

Data Science documentation in Microsoft Fabric

Learn more about the Data Science and AI experience in Microsoft Fabric

About Data Science



OVERVIEW

[What is Data Science?](#)

Copilot



OVERVIEW

[Overview](#)

[Chat-magics in Notebooks](#)



HOW-TO GUIDE

[Use the Copilot chat panel](#)

Fabric data agent (preview)



OVERVIEW

[What is a Fabric data agent \(preview\)?](#)



GET STARTED

[Fabric data agent over Adventure Works](#)

[Get started with Fabric data agents](#)

Get started

GET STARTED

[Introduction to Data science tutorials](#)

[Data science roles and permissions](#)

[Machine learning models](#)

Tutorials

TUTORIAL

[How to use end-to-end AI samples](#)

[Train and evaluate a time series forecasting model](#)

[Create, evaluate, and score a churn prediction model](#)

Prepare data

HOW-TO GUIDE

[How to accelerate data prep with Data Wrangler](#)

[How to read and write data with Pandas](#)

Train models

OVERVIEW

[Training Overview](#)

HOW-TO GUIDE

[Train with Spark MLlib](#)

[Train models with Scikit-learn](#)

[Train models with PyTorch](#)

Track models and experiments

 **HOW-TO GUIDE**

[Machine learning experiments](#)

[Fabric autologging](#)

AI with SynapseML

 **GET STARTED**

[Your First SynapseML Model](#)

[Install another version of SynapseML](#)

 **HOW-TO GUIDE**

[Use OpenAI for big data with SynapseML](#)

[Use AI Services with SynapseML](#)

Use semantic link

 **OVERVIEW**

[What is semantic link?](#)

 **HOW-TO GUIDE**

[Read from Power BI and write data consumable by Power BI](#)

[Detect, explore, and validate functional dependencies in your data](#)

[Explore and validate relationships in semantic models](#)

Reference docs

 **REFERENCE**

[SemPy Python SDK](#)

What is Data Science in Microsoft Fabric?

Article • 11/15/2023

Microsoft Fabric offers Data Science experiences to empower users to complete end-to-end data science workflows for the purpose of data enrichment and business insights. You can complete a wide range of activities across the entire data science process, all the way from data exploration, preparation and cleansing to experimentation, modeling, model scoring and serving of predictive insights to BI reports.

Microsoft Fabric users can access a Data Science Home page. From there, they can discover and access various relevant resources. For example, they can create machine learning Experiments, Models and Notebooks. They can also import existing Notebooks on the Data Science Home page.

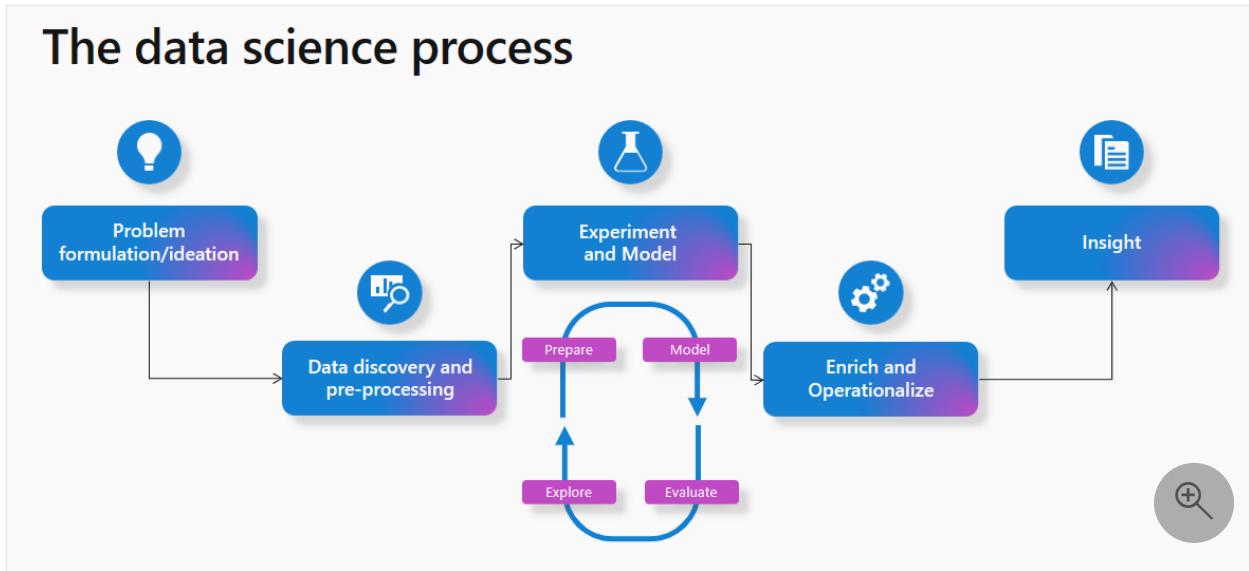
The screenshot shows the Microsoft Fabric Data Science Home page. At the top, it says "New" and "Current workspace: my_workspace". It indicates that items will be saved to this workspace. Below this, there are six buttons for creating new items: "ML model", "Experiment", "Notebook", "Environment (Preview)", "Import notebook", and "Use a sample". Under the heading "Recommended", there are three cards for "Getting Started with Data Science": "Getting Started with ML Models" (illustrated with a cloud icon), "Getting Started with ML Experiments" (illustrated with a laptop and charts), and "Getting Started with Notebooks" (illustrated with an open book and 3D cubes). A magnifying glass icon is in the bottom right corner of the recommended section.

You might know how a typical data science process works. As a well-known process, most machine learning projects follow it.

At a high level, the process involves these steps:

- Problem formulation and ideation
- Data discovery and pre-processing
- Experimentation and modeling
- Enrich and operationalize

- Gain insights



This article describes the Microsoft Fabric Data Science capabilities from a data science process perspective. For each step in the data science process, this article summarizes the Microsoft Fabric capabilities that can help.

Problem formulation and ideation

Data Science users in Microsoft Fabric work on the same platform as business users and analysts. Data sharing and collaboration becomes more seamless across different roles as a result. Analysts can easily share Power BI reports and datasets with data science practitioners. The ease of collaboration across roles in Microsoft Fabric makes hand-offs during the problem formulation phase much easier.

Data discovery and pre-processing

Microsoft Fabric users can interact with data in OneLake using the Lakehouse item. Lakehouse easily attaches to a Notebook to browse and interact with data.

Users can easily read data from a Lakehouse directly into a Pandas dataframe. For exploration, this makes seamless data reads from OneLake possible.

A powerful set of tools is available for data ingestion and data orchestration pipelines with data integration pipelines - a natively integrated part of Microsoft Fabric. Easy-to-build data pipelines can access and transform the data into a format that machine learning can consume.

Data exploration

An important part of the machine learning process is to understand data through exploration and visualization.

Depending on the data storage location, Microsoft Fabric offers a set of different tools to explore and prepare the data for analytics and machine learning. Notebooks become one of the quickest ways to get started with data exploration.

Apache Spark and Python for data preparation

Microsoft Fabric offers capabilities to transform, prepare, and explore your data at scale. With Spark, users can leverage PySpark/Python, Scala, and SparkR/SparklyR tools for data pre-processing at scale. Powerful open-source visualization libraries can enhance the data exploration experience to help better understand the data.

Data Wrangler for seamless data cleansing

The Microsoft Fabric Notebook experience added a feature to use Data Wrangler, a code tool that prepares data and generates Python code. This experience makes it easy to accelerate tedious and mundane tasks - for example, data cleansing, and build repeatability and automation through generated code. Learn more about Data Wrangler in the Data Wrangler section of this document.

Experimentation and ML modeling

With tools like PySpark/Python, SparklyR/R, notebooks can handle machine learning model training.

ML algorithms and libraries can help train machine learning models. Library management tools can install these libraries and algorithms. Users have therefore the option to leverage a large variety of popular machine learning libraries to complete their ML model training in Microsoft Fabric.

Additionally, popular libraries like Scikit Learn can also develop models.

MLflow experiments and runs can track the ML model training. Microsoft Fabric offers a built-in MLflow experience with which users can interact, to log experiments and models. Learn more about how to use MLflow to track experiments and manage models in Microsoft Fabric.

SynapseML

The SynapseML (previously known as MMLSpark) open-source library, that Microsoft owns and maintains, simplifies massively scalable machine learning pipeline creation. As a tool ecosystem, it expands the Apache Spark framework in several new directions. SynapseML unifies several existing machine learning frameworks and new Microsoft algorithms into a single, scalable API. The open-source SynapseML library includes a rich ecosystem of ML tools for development of predictive models, as well as leveraging pre-trained AI models from Azure AI services. Learn more about [SynapseML](#).

Enrich and operationalize

Notebooks can handle machine learning model batch scoring with open-source libraries for prediction, or the Microsoft Fabric scalable universal Spark Predict function, which supports MLflow packaged models in the Microsoft Fabric model registry.

Gain insights

In Microsoft Fabric, Predicted values can easily be written to OneLake, and seamlessly consumed from Power BI reports, with the Power BI Direct Lake mode. This makes it very easy for data science practitioners to share results from their work with stakeholders and it also simplifies operationalization.

Notebooks that contain batch scoring can be scheduled to run using the Notebook scheduling capabilities. Batch scoring can also be scheduled as part of data pipeline activities or Spark jobs. Power BI automatically gets the latest predictions without need for loading or refresh of the data, thanks to the Direct lake mode in Microsoft Fabric.

Data exploration with semantic link (preview)

Important

This feature is in **preview**.

Data scientists and business analysts spend lots of time trying to understand, clean, and transform data before they can start any meaningful analysis. Business analysts typically work with semantic models and encode their domain knowledge and business logic into Power BI measures. On the other hand, data scientists can work with the same data, but typically in a different code environment or language.

Semantic link (preview) allows data scientists to establish a connection between Power BI semantic models and the Synapse Data Science in Microsoft Fabric experience via the

[SemPy Python library](#). SemPy simplifies data analytics by capturing and leveraging data semantics as users perform various transformations on the semantic models. By leveraging semantic link, data scientists can:

- avoid the need to re-implement business logic and domain knowledge in their code
- easily access and use Power BI measures in their code
- use semantics to power new experiences, such as semantic functions
- explore and validate functional dependencies and relationships between data

Through the use of SemPy, organizations can expect to see:

- increased productivity and faster collaboration across teams that operate on the same datasets
- increased cross-collaboration across business intelligence and AI teams
- reduced ambiguity and an easier learning curve when onboarding onto a new model or dataset

For more information on semantic link, see [What is semantic link \(preview\)?](#).

Next steps

- Get started with end-to-end data science samples, see [Data Science Tutorials](#)
- Learn more about data preparation and cleansing with Data Wrangler, see [Data Wrangler](#)
- Learn more about tracking experiments, see [Machine learning experiment](#)
- Learn more about managing models, see [Machine learning model](#)
- Learn more about batch scoring with Predict, see [Score models with PREDICT](#)
- Serve predictions from Lakehouse to Power BI with [Direct lake Mode](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Ask the community ↗](#)

What is SynapseML?

Article • 12/20/2023

SynapseML (previously known as MMLSpark), is an open-source library that simplifies the creation of massively scalable machine learning (ML) pipelines. SynapseML provides simple, composable, and distributed APIs for a wide variety of different machine learning tasks such as text analytics, vision, anomaly detection, and many others.

SynapseML is built on the [Apache Spark distributed computing framework](#) and shares the same API as the [SparkML/MLLib library](#), allowing you to seamlessly embed SynapseML models into existing Apache Spark workflows.

With SynapseML, you can build scalable and intelligent systems to solve challenges in domains such as anomaly detection, computer vision, deep learning, text analytics, and others. SynapseML can train and evaluate models on single-node, multi-node, and elastically resizable clusters of computers. This lets you scale your work without wasting resources. SynapseML is usable across Python, R, Scala, Java, and .NET. Furthermore, its API abstracts over a wide variety of databases, file systems, and cloud data stores to simplify experiments no matter where data is located.

Installation

There are several ways to use SynapseML. Choose your desired method from the list on the [installation page](#) and follow the steps accordingly.

Visit this page for an [introduction tutorial](#) for creating your first pipeline.

Key features of SynapseML

SynapseML offers easy integration and pre trained resources to help you better understand and apply data to your business needs. SynapseML unifies several existing ML frameworks and new Microsoft algorithms in a single, scalable API that's usable across Python, R, Scala, and Java. SynapseML also helps developers understand model predictions by introducing new tools to reveal why models make certain predictions and how to improve the training dataset to eliminate biases.

A unified API for creating, training, and scoring models

SynapseML offers a unified API that simplifies developing fault-tolerant distributed programs. In particular, SynapseML exposes many different machine learning

frameworks under a single API that is scalable, data and language agnostic, and works for batch, streaming, and serving applications.

A unified API standardizes many tools, frameworks, algorithms and streamlines the distributed machine learning experience. It enables developers to quickly compose disparate machine learning frameworks, keeps code clean, and enables workflows that require more than one framework. For example, workflows such as web-supervised learning or search-engine creation require multiple services and frameworks. SynapseML shields users from this extra complexity.

Use pre-built intelligent models

Many tools in SynapseML don't require a large labeled training dataset. Instead, SynapseML provides simple APIs for pre-built intelligent services, such as Azure AI services, to quickly solve large-scale AI challenges related to both business and research. SynapseML enables developers to embed over 50 different state-of-the-art ML services directly into their systems and databases. These ready-to-use algorithms can parse a wide variety of documents, transcribe multi-speaker conversations in real time, and translate text to over 100 different languages. For more examples of how to use pre-built AI to solve tasks quickly, see [the SynapseML "cognitive" examples ↗](#).

To make SynapseML's integration with Azure AI services fast and efficient SynapseML introduces many optimizations for service-oriented workflows. In particular, SynapseML automatically parses common throttling responses to ensure that jobs don't overwhelm backend services. Additionally, it uses exponential back-offs to handle unreliable network connections and failed responses. Finally, Spark's worker machines stay busy with new asynchronous parallelism primitives for Spark. Asynchronous parallelism allows worker machines to send requests while waiting on a response from the server and can yield a tenfold increase in throughput.

Broad ecosystem compatibility with ONNX

SynapseML enables developers to use models from many different ML ecosystems through the Open Neural Network Exchange (ONNX) framework. With this integration, you can execute a wide variety of classical and deep learning models at scale with only a few lines of code. SynapseML automatically handles distributing ONNX models to worker nodes, batching and buffering input data for high throughput, and scheduling work on hardware accelerators.

Bringing ONNX to Spark not only helps developers scale deep learning models, it also enables distributed inference across a wide variety of ML ecosystems. In particular,

ONNXMLTools converts models from TensorFlow, scikit-learn, Core ML, LightGBM, XGBoost, H2O, and PyTorch to ONNX for accelerated and distributed inference using SynapseML.

Build responsible AI systems

After building a model, it's imperative that researchers and engineers understand its limitations and behavior before deployment. SynapseML helps developers and researchers build responsible AI systems by introducing new tools that reveal why models make certain predictions and how to improve the training dataset to eliminate biases. SynapseML dramatically speeds the process of understanding a user's trained model by enabling developers to distribute computation across hundreds of machines. More specifically, SynapseML includes distributed implementations of Shapley Additive Explanations (SHAP) and Locally Interpretable Model-Agnostic Explanations (LIME) to explain the predictions of vision, text, and tabular models. It also includes tools such as Individual Conditional Expectation (ICE) and partial dependence analysis to recognize biased datasets.

Enterprise support on Azure Synapse Analytics

SynapseML is generally available on Azure Synapse Analytics with enterprise support. You can build large-scale machine learning pipelines using Azure AI services, LightGBM, ONNX, and other [selected SynapseML features](#). It even includes templates to quickly prototype distributed machine learning systems, such as visual search engines, predictive maintenance pipelines, document translation, and more.

Next steps

- To learn more about SynapseML, see the [blog post](#).
- [Install SynapseML and get started with examples](#).
- [SynapseML GitHub repository](#).

Feedback

Was this page helpful?



[Provide product feedback](#) | [Ask the community](#)

Data science end-to-end scenario: introduction and architecture

Article • 04/21/2025

These tutorials present a complete end-to-end scenario in the Fabric data science experience. They cover each step, from

- Data ingestion
- Data cleaning
- Data preparation

to

- Machine learning model training
- Insight generation

and then cover consumption of those insights with visualization tools - for example, Power BI.

People new to Microsoft Fabric should visit [What is Microsoft Fabric?](#).

Introduction

A data science project lifecycle typically includes these steps:

- Understand the business rules
- Acquire the data
- Explore, clean, prepare, and visualize the data
- Train the model and track the experiment
- Score the model and generate insights

The steps often proceed iteratively. The goals and success criteria of each stage depend on collaboration, data sharing, and documentation. The Fabric data science experience involves multiple native-built features that enable seamless collaboration, data acquisition, sharing, and consumption.

These tutorials place you in the role of a data scientist who must explore, clean, and transform a dataset that contains the churn status of 10,000 bank customers. You then build a machine learning model to predict which bank customers will likely leave.

You perform the following activities in the tutorials:

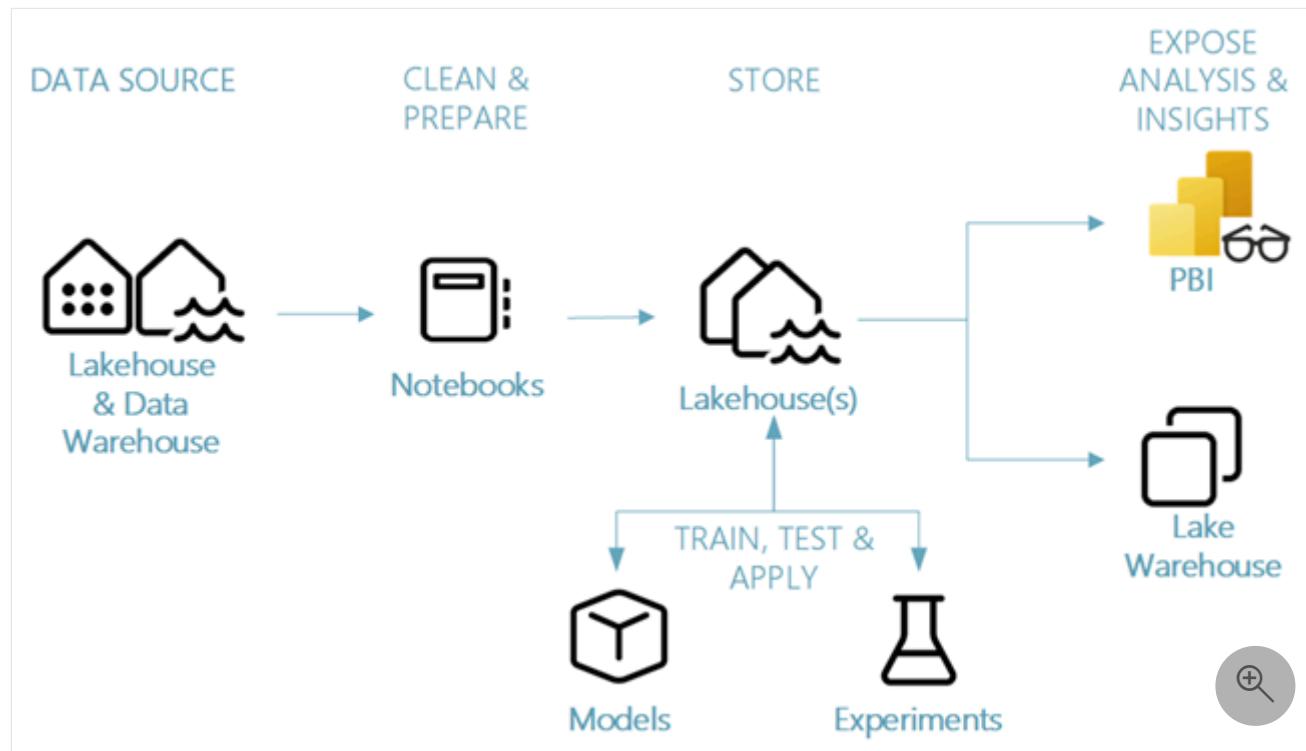
1. Use the Fabric notebooks for data science scenarios
2. Use Apache Spark to ingest data into a Fabric lakehouse

3. Load existing data from the lakehouse delta tables
4. Use Apache Spark and Python-based tools to clean and transform data
5. Create experiments and runs to train different machine learning models
6. Use MLflow and the Fabric UI to register and track trained models
7. Run scoring at scale, and save predictions and inference results to the lakehouse
8. Use DirectLake to visualize predictions in Power BI

Architecture

This tutorial series showcases a simplified end-to-end data science scenario involving:

1. Data ingestion from an external data source.
2. Data exploration and cleaning.
3. Machine learning model training and registration.
4. Batch scoring and prediction saving.
5. Prediction result visualization in Power BI.



Different components of the data science scenario

Data sources - To ingest data with Fabric, you can easily and quickly connect to Azure Data Services, other cloud platforms, and on-premises data resources. With Fabric Notebooks, you can ingest data from these resources:

- Built-in Lakehouses
- Data Warehouses

- Semantic models
- Various Apache Spark data sources
- Various data sources that support Python

This tutorial series focuses on data ingestion and loading from a lakehouse.

Explore, clean, and prepare - The Fabric data science experience supports data cleaning, transformation, exploration, and featurization. It uses built-in Spark experiences and Python-based tools - for example, Data Wrangler and SemPy Library. This tutorial showcases data exploration with the `seaborn` Python library, and data cleaning and preparation with Apache Spark.

Models and experiments - With Fabric, you can train, evaluate, and score machine learning models with built-in experiments. To register and deploy your models, and track experiments, [MLflow](#) offers seamless integration with Fabric as a way to model items. To build and share business insights, Fabric offers other features for model prediction at scale (PREDICT), to build and share business insights.

Storage - Fabric standardizes on [Delta Lake](#), which means all Fabric engines can interact with the same dataset stored in a lakehouse. With that storage layer, you can store both structured and unstructured data that support both file-based storage and tabular format. You can easily access the datasets and stored files through all Fabric experience items - for example, notebooks and pipelines.

Expose analysis and insights - Power BI, an industry-leading business intelligence tool, can consume lakehouse data for report and visualization generation. In notebook resources, Python or Spark native visualization libraries

- `matplotlib`
- `seaborn`
- `plotly`
- etc.

can visualize data persisted in a lakehouse. The SemPy library also supports data visualization. This library supports built-in rich, task-specific visualizations for

- The semantic data model
- Dependencies and their violations
- Classification and regression use cases

Next step

Prepare your system for the data science tutorial

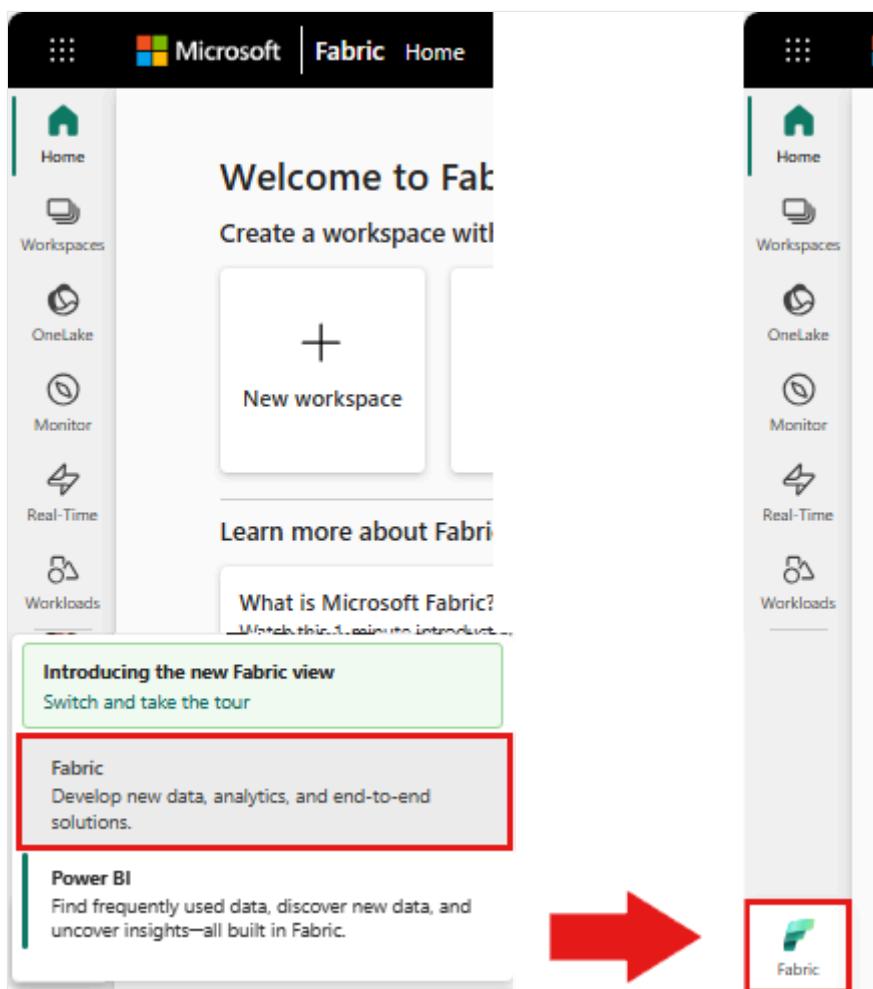
Prepare your system for data science tutorials

Article • 01/17/2025

Before you begin the data science end-to-end tutorial series, learn about prerequisites, how to import notebooks, and how to attach a lakehouse to those notebooks.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- If you don't have a Microsoft Fabric lakehouse, create one by following the steps in [Create a lakehouse in Microsoft Fabric](#).

Create a notebook

Each tutorial is available as a Jupyter notebook file in GitHub. Many of the tutorials are also available as samples in the Data Science workload. Use one of the following methods to access the tutorials:

- Create a new notebook, then copy and paste the code from the tutorial.
- Open the sample notebook (when available) in the Data Science workload:
 1. From the left pane, select **Workloads**.
 2. Select **Data Science**.
 3. From the **Explore a sample** card, select **Select**.
 4. Select the corresponding sample:
 - From the default **End-to-end workflows (Python)** tab, if the sample is for a Python tutorial.
 - From the **End-to-end workflows (R)** tab, if the sample is for an R tutorial.
 - From the **Quick tutorials** tab, if the sample is for a quick tutorial.
- Import the notebook from GitHub to your workspace:
 1. Download your notebook(s). Make sure to download the files by using the "Raw" file link in GitHub.
 - For the **Get started** notebooks, download the notebook(.ipynb) files from the parent folder: [data-science-tutorial](#).
 - For the **Tutorials** notebooks, download the notebooks(.ipynb) files from the parent folder [ai-samples](#).
 2. On the left navigation for the Fabric homepage, select your workspace.
 3. Select **Import > Notebook > From this computer**.

The screenshot shows the Microsoft Fabric interface for the workspace 'SG_ws'. On the left, there's a sidebar with icons for Home, Workspaces, OneLake, Monitor, Real-Time, and Workloads. A red box highlights the 'Workspaces' icon. In the center, there's a search bar with 'Search' and a 'Import' button with a dropdown menu. The dropdown menu has two options: 'Notebook' (highlighted with a red box) and 'Report or Paginated Report'. Below the dropdown, a tooltip says 'Import notebook source code files from your local drive.' To the right of the dropdown, there's a table listing four notebooks: '1-ingest-data', '2-explore-cleanse-data', '3-train-evaluate', and '4-predict', all categorized as 'Notebook' under 'Type'.

4. Select Upload and select the downloaded notebook file.

The screenshot shows the 'Import status' dialog box. It contains a message: 'Upload and import notebook from local computer, you can select multiple notebooks and import them in a batch.' Below the message is a button labeled 'Upload' with an upward arrow icon, which is highlighted with a red box and has a large red arrow pointing to it. There's also a magnifying glass icon.

5. Once the notebooks are imported, select **Go to workspace** in the import dialog box.

The screenshot shows a success dialog box with a green checkmark icon and the text 'Imported successfully'. It also says 'All files are listed in the page.' Below the message is a button labeled 'Go to workspace' which is highlighted with a red box and has a large red arrow pointing to it. There's also a magnifying glass icon.

6. The imported notebooks are now available in your workspace for use.

7. If the imported notebook includes output, select the **Edit** menu, then select **Clear all outputs**.

The screenshot shows the ribbon menu of a notebook. The 'Edit' tab is selected and highlighted with a red box. Other tabs include 'Run', 'Data', and 'View'. Below the tabs are various icons for file operations like 'AutoSave', 'Run', 'Data', 'View', and 'Edit'. A red box highlights the 'Clear all outputs' button, which has a trash bin icon. Other buttons include 'Add code cell', 'Browse code snippet', and 'Find and replace'.

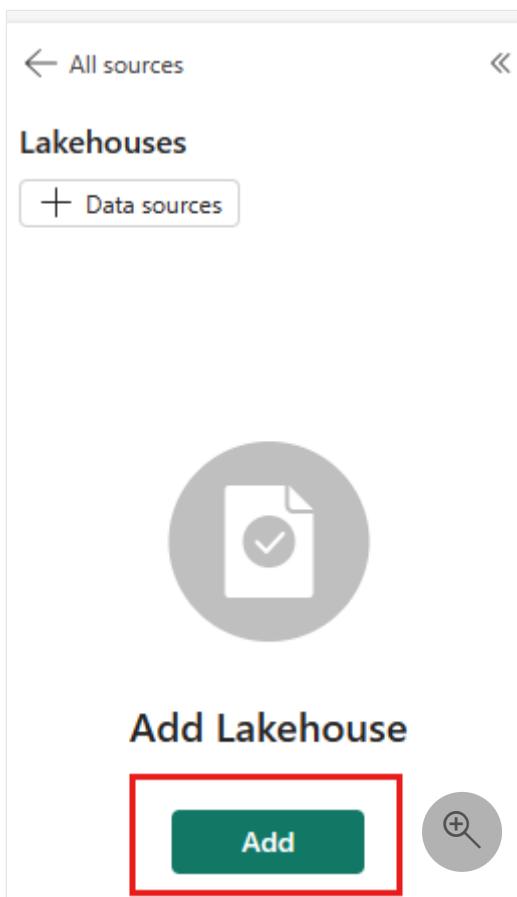
Attach a lakehouse to the notebooks

To demonstrate Fabric lakehouse features, many of the tutorials require attaching a default lakehouse to the notebooks. The following steps show how to add a lakehouse to a notebook in a Fabric-enabled workspace.

ⓘ Note

Before executing each notebook, you need to perform these steps on that notebook.

1. Open the notebook in the workspace.
2. Select **Add lakehouse** in the left pane.



3. Create a new lakehouse or use an existing lakehouse.
 - a. To create a new lakehouse, select **New**. Give the lakehouse a name and select **Create**.
 - b. To use an existing lakehouse, select **Existing lakehouse** to open the **Data hub** dialog box. Select the lakehouse you want to use and then select **Add**.
4. Once a lakehouse is added, it's visible in the lakehouse pane and you can view tables and files stored in the lakehouse.

Next step

Part 1: Ingest data into Fabric lakehouse using Apache Spark

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Tutorial Part 1: Ingest data into a Microsoft Fabric lakehouse using Apache Spark

Article • 01/22/2024

In this tutorial, you'll ingest data into Fabric lakehouses in delta lake format. Some important terms to understand:

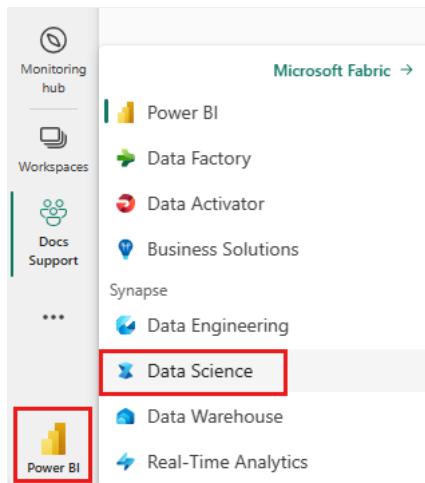
- **Lakehouse** - A lakehouse is a collection of files/folders/tables that represent a database over a data lake used by the Spark engine and SQL engine for big data processing and that includes enhanced capabilities for ACID transactions when using the open-source Delta formatted tables.
- **Delta Lake** - Delta Lake is an open-source storage layer that brings ACID transactions, scalable metadata management, and batch and streaming data processing to Apache Spark. A Delta Lake table is a data table format that extends Parquet data files with a file-based transaction log for ACID transactions and scalable metadata management.
- **Azure Open Datasets** are curated public datasets you can use to add scenario-specific features to machine learning solutions for more accurate models. Open Datasets are in the cloud on Microsoft Azure Storage and can be accessed by various methods including Apache Spark, REST API, Data factory, and other tools.

In this tutorial, you use the Apache Spark to:

- ✓ Read data from Azure Open Datasets containers.
- ✓ Write data into a Fabric lakehouse delta table.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



- Add a lakehouse to this notebook. You'll be downloading data from a public blob, then storing the data in the lakehouse.

Follow along in notebook

[1-ingest-data.ipynb](#) is the notebook that accompanies this tutorial.

To open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#), to import the notebook to your workspace.

If you'd rather copy and paste the code from this page, you can [create a new notebook](#).

Be sure to [attach a lakehouse to the notebook](#) before you start running code.

Bank churn data

The dataset contains churn status of 10,000 customers. It also includes attributes that could impact churn such as:

- Credit score
- Geographical location (Germany, France, Spain)
- Gender (male, female)
- Age
- Tenure (years of being bank's customer)
- Account balance
- Estimated salary
- Number of products that a customer has purchased through the bank
- Credit card status (whether a customer has a credit card or not)
- Active member status (whether an active bank's customer or not)

The dataset also includes columns such as row number, customer ID, and customer surname that should have no impact on customer's decision to leave the bank.

The event that defines the customer's churn is the closing of the customer's bank account. The column `Exited` in the dataset refers to customer's abandonment. There isn't much context available about these attributes so you have to proceed without having background information about the dataset. The aim is to understand how these attributes contribute to the `Exited` status.

Example rows from the dataset:

Expand table

"CustomerID"	"Surname"	"CreditScore"	"Geography"	"Gender"	"Age"	"Tenure"	"Balance"	"NumOfProducts"	"HasCrCard"	"IsActiveMembr"
15634602	Hargrave	619	France	Female	42	2	0.00	1	1	1
15647311	Hill	608	Spain	Female	41	1	83807.86	1	0	1

Download dataset and upload to lakehouse

Tip

By defining the following parameters, you can use this notebook with different datasets easily.

Python

```
IS_CUSTOM_DATA = False # if TRUE, dataset has to be uploaded manually

DATA_ROOT = "/lakehouse/default"
DATA_FOLDER = "Files/churn" # folder with data files
DATA_FILE = "churn.csv" # data file name
```

This code downloads a publicly available version of the dataset and then stores it in a Fabric lakehouse.

Important

Make sure you add a lakehouse to the notebook before running it. Failure to do so will result in an error.

Python

```
import os, requests
if not IS_CUSTOM_DATA:
    # Download demo data files into lakehouse if not exist
    remote_url = "https://synapseaisolutionsa.blob.core.windows.net/public/bankcustomerchurn"
    file_list = [DATA_FILE]
    download_path = f"{DATA_ROOT}/{DATA_FOLDER}/raw"

    if not os.path.exists("/lakehouse/default"):
        raise FileNotFoundError(
            "Default lakehouse not found, please add a lakehouse and restart the session."
        )
    os.makedirs(download_path, exist_ok=True)
    for fname in file_list:
        if not os.path.exists(f"{download_path}/{fname}"):
            r = requests.get(f"{remote_url}/{fname}", timeout=30)
            with open(f"{download_path}/{fname}", "wb") as f:
```

```
f.write(r.content)
print("Downloaded demo data files into lakehouse.")
```

Related content

You'll use the data you just ingested in:

[Part 2: Explore and visualize data using notebooks](#)

Feedback

Was this page helpful? [!\[\]\(06c63dadea3471d5a21d94122ad85b6d_img.jpg\) Yes](#) [!\[\]\(7e230521df3ad0d70cb1411ac38176b0_img.jpg\) No](#)

[Provide product feedback](#) | [Ask the community](#)

Tutorial Part 2: Explore and visualize data using Microsoft Fabric notebooks

Article • 10/20/2023

In this tutorial, you'll learn how to conduct exploratory data analysis (EDA) to examine and investigate the data while summarizing its key characteristics through the use of data visualization techniques.

You'll use `seaborn`, a Python data visualization library that provides a high-level interface for building visuals on dataframes and arrays. For more information about `seaborn`, see [Seaborn: Statistical Data Visualization](#).

You'll also use [Data Wrangler](#), a notebook-based tool that provides you with an immersive experience to conduct exploratory data analysis and cleaning.

ⓘ Important

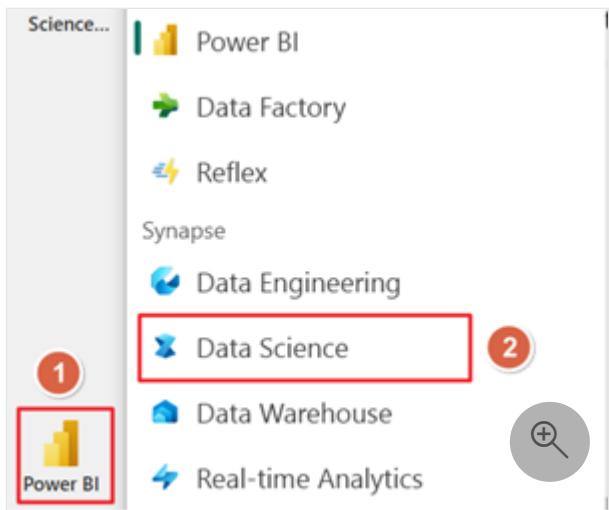
Microsoft Fabric is in [preview](#).

The main steps in this tutorial are:

1. Read the data stored from a delta table in the lakehouse.
2. Convert a Spark DataFrame to Pandas DataFrame, which python visualization libraries support.
3. Use Data Wrangler to perform initial data cleaning and transformation.
4. Perform exploratory data analysis using `seaborn`.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or sign up for a free [Microsoft Fabric \(Preview\) trial](#).
- Sign in to [Microsoft Fabric](#).
- Switch to the Data Science experience by using the experience switcher icon on the left side of your home page.



This is part 2 of 5 in the tutorial series. To complete this tutorial, first complete:

- Part 1: Ingest data into a Microsoft Fabric lakehouse using Apache Spark.

Follow along in notebook

[2-explore-cleanse-data.ipynb](#) is the notebook that accompanies this tutorial.

If you want to open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science](#) to import the tutorial notebooks to your workspace.

Or, if you'd rather copy and paste the code from this page, you can [create a new notebook](#).

Be sure to [attach a lakehouse to the notebook](#) before you start running code.

ⓘ Important

Attach the same lakehouse you used in Part 1.

Read raw data from the lakehouse

Read raw data from the **Files** section of the lakehouse. You uploaded this data in the previous notebook. Make sure you have attached the same lakehouse you used in Part 1 to this notebook before you run this code.

Python

```
df = (
    spark.read.option("header", True)
```

```
.option("inferSchema", True)
.csv("Files/churn/raw/churn.csv")
.cache()
)
```

Create a pandas DataFrame from the dataset

Convert the spark DataFrame to pandas DataFrame for easier processing and visualization.

Python

```
df = df.toPandas()
```

Display raw data

Explore the raw data with `display`, do some basic statistics and show chart views. Note that you first need to import the required libraries such as `Numpy`, `Pandas`, `Seaborn`, and `Matplotlib` for data analysis and visualization.

Python

```
import seaborn as sns
sns.set_theme(style="whitegrid", palette="tab10", rc = {'figure.figsize': (9,6)})
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
from matplotlib import rc, rcParams
import numpy as np
import pandas as pd
import itertools
```

Python

```
display(df, summary=True)
```

Use Data Wrangler to perform initial data cleaning

To explore and transform any pandas Dataframes in your notebook, launch Data Wrangler directly from the notebook.

(!) Note

Data Wrangler can not be opened while the notebook kernel is busy. The cell execution must complete prior to launching Data Wrangler.

1. Under the notebook ribbon **Data** tab, select **Launch Data Wrangler**. You'll see a list of activated pandas DataFrames available for editing.
2. Select the DataFrame you wish to open in Data Wrangler. Since this notebook only contains one DataFrame, `pd`, select `pd`.

The screenshot shows the Jupyter Notebook ribbon with the 'Data' tab highlighted. A dropdown menu is open under 'Launch Data Wrangler' with 'df' selected. The main area displays a table with columns: IsActiveMember (integer), Surname (string), and count (2792). Below the table, a section titled 'Use Data Wrangler to perform initial data clean' provides instructions for using Data Wrangler for data cleansing.

Data Wrangler launches and generates a descriptive overview of your data. The table in the middle shows each data column. The **Summary** panel next to the table shows information about the DataFrame. When you select a column in the table, the summary updates with information about the selected column. In some instances, the data displayed and summarized will be a truncated view of your DataFrame. When this happens, you'll see warning image in the summary pane. Hover over this warning to view text explaining the situation.

The screenshot shows the Data Wrangler interface for DataFrame 'df'. The left sidebar lists operations: Find and replace (4), Format (7), Formulas (4), Numeric (4), Schema (5), Sort and filter (2). The main area shows a preview of the DataFrame with columns RowNumber, CustomerId, and Surname. The 'Summary' panel on the right shows details: Data shape (5,000 rows x 14 columns), Columns (14), Rows (5,000), Rows with missing values (0.0%), Duplicate rows (0.0%), and Missing values (0). A warning icon is present in the summary panel.

Each operation you do can be applied in a matter of clicks, updating the data display in real time and generating code that you can save back to your notebook as a reusable function.

The rest of this section walks you through the steps to perform data cleaning with Data Wrangler.

Drop duplicate rows

On the left panel is a list of operations (such as **Find and replace**, **Format**, **Formulas**, **Numeric**) you can perform on the dataset.

1. Expand **Find and replace** and select **Drop duplicate rows**.

The screenshot shows the Data Wrangler interface. On the left, there's a sidebar with icons for Home, Create, Browse, OneLake data hub, Monitoring hub, Workspaces, and a workspace named SG_Happy_Path. The main area has a header with a back arrow, the title 'df (Data Wrangler)', and buttons for 'Add code to notebook', 'Copy code to clipboard', and 'Save as CSV'. Below the header is a section titled 'Operations' with a search bar. Under 'Operations', there are several categories: 'Find and replace (4)' (which is expanded and highlighted with a red box), 'Drop missing values', 'Fill missing values', 'Find and replace', 'Format (7)', 'Formulas (4)', and 'Numeric (4)'. To the right of the operations list is a preview pane showing a table with columns '# index', '# RowNumber', 'Missing:', and 'Distinct:'. The preview shows a series of blue bars for '# index' and '# RowNumber', with 'Missing:' and 'Distinct:' counts below them. Below the preview is a scrollable list of rows from 0 to 4. At the bottom of the preview pane, it says '2 New operation' and '1 Choose an operation'.

2. A panel appears for you to select the list of columns you want to compare to define a duplicate row. Select **RowNumber** and **CustomerId**.

In the middle panel is a preview of the results of this operation. Under the preview is the code to perform the operation. In this instance, the data appears to be unchanged. But since you're looking at a truncated view, it's a good idea to still apply the operation.

Operations

Drop duplicate rows

Target columns: RowNumber, CustomerId

Summary: 2048 Unique values

Cleaning steps:

```
1 # Drop duplicate rows in columns: 'RowNumber', 'CustomerId'
2 df = df.drop_duplicates(subset=['RowNumber', 'CustomerId'])
```

Data is unchanged

3. Select **Apply** (either at the side or at the bottom) to go to the next step.

Drop rows with missing data

Use Data Wrangler to drop rows with missing data across all columns.

1. Select **Drop missing values** from **Find and replace**.

2. Select **All columns** from the drop-down list.

Operations

Drop missing values

Target columns: Balance, NumOfProducts, HasCrCard, ...

Summary: 2048 Unique values

Cleaning steps:

```
1 # Drop rows with missing data across all columns
2 df = df.dropna()
```

Data is unchanged

3. Select **Apply** to go on to the next step.

Drop columns

Use Data Wrangler to drop columns that you don't need.

1. Expand **Schema** and select **Drop columns**.
2. Select **RowNumber**, **CustomerId**, **Surname**. These columns appear in red in the preview, to show they're changed by the code (in this case, dropped.)

The screenshot shows the Data Wrangler interface for a DataFrame named 'df'. On the left sidebar, under 'Operations', the 'Drop columns' option is selected. In the main area, the 'Target columns' dropdown contains 'RowNumber, CustomerId, Surname'. The preview pane shows a table with four columns: '# index', '# RowNumber', '# CustomerId', and '# Surname'. The '# RowNumber' and '# CustomerId' columns are marked with red checkmarks, indicating they are being dropped. The '# Surname' column is marked with a red question mark. Below the preview is a 'Cleaning steps' panel with a step labeled '4 Drop columns 3 columns' containing the code:

```
1 # Drop columns: 'RowNumber', 'CustomerId', 'Surname'  
2 df = df.drop(columns=['RowNumber', 'CustomerId', 'Surname'])
```

 At the bottom of the interface, there is a 'Previewing' button and 'Apply' and 'Discard' buttons.

3. Select **Apply** to go on to the next step.

Add code to notebook

Each time you select **Apply**, a new step is created in the **Cleaning steps** panel on the bottom left. At the bottom of the panel, select **Preview code for all steps** to view a combination of all the separate steps.

Select **Add code to notebook** at the top left to close Data Wrangler and add the code automatically. The **Add code to notebook** wraps the code in a function, then calls the function.

The screenshot shows the Data Wrangler interface. On the left sidebar, there are various navigation links like Home, Create, Browse, OneLake data hub, Monitoring hub, Workspaces, SG_Happy_Path, 4-predict(1), 3-train-evaluate, 2-explore-cleanse..., 1-ingest-data(1), and Data Science. The main area has a title bar with '← df (Data Wrangler)', a red box around '+ Add code to notebook', and buttons for 'Copy code to clipboard' and 'Save as CSV'. Below this is a section titled 'Operations' with a search bar and a list of operations: Find and replace (4), Format (7), Formulas (4), Numeric (4), and Schema (5). Under Schema (5), there are options for Change column type, Clone column, Drop columns, Rename column, and Select columns. A 'Cleaning steps' section follows, listing: Drop duplicate rows (2 columns), Drop missing values (14 columns), Drop columns (3 columns), and New operation. A red box highlights the 'Preview code for all steps' button. To the right, a preview of the DataFrame is shown with columns: # index, # CreditScore, Geography, and Gender. The CreditScore column has a histogram with a min of 350 and max of 850. The preview table shows 6 rows of data. Below the preview is a code editor with the following content:

```

1 # Loaded variable 'df' from kernel state
2
3 # Drop duplicate rows in columns: 'RowNumber', 'CustomerId'
4 df = df.drop_duplicates(subset=['RowNumber', 'CustomerId'])
5
6 # Drop rows with missing data across all columns
7 df = df.dropna()
8
9 # Drop columns: 'RowNumber', 'CustomerId', 'Surname'
10 df = df.drop(columns=['RowNumber', 'CustomerId', 'Surname'])

```

Below the code editor, it says 'Showing code - read only (Return to current step)'. At the bottom of the main area, it says 'Operation applied: Drop columns: 'RowNumber', 'CustomerId', 'Surname'' and 'Showing the first 5,000 rows'.

💡 Tip

The code generated by Data Wrangler won't be applied until you manually run the new cell.

If you didn't use Data Wrangler, you can instead use this next code cell.

This code is similar to the code produced by Data Wrangler, but adds in the argument `inplace=True` to each of the generated steps. By setting `inplace=True`, pandas will overwrite the original DataFrame instead of producing a new DataFrame as an output.

Python

```

# Modified version of code generated by Data Wrangler
# Modification is to add in-place=True to each step

# Define a new function that include all above Data Wrangler operations
def clean_data(df):
    # Drop rows with missing data across all columns
    df.dropna(inplace=True)
    # Drop duplicate rows in columns: 'RowNumber', 'CustomerId'
    df.drop_duplicates(subset=['RowNumber', 'CustomerId'], inplace=True)
    # Drop columns: 'RowNumber', 'CustomerId', 'Surname'

```

```
df.drop(columns=[ 'RowNumber', 'CustomerId', 'Surname'], inplace=True)
return df

df_clean = clean_data(df.copy())
df_clean.head()
```

Explore the data

Display some summaries and visualizations of the cleaned data.

Determine categorical, numerical, and target attributes

Use this code to determine categorical, numerical, and target attributes.

Python

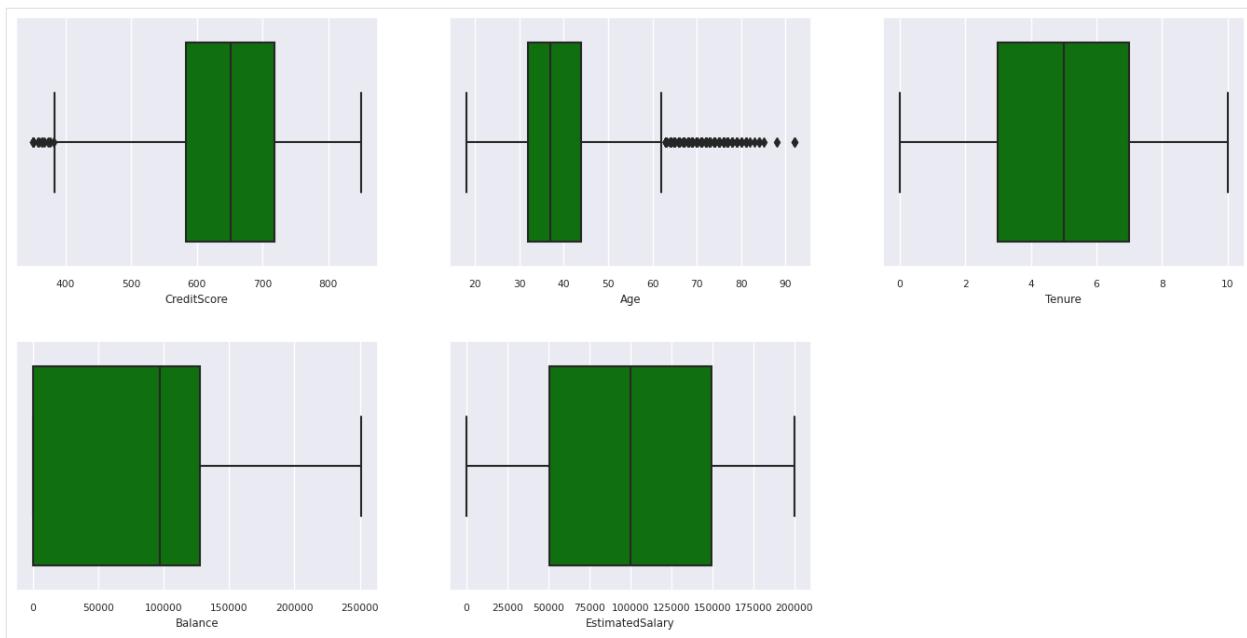
```
# Determine the dependent (target) attribute
dependent_variable_name = "Exited"
print(dependent_variable_name)
# Determine the categorical attributes
categorical_variables = [col for col in df_clean.columns if col in "0"
                           or df_clean[col].nunique() <=5
                           and col not in "Exited"]
print(categorical_variables)
# Determine the numerical attributes
numeric_variables = [col for col in df_clean.columns if df_clean[col].dtype
                     != "object"
                           and df_clean[col].nunique() >5]
print(numeric_variables)
```

The five-number summary

Show the five-number summary (the minimum score, first quartile, median, third quartile, the maximum score) for the numerical attributes, using box plots.

Python

```
df_num_cols = df_clean[numeric_variables]
sns.set(font_scale = 0.7)
fig, axes = plt.subplots(nrows = 2, ncols = 3, gridspec_kw =
dict(hspace=0.3), figsize = (17,8))
fig.tight_layout()
for ax,col in zip(axes.flatten(), df_num_cols.columns):
    sns.boxplot(x = df_num_cols[col], color='green', ax = ax)
fig.delaxes(axes[1,2])
```

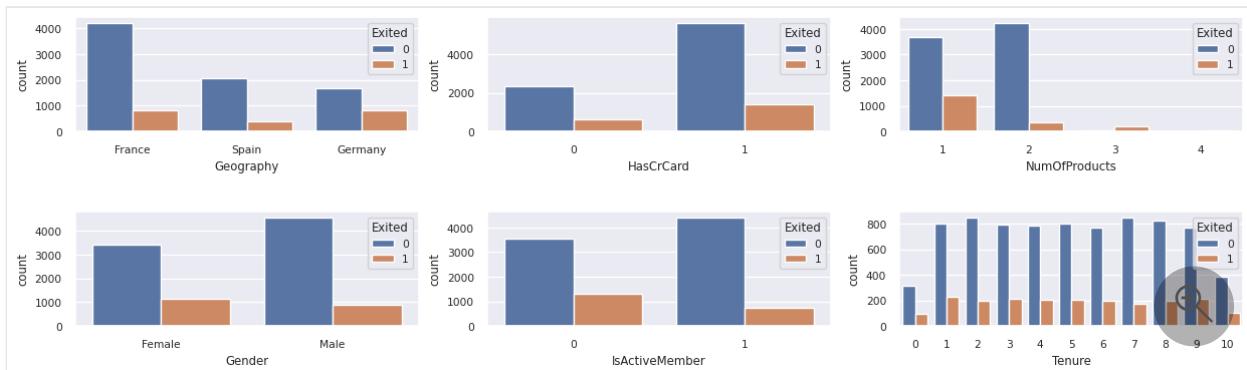


Distribution of exited and nonexited customers

Show the distribution of exited versus nonexited customers across the categorical attributes.

Python

```
attr_list = ['Geography', 'Gender', 'HasCrCard', 'IsActiveMember',
'NumOfProducts', 'Tenure']
fig, axarr = plt.subplots(2, 3, figsize=(15, 4))
for ind, item in enumerate(attr_list):
    sns.countplot(x = item, hue = 'Exited', data = df_clean, ax =
axarr[ind%2][ind//2])
fig.subplots_adjust(hspace=0.7)
```



Distribution of numerical attributes

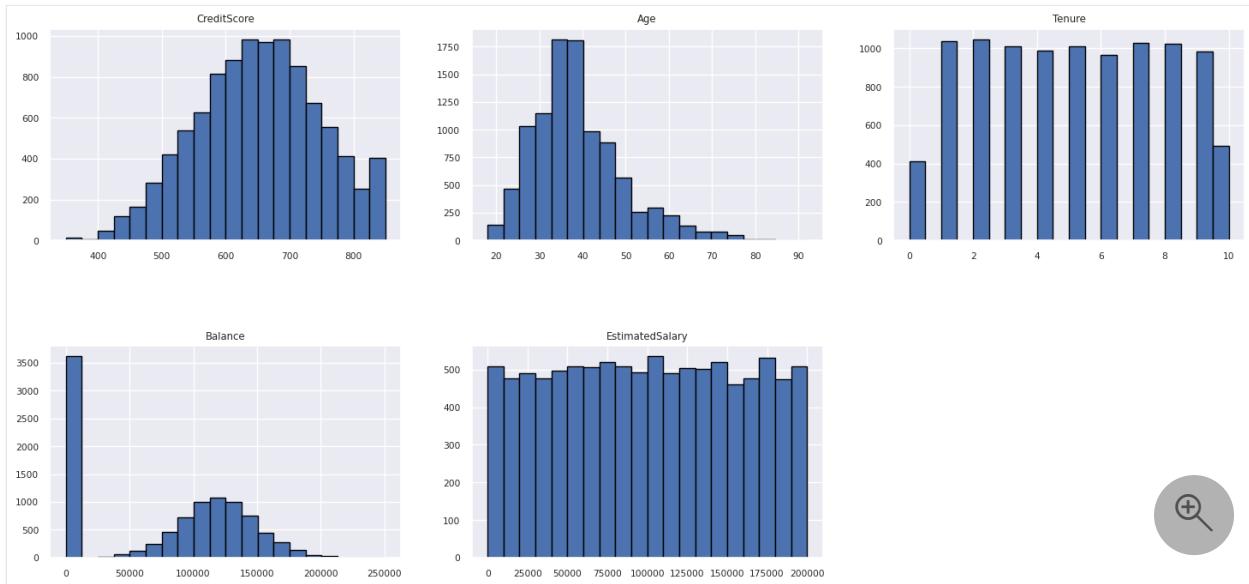
Show the frequency distribution of numerical attributes using histogram.

Python

```

columns = df_num_cols.columns[: len(df_num_cols.columns)]
fig = plt.figure()
fig.set_size_inches(18, 8)
length = len(columns)
for i,j in itertools.zip_longest(columns, range(length)):
    plt.subplot((length // 2), 3, j+1)
    plt.subplots_adjust(wspace = 0.2, hspace = 0.5)
    df_num_cols[i].hist(bins = 20, edgecolor = 'black')
    plt.title(i)
plt.show()

```



Perform feature engineering

Perform feature engineering to generate new attributes based on current attributes:

Python

```

df_clean[ "NewTenure" ] = df_clean[ "Tenure" ]/df_clean[ "Age" ]
df_clean[ "NewCreditsScore" ] = pd.qcut(df_clean[ 'CreditScore' ], 6, labels =
[1, 2, 3, 4, 5, 6])
df_clean[ "NewAgeScore" ] = pd.qcut(df_clean[ 'Age' ], 8, labels = [1, 2, 3, 4,
5, 6, 7, 8])
df_clean[ "NewBalanceScore" ] =
pd.qcut(df_clean[ 'Balance' ].rank(method="first"), 5, labels = [1, 2, 3, 4,
5])
df_clean[ "NewEstSalaryScore" ] = pd.qcut(df_clean[ 'EstimatedSalary' ], 10,
labels = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

```

Use Data Wrangler to perform one-hot encoding

Data Wrangler can also be used to perform one-hot encoding. To do so, re-open Data Wrangler. This time, select the `df_clean` data.

1. Expand **Formulas** and select **One-hot encode**.
2. A panel appears for you to select the list of columns you want to perform one-hot encoding on. Select **Geography** and **Gender**.

You could copy the generated code, close Data Wrangler to return to the notebook, then paste into a new cell. Or, select **Add code to notebook** at the top left to close Data Wrangler and add the code automatically.

If you didn't use Data Wrangler, you can instead use this next code cell:

Python

```
# This is the same code that Data Wrangler will generate

import pandas as pd

def clean_data(df_clean):
    # One-hot encode columns: 'Geography', 'Gender'
    df_clean = pd.get_dummies(df_clean, columns=['Geography', 'Gender'])
    return df_clean

df_clean_1 = clean_data(df_clean.copy())
df_clean_1.head()
```

Summary of observations from the exploratory data analysis

- Most of the customers are from France comparing to Spain and Germany, while Spain has the lowest churn rate comparing to France and Germany.
- Most of the customers have credit cards.
- There are customers whose age and credit score are above 60 and below 400, respectively, but they can't be considered as outliers.
- Very few customers have more than two of the bank's products.
- Customers who aren't active have a higher churn rate.
- Gender and tenure years don't seem to have an impact on customer's decision to close the bank account.

Create a delta table for the cleaned data

You'll use this data in the next notebook of this series.

Python

```
table_name = "df_clean"
# Create Spark DataFrame from pandas
sparkDF=spark.createDataFrame(df_clean_1)
sparkDF.write.mode("overwrite").format("delta").save(f"Tables/{table_name}")
print(f"Spark dataframe saved to delta table: {table_name}")
```

Next step

Train and register machine learning models with this data:

[Part 3: Train and register machine learning models](#) .

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Tutorial Part 3: Train and register a machine learning model

Article • 06/04/2024

In this tutorial, you'll learn to train multiple machine learning models to select the best one in order to predict which bank customers are likely to leave.

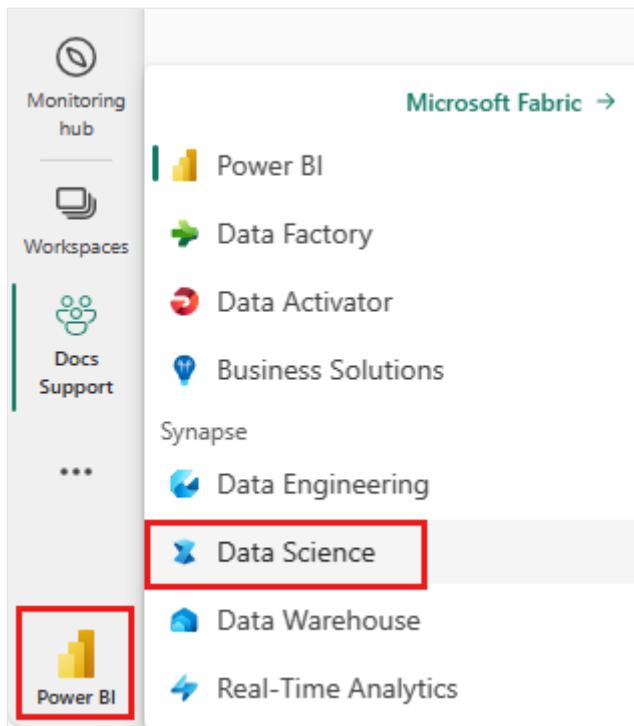
In this tutorial, you'll:

- ✓ Train Random Forrest and LightGBM models.
- ✓ Use Microsoft Fabric's native integration with the MLflow framework to log the trained machine learning models, the used hyperparameters, and evaluation metrics.
- ✓ Register the trained machine learning model.
- ✓ Assess the performances of the trained machine learning models on the validation dataset.

[MLflow](#) is an open source platform for managing the machine learning lifecycle with features like Tracking, Models, and Model Registry. MLflow is natively integrated with the Fabric Data Science experience.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



This is part 3 of 5 in the tutorial series. To complete this tutorial, first complete:

- Part 1: Ingest data into a Microsoft Fabric lakehouse using Apache Spark.
- Part 2: Explore and visualize data using Microsoft Fabric notebooks to learn more about the data.

Follow along in notebook

[3-train-evaluate.ipynb ↗](#) is the notebook that accompanies this tutorial.

To open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#), to import the notebook to your workspace.

If you'd rather copy and paste the code from this page, you can [create a new notebook](#).

Be sure to [attach a lakehouse to the notebook](#) before you start running code.

ⓘ Important

Attach the same lakehouse you used in part 1 and part 2.

Install custom libraries

For this notebook, you'll install imbalanced-learn (imported as `imblearn`) using `%pip install`. Imbalanced-learn is a library for Synthetic Minority Oversampling Technique (SMOTE) which is used when dealing with imbalanced datasets. The PySpark kernel will

be restarted after `%pip install`, so you'll need to install the library before you run any other cells.

You'll access SMOTE using the `imblearn` library. Install it now using the in-line installation capabilities (e.g., `%pip`, `%conda`).

Python

```
# Install imblearn for SMOTE using pip
%pip install imblearn
```

ⓘ Important

Run this install each time you restart the notebook.

When you install a library in a notebook, it's only available for the duration of the notebook session and not in the workspace. If you restart the notebook, you'll need to install the library again.

If you have a library you often use, and you want to make it available to all notebooks in your workspace, you can use a [Fabric environment](#) for that purpose. You can create an environment, install the library in it, and then your **workspace admin** can attach the environment to the workspace as its default environment. For more information on setting an environment as the workspace default, see [Admin sets default libraries for the workspace](#).

For information on migrating existing workspace libraries and Spark properties to an environment, see [Migrate workspace libraries and Spark properties to a default environment](#).

Load the data

Prior to training any machine learning model, you need to load the delta table from the lakehouse in order to read the cleaned data you created in the previous notebook.

Python

```
import pandas as pd
SEED = 12345
df_clean = spark.read.format("delta").load("Tables/df_clean").toPandas()
```

Generate experiment for tracking and logging the model using MLflow

This section demonstrates how to generate an experiment, specify the machine learning model and training parameters as well as scoring metrics, train the machine learning models, log them, and save the trained models for later use.

Python

```
import mlflow
# Setup experiment name
EXPERIMENT_NAME = "bank-churn-experiment" # MLflow experiment name
```

Extending the MLflow autologging capabilities, autologging works by automatically capturing the values of input parameters and output metrics of a machine learning model as it is being trained. This information is then logged to your workspace, where it can be accessed and visualized using the MLflow APIs or the corresponding experiment in your workspace.

All the experiments with their respective names are logged and you'll be able to track their parameters and performance metrics. To learn more about autologging, see [Autologging in Microsoft Fabric](#).

Set experiment and autologging specifications

Python

```
mlflow.set_experiment(EXPERIMENT_NAME)
mlflow.autolog(exclusive=False)
```

Import scikit-learn and LightGBM

With your data in place, you can now define the machine learning models. You'll apply Random Forrest and LightGBM models in this notebook. Use `scikit-learn` and `lightgbm` to implement the models within a few lines of code.

Python

```
# Import the required libraries for model training
from sklearn.model_selection import train_test_split
from lightgbm import LGBMClassifier
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.metrics import accuracy_score, f1_score, precision_score,
confusion_matrix, recall_score, roc_auc_score, classification_report
```

Prepare training, validation and test datasets

Use the `train_test_split` function from `scikit-learn` to split the data into training, validation, and test sets.

Python

```
y = df_clean["Exited"]
X = df_clean.drop("Exited",axis=1)
# Split the dataset to 60%, 20%, 20% for training, validation, and test
datasets
# Train-Test Separation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,
random_state=SEED)
# Train-Validation Separation
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=0.25, random_state=SEED)
```

Save test data to a delta table

Save the test data to the delta table for use in the next notebook.

Python

```
table_name = "df_test"
# Create PySpark DataFrame from Pandas
df_test=spark.createDataFrame(X_test)
df_test.write.mode("overwrite").format("delta").save(f"Tables/{table_name}")
print(f"Spark test DataFrame saved to delta table: {table_name}")
```

Apply SMOTE to the training data to synthesize new samples for the minority class

The data exploration in part 2 showed that out of the 10,000 data points corresponding to 10,000 customers, only 2,037 customers (around 20%) have left the bank. This indicates that the dataset is highly imbalanced. The problem with imbalanced classification is that there are too few examples of the minority class for a model to effectively learn the decision boundary. SMOTE is the most widely used approach to

synthesize new samples for the minority class. Learn more about SMOTE [here](#) and [here](#).

💡 Tip

Note that SMOTE should only be applied to the training dataset. You must leave the test dataset in its original imbalanced distribution in order to get a valid approximation of how the machine learning model will perform on the original data, which is representing the situation in production.

Python

```
from collections import Counter
from imblearn.over_sampling import SMOTE

sm = SMOTE(random_state=SEED)
X_res, y_res = sm.fit_resample(X_train, y_train)
new_train = pd.concat([X_res, y_res], axis=1)
```

💡 Tip

You can safely ignore the MLflow warning message that appears when you run this cell. If you see a **ModuleNotFoundError** message, you missed running the first cell in this notebook, which installs the `imblearn` library. You need to install this library each time you restart the notebook. Go back and re-run all the cells starting with the first cell in this notebook.

Model training

- Train the model using Random Forest with maximum depth of 4 and 4 features

Python

```
mlflow.sklearn.autolog(registered_model_name='rfc1_sm') # Register the
trained model with autologging
rfc1_sm = RandomForestClassifier(max_depth=4, max_features=4,
min_samples_split=3, random_state=1) # Pass hyperparameters
with mlflow.start_run(run_name="rfc1_sm") as run:
    rfc1_sm_run_id = run.info.run_id # Capture run_id for model prediction
    later
    print("run_id: {}; status: {}".format(rfc1_sm_run_id, run.info.status))
    # rfc1.fit(X_train,y_train) # Imbalanced training data
    rfc1_sm.fit(X_res, y_res.ravel()) # Balanced training data
    rfc1_sm.score(X_val, y_val)
```

```

y_pred = rfc1_sm.predict(X_val)
cr_rfc1_sm = classification_report(y_val, y_pred)
cm_rfc1_sm = confusion_matrix(y_val, y_pred)
roc_auc_rfc1_sm = roc_auc_score(y_res, rfc1_sm.predict_proba(X_res)[:, 1])

```

- Train the model using Random Forest with maximum depth of 8 and 6 features

Python

```

mlflow.sklearn.autolog(registered_model_name='rfc2_sm') # Register the trained model with autologging
rfc2_sm = RandomForestClassifier(max_depth=8, max_features=6, min_samples_split=3, random_state=1) # Pass hyperparameters
with mlflow.start_run(run_name="rfc2_sm") as run:
    rfc2_sm_run_id = run.info.run_id # Capture run_id for model prediction later
    print("run_id: {}; status: {}".format(rfc2_sm_run_id, run.info.status))
    # rfc2.fit(X_train,y_train) # Imbalanced training data
    rfc2_sm.fit(X_res, y_res.ravel()) # Balanced training data
    rfc2_sm.score(X_val, y_val)
    y_pred = rfc2_sm.predict(X_val)
    cr_rfc2_sm = classification_report(y_val, y_pred)
    cm_rfc2_sm = confusion_matrix(y_val, y_pred)
    roc_auc_rfc2_sm = roc_auc_score(y_res, rfc2_sm.predict_proba(X_res)[:, 1])

```

- Train the model using LightGBM

Python

```

# lgbm_model
mlflow.lightgbm.autolog(registered_model_name='lgbm_sm') # Register the trained model with autologging
lgbm_sm_model = LGBMClassifier(learning_rate = 0.07,
                               max_delta_step = 2,
                               n_estimators = 100,
                               max_depth = 10,
                               eval_metric = "logloss",
                               objective='binary',
                               random_state=42)

with mlflow.start_run(run_name="lgbm_sm") as run:
    lgbm1_sm_run_id = run.info.run_id # Capture run_id for model prediction later
    # lgbm_sm_model.fit(X_train,y_train) # Imbalanced training data
    lgbm_sm_model.fit(X_res, y_res.ravel()) # Balanced training data
    y_pred = lgbm_sm_model.predict(X_val)
    accuracy = accuracy_score(y_val, y_pred)
    cr_lgbm_sm = classification_report(y_val, y_pred)
    cm_lgbm_sm = confusion_matrix(y_val, y_pred)

```

```
roc_auc_lgbm_sm = roc_auc_score(y_res,
lgbm_sm_model.predict_proba(X_res)[:, 1])
```

Experiments artifact for tracking model performance

The experiment runs are automatically saved in the experiment artifact that can be found from the workspace. They're named based on the name used for setting the experiment. All of the trained machine learning models, their runs, performance metrics, and model parameters are logged.

To view your experiments:

1. On the left panel, select your workspace.
2. On the top right, filter to show only experiments, to make it easier to find the experiment you're looking for.

3. Find and select the experiment name, in this case *bank-churn-experiment*. If you don't see the experiment in your workspace, refresh your browser.

Name	Value
Run metrics (1)	accuracy_score_X_test 0.863
Run Parameters (27)	boosting_type gbdt colsample_bytree 1.0 learning_rate 0.09 max_depth 10 min_child_samples 20 min_child_weight 0.001 min_split_gain 0.0 n_jobs -1

Assess the performances of the trained models on the validation dataset

Once done with machine learning model training, you can assess the performance of trained models in two ways.

- Open the saved experiment from the workspace, load the machine learning models, and then assess the performance of the loaded models on the validation dataset.

Python

```
# Define run_uri to fetch the model
# mlflow client: mlflow.model.url, list model
load_model_rfc1_sm =
mlflow.sklearn.load_model(f"runs:{rfc1_sm_run_id}/model")
load_model_rfc2_sm =
mlflow.sklearn.load_model(f"runs:{rfc2_sm_run_id}/model")
load_model_lgbm1_sm =
mlflow.lightgbm.load_model(f"runs:{lgbm1_sm_run_id}/model")
# Assess the performance of the loaded model on validation dataset
ypred_rfc1_sm_v1 = load_model_rfc1_sm.predict(X_val) # Random Forest
with max depth of 4 and 4 features
ypred_rfc2_sm_v1 = load_model_rfc2_sm.predict(X_val) # Random Forest
with max depth of 8 and 6 features
ypred_lgbm1_sm_v1 = load_model_lgbm1_sm.predict(X_val) # LightGBM
```

- Directly assess the performance of the trained machine learning models on the validation dataset.

Python

```
ypred_rfc1_sm_v2 = rfc1_sm.predict(X_val) # Random Forest with max
depth of 4 and 4 features
ypred_rfc2_sm_v2 = rfc2_sm.predict(X_val) # Random Forest with max
depth of 8 and 6 features
ypred_lgbm1_sm_v2 = lgbm_sm_model.predict(X_val) # LightGBM
```

Depending on your preference, either approach is fine and should offer identical performances. In this notebook, you'll choose the first approach in order to better demonstrate the MLflow autologging capabilities in Microsoft Fabric.

Show True/False Positives/Negatives using the Confusion Matrix

Next, you'll develop a script to plot the confusion matrix in order to evaluate the accuracy of the classification using the validation dataset. The confusion matrix can be plotted using SynapseML tools as well, which is shown in Fraud Detection sample that is available [here](#).

Python

```
import seaborn as sns
sns.set_theme(style="whitegrid", palette="tab10", rc = {'figure.figsize': (9,6)})
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
from matplotlib import rc, rcParams
import numpy as np
import itertools

def plot_confusion_matrix(cm, classes,
                         normalize=False,
                         title='Confusion matrix',
                         cmap=plt.cm.Blues):
    print(cm)
    plt.figure(figsize=(4,4))
    plt.rcParams.update({'font.size': 10})
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45, color="blue")
    plt.yticks(tick_marks, classes, color="blue")

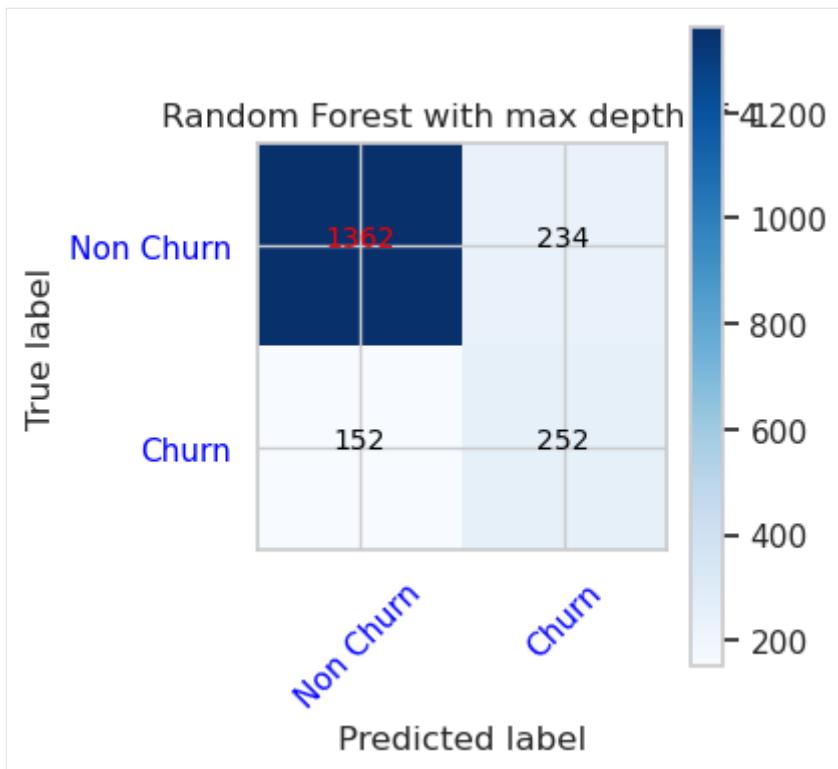
    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="red" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

- Confusion Matrix for Random Forest Classifier with maximum depth of 4 and 4 features

Python

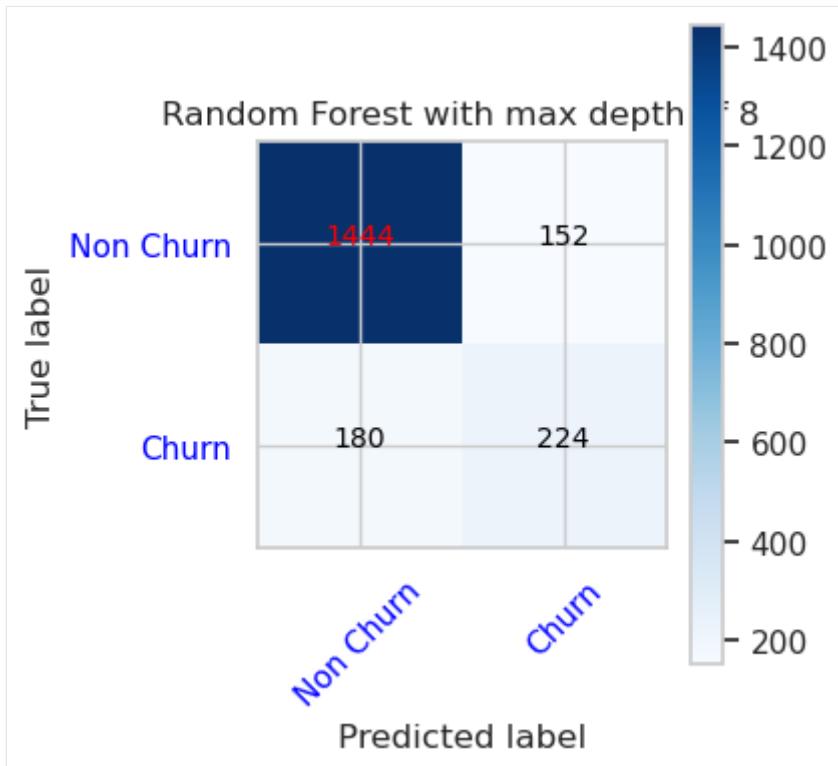
```
cfm = confusion_matrix(y_val, y_pred=ypred_rfc1_sm_v1)
plot_confusion_matrix(cfm, classes=['Non Churn', 'Churn'],
                      title='Random Forest with max depth of 4')
tn, fp, fn, tp = cfm.ravel()
```



- Confusion Matrix for Random Forest Classifier with maximum depth of 8 and 6 features

Python

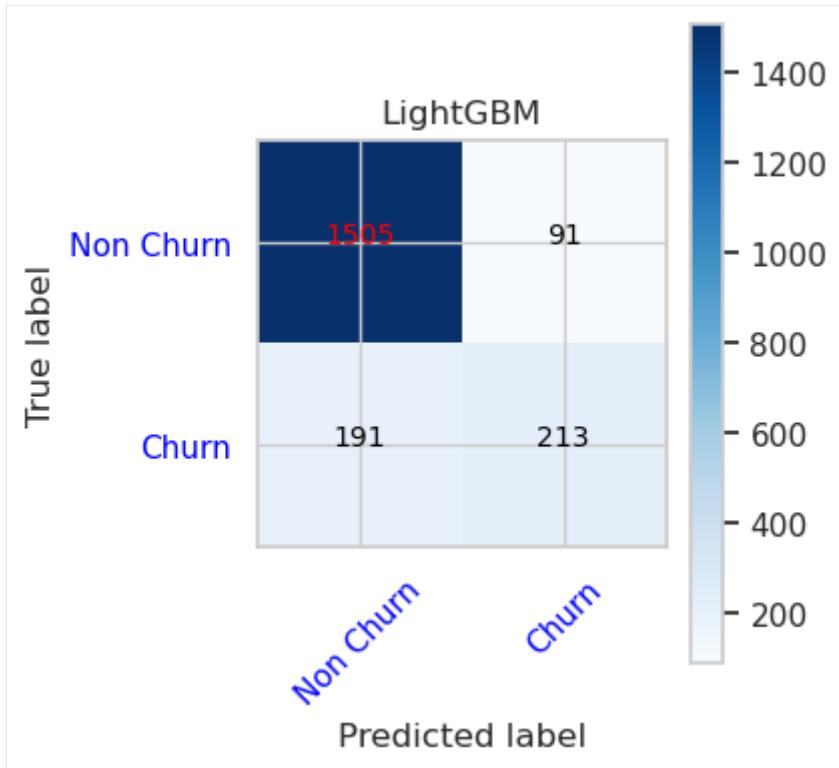
```
cfm = confusion_matrix(y_val, y_pred=ypred_rfc2_sm_v1)
plot_confusion_matrix(cfm, classes=['Non Churn', 'Churn'],
                      title='Random Forest with max depth of 8')
tn, fp, fn, tp = cfm.ravel()
```



- Confusion Matrix for LightGBM

```
Python
```

```
cfm = confusion_matrix(y_val, y_pred=ypred_lgbm1_sm_v1)
plot_confusion_matrix(cfm, classes=[ 'Non Churn', 'Churn'],
                      title='LightGBM')
tn, fp, fn, tp = cfm.ravel()
```



Next step

Part 4: Perform batch scoring and save predictions to a lakehouse

Feedback

Was this page helpful?

Yes

No

Provide product feedback | Ask the community

Tutorial Part 4: Perform batch scoring and save predictions to a lakehouse

Article • 07/08/2024

In this tutorial, you'll learn to import the registered LightGBMClassifier model that was trained in part 3 using the Microsoft Fabric MLflow model registry, and perform batch predictions on a test dataset loaded from a lakehouse.

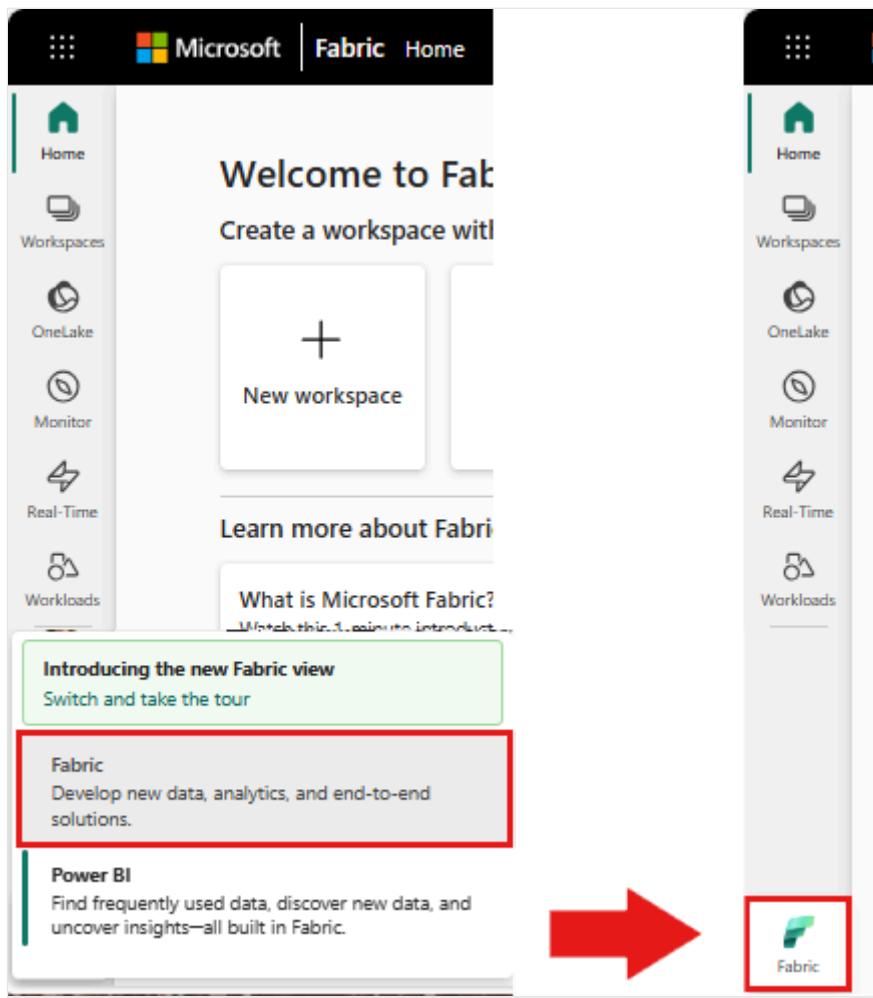
Microsoft Fabric allows you to operationalize machine learning models with a scalable function called PREDICT, which supports batch scoring in any compute engine. You can generate batch predictions directly from a Microsoft Fabric notebook or from a given model's item page. Learn about [PREDICT](#).

To generate batch predictions on the test dataset, you'll use version 1 of the trained LightGBM model that demonstrated the best performance among all trained machine learning models. You'll load the test dataset into a spark DataFrame and create an MLFlowTransformer object to generate batch predictions. You can then invoke the PREDICT function using one of following three ways:

- ✓ Transformer API from SynapseML
- ✓ Spark SQL API
- ✓ PySpark user-defined function (UDF)

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



This part 4 of 5 in the tutorial series. To complete this tutorial, first complete:

- Part 1: Ingest data into a Microsoft Fabric lakehouse using Apache Spark.
- Part 2: Explore and visualize data using Microsoft Fabric notebooks to learn more about the data.
- Part 3: Train and register machine learning models.

Follow along in notebook

[4-predict.ipynb](#) is the notebook that accompanies this tutorial.

- To open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#) to import the notebook to your workspace.
- If you'd rather copy and paste the code from this page, you can [create a new notebook](#).
- Be sure to [attach a lakehouse to the notebook](#) before you start running code.

ⓘ Important

Attach the same lakehouse you used in the other parts of this series.

Load the test data

Load the test data that you saved in Part 3.

Python

```
df_test = spark.read.format("delta").load("Tables/df_test")
display(df_test)
```

PREDICT with the Transformer API

To use the Transformer API from SynapseML, you'll need to first create an MLFlowTransformer object.

Instantiate MLFlowTransformer object

The MLFlowTransformer object is a wrapper around the MLFlow model that you registered in Part 3. It allows you to generate batch predictions on a given DataFrame. To instantiate the MLFlowTransformer object, you'll need to provide the following parameters:

- The columns from the test DataFrame that you need as input to the model (in this case, you would need all of them).
- A name for the new output column (in this case, predictions).
- The correct model name and model version to generate the predictions (in this case, `lgbm_sm` and version 1).

Python

```
from synapse.ml.predict import MLFlowTransformer

model = MLFlowTransformer(
    inputCols=list(df_test.columns),
    outputCol='predictions',
    modelName='lgbm_sm',
    modelVersion=1
)
```

Now that you have the MLFlowTransformer object, you can use it to generate batch predictions.

Python

```
import pandas

predictions = model.transform(df_test)
display(predictions)
```

PREDICT with the Spark SQL API

The following code invokes the PREDICT function with the Spark SQL API.

Python

```
from pyspark.ml.feature import SQLTransformer

# Substitute "model_name", "model_version", and "features" below with values
# for your own model name, model version, and feature columns
model_name = 'lgbm_sm'
model_version = 1
features = df_test.columns

sqlt = SQLTransformer().setStatement(
    f"SELECT PREDICT('{model_name}/{model_version}', {','.join(features)})"
    as predictions FROM __THIS__)

# Substitute "X_test" below with your own test dataset
display(sqlt.transform(df_test))
```

PREDICT with a user-defined function (UDF)

The following code invokes the PREDICT function with a PySpark UDF.

Python

```
from pyspark.sql.functions import col, pandas_udf, udf, lit

# Substitute "model" and "features" below with values for your own model
# name and feature columns
my_udf = model.to_udf()
features = df_test.columns

display(df_test.withColumn("predictions", my_udf(*[col(f) for f in
features])))
```

Note that you can also generate PREDICT code from a model's item page. Learn about [PREDICT](#).

Write model prediction results to the lakehouse

Once you have generated batch predictions, write the model prediction results back to the lakehouse.

Python

```
# Save predictions to lakehouse to be used for generating a Power BI report
table_name = "customer_churn_test_predictions"
predictions.write.format('delta').mode("overwrite").save(f"Tables/{table_name}")
print(f"Spark DataFrame saved to delta table: {table_name}")
```

Next step

Continue on to:

[Part 5: Create a Power BI report to visualize predictions](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Tutorial Part 5: Visualize predictions with a Power BI report

Article • 11/24/2024

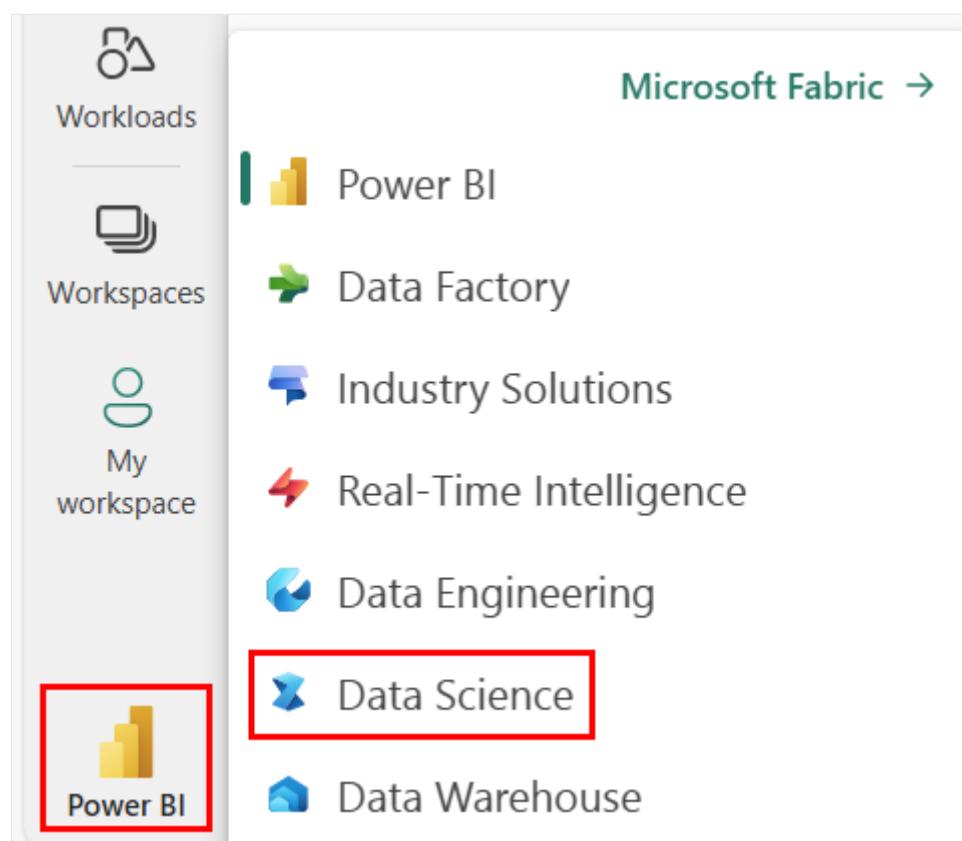
In this tutorial, you create a Power BI report from the predictions data generated in [Part 4: Perform batch scoring and save predictions to a lakehouse](#).

You'll learn how to:

- ✓ Create a semantic model from the predictions data.
- ✓ Add new measures to the data from Power BI.
- ✓ Create a Power BI report.
- ✓ Add visualizations to the report.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



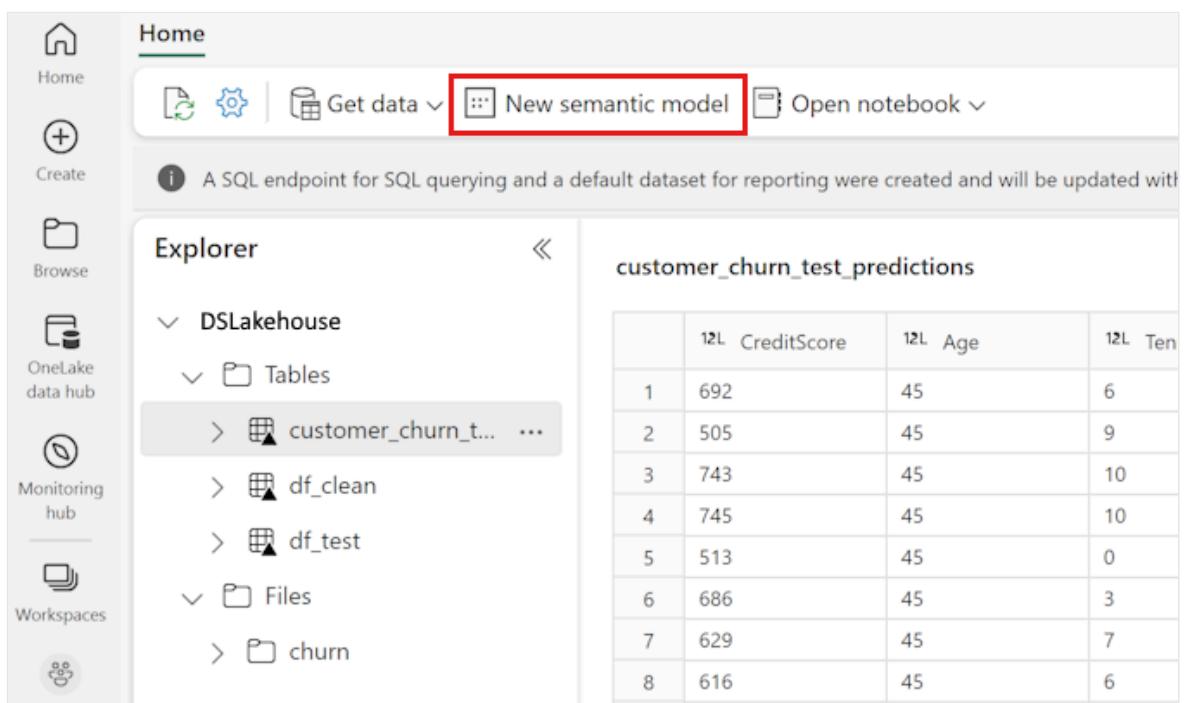
This is part 5 of 5 in the tutorial series. To complete this tutorial, first complete:

- Part 1: Ingest data into a Microsoft Fabric lakehouse using Apache Spark.
- Part 2: Explore and visualize data using Microsoft Fabric notebooks to learn more about the data.
- Part 3: Train and register machine learning models.
- Part 4: Perform batch scoring and save predictions to a lakehouse.

Create a semantic model

Create a new semantic model linked to the predictions data you produced in part 4:

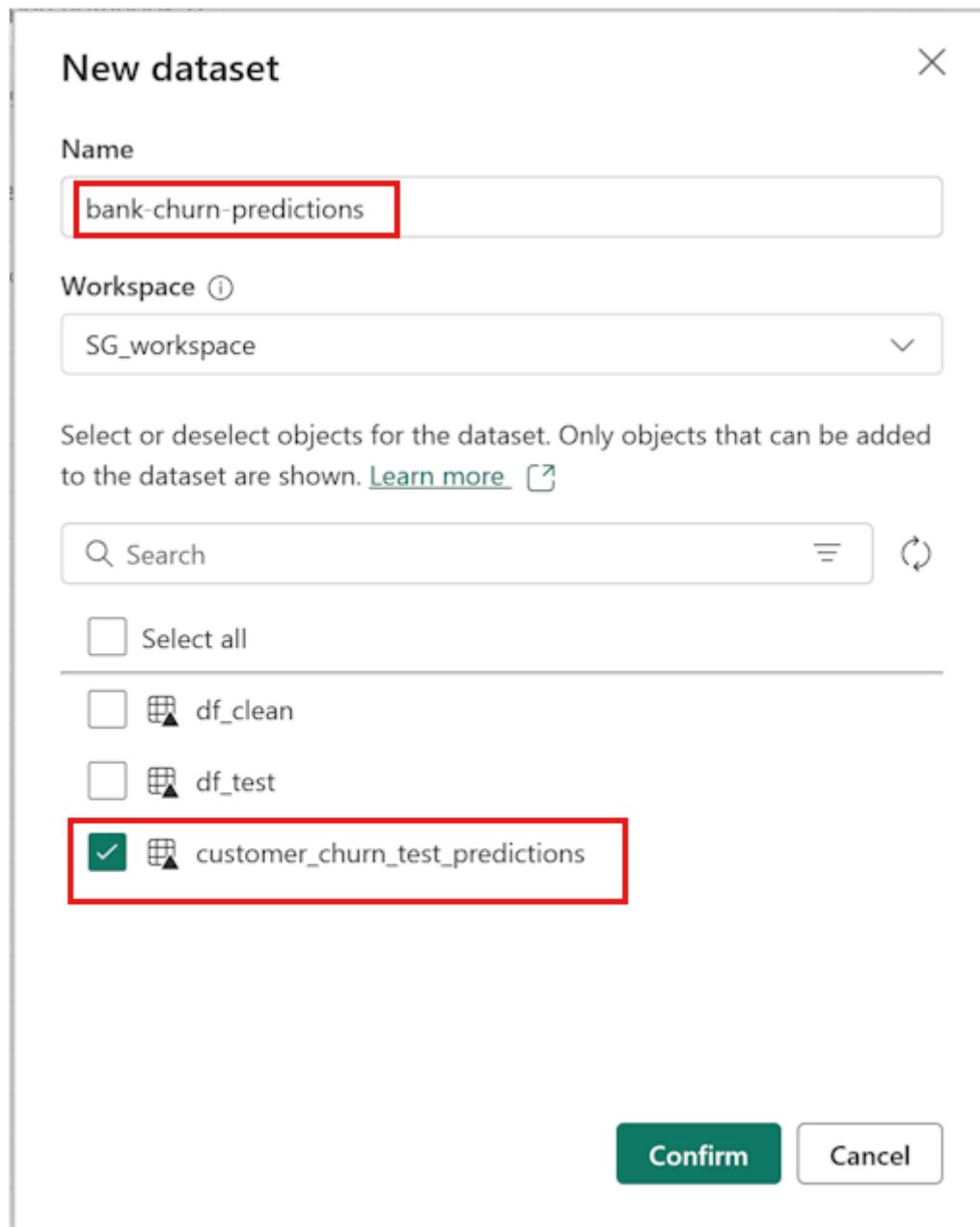
1. On the left, select your workspace.
2. On the top left, select **Lakehouse** as a filter.
3. Select the lakehouse that you used in the previous parts of the tutorial series.
4. Select **New semantic model** on the top ribbon.



The screenshot shows the Microsoft Fabric Home interface. On the left, there's a sidebar with icons for Home, Create, Browse, OneLake data hub, Monitoring hub, and Workspaces. The main area has a 'Home' tab selected at the top. Below it are buttons for 'Get data', 'New semantic model' (which is highlighted with a red box), and 'Open notebook'. A tooltip message says: 'A SQL endpoint for SQL querying and a default dataset for reporting were created and will be updated with the latest data from the source.' The 'Explorer' section on the left shows a tree view of 'DSLakehouse' containing 'Tables' (with 'customer_churn_t...' selected) and 'Files' (with 'churn' selected). To the right, a preview of the 'customer_churn_test_predictions' dataset is shown in a table format with columns: ID, CreditScore, Age, and Ten. The data rows are:

	CreditScore	Age	Ten
1	692	45	6
2	505	45	9
3	743	45	10
4	745	45	10
5	513	45	0
6	686	45	3
7	629	45	7
8	616	45	6

5. Give the semantic model a name, such as "bank churn predictions." Then select the **customer_churn_test_predictions** dataset.

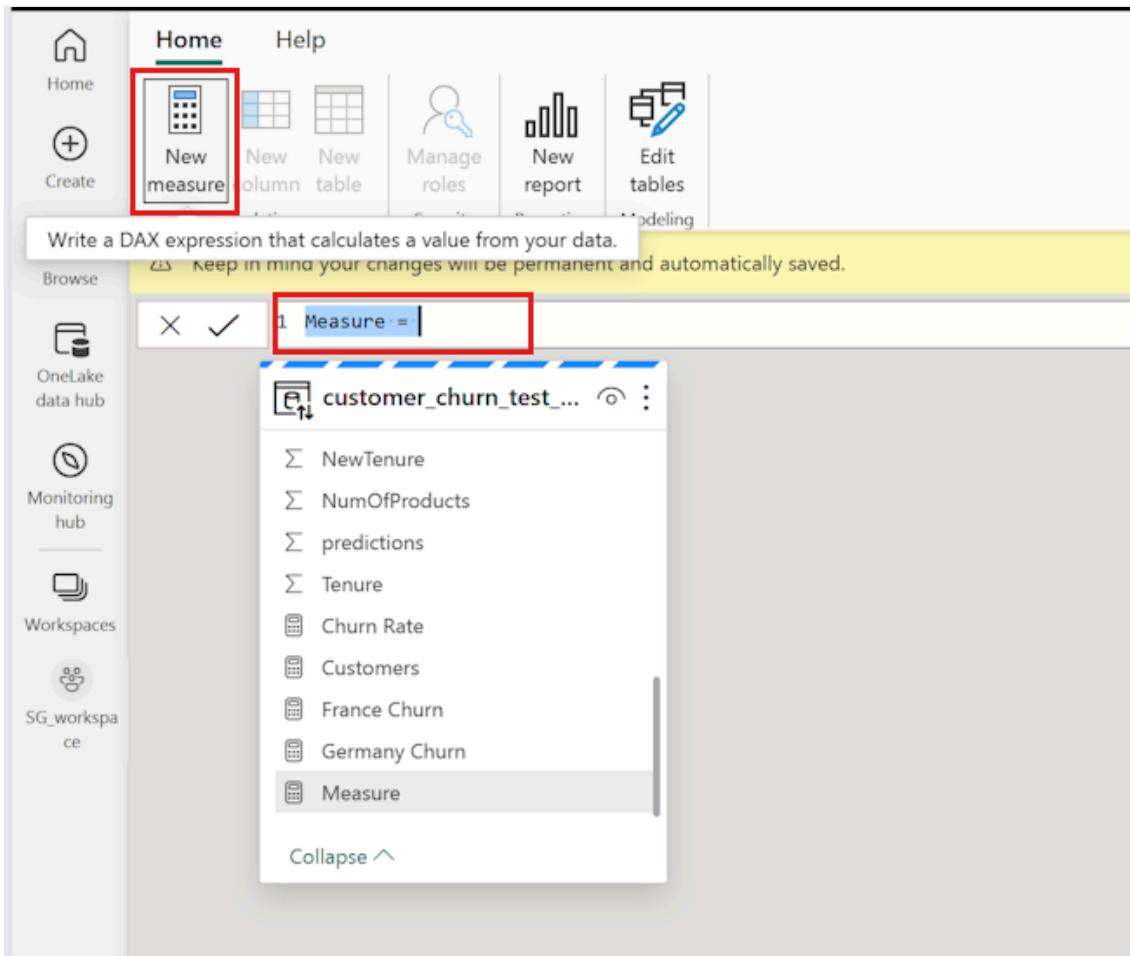


6. Select Confirm.

Add new measures

Now add a few measures to the semantic model:

1. Add a new measure for the churn rate.
 - a. Select **New measure** in the top ribbon. This action adds a new item named **Measure** to the **customer_churn_test_predictions** dataset, and opens a formula bar above the table.



- b. To determine the average predicted churn rate, replace **Measure =** in the formula bar with:

Python

