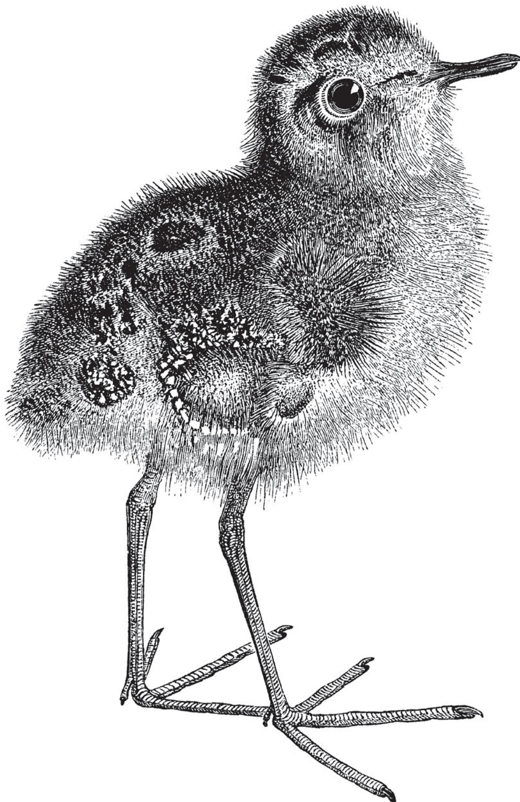
O'REILLY®

Building Machine Learning Systems with a Feature Store

Batch, Real-Time, and LLM Systems



Early Release

Raw & Unedited

Compliments of

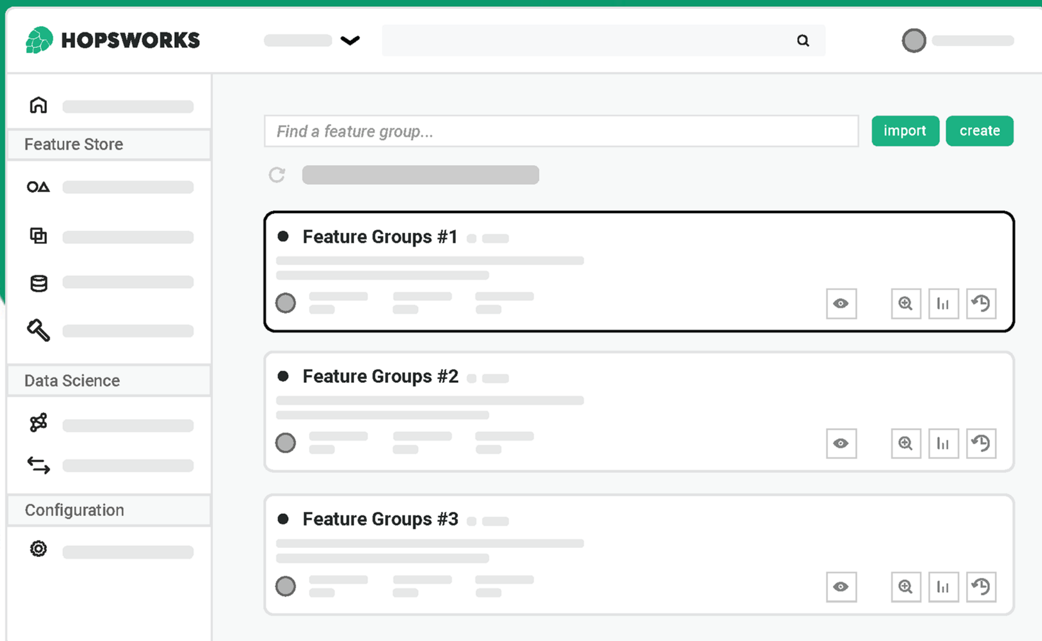
HOPSWORKS

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

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Building Machine Learning Systems with a Feature Store
by Jim Dowling

Copyright © 2025 O’Reilly Media, Inc. All rights reserved.

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
2024-02-07: First Release

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

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Building Machine Learning Systems with a Feature Store*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher’s views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O’Reilly and Hopsworks. See our statement of editorial independence.

