,
  bank.credit_rating as b_rating, merchant.chrgbk, label.fraud
  FROM credit_card_transactions as label
  ASOF LEFT JOIN aggregated_transactions as aggs
  ON label.cc_num=aggs.cc_num
  AND label.event_ts >= aggs.event_ts
  ASOF LEFT JOIN bank_details as bank
  ON aggs.bank_id=bank.bank_id
  AND label.event_ts >=bank.event_ts
  ASOF LEFT JOIN merchant_details as merchant
  ON label.merc_id=merchant.merc_id
  AND label.event_ts >=merchant.event_ts
  WHERE label.event_ts > '2022-01-01 00:00';
```

The above query returns all the rows in the label feature group where the *event_ts* is greater than the 1st of January 2022, and joins each row with 1 column from *aggrega-*

ted_transactions (*last_week*) and the 2 columns from *bank_details* (*rating* and country), and 1 column from the *merchant_details* table (*chrgbk*). For each row in the final output, the joined rows have the *event_ts* that is closest to, but less than, the value of *event_ts* in the label feature group. It is a LEFT JOIN, not an INNER JOIN, as the INNER JOIN excludes rows from the training data where a foreign key in the label table does not match a row in a feature table. In most cases, it is ok to have missing feature values as you can impute missing feature values in model-dependent transformations.

Feature Vectors for Online Inference with a Feature View

In online inference, the feature store provides APIs for retrieving precomputed features, computing on-demand `features`, and applying model-dependent transformations. Creating feature vectors for online inference involves reading precomputed features, computing on-demand features, and applying model-dependent transformations. In our example from Figure 4-13, there are 2 queries required to retrieve features (1) a primary key lookup for the mechant features using *merchant_id* and (2) a left join to read the aggregation and bank features using *cc_num*. The feature view provides an API to retrieve the precomputed features, as done in Hopsworks:

```
df = feature_view.get_feature_vectors(
entry = [{"cc_num": 1234567811112222, "merchant_id": 212}]
)
```

On-demand transformations and model-dependent transformations also need to be applied to the returned feature data, and we will look more at how feature views support them in Chapter 7.

Conclusions

Feature stores are the data layer for AI systems. We dived deep into the anatomy of a feature store and we looked at when it is appropriate for you to use one. We looked at how to organize your feature data in Feature Groups, and how to organize your data in a data model for batch and real-time AI systems. We also looked at how feature views help prevent skew between training and inference, and how they are used to query feature data for training and inference. In the next chapter we will look at the Hopsworks Feature Store in detail.

CHAPTER 5
Hopsworks Feature Store

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. The GitHub repo can be found at *https://github.com/featurestorebook/mlfs-book*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

In this chapter, we will look in depth at the Hopsworks. Hopsworks is a clustered platform that can be installed on as few as one server or as many as hundreds of servers. Hopsworks includes a feature store, as well as a complete MLOps platform, but we will focus on the feature store in this chapter. We will show how to implement the data model for our credit card fraud model from Chapter 4 in Hopsworks. We will also see how the feature store concepts from the previous chapter are represented in Hopsworks using code snippets in Python. You can create a free account by registering on *Hopsworks Serverless*. The account includes a generous free storage tier - more than enough to build and operate the AI systems from this book. It is also possible to install Hopsworks on a Kubernetes cluster (free for non-commercial use or startups). We will start with projects in Hopsworks - a secure, collaborative space for storing your feature data, training data, and models. If you are only interested in private AI projects and not interested in security, you can safely skip this section.

Hopsworks Projects

A Hopsworks cluster is organized into projects, where each project has a unique name. Hopsworks projects are a secure space for teams to collaborate and manage your data and models for AI. Similar to a repository in Github, a project has team members (with role-based access control), but instead of storing source code, Hopsworks projects store data for AI. Each project has its own feature store, a model registry, model deployments, and datasets for general purpose file storage.

The following code snippet shows how to get a reference to a project object when you login to Hopsworks. If you do not enter the name of the project, Hopsworks will return a reference to your main project (the project you created when you registered your account on hopsworks.ai). With your project, you can get a reference to its feature store as follows:

```
import hopsworks
project = hopsworks.login()
fs = project.get_feature_store()
```

The `hopsworks.login()` method also has parameters for the *hostname* (or IP) and port of the Hopsworks cluster, as well as the API key (either as a value or a file containing the API key). In this book, we will use serverless Hopsworks, which has a hostname of *c.app.hopsworks.ai* and a port of *443*. In this book, we call `hopsworks.login()`without parameters, instead setting HOPSWORKS_API_KEY as an environment variable in your program.

Storing Files in a Project

Every project in Hopsworks has directories where you can store data. From the UI or the Datasets API, you can upload and download files. For example, from the book's github repo, we can upload the titanic CSV file to a directory called *Resources* in your project as follows:

```
dataset = project.get_dataset_api()
uploaded_path = dataset_api.upload("data/titanic.csv", "Resources", over-
write=True)
```

Setting *overwrite=True* makes the upload operation idempotent. You can download a file from Hopsworks using its *path* (right-click on the file in the file explorer UI in Hopsworks to get its path):

```
dataset_api.download(uploaded_path, overwrite=True)
```

If you navigate to *Project Settings -> File Browser*, you will see the directories listed in Table 5-1 in your project.

Table 5-1. The names and a description of the directories in your Hopsworks project, where <proj> is the name of the project.

Directory	Description
Airflow/	Stores Airflow Python programs for this project (DAG files). *Not used in this book.*
DataValidation/	When expectations are attached to a Feature Group, every insertion/deletion creates a validation report that is stored in the subdirectory *<feature_group_name>/<version>* as a JSON file.
<proj> *_featurestore.db/*	The offline feature store directory containing the feature store lakehouse table files.
<proj> *_Training_Datasets/*	When you save training data as files, by default, they are saved here in the *<training_dataset_name>/ <version>* subdirectory (as .parquet, .tfrecords, .csv files).
Jupyter/	Store Jupyter notebooks run on Hopsworks in here. Typically check out git repositories in this directory. *Not used in this book.*
Logs/	For (Python, Spark, Flink) Jobs run in Hopsworks, their output is stored here in a subdirectory: [Spark/Python/Flink] */job_name/execution_id. Not used in this book.*
Models/	Models saved in the Hopsworks model registry are stored in the *<model_name>/<version>* subdirectory, along with its artifacts.
Resources/	A general purpose directory for files used in your project.
Statistics/	Statistics computed for feature groups and training datasets are stored in a subdirectory that follows the naming convention *<name>_<version>*.

Two of the directories in your project store programs (Jupyter notebooks, Airflow Dags). We will not use these directories in this book, however, as we will work with serverless Hopsworks - we will run our programs outside of Hopsworks. If, instead, you have your own Hopsworks cluster, you can use Hopsworks' Git/Bitbucket support to clone the book's source code to the Jupyter directory and run Jupyter notebooks and jobs from within Hopsworks.

Access Control within Projects

Projects support role-based access control (RBAC) inside the project. Each active project member has one of two possible roles: the *data owner* role that has administrator privileges within a project or the *data scientist* role that is a read-only role for the feature store but can create training data and train models. The privileges for the two roles are shown in Table 5-2.

Table 5-2. Privileges of the two roles for operations on Hopsworks services.

	Data Owner	Data Scientist
Project Membership	Add/Remove/Update	
Feature Store	Read/Write/Update	Read
Model Registry	Add/Remove	Add/Remove
Model Deployments	Create/Start/Stop	

	Data Owner	Data Scientist
Project Directories	Read/Write/Delete	Read/Write/Delete all except read only for <proj>_featurestore.db/
Data Sharing	Yes	No

Access Control Across Projects

Projects can also be used to implement access control by placing users and feature groups in different projects and selectively sharing access to feature groups across project boundaries. We will examine these capabilities through an example. In Figure 5-1, we can see how the five feature groups from Chapter 4 are organized inside a single project called *credit_card_transactions*. The project's members are Denzel, the project owner who is responsible for the feature pipelines and model deployment, with Jack and Tay, the data scientists, who train the models.

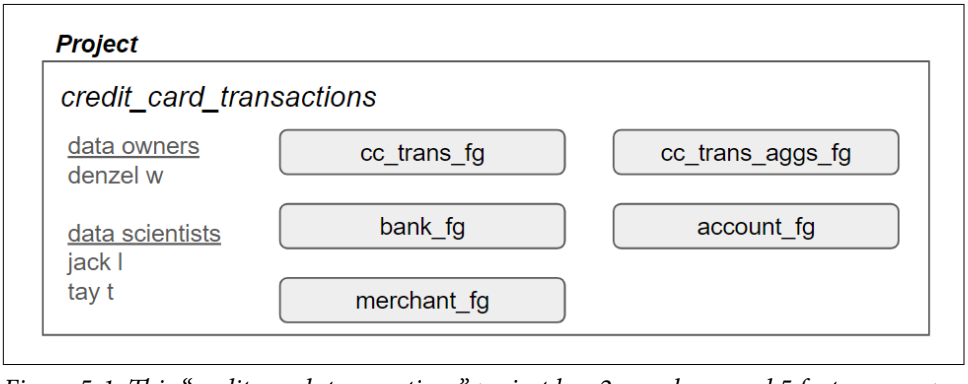


Figure 5-1. This "credit_card_transactions" project has 2 members, and 5 feature groups.

Hopsworks projects are a security boundary; they implement a multi-tenant security model, where each project is the tenant in the Hopsworks cluster. As such, Hopsworks supports project-level multi-tenancy. You can securely store data in a Hopsworks project on a shared cluster, and, by default, users who are not members of your project will not be able to access the resources in your project.

If you have your own Hopsworks cluster, any jobs you run on Hopsworks follow *dynamic RBAC*. Dynamic RBAC is when a user changes its role (and privileges) depending on some change in context. In Hopsworks, the context is the project from which you run the job - your job will have the same privileges as your user within the project the job is run from. If you go to a different project and run a job that that other project, the job will have different privileges - it will have the same privileges as your role in the other project. Hopsworks implements dynamic RBAC by every member of a project having a user identity unique to that project - enabling project-specific access control policies. When you perform an action from within a project, you execute that with your project-specific identity. Your project-specific user in one

project does not have access to other projects you are a member of. Every project you are a member of has a different project-specific user and, by default, access control policies for project-specific users restrict their privileges to only allow access to resources inside the project.

However, what happens if you want to share data from one project to another? Hopsworks supports secure sharing of feature groups and feature views with other projects. This enables us to re-factor our project from Figure 1 into smaller projects that share feature groups with one another, but that have tighter access control on the data. That is, you can implement the principle of least privilege (giving users the minimal set of privileges they need to get the job done, and no more) through a combination of putting sensitive data in their own projects with restricted membership and then sharing that data selectively only those projects that require access.

In Figure 5-2, we reorganized the feature groups from Figure 5-1 to move *account_fg* to a new *know_your_customer* project, and the *bank_fg* and *merchant_fg* to a new *commercial_banking* project.

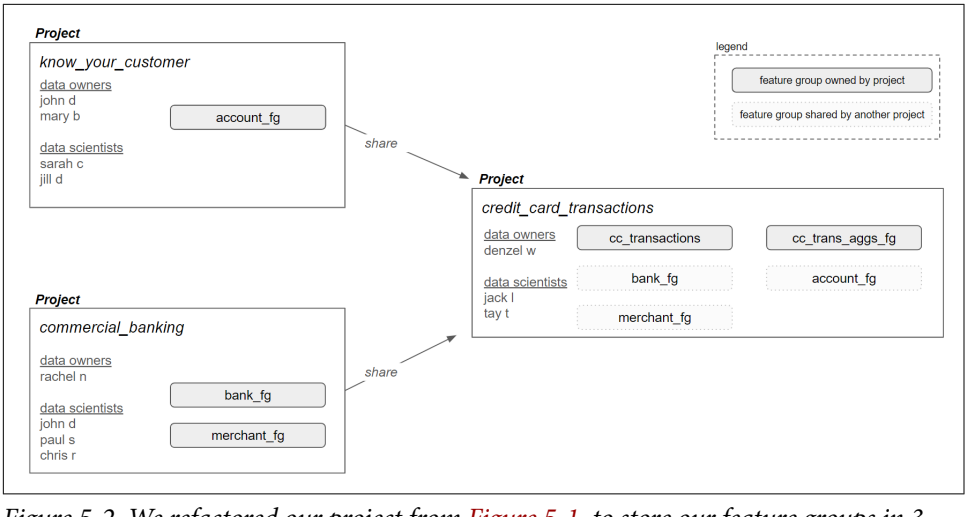


Figure 5-2. We refactored our project from Figure 5-1. to store our feature groups in 3 different projects. The new know_your_customer and commercial_banking projects share their feature groups (read-only) with the credit_card_transactions project. Members Jack, Tay, and Denzel of the credit_card_transactions project can now read feature data from all feature groups, but they can only write to the cc_trans_fg and cc_trans_aggs_fg feature groups.

Then, we share these feature groups read-only with the original *credit_card_transactions* project, whose members now the same read privileges to the data as earlier (when all feature groups were in a single project), but the data owner Denzel has lost write privileges to *account_fg*, *bank_fg*, and *merchant_fg*. This type of data organiza-

tion is often known as a *data mesh*, where instead of a central data team (in one project) managing all data, data ownership is distributed across different business domains (projects).

The best practice for organizing data and users in projects is informed by whether you are doing development, testing in staging, or running in production. For less friction in development, you should give each team/developer their own development project (with all users having the data owner role). For staging and production, you should follow the principle of least privilege - give the minimal read/write/execution privileges to users such that they can accomplish their tasks. One practice that I have often seen is to give read access to production data to developers by sharing read-only access to production data from development projects. Sometimes this is necessitated by huge data volumes, but, in general, this removes the need to metaphorically throw data over the wall to data scientists.

Feature Groups

A feature group in Hopsworks is a table of features, where a feature pipeline updates its feature data, and training/inference pipelines read its data via *feature views*. In Figure 5-3, we can see the backing stores for feature group data in Hopsworks.

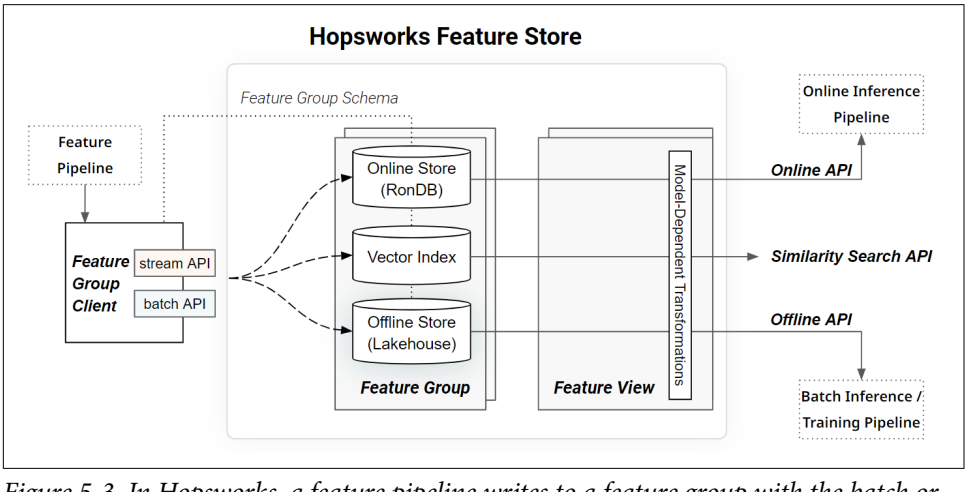


Figure 5-3. In Hopsworks, a feature pipeline writes to a feature group with the batch or stream API. Hopsworks ensures the consistency of feature data across online/offline stores and the vector index. You query/read feature data using a feature view (that may apply model-dependent transformations when reading data). Queries are mapped to one of the backends - the online store, offline store, or vector index.

Hopsworks' online store is *RonDB*, an open-source, distributed, highly-available, real-time database, developed by Hopsworks and forked from the open-source

MySQL NDB Cluster. The offline store is a Lakehouse table (Apache Hudi, Delta Lake, Apache Iceberg), stored either in Hopsworks or in a S3 compatible object store. It is also possible to create an external feature group where the offline store is an external data warehouse, such as Snowflake, BigQuery or Redshift. As such, the offline store can be a mix of external tables and Hopsworks managed lakehouse tables. You can also store vector embeddings in a vector index for a feature group. Clients typically read data from feature groups using feature views. The feature view provides both Offline and Online APIs that query data from the offline and online stores, respectively. There is also a similarity search API for feature groups that store vector embeddings, enabling you to find the *N* closest matching rows for a client provided vector embedding.

To create a feature group in Hopsworks, you first login and get a feature store object for your project, then you can use either *create_feature_group()*, which returns an error if the feature group already exists, or *get_or_create_feature_group()*, an idempotent operation that returns the feature group if it already exists. The following code snippet shows example code for creating an online feature group, with a vector embedding, using a DataFrame:

```
fs = hopsworks.login().get_feature_store()
df = # Read data into (Pandas/Polars/PySpark) DataFrame

# Use the default Embedding Index
emb = embedding.EmbeddingIndex()
# Define the column that contains vector embeddings
emb.add_embedding(df['col_with_embedding'])

expectation_suite = … # Define Data Validation Rules for ingestion

fg_cc_aggs = fs.create_feature_group(
    name="cc_trans_aggs_fg",
    version=1,
    description="Aggregated credit card transaction features",
    primary_key=['cc_num'],
    partition_key=['date'],
    event_time='datetime',
    online_enabled=True,
    time_travel_format='DELTA',
    embedding=emb,
    expectation_suite=expectation_suite,
)
fg_cc_aggs.insert(df)
```

The feature group must have a *name*, a *version*, and a *primary key*. You can provide an optional *description* for the feature group. It is also possible to set descriptions for individual features using the feature group object. The feature group can be either offline-only (*online_enabled=False*), which is default, or online (*online_enabled=True*), in which case tables are created in both the offline and online

stores for the feature group. For the offline tables, you can specify the table format for the offline tables (*time_travel_format='HUDI'* is default). Available table formats are Apache Hudi ('HUDI'), Delta Lake ('DELTA'), and Apache Iceberg ('ICEBERG'). The index columns included in a feature group definition are:

- a mandatory *primary key* defined on one or more columns,
- an optional *event_time* defined on one column (set for time-series data),
- an optional *partition key* defined on one or more columns,
- optional *foreign keys* defined on one or more columns.

The primary key for a feature group uniquely identifies an entity in the feature group. If the feature group has an *event_time* column, then there may be many rows in the feature group for that entity. Each row will have a different *event_time* value and have the feature values at that point in time. In this case, the unique identifier for each row is the combination of the *primary key* and *event_time*. For example, in our *cc_trans_fg* feature group from Chapter 4, there may be many rows with the same *cc_num*, but each row will have a different *event_ts* indicating when the transaction for the credit card with that *cc_num* took place. The primary key can be defined over one column or over two or more columns (*composite primary key*). For example, in the *bank_fg* feature group, we could make the primary key a combination of both the *bank_id* and the *country* column, so that the *bank_id* could refer to HSBC (which is located in many different countries). You can define a column as a *foreign key*, indicating that it refers to a primary key in another feature group. Foreign keys are used when creating a feature view (the selection of features for a model) to join the features from different feature groups together.

A foreign key is a column in a feature group that is used to join features from another feature group. The join column must point to a primary key in a different feature group. In Hopsworks, foreign keys are not statically bound to a specific feature group. Instead, they support late binding. That is, when you create a feature view, you specify the join key from one feature group to another. Hopsworks validates that the join key is a foreign key, and that it points to a primary key in the joined feature group. Foreign keys ensure that training data contains the correct number of rows. In row-oriented databases, foreign keys can also have constraints, such as 'ON DELETE CASCADE', but Hopsworks does not support foreign key constraints.

Hopsworks also supports data layout optimizations for the offline (lakehouse) tables, which can help when your queries become slow because your table stores a lot of data. You can define a *partition_key* on one or more column(s) to partition data in

the offline store (it has no effect on the online store, as RonDB automatically partitions data). The *partition_key* determines the subdirectory (of your feature group directory) in which the data (Parquet) files are written to in the offline store. That is, all rows in your feature group with the same partition key value(s) store their Parquet files in the same subdirectory of the feature group. In the above feature group creation code snippet, the *date* column is set as the partition key, so when you insert a DataFrame, all of its rows with the same *date* value will end up in the same subdirectory (in the feature group's directory). Then, when you query data from that feature group, for example, with *date="2024-11-11"*, only the Parquet files in the "*2024-11-11*" *sub*directory will be read - skipping the data files for all the other subdirectories for all other dates containing feature data. This is known as *Hive-style partitioning* and when a query can skip reading many of the data files, it is known as *data skipping*. *Hive-style partitioning* works well if you have one or columns with relatively low cardinality. If, however, you pick a *partition_key* with high cardinality, you will have a new directory for every unique value of your *partition_key*. So do not, for example, make the partition_key the same as the primary key!

The most common use case for partitioning is where you have a feature pipeline that runs once per hour/day/week and creates GBs/TBs of data, then you create a new *date* column (by extracting the date from your *event_time* column) and make it the *partition_key*. Every time your feature pipeline runs, a new directory will be created storing the data for that datetime in the feature group. Then, when you query the data and set a *filter* on the *date* for a given time period, only the data for the requested time period will be read from the offline store, speeding up queries.

Versioning

Hopsworks supports creating multiple versions of feature groups, where each version contains its own offline/online tables and vector indexes. Hopsworks also supports *data versioning* within a given version of a feature group. That is, every time data is added/updated/deleted to/from a feature group, Hopsworks stores the changes, enabling Git-like operations on feature groups. Data versioning is based on time-travel capabilities found in lakehouse tables.

Data Versioning in Feature Groups and Time-Travel

Hopsworks tracks mutations (appends, updates, deletions) to feature groups as *commits*. When data is either upserted (inserted or updated) or deleted to/from a feature group, each group of changes to the rows in a feature group is called a *commit*. Every commit has a unique ID and a timestamp, see Figure 5-4.

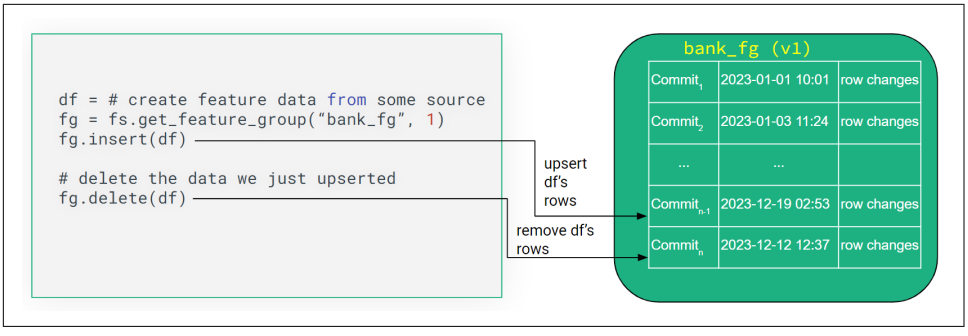


Figure 5-4. Every time you update data in a feature group, a new commit is performed on the feature group. A history of commits is stored on the feature group, enabling you to read changes made by a commit or to read the state of a feature group at a given commit (point-in-time).

A commit contains a set of updates/deletes/appends to rows in the feature group. Each commit has an associated timestamp, and as long as a commit has not been compacted, you can time-travel on a feature group to read its state 'as of' a given timestamp. In Figure 5-4, you can also see how rows in feature groups are removed by providing a DataFrame df containing the primary key values for the rows to be deleted, then calling fg.delete_rows(df).

Feature groups support both *time-travel* and *incremental* queries:

- time-travel to read data in the feature group *ASOF* a provided timestamp or commit-id. The timestamp here does not refer to the *event_time* column in a feature group, but rather the *ingestion time* for the commit.
- Incremental queries read the data changed in commits to a feature group during a specified time range, that is, the row-level *upserts* (inserts or updates).

You can call *read_changes()* to read records upserted within a specified time range as a DataFrame. The time range is specified with a starting timestamp and an optional ending timestamp. If no ending timestamp is set, the range returned will include all records since the starting timestamp. You can provide the ingestion time as a parameter to the *read()* method to read the state of the feature group *AS OF* that point in time, see Figure 5-5.

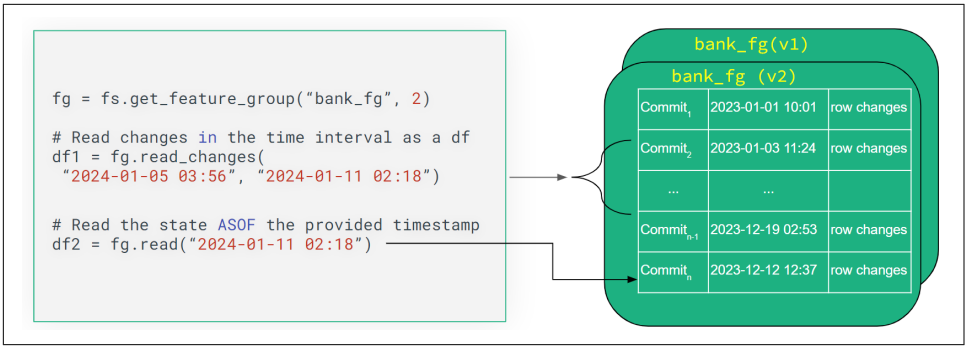


Figure 5-5. For version 2 of the bank_fg feature group, we read the changes in the provided date interval as df1 containing the rows updated/appended in the time range provided. We then read the state of the feature group into df2 as of the provided timestamp. If you omit the timestamp, read() returns the latest data.

Note that the ingestion time refers to the physical (actual) time at which that commit was ingested into Hopsworks. The ingestion time can be confusing, because your feature group may also have an *event_time* column indicating the value of a feature as of a point in time. Ingestion time and event time are different concepts. For example, imagine in our air quality project from Chapter 3 where a sensor was offline from days 4-9, as shown in Figure 5-6.

Figure 5-6. In this diagram, we see air quality measurements from days 4-9 arrive late on day ten. They arrived just after Training Dataset v1 was created. If we want to reproduce Training Dataset v1 at a later point in time, we should not include the late arriving data in it.

The weather updates came for every day, but on day ten, we received the missing six days of air quality measurements. They arrived late. The *event_time* values for these six late arrivals are the days 4-9, respectively, which makes sense as the *event_time* refers to the day the air quality measurement was taken. However, the ingestion time for the late arrivals is day ten - the event time doesn't match ingestion time. In real-world systems, late arriving data is a fact of life, and systems need to be designed to account for it.

If you read the feature group on day nine, it will not include any of the air quality measurements from days 4-9, but if you read it on day ten it will include days 4-0. The Training Dataset v1 was created on day nine, however, and it does not include days 4-9. If I later delete Training Dataset v1, but have to reproduce it, I would like it to be exactly the same as the original (compliance will demand this). I do not want it to include the air quality data for days 4-9. However, if I only used a query based on the *event time* to reproduce the training dataset, it would include the data from days 4-9. The solution is to use ingestion time to recreate Training Dataset v1 exactly as it was created on day nine. Luckily, Hopsworks does this transparently for you when you call any of its feature view methods to re-create training data using its version number, such as:

```
X, y = feature_view.get_train_test_split(training_dataset_version=1)
```

We have now seen the *ASOF* term twice now in different contexts. When you recreate a training dataset, you want to include the feature data *ASOF* its ingestion time (the feature data that existed at that time). But when you create point-in-time correct training data, you want the value of the features *ASOF* the event time, as you want to include the correct value for that feature at that point in time.

Updating and Versioning Feature Groups

Data versioning is only concerned with changes to the rows in feature groups. What if you want to add, remove, or update the features in a feature group? You can add a new feature to a feature group as follows, and existing clients of the feature group will work as before:

```
features = [
    Feature(name="limit", type="int", default_value=1000)
]
fg = fs.get_feature_group(name="cc_trans_fg", version=1)
fg.append_features(features)
```

However, if you want to change the data type for a feature or delete a feature from a feature group, then you are making a *breaking schema change*. Existing clients of the feature group will not work because one or more of the features they expect will

either have the wrong data type or not exist. Another less obvious breaking change is if you change how a feature is computed. You shouldn't mix the old feature values and new feature values in the same feature in a feature group. This will not break clients, but any models you train on the mixed feature data will not perform well.

The solution to breaking (schema) changes is to create a new version of the feature group with new feature(s). For example, in Figure 5-7, the *cc_fraud_v1* model is upgraded to *cc_fraud_v2* which uses a new version *v2* of the *account* feature group. When a model depends on a feature group for precomputed features, the model and feature versions are tightly coupled requiring synchronized upgrades and downgrades of model/feature versions.

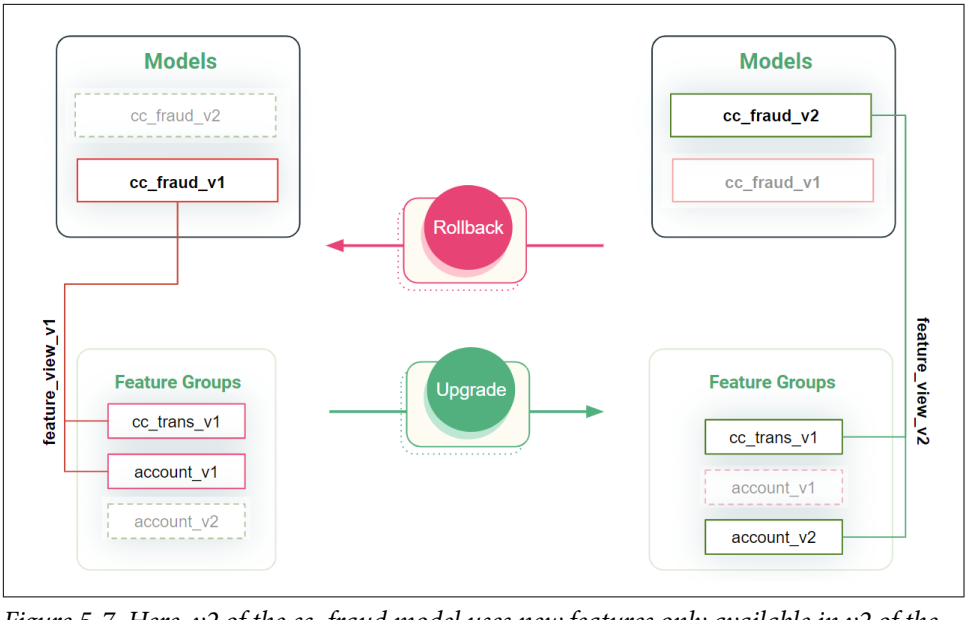


Figure 5-7. Here, v2 of the cc_fraud model uses new features only available in v2 of the account feature group. In order to be able to downgrade (in case of error), you need to maintain the older v1 of the account feature group.

When you create a new feature group version, new offline/online tables will be created, so you may need to backfill the new feature group version with data from the old feature group version. The backing table name in the offline/online stores is *<feature_group_name>__<version>*.

When a feature group has a large amount of data, you may want to avoid creating a new version of a feature group. Sometimes, you can just keep appending new features, leaving the old feature versions in the feature group (but maybe not used by the latest model version). That can also be expensive as appending a new feature requires updating all existing rows in the table with a *default_value*. For example, assume you

have a feature group with hundreds of columns that stores 10s of TBs of data, but you only want to change how one column is computed. You don't want to create a new version of the feature group and backfill the whole feature group. You don't either want to append a new feature as that will require updating all rows in the feature group with the new column and its default value - in lakehouse tables that will probably require rewriting all of the data files. Instead, you can create a new feature group with a different name, but with the same primary key as the original feature group, see Figure 5-8. You will need to backfill the new column for this feature group, but it will be a much less expensive operation than backfilling hundreds of columns.

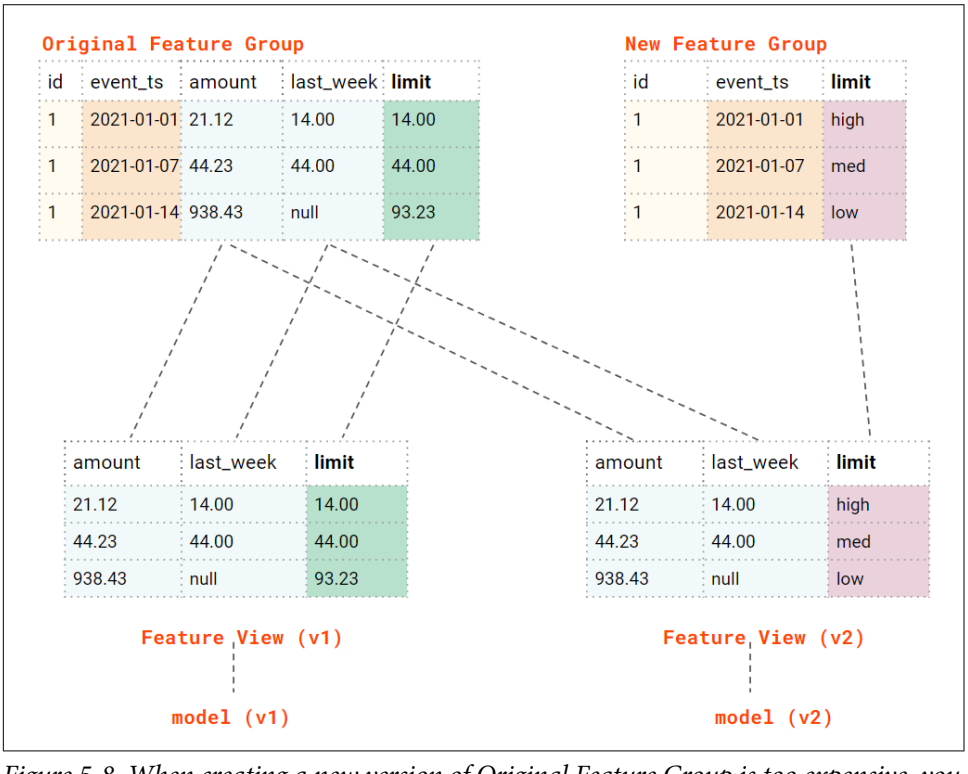


Figure 5-8. When creating a new version of Original Feature Group is too expensive, you can create New Feature Group that stores a new categorical version of the limit feature. A model that uses Feature View (v1) can be retrained with the new feature by updating its feature view to v2, replacing the old limit feature with the new one, but keeping the other features unchanged.

The new feature, from Figure 5-8, is a categorical *limit* feature in our new feature group that we will compute from the sparse *limit* feature. You need to write a transformation function that converts the numerical *limit* value to categorical value (*high*, *med*, or *low*). That transformation function can be used to backfill the new feature

group with all of the values from the original feature group, and it should also be included in a feature pipeline that will update the new feature group.

Now, assume we have a model *(v1)* that we want to update to (*v2*) to use the new categorical *limit* feature instead of the numerical *limit*. What we can do is create a new *feature view (v2)* that replaces the old numerical *limit* with the new categorical *limit*, but keeps all the other features from feature view (*v1*). Creating feature views is a metadata only operation, so it is cheap. The new feature view can now create new training data and train model (*v2*).

Assume, now that you have a model that uses the old feature and you want a new version of the model that instead uses the new version of the feature. For the new model, you create a new feature view that uses all the features from the feature view of the previous model, replacing the old feature with the new one. When you read training/inference data from the new feature view, it will join the original features (not including the feature you are replacing) with the new version of your feature. An example notebook for this pattern is available in the book's github repository.

Online Store

When you create a feature group, you have to decide whether the feature data will be stored in the online store or not. By default, a table is not created in the online store. To enable the online store, you have to specify *online_enabled=True* when you create the feature group. In contrast, a table is always needed in the offline store. You should make a feature group *online_enabled* if the feature data it stores will be read by interactive or real-time AI systems. If the feature data will only be used by batch AI systems, then do not make it *online_enabled*, as it will add cost in data storage and slow down writes. If you want an online-only feature group, with no data in the offline store, then you specify that writes should not be materialized to the offline store:

```
fg.insert(    df,
write_options={"start_offline_materialization":False}
)
```

Hopsworks' online stores feature data either in-memory or in on-disk columns. By default, it uses in-memory tables, which have lower latency and higher throughput compared to on-disk columns. However, in-memory tables require enough RAM to store the data, and when you have feature groups that will store many TBs of online data, it may be more cost efficient to use on-disk tables. You can specify that the online feature data will be stored on-disk when you create the feature group as follows (the *table_space ts_1* is the default parameter for the on-disk table in RonDB):

```
fs.create_feature_group( …
    online_enabled=True,
    online_config={'table_space': 'ts_1'}
)
```

Hopsworks' online store is RonDB, an open-source, distributed, real-time database that has both key-value and SQL APIs. It can be configured to be highly available either within a data center (with replication based on a non-blocking variant of the two-phase commit protocol) or across geographically separated data centers (using asynchronous replication). RonDB can scale to store in-memory tables with tens of TBs or store the feature columns as on-disk columns. The primary key and indexes are stored in-memory. RonDB has been designed to support feature store workloads, with support for projection pushdown, predicate pushdown, pushdown aggregations, composite primary keys, and pushdown left joins. For further reading on the performance impact of these capabilities, I recommend our research paper at SIGMOD 2024.

Time-to-Live

By default, the *event_time* column is not included in the online table, and the online table only stores the latest feature values for each entity. When you write new feature data for an entity, the row containing the feature data for that entity is overwritten. This bounds the size of your online table to the number of entities in your table.

However, what if you have hundreds of millions of entities and the feature data becomes stale for an entity after a period of time? Or what if you want to perform online aggregations for an entity, then you will need to include the *event_time* column in the online table to be able to store many rows for each entity? In both of these cases, you should specify a *time-to-live (TTL)* value for rows, whereby rows are removed from the database when they exceed the specified TTL defined on the feature group. For example, if the TTL is one hour, then one hour after the *event_time* for a row has passed, the row will be scheduled for deletion. You can define the TTL, at minute level granularity, when you create an *online_enabled* feature group:

```
fs.create_feature_group( …
    TTL_mins=120,
    online_aggregation=True
)
```

If you set *online_aggregation=True*, then the online store stores many rows for the same entity (with different event_time values). TTL expiration is a background process, and expired rows are typically deleted within 15 minutes of expiration, although under situations with high database load, it may take a bit longer.

It is important to note that Hopsworks also handles potential data leakage caused by the TTL. When you are creating training data, what should happen if the *label.event_time* is 01:00 and a *feature.event_time* for that label is 00:15, but the TTL is 30 minutes? You shouldn't include that feature value, otherwise there will be leakage. The reason why is that the online store would have removed the feature's row at 00:45, when its TTL expired. When the label event arrives at 01:00, there would be no

feature value to retrieve. This is a subtle, yet pernicious, form of data leakage that Hopsworks prevents. The general rule here is that when you create training data using features from a feature group with a TTL, the feature value will be null if the following holds:

```
label.event_time - feature.event_time > TTL
```

Vector Embeddings

Vector embeddings enable approximate nearest neighbor (ANN) search (also known as similarity search) for rows in *online_enabled* feature groups. You create a vector embedding by taking high dimensional data (such as text or images or a mix of data), passing it to an *embedding model* that then compresses the input data into a fixed size array of floating point numbers. The vector embedding is the output array of floating point numbers and what is astonishing about it is that, even after compression, it retains semantic information about the original input data. You can take millions of images or books of text (split into paragraphs), compute vector embeddings from them, and then pass in a new image or piece of text, and ANN search will find the closest images or paragraphs of text to the new data. And they work really well, even though it is a probabilistic matching.

To add vector embeddings to a feature group, you specify which columns in your DataFrame that contain the vector embeddings. The column values are then inserted into a vector index so that you can call *find_neighbors()* on the feature group to find rows with similar values. However, before inserting rows into an *embedding feature group*, you need to first compute the vector embeddings for the columns using an *embedding model*. There are many off-the-shelf embedding models that can be used, such as the *sentence transformers* model in the example below. You can also train your own embedding model.

When you design a data model that includes an embedding feature group, you should know that writing rows to a vector index is significantly slower than writing to the online feature store. The online feature store supports millions of concurrent writes per second, while the vector index is orders of magnitude slower. If you have non vector embedding columns in an embedding feature group that are updated more frequently than the vector embedding column, you should probably refactor your feature group to move the frequently updated columns to a separate feature group.

We will now look at our example credit card transaction fraud system and how we add support for vector embeddings. Suppose you are doing some EDA on fraudulent transactions and would like to find the most similar rows to a row marked as fraud. That's hard, as there may be tens of thousands of rows of fraudulent transactions or more. The *cc_fraud* table (in Postgres) that contains the fraud labels also has a string

column called *explanation*. The column contains a human written description of the reason the transaction was marked as fraudulent. We can add the *cc_fraud* table as a new feature group to enable similarity search for fraudulent transactions using the *explanation*. You can run the following code to create a vector embedding using an open-source sentence-transformers (embedding) model, that maps the *explanation* to a 384 dimensional array:

```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('all-MiniLM-L6-v2')

df = pd.read_csv("https://repo.hops.works/dev/jdowling/cc_fraud.csv")
embedding_body = model.encode(df['explanation'])

df['embed_explanation'] = pd.Series(embedding_body.tolist())
emb = embedding.EmbeddingIndex()
emb.add_embedding('name', model.get_sentence_embedding_dimension())

fg_fraud = fs.create_feature_group(
    name="cc_fraud_fg",
    version=1,
    description="Credit Card Fraud Data",
    primary_key=['tid'],
    event_time='datetime',
    embedding=emb
)
fg_fraud.insert(df)
```

You now perform similarity search on the new feature group, passing a vector embedding to the feature group's *find_neighbor()* method:

```
model = SentenceTransformer('all-MiniLM-L6-v2')
search_query = "Geographic attack in South Carolina"
fg_fraud.find_neighbors(model.encode(search_query), k=3)
res = fg_fraud.find_neighbors(model.encode(search_query), k=3)
```

The above code will return the 3 rows in the feature group that had an explanation column value that is most similar to the search string "Geographic attack in South Carolina".

Offline Store (Lakehouse Tables)

Hopsworks' offline store is lakehouse tables. Hopsworks supports three different types of lakehouse table, each with their own strengths: Apache Iceberg, Apache Hudi and Delta Lake. All three formats support time-travel, but there are other properties leveraged by Hopsworks:

primary key uniqueness
: Enforced by Hudi, but not by Iceberg or Delta;

data skipping
: Hive-style partitioning is supported across all three file formats, but additionally there is Z-Ordering (Hudi, Delta), Liquid Clustering (Delta), and Hilbert Space Fitting Curves (Hudi);

read_changes
: File formats support *CDC queries*, although full support will only come in Iceberg v3.

Delta and Iceberg do not enforce the uniqueness constraint for primary keys, and this means you have duplicate rows when you create training data. The ASOF LEFT JOIN (used to create training data from Chapter 4) joins features to labels and if there are multiple matching rows in a joined feature group, you will get multiple output rows for each row in your label feature group. That is not desired behavior, as a feature should have only one value for a given label. For this reason, Hopsworks uses Apache Hudi as the default offline store.

External Feature Groups

If you already have existing tables with feature data in a data warehouse or object store, you can create an *External Feature Group* from that table. In External Feature groups, the offline table is the external data warehouse (such as Snowflake, BigQuery, Redshift, or any JDBC-compatible database). No offline data will be stored in Hopsworks, only metadata.

An external feature group first needs a *storage connector* for your external store. Hopsworks supports storage connectors to JDBC data sources, GCS, S3, Snowflake, BigQuery, and other data sources. External feature groups also include a *query* parameter. This is a SQL statement, used to read the desired columns from the external table. The SQL query can also be used to perform any SQL operation supported by the external store (aggregations, filtering, etc). This means that feature data from external feature groups is computed when it is read. Here, we show you how to define *account_fg* as an external feature group:

```
query = """
SELECT account_id, cc_expiry_date, event_time FROM account_details
"""
external_fg = fs.create_external_feature_group(
            name="sales",
            version=1,
            description="Physical shop sales features",
            primary_key=['account_id'],
            event_time='event_time',
            online_enabled=True,
            query=query,
            storage_connector=sc
            )
```

```
external_fg.save()
```

If your external feature group is *online_enabled*, you need to explicitly synchronize the data from the offline store to the online store.

Data Statistics

When you write data to the offline feature group, by default, Hopsworks computes and saves descriptive statistics for features. Statistics are used for both EDA as well as to monitor for feature drift (see Chapter 13). Hopsworks can compute *histograms* for categorical variables (counts for each of the categories), a *correlation matrix* for the features (to help identify redundant features that can be removed), *descriptive statistics* for numerical features (min, max, mean, standard deviation), the sparsity of a feature through *exact_uniqueness* (values closer to 1 indicate more unique values). You provide the list of features that you want to compute features for in the *columns* parameter of the *statistics_config* dictionary.

```
fg_cc = feature_store.create_feature_group(name="cc_trans_fg",
    statistics_config={
        "enabled": True,
        "histograms": True,
        "correlations": True,
        "exact_uniqueness": False,
        "columns": ["feature1"]
    }
)
fg_cc.compute_statistics()
```

Note that computing statistics is expensive, particularly if they are computed on large volumes of data.

Change Data Capture (CDC) for Feature Groups

Sometimes, it is useful to build event-driven feature pipelines by executing actions when rows in a feature group have changed. One example use case is when you have a large number of entities and you want to make predictions for entities after changes in their feature values. You can do this by enabling change data capture (CDC) API for a feature group by providing a Kafka topic for the feature group:

```
kafka_api = project.get_kafka_api()
my_schema = kafka_api.create_schema(SCHEMA_NAME, schema)
my_topic = kafka_api.create_topic(TOPIC_NAME, SCHEMA_NAME, 1, replicas=3, parti-
tions=8)

fg_cc_ags = feature_store.create_feature_group(name="cc_trans_fg",
    notification_topic_name=TOPIC_NAME,
)
```

Rows that are updated in the *cc_trans_fg* feature group are published to the Kafka topic (*TOPIC_NAME*) and consumers of the changes can subscribe to the Kafka topic to consume the rows that were updated.

Feature Views

As introduced in Chapter 4, feature views bridge the gap between feature groups and models by storing the list of input and output features for a model. The main steps in creating and using feature views are:

- selecting the features and labels/targets that will be used by your model
- defining any model-dependent transformations you want to perform on your features
- creating the feature view from your feature selection and model-dependent transformations
- creating training data for your model with your feature view
- creating batch inference data for your model with your feature view
- creating online inference data for your model with your feature view.

We will work with the credit card fraud example and use the feature view to create training and inference data for our model.

Feature Selection

When you want to create a model, you will need to select columns from feature groups for your model. Most of these selected columns will be inputs (features) to your model, but, for supervised learning, one or more columns will be the output labels or targets for your model. For all feature groups that contain columns you want to include in your model, you need to be able to join those feature groups either directly (think star schema from Chapter 4) or transitively (snowflake schema) with the label feature group.

To start with, you should identify the label feature group for your feature view. Each feature view has at most one label feature group containing the labels. To join features with your label feature group, your label feature group needs to have a foreign key to the feature group that contains those features. In Chapter 10, we will look at how to add foreign keys to label feature groups, but for now we will assume those foreign keys exist. Any feature group joined to the label feature group can, in turn, have foreign key(s) to other feature groups that can also be included in the feature selection. You can also create a feature view without labels, for unsupervised learning, in which case the label feature group is just the *leftmost feature group* in a feature selection statement.

In our credit card fraud data model, *cc_num* in *cc_trans_fg* is a foreign key to *cc_trans_aggs_fg*. Similarly, *merchant_id* in *cc_trans_fg* is a foreign key *to merchant_fg*. We can also transitively include features from *bank_fg* and *account_fg*, as their primary keys are foreign keys in *cc_trans_aggs_fg*. We start by getting references to those feature groups.