```
Churn Rate = AVERAGE(customer_churn_test_predictions[predictions])
```

- c. To apply the formula, select the check mark in the formula bar. The new measure appears in the data table. The calculator icon shows it was created as a measure.
- d. Change the format from **General** to **Percentage** in the **Properties** panel.
- e. Scroll down in the **Properties** panel to change the **Decimal places** to 1.

The screenshot shows the Power BI desktop interface. In the top ribbon, the 'Home' tab is selected. The formula bar at the top contains the formula `1 Churn Rate = AVERAGE(customer_churn_test_predictions[predictions])`. Below the formula bar, the 'customer_churn_test...' dataset is expanded, showing columns like NewAgeScore, NewBalanceScore, etc., and the newly created 'Churn Rate' measure. On the right side, the 'Properties' pane is open, showing settings for the 'Churn Rate' measure. The 'Format' section is highlighted with a red box, specifically the 'Percentage' dropdown and the 'Decimal places' input field set to 1. Other properties like 'Description' and 'Display folder' are also visible.

2. Add a new measure that counts the total number of bank customers. You'll need it for the rest of the new measures.

- Select **New measure** in the top ribbon to add a new item named **Measure** to the `customer_churn_test_predictions` dataset. This action also opens a formula bar above the table.
- Each prediction represents one customer. To determine the total number of customers, replace `Measure =` in the formula bar with:

Python

```
Customers = COUNT(customer_churn_test_predictions[predictions])
```

- To apply the formula, select the check mark in the formula bar.

3. Add the churn rate for Germany.

- Select **New measure** in the top ribbon to add a new item named **Measure** to the `customer_churn_test_predictions` dataset. This action also opens a formula bar above the table.
- To determine the churn rate for Germany, replace `Measure =` in the formula bar with:

Python

```
Germany Churn =
CALCULATE(AVERAGE(customer_churn_test_predictions[predictions])), FILTER
```

```
ER(customer_churn_test_predictions,  
customer_churn_test_predictions[Geography_Germany] = TRUE()))
```

This filters the rows down to the ones with Germany as their geography (Geography_Germany equals one).

- c. To apply the formula, select the check mark in the formula bar.
4. Repeat the above step to add the churn rates for France and Spain.
 - Spain's churn rate:

Python

```
Spain Churn =  
CALCULATE(AVERAGE(customer_churn_test_predictions[predictions]),FI  
LTER(customer_churn_test_predictions,  
customer_churn_test_predictions[Geography_Spain] = TRUE()))
```

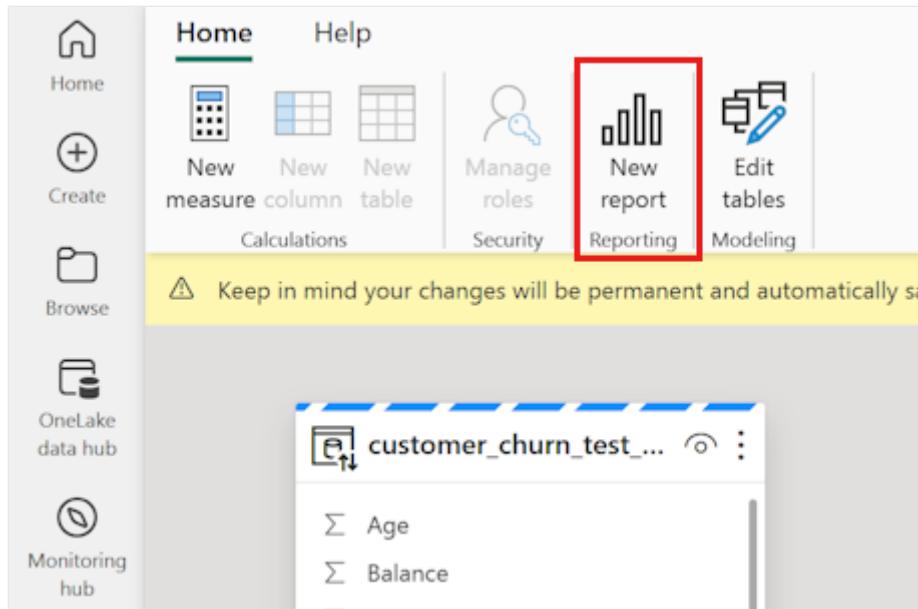
- France's churn rate:

Python

```
France Churn =  
CALCULATE(AVERAGE(customer_churn_test_predictions[predictions]),FI  
LTER(customer_churn_test_predictions,  
customer_churn_test_predictions[Geography_France] = TRUE()))
```

Create new report

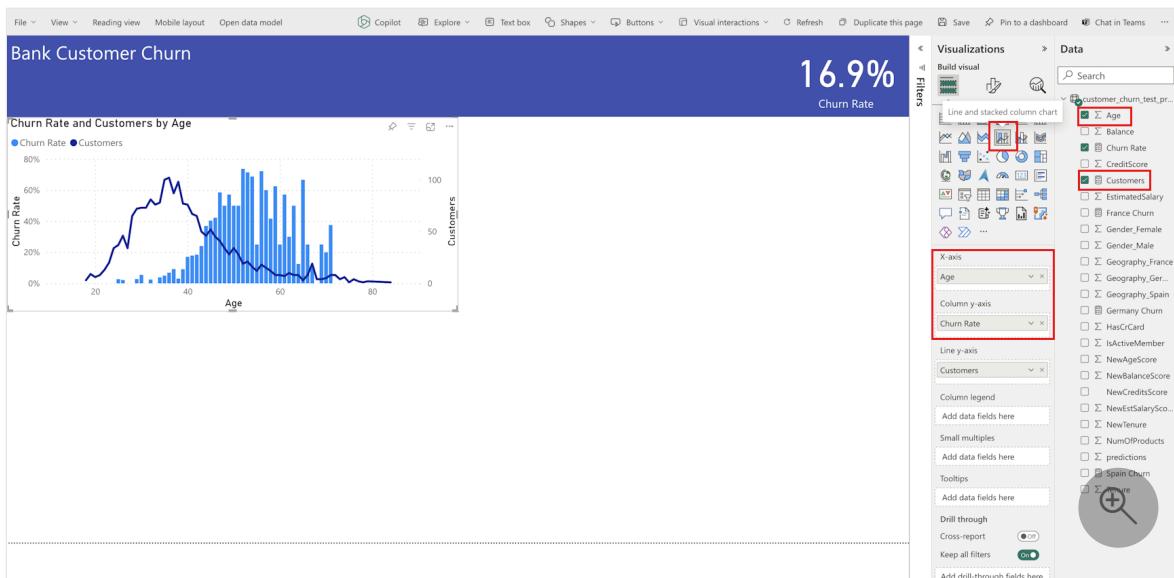
Once you're done with all operations, move on to the Power BI report authoring page by selecting **Create report** on the top ribbon.



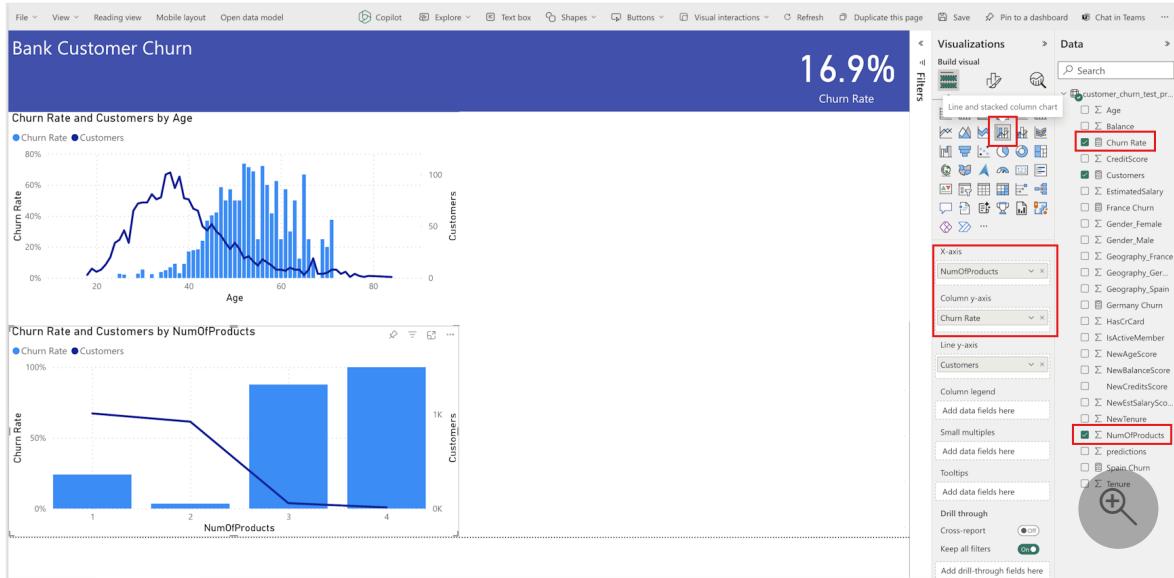
Once the report page appears, add these visuals:

1. Select the text box on the top ribbon and enter a title for the report, such as "Bank Customer Churn". Change the font size and background color in the Format panel. Adjust the font size and color by selecting the text and using the format bar.
2. In the Visualizations panel, select the **Card** icon. From the Data pane, select **Churn Rate**. Change the font size and background color in the Format panel. Drag this visualization to the top right of the report.

3. In the Visualizations panel, select the **Line and stacked column chart** icon. Select **age** for the x-axis, **Churn Rate** for column y-axis, and **Customers** for the line y-axis.



4. In the Visualizations panel, select the **Line and stacked column chart** icon. Select **NumOfProducts** for x-axis, **Churn Rate** for column y-axis, and **Customers** for the line y-axis.



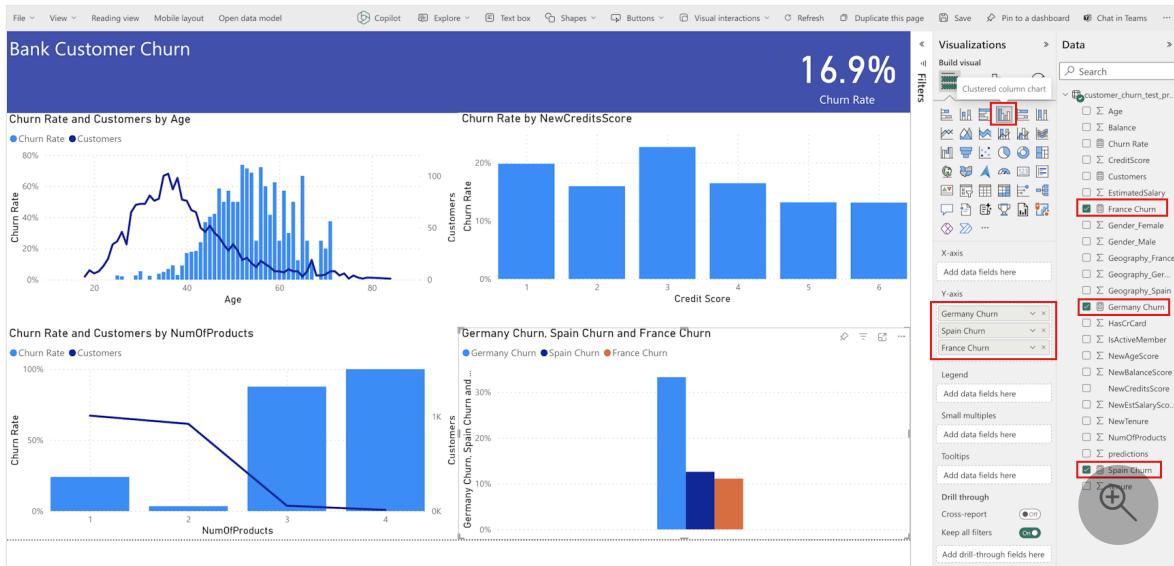
5. In the Visualizations panel, select the **Stacked column chart** icon. Select **NewCreditsScore** for x-axis and **Churn Rate** for y-axis.



Change the title "NewCreditsScore" to "Credit Score" in the Format panel.



6. In the Visualizations panel, select the **Clustered column chart** card. Select **Germany Churn, Spain Churn, France Churn** in that order for the y-axis.



Note

This report represents an illustrated example of how you might analyze the saved prediction results in Power BI. However, for a real customer churn use-case, the you may have to do more thorough ideation of what visualizations to create, based on your subject matter expertise, and what your firm and business analytics team has standardized as metrics.

The Power BI report shows:

- Bank customers who use more than two of the bank products have a higher churn rate, although few customers had more than two products. The bank should collect more data, but also investigate other features that correlate with more products (review the plot in the bottom left panel).
- Bank customers in Germany have a higher churn rate than in France and Spain (review the plot in the bottom right panel), suggesting that an investigation into what encouraged customers to leave could become helpful.
- There are more middle aged customers (between 25-45), and customers between 45-60 tend to exit more.
- Finally, customers with lower credit scores would most likely leave the bank for other financial institutions. The bank should look for ways to encourage customers with lower credit scores and account balances to stay with the bank.

Next step

This completes the five part tutorial series. See other end-to-end sample tutorials:

[How to use end-to-end AI samples in Microsoft Fabric](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Overview of Copilot for Data Science and Data Engineering (preview)

Article • 03/31/2025

ⓘ Important

This feature is in [preview](#).

Copilot for Data Science and Data Engineering is an AI assistant that helps analyze and visualize data. It works with Lakehouse tables and files, Power BI Datasets, and pandas/spark/fabric dataframes to provide answers and code snippets directly in a notebook. Connections to OneLake and default attached Lakehouses allow Copilot to provide contextualized code suggestions and natural language responses tailored to your data.

Copilot can help you understand your data better, and it can offer suggestions to begin your notebook, including generated code for the initial cells. After it identifies and adds data sources through the Fabric object explorer, Copilot Chat offers suggestions on model types to implement. You can copy these recommendations directly into your notebook to start development. If you're unsure of your next steps, you can invoke Copilot in-cell for model direction insights.

When you encounter errors, Copilot provides suggested fixes. For further help, you can chat with Copilot for more options, to avoid constant online searches.

You also benefit from automatic documentation, with a simple "Add Comments" feature that summarizes code and data changes. This makes cells clear for you and others. Throughout your workflow, you can consult Copilot at specific points, receiving real-time support and guidance to accelerate your development process.

ⓘ Note

With Spark 3.4 and later versions in Microsoft Fabric, no installation cell is required to use Copilot in your notebook. Previous versions that required an installation cell (Spark 3.3 and earlier) are no longer supported.

ⓘ Note

- Your administrator needs to enable the tenant switch before you start using Copilot. See the article [Copilot tenant settings](#) for details.
- Your F64 or P1 capacity needs to be in one of the regions listed in this article, [Fabric region availability](#).
- If your tenant or capacity is outside the US or France, Copilot is disabled by default unless your Fabric tenant admin enables the [Data sent to Azure](#), [OpenAI can be processed outside your tenant's geographic region](#), [compliance boundary, or national cloud instance](#) tenant setting in the Fabric Admin portal.
- Copilot in Microsoft Fabric isn't supported on trial SKUs. Only paid SKUs (F64 or higher, or P1 or higher) are supported.
- Copilot in Fabric is currently rolling out in public preview and is expected to be available for all customers by end of March 2024.
- See the article [Overview of Copilot in Fabric and Power BI](#) for more information.

Introduction to Copilot for Data Science and Data Engineering for Fabric Data Science

With Copilot for Data Science and Data Engineering, you can chat with an AI assistant that can help you handle your data analysis and visualization tasks. You can ask the Copilot questions about lakehouse tables, Power BI Datasets, or Pandas/Spark dataframes inside notebooks. Copilot answers in natural language or code snippets. Copilot can also generate data-specific code for you, depending on the task. For example, Copilot for Data Science and Data Engineering can generate code for:

- Chart creation
- Filtering data
- Applying transformations
- Machine learning models

First, select the Copilot icon in the notebooks ribbon. The Copilot chat panel opens, and a new cell appears at the top of your notebook. You might also select copilot at the top of your Fabric Notebooks cell as well.

To maximize Copilot effectiveness, load a table or dataset as a dataframe in your notebook. The AI can then access the data and understand its structure and content.

Next, start chatting with the AI. Select the chat icon in the notebook toolbar, and type your question or request in the chat panel. For example, you can ask:

- "What is the average age of customers in this dataset?"
- "Show me a bar chart of sales by region"
- etc.

Copilot responds with the answer or the code, which you can copy and paste it your notebook. Copilot for Data Science and Data Engineering is a convenient, interactive way to explore and analyze your data.

Using the Copilot Chat panel to interact with your data

To chat with your data and get insights, select the chat icon in the notebook toolbar to open the Copilot chat panel. Type your questions or requests in the chat panel. For example, you can ask:

- "What is the average age of customers in this dataset?"
- "Show me a bar chart of sales by region"
- etc.

Copilot responds with the answer or the code, which you can copy and paste into your notebook. Additionally, Copilot can suggest what to do next with your data. Copilot provides suggestions and generates relevant code snippets to help you proceed with your data analysis and visualization tasks.

To interact with the Copilot chat panel in Microsoft Fabric notebooks, follow these steps:

1. **Open the Copilot Chat Panel:** To open the Copilot chat panel, select the chat icon in the notebook toolbar.
2. **Ask Questions or Make Requests:** Type your questions or requests in the chat panel. Here are some specific examples for data science and data engineering:
 - **Data Exploration:**
 - "What is the distribution of the 'age' column in this dataset?"
 - "Show me a histogram of the 'income' column."
 - **Data Cleaning:**
 - "How can I handle missing values in this dataset?"
 - "Generate code to remove duplicates from this dataframe."

- **Data Transformation:**
 - "How do I normalize the 'sales' column?"
 - "Create a new column 'profit' by subtracting 'cost' from 'revenue'."
- **Visualization:**
 - "Plot a scatter plot of 'height' vs 'weight'."
 - "Generate a box plot for the 'salary' column."
- **Machine Learning:**
 - "Train a decision tree classifier on this dataset."
 - "Generate code for a k-means clustering algorithm with 3 clusters."
- **Model Evaluation:**
 - "How do I evaluate the accuracy of a logistic regression model?"
 - "Generate a confusion matrix for the predictions."

1. **Receive Responses:** Copilot responds with natural language explanations or code snippets. You can copy and paste the code into your notebook to execute it.

2. **Get Suggestions:** If you don't know how to proceed, ask Copilot for suggestions:

- "What should I do next with this dataset?"
- "What are some recommended feature engineering techniques for this data?"

1. **Use Generated Code:** Copy the generated code snippets from the chat panel, and paste them into your notebook cells to run them.

With these steps and the provided examples, you can effectively interact with the Copilot chat panel to enhance your data science and data engineering workflows in Microsoft Fabric notebooks.

Using the Copilot In-Cell Panel and Quick Actions

You can interact with Copilot directly within your notebook cells to generate code and perform quick actions on your code cells. Here's how to use the Copilot in-cell panel:

1. **Generate Code:** To generate code for specific tasks, you can use the Copilot in-cell panel. For example, you can type your request in the text panel above the code cell:

```
plaintext
```

Can you generate code for a logistic regression that fits this data?

Copilot provides the necessary code snippet directly in the cell below.

- 1. Fix Code:** You can ask Copilot to fix errors in your code. Type your request in the text panel above the code cell, and Copilot suggests corrections.
- 2. Add Comments:** To automatically document your code, use the "Add Comments" feature. Copilot generates comments that summarize the code and data changes, to make your notebook more readable.
- 3. Optimize Code:** For performance improvements, you can ask Copilot to optimize your code. Copilot provides suggestions to enhance the efficiency of your code.
- 4. Explain Code:** If you need clarification about a piece of code, ask Copilot for an explanation. Copilot provides a detailed explanation of the code's functionality.

Steps to Use Quick Actions

- 1. Invoke Copilot In-Cell:** Select the Copilot icon in the notebook toolbar to start interacting with Copilot.
- 2. Type Your Request:** Enter your request or question in the text panel above the code cell. For example:

plaintext

Explain the following code snippet.

- 1. Receive Suggestions:** Copilot responds with the relevant code, fixes, comments, optimizations, or explanations.
- 2. Apply Suggestions:** Copy the generated code or suggestions from Copilot and paste them into your notebook cells to execute them.

With the Copilot in-cell panel, you can generate code, fix errors, add comments, optimize performance, and understand your code better, all within your Microsoft Fabric notebooks.

1 %chat
2
3 Analyze the pandas dataframe and recommend a few machine learning models

6 sec -Command executed in 5 sec 954 ms

Based on the pandas dataframe, we can analyze the data and recommend a few machine learning models. Here are some steps you can follow:

- Data Exploration:** Begin by exploring the dataframe to understand its structure, features, and target variable(s). Use methods like `df.head()`, `df.info()`, and `df.describe()` to get an overview of the data.
- Data Preprocessing:** Preprocess the data by handling missing values, encoding categorical variables, and scaling numerical features if necessary. You can use techniques like one-hot encoding or label encoding for categorical variables and standardization or normalization for numerical variables.
- Feature Selection/Engineering:** Analyze the importance of features and perform feature selection or engineering techniques to improve model performance. Techniques like correlation analysis, recursive feature elimination, or principal component analysis (PCA) can be used.
- Model Selection:** Based on the type of problem (regression or classification) and the available data, select appropriate machine learning models. For regression problems, models like linear regression, decision trees, random forests, or gradient boosting algorithms (e.g., XGBoost or LightGBM) can be considered. For classification problems, logistic regression, support vector machines (SVM), random forests, or neural networks (e.g., deep learning models) can be suitable options.
- Model Evaluation:** Split your dataset into training and testing sets using techniques like train-test split or cross-validation. Train your selected models on the training set and evaluate their performance using appropriate evaluation metrics such as mean squared error (MSE), root mean squared error (RMSE), accuracy, precision, recall, F1-score, etc.
- Hyperparameter Tuning:** Fine-tune your selected models by optimizing their hyperparameters using techniques like grid search or random search. This step helps in finding the best combination of hyperparameters that maximizes model performance.
- Model Comparison:** Compare the performance of different models based on evaluation metrics to identify the best-performing model for your specific problem.

Remember that these steps are general guidelines for analyzing a pandas dataframe and selecting machine learning models. The specific approach may vary depending on your dataset's characteristics and problem requirements.

Let's work with your data

Choose an option below or type a request to get started.

- Load data from my lakehouse into a DataFrame
- Generate insights from data
- Suggest data visualizations

Type a request or click the sparkle icon for ideas.

AI-generated content can have mistakes. Make sure it's accurate and appropriate before using it. Read preview terms.

Copilot for Data Science and Data Engineering also has schema and metadata awareness of lakehouse tables. Copilot can provide relevant information in context of your data hosted in an attached lakehouse. For example, you can ask:

- "How many tables are in the lakehouse?"
- "What are the columns of the table customers?"

Copilot responds with the relevant information if you added the lakehouse to the notebook. Copilot also has awareness of the names of files added to any lakehouse attached to the notebook. You can refer to those files by name in your chat. For example, if you have a file named `sales.csv` in your lakehouse, you can ask Copilot to "Create a dataframe from sales.csv". Copilot generates the code and displays it in the chat panel. With Copilot for notebooks, you can easily access and query your data from different sources. You don't need the exact command syntax to do it.

Tips

- "Clear" your conversation in the Copilot chat panel with the broom located at the top of the chat panel. Copilot retains knowledge of any inputs or outputs during the session, but this helps if you find the current content distracting.
- Use the chat magics library to configure settings about Copilot, including privacy settings. The default sharing mode maximizes the context sharing Copilot can access. Therefore, limiting the information provided to copilot can directly and significantly affect the relevance of its responses.
- When Copilot first launches, it offers a set of helpful prompts that can help you get started. They can help kickstart your conversation with Copilot. To refer to prompts

later, you can use the sparkle button at the bottom of the chat panel.

- You can "drag" the sidebar of the copilot chat to expand the chat panel, to view the code more clearly or to improve the readability of the outputs on your screen.

Limitations

Copilot features in the Data Science experience are currently scoped to notebooks.

These features include the Copilot chat pane, IPython magic commands that can be

used within a code cell, and automatic code suggestions as you type in a code cell.

Copilot can also read Power BI semantic models using an integration of semantic link.

Copilot has two key intended uses:

- You can ask Copilot to examine and analyze data in your notebook (for example, by first loading a DataFrame and then asking Copilot about data inside the DataFrame).
- You can ask Copilot to generate a range of suggestions about your data analysis process - for example, what predictive models might be relevant, code to perform different types of data analysis, and documentation for a completed notebook.

Code generation with fast-moving or recently released libraries might include inaccuracies or fabrications.

Deletion and Export of data

Copilot in notebooks provides users with two essential commands to manage chat history within notebook cells: `show_chat_history` and `clear_chat_history`. The `show_chat_history` command exports the complete chat history for compliance purposes, to ensure that all necessary interactions are documented and accessible for review. For example, executing `show_chat_history` generates a comprehensive log of the chat history, which can then be reviewed or archived for compliance.

The `clear_chat_history` command removes all previous conversations from the notebook, so that the user can start fresh. This command clears out old interactions, to start a new conversation thread. For instance, executing `clear_chat_history` deletes all previous chat history, to leave the notebook free of any past conversations. These features enhance the overall functionality and user experience of Copilot in notebooks.

Related content

- [How to use Chat-magics](#)

- How to use the Copilot Chat Pane
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Overview of chat-magics in Microsoft Fabric notebooks (preview)

Article • 12/04/2024

ⓘ Important

This feature is in [preview](#).

The Chat-magics Python library enhances your data science and engineering workflow in Microsoft Fabric notebooks. It seamlessly integrates with the Fabric environment, and allows execution of specialized IPython magic commands in a notebook cell, to provide real-time outputs. IPython magic commands and more background on usage can be found here: <https://ipython.readthedocs.io/en/stable/interactive/magics.html#>.

ⓘ Note

- Your administrator needs to enable the tenant switch before you start using Copilot. See the article [Copilot tenant settings](#) for details.
- Your F64 or P1 capacity needs to be in one of the regions listed in this article, [Fabric region availability](#).
- If your tenant or capacity is outside the US or France, Copilot is disabled by default unless your Fabric tenant admin enables the [Data sent to Azure OpenAI can be processed outside your tenant's geographic region, compliance boundary, or national cloud instance](#) tenant setting in the Fabric Admin portal.
- Copilot in Microsoft Fabric isn't supported on trial SKUs. Only paid SKUs (F64 or higher, or P1 or higher) are supported.
- Copilot in Fabric is currently rolling out in public preview and is expected to be available for all customers by end of March 2024.
- See the article [Overview of Copilot in Fabric and Power BI](#) for more information.

Capabilities of Chat-magics

Instant query and code generation

The `%%chat` command allows you to ask questions about the state of your notebook.

The `%%code` enables code generation for data manipulation or visualization.

Dataframe descriptions

The `%describe` command provides summaries and descriptions of loaded dataframes.

This simplifies the data exploration phase.

Commenting and debugging

The `%%add_comments` and `%%fix_errors` commands help add comments to your code and fix errors respectively. This helps make your notebook more readable and error-free.

Privacy controls

Chat-magics also offers granular privacy settings, which allows you to control what data is shared with the Azure OpenAI Service. The `%set_sharing_level` and `%configure_privacy_settings` commands, for example, provide this functionality.

How can Chat-magics help you?

Chat-magics enhances your productivity and workflow in Microsoft Fabric notebooks. It accelerates data exploration, simplifies notebook navigation, and improves code quality. It adapts to multilingual code environments, and it prioritizes data privacy and security. Through cognitive load reductions, it allows you to more closely focus on problem-solving. Whether you're a data scientist, data engineer, or business analyst, Chat-magics seamlessly integrates robust, enterprise-level Azure OpenAI capabilities directly into your notebooks. This makes it an indispensable tool for efficient and streamlined data science and engineering tasks.

Get started with Chat-magics

1. Open a new or existing Microsoft Fabric notebook.
2. Select the **Copilot** button on the notebook ribbon to output the Chat-magics initialization code into a new notebook cell.
3. Run the cell when it is added at the top of your notebook.

Verify the Chat-magics installation

1. Create a new cell in the notebook, and run the `%chat_magics` command to display the help message. This step verifies proper Chat-magics installation.

Introduction to basic commands: `%%chat` and `%%code`

Using `%%chat` (Cell Magic)

1. Create a new cell in your notebook.
2. Type `%%chat` at the top of the cell.
3. Enter your question or instruction below the `%%chat` command - for example, **What variables are currently defined?**
4. Execute the cell to see the Chat-magics response.

Using `%%code` (Cell Magic)

1. Create a new cell in your notebook.
2. Type `%%code` at the top of the cell.
3. Below this, specify the code action you'd like - for example, **Load my_data.csv into a pandas dataframe.**
4. Execute the cell, and review the generated code snippet.

Customizing output and language settings

1. Use the `%set_output` command to change the default for how magic commands provide output. The options can be viewed by running `%set_output?`
2. Choose where to place the generated code, from options like
 - current cell
 - new cell
 - cell output
 - into a variable

Advanced commands for data operations

`%describe`, `%%add_comments`, and `%%fix_errors`

1. Use `%describe DataFrameName` in a new cell to obtain an overview of a specific dataframe.
2. To add comments to a code cell for better readability, type `%%add_comments` to the top of the cell you want to annotate and then execute. Be sure to validate the code is correct
3. For code error fixing, type `%%fix_errors` at the top of the cell that contained an error and execute it.

Privacy and security settings

1. By default, your privacy configuration shares previous messages sent to and from the Language Learning Model (LLM). However, it doesn't share cell contents, outputs, or any schemas or sample data from data sources.
2. Use `%set_sharing_level` in a new cell to adjust the data shared with the AI processor.
3. For more detailed privacy settings, use `%configure_privacy_settings`.

Context and focus commands

Using `%pin`, `%new_task`, and other context commands

1. Use `%pin DataFrameName` to help the AI focus on specific dataframes.
2. To clear the AI to focus on a new task in your notebook, type `%new_task` followed by a task that you are about to undertake. This clears the execution history that copilot knows about to this point and can make future responses more relevant.

Related content

- [How to use Copilot Pane](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Use the Copilot for Data Science and Data Engineering chat panel (preview)

Article • 12/04/2024

ⓘ Important

This feature is in [preview](#).

Copilot for Data Science and Data Engineering notebooks is an AI assistant that helps you analyze and visualize data. It works with lakehouse tables, Power BI Datasets, and pandas/spark dataframes, providing answers and code snippets directly in the notebook. The most effective way of using Copilot is to load your data as a dataframe. You can use the chat panel to ask your questions, and the AI provides responses or code to copy into your notebook. It understands your data's schema and metadata, and if data is loaded into a dataframe, it has awareness of the data inside of the data frame as well. You can ask Copilot to provide insights on data, create code for visualizations, or provide code for data transformations, and it recognizes file names for easy reference. Copilot streamlines data analysis by eliminating complex coding.

ⓘ Note

- Your administrator needs to enable the tenant switch before you start using Copilot. See the article [Copilot tenant settings](#) for details.
- Your F64 or P1 capacity needs to be in one of the regions listed in this article, [Fabric region availability](#).
- If your tenant or capacity is outside the US or France, Copilot is disabled by default unless your Fabric tenant admin enables the [Data sent to Azure](#), [OpenAI can be processed outside your tenant's geographic region, compliance boundary, or national cloud instance](#) tenant setting in the Fabric Admin portal.
- Copilot in Microsoft Fabric isn't supported on trial SKUs. Only paid SKUs (F64 or higher, or P1 or higher) are supported.
- Copilot in Fabric is currently rolling out in public preview and is expected to be available for all customers by end of March 2024.
- See the article [Overview of Copilot in Fabric and Power BI](#) for more information.

Azure OpenAI enablement

- Azure OpenAI must be enabled within Fabric at the tenant level.

⚠ Note

If your workspace is provisioned in a region without GPU capacity, and your data is not enabled to flow cross-geo, Copilot will not function properly and you will see errors.

Successful execution of Chat-magics installation cell

1. To use the Copilot pane, the installation cell for Chat-magics must successfully execute within your Spark session.

The screenshot shows a Jupyter Notebook interface with a sidebar containing a 'Lakehouses' section. A red box highlights a code cell in the main area:

```
1 #Run this cell to install the required packages for Copilot
2 %pip install https://aka.ms/chat-magics-0.0.0-py3-none-any.whl
3 %load_ext chat_magics
4
5
```

Below the cell, a message indicates it was executed in 15 seconds. The notebook also includes sections for 'Introduction' and a list of main steps in the workflow.

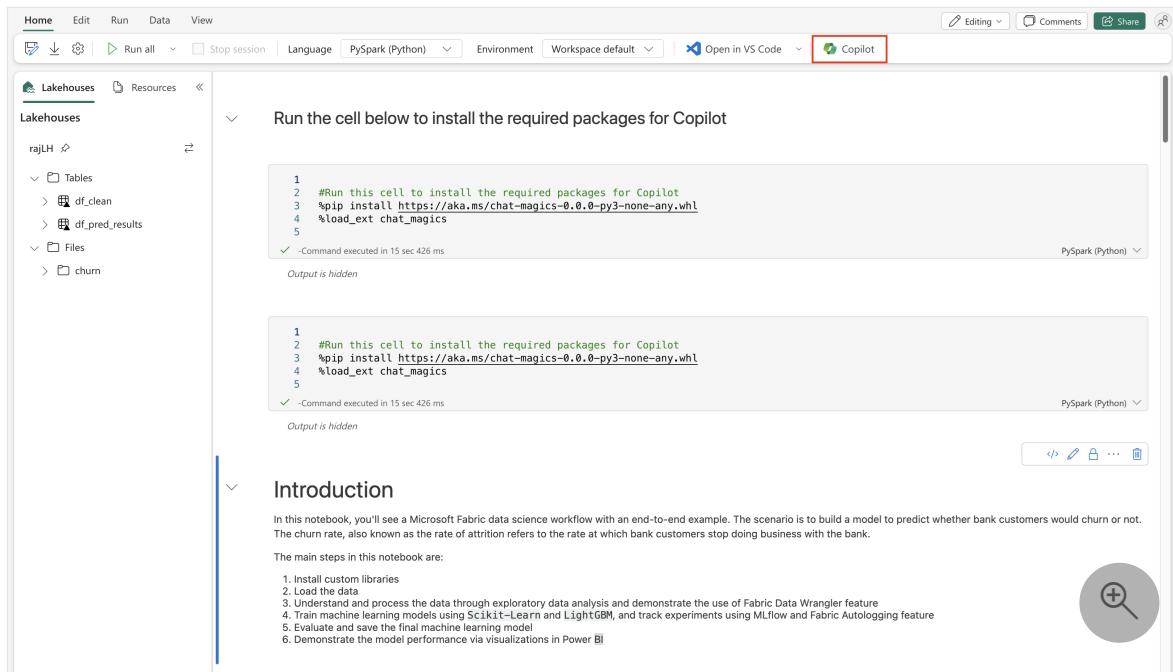
ⓘ Important

If your Spark session terminates, the context for Chat-magics will also terminate, also wiping the context for the Copilot pane.

2. Verify that all these conditions are met before proceeding with the Copilot chat pane.

Open Copilot chat panel inside the notebook

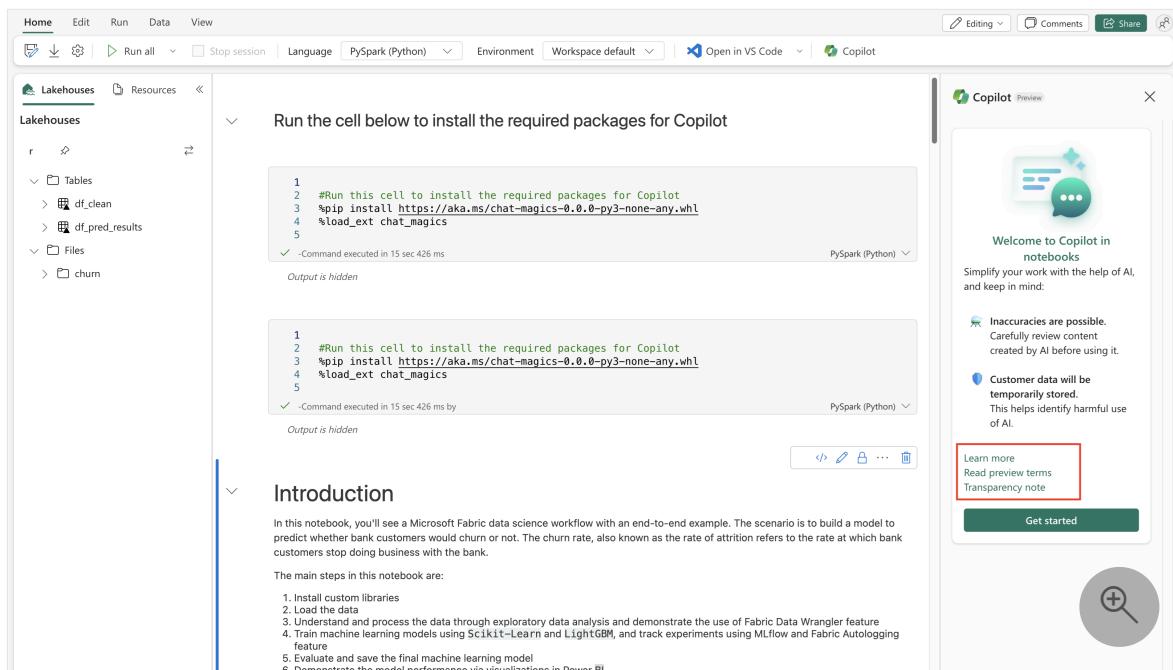
1. Select the Copilot button on the notebook ribbon.



2. To open Copilot, select the **Copilot** button at the top of the notebook.

3. The Copilot chat panel opens on the right side of your notebook.

4. A panel opens to provide overview information and helpful links.



Key capabilities

- **AI assistance:** Generate code, query data, and get suggestions to accelerate your workflow.

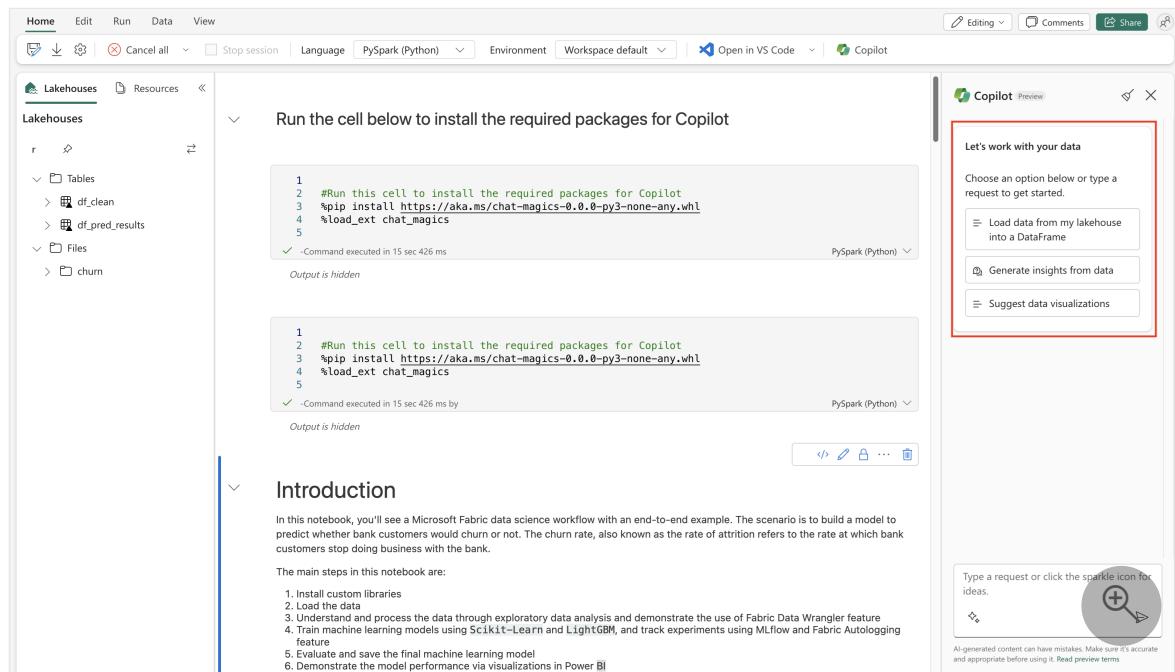
- **Data insights:** Quick data analysis and visualization capabilities.
- **Explanations:** Copilot can provide natural language explanations of notebook cells, and can provide an overview for notebook activity as it runs.
- **Fixing errors:** Copilot can also fix notebook run errors as they arise. Copilot shares context with the notebook cells (executed output) and can provide helpful suggestions.

Important notices

- **Inaccuracies:** Potential for inaccuracies exists. Review AI-generated content carefully.
- **Data storage:** Customer data is temporarily stored, to identify harmful use of AI.

Getting started with Copilot chat in notebooks

1. Copilot for Data Science and Data Engineering offers helpful starter prompts to get started. For example, "Load data from my lakehouse into a dataframe", or "Generate insights from data".



2. Each of these selections outputs chat text in the text panel. As the user, you must fill out the specific details of the data you'd like to use.
3. You can then input any type of request you have in the chat box.

Regular usage of the Copilot chat panel

- The more specifically you describe your goals in your chat panel entries, the more accurate the Copilot responses.
- You can "copy" or "insert" code from the chat panel. At the top of each code block, two buttons allow input of items directly into the notebook.
- To clear your conversation, select the  icon at the top to remove your conversation from the pane. It clears the pane of any input or output, but the context remains in the session until it ends.
- Configure the Copilot privacy settings with the %configure_privacy_settings command, or the %set_sharing_level command in the Chat-magics library.
- Transparency: Read our Transparency Note for details on data and algorithm use.

Related content

- [How to use Chat-magics](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Transform and enrich data seamlessly with AI functions (Preview)

Article • 03/05/2025

ⓘ Important

This feature is in [preview](#).

With Microsoft Fabric, all business professionals—from developers to analysts—can derive more value from their enterprise data through Generative AI, using experiences like [Copilot](#) and [Fabric data agents](#). Thanks to a new set of AI functions for data engineering, Fabric users can now harness the power of industry-leading large language models (LLMs) to transform and enrich data seamlessly.

AI functions harness the power of GenAI for summarization, classification, text generation, and so much more—all with a single line of code:

- **Calculate similarity with `ai.similarity`**: Compare the meaning of input text with a single common text value, or with corresponding text values in another column.
- **Categorize text with `ai.classify`**: Classify input text values according to labels you choose.
- **Detect sentiment with `ai.analyze_sentiment`**: Identify the emotional state expressed by input text.
- **Extract entities with `ai.extract`**: Find and extract specific types of information from input text, for example locations or names.
- **Fix grammar with `ai.fix_grammar`**: Correct the spelling, grammar, and punctuation of input text.
- **Summarize text with `ai.summarize`**: Get summaries of input text.
- **Translate text with `ai.translate`**: Translate input text into another language.
- **Answer custom user prompts with `ai.generate_response`**: Generate responses based on your own instructions.

It's seamless to incorporate these functions as part of data-science and data-engineering workflows, whether you're working with pandas or Spark. There is no detailed configuration, no complex infrastructure management, and no specific technical expertise needed.

Prerequisites

- To use AI functions with Fabric's built-in AI endpoint, your administrator needs to enable [the tenant switch for Copilot and other features powered by Azure OpenAI](#).
- You also need an F64 or higher SKU or a P SKU. With a smaller capacity resource, you need to provide AI functions with your own Azure OpenAI resource [using custom configurations](#).
- Depending on your location, you may need to enable a tenant setting for cross-geo processing. Learn more [here](#).

ⓘ Note

- AI functions are supported in the [Fabric 1.3 runtime](#) and higher.
- By default, AI functions are currently powered by the **gpt-3.5-turbo (0125)** model. To learn more about billing and consumption rates, visit [this article](#).
- Although the underlying model can handle several languages, most of the AI functions are optimized for use on English-language texts.
- During the initial rollout of AI functions, users are temporarily limited to 1,000 requests per minute with Fabric's built-in AI endpoint.

Getting started with AI functions

Use of the AI functions library in a Fabric notebook currently requires certain custom packages. The following code installs and imports those packages. Afterward, you can use AI functions with pandas or PySpark, depending on your preference.

This code cell installs the AI functions library and its dependencies.

⚠ Warning

The PySpark configuration cell takes a few minutes to finish executing. We appreciate your patience.

pandas

```
# Install fixed version of packages
%pip install -q openai==1.30
%pip install -q --force-reinstall httpx==0.27.0

# Install latest version of SynapseML-core
%pip install -q --force-reinstall
https://mmlspark.blob.core.windows.net/pip/1.0.9/synapseml_core-1.0.9-
```

```
py2.py3-none-any.whl
```

```
# Install SynapseML-Internal .whl with AI functions library from blob storage:  
%pip install -q --force-reinstall  
https://mmlspark.blob.core.windows.net/pip/1.0.10.0-spark3.4-6-0cb46b55-SNAPSHOT/synapseml_internal-1.0.10.0.dev1-py2.py3-none-any.whl
```

This code cell imports the AI functions library and its dependencies. The pandas cell also imports an optional Python library to display progress bars that track the status of every AI function call.

```
pandas
```

```
# Required imports  
import synapse.ml.aifunc as aifunc  
import pandas as pd  
import openai  
  
# Optional import for progress bars  
from tqdm.auto import tqdm  
tqdm.pandas()
```

Applying AI functions

Each of the following functions allows you to invoke Fabric's built-in AI endpoint to transform and enrich data with a single line of code. You can use AI functions to analyze pandas DataFrames or Spark DataFrames.

Tip

To learn about customizing the configuration of AI functions, visit [this article](#).

Calculate similarity with `ai.similarity`

The `ai.similarity` function invokes AI to compare input text values with a single common text value, or with pairwise text values in another column. The output similarity scores are relative, and they can range from -1 (opposites) to 1 (identical). A score of 0 indicates that the values are completely unrelated in meaning. For more detailed instructions about the use of `ai.similarity`, visit [this article](#).

Sample usage

```
pandas
```

```
# This code uses AI. Always review output for mistakes.  
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/  
  
df = pd.DataFrame([  
    ("Bill Gates", "Microsoft"),  
    ("Satya Nadella", "Toyota"),  
    ("Joan of Arc", "Nike")  
], columns=["names", "companies"])  
  
df["similarity"] = df["names"].ai.similarity(df["companies"])  
display(df)
```

Categorize text with `ai.classify`

The `ai.classify` function invokes AI to categorize input text according to custom labels you choose. For more information about the use of `ai.classify`, visit [this article](#).

Sample usage

```
pandas
```

```
# This code uses AI. Always review output for mistakes.  
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/  
  
df = pd.DataFrame([  
    "This duvet, lovingly hand-crafted from all-natural fabric, is perfect for a good night's sleep.",  
    "Tired of friends judging your baking? With these handy-dandy measuring cups, you'll create culinary delights.",  
    "Enjoy this *BRAND NEW CAR!* A compact SUV perfect for the professional commuter!"  
], columns=["descriptions"])  
  
df["category"] = df['descriptions'].ai.classify("kitchen", "bedroom", "garage", "other")  
display(df)
```

Detect sentiment with `ai.analyze_sentiment`

The `ai.analyze_sentiment` function invokes AI to identify whether the emotional state expressed by input text is positive, negative, mixed, or neutral. If AI can't make this determination, the output is left blank. For more detailed instructions about the use of `ai.analyze_sentiment`, visit [this article](#).

Sample usage

pandas

```
# This code uses AI. Always review output for mistakes.  
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/  
  
df = pd.DataFrame([  
    "The cleaning spray permanently stained my beautiful kitchen counter. Never again!",  
    "I used this sunscreen on my vacation to Florida, and I didn't get burned at all. Would recommend.",  
    "I'm torn about this speaker system. The sound was high quality, though it didn't connect to my roommate's phone.",  
    "The umbrella is OK, I guess."  
], columns=["reviews"])  
  
df["sentiment"] = df["reviews"].ai.analyze_sentiment()  
display(df)
```

Extract entities with `ai.extract`

The `ai.extract` function invokes AI to scan input text and extract specific types of information designated by labels you choose—for example, locations or names. For more detailed instructions about the use of `ai.extract`, visit [this article](#).

Sample usage

pandas

```
# This code uses AI. Always review output for mistakes.  
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/
```

```
df = pd.DataFrame([
    "MJ Lee lives in Tuscon, AZ, and works as a software engineer
for Microsoft.",
    "Kris Turner, a nurse at NYU Langone, is a resident of Jersey
City, New Jersey."
], columns=["descriptions"])

df_entities = df["descriptions"].ai.extract("name", "profession",
"city")
display(df_entities)
```

Fix grammar with `ai.fix_grammar`

The `ai.fix_grammar` function invokes AI to correct the spelling, grammar, and punctuation of input text. For more detailed instructions about the use of `ai.fix_grammar`, visit [this article](#).

Sample usage

pandas

```
# This code uses AI. Always review output for mistakes.
# Read terms: https://azure.microsoft.com/support/legal/preview-
supplemental-terms/

df = pd.DataFrame([
    "There are an error here.",
    "She and me go weigh back. We used to hang out every weeks.",
    "The big picture are right, but you're details is all wrong."
], columns=["text"])

df["corrections"] = df["text"].ai.fix_grammar()
display(df)
```

Summarize text with `ai.summarize`

The `ai.summarize` function invokes AI to generate summaries of input text (either values from a single column of a DataFrame, or row values across all the columns). For more detailed instructions about the use of `ai.summarize`, visit [this dedicated article](#).

Sample usage

pandas

```
# This code uses AI. Always review output for mistakes.
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/

df= pd.DataFrame([
    ("Microsoft Teams", "2017",
     """
        The ultimate messaging app for your organization—a workspace for
        real-time
        collaboration and communication, meetings, file and app sharing,
        and even the
        occasional emoji! All in one place, all in the open, all
        accessible to everyone.
    """),
    ("Microsoft Fabric", "2023",
     """
        An enterprise-ready, end-to-end analytics platform that unifies
        data movement,
        data processing, ingestion, transformation, and report building
        into a seamless,
        user-friendly SaaS experience. Transform raw data into
        actionable insights.
    """)
], columns=["product", "release_year", "description"])

df["summaries"] = df["description"].ai.summarize()
display(df)
```

Translate text with `ai.translate`

The `ai.translate` function invokes AI to translate input text to a new language of your choice. For more detailed instructions about the use of `ai.translate`, visit [this article](#).

Sample usage

pandas

```
# This code uses AI. Always review output for mistakes.
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/

df = pd.DataFrame([
    "Hello! How are you doing today?",
    "Tell me what you'd like to know, and I'll do my best to help.",
```

```
"The only thing we have to fear is fear itself."  
], columns=["text"])  
  
df["translations"] = df["text"].ai.translate("spanish")  
display(df)
```

Answer custom user prompts with `ai.generate_response`

The `ai.generate_response` function invokes AI to generate custom text based on your own instructions. For more detailed instructions about the use of `ai.generate_response`, visit [this article](#).

Sample usage

pandas

```
# This code uses AI. Always review output for mistakes.  
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/  
  
df = pd.DataFrame([  
    ("Scarves"),  
    ("Snow pants"),  
    ("Ski goggles")  
], columns=["product"])  
  
df["response"] = df.ai.generate_response("Write a short, punchy email  
subject line for a winter sale.")  
display(df)
```

Related content

- Calculate similarity with [ai.similarity](#).
- Detect sentiment with [ai.analyze_sentiment](#).
- Categorize text with [ai.classify](#).
- Extract entities with [ai_extract](#).
- Fix grammar with [ai.fix_grammar](#).
- Summarize text with [ai.summarize](#).
- Translate text with [ai.translate](#).
- Answer custom user prompts with [ai.generate_response](#).
- Learn how to [customize the configuration of AI functions](#).

- Did we miss a feature you need? Suggest it on the [Fabric Ideas forum](#).
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Calculate similarity with the `ai.similarity` function

Article • 03/05/2025

The `ai.similarity` function uses Generative AI to compare two string expressions and then calculate a semantic similarity score—all with a single line of code. You can compare text values from one column of a DataFrame with a single common text value or with pairwise text values in another column.

AI functions turbocharge data engineering by putting the power of Fabric's built-in large languages models into your hands. To learn more, visit [this overview article](#).

Important

This feature is in [preview](#), for use in the [Fabric 1.3 runtime](#) and higher.

- Review the prerequisites in [this overview article](#), including the [library installations](#) that are temporarily required to use AI functions.
- By default, AI functions are currently powered by the **gpt-3.5-turbo (0125)** model. To learn more about billing and consumption rates, visit [this article](#).
- Although the underlying model can handle several languages, most of the AI functions are optimized for use on English-language texts.
- During the initial rollout of AI functions, users are temporarily limited to 1,000 requests per minute with Fabric's built-in AI endpoint.

Use `ai.similarity` with pandas

The `ai.similarity` function extends the [pandas Series](#) class. Call the function on a [pandas DataFrame](#) text column to calculate the semantic similarity of each input row with respect to a single common text value. Alternatively, the function can calculate the semantic similarity of each row with respect to corresponding pairwise values in another column that has the same dimensions as the input column.

The function returns a pandas Series containing similarity scores, which can be stored in a new DataFrame column.

Syntax

Comparing with a single value

Python

```
df["similarity"] = df["col1"].ai.similarity("value")
```

Parameters

[+] Expand table

Name	Description
other Required	Either a string that contains a single common text value, which is used to compute similarity scores for each input row, OR another pandas Series with the same dimensions as the input, which contains text values that are used to compute pairwise similarity scores for each input row.

Returns

A [pandas Series](#) that contains similarity scores for each input text row. The output similarity scores are relative, and they're best used for ranking. Scores can range from -1 (opposites) to 1 (identical). A score of 0 indicates that the values are unrelated in meaning.

Example

Comparing with a single value

Python

```
# This code uses AI. Always review output for mistakes.  
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/  
  
df = pd.DataFrame([  
    ("Bill Gates"),  
    ("Satya Nadella"),  
    ("Joan of Arc")  
], columns=[ "name" ])
```

```
df["similarity"] = df["name"].ai.similarity("Microsoft")
display(df)
```

Use `ai.similarity` with PySpark

The `ai.similarity` function is also available for [Spark DataFrames](#). You must specify the name of an existing input column as a parameter. You must also specify a single common text value for comparisons, or the name of another column for pairwise comparisons.

The function returns a new DataFrame, with similarity scores for each row of input text stored in an output column.

Syntax

Comparing with a single value

Python

```
df.ai.similarity(input_col="col1", other="value",
                  output_col="similarity")
```

Parameters

[+] Expand table

Name	Description
<code>input_col</code> Required	A string that contains the name of an existing column with input text values to be used for computing similarity scores.
<code>other</code> or <code>other_col</code> Required	Only one of these parameters is required. The <code>other</code> parameter is a string that contains a single common text value used to compute similarity scores with respect to each row of input. The <code>other_col</code> parameter is a string that designates the name of a second existing column, with text values used to compute pairwise similarity scores.
<code>output_col</code> Optional	A string that contains the name of a new column to store calculated similarity scores for each input text row. If this parameter isn't set, a default name is generated for the output column.

Name	Description
<code>error_col</code> Optional	A string that contains the name of a new column that stores any OpenAI errors that result from processing each input text row. If this parameter isn't set, a default name is generated for the error column. If an input row has no errors, this column has a <code>null</code> value.

Returns

A [Spark DataFrame](#) with a new column that contains generated similarity scores for each input text row. The output similarity scores are relative, and they're best used for ranking. Scores can range from -1 (opposites) to 1 (identical). A score of 0 indicates that the values are unrelated in meaning.

Example

Comparing with a single value

Python

```
# This code uses AI. Always review output for mistakes.
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/

df = spark.createDataFrame([
    ("Bill Gates",),
    ("Sayta Nadella",),
    ("Joan of Arc",)
], ["names"])

similarity = df.ai.similarity(input_col="names", other="Microsoft",
                               output_col="similarity")
display(similarity)
```

Related content

- Categorize text with [ai.classify](#).
- Detect sentiment with [ai.analyze_sentiment](#).
- Extract entities with [ai_extract](#).
- Fix grammar with [ai.fix_grammar](#).
- Summarize text with [ai.summarize](#).
- Translate text with [ai.translate](#).

- Answer custom user prompts with [ai.generate_response](#).
 - Learn more about the full set of AI functions [here](#).
 - Learn how to customize the configuration of AI functions [here](#).
 - Did we miss a feature you need? Suggest it on the [Fabric Ideas forum ↗](#).
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Categorize text with the `ai.classify` function

Article • 03/05/2025

The `ai.classify` function uses Generative AI to categorize input text according to custom labels you choose—all with a single line of code.

AI functions turbocharge data engineering by putting the power of Fabric's built-in large languages models into your hands. To learn more, visit [this overview article](#).

Important

This feature is in [preview](#), for use in the [Fabric 1.3 runtime](#) and higher.

- Review the prerequisites in [this overview article](#), including the [library installations](#) that are temporarily required to use AI functions.
- By default, AI functions are currently powered by the **gpt-3.5-turbo (0125)** model. To learn more about billing and consumption rates, visit [this article](#).
- Although the underlying model can handle several languages, most of the AI functions are optimized for use on English-language texts.
- During the initial rollout of AI functions, users are temporarily limited to 1,000 requests per minute with Fabric's built-in AI endpoint.

Tip

We recommend using the `ai.classify` function with at least two input labels.

Use `ai.classify` with pandas

The `ai.classify` function extends the [pandas Series](#) class. Call the function on a text column of a [pandas DataFrame](#) to assign user-provided labels to each input row.

The function returns a pandas Series that contains classification labels, which can be stored in a new DataFrame column.

Syntax

Python

```
df["classification"] = df["text"].ai.classify("category1", "category2",
"category3")
```

Parameters

[+] Expand table

Name	Description
labels Required	One or more strings representing the set of classification labels to be matched to input text values.

Returns

The function returns a [pandas Series](#) that contains a classification label for each input text row. If a text value can't be classified, the corresponding label is `null`.

Example

Python

```
# This code uses AI. Always review output for mistakes.
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/

df = pd.DataFrame([
    "This duvet, lovingly hand-crafted from all-natural fabric, is
perfect for a good night's sleep.",
    "Tired of friends judging your baking? With these handy-dandy
measuring cups, you'll create culinary delights.",
    "Enjoy this *BRAND NEW CAR!* A compact SUV perfect for the
professional commuter!",
], columns=["descriptions"])

df["category"] = df['descriptions'].ai.classify("kitchen", "bedroom",
"garage", "other")
display(df)
```

Use `ai.classify` with PySpark

The `ai.classify` function is also available for [Spark DataFrames](#). The name of an existing input column must be specified as a parameter, along with a list of classification

labels.

The function returns a new DataFrame, with labels that match each row of input text stored in an output column.

Syntax

Python

```
df.ai.classify(labels=["category1", "category2", "category3"],  
    input_col="text", output_col="classification")
```

Parameters

[+] Expand table

Name	Description
labels Required	An array ↗ of strings ↗ that represents the set of classification labels to be matched to text values in the input column.
input_col Required	A string ↗ that contains the name of an existing column with input text values to be classified according to the custom labels.
output_col Optional	A string ↗ that contains the name of a new column to store a classification label for each input text row. If this parameter isn't set, a default name is generated for the output column.
error_col Optional	A string ↗ that contains the name of a new column. The new column stores any OpenAI errors that result from processing each row of input text. If this parameter isn't set, a default name is generated for the error column. If there are no errors for a row of input, the value in this column is <code>null</code> .

Returns

The function returns a [Spark DataFrame ↗](#) with a new column that contains classification labels that match each input text row. If a text value can't be classified, the corresponding label is `null`.

Example

Python

```
# This code uses AI. Always review output for mistakes.  
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/  
  
df = spark.createDataFrame([  
    ("This duvet, lovingly hand-crafted from all-natural fabric, is perfect for a good night's sleep.",),  
    ("Tired of friends judging your baking? With these handy-dandy measuring cups, you'll create culinary delights.",),  
    ("Enjoy this *BRAND NEW CAR!* A compact SUV perfect for the professional commuter!"),  
], ["descriptions"])  
  
categories = df.ai.classify(labels=["kitchen", "bedroom", "garage", "other"], input_col="descriptions", output_col="categories")  
display(categories)
```

Related content

- Calculate similarity with [ai.similarity](#).
- Detect sentiment with [ai.analyze_sentiment](#).
- Extract entities with [ai_extract](#).
- Fix grammar with [ai.fix_grammar](#).
- Summarize text with [ai.summarize](#).
- Translate text with [ai.translate](#).
- Answer custom user prompts with [ai.generate_response](#).
- Learn more about the full set of AI functions [here](#).
- Learn how to customize the configuration of AI functions [here](#).
- Did we miss a feature you need? Suggest it on the [Fabric Ideas forum](#) ↗.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) ↗ | [Ask the community](#) ↗

Detect sentiment with the `ai.analyze_sentiment` function

Article • 03/05/2025

The `ai.analyze_sentiment` function uses Generative AI to detect whether the emotional state expressed by input text is positive, negative, mixed, or neutral—all with a single line of code. If the function can't determine the sentiment, it leaves the output blank.

AI functions turbocharge data engineering by putting the power of Fabric's built-in large languages models into your hands. To learn more, visit [this overview article](#).

ⓘ Important

This feature is in [preview](#), for use in the [Fabric 1.3 runtime](#) and higher.

- Review the prerequisites in [this overview article](#), including the [library installations](#) that are temporarily required to use AI functions.
- By default, AI functions are currently powered by the `gpt-3.5-turbo (0125)` model. To learn more about billing and consumption rates, visit [this article](#).
- Although the underlying model can handle several languages, most of the AI functions are optimized for use on English-language texts.
- During the initial rollout of AI functions, users are temporarily limited to 1,000 requests per minute with Fabric's built-in AI endpoint.

Use `ai.analyze_sentiment` with pandas

The `ai.analyze_sentiment` function extends the [pandas Series](#) class. Call the function on [pandas DataFrame](#) text column to detect the sentiment of each input row.

The function returns a pandas Series that contains sentiment labels, which can be stored in a new column of the DataFrame.

Syntax

Python

```
df["sentiment"] = df["text"].ai.analyze_sentiment()
```

Parameters

None

Returns

The function returns a [pandas Series](#) that contains sentiment labels for each input text row. Each sentiment label is `positive`, `negative`, `neutral`, or `mixed`. If a sentiment can't be determined, the return value is `null`.

Example

Python

```
# This code uses AI. Always review output for mistakes.  
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/  
  
df = pd.DataFrame([  
    "The cleaning spray permanently stained my beautiful kitchen  
counter. Never again!",  
    "I used this sunscreen on my vacation to Florida, and I didn't get  
burned at all. Would recommend.",  
    "I'm torn about this speaker system. The sound was high quality,  
though it didn't connect to my roommate's phone.",  
    "The umbrella is OK, I guess."  
], columns=["reviews"])  
  
df["sentiment"] = df["reviews"].ai.analyze_sentiment()  
display(df)
```

Use `ai.analyze_sentiment` with PySpark

The `ai.analyze_sentiment` function is also available for [Spark DataFrames](#). The name of an existing input column must be specified as a parameter.

The function returns a new DataFrame, with sentiment labels for each input text row stored in an output column.

Syntax

Python

```
df.ai.analyze_sentiment(input_col="text", output_col="sentiment")
```

Parameters

[+] Expand table

Name	Description
<code>input_col</code> Required	A string that contains the name of an existing column with input text values to be analyzed for sentiment.
<code>output_col</code> Optional	A string that contains the name of a new column to store the sentiment label for each row of input text. If this parameter isn't set, a default name is generated for the output column.
<code>error_col</code> Optional	A string that contains the name of a new column to store any OpenAI errors that result from processing each row of input text. If this parameter isn't set, a default name is generated for the error column. If an input row has no errors, the value in this column is <code>null</code> .

Returns

A [Spark DataFrame](#) with a new column containing sentiment labels that match each row of text in the input column. Each sentiment label is `positive`, `negative`, `neutral`, or `mixed`. If a sentiment can't be determined, the return value is `null`.

Example

Python

```
# This code uses AI. Always review output for mistakes.
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/

df = spark.createDataFrame([
    ("The cleaning spray permanently stained my beautiful kitchen counter. Never again!"),
    ("I used this sunscreen on my vacation to Florida, and I didn't get burned at all. Would recommend."),
    ("I'm torn about this speaker system. The sound was high quality, though it didn't connect to my roommate's phone."),
    ("The umbrella is OK, I guess."),
], ["reviews"])

sentiment = df.ai.analyze_sentiment(input_col="reviews",
```

```
output_col="sentiment")  
display(sentiment)
```

Related content

- Calculate similarity with [ai.similarity](#).
- Categorize text with [ai.classify](#).
- Extract entities with [ai_extract](#).
- Fix grammar with [ai.fix_grammar](#).
- Summarize text with [ai.summarize](#).
- Translate text with [ai.translate](#).
- Answer custom user prompts with [ai.generate_response](#).
- Learn more about the full set of AI functions [here](#).
- Learn how to customize the configuration of AI functions [here](#).
- Did we miss a feature you need? Suggest it on the [Fabric Ideas forum](#).

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Extract entities with the `ai.extract` function

Article • 03/05/2025

The `ai.extract` function uses Generative AI to scan input text and extract specific types of information designated by labels you choose—for example, locations or names—all with a single line of code.

AI functions turbocharge data engineering by putting the power of Fabric's built-in large languages models into your hands. To learn more, visit [this overview article](#).

ⓘ Important

This feature is in [preview](#), for use in the [Fabric 1.3 runtime](#) and higher.

- Review the prerequisites in [this overview article](#), including the [library installations](#) that are temporarily required to use AI functions.
- By default, AI functions are currently powered by the `gpt-3.5-turbo (0125)` model. To learn more about billing and consumption rates, visit [this article](#).
- Although the underlying model can handle several languages, most of the AI functions are optimized for use on English-language texts.
- During the initial rollout of AI functions, users are temporarily limited to 1,000 requests per minute with Fabric's built-in AI endpoint.

Use `ai.extract` with pandas

The `ai.extract` function extends the [pandas Series](#) class. Call the function on a [pandas DataFrame](#) text column to extract custom entity types from each row of input.

Unlike other AI functions, `ai.extract` returns a pandas DataFrame, instead of a Series, with a separate column for each specified entity type that contains extracted values for each input row.

Syntax

Python

```
df_entities = df["text"].ai.extract("entity1", "entity2", "entity3")
```

Parameters

[\[+\] Expand table](#)

Name	Description
labels Required	One or more strings representing the set of entity types to be extracted from the input text values.

Returns

The function returns a [pandas DataFrame](#) with a column for each specified entity type. The column or columns contain the entities extracted for each row of input text. If the function identifies more than one match for a given entity, it returns only one of those matches. If no match is found, the result is `null`.

Example

Python

```
# This code uses AI. Always review output for mistakes.
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/

df = pd.DataFrame([
    "MJ Lee lives in Tuscon, AZ, and works as a software engineer for Microsoft.",
    "Kris Turner, a nurse at NYU Langone, is a resident of Jersey City, New Jersey."]
    , columns=["descriptions"])

df_entities = df["descriptions"].ai.extract("name", "profession", "city")
display(df_entities)
```

Use `ai.extract` with PySpark

The `ai.extract` function is also available for [Spark DataFrames](#). The name of an existing input column must be specified as a parameter, along with a list of entity types to extract from each row of text.

The function returns a new DataFrame, with a separate column for each specified entity type that contains extracted values for each input row.

Syntax

Python

```
df.ai.extract(labels=["entity1", "entity2", "entity3"], input_col="text")
```

Parameters

[Expand table

Name	Description
<code>labels</code> Required	An array ↗ of strings ↗ that represents the set of entity types to be extracted from the text values in the input column.
<code>input_col</code> Required	A string ↗ that contains the name of an existing column with input text values to be scanned for the custom entities.
<code>error_col</code> Optional	A string ↗ that contains the name of a new column to store any OpenAI errors that result from processing each input text row. If this parameter isn't set, a default name is generated for the error column. If an input row has no errors, the value in this column is <code>null</code> .

Returns

The function returns a [Spark DataFrame ↗](#) with a new column for each specified entity type. The column or columns contain the entities extracted for each row of input text. If the function identifies more than one match for a given entity, it returns only one of those matches. If no match is found, the result is `null`.

Example

Python

```
# This code uses AI. Always review output for mistakes.  
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/  
  
df = spark.createDataFrame([  
    ("MJ Lee lives in Tuscon, AZ, and works as a software engineer for Microsoft."),
```

```
("Kris Turner, a nurse at NYU Langone, is a resident of Jersey City,  
New Jersey.",)  
], ["descriptions"])
```

```
df_entities = df.ai.extract(labels=["name", "profession", "city"],  
input_col="descriptions")  
display(df_entities)
```

Related content

- Calculate similarity with [ai.similarity](#).
- Categorize text with [ai.classify](#).
- Detect sentiment with [ai.analyze_sentiment](#).
- Fix grammar with [ai.fix_grammar](#).
- Summarize text with [ai.summarize](#).
- Translate text with [ai.translate](#).
- Answer custom user prompts with [ai.generate_response](#).
- Learn more about the full set of AI functions [here](#).
- Learn how to customize the configuration of AI functions [here](#).
- Did we miss a feature you need? Suggest it on the [Fabric Ideas forum](#) ↗.

Feedback

Was this page helpful?



[Provide product feedback](#) ↗ | [Ask the community](#) ↗

Fix grammar with the `ai.fix_grammar` function

Article • 03/05/2025

The `ai.fix_grammar` function uses Generative AI to correct the spelling, grammar, and punctuation of input text—all with a single line of code.

AI functions turbocharge data engineering by putting the power of Fabric's built-in large languages models into your hands. To learn more, visit [this overview article](#).

Important

This feature is in [preview](#), for use in the [Fabric 1.3 runtime](#) and higher.

- Review the prerequisites in [this overview article](#), including the [library installations](#) that are temporarily required to use AI functions.
- By default, AI functions are currently powered by the **gpt-3.5-turbo (0125)** model. To learn more about billing and consumption rates, visit [this article](#).
- Although the underlying model can handle several languages, most of the AI functions are optimized for use on English-language texts.
- During the initial rollout of AI functions, users are temporarily limited to 1,000 requests per minute with Fabric's built-in AI endpoint.

Use `ai.fix_grammar` with pandas

The `ai.fix_grammar` function extends the [pandas Series](#) class. Call the function on a [pandas DataFrame](#) text column to correct the spelling, grammar, and punctuation of each row of input.

The function returns a pandas Series that contains corrected text values, which can be stored in a new DataFrame column.

Syntax

Python

```
df["corrections"] = df["text"].ai.fix_grammar()
```

Parameters

None

Returns

The function returns a [pandas Series](#) that contains corrected text for each input text row. If the input text is `null`, the result is `null`.

Example

Python

```
# This code uses AI. Always review output for mistakes.  
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/  
  
df = pd.DataFrame([  
    "There are an error here.",  
    "She and me go weigh back. We used to hang out every weeks.",  
    "The big picture are right, but you're details is all wrong."  
, columns=["text"])  
  
df["corrections"] = df["text"].ai.fix_grammar()  
display(df)
```

Use `ai.fix_grammar` with PySpark

The `ai.fix_grammar` function is also available for [Spark DataFrames](#). The name of an existing input column must be specified as a parameter.

The function returns a new DataFrame, with corrected text for each input text row stored in an output column.

Syntax

Python

```
df.ai.fix_grammar(input_col="text", output_col="corrections")
```

Parameters

Name	Description
input_col Required	A string containing the name of an existing column with input text values to be corrected for spelling, grammar, and punctuation.
output_col Optional	A string containing the name of a new column to store corrected text for each row of input text. If this parameter isn't set, a default name is generated for the output column.
error_col Optional	A string containing the name of a new column to store any OpenAI errors that result from processing each row of input text. If this parameter isn't set, a default name is generated for the error column. If there are no errors for a row of input, the value in this column is <code>null</code> .

Returns

A [Spark DataFrame](#) with a new column containing corrected text for each row of text in the input column. If the input text is `null`, the result is `null`.

Example

Python

```
# This code uses AI. Always review output for mistakes.
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/

df = spark.createDataFrame([
    ("There are an error here.",),
    ("She and me go weigh back. We used to hang out every weeks.",),
    ("The big picture are right, but you're details is all wrong.",)
], ["text"])

results = df.ai.fix_grammar(input_col="text", output_col="corrections")
display(results)
```

Related content

- Calculate similarity with [ai.similarity](#).
- Categorize text with [ai.classify](#).
- Detect sentiment with [ai.analyze_sentiment](#).
- Extract entities with [ai_extract](#).
- Summarize text with [ai.summarize](#).

- Translate text with [ai.translate](#).
 - Answer custom user prompts with [ai.generate_response](#).
 - Learn more about the full set of AI functions [here](#).
 - Learn how to customize the configuration of AI functions [here](#).
 - Did we miss a feature you need? Suggest it on the [Fabric Ideas forum](#).
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Summarize text with the `ai.summarize` function

Article • 03/05/2025

The `ai.summarize` function uses Generative AI to produce summaries of input text—either values from one column of a DataFrame or values across all the columns—with a single line of code.

AI functions turbocharge data engineering by putting the power of Fabric's built-in large languages models into your hands. To learn more, visit [this overview article](#).

Important

This feature is in [preview](#), for use in the [Fabric 1.3 runtime](#) and higher.

- Review the prerequisites in [this overview article](#), including the [library installations](#) that are temporarily required to use AI functions.
- By default, AI functions are currently powered by the `gpt-3.5-turbo (0125)` model. To learn more about billing and consumption rates, visit [this article](#).
- Although the underlying model can handle several languages, most of the AI functions are optimized for use on English-language texts.
- During the initial rollout of AI functions, users are temporarily limited to 1,000 requests per minute with Fabric's built-in AI endpoint.

Use `ai.summarize` with pandas

The `ai.summarize` function extends the [pandas Series](#) class. Call the function on a [pandas DataFrame](#) text column to summarize each row value from that column alone. Alternatively, you can call the `ai.summarize` function on an entire DataFrame, to summarize values across all the columns.

The function returns a pandas Series that contains summaries, which can be stored in a new DataFrame column.

Syntax

Summarizing values from a single column

Python

```
df["summaries"] = df["text"].ai.summarize()
```

Parameters

None

Returns

A [pandas Series](#) that contains summaries for each input text row. If the input text is `null`, the result is `null`.

Example

Summarizing values from a single column

Python

```
# This code uses AI. Always review output for mistakes.
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/

df= pd.DataFrame([
    ("Microsoft Teams", "2017",
     """
        The ultimate messaging app for your organization—a workspace for
        real-time
        collaboration and communication, meetings, file and app sharing,
        and even the
        occasional emoji! All in one place, all in the open, all
        accessible to everyone.
        """
    ),
    ("Microsoft Fabric", "2023",
     """
        An enterprise-ready, end-to-end analytics platform that unifies
        data movement,
        data processing, ingestion, transformation, and report building
        into a seamless,
        user-friendly SaaS experience. Transform raw data into
        actionable insights.
        """
    )
], columns=[ "product", "release_year", "description"])
```

```
df["summaries"] = df["description"].ai.summarize()  
display(df)
```

Use `ai.summarize` with PySpark

The `ai.summarize` function is also available for [Spark DataFrames](#). If you specify the name of an existing input column as a parameter, the function summarizes each value from that column alone. Otherwise, the function summarizes values across all columns of the DataFrame, row by row.

The function returns a new DataFrame with summaries for each input text row, from a single column or across all the columns, stored in an output column.

Syntax

Summarizing values from a single column

Python

```
df.ai.summarize(input_col="text", output_col="summaries")
```

Parameters

[] [Expand table](#)

Name	Description
<code>input_col</code> Optional	A string that contains the name of an existing column with input text values to summarize. If this parameter isn't set, the function summarizes values across all columns in the DataFrame, instead of values from a specific column.
<code>output_col</code> Optional	A string that contains the name of a new column to store summaries for each input text row. If this parameter isn't set, a default name is generated for the output column.
<code>error_col</code> Optional	A string that contains the name of a new column to store any OpenAI errors that result from processing each input text row. If this parameter isn't set, a default name is generated for the error column. If an input row has no errors, the value in this column is <code>null</code> .

Returns

A [Spark DataFrame](#) with a new column that contains summarized text for each input text row. If the input text is `null`, the result is `null`. If no input column is specified, the function summarizes values across all columns in the DataFrame.

Example

Summarizing values from a single column

Python

```
# This code uses AI. Always review output for mistakes.
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/

df = spark.createDataFrame([
    ("Microsoft Teams", "2017",
     """
        The ultimate messaging app for your organization—a workspace for
        real-time
        collaboration and communication, meetings, file and app sharing,
        and even the
        occasional emoji! All in one place, all in the open, all
        accessible to everyone.
        """),
    ("Microsoft Fabric", "2023",
     """
        An enterprise-ready, end-to-end analytics platform that unifies
        data movement,
        data processing, ingestion, transformation, and report building
        into a seamless,
        user-friendly SaaS experience. Transform raw data into
        actionable insights.
        """),
], ["product", "release_year", "description"])

summaries = df.ai.summarize(input_col="description",
                            output_col="summaries")
display(summaries)
```

Related content

- Calculate similarity with [ai.similarity](#).
- Categorize text with [ai.classify](#).
- Detect sentiment with [ai.analyze_sentiment](#).

- Extract entities with [ai_extract](#).
 - Fix grammar with [ai.fix_grammar](#).
 - Translate text with [ai.translate](#).
 - Answer custom user prompts with [ai.generate_response](#).
 - To learn more about the full set of AI functions, visit [this overview article](#).
 - Learn how to customize the configuration of AI functions [here](#).
 - Did we miss a feature you need? Suggest it on the [Fabric Ideas forum](#).
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Translate text with the `ai.translate` function

Article • 03/05/2025

The `ai.translate` function uses Generative AI to translate input text to a new language of your choice—all with a single line of code.

AI functions turbocharge data engineering by putting the power of Fabric's built-in large languages models into your hands. To learn more, visit [this overview article](#).

Important

This feature is in [preview](#), for use in the [Fabric 1.3 runtime](#) and higher.

- Review the prerequisites in [this overview article](#), including the [library installations](#) that are temporarily required to use AI functions.
- By default, AI functions are currently powered by the **gpt-3.5-turbo (0125)** model. To learn more about billing and consumption rates, visit [this article](#).
- Although the underlying model can handle several languages, most of the AI functions are optimized for use on English-language texts.
- During the initial rollout of AI functions, users are temporarily limited to 1,000 requests per minute with Fabric's built-in AI endpoint.

Tip

The `ai.translate` function was tested with 10 languages: **Czech, English, Finnish, French, German, Greek, Italian, Polish, Spanish, and Swedish**. Your results with other languages may vary.

Use `ai.translate` with pandas

The `ai.translate` function extends the [pandas Series](#) class. Call the function on a [pandas DataFrame](#) text column to translate each input row into a target language of your choosing.

The function returns a pandas Series that contains translations, which you can store in a new DataFrame column.

Syntax

Python

```
df["translations"] = df["text"].ai.translate("target_language")
```

Parameters

[+] Expand table

Name	Description
<code>to_lang</code> Required	A string representing the target language for text translations.

Returns

A [pandas Series](#) that contains translations for each row of input text. If the input text is `null`, the result is `null`.

Example

Python

```
# This code uses AI. Always review output for mistakes.
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/

df = pd.DataFrame([
    "Hello! How are you doing today?",
    "Tell me what you'd like to know, and I'll do my best to help.",
    "The only thing we have to fear is fear itself."
], columns=["text"])

df["translations"] = df["text"].ai.translate("spanish")
display(df)
```

Use `ai.translate` with PySpark

The `ai.translate` function is also available for [Spark DataFrames](#). You must specify an existing input column name as a parameter, along with a target language.

The function returns a new DataFrame, with translations for each input text row stored in an output column.

Syntax

Python

```
df.ai.translate(to_lang="spanish", input_col="text",
output_col="translations")
```

Parameters

[+] Expand table

Name	Description
to_lang Required	A string that represents the target language for text translations.
input_col Required	A string that contains the name of an existing column with input text values to be translated.
output_col Optional	A string that contains the name of a new column that stores translations for each input text row. If this parameter isn't set, a default name is generated for the output column.
error_col Optional	A string that contains the name of a new column that stores any OpenAI errors that result from processing each input text row. If this parameter isn't set, a default name is generated for the error column. If an input row has no errors, the value in this column is <code>null</code> .

Returns

A [Spark DataFrame](#) with a new column that contains translations for the text in the input column row. If the input text is `null`, the result is `null`.

Example

Python

```
# This code uses AI. Always review output for mistakes.
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/
```

```
df = spark.createDataFrame([
    ("Hello! How are you doing today?",),
    ("Tell me what you'd like to know, and I'll do my best to help.",),
    ("The only thing we have to fear is fear itself.",),
], ["text"])

translations = df.ai.translate(to_lang="spanish", input_col="text",
output_col="translations")
display(translations)
```

Related content

- Calculate similarity with [ai.similarity](#).
- Categorize text with [ai.classify](#).
- Detect sentiment with [ai.analyze_sentiment](#).
- Extract entities with [ai_extract](#).
- Fix grammar with [ai.fix_grammar](#).
- Summarize text with [ai.summarize](#).
- Answer custom user prompts with [ai.generate_response](#).
- To learn more about the full set of AI functions, visit [this overview article](#).
- Learn how to customize the configuration of AI functions [here](#).
- Did we miss a feature you need? Suggest it on the [Fabric Ideas forum](#).

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Answer custom user prompts with the `ai.generate_response` function

Article • 03/05/2025

The `ai.generate_response` function uses Generative AI to generate custom text responses based on your own instructions—all with a single line of code.

AI functions turbocharge data engineering by putting the power of Fabric's built-in large languages models into your hands. To learn more, visit [this overview article](#).

Important

This feature is in [preview](#), for use in the [Fabric 1.3 runtime](#) and higher.

- Review the prerequisites in [this overview article](#), including the [library installations](#) that are temporarily required to use AI functions.
- By default, AI functions are currently powered by the **gpt-3.5-turbo (0125)** model. To learn more about billing and consumption rates, visit [this article](#).
- Although the underlying model can handle several languages, most of the AI functions are optimized for use on English-language texts.
- During the initial rollout of AI functions, users are temporarily limited to 1,000 requests per minute with Fabric's built-in AI endpoint.

Use `ai.generate_response` with pandas

The `ai.generate_response` function extends the [pandas DataFrame](#) class. The `ai.generate_response` function differs from the other AI functions, because those functions extend the [pandas Series](#) class. Call this function on an entire pandas DataFrame to generate custom text responses row by row. Your prompt can be a literal string, in which case the function considers all columns of the DataFrame while generating responses. Or your prompt can be a format string, in which case the function considers only those column values that appear between curly braces in the prompt.

The function returns a pandas Series that contains custom text responses for each row of input. The text responses can be stored in a new DataFrame column.

Syntax

Generating responses with a simple prompt

Python

```
df["response"] = df.ai.generate_response(prompt="Instructions for a  
custom response based on all column values")
```

Parameters

[+] Expand table

Name	Description
prompt Required	A string that contains prompt instructions to be applied to input text values for custom responses.
is_prompt_template Optional	A boolean that indicates whether the prompt is a format string or a literal string. If this parameter is set to <code>True</code> , then the function considers only the specific row values from each column name that appears in the format string. In this case, those column names must appear between curly braces, and other columns are ignored. If this parameter is set to its default value of <code>False</code> , then the function considers all column values as context for each input row.

Returns

The function returns a [pandas DataFrame](#) that contains custom text responses to the prompt for each input text row.

Example

Generating responses with a simple prompt

Python

```
# This code uses AI. Always review output for mistakes.  
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/  
  
df = pd.DataFrame([  
    ("Scarves"),  
    ("Snow pants"),  
    ("Ski goggles")
```

```
[], columns=[ "product"])

df[ "response" ] = df.ai.generate_response("Write a short, punchy email
subject line for a winter sale.")
display(df)
```

Use `ai.generate_response` with PySpark

The `ai.generate_response` function is also available for [Spark DataFrames](#). You must specify the name of an existing input column as a parameter. You must also specify a string-based prompt, and a boolean that indicates whether that prompt should be treated as a format string.

The function returns a new DataFrame, with custom responses for each input text row stored in an output column.

Syntax

Generating responses with a simple prompt

Python

```
df.ai.generate_response(prompt="Instructions for a custom response based
on all column values", output_col="response")
```

Parameters

[] [Expand table](#)

Name	Description
<code>prompt</code> Required	A string that contains prompt instructions to be applied to input text values, for custom responses.
<code>is_prompt_template</code> Optional	A boolean that indicates whether the prompt is a format string or a literal string. If this parameter is set to <code>True</code> , then the function considers only the specific row values from each column that appears in the format string. In this case, those column names must appear between curly braces, and other columns are ignored. If this parameter is set to its default value of <code>False</code> , then the function considers all column values as context for each input row.

Name	Description
<code>output_col</code> Optional	A string that contains the name of a new column to store custom responses for each row of input text. If this parameter isn't set, a default name is generated for the output column.
<code>error_col</code> Optional	A string that contains the name of a new column to store any OpenAI errors that result from processing each row of input text. If this parameter isn't set, a default name is generated for the error column. If there are no errors for a row of input, the value in this column is <code>null</code> .

Returns

A [Spark DataFrame](#) with a new column that contains custom text responses to the prompt for each input text row.

Example

Generating responses with a simple prompt

Python

```
# This code uses AI. Always review output for mistakes.
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/

df = spark.createDataFrame([
    ("Scarves",),
    ("Snow pants",),
    ("Ski goggles",)
], ["product"])

responses = df.ai.generate_response(prompt="Write a short, punchy email subject line for a winter sale.", output_col="response")
display(responses)
```

Related content

- Calculate similarity with [ai.similarity](#).
- Categorize text with [ai.classify](#).
- Detect sentiment with [ai.analyze_sentiment](#).
- Extract entities with [ai_extract](#).
- Fix grammar with [ai.fix_grammar](#).

- Summarize text with [ai.summarize](#).
 - Translate text with [ai.translate](#).
 - Learn more about the full set of AI functions [here](#).
 - Learn how to customize the configuration of AI functions [here](#).
 - Did we miss a feature you need? Suggest it on the [Fabric Ideas forum](#).
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Customize the configuration of AI functions

Article • 03/05/2025

AI functions, currently in public preview, allow users to harness the power of Fabric's native large language models (LLMs) to [transform and enrich their enterprise data](#). They're designed to work out-of-the-box, with the underlying model and settings configured by default. Users who want more flexible configurations, however, can customize their solutions with a few extra lines of code.

Important

This feature is in [preview](#), for use in the [Fabric 1.3 runtime](#) and higher.

- Review the prerequisites in [this overview article](#), including the [library installations](#) that are temporarily required to use AI functions.
- Although the underlying model can handle several languages, most of the AI functions are optimized for use on English-language texts.
- During the initial rollout of AI functions, users are temporarily limited to 1,000 requests per minute with Fabric's built-in AI endpoint.

Customizing AI functions with pandas

By default, AI functions are powered by Fabric's built-in AI endpoint. The LLM's settings are globally configured in the `aifunc Conf` class. If you work with AI functions in pandas, you can use the `aifunc Conf` class to modify some or all of these settings:

 Expand table

Parameter	Description	Default
<code>model_deployment_name</code> Optional	A string that designates the name of the language model deployment that powers AI functions.	<code>gpt-35-turbo-0125</code>
<code>embedding_deployment_name</code> Optional	A string that designates the name of the embedding model deployment that powers AI functions.	<code>text-embedding-ada-002</code>

Parameter	Description	Default
<code>temperature</code> Optional	A <code>float</code> between 0.0 and 1.0 that designates the temperature of the underlying model. Higher temperatures increase the randomness or creativity of the model's outputs.	0.0
<code>seed</code> Optional	An <code>int</code> that designates the seed to use for the response of the underlying model. The default behavior randomly picks a seed value for each row. The choice of a constant value improves the reproducibility of your experiments.	<code>openai.NOT_GIVEN</code>
<code>timeout</code> Optional	An <code>int</code> that designates the number of seconds before an AI function raises a time-out error. By default, there's no time-out.	<code>None</code>
<code>max_concurrency</code> Optional	An <code>int</code> that designates the maximum number of rows to be processed in parallel with asynchronous requests to the model. Higher values speed up processing time (if your capacity can accommodate it).	4

The next code sample shows how to override `aifunc Conf` settings globally, so that they apply to all AI function calls in a given session:

Python

```
# This code uses AI. Always review output for mistakes.
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/

aifunc.default_conf.temperature = 0.5 # Default: 0.0
aifunc.default_conf.maxConcurrency = 10 # Default: 4

df = pd.DataFrame([
    "Hello! How are you doing today?",
    "Tell me what you'd like to know, and I'll do my best to help.",
    "The only thing we have to fear is fear itself."
], columns=["text"])

df["translations"] = df["text"].ai.translate("spanish")
df["sentiment"] = df["text"].ai.analyze_sentiment()
display(df)
```

You can also customize these settings for each individual function call. Each AI function accepts an optional `conf` parameter. The next code sample modifies the default `aifunc`

settings for only the `ai.translate` function call, using a custom temperature value. (The `ai.analyze_sentiment` call still uses the default values, because no custom values are set.)

Python

```
# This code uses AI. Always review output for mistakes.  
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/  
  
from synapse.ml.aifunc import Conf  
  
df = pd.DataFrame([  
    "Hello! How are you doing today?",  
    "Tell me what you'd like to know, and I'll do my best to help.",  
    "The only thing we have to fear is fear itself."  
, columns=["text"])  
  
df["translations"] = df["text"].ai.translate("spanish",  
conf=Conf(temperature=0.5))  
df["sentiment"] = df["text"].ai.analyze_sentiment()  
display(df)
```

To substitute a custom Azure OpenAI LLM resource in place of the native Fabric LLM, you can use the `aifunc.setup` function with your own client, as shown in the next code sample:

Python

```
from openai import AzureOpenAI  
  
# Example of creating a custom client  
client = AzureOpenAI(  
    api_key="your-api-key",  
    azure_endpoint="https://your-openai-endpoint.openai.azure.com/",  
    api_version=aifunc.session.api_version, # Default "2024-10-21"  
    max_retries=aifunc.session.max_retries, # Default: sys.maxsize ~= 9e18  
)  
  
aifunc.setup(client) # Set the client for all functions
```

Customizing AI functions with PySpark

If you're working with AI functions in PySpark, you can use the `OpenAIDefaults` class to modify the underlying language model that powers the functions. As an example, the following code sample uses placeholder values to show how you can override the built-in Fabric AI endpoint with a custom Azure OpenAI LLM deployment:

Python

```
from synapse.ml.services.openai import OpenAIDefaults
defaults = OpenAIDefaults()

defaults.set_deployment_name("your-deployment-name")
defaults.set_subscription_key("your-subscription-key")
defaults.set_URL("https://your-openai-endpoint.openai.azure.com/")
defaults.set_temperature(0.05)
```

You can substitute your own values for the deployment name, subscription key, endpoint URL, and custom temperature value:

[+] Expand table

Parameter	Description
<code>deployment_name</code>	A string value that designates the custom name of your model deployment in Azure OpenAI or Azure AI Foundry. In the Azure portal, this value appears under Resource Management > Model Deployments . In the Azure AI Foundry portal, the value appears on the Deployments page. By default, the native Fabric LLM endpoint deployment is set to gpt-35-turbo-0125 .
<code>subscription_key</code>	An API key used for authentication with your LLM resource. In the Azure portal, this value appears in the Keys and Endpoint section.
<code>URL</code>	A URL designating the endpoint of your LLM resource. In the Azure portal, this value appears in the Keys and Endpoint section. For example: "https://your-openai-endpoint.openai.azure.com/".
<code>temperature</code>	A numeric value between 0.0 and 1.0 . Higher temperatures increase the randomness or creativity of the underlying model's outputs. By default, the Fabric LLM endpoint's temperature is set to 0.0 .

You can retrieve and print each of the `OpenAIDefaults` parameters with the next code sample:

Python

```
print(defaults.get_deployment_name())
print(defaults.get_subscription_key())
print(defaults.get_URL())
print(defaults.get_temperature())
```

You can also reset the parameters as easily as you modified them. The following code sample resets the AI functions library so that it uses the default Fabric LLM endpoint:

Python