978-1-098-16517-8

[TO COME]

Table of Contents

Brief Table of Contents (*Not Yet Final*) vii

Preface ix

Introduction xiii

1. Building Machine Learning Systems 21

The Evolution of Machine Learning Systems 23
The Anatomy of a Machine Learning System 26
  Types of Machine Learning 27
  Data Sources 29
  Incremental Datasets 32
What is a ML Pipeline ? 34
  Principles of MLOps 37
Machine Learning Systems with a Feature Store 41
  Three Types of ML System with a Feature Store 42
  ML Frameworks and ML Infrastructure used in this book 44
Summary 45

Brief Table of Contents (*Not Yet Final*)

Preface

Introduction

Chapter 1: Building Machine Learning Systems

Chapter 2: ML Pipelines (unavailable)

Chapter 3: Build an Air Quality Prediction ML System (unavailable)

Chapter 4: Feature Stores for ML (unavailable)

Chapter 5: Hopsworks Feature Store (unavailable)

Chapter 6: Model-Independent Transformations (unavailable)

Chapter 7: Model-Dependent Transformations (unavailable)

Chapter 8: Batch Feature Pipelines (unavailable)

Chapter 9: Streaming Feature Pipelines (unavailable)

Chapter 10: Training Pipelines (unavailable)

Chapter 11: Inference Pipelines (unavailable)

Chapter 12: MLOps (unavailable)

Chapter 13: Feature and Model Monitoring (unavailable)

Chapter 14: Vector Databases (unavailable)

Chapter 15: Case Study: Personalized Recommendations (unavailable)

Preface

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

Prediction Problem	Data Source(s)
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. Please note that the GitHub repo will be made active later on.

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.

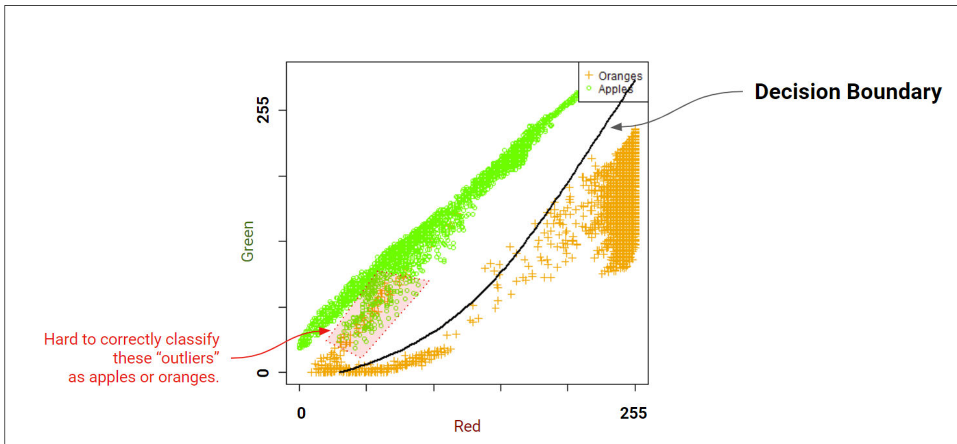


Figure I-2. When we plot all of our example apples and oranges using the observed values for the red and green color channels, we can see that most apples are on the left of the decision boundary, and most oranges are on the right. Some apples and oranges are, however, difficult to differentiate based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

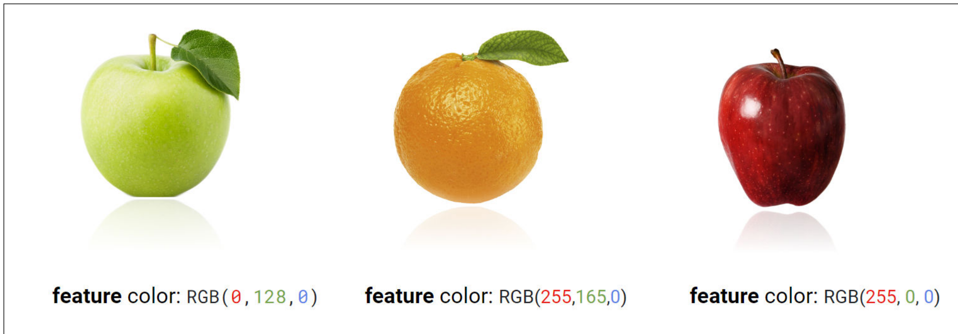


Figure I-3. The red apple complicates our prediction problem because there is no longer a linear decision boundary between the apples and oranges using only color as a feature.

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.