```
labels = fs.get_feature_group("cc_trans_fg", version=1)
aggs = fs.get_feature_group("cc_trans_aggs_fg", version=1)
merchant = fs.get_feature_group("merchant", version=1)
bank = fs.get_feature_group("bank_fg", version=1)
account = fs.get_feature_group("account_fg", version=1)
```

You specify which features to join by calling one of the *select* methods on a feature group:

- *select_features()* selects all the feature columns (not index columns and foreign keys)
- *select_all()* selects all the columns (includes index columns and foreign keys)
- *select_except(['f1', 'f2', …])* selects all the columns except those in the provided list
- *select(['f1', 'f2', …])* selects only those columns in the provided list

The *select* methods return a *Query* object that represents the selection of features. You can read feature data with a *Query* object, add a filter to read a subset of feature data, inspect the temporal query string used to create the feature data, and most importantly, you can call *join*() on it to join with other *Query* objects (that represent features selected from other feature groups). Here are the *select* and *join* methods that are used to create the selection of features (and the label) used in our credit card fraud model:

```
aggs_subtree = aggs.select_features()
.join(bank.select_features())
.join(account.select_features()

selection = labels.select_features()
.join(merchant.select_features())
.join(aggs_subtree)
```

In the above code, we do not specify any *join key* explicitly. Hopsworks' looks for the column(s) in the left-hand feature group that has the same name and type as the primary key in the right-hand (joined) feature group. If there is no match, you have to explicitly define the *join key*. For example, if the primary key of *account_fg* were *id* (instead of *account_id*) we would have to construct the join as follows:

```
aggs.select_features().join(bank.select_features(),
left_on=["account_id"],right_on=["id"])
```

If there is a clash between feature names from the left and right feature groups (both feature groups have a feature with the same name), in the *join* method, you can use the parameter *prefix="abc_"* to add a prefix to the feature names from the right hand feature group.

Model-Dependent Transformations

In Hopsworks, you can declaratively attach a transformation function to any of the selected features in your feature view. The transformation functions are executed in the client after data has been read from the feature store with a feature view. As feature views are only used in training and inference pipelines, these transformation functions are model-dependent transformations. You can use either built-in transformations (such as *min_max_scaler*) or define your own custom transformation function, such as here:

```
from hopsworks import udf

@udf(float)
def f1(amount, days_until_expired, stats: TransformationStatistics):
    return (amount * days_until_expired) / stats["amount"].median
```

In the above example, we can see that the transformation function is parameterized by the *TransformationStatistics* object that contains statistics that were computed over features in the training dataset. Many transformation functions are parameterized by statistics computed on the training dataset, such as those that encode categorical features or scale numerical features.

Transformation functions can be defined either as a Python User-Defined Functions (UDFs) or a Pandas UDF. Pandas UDFs scale to process large data volumes, for example, in PySpark training dataset pipelines, but they add a small amount of latency in online inference pipelines. Python UDFs, in contrast, scale poorly when data volumes increase, but have lower latency in online inference pipelines.

Creating Feature Views

Once you have selected your features and defined your model-dependent transformations, you can create a feature view as follows:

```
feature_view = fs.create_feature_view(
    name='cc_fraud',
    query=selection,
    labels=["is_fraud"],
    transformation_functions = [ MinMaxScaler("amount") ],
    inference_helper_columns=[‘cc_expiry_date’,‘prev_loc_transaction’,
‘prev_ts_transaction’]
)
```

Typically a feature view is created for one model or a family of related models. For example, if you have models for customers in different geographic regions, you could use the same feature view to represent the models for all of your customers, and then apply filters when creating training data or batch inference data to only return the data for the model's geographic region:

```
feature_view.filter(Feature("region")=="Europe").training_data()
```

When you use one or more filters to create training data, the filter(s) is stored as metadata in Hopsworks, so that you can later reproduce the training data using only metadata.

A feature view does not have a primary key, instead it has *serving keys*. When you use a feature view to retrieve one or more rows of features (*feature vectors*) via the Online API, you have to provide values for the serving keys. The serving keys are the foreign keys in the label feature group for the feature view. In our credit card fraud example, the serving keys from *cc_trans_fg* are (*cc_num*, *merchant_id*), as both of these foreign keys were used to create our feature view. You can inspect a feature view's serving keys as follows:

```
print(feature_view.serving_keys)
```

Other parameters that can be provided when creating a feature view are *training_helper_columns* and *inference_helper_columns*. Sometimes during training or inference, you need helper columns that will not be used as features. For example, helper columns could be used as inputs to transformation functions, but will not themselves be features. In our credit card fraud system, we define three columns as *inference_helper_columns*, as they are all used as parameters in transformation functions used to compute on-demand features: *haversine_distance*, *time_since_last_trans*, and *days_to_card_expiry*. When you read online inference data with the feature view, you will receive these columns and then use them to compute the on-demand features (they are parameters to the transformation functions). However, you will not include them as input parameters when calling *model.predict()*. When you use the same feature view to read training data, *fv.training_data()*, it will not return the *inference_helper_columns*, as they are only needed during inference (there are no on-demand transformation functions in training pipelines). Similarly, *training_helper_columns* are returned when you create training data, but not returned when you read (batch or online) inference data.

Training Data as either DataFrames or Files

With your feature view, you can read training data as Pandas DataFrames or create training data as files, see Table 5-3.

Table 5-3. You can read training data as either Pandas DataFrames or create training data as files. Deep learning models often create large training datasets as files, while decision tree models have smaller training datasets that can often be read directly as DataFrames.

	When to Use	Most Common ML Frameworks
Pandas DataFrames (Training Pipeline)	Tabular data < 10 GBs	Scikit-Learn, XGBoost, Catboost, Prophet
Files: .tfrecord, .parquet, .csv (Training Dataset Pipeline)	Tabular data > 1 GB or Tensor datasets	PyTorch, TensorFlow, Jax

You often split training data into a training set and a test set. The feature view provides convenience methods that read and split the training data in a single method call.

Typically, you read data from files, but query a feature store. Querying is more powerful than just reading training data as files, as you can filter data based on your needs. For example, you might need a time window of data or user data from a geographic region and querying will only read the data requested.

Random, Time-Series, and Stratified Splits

You can read your training data, split using a *random split* into training and test sets of features (*X_*) and labels (*y_*), as follows:

```
X_train, X_test, y_train, y_test = fv.train_test_split(test_size=0.2)
```

The above example gives you 80% of the data in the training set (*X_train*, *y_train*) and 20% in the test set (*X_test*, *y_test*). Sometimes, you also need a validation set in addition to the training and test sets. For example, if you want to perform hyperparameter tuning, you should not evaluate model performance using the test set (otherwise the test set can leak into model training). Instead, you can create an additional validation set, on which you evaluate training runs with different hyperparameters:

```
X_train, X_validation, X_test, y_train, y_validation, y_test = fv.train_valida-
tion_test_split(validation_size=0.15, test_size=0.15)
```

In this case, the test set is the holdout set used to evaluate final model performance, after hyperparameter tuning is finished.

The same *train_test_split* and *train_validation_test_split* functions can also return a time-series split of your training data. As a rule, you should never create a random split of time-series data - as temporal patterns and trends get lost in randomization. Instead, specify a time-range for each of your training, validation, and test sets. In the sample code below, the training set time window is from the 1st-31st January 2024, and the test data is the data that arrived after the 1st-7th of February 2024:

```
X_train, X_test, y_train, y_test =
  fv.train_test_split(start_train_time="20240101", end_train_time="20240131",
start_test_time="20240201", end_test_time="20240207")
```

If you omit the *start_test_time*, the test set will start after *end_train_time*. Also, if you omit *end_test_time*, the test set will include all data that arrived after 1st February 2024.

Sometimes, you need a more sophisticated way to split your training data than a random or time-series split. For example, when predicting credit card fraud, you can train a binary classifier, but the positive class (fraud) is massively underrepresented compared to the negative class (no-fraud). The imbalance ratio could be 1000s to 1 or higher. There is a high risk when you split your data into training and test sets that the ratio of positive and negative classes will not be the same, which would result in poor evaluation of model performance as the distribution of labels would not be the same in training and test sets.

In this case, and, in general, if you have an imbalanced dataset, you should use a stratified split. For this, you should read your training data as a single DataFrame, and then implement the stratified split yourself, using an appropriate library, such as scikit-learn, if needed:

```
training_data = fv.training_data()
# apply custom splits into training and test/validation sets
```

Supervised learning does not work well when the class distribution is skewed. For binary classifiers, you should upsample or downsample one of the classes to improve balance between the classes. In Python, the *imbalance* library is widely used for up/down sampling. If imbalance is too high, you may need to consider an alternative technique, such as anomaly detection with unsupervised learning instead of a binary classifier.

Reproducible Training Data

When you read training data as DataFrames or create training data as files, Hopsworks stores metadata about the training data created, including the feature view used, any filters used when creating training data, the training dataset ID, any random number seed, and the commit-ids for the feature groups that the training data was read from. This way, you can delete the training data, and Hopsworks can still reproduce that training data exactly using only the training dataset id:

```
X_train, X_test, y_train, y_test =   fv.get_train_test_split(train-
ing_data_id=111)
```

Sometimes, you will need to delete training datasets due to storage costs or for compliance reasons (data retention policies). In these cases, the ability to accurately recreate training data is important.

Data science has aspired to be more science than engineering, with an emphasis on reproducibility and replicability as they are cornerstones of the scientific method. This has led to the growth in popularity of experiment tracking platforms that store hyperparameters from training runs, enabling models to be reproduced using experiment tracking metadata. Reproducible training data has received comparatively less attention, but is now possible with feature stores, and should grow in importance with the coming regulation of AI.

Batch Inference Data

You can read batches of inference data from the offline store with a feature view. A popular use case in batch inference pipelines is to read all new data that has arrived since the last time the batch inference pipeline ran:

```
last_run_timestamp = "2024-05-10 00:01"
fv = fs.get_feature_view(...)
fv.init_batch_scoring(training_data_id=1)
df = fv.get_batch_data(start_timestamp=last_run_timestamp)
df["prediction"] = model.predict(df)
```

Here we call *init_batch_inference* on the feature view to tell it which training dataset ID to use if it has to compute model-dependent transformations. Then we read a Pandas DataFrame, *df*, containing the transformed input features, read from the feature store. Finally, we make our predictions with the *model* on *df* (assuming the model can take a Pandas DataFrame as its input, which is possible for XGBoost and Scikit-Learn models). You can also log predictions and the feature values using *fv.log(df)*.

Sometimes, you need more flexibility when reading batch inference data. For example, imagine if you want to read the latest feature data for all entities with your feature view (such as the latest transactions and fraud features for all credit cards). For this, you can use a spine group. A spine group contains rows of serving keys, for reading your features for your feature view, along with a timestamp value for every serving key. It is called a spine as it is the structure around which the training data or batch inference data is built. Spine groups are not used in online inference and a spine group has to be the label feature group in a feature view. You can define a spine group as follows:

```
trans_spine = fs.get_or_create_spine_group(
    name="cc_trans_spine_fg",
    …
    dataframe=trans_df
)
```

Notice that you have to include a DataFrame, *trans_df*, to provide the schema for the feature group. A spine group does not materialize any data to the feature store itself,

and its data always needs to be provided when retrieving features for training or batch inference. You can think of it as a temporary feature group, to be replaced by a DataFrame when data is read from it. When you want to create training data with a feature view that contains a spine group as its label feature group, you can do so as follows:

```
df = # (serving keys, timestamp for label values)
X_train, X_test, y_train, y_test =
feature_view.train_test_split(0.2, spine=df)
```

Similarly for batch inference, you can read inference data as follows:

```
input_df = # (serving keys, timestamp for feature values)
output_df = feature_view.get_batch_data(spine=input_df)
predictions = model.predict(output_df)
```

If you can avoid spine groups, you should, as they add complexity and externalize much of the work for building training datasets and batch inference data to clients.

Online Inference

Feature views are also used to retrieve rows of features from the online store at low latency. In our fraud example, the *get_feature_vector()* method call retrieves a row of precomputed features for a given credit card number (serving key):

```
feature_vector = feature_view.get_feature_vector(entry={"cc_num":
"1234", "merchant_id": 4321}, return_type = "pandas")
```

The result, the feature vector, is returned as a Pandas DataFrame, but you can also read a *numpy array* or list type (default). There is also a version of this method call that retrieves many rows called *get_feature_vectors*, where the *entry* parameter is a list of serving keys.

The transformation functions, introduced earlier, can also be used to define on-demand transformation functions. For example, the on-demand

from earlier feature *cards_to_card_expiry* can be computed as follows:

```
@udf(float)
def days_to_card_expiry(expiry_date):
    return datetime.today().date - expiry_date
```

You call this transformation function in an online inference pipeline as follows:

```
feature_vector = feature_view.get_feature_vector(entry={"cc_num":
"1234", "merchant_id": 4321}, return_type = "pandas")
cc_expiry = days_to_card_expiry(feature_vector["expiry_date"])
feature_vector = feature_vector.drop(columns=["expiry_date"])
prediction = model.predict(feature_vector)
```

Note that the *expiry_date* parameter is retrieved using the feature view as an inference helper column. You need to drop any inference helper columns before you call *model.predict()*.

In Chapter 11, we will bring all online inference steps together, including also model-dependent transformations, logging the prediction/feature values, and monitoring the features/models.

Faster Queries for Feature Data

We finish this chapter by looking at how to read feature data using filters. Applying filters can lead to huge performance improvements when reading a subset of feature data. For example, in the offline store, when data volumes are large, reading large amounts of data into (Pandas or PySpark) DataFrames and then dropping the columns and rows you do not need incurs huge overhead. It is going to be either very slow or may not work due to out-of-memory errors. The two main techniques for reducing the amount of data read in a query are:

projection pushdown
: Read only the columns you request

pushdown filters
: Read only the data for the filter value(s) you provide. This includes both *partition pruning* and *predicate pushdown.*

When you read a subset of the features in a feature group and only the data for those features is returned to the client, it is known as *projection pushdown*. Hopsworks supports projection pushdown out of the box - you don't need to do anything to get the benefits. When you define a feature view that only uses a subset of the features in a feature group, reads using that feature view will read with projection pushdown. Both Hopsworks' online feature store, RonDB, and its offline store (lakehouse tables) support projection pushdown. Online stores without projection pushdown, for example Redis, require the client to read all of the columns in feature groups and only in the client will it filter out the data it doesn't need. Projection pushdown is particularly needed in cases such as when you have a wide feature group with many columns, and a subset of those columns are used in many different models. When you query data from a feature view (e.g., read training data or batch inference data), you can provide a filter such as