```
defaults.reset_deployment_name()  
defaults.reset_subscription_key()  
defaults.reset_URL()  
defaults.reset_temperature()
```

Related content

- Calculate similarity with [ai.similarity](#).
- Detect sentiment with [ai.analyze_sentiment](#).
- Categorize text with [ai.classify](#).
- Extract entities with [ai_extract](#).
- Fix grammar with [ai.fix_grammar](#).
- Summarize text with [ai.summarize](#).
- Translate text with [ai.translate](#).
- Answer custom user prompts with [ai.generate_response](#).
- Learn more about the full set of AI functions [here](#).
- Did we miss a feature you need? Let us know! Suggest it at the [Fabric Ideas forum](#) ↗

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) ↗ | [Ask the community](#) ↗

Fabric data agent concepts (preview)

Article • 03/31/2025

Data agent in Microsoft Fabric is a new Microsoft Fabric feature that allows you to build your own conversational Q&A systems using generative AI. A Fabric data agent makes data insights more accessible and actionable for everyone in your organization. With a Fabric data agent, your team can have conversations, with plain English-language questions, about the data that your organization stored in Fabric OneLake and then receive relevant answers. This way, even people without technical expertise in AI or a deep understanding of the data structure can receive precise and context-rich answers.

You can also add organization-specific instructions, examples, and guidance to fine-tune the Fabric data agent. This ensures that responses align with your organization's needs and goals, allowing everyone to engage with data more effectively. Fabric data agent fosters a culture of data-driven decision-making because it lowers barriers to insight accessibility, it facilitates collaboration, and it helps your organization extract more value from its data.

 **Important**

This feature is in [preview](#).

Prerequisites

- A paid F64 or higher Fabric capacity resource
- Fabric data agent tenant settings is enabled.
- Copilot tenant switch is enabled.
- Cross-geo processing for AI is enabled.
- Cross-geo storing for AI is enabled.
- At least one of these: A warehouse, a lakehouse, one or more Power BI semantic models, or a KQL database with data.
- Power BI semantic models via XMLA endpoints tenant switch is enabled for Power BI semantic model data sources.

How the Fabric data agent works

Fabric data agent uses large language models (LLMs) to help users interact with their data naturally. Fabric data agent applies Azure OpenAI Assistant APIs, and it behaves like an agent. It processes user questions, determines the most relevant data source

(Lakehouse, Warehouse, Power BI dataset, KQL databases), and it invokes the appropriate tool to generate, validate, and execute queries. Users can then ask questions in plain language and receive structured, human-readable answers—eliminating the need to write complex queries and ensuring accurate and secure data access.

Here's how it works in detail:

Question Parsing & Validation: The Fabric data agent applies Azure OpenAI Assistant APIs as the underlying agent to process user questions. This approach ensures that the question complies with security protocols, responsible AI (RAI) policies, and user permissions. The Fabric data agent strictly enforces read-only access, maintaining read-only data connections to all data sources.

Data Source Identification: The Fabric data agent uses the user's credentials to access the schema of the data source. This ensures that the system fetches data structure information that the user has permission to view. It then evaluates the user's question against all available data sources, including relational databases (Lakehouse and Warehouse), Power BI datasets (Semantic Models), and KQL databases. It might also reference user-provided data agent instructions to determine the most relevant data source.

Tool Invocation & Query Generation: Once the correct data source or sources are identified, the Fabric data agent rephrases the question for clarity and structure, and then invokes the corresponding tool to generate a structured query:

- Natural language to SQL (NL2SQL) for relational databases (Lakehouse/Warehouse).
- Natural language to DAX (NL2DAX) for Power BI datasets (Semantic Models).
- Natural language to KQL (NL2KQL) for KQL databases.

The selected tool generates a query based on the provided schema, metadata, and context that the agent underlying the Fabric data agent then passes.

Query Validation: The tool performs validation to ensure the query is correctly formed and adheres to its own security protocols, and RAI policies.

Query Execution & Response: Once validated, the Fabric data agent executes the query against the chosen data source. The results are formatted into a human-readable response, which might include structured data such as tables, summaries, or key insights.

This approach ensures that users can interact with their data using natural language, while the Fabric data agent handles the complexities of query generation, validation,

and execution—all without requiring users to write SQL, DAX, or KQL themselves.

Fabric data agent configuration

Configuring a Fabric data agent is similar to building a Power BI report—you start by designing and refining it to ensure it meets your needs, then publish and share it with colleagues so they can interact with the data. Setting up a Fabric data agent involves:

Selecting Data Sources: A Fabric data agent supports up to five data sources in any combination, including lakehouses, warehouses, KQL databases, and Power BI semantic models. For example, a configured Fabric data agent could include five Power BI semantic models. It could include a mix of two Power BI semantic models, one lakehouse, and one KQL database. You have many available options.

Choosing Relevant Tables: After you select the data sources, you need to add them one at a time, and define the specific tables from each source that the Fabric data agent will use. This step ensures that the Fabric data agent retrieves accurate results by focusing only on relevant data.

Adding Context: To improve the Fabric data agent accuracy, you can provide more context through Fabric data agent instructions and example queries. As the underlying agent for the Fabric data agent, the context helps the Azure OpenAI Assistant API make more informed decisions about how to process user questions, and determine which data source is best suited to answer them.

- **Data agent instructions:** You can add instructions to guide the agent that underlies the Fabric data agent, in determining the best data source to answer specific types of questions. You can also provide custom rules or definitions that clarify organizational terminology or specific requirements. These instructions can provide more context or preferences that influence how the agent selects and queries data sources.
 - Direct questions about **financial metrics** to a Power BI semantic model.
 - Assign queries involving **raw data exploration** to the lakehouse.
 - Route questions requiring **log analysis** to the KQL database.
- **Example queries:** You can add sample question-query pairs to illustrate how the Fabric data agent should respond to common queries. These examples serve as a guide for the agent, which helps it understand how to interpret similar questions and generate accurate responses.

 **Note**

Adding sample query/question pairs isn't currently supported for Power BI semantic model data sources.

By combining clear AI instructions and relevant example queries, you can better align the Fabric data agent with your organization's data needs, ensuring more accurate and context-aware responses.

Difference between a Fabric data agent and a copilot

While both Fabric data agents and Fabric copilots use generative AI to process and reason over data, there are key differences in their functionality and use cases:

Configuration Flexibility: Fabric data agents are highly configurable. You can provide custom instructions and examples to tailor their behavior to specific scenarios. Fabric copilots, on the other hand, are preconfigured, and they don't offer this level of customization.

Scope and Use Case: Fabric copilots are designed to assist with tasks within Microsoft Fabric, such as generation of notebook code or warehouse queries. Fabric data agents, in contrast, are standalone artifacts. To make Fabric data agents more versatile for broader use cases, they can integrate with external systems like Microsoft Copilot Studio, Azure AI Foundry, Microsoft Teams, or other tools outside Fabric.

Evaluation of the Fabric data agent

The quality and safety of Fabric data agent responses went through rigorous evaluation:

Benchmark Testing: The product team tested Fabric data agents across a range of public and private datasets to ensure high-quality and accurate responses.

Enhanced Harm Mitigations: More safeguards are in place to ensure that Fabric data agent outputs remain focused on the context of selected data sources, to reduce the risk of irrelevant or misleading answers.

Limitations

The Fabric data agent is currently in public preview and it has limitations. Updates will improve the Fabric data agent over time.

- The Fabric data agent can retrieve data by generating structured queries (SQL, DAX, or KQL) for questions that involve facts, totals, rankings, or filters. However, it can't interpret trends, provide explanations, or analyze underlying causes.
- The Fabric data agent only generates SQL/DAX/KQL "read" queries. It doesn't generate SQL/DAX/KQL queries that create, update, or delete data.
- The Fabric data agent can only access data that you provide. It only uses the data resource configurations that you provide.
- The Fabric data agent has data access permissions that match the permissions granted to the user interacting with the Fabric data agent. This is true when the Fabric data agent is published to other locations—for example, Microsoft Copilot Studio, Azure AI Foundry, and Microsoft Teams.
- You can't add more than five data sources to the Fabric data agent.
- You can't use the Fabric data agent to access unstructured data resources. These resources include .pdf, .docx, or .txt files, for example.
- The Fabric data agent blocks non-English language questions or instructions.
- You can't change the LLM that the Fabric data agent uses.
- You can't add a KQL database as a data source if it has more than 1,000 tables or any table with over 100 columns.
- You can't add a Power BI semantic model as a data source if it contains more than a total of 100 columns and measures.
- The Fabric data agent works best with 25 or fewer tables selected across all data sources.
- Nondescriptive data resource column and table names have a significant, negative impact on generated SQL/DAX/KQL query quality. We recommend the use of descriptive names.
- Use of too many columns and tables might lower Fabric data agent performance.
- The Fabric data agent is currently designed to handle simple queries. Complex queries that require many joins or sophisticated logic tend to have lower reliability.
- If you add a Power BI semantic model as a data source, the Fabric data agent doesn't use any hidden tables, columns, or measures.
- If you previously created a Fabric data agent that used a warehouse as a data source, and the warehouse was located in a workspace that doesn't host that Fabric data agent, you might encounter an error. To resolve this issue, delete the existing data source and add it again.
- To add a Power BI semantic model as a data source for Fabric data agent, you need read/write permissions for that Power BI semantic model. Querying a Fabric data agent that uses a Power BI semantic model also requires that you have read/write permissions for the underlying Power BI semantic model.
- The Fabric data agent might return incorrect answers. You should test the Fabric data agent with your colleagues to verify that it answers questions as expected. If it makes mistakes, provide it with more examples and instructions.

- If you previously created and published a Fabric data agent, and you have used its URL programmatically, the URL will no longer work if you open the Fabric data agent in the Fabric data agent new user interface page. To resolve this, you must republish the Fabric data agent, and use the new URL based on the Assistants API.

Related content

- [Fabric data agent scenario](#)
- [Create a Fabric data agent](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Create a Fabric data agent (preview)

Article • 03/31/2025

With a data agent in Microsoft Fabric, you can create conversational AI experiences that answer questions about data stored in lakehouses, warehouses, Power BI semantic models, and KQL databases in Fabric. Your data insights become accessible. Your colleagues can ask questions in plain English and receive data-driven answers, even if they aren't AI experts or deeply familiar with the data.

 **Important**

This feature is in [preview](#).

Prerequisites

- A paid F64 or higher Fabric capacity resource
- Fabric data agent tenant settings is enabled.
- Copilot tenant switch is enabled.
- Cross-geo processing for AI is enabled.
- Cross-geo storing for AI is enabled.
- At least one of these: A warehouse, a lakehouse, one or more Power BI semantic models, or a KQL database with data.
- Power BI semantic models via XMLA endpoints tenant switch is enabled for Power BI semantic model data sources.

End-to-End Flow for creating and consuming Fabric data agents

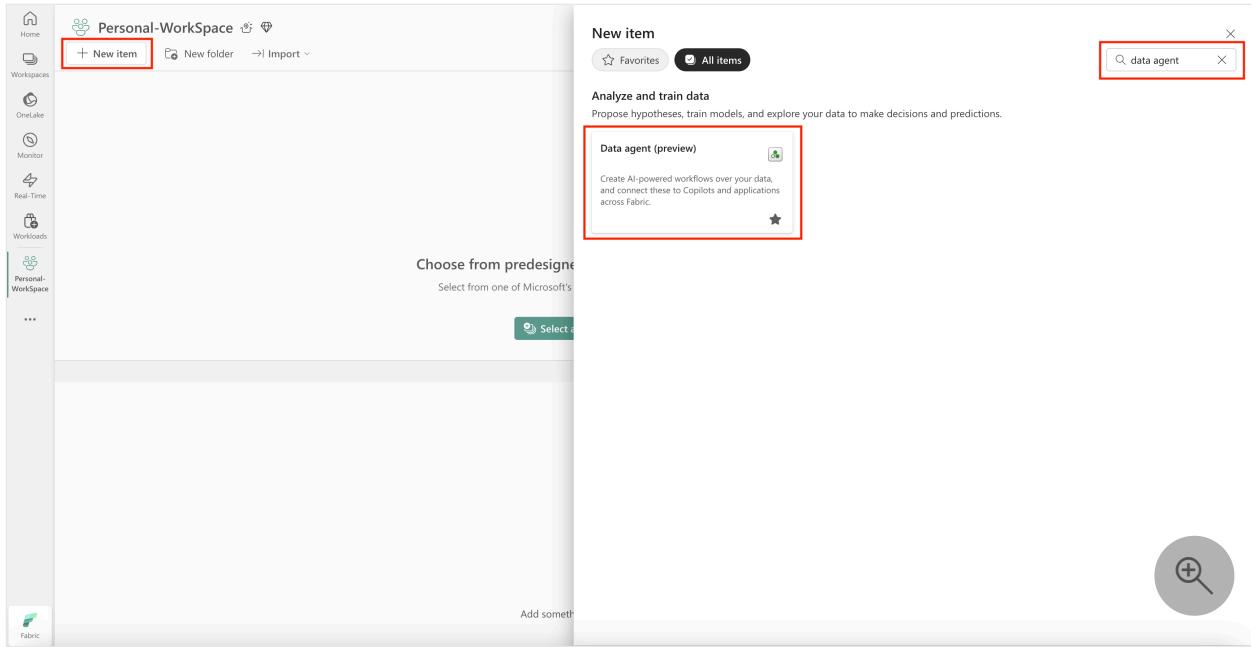
This section outlines the key steps to create, validate, and share a Fabric data agent in Fabric, making it accessible for consumption.

The process is straightforward and you can begin testing the Fabric data agent resources in minutes.

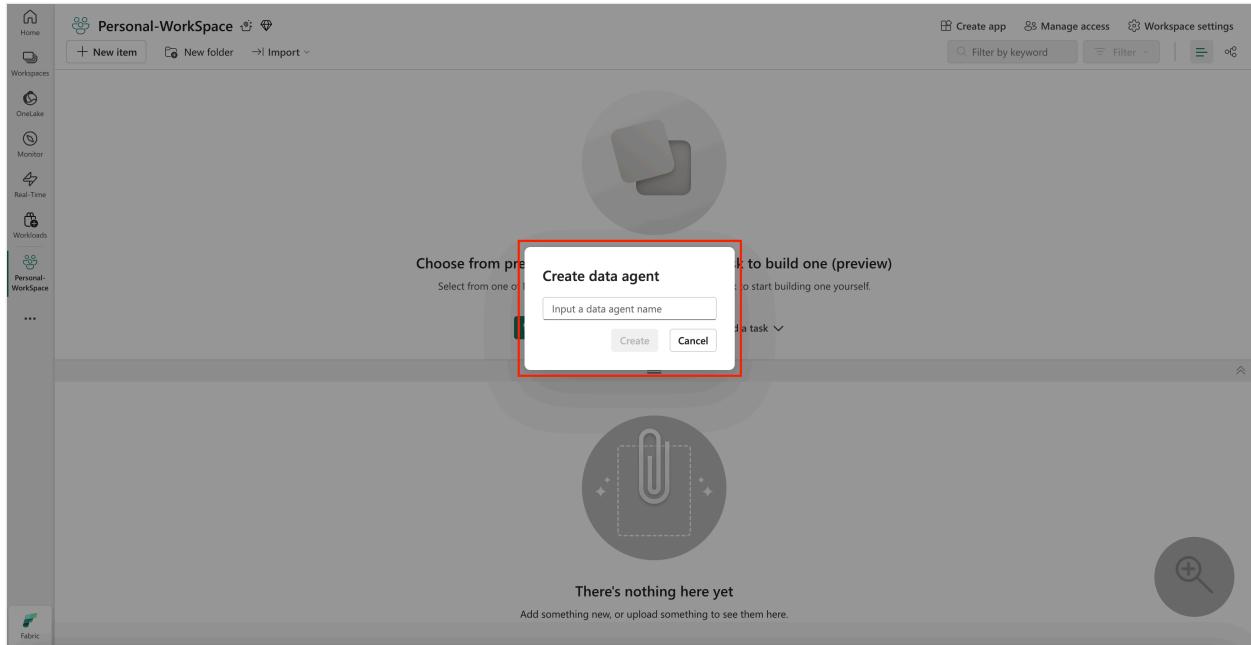
Create a new Fabric data agent

To create a new Fabric data agent, first navigate to your workspace, and then select the + New Item button. In the All items tab, search for **Fabric data agent** to locate the

appropriate option, as shown in this screenshot:



Once selected, you're prompted to provide a name for your Fabric data agent, as shown in this screenshot:



Refer to the provided screenshot for a visual guide on naming the Fabric data agent. After entering the name, proceed with the configuration to align the Fabric data agent with your specific requirements.

Select your data

After you create a Fabric data agent, you can add up to five data sources, including lakehouses, warehouses, Power BI semantic models, and KQL databases in any

combination. For example, you could add five Power BI semantic models, or two Power BI semantic models, one lakehouse, and one KQL database.

When you create a Fabric data agent for the first time, and provide a name, the OneLake catalog automatically appears, allowing you to add data sources. To add a data source, select it from the catalog as shown on the next screen, then select **Add**. Each data source must be added individually. For example, you can add a lakehouse, select **Add**, and then proceed to add another data source. To filter the data source types, select the filter icon and then select the desired type. You can view only the data sources of the selected type, making it easier to locate and connect the appropriate sources for your Fabric data agent.

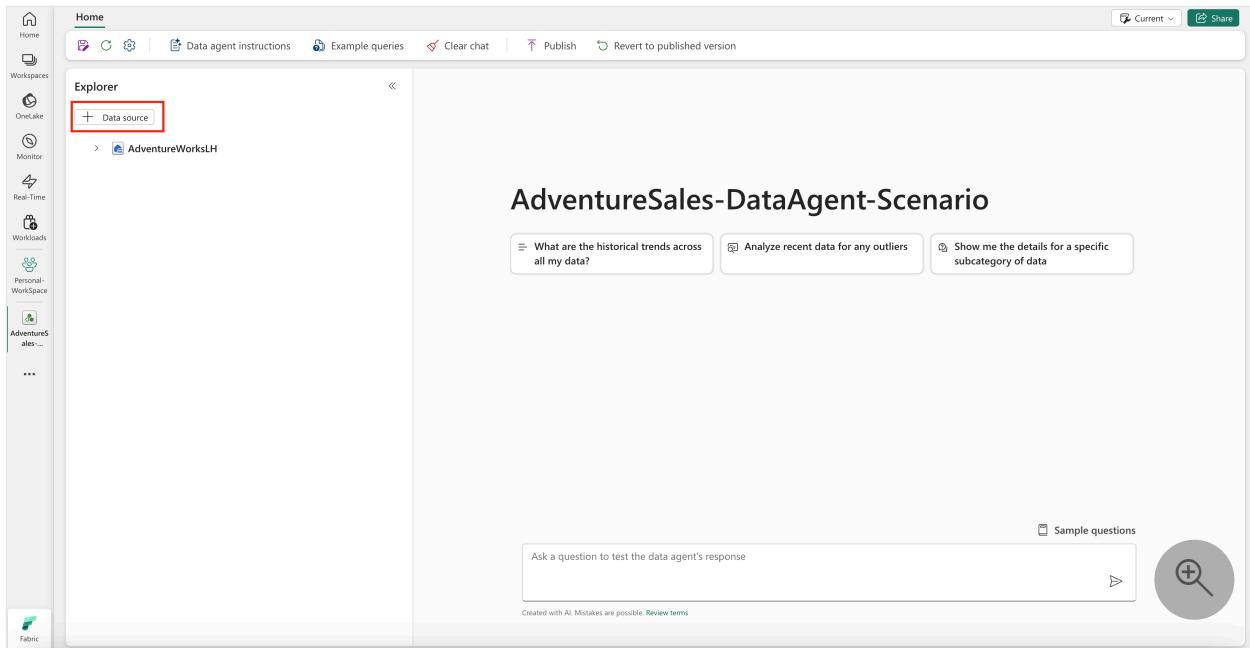
Once you add the data source, the **Explorer** on the left pane of the Fabric data agent page populates with the available tables in each selected data source, where you can use the checkboxes to make tables available or unavailable to the AI as shown in the following screenshot:

The screenshot shows the 'Add a data source' dialog. On the left, there's an 'Explorer' sidebar. The main area lists data sources with columns for Name, Type, Owner, and Refreshed. A 'Filter' dropdown is open, showing a hierarchy of data source types: KQL Database, Lakehouse, Semantic model, and Warehouse. The 'Semantic model' node is expanded, showing several entries. At the bottom right are 'Add' and 'Cancel' buttons.

ⓘ Note

You need read/write permission to add a Power BI semantic model as a data source to the Fabric data agent.

For subsequent additions of data sources, navigate to the **Explorer** on the left pane of the Fabric data agent page, and select **+ Data source**, as shown in this screenshot:



The OneLake catalog opens again, and you can seamlessly add more data sources as needed.

💡 Tip

Make sure to use descriptive names for both tables and columns. A table named `SalesData` is more meaningful than `TableA`, and column names like `ActiveCustomer` or `IsCustomerActive` are clearer than `C1` or `ActCu`. Descriptive names help the AI generate more accurate and reliable queries.

Ask questions

After you add the data sources and select the relevant tables for each data source, you can start asking questions. The system handles questions as shown in this screenshot:

The screenshot shows the Microsoft Fabric Data Agent Scenario interface. On the left, there's a sidebar with icons for Home, Workspaces, OneLake, Monitor, Real-Time Workloads, Personal Workspace, and a Fabric icon. The main area has a title bar with 'Home', 'Data agent instructions', 'Example queries', 'Clear chat', 'Publish', and 'Revert to published version'. Below the title bar is an 'Explorer' pane showing a tree view of a 'AdventureWorksLH' database, specifically the 'dbo' schema, with various tables like dimaccount, dimcurrency, dimcustomer, dimdate, etc., some of which have checkboxes next to them. To the right of the explorer is a large workspace titled 'AdventureSales-DataAgent-Scenario'. It contains a text input field with the question 'Who are the top 5 customers by yearly income, and what are their state or province names?'. Below this input field is a note: 'Created with AI. Mistakes are possible. Review terms.' There are also three buttons at the top of the workspace: 'What are the historical trends across all my data?', 'Analyze recent data for any outliers', and 'Show me the details for a specific subcategory of data'. In the bottom right corner of the workspace, there's a magnifying glass icon labeled 'Sample questions'.

Questions similar to these examples should also work:

- "What were our total sales in California in 2023?"
- "What are the top 5 products with the highest list prices, and what are their categories?"
- "What are the most expensive items that have never been sold?"

Questions like this are suitable because the system can translate them into structured queries (T-SQL, DAX, or KQL), execute them against databases, and then return concrete answers based on stored data.

However, questions like these are out of scope:

- "Why is our factory productivity lower in Q2 2024?"
- "What is the root cause of our sales spike?"

These questions are currently out of scope because they require complex reasoning, correlation analysis, or external factors not directly available in the database. The Fabric data agent currently doesn't perform advanced analytics, machine learning, or causal inference. It simply retrieves and processes structured data based on the user's query.

When you ask a question, the Fabric data agent uses the Azure OpenAI Assistant API to process the request. The flow operates this way:

Schema access with user credentials

The system first uses the credentials of the user to access the schema of the data source (for example, lakehouse, warehouse, PBI semantic model, or KQL databases). This

ensures that the system fetches data structure information that the user has permission to view.

Constructing the prompt

To interpret the user's question, the system combines:

1. User Query: The natural language question provided by the user.
2. Schema Information: Metadata and structural details of the data source retrieved in the previous step.
3. Examples and Instructions: Any predefined examples (for example, sample questions and answers) or specific instructions provided when setting up the Fabric data agent. These examples and instructions help refine the AI's understanding of the question, and guide how the AI interacts with the data.

All this information is used to construct a prompt. This prompt serves as an input to the Azure OpenAI Assistant API, which behaves as an agent underlying the Fabric data agent. This essentially instructs the Fabric data agent about how to process the query, and the type of answer to produce.

Tool invocation based on query needs

The agent analyzes the constructed prompt, and decides which tool to invoke to retrieve the answer:

- Natural Language to SQL (NL2SQL): Used to generate SQL queries when the data resides in a lakehouse or warehouse
- Natural Language to DAX (NL2DAX): Used to create DAX queries to interact with semantic models in Power BI data sources
- Natural Language to KQL (NL2KQL): Used to construct KQL queries to query data in KQL databases

The selected tool generates a query using the schema, metadata, and context that the agent underlying the Fabric data agent provides. Then the tool validates the query, to ensure proper formatting and compliance with its security protocols, and its own Responsible AI (RAI) policies.

Response construction

The agent underlying the Fabric data agent executes the query and ensures that the response is structured and formatted appropriately. The agent often includes extra

context to make the answer user-friendly. Finally, the answer is displayed to the user in a conversational interface, as shown in the following screenshot:

The screenshot shows the Fabric data agent interface. On the left is a sidebar with icons for Home, Workspaces, OneLake, Monitor, Real-Time, Workloads, Personal Workspace, and a section for AdventureS.les. The main area has tabs for Home, Data agent instructions, Example queries, Clear chat, Publish, and Revert to published version. The Home tab is selected. In the center, there's an 'Explorer' pane showing a tree view of a database named 'AdventureWorksLT'. Under 'dbo', several tables are listed with checkboxes next to them, some of which are checked. A red box highlights a dropdown menu that says 'The top 5 customers by yearly income and their state or province names are:' followed by a list of five customers with their details. Below this, it says '1 step completed' and 'Response time: 8 sec'. At the bottom, there's a search bar with 'Ask a question to test the data agent's response' and a 'Sample questions' button.

The agent presents both the result and the intermediate steps that it took to retrieve the final answer. This approach enhances transparency and allows validation of those steps, if necessary. Users can expand the dropdown for the steps to view all the steps the Fabric data agent took to retrieve the answer, as shown in the following screenshot:

This screenshot is similar to the previous one but shows the expanded steps for the query. The 'Step 1. analyze_database' section is highlighted with a red box. It shows the original question 'Who are the top 5 customers by yearly income, and what are their state or province names?' followed by the generated SQL code: 'SELECT TOP 5 c.FirstName + ' ' + c.LastName AS CustomerName, g.StateProvinceName FROM dbo.dimcustomer c JOIN dbo.dimgeography g ON c.GeographyKey = g.GeographyKey ORDER BY c.YearlyIncome DESC;'. The rest of the interface is identical to the first screenshot.

Additionally, the Fabric data agent provides the generated code used to query the corresponding data source, offering further insight into how the response was constructed.

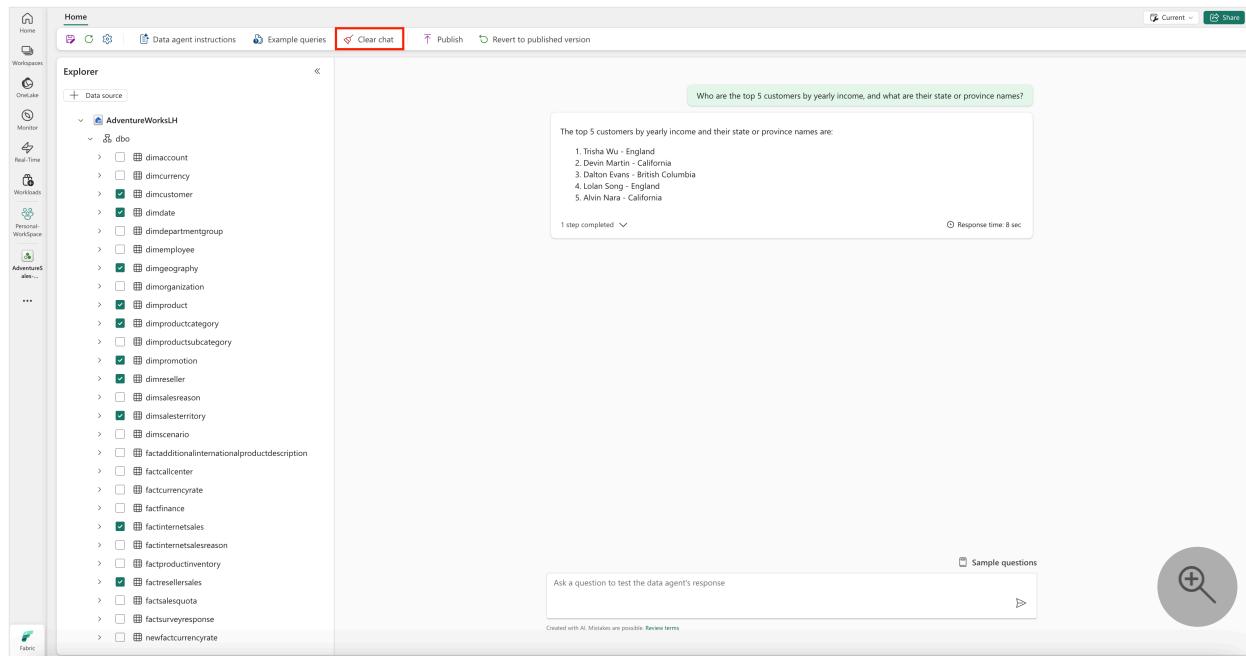
These queries are designed exclusively for querying data. Operations that involve

- data creation

- data updates
- data deletions
- any type of data change

aren't allowed, to protect the integrity of your data.

At any point, you can select the **Clear chat** button to clear the chat, as shown in the following screenshot:



The Clear chat feature erases all chat history and starts a new session. Once you delete your chat history, you can't retrieve it.

Change the data source

To remove a data source, hover over the data source name in the **Explorer** on the left pane of the Fabric data agent page until the three-dot menu appears. Select the three dots to reveal the options, then select **Remove** to delete the data source as shown in the following screenshot:

The screenshot shows the Fabric interface with the 'Home' tab selected. On the left, there's a sidebar with icons for Home, Workspaces, Create, Monitor, Real-Time, Workloads, Personal Workspace, and a 'Fabric' icon. The main area has tabs for Data agent instructions, Example queries, Clear chat, Publish, and Revert to published version. In the center, there's an 'Explorer' pane showing a tree view of a database named 'AdventureWorksLT'. A context menu is open over the 'dbo' node, with options: Open (highlighted with a red box), Refresh (highlighted with a red box), and Remove. To the right of the explorer, a query card displays the results of a query: 'Who are the top 5 customers by yearly income, and what are their state or province names?'. The results list five customers: 1. Trisha Wu - England, 2. Devin Martin - California, 3. Dalton Evans - British Columbia, 4. Lolan Song - England, 5. Alvin Nara - California. Below the results, it says '1 step completed' and 'Response time: 8 sec'. At the bottom, there's a search bar with 'Ask a question to test the data agent's response' and a 'Sample questions' button.

Alternatively, if your data source changed, you can select **Refresh** within the same menu, as shown in the following screenshot:

This screenshot is identical to the one above, but the 'Refresh' option in the context menu over the 'dbo' node is highlighted with a red box.

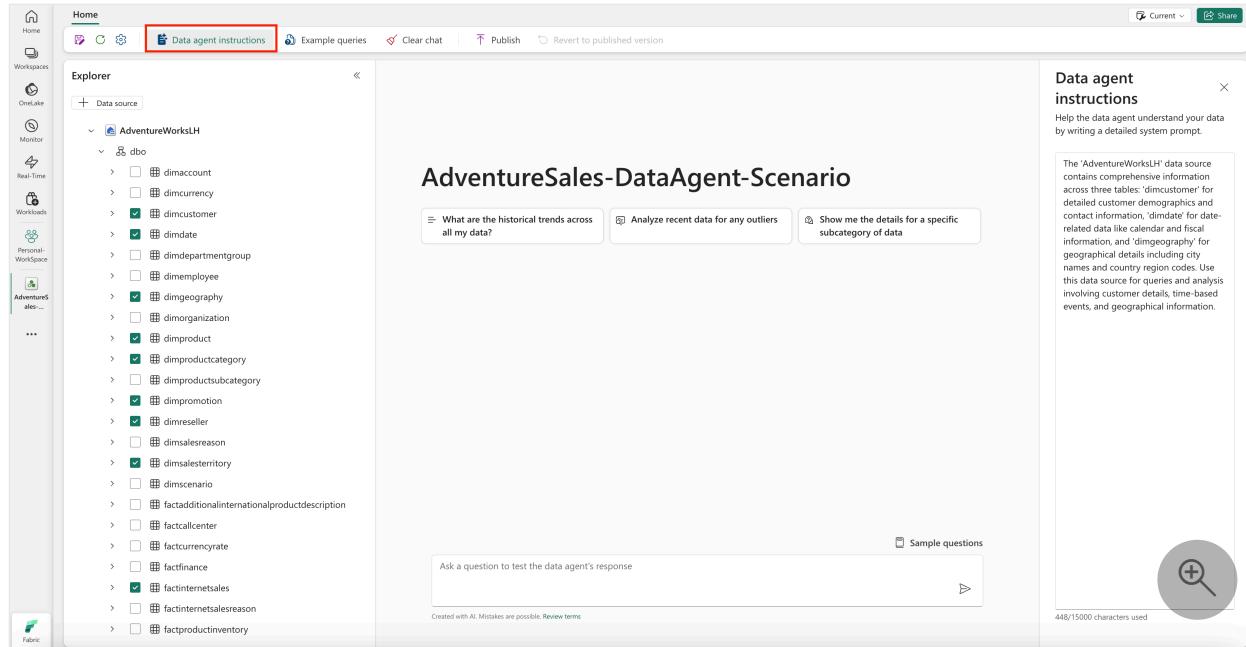
This ensures that any data source updates are both reflected and correctly populated in the explorer, to keep your Fabric data agent in sync with the latest data.

Fabric data agent configuration

The Fabric data agent offers several configuration options that allow users to customize Fabric data agent behavior, to better match the needs of your organization. As the Fabric data agent processes and presents data, these configurations offer flexibility that enables more control over the outcomes.

Provide instructions

You can provide specific instructions to guide the AI's behavior. To add them in the Fabric data agent instructions pane, select **Data agent instructions** as shown in the following screenshot:



Here, you can write up to 15,000 characters in plain English-language text, to instruct the AI about how to handle queries.

For example, you can specify the exact data source to use for certain types of questions. Examples of data source choices could involve directing the AI to use

- Power BI semantic models for financial queries
- a lakehouse for sales data
- a KQL database for operational metrics

These instructions ensure that the AI generates appropriate queries, whether SQL, DAX, or KQL, based on your guidance and the context of the questions.

If your AI resource consistently misinterprets certain words, acronyms, or terms, you can try to provide clear definitions in this section, to ensure that the AI understands and processes them correctly. This becomes especially useful for domain-specific terminology or unique business jargon.

By tailoring these instructions and defining terms, you enhance the AI's ability to deliver precise and relevant insights, in full alignment with your data strategy and business requirements.

Provide example queries

You can enhance the accuracy of the Fabric data agent responses when you provide example queries tailored to each data source, such as lakehouse, warehouse, and KQL databases. This approach, known as **Few-Shot Learning** in generative AI, helps guide the Fabric data agent to generate responses that better align with your expectations.

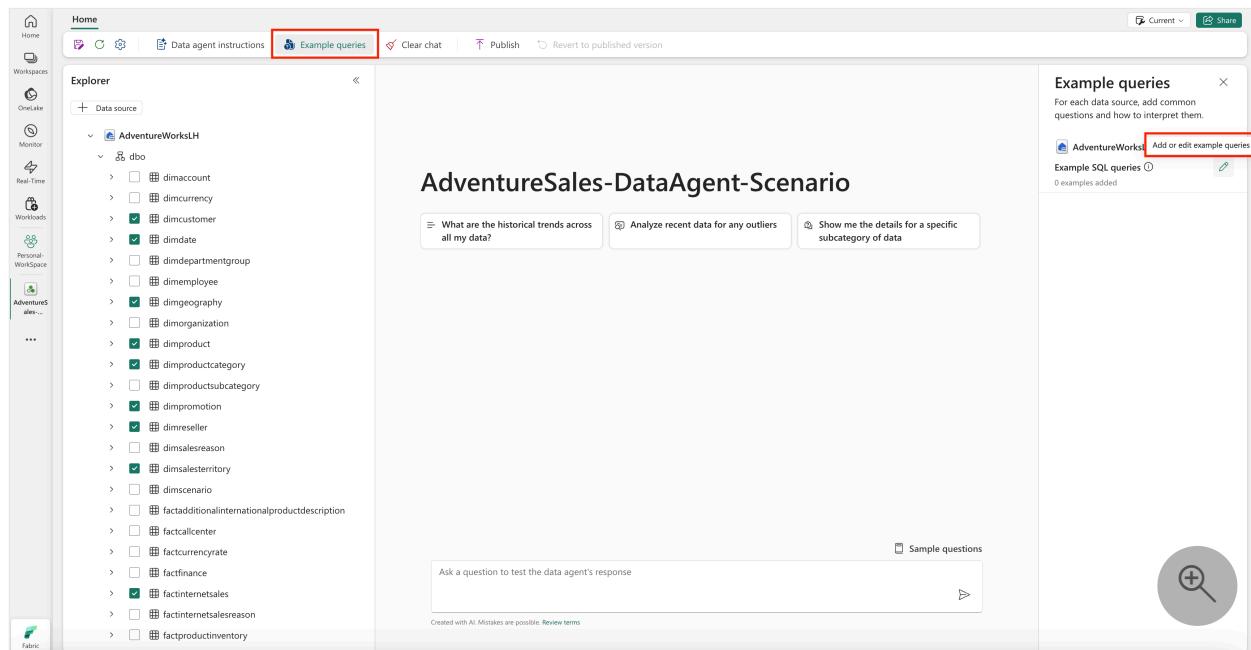
When you provide the AI with sample query/question pairs, it references these examples when it answers future questions. Matching new queries to the most relevant examples helps the AI incorporate business-specific logic, and respond effectively to commonly asked questions. This functionality enables fine-tuning for individual data sources, and ensures generation of more accurate SQL or KQL queries.

Power BI semantic model data don't support adding sample query/question pairs at this time. However, for supported data sources such as lakehouse, warehouse, and KQL databases, providing more examples can significantly improve the AI's ability to generate precise queries when its default performance needs adjustment.

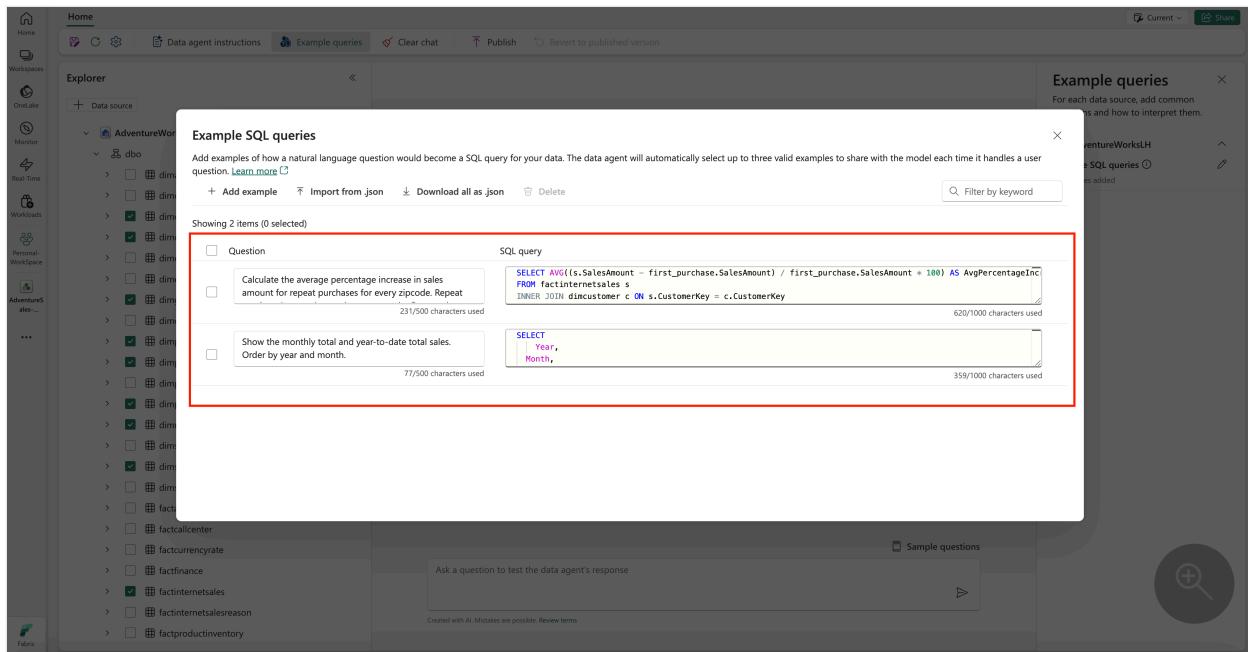
Tip

A diverse set of example queries enhances the ability of a Fabric data agent to generate accurate and relevant SQL/KQL queries.

To add or edit example queries, select the **Example queries** button to open the example queries pane, as shown in the following screenshot:



This pane provides options to add or edit example queries for all supported data sources except Power BI semantic models. For each data source, you can select **Add or Edit Example Queries** to input the relevant examples, as shown in the following screenshot:

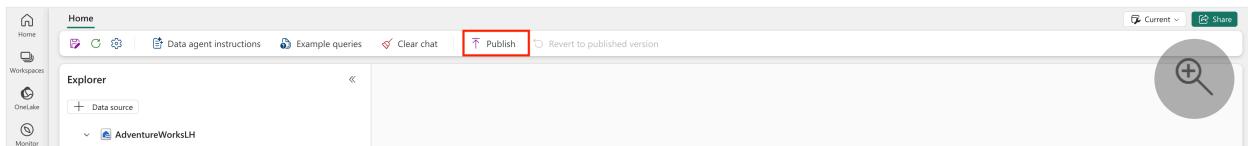


ⓘ Note

The Fabric data agent only refers to queries that contain valid SQL/KQL syntax and that match the schema of the selected tables. The Fabric data agent doesn't use queries that haven't completed their validation. Make sure that all example queries are valid and correctly aligned with the schema to ensure that the Fabric data agent utilizes them effectively.

Publish and share a Fabric data agent

After you test the performance of your Fabric data agent across various questions, and you confirm that it generates accurate SQL, DAX, OR KQL queries, you can share it with your colleagues. At that point, select **Publish**, as shown in the following screenshot:



This step opens a window that asks for a description of the Fabric data agent. Here, provide a detailed description of what the Fabric data agent does. These details guide your colleagues about the functionality of the Fabric data agent, and assist other AI systems/orchestrators to effectively invoke the Fabric data agent.

After you publish the Fabric data agent, you'll have two versions of it. One version is the current draft version, which you can continue to refine and improve. The second version is the published version, which you can share with your colleagues who want to query the Fabric data agent to get answers to their questions. You can incorporate feedback

from your colleagues into your current draft version as you develop it, to further enhance the Fabric data agent's performance.

Related content

- [Data agent concept](#)
 - [Data agent scenario](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Fabric data agent example with the AdventureWorks dataset (preview)

Article • 03/31/2025

This article describes how to set up a data agent in Microsoft Fabric, using a lakehouse as a data source. To illustrate the process, we first create a lakehouse, and then add data to it. Then, we create a Fabric data agent and configure the lakehouse as its data source. If you already have a Power BI semantic model (with the necessary read/write permissions), a warehouse, or a KQL database, you can follow the same steps after you create the Fabric data agent to add your data sources. While the steps shown here focus on the lakehouse, the process is similar for other data sources—you just need to make adjustments based on your specific selection.

ⓘ **Important**

This feature is in [preview](#).

Prerequisites

- [A paid F64 or higher Fabric capacity resource](#)
- [Fabric data agent tenant settings](#) is enabled.
- [Copilot tenant switch](#) is enabled.
- [Cross-geo processing for AI](#) is enabled.
- [Cross-geo storing for AI](#) is enabled.
- At least one of these: A warehouse, a lakehouse, one or more Power BI semantic models, or a KQL database with data.
- [Power BI semantic models via XMLA endpoints tenant switch](#) is enabled for Power BI semantic model data sources.

Create a lakehouse with AdventureWorksLH

First, create a lakehouse and populate it with the necessary data.

If you already have an instance of AdventureWorksLH in a lakehouse (or a warehouse), you can skip this step. If not, you can use the following instructions from a Fabric notebook to populate the lakehouse with the data.

1. Create a new notebook in the workspace where you want to create your Fabric data agent.
2. On the left side of the **Explorer** pane, select **+ Data sources**. This option allows you to add an existing lakehouse or creates a new lakehouse. For sake of clarity, create a new lakehouse and assign a name to it.
3. In the top cell, add the following code snippet:

```
Python

import pandas as pd
from tqdm.auto import tqdm
base =
"https://synapseaisolutionsa.blob.core.windows.net/public/AdventureWork
s"

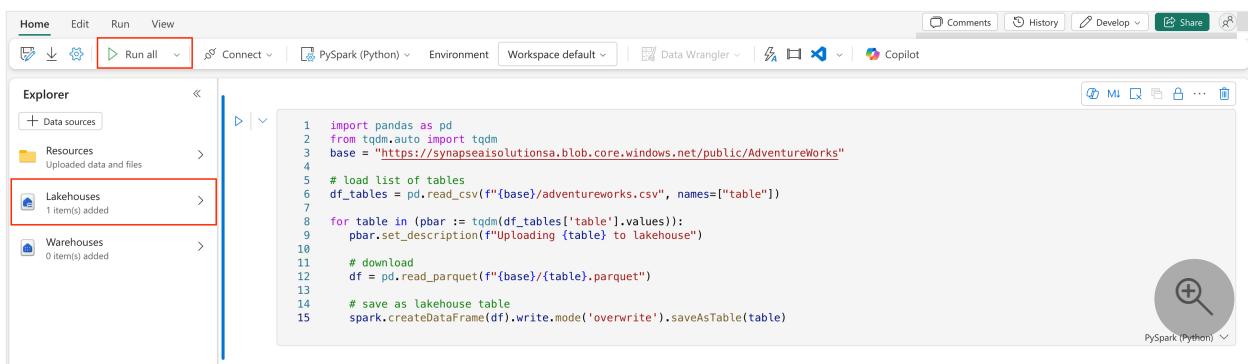
# load list of tables
df_tables = pd.read_csv(f"{base}/adventureworks.csv", names=["table"])

for table in (pbar := tqdm(df_tables['table'].values)):
    pbar.set_description(f"Uploading {table} to lakehouse")

    # download
    df = pd.read_parquet(f"{base}/{table}.parquet")

    # save as lakehouse table
    spark.createDataFrame(df).write.mode('overwrite').saveAsTable(table)
```

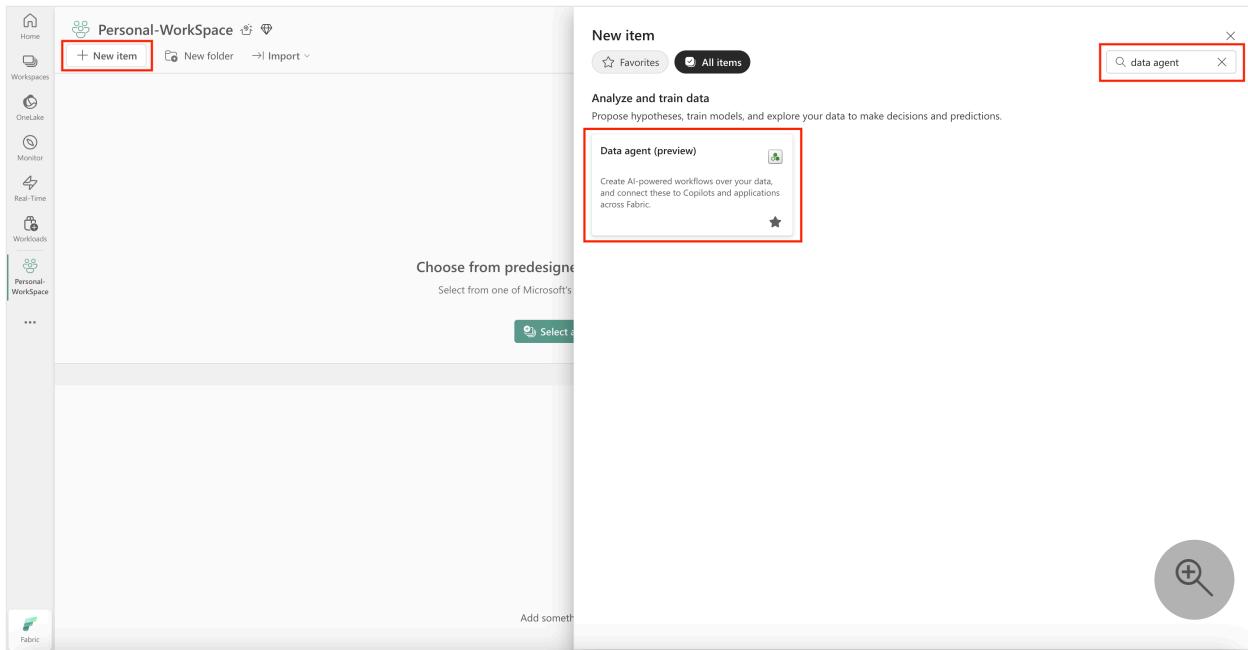
4. Select **Run all**.



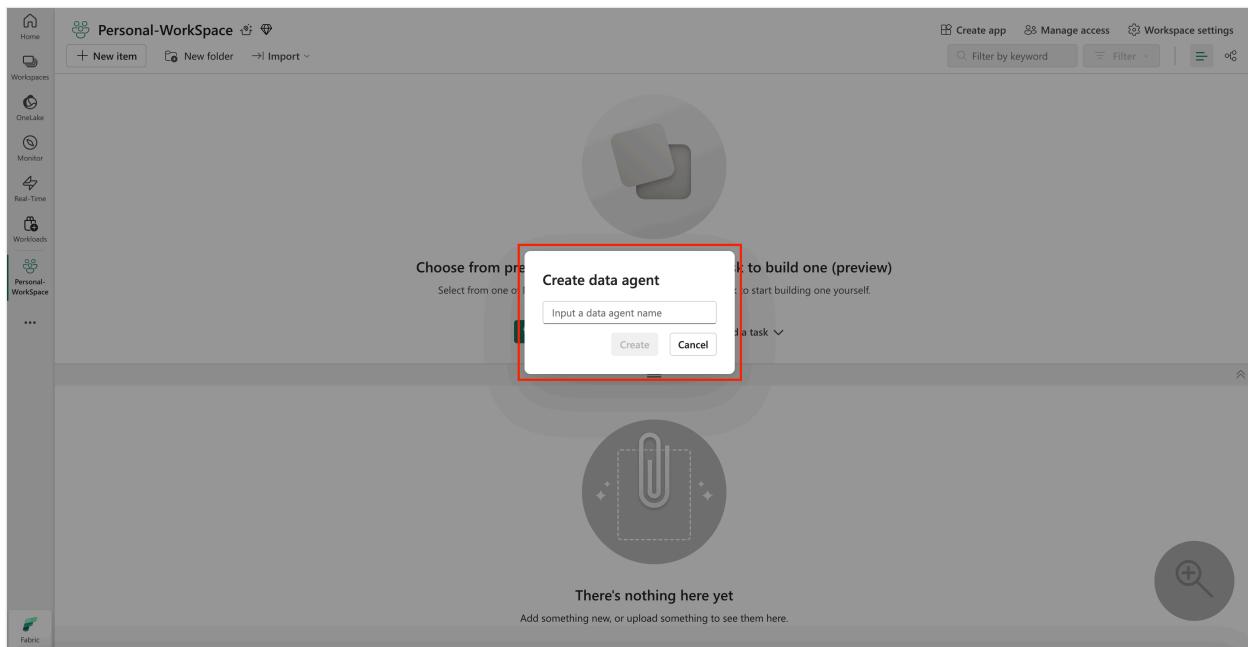
After a few minutes, the lakehouse populates with the necessary data.

Create a Fabric data agent

To create a new Fabric data agent, navigate to your workspace and select the **+ New Item** button, as shown in this screenshot:



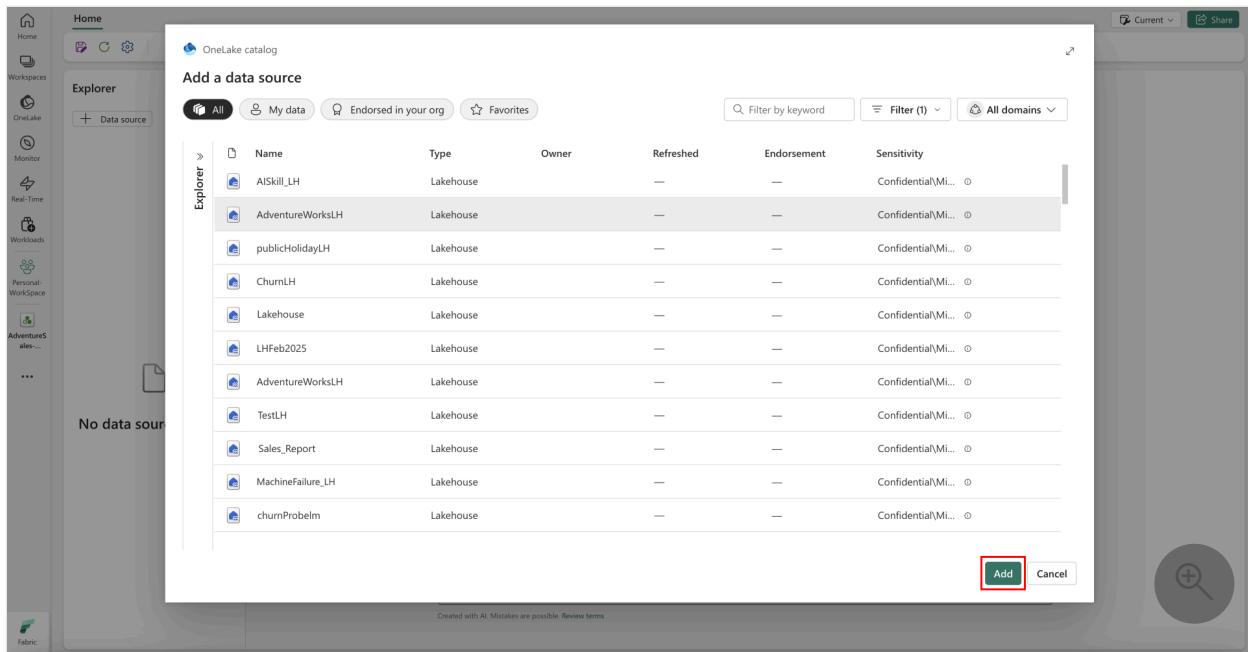
In the All items tab, search for **Fabric data agent** to locate the appropriate option. Once selected, a prompt asks you to provide a name for your Fabric data agent, as shown in this screenshot:



After you enter the name, proceed with the following steps to align the Fabric data agent with your specific requirements.

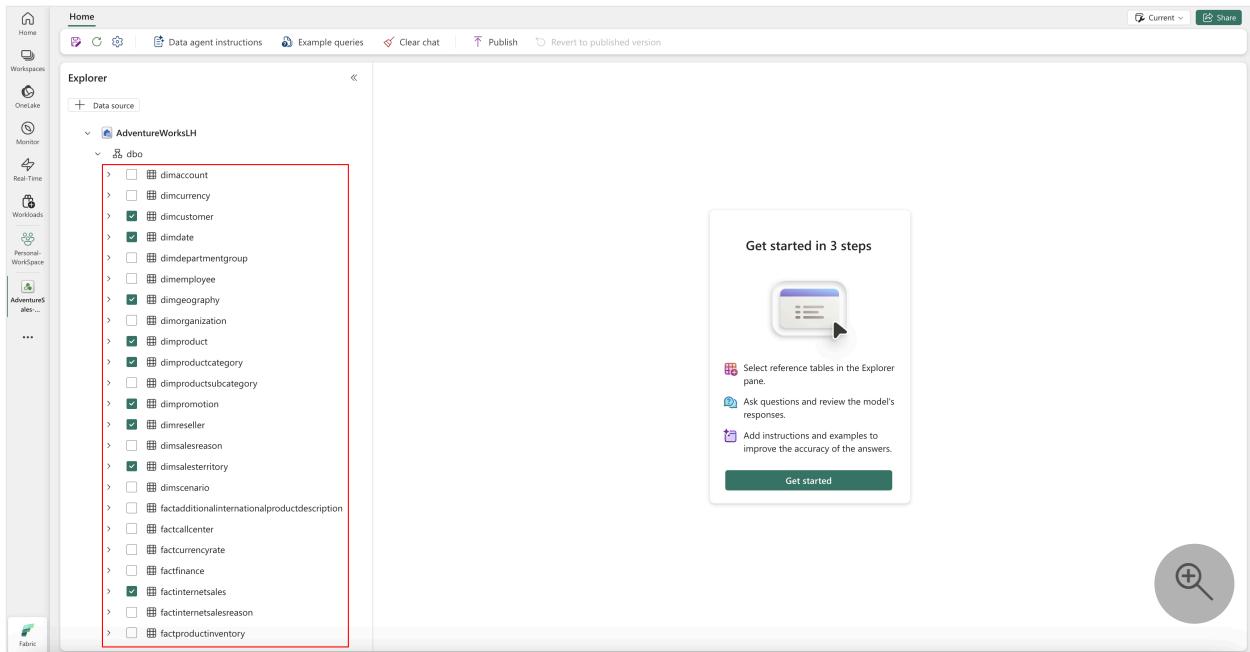
Select the data

Select the lakehouse you created in the previous step, and then select **Add**, as shown in the following screenshot:



Once the lakehouse is added as a data source, the **Explorer** pane on the left side of the Fabric data agent page shows the lakehouse name. Select the lakehouse to view all available tables. Use the checkboxes to select the tables you want to make available to the AI. For this scenario, select these tables:

- `dimcustomer`
- `dimdate`
- `dimgeography`
- `dimproduct`
- `dimproductcategory`
- `dimpromotion`
- `dimreseller`
- `dimsalesterritory`
- `factinternetsales`
- `cactresellersales`



Provide instructions

To add Fabric data agent instructions, select the **Data agent instructions** button to open the Fabric data agent instructions pane on the right. You can add the following instructions.

The `AdventureWorksLH` data source contains information from three tables:

- `dimcustomer`, for detailed customer demographics and contact information
- `dimdate`, for date-related data - for example, calendar and fiscal information
- `dimgeography`, for geographical details including city names and country region codes.

Use this data source for queries and analyses that involve customer details, time-based events, and geographical locations.

The screenshot shows the Fabric interface with the 'Data agent instructions' tab selected. The left sidebar includes icons for Home, Workspaces, OneLake, Monitor, Real-Time, Workloads, Personal Workspace, and a selected 'AdventureS' item. The main area displays the 'AdventureSales-DataAgent-Scenario' with three example queries: 'What are the historical trends across all my data?', 'Analyze recent data for any outliers', and 'Show me the details for a specific subcategory of data'. The right pane contains detailed information about the 'AdventureWorksLH' data source, including tables like dimcustomer, dimdate, and dimproduct, along with sample questions and a search icon.

Provide examples

To add example queries, select the **Example queries** button to open the example queries pane on the right. This pane provides options to add or edit example queries for all supported data sources. For each data source, you can select **Add or Edit Example Queries** to input the relevant examples, as shown in the following screenshot:

This screenshot is similar to the previous one but with the 'Example queries' tab highlighted in red. The right pane now shows the 'Example queries' section for the 'AdventureWorks' data source, which currently has 0 examples added. A red box highlights the 'Add or edit example queries' button next to the data source name.

Here, you should add Example queries for the lakehouse data source that you created.

Question: Calculate the average percentage increase in sales amount for repeat purchases for every zipcode. Repeat purchase is a purchase subsequent to the first purchase (the average should always be computed relative to the first purchase)

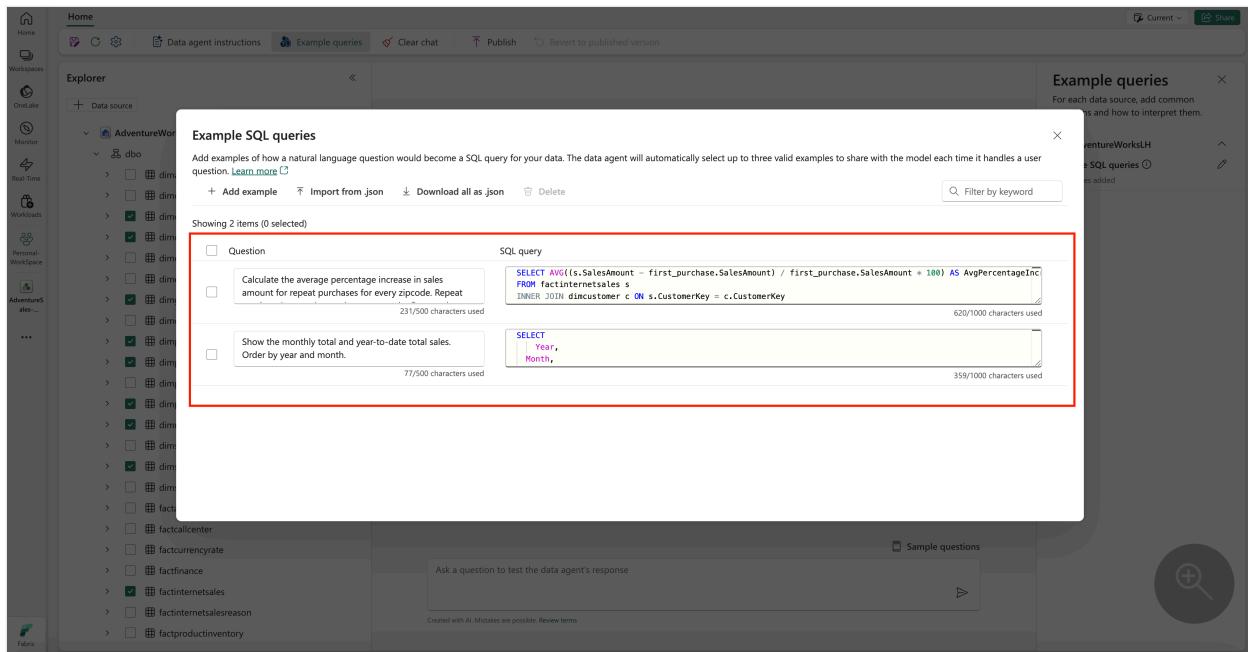
SQL

```
SELECT AVG((s.SalesAmount - first_purchase.SalesAmount) /  
first_purchase.SalesAmount * 100) AS AvgPercentageIncrease  
FROM factinternetsales s  
INNER JOIN dimcustomer c ON s.CustomerKey = c.CustomerKey  
INNER JOIN dimgeography g ON c.GeographyKey = g.GeographyKey  
INNER JOIN (  
    SELECT *  
    FROM (  
        SELECT  
            CustomerKey,  
            SalesAmount,  
            OrderDate,  
            ROW_NUMBER() OVER (PARTITION BY CustomerKey ORDER BY OrderDate)  
        AS RowNumber  
        FROM factinternetsales  
    ) AS t  
    WHERE RowNumber = 1  
) first_purchase ON s.CustomerKey = first_purchase.CustomerKey  
WHERE s.OrderDate > first_purchase.OrderDate  
GROUP BY g.PostalCode;
```

Question: Show the monthly total and year-to-date total sales. Order by year and month.

SQL

```
SELECT  
    Year,  
    Month,  
    MonthlySales,  
    SUM(MonthlySales) OVER (PARTITION BY Year ORDER BY Year, Month ROWS  
    BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS CumulativeTotal  
FROM (  
    SELECT  
        YEAR(OrderDate) AS Year,  
        MONTH(OrderDate) AS Month,  
        SUM(SalesAmount) AS MonthlySales  
    FROM factinternetsales  
    GROUP BY YEAR(OrderDate), MONTH(OrderDate)  
) AS t
```



ⓘ Note

Adding sample query/question pairs isn't currently supported for Power BI semantic model data sources.

Test and revise the Fabric data agent

Now that you configured the Fabric data agent, added Fabric data agent instructions, and provided example queries for the lakehouse, you can interact with it by asking questions and receiving answers. As you continue testing, you can add more examples, and refine the instructions, to further improve the performance of the Fabric data agent. Collaborate with your colleagues to gather feedback, and based on their input, ensure the provided example queries and instructions align with the types of questions they want to ask.

Use the Fabric data agent programmatically

You can use the Fabric data agent programmatically within a Fabric notebook. To determine whether or not the Fabric data agent has a published URL value, select **Settings**, as shown in this screenshot:

The screenshot shows the Power BI Home page. In the top navigation bar, the 'Home' tab is selected. Below it, the 'Data agent instructions' and 'Publish' buttons are visible. The 'Explorer' pane on the left shows a tree structure of a data source named 'AdventureWorksLH'. Under 'AdventureWorksLH', there is a 'dbo' folder containing various tables like 'dimaccount', 'dimcustomer', etc. The 'Publish' button is located in the top right corner of the main workspace area.

Before you publish the Fabric data agent, it doesn't have a published URL value, as shown in this screenshot:

This screenshot is similar to the previous one, but it shows the 'Data Agent' dialog box open over the main workspace. The dialog box has a title 'AdventureSales-DataAgent-Scenario' and a sub-section 'Data agent'. It contains fields for 'Published URL', 'About', 'Sensitivity label', and 'Endorsement'. The 'Endorsement' section is highlighted with a red box. A message in the 'Published URL' section says: 'This data agent hasn't been published. To access it remotely, select Publish from the Home tab of the ribbon.'

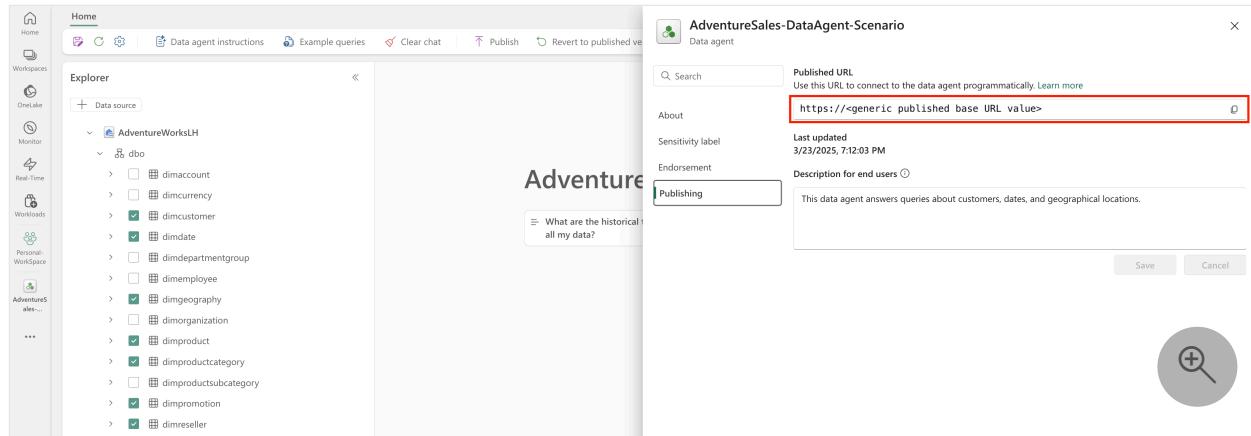
After you validate the performance of the Fabric data agent, you might decide to publish it so you can then share it with your colleagues who want to do Q&A over data. In this case, select **Publish**, as shown in this screenshot:

The screenshot shows the Power BI Home page again, but now the 'Publish' button in the top navigation bar is highlighted with a red box. This indicates that the user has selected the 'Publish' option.

The **Publish data agent** box opens, as shown in this screenshot:

This screenshot shows the 'Publish data agent' dialog box open. It has a title 'AdventureSales-DataAgent-Scenario' and a sub-section 'Publish data agent'. It includes fields for 'Name' (set to 'AdventureSales-DataAgent-Scenario') and 'Description for end users' (containing the text 'This data agent answers queries about customers, dates, and geographical locations.'). The 'Publish' button at the bottom of the dialog box is highlighted with a red box. The background shows the Power BI Home page with the 'Explorer' pane and the 'AdventureWorksLH' data source selected.

In this box, select **Publish** to publish the Fabric data agent. The published URL for the Fabric data agent appears, as shown in this screenshot:



You can then copy the published URL and use it in the Fabric notebook. This way, you can query the Fabric data agent by making calls to the Fabric data agent API in a Fabric notebook. Paste the copied URL in this code snippet. Then, replace the question with any query relevant to your Fabric data agent. This example uses `\<generic published URL value\>` as the URL.

Python

```
%pip install "openai==1.14.1"
```

Python

```
%pip install httpx==0.27.2
```

Python

```
import requests
import json
import pprint
import typing as t
import time
import uuid

from openai import OpenAI
from openai._exceptions import APIStatusError
from openai._models import FinalRequestOptions
from openai._types import Omit
from openai._utils import is_given
from synapse.ml.mlflow import get_mlflow_env_config
from sempy.fabric._token_provider import SynapseTokenProvider

base_url = "https://<generic published base URL value>"
question = "What datasources do you have access to?"
```

```

configs = get_mlflow_env_config()

# Create OpenAI Client
class FabricOpenAI(OpenAI):
    def __init__(
        self,
        api_version: str ="2024-05-01-preview",
        **kwargs: t.Any,
    ) -> None:
        self.api_version = api_version
        default_query = kwargs.pop("default_query", {})
        default_query["api-version"] = self.api_version
        super().__init__(
            api_key="",
            base_url=base_url,
            default_query=default_query,
            **kwargs,
        )

    def _prepare_options(self, options: FinalRequestOptions) -> None:
        headers: dict[str, str | Omit] = (
            {**options.headers} if is_given(options.headers) else {}
        )
        options.headers = headers
        headers["Authorization"] = f"Bearer {configs.driver_aad_token}"
        if "Accept" not in headers:
            headers["Accept"] = "application/json"
        if "ActivityId" not in headers:
            correlation_id = str(uuid.uuid4())
            headers["ActivityId"] = correlation_id

        return super()._prepare_options(options)

# Pretty printing helper
def pretty_print(messages):
    print("---Conversation---")
    for m in messages:
        print(f"{m.role}: {m.content[0].text.value}")
    print()

fabric_client = FabricOpenAI()
# Create assistant
assistant = fabric_client.beta.assistants.create(model="not used")
# Create thread
thread = fabric_client.beta.threads.create()
# Create message on thread
message = fabric_client.beta.threads.messages.create(thread_id=thread.id,
role="user", content=question)
# Create run
run = fabric_client.beta.threads.runs.create(thread_id=thread.id,
assistant_id=assistant.id)

# Wait for run to complete
while run.status == "queued" or run.status == "in_progress":
    run = fabric_client.beta.threads.runs.retrieve(

```

```
        thread_id=thread.id,
        run_id=run.id,
    )
print(run.status)
time.sleep(2)

# Print messages
response = fabric_client.beta.threads.messages.list(thread_id=thread.id,
order="asc")
pretty_print(response)

# Delete thread
fabric_client.beta.threads.delete(thread_id=thread.id)
```

Related content

- [Fabric data agent concept](#)
- [Create a Fabric data agent](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Ask the community ↗](#)

Consume Fabric data agent (preview)

Article • 04/15/2025

Data agent in Microsoft Fabric transforms enterprise data into conversational Q&A systems, and it enables users to interact with their data through chat, to uncover actionable insights. One way to consume Fabric data agent is through Azure AI Agent Service, a core component of Azure AI Foundry. Through integration of Fabric data agents with Azure AI Foundry, your Azure AI agents can directly tap into the rich, structured, and semantic data available in Microsoft Fabric OneLake. This integration provides immediate access to high-quality enterprise data, and it empowers your Azure AI agents to generate actionable insights and streamline analytical workflows. Organizations can then enhance data-driven decision-making with Fabric data agent as a powerful knowledge source within their Azure AI environments.

Important

This feature is in [preview](#).

Prerequisites

- A paid F64 or higher Fabric capacity resource
- Fabric data agent tenant settings is enabled.
- Copilot tenant switch is enabled.
- Cross-geo processing for AI is enabled.
- Cross-geo storing for AI is enabled.
- At least one of the following: A warehouse, a lakehouse, one or more Power BI semantic models, or a KQL database with data.
- Power BI semantic models via XMLA endpoints tenant switch is enabled for Power BI semantic model data sources.
- Developers and end users in Azure AI Foundry must at least have the `AI Developer` Role-Based Access Control (RBAC) role.

How it works

Agent Setup: In Azure AI Agent Service, create a new agent and add Fabric data agent as one of its knowledge sources. To establish this connection, you need the workspace ID and artifact ID for the Fabric data agent. The setup enables your Azure AI agent to evaluate available sources when it receives a query, ensuring that it invokes the correct tool to process the request. Currently, you can only add one Fabric data agent as a knowledge source to your Azure AI agent.

Note

The model you select in Azure AI Agent setup is only used for Azure AI agent orchestration and response generation. It doesn't affect the model that Fabric data agent uses.

Query Processing: When a user sends a query from the Foundry playground, the Azure AI Agent Service determines whether or not Fabric data agent is the best tool for the task. If it is, the Azure AI agent:

- Uses the end user's identity to generate secure queries over the data sources the user is permitted to access within the Fabric data agent.
- Invokes Fabric to fetch and process the data, to ensure a smooth, automated experience.
- Combines the results from Fabric data agent with its own logic to generate comprehensive responses. Identity Passthrough (On-Behalf-Of) authorization secures this flow, to ensure robust security and proper access control across enterprise data.

Note

The Fabric data agent and the Azure AI Foundry resources should be on the same tenant.

Adding Fabric data agent to your Azure AI Agent

You can add Fabric data agent to your Azure AI agent either programmatically or with the user interface (UI). Detailed code examples and further instructions are available in the Azure AI Agent integration documentation.

Add Fabric data agent through UI:

- Navigate to the left pane. Under **Build and Customize**, select **Agents**, as shown in the following screenshot:

The screenshot shows the 'Agents' section of the Azure AI Foundry portal. On the left sidebar, under 'My assets', the 'Agents' option is selected and highlighted with a red box. The main area displays a table of existing agents with columns for Name, ID, Created, and Model. A red box highlights the '+ New agent' button at the top left of the table. To the right, there's a 'Setup' panel for creating a new agent, including fields for Agent id, Agent name, Azure AI Services or Azure OpenAI resource, Deployment, Instructions, and Knowledge.

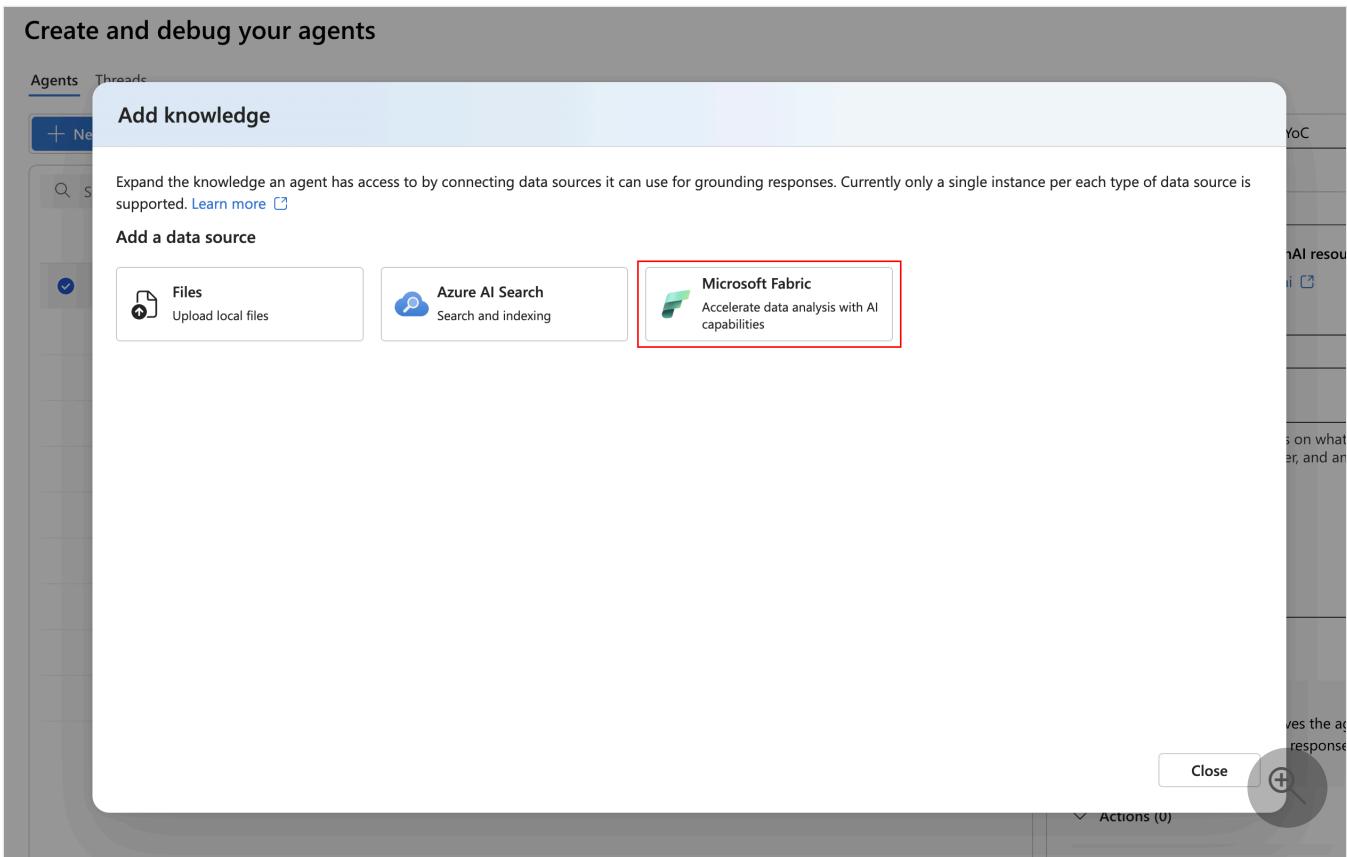
This displays the list of your existing Azure AI agents. You can add Fabric to one of these agents, or you can select **New Agent** to create a new agent. New agent creation generates a unique agent ID, and a default name. You can change that name at any time. For more information, please visit [What is Azure OpenAI in Azure AI Foundry portal](#).

- Initiate Adding a Knowledge Source: Select the **Add** button, as shown in the following screenshot:

The screenshot shows the same 'Agents' section as before, but the 'Knowledge' section on the right is expanded. It includes a 'Knowledge' icon, a description stating 'Knowledge gives the agent access to data sources for grounding responses.', and a '+ Add' button. The '+ Add' button is highlighted with a red box.

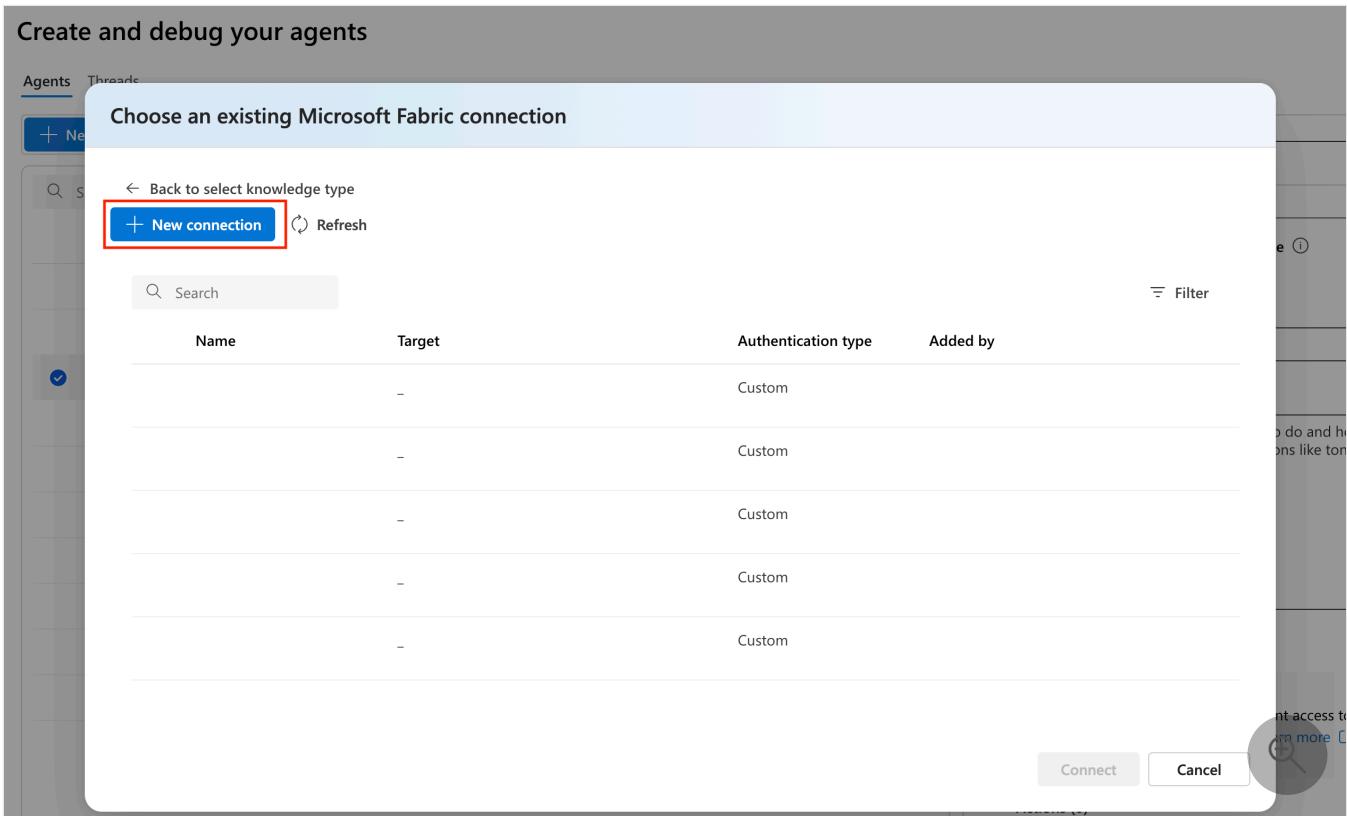
This opens a menu of supported knowledge source types.

- Select Microsoft Fabric as the Source: From the list, choose **Microsoft Fabric**, as shown in the following screenshot:

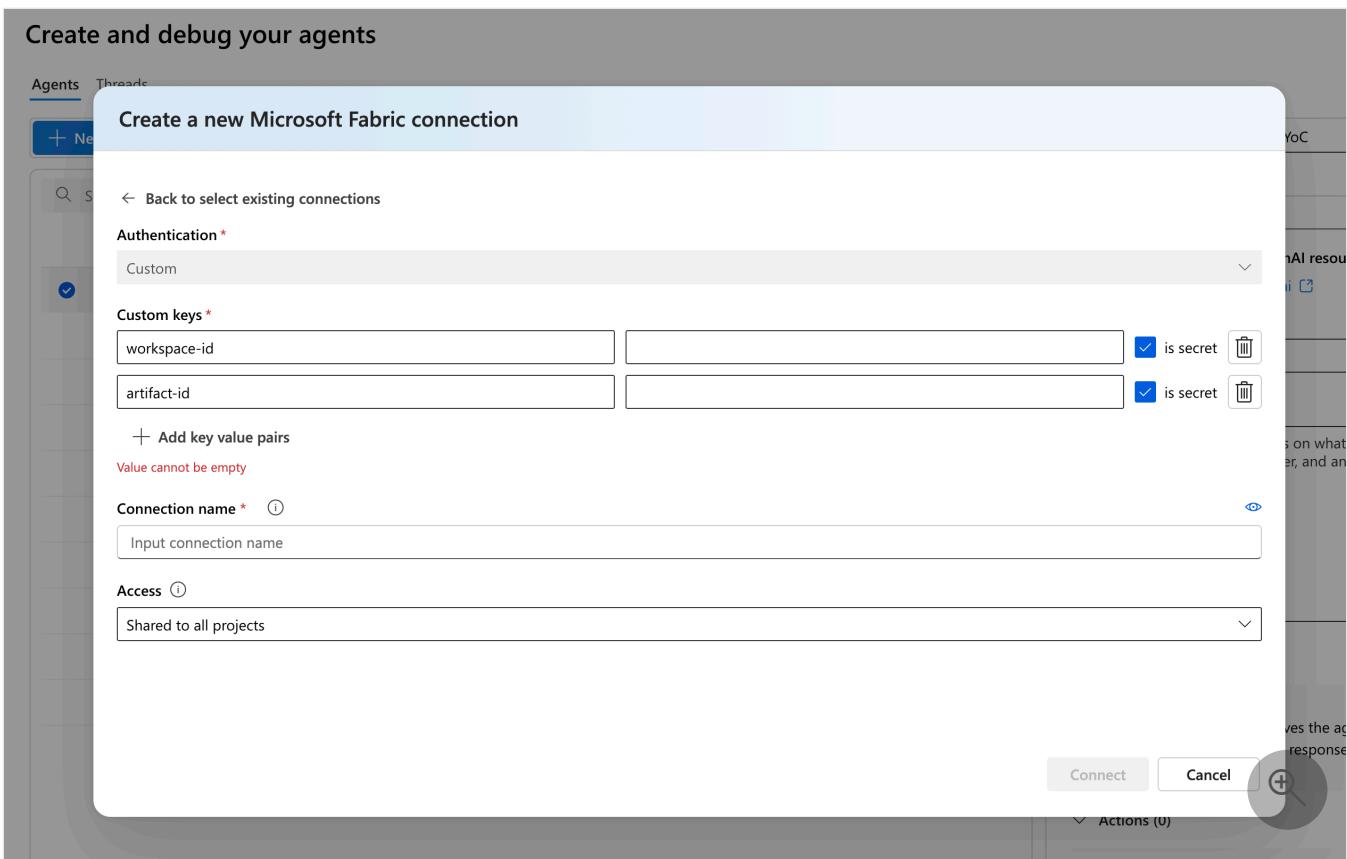


With this option, your agent can access Fabric data agent.

- Create a connection: If you previously established a connection to a Fabric data agent, you can reuse that connection for your new Azure AI Agent. Otherwise, select **New Connection** to create a connection, as shown in this screenshot:



The **Create a new Microsoft Fabric connection** window opens, as shown in this screenshot:



When you set up the connection, provide the Fabric data agent `workspace-id` and `artifact-id` values as custom keys. You can find the `workspace-id` and `artifact-id` values in the published Fabric data agent endpoint. Your Fabric data agent endpoint has this format:

<https://fabric.microsoft.com/groups/> <workspace_id>/aiskills/<artifact-id>, and select the **Is Secret** checkbox

Finally, assign a name to your connection, and choose whether to make it available to all projects in Azure AI Foundry or to restrict it to the current project.

Add Fabric data agent programmatically: The following steps describe how to add Fabric data agent programmatically to your Azure AI agent in Python. For other languages (C#, JavaScript) you can refer to [here](#).

Step 1: Create a project client

Create a client object that contains the connection string to connect to your AI project and other resources.

Python

```
import os
from azure.ai.projects import AIProjectClient
from azure.identity import DefaultAzureCredential
from azure.ai.projects.models import FabricTool
```

Step 2: Create an Agent with the Microsoft Fabric tool enabled

To make the Fabric data agent tool available to your Azure AI agent, use a connection to initialize the tool and attach it to the agent. You can find your connection in the [connected resources](#) section of your project in the Azure AI Foundry portal.

Python

```
# The Fabric connection ID can be found in the Azure AI Foundry project as a
# property of the Fabric tool
# Your connection ID is in the format /subscriptions/<your-subscription-
# id>/resourceGroups/<your-resource-
# group>/providers/Microsoft.MachineLearningServices/workspaces/<your-project-
# name>/connections/<your-fabric-connection-name>
conn_id = "your-connection-id"

# Initialize agent Fabric tool and add the connection ID
fabric = FabricTool(connection_id=conn_id)

# Create agent with the Fabric tool and process assistant run
with project_client:
    agent = project_client.agents.create_agent(
        model="gpt-4o",
        name="my-assistant",
        instructions="You are a helpful assistant",
        tools=fabric.definitions,
        headers={"x-ms-enable-preview": "true"},
    )
    print(f"Created agent, ID: {agent.id}")
```

Step 3: Create a thread

Python

```
# Create thread for communication
thread = project_client.agents.create_thread()
print(f"Created thread, ID: {thread.id}")

# Create message to thread
# Remember to update the message with your data
message = project_client.agents.create_message(
    thread_id=thread.id,
    role="user",
    content="what is top sold product in Contoso last month?",
)
print(f"Created message, ID: {message.id}")
```

Step 4: Create a run and check the output

Create a run and observe that the model uses the Fabric data agent tool to provide a response to the user's question.

Python

```
# Create and process agent run in thread with tools
run = project_client.agents.create_and_process_run(thread_id=thread.id,
assistant_id=agent.id)
print(f"Run finished with status: {run.status}")

if run.status == "failed":
    print(f"Run failed: {run.last_error}")

# Delete the assistant when done
project_client.agents.delete_agent(agent.id)
print("Deleted agent")

# Fetch and log all messages
messages = project_client.agents.list_messages(thread_id=thread.id)
print(f"Messages: {messages}")
```

Related content

- [Data agent concept](#)
- [Data agent scenario](#)

Configure Fabric data agent tenant setting

Article • 03/31/2025

To use a data agent in Microsoft Fabric, you must configure the required tenant settings. Additionally, if your Fabric data agent uses a Power BI semantic model as a data source, specific tenant settings must be enabled to allow connectivity. This guide walks you through the necessary configurations for a seamless setup.

Accessing tenant settings

To configure the required settings, you need administrative privileges to access the [Admin Portal](#) in Microsoft Fabric.

1. Sign in to Microsoft Fabric with an admin account.
2. Open the Admin Portal:
 - Select the gear icon in the top-right corner.
 - Select **Admin Portal**.
3. Navigate to Tenant Settings:
 - In the Admin Portal, select **Tenant settings** from the left-hand navigation pane.

Once you are in **Tenant Settings**, you can proceed with enabling the necessary configurations.

Enable Copilot and Azure OpenAI tenant switch

For a Fabric data agent to function properly, the [Copilot and Azure OpenAI Service](#) tenant settings must be enabled. These settings control user access and data processing policies.

Required settings

- Users can use Copilot and other features powered by Azure OpenAI:
 - This must be enabled to allow users to access Copilot-powered features, including Fabric data agent. This setting can be managed at both the tenant and the capacity levels. For more information, see [Overview of Copilot in Fabric](#).

- To enable this setting, check the option in **Tenant Settings** as shown in the next screenshot:

Admin portal

Tenant settings New

- Usage metrics
- Users
- Premium Per User
- Audit logs
- Domains New
- Workloads
- Tags (preview) New
- Capacity settings
- Refresh summary
- Embed Codes
- Organizational visuals
- Azure connections
- Workspaces
- Custom branding
- Fabric identities
- Featured content
- Microsoft Purview setting
- Help + support
- Data Policies

Copilot and Azure OpenAI Service

⚠️ Users can use Copilot and other features powered by Azure OpenAI
Enabled for the entire organization

When this setting is enabled, users can access the features powered by Azure OpenAI, including Copilot. This setting can be managed at both the tenant and the capacity levels. [Learn More](#)

For customers in the EU Data Boundary, this setting adheres to Microsoft Fabric's EU Data Boundary commitments. [Learn More](#)

By enabling this setting, you agree to the [Preview Terms](#).

Enabled

⚠️ Note: Copilot in Fabric is now generally available, starting with the Microsoft Power BI experience. The Copilot in Fabric experiences for Data Factory, Data Engineering, Data Science, Data Warehouse, and Real-Time Intelligence are in preview.

⚠️ Note: If Azure OpenAI is not available in your geographic region, your data may need to be processed outside your capacity's geographic region, compliance boundary, or national cloud instance. To allow data to be processed outside your capacity's geographic region, turn on the related setting, "Data sent to Azure OpenAI can be processed outside your capacity's geographic region, compliance boundary, or national cloud instance".

Apply to:

The entire organization

Specific security groups

Except specific security groups

Delegate setting to other admins ⓘ

Select the admins who can view and change this setting, including any security group selections you've made.

Capacity admins can enable/disable

- **Data sent to Azure OpenAI can be processed outside your capacity's geographic region, compliance boundary, or national cloud instance**
 - Required for customers using Fabric data agent whose capacity's geographic region is outside of the EU data boundary and the US.
 - To enable this setting, check the option in **Tenant Settings** as shown in the next screenshot:

Admin portal

Tenant settings New

Copilot and Azure OpenAI Service

Users can use Copilot and other features powered by Azure OpenAI
Enabled for the entire organization

Data sent to Azure OpenAI can be processed outside your capacity's geographic region, compliance boundary, or national cloud instance
Enabled for the entire organization

This setting is only applicable for customers who want to use Copilot and AI features in Fabric powered by Azure OpenAI, and whose capacity's geographic region is outside of EU Data Boundary and US. [Learn More](#)

When this setting is enabled, data sent to Azure OpenAI can be processed outside your capacity's geographic boundary or national cloud boundary. This setting can be managed at both the tenant and the capacity levels. [Learn More](#)

By enabling this setting, you agree to the [Preview Terms](#).

Enabled

Note: Even if this setting is on, you will also need to turn on the related setting "Users can use Copilot and other features powered by Azure OpenAI" for these features to work.

Apply to:

The entire organization

Specific security groups

Except specific security groups

Delegate setting to other admins

Select the admins who can view and change this setting, including any security group selections you've made.

Capacity admins can enable/disable

Apply Cancel

- **Data sent to Azure OpenAI can be stored outside your capacity's geographic region, compliance boundary, or national cloud instance**
 - Required for customers using Fabric data agent whose capacity's geographic region is outside of the EU data boundary and the US.
 - To enable this setting, check the option in **Tenant Settings** as shown in the next screenshot:

Admin portal

Tenant settings New

Users can use Copilot and other features powered by Azure OpenAI
Enabled for the entire organization

Data sent to Azure OpenAI can be processed outside your capacity's geographic region, compliance boundary, or national cloud instance
Enabled for the entire organization

Capacities can be designated as Fabric Copilot capacities
Enabled for the entire organization

Data sent to Azure OpenAI can be stored outside your capacity's geographic region, compliance boundary, or national cloud instance
Enabled for the entire organization

This setting is only applicable for customers who want to use Copilot and AI features in Fabric powered by Azure OpenAI, and whose capacity's geographic region is outside of EU Data Boundary and US. [Learn More](#)

When this setting is enabled, data sent to Azure OpenAI can be stored outside your capacity's geographic boundary or national cloud boundary. This setting can be managed at both the tenant and the capacity levels. [Learn More](#)

By enabling this setting, you agree to the [Preview Terms](#).

Enabled

Note: Even if this setting is on, you will also need to turn on the related setting "Users can use Copilot and other features powered by Azure OpenAI" for these features to work.

Note: This setting is only applicable for customers who want to use a preview of generative AI experiences in Fabric, such as Copilot for Data Science and Data Engineering and AI skills.

Apply to:

The entire organization

Specific security groups

Except specific security groups

Apply Cancel

Enable Fabric data agent tenant settings

By default, the Fabric data agent feature is disabled at the tenant level. To allow users to create and share Fabric data agent items, administrators must enable this setting. This activation allows users to craft natural language Q&A experiences using generative AI, and then share the Fabric data agent within the organization.

Steps to enable Fabric data agent

1. In **Tenant Settings**, locate the **Fabric data agent** section.
2. To enable this setting, check the option in **Tenant Settings** as shown in the next screenshot:

The screenshot shows the Microsoft Admin portal's Tenant settings page. On the left, there's a sidebar with various administrative options like Usage metrics, Users, Premium Per User, Audit logs, Domains, Workloads, Tags, Capacity settings, Refresh summary, Embedded Codes, Organizational visuals, Azure connections, Workspaces, Custom branding, Fabric identities, Featured content, and Microsoft Purview setting. The 'Tenant settings' tab is selected. In the main content area, there's a message: 'There are new or updated tenant settings. Expand to review the changes.' Below it, under the 'Microsoft Fabric' section, there's a box titled 'Users can create and share Data agent item types (preview)' with the subtext 'Enabled for the entire organization'. A note below says 'Users can create natural language data question and answer (Q&A) experiences using generative AI and then save them as Data agent items. Data agent items can be shared with others in the organization.' A 'Learn More' link is present. A red box highlights the 'Enabled' toggle switch, which is currently turned on. Below the toggle, there are 'Apply to:' options: 'The entire organization' (selected), 'Specific security groups', and 'Except specific security groups'. There's also a 'Delegate setting to other admins' section and a 'Capacity admins can enable/disable' checkbox. At the bottom are 'Apply' and 'Cancel' buttons, and a magnifying glass icon.

Enable integration of Power BI semantic models via XMLA endpoints

Fabric data agents can query and manage Power BI semantic models programmatically via XMLA (XML for Analysis) endpoints. To enable this functionality, XMLA endpoints must be configured correctly.

Steps to enable XMLA endpoints

1. In **Tenant Settings**, navigate to the **Integration settings** section.
2. Locate **Allow XMLA endpoints** and **Analyze in Excel with on-premises datasets** and then enable it, as shown in the next screenshot:

Admin portal

Tenant settings New

- Usage metrics
- Users
- Premium Per User
- Audit logs
- Domains New
- Workloads
- Tags (preview) New
- Capacity settings
 - Refresh summary
 - Embed Codes
 - Organizational visuals
 - Azure connections
 - Workspaces
 - Custom branding
 - Protection metrics
 - Fabric identities
 - Featured content
- Help + support

There are new or updated tenant settings. Expand to review the changes.

Export and sharing settings

Users can work with semantic models in Excel using a live connection
Enabled for the entire organization

Integration settings

Allow **XMLA** endpoints and Analyze in Excel with on-premises semantic models
Enabled for the entire organization

Users in the organization can use Excel to view and interact with on-premises Power BI semantic models. This also allows connections to **XMLA** endpoints.

Enabled

Apply to:

The entire organization

Specific security groups

Except specific security groups

Apply Cancel

xmla



Related content

- [Data agent concept](#)
- [About tenant settings](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Fabric data agent sharing and permission management (preview)

Article • 03/31/2025

ⓘ Important

This feature is in [preview](#).

Prerequisites

- A paid F64 or higher Fabric capacity resource
- Fabric data agent tenant settings is enabled.
- Copilot tenant switch is enabled.
- Cross-geo processing for AI is enabled.
- Cross-geo storing for AI is enabled.
- At least one of these: A warehouse, a lakehouse, one or more Power BI semantic models, or a KQL database with data.
- Power BI semantic models via XMLA endpoints tenant switch is enabled for Power BI semantic model data sources.

Publishing and versioning

Creation of a data agent in Microsoft Fabric is an iterative process. It involves refinement of various configurations, for example

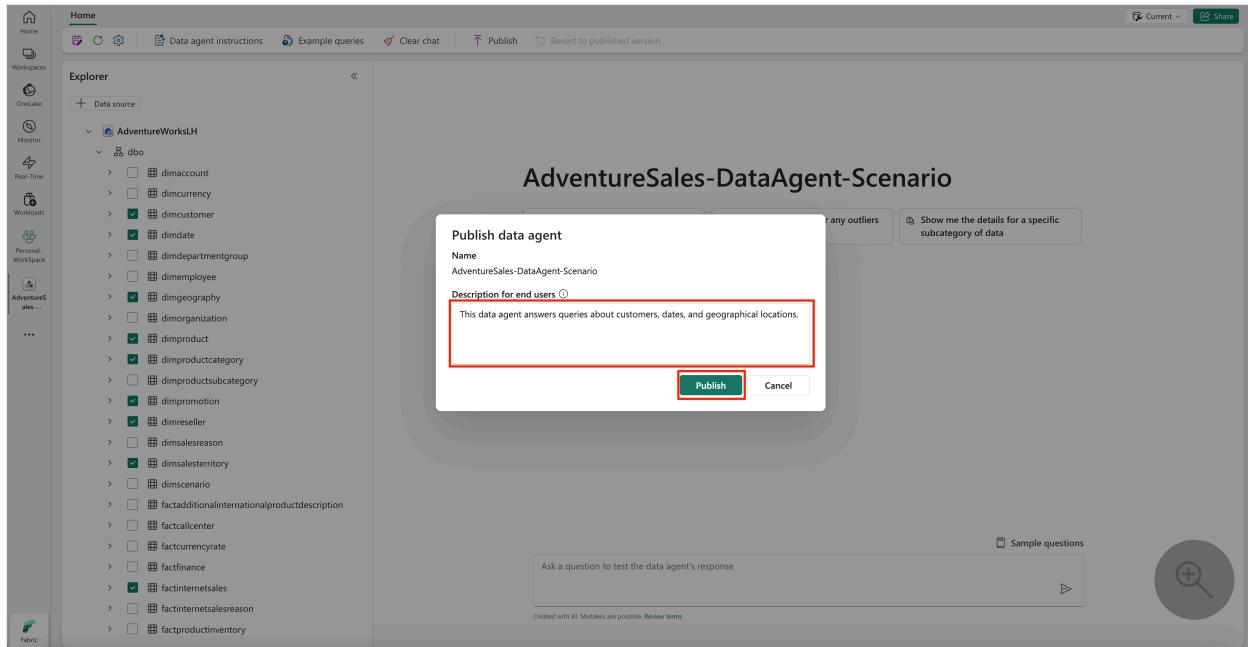
- selection of relevant tables
- Fabric data agent instruction definition
- production of example queries for each data source

As you make adjustments to enhance the performance of the Fabric data agent, you can eventually publish that Fabric data agent. Once published, a read-only version is generated, which you can share with others.

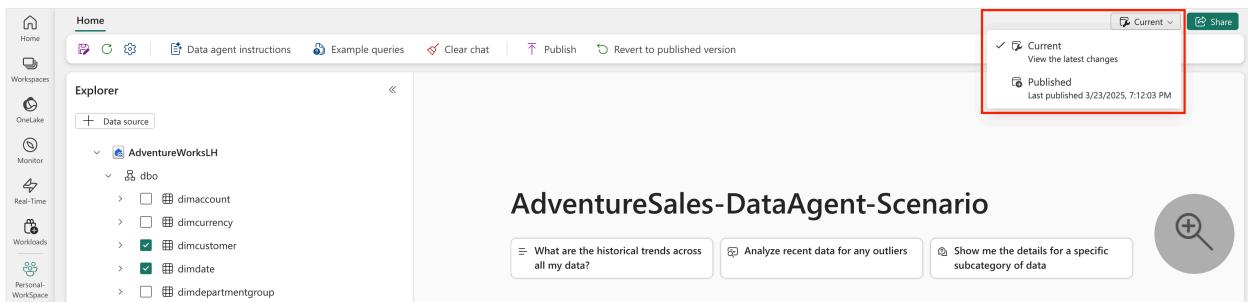
When you try to publish the Fabric data agent, you can include a description that explains what the Fabric data agent does. The description is available to consumers of the Fabric data agent, to help them understand its purpose and functionality. Other automated systems and orchestrators can also use the description, to invoke the Fabric data agent outside of Microsoft Fabric.

(!) Note

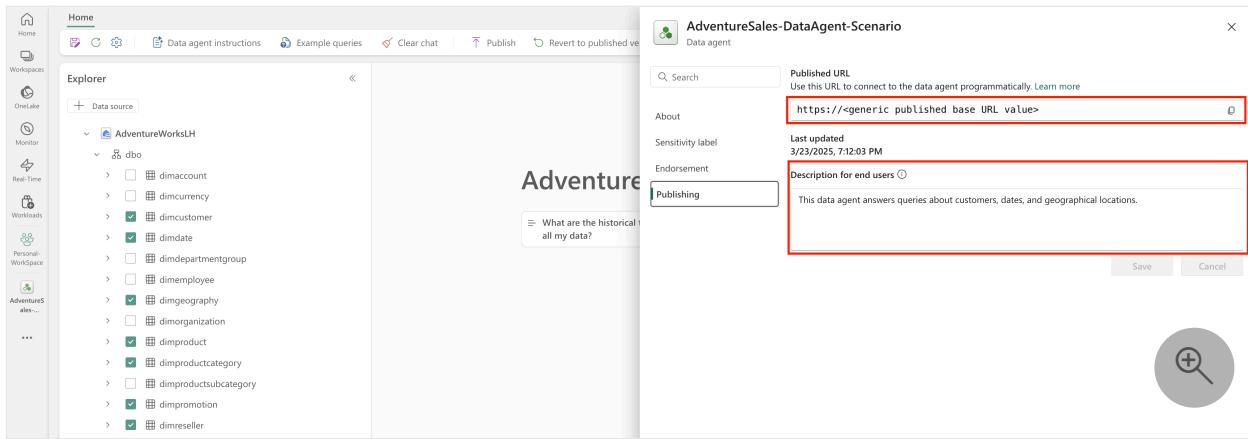
You can ask the Fabric data agent to describe what it does. You can then refine and summarize the response to use as its description when publishing.



After you publish your Fabric data agent, you can continue to refine its current draft version to enhance its performance, without affecting the published version that other people use. This way, you can iterate with confidence, knowing that your changes remain isolated from the published version. You can seamlessly switch between the published and draft versions, testing the same set of queries on both to compare their performance. This helps you assess the effects of your changes, and you can gain valuable insights into how they improve the effectiveness of your Fabric data agent. The following screenshot shows how to switch between published and developed Fabric data agent versions:

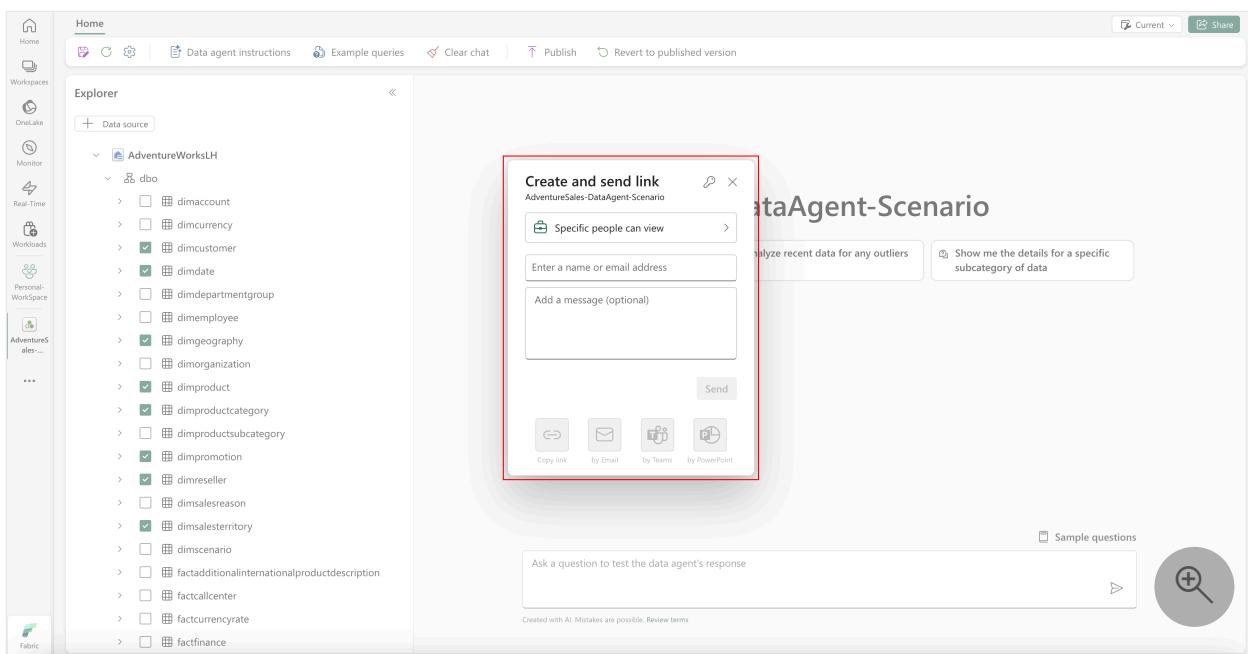


To update the Fabric data agent description without making any other changes, navigate to **Settings**, select **Publishing**, and then update the description, as shown in this screenshot:



Permission models for sharing the Fabric data agent

The **Fabric data agent sharing** feature allows you to share your Fabric data agents with others, with a range of permission models, as shown in this screenshot:



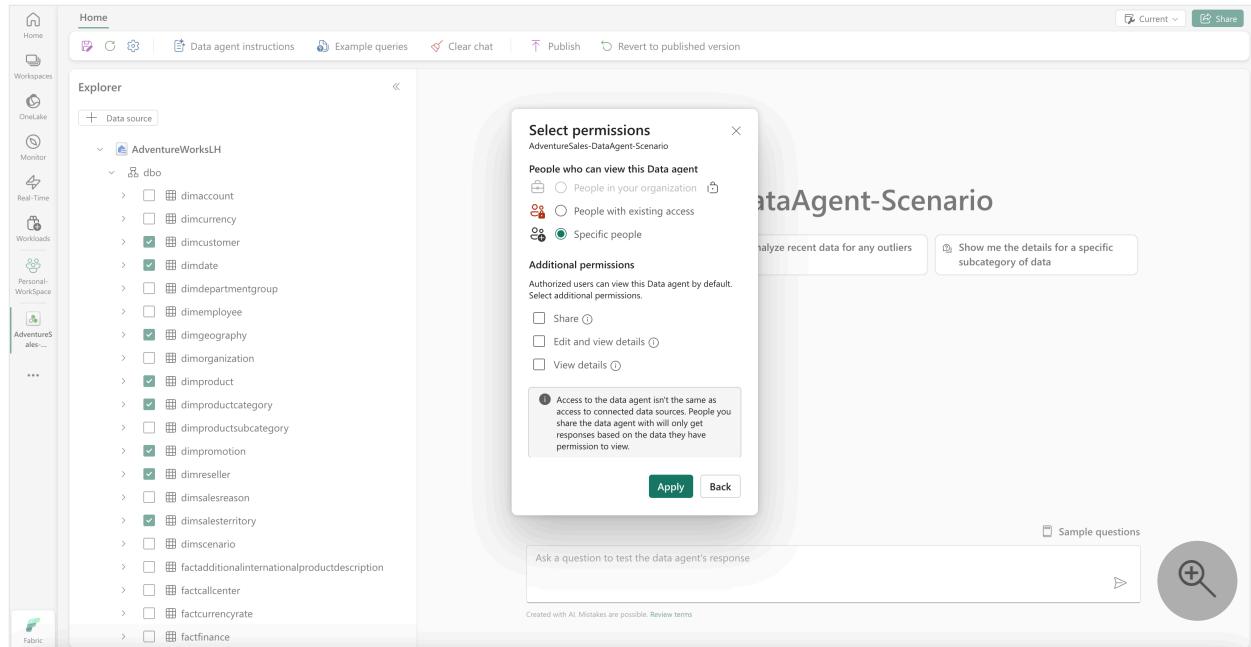
You have complete control over access to your Fabric data agent, and complete control of its use. Additionally, when you share the Fabric data agent, you must also share access to the underlying data it uses. The Fabric data agent honors all user permissions to the data, including Row-Level Security (RLS) and Column-Level Security (CLS).

- No permission selected:** If you don't select any other permission, users can only query the **published** version of the Fabric data agent. They have no access to edit or even view any configurations or details. This maintains the integrity of your Fabric data agent set-up.
- View details:** Users can view the details and configurations of both the published and draft versions of the Fabric data agent, but they can't make any changes to it.

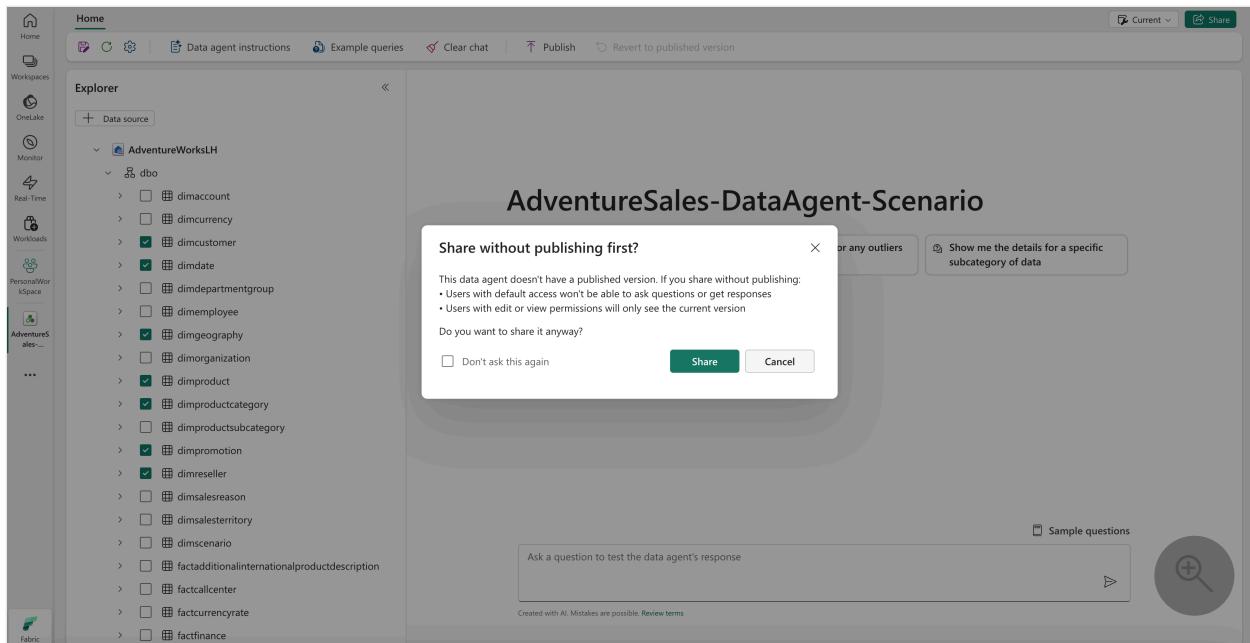
However, they can still query the Fabric data agent, and build informative insights without risk of unintended modifications.

- **Edit and view details:** Users have full access to view and edit all the details and configurations of both the published and draft versions of the Fabric data agent. They can also query the Fabric data agent, which makes it ideal for collaborative work.

The following screenshot shows the actual permissions that you can select:



If you share a Fabric data agent before you publish it, users with default permissions (without any other permissions) can't query it. It works this way because the default permission allows users to query only the published version—if a published version doesn't yet exist, users can't query the Fabric data agent. Users with other permissions (**View details**, or **Edit and view details**) can only access the draft version. The following screenshot shows the option to share a Fabric data agent without publishing it:



Related content

- [Data agent concept](#)
- [Fabric data agent tenant settings](#)
- [Create a Fabric data agent](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Fabric Data Agent Python SDK

Article • 03/31/2025

The Fabric Data Agent Python SDK library facilitates programmatic access to Fabric Data Agent artifacts. The SDK is designed for code-first users, and it simplifies the creation, management, and utilization of Fabric data agents within Microsoft Fabric notebooks. It provides a set of straightforward APIs to integrate and manage data sources, automate workflow operations, and interact with the Fabric data agent, based on the OpenAI Assistants API within Microsoft Fabric notebook.

Prerequisites

- **Python Version:** A compatible version of Python (typically Python ≥ 3.10).
- **Dependencies:** The SDK might require other packages. Pip automatically installs these packages.
- **Environment:** This SDK is designed to work exclusively within Microsoft Fabric notebooks. It isn't supported for local execution.

Features

- **Programmatic Management:** Create, update, and delete Data Agent artifacts seamlessly.
- **Data Source Integration:** Easily connect to and integrate multiple data sources, for enhanced data analysis and insight generation.
- **OpenAI Assistants API Support:** Use the OpenAI Assistants API for rapid prototyping and experimentation.
- **Workflow Automation:** Automate routine tasks to reduce manual efforts and improve operational efficiency.
- **Resource Optimization:** Optimize configuration and management of Data Agent resources to better align with specific customer needs.

Installation

Use pip to install the Fabric data agent Python SDK:

```
pip install fabric-data-agent-sdk
```

Quick Start Example

For more information about the sample notebooks that show how to use the Fabric Data Agent Python SDK, visit [this GitHub repo](#) resource.

Related content

- [Fabric data agent creation](#)
 - [Fabric data agent scenario](#)
 - [Fabric data agent sharing](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

How to use Microsoft Fabric notebooks

Article • 04/08/2025

The Microsoft Fabric notebook is a primary code item for developing Apache Spark jobs and machine learning experiments. It's a web-based interactive surface used by data scientists and data engineers to write code benefiting from rich visualizations and Markdown text. Data engineers write code for data ingestion, data preparation, and data transformation. Data scientists also use notebooks to build machine learning solutions, including creating experiments and models, model tracking, and deployment.

With a Fabric notebook, you can:

- Get started with zero set-up effort.
- Easily explore and process data with intuitive low-code experience.
- Keep data secure with built-in enterprise security features.
- Analyze data across raw formats (CSV, txt, JSON, etc.), processed file formats (parquet, Delta Lake, etc.), using powerful Spark capabilities.
- Be productive with enhanced authoring capabilities and built-in data visualization.

This article describes how to use notebooks in data science and data engineering experiences.

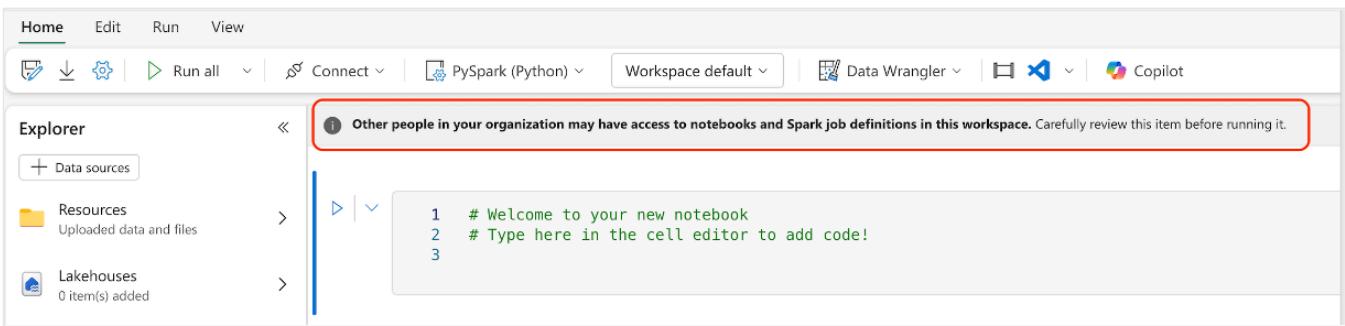
Security context of running notebook

The execution of a notebook can be triggered by three different manners in Fabric with full flexibility to meet different scenarios:

- **Interactive run:** User manually triggers the execution via the different UX entries or calling the REST API. The execution would be running under the current user's security context.
- **Run as pipeline activity:** The execution is triggered from Fabric Data Factory pipeline. You can find the detail steps in the [Notebook Activity](#). The execution would be running under the pipeline owner's security context.
- **Scheduler:** The execution is triggered from a scheduler plan. The execution would be running under the security context of the user who setup/update the scheduler plan.

The flexibility of these execution options with different security context allows you to meet different scenarios and requirements, but also requires you to be aware of the security context when you design and develop your notebook, otherwise it may cause unexpected behavior and even some security issues.

The first time when a notebook is created, a warning message is shown to remind you the risk of running the code without reviewing it.



Here are some best practices to help you avoid security issues:

- Before you manually run the notebook, Open the Notebook setting and check the Detail section under the About panel for the modification update, make sure you are OK with the latest change.
- Before you add a notebook activity to a pipeline, Open the Notebook setting and check the Detail section under the About panel for the modification update, make sure you are OK with the latest change. If you are not sure about the latest change, better open the Notebook to review the change before you add it into the pipeline.
- Before you update the scheduler plan, Open the Notebook setting and check the Detail section under the About panel for the modification update, make sure you are OK with the latest change. If you are not sure about the latest change, better open the Notebook to review the change before you update the scheduler plan.
- Separate the workspace into different stage (dev, test, prod) and control the access of different stage to avoid the security issue. Only add the user who you trust to the prod stage.

Create notebooks

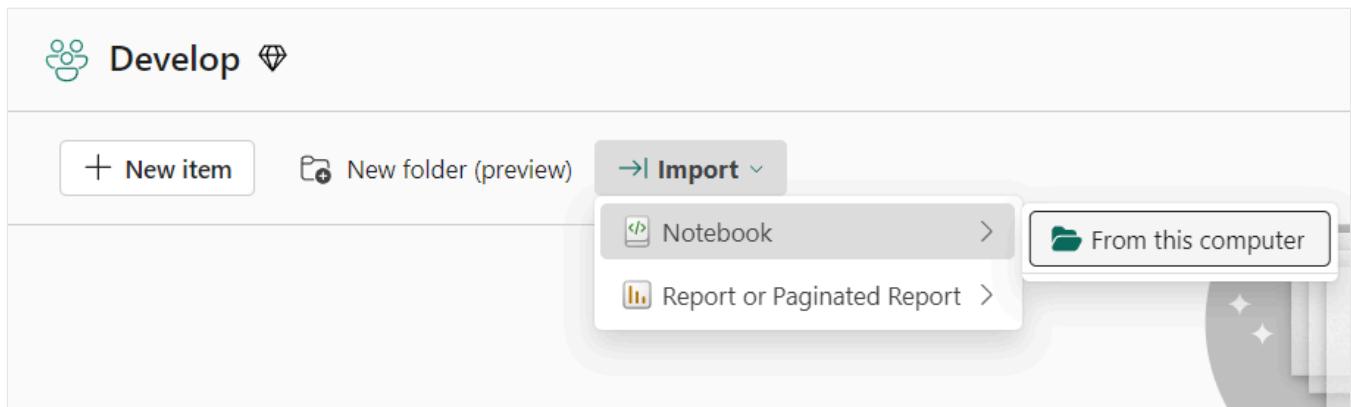
You can either create a new notebook or import an existing notebook.

Create a new notebook

Like other standard Fabric item creation processes, you can easily create a new notebook from the Fabric **Data Engineering** homepage, the workspace **New** option, or the **Create Hub**.

Import existing notebooks

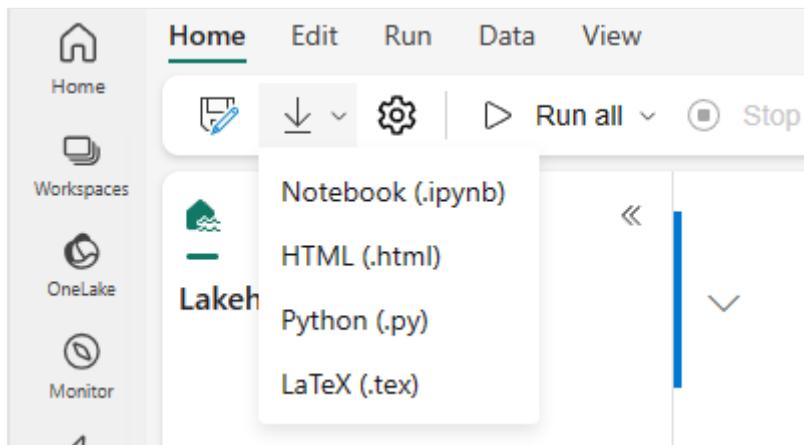
You can import one or more existing notebooks from your local computer using the entry in the workspace toolbar. Fabric notebooks recognize the standard Jupyter Notebook *.ipynb* files, and source files like *.py*, *.scala*, and *.sql*, and create new notebook items accordingly.



Export a notebook

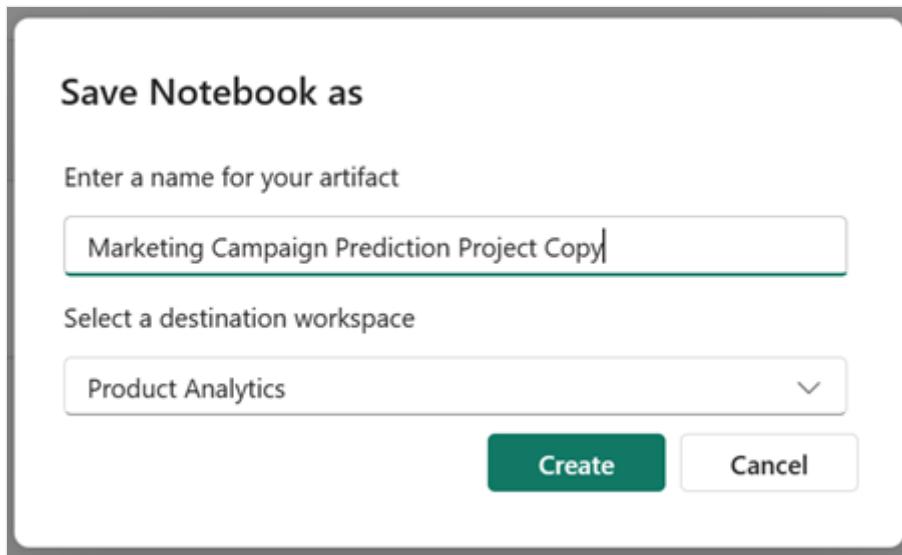
You can export your notebook to other standard formats. Synapse notebook can be exported into:

- The standard notebook file (.ipynb) that is used for Jupyter notebooks.
- An HTML file (.html) that can be opened from a browser directly.
- A Python file (.py).
- A Latex file (.tex).

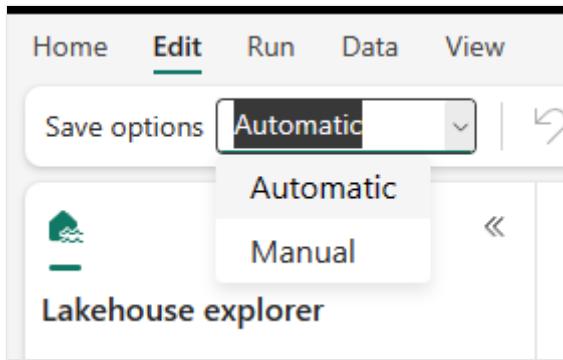


Save a notebook

In Fabric, a notebook will by default save automatically after you open and edit it; you don't need to worry about losing code changes. You can also use **Save a copy** to clone another copy in the current workspace or to another workspace.



If you prefer to save a notebook manually, you can switch to the **Manual** save option to have a local branch of your notebook item, and then use **Save** or **CTRL+s** to save your changes.

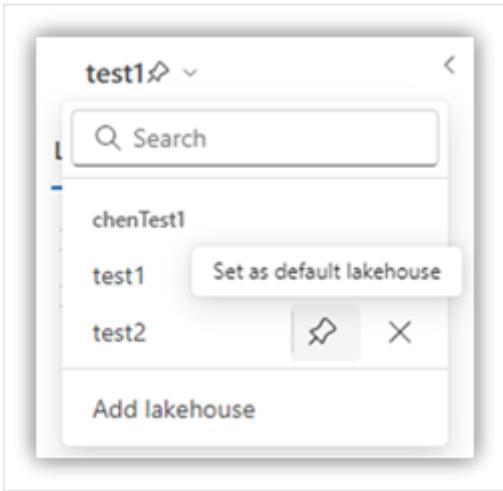


You can also switch to manual save mode by selecting **Edit** -> **Save options** -> **Manual**. To turn on a local branch of your notebook then save it manually, select **Save** or use the **Ctrl+s** keyboard shortcut.

Connect lakehouses and notebooks

Fabric notebooks now support close interactions with lakehouses; you can easily add a new or existing lakehouse from the Lakehouse explorer.

You can navigate to different lakehouses in the Lakehouse explorer and set one lakehouse as the default by pinning it. Your default is then mounted to the runtime working directory, and you can read or write to the default lakehouse using a local path.



(!) Note

You must restart the session after pinning a new lakehouse or renaming the default lakehouse.

Add or remove a lakehouse

Selecting the X icon beside a lakehouse name removes it from the notebook tab, but the lakehouse item still exists in the workspace.

Select **Add lakehouse** to add more lakehouses to the notebook, either by adding an existing one or creating a new lakehouse.

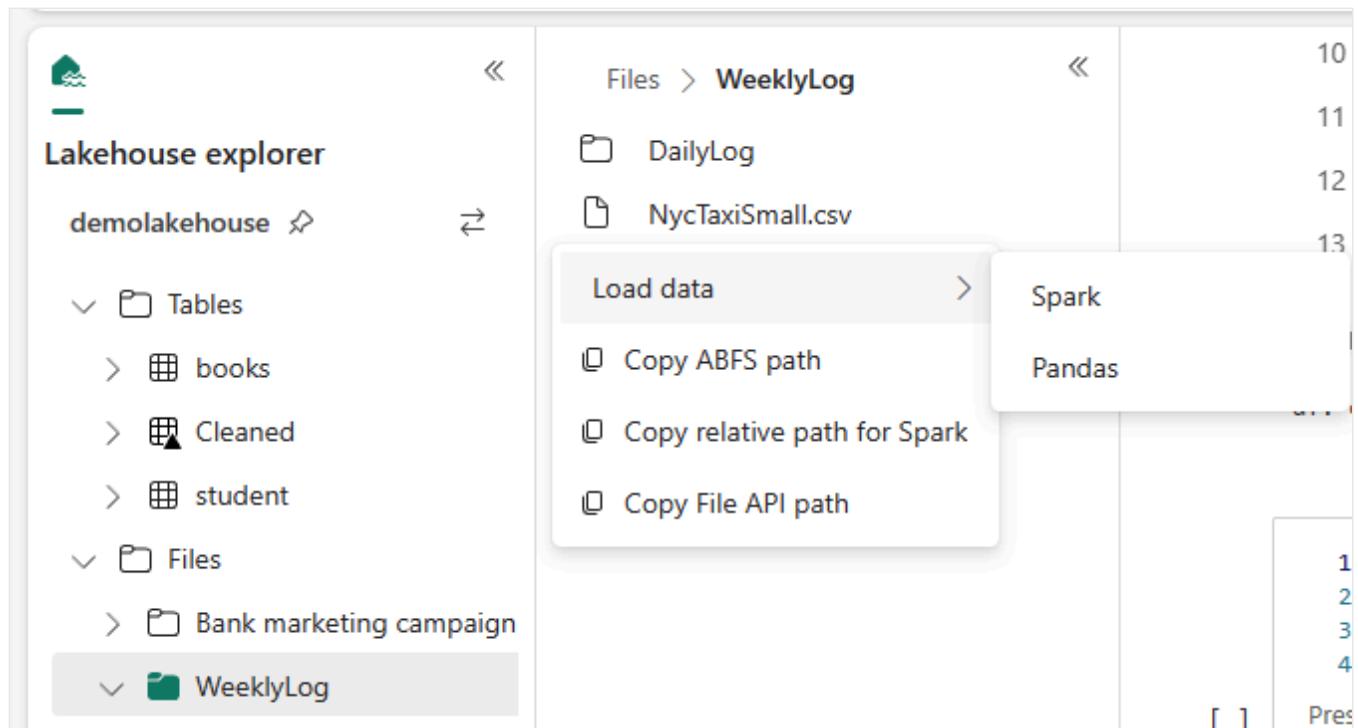
Explore a lakehouse file

The subfolder and files under the **Tables** and **Files** section of the **Lake** view appear in a content area between the lakehouse list and the notebook content. Select different folders in the **Tables** and **Files** section to refresh the content area.

Folder and file operations

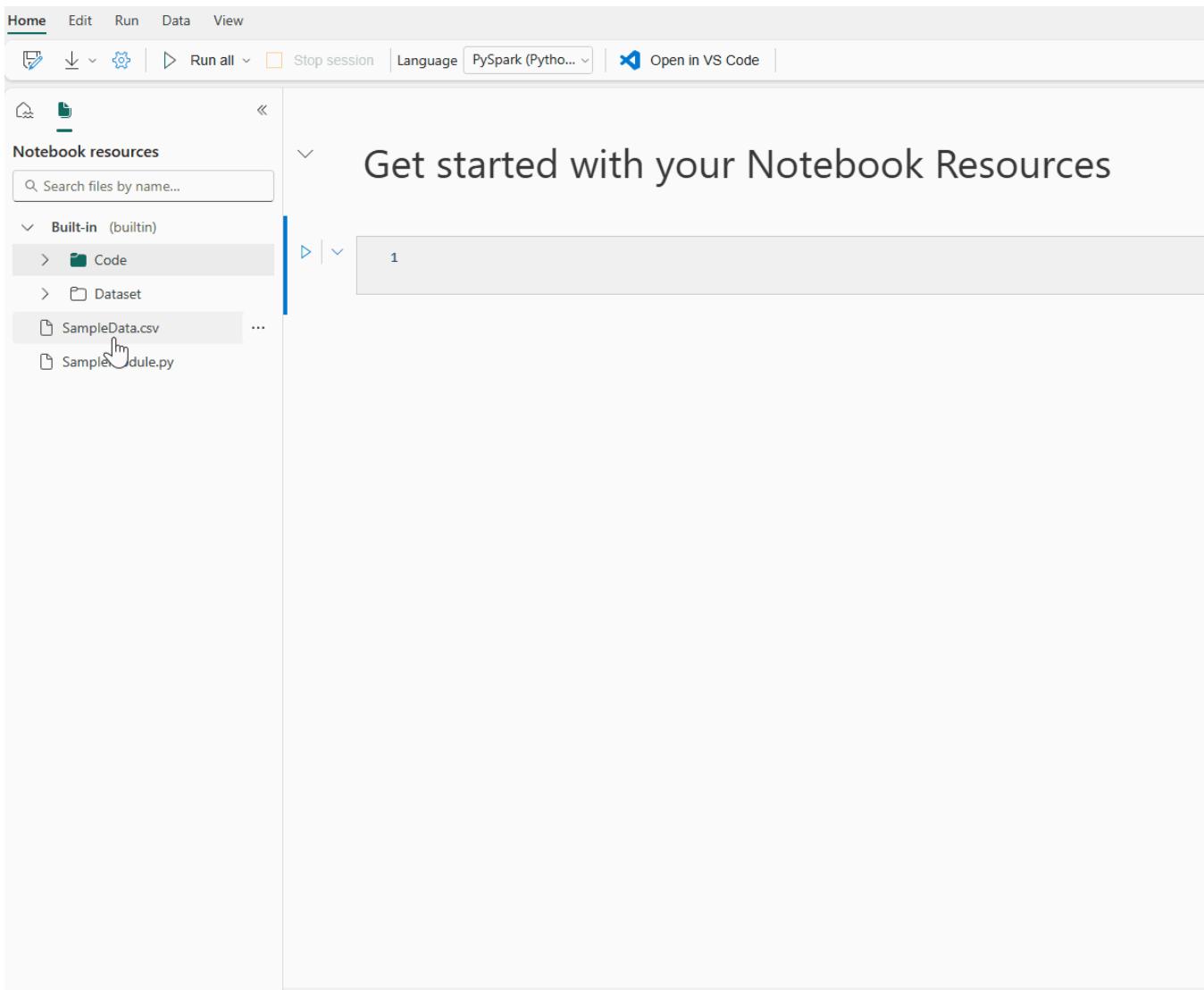
If you select a file (.csv,.parquet,.txt,.jpg,.png, etc.) with a right mouse click, you can use the Spark or Pandas API to load the data. A new code cell is generated and inserted beneath the focus cell.

You can easily copy a path with a different format from the select file or folder and use the corresponding path in your code.



Notebook resources

The notebook resource explorer provides a Unix-like file system to help you manage your folders and files. It offers a writeable file system space where you can store small-sized files, such as code modules, semantic models, and images. You can easily access them with code in the notebook as if you were working with your local file system.



(!) Note

- The maximum Resource storages for both built-in folder and environment folder are **500 MB**, with a single file size up to **100 MB**. They both allow up to **100** file/folder instances in total.
- When using `notebookutils.notebook.run()`, use the `notebookutils.nbResPath` command to access the target notebook resource. The relative path **builtin/** will always point to the root notebook's built-in folder.

Built-in resources folder

The built-in resources folder is a system-defined folder unique to each notebook. It is recommended to use built-in resource folder to storage any data used in the current notebook. Here are the key capabilities for the notebook resources.

- You can use common operations such as create/delete, upload/download, drag/drop, rename, duplicate, and search through the UI.

- You can use relative paths like `builtin/YourData.txt` for quick exploration. The `notebookutils.nbResPath` method helps you compose the full path.
- You can easily move your validated data to a lakehouse via the **Write to lakehouse** option. Fabric embeds rich code snippets for common file types to help you quickly get started.
- These resources are also available for use in the [Reference notebook run](#) case via `notebookutils.notebook.run()`.

Environment resources folder

Environment Resources Folder is a shared repository designed to streamline collaboration across multiple notebooks.

- You can find the **Resources** tab inside the environment and have the full operations to manage the resource files here. These files can be shared across multiple notebooks once the notebook is attached to the current environment.

	Name	Size	Last modified	Type
<input type="checkbox"/>	Dataset	-	1/1/1970 8:00 AM	Folder

- In the Notebook page, you can easily find a second root folder under Resources inherited from the attached environment.

The screenshot shows the Databricks notebook interface. On the left, the 'Resources' sidebar displays the file tree. A red box highlights the 'envtest (env)' folder, which contains a 'Dataset' folder and files 'train.csv' and 'userdata1.parquet'. In the main area, three code cells are visible, each with syntax highlighting and numbered lines. The first cell shows code for loading a Python file and using its members. The second cell shows code for reading and plotting an image. The third cell shows code for reading and processing various file types (txt, json, yaml). The status bar at the bottom indicates 'Not connected' and 'Copilot completions: Off'.

```

1 # Load python file
2 import builtin.Test as Test
3 # Now use the exported members from this module with identifier: `Test`
4 dir(test)

[1] - Session ready in 14 sec 608 ms. Command executed in 4 sec 368 ms by

...[['jack', 34, '36'], ['Riti', 30, 'Delhi'], ['Aadi', 16, 'New York']]
[__builtins__,
 __cached__,
 __doc__,
 __file__,
 __loader__,
 __name__,
 __package__,
 __spec__,
 students]

1 # Note: NotebookUtils is only supported on runtime v1.2 and above. If you are using runtime v1.1, please use mssparkutils instead.
2 # Load image
3 import matplotlib.pyplot as plt
4 import matplotlib.image as mpimg
5 image = mpimg.imread(f"{notebookutils.nbResPath}/builtin/yourFile")
6 # Let the axes disappear
7 plt.axis('off')
8 # Plot image in the output
9 image_plot = plt.imshow(image)

1 # Note: NotebookUtils is only supported on runtime v1.2 and above. If you are using runtime v1.1, please use mssparkutils instead.
2 import pandas as pd
3 # Load txt file
4 df_txt = pd.read_csv(f"{notebookutils.nbResPath}/builtin/yourFile.txt", sep="\n")
5
6 # Load json file
7 df_json = pd.read_json(f"{notebookutils.nbResPath}/builtin/yourFile.json")
8
9 # Load yaml file
10 df_yaml = pd.read_csv(f"{notebookutils.nbResPath}/builtin/yourFile.yaml", sep="\s+")
11

```

- You can also operate on the files/folders same with the Built-in resources folder.
- The Environment resource path is automatically mounted to the notebook cluster. You can use the relative path `/env` to access the environment resources.

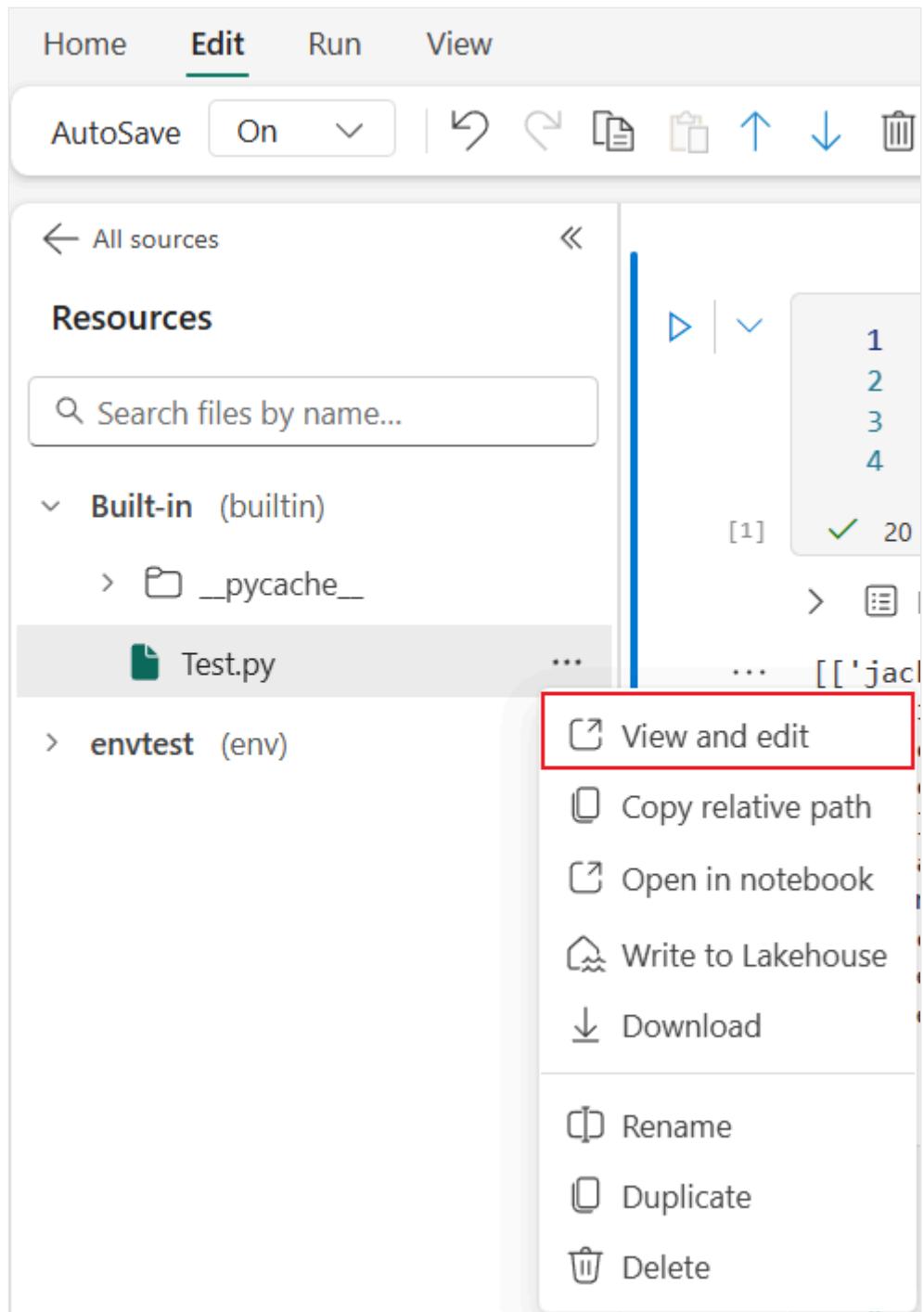
(!) Note

Reading/writing with a relative path is not functioning in a [High concurrency session](#).

File editor

The file editor allows you to view and edit files directly within the notebook's resource folder and environment resource folder in notebook. Supported file types include **CSV**, **TXT**, **HTML**, **YML**, **PY**, **SQL**, and more. With the file editor, you can easily access and modify files within the notebook, it supports Keyword highlighting and provides necessary language service when opening and editing code files like `.py` and `.sql`.

- You can access this feature through '**View and edit**' in the file menu. Double-click on file is a faster way.



- Content change on file editor needs to be saved manually by clicking the **Save** button or keyboard shortcut: **Ctrl+S**, file editor doesn't support autosave.
- **notebook mode** also affects the file editor. You can only view files but cannot edit them if you are in the notebook mode without editing permission.

ⓘ Note

Here are some limitations for file editor.

- File size limit is **1 MB**.
- These file types are not supported for viewing and editing: **.xlsx** and **.parquet**.

Collaborate in a notebook

The Fabric notebook is a collaborative item that supports multiple users editing the same notebook.

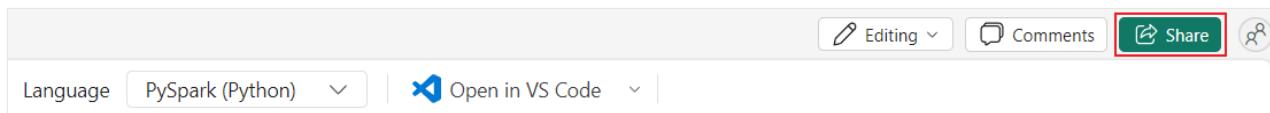
When you open a notebook, you enter the coediting mode by default, and every notebook edit is automatically saved. If your colleagues open the same notebook at the same time, you see their profile, run output, cursor indicator, selection indicator, and editing trace. By using the collaboration features, you can easily accomplish pair programming, remote debugging, and tutoring scenarios.

The screenshot shows a Fabric notebook interface. On the left, there's a sidebar titled 'Lakehouse explorer' showing a tree structure of tables and files under 'Marketing_LH'. The main area has a title 'Project scope' with a description about marketing campaign data. Below it, sections for 'Objectives' and 'Step 1: Use magic command to install dependencies' (containing a code cell for pip install scikit-learn) are visible. A third section, 'Step 2: Load data from Lakehouse', also contains a code cell. At the bottom, there's a section for 'Step 3: Use summary tool to identify the quality issue and clean the data' with another code cell. The top right shows a user profile 'PM' and a message 'On cell 7 of 18'. The bottom right shows '1 of 18 cells'.

Share a notebook

Sharing a notebook is a convenient way for you to collaborate with team members. Authorized workspace roles can view or edit/run notebooks by default. You can share a notebook with specified permissions granted.

1. Select **Share** on the notebook toolbar.



2. Select the corresponding category of **people who can view this notebook**. You can choose **Share**, **Edit**, or **Run** permissions for the recipients.

Select permissions

X

1Test Widgets

People who can view this Notebook

-  People in your organization
-  People with existing access
-  Specific people

Additional permissions

Authorized users can view this Notebook by default.

Select additional permissions.

- Share
- Edit
- Run

 Notebook artifact Sharing

Apply

Back

3. After you select **Apply**, you can either send the notebook directly or copy the link to others. Recipients can then open the notebook with the corresponding view granted by their permission level.

Create and send link



1Test Widgets



People in your organization can
view



Enter a name or email address

Add a message (optional)

Send



Copy link



by Email



by Teams



by PowerPoint

4. To further manage your notebook permissions, select **Workspace item list > More options**, and then select **Manage permissions**. From that screen, you can update the existing notebook access and permissions.

The screenshot shows a list of notebooks on the left and a context menu open over 'Notebook 1' on the right. The menu items are:

- Open
- Delete
- Settings
- Add to Favorites
- View lineage
- View details
- Schedule
- Recent runs
- Manage permissions
- Share

Comment a code cell

Commenting is another useful feature for collaborative scenarios. Currently, Fabric supports adding cell-level comments.

1. Select the **Comments** button on the notebook toolbar or cell comment indicator to open the **Comments** pane.

The screenshot shows the Fabric interface with the 'Comments' pane open. The pane displays a single comment:

z Use the specific Featureization module to select the most relevant features.
April 25, 2023 at 3:11 PM

2. Select code in the code cell, select **New** in the **Comments** pane, add comments, and then select **Post comment** to save.

Step 6: Build machine learning models to fit the data

```

1 import sklearn
2 from sklearn.svm import SVC
3 from sklearn.preprocessing import StandardScaler
4 from sklearn.ensemble import ExtraTreesClassifier
5 from sklearn.linear_model import LogisticRegression
6 from sklearn.tree import DecisionTreeClassifier
7 from sklearn.naive_bayes import BernoulliNB
8 from sklearn.neighbors import KNeighborsClassifier
9 from sklearn.model_selection import GridSearchCV
10

```

Comments

New

Use the specific Featureization module to select the most relevant features.
April 25, 2023 at 3:11 PM
Reply...

3. If you need them, find the **Edit comment**, **Resolve thread**, and **Delete thread** options by selecting the More option next to your comment.

Tagging others in a comment

"Tagging" refers to mentioning and notifying a user in a comment thread, enhancing collaboration efficiently on the specifics.

1. Select a section of code in a cell and new a comment thread.
2. Input user name and choose the correct one on the suggestion list if you want to mention someone for discussion about a certain section.
3. Share your insights and **Post** them.
4. An Email notification is triggered, and user clicks on **Open Comments** link to quickly locate this cell.
5. Moreover, authorize and configure the permissions for users when tagging someone who doesn't have access, ensuring that your code assets are well managed.

Connect ▾ | PySpark (Python) ▾ | Workspace default ▾ | Data Wrangler ▾ | Copilot

Now, plot the uplift curve for the test dataset prediction. Note that you must convert the PySpark DataFrame to a Pandas DataFrame before plotting.

```

1 def uplift_plot(uplift_df):
2     """
3         Plot the uplift curve
4     """
5     gain_x = uplift_df.percent_rank
6     gain_y = uplift_df.group_uplift
7     # Plot the data
8     fig = plt.figure(figsize=(10, 6))
9     mpl.rcParams["font.size"] = 8
10
11    ax = plt.plot(gain_x, gain_y, color="#2077B4", label="Normalized Uplift Model")
12
13    plt.plot(
14        [0, gain_x.max()],
15        [0, gain_y.max()],
16        "-",
17        color="tab:orange",
18        label="Random Treatment",
19    )
20
21    plt.legend()
22    plt.xlabel("Proportion Targeted")
23    plt.ylabel("Uplift")
24    plt.grid()
25
26    return fig, ax
27
28 test_ranked_pd_df = test_ranked_df.select(["pred_uplift", "percent_rank", "group_uplift"]).toPandas()
29 fig, ax = uplift_plot(test_ranked_pd_df)
30
31 mlflow.log_figure(fig, "UpliftCurve.png")
32

```

Comments

New

There are no comments in this notebook

ⓘ Note

For a comment item, the tagged user will not receive an Email notification anymore if you update the comment within one hour. But it sends Email notification to the new tagged user.

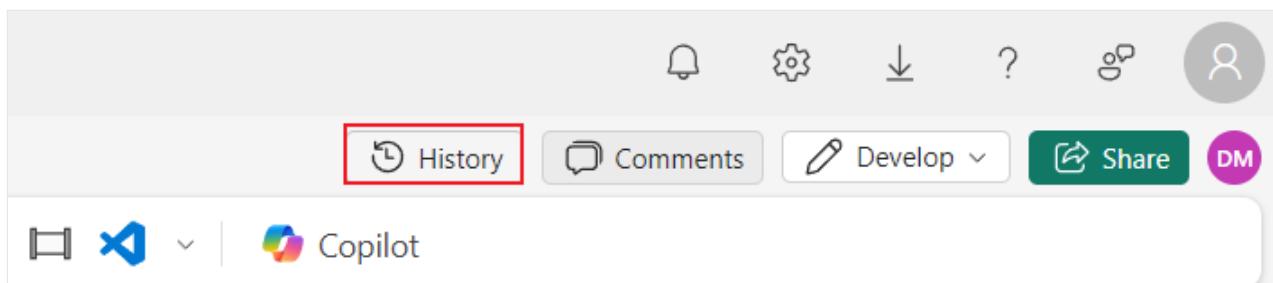
Version history

ⓘ Important

This feature is in [preview](#).

Version history allows you to easily version your live notebook changes. It supports history tracking and notebook management with robust built-in version control capabilities, which is especially helpful for collaborative work with multiple contributors on the same notebook.

1. Access version **history** from notebook global toolbar.



2. Notebook supports two ways of creating checkpoints.

- Manual checkpoint: You can manually **create new version** to record your development milestones, providing flexibility for managing your notebook versions.

Sample_notebook | Saved Fabric Trial: 11 days left

[Back to notebook](#) [History](#)

A Current 10/22/2024, 1:37:4PM

```

1 df = spark.read.parquet("Files/green_tripdata_2022-08.parquet")
2 # df now is a Spark DataFrame containing parquet data from "Files/green_tripdata_2022-08.parquet".
3 display(df)

1 # Specify the name of the Delta table
2 table_name = 'green_tripdata_2022_08'
3
4 # Write the Dataframe to a Delta Lake table, using the specified name and path
5 df.write.format('delta').mode('overwrite').saveAsTable(table_name)

1 df = spark.sql("SELECT * FROM green_tripdata_2022_08 LIMIT 1000")
2
3 display(df)

1 df.printSchema()

1 df.show(5)

```

Objective: Cleanse the data and filter out invalid records for further analysis.

```

1 from pyspark.sql.functions import col, when
2
3 # Load data from source

```

Version history

+ Version

Name * Name Description 0/512 maximum characters

New dev member

Oct 21 (2) Edited by dev member October 21, 2024 at 3:56 PM dev member October 21, 2024 at 3:02 PM dev member

Exploratory data analysis In data engineering, EDA is often done to identify data quality issues, anomalies, or other problems that need to be addressed before data can be used for analysis or modeling. October 21, 2024 at 2:59 PM Created by dev contributor

Data cleaning and transformation October 21, 2024 at 2:56 PM Created by dev member

Load data into a Spark DF October 21, 2024 at 2:53 PM Created by dev member

- System checkpoint: These checkpoints are created automatically every 5 minutes based on editing time interval by Notebook system, ensuring that your work is consistently saved and versioned. You can find the modification records from all the contributors in the system checkpoint timeline list.

[Back to notebook](#) [History](#)

B Current 11/25/2024, 10:52:55AM

Creating, Evaluating, and Training a Churn Prediction Model

Introduction

In this notebook, we'll demonstrate Microsoft Fabric data science workflow with an end-to-end example. The scenario is to build a model to predict whether bank customers would churn or not. The churn rate, also known as the rate of attrition refers to the rate at which bank customers stop doing business with the bank.

The summary of main steps we take in this notebook are as following:

1. Load the data from lakehouse
2. Understanding and exploring data
3. Cleaning the data and verify
4. Train machine learning models using Scikit-Learn

Get started with Live pool!

Step 1: Load and Process the Data

The dataset contains churn status of 10000 customers along with 14 attributes that include credit score, geographical location (Germany, France, Spain), gender (male, female), age, tenure (years of being bank's customer), account balance, estimated salary, number of products that a customer has purchased through the bank, credit card status (whether a customer has a credit card or not), and active member status (whether an active bank's customer or not).

Out of the 10000 customers, only 2037 customers (around 20%) have left the bank. Therefore, given the class imbalance ratio, we recommend to generate synthetic data. Moreover, confusion matrix accuracy may not be meaningful for imbalanced classification and it might be better to also measure the accuracy using the Area Under the Precision-Recall Curve (AUPRC).

- * churn.csv

~CustomerID~	~Surname~	~CreditScore~	~Geography~	~Gender~	~Age~	~Tenure~	~Balance~	~NumOfProducts~	~HasCrCard~	~IsActiveMember~	~EstimatedSalary~
--------------	-----------	---------------	-------------	----------	-------	----------	-----------	-----------------	-------------	------------------	-------------------

Version history

> Nov 1, 5:34 PM (2) Edited by Jene

> Oct 31, 4:13 PM (5) Edited by Jene

> Oct 30, 6:55 PM (5) Edited by Jene Fei

> Oct 29, 3:47 PM (1) Edited by Jene Lijing

Train model and validate Add final step to validate the final result October 29, 2024 at 3:43 PM Created by Lijing

> Oct 29, 3:42 PM (5) Edited by Jene Lijing Fei

October 29, 2024 at 3:42 PM Jene

October 29, 2024 at 3:37 PM Lijing Jene Fei

October 29, 2024 at 3:27 PM Jene

October 29, 2024 at 3:12 PM Jene

October 29, 2024 at 3:07 PM Lijing Jene

Data exploration v1 Finish data loading and cleaning

3. You can click on a checkpoint to open the diff view, it highlights the content differences between the selected checkpoint and the current live version, including the differences of cell content, cell output, and metadata. The version of this checkpoint can be managed individually in 'more options' menu.

The screenshot shows a Jupyter Notebook interface with two code cells and a sidebar.

Code Cell 1:

```

4
5
6 Out of the 10000 customers, only 2037 customers (around 20%) have left the bank. Th
7
8 - churn.csv
9
10 ["CustomerID","Surname","CreditScore","Geography","Gender","Age","Tenure","Balance"
11 |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
12 [11111111]Hargrave[619|France|Female|42|2|0.00|1|1|101348.88|1]
13 [22222222]Hill[608|Spain|Female|41|1|83807.86|1|0|1|12542.58|0]
14
15
16

```

Code Cell 2:

```

1 #### Read raw data from the lakehouse
2
3 **Reads raw data from the _Files_ section of the lakehouse, adds additional columns
4
5 We can explore the raw data with `display`, do some basic statistics or even show c

```

Code Cell 3:

```

1 import pandas as pd
2
3 raw_data_path = "/lakehouse/default/Files/bank-additional-full.csv"
4 df = pd.read_csv(raw_data_path)
5+display(df)

```

Code Cell 4:

```

1 import pandas as pd
2
3 raw_data_path = "/lakehouse/default/Files/bank-additional-full.csv"
4 df = pd.read_csv(raw_data_path)
5+display(df)

```

Version History:

- Current (November 25, 2024 at 10:52 AM)
- Nov 20, 4:07 PM (1) Edited by Jen
- Nov 8, 9:39 AM (1) Edited by Kwangle
- Nov 7, 8:22 AM (2) Edited by Lijing Kwangle
- Nov 5, 8:30 AM (1) Edited by Fei
- Nov 1, 5:34 PM (2) Edited by Jen
- Oct 31, 4:13 PM (5) Edited by Jen
- Oct 30, 6:55 PM (5) Edited by Jen Fei
- Oct 29, 3:47 PM (1) Edited by Jen Lijing
- Train model and validate (Add final step to validate the final result October 29, 2024 at 3:43 PM Created by Lijing)
- Data exploration v1 (Finish data loading and cleaning October 29, 2024 at 2:59 PM)

4. You can manage the version from the checkpoint drop-down menu, if you want to keep a previous version, click **restore** from checkpoint and overwrite the current notebook, or using **save as copy** to clone it to a new notebook.

The screenshot shows a Jupyter Notebook interface with two code cells and a sidebar.

Code Cell 1:

```

import matplotlib.pyplot as plt
# assuming `df` is your Spark DataFrame containing the columns `fare_amount` and `trip_distance`
df_pd = df.select(['fare_amount', 'trip_distance']).toPandas()

# create scatter plot
fig, ax = plt.subplots()
ax.scatter(x=df_pd['trip_distance'], y=df_pd['fare_amount'], alpha=0.5)

# set axis labels and title
ax.set_xlabel('Trip Distance')
ax.set_ylabel('Fare Amount')
ax.set_title('Correlation between Fare Amount and Trip Distance')

# show the plot
plt.show()

```

Code Cell 2:

```

1 df = spark.read.parquet("Files/green_tripdata_2022-08.parquet")
2 # df now is a Spark DataFrame containing parquet data from "Files/green_tripdata_2022-08.parquet"
3 display(df)

```

Code Cell 3:

```

1 # Specify the name of the Delta table
2 table_name = 'green_tripdata_2022_08'
3

```

Code Cell 4:

```

1 # Specify the name of the Delta table
2 table_name = 'green_tripdata_2022_08'
3

```

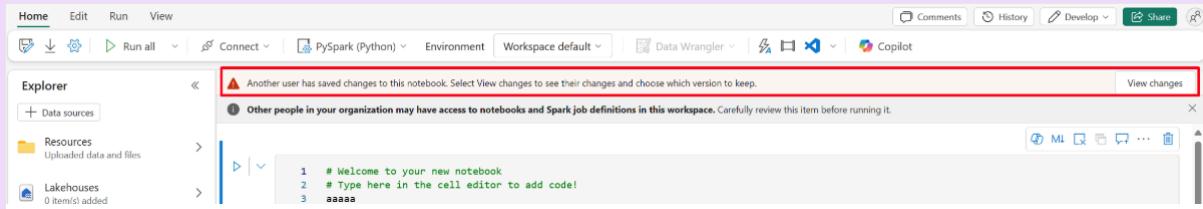
Version History:

- Current (October 22, 2024 at 11:29 AM)
- Oct 21 (2)
 - Edited by dev member (October 21, 2024 at 3:56 PM)
 - Created by dev contrib (October 21, 2024 at 3:02 PM)
- Exploratory data analysis (In data engineering, EDA is often used to identify data quality issues, anomalies, and problems that need to be addressed. It can be used for analysis or monitoring. October 21, 2024 at 2:59 PM)
 - Restore
 - Edit details
 - Save as copy
 - Delete
- Data cleaning and transformation (October 21, 2024 at 2:56 PM)
 - Created by dev member
- Load data into a Spark DF (October 21, 2024 at 2:53 PM)
 - Created by dev member
- Oct 21 (1)
 - Edited by dev member

! Note

- Known limitation: After clicking on the **Restore** button and navigate **Back to notebook**, the notebook won't be immediately recovered from the checkpoint. A message bar prompts you to view the changes. You need to click the **View changes**

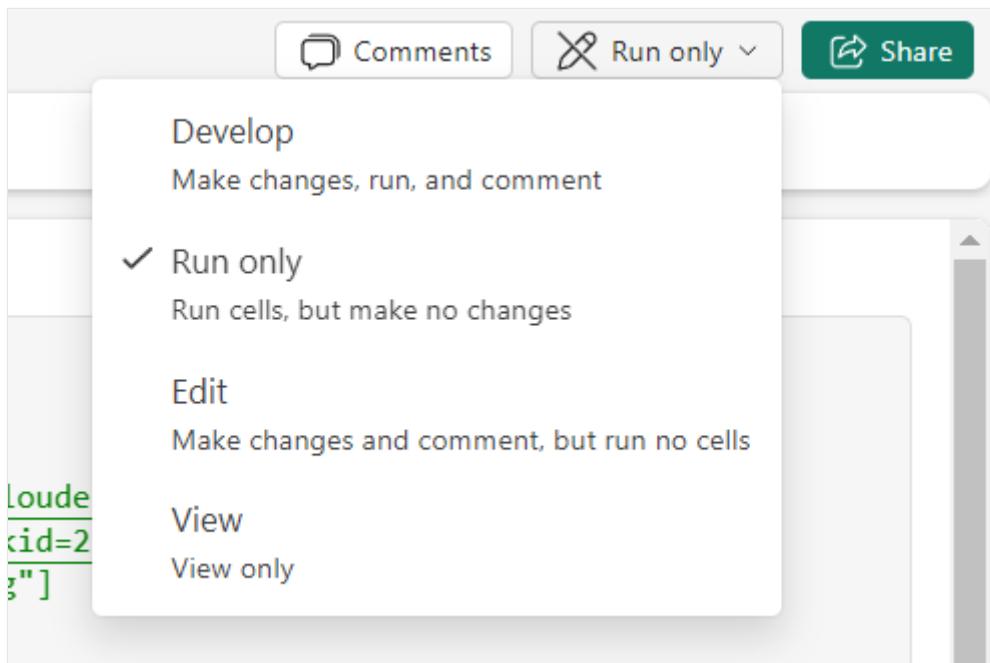
button and select keeping the *Saved version* to finish the restore.



- System checkpoints will expire after 1 year.

Notebook mode switcher

Fabric notebooks support four modes that you can easily switch: **Develop** mode, **Run only** mode, **Edit** mode, and **View** mode. Each mode maps to a specific permission combination. When sharing the notebook to other team members, you can grant proper permissions to the recipients. They can see the best available notebook mode according to their permission, and they are able to switch between the mode they have permission to.



- **Develop mode**: Read, execute, write permission needed.
- **Run only mode**: Read, execute permission needed.
- **Edit mode**: Read, write permission needed.
- **View mode**: Read permission needed.

Related content

- [Author and execute notebooks](#)

Develop, execute, and manage Microsoft Fabric notebooks

Article • 03/31/2025

A Microsoft Fabric notebook is a primary code item for developing Apache Spark jobs and machine learning experiments. It's a web-based interactive surface used by data scientists and data engineers to write code benefiting from rich visualizations and Markdown text. This article explains how to develop notebooks with code cell operations and run them.

Develop notebooks

Notebooks consist of cells, which are individual blocks of code or text that can be run independently or as a group.

We provide rich operations to develop notebooks:

- [Add a cell](#)
- [Set a primary language](#)
- [Use multiple languages](#)
- [IDE-style IntelliSense](#)
- [Code snippets](#)
- [Drag and drop to insert snippets](#)
- [Drag and drop to insert images](#)
- [Format text cell with toolbar buttons](#)
- [Undo or redo cell operation](#)
- [Move a cell](#)
- [Delete a cell](#)
- [Collapse a cell input](#)
- [Collapse a cell output](#)
- [Cell output security](#)
- [Lock or freeze a cell](#)
- [Notebook contents](#)
- [Markdown folding](#)
- [Find and replace](#)

Add a cell

There are multiple ways to add a new cell to your notebook.

1. Hover over the space between two cells and select **Code** or **Markdown**.

2. Use [Shortcut keys in command mode](#). Press **A** to insert a cell above the current cell.

Press **B** to insert a cell below the current cell.

Set a primary language

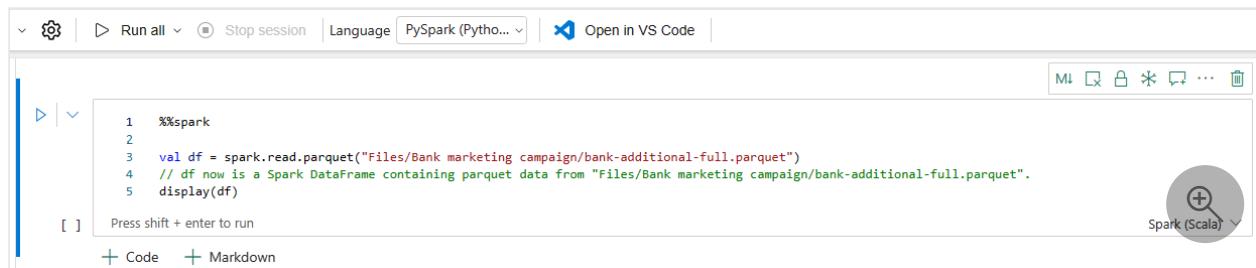
Fabric notebooks currently support four Apache Spark languages:

- PySpark (Python)
- Spark (Scala)
- Spark SQL
- SparkR

You can set the primary language for new added cells from the drop-down list in the top command bar.

Use multiple languages

You can use multiple languages in a notebook by specifying the language magic command at the beginning of a cell. You can also switch the cell language from the language picker. The following table lists the magic commands for switching cell languages.



The screenshot shows a Fabric notebook interface. A code cell contains the following Scala code:

```
1 %%spark
2
3 val df = spark.read.parquet("Files/Bank marketing campaign/bank-additional-full.parquet")
4 // df now is a Spark DataFrame containing parquet data from "Files/Bank marketing campaign/bank-additional-full.parquet".
5 display(df)
```

The cell has a status bar indicating "[]" and "Press shift + enter to run". To the right of the cell is a language picker button labeled "Spark (Scala)".

[Expand table](#)

Magic command	Language	Description
%%pyspark	Python	Execute a Python query against Apache Spark Context.
%%spark	Scala	Execute a Scala query against Apache Spark Context.
%%sql	SparkSQL	Execute a SparkSQL query against Apache Spark Context.
%%html	Html	Execute a HTML query against Apache Spark Context.
%%sparkr	R	Execute a R query against Apache Spark Context.

IDE-style IntelliSense

Fabric notebooks are integrated with the Monaco editor to bring IDE-style IntelliSense to the cell editor. Syntax highlight, error marker, and automatic code completions help you to quickly write code and identify issues.

The IntelliSense features are at different levels of maturity for different languages. The following table shows what Fabric supports:

[Expand table]

Languages	Syntax highlight	Syntax error marker	Syntax code completion	Variable code completion	System function code completion	User function code completion	Smart indent	Code folding
PySpark (Python)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Python	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Spark (Scala)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
SparkSQL	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
SparkR	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
T-SQL	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes

Note

You must have an active Apache Spark session to use IntelliSense code completion.

Enhance Python Development with Pylance

Note

Currently, the feature is in preview.

Pylance, a powerful and feature-rich language server, is now available in Fabric notebook. Pylance makes Python development easier with smart completions, better error detection, and improved code insights. Key improvements include smarter auto-completion, enhanced lambda support, parameter suggestions, improved hover information, better docstring rendering, and error highlighting. With Pylance, writing Python and PySpark code becomes faster, more accurate, and more efficient.

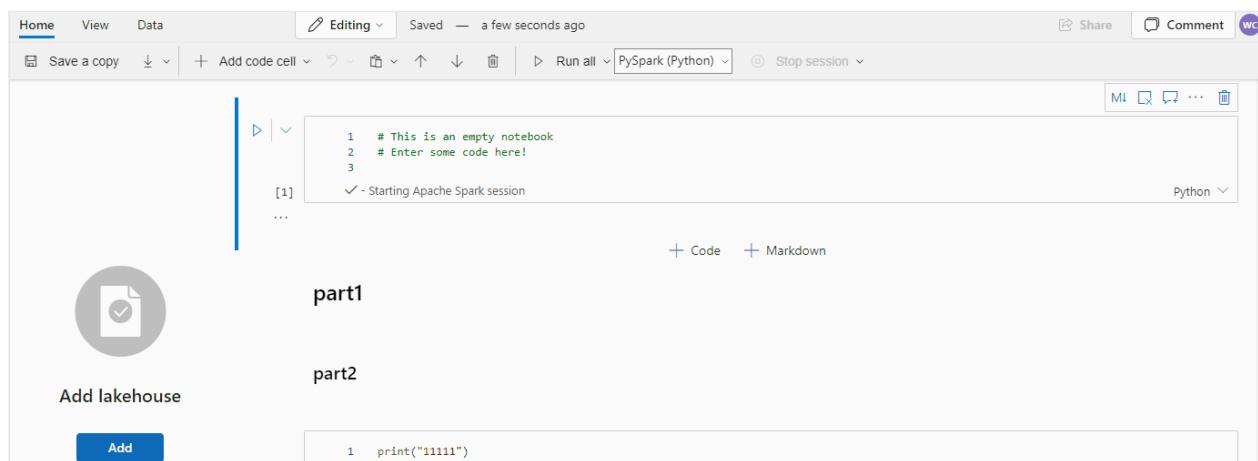
Code snippets

Fabric notebooks provide code snippets that help you easily write commonly used code patterns, like:

- Reading data as an Apache Spark DataFrame
- Drawing charts with Matplotlib

Snippets appear in **Shortcut keys of IDE style IntelliSense** mixed with other suggestions.

The code snippet contents align with the code cell language. You can see available snippets by typing **Snippet**. You can also type any keyword to see a list of relevant snippets. For example, if you type **read**, you see the list of snippets to read data from various data sources.



Drag and drop to insert snippets

Use drag and drop to read data from Lakehouse explorer conveniently. Multiple file types are supported here; you can operate on text files, tables, images, etc. You can either drop to an existing cell or to a new cell. The notebook generates the code snippet accordingly to preview the data.

A screenshot of a Fabric notebook interface. On the left, there's a sidebar titled "TestLH" with "Lake view" selected. It shows a "Tables" section with "Tables" and "Files" options, and a "Files" section listing "aisles.csv", "order_products__train.csv", "orders.csv", "products.csv", and "test". The main workspace has a title "Predict NYC Taxi Tips using Spark ML and Azure Open Datasets" with a subtitle "The notebook ingests, visualizes, prepares and then trains a model based on an Open Dataset that tracks NYC Yellow Taxi trips and v for a given trip whether there will be a tip or not." Below the title, there's a section titled "Test Drag & Drop" with three code cells. The first cell contains "print("code cell #1")" and a note "Press shift + enter to run". The second cell contains "print("code cell #2")" and a note "Press shift + enter to run". The third cell contains "import matplotlib.pyplot as plt" and "from pyspark.sql.functions import unix_timestamp". There are "Code" and "Markdown" buttons at the bottom of the workspace.

Drag and drop to insert images

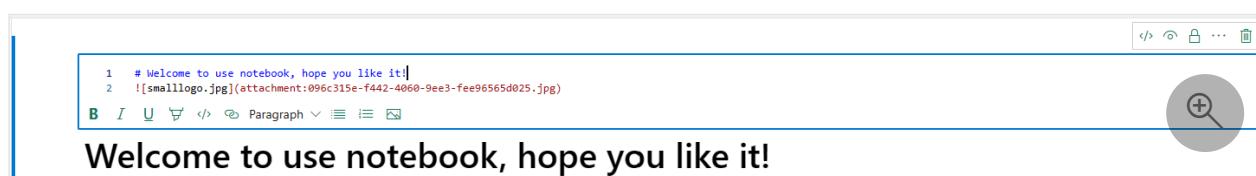
Use drag and drop to easily insert images from your browser or local computer to a markdown cell.

The screenshot shows a Jupyter Notebook interface. At the top, there are two tabs: 'Code' and 'Markdown'. Below the tabs, a message says 'Empty Markdown cell. Double click or press enter to add content.' In the main area, there is a code cell containing the following Python code:

```
1 import random
2 import datetime
3 import pathlib
4 import os
5 import numpy as np
6 import pandas as pd
7
8 # less than 50M
9 # [0, 5], add 0.1% float/string, 10% missing value
10 # date, 1 months, 0.1 incorrect date
11 # UID, report
12 # PID, product id
13
14
15 def convert(line, month, day):
```

Format text cell with toolbar buttons

To complete common markdown actions, use the format buttons in the text cell toolbar.



Undo or redo cell operations

Select **Undo** or **Redo**, or press **Z** or **Shift+Z** to revoke the most recent cell operations. You can undo or redo up to 10 of the latest historical cell operations.



Supported undo cell operations:

- Insert or delete cell. You can revoke the deleted operations by selecting **Undo** (the text content is kept along with the cell).
- Reorder cell.
- Toggle parameter.
- Convert between code cell and Markdown cell.

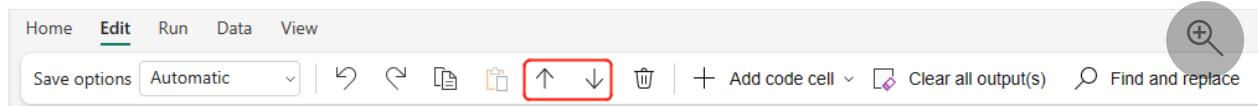
ⓘ Note

In-cell text operations and code cell commenting operations can't be undone. You can undo or redo up to 10 of the latest historical cell operations.

Move a cell

You can drag from the empty part of a cell and drop it to the desired position.

You can also move the selected cell using **Move up** and **Move down** on the ribbon.



Delete a cell

To delete a cell, select the delete button at the right side of the cell.

You can also use [shortcut keys in command mode](#). Press **Shift+D** to delete the current cell.

Collapse a cell input

Select the **More commands** ellipses (...) on the cell toolbar and **Hide input** to collapse the current cell's input. To expand it again, select **Show input** when the cell is collapsed.

Collapse a cell output

Select the **More commands** ellipses (...) on the cell toolbar and **Hide output** to collapse the current cell's output. To expand it again, select **Show output** when the cell output is collapsed.

Cell output security

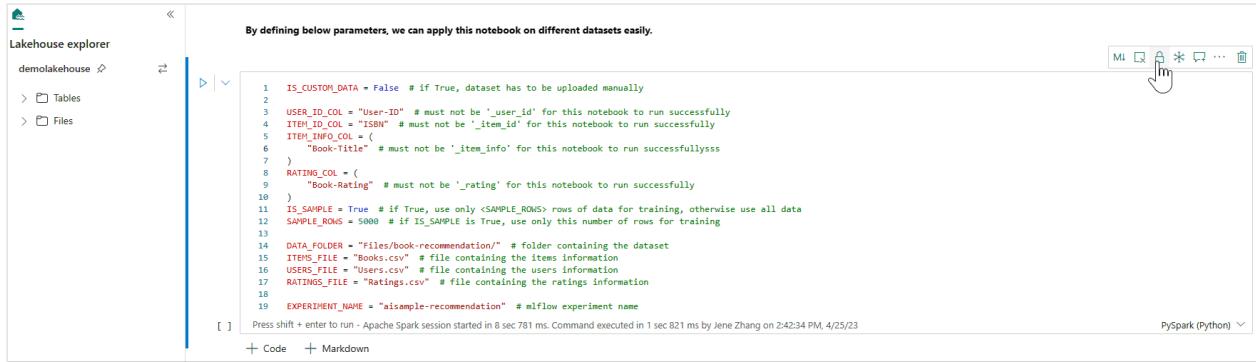
Using [OneLake data access roles \(preview\)](#), users can configure access to only specific folders in a lakehouse during notebook queries. Users without access to a folder or table see an unauthorized error during query execution.

i Important

Security only applies during query execution and any notebook cells containing query results can be viewed by users that are not authorized to run queries against the data directly.

Lock or freeze a cell

The lock and freeze cell operations allow you to make cells read-only or stop code cells from being run on an individual basis.

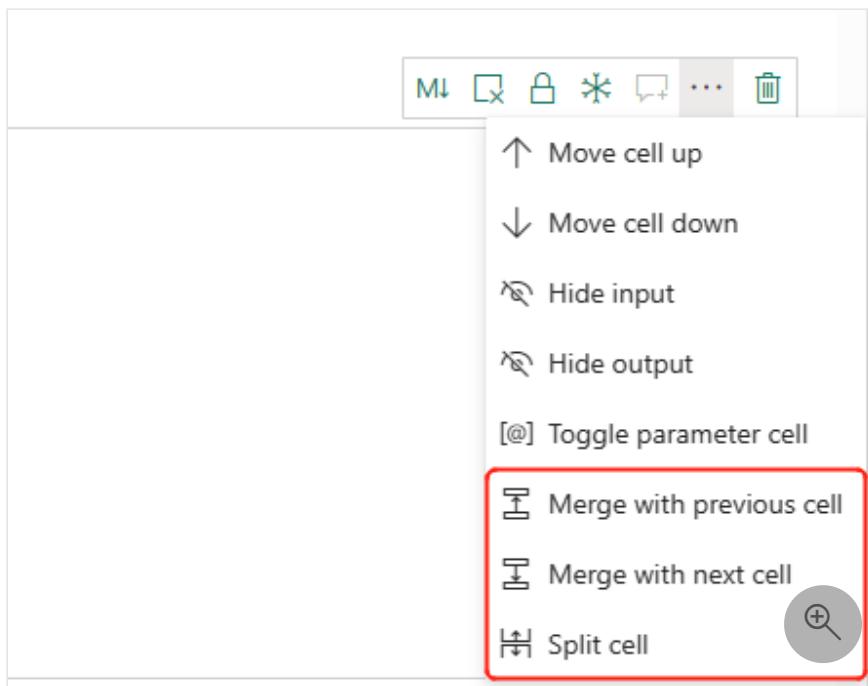


A screenshot of a Jupyter Notebook interface. On the left, there's a sidebar titled "Lakehouse explorer" with sections for "Tables" and "Files". The main area contains a code cell with Python code related to a recommendation system. The code includes variables like IS_CUSTOM_DATA, USER_ID_COL, ITEM_ID_COL, ITEM_INFO_COL, RATING_COL, IS_SAMPLE, SAMPLE_ROWS, DATA_FOLDER, ITEMS_FILE, USERS_FILE, and RATINGS_FILE. A status bar at the bottom indicates "Press shift + enter to run - Apache Spark session started in 8 sec 781 ms. Command executed in 1 sec 821 ms by Jene Zhang on 2:42:34 PM, 4/25/23". The command bar at the top right includes icons for "M", "Lock", "Star", "Copy", "Run", and "More".

Merge and split cells

You can use **Merge with previous cell** or **Merge with next cell** to merge related cells conveniently.

Selecting **Split cell** helps you split irrelevant statements to multiple cells. The operation splits the code according to your cursor's line position.



Notebook contents

Selecting **Outlines** or **Table of Contents** presents the first markdown header of any markdown cell in a sidebar window for quick navigation. The Outlines sidebar is resizable and collapsible to fit the screen in the best way possible. Select the **Contents** button on the notebook command bar to open or hide the sidebar.

Markdown folding

The markdown folding option allows you to hide cells under a markdown cell that contains a heading. The markdown cell and its hidden cells are treated the same as a set of contiguous multi-selected cells when performing cell operations.

Find and replace

The find and replace option can help you match and locate the keywords or expression within your notebook content. You can also easily replace the target string with a new string.

The screenshot shows the Lakehouse Explorer interface. On the left, there's a sidebar with a 'Find' section containing a search bar with 'sample' typed in, and a 'Replace' section with a 'Replace with...' input field. Below these are buttons for 'Replace' and 'Replace all'. A message '5 results' is displayed. The main area contains three code cells:

```
1 # Use the 2 magic commands below to reload the modules if your module has updates during the cu
2 # %load_ext autoreload
```

```
1 # %autoreload 2
2
3 import builtin.Code.SampleModule as SampleModule
4 # Now use the exported members from this module with identifier: `SampleModule`
5 dir(SampleModule)
6
```

```
1 df = spark.read.parquet("Files/SampleData.parquet")
2
3 display(df)
```

Run notebooks

You can run the code cells in your notebook individually or all at once. The status and progress of each cell is displayed in the notebook.

Run a cell

There are several ways to run the code in a cell.

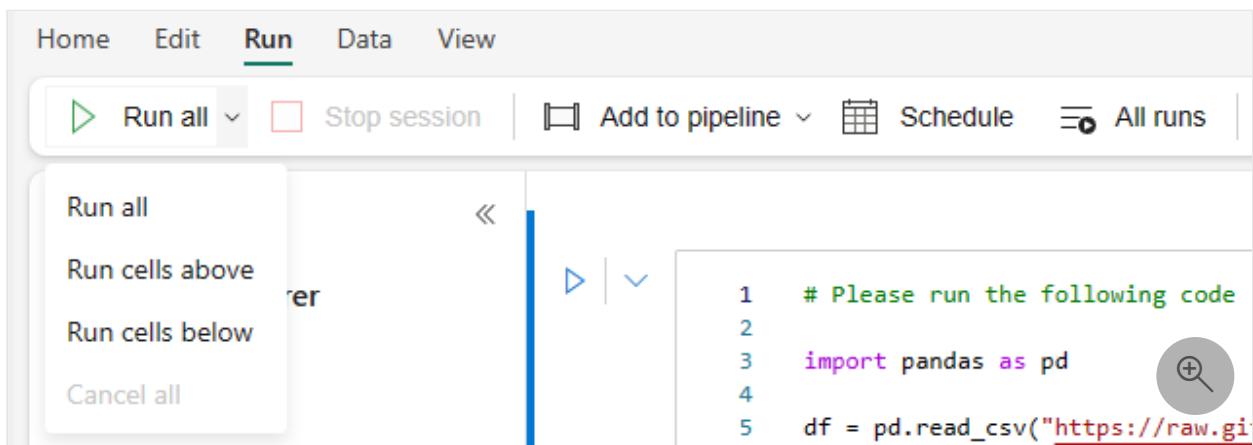
- Hover on the cell you want to run and select the **Run cell** button or press **Ctrl+Enter**.
- Use [Shortcut keys in command mode](#). Press **Shift+Enter** to run the current cell and select the next cell. Press **Alt+Enter** to run the current cell and insert a new cell.

Run all cells

Select the **Run all** button to run all the cells in the current notebook in sequence.

Run all cells above or below

Expand the drop-down list from **Run all**, then select **Run cells above** to run all the cells above the current in sequence. Select **Run cells below** to run the current cell and all the cells below the current in sequence.

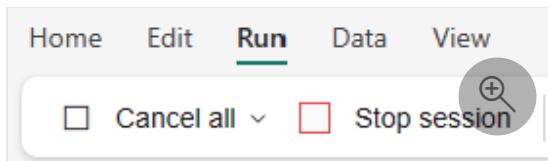


Cancel all running cells

Select **Cancel all** to cancel the running cells or cells waiting in the queue.

Stop session

Stop session cancels the running and waiting cells and stops the current session. You can restart a brand new session by selecting the run option again.



Reference run

Reference run a Notebook

In addition to [notebookutils reference run API](#), you can also use the `%run <notebook name>` magic command to reference another notebook within current notebook's context. All the variables defined in the reference notebook are available in the current notebook. The `%run` magic command supports nested calls but doesn't support recursive calls. You receive an exception if the statement depth is larger than `five`.

Example: `%run Notebook1 { "parameterInt": 1, "parameterFloat": 2.5, "parameterBool": true, "parameterString": "abc" }.`

Notebook reference works in both interactive mode and pipeline.

(!) Note

- The `%run` command currently only supports reference notebooks in the same workspace with the current notebook.
- The `%run` command currently only supports up to four parameter value types: `int`, `float`, `bool`, and `string`. Variable replacement operation is not supported.
- The `%run` command doesn't support nested reference with a depth larger than five.

Reference run a script

The `%run` command also allows you to run Python or SQL files that are stored in the notebook's built-in resources, so you can execute your source code files in notebook conveniently.

```
%run [-b/--builtin -c/--current] [script_file.py/.sql] [variables ...]
```

For options:

- **-b/--builtin:** This option indicates that the command finds and runs the specified script file from the notebook's built-in resources.
- **-c/--current:** This option ensures that the command always uses the current notebook's built-in resources, even if the current notebook is referenced by other notebooks.

Examples:

- To run `script_file.py` from the built-in resources: `%run -b script_file.py`
- To run `script_file.sql` from the built-in resources: `%run -b script_file.sql`
- To run `script_file.py` from the built-in resources with specific variables: `%run -b script_file.py { "parameterInt": 1, "parameterFloat": 2.5, "parameterBool": true, "parameterString": "abc" }`

ⓘ Note

If the command does not contain `-b/--builtin`, it will attempt to find and execute notebook item inside the same workspace rather than the built-in resources.

Usage example for nested run case:

- Suppose we have two notebooks.
 - **Notebook1:** Contains `script_file1.py` in its built-in resources

- **Notebook2**: Contains `script_file2.py` in its built-in resources
- Let's use **Notebook1** work as a root notebook with the content: `%run Notebook2`.
- Then in the **Notebook2** the usage instruction is:
 - To run `script_file1.py` in **Notebook1**(the root Notebook), the code would be: `%run -b script_file1.py`
 - To run `script_file2.py` in **Notebook2**(the current Notebook), the code would be: `%run -b -c script_file2.py`

Variable explorer

Fabric notebooks provide a built-in variables explorer that displays the list of the variables name, type, length, and value in the current Spark session for PySpark (Python) cells. More variables show up automatically as they're defined in the code cells. Clicking on each column header sorts the variables in the table.

To open or hide the variable explorer, select **Variables** on the notebook ribbon **View**.

Name	Type	Length	Value
a	int	10	
displayHTML	DisplayHTML		
HiveContext	type		
sc	SparkContext		
spark	SparkSession		
sqlContext	SQLContext		
StreamingContext	type		
VarA	str	14	'hello notebook'
VarB	int		128

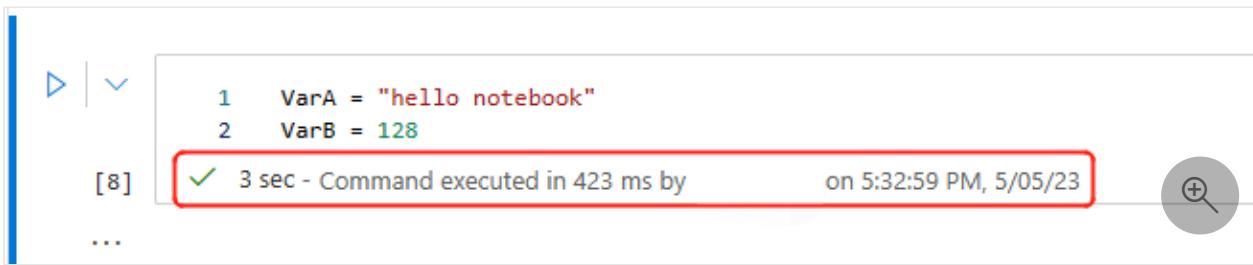
! Note

The variable explorer only supports Python.

Cell status indicator

A step-by-step cell execution status is displayed beneath the cell to help you see its current progress. Once the cell run is complete, an execution summary with the total duration and

end time appears and is stored there for future reference.



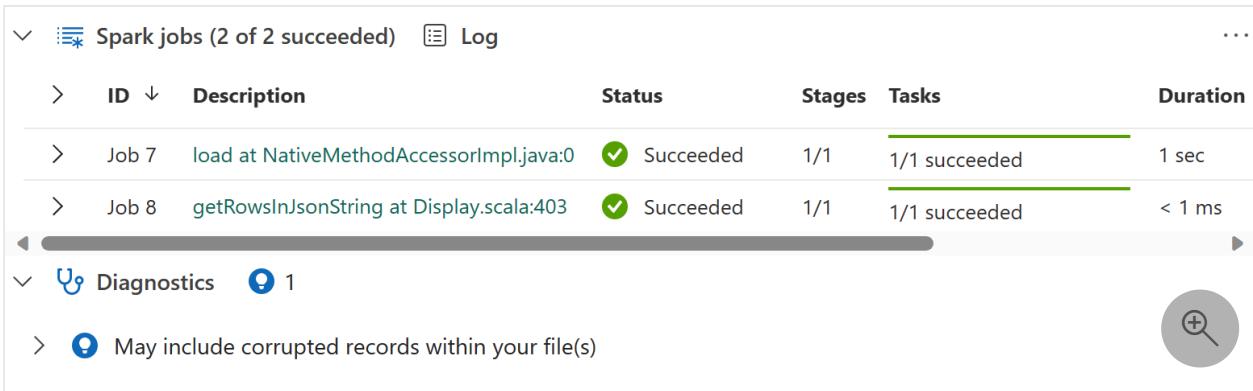
```
1 VarA = "hello notebook"
2 VarB = 128
```

[8] ✓ 3 sec - Command executed in 423 ms by on 5:32:59 PM, 5/05/23

Inline Apache Spark job indicator

The Fabric notebook is Apache Spark based. Code cells are executed on the Apache Spark cluster remotely. A Spark job progress indicator is provided with a real-time progress bar that appears to help you understand the job execution status. The number of tasks per each job or stage helps you to identify the parallel level of your Spark job. You can also drill deeper to the Spark UI of a specific job (or stage) via selecting the link on the job (or stage) name.

You can also find the **Cell level real-time log** next to the progress indicator, and **Diagnostics** can provide you with useful suggestions to help refine and debug the code.

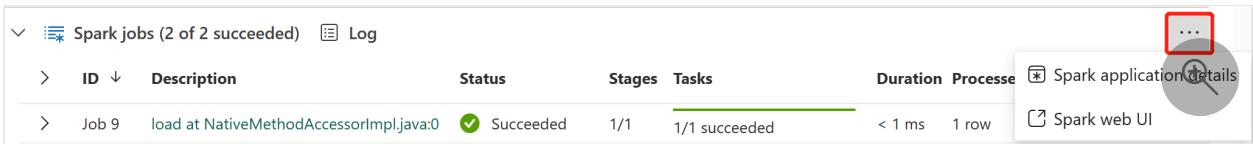


ID	Description	Status	Stages	Tasks	Duration
Job 7	load at NativeMethodAccessorImpl.java:0	Succeeded	1/1	1/1 succeeded	1 sec
Job 8	getRowsInJsonString at Display.scala:403	Succeeded	1/1	1/1 succeeded	< 1 ms

✓ Diagnostics

May include corrupted records within your file(s)

In **More actions**, you can easily navigate to the **Spark application details** page and **Spark web UI** page.



Secret redaction

To prevent credentials being accidentally leaked when running notebooks, Fabric notebooks support **Secret redaction** to replace the secret values that are displayed in cell output with **[REDACTED]**. Secret redaction is applicable for **Python**, **Scala**, and **R**.

User-oriented interface to get secret.

```
[6]  1 %%pyspark  
  2 mssparkutils.credentials.getSecret("https://kvtest.vault.azure.net/", "secret name")  
✓ - Command executed in 404 ms by [REDACTED] on 4:21:07 PM, 4/25/23
```

PySpark (Python) ▾

User-oriented interface to get token.

```
[7]  1 %%pyspark  
  2 print(mssparkutils.credentials.getToken("https://kusto.kusto.windows.net"))  
  3 print(mssparkutils.credentials.getToken("pbi"))  
✓ - Command executed in 2 sec 612 ms by [REDACTED] on 4:21:21 PM, 4/25/23
```

PySpark (Python) ▾

Magic commands in a notebook

Built-in magic commands

You can use familiar Ipython magic commands in Fabric notebooks. Review the following list of currently available magic commands.

⚠ Note

These are the only magic commands supported in Fabric pipeline: %%pyspark, %%spark, %%csharp, %%sql, %%configure.

Available line magic commands: [%%lsmagic](#), [%%time](#), [%%timeit](#), [%%history](#), [%%run](#), [%%load](#), [%%alias](#), [%%alias_magic](#), [%%autoawait](#), [%%autocall](#), [%%automagic](#), [%%bookmark](#), [%%cd](#), [%%colors](#), [%%dhist](#), [%%dirs](#), [%%doctest_mode](#), [%%killbgscripts](#), [%%load_ext](#), [%%logoff](#), [%%logon](#), [%%logstart](#), [%%logstate](#), [%%logstop](#), [%%magic](#), [%%matplotlib](#), [%%page](#), [%%pastebin](#), [%%pdef](#), [%%pfile](#), [%%pinfo](#), [%%pinfo2](#), [%%popd](#), [%% pprint](#), [%%precision](#), [%%prun](#), [%%psearch](#), [%%psource](#), [%%pushd](#), [%%pwd](#), [%%pycat](#), [%%quickref](#), [%%rehashx](#), [%%reload_ext](#), [%%reset](#), [%%reset_selective](#), [%%sx](#), [%%system](#), [%%tb](#), [%%unalias](#), [%%unload_ext](#), [%%who](#), [%%who_ls](#), [%%whos](#), [%%xdel](#), [%%xmode](#).

Fabric notebook also supports the improved library management commands [%%pip](#) and [%%conda](#). For more information about usage, see [Manage Apache Spark libraries in Microsoft Fabric](#).

Available cell magic commands: [%%time](#), [%%timeit](#), [%%capture](#), [%%writefile](#), [%%sql](#), [%%pyspark](#), [%%spark](#), [%%csharp](#), [%%configure](#), [%%html](#), [%%bash](#), [%%markdown](#), [%%perl](#), [%%script](#), [%%sh](#).

Custom magic commands

You can also build out more custom magic commands to meet your specific needs. Here's an example:

1. Create a notebook with name "MyLakehouseModule".

Register My magic command with IPython in Notebook: MyLakehouseModule

```
1 @magics_class
2 class MyLakeHouseMagic(Magics):
3     @line_magic
4     def list_lh(self):
5         "list all my lakehouse account"
6         # do operations
7         return ['LH1', 'LH2']
8
9     @line_magic
10    def create_lh(self, name):
11        "create lakehouse with name"
12        # do operations
13        return "create Lakehouse with name: " + name
14
15
16 Press shift + enter to run
```



2. In another notebook, reference the "MyLakehouseModule" and its magic commands. This process is how you can conveniently organize your project with notebooks that use different languages.

```
1 %run MyLakehouseModule
Press shift + enter to run
```

Use it everywhere, in every language

```
1 %create_lh "mylakehouse"
✓ 20 sec - Command executed in 9 sec 973 ms by mgr nbs on 4:32:09 PM, 6/23/22
'create Lakehouse with name: "mylakehouse"'
```



IPython Widgets

IPython Widgets are eventful Python objects that have a representation in the browser. You can use IPython Widgets as low-code controls (for example, slider or text box) in your notebook, just like the Jupyter notebook. Currently it only works in a Python context.

To use IPython Widgets

1. Import the *ipywidgets* module first to use the Jupyter Widget framework.

Python

```
import ipywidgets as widgets
```

2. Use the top-level `display` function to render a widget, or leave an expression of `widget` type at the last line of the code cell.

Python

```
slider = widgets.IntSlider()  
display(slider)
```

3. Run the cell. The widget displays in the output area.

Python

```
slider = widgets.IntSlider()  
display(slider)
```

```
1 import ipywidgets as widgets  
2 slider = widgets.IntSlider()  
3 display(slider)
```

✓ 3 sec - Command executed in 353 ms by

6:13:08 PM, 5/05/23

PySpark (Python) ▾



57

4. Use multiple `display()` calls to render the same widget instance multiple times. They remain in sync with each other.

Python

```
slider = widgets.IntSlider()  
display(slider)  
display(slider)
```

```
1 slider = widgets.IntSlider()  
2 display(slider)  
3 display(slider)
```

✓ 4 sec - Command executed in 358 ms by

on 6:14:17 PM, 5/05/23

PySpark (Python) ▾



21



21

5. To render two widgets independent of each other, create two widget instances:

Python

```
slider1 = widgets.IntSlider()  
slider2 = widgets.IntSlider()  
display(slider1)  
display(slider2)
```

```

1 slider1 = widgets.IntSlider()
2 slider2 = widgets.IntSlider()
3 display(slider1)
4 display(slider2)

```

4 sec - Command executed in 361 ms by on 6:14:08 PM, 5/05/23

PySpark (Python) ▾

90
28

Supported widgets

[Expand table](#)

Widgets type	Widgets
Numeric widgets	IntSlider, FloatSlider, FloatLogSlider, IntRangeSlider, FloatRangeSlider, IntProgress, FloatProgress, BoundedIntText, BoundedFloatText, IntText, FloatText
Boolean widgets	ToggleButton, Checkbox, Valid
Selection widgets	Dropdown, RadioButtons, Select, SelectionSlider, SelectionRangeSlider, ToggleButtons, SelectMultiple
String widgets	Text, Text area, Combobox, Password, Label, HTML, HTML Math, Image, Button
Play (animation) widgets	Date picker, Color picker, Controller
Container or layout widgets	Box, HBox, VBox, GridBox, Accordion, Tabs, Stacked

Known limitations

- The following widgets aren't supported yet. The following workarounds are available:

[Expand table](#)

Functionality	Workaround
<i>Output</i> widget	You can use <i>print()</i> function instead to write text into stdout.
<i>widgets.jslink()</i>	You can use <i>widgets.link()</i> function to link two similar widgets.
<i>FileUpload</i> widget	Not supported yet.

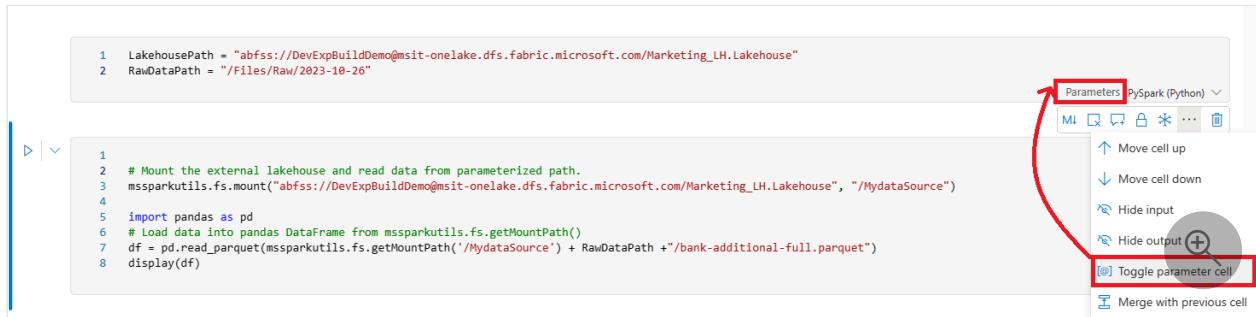
- The Fabric global *display* function doesn't support displaying multiple widgets in one call (for example, *display(a, b)*). This behavior is different from the IPython *display* function.

- If you close a notebook that contains an IPython widget, you can't see or interact with it until you execute the corresponding cell again.
- The interact function (ipywidgets.interact) is not supported.

Integrate a notebook

Designate a parameters cell

To parameterize your notebook, select the ellipses (...) to access the **More** commands at the cell toolbar. Then select **Toggle parameter cell** to designate the cell as the parameters cell.



The parameter cell is useful for integrating a notebook in a pipeline. Pipeline activity looks for the parameters cell and treats this cell as the default for the parameters passed in at execution time. The execution engine adds a new cell beneath the parameters cell with input parameters in order to overwrite the default values.

Assign parameters values from a pipeline

After you create a notebook with parameters, you can execute it from a pipeline with the Fabric notebook activity. After you add the activity to your pipeline canvas, you can set the parameters values under the **Base parameters** section of the **Settings** tab.

The screenshot shows the 'Notebook' activity configuration in the Azure Data Factory pipeline canvas. The 'General' tab is selected, showing the workspace as 'Fabrictest' and the notebook as 'Notebook1'. The 'Settings' tab is selected, showing the 'Base parameters' section. It contains two entries:

Name	Type	Value	Treat as null
LakehousePath	String	anotherLakehouse	<input type="checkbox"/>
RawDataPath	String	@pipeline().TriggerTime	<input type="checkbox"/>

When assigning parameter values, you can use the [pipeline expression language](#) or [functions and variables](#).

Spark session configuration magic command

You can personalize your Spark session with the magic command `%%configure`. Fabric notebook supports customized vCores, Memory of the Driver and Executor, Apache Spark properties, mount points, pool, and the default lakehouse of the notebook session. They can be used in both interactive notebook and pipeline notebook activities. We recommend that you run the `%%configure` command at the beginning of your notebook, or you must restart the Spark session to make the settings take effect.

JSON

```
%%configure
{
    // You can get a list of valid parameters to config the session from
    // https://github.com/cloudera/livy#request-body.
    "driverMemory": "28g", // Recommended values: ["28g", "56g", "112g",
    "224g", "400g"]
    "driverCores": 4, // Recommended values: [4, 8, 16, 32, 64]
    "executorMemory": "28g",
    "executorCores": 4,
    "jars": ["abfs[s]":
        //<file_system>@<account_name>.dfs.core.windows.net/<path>/myjar.jar",
        "wasb[s]":
            //<containername>@<accountname>.blob.core.windows.net/<path>/myjar1.jar"],
        "conf":
        {
            // Example of customized property, you can specify count of lines that
            // Spark SQL returns by configuring "livy.rsc.sql.num-rows".
            "livy.rsc.sql.num-rows": "3000",
            "spark.log.level": "ALL"
        },
        "defaultLakehouse": { // This overwrites the default lakehouse for
            current session
            "name": "<lakehouse-name>",
            "id": "<(optional) lakehouse-id>",
            "workspaceId": "<(optional) workspace-id-that-contains-the-lakehouse>"
        }
        // Add workspace ID if it's from another workspace
    },
    "mountPoints": [
        {
            "mountPoint": "/myMountPoint",
            "source": "abfs[s]://<file_system>@<account_name>.dfs.core.windows.net/<path>"
        },
        {
            "mountPoint": "/myMountPoint1",
            "source": "abfs[s]://<file_system>@<account_name>.dfs.core.windows.net/<path1>"
        },
    ]
}
```

```
],
  "environment": {
    "id": "<environment-id>",
    "name": "<environment-name>"
  },
  "sessionTimeoutInSeconds": 1200,
  "useStarterPool": false, // Set to true to force using starter pool
  "useWorkspacePool": "<workspace-pool-name>"
}
```

ⓘ Note

- We recommend that you set the same value for "DriverMemory" and "ExecutorMemory" in %%configure. The "driverCores" and "executorCores" values should also be the same.
- The "defaultLakehouse" will overwrite your pinned lakehouse in Lakehouse explorer, but that only works in your current notebook session.
- You can use %%configure in Fabric pipelines, but if it's not set in the first code cell, the pipeline run will fail due to cannot restart session.
- The %%configure used in notebookutils.notebook.run will be ignored but used in %run notebook will continue executing.
- The standard Spark configuration properties must be used in the "conf" body. Fabric does not support first level reference for the Spark configuration properties.
- Some special Spark properties, including "spark.driver.cores", "spark.executor.cores", "spark.driver.memory", "spark.executor.memory", and "spark.executor.instances" don't take effect in "conf" body.

Parameterized session configuration from a pipeline

Parameterized session configuration allows you to replace the value in %%configure magic with the pipeline run notebook activity parameters. When preparing %%configure code cell, you can override default values (also configurable, 4 and "2000" in the below example) with an object like this:

```
{
  "parameterName": "paramterNameInPipelineNotebookActivity",
```

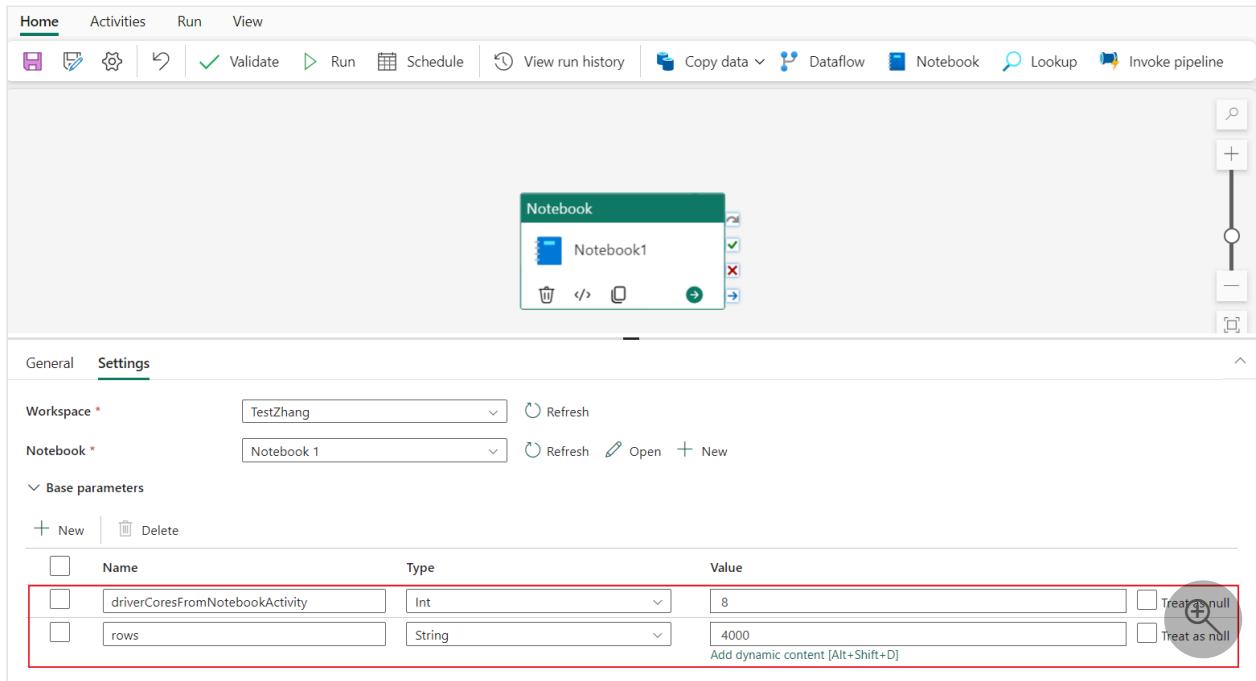
```
        "defaultValue": "defaultValueIfNoParameterFromPipelineNotebookActivity"  
    }  
}
```

Python

```
%configure  
  
{  
    "driverCores":  
    {  
        "parameterName": "driverCoresFromNotebookActivity",  
        "defaultValue": 4  
    },  
    "conf":  
    {  
        "livy.rsc.sql.num-rows":  
        {  
            "parameterName": "rows",  
            "defaultValue": "2000"  
        }  
    }  
}
```

A notebook uses the default value if you run a notebook in interactive mode directly or if the pipeline notebook activity gives no parameter that matches "activityParameterName."

During a pipeline run, you can configure pipeline notebook activity settings as follows:



If you want to change the session configuration, pipeline notebook activity parameters name should be same as `parameterName` in the notebook. In this example of running a pipeline, `driverCores` in `%%configure` are replaced by 8, and `livy.rsc.sql.num-rows` are replaced by 4000.

(!) Note

- If a pipeline run fails because you used the %%configure magic command, find more error information by running the %%configure magic cell in the interactive mode of the notebook.
- Notebook scheduled runs don't support parameterized session configuration.

Python logging in a notebook

You can find Python logs and set different log levels and format like the sample code shown here:

Python

```
import logging

# Customize the logging format for all loggers
FORMAT = "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
formatter = logging.Formatter(fmt=FORMAT)
for handler in logging.getLogger().handlers:
    handler.setFormatter(formatter)

# Customize log level for all loggers
logging.getLogger().setLevel(logging.INFO)

# Customize the log level for a specific logger
customizedLogger = logging.getLogger('customized')
customizedLogger.setLevel(logging.WARNING)

# logger that use the default global log level
defaultLogger = logging.getLogger('default')
defaultLogger.debug("default debug message")
defaultLogger.info("default info message")
defaultLogger.warning("default warning message")
defaultLogger.error("default error message")
defaultLogger.critical("default critical message")

# logger that use the customized log level
customizedLogger.debug("customized debug message")
customizedLogger.info("customized info message")
customizedLogger.warning("customized warning message")
customizedLogger.error("customized error message")
customizedLogger.critical("customized critical message")
```

View the history of input commands

Fabric notebook support magic command `%history` to print the input command history that executed in the current session, comparing to the standard Jupyter Ipython command the `%history` works for multiple languages context in notebook.