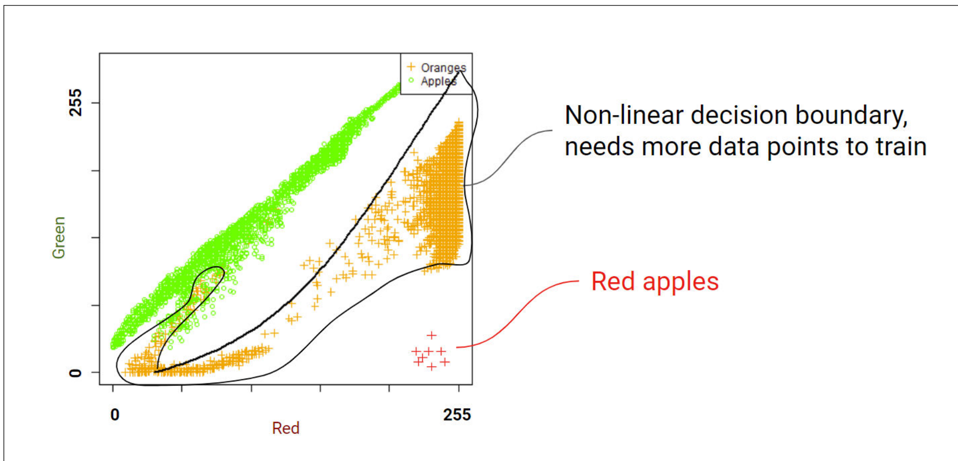


Figure I-4. When we add red apples to our training examples, we can see that we can no longer use a straight line to classify fruit as orange or apple. We now need a non-linear decision boundary to separate apples from oranges, and in order to learn the decision boundary, we need a more complex model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

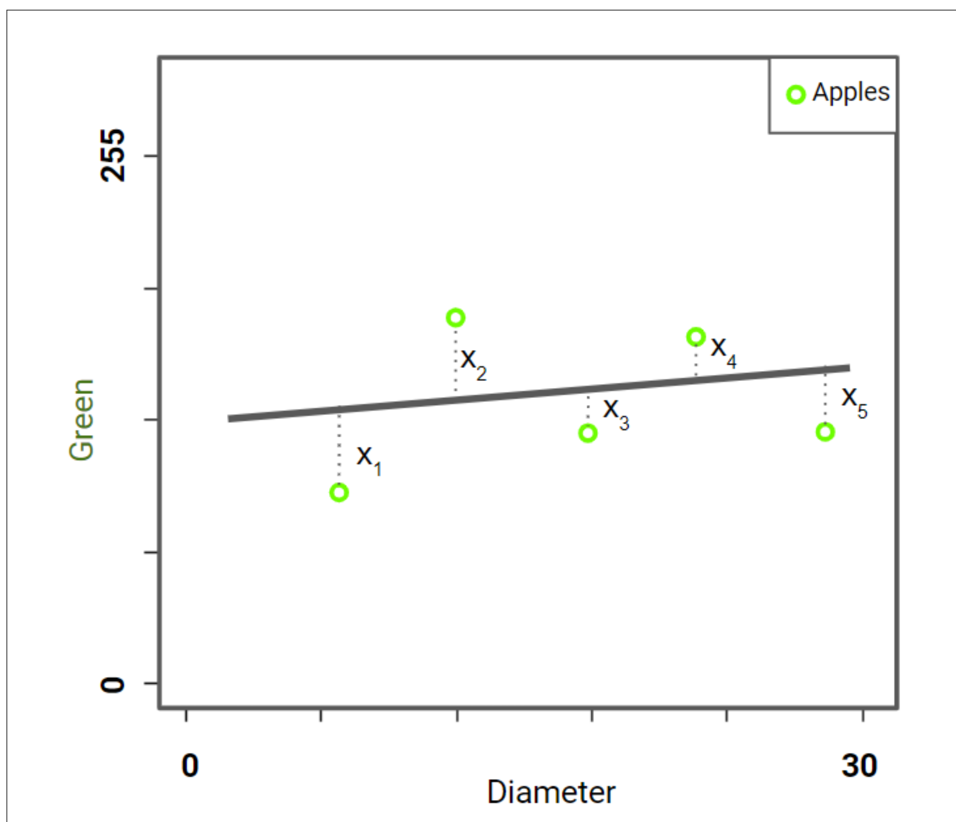


Figure I-5. This regression problem of predicting the weight of an apple can be solved using a linear model that minimizes the mean-squared error

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

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.