```
X_features, y_labels =
     fv.filter(Feature("event_time")=="2024-01-10").training_data()
```

You can also read data directly from feature groups using filters:

```
df = fg.filter(Feature("event_time") > "2024-01-10").read()
```

In the case, where you have a feature view that contains features from multiple feature groups, you can chain filters that can all potentially be pushed down to the backing feature groups. For example, assume we have a feature view that contains features from two feature groups. The first feature group is partitioned by the *event_time* column and the second one is partitioned by the *country* column. In this case, we chain filter function calls. In the following example, we show an alternative way to identify a feature (that doesn't require the feature group object):

```
df = fv.filter(Feature("event_time")=="2024-01-10")
  .filter(Feature("country") == "Ireland")
  .training_data()
```

We already covered partitioning earlier, but we didn't cover how to write filtered queries for multi-column partition keys. For example, if you define 2 columns as your partition key, the order of the columns is important. If you have `['event_time', 'country']` as the partition key, a query that filters on a given *event_time*, it will only read the files that contain rows that contain the date value in your filter. However, it will return data for all the countries in your feature group. To read only feature data for a given *event_time* and *country*, you call:

```
df = fg.filter( (Feature("event_time")=="2024-01-10") |
     (Feature("country")=="Ireland")
    ).read()
```

If, however, you query with a filter on only the *country* name, the partition key filter will not be pushed down. The Hopsworks client will then attempt to execute the filter as a *pushdown predicate*. The Hopsworks Feature Query Service performs data skipping at both the Parquet file level as well as the row group level. It leverages column level statistics collected by the backing lakehouse table (for example, min/max column values for a Parquet file) to skip files when reading data and *zone maps* in the Parquet file's metadata to enable the reader to only fetch row groups with parameter values provided in the query. The Hopsworks Feature Query Service uses Apache Arrow both as a query processing format and an over-the-network format, improving performance significantly compared to feature stores that use JDBC/ODBC servers for querying data.

The role of an Index is to help skip as much data as possible when querying data. Lakehouse tables store their data as Parquet files. Lakehouse tables can have thousands of Parquet files. A well designed feature pipeline will ensure that Parquet file sizes are uniform and of reasonable size (tens of MBs to a few GBs). Too many small files hurts query performance as there are too many files to process. Too few files or skewed file sizes results in inefficient data skipping during query execution. Hopsworks has table services that can run periodically to dynamically adjust file sizes and garbage collect unused files.

Summary

This chapter explores the Hopsworks Feature Store, emphasizing API calls to create and use both feature groups and feature views. We started by looking at how to implement access control for feature data using Hopsworks projects and RBAC. We looked at the internals of the feature groups: the offline store (a lakehouse), the online store (RonDB), and vector index(es). We looked at how to create feature views and use them to create both training data and inference data. Finally, we gave some advice on how to improve the performance of feature store queries using filters.

CHAPTER 6
Model-Independent Transformations

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. The GitHub repo can be found at *https://github.com/featurestorebook/mlfs-book*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

Our focus now switches to how to write the data transformation logic for feature pipelines. As introduced in Chapter 2, feature pipelines are the programs that execute model-independent data transformations to produce reusable features that are stored in the feature store. That is, the feature data created could be used by potentially many different models - not just the first model you are developing the feature pipeline for. Feature reuse results in higher quality features through increased usage and testing, reduced storage costs, reduced feature development and operational costs. And remember, the lowest cost feature pipeline is the one you don't have to create.

Examples of model-independent transformations we will cover in this include the "EVAC" transformations:

- feature Extraction (lagged features, binning, and chunking for LLMs),
- data Validation (with Great Expectations),
- Aggregations (counts and sums for time windows), and

- Compression (vector embeddings).

We will also look at how we can compose transformations in feature pipelines to improve the modularity, testability, and performance of your feature pipelines. However, we will start by setting up our development process - how to organize the source code into packages, and what technologies we can use to implement our transformations in feature pipelines.

Source Code Organization

We will use the source code for our credit card fraud project as a template for how to organize source code such that it follows production best-practices for developing ML pipelines. We need to move beyond just writing notebooks if we are to build production quality pipelines, and that means following software engineering practices such as test-driven development with continuous integration and continuous development (CI/CD). If you make changes to your source code, tests will give you increased confidence that the changes you made do not break either a pipeline or a client that is dependent on an artifact created by your pipeline - whether that artifact is a feature, a training dataset, a model, or a prediction. By automating the execution of the tests, they will not slow down your iteration speed when developing. If you have never written a unit test before, don't worry - LLMs (such as ChatGPT) can help you fix syntactic issues in the source code for your tests, and sometimes even help you identify what to test.

We use a directory structure that organizes all the source code we need to build, test, and run our entire credit card fraud prediction system, see Figure 6-1.

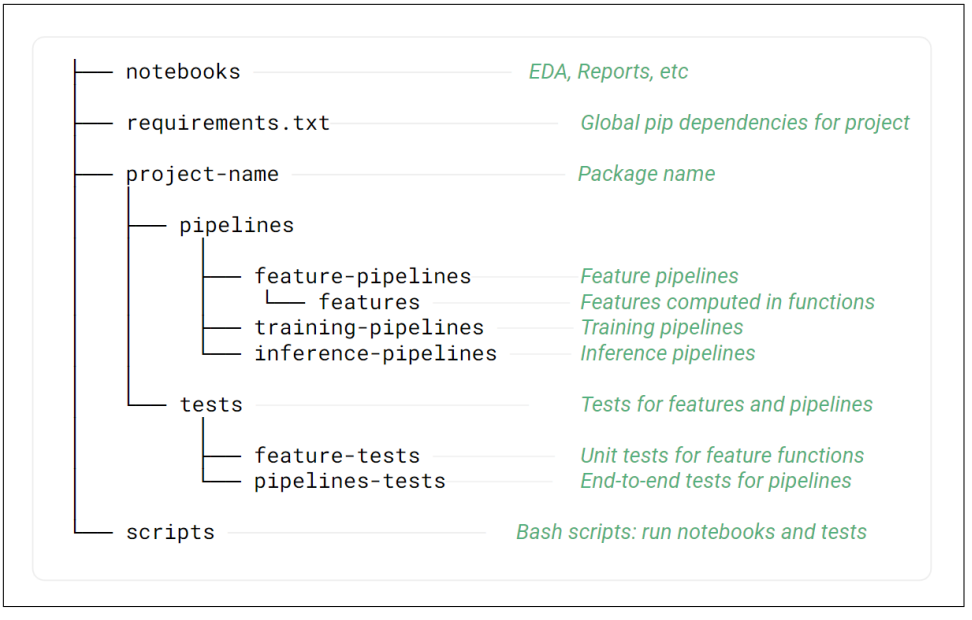


Figure 6-1. For an AI system built with Python, we organize our source code for production by placing the different programs, functions, and tests into different directories, separating production code in the project from EDA in notebooks and helper scripts.

The source code for the different feature, training, and inference pipelines is stored in their own respective directories (*feature-pipelines*, *training-pipelines*, *inference-pipelines*). For easier maintenance, we will store the *tests* in separate files in a dedicated directory outside of our pipeline programs, as this separates the code for our pipelines from the code for testing. We will have two different types of tests - *feature-tests*, unit tests for computing features, and *pipeline-tests*, end-to-end tests for pipelines. Similarly, it is a good idea to separate the functions used to compute features from the programs that implement the feature/training/inference pipelines. We place feature functions in the *features* directory. If you follow this code structure, you will be able to iterate quickly, and not have to later refactor your code for production.

We call this type of project structure a *monorepo*, as the source code for our entire AI system is in a single source code (git) repository. The advantage of a monorepo over separate git repositories for the feature/training/inference pipelines is that we don't have to create and manage installable Python libraries for any shared code between the ML pipelines. The monorepo also does not hinder creating separate production quality deployments for the feature, training, and inference pipelines. For example, each ML pipeline can have its own *requirements.txt* file in its own directory that will be used to build an executable container image for the ML pipeline.

Notice that *notebooks* is a separate directory. It is not part of the production code in the project. It is there to create insights into creating production code - to perform EDA to understand the data and the prediction problem, and to communicate those insights with other stakeholders. Similarly, the *scripts* directory is not part of the production code, and is there to store utility shell scripts for running tests or pipelines during development.

In contrast, Python library dependencies are part of production and included in the project directory as at least one global *requirements.txt* file (for all ML pipelines). Most Python developers have opened the gates of pip dependency hell. For example, I had problems with Pandas 2.x together with Great Expectations 0.16.x. Pandas depends on version 3.1+ of the jinja2 library, while Great Expectations required an earlier version of jinja2, but jinja2 didn't maintain backwards compatibility. For a while, I couldn't use Pandas 2.x with Great Expectations, until Great Expectations upgraded its jinja2 dependency to version 3.1+.

In our credit card fraud project, I included different versioned Python library dependencies for each of our three ML pipelines. That is, the feature, training, and inference pipeline directories each have their own *requirements.txt* file. Within each *requirements.txt*, I pinned the versions to ensure the builds are reproducible by you. If I didn't pin the versions, it is possible an upgrade to a dependency would cause your program to fail. That is unacceptable, so we pin our versions. You can install the Python dependencies in your virtual environment by calling:

```
pip install -r requirements.txt
```

If, instead of pip and a requirements.txt file, you prefer to use a more feature-rich dependency management library, such as Poetry, then please do so. Poetry is great for large projects and manages the Python virtual environment lifecycle using a *pyproject.toml* file. We will use pip and *requirements.txt* files as they have a lower barrier to entry and better integration with cloud platforms that build container images from *requirements.txt* files.

Feature Pipelines

Feature pipelines read data from some data sources, create features from the data read, and write their output feature data to the feature store. Before we dive deep into feature engineering, we will first look at a number of popular open-source data transformation engines, introduced in Chapter 2. Given a group of features you want to compute together (and write to a feature group), you should understand the trade-offs between using different available engines, based on the expected data volume and the freshness requirements for the features. Most compute engines for feature engineering fall into one of the following computing paradigms:

- Stream processing for streaming feature pipelines (Python, Java, or SQL)

- DataFrames for batch feature pipelines (Python)
- Data warehouses for batch feature pipelines (SQL)

There are also other specialist compute engines for feature engineering, including some that leverage GPUs, but due to space considerations we restrict ourselves to widely adopted open-source engines: Pandas, Polars, Apache Spark, Apache Flink, and Feldera (a stream processing engine using SQL). In Figure 6-2, you can see how to implement data transformations in these frameworks, organized by whether they:

- scale to process data that is too big to be processed by a single server (PySpark, Apache Flink),
- are stream processing frameworks (Feldera, Apache Flink),
- support real-time computation of feature data in prediction requests (Python UDFs),
- are batch data transformations (Pandas, Polars, DuckDB, and PySpark).

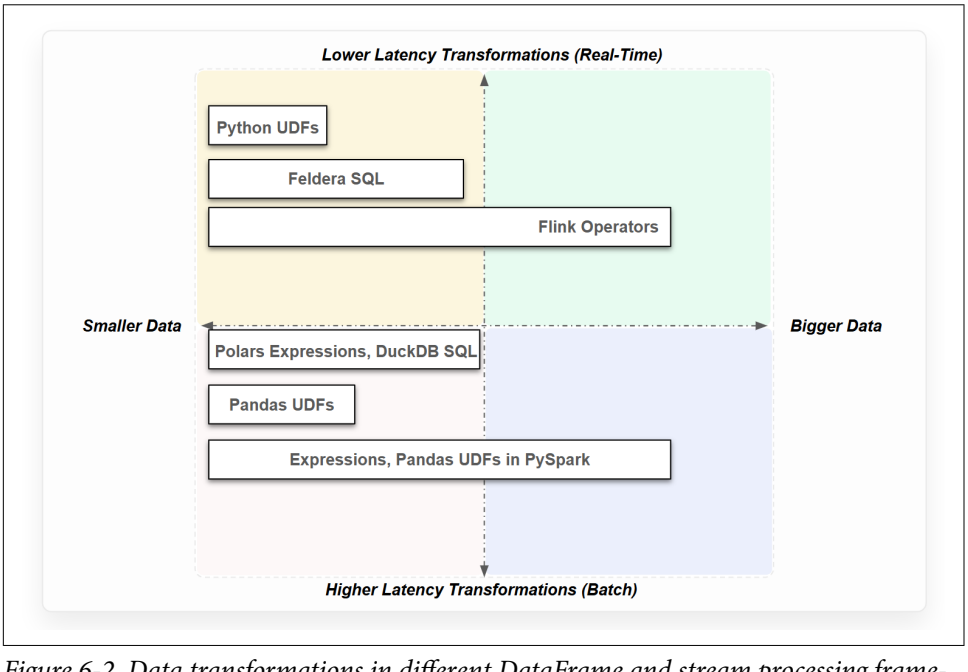


Figure 6-2. Data transformations in different DataFrame and stream processing frameworks have different latency and scalability properties. For each feature pipeline, you should select the best framework, given the scale and latency requirements for the features it creates.

For stream processing, Apache Flink and Spark Streaming are widely used as distributed, scalable frameworks. Both, however, have steep learning curves and high operational overhead. Feldera is another single-machine stream processing engine with support for incremental computation with SQL, see Chapter 9.

For batch processing with DataFrames, Pandas, Polars, and PySpark are the main frameworks that we will work with in this chapter. Batch processing with SQL can be performed in data warehouses, such as Snowflake, BigQuery, Databricks Photon, or Redshift, or on single-host SQL engines, such as DuckDB. DBT has become a popular framework for orchestrating feature engineering pipelines as a series of SQL commands. Table 6-1 provides a guide as to when you should choose one engine over another.

Table 6-1. Example AI systems and the candidate frameworks that are best suited for computing features, based on whether the AI systems require fresh features (stream processing) or not (batch processing), and whether the feature pipelines will process big data (distributed compute engines) or not (single server compute engines).

Data Volume	Feature Freshness	Candidate Frameworks	Example Feature Pipelines for AI Systems
Large	1-3 secs	Flink (Java)	Clickstreams for TikTok Scale Recommenders
Small-Medium	1-3 secs	Feldera (SQL)	Real-Time Logistics (delivery, drivers), smaller clickstream processing, cybersecurity events.
Small	1-3 secs	Python: Pathway, Quix, Bytewax	Streaming ML systems (Intrusion detection, Industry 4.0, Edge),
Large	Mins to hrs	PySpark or dbt/SQL	Personalized marketing campaigns and segmentation, batch fraud, customer churn, credit scoring, demand forecasting
Large Unstructured	Mins to hrs	PySpark	Image augmentation, Text processing (e.g., chunking), video pre-processing (PySpark)
Small-Medium	Mins to hrs	Pandas, Polars, DuckDB	Same as previous for smaller data volumes, data fetching from APIs,
Small- Large	Mins to hrs	Optionally with GPUs: Pandas, Polars, PySpark	Vector embedding text chunking pipelines for RAG, Video pre-processing

In general, you should choose stream processing if you are building a real-time AI system that needs fresh precomputed features. If feature freshness is not important, you should probably write a batch feature pipeline as they have lower operational costs. You should prefer DataFrame compute engines (Pandas/Polars/PySpark) over SQL when:

- you need to fetch data from APIs,
- extensive data cleaning is required,
- you need to transform unstructured data (images, video, text),
- you need to use feature engineering libraries that are only available in Python,

- you need to write transformations with custom logic.

Feature engineering with DataFrames can be scaled up on a single machine by switching from Pandas to Polars, which makes better use of available memory and CPUs. When data volumes are too large for a single machine, you can use PySpark that can be scaled out over many workers to TB or PB-sized workloads.

We will now briefly cover SQL for feature engineering. SQL should be used over DataFrames when you have a batch feature pipeline, all of the source data is in the data warehouse or lakehouse, and your feature engineering can be implemented in SQL. SQL-based feature engineering is declarative, leveraging the power of relational operations and the scale of data warehouses or query engines on top of lakehouse tables.

For example, in Hopsworks, SQL-based transformations for a batch data source can be defined in an *external feature group* and the SQL that performs the transformations and returns the transformed data is run when you read data from the feature group (typically in a training or batch inference pipeline). In this case, there is no feature pipeline to schedule as features are computed when they are read.

SparkSQL transformations such as