```
%history [-n] [range [range ...]]
```

For options:

- `-n`: Print execution number.

Where range can be:

- `N`: Print code of **N**th executed cell.
- `M-N`: Print code from **M**th to **N**th executed cell.

Example:

- Print input history from 1st to 2nd executed cell: `%history -n 1-2`

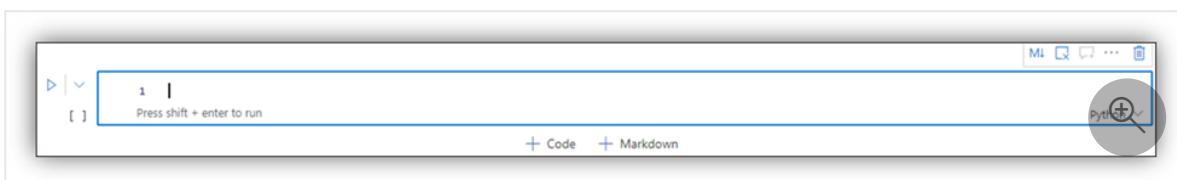
Shortcut keys

Similar to Jupyter Notebooks, Fabric notebooks have a modal user interface. The keyboard does different things depending on which mode the notebook cell is in. Fabric notebooks support the following two modes for a given code cell: Command mode and Edit mode.

- A cell is in Command mode when there's no text cursor prompting you to type. When a cell is in Command mode, you can edit the notebook as a whole but not type into individual cells. Enter Command mode by pressing ESC or using the mouse to select outside of a cell's editor area.



- Edit mode can be indicated from a text cursor that prompting you to type in the editor area. When a cell is in Edit mode, you can type into the cell. Enter Edit mode by pressing Enter or using the mouse to select a cell's editor area.



Shortcut keys in command mode

[Expand table](#)

Action	Notebook shortcuts
Run the current cell and select below	Shift+Enter
Run the current cell and insert below	Alt+Enter
Run current cell	Ctrl+Enter
Select cell above	Up
Select cell below	Down
Select previous cell	K
Select next cell	J
Insert cell above	A
Insert cell below	B
Delete selected cells	Shift + D
Switch to edit mode	Enter

Shortcut keys in edit mode

Using the following keystroke shortcuts, you can easily navigate and run code in Fabric notebooks when in Edit mode.

[Expand table](#)

Action	Notebook shortcuts
Move up cursor	Up
Move down cursor	Down
Undo	Ctrl + Z
Redo	Ctrl + Y
Comment or Uncomment	Ctrl + / Comment: Ctrl + K + C Uncomment: Ctrl + K + U
Delete word before	Ctrl + Backspace
Delete word after	Ctrl + Delete
Go to cell start	Ctrl + Home

Action	Notebook shortcuts
Go to cell end	Ctrl + End
Go one word left	Ctrl + Left
Go one word right	Ctrl + Right
Select all	Ctrl + A
Indent	Ctrl +]
Dedent	Ctrl + [
Switch to command mode	Esc

To find all shortcut keys, select **View** on the notebook ribbon, and then select **Keybindings**.

Related content

- [Notebook visualization](#)
- [Introduction of Fabric NotebookUtils](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Ask the community ↗](#)

Use Python experience on Notebook

Article • 03/31/2025

ⓘ Note

Currently, the feature is in preview.

The Python notebook is a new experience built on top of Fabric notebook. It is a versatile and interactive tool designed for data analysis, visualization, and machine learning. It provides a seamless developing experience for writing and executing Python code. This capability makes it an essential tool for data scientists, analysts, and BI developers, especially for exploration tasks that don't require big data and distributed computing.

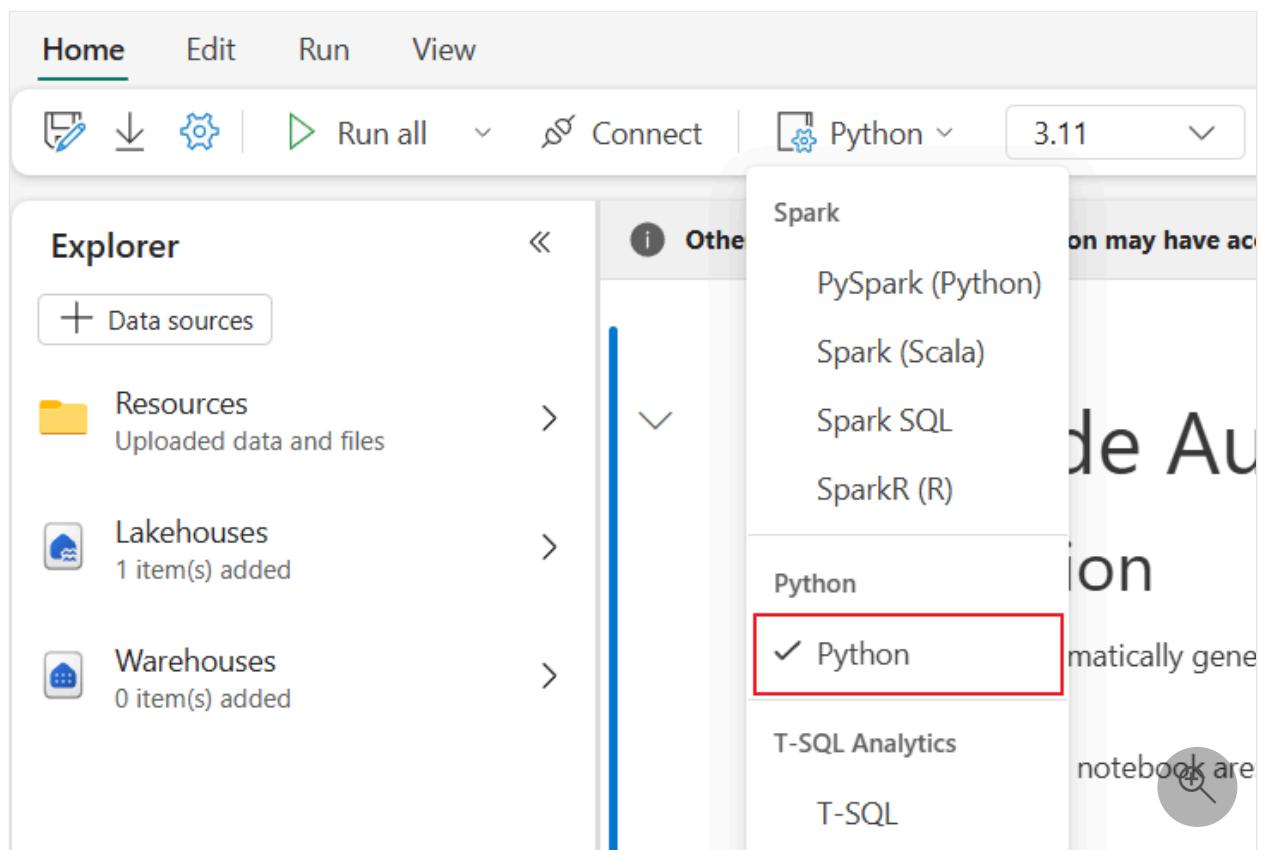
With a Python notebook, you can get:

- **Multiple built-in python kernels:** Python notebooks offer a pure Python coding environment without Spark, with two versions of Python kernel - Python 3.10 and 3.11 available by default, and the native ipython features supported such as iPyWidget, magic commands.
- **Cost effective:** The new Python notebook offers cost-saving benefits by running on a single node cluster with 2vCores/16GB memory by default. This setup ensures efficient resource utilization for data exploration projects with smaller data size.
- **Lakehouse & Resources are natively available:** The Fabric Lakehouse together with Notebook built-in Resources full functionality are available in Python notebook. This feature enables users to easily bring the data to python notebook, just try drag & drop to get the code snippet.
- **Mix programming with T-SQL:** Python notebook offers an easy way to interact with Data Warehouse and SQL endpoints in explorer, by using notebookutils data connector, you can easily execute the T-SQL scripts under the context of python.
- **Support for Popular Data Analytic libraries:** Python notebooks come with pre-installed libraries such as DuckDB, Polars, and Scikit-learn, providing a comprehensive toolkit for data manipulation, analysis, and machine learning.
- **Advanced intellisense:** Python notebook is adopting Pylance as the intellisense engine, together with other Fabric customized language service, aiming to provide state of art coding experience for notebook developers.

- **NotebookUtils & Semantic link:** Powerful API toolkits empower you easily use Fabric and Power BI capabilities with code-first experience.
- **Rich Visualization Capabilities:** Except for the popular rich dataframe preview 'Table' function and 'Chart' function, we also support popular visualization libraries like Matplotlib, Seaborn, and Plotly. The PowerBIClient also supports these libraries to help users better understanding data patterns and insights.
- **Common Capabilities for Fabric Notebook:** All the Notebook level features are naturally applicable for Python notebook, such as editing features, AutoSave, collaboration, sharing and permission management, Git integration, import/export, etc.
- **Full stack Data Science Capabilities:** The advanced low-code toolkit Data Wrangler, the machine learning framework MLFlow, and powerful Copilot are all available on Python notebook.

How to access Python Notebook

After opening a Fabric Notebook, you can switch to *Python* in the language dropdown menu at **Home** tab and convert the entire notebook set-up to Python.



Most of the common features are supported as a notebook level, you can refer the [How to use Microsoft Fabric notebooks](#) and [Develop, execute, and manage Microsoft Fabric](#)

[notebooks](#) to learn the detailed usage. Here we list some key capabilities specific for Python scenarios.

Run Python notebooks

Python notebook supports multiple job execution ways:

- **Interactive run:** You can run a Python notebook interactively like a native Jupyter notebook.
- **Schedule run:** You can use the light-weighted scheduler experience on the notebook settings page to run Python notebook as a batch job.
- **Pipeline run:** You can orchestrate Python notebooks as notebook activities in [Data pipeline](#). Snapshot will be generated after the job execution.
- **Reference run:** You can use `notebookutils.notebook.run()` or `notebookutils.notebook.runMultiple()` to reference run Python notebooks in another Python notebook as batch job. Snapshot will be generated after the reference run finished.
- **Public API run:** You can schedule your python notebook run with the [notebook run public API](#), make sure the language and kernel properties in notebook metadata of the public API payload are set properly.

You can monitor the Python notebook job run details on the ribbon tab **Run -> View all runs**.

Data interaction

You can interact with Lakehouse, Warehouses, SQL endpoints, and built-in resources folders on Python notebook.

ⓘ Note

The Python Notebook runtime comes pre-installed with [delta-rs](#) and [duckdb](#) libraries to support both reading and writing Delta Lake data. However, please note that some Delta Lake features may not be fully supported at this time. For more details and the latest updates, kindly refer to the official [delta-rs](#) and [duckdb](#) websites.

Lakehouse interaction

You can set a Lakehouse as the default, or you can also add multiple Lakehouses to explore and use them in notebooks.

If you are not familiar with reading the data objects like *delta table*, try drag and drop the file and delta table to the notebook canvas, or use the *Load data* in the object's dropdown menu. Notebook automatically inserts code snippet into code cell and generating code for reading the target data object.

① Note

If you encounter OOM when loading large volume of data, try using DuckDB, Polars or PyArrow dataframe instead of pandas.

You can find the write Lakehouse operation in **Browse code snippet -> Write data to delta table**.

The screenshot shows a Jupyter Notebook interface with the following details:

- Code Snippets:** A sidebar titled "Code snippet library" contains snippets for PySpark, R, Python, Scala, SparkSQL, and TSQL. The "Python" tab is selected, and the "Write data to delta table" snippet is highlighted.
- Code Cells:** Two code cells are visible:
 - Cell 3 (Python):

```
1 import pandas as pd
2 from deltalake import write_deltalake
3 table_path = "/abfss://dlt160e23-d798-4dc7-a7ac-bb507060ded@dlt-onelake.dfs.fabric.microsoft.com/fib39ea5-960c-4cc5-8a3c-6dc3b7b82baa/Tables/customerlist"
4 storage_options = {"bearer_token": notebookutils.credentials.getToken("storage"), "use_fabric_endpoint": "true"}
5 df = pd.DataFrame({"id": range(5, 10)})
6 write_deltalake(table_path, df, mode='overwrite', schema_mode='merge', engine='rust', storage_options=storage_options)
7
8 ## Write to mounted path
9 # delta_table_path = "/Lakehouse/default/Tables/yourTable" #Fill in your delta table path
10 # df = pd.DataFrame({"id": range(5, 10)})
11 # write_deltalake(delta_table_path, df, mode='overwrite', schema_mode='merge', engine='rust', storage_options={"allow_unsafe_rename": "true"})
```
 - Cell 4 (Python):

```
1 from deltalake import DeltaTable, write_deltalake
2 table_path = "/abfss://dlt160e23-d798-4dc7-a7ac-bb507060ded@dlt-onelake.dfs.fabric.microsoft.com/fib39ea5-960c-4cc5-8a3c-6dc3b7b82baa/Tables/customerlist"
3 storage_options = {"bearer_token": notebookutils.credentials.getToken("storage"), "use_fabric_endpoint": "true"}
4 dt = DeltaTable(table_path, storage_options=storage_options)
5 limited_data = dt.to_pyarrow_dataset().head(1000).to_pandas()
6 display(limited_data)
7
8 # Write data frame to Lakehouse
9 # write_deltalake(table_path, limited_data, mode='overwrite')
10
11 # If the table is too large and might cause an Out of Memory (OOM) error,
12 # you can try using the code below. However, please note that delta_scan with default lakehouse is currently in preview.
13 # import duckdb
14 # display(duckdb.sql("select * from delta_scan('/Lakehouse/default/Tables/dbo/bigdeltatable') limit 1000").df())
```
- Table View:** A table view titled "Table view" shows a single row with columns "id" and values 5, 6, 7.
- System Status:** At the bottom left, it says "Session ready" and "CPU 2vCores (1.2%), RAM 16G (13.0%)".
- Code Completion:** It says "Copilot completions: Off".

Warehouse interaction and mix programming with T-SQL

You can add Data Warehouses or SQL endpoints from the Warehouse explorer of Notebook. Similarly, you can drag and drop the tables into the notebook canvas, or use the shortcut operations in the table dropdown menu. Notebook automatically generates code snippet for you. You can use the [notebookutils.data utilities](#) to establish a connection with Warehouses and query the data using T-SQL statement in the context of Python.

The screenshot shows a Microsoft Power BI Data Studio interface. The top navigation bar includes Home, Edit, Run, View, and a Python 3.11 kernel selector. The left sidebar shows 'Warehouses' with a '+ Warehouses' button. Below it is 'BankCustomerChurnW...', expanded to show 'Schemas' (with 'dbo'), 'Tables' (with 'ChurnIn'), 'Views', 'Functions', 'Stored Proc.', 'INFORMATION_SCHEMA', and 'queryinsights'. A context menu is open over 'ChurnIn' with options: 'SELECT TOP 100', 'CREATE', 'DROP', and 'DROP and CREATE'. The main panel features a large title 'Create, evaluate prediction m...' followed by 'Introduction'. The introduction text discusses building a model to predict bank customer churn. It lists four main steps: 1. Load the data, 2. Understand and process the data using Wrangler feature, 3. Train machine learning models using Autologging feature, and 4. Evaluate and save the final machine learning model.

Note

SQL endpoints are read-only here.

Notebook resources folder

The [Notebook resources](#) built-in resources folder is natively available on Python Notebook. You can easily interact with the files in built-in resources folder using Python code as if you are working with your local file system. Currently, the Environment resource folder is not supported.

Kernel operations

Python notebook support two built-in kernels right now, they are *Python 3.10* and *Python 3.11*, the default selected kernel is *Python 3.11*. you can easily switch between

them.

You can interrupt, restart, or switch kernel on the **Home** tab of the ribbon. Interrupting kernel in Python notebooks is same as canceling cell in Spark notebook.



Abnormal kernel exit causes code execution to be interrupted and losing variables, but it doesn't stop the notebook session.

There are commands that can lead to kernel died. For example, `quit()`, `exit()`.

Library management

You can use `%pip` and `%conda` commands for inline installations, the commands support both public libraries and customized libraries.

For customized libraries, you can upload the lib files to the **Built-in resources** folder. We support multiple types of libraries like `.whl`, `.jar`, `.dll`, `.py`, etc., just try drag&drop to the file and the code snippet is generated automatically.

You may need to restart the kernel to use the updated packages.

Session configuration magic command

Similar with personalizing a [Spark session configuration](#) in notebook, you can also use `%%configure` in Python notebook too. Python notebook supports customizing compute node size, mount points and default lakehouse of the notebook session. They can be used in both interactive notebook and pipeline notebook activities. We recommend using `%%configure` command at the beginning of your notebook, or you must restart the notebook session to make the settings take effect.

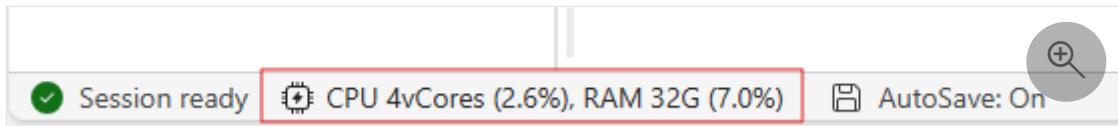
Here are the supported properties in Python notebook `%%configure`:

```
JSON

%%configure -f
{
    "vCores": 4, // Recommended values: [4, 8, 16, 32, 64], Fabric will
    // allocate matched memory according to the specified vCores.
    "defaultLakehouse": {
        // Will overwrites the default lakehouse for current session
        "name": "<lakehouse-name>",
    }
}
```

```
        "id": "<(optional) lakehouse-id>",
        "workspaceId": "<(optional) workspace-id-that-contains-the-
lakehouse>" // Add workspace ID if it's from another workspace
    },
    "mountPoints": [
        {
            "mountPoint": "/myMountPoint",
            "source": "abfs[s]://<file_system>@<account_name>.dfs.core.windows.net/<path>"
        },
        {
            "mountPoint": "/myMountPoint1",
            "source": "abfs[s]://<file_system>@<account_name>.dfs.core.windows.net/<path1>"
        },
    ],
}
```

You can view the compute resources update on notebook status bar, and monitor the CPU and Memory usage of the compute node in real-time.



NotebookUtils

Notebook Utilities (NotebookUtils) is a built-in package to help you easily perform common tasks in Fabric Notebook. It is pre-installed on Python runtime. You can use NotebookUtils to work with file systems, to get environment variables, to chain notebooks together, to access external storage, and to work with secrets.

You can use `notebookutils.help()` to list available APIs and also get help with methods, or referencing the doc [NotebookUtils](#).

Data utilities

Note

Currently, the feature is in preview.

You can use `notebookutils.data` utilities to establish a connection with provided data source and then read and query data using T-SQL statement.

Run the following command to get an overview of the available methods:

Python

```
notebookutils.data.help()
```

Output:

Console

```
Help on module notebookutils.data in notebookutils:
```

NAME

```
    notebookutils.data - Utility for read/query data from connected data  
sources in Fabric
```

FUNCTIONS

```
    connect_to_artifact(artifact: str, workspace: str = '', artifact_type:  
str = '', **kwargs)
```

```
        Establishes and returns an ODBC connection to a specified artifact  
within a workspace
```

```
        for subsequent data queries using T-SQL.
```

```
:param artifact: The name or ID of the artifact to connect to.
```

```
:param workspace: Optional; The workspace in which the provided  
artifact is located, if not provided,
```

```
                use the workspace where the current notebook is  
located.
```

```
:param artifactType: Optional; The type of the artifact, Currently  
supported type are Lakehouse, Warehouse and MirroredDatabase.
```

```
        If not provided, the method will try to  
determine the type automatically.
```

```
:param **kwargs Optional: Additional optional configuration.
```

```
Supported keys include:
```

```
    - tds_endpoint : Allow user to specify a custom TDS endpoint to  
use for connection.
```

```
:return: A connection object to the specified artifact.
```

```
:raises UnsupportedArtifactException: If the specified artifact type  
is not supported to connect.
```

```
:raises ArtifactNotFoundException: If the specified artifact is not  
found within the workspace.
```

Examples:

```
    sql_query = "SELECT DB_NAME()  
    with
```

```
notebookutils.data.connect_to_artifact("ARTIFACT_NAME_OR_ID",  
"WORKSPACE_ID", "ARTIFACT_TYPE") as conn:
```

```
    df = conn.query(sql_query)  
    display(df)
```

```
    help(method_name: str = '') -> None
```

```
        Provides help for the notebookutils.data module or the specified  
method.
```

```
Examples:  
notebookutils.data.help()  
notebookutils.data.help("connect_to_artifact")  
:param method_name: The name of the method to get help with.
```

```
DATA
```

```
__all__ = ['help', 'connect_to_artifact']
```

```
FILE
```

```
/home/trusted-service-user/jupyter-env/python3.10/lib/python3.10/site-packages/notebookutils/data.py
```

Query data from Lakehouse

```
Python
```

```
conn = notebookutils.data.connect_to_artifact("lakehouse_name_or_id",  
"optional_workspace_id", "optional_lakehouse_type")  
df = conn.query("SELECT * FROM sys.schemas;")
```

Query data from Warehouse

```
Python
```

```
conn = notebookutils.data.connect_to_artifact("warehouse_name_or_id",  
"optional_workspace_id", "optional_warehouse_type")  
df = conn.query("SELECT * FROM sys.schemas;")
```

ⓘ Note

The Data utilities in NotebookUtils are only available on Python notebook for now.

Browse code snippets

You can find useful python code snippets on **Edit tab-> Browse code snippet**, new Python samples are now available. You can learn from the Python code snippet to start exploring the notebook.

The screenshot shows a Microsoft Fabric Python notebook interface. On the left, there's a sidebar with 'Lakehouses' (CustomerRawLakehouse) and 'Tables'. The main area contains two code cells. Cell [10] shows a snippet for data preprocessing:

```

1 # Data preprocessing
2 cols = ['Row ID', 'Order ID', 'Ship Date', 'Ship Mode', 'Customer ID', 'Customer Name',
3 'Segment', 'Country', 'City', 'State', 'Postal Code', 'Region', 'Product ID', 'Category',
4 'Sub-Category', 'Product Name', 'Quantity', 'Discount', 'Profit']
5 # Drop unnecessary columns
6 furniture.drop(cols, axis=1, inplace=True)
7 furniture = furniture.sort_values('Order Date')
8 furniture.isnull().sum()

```

Cell [11] shows a snippet for data preparation:

```

1 # Data Preparation
2 furniture = furniture.groupby('Order Date')['Sales'].sum().reset_index()
3 furniture = furniture.set_index('Order Date')
4 furniture.index
5 y = furniture['Sales'].resample('MS').mean()
6 y = y.reset_index()
7 y['Order Date'] = pd.to_datetime(y['Order Date'])
8 y['Order Date'] = [i+pd.DateOffset(months=6) for i in y['Order Date']]
9 y = y.set_index(['Order Date'])
10 maximim_date = y.reset_index()['Order Date'].max()

```

The right side features a 'Code snippet library' panel with tabs for PySpark, R, Python (selected), Scala, SparkSQL, and TSQL. It includes sections for 'Notebookutils' (Get secret, Get token, Check token, Append content to a file) and 'Get secret' (Description: GetSecret returns an Azure Key Vault secret for a given Azure Key Vault endpoint and secret name using user credentials). A code snippet for 'notebookutils.credentials.getSecret' is shown.

Semantic link

Semantic link is a feature that allows you to establish a connection between [semantic models](#) and Synapse Data Science in Microsoft Fabric. It is natively supported on Python notebook. BI engineers and Power BI developers can use Semantic link connect and manage semantic model easily. Read the [public document](#) to learn more about Semantic link.

Visualization

In addition to drawing charts with libraries, the [built-in visualization](#) function allows you to turn DataFrames into rich format data visualizations. You can use the `display()` function on dataframes to produce the rich dataframe table view and chart view.

```

1 import pandas as pd
2 # Load data into pandas DataFrame from "/lakehouse/default/Files/churn.csv"
3 df = pd.read_csv("/Lakehouse/default/Files/churn.csv")
4 display(df)
5

```

[1] ✓ - Session ready in 6 sec 812 ms. Command executed in 3 sec 354 ms by ...

Table Country HasCrCards + New chart

Table view

14 columns, 1000 rows ...

Inspect

12L RowNumber

Missing: 0 (0%) Unique: 1000 (100%) Invalid: 0 (0%)

Min 1 Max 1000

12L CustomerId

Missing: 0 (0%) Unique: 1000 (100%) Invalid: 0 (0%)

Min 15566091 Max 15815364

ABC Surname

Missing: 0 (0%) Unique: 740 (74%)

① Note

The chart configurations will be persisted in Python notebook, which means after rerunning the code cell, if the target dataframe schema hasn't change, the saved charts are still persisted.

Code intelliSense

Python notebook also uses Pylance as the language server. For more information, see [enhance Python Development with Pylance](#).

Data science capabilities

Visit [Data Science documentations in Microsoft Fabric](#) to learn more data science and AI experience in Fabric. Here we list a few key data science features that are natively supported on Python notebook.

- **Data Wrangler:** Data Wrangler is a notebook-based tool that provides an immersive interface for exploration data analysis. This feature combines a grid-like data display with dynamic summary statistics, built-in visualizations, and a library of common data cleaning operations. It provides data cleaning, data transformation, and integration, which accelerates data preparation with Data Wrangler.

- **MLflow:** A machine learning experiment is the primary unit of organization and control for all related machine learning runs. A run corresponds to a single execution of model code.
- **Fabric Auto Logging:** Synapse Data Science in Microsoft Fabric includes autologging, which significantly reduces the amount of code required to automatically log the parameters, metrics, and items of a machine learning model during training.

Autologging extends MLflow Tracking capabilities. Autologging can capture various metrics, including accuracy, loss, F1 score, and custom metrics you define. By using autologging, developers and data scientists can easily track and compare the performance of different models and experiments without manual tracking.

- **Copilot:** Copilot for Data Science and Data Engineering notebooks is an AI assistant that helps you analyze and visualize data. It works with lakehouse tables, Power BI Datasets, and pandas/spark dataframes, providing answers and code snippets directly in the notebook. You can use the Copilot chat panel and Char-magics in notebook, and the AI provides responses or code to copy into your notebook.

Public preview known limitations

- Live pool experience is not guaranteed for every python notebook run. The session start time may take up to 3 minutes if the notebook run does not hit the live pool. As Python notebook usage grows, our intelligent pooling methods gradually increase the live pool allocation to meet the demand.
- Environment integration is not available on Python notebook by public preview.
- Set session timeout is not available for now.
- Copilot may generate Spark statement, which may not executable in Python notebook.
- Currently, Copilot on Python notebook is not fully supported in several regions. The deployment process is still ongoing stay tuned as we continue to roll out support in more regions.

Related content

- [How to use Microsoft Fabric notebooks](#)

- Develop, execute, and manage Microsoft Fabric notebooks
 - Introduction of Fabric NotebookUtils
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Format code in Microsoft Fabric notebooks

Article • 07/25/2024

There are many benefits to adopting good style and conventions when you write a Python notebook or Apache Spark job definition. By consistently formatting your code, you can:

- Make it easier to read the code.
- Increases maintainability of the code.
- Conduct faster code reviews.
- Perform more accurate diffs, which detect changes between versions.

Specifically, this article describes how you can extend a Fabric notebook to use a [PEP 8-compliant](#) code formatter.

ⓘ Note

PEP is the acronym for Python Enhancement Proposals. PEP 8 is a style guide that describes coding conventions for Python code.

Extend Fabric notebooks

You can extend a Fabric notebook by using a *notebook extension*. A notebook extension is a software component that adds new functionality to the notebook interface. You install an extension as a library, and then you set it up to meet your specific needs.

This article considers two extensions that you can use to format Python code in a Fabric notebook. Both extensions are freely available from GitHub.

- The [black](#) Python code formatter extension.
- The [jupyter-black](#) formatter extension, which you can also use to automatically format code in a Jupyter Notebook or Jupyter Lab.

Set up a code formatter extension

There are two methods to set up a code formatter extension in a Fabric notebook. The first method involves workspace settings, while the second method involves in-line installation. You enable code formatting after you install the extension.

Workspace settings

Use the workspace settings to set up the working environment for a Fabric workspace. To make your libraries available for use in any notebooks and Spark job definitions in the workspace, you can create the environment, install the libraries in it, and then your workspace admin can attach the environment as the default for the workspace. Therefore, when a code formatter extension is installed in the workspace's default environment, all notebooks within the workspace can benefit from it.

For more information on environments, see [create, configure, and use an environment in Microsoft Fabric](#).

In-line installation

Use the in-line installation method when you want to install a library for a specific notebook, rather than all notebooks in a workspace. This approach is convenient when you want a temporary or quick solution that shouldn't affect other notebooks in the workspace.

To learn how to perform an in-line installation, see [In-line installation](#).

In the following example, the %pip command is used to install the latest version of the *Jupyter-black* extension on both the driver and executor nodes.

Python

```
# Install the latest version by using %pip command
# It will be available on both driver and executor nodes
%pip install jupyter-black
```

Enable code formatting

After you install the code formatting extension, you must enable code formatting for the notebook. You do that by loading the extension, which can be done in one of two ways.

Either use the `%load_ext` magic command.

Python

```
# Load the jupyter-black extension
%load_ext jupyter_black
```

Or, use the load extension by using the programming API.

Python

```
import jupyter_black  
jupyter_black.load()
```

💡 Tip

To ensure that all notebooks enable code formatting, which can be helpful in enterprise development scenarios, enable formatting in a template notebook.

Format code

To format code, select the lines of code you want to format, and then press **Shift+Enter**.

The following 11 lines of code aren't yet properly formatted.

Python

```
def unique(list_input):  
    list_set = set(list_input  
    )  
    unique_list = (list(  
        list_set  
        )  
    )  
    for x in unique_list:  
        print(  
            x  
        )
```

After formatting has been applied, the extension reduced the script to only five lines.

The code now adopts good style and conventions.

Python

```
def unique(list_input):  
    list_set = set(list_input)  
    unique_list = list(list_set)  
    for x in unique_list:  
        print(x)
```

Related content

For more information about Fabric notebooks, see the following articles.

- Manage Apache Spark libraries in Microsoft Fabric
 - Questions? Try asking the [Fabric Community](#).
 - Suggestions? [Contribute ideas to improve Fabric](#).
-

Feedback

Was this page helpful?



Yes



No

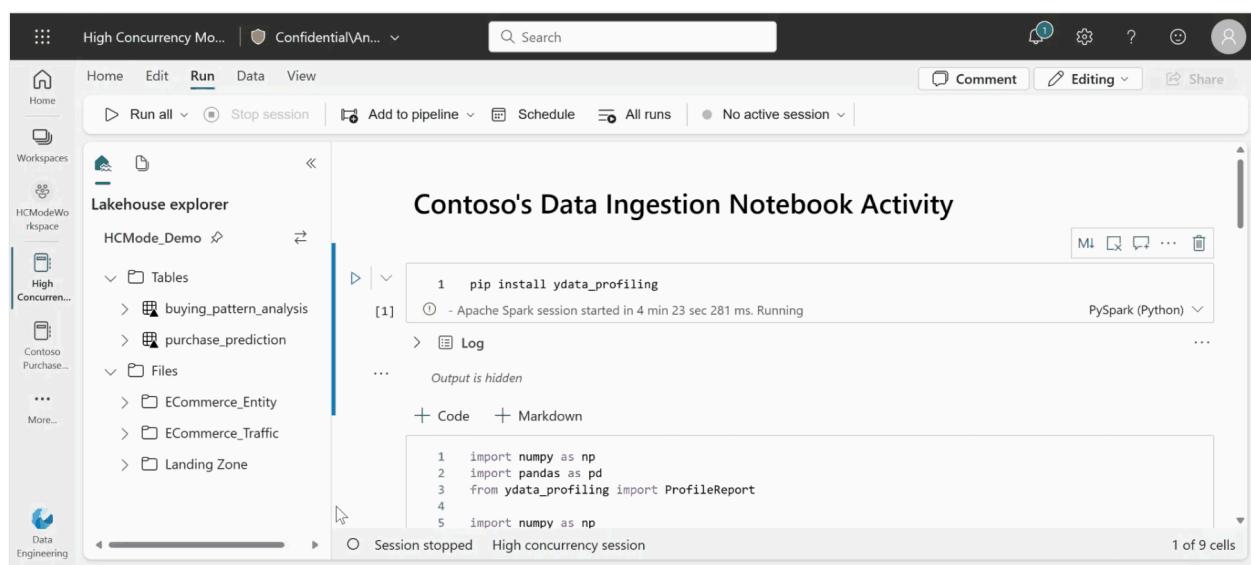
[Provide product feedback](#) | [Ask the community](#)

Configure high concurrency mode for Fabric notebooks

Article • 01/17/2025

When you run a notebook in Microsoft Fabric, an Apache Spark session is started and is used to run the queries submitted as part of the notebook cell executions. With high concurrency mode enabled, there's no need to start new spark sessions every time to run a notebook.

If you already have a high concurrency session running, you could attach notebooks to the high concurrency session getting a spark session instantly to run the queries and achieve a greater session utilization rate.



Note

The high concurrency mode-based session sharing is always within a single user boundary. The notebooks need to have matching spark configurations, should be part of the same workspace, share the same default lakehouse and libraries to share a single spark session.

Session sharing conditions

For notebooks to share a single Spark session, they must:

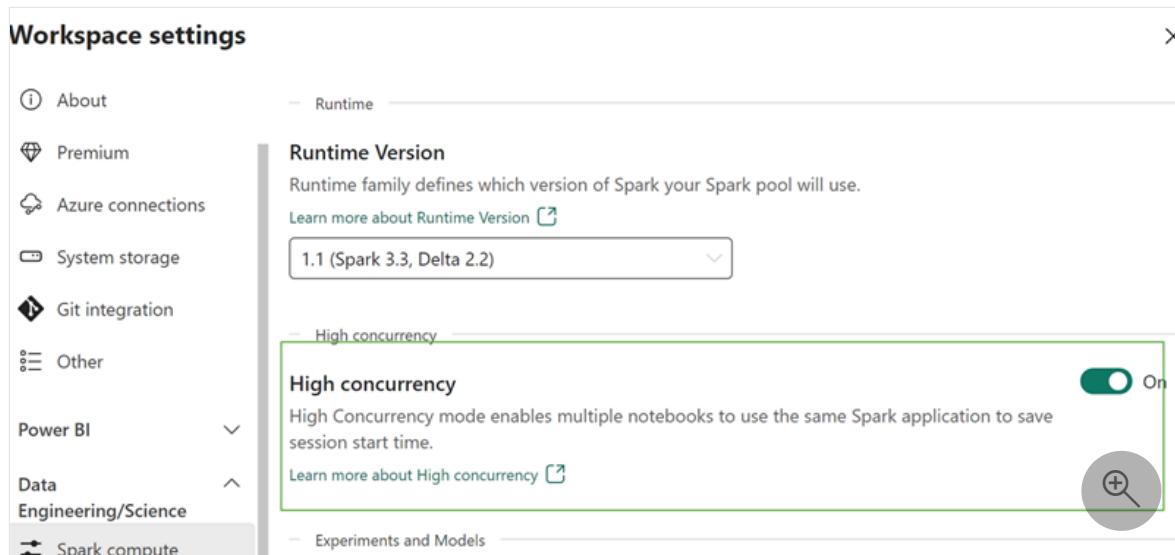
- Be run by the same user.
- Have the same default lakehouse. Notebooks without a default lakehouse can share sessions with other notebooks that don't have a default lakehouse.

- Have the same Spark compute configurations.
- Have the same library packages. You can have different inline library installations as part of notebook cells and still share the session with notebooks having different library dependencies.

Configure high concurrency mode

By default, all the Fabric workspaces are enabled with high concurrency mode. Use the following steps to configure the high concurrency feature:

1. Click on **Workspace settings** option in your Fabric workspace.
2. Navigate to the **Data Engineering/Science** section > **Spark settings** > **High concurrency**.
3. In the **High concurrency** section, enable the **For notebooks** setting. You can choose to **enable** or **disable** the setting from this pane.



4. Enabling the high concurrency option allows users to start a high concurrency session in their notebooks or attach to existing high concurrency session.
5. Disabling the high concurrency mode hides the section to configure the time period of inactivity and also hides the option to start a new high concurrency session from the notebook menu.

High concurrency

High Concurrency mode enables multiple notebooks to use the same Spark application to save session start time.

[Learn more about high concurrency mode](#)

Configurations

Spark properties

Spark properties allows you to define many Spark runtime properties.

[Learn more about Spark properties](#)

Show inherit properties

+ Add Delete

Run notebooks in high concurrency session

1. Open the Fabric workspace.
2. Create a notebook or open an existing notebook.
3. Navigate to the Run tab in the menu ribbon and select the **session type** dropdown that has **Standard** selected as the default option.

Predict NYC Taxi Tips using Spark ML and Azure Open Datasets | Status 12/12/20

Home Edit Run Data science View

Run all Stop session Add to pipeline Schedule run All runs Standard session

4. Select **New high concurrency session**.
5. Once the high concurrency session has started, you could now add upto 5 notebooks in the high concurrency session.

Predict NYC Taxi Tips using Spark ML and Azure Open Datasets | Status 12/12/20

Home Edit Run Data science View

Run all Stop session Add to pipeline Schedule run All runs Standard session

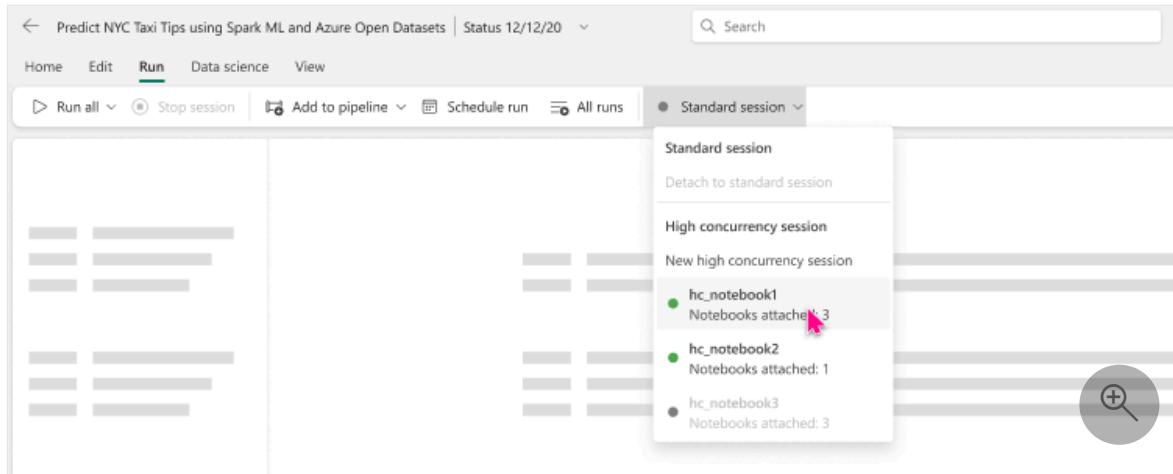
Standard session
Detach to standard session

High concurrency session
New high concurrency session

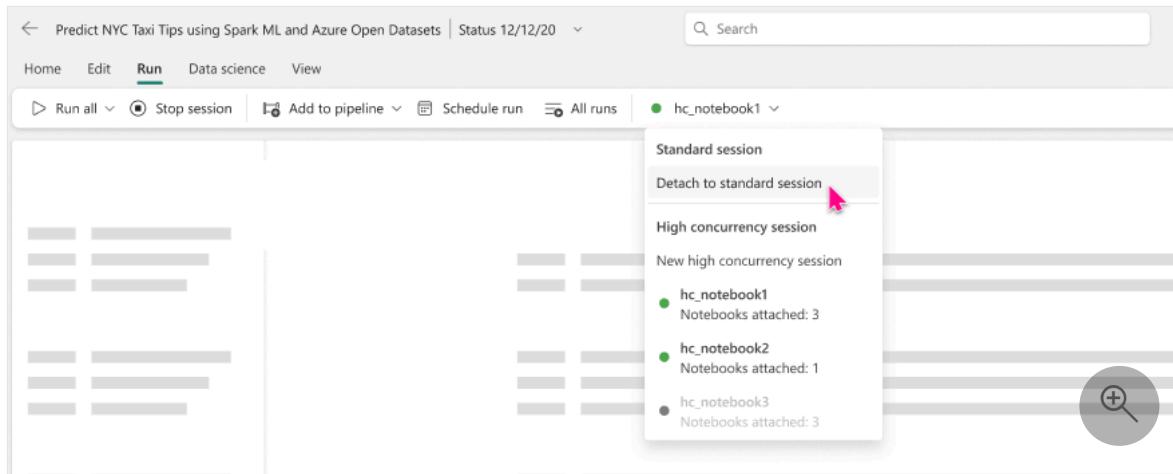
- hc_notebook1 Notebooks attached: 3
- hc_notebook2 Notebooks attached: 1
- hc_notebook3 Notebooks attached: 3

6. Create a new notebook and by navigating to the **Run** menu as mentioned in the above steps, in the drop-down menu you will now see the newly created high concurrency session listed.

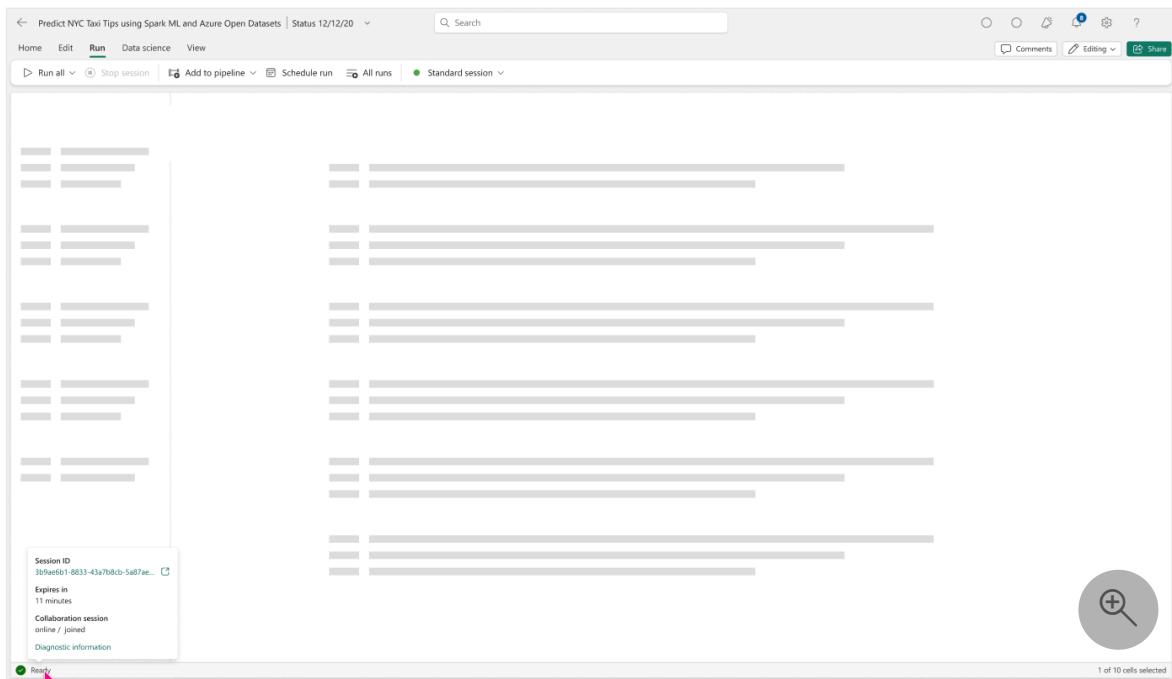
7. Selecting the existing high concurrency session attaches the second notebook to the session.



8. Once the notebook has been attached, you can start executing the notebook steps instantly.
9. The high concurrency session status also shows the number of notebooks attached to a given session at any point in time.
10. At any point in time if you feel the notebook attached to a high concurrency session requires more dedicated compute, you can choose to switch the notebook to a standard session by selecting the option to detach the notebook from the high concurrency in the Run menu tab.



11. You can view the session status, type, and ID in the **status bar**. Select the **Session ID** to explore the jobs executed in this high concurrency session and to view logs of the spark session on the monitoring detail page.



Monitoring and debugging notebooks running in high concurrency session

Monitoring and debugging are often a non-trivial task when you are running multiple notebooks in a shared session. For high concurrency mode in Fabric, separation of logs is offered which would allow users to trace the logs emitted by spark events from different notebooks.

1. When the session is in progress or in completed state, you can view the session status by navigating to the **Run** menu and selecting the **All Runs** option
2. This would open up the run history of the notebook showing the list of current active and historic spark sessions

The screenshot shows the Databricks interface with the 'Run' menu selected. A modal window titled 'All runs' is displayed, showing a table of runs for 'Notebook 1'. The table includes columns for Application name, Submitted, Submitter, Status, Total dur., Run kind, and Livy Id. One specific run is highlighted with a red box, showing 'HC_Notebook_2_d2d5' as the application name, submitted on 3/23/23 4:00:44 PM, by Jessie Irwin, and currently 'Running'. The total duration is 12m 17s and it's a manual run.

3. Users by selecting a session, can access the monitoring detail view, which shows the list of all the spark jobs that have been run in the session.
4. In the case of high concurrency session, users could identify the jobs and its associated logs from different notebooks using the **Related notebook** tab, which shows the notebook from which that job has been run.

HC_Notebook 2_122f2663-1b77-4aba-8b34-668aa1d5cfdb									
Jobs		Logs	Data	Related items					
ID	Description	Status	Stages	Tasks	Duration	Rows	Data read	Data written	Related notebook
> Job 0	save at <console>-31	✓ Succeeded	1/1	3/3 succeeded	15 sec	3	0 B	3.88 KB	Notebook 2 ↗
> Job 1		✓ Succeeded	0/0	0/0 succeeded	< 1 ms	0	0 B	0 B	Notebook 1 ↗
> Job 2	\$anonfun\$recordDeltaOperation\$\$ at SynapseLoggingShim.scala:95	✓ Succeeded	1/1	1/1 succeeded	2 sec	12	2.14 KB	2.4 KB	● Not found ⚠
> Job 3	\$anonfun\$recordDeltaOperation\$\$ at SynapseLoggingShim.scala:95	✓ Succeeded	1/1	50/50 succeeded	9 sec	56	2.4 KB	4.24 KB	Notebook 2 ↗
> Job 4	\$anonfun\$recordDeltaOperation\$\$ at SynapseLoggingShim.scala:95	✓ Succeeded	1/1	1/1 succeeded	< 1 ms	50	4.24 KB	0 B	Notebook 1 ↗
> Job 5	\$anonfun\$recordDeltaOperation\$\$ at SynapseLoggingShim.scala:95	✓ Succeeded	1/1	50/50 succeeded	1 sec	5	2.29 KB	0 B	● Not found ⚠
> Job 6	takeAsList at <console>-36	✓ Succeeded	1/1	1/1 succeeded	1 sec	1	3 KB	0 B	Notebook 2 ↗
> Job 7	save at <console>-31	✓ Succeeded	1/1	3/3 succeeded	7 sec	3	0 B	3.88 KB	Notebook 1 ↗
> Job 8	\$anonfun\$recordDeltaOperation\$\$ at SynapseLoggingShim.scala:95	✓ Succeeded	1/1	50/50 succeeded	< 1 ms	5	2.29 KB	0 B	● Not found ⚠
> Job 9	\$anonfun\$recordDeltaOperation\$\$ at SynapseLoggingShim.scala:95	✓ Succeeded	1/1	2/2 succeeded	1 sec	26	4.53 KB	5.69 KB	Notebook 2 ↗
> Job 10	\$anonfun\$recordDeltaOperation\$\$ at SynapseLoggingShim.scala:95	✓ Succeeded	1/1	50/50 succeeded	1 sec	63	5.69 KB	4.26 KB	Notebook 1 ↗
> Job 11	\$anonfun\$recordDeltaOperation\$\$ at SynapseLoggingShim.scala:95	✓ Succeeded	1/1	1/1 succeeded	< 1 ms	50	4.26 KB	0 B	● Not found ⚠
> Job 12	\$anonfun\$recordDeltaOperation\$\$ at SynapseLoggingShim.scala:95	✓ Succeeded	1/1	50/50 succeeded	< 1 ms	8	2.92 KB	0 B	Notebook 2 ↗
> Job 13	takeAsList at <console>-36	✓ Succeeded	1/1	1/1 succeeded	< 1 ms	1	3 KB	0 B	Notebook 1 ↗
> Job 14	count at <console>-53	✓ Succeeded	2/2	600/600 succeeded	47 sec	8000000	9.83 GB	9.83 GB	● Failed to load ⚠
> Job 15	count at <console>-53	✓ Succeeded	2/2	600/600 succeeded	46 sec	8000000	9.83 GB	9.83 GB	● Failed to load ⚠

Related notebook

Code snippet

Diagnostics 0 1 ▲ 1

- Spark_User_AutoClassification_pre_run_storage_mount

No TSG is available yet for this error code. (Notebook: Notebook 2)

- Data and time skew

Name Skewed task(%) Max. task exec... Avg. executio... Max. data read Min. data read Avg. data read Skewness

Stage 23	1 (0.5%)	18 seconds (task ...	452 ms	3.91 GB	30.21 MB	50.31 MB	0.21265937890083883
Stage 25	1 (0.5%)	19 seconds (task ...	420 ms	3.91 GB	30.12 MB	50.31 MB	0.2126084934770817



Related content

In this document, you get a basic understanding of a session sharing through high concurrency mode in notebooks. Advance to the next articles to learn how to create and get started with your own Data Engineering experiences using Lakehouse and Notebooks:

- To get started with Lakehouse, see [Create a lakehouse in Microsoft Fabric](#).
- To get started with notebooks, see [How to use a Microsoft Fabric notebooks](#).

Feedback

Was this page helpful?



Provide product feedback ↗ | Ask the community ↗

Notebook visualization in Microsoft Fabric

Article • 01/06/2025

Microsoft Fabric is an integrated analytics service that accelerates time to insight across data warehouses and big data analytics systems. Data visualization in notebooks is a key component that allows you to gain insight into your data. It helps make large and small data easier for humans to understand. It also makes it easier to detect patterns, trends, and outliers in groups of data.

When you use Apache Spark in Fabric, there are various built-in options to help you visualize your data, including Fabric notebook chart options, and access to popular open-source libraries.

When using a Fabric notebook, you can turn your tabular results view into a customized chart using chart options. Here, you can visualize your data without having to write any code.

Built-in visualization command - `display()` function

The Fabric built-in visualization function allows you to turn Apache Spark DataFrames, Pandas DataFrames, and SQL query results into rich format data visualizations.

You can use the `display` function on dataframes that created in PySpark and Scala on Spark DataFrames or Resilient Distributed Datasets (RDD) functions to produce the rich dataframe table view and chart view.

You can specify the row count of the dataframe being rendered. The default value is **1000**. Notebook `display` output widget supports to view and profile **10000** rows of a dataframe at most.

```

1 import pandas as pd
2 # Load data into pandas DataFrame from "/lakehouse/default/Files/churn.csv"
3 df = pd.read_csv("/lakehouse/default/Files/churn.csv")
4 display(df)
5

```

[2] ✓ 1 sec - Command executed in 1 sec 191 ms by PySpark (Python) ▾

> Log ...

Table view

	RowNumber	CustomerID	Surname	CreditScore	Country	Gender	Age	Tenure	Balans
1	1	15634602	Hargrave	619	France	Female	42	2	0.0
2	2	15647311	Hill	608	Spain	Female	41	1	83807.8
3	3	15619304	Onio	502	France	Female	42	8	159660.
4	4	15701354	Boni	699	France	Female	39	1	0.0
5	5	15737888	Mitchell	850	Spain	Female	43	2	125510.
6	6	15574012	Chu	645	Spain	Male	44	8	113755.
7	7	15592531	Bartlett	822	France	Male	50	7	0.0
8	8	15656148	Obinna	376	Germany	Female	29	4	115046.
9	9	15792365	He	501	France	Male	44	4	142051.07
10	10	15592389	H?	684	France	Male	27	2	134603.88
11	11	15767821	Bearce	528	France	Male	31	6	102016.72
12	12	15737173	Andrews	497	Spain	Male	24	3	0.0
13	13	15632264	Kay	476	France	Female	34	10	0.0
14	14	15691483	Chin	549	France	Female	25	5	0.0
15	15	15600882	Scott	635	Spain	Female	35	7	0.0
16	16	15643966	Goforth	616	Germany	Male	45	3	143129.41

You can use the filter function on the global toolbar to filter the data that mapping with your customized rule efficiently, the condition is applied to the specified column, and the filter result reflects on both table view and chart view.

```

1 import pandas as pd
2 # Load data into pandas DataFrame from "/lakehouse/default/Files/churn.csv"
3 df = pd.read_csv("/lakehouse/default/Files/churn.csv")
4 display(df)
5

```

[2] ✓ 1 sec - Command executed in 1 sec 191 ms by PySpark (Python) ▾

> Log ...

Table view

	RowNumber	CustomerID	Surname	CreditScore	Country	Gender	Age	Tenure	Balans
1	1	15634602	Hargrave	619	France	Female	42	2	0.0
2	2	15647311	Hill	608	Spain	Female	41	1	83807.8
3	3	15619304	Onio	502	France	Female	42	8	159660.
4	4	15701354	Boni	699	France	Female	39	1	0.0
5	5	15737888	Mitchell	850	Spain	Female	43	2	125510.82
6	6	15574012	Chu	645	Spain	Male	44	8	113755.78
7	7	15592531	Bartlett	822	France	Male	50	7	0.0
8	8	15656148	Obinna	376	Germany	Female	29	4	115046.74
9	9	15792365	He	501	France	Male	44	4	142051.07
10	10	15592389	H?	684	France	Male	27	2	134603.88
11	11	15767821	Bearce	528	France	Male	31	6	102016.72
12	12	15737173	Andrews	497	Spain	Male	24	3	0.0
13	13	15632264	Kay	476	France	Female	34	10	0.0
14	14	15691483	Chin	549	France	Female	25	5	0.0
15	15	15600882	Scott	635	Spain	Female	35	7	0.0
16	16	15643966	Goforth	616	Germany	Male	45	3	143129.41

The output of SQL statement adopts the same output widget with `display()` by default.

Rich dataframe table view

Free selection support on table view

Table view is rendered by default when using `display()` command. The rich dataframe preview in the notebook offers a free selection function designed to enhance the data analysis experience through flexible and intuitive selection capabilities. This feature allows users to interact with dataframes more efficiently and gain deeper insights with ease.

- **Column selection**
 - **Single column:** Click the column header to select the entire column.
 - **Multiple columns:** After selecting a single column, press and hold the 'Shift' key, then click another column header to select multiple columns.
- **Row selection**
 - **Single row:** Click on a row header to select the entire row.
 - **Multiple rows:** After selecting a single row, press and hold the 'Shift' key, then click another row header to select multiple rows.
- **Cell content preview:** Preview the content of individual cells to get a quick and detailed look at the data without the need to write additional code.
- **Column summary:** Get a summary of each column, including data distribution and key statistics, to quickly understand the characteristics of the data.
- **Free area selection:** Select any continuous segment of the table to get an overview of the total selected cells and the numeric values in the selected area.
- **Copying Selected Content:** In all selection cases, you can quickly copy the selected content using the 'Ctrl + C' shortcut. The selected data is copied in CSV format, making it easy to process in other applications.

The screenshot shows a Jupyter Notebook interface with a code cell and a data visualization area.

Code Cell:

```
1 import pandas as pd
2 # Load data into pandas DataFrame from "/lakehouse/default/files/CSV/Euro2012.csv"
3 df = pd.read_csv("/Lakehouse/default/Files/CSV/Euro2012.csv")
4 display(df)
5
```

Data View:

A table titled "Table view" displays 16 rows of data for various countries. The columns include:

	ABC Team	12L Goals	12L Shots off target	12L Total shots (inc. Blocked)	ABC % Goals-to-shots	12L Hit W
1	Croatia	4	13	12	51.9%	16.0%
2	Czech Rep...	4	13	18	41.9%	12.9%
3	Denmark	4	10	10	50.0%	20.0%
4	England	5	11	18	50.0%	17.2%
5	France	3	22	24	37.9%	6.5%
6	Germany	10	32	32	47.8%	15.6%
7	Greece	5	8	18	30.7%	19.2%
8	Italy	6	34	45	43.0%	7.5%
9	Netherlands	2	12	36	25.0%	4.1%
10	Poland	2	15	23	39.4%	5.2%
11	Portugal	6	22	42	34.3%	9.3%
12	Republic of...	1	7	12	36.8%	5.2%
13	Russia	5	9	31	22.5%	12.5%
14	Spain	12	42	33	55.9%	16.0%
15	Sweden	5	17	19	47.2%	13.8%
16	Ukraine	2	7	26	21.2%	6.0%

Inspection Tools:

- ABC Team:** Histogram showing 16 unique teams.
- 12L Goals:** Histogram showing 8 unique goals.
- 12L Shots on target:** Histogram showing 13 unique shots on target.

Data profiling support via Inspect pane

The screenshot shows a PySpark (Python) notebook interface. At the top, there is a code cell containing Python code to load a CSV file into a DataFrame:

```
1 import pandas as pd
2 # Load data into pandas DataFrame from "/lakehouse/default/Files/churn.csv"
3 df = pd.read_csv("/lakehouse/default/Files/churn.csv")
4 display(df)
5
```

The output of the cell shows a success message: "1 sec - Command executed in 1 sec 191 ms by". Below the code cell is a "Log" section.

The main area displays a "Table view" of the DataFrame. The table has 14 columns and 1000 rows. The columns are labeled: RowNumber, CustomerId, Surname, CreditScore, Country, Gender, Age, Tenure, Balance, NumOfProducts, HasC, and another unnamed column. The "Inspect" button is located on the far right of the table header.

	RowNumber	CustomerId	Surname	CreditScore	Country	Gender	Age	Tenure	Balance	NumOfProducts	HasC	
1	1	15634602	Hargrave	619	France	Female	42	2	0.0	1	1	
2	2	15647311	Hill	608	Spain	Female	41	1	83807.86	1	0	
3	3	15619304	Onio	502	France	Female	42	8	159660.8	3	1	
4	4	15701354	Boni	699	France	Female	39	1	0.0	2	0	
5	5	15737888	Mitchell	850	Spain	Female	43	2	125510.82	1	1	
6	6	15574012	Chu	645	Spain	Male	44	8	113755.78	2	1	
7	7	15592531	Bartlett	822	France	Male	50	7	0.0	2	1	
8	8	15656148	Obinna	376	Germany	Female	29	4	115046.74	4	1	
9	9	15792365	He	501	France	Male	44	4	142051.07	2	0	
10	10	15592389	H?	684	France	Male	27	2	134603.88	1	1	
11	11	15767821	Bearce	528	France	Male	31	6	102016.72	2	0	
12	12	15737173	Andrews	497	Spain	Male	24	3	0.0	2	1	
13	13	15632264	Kay	476	France	Female	34	10	0.0	2	1	
14	14	15691483	Chin	549	France	Female	25	5	0.0	2	0	
15	15	15600082	Scott	635	Spain	Female	35	7	0.0	2	1	
16	16	15643966	Goforth	616	Germany	Male	45	3	143129.41	2	0	

1. You can profile your dataframe by clicking on **Inspect** button. It provides the summarized data distribution and showing statistics of each column.
2. Each card in the "Inspect" side pane maps to a column of the dataframe, you can view more details by clicking on the card or selecting a column in the table.
3. You can view the cell details by clicking on the cell of the table. This feature is useful when the dataframe contains long string type of contents.

New rich dataframe chart view

⚠ Note

Currently, the feature is in preview.

The improved chart view is available on `display()` command. It provides a more intuitive and powerful experience for visualizing your data by using the `display()` command.

1. Now you can add up to 5 charts in one `display()` output widget by clicking **New chart**, allowing you to create multiple charts based on different columns, and compare charts easily.
2. You can get a list of chart recommendations based on the target dataframe when creating new charts. You can choose to edit a recommended chart or build your

own chart from scratch.

The screenshot shows a Data Wrangler interface. At the top, there is a code editor window containing Python code for loading a CSV file into a pandas DataFrame:

```
1 import pandas as pd
2 # Load data into pandas DataFrame from "/lakehouse/default/Files/churn.csv"
3 df = pd.read_csv("/lakehouse/default/Files/churn.csv")
4 display(df)
5
```

Below the code editor is a log message: "[2] 1 sec - Command executed in 1 sec 191 ms by". To the right of the log is the text "PySpark (Python) ▾".

Below the code editor is a table view window titled "Table view". It displays a data frame with 16 rows and 14 columns. The columns are labeled: RowNumber, CustomerId, Surname, CreditScore, Country, Gender, Age, Tenure, Balance, NumOfProducts, HasC, and another unnamed column. The data includes various customer details like names, ages, and balance amounts. On the right side of the table view, there are buttons for "Download" and "Search", and a vertical "Inspect" panel.

3. You can now customize your visualization by specifying the following settings. The setting options might change according to the selected chart type:

[Expand table](#)

Category	Basic settings	Description
	Chart type	The display function supports a wide range of chart types, including bar charts, scatter plots, line graphs, pivot table, and more.
Title	Title	The title of the chart.
Title	Subtitle	The subtitle of the chart with more descriptions.
Data	X-axis	Specify the key of the chart.
Data	Y-axis	Specify the values of the chart.
Legend	Show Legend	Enable/disable the legend.
Legend	Position	Customize the position of legend.
Other	Series group	Use this configuration to determine the groups for the aggregation.
Other	Aggregation	Use this method to aggregate data in your visualization.

Category	Basic settings	Description
Other	Stacked	Configure the display style of result.

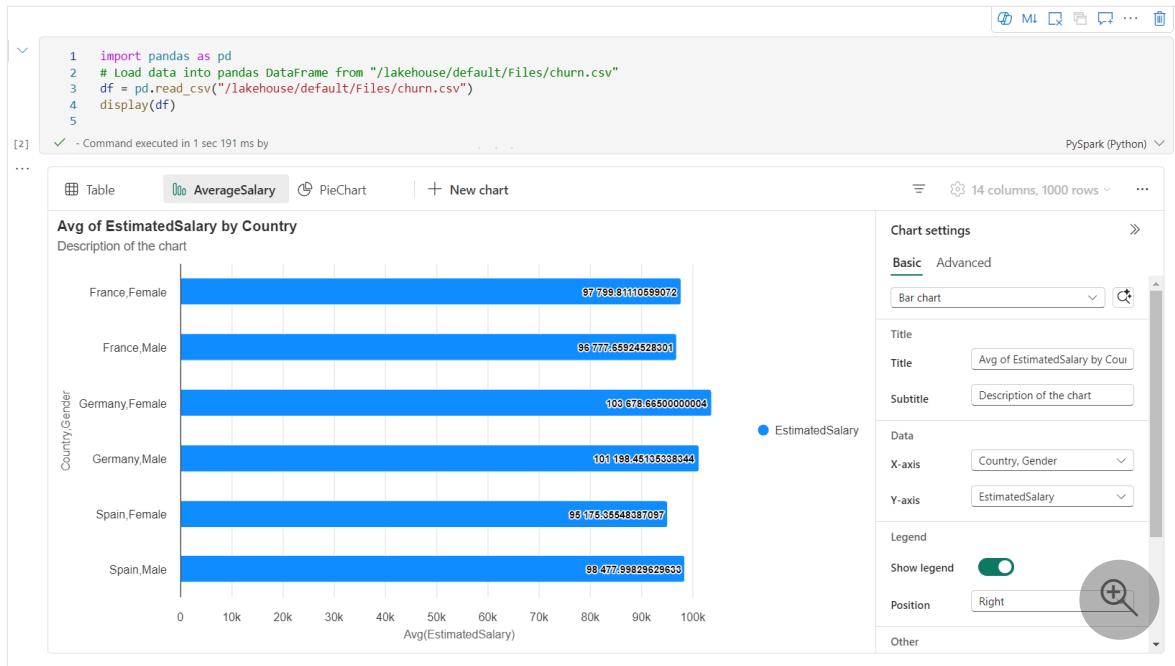
(!) Note

By default the `display(df)` function only takes the first 1,000 rows of the data to render the charts. Select **Aggregation over all results** and then select **Apply** to apply the chart generation from the whole dataframe. A Spark job is triggered when the chart setting changes. It might take several minutes to complete the calculation and render the chart.

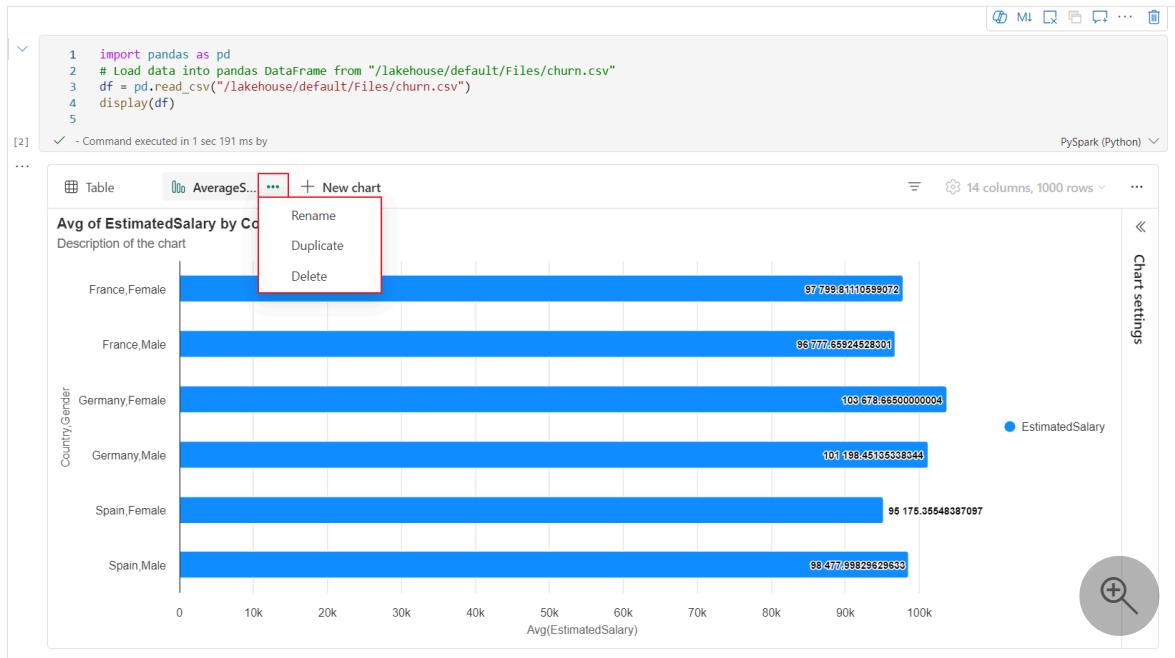
[\[+\] Expand table](#)

Category	Advanced settings	Description
Color	Theme	Define the theme color set of the chart.
X-axis	Label	Specify a label to the X-axis.
X-axis	Scale	Specify the scale function of the X-axis.
X-axis	Range	Specify the value range X-axis.
Y-axis	Label	Specify a label to the Y-axis.
Y-axis	Scale	Specify the scale function of the Y-axis.
Y-axis	Range	Specify the value range Y-axis.
Display	Show labels	Show/hide the result labels on the chart.

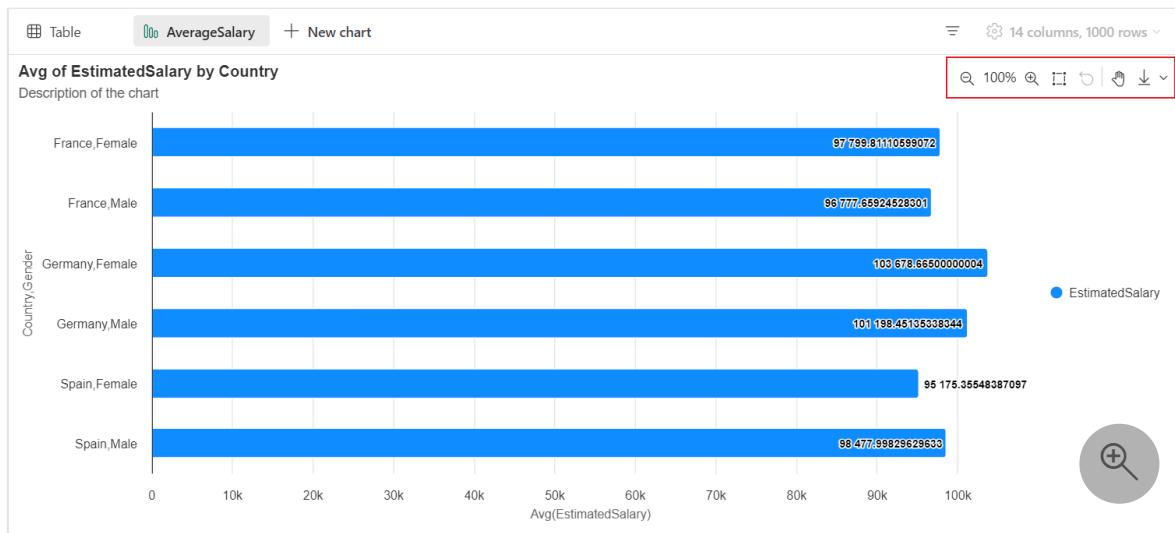
The changes of configurations take effect immediately, and all the configurations are autosaved in notebook content.



4. You can easily rename, duplicate, or delete charts in the chart tab menu.



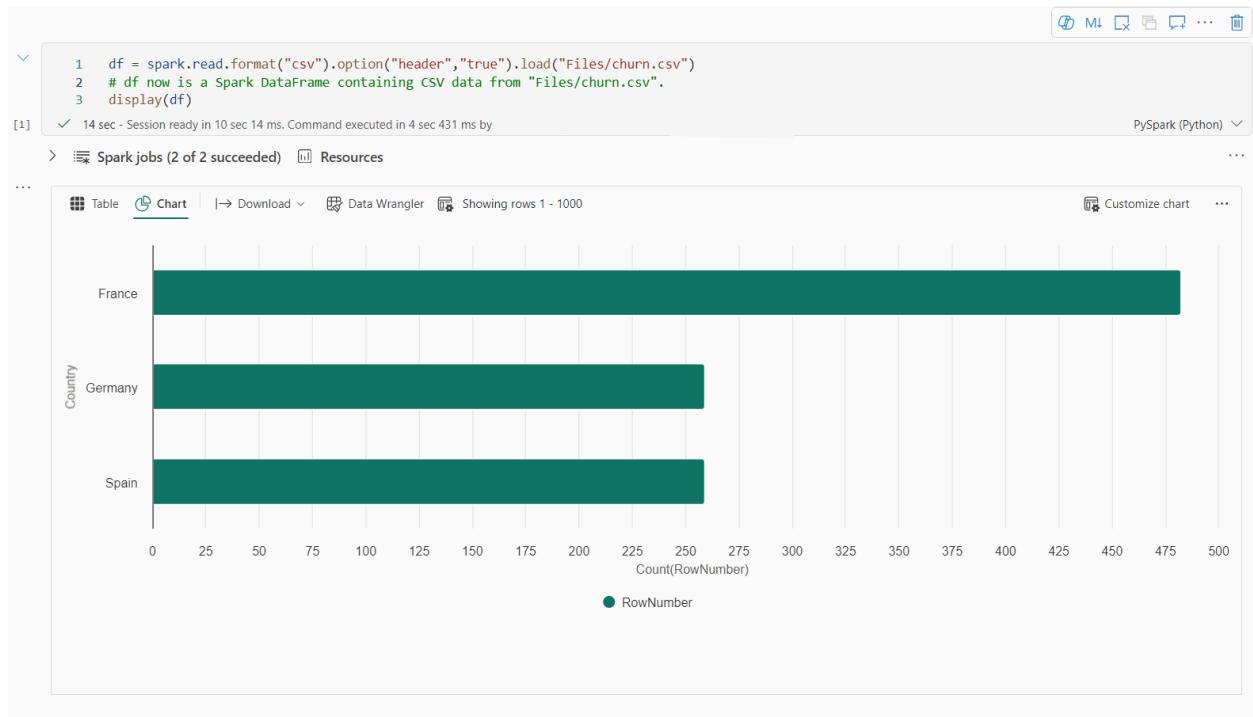
5. An interactive toolbar is available in the new chart experience when user hovers on a chart. Support operations like zoom in, zoom out, select to zoom, reset, panning, etc.



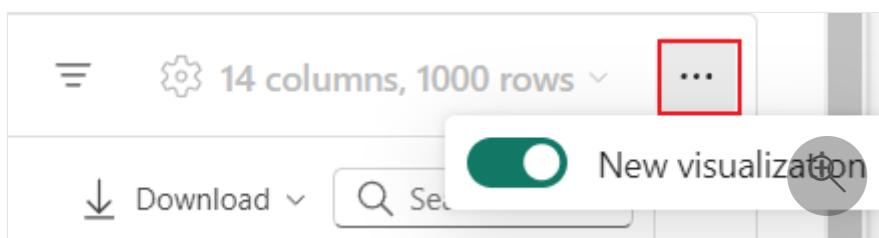
Legacy chart view

ⓘ Note

The legacy chart view will be deprecated after the new chart view finishes preview.



1. You can switch back to the legacy chart view by toggling off 'New visualization'.
The new experience is enabled by default.



- Once you have a rendered table view, switch to the **Chart** view.
- Fabric notebook automatically recommends charts based on the target dataframe, to make the chart meaningful with data insights.
- You can now customize your visualization by specifying the following values:

 Expand table

Configuration	Description
Chart type	The display function supports a wide range of chart types, including bar charts, scatter plots, line graphs, and more.
Key	Specify the range of values for the x-axis.
Value	Specify the range of values for the y-axis values.
Series group	Use this configuration to determine the groups for the aggregation.
Aggregation	Use this method to aggregate data in your visualization.

The configurations are autosaved in the Notebook output content.

 Note

By default the `display(df)` function only take the first 1,000 rows of the data to render the charts. Select **Aggregation over all results** and then select **Apply** to apply the chart generation from the whole dataframe. A Spark job is triggered when the chart setting changes. It might take several minutes to complete the calculation and render the chart.

- When the job is complete, you can view and interact with your final visualization.

display() summary view

Use `display(df, summary = true)` to check the statistics summary of a given Apache Spark DataFrame. The summary includes the column name, column type, unique values, and missing values for each column. You can also select a specific column to see its minimum value, maximum value, mean value, and standard deviation.

```

1 df = spark.read.format("csv").option("header","true").load("Files/churn.csv")
2
3 display(df, summary=True)

```

[3] ✓ 12 sec - Command executed in 11 sec 215 ms by PySpark (Python) ...

> Spark jobs (4 of 4 succeeded) Resources Log

...

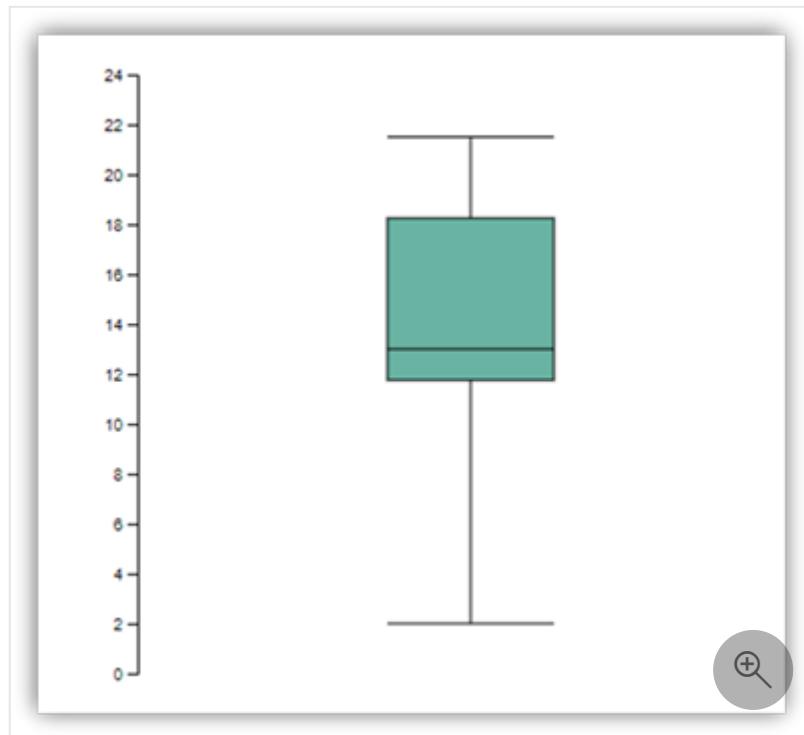
name	type	unique	missing
Balance	string	6473	0
CustomerId	string	9425	0
Country	string	3	0
Tenure	string	11	0
HasCrCard	string	2	0
EstimatedSalary	string	10000	0
NumOfProducts	string	4	0
Age	string	73	0

Select a row
Preview the statistics summary.

displayHTML() option

Fabric notebooks support HTML graphics using the *displayHTML* function.

The following image is an example of creating visualizations using [D3.js](#).



To create this visualization, run the following code.

Python

```

displayHTML("""<!DOCTYPE html>
<meta charset="utf-8">

<!-- Load d3.js -->
<script src="https://d3js.org/d3.v4.js"></script>

<!-- Create a div where the graph will take place -->
<div id="my_dataviz"></div>
<script>

// set the dimensions and margins of the graph
var margin = {top: 10, right: 30, bottom: 30, left: 40},
    width = 400 - margin.left - margin.right,
    height = 400 - margin.top - margin.bottom;

// append the svg object to the body of the page
var svg = d3.select("#my_dataviz")
.append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
.append("g")
    .attr("transform",
        "translate(" + margin.left + "," + margin.top + ")");

// Create Data
var data = [12,19,11,13,12,22,13,4,15,16,18,19,20,12,11,9]

// Compute summary statistics used for the box:
var data_sorted = data.sort(d3.ascending)
var q1 = d3.quantile(data_sorted, .25)
var median = d3.quantile(data_sorted, .5)
var q3 = d3.quantile(data_sorted, .75)
var interQuantileRange = q3 - q1
var min = q1 - 1.5 * interQuantileRange
var max = q1 + 1.5 * interQuantileRange

// Show the Y scale
var y = d3.scaleLinear()
    .domain([0,24])
    .range([height, 0]);
svg.call(d3.axisLeft(y))

// a few features for the box
var center = 200
var width = 100

// Show the main vertical line
svg
.append("line")
    .attr("x1", center)
    .attr("x2", center)
    .attr("y1", y(min) )
    .attr("y2", y(max) )
    .attr("stroke", "black")

```

```

// Show the box
svg
.append("rect")
.attr("x", center - width/2)
.attr("y", y(q3) )
.attr("height", (y(q1)-y(q3)) )
.attr("width", width )
.attr("stroke", "black")
.style("fill", "#69b3a2")

// show median, min and max horizontal lines
svg
.selectAll("toto")
.data([min, median, max])
.enter()
.append("line")
.attr("x1", center-width/2)
.attr("x2", center+width/2)
.attr("y1", function(d){ return(y(d))} )
.attr("y2", function(d){ return(y(d))} )
.attr("stroke", "black")
</script>

"""
)

```

Embed a Power BI report in a notebook

ⓘ Important

This feature is currently in PREVIEW. This information relates to a prerelease product that might be substantially modified before it reached General Available. Microsoft makes no warranties, expressed or implied, with respect to the information provided here.

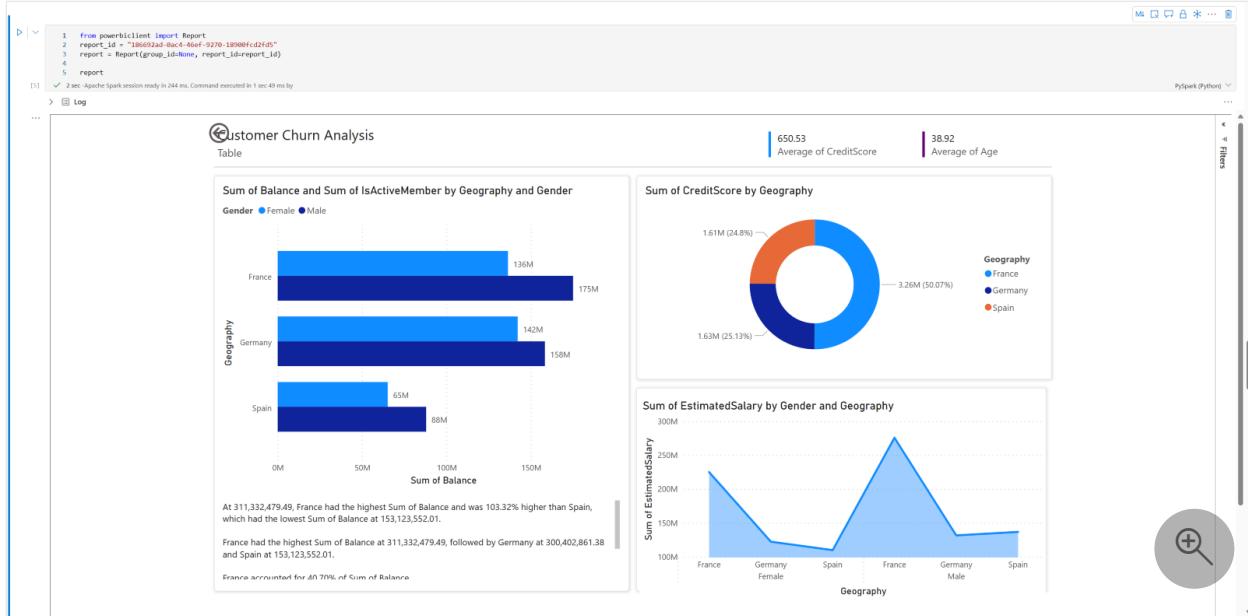
The [Powerbiclient](#) Python package is now natively supported in Fabric notebooks. You don't need to do any extra setup (like authentication process) on Fabric notebook Spark runtime 3.4. Just import `powerbiclient` and then continue your exploration. To learn more about how to use the powerbiclient package, see the [powerbiclient documentation](#).

Powerbiclient supports the following key features.

Render an existing Power BI report

You can easily embed and interact with Power BI reports in your notebooks with just a few lines of code.

The following image is an example of rendering existing Power BI report.



Run the following code to render an existing Power BI report.

Python

```
from powerbiclient import Report

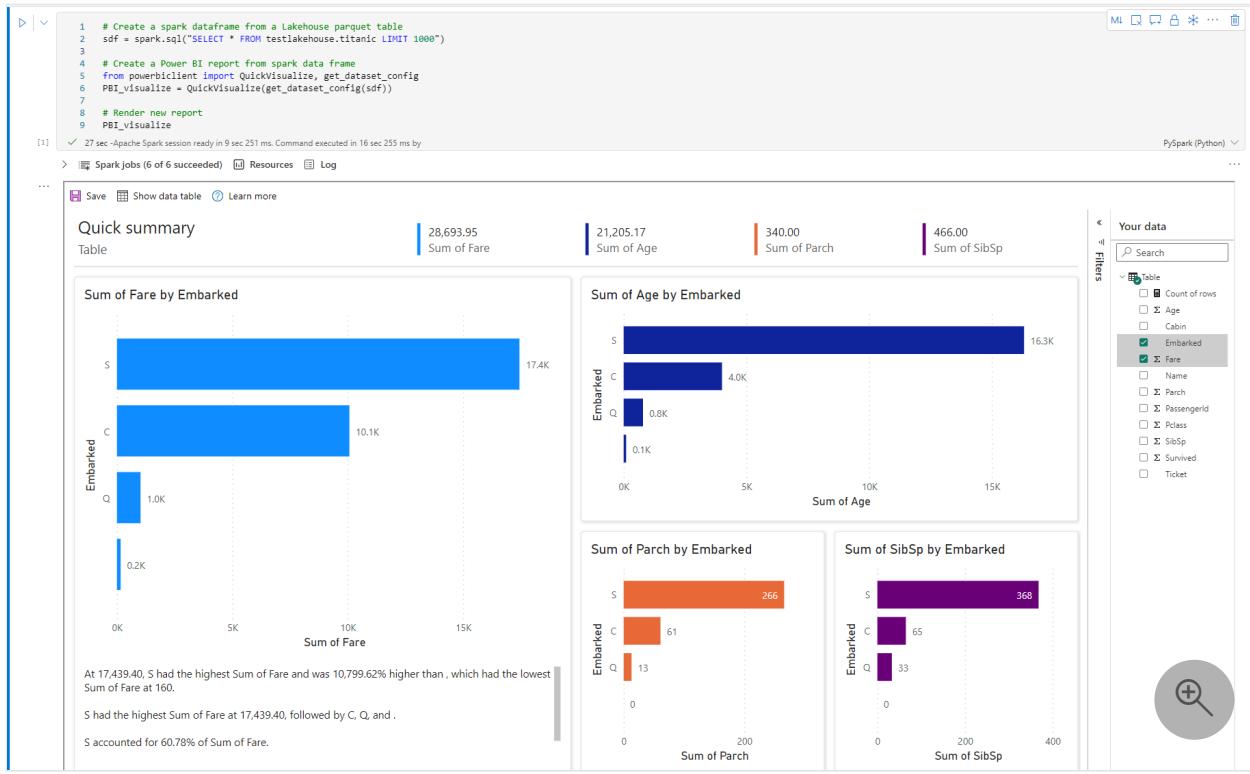
report_id="Your report id"
report = Report(group_id=None, report_id=report_id)

report
```

Create report visuals from a Spark DataFrame

You can use a Spark DataFrame in your notebook to quickly generate insightful visualizations. You can also select **Save** in the embedded report to create a report item in a target workspace.

The following image is an example of a `QuickVisualize()` from a Spark DataFrame.



Run the following code to render a report from a Spark DataFrame.

Python

```
# Create a spark dataframe from a Lakehouse parquet table
sdf = spark.sql("SELECT * FROM testlakehouse.table LIMIT 1000")

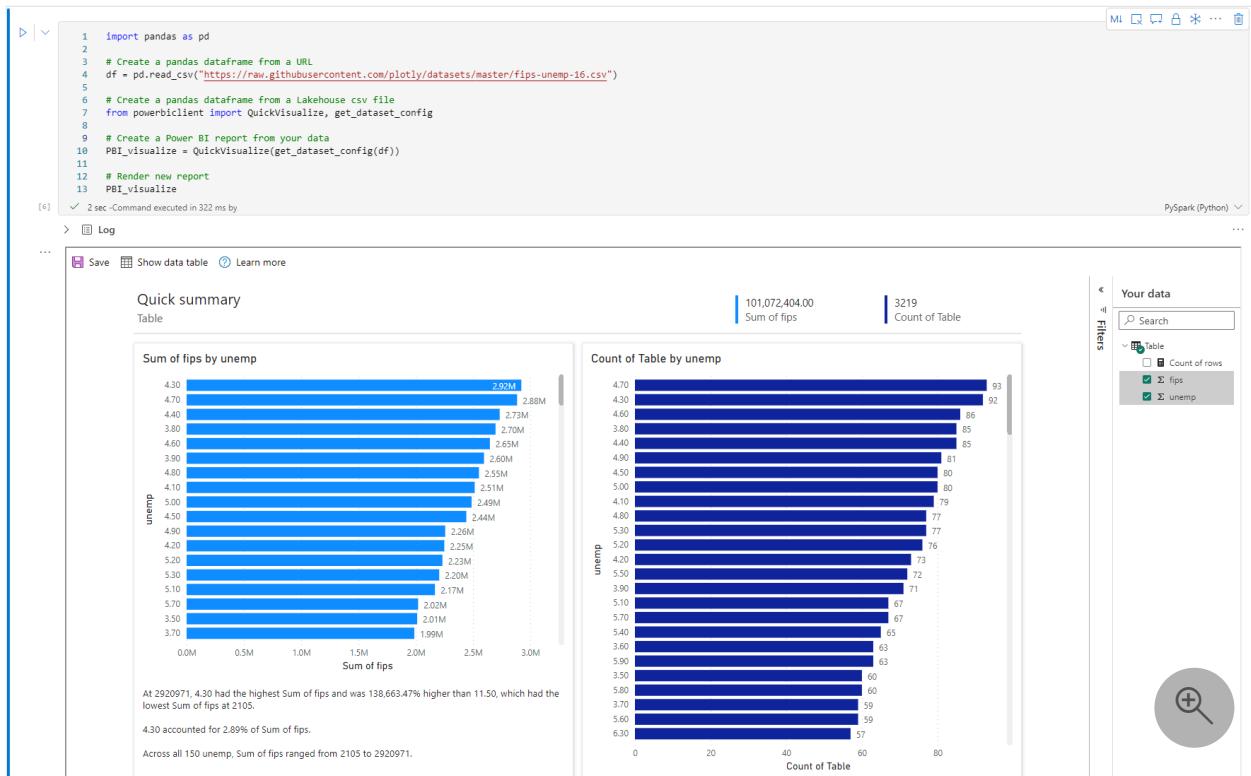
# Create a Power BI report object from spark data frame
from powerbiclient import QuickVisualize, get_dataset_config
PBI_visualize = QuickVisualize(get_dataset_config(sdf))

# Render new report
PBI_visualize
```

Create report visuals from a pandas DataFrame

You can also create reports based on a pandas DataFrame in notebook.

The following image is an example of a `QuickVisualize()` from a pandas DataFrame.



Run the following code to render a report from a Spark DataFrame.

Python

```

import pandas as pd

# Create a pandas dataframe from a URL
df =
pd.read_csv("https://raw.githubusercontent.com/plotly/datasets/master/fips-
unemp-16.csv")

# Create a pandas dataframe from a Lakehouse csv file
from powerbiclient import QuickVisualize, get_dataset_config

# Create a Power BI report object from your data
PBI_visualize = QuickVisualize(get_dataset_config(df))

# Render new report
PBI_visualize

```

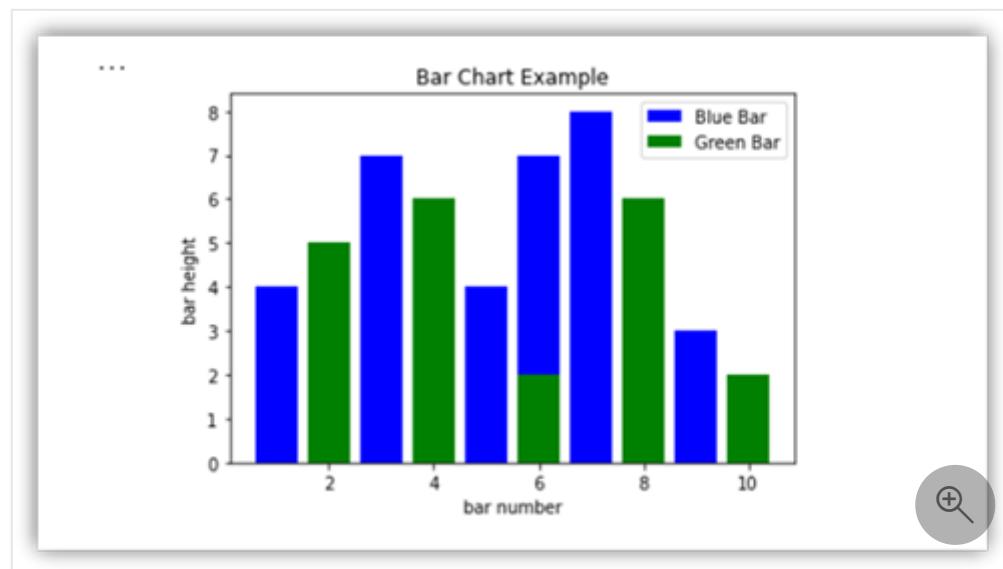
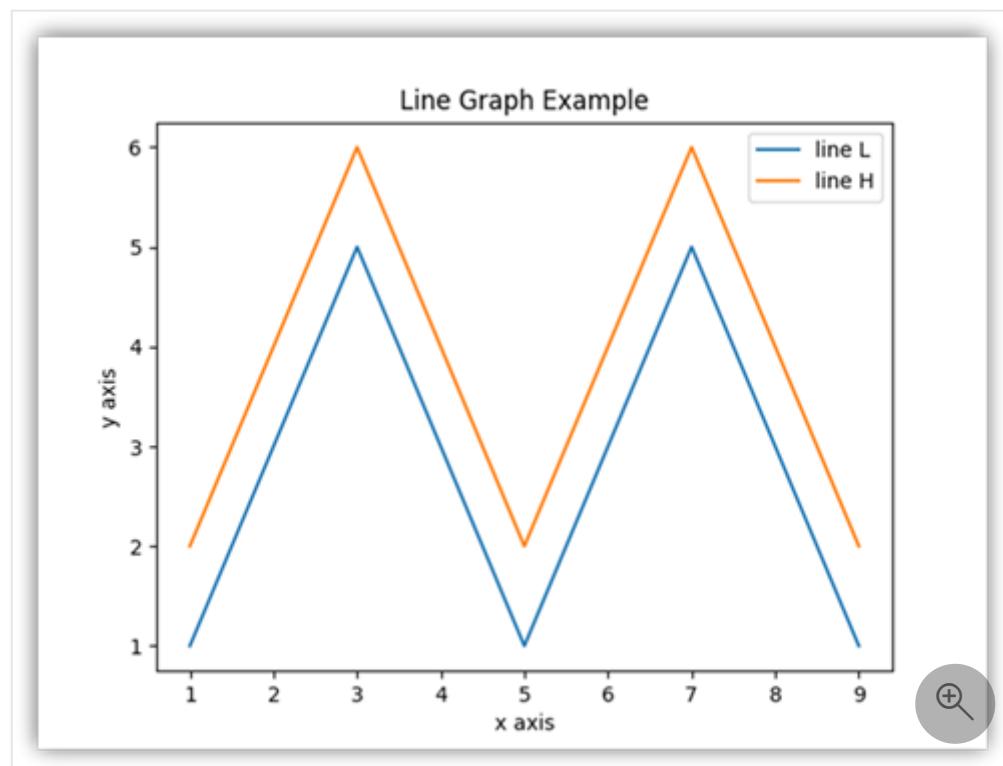
Popular libraries

When it comes to data visualization, Python offers multiple graphing libraries that come packed with many different features. By default, every Apache Spark pool in Fabric contains a set of curated and popular open-source libraries.

Matplotlib

You can render standard plotting libraries, like Matplotlib, using the built-in rendering functions for each library.

The following image is an example of creating a bar chart using **Matplotlib**.



Run the following sample code to draw this bar chart.

Python

```
# Bar chart

import matplotlib.pyplot as plt

x1 = [1, 3, 4, 5, 6, 7, 9]
y1 = [4, 7, 2, 4, 7, 8, 3]
```

```

x2 = [2, 4, 6, 8, 10]
y2 = [5, 6, 2, 6, 2]

plt.bar(x1, y1, label="Blue Bar", color='b')
plt.bar(x2, y2, label="Green Bar", color='g')
plt.plot()

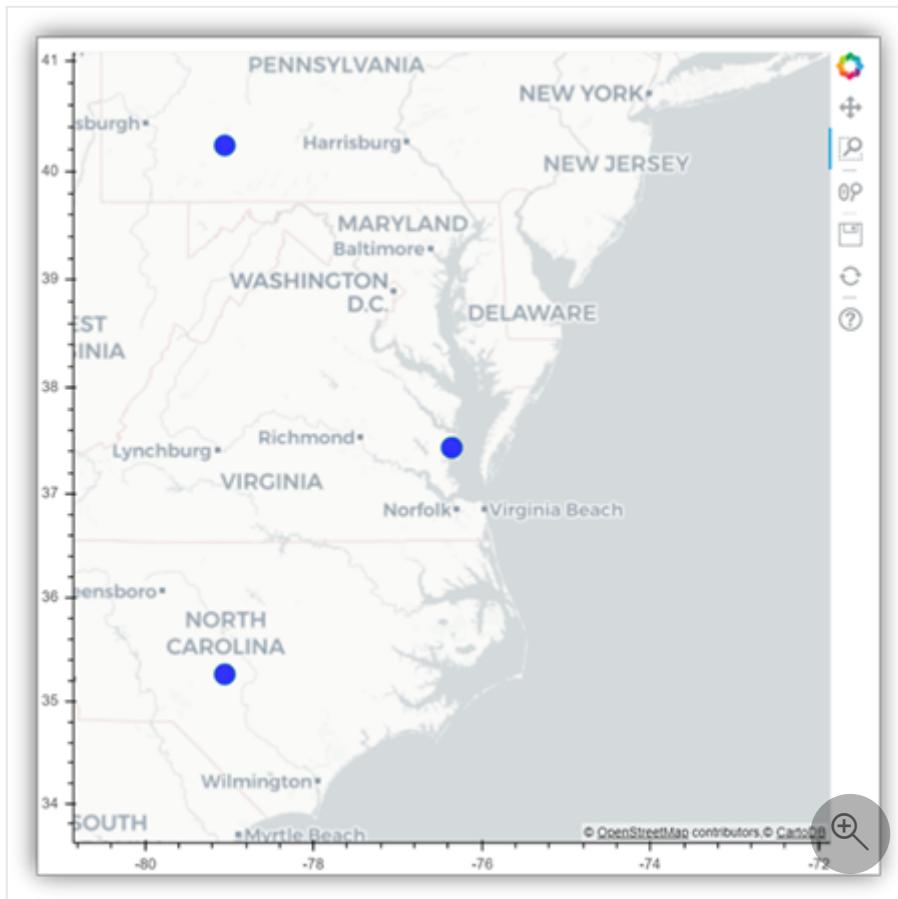
plt.xlabel("bar number")
plt.ylabel("bar height")
plt.title("Bar Chart Example")
plt.legend()
plt.show()

```

Bokeh

You can render HTML or interactive libraries, like **bokeh**, using the *displayHTML(df)*.

The following image is an example of plotting glyphs over a map using **bokeh**.



To draw this image, run the following sample code.

Python

```

from bokeh.plotting import figure, output_file
from bokeh.tile_providers import get_provider, Vendors
from bokeh.embed import file_html
from bokeh.resources import CDN

```

```

from bokeh.models import ColumnDataSource

tile_provider = get_provider(Vendors.CARTODBPOSITRON)

# range bounds supplied in web mercator coordinates
p = figure(x_range=(-9000000,-8000000), y_range=(4000000,5000000),
            x_axis_type="mercator", y_axis_type="mercator")
p.add_tile(tile_provider)

# plot datapoints on the map
source = ColumnDataSource(
    data=dict(x=[ -8800000, -8500000 , -8800000],
              y=[4200000, 4500000, 4900000])
)

p.circle(x="x", y="y", size=15, fill_color="blue", fill_alpha=0.8,
          source=source)

# create an html document that embeds the Bokeh plot
html = file_html(p, CDN, "my plot1")

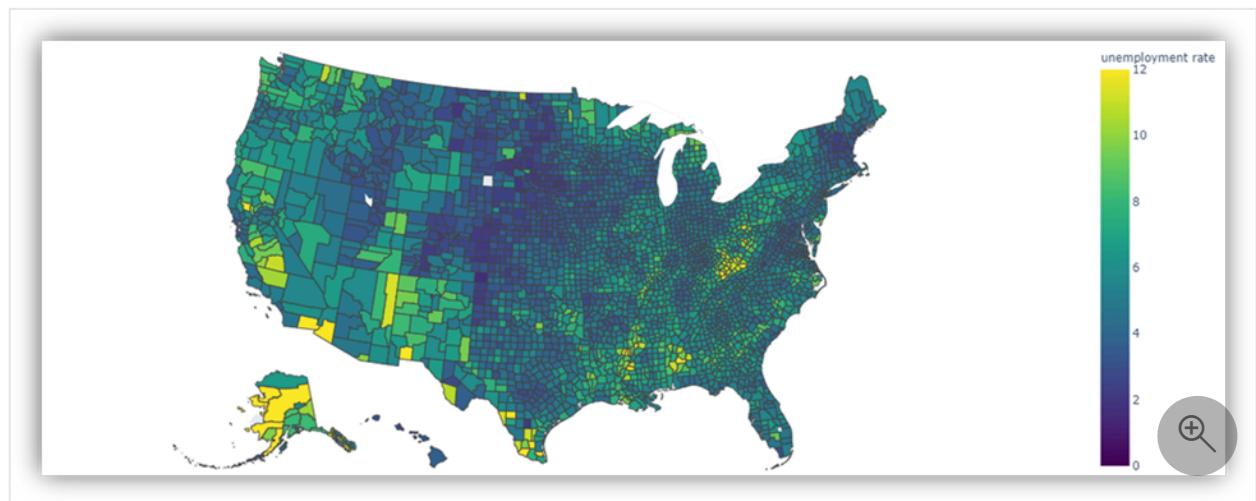
# display this html
displayHTML(html)

```

Plotly

You can render HTML or interactive libraries like **Plotly**, using the `displayHTML()`.

To draw this image, run the following sample code.



Python

```

from urllib.request import urlopen
import json
with
urlopen('https://raw.githubusercontent.com/plotly/datasets/master/geojson-
counties-fips.json') as response:

```

```

counties = json.load(response)

import pandas as pd
df =
pd.read_csv("https://raw.githubusercontent.com/plotly/datasets/master/fips-
unemp-16.csv",
            dtype={"fips": str})

import plotly
import plotly.express as px

fig = px.choropleth(df, geojson=counties, locations='fips', color='unemp',
                     color_continuous_scale="Viridis",
                     range_color=(0, 12),
                     scope="usa",
                     labels={'unemp': 'unemployment rate'}
                    )
fig.update_layout(margin={"r":0, "t":0, "l":0, "b":0})

# create an html document that embeds the Plotly plot
h = plotly.offline.plot(fig, output_type='div')

# display this html
displayHTML(h)

```

Pandas

You can view HTML output of pandas DataFrames as the default output. Fabric notebooks automatically show the styled HTML content.

Model:	Decision Tree		Regression		Random	
Predicted:	Tumour	Non-Tumour	Tumour	Non-Tumour	Tumour	Non-Tumour
Actual Label:						
Tumour (Positive)	38.0	2.0	18.0	22.0	21	NaN
Non-Tumour (Negative)	19.0	439.0	6.0	452.0	226	232.0



Python