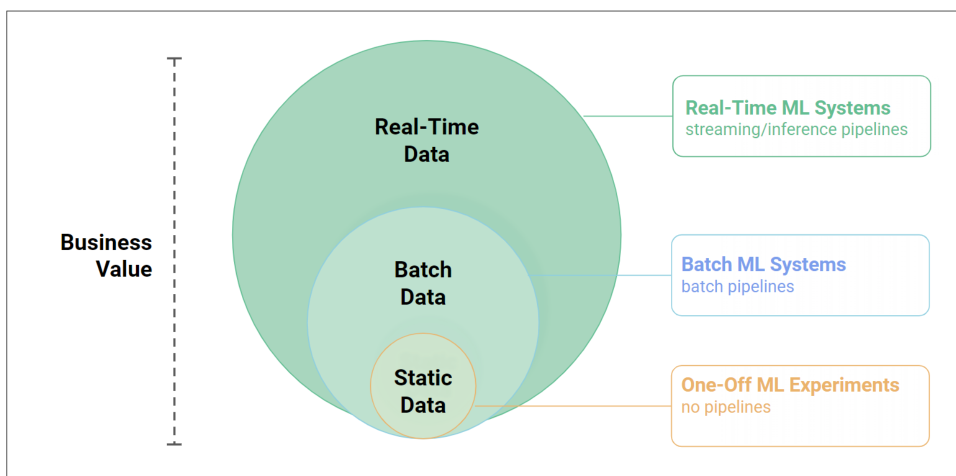


Figure 1-1. A ML system that only generates a one-off prediction on a static dataset generates less business value than a ML system that can make predictions on a schedule with batches of input data. ML systems that can make predictions with real-time data are more technically challenging, but can create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

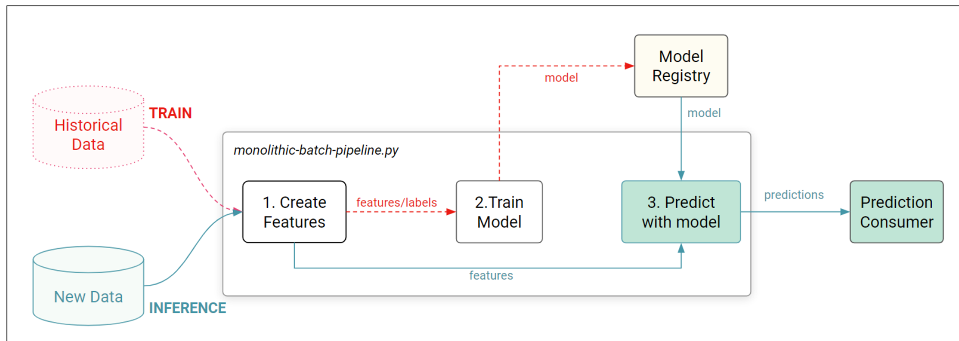


Figure 1-2. A monolithic batch ML system that can run in either (1) training mode or (2) inference mode.

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.

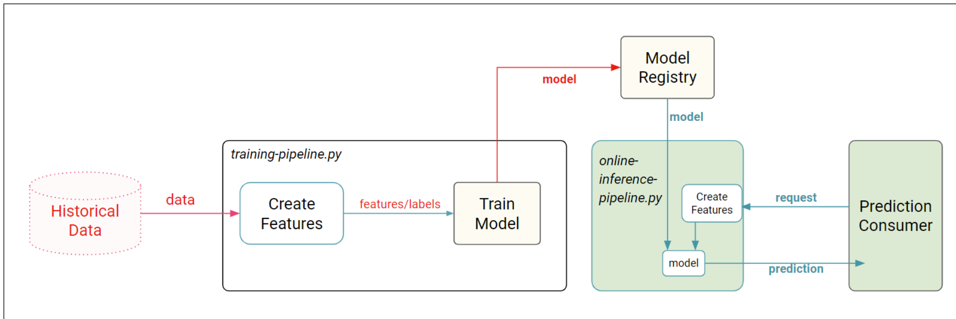


Figure 1-3. A (real-time) interactive ML system requires a separate offline training system from the online inference systems.

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.