```
df.createOrReplaceTempView("data")
df_features = spark.sql("SELECT *, (col1 + col2) AS feature_sum FROM data")
```

Data Transformations for DataFrames

Feature engineering with both DataFrames and SQL tables involves performing row-wise and column-wise transformations on the data. I find one useful way to understand each data transformation is how it changes the rows and columns in your DataFrame(s) or SQL table(s).

You need to know what the result of the data transformations will be - will it add or remove columns, reduce the number of rows, or add more rows? Figure 6-3 shows the different classes of transformations that can be performed on tabular data. In the discussion below, we will restrict ourselves to data transformations on DataFrames. The code snippets are in Polars, as most of the batch transformations in our credit card fraud detection system are written in Polars. Similar to Pandas, Polars is a DataFrame engine that runs on a single machine, but it scales to handle much larger data volumes thanks to better memory management and multi-core support.

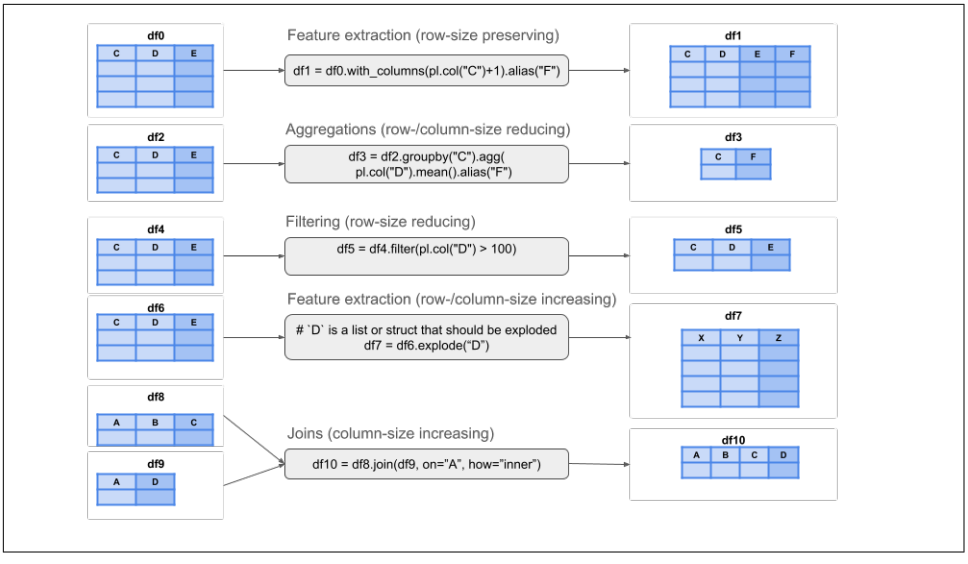


Figure 6-3. Data transformations produce output DataFrames that often do not match the shape of the input DataFrame(s). Some transformations add rows and/or columns, some keep the same number of rows, and some reduce the number of rows and/or columns.

We can classify DataFrame transformations into the following *cardinalities*:

- *row-size preserving transformation* where you **add a new column** to an existing DataFrame without changing the number of rows. Feature extraction is a typical example of one such data transformation.
- *row-/column-size reducing transformation* where the input DataFrame has more rows than the output DataFrame. Examples of such transformations include *group-by-aggregations*, *filtering*, or data compression (*vector embeddings, PCA*),
- *row-/column-size increasing transformation* where the input DataFrame has less rows than the output DataFrame. A common example is feature extraction that involves exploding JSON objects, lists, or dicts stored in columns in DataFrames. Cross-joins also belong here, too. As do user-defined table functions (in PySpark).
- *Join transformations* involve merging together two input DataFrames to produce a single DataFrame (with more columns than either of the input DataFrames). Joins are needed when you have data from different sources and you want to compute features using data from both sources. Joins are sometimes needed to build the final DataFrame that is written to a feature group.

Row-Size Preserving Transformations

Here is an example of a row-size preserving transformation, implemented as a Pandas UDF, that identifies rows that include outliers by setting a boolean value for *is_outlier* in a new column in the DataFrame:

```
def detect_outliers(value_series: pd.Series) -> pd.Series:
    """Add a column that indicates whether the row is an outlier"""
    mean = value_series.mean()
    std_dev = value_series.std()
    z_scores = (value_series - mean) / std_dev
    return np.abs(z_scores) > 3

df = df.withColumn("is_outlier", detect_outliers(df["value"]))
```

We may compose this transformation with row-reducing transformation that removes the rows that are considered outliers:

```
def remove_outliers(df: pd.DataFrame) -> pd.DataFrame:
    """Remove the rows in the DataFrame where is_outlier is True"""
    return df[df["is_outlier"] == False)]
df_filtered = remove_outliers(df)
```

Other examples of row-size preserving data transformations include:

- applying a UDF as a lambda function in Polars (or an *apply* in Pandas, or a Pandas UDF in PySpark). This code (that stores the squared value of a column in *new_col*) applies the lambda function to *col1* using the *map* function:

```
df.with_columns(df["col1"].map(lambda x: x * 2).alias("new_col"))
```

- a rolling window expression that computes the mean amount spent on a credit card for the previous 3 days:

```
df.with_columns((col("amount").rolling_mean(window_size=3)
                .over("cc_num")).alias("rolling_avg"))
```

- conditional transformations (*when, then, otherwise, select*). Here, if *col* is 0, the set *new_col* to *positive*, else set it to *non_positive*:

```
df.with_columns((pl.when(df["col"]==0).then("positive")
    .otherwise("non_positive")).alias("new_col"))
```

- temporal transformations that capture time-related information about the data. Here, we compute the number of days since the bank's credit rating was last changed:

```
df.with_columns((pl.lit(datetime.now()) - pl.col("last_modi-
fied")).alias("days_since_bank_cr_changed")
```

- sorting and ranking. This code computes in *rank_col* the rank of each value in *col*.

```
df.with_columns(df["col"].rank().alias("rank_col"))
```

- mathematical transformations. Here, we store the sum of *col1* and *col2* in *sum_col*:

```
df.with_columns((df["col1"] + df["col2"]).alias("sum_col"))
```

- String transformations. This transformation uppercases the string in *name* and stores it in *uppercase_name*:

```
df.with_columns(df["name"].str.to_uppercase().alias("uppercase_name"))
```

- Lag and lead. This code stores yesterday's *pm25* value in *lagged_pm25*:

```
df.with_columns(df["pm25"].shift(1).alias("lagged_pm25"))
```

Row- and Column-Size Reducing Transformations

Aggregations are an example of a well-known data transformation that reduces the number of rows from the input DataFrame (or table). Aggregations summarize data over a column and optionally an additional time window (a time range of data), capturing trends or temporal patterns. Aggregations are useful in AI systems with sparse data and temporal patterns, such as fraud detection, recommendation engines, and predictive maintenance applications.

Aggregations are functions that summarize a window of data. The data could include all of the input data or a time window, a period over which the aggregation is performed. Common aggregation functions include:

Count
: Number of events.

Sum
: Total value (e.g., total transaction amount).

Mean/Median
: Average value.

Max/Min
: Extreme values.

Standard Deviation/Variance
: Measure of variability.

Percentiles
: Specific thresholds, such as the 90th percentile.

Aggregations are computed for entities, for example:

- Per credit card
- Per customer
- Per merchant/bank
- Per product/item

In SQL and PySpark you use *group by* and a *window*. Polars supports grouping by time windows through the *groupby_rolling* and *groupby_dynamic* methods and then applying aggregations. Pandas supports time-based grouping through *resample* and *rolling*, which can be combined with aggregation functions. Here is example aggregation in Polars without a time window that handles missing data filling missing values with the forward fill strategy (replace null values with the last valid (non-null) value that appeared earlier in the data):

```
filled_df = (
    df.groupby("cc_num", maintain_order=True)
    .agg([
        pl.col("event_time"),  # Keep transaction_time
        pl.col("amount").fill_null(strategy="forward")
        .sum().alias("total_amount"),
    ])
)
```

In the previous code snippet, the output DataFrame, *filled_df*, includes the *event_time* column from *df* and adds the new *total_amount* column containing the result of the aggregation. All other columns from *df* were not retained, as aggregations typically reduce the number of columns. For example, if you are computing the sum of the transactions for a credit card number, it is not meaningful to retain the *category* column in that transformation. If you want to compute an aggregate for the *category* column, you perform a separate transformation on that column.

Aggregations support different types of time windows, some of which are row-size reducing and some of which are not. Rolling window aggregations compute an output for every row in source DataFrame, and are therefore not row-rize reducing. In contrast, tumbling windows compute an output for all events in a window length, so they typically reduce the number of rows. For example, if your window length is 1 week and there are, on average, 20 transactions per week, you will reduce the number of rows, on average, by a factor of 20.

Sometimes aggregations require composing transformations. For example, if we want to compute the following: "find the maximum amount for each *cc_num* that has 2 or more transactions from the same category". Here, we need to group by *cc_num*, then

we have to remove those transactions that have only one entry for a given category, then for each remaining category (with >1 transaction), find the maximum amount. This might seem like a complex example, but it is not uncommon when you need to find specific signals in the data that are predictive for your problem at hand. Polars lets us elegantly and efficiently compose group-by aggregations and expressions.

```
df3 = df.group_by("cc_num").agg(
    pl.col("amount").filter(pl.col("category").count() > 1).max()
)
```

Vector embeddings are another data transformation that compresses input data into a smaller number of rows and columns. You create a vector embedding from some high dimensional input data (rows and columns) by passing it through an *embedding model* that then outputs a vector. The vector is a fixed sized array (its length is known as its *dimension*) containing (normally 32-bit) floating point numbers. The embedding model is a deep learning model, so if you are transforming a large volume of data into vector embeddings, you may be able to speed it up considerably by performing the data transformations on GPUs rather than CPUs. In this example code, we encode the *explanation* string for a fraudulent credit card transaction with the *SentenceTransformer* embedding model:

```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('all-MiniLM-L6-v2')
embeddings = model.encode(df["explanation"].to_list())
df = df.with_columns(pl.Series("embedding_explanation", embeddings))
```

If you write this vector embedding to a vector database (or a feature group in Hopsworks), you can then search for records with similar explanation strings using k-nearest neighbor (kNN) search. KNN search is a probabilistic algorithm that returns *k* records containing vector embeddings that are semantically close to the provided vector embedding. The size of *k* can range from a few to a few hundred records.

Row-/Column-Size Increasing Transformations

It is becoming more common to store JSON objects in columns in tables. In order to create features from values in the JSON object, you may need to first extract the values in the JSON object as new columns and/or new rows. You can do this by exploding the column containing the JSON object. In Polars, this involves first casting the column to a *struct*, then calling *unnest* to explode the struct into separate columns:

```
df = pl.DataFrame({
    "json_col": [
        {"name": "Alice", "age": 25, "city": "Palo Alto"},
        {"name": "Bob", "age": 30, "city": "Dublin"}, …
    ]})
df = df.with_columns(pl.struct("json_col").alias("json_struct"))
df_exploded = df.unnest("json_struct")
```

If you have json objects in a column, in Polars, you can define them first as a struct and then *unnest* the column to explode *details* into separate columns. At the end, *df_exploded* contains the columns *["name", "age", "city"]*.

In PySpark, user-defined table functions (UDTFs) are functions that transform a single input row into multiple output rows. In contrast, UDFs work on a row-to-row basis. UDTFs can, for example, explode a JSON structure in a column to multiple rows based on deeply nested fields. UDTFs are not available in Polars or Pandas. UDTF execution is parallelized on Spark, but they are not inherently vectorized like Pandas UDFs or Arrow-based functions. This means they introduce overhead due to serialization, deserialization, and Python-JVM boundary crossing. As of Spark 3.x, PySpark doesn't support custom User-Defined Table Functions (UDTFs). Custom UDTFs can be written in Java/Scala Spark.

Exploding JSON objects is not the only row-size increasing data transformation. Imagine we want to create a feature for the total spending of each customer per transaction category. However, transactions are organized by *cc_num* (entity ID), so we need to pivot the DataFrame to transform columns into rows and compute a *spend_category* column:

```
pivot = (
    df.groupby(["cc_num", "category"])
    .agg(pl.col("amount").sum())
    .pivot(values="amount", index="cc_num", columns="category")
    .fill_null(0)  # Replace nulls with 0
)
pivot = pivot.rename({col: f"spend_{col}" for col in pivot.columns if col !=
"cc_num"})
```

Similarly, you also unpivot columns into rows using *melt*:

```
dv_unpivot = df.melt(id_vars=["cc_num"], value_vars=["category"])
```

Join Transformations

A common requirement when selecting features for a model is to include features that "belong" to different entities. For example, you could have features in different feature groups with different entity IDs (e.g., *cc_num* and *account_id*), but you would like to use features from both feature groups in your model. In this case, we often need to join two or more DataFrames together using a common *join key*.

The following is an example of joining two DataFrames together in Polars. Note that Pandas uses the method *merge* instead of *join* for this operation (PySpark uses *join*).