```

import pandas as pd
import numpy as np

df = pd.DataFrame([[38.0, 2.0, 18.0, 22.0, 21, np.nan],[19, 439, 6, 452,
226,232]],

index=pd.Index(['Tumour (Positive)', 'Non-Tumour

```

```
(Negative)'], name='Actual Label'),  
  
        columns=pd.MultiIndex.from_product([[ 'Decision Tree',  
'Regression', 'Random'],[ 'Tumour', 'Non-Tumour']], names=[ 'Model:',  
'Predicted:']))  
  
df
```

Related content

- Explore the data in your lakehouse with a notebook
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Explore the data in your lakehouse with a notebook

Article • 11/29/2023

In this tutorial, learn how to explore the data in your lakehouse with a notebook.

Prerequisites

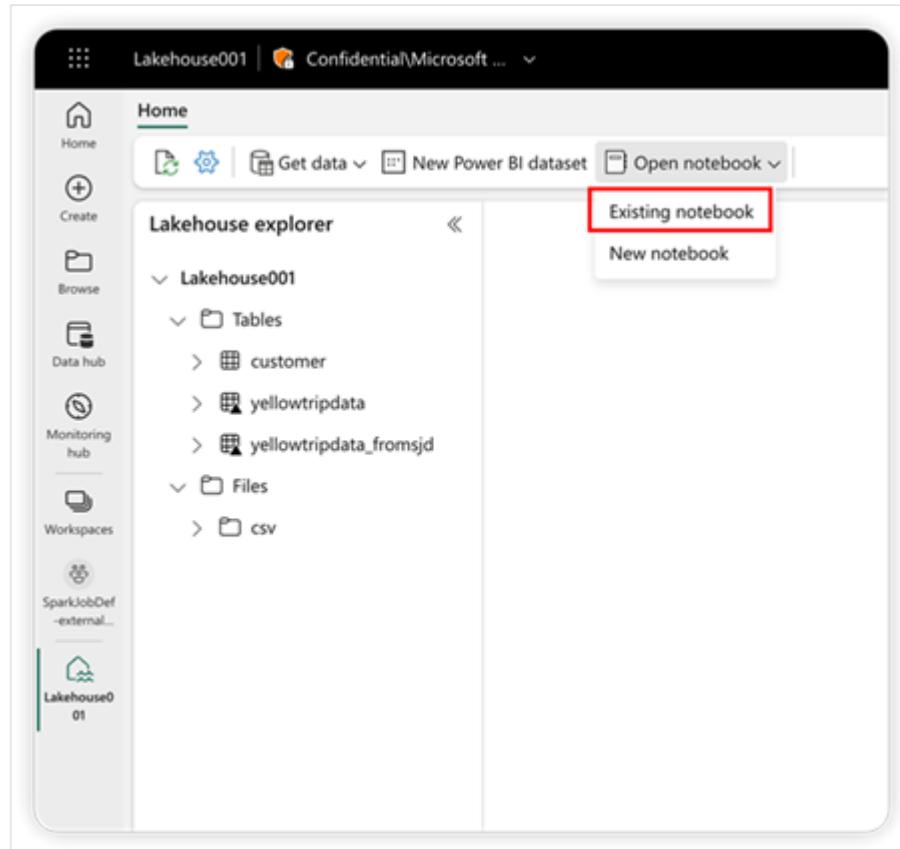
To get started, you need the following prerequisites:

- A Microsoft Fabric tenant account with an active subscription. [Create an account for free](#).
- Read the [Lakehouse overview](#).

Open or create a notebook from a lakehouse

To explore your lakehouse data, you can add the lakehouse to an existing notebook or create a new notebook from the lakehouse.

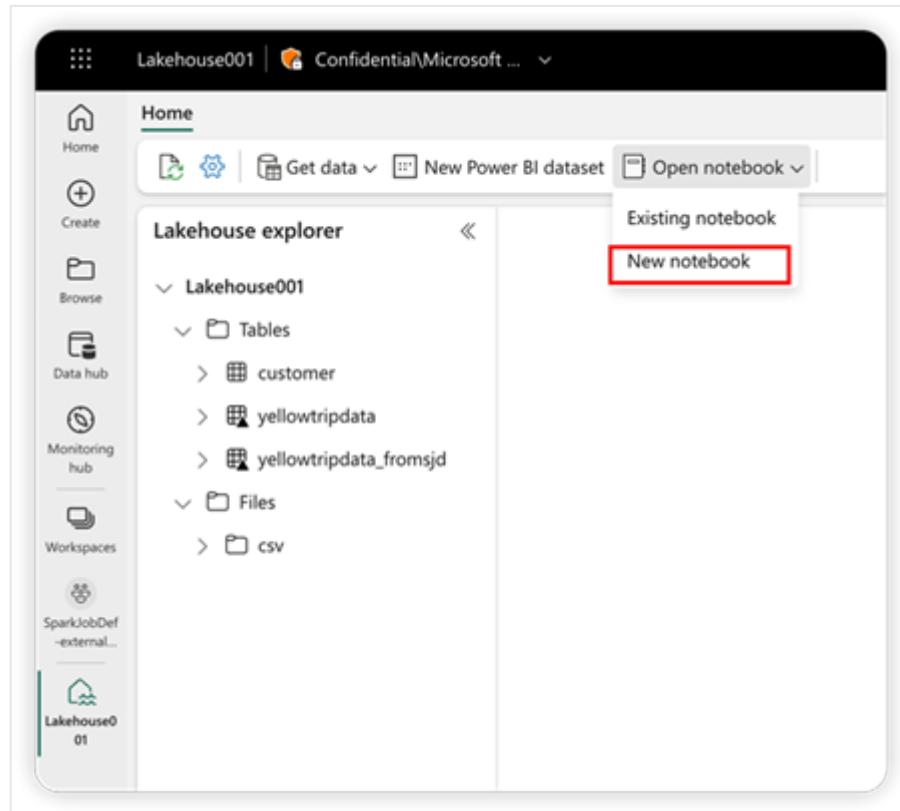
Open a lakehouse from an existing notebook



Select the notebook from the notebook list and then select **Add**. The notebook opens with your current lakehouse added to the notebook.

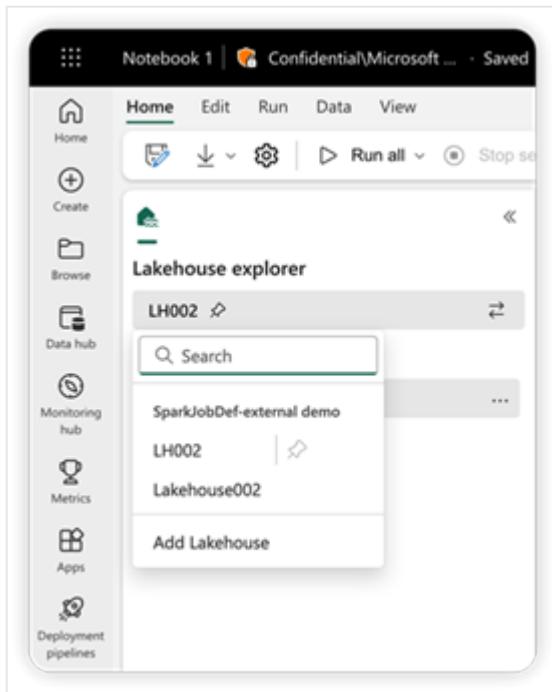
Open a lakehouse from a new notebook

You can create a new notebook in the same workspace and the current lakehouse appears in that notebook.



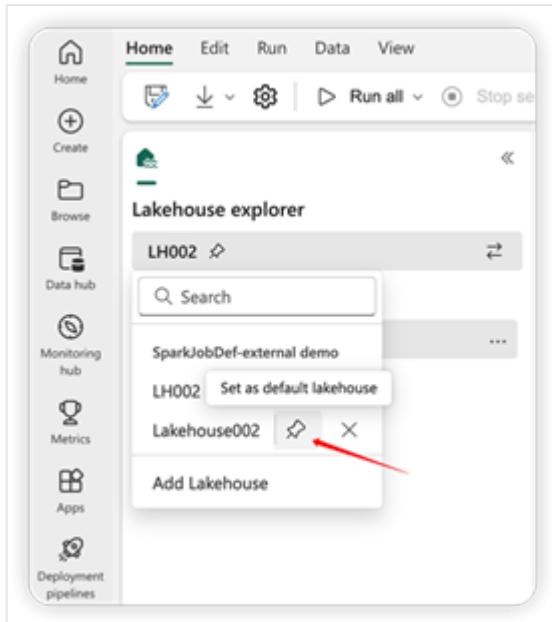
Switch lakehouses and set a default

You can add multiple lakehouses to the same notebook. By switching the available lakehouse in the left panel, you can explore the structure and the data from different lakehouses.



In the lakehouse list, the pin icon next to the name of a lakehouse indicates that it's the default lakehouse in your current notebook. In the notebook code, if only a relative path is provided to access the data from the Microsoft Fabric OneLake, then the default lakehouse is served as the root folder at run time.

To switch to a different default lakehouse, move the pin icon.



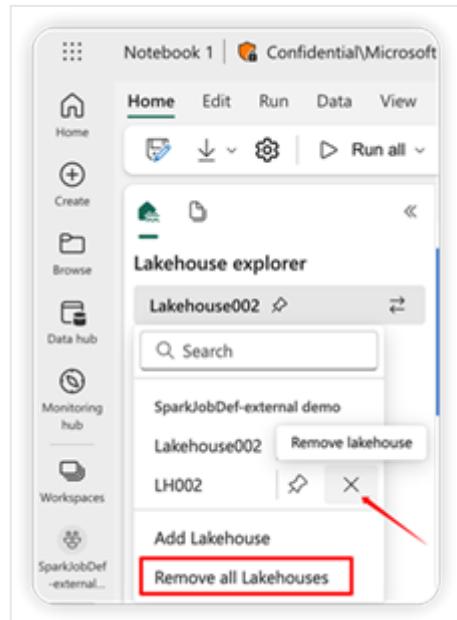
ⓘ Note

The default lakehouse decide which Hive metastore to use when running the notebook with Spark SQL. If multiple lakehouse are added into the notebook, make sure when Spark SQL is used, the target lakehouse and the current default lakehouse are from the same workspace.

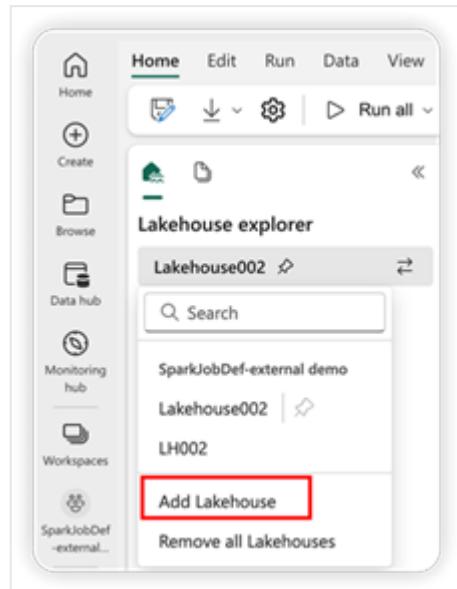
Add or remove a lakehouse

Selecting the X icon next to a lakehouse name removes it from the notebook, but the lakehouse item still exists in the workspace.

To remove all the lakehouses from the notebook, click "Remove all Lakehouses" in the lakehouse list.



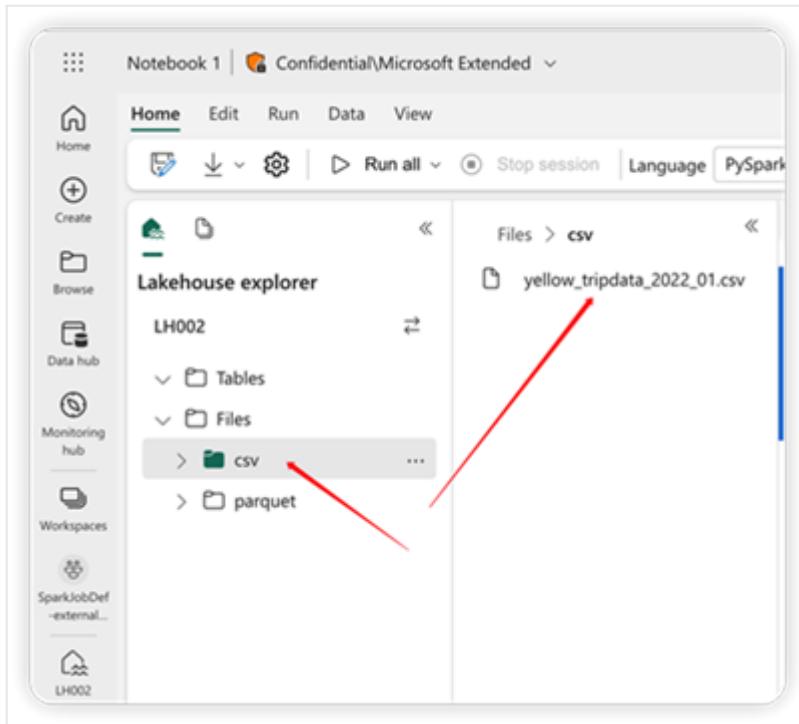
Select **Add lakehouse** to add more lakehouses to the notebook. You can either add an existing one or create a new one.



Explore the lakehouse data

The structure of the Lakehouse shown in the Notebook is the same as the one in the Lakehouse view. For the detail please check [Lakehouse overview](#). When you select a file

or folder, the content area shows the details of the selected item.



ⓘ Note

The notebook will be created under your current workspace.

Related content

- [How to use a notebook to load data into your lakehouse](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Use a notebook to load data into your lakehouse

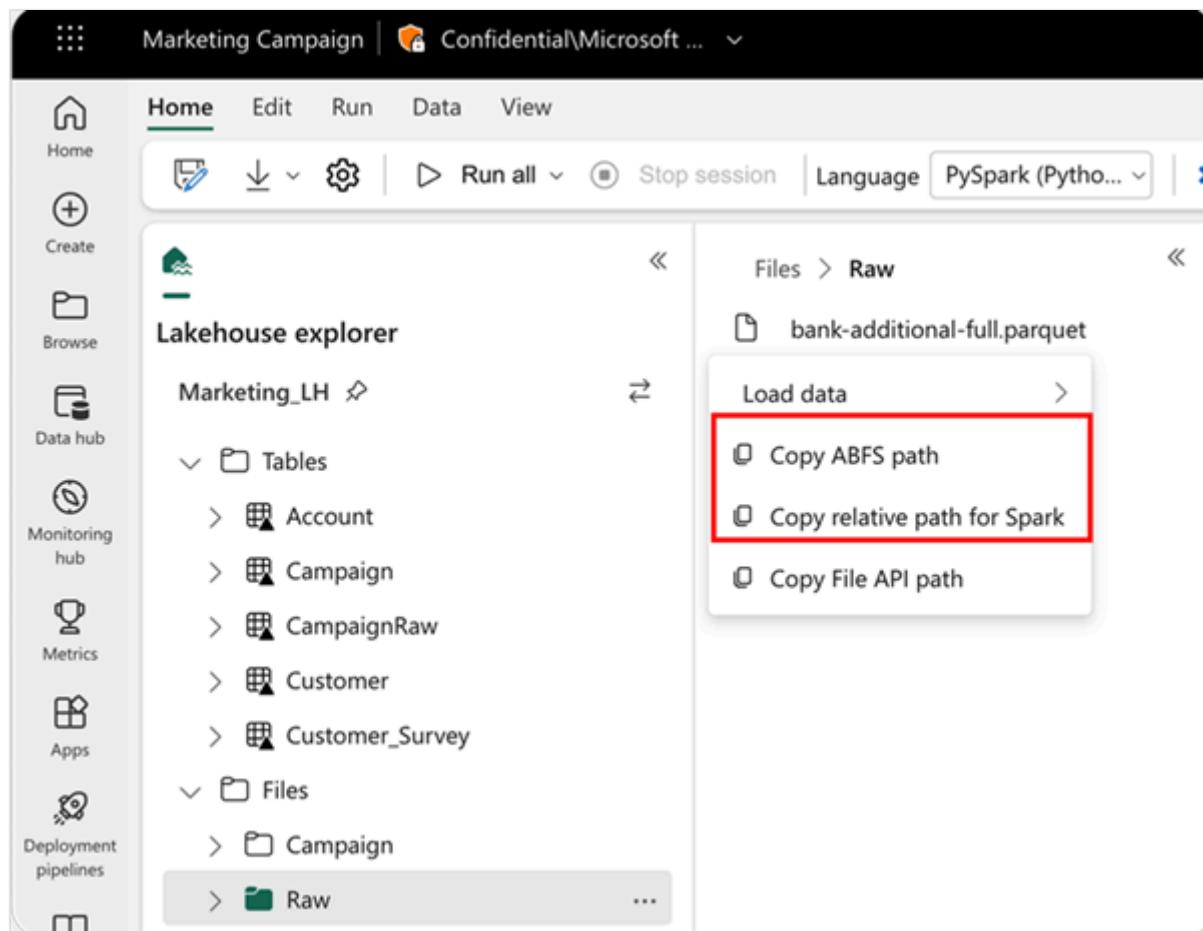
Article • 05/24/2024

In this tutorial, learn how to read/write data into your Fabric lakehouse with a notebook. Fabric supports Spark API and Pandas API are to achieve this goal.

Load data with an Apache Spark API

In the code cell of the notebook, use the following code example to read data from the source and load it into **Files**, **Tables**, or both sections of your lakehouse.

To specify the location to read from, you can use the relative path if the data is from the default lakehouse of your current notebook. Or, if the data is from a different lakehouse, you can use the absolute Azure Blob File System (ABFS) path. Copy this path from the context menu of the data.



Copy ABFS path: This option returns the absolute path of the file.

Copy relative path for Spark: This option returns the relative path of the file in your default lakehouse.

Python

```
df = spark.read.parquet("location to read from")

# Keep it if you want to save dataframe as CSV files to Files section of the
default lakehouse

df.write.mode("overwrite").format("csv").save("Files/ " + csv_table_name)

# Keep it if you want to save dataframe as Parquet files to Files section of
the default lakehouse

df.write.mode("overwrite").format("parquet").save("Files/" +
parquet_table_name)

# Keep it if you want to save dataframe as a delta lake, parquet table to
Tables section of the default lakehouse

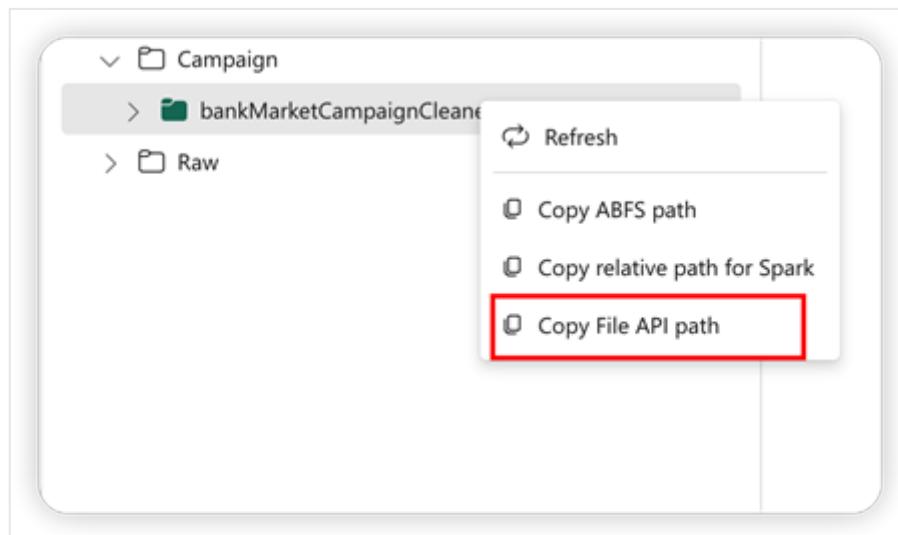
df.write.mode("overwrite").format("delta").saveAsTable(delta_table_name)

# Keep it if you want to save the dataframe as a delta lake, appending the
data to an existing table

df.write.mode("append").format("delta").saveAsTable(delta_table_name)
```

Load data with Pandas API

To support Pandas API, the default lakehouse is automatically mounted to the notebook. The mount point is '/lakehouse/default/'. You can use this mount point to read/write data from/to the default lakehouse. The "Copy File API Path" option from the context menu returns the File API path from that mount point. The path returned from the option **Copy ABFS path** also works for Pandas API.



Copy File API Path: This option returns the path under the mount point of the default lakehouse.

Python

```
# Keep it if you want to read parquet file with Pandas from the default
# lakehouse mount point

import pandas as pd
df = pd.read_parquet("/lakehouse/default/Files/sample.parquet")

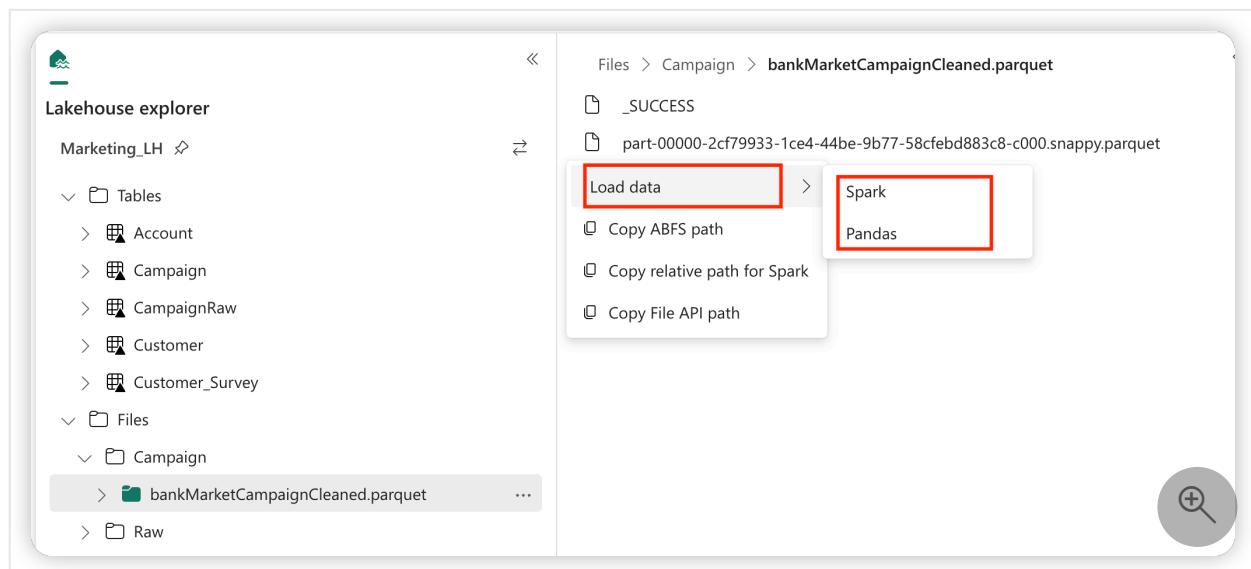
# Keep it if you want to read parquet file with Pandas from the absolute
# abfss path

import pandas as pd
df = pd.read_parquet("abfss://DevExpBuildDemo@msit-
onelake.dfs.fabric.microsoft.com/Marketing_LH.Lakehouse/Files/sample.parquet
")
```

Tip

For Spark API, please use the option of **Copy ABFS path** or **Copy relative path for Spark** to get the path of the file. For Pandas API, please use the option of **Copy ABFS path** or **Copy File API path** to get the path of the file.

The quickest way to have the code to work with Spark API or Pandas API is to use the option of **Load data** and select the API you want to use. The code is automatically generated in a new code cell of the notebook.



Related content

- Explore the data in your lakehouse with a notebook
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Notebook source control and deployment

Article • 10/08/2024

This article explains how Git integration and deployment pipelines work for notebooks in Microsoft Fabric. Learn how to set up a connection to your repository, manage your notebooks, and deploy them across different environments.

Notebook Git integration

Fabric notebooks offer Git integration for source control with Azure DevOps. With Git integration, you can back up and version your notebook, revert to previous stages as needed, collaborate or work alone using Git branches, and manage your notebook content lifecycle entirely within Fabric.

ⓘ Note

Start from October 2024, Notebook git integration supports persisting the mapping relationship of the attached Environment when syncing to new workspace, which means when you commit the notebook and attached environment together to git repo, and sync it to another workspace, the newly generated notebook and environment will be bound together. This upgrade will have impact to existing Notebooks and dependent Environments that are versioned in git, the **Physical id** of attached environment in notebook metadata content will be replaced with an **Logical id**, the change will get reflected on the diff view.

Set up a connection

From your workspace settings, you can easily set up a connection to your repo to commit and sync changes. To set up the connection, see [Get started with Git integration](#). Once connected, your items, including notebooks, appear in the **Source control** panel.

The screenshot shows the Microsoft Azure DevOps Source control interface for a project named "Git Integration Demo".

At the top, there are navigation links: "Create deployment pipeline", "Create app", "Manage access", and "Workspace settings". Below the navigation is a toolbar with buttons for "New item", "New folder (preview)", "Import", "Source control" (highlighted in red), "Filter by keyword", "Filter", and "Workspace settings".

The main area features a large circular icon with a plus sign and three stacked squares, followed by the text "Choose from predesigned task flows or add a task to build one (preview)". A note below it says "Select from one of Microsoft's predesigned task flows or add a task to start building one yourself." There are two buttons: "Select a predesigned task flow" and "Add a task".

A table lists items:

Name	Git status	Type	Task	Owner	Refreshed	Next refresh	Endorser
environment	Synced	Environment	—	—	—	—	—
Notebook 1	Uncommitted	Notebook	—	—	—	—	—

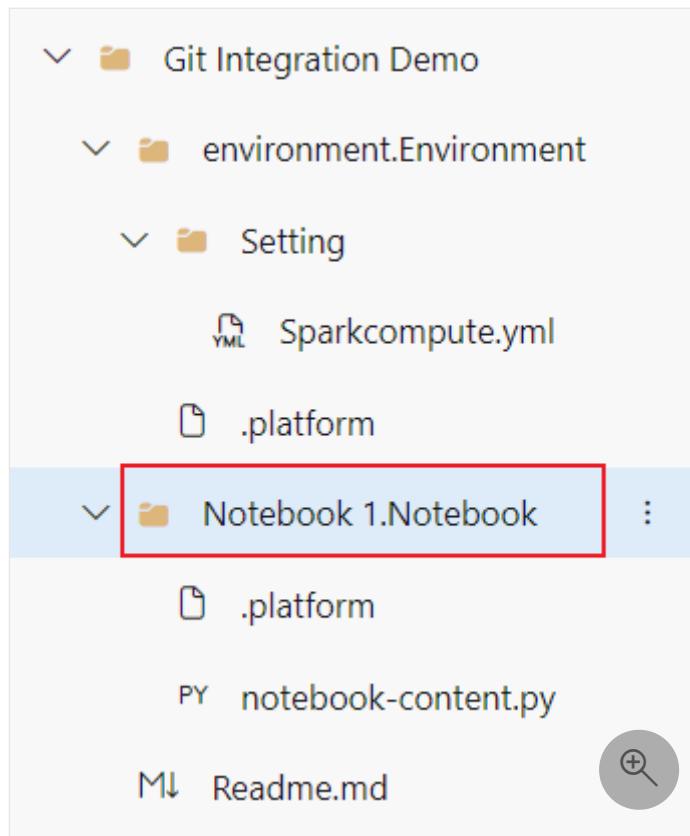
To the right, a sidebar titled "Source control" shows the "Current branch: NotebookGitDemo". It has tabs for "Changes" (1) and "Updates". An optional commit message field is present. Below the table, a commit history shows "Notebook 1". At the bottom right are "Commit" and "Undo" buttons.

After you successfully commit the notebook instances to the Git repo, you see the notebook folder structure in the repo.

You can now execute future operations, like [Create pull request](#).

Notebook representation in Git

The following image is an example of the file structure of each notebook item in the repo:



When you commit the notebook item to the Git repo, the notebook code is converted to a source code format, instead of a standard .ipynb file. For example, a PySpark notebook converts to a notebook-content.py file. This approach allows for easier code reviews using built-in diff features.

In the item content source file, metadata (including the default lakehouse and attached environment), markdown cells, and code cells are preserved and distinguished. This approach supports a precise recovery when you sync back to a Fabric workspace.

Notebook cell output isn't included when syncing to Git.

```
1 # Fabric notebook source
2
3 # METADATA ****
4
5 # META {
6 # META   "kernel_info": {
7 # META     "name": "synapse_pyspark"
8 # META   },
9 # META   "dependencies": {
10 # META     "lakehouse": {
11 # META       "default_lakehouse": "91f8a371-9467-436e-9082-777458f8ae6f",
12 # META       "default_lakehouse_name": "Sample_lakehouse",
13 # META       "default_lakehouse_workspace_id": "a08102bf-6da4-46d0-9ed4-995e6a0ccb7c"
14 # META     },
15 # META   "environment": {
16 # META     "environmentId": "d1fc11bd-b778-439f-b337-243e52c1ab6c",
17 # META     "workspaceId": "00000000-0000-0000-0000-000000000000"
18 # META   }
19 # META }
20 # META }
21
22 # CELL ****
23
24 df = spark.read.parquet("Files/green_tripdata_2022-08.parquet")
25 # df now is a Spark DataFrame containing parquet data from "Files/green_tripdata_2022-08.parquet".
26 display(df)
27
```

Note

- Currently, files in **Notebook resources** aren't committed to the repo. Committing these files is supported in an upcoming release.
- We recommend you to manage the notebooks and their dependent environment in the same workspace, and use git to version control both notebook and [environment](#) items, Fabric Git system will handle the mapping relationship when syncing the notebook and attached environment to new workspaces.
- The default lakehouse ID persists in the notebook when you sync from the repo to a Fabric workspace. If you commit a notebook with the default lakehouse, you must refer a newly created lakehouse item manually. For more information, see [Lakehouse Git integration](#).

Notebook in deployment pipelines

You can also use Deployment pipeline to deploy your notebook code across different environments, such as development, test, and production. This feature can enable you to streamline your development process, ensure quality and consistency, and reduce manual errors with lightweight low-code operations. You can also use deployment rules to customize the behavior of your notebooks when they're deployed, such as changing the default lakehouse of a notebook.

Note

- You are using the new design of deployment pipeline now, the old UI can be accessed by turning off 'New Deployment pipeline'.
- Start from October, Fabric notebook supports auto-binding feature that will bind the default lakehouse and attached environment within the same workspace when deploying to next stage. The change will have impacts to existing notebooks in deployment pipeline.
 - The default lakehouse and attached environment (when all dependent items are in the same workspace) will be replaced by newly generated items in target workspace, the notebook metadata change will be highlighted in the diff view in next round of deployment.

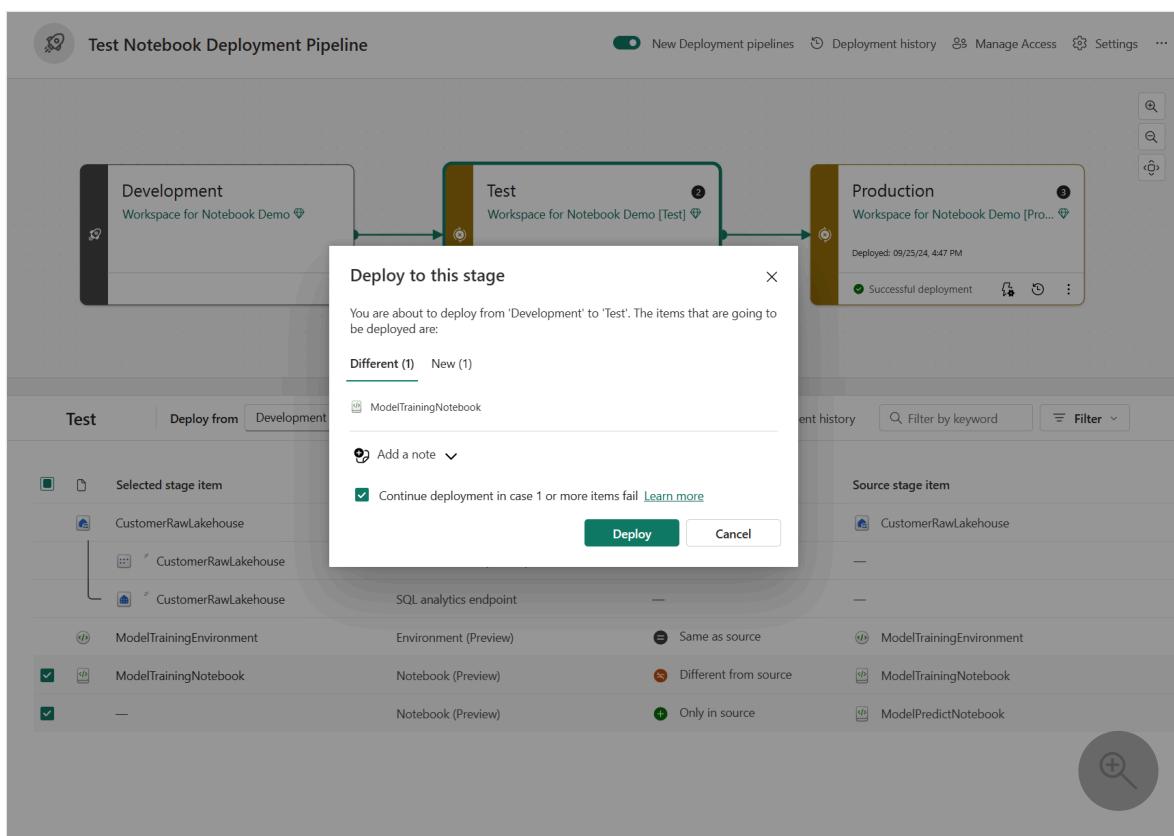
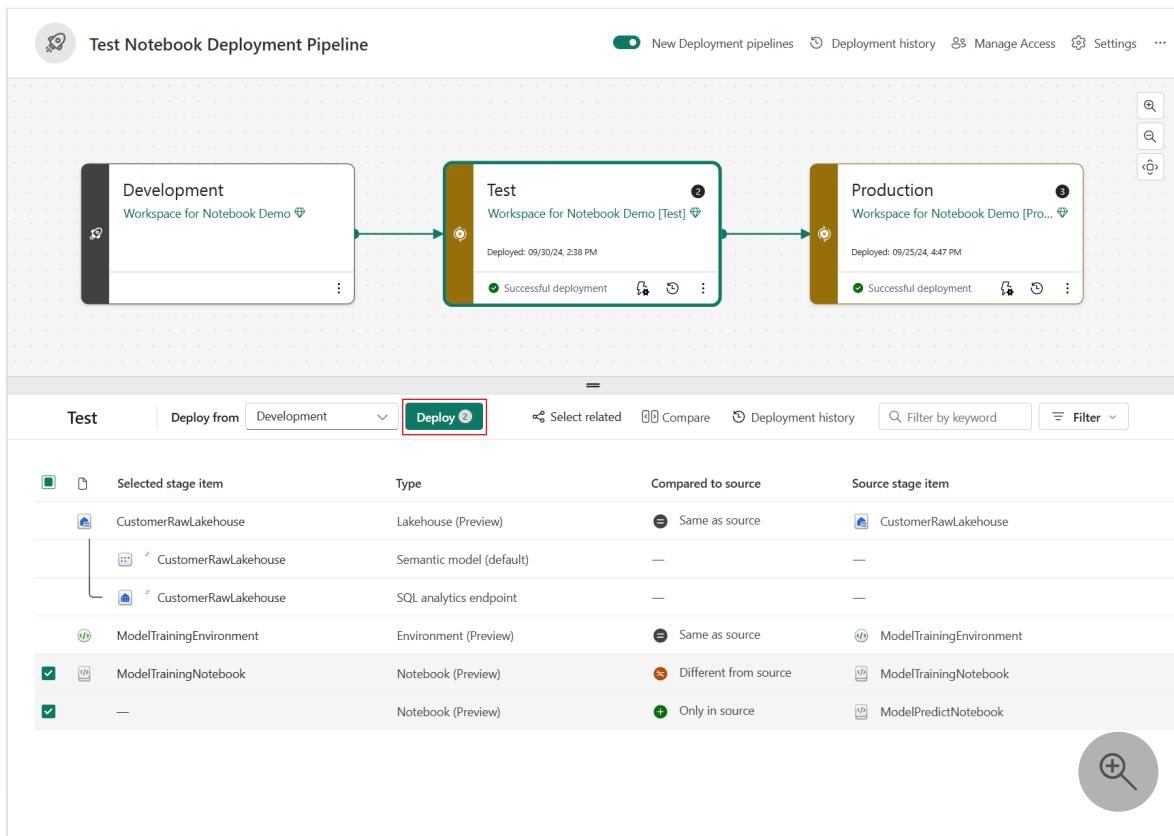
- You can set deployment rules for default lakehouse to override the auto-bound lakehouse.

Use the following steps to complete your notebook deployment using the deployment pipeline.

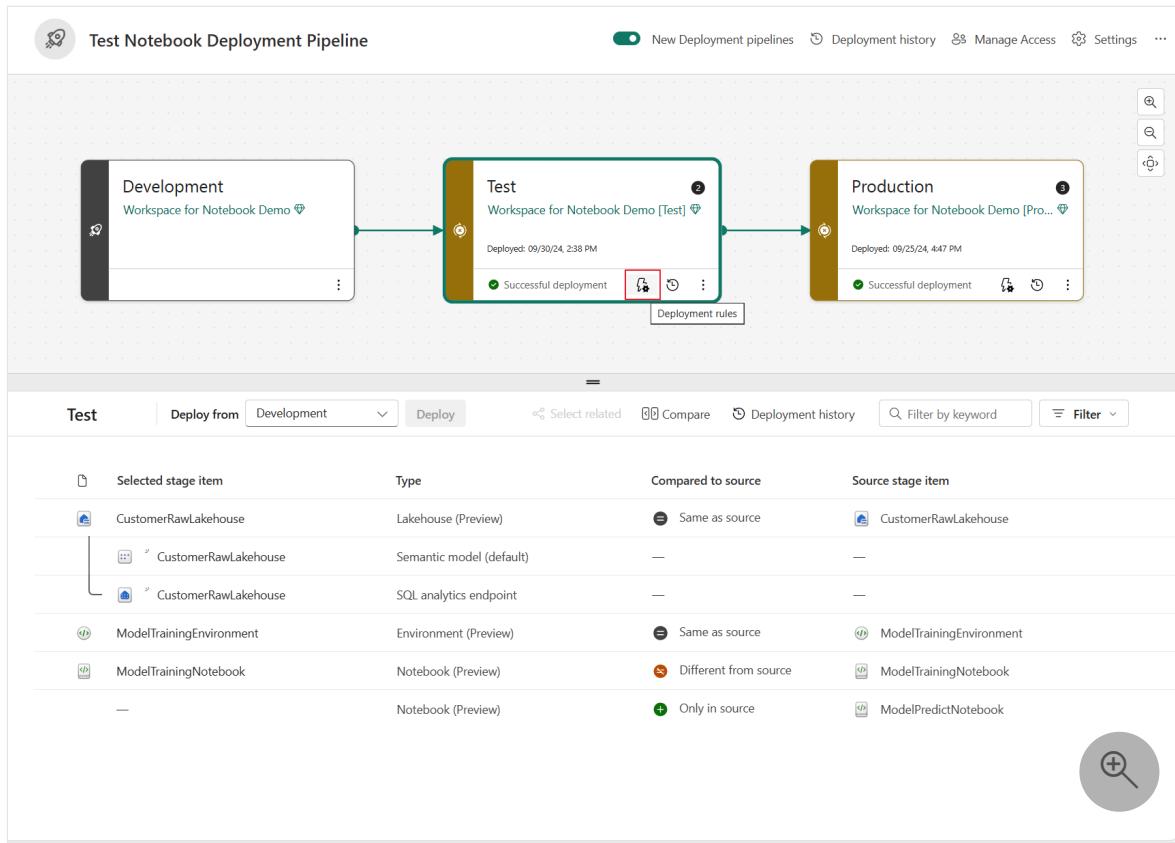
1. Create a new deployment pipeline or open an existing deployment pipeline. (For more information, see [Get started with deployment pipelines](#).)
2. Assign workspaces to different stages according to your deployment goals.
3. Select, view, and compare items including notebooks between different stages, as shown in the following example. The highlighted badge indicating changed item count between the previous stage and current stage.

Selected stage item	Type	Compared to source	Source stage item
CustomerRawLakehouse	Lakehouse (Preview)	Same as source	CustomerRawLakehouse
CustomerRawLakehouse	Semantic model (default)	—	—
CustomerRawLakehouse	SQL analytics endpoint	—	—
ModelTrainingEnvironment	Environment (Preview)	Same as source	ModelTrainingEnvironment
ModelTrainingNotebook	Notebook (Preview)	Different from source	ModelTrainingNotebook
—	Notebook (Preview)	Only in source	ModelPredictNotebook

4. Select **Deploy** to deploy your notebooks across the Development, Test, and Production stages.



5. (Optional.) You can select **Deployment rules** to create deployment rules for a deployment process. Deployment rules entry is on the target stage for a deployment process.



Fabric supports parameterizing the default lakehouse for **each** notebook instance when deploying with deployment rules. Three options are available to specify the target default lakehouse: Same with source lakehouse, N/A(no default lakehouse), and other lakehouse.

[← ModelTrainingNotebook](#)

X

Notebook deployment rules

Set deployment rules

Create rules to run when deploying this item. [Learn more](#) ↗

⚡ Default lakehouse

^

Define default lakehouse that will be connected to this item. [Learn more](#)

From

To

CustomerRawLakehouse

Other



Lakehouse id

469b9825-

Lakehouse name

CustomerNewLakehouse

LakehouseWorkspaceld

efac4a27-

+ Add rule



You can achieve secured data isolation by setting up this rule. Your notebook's default lakehouse is replaced by the one you specified as target during deployment.

ⓘ Note

When setting default lakehouse in deployment rules, the **Lakehouse ID** is must have. You can get the lakehouse id from the item URL link. The deployment rules has higher priority than auto-binding, the auto-bound lakehouse will be overwritten when there's deployment rule configured.

6. Monitor the deployment status from [Deployment history](#).

Related content

- [Manage and execute Fabric notebooks with public APIs](#)
- [Introduction to Git integration](#)

- Introduction to deployment pipelines
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Manage and execute notebooks in Fabric with APIs

Article • 01/28/2025

The Microsoft Fabric REST API provides a service endpoint for the create, read, update, and delete (CRUD) operations of a Fabric item. This article describes the available notebook REST APIs and their usage.

Important

This feature is in [preview](#).

With the notebook APIs, data engineers and data scientists can automate their own pipelines and conveniently and efficiently establish CI/CD. These APIs also make it easy for users to manage and manipulate Fabric notebook items, and integrate notebooks with other tools and systems.

These **Item management** actions are available for notebooks:

 Expand table

Action	Description
Create item	Creates a notebook inside a workspace.
Update item	Updates the metadata of a notebook.
Update item definition	Updates the content of a notebook.
Delete item	Deletes a notebook.
Get item	Gets the metadata of a notebook.
Get item definition	Gets the content of a notebook.
List item	List all items in a workspace.

For more information, see [Items - REST API](#).

The following **Job scheduler** actions are available for notebooks:

 Expand table

Action	Description
Run on demand Item Job	Run notebook with parameterization.
Cancel Item Job Instance	Cancel notebook job run.
Get Item Job Instance	Get notebook run status.

For more information, see [Job Scheduler](#).

ⓘ Note

Service principal authentication is available for Notebook CRUD API and Job scheduler API, meaning you can use service principal to do the CRUD operations and trigger/cancel notebook runs, and get the run status. You need to add the service principal to the workspace with the appropriate role.

Notebook REST API usage examples

Use the following instructions to test usage examples for specific notebook public APIs and verify the results.

ⓘ Note

These scenarios only cover notebook-unique usage examples. Fabric item common API examples are not covered here.

Prerequisites

The Fabric Rest API defines a unified endpoint for operations. Replace the placeholders `{WORKSPACE_ID}` and `{ARTIFACT_ID}` with appropriate values when you follow the examples in this article.

Create a notebook with a definition

Create a notebook item with an existing .ipynb file and other type of source files.

Request

HTTP

```
POST https://api.fabric.microsoft.com/v1/workspaces/{{WORKSPACE_ID}}/items
```

```
{  
    "displayName": "Notebook1",  
    "type": "Notebook",  
    "definition": {  
        "format": "ipynb", // Use "fabricGitSource" for source file format.  
        "parts": [  
            {  
                "path": "notebook-content.ipynb", // fabric source file  
format, .py, .scala, .sql files are supported.  
                "payload":  
"eyJuYmZvcm1hdCI6NCwibmJmb3JtYXRfbWlub3Ii0jUsImNlbGxzIjpbe...  
                "payloadType": "InlineBase64"  
            }  
        ]  
    }  
}
```

The payload in the request is a base64 string converted from the following sample notebook.

JSON

```
{  
    "nbformat": 4,  
    "nbformat_minor": 5,  
    "cells": [  
        {  
            "cell_type": "code",  
            "source": [  
                "# Welcome to your new notebook\n# Type here in the cell  
editor to add code!\n"],  
            "execution_count": null,  
            "outputs": [],  
            "metadata": {}  
        }  
    ],  
    "metadata": {  
        "language_info": {  
            "name": "python"  
        },  
        "dependencies": {  
            "environment": {  
                "environmentId": "6524967a-18dc-44ae-86d1-0ec903e7ca05",  
                "workspaceId": "c31eddd2-26e6-4aa3-9abb-c223d3017004"  
            },  
            "environmentVariables": {}  
        }  
    }  
},  
"kernelspec": {  
    "display_name": "Python 3",  
    "language": "python",  
    "name": "python3"  
},  
"display_data": {}  
}
```

```
        "lakehouse": {
            "default_lakehouse": "5b7cb89a-81fa-4d8f-87c9-3c5b30083bee",
            "default_lakehouse_name": "lakehouse_name",
            "default_lakehouse_workspace_id": "c31eddd2-26e6-4aa3-9abb-
c223d3017004"
        }
    }
}
```

ⓘ Note

You can change the notebook default lakehouse or attached environment by changing notebook content `metadata.trident.lakehouse` or `metadata.trident.environment`.

Get a notebook with a definition

Use the following API to get the notebook content. Fabric supports you setting the format as `.ipynb` in the query string to get an `.ipynb` notebook.

Request

HTTP

```
POST  
https://api.fabric.microsoft.com/v1/workspaces/{{WORKSPACE\_ID}}/items/{{ARTIFACT\_ID}}/GetDefinition?format=ipynb
```

Response

Status code: 200

JSON

```
{  
    "definition": {  
        "parts": [  
            {  
                "path": "notebook-content.ipynb",  
                "payload":  
                    "eyJuYmZvcmlhdCI6NCwibmJmb3JtYXRfbWlub3Ii0jUsImNlbGxzIjpbe...  
                    jb2RlIiwic291cmNlIjpBIiMgV2VsY29tZSB0byB5b3VyIG5ldyBub3RlYm9va1xuIyBUeXBIGH  
                    lcmUgaW4gdGhIGN1bGwgZWRpdyG9yIHRvIGFkZCBjb2RlIVxuIl0sImV4ZW...  
                    6bnVsbcWib3V0cHV0cyI6W10sIm1ldGFkYXRhIjp7fx1dLCJtZXRhZGF0YSI6eyJsYW5ndWFnZV9  
                    pbmZvIjp7Im5hbWUiOijweXRob24ifX19",  
            }  
        ]  
    }  
}
```

```
        "payloadType": "InlineBase64"
    }
]
}
}
```

Run a notebook on demand

Schedule your notebook run with the following API. The Spark job starts executing after a successful request.

Fabric supports passing `parameters` in the request body to parameterize the notebook run. The values are consumed by the [notebook parameter cell](#).

You can also use `configuration` to personalize the Spark session of notebook run.

`configuration` shares the same contract with the [Spark session configuration magic command](#).

Request

HTTP

POST

https://api.fabric.microsoft.com/v1/workspaces/{{WORKSPACE_ID}}/items/{{ARTIFACT_ID}}/jobs/instances?jobType=RunNotebook

```
{
    "executionData": {
        "parameters": {
            "parameterName": {
                "value": "new value",
                "type": "string"
            }
        },
        "configuration": {
            "conf": {
                "spark.conf1": "value"
            },
            "environment": {
                "id": "<environment_id>",
                "name": "<environment_name>"
            },
            "defaultLakehouse": {
                "name": "<lakehouse-name>",
                "id": "<lakehouse-id>",
                "workspaceId": "<(optional) workspace-id-that-contains-the-lakehouse>"
            },
            "useStarterPool": false,
            "useWorkspacePool": "<workspace-pool-name>"
        }
    }
}
```

```
        }
    }
}
```

Response

Status code: 202

HTTP

```
Location: https://api.fabric.microsoft.com/v1/workspaces/4b218778-e7a5-4d73-  
8187-f10824047715/items/431e8d7b-4a95-4c02-8ccd-  
6faef5ba1bd7/jobs/instances/f2d65699-dd22-4889-980c-15226deb0e1b  
Retry-After: 60
```

With `location`, you can use [Get Item Job Instance](#) to view job status or use [Cancel Item Job Instance](#) to cancel the current notebook run.

Related content

- [Develop, execute, and manage Microsoft Fabric notebooks](#)
- [Notebook source control and deployment](#)
- [Microsoft Spark Utilities \(MSSparkUtils\) for Fabric](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Create, configure, and use an environment in Microsoft Fabric

Article • 03/31/2025

Microsoft Fabric environment is a consolidated item for all your hardware and software settings. In an environment, you can select different Spark runtimes, configure your compute resources, install libraries from public repositories or local directory and more.

This tutorial gives you an overview of creating, configuring, and using an environment.

Create an environment

There are multiple entry points of creating new environments.

- Standard entry point

In the creation hub or the **New** section of your workspace, you can find the option of creating new environment like other Fabric items.

- Create during selection

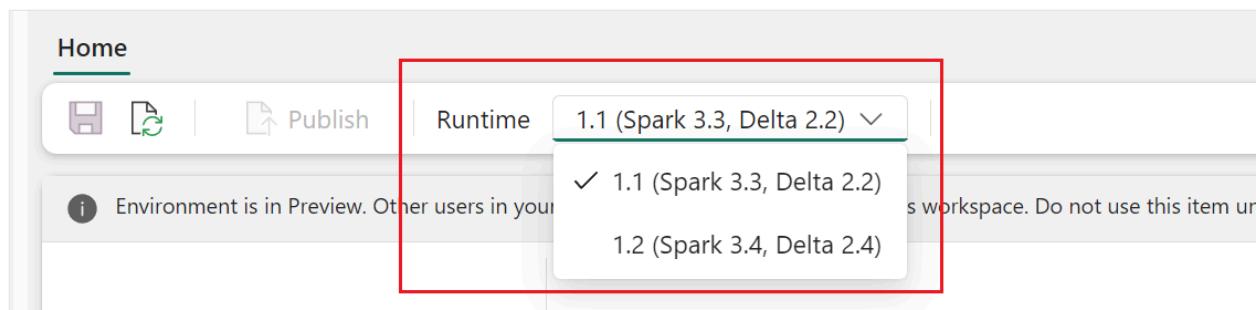
When you select the environment for your notebook, Spark job definition and the workspace default, you can find the option to create new environment.

Configure an environment

There are three major components in an environment, which are Spark compute that includes Spark runtime, libraries, and resource. The Spark compute and libraries configurations are required for the publishing to be effective, while resources are a shared storage that can change in real-time. See [Save and publish changes](#) section for more details.

Configure Spark compute

For an environment, you can choose from various [Spark runtimes](#), each with its own default settings and preinstalled packages. To view the available runtimes, navigate to the **Home** tab of the environment and select **Runtime**. Select the runtime that best suits your needs.



ⓘ Important

- If you are updating the runtime of an environment with existing configurations or libraries, you must republish the contents based on the updated runtime version.
- If the existing configurations or libraries are not compatible with the newly updated runtime version, the publishing fails. You must remove the incompatible configurations or libraries and publish the environment again.

Microsoft Fabric Spark compute provides unparalleled speed and efficiency running on Spark and requirement-tailored experiences. In your environment, you can choose from various pools created by workspace admins and capacity admins. You can further adjust the configurations and manage Spark properties to be effective in Spark sessions. For more information, see [Spark compute configuration settings in Fabric environments](#).

Manage libraries

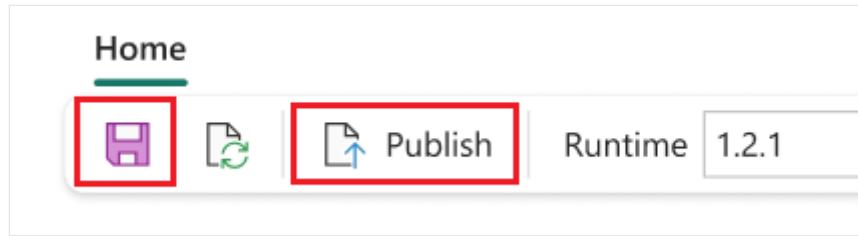
Except for the built-in libraries provided by each Spark runtime, the Fabric environment allows you to install libraries from public sources or upload custom libraries built by you or your organization. Once you successfully install the libraries, they're available in your Spark sessions. For more information, see [Library management in Fabric environments](#). You can also find the best practices of managing libraries in Microsoft Fabric. [Manage Apache Spark libraries in Microsoft Fabric](#)

Resources

The Resources section in environment facilitates the ability to manage small resources during the development phase. Files uploaded to the environment are accessible across notebooks when attached. For more information, see [Manage the resources in Fabric environment](#)

Save and publish changes

In the **Home** tab of environment ribbon, you can easily find two buttons called **Save** and **Publish**. They will be activated when there are unsaved or unpublished pending changes in the Libraries and Spark compute sections.



You will also see a banner prompting these two buttons when there are pending changes in the the Libraries and Spark compute sections, they have the same functionalities with the ones in the ribbon.

- The unsaved changes are lost if you refresh or leave the browser open. Select the **Save** button to make sure your changes are recorded before leaving. Saving doesn't apply the configuration but caches them in the system.
- Select **Publish** to apply the changes to Libraries and Spark compute. The **Pending changes** page will appear for final review before publishing. Next select **Publish all** to initiate configuration in the Fabric environment. This process may take some time, especially if library changes are involved.
- To cancel a publishing process, select **View progress** in the banner and **Cancel** the operation.
- A notification appears upon publishing completion. An error notification occurs if there are any issues during the process.

Note

An environment accepts only one publish at a time. No further changes can be made to the libraries or the Spark compute section during an ongoing publish. Publishing doesn't impact adding, deleting, or editing the files and folders in **Resources** section. The actions to manage resources are in real-time, publish doesn't block changes in resources section.

Share an existing environment

Microsoft Fabric supports sharing an item with different level of permissions.

The screenshot shows the 'Select permissions' dialog box over a Microsoft Fabric environment interface. The dialog box has sections for 'People who can view this Environment' (with 'Specific people' selected), 'Additional permissions' (with 'Share' and 'Edit' options), and buttons for 'Apply' and 'Back'. In the background, the environment library list shows items like 'emoji', 'et-xmlfile', 'fuzzywuzzy', and 'wordcloud'. A preview pane on the right shows a table of dependencies with columns 'Source', 'Status', and 'Last updated'.

When you share an environment item, recipients automatically receive **Read permission**. With this permission, they can explore the environment's configurations and attach it to notebooks or Spark jobs. For smooth code execution, ensure to grant read permissions for attached environments when sharing notebooks and Spark job definitions.

Additionally, you can share the environment with **Share** and **Edit** permissions. Users with **Share permission** can continue sharing the environment with others. Meanwhile, recipients with **Edit permission** can update the environment's content.

Attach an environment

Microsoft Fabric environment can be attached to your **Data Engineering/Science** workspaces or your notebooks and Spark job definitions.

Attach an environment as workspace default

ⓘ Important

Once an environment is selected as workspace default, only workspace admins can update the contents of the default environment.

Find the **Environment** tab by selecting **Workspace settings > Data Engineering/Science > Spark settings**.

Workspace admins can define the default workload for entire workspaces. The values configured here are effective for notebooks and Spark job definitions that attach to **Workspace settings**.

The **Set default environment** toggle can enhance the user experience. By default, this toggle is set to **Off**. If there's no default Spark property or library required as the workspace default, admins can define the Spark runtime in this circumstance. However, if an admin wants to prepare a default Spark compute and libraries for the workspace, they can switch the toggle to **On** and easily attach an environment as the workspace default. This option makes all configurations in the environment effective as the **Workspace settings**.

Workspace settings

Search

This section contains unsaved changes.

Spark settings

Configure and manage settings for Spark workloads and the default environment for the workspace.

About

Premium

Azure connections

Pool

Environment

High concurrency

Automatic log

System storage

Git integration

Other

Power BI

Data
Engineering/Science

Spark settings

Set default environment

On

The default environment will provide Spark properties, libraries, and developer settings for notebooks and Spark job definitions in this workspace when users don't select a different environment.

[Learn more about Set default environment](#)

env

Filter by keyword

Items

env

Runtime: 1.2 (Spark 3.4, Delta 2.4), Compute: Medium, 1 - 10 nodes

env11

Runtime: 1.2 (Spark 3.4, Delta 2.4), Compute: Medium, 1 - 10 nodes

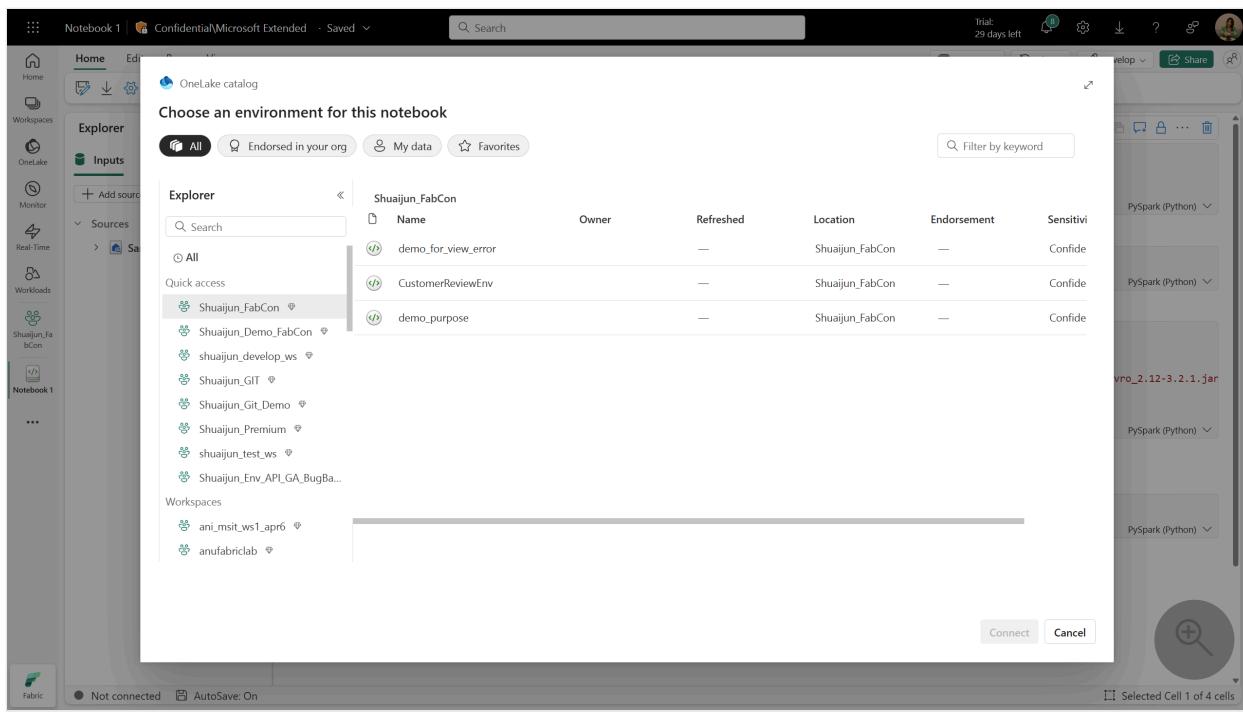
New Environment

Save

Discard

Attach an environment to a notebook or a Spark job definition

The **Environment** is available in both the Notebook and Spark Job Definition Home tabs. Attaching to an environment enables Notebooks and Spark job definitions to access its libraries, compute configurations, and resources. The explorer will list all available environments, including those shared with you, from the current workspace, and from other workspaces you have access to.



ⓘ Note

If you switch to a different environment during an active session, the newly selected environment will not take effect until the next session. When you attach an environment from another workspace, both workspaces must have the **same capacity** and **network security settings**. Although you can select environments from workspaces with different capacities or network security settings, the session will fail to start. When you attach an environment from another workspace, the compute configuration in that environment is ignored. Instead, the pool and compute configurations will default to the settings of your current workspace.

Related content

- [Spark compute configuration settings in Fabric environments](#)
- [Library management in Fabric environments](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Spark compute configuration settings in Fabric environments

Article • 03/13/2025

Microsoft Fabric Data Engineering and Data Science experiences operate on a fully managed Spark compute platform. This platform is designed to deliver unparalleled speed and efficiency. It includes starter pools and custom pools.

A Fabric environment contains a collection of configurations, including Spark compute properties that allow users to configure the Spark session after they're attached to notebooks and Spark jobs. With an environment, you have a flexible way to customize compute configurations for running your Spark jobs. In an environment, the compute section allows you to configure the Spark session level properties to customize the memory and cores of executors based on workload requirements. The Spark properties set via `spark.conf.set` control application-level parameters and they aren't related to environment variables.

Workspace admins can enable or disable compute customizations with the **Customize compute configurations for items** switch in the **Pool** tab of the **Data Engineering/Science** section in the **Workspace settings** screen.

Workspace admins can delegate the members and contributors to change the default session level compute configurations in a Fabric environment by enabling this setting.

Workspace settings

Search

- About
- Premium
- Azure connections
- System storage
- Git integration
- Other
- Power BI

Use the automatically created Starter pool or create custom pools for workspaces and items in the capacity. If the setting Customize compute configurations for items is turned off, this pool will be used for all environments in this workspace.

StarterPool

Pool details		
Node family	Node size	Number of nodes
Memory optimized	Medium	1 - 1

Customize compute configurations for items On

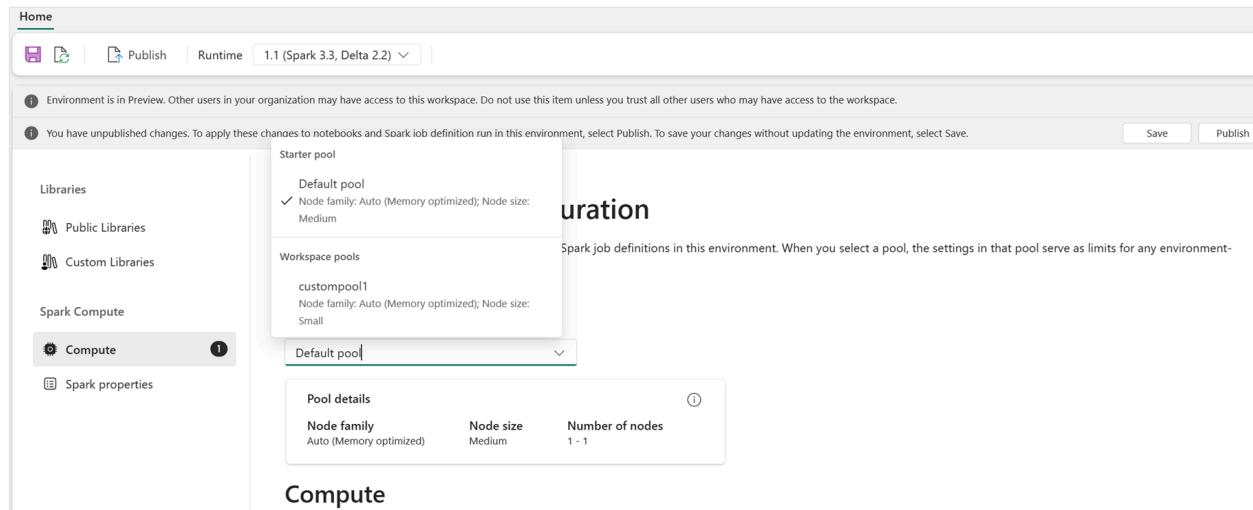
When turned on, users can adjust compute configuration for individual items such as notebooks and Spark job definitions.

[Learn more about Customize compute configurations for items](#)

If the workspace admin disables this option in the workspace settings, the compute section of the environment is disabled and the default pool compute configurations for the workspace are used for running Spark jobs.

Customizing session level compute properties in an environment

As a user, you can select a pool for the environment from the list of pools available in the Fabric workspace. The Fabric workspace admin creates the default starter pool and custom pools.



After you select a pool in the **Compute** section, you can tune the cores and memory for the executors within the bounds of the node sizes and limits of the selected pool.

For example: You select a custom pool with node size of large, which is 16 Spark vCores, as the environment pool. You can then choose the driver/executor core to be either 4, 8 or 16, based on your job level requirement. For the memory allocated to driver and executors, you can choose 28 g, 56 g, or 112 g, which are all within the bounds of a large node memory limit.

Spark Compute

Compute 1

Spark properties

Environment pool

custompool1

Pool details

Node family	Node size	Number of nodes
Auto (Memory optimized)	Large	1 - 8

Compute

Spark driver core

4

4

8

16

4

Spark executor memory

28g

Dynamically allocate executors

Enable allocate

For more information about Spark compute sizes and their cores or memory options, see [What is Spark compute in Microsoft Fabric?](#).

Related content

- [Create, configure, and use an environment in Microsoft Fabric.](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Library management in Fabric environments

Article • 11/08/2024

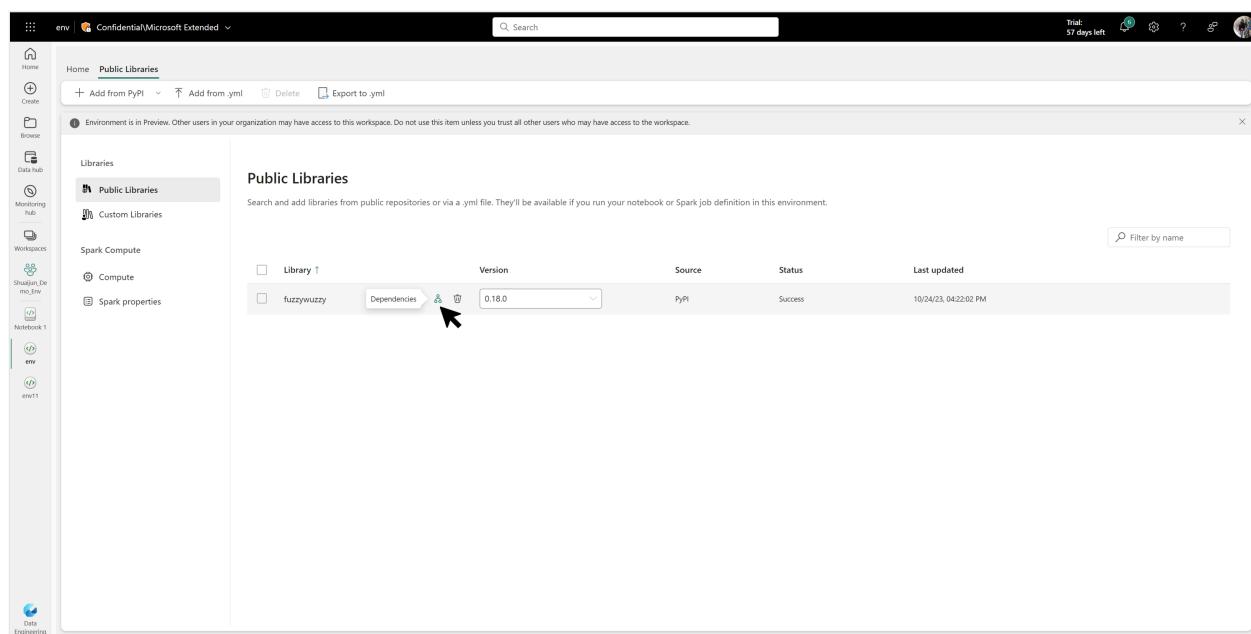
Microsoft Fabric environments provide flexible configurations for running your Spark jobs. Libraries provide reusable code that developers want to include in their work. Except for the built-in libraries that come with each Spark runtime, you can install public and custom libraries in your Fabric environments. And you can easily attach environments to your notebooks and Spark job definitions.

⚠ Note

Modifying the version of a specific package could potentially break other packages that depend on it. For instance, downgrading `azure-storage-blob` might cause problems with `Pandas` and various other libraries that rely on `Pandas`, including `mssparkutils`, `fsspec_wrapper`, and `notebookutils`. You can view the list of preinstalled packages and their versions for each runtime [here](#). Check more options and best practices of using libraries in Microsoft Fabric: [Manage Apache Spark libraries in Microsoft Fabric](#)

Public libraries

Public libraries are sourced from repositories such as PyPI and Conda, which Fabric currently supports.



The screenshot shows the Microsoft Fabric Data Engineering workspace interface. On the left, there's a sidebar with icons for Home, Create, Browse, Data Hub, Monitoring hub, Workspaces, and a Notebook section containing 'env' and 'env11'. The main area has a header with 'Home' and 'Public Libraries' selected. Below the header is a toolbar with buttons for 'Add from PyPI', 'Add from .yml', 'Delete', and 'Export to .yml'. A note says 'Environment is in Preview. Other users in your organization may have access to this workspace. Do not use this item unless you trust all other users who may have access to the workspace.' The central part is titled 'Public Libraries' with a sub-instruction 'Search and add libraries from public repositories or via a .yml file. They'll be available if you run your notebook or Spark job definition in this environment.' A table lists a single library entry:

Library	Version	Source	Status	Last updated
fuzzywuzzy	0.18.0	PyPI	Success	10/24/23, 04:22:02 PM

A cursor is hovering over the trash icon in the table row for 'fuzzywuzzy'.

Add a new public library

To add a new public library, select a source and specify the name and version of the library. Alternatively, you can upload a Conda environment specification .yml file to specify the public libraries. The content of the uploaded .yml file is extracted and appended to the list.

Note

The auto-completion feature for library names during adding is limited to the most popular libraries. If the library you want to install is not on that list, you don't receive an auto-completion prompt. Instead, search for the library directly in PyPI or Conda by entering its full name. If the library name is valid, you see the available versions. If the library name is not valid, you get a warning that the library doesn't exist.

Add public libraries in a batch

Environments support uploading the YAML file to manage multiple public libraries in a batch. The contents of the YAML are extracted and appended in the public library list.

Note

The custom conda channels in YAML file are currently not supported. Only the libraries from PyPI and conda are recognized.

Filter public libraries

Enter keywords in the search box on the **Public Libraries** page, to filter the list of public libraries and find the one you need.

Update public libraries

To update the version of an existing public library, navigate to your environment and open the **Public libraries** or **Custom libraries**. Choose the required library, select the version drop-down, and update its version.

Delete public libraries

The trash option for each library appears when you hover over the corresponding row. Alternatively, you can delete multiple public libraries by selecting them and then selecting **Delete** on the ribbon.

View dependency

Each public library has various dependencies. The view dependency option appears when you hover over the corresponding row.

Export to yaml

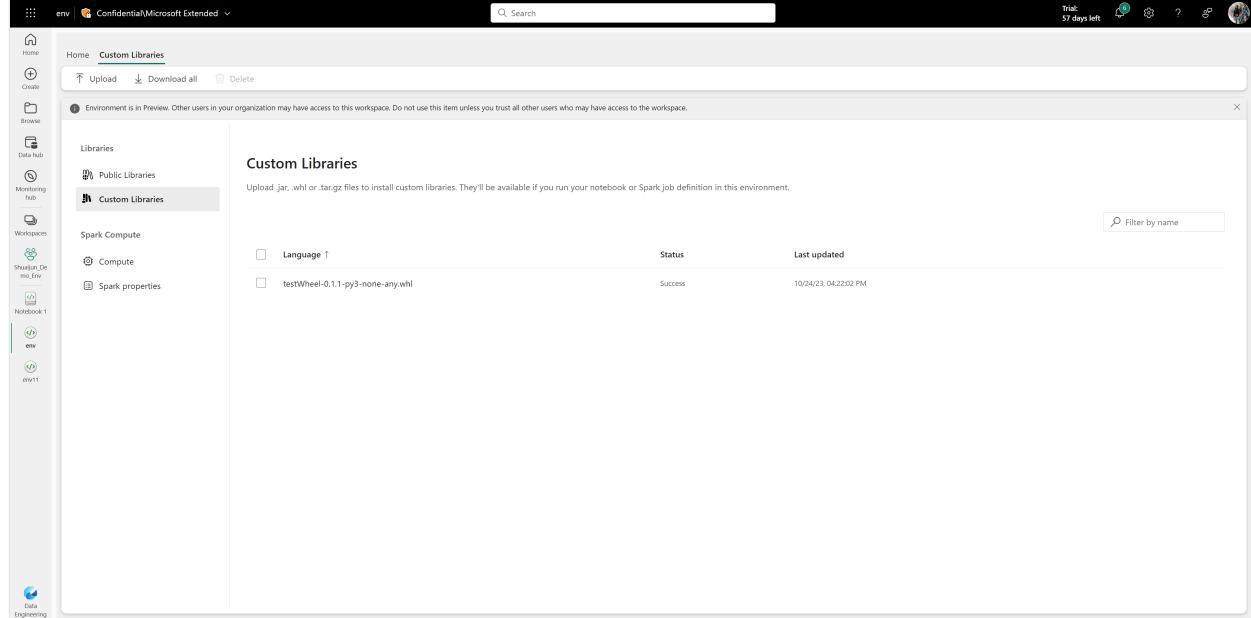
Fabric provides the option to export the full public library list to a YAML file and download it to your local directory.

Custom libraries

Custom libraries refer to code built by you or your organization. Fabric supports custom library files in .whl, .jar, and .tar.gz formats.

⚠ Note

Fabric only supports *.tar.gz* files for R language. Use the *.whl* file format for Python language.



The screenshot shows the Fabric interface with the 'Custom Libraries' tab selected in the left sidebar. The main area displays a table of custom libraries:

Language	Status	Last updated
testWheel-0.1.1-py3-none-any.whl	Success	10/24/23, 04:22:02 PM

At the bottom of the table is a 'Filter by name' search bar. The left sidebar also includes sections for 'Public Libraries' and 'Spark Compute'.

Upload the custom library

You can upload custom libraries from your local directory to the Fabric environment.

Delete the custom library

The trash option for each library appears when you hover the corresponding row. Alternatively, you can delete multiple custom libraries by selecting them and then selecting **Delete** on the ribbon.

Download all custom libraries

If clicked, custom libraries download one by one to your local default download directory.

Related content

- [Create, configure, and use an environment in Microsoft Fabric](#)
- [Manage Apache Spark libraries in Microsoft Fabric](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Ask the community ↗](#)

Migrate workspace libraries and Spark properties to a default environment

Article • 11/15/2023

Microsoft Fabric environments provide flexible configurations for running your Spark jobs. In an environment, you can select different Spark runtimes, configure your compute resources, and install libraries from public repositories or upload local custom-built libraries. You can easily attach environments to your notebooks and Spark job definitions.

Data Engineering and Data Science workspace settings are upgraded to include Fabric environments. As a part of this upgrade, Fabric no longer supports adding new libraries and Spark properties in workspace settings. Instead, you can create a Fabric environment, configure the library and property in it and attach it as the workspace default environment. After you create an environment and set it as the default, you can migrate the existing libraries and Spark properties to that default environment.

In this tutorial, learn how to migrate the existing workspace libraries and Spark properties to an environment.

Important

- Workspace settings are restricted to admins.
- Your existing workspace settings remain effective for your notebooks or Spark job definitions if no environment is attached to them. However, you aren't able to make further changes to those settings. We **STRONGLY RECOMMEND** you migrate your existing settings to an environment.
- The migration process includes a step that **permanently removes all existing configurations**. Please **carefully** follow these instructions. There is no way to bring back files if they are deleted accidentally.

Prepare the files for migration

In **Workspace settings**, review your existing configurations.

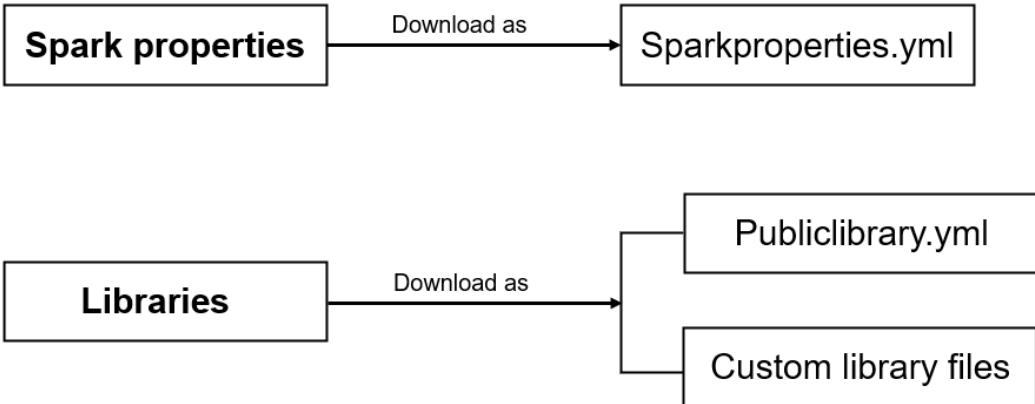
The screenshot shows the 'Spark settings' page within the 'Workspace settings' interface. On the left, there's a sidebar with sections like 'About', 'Premium', 'Azure connections', 'System storage', 'Git integration', and 'Other'. Below that is a 'Power BI' section and a 'Data Engineering/Science' section where 'Spark settings' is selected. A large red box highlights the 'Runtime' section, which displays '1.2' and tabs for 'Spark properties' and 'Libraries'. Under 'Spark properties', there's a table with one row: 'spark.acls.enable' with 'true' as its value. At the bottom are 'Save' and 'Discard' buttons.

Property	Value
spark.acls.enable	true

1. Make a note of the current **Runtime** version.
2. Download existing configurations by selecting **Download all files**.

The content is downloaded as different files. **Sparkproperties.yml** contains all of the Spark properties key value pairs. The **Publiclibrary.yml** file contains all of the public library definitions. Any custom packages uploaded by you or your

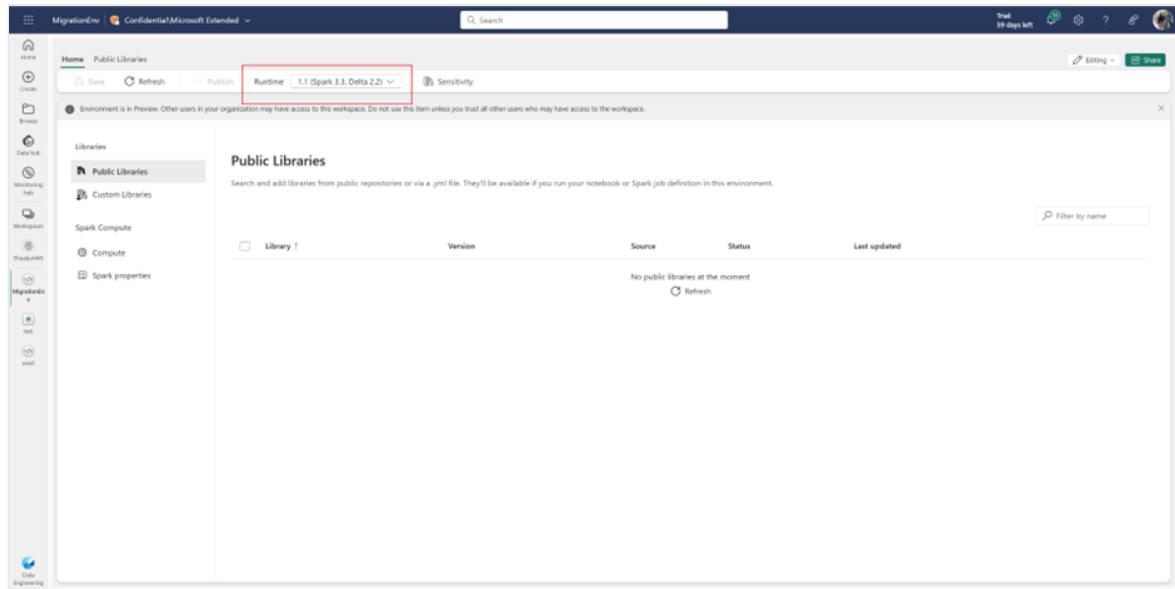
organization are downloaded as files one by one.



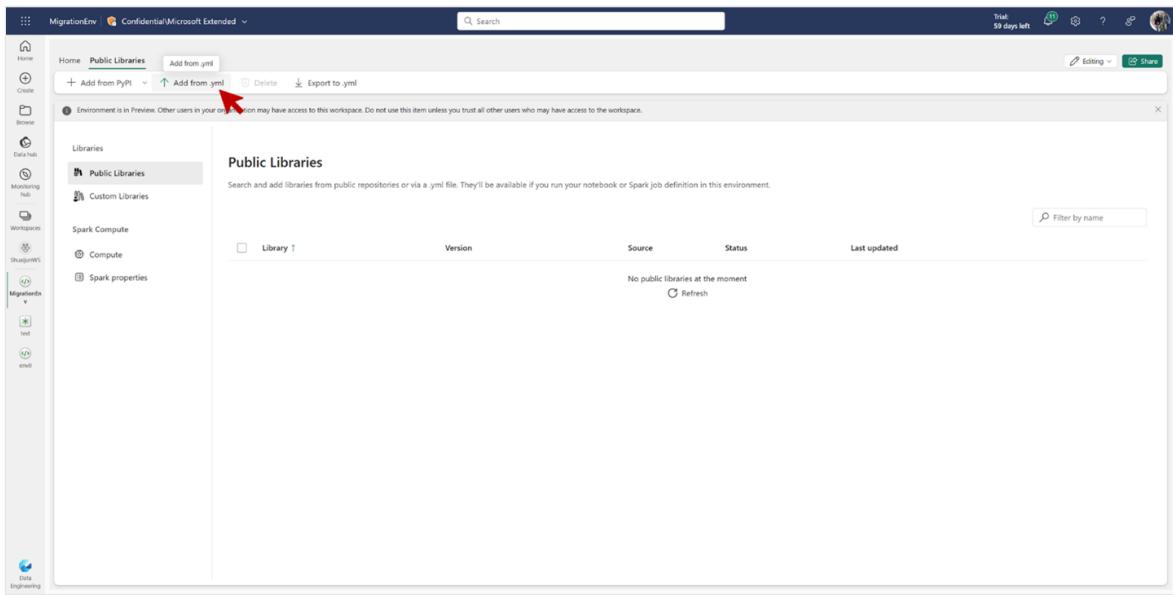
After the files are downloaded, you can migrate.

Create and configure an environment

1. Create an environment in the workspace list/creation hub. After you create a new environment, the Environment page appears.
2. In the **Home** tab of the environment, make sure the **Runtime** version is the same as your existing workspace Runtime.

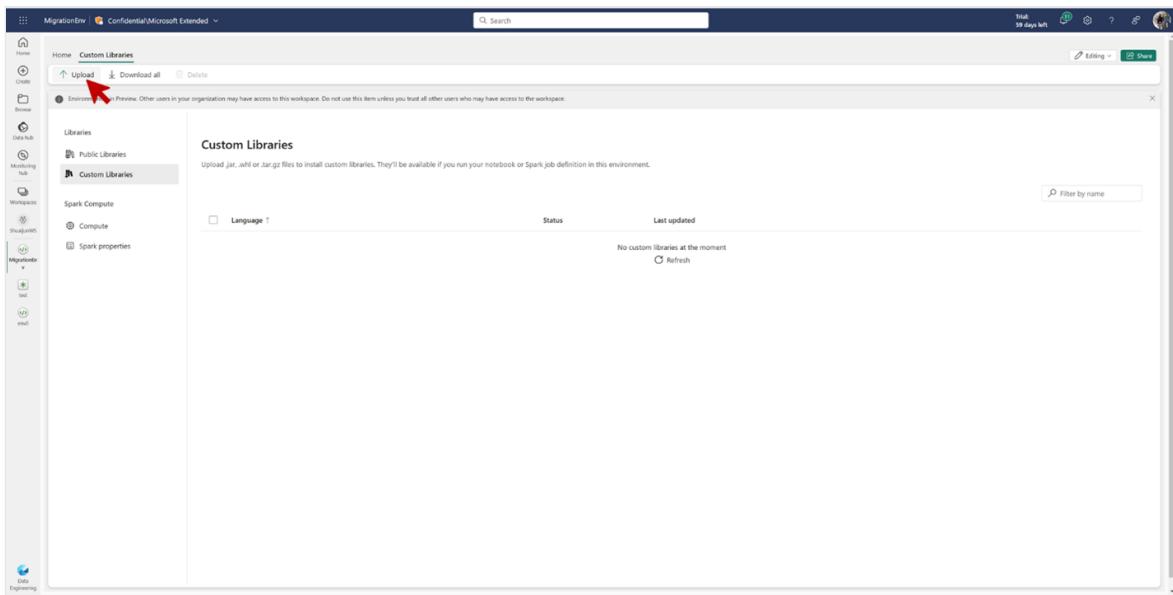


3. Skip this step if you didn't have any public libraries in your workspace settings. Navigate to the **Public Libraries** section and select **Add from .yml** on the ribbon. Upload **Publiclibrary.yml**, which you downloaded from the existing workspace settings.



4. Skip this step if you didn't have any custom libraries in your workspace settings.

Navigate to the **Custom Libraries** section and select **Upload** on the ribbon. Upload the custom library files, which you downloaded from the existing workspace settings.



5. Skip this step if you didn't have any Spark properties in your workspace settings.

Navigate to the **Spark properties** section and select **Upload** on the ribbon. Upload the **Sparkproperties.yml** file, which you downloaded from the existing workspace settings.

6. Select **Publish** and carefully review the changes again. If everything is correct, publish the changes. Publishing takes several minutes to finish.

After publishing is complete, you have successfully configured your environment.

Enable and select a default environment in workspace settings

Important

All existing configurations will be **discarded** when you select **Enable environment**. Make sure that you have downloaded all existing configurations and installed them successfully in an environment before proceeding.

1. Navigate to **Workspace settings** -> **Data Engineering/Science** -> **Environment**, and select **Enable environment**. This action removes the existing configurations and begins your workspace-level environment experience.

The following screen appears when you successfully delete the existing configurations.

Trial:
59 days left

?

?

Workspace settings

Search

Update now to Runtime 1.2. The new Fabric runtime includes Spark 3.4 and Delta 2.4.

About

Premium

Azure connections

System storage

Git integration

Other

Power BI

Data

Engineering/Science

Spark settings

Spark settings

Configure and manage settings for Spark workloads and the default environment for the workspace.

Learn more about Spark settings

Pool Environment High concurrency Automatic log

Customize environment

The default environment will provide Spark properties, libraries, and developer settings for notebooks and Spark job definitions in this workspace when users don't select a different environment.

Learn more about Customize environment

Runtime Version

Runtime version defines which version of Spark your Spark pool will use.

Learn more about Runtime Version

1.1 (Spark 3.3, Delta 2.2)

Off

Save Discard

2. Move the **Customize environment** toggle to the **On** position. This option allows you to attach an environment as a workspace default.

Trial:
59 days left

?

?

Workspace settings

Search

Update now to Runtime 1.2. The new Fabric runtime includes Spark 3.4 and Delta 2.4.

About

Premium

Azure connections

System storage

Git integration

Other

Power BI

Data

Engineering/Science

Spark settings

Spark settings

Configure and manage settings for Spark workloads and the default environment for the workspace.

Learn more about Spark settings

Pool Environment High concurrency Automatic log

Customize environment

The default environment will provide Spark properties, libraries, and developer settings for notebooks and Spark job definitions in this workspace when users don't select a different environment.

Learn more about Customize environment

Runtime Version

Runtime version defines which version of Spark your Spark pool will use.

Learn more about Runtime Version

1.1 (Spark 3.3, Delta 2.2)

Off

Save Discard

A screenshot of the Azure Databricks workspace settings page. On the left, there's a sidebar with various options like About, Premium, Azure connections, System storage, Git integration, Other, Power BI, Data, Engineering/Science, and Spark settings. The Spark settings option is selected and highlighted with a grey background. The main area is titled 'Spark settings' and contains a message about updating to Runtime 1.2. Below that, there are tabs for Pool, Environment (which is selected), High concurrency, and Automatic log. Under the Environment tab, there's a section for 'Customize environment' with a note that it provides default Spark properties, libraries, and developer settings. A toggle switch is shown, and a red arrow points to the 'Off' position. Further down, there's a 'Runtime Version' section with a dropdown menu set to '1.1 (Spark 3.3, Delta 2.2)'. At the bottom, there are 'Save' and 'Discard' buttons.

3. Select the environment you configured in the previous steps as the workspace default, and select **Save**.

Trial:
59 days left

?

Spark settings

Search

Update now to Runtime 1.2. The new Fabric runtime includes Spark 3.4 and Delta 2.4.

About

Premium

Azure connections

System storage

Git integration

Other

Power BI

Data

Engineering/Science

Spark settings

Spark settings

Configure and manage settings for Spark workloads and the default environment for the workspace.

Learn more about Spark settings

Pool Environment High concurrency Automatic log

Customize environment

The default environment will provide Spark properties, libraries, and developer settings for notebooks and Spark job definitions in this workspace when users don't select a different environment.

Learn more about Customize environment

On

Default environment

Environment is in Preview. Other users in your organization may have access to this workspace. Do not use this item unless you trust all other users who may have access to the workspace.

Default environment for workspace

Set the default environment for this workspace. Individual items can be run in different environments.

Learn more about Default environment for workspace

Workspace default

Filter by keyword

Items

MigrationEnv
Runtime: 1.1 (Spark 3.3, Delta 2.2), Compute: Medium, 1 - 10 nodes

New Environment

Save Discard

A screenshot of the Azure Databricks workspace settings page. The 'Spark settings' tab is selected. On the left, there's a sidebar with various options like About, Premium, and Spark settings (which is highlighted). The main area shows the 'Customize environment' section with a toggle switch set to 'On'. Below it is the 'Default environment for workspace' section, which lists 'MigrationEnv' as the current default environment. A red arrow points to the 'MigrationEnv' entry in the list. At the bottom of the modal, there are 'Save' and 'Discard' buttons.

4. Confirm that your new environment now appears under **Default environment for workspace** on the **Spark settings** page.



Workspace settings

 Search

⚠️ Update now to Runtime 1.2. The new Fabric runtime includes Spark 3.4 and Delta

- [ⓘ About](#)
- [⚡ Premium](#)
- [☁ Azure connections](#)
- [📂 System storage](#)
- [⚡ Git integration](#)
- [☰ Other](#)
- [Power BI](#)
- [Data](#)
- [Engineering/Science](#)
- [🔍 Spark settings](#)

Spark settings

Configure and manage settings for Spark workloads and the default environment for the workspace.

[Learn more about Spark settings](#)

Pool **Environment** High concurrency Automatic log

Customize environment

On

The default environment will provide Spark properties, libraries, and developer settings for notebooks and Spark job definitions in this workspace when users don't select a different environment.

[Learn more about Customize environment](#)

⚠️ Environment is in Preview. Other users in your organization may have access to this workspace.
Do not use this item unless you trust all other users who may have access to the workspace.

Default environment for workspace

Set the default environment for this workspace. Individual items can be run in different environments.

[Learn more about Default environment for workspace](#)

MigrationEnv

Runtime

1.1

Spark settings Spark properties Libraries

Spark driver core

8

Spark driver memory

56g

Spark executor core

8

Spark executor memory

56g

Dynamically allocate executors

Enabled

Spark executor instances

1-9

Save

Discard

Related content

- Create, configure, and use an environment in Microsoft Fabric.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Use end-to-end AI samples in Microsoft Fabric

Article • 02/07/2024

The Synapse Data Science software as a service (SaaS) experience in Microsoft Fabric can help machine learning professionals build, deploy, and operationalize their machine learning models in a single analytics platform, while collaborating with other key roles. This article describes both the capabilities of the Synapse Data Science experience, and how machine learning models can address common business problems.

Install Python libraries

Some of the end-to-end AI samples require other libraries for machine learning model development or ad hoc data analysis. You can choose one of these options to quickly install those libraries for your Apache Spark session.

Install with inline installation capabilities

Use the [Python inline installation capabilities](#)- for example, `%pip` or `%conda` - in your notebook, to install new libraries. This option installs the libraries only in the current notebook, and not in the workspace. Use this code to install a library. Replace `<library name>` with the name of your library: `imblearn` or `wordcloud`.

Python

```
# Use pip to install libraries  
%pip install <library name>  
  
# Use conda to install libraries  
%conda install <library name>
```

Set default libraries for the workspace

To make your libraries available for use in any notebooks in the workspace, you can use a [Fabric environment](#) for that purpose. You can create an environment, install the library in it, and then your **workspace admin** can attach the environment to the workspace as its default environment. For more information on setting an environment as the workspace default, see [Admin sets default libraries for the workspace](#).

Important

Library management at the workspace setting is no longer supported. You can follow "[Migrate workspace libraries and Spark properties to a default environment](#)" to migrate existing workspace libraries to an environment and attach it as the workspace default.

Follow tutorials to create machine learning models

These tutorials provide end-to-end samples for common scenarios.

Customer churn

Build a model to predict the churn rate for bank customers. The churn rate, also called the rate of attrition, is the rate at which customers stop doing business with the bank.

Follow along in the [predicting customer churn](#) tutorial.

Recommendations

An online bookstore wants to provide customized recommendations to increase sales. With customer book rating data, you can develop and deploy a recommendation model to make predictions.

Follow along in the [training a retail recommendation model](#) tutorial.

Fraud detection

As unauthorized transactions increase, real-time credit card fraud detection can help financial institutions provide customers faster turnaround time on resolution. A fraud detection model includes preprocessing, training, model storage, and inferencing. The training part reviews multiple models and methods that address challenges like imbalanced examples and trade-offs between false positives and false negatives.

Follow along in the [fraud detection](#) tutorial.

Forecasting

With historical New York City property sales data, and Facebook Prophet, build a time series model with trend and seasonality information to forecast what sales in future cycles.

Follow along in the [time series forecasting](#) tutorial.

Text classification

Apply text classification with word2vec and a linear regression model in Spark, to predict whether or not a book in the British Library is fiction or nonfiction, based on book metadata.

Follow along in the [text classification](#) tutorial.

Uplift model

Estimate the causal impact of certain medical treatments on an individual's behavior, with an uplift model. Touch on four core areas in these modules:

- Data-processing module: extracts features, treatments, and labels.
- Training module: predict the difference in an individual's behavior when treated and when not treated, with a classical machine learning model - for example, LightGBM.
- Prediction module: calls the uplift model for predictions on test data.
- Evaluation module: evaluates the effect of the uplift model on test data.

Follow along in the [causal impact of medical treatments](#) tutorial.

Predictive maintenance

Train multiple models on historical data, to predict mechanical failures such as temperature and rotational speed. Then, determine which model is the best fit to predict future failures.

Follow along in the [predictive maintenance](#) tutorial.

Sales forecast

Predict future sales for superstore product categories. Train a model on historical data to do so.

Follow along in the [sales forecasting](#) tutorial.

Related content

- [How to use Microsoft Fabric notebooks](#)
 - [Machine learning model in Microsoft Fabric](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Tutorial: Create, evaluate, and score a recommendation system

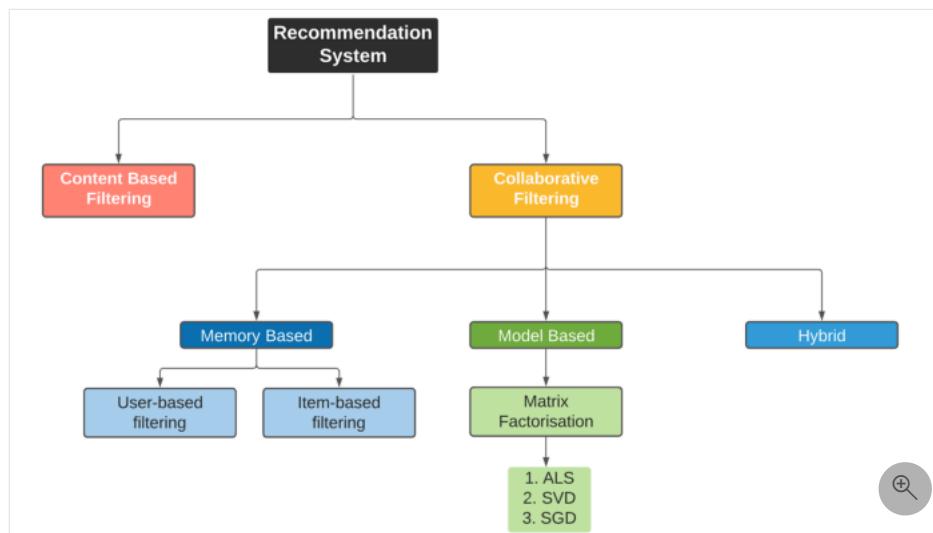
Article • 01/17/2025

This tutorial presents an end-to-end example of a Synapse Data Science workflow in Microsoft Fabric. The scenario builds a model for online book recommendations.

This tutorial covers these steps:

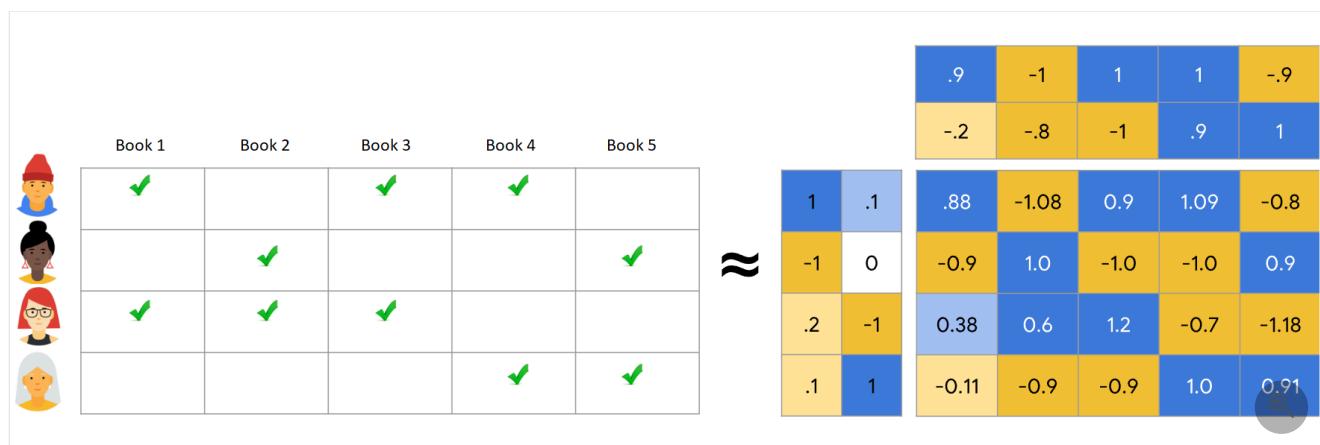
- ✓ Upload the data into a lakehouse
- ✓ Perform exploratory analysis on the data
- ✓ Train a model, and log it with MLflow
- ✓ Load the model and make predictions

We have many types of recommendation algorithms available. This tutorial uses the Alternating Least Squares (ALS) matrix factorization algorithm. ALS is a model-based collaborative filtering algorithm.



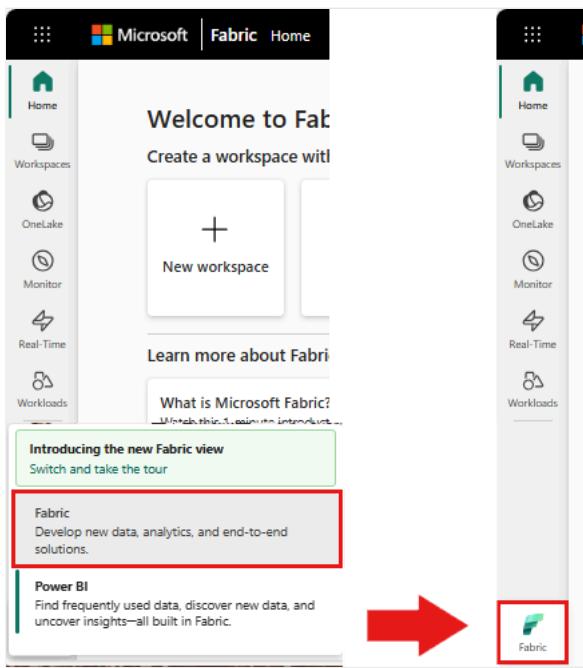
ALS tries to estimate the ratings matrix R as the product of two lower-rank matrices, U and V . Here, $R = U * V^T$. Typically, these approximations are called *factor* matrices.

The ALS algorithm is iterative. Each iteration holds one of the factor matrices constant, while it solves the other using the method of least squares. It then holds that newly solved factor matrix constant while it solves the other factor matrix.



Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- If necessary, create a Microsoft Fabric lakehouse as described in [Create a lakehouse in Microsoft Fabric](#).

Follow along in a notebook

You can choose one of these options to follow along in a notebook:

- Open and run the built-in notebook.
- Upload your notebook from GitHub.

Open the built-in notebook

The sample **Book recommendation** notebook accompanies this tutorial.

1. To open the sample notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#).
2. Make sure to [attach a lakehouse to the notebook](#) before you start running code.

Import the notebook from GitHub

The [Alsample - Book Recommendation.ipynb](#) notebook accompanies this tutorial.

- To open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#) to import the notebook to your workspace.
- If you'd rather copy and paste the code from this page, you can [create a new notebook](#).
- Be sure to [attach a lakehouse to the notebook](#) before you start running code.

Step 1: Load the data

The book recommendation dataset in this scenario consists of three separate datasets:

- *Books.csv*: An International Standard Book Number (ISBN) identifies each book, with invalid dates already removed. The data set also includes the title, author, and publisher. For a book with multiple authors, the *Books.csv* file lists only the first author. URLs point to Amazon website resources for the cover images, in three sizes.

[Expand table](#)

ISBN	Book-Title	Book-Author	Year-Of-Publication	Publisher	Image-URL-S	Image-URL-L
0195153448	Classical Mythology	Mark P. O.	2002	Oxford University Press	http://images.amazon.com/images/P/0195153448.01.THUMBZZZ.jpg	http://images.amazon.com/images/P/0195153448.01.LARGE.jpg

ISBN	Book-Title	Book-Author	Year-Of-Publication	Publisher	Image-URL-S	Image-URL-I
	Morford					
0002005018	Clara Callan	Richard Bruce Wright	2001	HarperFlamingo Canada	http://images.amazon.com/images/P/0002005018.01.THUMBZZZ.jpg	http://images.amazon.com/images/P/0002005018.01.THUMBZZZ.jpg

- *Ratings.csv*: Ratings for each book are either explicit (provided by users, on a scale of 1 to 10) or implicit (observed without user input, and indicated by 0).

[Expand table](#)

User-ID	ISBN	Book-Rating
276725	034545104X	0
276726	0155061224	5

- *Users.csv*: User IDs are anonymized and mapped to integers. Demographic data - for example, location and age - are provided, if available. If this data is unavailable, these values are `null`.

[Expand table](#)

User-ID	Location	Age
1	"nyc new york usa"	
2	"stockton california usa"	18.0

Define these parameters, so that you can this notebook with different datasets:

```
Python

IS_CUSTOM_DATA = False # If True, the dataset has to be uploaded manually

USER_ID_COL = "User-ID" # Must not be '_user_id' for this notebook to run successfully
ITEM_ID_COL = "ISBN" # Must not be '_item_id' for this notebook to run successfully
ITEM_INFO_COL = (
    "Book-Title" # Must not be '_item_info' for this notebook to run successfully
)
RATING_COL = (
    "Book-Rating" # Must not be '_rating' for this notebook to run successfully
)
IS_SAMPLE = True # If True, use only <SAMPLE_ROWS> rows of data for training; otherwise, use all data
SAMPLE_ROWS = 5000 # If IS_SAMPLE is True, use only this number of rows for training

DATA_FOLDER = "Files/book-recommendation/" # Folder that contains the datasets
ITEMS_FILE = "Books.csv" # File that contains the item information
USERS_FILE = "Users.csv" # File that contains the user information
RATINGS_FILE = "Ratings.csv" # File that contains the rating information

EXPERIMENT_NAME = "aisample-recommendation" # MLflow experiment name
```

Download and store the data in a lakehouse

This code downloads the dataset, and then stores it in the lakehouse.

Important

Be sure to [add a lakehouse](#) to the notebook before you run it. Otherwise, you'll get an error.