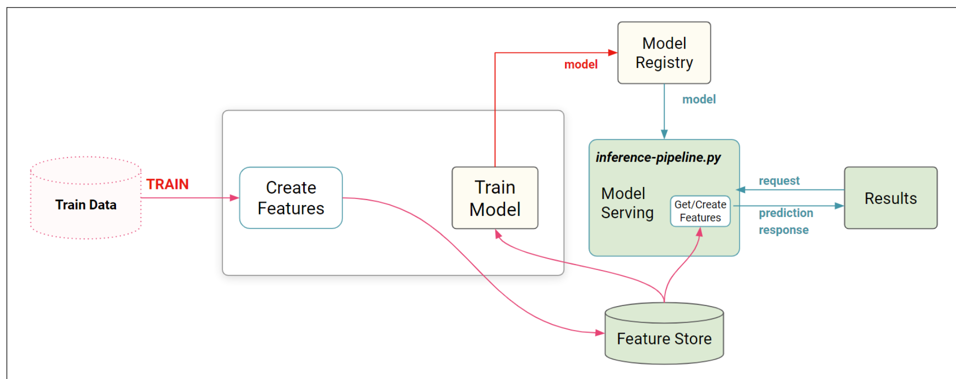


Figure 1-4. Many (real-time) interactive ML systems also require history and context to make personalized predictions. The feature store enables personalized history and context to be retrieved at low latency as precomputed features for online models.

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

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."

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.

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;

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

- 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, 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 for-

mats: 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 plat-

forms (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;

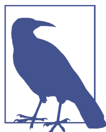
test.csv
: the test set you should use to evaluate the performance of your trained model.

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

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.

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 available 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.

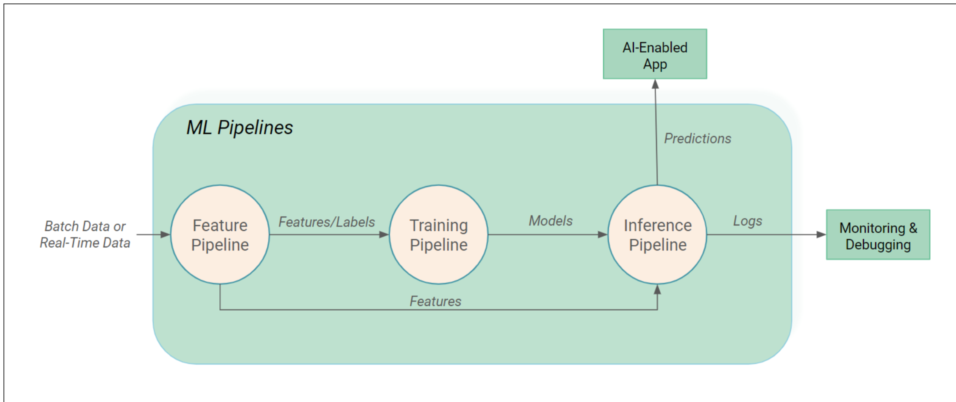


Figure 1-5. A ML pipeline has well-defined inputs and outputs. The outputs of ML pipelines can be inputs to other ML pipelines or to external ML Systems that use the predictions and prediction logs to make them "AI-enabled".

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.