```
merged_df = transaction_df.join(account_df, on="cc_num", how="left")
```

Here, we perform an *inner join*, which will take every row in *transaction_df* and look for a matching *cc_num* in *account_df*. It will skip rows in *transaction_df* that do not have a matching *cc_num* in *account_df*. What if there is no account information for a

transaction, but we still would like to include the transaction (as we can infer reasonable values for the account during training or inference)? In this case, we can change the policy to a left (outer) join, with `how="left"`. Inner and left joins are the most widely used joins for feature engineering. Note that an *left outer join* will be a row-size preserving transformation for the left-hand DataFrame in the join operation, but an *inner join* will either be a row-size preserving or row-size reducing transformation, depending on whether there all matching rows in the right-hand DataFrame for all rows in the left-hand DataFrame (preserving) or not (reducing).

DAG of Feature Functions

In Chapter 2, we argued that transformation logic should be factored into feature functions in order to improve code modularity and make transformations unit testable. A feature pipeline is a series of well defined steps that transform source data into features that are written in the feature store:

1. read data from one or more data sources in ti one or more DataFrames
2. apply feature functions to transform data into features and to join features together
3. write DataFrame containing featurized data to the corresponding feature group.

The feature pipeline should be parameterized by its data input so that you can run the feature pipeline either with historical data or with new incremental data. Assuming your data source supports data skipping, you should only select the columns you need and filter out the rows you don't need. If you work with small data, you may get away with reading all the data from your data source into a DataFrame, and then dropping the extra columns and filtering out the data you don't need. However, with large data volumes, this is not possible and you need to push down your selections and filters to the data source.

Once you have read your source data into DataFrame(s), the feature pipeline organizes the feature functions in a dataflow graph. A dataflow graph is a directed acyclic graph (DAG) that has inputs (data sources), nodes (DataFrames), edges (feature functions), and outputs (feature groups). Figure 6-4 shows three different feature functions *g()*, *h()*, and *j()*, where *df* is read from the data source, *g()* is applied to *df* to produce *df1*. Then in parallel, *h()* and *j()* are applied to (potentially different columns) in df1 in parallel, producing *dfM* and *dfN*, respectively. (Note, PySpark and Polars support parallel executions, while Pandas does not).

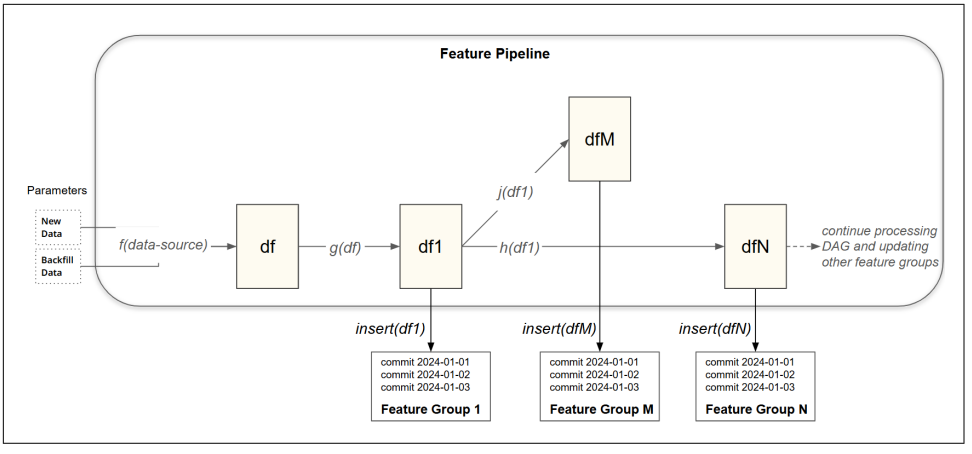


Figure 6-4. A feature pipeline reads new data or backfill data into a DataFrame (df), and then applies a directed acyclic graph of data transformations on df using feature functions f, g, h, and j. The output of each feature function g, h, and j is a DataFrame that is written to feature group 1, M, and N, respectively.

The graph structure inherently represents dependencies between the transformations, as one featurized DataFrame can be the input to another. When the output of one transformation is used as the input to another transformation, we say that the data transformations have been composed, as presented in Chapter 4. Both intermediate and leaf nodes in the DAG can write DataFrames to feature groups. Here, *df1* is written to feature group *1*, *dfM* to feature group *M*, and *dfN* to feature group *N*.

Lazy DataFrames

Pandas supports *eager evaluation* of operations on DataFrames. Each command is processed right away. In a Jupyter notebook, you see the result of the operation directly after it has been executed. This is a powerful approach for learning to write data transformations in Pandas. In contrast, DataFrame frameworks that support *lazy evaluation*, such as Polars and PySpark, can wait across multiple steps before the commands are executed. Waiting provides the possibility to optimize the execution of the steps. But how long do you wait before executing? Lazy DataFrames are like a quantum state, where the act of observing gives you the result. With Lazy DataFrames, an "action" (reading the contents of a DataFrame or writing it to external storage) triggers the execution of the transformations on it. While eager evaluation is great for beginners, it is not great for performance. As data volumes inexorably increase, you should learn to work with Lazy DataFrames. Both Polars and PySpark are built around Lazy DataFrames.

The following code snippet in Polars creates a Lazy DataFrame from a CSV file, then computes the *mean* value of the *amount* column, and then computes the *devia-*

tion_from_mean by subtracting the *mean* from the *amount*. This is a useful feature in credit card fraud. However, all of these steps are only executed when the code reaches the last line - an action, *collect()*, to read its contents:

```
# Lazy loading with pl.scan_csv
lazy_df = pl.scan_csv("transactions.csv")

# Compute the mean, then create a new column for deviation from mean
lazy_df = lazy_df.with_columns([
    (pl.col("amount") - pl.col("amount").mean()).alias("deviation_from_mean")
])

# Trigger execution and collect the result
result = lazy_df.collect()
```

Vectorized Compute, Multi-Core, and Arrow

For performance reasons, we avoid writing data transformation code using DataFrames and native Python language features such as for/while loops, list comprehensions, and map/reduce functions. The code examples we have introduced thus far are based on idioms such as *with_columns(...)* and Pandas UDFs. DataFrame transformations that follow these idioms are executed by a vectorized compute engine and not executed in native Python code. They are orders of magnitude faster than native Python code for two main reasons. Firstly, Python's standard execution model is interpreted bytecode, lacking native vectorization. Secondly, Python programs are constrained by the Global Interpreter Lock, which prevents efficient scalability on multiple CPU cores.

A vectorized compute engine performs operations on large arrays or data structures by applying single instructions to multiple data points simultaneously (SIMD). These operations can also be parallelized across multiple CPU cores to further improve scalability. Pandas, Polars, and PySpark all have vectorized compute engines. Polars and PySpark both have good multi-core support, while Pandas (up to version 2) does not.

You should write your data transformations so that they are executed in the vectorized compute engine rather than run in Python as interpreted bytecode, see Figure 6-5. A trivial example would be a for-loop to process a Pandas DataFrame. Please, don't do this. A more common performance bottleneck in Pandas is a Python UDF that you *apply* to a DataFrame. This will involve the data being copied from the backing store (Arrow supported in Pandas v2) to Python, where the data will be the UDF is applied, and then back to Arrow.

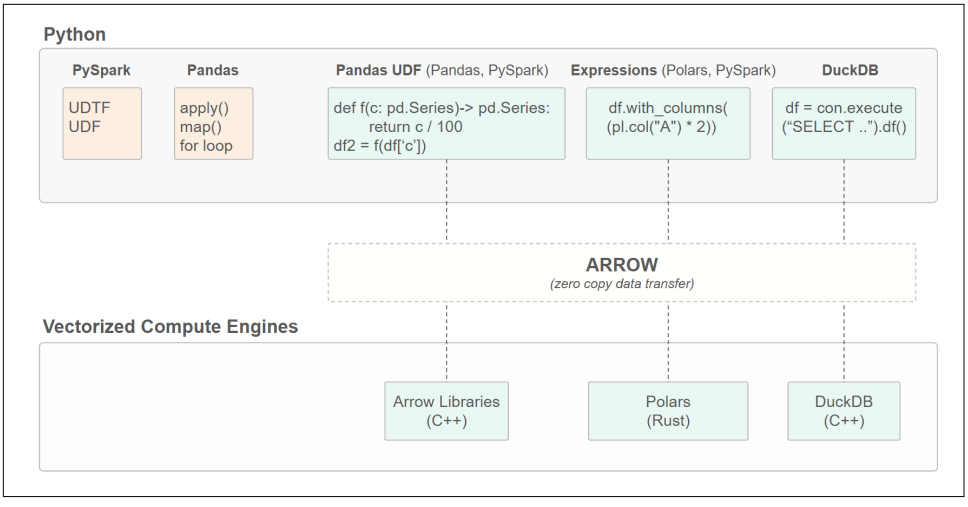


Figure 6-5. When you define data transformations in Python, it is important to use transformations that have vectorized implementations. Pandas supports both NumPy and more recently Arrow transformations (shown). Polars has a rust engine, while DuckDB has a C++ engine.

For example, the following Python UDF, executed with apply in Pandas, takes 7.35 seconds on my laptop (Windows Subsystem for Linux, 32GB RAM, 8 CPUs).

```
num_rows = 10_000_000
df = pd.DataFrame({ 'value': np.random.rand(num_rows) * 100})

def python_udf(val: float) -> float:
    return val * 1.1 + math.sin(val)

df['apply_result'] = df['value'].apply(python_udf)
```

If I rewrite the same UDF as a Pandas UDF, using NumPy as a vectorized compute engine, it completes in only 0.28 seconds:

```
import numpy as np

def pandas_udf(series: pd.Series) -> pd.Series:
    return series * 1.1 + np.sin(series)

df['pandas_udf_result'] = pandas_udf(df['value'])
```

I can also rewrite the same code as an expression in Polars, and it has roughly the same execution time as the vectorized Pandas UDF:

```
df_polars_expr = df.with_columns(
    (pl.col("value") * 1.1 + pl.col("value").sin()).alias("result")
)
```

In this case, Polars is not faster than Pandas, as there is no parallelization over multiple CPUs. Polars, however, has superior memory management for larger data volumes. I can run this Polars code with 500m rows, but the Pandas code crashes at that scale.

We can also rewrite the above code as a PySpark program. PySpark supports lazy evaluation, *withColumn* expressions, and Pandas UDFs:

```
@pandas_udf("double")
def pandas_udf(value: pd.Series) -> pd.Series:
    return value * 1.1 + np.sin(value)

df["result"] = df.withColumn("pandas_udf_result", pandas_udf(col("value")))
```

The above code uses Arrow to `efficiently` transfer data between PySpark's Java Virtual Machine (JVM) and Python. We can also rewrite the previous code in PySpark as a *withColumn* expression:

```
from pyspark.sql.functions import col, sin

df = df.withColumn(
    "result", (col("value") * 1.1 + sin(col("value")))
)
```

This code uses PySpark's SQL expression API and is performed natively in the Spark engine, without the need to transfer data from the JVM to the Pandas UDF.

Lastly, we can re-write the above code in Python using DuckDB, an embedded SQL engine that can outperform even Polars for some classes of data transformation:

```
import duckdb
con = duckdb.connect()
con.register("input_df", df)

result_df = con.execute("""
    SELECT
        value,
        value * 1.1 + SIN(value) AS result
    FROM input_df
""").fetchdf()
```

This returns *result_df* as a Pandas DataFrame and transfers data to and from Pandas using Arrow.

Pandas, Polars, PySpark, and DuckDB all can natively transfer their data as Arrow tables. So, you can, with zero cost, move DataFrames between Pandas, Polars, and DuckDB by reading the source DataFrame as an ArrowTable and then creating a DataFrame from that Arrow table in your target framework. This way you can write feature pipelines that perform some data transformations in DuckDB, some in Pandas, and some in Polars - without any overhead when moving DataFrames between the different engines. PySpark, in contrast, is a distributed compute engine, where

DataFrames are partitioned across workers. Converting a PySpark DataFrame to a Pandas DataFrame requires first collecting the distributed PySpark DataFrame on the driver node - a process that can potentially overload the driver resulting in an out-of-memory error.

Arrow is a language independent in-memory columnar format that is an efficient data interchange format between different programming languages and frameworks and supports dictionary compression. Since Arrow data is already in a serialized format, it can be directly sent over the network or shared between processes without converting to or from other formats. For example, Arrow Flight is a gRPC-based network protocol for transferring Arrow data between systems. Arrow is also efficient for feature engineering tasks such as computing aggregations on columns as it is an in-memory columnar format. PyArrow is a popular Python library for working Arrow data.

In the following code snippet, we show how to write a feature pipeline that performs processing steps in different compute engines, using Arrow to efficiently transfer data between the engines. First, we first create a Pandas DataFrame *pdf* containing three input columns: employee's name, age, and salary. We then transform the employee's age from a number to a categorical variable (junior or senior) in Polars. Then we transform their salary from a number to a categorical variable (junior, mid-level, or senior) in DuckDB.

```
import polars as pl
import duckdb
import pyarrow as pa

pdf = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Charlie', 'David'],
    'age': [25, 30, 35, 40],
    'salary': [50000, 60000, 75000, 90000]
})
# Convert Pandas DataFrame to PyArrow Table (zero-copy)
arrow_table = pa.Table.from_pandas(pdf)

# Convert to Polars DataFrame (zero-copy)
pldf = pl.from_arrow(arrow_table)

pldf_transformed = pldf.with_columns([
    pl.when(pl.col('age') < 35)
    .then(pl.lit('Junior'))  # Use pl.lit() for string literals
    .otherwise(pl.lit('Senior'))
    .alias('age_category')
])
print(pldf_transformed)
arrow_table_transformed = pldf_transformed.to_arrow()
```

```
# Create in-memory DuckDB database and register Arrow Table
con = duckdb.connect(':memory:')
con.register('employee_table', arrow_table_transformed)

# Perform a transformation in DuckDB, returns Pandas DF
result_df = con.execute("""
    SELECT name, age_category,
        CASE
            WHEN salary < 60000 THEN 'Junior'
            WHEN salary BETWEEN 60000 AND 80000 THEN 'Mid-level'
            ELSE 'Senior'
        END as salary_band
    FROM employee_table
""").df()
con.close()
print(result_df)
```

We start by creating a Pandas DataFrame *pdf* containing our data and convert it to a PyArrow table, *arrow_table*. We then create a Polars DataFrame *pldf* using *arrow_table*, again without copying data and create a categorical feature by transforming *age* - we categorize people as either *Junior* or *Senior*. Then, we create a DuckDB instance and register *arrow_table_transformed*, retrieved from the Polars DataFrame without copying, as *employee_table*. Finally, we compute a categorical variable *salary_band* using *salary*.

Data Types

When you write code in ML pipelines, you work with the corresponding Polars/Pandas/PySpark/SQL data types. However, ML pipelines interoperate via the shared feature store layer, and every feature store has its own set of supported data types. One complication can arrive if you use a different framework in a feature pipeline compared to the training/inference pipelines. For example, the feature pipeline could run in PySpark, while the training pipeline uses Pandas to feed samples to the model. However, PySparks supports a different set of data types compared to Pandas. The feature store connects these two pipelines by storing data in its native data types, and casting data to/from the framework's data types.

In Figure 6-6, you can see how a feature pipeline written in PySpark writes a DataFrame containing four columns with the native PySpark data types TimestampType, DateType, StringType, and BinaryType.

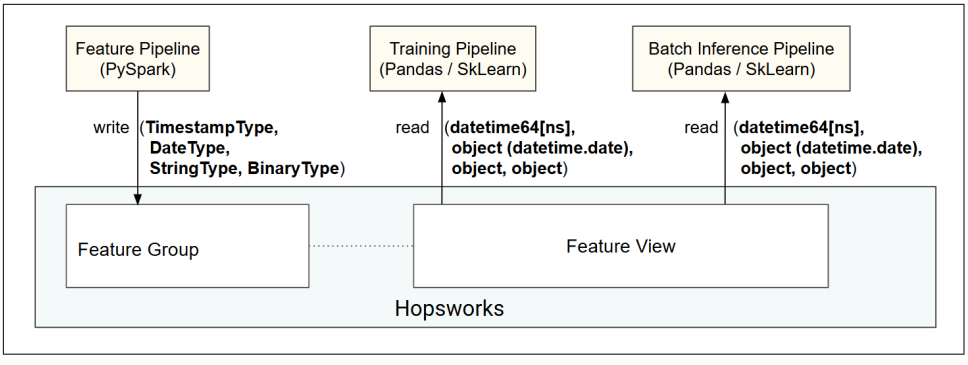


Figure 6-6. When you write typed data to the feature store (in a feature pipeline), the training and inference pipelines should read the data with the same (or compatible) data types. The offline and online feature groups store the data in their own native types.

Hopsworks stores them as Hive data types in its offline store, and when Pandas clients in the training/inference pipelines read the features they read them as Pandas data types datetime64[ns], datetime64[ns], object, and object.

One potential problem when writing ML pipelines is a mismatch between data types in your ML pipeline and data types supported by your feature store. For example, compute engines support a wider variety of data types than are supported by feature stores. This can lead to a loss of precision. For example, an unsigned 8-bit int could be cast to a signed 16-bit int. Or in Figure 6-7, a PySpark DateType is cast to an object dtype in Pandas containing a *datetime.date* object.

The feature store is responsible for storing the feature data in its native data types and ensuring that different combinations of frameworks can read and write data as expected. It should ensure that irrespective of whether you use SQL/Pandas/Polars/PySpark/Flink for the feature pipeline, the training and inference pipelines should be able to read the feature data in supported DataFrame engines. There is also the added complication that the feature store stores data in both offline tables and online tables, each of which may support different data types. For example, in Hopsworks the offline table uses Hive data types, while the online table uses MySQL data types.

Arrays, Structs, Maps, and Tensors

Apart from primitive data types, other data types can be stored in feature stores. For example, arrays, structs, and maps are all supported by Hopsworks. Vector embeddings are stored as an array of floats. The other main data structure in machine learning is the tensor. A tensor is a multi-dimensional numerical data structure that can represent data in one or more dimensions. Unlike traditional matrices, which are two-dimensional, tensors extend to three or more dimensions. In deep learning, tensors are commonly constructed from unstructured data, such as images (3D tensors),

videos (4D tensors), or audio signals (1D tensors), enabling the representation and processing of complex data formats. PyTorch is the most popular framework for deep learning. PyTorch represents tensors as instances of the *torch.Tensor* class, with the default data type being *torch.float32* (*torch.int64* is the default for integer tensors). You can print the shape of a tensor using the *shape* attribute of *torch.Tensor*: `print(tensor.shape)`.

We typically do not store tensors in a feature store. Instead, in training/inference pipelines, unstructured data (in compressed file formats such as png, mp4, and mp3 for images, video, and sound, respectively) is transformed into tensors when it is read:

```
import torch
from torchvision import transforms
from PIL import Image
image = Image.open("path/to/your/image.png")

# Define a transformation pipeline to convert the image into a tensor
transform = transforms.Compose([ transforms.ToTensor() ])
image_tensor = transform(image)
```

It is, however, sometimes desirable to preprocess the files in a training dataset pipeline that outputs tensors in files, such as *.tfrecord* files. TFRecord is a file format that can natively store serialized tensors. Using *.tfrecord* files can reduce the amount of CPU preprocessing needed in training pipelines by removing the need to convert unstructured data into tensors, helping improve GPU utilization levels - assuming CPU preprocessing is a bottleneck in the training pipeline.

Implicit or Explicit Schemas for Feature Groups

In Chapter 3, we saw how convenient it is to infer the schema of a feature group from a DataFrame. You may already have written programs that read CSV files into DataFrames in Pandas, Polars, or PySpark and noticed that they don't always infer the "correct" datatypes. By correct, we mean the data type you wanted, not the one you got. For example, Pandas can infer the schema of columns when reading CSV files, but if one of the columns is a datetime column, Pandas by default infers it is an *object* (string) dtype. You can fix this by passing a parameter with the columns that contains dates (`parse_dates=['col1',..,'colN']`). PySpark is not much better with CSV files, as it assumes all columns are strings, unless you set `inferSchema=True`.

In production feature pipelines, it is generally considered best practice to explicitly specify the schema for a feature group, helping prevent any type inference errors or precision errors when inferring data types. If in doubt, spell it (the schema) out. Here is an example for specifying an explicit schema for a feature group in Hopsworks:

```
from hsfs.feature import Feature
features = [
```

```
    Feature(name="id",type="int",online_type="int"),
    Feature(name="name",type="string",online_type="varchar(2000)")
]

fg = fs.create_feature_group(name="fg_with_explicit_schema",
                             features=features,
                             …)
fg.save(features)
```

Note that you can also explicitly define the data types for the offline store and the offline store as part of the feature group schema.

Credit Card Fraud Features

We now revisit our credit card fraud detection system. We start by noting the data-related challenges in building a robust credit card fraud detection system. They include:

- class imbalance - we have very few examples of fraud compared to non-fraud transactions,
- non-stationary prediction problem, as fraudsters constantly come up with novel strategies for fraud, so we will need to frequently retrain our model on the latest data,
- data drift, where unseen patterns in transaction activity are common,
- ML fraud models are typically used in addition to rule-based approaches that detect simple fraud schemes and patterns.

In Chapter 4, we introduced the features we want to create from our source data. We now present the model-independent data transformations used to create those features. Figure 6-8 shows the feature pipeline that uses the tables (and event bus) in our data mart as the data sources. The data mart includes credit card transactions as events in an event bus, a fact table that the credit card transaction events are persisted to, the four dimension "details" tables, and the table *cc_fraud* containing labels, see Figure 6-7.

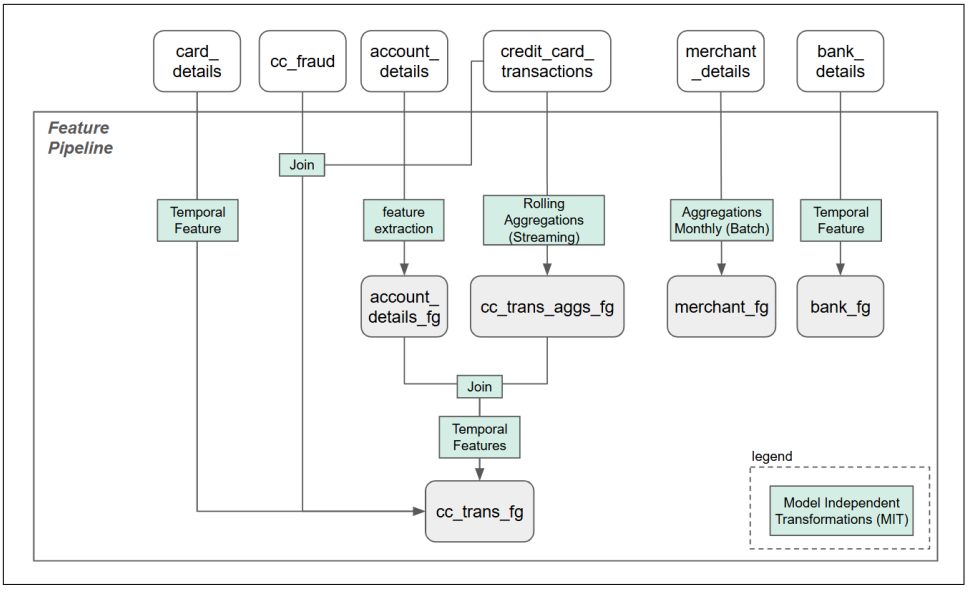


Figure 6-7. Dataflow graph from the data mart to the feature groups via model-independent transformations. Notice that some data transformations are composed from other transformations (the input of a transformation is the output of another transformation), and that joins bring features from different entities (cards, accounts, merchants) together.

We will now take a new approach to defining our transformation logic. Instead of presenting the source code, we will present the prompts that I used to create the transformation logic using a LLM. Table 6-2 shows the prompts I used to create the transformation code in the book's source code repository. As of early 2025, LLMs are very good at creating working Pandas, Polars, and PySpark code. Typically, you also have to pre-pend the *logical models* for your tables (see Chapter 8), so that the LLM understands the data types and the semantics of the columns it is working with.

Table 6-2. Here are some sample prompts that can be used to create the Polars code that creates features from our data sources.

Feature	Prompt to write code for feature
chargeback_rate_prev_week	From *merchant_details*, write Polars code to compute a 28 day tumbling window using *chargeback_rate_prev_week*. Read up from the FG with overlap for the 28 days before our start date, as we don't want empty first. We want this feature function to take start/end dates, so it can both backfill and take new data.
time_since_last_trans	Join *cc_trans_aggs_fg* with *cc_trans_fg* using *cc_num* to produce DataFrame df. Then, compute *time_since_last_trans* in a Python UDF using Polars by subtracting *prev_ts_transaction* from *event_time*. Apply the Python UDF to df to compute the new feature.

Feature	Prompt to write code for feature
days_to_ card_expiry	Join *card_details* with cc_trans_fg using cc_num to produce DataFrame df. Then, compute *days_to_card_expiry* in a Pandas UDF by subtracting *event_time* from *expiry_date*. Apply the Pandas UDF to df to compute the new feature.

The features are a mix of simple features (copied directly from the source table), some computed using map functions (*days_since_credit_rating_changed*, *days_until_expired*), and a lot of features that require maintaining state across data transformations, such as those that summarize observed events over windows of time (like an hour, minute, or day). In particular, all the features computed for the *cc_trans_aggs_fg* feature group require stateful data transformations. In Chapter 9, we will look at how to implement these model-independent data transformations in streaming feature pipelines.

There are many other data transformations that are not included in our credit card example system. Some examples of useful prompts that you can use to create data transformations are shown in Table 6-4.

Table 6-3. Prompts that produce data transformation code to create features.

Transformation	Example LLM Prompt
Data Cleaning	Write code to remove missing values for column Y in DataFrame df.
Filtering	Write code to read columns a, b, c from the data source into a DataFrame. Filter out rows where the *event_time* is greater than Jan 1st 2025.
Grouped Aggregations	Write code to group the rows in DataFrame df by column X and then compute the min/max/average/ median/standard deviation for column Y
Binning	Write code to compute 10 bins for the numerical feature Y in DataFrame df, where each bin should have a roughly equivalent number of entries.
Left Join	Write code to join DataFrames df1 and df2 using *merchant_id* as the join key. df1 should have more rows, and if there is no matching join key in df2, include nulls for the missing column values.
Inner Join	Write code to join DataFrames df1 and df2 using *merchant_id* as the join key. If there is no matching join key in df2 for a row in df1, do not include that row in the output DataFrame.
Lagged Feature	Write code to create a lagged feature, pm25_1day, for pm25 in the air_quality DataFrame.
Temporal Feature Extraction	Write code to extract the date from the *event_time* column in DataFrame df. The date format should be 'YYYY-MM-DD'.
Data Compression	Write code that downloads the *sentence_transformers* embedding model from Hugging Face, and uses it to encode the string column Y in DataFrame df.

Sometimes the code generated has bugs. For example, sometimes GPT-4o hallucinates that Polars DataFrames support the widely used Pandas DataFrame *apply* function, used to apply a UDF to the DataFrame. When I get errors, I paste the error log into my LLM's prompt and ask it to fix the bug. Generally, this works. But you still need to understand the code produced. Ultimately, you sign off on the code being correct. For this reason, unit testing your feature functions becomes even more criti-

cal. Again, I use LLMs to generate the unit tests for the feature functions I write. Again, I inspect the generated unit tests for correctness before I incorporate them.

Composition of Transformations

In batch pipelines, we often compute aggregations (such as min, max, mean, median, standard deviation) over a window of time, such as an hour or a day. Often more than one time window contains useful predictive signals for models. For example, we could compute aggregates once per day, but also trailing 7-day and trailing 30-day aggregates, as shown in Figure 6-8.

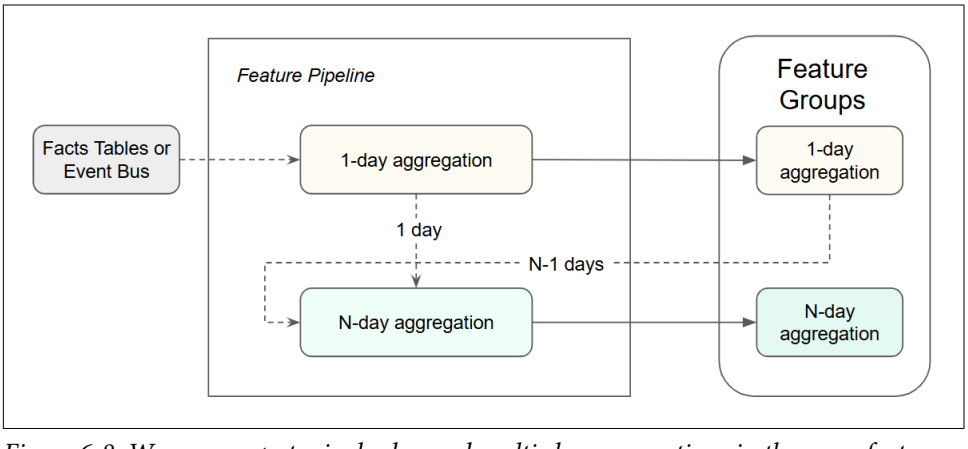


Figure 6-8. We can compute single-day and multi-day aggregations in the same feature pipeline. Multi-day aggregations combine the current daily aggregation with the historical daily aggregations read from the feature store.

Ideally, we should compute the larger windows (30-day, 7-day) from the smallest window (1-day), reducing the amount of work needed to compute aggregations. Table 6-4 shows how to compute popular aggregations for larger windows from smaller windows.

Table 6-4. Most multi-day aggregations can be computed from 1-day aggregations, resulting in large computation savings. Sometimes, however, they need additional state to be computed.

Aggregation	How to compute 7-day aggregations from 1-day aggregations
count	Sum the previous 7 days together.
sum	Sum the previous 7 days together.
max/min	Get the max/min over all the previous 7 days.
standard deviation	We need to compute and store additional daily data. For each day, we also need the count of records. Then, we can compute the 7-day aggregate using the sum of squares.

Aggregation	How to compute 7-day aggregations from 1-day aggregations
mean	We need to compute and store additional daily data. For each day, we also need the count of records. Then, we can compute the 7-day aggregate as a weighted mean.
approxQuantile	We need to compute and store complete sorted lists of daily values. With approximate summaries like T-Digests or histograms, 7-day quantiles can be approximated by merging daily distributions.
distinct count	For an accurate result, we need to store the unique values for each day and perform a set union. Approximate answers are possible with *HyperLogLog* (memory efficient, but worst accuracy) or *Bitmap/Bloom Filters* (moderate memory efficiency, better accuracy)

For example, in PySpark, we can compute a multi-day mean using the weighted man approach. The sum-of-squares is an alternative approach we could have used, but it requires an additional column storing the sum of squares, so we prefer the weighted mean approach as it requires one less column to store in our daily aggregations feature group. The PySpark code looks as follows:

```
def compute_mean(days):
    window_spec =
        Window.partitionBy("user_id").orderBy("date").rowsBetween(-days, 0)
        df = df.withColumn(f"{days}d_avg",
        F.sum(F.col("daily_mean") * F.col("daily_count")).over(window_spec) /
        F.sum("daily_count").over(window_spec)
```

Summary

In this chapter, we introduced some fundamentals for writing model-independent transformations in feature pipelines. There were extensive preliminaries on how to organize the source code for your system in a monrepo, what the common data sources for feature pipelines are, and the data types you need to work with when writing feature pipelines. We looked at how we should wrap data transformations in feature functions to make features testable and easier to maintain. We finally introduced examples of model-independent data transformations for our credit card fraud system, including binning for categorical data, mapping functions, RFM features, and aggregations.

Exercises

- You are tasked with developing a credit card fraud detection AI system. The credit card issuer estimates that there will be at most 50k transactions per day for the current year, growing to at most 100k transactions per day for the next 2 years. You have 12 months of historical transaction data. Your team does not have a strong data engineering background. Your data mart tables are stored on Iceberg on S3. Motivate which data engineering framework you would choose for writing your batch feature pipelines?

- Answer the previous question again, but this time when data volumes are 10 million transactions per day.
- Assume you have a new column *email* in the *account_details* table. Use a LLM to help write a feature function that transforms an email address into a numerical feature that represents the quality of the email address. Hint, use a LLM and tell it to use the *email-validator* Python library and tell it to use the email address domain name to help determine the "score" for the email address.

About the Author

Jim Dowling is CEO of Hopsworks and an Associate Professor at KTH Royal Institute of Technology. He's led the development of Hopsworks that includes the first open-source feature store for machine learning. He has a unique background in the intersection of data and AI. For data, he worked at MySQL and later led the development of HopsFS, a distributed file system that won the IEEE Scale Prize in 2017. For AI, his PhD introduced Collaborative Reinforcement Learning, and he developed and taught the first course on Deep Learning in Sweden in 2016. He also released a popular online course on serverless machine learning using Python at *serverless-ml.org*. This combined background of Data and AI helped him realize the vision of a feature store for machine learning based on general purpose programming languages, rather than the earlier feature store work at Uber on DSLs. He was the first evangelist for feature stores, helping to create the feature store product category through talks at industry conferences, like Data/AI Summit, PyData, OSDC, and educational articles on feature stores. He is the organizer of the annual feature store summit conference and the featurestore.org community, as well as co-organizer of PyData Stockholm.