```
Python
```

```
if not IS_CUSTOM_DATA:
    # Download data files into a lakehouse if they don't exist
    import os, requests

    remote_url = "https://synapseaisolutionsa.blob.core.windows.net/public/Book-Recommendation-Dataset"
    file_list = ["Books.csv", "Ratings.csv", "Users.csv"]
    download_path = f"/lakehouse/default/{DATA_FOLDER}/raw"
```

```

if not os.path.exists("/lakehouse/default"):
    raise FileNotFoundError(
        "Default lakehouse not found, please add a lakehouse and restart the session."
    )
os.makedirs(download_path, exist_ok=True)
for fname in file_list:
    if not os.path.exists(f"{download_path}/{fname}"):
        r = requests.get(f"{remote_url}/{fname}", timeout=30)
        with open(f"{download_path}/{fname}", "wb") as f:
            f.write(r.content)
print("Downloaded demo data files into lakehouse.")

```

Set up the MLflow experiment tracking

Use this code to set up the MLflow experiment tracking. This example disables autologging. For more information, see the [Autologging in Microsoft Fabric](#) article.

Python

```

# Set up MLflow for experiment tracking
import mlflow

mlflow.set_experiment(EXPERIMENT_NAME)
mlflow.autolog(disable=True) # Disable MLflow autologging

```

Read data from the lakehouse

After the correct data is placed in the lakehouse, read the three datasets into separate Spark DataFrames in the notebook. The file paths in this code use the parameters defined earlier.

Python

```

df_items = (
    spark.read.option("header", True)
    .option("inferSchema", True)
    .csv(f"{DATA_FOLDER}/raw/{ITEMS_FILE}")
    .cache()
)

df_ratings = (
    spark.read.option("header", True)
    .option("inferSchema", True)
    .csv(f"{DATA_FOLDER}/raw/{RATINGS_FILE}")
    .cache()
)

df_users = (
    spark.read.option("header", True)
    .option("inferSchema", True)
    .csv(f"{DATA_FOLDER}/raw/{USERS_FILE}")
    .cache()
)

```

Step 2: Perform exploratory data analysis

Display raw data

Explore the DataFrames with the `display` command. With this command, you can view high-level DataFrame statistics, and understand how different dataset columns relate to each other. Before you explore the datasets, use this code to import the required libraries:

Python

```

import pyspark.sql.functions as F
from pyspark.ml.feature import StringIndexer
import matplotlib.pyplot as plt
import seaborn as sns
color = sns.color_palette() # Adjusting plotting style
import pandas as pd # DataFrames

```

Use this code to look at the DataFrame that contains the book data:

```
Python  
display(df_items, summary=True)
```

Add an `_item_id` column for later use. The `_item_id` value must be an integer for recommendation models. This code uses `StringIndexer` to transform `ITEM_ID_COL` to indices:

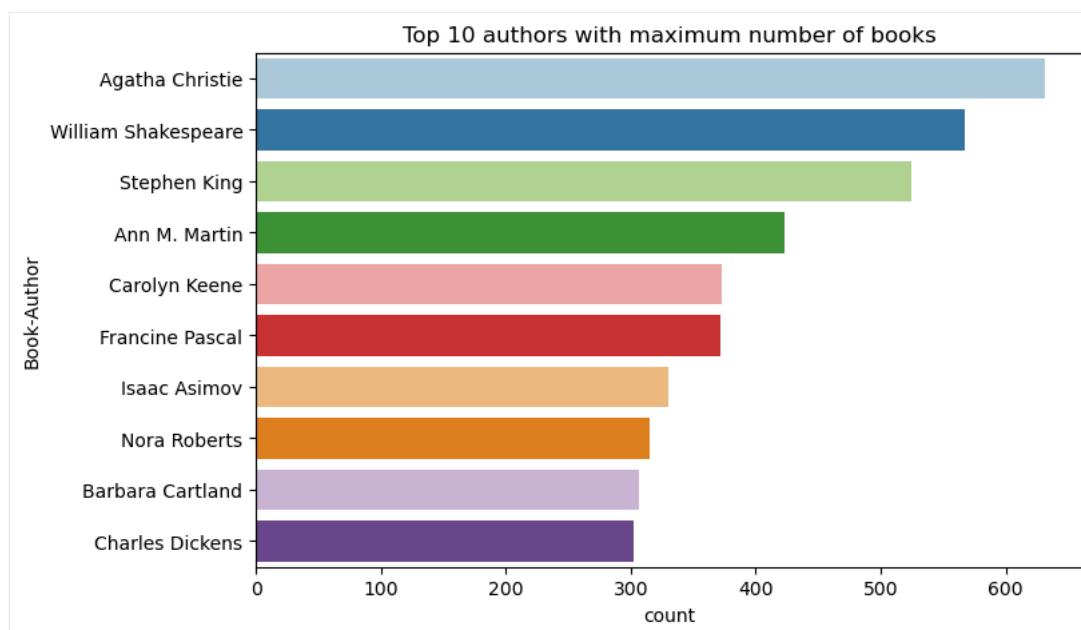
```
Python  
  
df_items = (  
    StringIndexer(inputCol=ITEM_ID_COL, outputCol="_item_id")  
    .setHandleInvalid("skip")  
    .fit(df_items)  
    .transform(df_items)  
    .withColumn("_item_id", F.col("_item_id").cast("int"))  
)
```

Display the DataFrame, and check whether the `_item_id` value increases monotonically and successively, as expected:

```
Python  
display(df_items.sort(F.col("_item_id").desc()))
```

Use this code to plot the top 10 authors, by number of books written, in descending order. Agatha Christie is the leading author with more than 600 books, followed by William Shakespeare.

```
Python  
  
df_books = df_items.toPandas() # Create a pandas DataFrame from the Spark DataFrame for visualization  
plt.figure(figsize=(8,5))  
sns.countplot(y="Book-Author", palette = 'Paired', data=df_books, order=df_books['Book-Author'].value_counts().index[0:10])  
plt.title("Top 10 authors with maximum number of books")
```



Next, display the DataFrame that contains the user data:

```
Python  
display(df_users, summary=True)
```

If a row has a missing `User-ID` value, drop that row. Missing values in a customized dataset don't cause problems.

```
Python  
  
df_users = df_users.dropna(subset=(USER_ID_COL))
```

```
display(df_users, summary=True)
```

Add a `_user_id` column for later use. For recommendation models, the `_user_id` value must be an integer. The following code sample uses `StringIndexer` to transform `USER_ID_COL` to indices.

The book dataset already has an integer `User-ID` column. However, adding a `_user_id` column for compatibility with different datasets makes this example more robust. Use this code to add the `_user_id` column:

Python

```
df_users = (
    StringIndexer(inputCol=USER_ID_COL, outputCol="_user_id")
    .setHandleInvalid("skip")
    .fit(df_users)
    .transform(df_users)
    .withColumn("_user_id", F.col("_user_id").cast("int"))
)
```

Python

```
display(df_users.sort(F.col("_user_id").desc()))
```

Use this code to view the rating data:

Python

```
display(df_ratings, summary=True)
```

Obtain the distinct ratings, and save them for later use in a list named `ratings`:

Python

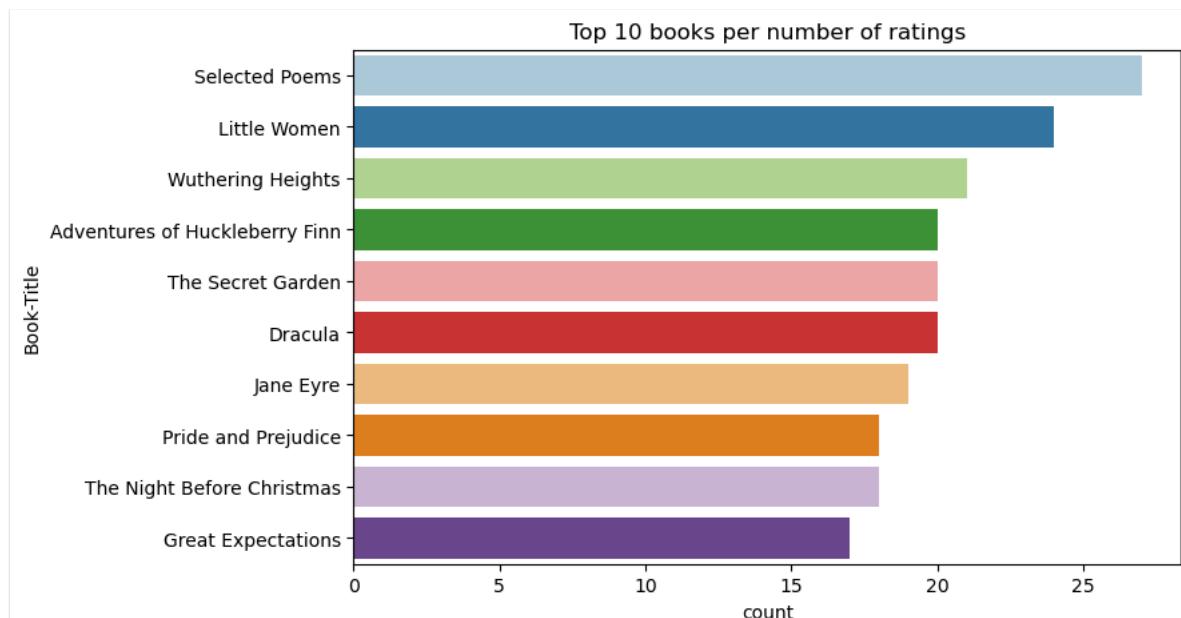
```
ratings = [i[0] for i in df_ratings.select(RATING_COL).distinct().collect()]
print(ratings)
```

Use this code to show the top 10 books with the highest ratings:

Python

```
plt.figure(figsize=(8,5))
sns.countplot(y="Book-Title", palette = 'Paired', data=df_books, order=df_books['Book-Title'].value_counts().index[0:10])
plt.title("Top 10 books per number of ratings")
```

According to the ratings, *Selected Poems* is the most popular book. *Adventures of Huckleberry Finn*, *The Secret Garden*, and *Dracula* have the same rating.



Merge data

Merge the three DataFrames into one DataFrame for a more comprehensive analysis:

Python

```
df_all = df_ratings.join(df_users, USER_ID_COL, "inner").join(
    df_items, ITEM_ID_COL, "inner"
)
df_all_columns = [
    c for c in df_all.columns if c not in ["_user_id", "_item_id", RATING_COL]
]

# Reorder the columns to ensure that _user_id, _item_id, and Book-Rating are the first three columns
df_all = (
    df_all.select(["_user_id", "_item_id", RATING_COL] + df_all_columns)
    .withColumn("id", F.monotonically_increasing_id())
    .cache()
)

display(df_all)
```

Use this code to display a count of the distinct users, books, and interactions:

Python

```
print(f"Total Users: {df_users.select('_user_id').distinct().count()}")
print(f"Total Items: {df_items.select('_item_id').distinct().count()}")
print(f"Total User-Item Interactions: {df_all.count()}")
```

Compute and plot the most popular items

Use this code to compute and display the top 10 most popular books:

Python

```
# Compute top popular products
df_top_items = (
    df_all.groupby(["_item_id"])
    .count()
    .join(df_items, "_item_id", "inner")
    .sort(["count"], ascending=[0])
)

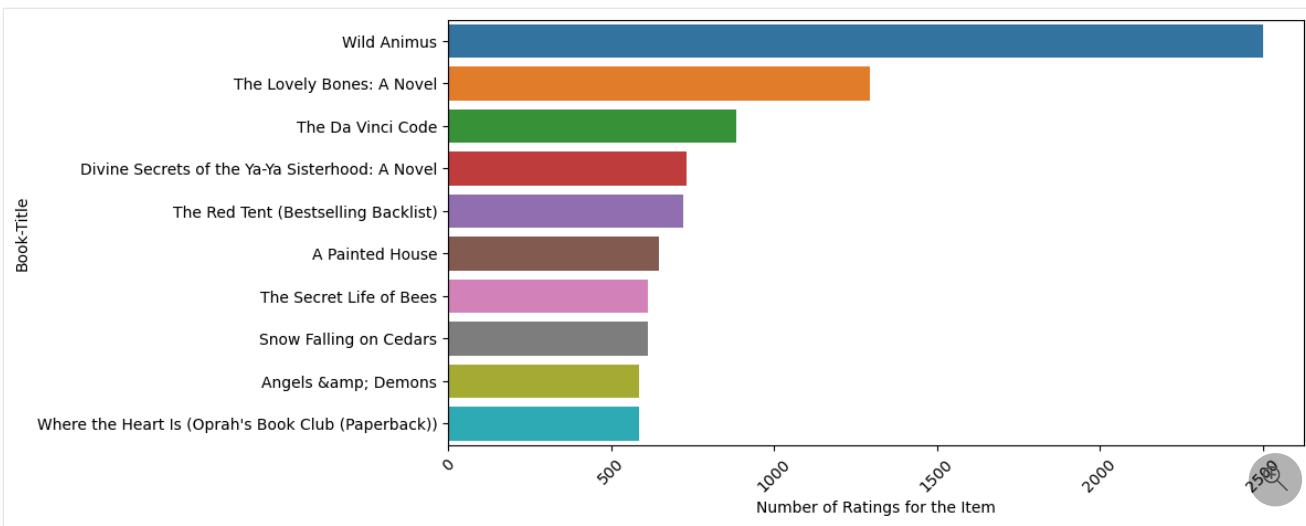
# Find top <topn> popular items
topn = 10
pd_top_items = df_top_items.limit(topn).toPandas()
pd_top_items.head(10)
```



Use the `<topn>` value for Popular or Top purchased recommendation sections.

Python

```
# Plot top <topn> items
f, ax = plt.subplots(figsize=(10, 5))
plt.xticks(rotation="vertical")
sns.barplot(y=ITEM_INFO_COL, x="count", data=pd_top_items)
ax.tick_params(axis='x', rotation=45)
plt.xlabel("Number of Ratings for the Item")
plt.show()
```



2500

Prepare training and test datasets

The ALS matrix requires some data preparation before training. Use this code sample to prepare the data. The code performs these actions:

- Cast the rating column to the correct type
- Sample the training data with user ratings
- Split the data into training and test datasets

Python

```

if IS_SAMPLE:
    # Must sort by '_user_id' before performing limit to ensure that ALS works normally
    # If training and test datasets have no common _user_id, ALS will fail
    df_all = df_all.sort("_user_id").limit(SAMPLE_ROWS)

# Cast the column into the correct type
df_all = df_all.withColumn(RATING_COL, F.col(RATING_COL).cast("float"))

# Using a fraction between 0 and 1 returns the approximate size of the dataset; for example, 0.8 means 80% of the dataset
# Rating = 0 means the user didn't rate the item, so it can't be used for training
# We use the 80% of the dataset with rating > 0 as the training dataset
fractions_train = {0: 0}
fractions_test = {0: 0}
for i in ratings:
    if i == 0:
        continue
    fractions_train[i] = 0.8
    fractions_test[i] = 1
# Training dataset
train = df_all.sampleBy(RATING_COL, fractions=fractions_train)

# Join with leftanti will select all rows from df_all with rating > 0 and not in the training dataset; for example, the
# remaining 20% of the dataset
# test dataset
test = df_all.join(train, on="id", how="leftanti").sampleBy(
    RATING_COL, fractions=fractions_test
)

```

Sparsity refers to sparse feedback data, which can't identify similarities in users' interests. For a better understanding of both the data and the current problem, use this code to compute the dataset sparsity:

Python

```

# Compute the sparsity of the dataset
def get_mat_sparsity(ratings):
    # Count the total number of ratings in the dataset - used as numerator
    count_nonzero = ratings.select(RATING_COL).count()
    print(f"Number of rows: {count_nonzero}")

    # Count the total number of distinct user_id and distinct product_id - used as denominator
    total_elements = (
        ratings.select("_user_id").distinct().count()
        * ratings.select("_item_id").distinct().count()
    )

    # Calculate the sparsity by dividing the numerator by the denominator

```

```
sparsity = (1.0 - (count_nonzero * 1.0) / total_elements) * 100
print("The ratings DataFrame is ", "%."4f" % sparsity + "% sparse.")

get_mat_sparsity(df_all)
```

Python

```
# Check the ID range
# ALS supports only values in the integer range
print(f"max user_id: {df_all.agg({'_user_id': 'max'}).collect()[0][0]}")
print(f"max item_id: {df_all.agg({'_item_id': 'max'}).collect()[0][0]}")
```

Step 3: Develop and train the model

Train an ALS model to give users personalized recommendations.

Define the model

Spark ML provides a convenient API for building the ALS model. However, the model doesn't reliably handle problems like data sparsity and cold start (making recommendations when the users or items are new). To improve model performance, combine cross-validation and automatic hyperparameter tuning.

Use this code to import the libraries required for model training and evaluation:

Python

```
# Import Spark required libraries
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator, TrainValidationSplit

# Specify the training parameters
num_epochs = 1 # Number of epochs; here we use 1 to reduce the training time
rank_size_list = [64] # The values of rank in ALS for tuning
reg_param_list = [0.01, 0.1] # The values of regParam in ALS for tuning
model_tuning_method = "TrainValidationSplit" # TrainValidationSplit or CrossValidator
# Build the recommendation model by using ALS on the training data
# We set the cold start strategy to 'drop' to ensure that we don't get NaN evaluation metrics
als = ALS(
    maxIter=num_epochs,
    userCol="_user_id",
    itemCol="_item_id",
    ratingCol=RATING_COL,
    coldStartStrategy="drop",
    implicitPrefs=False,
    nonnegative=True,
)
```

Tune model hyperparameters

The next code sample constructs a parameter grid, to help search over the hyperparameters. The code also creates a regression evaluator that uses the root-mean-square error (RMSE) as the evaluation metric:

Python

```
# Construct a grid search to select the best values for the training parameters
param_grid = (
    ParamGridBuilder()
    .addGrid(als.rank, rank_size_list)
    .addGrid(als.regParam, reg_param_list)
    .build()
)

print("Number of models to be tested: ", len(param_grid))

# Define the evaluator and set the loss function to the RMSE
evaluator = RegressionEvaluator(
    metricName="rmse", labelCol=RATING_COL, predictionCol="prediction"
)
```

The next code sample initiates different model tuning methods based on the preconfigured parameters. For more information about model tuning, see [ML Tuning: model selection and hyperparameter tuning](#) at the Apache Spark website.

Python

```
# Build cross-validation by using CrossValidator and TrainValidationSplit
if model_tuning_method == "CrossValidator":
    tuner = CrossValidator(
        estimator=als,
        estimatorParamMaps=param_grid,
        evaluator=evaluator,
        numFolds=5,
        collectSubModels=True,
    )
elif model_tuning_method == "TrainValidationSplit":
    tuner = TrainValidationSplit(
        estimator=als,
        estimatorParamMaps=param_grid,
        evaluator=evaluator,
        # 80% of the training data will be used for training; 20% for validation
        trainRatio=0.8,
        collectSubModels=True,
    )
else:
    raise ValueError(f"Unknown model_tuning_method: {model_tuning_method}")
```

Evaluate the model

You should evaluate modules against the test data. A well-trained model should have high metrics on the dataset.

An overfitted model might need an increase in the size of the training data, or a reduction of some of the redundant features. The model architecture might need to change, or its parameters might need some fine tuning.

 Note

A negative R-squared metric value indicates that the trained model performs worse than a horizontal straight line. This finding suggests that the trained model doesn't explain the data.

To define an evaluation function, use this code:

Python

```
def evaluate(model, data, verbose=0):
    """
    Evaluate the model by computing rmse, mae, r2, and variance over the data.
    """

    predictions = model.transform(data).withColumn(
        "prediction", F.col("prediction").cast("double")
    )

    if verbose > 1:
        # Show 10 predictions
        predictions.select("_user_id", "_item_id", RATING_COL, "prediction").limit(
            10
        ).show()

    # Initialize the regression evaluator
    evaluator = RegressionEvaluator(predictionCol="prediction", labelCol=RATING_COL)

    _evaluator = lambda metric: evaluator.setMetricName(metric).evaluate(predictions)
    rmse = _evaluator("rmse")
    mae = _evaluator("mae")
    r2 = _evaluator("r2")
    var = _evaluator("var")

    if verbose > 0:
        print(f"RMSE score = {rmse}")
        print(f"MAE score = {mae}")
        print(f"R2 score = {r2}")
        print(f"Explained variance = {var}")

    return predictions, (rmse, mae, r2, var)
```

Track the experiment by using MLflow

Use MLflow to track all the experiments and to log parameters, metrics, and models. To start model training and evaluation, use this code:

Python

```
from mlflow.models.signature import infer_signature

with mlflow.start_run(run_name="als"):
    # Train models
    models = tuner.fit(train)
    best_metrics = {"RMSE": 10e6, "MAE": 10e6, "R2": 0, "Explained variance": 0}
    best_index = 0
    # Evaluate models
    # Log models, metrics, and parameters
    for idx, model in enumerate(models.subModels):
        with mlflow.start_run(nested=True, run_name=f"als_{idx}") as run:
            print("\nEvaluating on test data:")
            print(f"subModel No. {idx + 1}")
            predictions, (rmse, mae, r2, var) = evaluate(model, test, verbose=1)

            signature = infer_signature(
                train.select(["_user_id", "_item_id"]),
                predictions.select(["_user_id", "_item_id", "prediction"]),
            )
            print("log model:")
            mlflow.spark.log_model(
                model,
                f"{EXPERIMENT_NAME}-alsmodel",
                signature=signature,
                registered_model_name=f"{EXPERIMENT_NAME}-alsmodel",
                dfs_tmpdir="Files/spark",
            )
            print("log metrics:")
            current_metric = {
                "RMSE": rmse,
                "MAE": mae,
                "R2": r2,
                "Explained variance": var,
            }
            mlflow.log_metrics(current_metric)
            if rmse < best_metrics["RMSE"]:
                best_metrics = current_metric
                best_index = idx

            print("log parameters:")
            mlflow.log_params(
                {
                    "subModel_idx": idx,
                    "num_epochs": num_epochs,
                    "rank_size_list": rank_size_list,
                    "reg_param_list": reg_param_list,
                    "model_tuning_method": model_tuning_method,
                    "DATA_FOLDER": DATA_FOLDER,
                }
            )
    # Log the best model and related metrics and parameters to the parent run
    mlflow.spark.log_model(
        models.subModels[best_index],
        f"{EXPERIMENT_NAME}-alsmodel",
        signature=signature,
        registered_model_name=f"{EXPERIMENT_NAME}-alsmodel",
        dfs_tmpdir="Files/spark",
    )
    mlflow.log_metrics(best_metrics)
    mlflow.log_params(
        {
            "subModel_idx": idx,
            "num_epochs": num_epochs,
            "rank_size_list": rank_size_list,
            "reg_param_list": reg_param_list,
            "model_tuning_method": model_tuning_method,
            "DATA_FOLDER": DATA_FOLDER,
        }
    )
```

Select the experiment named `aisample-recommendation` from your workspace to view the logged information for the training run. If you changed the experiment name, select the experiment that has the new name. The logged information resembles this image:

The screenshot shows the Azure Machine Learning Studio interface. At the top, there are navigation links for Home, View, Save as ML model, Download run files, Delete run, and New AutoML run. Below this, a search bar and a tree view show runs: 'aisample-recommendat...' (als, 14:13 AM), 'als_1' (14:51 AM), and 'als_0' (14:45 AM). A central panel displays the properties of the 'als' run, including Run name (als), Start date (1/15/2025 1:41 AM), Duration (5m 50s), Status (Completed), Run ID (3f49696f-43c4...), Created by (Book recommend...), and Source (als). To the right, there are three cards: 'Compare runs' (Compare run details in a list view. Select certain runs to visually compare run metrics. Learn more), 'Save run as an ML model' (When an experimental run yield the desired result, save the run as an ML model for shared usage and tracking. Learn more), and a 'View run list' button.

Step 4: Load the final model for scoring and make predictions

After you finish the model training, and then select the best model, load the model for scoring (sometimes called inferencing). This code loads the model and uses predictions to recommend the top 10 books for each user:

Python

```
# Load the best model
# MLflow uses PipelineModel to wrap the original model, so we extract the original ALSModel from the stages
model_uri = f"models://{EXPERIMENT_NAME}-alsmodel/1"
loaded_model = mlflow.spark.load_model(model_uri, dls_tmpdir="Files/spark").stages[-1]

# Generate top 10 book recommendations for each user
userRecs = loaded_model.recommendForAllUsers(10)

# Represent the recommendations in an interpretable format
userRecs = (
    userRecs.withColumn("rec_exp", F.explode("recommendations"))
    .select("_user_id", F.col("rec_exp._item_id"), F.col("rec_exp.rating"))
    .join(df_items.select(["_item_id", "Book-Title"]), on="_item_id")
)
userRecs.limit(10).show()
```

The output resembles this table:

[Expand table](#)

_item_id	_user_id	rating	Book-Title
44865	7	7.9996786	Lasher: Lives of ...
786	7	6.2255826	The Piano Man's D...
45330	7	4.980466	State of Mind
38960	7	4.980466	All He Ever Wanted
125415	7	4.505084	Harry Potter and ...
44939	7	4.3579073	Taltos: Lives of ...
175247	7	4.3579073	The Bonesetter's ...
170183	7	4.228735	Living the Simple...
88503	7	4.221206	Island of the Blu...
32894	7	3.9031885	Winter Solstice

Save the predictions to the lakehouse

Use this code to write the recommendations back to the lakehouse:

Python

```
# Code to save userRecs into the lakehouse
userRecs.write.format("delta").mode("overwrite").save(
    f"{DATA_FOLDER}/predictions/userRecs"
)
```

Related content

- [Train and evaluate a text classification model](#)
- [Machine learning model in Microsoft Fabric](#)
- [Train machine learning models](#)
- [Machine learning experiments in Microsoft Fabric](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Tutorial: Create, evaluate, and score a churn prediction model

Article • 01/17/2025

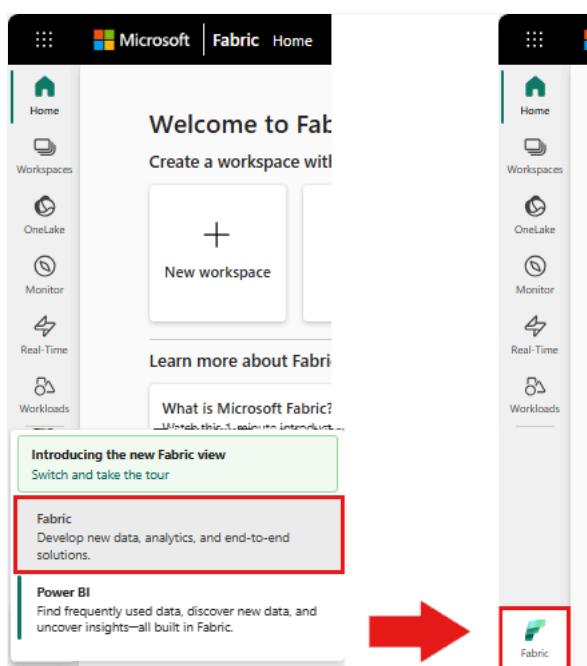
This tutorial presents an end-to-end example of a Synapse Data Science workflow in Microsoft Fabric. The scenario builds a model to predict whether or not bank customers churn. The churn rate, or the rate of attrition, involves the rate at which bank customers end their business with the bank.

This tutorial covers these steps:

- ✓ Install custom libraries
- ✓ Load the data
- ✓ Understand and process the data through exploratory data analysis, and show the use of the Fabric Data Wrangler feature
- ✓ Use scikit-learn and LightGBM to train machine learning models, and track experiments with the MLflow and Fabric Autologging features
- ✓ Evaluate and save the final machine learning model
- ✓ Show the model performance with Power BI visualizations

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- If necessary, create a Microsoft Fabric lakehouse as described in [Create a lakehouse in Microsoft Fabric](#).

Follow along in a notebook

You can choose one of these options to follow along in a notebook:

- Open and run the built-in notebook.
- Upload your notebook from GitHub.

Open the built-in notebook

The sample Customer churn notebook accompanies this tutorial.

1. To open the sample notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#).

2. Make sure to [attach a lakehouse to the notebook](#) before you start running code.

Import the notebook from GitHub

The [Alsample - Bank Customer Churn.ipynb](#) notebook accompanies this tutorial.

- To open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#) to import the notebook to your workspace.
- If you'd rather copy and paste the code from this page, you can [create a new notebook](#).
- Be sure to [attach a lakehouse to the notebook](#) before you start running code.

Step 1: Install custom libraries

For machine learning model development or ad-hoc data analysis, you might need to quickly install a custom library for your Apache Spark session. You have two options to install libraries.

- Use the inline installation capabilities (`%pip` or `%conda`) of your notebook to install a library, in your current notebook only.
- Alternatively, you can create a Fabric environment, install libraries from public sources or upload custom libraries to it, and then your workspace admin can attach the environment as the default for the workspace. All the libraries in the environment will then become available for use in any notebooks and Spark job definitions in the workspace. For more information on environments, see [create, configure, and use an environment in Microsoft Fabric](#).

For this tutorial, use `%pip install` to install the `imblearn` library in your notebook.

⚠ Note

The PySpark kernel restarts after `%pip install` runs. Install the needed libraries before you run any other cells.

Python

```
# Use pip to install libraries
%pip install imblearn
```

Step 2: Load the data

The dataset in `churn.csv` contains the churn status of 10,000 customers, along with 14 attributes that include:

- Credit score
- Geographical location (Germany, France, Spain)
- Gender (male, female)
- Age
- Tenure (number of years the person was a customer at that bank)
- Account balance
- Estimated salary
- Number of products that a customer purchased through the bank
- Credit card status (whether or not a customer has a credit card)
- Active member status (whether or not the person is an active bank customer)

The dataset also includes row number, customer ID, and customer surname columns. Values in these columns shouldn't influence a customer's decision to leave the bank.

A customer bank account closure event defines the churn for that customer. The dataset `Exited` column refers to the customer's abandonment. Since we have little context about these attributes, we don't need background information about the dataset. We want to understand how these attributes contribute to the `Exited` status.

Out of those 10,000 customers, only 2037 customers (roughly 20%) left the bank. Because of the class imbalance ratio, we recommend generation of synthetic data. Confusion matrix accuracy might not have relevance for imbalanced classification. We might want to measure the accuracy using the Area Under the Precision-Recall Curve (AUPRC).

- This table shows a preview of the `churn.csv` data:

CustomerID	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary
15634602	Hargrave	619	France	Female	42	2	0.00	1	1	1	101348.88
15647311	Hill	608	Spain	Female	41	1	83807.86	1	0	1	112542.58

Download the dataset and upload to the lakehouse

Define these parameters, so that you can use this notebook with different datasets:

Python

```
IS_CUSTOM_DATA = False # If TRUE, the dataset has to be uploaded manually

IS_SAMPLE = False # If TRUE, use only SAMPLE_ROWS of data for training; otherwise, use all data
SAMPLE_ROWS = 5000 # If IS_SAMPLE is True, use only this number of rows for training

DATA_ROOT = "/lakehouse/default"
DATA_FOLDER = "Files/churn" # Folder with data files
DATA_FILE = "churn.csv" # Data file name
```

This code downloads a publicly available version of the dataset, and then stores that dataset in a Fabric lakehouse:

i Important

[Add a lakehouse](#) to the notebook before you run it. Failure to do so will result in an error.

Python

```
import os, requests
if not IS_CUSTOM_DATA:
    # With an Azure Synapse Analytics blob, this can be done in one line

# Download demo data files into the lakehouse if they don't exist
remote_url = "https://synapseaisolutionsa.blob.core.windows.net/public/bankcustomerchurn"
file_list = ["churn.csv"]
download_path = "/lakehouse/default/Files/churn/raw"

if not os.path.exists("/lakehouse/default"):
    raise FileNotFoundError(
        "Default lakehouse not found, please add a lakehouse and restart the session."
    )
os.makedirs(download_path, exist_ok=True)
for fname in file_list:
    if not os.path.exists(f"{download_path}/{fname}"):
        r = requests.get(f"{remote_url}/{fname}", timeout=30)
        with open(f"{download_path}/{fname}", "wb") as f:
            f.write(r.content)
print("Downloaded demo data files into lakehouse.")
```

Start recording the time needed to run the notebook:

Python

```
# Record the notebook running time
import time

ts = time.time()
```

Read raw data from the lakehouse

This code reads raw data from the **Files** section of the lakehouse, and adds more columns for different date parts. Creation of the partitioned delta table uses this information.

Python

```
df = (
    spark.read.option("header", True)
```

```
.option("inferSchema", True)
.csv("Files/churn/raw/churn.csv")
.cache()
)
```

Create a pandas DataFrame from the dataset

This code converts the Spark DataFrame to a pandas DataFrame, for easier processing and visualization:

Python

```
df = df.toPandas()
```

Step 3: Perform exploratory data analysis

Display raw data

Explore the raw data with `display`, calculate some basic statistics, and show chart views. You must first import the required libraries for data visualization - for example, [seaborn](#). Seaborn is a Python data visualization library, and it provides a high-level interface to build visuals on dataframes and arrays.

Python

```
import seaborn as sns
sns.set_theme(style="whitegrid", palette="tab10", rc = {'figure.figsize':(9,6)})
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
from matplotlib import rc, rcParams
import numpy as np
import pandas as pd
import itertools
```

Python

```
display(df, summary=True)
```

Use Data Wrangler to perform initial data cleaning

Launch Data Wrangler directly from the notebook to explore and transform pandas dataframes. Select the Data Wrangler dropdown from the horizontal toolbar to browse the activated pandas DataFrames available for editing. Select the DataFrame you want to open in Data Wrangler.

Note

Data Wrangler cannot be opened while the notebook kernel is busy. The cell execution must finish before you launch Data Wrangler.
[Learn more about Data Wrangler](#).

Home Edit Run View

Standard session PySpark (Python) Environment Workspace default Data Wrangler Copilot

All sources Lakehouses + Lakehouse lakehouse7 Tables Files

Summary of observations from the exploratory analysis

- Most of the customers are from France comparing to Spain and Germany, while Spain has the highest percentage of missing values.
- Most of the customers have credit cards.
- There are customers whose age and credit score are above 60 and below 400, respectively.
- Very few customers have more than two of the bank's products.
- Customers who aren't active have a higher churn rate.
- Gender and tenure years don't seem to have an impact on customer's decision to close the account.

Step 4: Model training and tracking

With your data in place, you can now define the model. You'll apply Random Forrest and LightGBM to implement the models within a few lines of code.

Use `scikit-learn` and `lightgbm` to implement the models within a few lines of code.

Here you'll load the delta table from the lakehouse. You may use other delta tables consider-

Data Wrangler

pandas DataFrames

- X
- X_res
- X_test
- X_train
- df
- df_clean
- df_num_cols
- df_pred
- new_train

Spark DataFrames

- sparkDF

Choose custom sample Default: First 5,000 rows

After the Data Wrangler launches, a descriptive overview of the data panel is generated, as shown in the following images. The overview includes information about the dimension of the DataFrame, any missing values, etc. You can use Data Wrangler to generate the script to drop the rows with missing values, the duplicate rows and the columns with specific names. Then, you can copy the script into a cell. The next cell shows that copied script.

← Data Wrangler: df

Add code to notebook Copy code to clipboard Save as CSV Views

Other people in your organization may have access to notebooks and Spark job definitions in this workspace. Carefully review this item before running it.

Operations

Find and replace (4)

- Drop duplicate rows
- Drop missing values
- Fill missing values
- Find and replace

Format (7)

Formulas (4)

Numeric (4)

Schema (5)

Sort and filter (2)

Cleaning steps

Load data from variable

New operation

Choose an operation to get started

Or, start typing code to see a live preview of the transformation on your data (e.g. `df = df.drop(columns=['RowNumber'])`)

Summary

Data shape 5,000 rows x 14 columns

Columns 14

Rows (5,000)

Rows with missing values 0 (0.0%)

Duplicate rows 0 (0.0%)

Missing values (0)

No missing values

← Data Wrangler: df

Add code to notebook Copy code to clipboard Save as CSV Views

Other people in your organization may have access to notebooks and Spark job definitions in this workspace. Carefully review this item before running it.

Operations

Back to all operations

Drop duplicate rows

Drops duplicate rows. Duplicate rows are identified using the selected columns only.

Target columns Balance, NumOfProducts, HasCrCard, IsActive...

Apply Discard

Cleaning steps

Load data from variable

Drop duplicate rows

Drop duplicate rows across all columns

```
1 # Drop duplicate rows across all columns
2 df = df.drop_duplicates()
```

Data is unchanged

Preview code for all steps

Summary

RowCount RowNumber

Data type int32

Rows 5,000

Distinct values 5,000

Missing values 0

Statistics

Mean 2500.5

Standard deviation 1443.5200033252052

Minimum 1.0

25th percentile 1250.75

Median 2500.5

75th percentile 3750.25

Maximum 5000.0

Advanced Statistics

Kurtosis -1.2

Skew 0.0

```

def clean_data(df):
    # Drop rows with missing data across all columns
    df.dropna(inplace=True)
    # Drop duplicate rows in columns: 'RowNumber', 'CustomerId'
    df.drop_duplicates(subset=['RowNumber', 'CustomerId'], inplace=True)
    # Drop columns: 'RowNumber', 'CustomerId', 'Surname'
    df.drop(columns=['RowNumber', 'CustomerId', 'Surname'], inplace=True)
    return df

df_clean = clean_data(df.copy())

```

Determine attributes

This code determines the categorical, numerical, and target attributes:

Python

```

# Determine the dependent (target) attribute
dependent_variable_name = "Exited"
print(dependent_variable_name)
# Determine the categorical attributes
categorical_variables = [col for col in df_clean.columns if col in "0"
                         or df_clean[col].nunique() <=5
                         and col not in "Exited"]
print(categorical_variables)
# Determine the numerical attributes
numeric_variables = [col for col in df_clean.columns if df_clean[col].dtype != "object"
                      and df_clean[col].nunique() >5]
print(numeric_variables)

```

Show the five-number summary

Use box plots to show the five-number summary

- the minimum score
- first quartile
- median
- third quartile
- maximum score

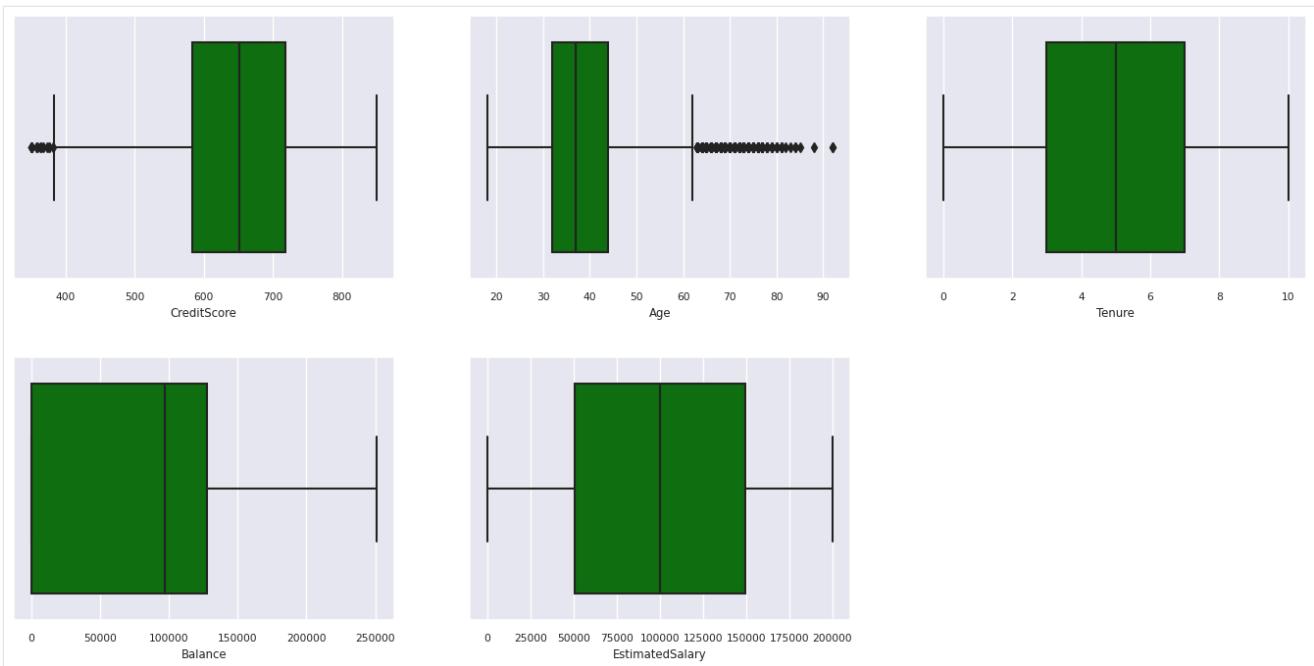
for the numerical attributes.

Python

```

df_num_cols = df_clean[numeric_variables]
sns.set(font_scale = 0.7)
fig, axes = plt.subplots(nrows = 2, ncols = 3, gridspec_kw = dict(hspace=0.3), figsize = (17,8))
fig.tight_layout()
for ax,col in zip(axes.flatten(), df_num_cols.columns):
    sns.boxplot(x = df_num_cols[col], color='green', ax = ax)
# fig.suptitle('visualize and compare the distribution and central tendency of numerical attributes', color = 'k', fontsize = 12)
fig.delaxes(axes[1,2])

```

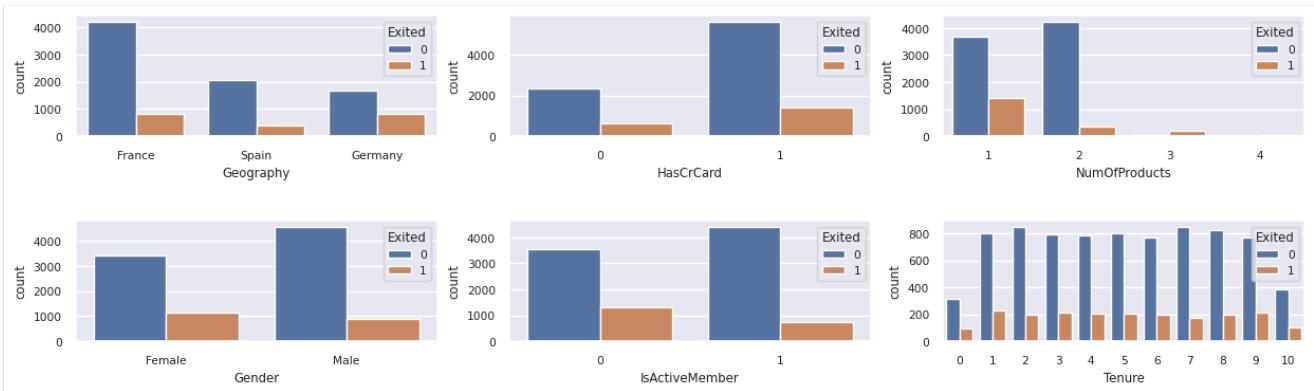


Show the distribution of exited and non-exited customers

Show the distribution of exited versus non-exited customers, across the categorical attributes:

Python

```
attr_list = ['Geography', 'Gender', 'HasCrCard', 'IsActiveMember', 'NumOfProducts', 'Tenure']
fig, axarr = plt.subplots(2, 3, figsize=(15, 4))
for ind, item in enumerate(attr_list):
    sns.countplot(x = item, hue = 'Exited', data = df_clean, ax = axarr[ind//2][ind%2])
fig.subplots_adjust(hspace=0.7)
```

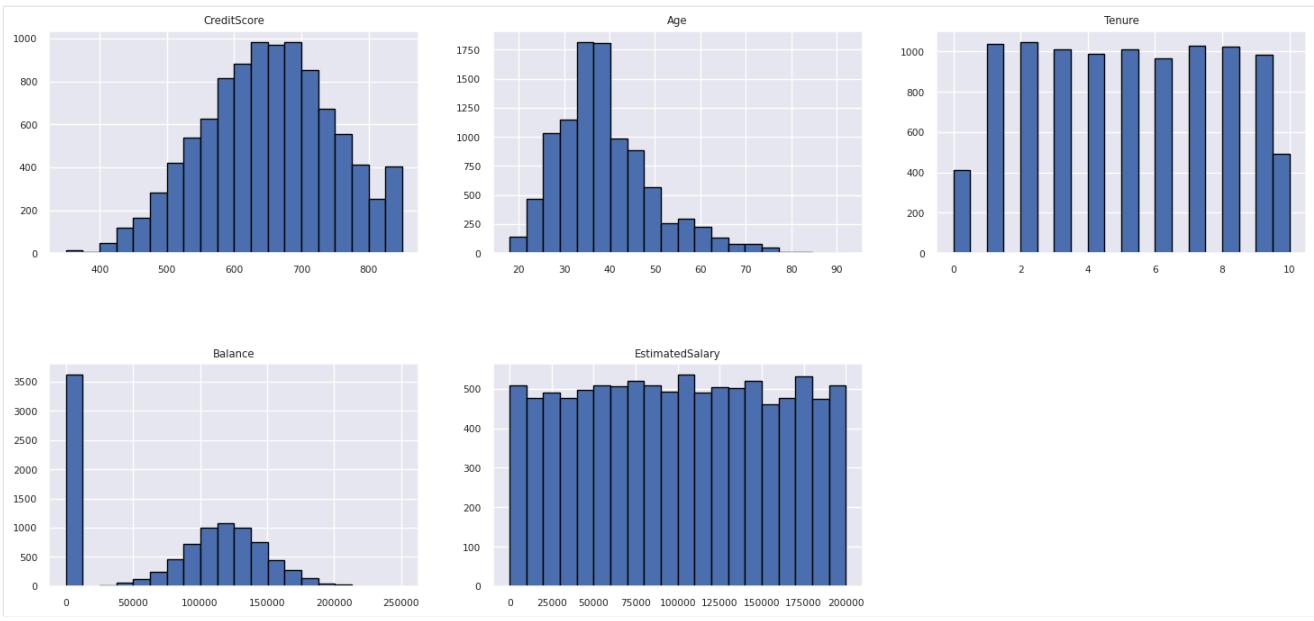


Show the distribution of numerical attributes

Use a histogram to show the frequency distribution of numerical attributes:

Python

```
columns = df_num_cols.columns[: len(df_num_cols.columns)]
fig = plt.figure()
fig.set_size_inches(18, 8)
length = len(columns)
for i,j in itertools.zip_longest(columns, range(length)):
    plt.subplot((length // 2), 3, j+1)
    plt.subplots_adjust(wspace = 0.2, hspace = 0.5)
    df_num_cols[i].hist(bins = 20, edgecolor = 'black')
    plt.title(i)
# fig = fig.suptitle('distribution of numerical attributes', color = 'r' ,fontsize = 14)
plt.show()
```



Perform feature engineering

This feature engineering generates new attributes based on the current attributes:

Python

```
df_clean["NewTenure"] = df_clean["Tenure"]/df_clean["Age"]
df_clean["NewCreditsScore"] = pd.qcut(df_clean['CreditScore'], 6, labels = [1, 2, 3, 4, 5, 6])
df_clean["NewAgeScore"] = pd.qcut(df_clean['Age'], 8, labels = [1, 2, 3, 4, 5, 6, 7, 8])
df_clean["NewBalanceScore"] = pd.qcut(df_clean['Balance'].rank(method="first"), 5, labels = [1, 2, 3, 4, 5])
df_clean["NewEstSalaryScore"] = pd.qcut(df_clean['EstimatedSalary'], 10, labels = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Use Data Wrangler to perform one-hot encoding

With the same steps to launch Data Wrangler, as discussed earlier, use the Data Wrangler to perform one-hot encoding. This cell shows the copied generated script for one-hot encoding:

← Data Wrangler: df

Add code to notebook Copy code to clipboard Save as CSV Views

Operations

Search for operations...

- Find and replace (4)
 - Drop duplicate rows
 - Drop missing values
 - Fill missing values
 - Find and replace
- Format (7)
- Formulas (4)
 - Multi-label binarizer
 - One-hot encode
 - Calculate text length
 - Create column from formula
- Numeric (4)

Cleaning steps

- Load data from variable
- New operation

2 New operation

Choose an operation to get
Or, start typing code to se

← Data Wrangler: df

Add code to notebook Copy code to clipboard Save as CSV Views

Operations

Back to all operations

One-hot encode

Split categorical data into a new column for each category, where each new column contains a 1 in rows that match that category, and 0 otherwise.

Target columns

Geography, Gender

Select all

Surname

Geography

Gender

index

RowNumber

CustomerID

CreditScore

Age

Tenure

Balance

NumOfProducts

HasCrCard

IsActiveMember

# RowNumber	# CustomerID	# Surname	# CreditScore	Geography_France
0	1564602	Hargrave	619	true
1	15647311	Hill	608	false
2	15619304	Ohio	502	true
3	15701354	Bono	699	true
4	15737886	Mitchell	850	false
5	15574012	Chu	645	false
6	15592531	Bartlett	822	true
7	15656148	Obinna	376	false
8	15792365	He	501	true
9	15592389	H?	684	true
10	15767821	Bearce	678	true
11				

2 One-hot encode

```

1 # One-hot encode columns: 'Geography', 'Gender'
2 import pandas as pd
3 for column in ['Geography', 'Gender']:
4     insert_loc = df.columns.get_loc(column)
5     df = pd.concat([df.iloc[:, :insert_loc], pd.get_dummies(df.loc[:, [column]]), df.iloc[:, insert_loc+1:]], axis=1)

```

Previewing

Summary RowNumber

Data type int32

Rows 5,000

Distinct values 5,000

Missing values 0

Statistics

Mean 2500.5

Standard deviation 1443.520033252052

Minimum 1.0

25th percentile 1250.75

Median 2500.5

75th percentile 3750.25

Maximum 5000.0

Advanced Statistics

Kurtosis -1.2

Skew 0.0

Python

```
df_clean = pd.get_dummies(df_clean, columns=['Geography', 'Gender'])
```

Create a delta table to generate the Power BI report

Python

```

table_name = "df_clean"
# Create a PySpark DataFrame from pandas
sparkDF=spark.createDataFrame(df_clean)

```

```
sparkDF.write.mode("overwrite").format("delta").save(f"Tables/{table_name}")
print(f"Spark DataFrame saved to delta table: {table_name}")
```

Summary of observations from the exploratory data analysis

- Most of the customers are from France. Spain has the lowest churn rate, compared to France and Germany.
- Most customers have credit cards
- Some customers are both over the age of 60 and have credit scores below 400. However, they can't be considered as outliers
- Very few customers have more than two bank products
- Inactive customers have a higher churn rate
- Gender and tenure years have little impact on a customer's decision to close a bank account

Step 4: Perform model training and tracking

With the data in place, you can now define the model. Apply random forest and LightGBM models in this notebook.

Use the scikit-learn and LightGBM libraries to implement the models, with a few lines of code. Additionally, use MLflow and Fabric Autologging to track the experiments.

This code sample loads the delta table from the lakehouse. You can use other delta tables that themselves use the lakehouse as the source.

Python

```
SEED = 12345
df_clean = spark.read.format("delta").load("Tables/df_clean").toPandas()
```

Generate an experiment for tracking and logging the models by using MLflow

This section shows how to generate an experiment, and it specifies the model and training parameters and the scoring metrics. Additionally, it shows how to train the models, log them, and save the trained models for later use.

Python

```
import mlflow

# Set up the experiment name
EXPERIMENT_NAME = "sample-bank-churn-experiment" # MLflow experiment name
```

Autologging automatically captures both the input parameter values and the output metrics of a machine learning model, as that model is trained. This information is then logged to your workspace, where the MLflow APIs or the corresponding experiment in your workspace can access and visualize it.

When complete, your experiment resembles this image:

The screenshot shows the Microsoft Fabric interface for an experiment named "sample-bank-churn-exp...". On the left, there's a sidebar with a search bar and a list of runs: "lgbm_sm" (4:04:49 PM), "feature_importance_gai...", "feature_importance_gai...", "feature_importance_spl...", "feature_importance_spl...", "metric_info.json", "model" (MLmodel, conda.yaml, metadata, model.pkl, python_env.yaml, requirements.txt), "rfc2_sm" (4:04:28 PM), "rfc1_sm" (4:04:00 PM), and "sad_spoon_0..." (4:03:47 PM). The main panel displays the properties of the "lgbm_sm" run, which started at 1/14/2025 4:04 PM, created by "Customer churn-429", and has a status of "Completed". It also shows "Run details" including "Run metrics (1)" (accuracy_score_X_test: 0.8675) and "Run parameters (23)". Parameters include boosting_type: gbdt, colsample_bytree: 1.0, learning_rate: 0.07, max_depth: 10, min_child_samples: 20, min_child_weight: 0.001, min_split_gain: 0.0, num_leaves: 31, and random_state: 42. To the right are buttons for "Compare runs" and "Save as an ML model". A magnifying glass icon is in the bottom right corner.

All the experiments with their respective names are logged, and you can track their parameters and performance metrics. To learn more about autologging, see [Autologging in Microsoft Fabric](#).

Set experiment and autologging specifications

Python

```
mlflow.set_experiment(EXPERIMENT_NAME) # Use a date stamp to append to the experiment
mlflow.autolog(exclusive=False)
```

Import scikit-learn and LightGBM

Python

```
# Import the required libraries for model training
from sklearn.model_selection import train_test_split
from lightgbm import LGBMClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, f1_score, precision_score, confusion_matrix, recall_score, roc_auc_score,
classification_report
```

Prepare training and test datasets

Python

```
y = df_clean["Exited"]
X = df_clean.drop("Exited",axis=1)
# Train/test separation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=SEED)
```

Apply SMOTE to the training data

Imbalanced classification has a problem, because it has too few examples of the minority class for a model to effectively learn the decision boundary. To handle this, Synthetic Minority Oversampling Technique (SMOTE) is the most widely used technique to synthesize new samples for the minority class. Access SMOTE with the `imblearn` library that you installed in step 1.

Apply SMOTE only to the training dataset. You must leave the test dataset in its original imbalanced distribution, to get a valid approximation of model performance on the original data. This experiment represents the situation in production.

Python

```
from collections import Counter
from imblearn.over_sampling import SMOTE
```

```

sm = SMOTE(random_state=SEED)
X_res, y_res = sm.fit_resample(X_train, y_train)
new_train = pd.concat([X_res, y_res], axis=1)

```

For more information, see [SMOTE](#) and [From random over-sampling to SMOTE and ADASYN](#). The imbalanced-learn website hosts these resources.

Train the model

Use Random Forest to train the model, with a maximum depth of four, and with four features:

Python

```

mlflow.sklearn.autolog(registered_model_name='rfc1_sm') # Register the trained model with autologging
rfc1_sm = RandomForestClassifier(max_depth=4, max_features=4, min_samples_split=3, random_state=1) # Pass hyperparameters
with mlflow.start_run(run_name="rfc1_sm") as run:
    rfc1_sm_run_id = run.info.run_id # Capture run_id for model prediction later
    print("run_id: {}; status: {}".format(rfc1_sm_run_id, run.info.status))
    # rfc1.fit(X_train,y_train) # Imbalanced training data
    rfc1_sm.fit(X_res, y_res.ravel()) # Balanced training data
    rfc1_sm.score(X_test, y_test)
    y_pred = rfc1_sm.predict(X_test)
    cr_rfc1_sm = classification_report(y_test, y_pred)
    cm_rfc1_sm = confusion_matrix(y_test, y_pred)
    roc_auc_rfc1_sm = roc_auc_score(y_res, rfc1_sm.predict_proba(X_res)[:, 1])

```

Use Random Forest to train the model, with a maximum depth of eight, and with six features:

Python

```

mlflow.sklearn.autolog(registered_model_name='rfc2_sm') # Register the trained model with autologging
rfc2_sm = RandomForestClassifier(max_depth=8, max_features=6, min_samples_split=3, random_state=1) # Pass hyperparameters
with mlflow.start_run(run_name="rfc2_sm") as run:
    rfc2_sm_run_id = run.info.run_id # Capture run_id for model prediction later
    print("run_id: {}; status: {}".format(rfc2_sm_run_id, run.info.status))
    # rfc2.fit(X_train,y_train) # Imbalanced training data
    rfc2_sm.fit(X_res, y_res.ravel()) # Balanced training data
    rfc2_sm.score(X_test, y_test)
    y_pred = rfc2_sm.predict(X_test)
    cr_rfc2_sm = classification_report(y_test, y_pred)
    cm_rfc2_sm = confusion_matrix(y_test, y_pred)
    roc_auc_rfc2_sm = roc_auc_score(y_res, rfc2_sm.predict_proba(X_res)[:, 1])

```

Train the model with LightGBM:

Python

```

# lgbm_model
mlflow.lightgbm.autolog(registered_model_name='lgbm_sm') # Register the trained model with autologging
lgbm_sm_model = LGBMClassifier(learning_rate = 0.07,
                               max_delta_step = 2,
                               n_estimators = 100,
                               max_depth = 10,
                               eval_metric = "logloss",
                               objective='binary',
                               random_state=42)

with mlflow.start_run(run_name="lgbm_sm") as run:
    lgbm1_sm_run_id = run.info.run_id # Capture run_id for model prediction later
    # lgbm_sm_model.fit(X_train,y_train) # Imbalanced training data
    lgbm_sm_model.fit(X_res, y_res.ravel()) # Balanced training data
    y_pred = lgbm_sm_model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    cr_lgbm_sm = classification_report(y_test, y_pred)
    cm_lgbm_sm = confusion_matrix(y_test, y_pred)
    roc_auc_lgbm_sm = roc_auc_score(y_res, lgbm_sm_model.predict_proba(X_res)[:, 1])

```

View the experiment artifact to track model performance

The experiment runs are automatically saved in the experiment artifact. You can find that artifact in the workspace. An artifact name is based on the name used to set the experiment. All of the trained models, their runs, performance metrics and model parameters are

logged on the experiment page.

To view your experiments:

1. On the left panel, select your workspace.
2. Find and select the experiment name, in this case, **sample-bank-churn-experiment**.

The screenshot shows the Azure Machine Learning studio interface. The left sidebar lists four runs: lgbm_sm, rfc2_sm, rfc1_sm, and sad_spoon_0... The main area displays the properties of the lgbm_sm run, including its start date (1/14/2025 4:04 PM), duration (15s), and status (Completed). It also shows run details such as accuracy_score_X_test (0.8675) and various run parameters like boosting_type (gbdt) and colsample_bytree (1.0).

Step 5: Evaluate and save the final machine learning model

Open the saved experiment from the workspace to select and save the best model:

```
# Define run_uri to fetch the model
# MLflow client: mlflow.model.url, list model
load_model_rfc1_sm = mlflow.sklearn.load_model(f"runs:{rfc1_sm_run_id}/model")
load_model_rfc2_sm = mlflow.sklearn.load_model(f"runs:{rfc2_sm_run_id}/model")
load_model_lgbm1_sm = mlflow.lightgbm.load_model(f"runs:{lgbm1_sm_run_id}/model")
```

Assess the performance of the saved models on the test dataset

```
ypred_rfc1_sm = load_model_rfc1_sm.predict(X_test) # Random forest with maximum depth of 4 and 4 features
ypred_rfc2_sm = load_model_rfc2_sm.predict(X_test) # Random forest with maximum depth of 8 and 6 features
ypred_lgbm1_sm = load_model_lgbm1_sm.predict(X_test) # LightGBM
```

Show true/false positives/negatives by using a confusion matrix

To evaluate the accuracy of the classification, build a script that plots the confusion matrix. You can also plot a confusion matrix using SynapseML tools, as shown in the [Fraud Detection sample](#).

```
def plot_confusion_matrix(cm, classes,
                         normalize=False,
                         title='Confusion matrix',
                         cmap=plt.cm.Blues):

    print(cm)
    plt.figure(figsize=(4,4))
    plt.rcParams.update({'font.size': 10})
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45, color="blue")
    plt.yticks(tick_marks, classes)
```

```

plt.yticks(tick_marks, classes, color="blue")

fmt = '.2f' if normalize else 'd'
thresh = cm.max() / 2.
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, format(cm[i, j], fmt),
             horizontalalignment="center",
             color="red" if cm[i, j] > thresh else "black")

plt.tight_layout()
plt.ylabel('True label')
plt.xlabel('Predicted label')

```

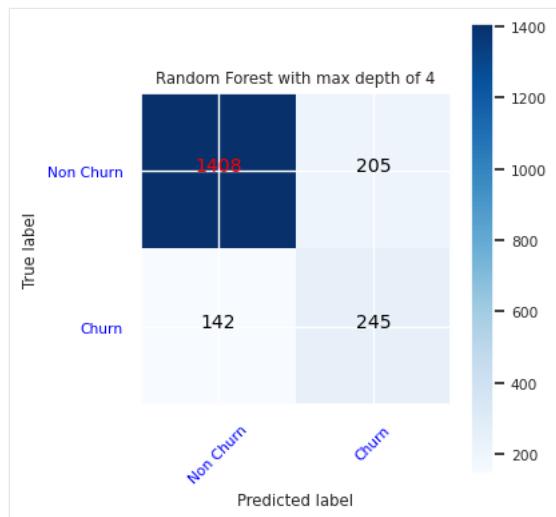
Create a confusion matrix for the random forest classifier, with a maximum depth of four, with four features:

Python

```

cfm = confusion_matrix(y_test, y_pred=ypred_rfc1_sm)
plot_confusion_matrix(cfm, classes=['Non Churn','Churn'],
                      title='Random Forest with max depth of 4')
tn, fp, fn, tp = cfm.ravel()

```



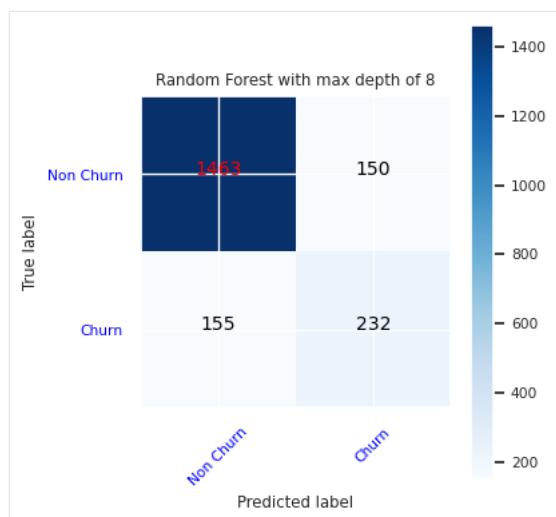
Create a confusion matrix for the random forest classifier with maximum depth of eight, with six features:

Python

```

cfm = confusion_matrix(y_test, y_pred=ypred_rfc2_sm)
plot_confusion_matrix(cfm, classes=['Non Churn','Churn'],
                      title='Random Forest with max depth of 8')
tn, fp, fn, tp = cfm.ravel()

```



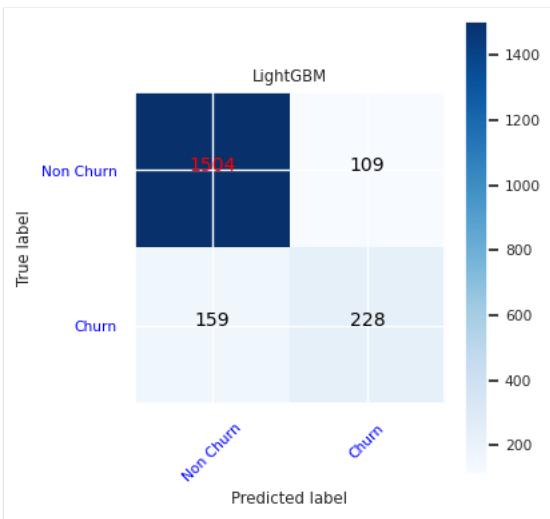
Create a confusion matrix for LightGBM:

Python

```

cfm = confusion_matrix(y_test, y_pred=ypred_lgbm1_sm)
plot_confusion_matrix(cfm, classes=['Non Churn','Churn'],
                      title='LightGBM')
tn, fp, fn, tp = cfm.ravel()

```



Save results for Power BI

Save the delta frame to the lakehouse, to move the model prediction results to a Power BI visualization.

Python

```

df_pred = X_test.copy()
df_pred['y_test'] = y_test
df_pred['ypred_rfc1_sm'] = ypred_rfc1_sm
df_pred['ypred_rfc2_sm'] = ypred_rfc2_sm
df_pred['ypred_lgbm1_sm'] = ypred_lgbm1_sm
table_name = "df_pred_results"
sparkDF=spark.createDataFrame(df_pred)
sparkDF.write.mode("overwrite").format("delta").option("overwriteSchema", "true").save(f"Tables/{table_name}")
print(f"Spark DataFrame saved to delta table: {table_name}")

```

Step 6: Access visualizations in Power BI

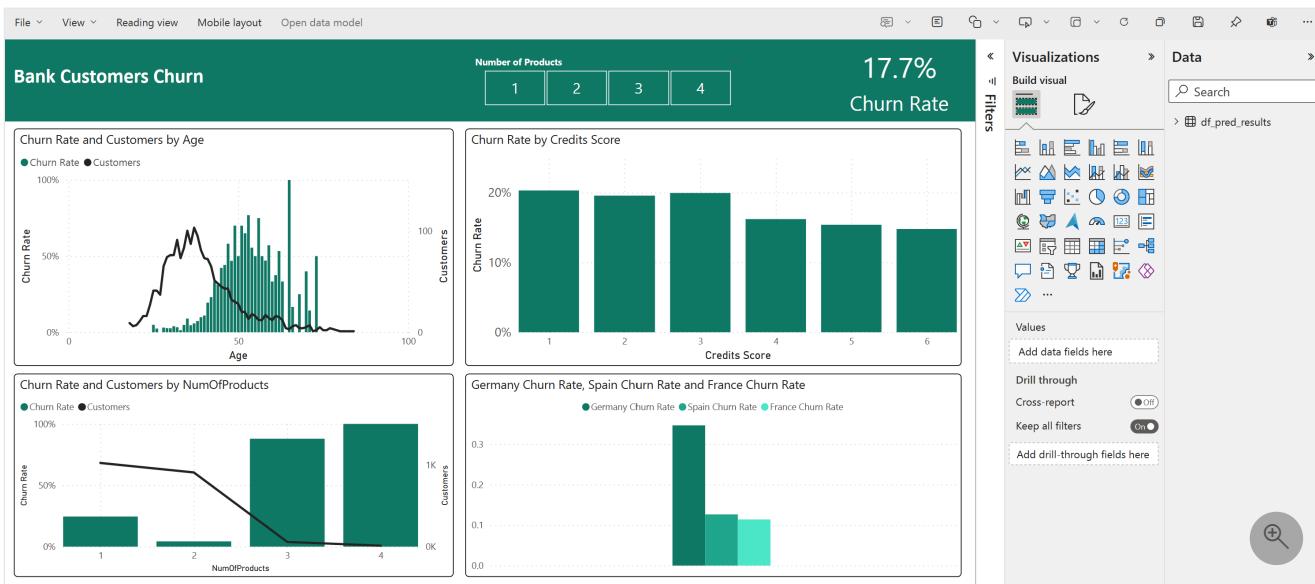
Access your saved table in Power BI:

1. On the left, select **OneLake**.
2. Select the lakehouse that you added to this notebook.
3. In the **Open this Lakehouse** section, select **Open**.
4. On the ribbon, select **New semantic model**. Select `df_pred_results`, and then select **Confirm** to create a new Power BI semantic model linked to the predictions.
5. Open new semantic model. You can find it in OneLake.
6. Select **Create New report** under file from the tools at the top of the semantic models page, to open the Power BI report authoring page.

The following screenshot shows some example visualizations. The data panel shows the delta tables and columns to select from a table. After selection of appropriate category (x) and value (y) axis, you can choose the filters and functions - for example, sum or average of the table column.

Note

In this screenshot, the illustrated example describes the analysis of the saved prediction results in Power BI:



However, for a real customer churn use-case, the user might need a more thorough set of requirements of the visualizations to create, based on subject matter expertise, and what the firm and business analytics team and firm have standardized as metrics.

The Power BI report shows that customers who use more than two of the bank products have a higher churn rate. However, few customers had more than two products. (See the plot in the bottom left panel.) The bank should collect more data, but should also investigate other features that correlate with more products.

Bank customers in Germany have a higher churn rate compared to customers in France and Spain. (See the plot in the bottom right panel). Based on the report results, an investigation into the factors that encouraged customers to leave might help.

There are more middle-aged customers (between 25 and 45). Customers between 45 and 60 tend to exit more.

Finally, customers with lower credit scores would most likely leave the bank for other financial institutions. The bank should explore ways to encourage customers with lower credit scores and account balances to stay with the bank.

Python

```
# Determine the entire runtime
print(f"Full run cost {int(time.time() - ts)} seconds.")
```

Related content

- [Machine learning model in Microsoft Fabric](#)
- [Train machine learning models](#)
- [Machine learning experiments in Microsoft Fabric](#)

Feedback

Was this page helpful?

Yes

No

Provide product feedback | Ask the community

Tutorial: Create, evaluate, and score a fraud detection model

Article • 01/17/2025

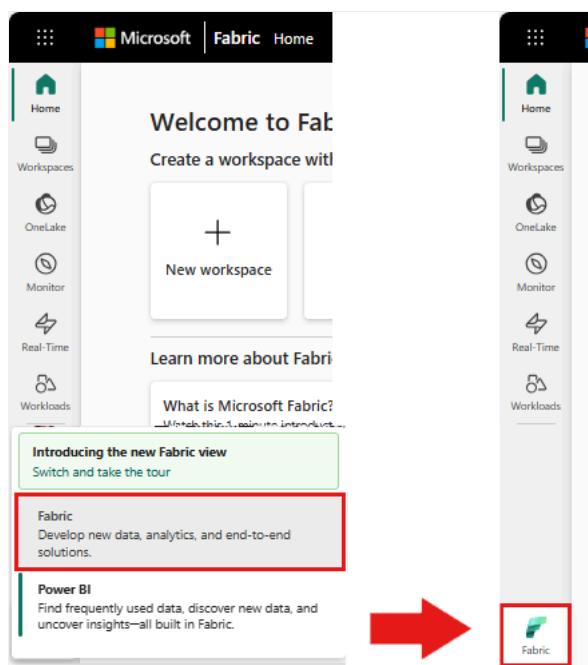
This tutorial presents an end-to-end example of a Synapse Data Science workflow, in Microsoft Fabric. The scenario builds a fraud detection model with machine learning algorithms trained on historical data. It then uses the model to detect future fraudulent transactions.

This tutorial covers these steps:

- ✓ Install custom libraries
- ✓ Load the data
- ✓ Understand and process the data through exploratory data analysis
- ✓ Use scikit-learn to train a machine learning model, and track experiments with the MLflow and Fabric Autologging features
- ✓ Save and register the machine learning model that has the highest performance
- ✓ Load the machine learning model for scoring and to make predictions

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- If necessary, create a Microsoft Fabric lakehouse as described in [Create a lakehouse in Microsoft Fabric](#).

Follow along in a notebook

You can choose one of these options to follow along in a notebook:

- Open and run the built-in notebook.
- Upload your notebook from GitHub.

Open the built-in notebook

The sample Fraud detection notebook accompanies this tutorial.

1. To open the sample notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#).
2. Make sure to [attach a lakehouse to the notebook](#) before you start running code.

Import the notebook from GitHub

The [Alsample - Fraud Detection.ipynb](#) notebook accompanies this tutorial.

- To open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#) to import the notebook to your workspace.
- If you'd rather copy and paste the code from this page, you can [create a new notebook](#).
- Be sure to [attach a lakehouse to the notebook](#) before you start running code.

Step 1: Install custom libraries

For machine learning model development or ad-hoc data analysis, you might need to quickly install a custom library for your Apache Spark session. You have two options to install libraries.

- Use the inline installation capabilities (`%pip` or `%conda`) of your notebook to install a library, in your current notebook only.
- Alternatively, you can create a Fabric environment, install libraries from public sources or upload custom libraries to it, and then your workspace admin can attach the environment as the default for the workspace. All the libraries in the environment will then become available for use in any notebooks and Spark job definitions in the workspace. For more information on environments, see [create, configure, and use an environment in Microsoft Fabric](#).

For this tutorial, use `%pip install` to install the `imblearn` library in your notebook.

① Note

The PySpark kernel restarts after `%pip install` runs. Install the needed libraries before you run any other cells.

Python

```
# Use pip to install imblearn
%pip install imblearn
```

Step 2: Load the data

The fraud detection dataset contains credit card transactions, from September 2013, that European cardholders made over the course of two days. The dataset contains only numerical features because of a Principal Component Analysis (PCA) transformation applied to the original features. PCA transformed all features except for `Time` and `Amount`. To protect confidentiality, we can't provide the original features or more background information about the dataset.

These details describe the dataset:

- The `V1`, `V2`, `V3`, ..., `V28` features are the principal components obtained with PCA
- The `Time` feature contains the elapsed seconds between a transaction and the first transaction in the dataset
- The `Amount` feature is the transaction amount. You can use this feature for example-dependent, cost-sensitive learning
- The `Class` column is the response (target) variable. It has the value `1` for fraud, and `0` otherwise

Only 492 transactions, out of 284,807 transactions total, are fraudulent. The dataset is highly imbalanced, because the minority (fraudulent) class accounts for only about 0.172% of the data.

This table shows a preview of the `creditcard.csv` data:

[Expand table

Time	V1	V2	V3	V4	V5	V6	V7
0	-1.3598071336738	-0.0727811733098497	2.53634673796914	1.37815522427443	-0.338320769942518	0.46238777762292	0.23959855406125
0	1.19185711131486	0.26615071205963	0.16648011335321	0.448154078460911	0.0600176492822243	-0.0823608088155687	-0.0788029833323

Download the dataset and upload to the lakehouse

Define these parameters, so that you can use this notebook with different datasets:

Python

```
IS_CUSTOM_DATA = False # If True, the dataset has to be uploaded manually

TARGET_COL = "Class" # Target column name
IS_SAMPLE = False # If True, use only <SAMPLE_ROWS> rows of data for training; otherwise, use all data
SAMPLE_ROWS = 5000 # If IS_SAMPLE is True, use only this number of rows for training

DATA_FOLDER = "Files/fraud-detection/" # Folder with data files
DATA_FILE = "creditcard.csv" # Data file name

EXPERIMENT_NAME = "aisample-fraud" # MLflow experiment name
```

This code downloads a publicly available version of the dataset, and then stores it in a Fabric lakehouse.

Important

Be sure to [add a lakehouse](#) to the notebook before you run it. Otherwise, you'll get an error.

Python

```
if not IS_CUSTOM_DATA:
    # Download data files into the lakehouse if they're not already there
    import os, requests

    remote_url = "https://synapseaisolutionsa.blob.core.windows.net/public/Credit_Card_Fraud_Detection"
    fname = "creditcard.csv"
    download_path = f"/lakehouse/default/{DATA_FOLDER}/raw"

    if not os.path.exists("/lakehouse/default"):
        raise FileNotFoundError("Default lakehouse not found, please add a lakehouse and restart the session.")
    os.makedirs(download_path, exist_ok=True)
    if not os.path.exists(f"{download_path}/{fname}"):
        r = requests.get(f"{remote_url}/{fname}", timeout=30)
        with open(f"{download_path}/{fname}", "wb") as f:
            f.write(r.content)
    print("Downloaded demo data files into lakehouse.")
```

Set up MLflow experiment tracking

The experiment tracking process saves all relevant experiment-related information for every experiment that you run. Sometimes, you have no way to obtain better results when you run a specific experiment. In those cases, you should stop the experiment and try a new one.

The Synapse Data Science experience in Microsoft Fabric includes an autologging feature. This feature reduces the amount of code needed to automatically log the parameters, metrics, and items of a machine learning model during training. The feature extends the MLflow autologging capabilities. It has deep integration in the Data Science experience.

With autologging, you can easily track and compare the performance of different models and experiments, without the need for manual tracking. For more information, see [Autologging in Microsoft Fabric](#).

To disable Microsoft Fabric autologging in a notebook session, call `mlflow.autolog()` and set `disable=True`:

Python

```
# Set up MLflow for experiment tracking
import mlflow

mlflow.set_experiment(EXPERIMENT_NAME)
mlflow.autolog(disable=True) # Disable MLflow autologging
```

Read raw data from the lakehouse

This code reads raw data from the lakehouse:

Python

```
df = (
    spark.read.format("csv")
    .option("header", "true")
    .option("inferSchema", True)
```

```
.load(f"{{DATA_FOLDER}}/raw/{{DATA_FILE}}")
.cache()
)
```

Step 3: Perform exploratory data analysis

In this section, you first explore the raw data and high-level statistics. Then, to transform the data, cast the columns into the correct types, and convert them from the Spark DataFrame into a pandas DataFrame for easier visualization. Finally, you explore and visualize the class distributions in the data.

Display the raw data

1. Explore the raw data, and view high-level statistics, with the `display` command. For more information about data visualization, see [Notebook visualization in Microsoft Fabric](#).

Python

```
display(df)
```

2. Print some basic information about the dataset:

Python

```
# Print dataset basic information
print("records read: " + str(df.count()))
print("Schema: ")
df.printSchema()
```

Transform the data

1. Cast the dataset columns into the correct types:

Python

```
import pyspark.sql.functions as F

df_columns = df.columns
df_columns.remove(TARGET_COL)

# Ensure that TARGET_COL is the last column
df = df.select(df_columns + [TARGET_COL]).withColumn(TARGET_COL, F.col(TARGET_COL).cast("int"))

if IS_SAMPLE:
    df = df.limit(SAMPLE_ROWS)
```

2. Convert the Spark DataFrame to a pandas DataFrame for easier visualization and processing:

Python

```
df_pd = df.toPandas()
```

Explore the class distribution in the dataset

1. Display the class distribution in the dataset:

Python

```
# The distribution of classes in the dataset
print('No Frauds', round(df_pd['Class'].value_counts()[0]/len(df_pd) * 100,2), '% of the dataset')
print('Frauds', round(df_pd['Class'].value_counts()[1]/len(df_pd) * 100,2), '% of the dataset')
```

The code returns this dataset class distribution: 99.83% `No Frauds` and 0.17% `Frauds`. This class distribution shows that most of the transactions are nonfraudulent. Therefore, data preprocessing is required before model training, to avoid overfitting.

2. Use a plot to show the class imbalance in the dataset, by viewing the distribution of fraudulent versus nonfraudulent transactions:

Python

```
import seaborn as sns
import matplotlib.pyplot as plt

colors = ["#0101DF", "#DF0101"]
sns.countplot(x='Class', data=df_pd, palette=colors)
plt.title('Class Distributions \n (0: No Fraud || 1: Fraud)', fontsize=10)
```

3. Show the five-number summary (minimum score, first quartile, median, third quartile, and maximum score) for the transaction amount, with box plots:

Python

```
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(12,5))
s = sns.boxplot(ax = ax1, x="Class", y="Amount", hue="Class", data=df_pd, palette="PRGn", showfliers=True) # Remove
outliers from the plot
s = sns.boxplot(ax = ax2, x="Class", y="Amount", hue="Class", data=df_pd, palette="PRGn", showfliers=False) # Keep
outliers from the plot
plt.show()
```

For highly imbalanced data, box plots might not show accurate insights. However, you can address the `Class` imbalance problem first, and then create the same plots for more accurate insights.

Step 4: Train and evaluate the models

Here, you train a LightGBM model to classify the fraud transactions. You train a LightGBM model on both the imbalanced dataset and the balanced dataset. Then, you compare the performance of both models.

Prepare training and test datasets

Before training, split the data into the training and test datasets:

Python

```
# Split the dataset into training and testing sets
from sklearn.model_selection import train_test_split

train, test = train_test_split(df_pd, test_size=0.15)
feature_cols = [c for c in df_pd.columns.tolist() if c not in [TARGET_COL]]
```

Apply SMOTE to the training dataset

The `imblearn` library uses the Synthetic Minority Oversampling Technique (SMOTE) approach to address the problem of imbalanced classification. Imbalanced classification happens when too few examples of the minority class are available, for a model to effectively learn the decision boundary. SMOTE is the most widely used approach to synthesize new samples for the minority class.

Apply SMOTE only to the training dataset, instead of the test dataset. When you score the model with the test data, you need an approximation of the model performance on unseen data in production. For a valid approximation, your test data relies on the original imbalanced distribution to represent production data as closely as possible.

Python

```
# Apply SMOTE to the training data
import pandas as pd
from collections import Counter
from imblearn.over_sampling import SMOTE

X = train[feature_cols]
y = train[TARGET_COL]
print("Original dataset shape %s" % Counter(y))

sm = SMOTE(random_state=42)
X_res, y_res = sm.fit_resample(X, y)
print("Resampled dataset shape %s" % Counter(y_res))

new_train = pd.concat([X_res, y_res], axis=1)
```

For more information about SMOTE, see the [scikit-learn reference page for the SMOTE method](#) and the [scikit-learn user guide on oversampling](#) resources.

Train machine learning models and run experiments

Apache Spark, in Microsoft Fabric, enables machine learning with big data. With Apache Spark, you can get valuable insights from large amounts of structured, unstructured, and fast-moving data.

You have several available options to train machine learning models with Apache Spark in Microsoft Fabric: Apache Spark MLlib, SynapseML, and other open-source libraries. For more information, see [Train machine learning models in Microsoft Fabric](#).

A *machine learning experiment* serves as the primary unit of organization and control for all related machine learning runs. A *run* corresponds to a single execution of model code. Machine learning *experiment tracking* involves the management of all the experiments and their components, such as parameters, metrics, models, and other artifacts.

For experiment tracking, you can organize all the required components of a specific machine learning experiment. Additionally, you can easily reproduce past results with saved experiments. For more information about machine learning experiments, see [Machine learning experiments in Microsoft Fabric](#).

1. To track more metrics, parameters, and files, set `exclusive=False` to update the MLflow autologging configuration:

```
Python
```

```
mlflow.autolog(exclusive=False)
```

2. Train two models with LightGBM. One model handles the imbalanced dataset, and the other model handles the balanced dataset (via SMOTE). Then compare the performance of the two models.

```
Python
```

```
import lightgbm as lgb

model = lgb.LGBMClassifier(objective="binary") # Imbalanced dataset
smote_model = lgb.LGBMClassifier(objective="binary") # Balanced dataset
```

```
Python
```

```
# Train LightGBM for both imbalanced and balanced datasets and define the evaluation metrics
print("Start training with imbalanced data:\n")
with mlflow.start_run(run_name="raw_data") as raw_run:
    model = model.fit(
        train[feature_cols],
        train[TARGET_COL],
        eval_set=[(test[feature_cols], test[TARGET_COL])],
        eval_metric="auc",
        callbacks=[
            lgb.log_evaluation(10),
        ],
    )

print("\n\nStart training with balanced data:\n")
with mlflow.start_run(run_name="smote_data") as smote_run:
    smote_model = smote_model.fit(
        new_train[feature_cols],
        new_train[TARGET_COL],
        eval_set=[(test[feature_cols], test[TARGET_COL])],
        eval_metric="auc",
        callbacks=[
            lgb.log_evaluation(10),
        ],
    )
```

Determine feature importance for training

1. Determine feature importance for the model that you trained on the imbalanced dataset:

```
Python
```

```
with mlflow.start_run(run_id=raw_run.info.run_id):
    importance = lgb.plot_importance(
```

```

        model, title="Feature importance for imbalanced data"
    )
importance.figure.savefig("feauture_importance.png")
mlflow.log_figure(importance.figure, "feature_importance.png")

```

2. Determine feature importance for the model that you trained on balanced data. SMOTE generated the balanced data:

Python

```

with mlflow.start_run(run_id=smote_run.info.run_id):
    smote_importance = lgb.plot_importance(
        smote_model, title="Feature importance for balanced (via SMOTE) data"
    )
    smote_importance.figure.savefig("feauture_importance_smote.png")
    mlflow.log_figure(smote_importance.figure, "feauture_importance_smote.png")

```

To train a model with the imbalanced dataset, the important features have significant differences when compared with a model trained with the balanced dataset.

Evaluate the models

Here, you evaluate the two trained models:

- `model` trained on raw, imbalanced data
- `smote_model` trained on balanced data

Compute model metrics

1. Define a `prediction_to_spark` function that performs predictions, and converts the prediction results into a Spark DataFrame. You can then compute model statistics on the prediction results with [SynapseML](#).

Python

```

from pyspark.sql.functions import col
from pyspark.sql.types import IntegerType, DoubleType

def prediction_to_spark(model, test):
    predictions = model.predict(test[feature_cols], num_iteration=model.best_iteration_)
    predictions = tuple(zip(test[TARGET_COL].tolist(), predictions.tolist()))
    dataColumns = [TARGET_COL, "prediction"]
    predictions = (
        spark.createDataFrame(data=predictions, schema=dataColumns)
        .withColumn(TARGET_COL, col(TARGET_COL).cast(IntegerType()))
        .withColumn("prediction", col("prediction").cast(DoubleType()))
    )
    return predictions

```

2. Use the `prediction_to_spark` function to perform predictions with the two models, `model` and `smote_model`:

Python

```

predictions = prediction_to_spark(model, test)
smote_predictions = prediction_to_spark(smote_model, test)
predictions.limit(10).toPandas()

```

3. Compute metrics for the two models:

Python

```

from synapse.ml.train import ComputeModelStatistics

metrics = ComputeModelStatistics(
    evaluationMetric="classification", labelCol=TARGET_COL, scoredLabelsCol="prediction"
).transform(predictions)

smote_metrics = ComputeModelStatistics(
    evaluationMetric="classification", labelCol=TARGET_COL, scoredLabelsCol="prediction"
).transform(smote_predictions)
display(metrics)

```

Evaluate model performance with a confusion matrix

A *confusion matrix* displays the number of

- true positives (TP)
- true negatives (TN)
- false positives (FP)
- false negatives (FN)

that a model produces when scored with test data. For binary classification, the model returns a `2x2` confusion matrix. For multiclass classification, the model returns an `nxn` confusion matrix, where `n` is the number of classes.

1. Use a confusion matrix to summarize the performances of the trained machine learning models on the test data:

Python

```
# Collect confusion matrix values
cm = metrics.select("confusion_matrix").collect()[0][0].toArray()
smote_cm = smote_metrics.select("confusion_matrix").collect()[0][0].toArray()
print(cm)
```

2. Plot the confusion matrix for the predictions of `smote_model` (trained on balanced data):

Python

```
# Plot the confusion matrix
import seaborn as sns

def plot(cm):
    """
    Plot the confusion matrix.
    """
    sns.set(rc={"figure.figsize": (5, 3.5)})
    ax = sns.heatmap(cm, annot=True, fmt=".20g")
    ax.set_title("Confusion Matrix")
    ax.set_xlabel("Predicted label")
    ax.set_ylabel("True label")
    return ax

with mlflow.start_run(run_id=smote_run.info.run_id):
    ax = plot(smote_cm)
    mlflow.log_figure(ax.figure, "ConfusionMatrix.png")
```

3. Plot the confusion matrix for the predictions of `model` (trained on raw, imbalanced data):

Python

```
with mlflow.start_run(run_id=raw_run.info.run_id):
    ax = plot(cm)
    mlflow.log_figure(ax.figure, "ConfusionMatrix.png")
```

Evaluate model performance with AUC-ROC and AUPRC measures

The Area Under the Curve Receiver Operating Characteristic (AUC-ROC) measure assesses the performance of binary classifiers. The AUC-ROC chart visualizes the trade-off between the true positive rate (TPR) and the false positive rate (FPR).

In some cases, it's more appropriate to evaluate your classifier based on the Area Under the Precision-Recall Curve (AUPRC) measure. The AUPRC curve combines these rates:

- The precision, or the positive predictive value (PPV)
- The recall, or TPR

To evaluate performance with the AUC-ROC and AUPRC measures:

1. Define a function that returns the AUC-ROC and AUPRC measures:

Python

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator

def evaluate(predictions):
```

```

"""
Evaluate the model by computing AUROC and AUPRC with the predictions.
"""

# Initialize the binary evaluator
evaluator = BinaryClassificationEvaluator(rawPredictionCol="prediction", labelCol=TARGET_COL)

_evaluator = lambda metric: evaluator.setMetricName(metric).evaluate(predictions)

# Calculate AUROC, baseline 0.5
auroc = _evaluator("areaUnderROC")
print(f"The AUROC is: {auroc:.4f}")

# Calculate AUPRC, baseline positive rate (0.172% in the data)
auprc = _evaluator("areaUnderPR")
print(f"The AUPRC is: {auprc:.4f}")

return auroc, auprc

```

2. Log the AUC-ROC and AUPRC metrics for the model that you trained on imbalanced data:

Python

```

with mlflow.start_run(run_id=raw_run.info.run_id):
    auroc, auprc = evaluate(predictions)
    mlflow.log_metrics({"AUPRC": auprc, "AUROC": auroc})
    mlflow.log_params({"Data_Enhancement": "None", "DATA_FILE": DATA_FILE})

```

3. Log the AUC-ROC and AUPRC metrics for the model that you trained on balanced data:

Python

```

with mlflow.start_run(run_id=smote_run.info.run_id):
    auroc, auprc = evaluate(smote_predictions)
    mlflow.log_metrics({"AUPRC": auprc, "AUROC": auroc})
    mlflow.log_params({"Data_Enhancement": "SMOTE", "DATA_FILE": DATA_FILE})

```

The model trained on the balanced data returns higher AUC-ROC and AUPRC values compared to the model trained on the imbalanced data. Based on these measures, SMOTE seems like an effective technique to enhance model performance when working with highly imbalanced data.

As the next image shows, any experiment is logged with its respective name. You can track the experiment parameters and performance metrics in your workspace.

Run	Metrics	Parameters
smote_data	valid_0-auc, valid_0-binary_logloss, AUPRC, AUROC	boosting_type: gbdt, colsample_bytree: 1.0, learning_rate: 0.1, max_depth: -1, min_child_samples: 20, min_child_weight: 0.001
raw_data		boosting_type: gbdt, colsample_bytree: 1.0, learning_rate: 0.1, max_depth: -1, min_child_samples: 20, min_child_weight: 0.001

This image shows the performance metrics for the model trained on the balanced dataset (in Version 2):

You can select **Version 1** to see the metrics for the model trained on the imbalanced dataset. When you compare the metrics, the AUROC is higher for the model trained with the balanced dataset. These results indicate that this model is better at correctly predicting **0** classes as **0**, and predicting **1** classes as **1**.

Step 5: Register the models

Use MLflow to register the two models:

Python

```
# Register the model
registered_model_name = f"{EXPERIMENT_NAME}-lightgbm"

raw_model_uri = "runs:{}/model".format(raw_run.info.run_id)
mlflow.register_model(raw_model_uri, registered_model_name)

smote_model_uri = "runs:{}/model".format(smote_run.info.run_id)
mlflow.register_model(smote_model_uri, registered_model_name)
```

Step 6: Save the prediction results

Microsoft Fabric allows users to operationalize machine learning models with the `PREDICT` scalable function. This function supports batch scoring (or batch inferencing) in any compute engine.

You can generate batch predictions directly from the Microsoft Fabric notebook or from a model's item page. For more information about `PREDICT`, see [Model scoring with PREDICT in Microsoft Fabric](#).

1. Load the better-performing model (**Version 2**) for batch scoring, and generate the prediction results:

Python

```
from synapse.ml.predict import MLFlowTransformer

spark.conf.set("spark.synapse.ml.predict.enabled", "true")

model = MLFlowTransformer(
    inputCols=feature_cols,
    outputCol="prediction",
    modelName=f"{EXPERIMENT_NAME}-lightgbm",
    modelVersion=2,
)

test_spark = spark.createDataFrame(data=test, schema=test.columns.to_list())

batch_predictions = model.transform(test_spark)
```

2. Save predictions to the lakehouse:

Python

```
# Save the predictions to the lakehouse
batch_predictions.write.format("delta").mode("overwrite").save(f"{DATA_FOLDER}/predictions/batch_predictions")
```

Related content

- [How to use Microsoft Fabric notebooks](#)
- [Machine learning model in Microsoft Fabric](#)
- [Train machine learning models](#)
- [Machine learning experiments in Microsoft Fabric](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Tutorial: create, evaluate, and score a machine fault detection model

Article • 01/17/2025

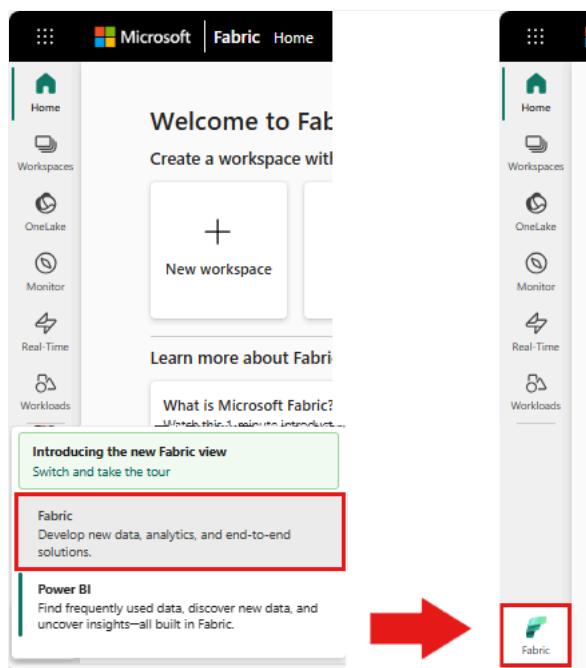
This tutorial presents an end-to-end example of a Synapse Data Science workflow in Microsoft Fabric. The scenario uses machine learning for a more systematic approach to fault diagnosis, to proactively identify issues and to take actions before an actual machine failure. The goal is to predict whether a machine would experience a failure based on process temperature, rotational speed, etc.

This tutorial covers these steps:

- ✓ Install custom libraries
- ✓ Load and process the data
- ✓ Understand the data through exploratory data analysis
- ✓ Use scikit-learn, LightGBM, and MLflow to train machine learning models, and use the Fabric Autologging feature to track experiments
- ✓ Score the trained models with the Fabric `PREDICT` feature, save the best model, and load that model for predictions
- ✓ Show the loaded model performance with Power BI visualizations

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- If necessary, create a Microsoft Fabric lakehouse as described in [Create a lakehouse in Microsoft Fabric](#).

Follow along in a notebook

You can choose one of these options to follow along in a notebook:

- Open and run the built-in notebook.
- Upload your notebook from GitHub.

Open the built-in notebook

The sample **Machine failure** notebook accompanies this tutorial.

1. To open the sample notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#).

2. Make sure to [attach a lakehouse to the notebook](#) before you start running code.

Import the notebook from GitHub

The [AI Sample - Predictive Maintenance](#) notebook accompanies this tutorial.

- To open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#) to import the notebook to your workspace.
- If you'd rather copy and paste the code from this page, you can [create a new notebook](#).
- Be sure to [attach a lakehouse to the notebook](#) before you start running code.

Step 1: Install custom libraries

For machine learning model development or ad-hoc data analysis, you might need to quickly install a custom library for your Apache Spark session. You have two options to install libraries.

- Use the inline installation capabilities (`%pip` or `%conda`) of your notebook to install a library, in your current notebook only.
- Alternatively, you can create a Fabric environment, install libraries from public sources or upload custom libraries to it, and then your workspace admin can attach the environment as the default for the workspace. All the libraries in the environment will then become available for use in any notebooks and Spark job definitions in the workspace. For more information on environments, see [create, configure, and use an environment in Microsoft Fabric](#).

For this tutorial, use `%pip install` to install the `imblearn` library in your notebook.

⚠ Note

The PySpark kernel restarts after `%pip install` runs. Install the needed libraries before you run any other cells.

Python

```
# Use pip to install imblearn
%pip install imblearn
```

Step 2: Load the data

The dataset simulates logging of a manufacturing machine's parameters as a function of time, which is common in industrial settings. It consists of 10,000 data points stored as rows with features as columns. The features include:

- A Unique Identifier (UID) that ranges from 1 to 10000
- Product ID, consisting of a letter L (for low), M (for medium), or H (for high), to indicate the product quality variant, and a variant-specific serial number. Low, medium, and high-quality variants make up 60%, 30%, and 10% of all products, respectively
- Air temperature, in degrees Kelvin (K)
- Process Temperature, in degrees Kelvin
- Rotational Speed, in revolutions per minute (RPM)
- Torque, in Newton-Meters (Nm)
- Tool wear, in minutes. The quality variants H, M, and L add 5, 3, and 2 minutes of tool wear respectively to the tool used in the process
- A Machine Failure Label, to indicate whether the machine failed in the specific data point. This specific data point can have any of the following five independent failure modes:
 - Tool Wear Failure (TWF): the tool is replaced or fails at a randomly selected tool wear time, between 200 and 240 minutes
 - Heat Dissipation Failure (HDF): heat dissipation causes a process failure if the difference between the air temperature and the process temperature is less than 8.6 K, and the tool's rotational speed is less than 1380 RPM
 - Power Failure (PWF): the product of torque and rotational speed (in rad/s) equals the power required for the process. The process fails if this power falls below 3,500 W or exceeds 9,000 W
 - OverStrain Failure (OSF): if the product of tool wear and torque exceeds 11,000 minimum Nm for the L product variant (12,000 for M, 13,000 for H), the process fails due to overstrain

- o Random Failures (RNF): each process has a failure chance of 0.1%, regardless of the process parameters

 Note

If at least one of the above failure modes is true, the process fails, and the "machine failure" label is set to 1. The machine learning method can't determine which failure mode caused the process failure.

Download the dataset and upload to the lakehouse

Connect to the Azure Open Datasets container, and load the Predictive Maintenance dataset. This code downloads a publicly available version of the dataset, and then stores it in a Fabric lakehouse:

 Important

Add a lakehouse to the notebook before you run it. Otherwise, you'll get an error. For information about adding a lakehouse, see [Connect lakehouses and notebooks](#).

Python

```
# Download demo data files into the lakehouse if they don't exist
import os, requests
DATA_FOLDER = "Files/predictive_maintenance/" # Folder that contains the dataset
DATA_FILE = "predictive_maintenance.csv" # Data file name
remote_url = "https://synapseaisolutionsa.blob.core.windows.net/public/MachineFaultDetection"
file_list = ["predictive_maintenance.csv"]
download_path = f"/lakehouse/default/{DATA_FOLDER}/raw"

if not os.path.exists("/lakehouse/default"):
    raise FileNotFoundError(
        "Default lakehouse not found, please add a lakehouse and restart the session."
    )
os.makedirs(download_path, exist_ok=True)
for fname in file_list:
    if not os.path.exists(f"{download_path}/{fname}"):
        r = requests.get(f'{remote_url}/{fname}', timeout=30)
        with open(f'{download_path}/{fname}', "wb") as f:
            f.write(r.content)
print("Downloaded demo data files into lakehouse.")
```

After you download the dataset into the lakehouse, you can load it as a Spark DataFrame:

Python

```
df = (
    spark.read.option("header", True)
    .option("inferSchema", True)
    .csv(f'{DATA_FOLDER}raw/{DATA_FILE}')
    .cache()
)
df.show(5)
```

This table shows a preview of the data:

 Expand table

UDI	Product ID	Type	Air temperature [K]	Process temperature [K]	Rotational speed [rpm]	Torque [Nm]	Tool wear [min]	Target	Failure Type
1	M14860	M	298.1	308.6	1551	42.8	0	0	No Failure
2	L47181	L	298.2	308.7	1408	46.3	3	0	No Failure
3	L47182	L	298.1	308.5	1498	49.4	5	0	No Failure
4	L47183	L	298.2	308.6	1433	39.5	7	0	No Failure
5	L47184	L	298.2	308.7	1408	40.0	9	0	No Failure

Write a Spark DataFrame to a lakehouse delta table

Format the data (for example, replace the spaces with underscores) to facilitate Spark operations in subsequent steps:

Python

```
# Replace the space in the column name with an underscore to avoid an invalid character while saving
df = df.toDF(*[c.replace(' ', '_') for c in df.columns])
table_name = "predictive_maintenance_data"
df.show(5)
```

This table shows a preview of the data with reformatted column names:

[Expand table](#)

UDI	Product_ID	Type	Air_temperature_[K]	Process_temperature_[K]	Rotational_speed_[rpm]	Torque_[Nm]	Tool_wear_[min]	Target	Failure_Ty
1	M14860	M	298.1	308.6	1551	42.8	0	0	No Failure
2	L47181	L	298.2	308.7	1408	46.3	3	0	No Failure
3	L47182	L	298.1	308.5	1498	49.4	5	0	No Failure
4	L47183	L	298.2	308.6	1433	39.5	7	0	No Failure
5	L47184	L	298.2	308.7	1408	40.0	9	0	No Failure

Python

```
# Save data with processed columns to the lakehouse
df.write.mode("overwrite").format("delta").save(f"Tables/{table_name}")
print(f"Spark DataFrame saved to delta table: {table_name}")
```

Step 3: Preprocess data and perform exploratory data analysis

Convert the Spark DataFrame to a pandas DataFrame, to use Pandas-compatible popular plotting libraries.

 Tip

For a large dataset, you might need to load a portion of that dataset.

Python

```
data = spark.read.format("delta").load("Tables/predictive_maintenance_data")
SEED = 1234
df = data.toPandas()
df.drop(['UDI', 'Product_ID'], axis=1, inplace=True)
# Rename the Target column to IsFail
df = df.rename(columns = {'Target': "IsFail"})
df.info()
```

Convert specific columns of the dataset to floats or integer types as required, and map strings ('L', 'M', 'H') to numerical values (0, 1, 2):

Python

```
# Convert temperature, rotational speed, torque, and tool wear columns to float
df['Air_temperature_[K]'] = df['Air_temperature_[K]'].astype(float)
df['Process_temperature_[K]'] = df['Process_temperature_[K]'].astype(float)
df['Rotational_speed_[rpm]'] = df['Rotational_speed_[rpm]'].astype(float)
df['Torque_[Nm]'] = df['Torque_[Nm]'].astype(float)
df['Tool_wear_[min]'] = df['Tool_wear_[min]'].astype(float)

# Convert the 'Target' column to an integer
df['IsFail'] = df['IsFail'].astype(int)
# Map 'L', 'M', 'H' to numerical values
df['Type'] = df['Type'].map({'L': 0, 'M': 1, 'H': 2})
```

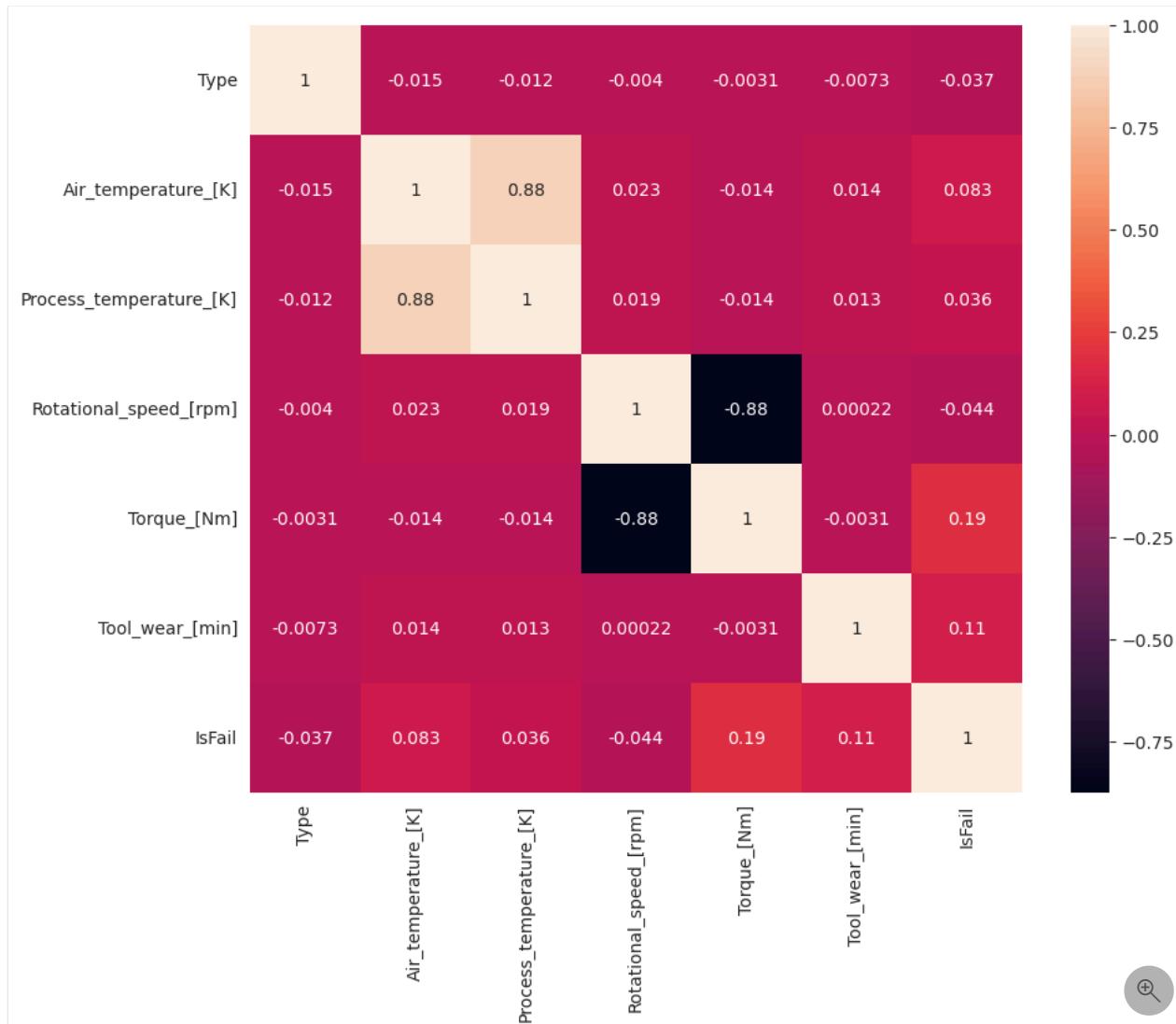
Explore data through visualizations

Python

```
# Import packages and set plotting style
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
sns.set_style('darkgrid')

# Create the correlation matrix
corr_matrix = df.corr(numeric_only=True)

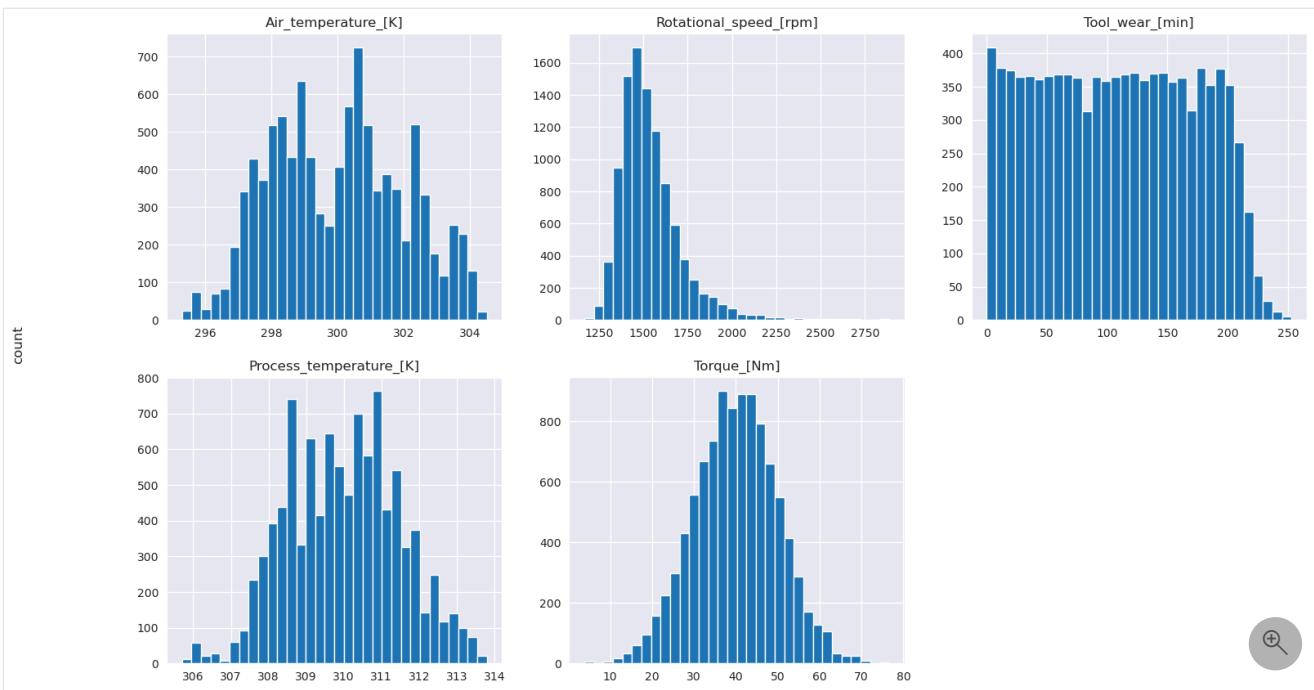
# Plot a heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, annot=True)
plt.show()
```



As expected, failure (`IsFail`) correlates with the selected features (columns). The correlation matrix shows that `Air_temperature`, `Process_temperature`, `Rotational_speed`, `Torque`, and `Tool_wear` have the highest correlation with the `IsFail` variable.

Python

```
# Plot histograms of select features
fig, axes = plt.subplots(2, 3, figsize=(18,10))
columns = ['Air_temperature_[K]', 'Process_temperature_[K]', 'Rotational_speed_[rpm]', 'Torque_[Nm]', 'Tool_wear_[min]']
data=df.copy()
for ind, item in enumerate (columns):
    column = columns[ind]
    df_column = data[column]
    df_column.hist(ax = axes[ind%2][ind//2], bins=32).set_title(item)
fig.supylabel('count')
fig.subplots_adjust(hspace=0.2)
fig.delaxes(axes[1,2])
```



As the plotted graphs show, the `Air_temperature`, `Process_temperature`, `Rotational_speed`, `Torque`, and `Tool_wear` variables aren't sparse. They seem to have good continuity in the feature space. These plots confirm that training a machine learning model on this dataset likely produces reliable results that can generalize to a new dataset.

Inspect the target variable for class imbalance

Count the number of samples for failed and unfailed machines, and inspect the data balance for each class (`IsFail=0`, `IsFail=1`):

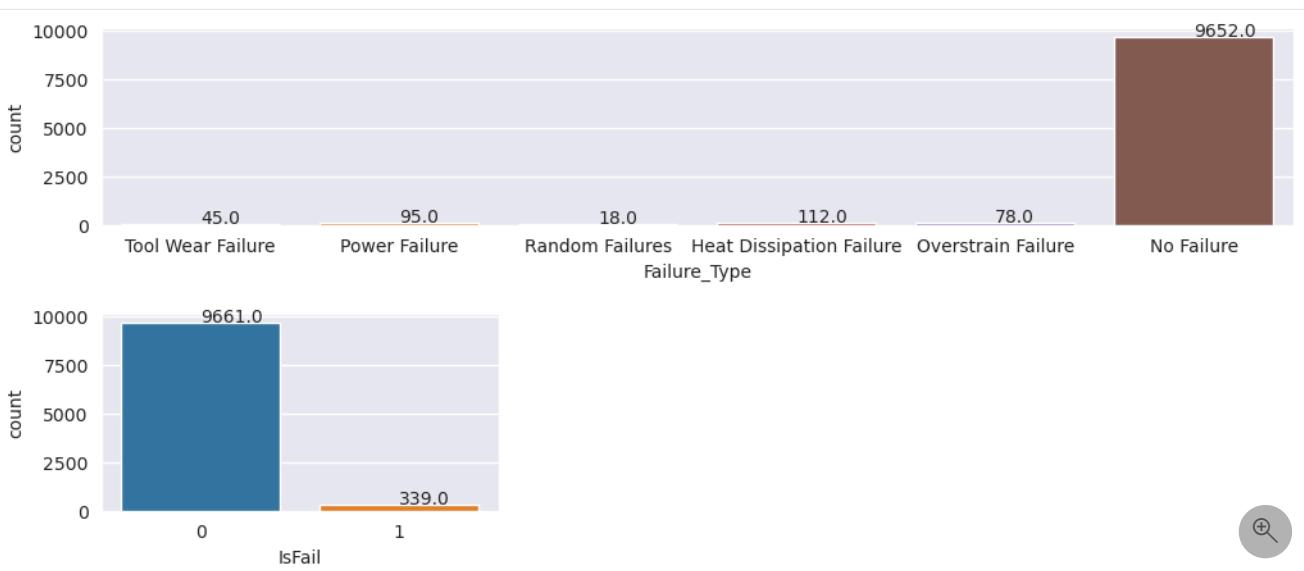
Python

```
# Plot the counts for no failure and each failure type
plt.figure(figsize=(12, 2))
ax = sns.countplot(x='Failure_Type', data=df)
for p in ax.patches:
    ax.annotate(f'{p.get_height()}', (p.get_x()+0.4, p.get_height()+50))

plt.show()

# Plot the counts for no failure versus the sum of all failure types
plt.figure(figsize=(4, 2))
ax = sns.countplot(x='IsFail', data=df)
for p in ax.patches:
    ax.annotate(f'{p.get_height()}', (p.get_x()+0.4, p.get_height()+50))

plt.show()
```



The plots indicate that the no-failure class (shown as `IsFail=0` in the second plot) constitutes most of the samples. Use an oversampling technique to create a more balanced training dataset:

```
Python

# Separate features and target
features = df[['Type', 'Air_temperature_[K]', 'Process_temperature_[K]', 'Rotational_speed_[rpm]', 'Torque_[Nm]', 'Tool_wear_[min]']]
labels = df['IsFail']

# Split the dataset into the training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.2, random_state=42)

# Ignore warnings
import warnings
warnings.filterwarnings('ignore')
# Save test data to the lakehouse for use in future sections
table_name = "predictive_maintenance_test_data"
df_test_X = spark.createDataFrame(X_test)
df_test_X.write.mode("overwrite").format("delta").save(f"Tables/{table_name}")
print(f"Spark DataFrame saved to delta table: {table_name}")
```

Oversample to balance classes in the training dataset

The previous analysis showed that the dataset is highly imbalanced. That imbalance becomes a problem, because the minority class has too few examples for the model to effectively learn the decision boundary.

[SMOTE](#) can solve the problem. SMOTE is a widely used oversampling technique that generates synthetic examples. It generates examples for the minority class based on the Euclidian distances between data points. This method differs from random oversampling, because it creates new examples that don't just duplicate the minority class. The method becomes a more effective technique to handle imbalanced datasets.