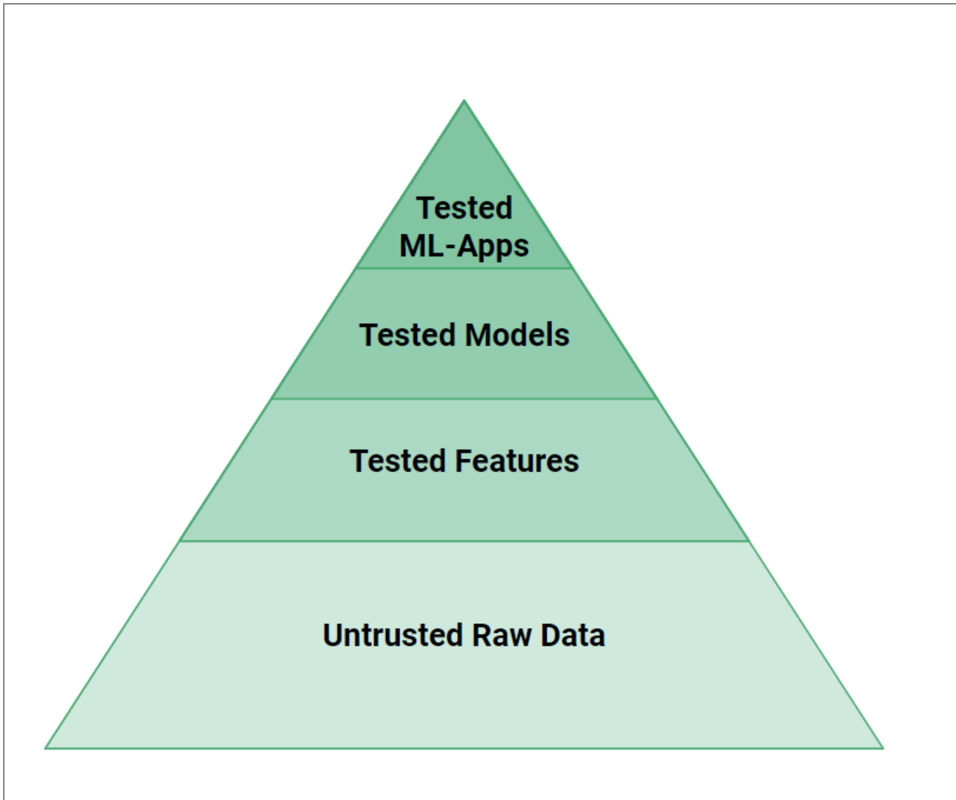


Figure 1-6. The testing pyramid for ML Systems is higher than traditional software systems, as both code and data need to be tested, not just code.

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.

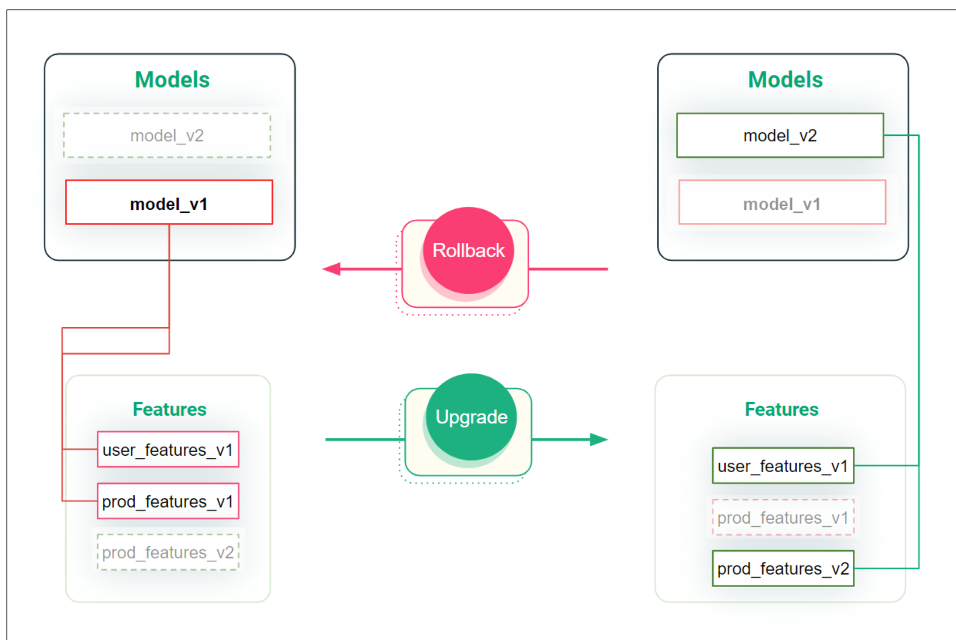


Figure 1-7. Versioning of features and models is needed to be able to easily upgrade ML systems and rollback upgrades in case of failure.

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.

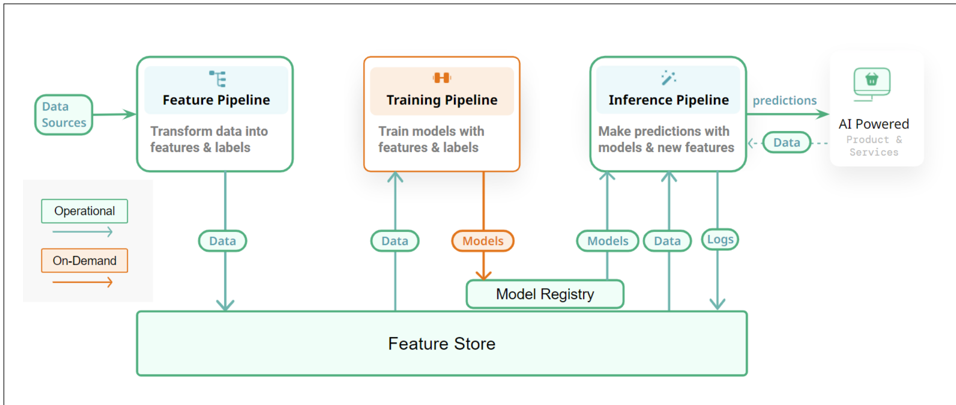


Figure 1-8. A ML system with a feature store supports 3 different types of ML pipeline: a feature pipeline, a training pipeline, and inference pipeline. Logging pipelines help implement observability for ML systems.

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.

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.