```
Python

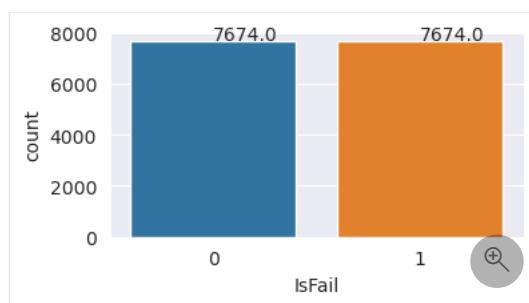
# Disable MLflow autologging because you don't want to track SMOTE fitting
import mlflow

mlflow.autolog(disable=True)

from imblearn.combine import SMOTETomek
smt = SMOTETomek(random_state=SEED)
X_train_res, y_train_res = smt.fit_resample(X_train, y_train)

# Plot the counts for both classes
plt.figure(figsize=(4, 2))
ax = sns.countplot(x='IsFail', data=pd.DataFrame({'IsFail': y_train_res.values}))
for p in ax.patches:
    ax.annotate(f'{p.get_height()}', (p.get_x()+0.4, p.get_height()+50))

plt.show()
```



You successfully balanced the dataset. You can now move to model training.

Step 4: Train and evaluate the models

[MLflow](#) registers models, trains and compares various models, and picks the best model for prediction purposes. You can use the following three models for model training:

- Random forest classifier

- Logistic regression classifier
- XGBoost classifier

Train a random forest classifier

Python

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from mlflow.models.signature import infer_signature
from sklearn.metrics import f1_score, accuracy_score, recall_score

mlflow.set_experiment("Machine_Failure_Classification")
mlflow.autolog(exclusive=False) # This is needed to override the preconfigured autologging behavior

with mlflow.start_run() as run:
    rfc_id = run.info.run_id
    print(f"run_id {rfc_id}, status: {run.info.status}")
    rfc = RandomForestClassifier(max_depth=5, n_estimators=50)
    rfc.fit(X_train_res, y_train_res)
    signature = infer_signature(X_train_res, y_train_res)

    mlflow.sklearn.log_model(
        rfc,
        "machine_failure_model_rf",
        signature=signature,
        registered_model_name="machine_failure_model_rf"
    )

    y_pred_train = rfc.predict(X_train)
    # Calculate the classification metrics for test data
    f1_train = f1_score(y_train, y_pred_train, average='weighted')
    accuracy_train = accuracy_score(y_train, y_pred_train)
    recall_train = recall_score(y_train, y_pred_train, average='weighted')

    # Log the classification metrics to MLflow
    mlflow.log_metric("f1_score_train", f1_train)
    mlflow.log_metric("accuracy_train", accuracy_train)
    mlflow.log_metric("recall_train", recall_train)

    # Print the run ID and the classification metrics
    print("F1 score_train:", f1_train)
    print("Accuracy_train:", accuracy_train)
    print("Recall_train:", recall_train)

    y_pred_test = rfc.predict(X_test)
    # Calculate the classification metrics for test data
    f1_test = f1_score(y_test, y_pred_test, average='weighted')
    accuracy_test = accuracy_score(y_test, y_pred_test)
    recall_test = recall_score(y_test, y_pred_test, average='weighted')

    # Log the classification metrics to MLflow
    mlflow.log_metric("f1_score_test", f1_test)
    mlflow.log_metric("accuracy_test", accuracy_test)
    mlflow.log_metric("recall_test", recall_test)

    # Print the classification metrics
    print("F1 score_test:", f1_test)
    print("Accuracy_test:", accuracy_test)
    print("Recall_test:", recall_test)
```

From the output, both the training and test datasets yield an F1 score, accuracy and recall of about 0.9 when using the random forest classifier.

Train a logistic regression classifier

Python

```
from sklearn.linear_model import LogisticRegression

with mlflow.start_run() as run:
    lr_id = run.info.run_id
    print(f"run_id {lr_id}, status: {run.info.status}")
    lr = LogisticRegression(random_state=42)
    lr.fit(X_train_res, y_train_res)
    signature = infer_signature(X_train_res, y_train_res)
```

```

mlflow.sklearn.log_model(
    lr,
    "machine_failure_model_lr",
    signature=signature,
    registered_model_name="machine_failure_model_lr"
)

y_pred_train = lr.predict(X_train)
# Calculate the classification metrics for training data
f1_train = f1_score(y_train, y_pred_train, average='weighted')
accuracy_train = accuracy_score(y_train, y_pred_train)
recall_train = recall_score(y_train, y_pred_train, average='weighted')

# Log the classification metrics to MLflow
mlflow.log_metric("f1_score_train", f1_train)
mlflow.log_metric("accuracy_train", accuracy_train)
mlflow.log_metric("recall_train", recall_train)

# Print the run ID and the classification metrics
print("F1 score_train:", f1_train)
print("Accuracy_train:", accuracy_train)
print("Recall_train:", recall_train)

y_pred_test = lr.predict(X_test)
# Calculate the classification metrics for test data
f1_test = f1_score(y_test, y_pred_test, average='weighted')
accuracy_test = accuracy_score(y_test, y_pred_test)
recall_test = recall_score(y_test, y_pred_test, average='weighted')

# Log the classification metrics to MLflow
mlflow.log_metric("f1_score_test", f1_test)
mlflow.log_metric("accuracy_test", accuracy_test)
mlflow.log_metric("recall_test", recall_test)

```

Train an XGBoost classifier

Python

```

from xgboost import XGBClassifier

with mlflow.start_run() as run:
    xgb = XGBClassifier()
    xgb_id = run.info.run_id
    print(f"run_id {xgb_id}, status: {run.info.status}")
    xgb.fit(X_train_res.to_numpy(), y_train_res.to_numpy())
    signature = infer_signature(X_train_res, y_train_res)

    mlflow.xgboost.log_model(
        xgb,
        "machine_failure_model_xgb",
        signature=signature,
        registered_model_name="machine_failure_model_xgb"
    )

    y_pred_train = xgb.predict(X_train)
    # Calculate the classification metrics for training data
    f1_train = f1_score(y_train, y_pred_train, average='weighted')
    accuracy_train = accuracy_score(y_train, y_pred_train)
    recall_train = recall_score(y_train, y_pred_train, average='weighted')

    # Log the classification metrics to MLflow
    mlflow.log_metric("f1_score_train", f1_train)
    mlflow.log_metric("accuracy_train", accuracy_train)
    mlflow.log_metric("recall_train", recall_train)

    # Print the run ID and the classification metrics
    print("F1 score_train:", f1_train)
    print("Accuracy_train:", accuracy_train)
    print("Recall_train:", recall_train)

    y_pred_test = xgb.predict(X_test)
    # Calculate the classification metrics for test data
    f1_test = f1_score(y_test, y_pred_test, average='weighted')
    accuracy_test = accuracy_score(y_test, y_pred_test)
    recall_test = recall_score(y_test, y_pred_test, average='weighted')

    # Log the classification metrics to MLflow
    mlflow.log_metric("f1_score_test", f1_test)

```

```
mlflow.log_metric("accuracy_test", accuracy_test)
mlflow.log_metric("recall_test", recall_test)
```

Step 5: Select the best model and predict outputs

In the previous section, you trained three different classifiers: random forest, logistic regression, and XGBoost. You now have the choice to either programmatically access the results, or use the user interface (UI).

For the UI path option, navigate to your workspace and filter the models.

The screenshot shows the MLflow UI for a workspace named 'workspace123'. On the left, there's a sidebar with options like 'New item', 'Import', and a search/filter bar. The main area displays a table of registered models. The columns include Name, Location, Type, Task, Owner, Refreshed, Next refresh, and Endorsement. There are four entries:

Name	Location	Type	Task	Owner	Refreshed	Next refresh	Endorsement
aisample-fraud-lightgbm	workspace123	ML model	—	—	—	—	—
aisample-timeseries-prophet	workspace123	ML model	—	—	—	—	—
aisample-upliftmodelling-controlmodel	workspace123	ML model	—	—	—	—	—
aisample-upliftmodelling-treatmentmodel	workspace123	ML model	—	—	—	—	—

To the right of the table is a sidebar with filters for 'Type' (Experiment, Lakehouse, ML model, Notebook, SQL analytics endpoint, Semantic model, Semantic model (default)), 'Workload' (Owner), and a search bar.

Select individual models for details of the model performance.

This screenshot shows the details for a specific version of an ML model named 'machine_failure_model....'. The top navigation bar includes 'Home', 'View', and other options. The main content area is divided into sections:

- Properties:** Shows version name (Version 2), created time (1/15/2025 1:29 AM), last modified (1/15/2025 1:29 AM), experiment name (Machine_Failure...), run name (good_wing_ppp3...), and a 'Description' field.
- Apply this version:** A button to apply the model version to generate predictions.
- Compare ML model versions in a list:** A link to compare multiple versions visually.
- Version details:** A table of ML model version metrics:

Metric	Value
f1_score_train	0.9980246414136738
accuracy_train	0.998
recall_train	0.998
f1_score_test	0.9694791987673345
accuracy_test	0.9675
recall_test	0.9675

This example shows how to programmatically access the models through MLflow:

The code editor shows a Python script with the following content:

```
Python

runs = {'random forest classifier': rfc_id,
        'logistic regression classifier': lr_id,
        'xgboost classifier': xgb_id}

# Create an empty DataFrame to hold the metrics
df_metrics = pd.DataFrame()

# Loop through the run IDs and retrieve the metrics for each run
for run_name, run_id in runs.items():
    metrics = mlflow.get_run(run_id).data.metrics
    metrics["run_name"] = run_name
    df_metrics = df_metrics.append(metrics, ignore_index=True)

# Print the DataFrame
print(df_metrics)
```

Although XGBoost yields the best results on the training set, it performs poorly on the test data set. That poor performance indicates overfitting. The logistic regression classifier performs poorly on both training and test datasets. Overall, random forest strikes a good balance between training performance and avoidance of overfitting.

In the next section, choose the registered random forest model, and perform a prediction with the [PREDICT](#) feature:

The code editor has a single line of code:

```
Python
print(mlflow.PREDICT)
```

```

from synapse.ml.predict import MLFlowTransformer

model = MLFlowTransformer(
    inputCols=list(X_test.columns),
    outputCol='predictions',
    modelName='machine_failure_model_rf',
    modelVersion=1
)

```

With the `MLFlowTransformer` object you created to load the model for inferencing, use the Transformer API to score the model on the test dataset:

Python

```

predictions = model.transform(spark.createDataFrame(X_test))
predictions.show()

```

This table shows the output:

[Expand table](#)

Type	Air_temperature_[K]	Process_temperature_[K]	Rotational_speed_[rpm]	Torque_[Nm]	Tool_wear_[min]	predictions
0	300.6	309.7	1639.0	30.4	121.0	0
0	303.9	313.0	1551.0	36.8	140.0	0
1	299.1	308.6	1491.0	38.5	166.0	0
0	300.9	312.1	1359.0	51.7	146.0	1
0	303.7	312.6	1621.0	38.8	182.0	0
0	299.0	310.3	1868.0	24.0	221.0	1
2	297.8	307.5	1631.0	31.3	124.0	0
0	297.5	308.2	1327.0	56.5	189.0	1
0	301.3	310.3	1460.0	41.5	197.0	0
2	297.6	309.0	1413.0	40.2	51.0	0
1	300.9	309.4	1724.0	25.6	119.0	0
0	303.3	311.3	1389.0	53.9	39.0	0
0	298.4	307.9	1981.0	23.2	16.0	0
0	299.3	308.8	1636.0	29.9	201.0	0
1	298.1	309.2	1460.0	45.8	80.0	0
0	300.0	309.5	1728.0	26.0	37.0	0
2	299.0	308.7	1940.0	19.9	98.0	0
0	302.2	310.8	1383.0	46.9	45.0	0
0	300.2	309.2	1431.0	51.3	57.0	0
0	299.6	310.2	1468.0	48.0	9.0	0

Save the data into the lakehouse. The data then becomes available for later uses - for example, a Power BI dashboard.

Python

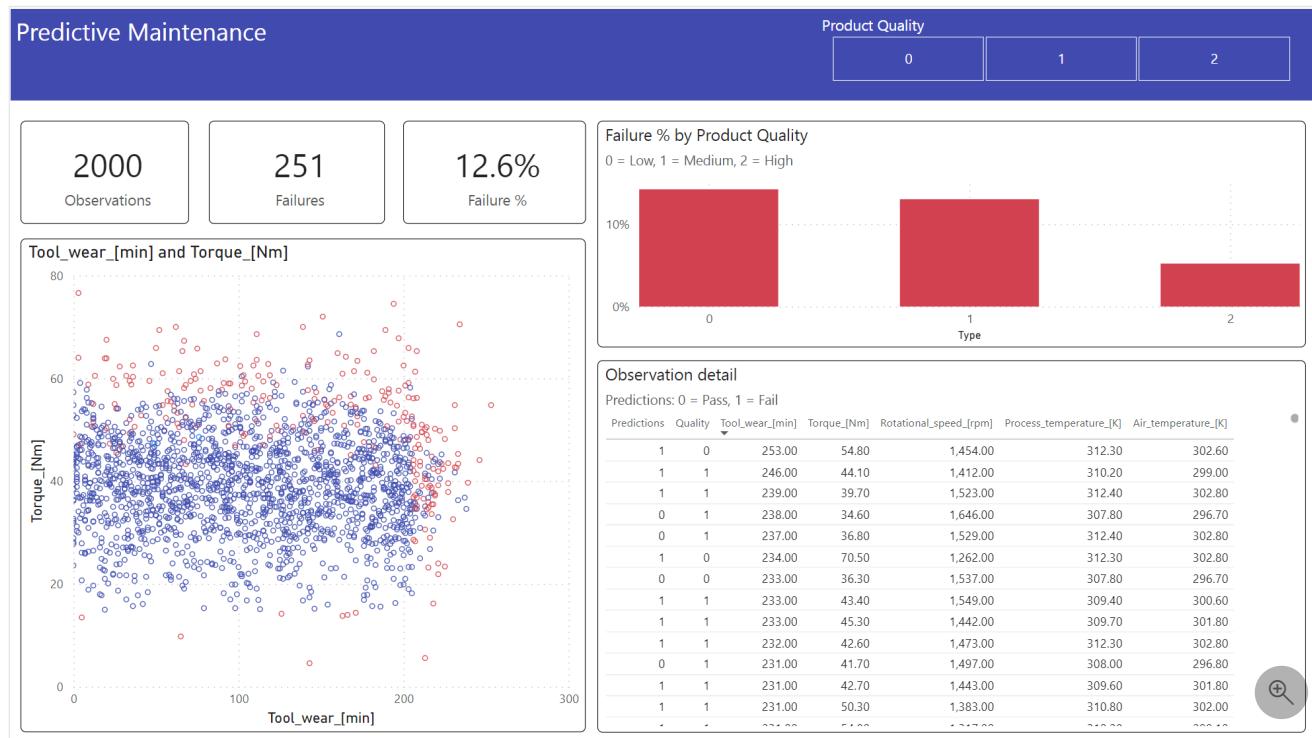
```

# Save test data to the lakehouse for use in the next section.
table_name = "predictive_maintenance_test_with_predictions"
predictions.write.mode("overwrite").format("delta").save(f"Tables/{table_name}")
print(f"Spark DataFrame saved to delta table: {table_name}")

```

Step 6: View business intelligence via visualizations in Power BI

Show the results in an offline format, with a Power BI dashboard.



The dashboard shows that `Tool_wear` and `Torque` create a noticeable boundary between failed and unfailed cases, as expected from the earlier correlation analysis in step 2.

Related content

- [Train and evaluate a text classification model](#)
- [Machine learning model in Microsoft Fabric](#)
- [Train machine learning models](#)
- [Machine learning experiments in Microsoft Fabric](#)

Feedback

Was this page helpful?

Yes

No

Provide product feedback | [Ask the community](#)

Develop, evaluate, and score a forecasting model for superstore sales

Article • 01/22/2024

This tutorial presents an end-to-end example of a Synapse Data Science workflow in Microsoft Fabric. The scenario builds a forecasting model that uses historical sales data to predict product category sales at a superstore.

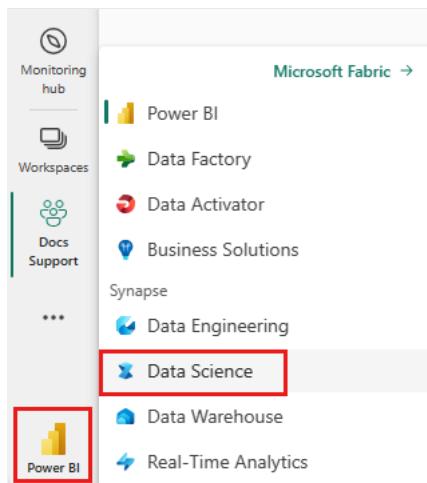
Forecasting is a crucial asset in sales. It combines historical data and predictive methods to provide insights into future trends. Forecasting can analyze past sales to identify patterns, and learn from consumer behavior to optimize inventory, production, and marketing strategies. This proactive approach enhances adaptability, responsiveness, and overall performance of businesses in a dynamic marketplace.

This tutorial covers these steps:

- ✓ Load the data
- ✓ Use exploratory data analysis to understand and process the data
- ✓ Train a machine learning model with an open-source software package, and track experiments with MLflow and the Fabric autologging feature
- ✓ Save the final machine learning model, and make predictions
- ✓ Show the model performance with Power BI visualizations

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



- If necessary, create a Microsoft Fabric lakehouse as described in [Create a lakehouse in Microsoft Fabric](#).

Follow along in a notebook

You can choose one of these options to follow along in a notebook:

- Open and run the built-in notebook in the Synapse Data Science experience
- Upload your notebook from GitHub to the Synapse Data Science experience

Open the built-in notebook

The sample **Sales forecasting** notebook accompanies this tutorial.

To open the tutorial's built-in sample notebook in the Synapse Data Science experience:

1. Go to the Synapse Data Science home page.
2. Select **Use a sample**.
3. Select the corresponding sample:

- From the default **End-to-end workflows (Python)** tab, if the sample is for a Python tutorial.
- From the **End-to-end workflows (R)** tab, if the sample is for an R tutorial.
- From the **Quick tutorials** tab, if the sample is for a quick tutorial.

4. [Attach a lakehouse to the notebook](#) before you start running code.

Import the notebook from GitHub

The [AISample - Superstore Forecast.ipynb](#) notebook accompanies this tutorial.

To open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#), to import the notebook to your workspace.

If you'd rather copy and paste the code from this page, you can [create a new notebook](#).

Be sure to [attach a lakehouse to the notebook](#) before you start running code.

Step 1: Load the data

The dataset contains 9,995 instances of sales of various products. It also includes 21 attributes. This table is from the *Superstore.xlsx* file used in this notebook:

[Expand table](#)

Row ID	Order ID	Order Date	Ship Date	Ship Mode	Customer ID	Customer Name	Segment	Country	City	State	Postal Code	Region	Product ID	Category
4	US-2015-108966	2015-10-11	2015-10-18	Standard Class	SO-20335	Sean O'Donnell	Consumer	United States	Fort Lauderdale	Florida	33311	South	FUR-TA-10000577	Furniture
11	CA-2014-115812	2014-06-09	2014-06-09	Standard Class	Standard Class	Brosina Hoffman	Consumer	United States	Los Angeles	California	90032	West	FUR-TA-10001539	Furniture
31	US-2015-150630	2015-09-17	2015-09-21	Standard Class	TB-21520	Tracy Blumstein	Consumer	United States	Philadelphia	Pennsylvania	19140	East	OFF-EN-10001509	Office Supplies

Define these parameters, so that you can use this notebook with different datasets:

```
Python

IS_CUSTOM_DATA = False # If TRUE, the dataset has to be uploaded manually

IS_SAMPLE = False # If TRUE, use only rows of data for training; otherwise, use all data
SAMPLE_ROWS = 5000 # If IS_SAMPLE is True, use only this number of rows for training

DATA_ROOT = "/lakehouse/default"
DATA_FOLDER = "Files/salesforecast" # Folder with data files
DATA_FILE = "Superstore.xlsx" # Data file name

EXPERIMENT_NAME = "aisample-superstore-forecast" # MLflow experiment name
```

Download the dataset and upload to the lakehouse

This code downloads a publicly available version of the dataset, and then stores it in a Fabric lakehouse:

Important

Be sure to [add a lakehouse](#) to the notebook before you run it. Otherwise, you'll get an error.

Python

```

import os, requests
if not IS_CUSTOM_DATA:
    # Download data files into the lakehouse if they're not already there
    remote_url = "https://synapseaisolutionsa.blob.core.windows.net/public/Forecast_Superstore_Sales"
    file_list = ["Superstore.xlsx"]
    download_path = "/lakehouse/default/Files/salesforecast/raw"

    if not os.path.exists("/lakehouse/default"):
        raise FileNotFoundError(
            "Default lakehouse not found, please add a lakehouse and restart the session."
        )
    os.makedirs(download_path, exist_ok=True)
    for fname in file_list:
        if not os.path.exists(f"{download_path}/{fname}"):
            r = requests.get(f"{remote_url}/{fname}", timeout=30)
            with open(f"{download_path}/{fname}", "wb") as f:
                f.write(r.content)
print("Downloaded demo data files into lakehouse.")

```

Set up MLflow experiment tracking

Microsoft Fabric automatically captures the values of input parameters and output metrics of a machine learning model as you train it. This extends MLflow autologging capabilities. The information is then logged to the workspace, where you can access and visualize it with the MLflow APIs or the corresponding experiment in the workspace. To learn more about autologging, see [Autologging in Microsoft Fabric](#).

To turn off Microsoft Fabric autologging in a notebook session, call `mlflow.autolog()` and set `disable=True`:

Python

```

# Set up MLflow for experiment tracking
import mlflow

mlflow.set_experiment(EXPERIMENT_NAME)
mlflow.autolog(disable=True) # Turn off MLflow autologging

```

Read raw data from the lakehouse

Read raw data from the `Files` section of the lakehouse. Add more columns for different date parts. The same information is used to create a partitioned delta table. Because the raw data is stored as an Excel file, you must use pandas to read it:

Python

```

import pandas as pd
df = pd.read_excel("/lakehouse/default/Files/salesforecast/raw/Superstore.xlsx")

```

Step 2: Perform exploratory data analysis

Import libraries

Before any analysis, import the required libraries:

Python

```

# Importing required libraries
import warnings
import itertools
import numpy as np
import matplotlib.pyplot as plt
warnings.filterwarnings("ignore")
plt.style.use('fivethirtyeight')
import pandas as pd
import statsmodels.api as sm
import matplotlib
matplotlib.rcParams['axes.labelsize'] = 14
matplotlib.rcParams['xtick.labelsize'] = 12
matplotlib.rcParams['ytick.labelsize'] = 12
matplotlib.rcParams['text.color'] = 'k'
from sklearn.metrics import mean_squared_error, mean_absolute_percentage_error

```

Display the raw data

Manually review a subset of the data, to better understand the dataset itself, and use the `display` function to print the DataFrame. Additionally, the `Chart` views can easily visualize subsets of the dataset.

Python

```
display(df)
```

This notebook primarily focuses on forecasting the `Furniture` category sales. This speeds up the computation, and helps show the performance of the model. However, this notebook uses adaptable techniques. You can extend those techniques to predict the sales of other product categories.

Python

```
# Select "Furniture" as the product category
furniture = df.loc[df['Category'] == 'Furniture']
print(furniture['Order Date'].min(), furniture['Order Date'].max())
```

Preprocess the data

Real-world business scenarios often need to predict sales in three distinct categories:

- A specific product category
- A specific customer category
- A specific combination of product category and customer category

First, drop unnecessary columns to preprocess the data. Some of the columns (`Row ID`, `Order ID`, `Customer ID`, and `Customer Name`) are unnecessary because they have no impact. We want to forecast the overall sales, across the state and region, for a specific product category (`Furniture`), so we can drop the `State`, `Region`, `Country`, `City`, and `Postal Code` columns. To forecast sales for a specific location or category, you might need to adjust the preprocessing step accordingly.

Python

```
# Data preprocessing
cols = ['Row ID', 'Order ID', 'Ship Date', 'Ship Mode', 'Customer ID', 'Customer Name',
'Segment', 'Country', 'City', 'State', 'Postal Code', 'Region', 'Product ID', 'Category',
'Sub-Category', 'Product Name', 'Quantity', 'Discount', 'Profit']
# Drop unnecessary columns
furniture.drop(cols, axis=1, inplace=True)
furniture = furniture.sort_values('Order Date')
furniture.isnull().sum()
```

The dataset is structured on a daily basis. We must resample on the column `Order Date`, because we want to develop a model to forecast the sales on a monthly basis.

First, group the `Furniture` category by `Order Date`. Then, calculate the sum of the `Sales` column for each group, to determine the total sales for each unique `Order Date` value. Resample the `Sales` column with the `MS` frequency, to aggregate the data by month. Finally, calculate the mean sales value for each month.

Python

```
# Data preparation
furniture = furniture.groupby('Order Date')['Sales'].sum().reset_index()
furniture = furniture.set_index('Order Date')
furniture.index
y = furniture['Sales'].resample('MS').mean()
y = y.reset_index()
y['Order Date'] = pd.to_datetime(y['Order Date'])
y['Order Date'] = [i+pd.DateOffset(months=67) for i in y['Order Date']]
y = y.set_index(['Order Date'])
maximum_date = y.reset_index()['Order Date'].max()
```

Demonstrate the impact of `Order Date` on `Sales` for the `Furniture` category:

Python

```
# Impact of order date on the sales
y.plot(figsize=(12, 3))
plt.show()
```

Before any statistical analysis, you must import the `statsmodels` Python module. It provides classes and functions for the estimation of many statistical models. It also provides classes and functions to conduct statistical tests and statistical data exploration.

Python

```
import statsmodels.api as sm
```

Perform statistical analysis

A time series tracks these data elements at set intervals, to determine the variation of those elements in the time series pattern:

- **Level:** The fundamental component representing the average value for a specific time period
- **Trend:** Describes whether the time series decreases, stays constant, or increases over time
- **Seasonality:** Describes the periodic signal in the time series, and looks for cyclic occurrences that impact the increasing or decreasing time series patterns
- **Noise/Residual:** Refers to the random fluctuations and variability in the time series data that the model can't explain.

In this code, you observe those elements for your dataset after the preprocessing:

Python

```
# Decompose the time series into its components by using statsmodels
result = sm.tsa.seasonal_decompose(y, model='additive')

# Labels and corresponding data for plotting
components = [('Seasonality', result.seasonal),
               ('Trend', result.trend),
               ('Residual', result.resid),
               ('Observed Data', y)]

# Create subplots in a grid
fig, axes = plt.subplots(nrows=4, ncols=1, figsize=(12, 7))
plt.subplots_adjust(hspace=0.8) # Adjust vertical space
axes = axes.ravel()

# Plot the components
for ax, (label, data) in zip(axes, components):
    ax.plot(data, label=label, color='blue' if label != 'Observed Data' else 'purple')
    ax.set_xlabel('Time')
    ax.set_ylabel(label)
    ax.set_xlabel('Time', fontsize=10)
    ax.set_ylabel(label, fontsize=10)
    ax.legend(fontsize=10)

plt.show()
```

The plots describe the seasonality, trends, and noise in the forecasting data. You can capture the underlying patterns, and develop models that make accurate predictions that are resilient to random fluctuations.

Step 3: Train and track the model

Now that you have the data available, define the forecasting model. In this notebook, apply the forecasting model called *seasonal autoregressive integrated moving average with exogenous factors* (SARIMAX). SARIMAX combines autoregressive (AR) and moving average (MA) components, seasonal differencing, and external predictors to make accurate and flexible forecasts for time series data.

You also use MLflow and Fabric autologging to track the experiments. Here, load the delta table from the lakehouse. You might use other delta tables that consider the lakehouse as the source.

Python

```
# Import required libraries for model evaluation
from sklearn.metrics import mean_squared_error, mean_absolute_percentage_error
```

Tune hyperparameters

SARIMAX takes into account the parameters involved in regular autoregressive integrated moving average (ARIMA) mode (`p`, `d`, `q`), and adds the seasonality parameters (`P`, `D`, `Q`, `s`). These SARIMAX model arguments are called *order* (`p`, `d`, `q`) and *seasonal order* (`P`, `D`, `Q`, `s`), respectively. Therefore, to train the model, we must first tune seven parameters.

The order parameters:

- `p`: The order of the AR component, representing the number of past observations in the time series used to predict the current value.

Typically, this parameter should be a non-negative integer. Common values are in the range of `0` to `3`, although higher values are possible, depending on the specific data characteristics. A higher `p` value indicates a longer memory of past values in the model.

- `d`: The differencing order, representing the number of times that the time series needs to be differenced, to achieve stationarity.

This parameter should be a non-negative integer. Common values are in the range of `0` to `2`. A `d` value of `0` means the time series is already stationary. Higher values indicate the number of differencing operations required to make it stationary.

- `q`: The order of the MA component, representing the number of past white-noise error terms used to predict the current value.

This parameter should be a non-negative integer. Common values are in the range of `0` to `3`, but higher values might be necessary for certain time series. A higher `q` value indicates a stronger reliance on past error terms to make predictions.

The seasonal order parameters:

- `P`: The seasonal order of the AR component, similar to `p` but for the seasonal part
- `D`: The seasonal order of differencing, similar to `d` but for the seasonal part
- `Q`: The seasonal order of the MA component, similar to `q` but for the seasonal part
- `s`: The number of time steps per seasonal cycle (for example, 12 for monthly data with a yearly seasonality)

Python

```
# Hyperparameter tuning
p = d = q = range(0, 2)
pdq = list(itertools.product(p, d, q))
seasonal_pdq = [(x[0], x[1], x[2], 12) for x in list(itertools.product(p, d, q))]
print('Examples of parameter combinations for Seasonal ARIMA...')
print('SARIMAX: {} x {}'.format(pdq[1], seasonal_pdq[1]))
print('SARIMAX: {} x {}'.format(pdq[1], seasonal_pdq[2]))
print('SARIMAX: {} x {}'.format(pdq[2], seasonal_pdq[3]))
print('SARIMAX: {} x {}'.format(pdq[2], seasonal_pdq[4]))
```

SARIMAX has other parameters:

- `enforce_stationarity`: Whether or not the model should enforce stationarity on the time series data, before fitting the SARIMAX model.

If `enforce_stationarity` is set to `True` (the default), it indicates that the SARIMAX model should enforce stationarity on the time series data. The SARIMAX model then automatically applies differencing to the data, to make it stationary, as specified by the `d` and `D` orders, before fitting the model. This is a common practice because many time series models, including SARIMAX, assume that the data is stationary.

For a nonstationary time series (for example, it exhibits trends or seasonality), it's good practice to set `enforce_stationarity` to `True`, and let the SARIMAX model handle the differencing to achieve stationarity. For a stationary time series (for example, one with no trends or seasonality), set `enforce_stationarity` to `False` to avoid unnecessary differencing.

- `enforce_invertibility`: Controls whether or not the model should enforce invertibility on the estimated parameters during the optimization process.

If `enforce_invertibility` is set to `True` (the default), it indicates that the SARIMAX model should enforce invertibility on the estimated parameters. Invertibility ensures that the model is well defined, and that the estimated AR and MA coefficients land within the range of stationarity.

Invertibility enforcement helps ensure that the SARIMAX model adheres to the theoretical requirements for a stable time series model. It also helps prevent issues with model estimation and stability.

The default is an `AR(1)` model. This refers to `(1, 0, 0)`. However, it's common practice to try different combinations of the order parameters and seasonal order parameters, and evaluate the model performance for a dataset. The appropriate values can vary from one

time series to another.

Determination of the optimal values often involves analysis of the autocorrelation function (ACF) and partial autocorrelation function (PACF) of the time series data. It also often involves use of model selection criteria - for example, the Akaike information criterion (AIC) or the Bayesian information criterion (BIC).

Tune the hyperparameters:

Python

```
# Tune the hyperparameters to determine the best model
for param in pdq:
    for param_seasonal in seasonal_pdq:
        try:
            mod = sm.tsa.statespace.SARIMAX(y,
                                              order=param,
                                              seasonal_order=param_seasonal,
                                              enforce_stationarity=False,
                                              enforce_invertibility=False)
            results = mod.fit(disp=False)
            print('ARIMA{}x{}12 - AIC:{}'.format(param, param_seasonal, results.aic))
        except:
            continue
```

After evaluation of the preceding results, you can determine the values for both the order parameters and the seasonal order parameters. The choice is `order=(0, 1, 1)` and `seasonal_order=(0, 1, 1, 12)`, which offer the lowest AIC (for example, 279.58). Use these values to train the model.

Train the model

Python

```
# Model training
mod = sm.tsa.statespace.SARIMAX(y,
                                 order=(0, 1, 1),
                                 seasonal_order=(0, 1, 1, 12),
                                 enforce_stationarity=False,
                                 enforce_invertibility=False)
results = mod.fit(disp=False)
print(results.summary().tables[1])
```

This code visualizes a time series forecast for furniture sales data. The plotted results show both the observed data and the one-step-ahead forecast, with a shaded region for confidence interval.

Python

```
# Plot the forecasting results
pred = results.get_prediction(start=maximim_date, end=maximim_date+pd.DateOffset(months=6), dynamic=False) # Forecast for the next 6 months (months=6)
pred_ci = pred.conf_int() # Extract the confidence intervals for the predictions
ax = y['2019'].plot(label='observed')
pred.predicted_mean.plot(ax=ax, label='One-step ahead forecast', alpha=.7, figsize=(12, 7))
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.2)
ax.set_xlabel('Date')
ax.set_ylabel('Furniture Sales')
plt.legend()
plt.show()
```

Python

```
# Validate the forecasted result
predictions = results.get_prediction(start=maximim_date-pd.DateOffset(months=6-1), dynamic=False)
# Forecast on the unseen future data
predictions_future = results.get_prediction(start=maximim_date+ pd.DateOffset(months=1),end=maximim_date+
pd.DateOffset(months=6),dynamic=False)
```

Use `predictions` to assess the model's performance, by contrasting it with the actual values. The `predictions_future` value indicates future forecasting.

Python

```
# Log the model and parameters
model_name = f"{EXPERIMENT_NAME}-Sarimax"
with mlflow.start_run(run_name="Sarimax") as run:
    mlflow.statsmodels.log_model(results, model_name, registered_model_name=model_name)
    mlflow.log_params({"order":(0,1,1), "seasonal_order":(0, 1, 1, 12), "enforce_stationarity":False, "enforce_invertibility":False})
    model_uri = f"runs:{run.info.run_id}/{model_name}"
    print("Model saved in run %s" % run.info.run_id)
    print(f"Model URI: {model_uri}")
mlflow.end_run()
```

Python

```
# Load the saved model
loaded_model = mlflow.statsmodels.load_model(model_uri)
```

Step 4: Score the model and save predictions

Integrate the actual values with the forecasted values, to create a Power BI report. Store these results in a table within the lakehouse.

Python

```
# Data preparation for Power BI visualization
Future = pd.DataFrame(predictions_future.predicted_mean).reset_index()
Future.columns = ['Date', 'Forecasted_Sales']
Future['Actual_Sales'] = np.NAN
Actual = pd.DataFrame(predictions.predicted_mean).reset_index()
Actual.columns = ['Date', 'Forecasted_Sales']
y_truth = y['2023-02-01':]
Actual['Actual_Sales'] = y_truth.values
final_data = pd.concat([Actual, Future])
# Calculate the mean absolute percentage error (MAPE) between 'Actual_Sales' and 'Forecasted_Sales'
final_data['MAPE'] = mean_absolute_percentage_error(Actual['Actual_Sales'], Actual['Forecasted_Sales']) * 100
final_data['Category'] = "Furniture"
final_data[final_data['Actual_Sales'].isnull()]
```

Python

```
input_df = y.reset_index()
input_df.rename(columns = {'Order Date':'Date', 'Sales':'Actual_Sales'}, inplace=True)
input_df['Category'] = 'Furniture'
input_df['MAPE'] = np.NAN
input_df['Forecasted_Sales'] = np.NAN
```

Python

```
# Write back the results into the lakehouse
final_data_2 = pd.concat([input_df, final_data[final_data['Actual_Sales'].isnull()]])
table_name = "Demand_Forecast_New_1"
spark.createDataFrame(final_data_2).write.mode("overwrite").format("delta").save(f"Tables/{table_name}")
print(f"Spark DataFrame saved to delta table: {table_name}")
```

Step 5: Visualize in Power BI

The Power BI report shows a mean absolute percentage error (MAPE) of 16.58. The MAPE metric defines the accuracy of a forecasting method. It represents the accuracy of the forecasted quantities, in comparison with the actual quantities.

MAPE is a straightforward metric. A 10% MAPE represents that the average deviation between the forecasted values and actual values is 10%, regardless of whether the deviation was positive or negative. Standards of desirable MAPE values vary across industries.

The light blue line in this graph represents the actual sales values. The dark blue line represents the forecasted sales values. Comparison of actual and forecasted sales reveals that the model effectively predicts sales for the Furniture category during the first six months of 2023.



Based on this observation, we can have confidence in the forecasting capabilities of the model, for the overall sales in the last six months of 2023, and extending into 2024. This confidence can inform strategic decisions about inventory management, procurement of raw materials, and other business-related considerations.

Related content

- [How to use Microsoft Fabric notebooks](#)
- [Machine learning model in Microsoft Fabric](#)
- [Train machine learning models](#)
- [Machine learning experiments in Microsoft Fabric](#)

Feedback

Was this page helpful?

Provide product feedback [↗](#) | [Ask the community ↗](#)

Tutorial: Create, evaluate, and score a text classification model

Article • 04/20/2025

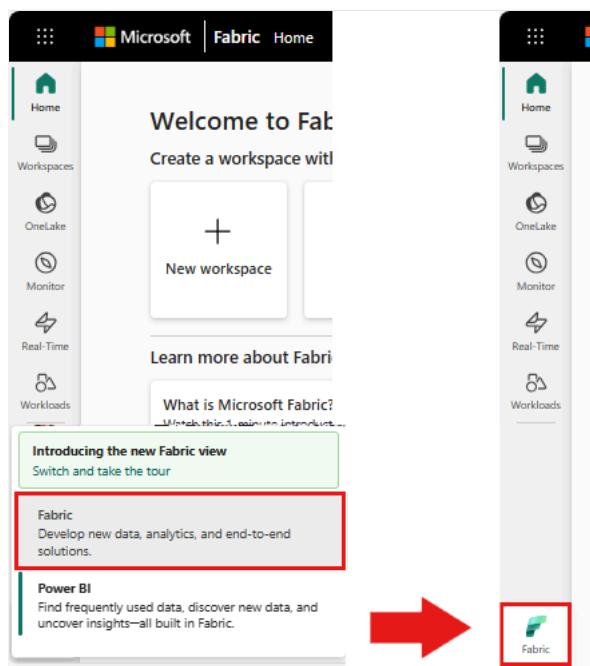
This tutorial presents an end-to-end example of a Synapse Data Science workflow for a text classification model, in Microsoft Fabric. The scenario uses both Word2vec natural language processing (NLP), and logistic regression, on Spark, to determine the genre of a book from the British Library book dataset. The determination is solely based on the title of the book.

This tutorial covers these steps:

- ✓ Install custom libraries
- ✓ Load the data
- ✓ Understand and process the data with exploratory data analysis
- ✓ Train a machine learning model with both Word2vec NLP and logistic regression, and track experiments with MLflow and the Fabric autologging feature
- ✓ Load the machine learning model for scoring and predictions

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- If you don't have a Microsoft Fabric lakehouse, follow the steps in the [Create a lakehouse in Microsoft Fabric](#) resource to create one.

Follow along in a notebook

To follow along in a notebook, you have these options:

- Open and run the built-in notebook.
- Upload your notebook from GitHub.

Open the built-in notebook

The sample Title genre classification notebook accompanies this tutorial.

1. To open the sample notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#).
2. Make sure to [attach a lakehouse to the notebook](#) before you start running code.

Import the notebook from GitHub

[Alsample - Title Genre Classification.ipynb](#) is the notebook that accompanies this tutorial.

- To open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#) to import the notebook to your workspace.
- If you'd rather copy and paste the code from this page, you can [create a new notebook](#).
- Be sure to [attach a lakehouse to the notebook](#) before you start running code.

Step 1: Install custom libraries

For machine learning model development or ad-hoc data analysis, you might need to quickly install a custom library for your Apache Spark session. You have two options to install a library.

- To install a library, in your current notebook only, use the inline installation capabilities (`%pip` or `%conda`) of your notebook.
- As an alternative, you can create a Fabric environment, and install libraries from public sources or upload custom libraries to it. Then, your workspace admin can attach the environment as the default for the workspace. At that point, all the libraries in the environment become available for use in all notebooks and all Spark job definitions in that workspace. For more information about environments, visit the [create, configure, and use an environment in Microsoft Fabric](#) resource.

For the classification model, use the `wordcloud` library to represent the word frequency in text. In `wordcloud` resources, the size of a word represents its frequency. For this tutorial, use `%pip install` to install `wordcloud` in your notebook.

① Note

The PySpark kernel restarts after `%pip install` runs. Install the libraries you need before you run any other cells.

Python

```
# Install wordcloud for text visualization by using pip
%pip install wordcloud
```

Step 2: Load the data

The British Library book dataset has metadata about books from the British Library. A collaboration between the library and Microsoft digitized the original resources that became the dataset. The metadata is classification information that indicates whether or not a book is fiction or nonfiction. The following diagram shows a row sample of the dataset.

[Expand table](#)

BL record ID	Type of resource	Name	Dates associated with name	Type of name	Role	All names	Title	Variant titles	Series title	Number within series	Country of publication	Place of publication	Publisher	Date pub
014602826	Monograph	Yearsley, Ann	1753-1806	person		More, Hannah, 1745-1833 [person]; Yearsley, Ann, 1753-1806 [person]	Poems on several occasions [With a prefatory letter by Hannah More.]				England	London		1786
014602830	Monograph	A, T.		person		Oldham, John, 1653-1683 [person]; A, T. [person]	A Satyr against Vertue. (A poem: supposed to be spoken by a				England	London		1679

BL record ID	Type of resource	Name	Dates associated with name	Type of name	Role	All names	Title	Variant titles	Series title	Number within series	Country of publication	Place of publication	Publisher	Date pub
							Town-Hector [By John Oldham. The preface signed: T. A.])							

With this dataset, our goal is to train a classification model that determines the genre of a book, based only on the book title.

Define the following parameters, to apply this notebook on different datasets:

Python

```
IS_CUSTOM_DATA = False # If True, the user must manually upload the dataset
DATA_FOLDER = "Files/title-genre-classification"
DATA_FILE = "blbooksgenre.csv"

# Data schema
TEXT_COL = "Title"
LABEL_COL = "annotator_genre"
LABELS = ["Fiction", "Non-fiction"]

EXPERIMENT_NAME = "sample-aisample-textclassification" # MLflow experiment name
```

Download the dataset and upload to the lakehouse

The following code snippet downloads a publicly available version of the dataset, and then stores it in a Fabric lakehouse:

ⓘ Important

[Add a lakehouse](#) to the notebook before you run it. Failure to do so resultS in an error.

Python

```
if not IS_CUSTOM_DATA:
    # Download demo data files into the lakehouse, if they don't exist
    import os, requests

    remote_url = "https://synapseaisolutionsa.blob.core.windows.net/public/Title_Genre_Classification"
    fname = "blbooksgenre.csv"
    download_path = f"/lakehouse/default/{DATA_FOLDER}/raw"

    if not os.path.exists("/lakehouse/default"):
        # Add a lakehouse, if no default lakehouse was added to the notebook
        # A new notebook won't link to any lakehouse by default
        raise FileNotFoundError(
            "Default lakehouse not found, please add a lakehouse and restart the session."
        )
    os.makedirs(download_path, exist_ok=True)
    if not os.path.exists(f"{download_path}/{fname}"):
        r = requests.get(f"{remote_url}/{fname}", timeout=30)
        with open(f"{download_path}/{fname}", "wb") as f:
            f.write(r.content)
    print("Downloaded demo data files into lakehouse.")
```

Import required libraries

Before any processing, you must import the required libraries, including the libraries for [Spark](#) and [SynapseML](#):

Python

```
import numpy as np
from itertools import chain
```

```
from wordcloud import WordCloud
import matplotlib.pyplot as plt
import seaborn as sns

import pyspark.sql.functions as F

from pyspark.ml import Pipeline
from pyspark.ml.feature import *
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import (
    BinaryClassificationEvaluator,
    MulticlassClassificationEvaluator,
)

from synapse.ml.stages import ClassBalancer
from synapse.ml.train import ComputeModelStatistics

import mlflow
```

Define hyperparameters

The following code snippet defines the necessary hyperparameters for model training:

ⓘ Important

Modify these hyperparameters only if you understand each parameter.

Python

```
# Hyperparameters
word2vec_size = 128 # The length of the vector for each word
min_word_count = 3 # The minimum number of times that a word must appear to be considered
max_iter = 10 # The maximum number of training iterations
k_folds = 3 # The number of folds for cross-validation
```

Start recording the time needed to run this notebook:

Python

```
# Record the notebook running time
import time

ts = time.time()
```

Set up MLflow experiment tracking

Autologging extends the MLflow logging capabilities. Autologging automatically captures the input parameter values and output metrics of a machine learning model as you train it. You then log this information to the workspace. In the workspace, you can access and visualize the information with the MLflow APIs, or the corresponding experiment, in the workspace. For more information about autologging, visit the [Autologging in Microsoft Fabric](#) resource.

To disable Microsoft Fabric autologging in a notebook session, call `mlflow.autolog()` and set `disable=True`:

Python

```
# Set up Mlflow for experiment tracking

mlflow.set_experiment(EXPERIMENT_NAME)
mlflow.autolog(disable=True) # Disable Mlflow autologging
```

Read raw date data from the lakehouse

Python

```
raw_df = spark.read.csv(f"{DATA_FOLDER}/raw/{DATA_FILE}", header=True, inferSchema=True)
```

Step 3: Perform exploratory data analysis

Explore the dataset with the `display` command, to view high-level statistics for the dataset and to show the chart views:

```
Python
```

```
display(raw_df.limit(20))
```

Prepare the data

To clean the data, remove the duplicates:

```
Python
```

```
df = (
    raw_df.select([TEXT_COL, LABEL_COL])
    .where(F.col(LABEL_COL).isin(LABELS))
    .dropDuplicates([TEXT_COL])
    .cache()
)

display(df.limit(20))
```

Apply class balancing to address any bias:

```
Python
```

```
# Create a ClassBalancer instance, and set the input column to LABEL_COL
cb = ClassBalancer().setInputCol(LABEL_COL)

# Fit the ClassBalancer instance to the input DataFrame, and transform the DataFrame
df = cb.fit(df).transform(df)

# Display the first 20 rows of the transformed DataFrame
display(df.limit(20))
```

To tokenize the dataset, split the paragraphs and sentences into smaller units. This way, it becomes easier to assign meaning. Next, remove the stopwords to improve the performance. Stopword removal involves removal of words that commonly occur across all documents in the corpus. Stopword removal is one of the most commonly used preprocessing steps in natural language processing (NLP) applications. The following code snippet covers these steps:

```
Python
```

```
# Text transformer
tokenizer = Tokenizer(inputCol=TEXT_COL, outputCol="tokens")
stopwords_remover = StopWordsRemover(inputCol="tokens", outputCol="filtered_tokens")

# Build the pipeline
pipeline = Pipeline(stages=[tokenizer, stopwords_remover])

token_df = pipeline.fit(df).transform(df)

display(token_df.limit(20))
```

Display the wordcloud library for each class. A wordcloud library presents keywords that appear frequently in text data, is a visually prominent presentation. The wordcloud library is effective because the keyword rendering forms a cloudlike color picture, to better capture the main text data at a glance. Visit [this resource](#) for more information about wordcloud.

The following code snippet covers these steps:

```
Python
```

```
# WordCloud
for label in LABELS:
    tokens = (
        token_df.where(F.col(LABEL_COL) == label)
        .select(F.explode("filtered_tokens").alias("token"))
        .where(F.col("token").rlike(r"^\w+$"))
    )

    top50_tokens = (
```

```

        tokens.groupBy("token").count().orderBy(F.desc("count")).limit(50).collect()
    )

    # Generate a wordcloud image
    wordcloud = WordCloud(
        scale=10,
        background_color="white",
        random_state=42, # Make sure the output is always the same for the same input
    ).generate_from_frequencies(dict(top50_tokens))

    # Display the generated image by using matplotlib
    plt.figure(figsize=(10, 10))
    plt.title(label, fontsize=20)
    plt.axis("off")
    plt.imshow(wordcloud, interpolation="bilinear")

```

Finally, use Word2vec NLP to vectorize the text. The Word2vec NLP technique creates a vector representation of each word in the text. Words used in similar contexts, or that have semantic relationships, are captured effectively through their closeness in the vector space. This closeness indicates that similar words have similar word vectors. The following code snippet covers these steps:

Python

```

# Label transformer
label_indexer = StringIndexer(inputCol=LABEL_COL, outputCol="labelIdx")
vectorizer = Word2Vec(
    vectorSize=word2vec_size,
    minCount=min_word_count,
    inputCol="filtered_tokens",
    outputCol="features",
)

# Build the pipeline
pipeline = Pipeline(stages=[label_indexer, vectorizer])
vec_df = (
    pipeline.fit(token_df)
    .transform(token_df)
    .select([TEXT_COL, LABEL_COL, "features", "labelIdx", "weight"])
)
display(vec_df.limit(20))

```

Step 4: Train and evaluate the model

With the data in place, define the model. In this section, you train a logistic regression model to classify the vectorized text.

Prepare training and test datasets

The following code snippet splits the dataset:

Python

```

# Split the dataset into training and testing
(train_df, test_df) = vec_df.randomSplit((0.8, 0.2), seed=42)

```

Track machine learning experiments

Machine learning experiment tracking manages all the experiments and their components - for example, parameters, metrics, models, and other artifacts. Tracking enables the organization and management of all the components that a specific machine learning experiment requires. It also enables the easy reproduction of past results with saved experiments. Visit [Machine learning experiments in Microsoft Fabric](#) for more information.

A machine learning experiment is the primary unit of organization and control for all related machine learning runs. A run corresponds to a single execution of model code. The following code snippet covers these steps:

Python

```

# Build the logistic regression classifier
lr = (
    LogisticRegression()
    .setMaxIter(max_iter)
    .setFeaturesCol("features")

```

```
.setLabelCol("labelIdx")
.setWeightCol("weight")
)
```

Tune hyperparameters

Build a grid of parameters to search over the hyperparameters. Then build a cross-evaluator estimator, to produce a `CrossValidator` model, as shown in the following code snippet:

Python

```
# Build a grid search to select the best values for the training parameters
param_grid = (
    ParamGridBuilder()
    .addGrid(lr.regParam, [0.03, 0.1])
    .addGrid(lr.elasticNetParam, [0.0, 0.1])
    .build()
)

if len(LABELS) > 2:
    evaluator_cls = MulticlassClassificationEvaluator
    evaluator_metrics = ["f1", "accuracy"]
else:
    evaluator_cls = BinaryClassificationEvaluator
    evaluator_metrics = ["areaUnderROC", "areaUnderPR"]
evaluator = evaluator_cls(labelCol="labelIdx", weightCol="weight")

# Build a cross-evaluator estimator
crossval = CrossValidator(
    estimator=lr,
    estimatorParamMaps=param_grid,
    evaluator=evaluator,
    numFolds=k_folds,
    collectSubModels=True,
)
```

Evaluate the model

We can evaluate the models on the test dataset, to compare them. A well-trained model should demonstrate high performance, on the relevant metrics, when run against the validation and test datasets. The following code snippet covers these steps:

Python

```
def evaluate(model, df):
    log_metric = {}
    prediction = model.transform(df)
    for metric in evaluator_metrics:
        value = evaluator.evaluate(prediction, {evaluator.metricName: metric})
        log_metric[metric] = value
        print(f"{metric}: {value:.4f}")
    return prediction, log_metric
```

Track experiments by using MLflow

Start the training and evaluation process. Use MLflow to track all experiments, and log the parameters, metrics, and models. In the workspace, all of this information is logged under the experiment name. The following code snippet covers these steps:

Python

```
with mlflow.start_run(run_name="lr"):
    models = crossval.fit(train_df)
    best_metrics = {k: 0 for k in evaluator_metrics}
    best_index = 0
    for idx, model in enumerate(models.subModels[0]):
        with mlflow.start_run(nested=True, run_name=f"lr_{idx}") as run:
            print("\nEvaluating on test data:")
            print(f"subModel No. {idx + 1}")
            prediction, log_metric = evaluate(model, test_df)

            if log_metric[evaluator_metrics[0]] > best_metrics[evaluator_metrics[0]]:
                best_metrics = log_metric
                best_index = idx
```

```

        print("log model")
        mlflow.spark.log_model(
            model,
            f"{EXPERIMENT_NAME}-lrmrmodel",
            registered_model_name=f"{EXPERIMENT_NAME}-lrmrmodel",
            dfs_tmpdir="Files/spark",
        )

        print("log metrics")
        mlflow.log_metrics(log_metric)

        print("log parameters")
        mlflow.log_params(
            {
                "word2vec_size": word2vec_size,
                "min_word_count": min_word_count,
                "max_iter": max_iter,
                "k_folds": k_folds,
                "DATA_FILE": DATA_FILE,
            }
        )

# Log the best model and its relevant metrics and parameters to the parent run
mlflow.spark.log_model(
    models.subModels[0][best_index],
    f"{EXPERIMENT_NAME}-lrmrmodel",
    registered_model_name=f"{EXPERIMENT_NAME}-lrmrmodel",
    dfs_tmpdir="Files/spark",
)
mlflow.log_metrics(best_metrics)
mlflow.log_params(
    {
        "word2vec_size": word2vec_size,
        "min_word_count": min_word_count,
        "max_iter": max_iter,
        "k_folds": k_folds,
        "DATA_FILE": DATA_FILE,
    }
)

```

To view your experiments:

1. Select your workspace in the left nav
2. Find and select the experiment name - in this case, **sample_aisample-textclassification**

Properties

- Run name: Ir_3
- Start date: 1/14/2025 6:19 AM
- Duration: 21s
- Status: Completed

Run details

Run metrics (2)	Value
areaUnderPR	0.6788731421242457
areaUnderROC	0.753856165043864

Run parameters (5)	Value
word2vec_size	128
min_word_count	3
max_iter	10
k_folds	3
DATA_FILE	blbooksgenre.csv

Run tags (0)

No run tags

Input schema (0)

Step 5: Score and save prediction results

Microsoft Fabric allows users to operationalize machine learning models with the scalable `PREDICT` function. This function supports batch scoring (or batch inferencing) in any compute engine. You can create batch predictions straight from a notebook or from the item page for a particular model. For more information about the `PREDICT` function, and how to use it in Fabric, visit [Machine learning model scoring with PREDICT in Microsoft Fabric](#).

From our evaluation results, model 1 has the largest metrics for both Area Under the Precision-Recall Curve (AUPRC) and for Area Under the Curve Receiver Operating Characteristic (AUC-ROC). Therefore, you should use model 1 for prediction.

The AUC-ROC measure is widely used to measure binary classifiers performance. However, it sometimes becomes more appropriate to evaluate the classifier based on AUPRC measurements. The AUC-ROC chart visualizes the trade-off between true positive rate (TPR) and false positive rate (FPR). The AUPRC curve combines both precision (positive predictive value or PPV) and recall (true positive rate or TPR) in a single visualization. The following code snippets cover these steps:

Python

```
# Load the best model
model_uri = f"models:/{{EXPERIMENT_NAME}}-lrmodel/1"
loaded_model = mlflow.spark.load_model(model_uri, dfs_tmpdir="Files/spark")

# Verify the loaded model
batch_predictions = loaded_model.transform(test_df)
batch_predictions.show(5)
```

Python

```
# Code to save userRecs in the lakehouse
batch_predictions.write.format("delta").mode("overwrite").save(
    f"{{DATA_FOLDER}}/predictions/batch_predictions"
)
```

Python

```
# Determine the entire runtime
print(f"Full run cost {int(time.time() - ts)} seconds.")
```

Related content

- [Machine learning model in Microsoft Fabric](#)
- [Train machine learning models](#)
- [Machine learning experiments in Microsoft Fabric](#)

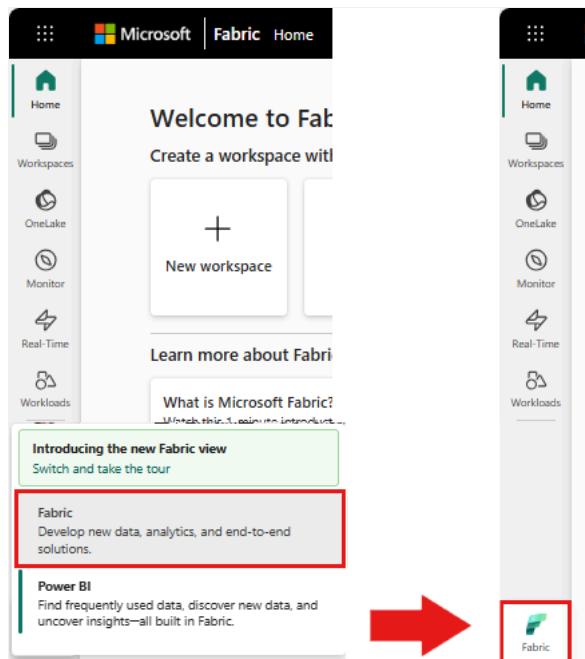
Train and evaluate a time series forecasting model

Article • 01/17/2025

In this notebook, we build a program to forecast time series data that has seasonal cycles. We use the [NYC Property Sales dataset](#) with dates ranging from 2003 to 2015 published by NYC Department of Finance on the [NYC Open Data Portal](#).

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- Familiarity with [Microsoft Fabric notebooks](#).
- A lakehouse to store data for this example. For more information, see [Add a lakehouse to your notebook](#).

Follow along in a notebook

You can follow along in a notebook one of two ways:

- Open and run the built-in notebook.
- Upload your notebook from GitHub.

Open the built-in notebook

The sample Time series notebook accompanies this tutorial.

1. To open the sample notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#).
2. Make sure to [attach a lakehouse to the notebook](#) before you start running code.

Import the notebook from GitHub

[Alsample - Time Series Forecasting.ipynb](#) is the notebook that accompanies this tutorial.

- To open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#) to import the notebook to your workspace.
- If you'd rather copy and paste the code from this page, you can [create a new notebook](#).

- Be sure to [attach a lakehouse to the notebook](#) before you start running code.

Step 1: Install custom libraries

When you develop a machine learning model, or you handle ad-hoc data analysis, you may need to quickly install a custom library (for example, `prophet` in this notebook) for the Apache Spark session. To do this, you have two choices.

1. You can use the in-line installation capabilities (for example, `%pip`, `%conda`, etc.) to quickly get started with new libraries. This would only install the custom libraries in the current notebook, not in the workspace.

Python

```
# Use pip to install libraries
%pip install <library name>

# Use conda to install libraries
%conda install <library name>
```

2. Alternatively, you can create a Fabric environment, install libraries from public sources or upload custom libraries to it, and then your workspace admin can attach the environment as the default for the workspace. All the libraries in the environment will then become available for use in any notebooks and Spark job definitions in the workspace. For more information on environments, see [create, configure, and use an environment in Microsoft Fabric](#).

For this notebook, you use `%pip install` to install the `prophet` library. The PySpark kernel will restart after `%pip install`. This means that you must install the library before you run any other cells.

Python

```
# Use pip to install Prophet
%pip install prophet
```

Step 2: Load the data

Dataset

This notebook uses the NYC Property Sales data dataset. It covers data from 2003 to 2015, published by the NYC Department of Finance on the [NYC Open Data Portal](#).

The dataset includes a record of every building sale in the New York City property market, within a 13 year period. Refer to the [Glossary of Terms for Property Sales Files](#) for a definition of the columns in the dataset.

[] Expand table

borough	neighborhood	building_class_category	tax_class	block	lot	eastment	building_class_at_present	address	apartment_number	zip_
Manhattan	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	0.0	384.0	17.0		C4	225 EAST 2ND STREET		1000000000000000000
Manhattan	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2.0	405.0	12.0		C7	508 EAST 12TH STREET		1000000000000000000

The goal is to build a model that forecasts the monthly total sales, based on historical data. For this, you use [Prophet](#), an open source forecasting library developed by Facebook. Prophet is based on an additive model, where nonlinear trends are fit with daily, weekly, and yearly seasonality, and holiday effects. Prophet works best on time series datasets that have strong seasonal effects, and several seasons of historical data. Additionally, Prophet robustly handles missing data, and data outliers.

Prophet uses a decomposable time series model, consisting of three components:

- **trend:** Prophet assumes a piece-wise constant rate of growth, with automatic change point selection
- **seasonality:** By default, Prophet uses Fourier Series to fit weekly and yearly seasonality

- **holidays**: Prophet requires all past and future occurrences of holidays. If a holiday doesn't repeat in the future, Prophet won't include it in the forecast.

This notebook aggregates the data on a monthly basis, so it ignores the holidays.

Read [the official paper](#) for more information about the Prophet modeling techniques.

Download the dataset, and upload to a lakehouse

The data source consists of 15 `.csv` files. These files contain property sales records from five boroughs in New York, between 2003 and 2015. For convenience, the `nyc_property_sales.tar` file holds all of these `.csv` files, compressing them into one file. A publicly available blob storage hosts this `.tar` file.

Tip

With the parameters shown in this code cell, you can easily apply this notebook to different datasets.

Python

```
URL = "https://synapseaisolutionsa.blob.core.windows.net/public/NYC_Property_Sales_Dataset/"
TAR_FILE_NAME = "nyc_property_sales.tar"
DATA_FOLDER = "Files/NYC_Property_Sales_Dataset"
TAR_FILE_PATH = f"/lakehouse/default/{DATA_FOLDER}/tar/"
CSV_FILE_PATH = f"/lakehouse/default/{DATA_FOLDER}/csv/"

EXPERIMENT_NAME = "aisample-timeseries" # MLflow experiment name
```

This code downloads a publicly available version of the dataset, and then stores that dataset in a Fabric Lakehouse.

Important

Make sure you [add a lakehouse](#) to the notebook before running it. Failure to do so will result in an error.

Python

```
import os

if not os.path.exists("/lakehouse/default"):
    # Add a lakehouse if the notebook has no default lakehouse
    # A new notebook will not link to any lakehouse by default
    raise FileNotFoundError(
        "Default lakehouse not found, please add a lakehouse for the notebook."
    )
else:
    # Verify whether or not the required files are already in the lakehouse, and if not, download and unzip
    if not os.path.exists(f"{TAR_FILE_PATH}{TAR_FILE_NAME}"):
        os.makedirs(TAR_FILE_PATH, exist_ok=True)
        os.system(f"wget {URL}{TAR_FILE_NAME} -O {TAR_FILE_PATH}{TAR_FILE_NAME}")

    os.makedirs(CSV_FILE_PATH, exist_ok=True)
    os.system(f"tar -zxf {TAR_FILE_PATH}{TAR_FILE_NAME} -C {CSV_FILE_PATH}")
```

Start recording the run-time of this notebook.

Python

```
# Record the notebook running time
import time

ts = time.time()
```

Set up the MLflow experiment tracking

To extend the MLflow logging capabilities, autologging automatically captures the values of input parameters and output metrics of a machine learning model during its training. This information is then logged to the workspace, where the MLflow APIs or the corresponding experiment in the workspace can access and visualize it. Visit [this resource](#) for more information about autologging.

Python

```
# Set up the MLflow experiment
import mlflow

mlflow.set_experiment(EXPERIMENT_NAME)
mlflow.autolog(disable=True) # Disable MLflow autologging
```

ⓘ Note

If you want to disable Microsoft Fabric autologging in a notebook session, call `mlflow.autolog()` and set `disable=True`.

Read raw date data from the lakehouse

Python

```
df = (
    spark.read.format("csv")
    .option("header", "true")
    .load("Files/NYC_Property_Sales_Dataset/csv")
)
```

Step 3: Begin exploratory data analysis

To review the dataset, you could manually examine a subset of data to gain a better understanding of it. You can use the `display` function to print the DataFrame. You can also show the Chart views, to easily visualize subsets of the dataset.

Python

```
display(df)
```

A manual review of the dataset leads to some early observations:

- Instances of \$0.00 sales prices. According to the [Glossary of Terms](#), this implies a transfer of ownership with no cash consideration. In other words, no cash flowed in the transaction. You should remove sales with \$0.00 `sales_price` values from the dataset.
- The dataset covers different building classes. However, this notebook will focus on residential buildings which, according to the [Glossary of Terms](#), are marked as type "A". You should filter the dataset to include only residential buildings. To do this, include either the `building_class_at_time_of_sale` or the `building_class_at_present` columns. You must only include the `building_class_at_time_of_sale` data.
- The dataset includes instances where `total_units` values equal 0, or `gross_square_feet` values equal 0. You should remove all the instances where `total_units` or `gross_square_units` values equal 0.
- Some columns - for example, `apartment_number`, `tax_class`, `build_class_at_present`, etc. - have missing or NULL values. Assume that the missing data involves clerical errors, or nonexistent data. The analysis doesn't depend on these missing values, so you can ignore them.
- The `sale_price` column is stored as a string, with a prepended "\$" character. To proceed with the analysis, represent this column as a number. You should cast the `sale_price` column as integer.

Type conversion and filtering

To resolve some of the identified issues, import the required libraries.

Python

```
# Import libraries
import pyspark.sql.functions as F
from pyspark.sql.types import *
```

Cast the sales data from string to integer

Use regular expressions to separate the numeric portion of the string from the dollar sign (for example, in the string `$300,000`, split `$` and `300,000`), and then cast the numeric portion as an integer.

Next, filter the data to only include instances that meet all of these conditions:

1. The `sales_price` is greater than 0
2. The `total_units` is greater than 0
3. The `gross_square_feet` is greater than 0
4. The `building_class_at_time_of_sale` is of type A

Python

```
df = df.withColumn(  
    "sale_price", F regexp_replace("sale_price", "[\$,.]", "").cast(IntegerType()))  
)  
df = df.select("*").where(  
    'sale_price > 0 and total_units > 0 and gross_square_feet > 0 and building_class_at_time_of_sale like "A%"')  
)
```

Aggregation on monthly basis

The data resource tracks property sales on a daily basis, but this approach is too granular for this notebook. Instead, aggregate the data on a monthly basis.

First, change the date values to show only month and year data. The date values would still include the year data. You could still distinguish between, for example, December 2005 and December 2006.

Additionally, only keep the columns relevant to the analysis. These include `sales_price`, `total_units`, `gross_square_feet` and `sales_date`. You must also rename `sales_date` to `month`.

Python

```
monthly_sale_df = df.select(  
    "sale_price",  
    "total_units",  
    "gross_square_feet",  
    F.date_format("sale_date", "yyyy-MM").alias("month"),  
)  
display(monthly_sale_df)
```

Aggregate the `sale_price`, `total_units` and `gross_square_feet` values by month. Then, group the data by `month`, and sum all the values within each group.

Python

```
summary_df = (  
    monthly_sale_df.groupBy("month")  
    .agg(  
        F.sum("sale_price").alias("total_sales"),  
        F.sum("total_units").alias("units"),  
        F.sum("gross_square_feet").alias("square_feet"),  
    )  
    .orderBy("month")  
)  
  
display(summary_df)
```

Pyspark to Pandas conversion

Pyspark DataFrames handle large datasets well. However, due to data aggregation, the DataFrame size is smaller. This suggests that you can now use pandas DataFrames.

This code casts the dataset from a pyspark DataFrame to a pandas DataFrame.

Python

```
import pandas as pd
```

```
df_pandas = summary_df.toPandas()
display(df_pandas)
```

Visualization

You can examine the property trade trend of New York City to better understand the data. This leads to insights into potential patterns and seasonality trends. Learn more about Microsoft Fabric data visualization at [this](#) resource.

Python

```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

f, (ax1, ax2) = plt.subplots(2, 1, figsize=(35, 10))
plt.sca(ax1)
plt.xticks(np.arange(0, 15 * 12, step=12))
plt.ticklabel_format(style="plain", axis="y")
sns.lineplot(x="month", y="total_sales", data=df_pandas)
plt.ylabel("Total Sales")
plt.xlabel("Time")
plt.title("Total Property Sales by Month")

plt.sca(ax2)
plt.xticks(np.arange(0, 15 * 12, step=12))
plt.ticklabel_format(style="plain", axis="y")
sns.lineplot(x="month", y="square_feet", data=df_pandas)
plt.ylabel("Total Square Feet")
plt.xlabel("Time")
plt.title("Total Property Square Feet Sold by Month")
plt.show()
```

Summary of observations from the exploratory data analysis

- The data shows a clear recurring pattern on a yearly cadence; this means the data has a **yearly seasonality**
- The summer months seem to have higher sales volumes compared to winter months
- In a comparison of years with high sales and years with low sales, the revenue difference between high sales months and low sales months in high sales years exceeds - in absolute terms - the revenue difference between high sales months and low sales months in low sale years.

For example, in 2004, the revenue difference between the highest sales month and the lowest sales month is about:

$\$900,000,000 - \$500,000,000 = \$400,000,000$

For 2011, that revenue difference calculation is about:

$\$400,000,000 - \$300,000,000 = \$100,000,000$

This becomes important later, when you must decide between **multiplicative** and **additive** seasonality effects.

Step 4: Model training and tracking

Model fitting

[Prophet](#) input is always a two-column DataFrame. One input column is a time column named `ds`, and one input column is a value column named `y`. The time column should have a date, time, or datetime data format (for example, `YYYY-MM`). The dataset here meets that condition. The value column must be a numerical data format.

For the model fitting, you must only rename the time column to `ds` and value column to `y`, and pass the data to Prophet. Read the [Prophet Python API documentation](#) for more information.

Python

```
df_pandas["ds"] = pd.to_datetime(df_pandas["month"])
df_pandas["y"] = df_pandas["total_sales"]
```

Prophet follows the [scikit-learn](#) convention. First, create a new instance of Prophet, set certain parameters (for example, `seasonality_mode`), and then fit that instance to the dataset.

- Although a constant additive factor is the default seasonality effect for Prophet, you should use the '**multiplicative**' **seasonality** for the seasonality effect parameter. The analysis in the previous section showed that because of changes in seasonality amplitude, a simple additive seasonality won't fit the data well at all.
- Set the `weekly_seasonality` parameter to `off`, because the data was aggregated by month. As a result, weekly data isn't available.
- Use **Markov Chain Monte Carlo (MCMC)** methods to capture the seasonality uncertainty estimates. By default, Prophet can provide uncertainty estimates on the trend and observation noise, but not for the seasonality. MCMC require more processing time, but they allow the algorithm to provide uncertainty estimates on the seasonality, and the trend and observation noise. Read the [Prophet Uncertainty Intervals documentation](#) for more information.
- Tune the automatic change point detection sensitivity through the `changepoint_prior_scale` parameter. The Prophet algorithm automatically tries to find instances in the data where the trajectories abruptly change. It can become difficult to find the correct value. To resolve this, you can try different values and then select the model with the best performance. Read the [Prophet Trend Changepoints documentation](#) for more information.

```
Python
```

```
from prophet import Prophet

def fit_model(dataframe, seasonality_mode, weekly_seasonality, chpt_prior, mcmc_samples):
    m = Prophet(
        seasonality_mode=seasonality_mode,
        weekly_seasonality=weekly_seasonality,
        changepoint_prior_scale=chpt_prior,
        mcmc_samples=mcmc_samples,
    )
    m.fit(dataframe)
    return m
```

Cross validation

Prophet has a built-in cross-validation tool. This tool can estimate the forecasting error, and find the model with the best performance.

The cross-validation technique can validate model efficiency. This technique trains the model on a subset of the dataset, and runs tests on a previously-unseen subset of the dataset. This technique can check how well a statistical model generalizes to an independent dataset.

For cross-validation, reserve a particular sample of the dataset, which wasn't part of the training dataset. Then, test the trained model on that sample, prior to deployment. However, this approach doesn't work for time-series data, because if the model has seen data from the months of January 2005 and March 2005, and you try to predict for the month February 2005, the model can essentially *cheat*, because it could see where the data trend leads. In real applications, the aim is to forecast for the *future*, as the unseen regions.

To handle this, and make the test reliable, split the dataset based on the dates. Use the dataset up to a certain date (for example, the first 11 years of data) for training, and then use the remaining unseen data for prediction.

In this scenario, start with 11 years of training data, and then make monthly predictions using a one-year horizon. Specifically, the training data contains everything from 2003 through 2013. Then, the first run handles predictions for January 2014 through January 2015. The next run handles predictions for February 2014 through February 2015, and so on.

Repeat this process for each of the three trained models, to see which model performs the best. Then, compare these predictions with real-world values, to establish the prediction quality of the best model.

```
Python
```

```
from prophet.diagnostics import cross_validation
from prophet.diagnostics import performance_metrics

def evaluation(m):
    df_cv = cross_validation(m, initial="4017 days", period="30 days", horizon="365 days")
    df_p = performance_metrics(df_cv, monthly=True)
    future = m.make_future_dataframe(periods=12, freq="M")
    forecast = m.predict(future)
    return df_p, future, forecast
```

Log model with MLflow

Log the models, to keep track of their parameters, and save the models for later use. All relevant model information is logged in the workspace, under the experiment name. The model, parameters, and metrics, along with MLflow autologging items, is saved in one MLflow run.

Python

```
# Setup MLflow
from mlflow.models.signature import infer_signature
```

Conduct experiments

A machine learning experiment serves as the primary unit of organization and control, for all related machine learning runs. A run corresponds to a single execution of model code. Machine learning experiment tracking refers to the management of all the different experiments and their components. This includes parameters, metrics, models and other artifacts, and it helps organize the required components of a specific machine learning experiment. Machine learning experiment tracking also allows for the easy duplication of past results with saved experiments. Learn more about [machine learning experiments in Microsoft Fabric](#). Once you determine the steps you intend to include (for example, fitting and evaluating the Prophet model in this notebook), you can run the experiment.

Python

```
model_name = f"{EXPERIMENT_NAME}-prophet"

models = []
df_metrics = []
forecasts = []
seasonality_mode = "multiplicative"
weekly_seasonality = False
changepoint_priors = [0.01, 0.05, 0.1]
mcmc_samples = 100

for chpt_prior in changepoint_priors:
    with mlflow.start_run(run_name=f"prophet_changepoint_{chpt_prior}"):
        # init model and fit
        m = fit_model(df_pandas, seasonality_mode, weekly_seasonality, chpt_prior, mcmc_samples)
        models.append(m)
        # Validation
        df_p, future, forecast = evaluation(m)
        df_metrics.append(df_p)
        forecasts.append(forecast)
        # Log model and parameters with MLflow
        mlflow.prophet.log_model(
            m,
            model_name,
            registered_model_name=model_name,
            signature=infer_signature(future, forecast),
        )
        mlflow.log_params(
            {
                "seasonality_mode": seasonality_mode,
                "mcmc_samples": mcmc_samples,
                "weekly_seasonality": weekly_seasonality,
                "changepoint_prior": chpt_prior,
            }
        )
    metrics = df_p.mean().to_dict()
    metrics.pop("horizon")
    mlflow.log_metrics(metrics)
```

Properties

Description: prophet_change...

Run name: prophet_change...

Start date: 1/15/2025 2:09 AM

Duration: 28s

Status: Completed

Run ID: 18fd5de3-47b7...

Created by: Time series-829

Source: + 2

Run details

Run metrics (7)

Metric	Value
coverage	0.8344155844155844
mae	55312114.724166036
mape	0.09240464833916053
mdape	0.07229433467604929
mse	5078181985435596
rmse	70717769.07835788
smape	0.09943921881468122

Run parameters (4)

Parameter	Value
seasonality_mode	multiplicative
mcmc_samples	100
weekly_seasonality	False

Compare runs

Save run as an ML model

When an experimental run yields the desired result, save the run as an ML model for shared usage and tracking. Learn more

View run list

Save

Visualize a model with Prophet

Prophet has built-in visualization functions, which can show the model fitting results.

The black dots denote the data points that are used to train the model. The blue line is the prediction, and the light blue area shows the uncertainty intervals. You have built three models with different `changepoint_prior_scale` values. The predictions of these three models are shown in the results of this code block.

Python

```
for idx, pack in enumerate(zip(models, forecasts)):
    m, forecast = pack
    fig = m.plot(forecast)
    fig.suptitle(f"changepoint = {changepoint_priors[idx]}")
```

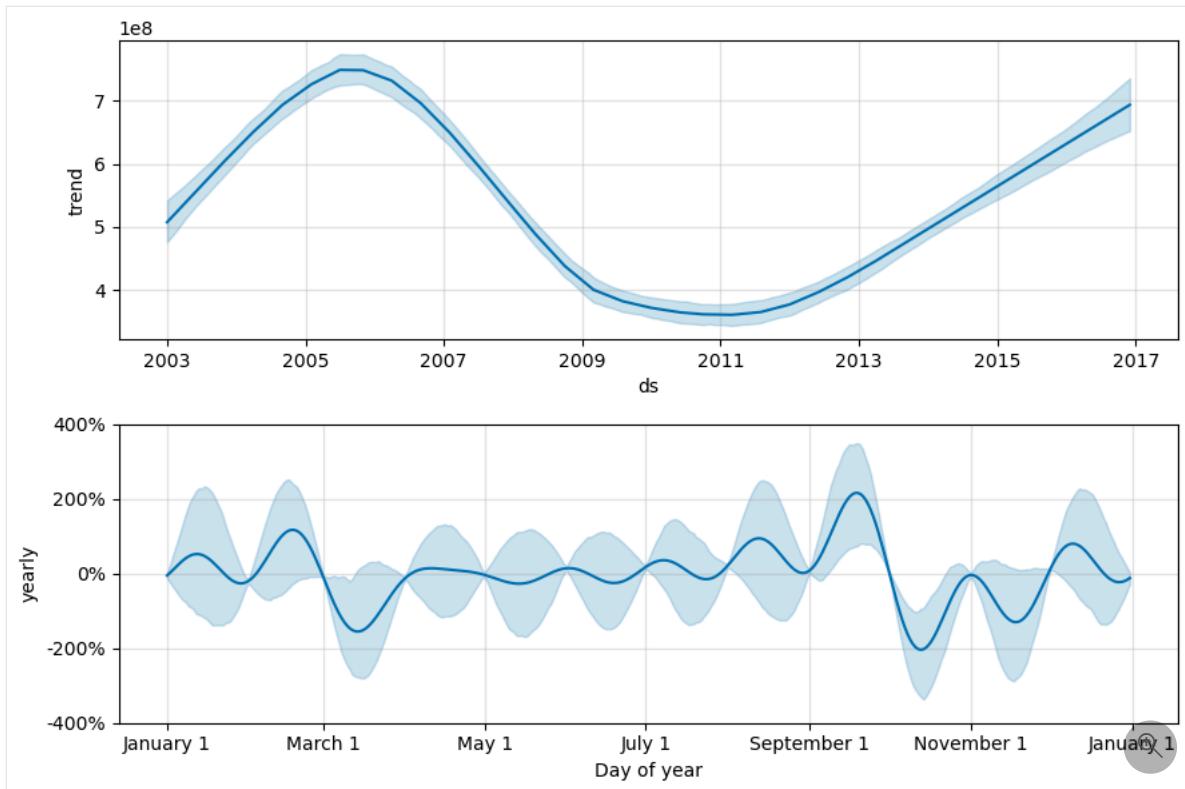
The smallest `changepoint_prior_scale` value in the first graph leads to an underfitting of trend changes. The largest `changepoint_prior_scale` in the third graph could result in overfitting. So, the second graph seems to be the optimal choice. This implies that the second model is the most suitable.

Visualize trends and seasonality with Prophet

Prophet can also easily visualize the underlying trends and seasonalities. Visualizations of the second model are shown in the results of this code block.

Python

```
BEST_MODEL_INDEX = 1 # Set the best model index according to the previous results
fig2 = models[BEST_MODEL_INDEX].plot_components(forecast)
```



In these graphs, the light blue shading reflects the uncertainty. The top graph shows a strong, long-period oscillating trend. Over a few years, the sales volumes rise and fall. The lower graph shows that sales tend to peak in February and September, reaching their maximum values for the year in those months. Shortly after those months, in March and October, they fall to the year's minimum values.

Evaluate the performance of the models using various metrics, for example:

- mean squared error (MSE)
- root mean squared error (RMSE)
- mean absolute error (MAE)
- mean absolute percent error (MAPE)
- median absolute percent error (MDAPE)
- symmetric mean absolute percentage error (SMAPE)

Evaluate the coverage using the `yhat_lower` and `yhat_upper` estimates. Note the varying horizons where you predict one year in the future, 12 times.

Python

```
display(df_metrics[BEST_MODEL_INDEX])
```

With the MAPE metric, for this forecasting model, predictions that extend one month into the future typically involve errors of roughly 8%. However, for predictions one year into the future, the error increases to roughly 10%.

Step 5: Score the model and save prediction results

Now score the model, and save the prediction results.

Make predictions with Predict Transformer

Now, you can load the model and use it to make predictions. Users can operationalize machine learning models with **PREDICT**, a scalable Microsoft Fabric function that supports batch scoring in any compute engine. Learn more about **PREDICT**, and how to use it within Microsoft Fabric, at [this resource](#).

Python

```
from synapse.ml.predict import MLFlowTransformer
spark.conf.set("spark.synapse.ml.predict.enabled", "true")
```

```
model = MLFlowTransformer(  
    inputCols=future.columns.values,  
    outputCol="prediction",  
    modelName=f"{EXPERIMENT_NAME}-prophet",  
    modelVersion=BEST_MODEL_INDEX,  
)  
  
test_spark = spark.createDataFrame(data=future, schema=future.columns.to_list())  
  
batch_predictions = model.transform(test_spark)  
  
display(batch_predictions)
```

Python

```
# Code for saving predictions into lakehouse  
batch_predictions.write.format("delta").mode("overwrite").save(  
    f"{DATA_FOLDER}/predictions/batch_predictions"  
)
```

Python

```
# Determine the entire runtime  
print(f"Full run cost {int(time.time() - ts)} seconds.")
```

Related content

- Machine learning model in Microsoft Fabric
- Train machine learning models
- Machine learning experiments in Microsoft Fabric

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Ask the community 

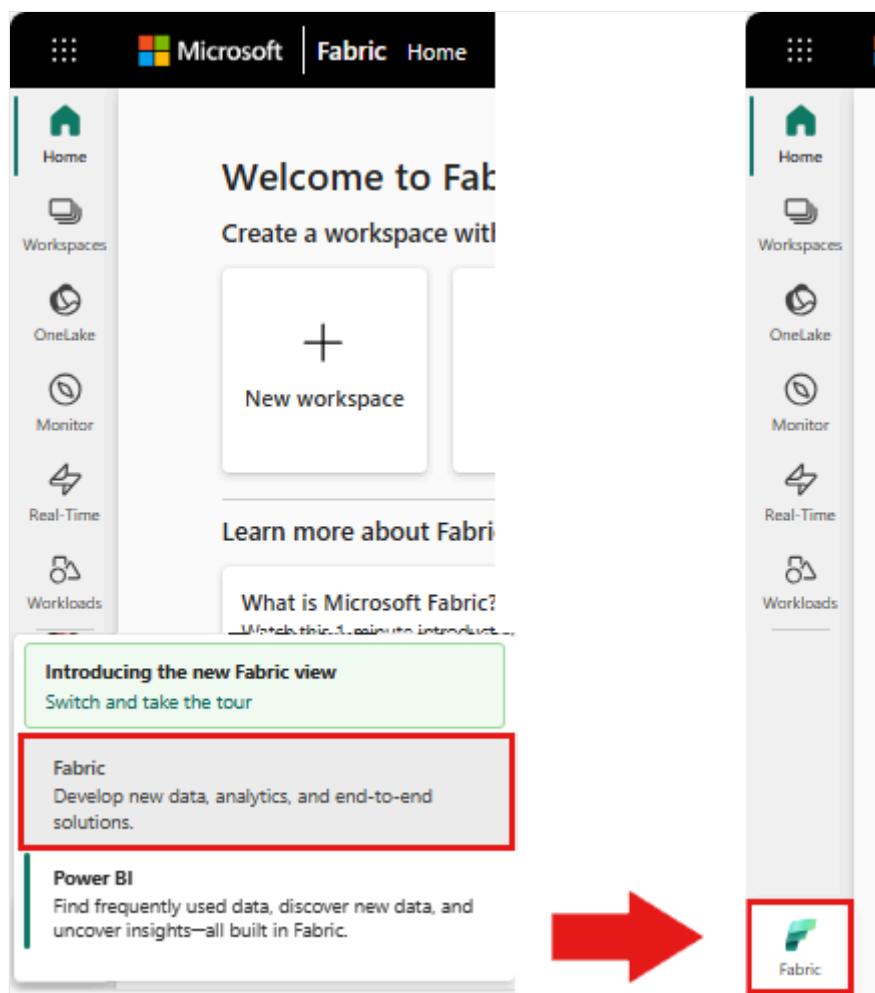
Tutorial: Create, train, and evaluate an uplift model

Article • 01/17/2025

This tutorial presents an end-to-end example of a Synapse Data Science workflow, in Microsoft Fabric. You learn how to create, train, and evaluate uplift models and apply uplift modeling techniques.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- Familiarity with [Microsoft Fabric notebooks](#)

- A lakehouse for this notebook, to store data for this example. For more information, visit [Add a lakehouse to your notebook](#)

Follow along in a notebook

You can follow along in a notebook in one of two ways:

- Open and run the built-in notebook.
- Upload your notebook from GitHub.

Open the built-in notebook

The sample **Uplift modeling** notebook accompanies this tutorial.

1. To open the sample notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#).
2. Make sure to [attach a lakehouse to the notebook](#) before you start running code.

Import the notebook from GitHub

The [Alsample - Uplift Modeling.ipynb](#) notebook accompanies this tutorial.

To open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#), to import the notebook to your workspace.

You can [create a new notebook](#) if you'd rather copy and paste the code from this page.

Be sure to [attach a lakehouse to the notebook](#) before you start running code.

Step 1: Load the data

Dataset

The Criteo AI Lab created the dataset. That dataset has 13M rows. Each row represents one user. Each row has 12 features, a treatment indicator, and two binary labels that include visit and conversion.

f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	treatment	conversion	visit	+	🔍
----	----	----	----	----	----	----	----	----	----	-----	-----	-----------	------------	-------	---	---

- **f0 - f11:** feature values (dense, floating values)

- **treatment**: whether or not a user was randomly target for treatment (for example, advertising) (1 = treatment, 0 = control)
- **conversion**: whether a conversion occurred (for example, made a purchase) for a user (binary, label)
- **visit**: whether a conversion occurred (for example, made a purchase) for a user (binary, label)

Citation

- Dataset homepage: <https://ailab.criteo.com/criteo-uplift-prediction-dataset/>

The dataset used for this notebook requires this BibTex citation:

```
@inproceedings{Diemert2018,
author = {{Diemert Eustache, Betlei Artem} and Renaudin, Christophe and Massih-Reza, Amini},
title={A Large Scale Benchmark for Uplift Modeling},
publisher = {ACM},
booktitle = {Proceedings of the AdKDD and TargetAd Workshop, KDD, London, United Kingdom, August, 20, 2018},
year = {2018}
}
```

💡 Tip

By defining the following parameters, you can apply this notebook on different datasets easily.

Python

```
IS_CUSTOM_DATA = False # If True, the user must upload the dataset manually
DATA_FOLDER = "Files/uplift-modelling"
DATA_FILE = "criteo-research-uplift-v2.1.csv"

# Data schema
FEATURE_COLUMNS = [f"f{i}" for i in range(12)]
TREATMENT_COLUMN = "treatment"
LABEL_COLUMN = "visit"

EXPERIMENT_NAME = "aisample-upliftmodelling" # MLflow experiment name
```

Import libraries

Before processing, you must import required Spark and SynapseML libraries. You must also import a data visualization library - for example, Seaborn, a Python data visualization library. A data visualization library provides a high-level interface to build visual resources on DataFrames and arrays. Learn more about [Spark](#), [SynapseML](#), and [Seaborn](#).

Python

```
import os
import gzip

import pyspark.sql.functions as F
from pyspark.sql.window import Window
from pyspark.sql.types import *

import numpy as np
import pandas as pd

import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.style as style
import seaborn as sns

%matplotlib inline

from synapse.ml.featurize import Featurize
from synapse.ml.core.spark import FluentAPI
from synapse.ml.lightgbm import *
from synapse.ml.train import ComputeModelStatistics

import mlflow
```

Download a dataset and upload to lakehouse

This code downloads a publicly available version of the dataset, and then stores that data resource in a Fabric lakehouse.

 **Important**

Make sure you [Add a lakehouse](#) to the notebook before you run it. Failure to do so will result in an error.

Python

```
if not IS_CUSTOM_DATA:
    # Download demo data files into lakehouse if not exist
    import os, requests
```

```
remote_url = "http://go.criteo.net/criteo-research-uplift-v2.1.csv.gz"
download_file = "criteo-research-uplift-v2.1.csv.gz"
download_path = f"/lakehouse/default/{DATA_FOLDER}/raw"

if not os.path.exists("/lakehouse/default"):
    raise FileNotFoundError("Default lakehouse not found, please add a
lakehouse and restart the session.")
os.makedirs(download_path, exist_ok=True)
if not os.path.exists(f"{download_path}/{DATA_FILE}"):
    r = requests.get(f"{remote_url}", timeout=30)
    with open(f"{download_path}/{download_file}", "wb") as f:
        f.write(r.content)
    with gzip.open(f"{download_path}/{download_file}", "rb") as fin:
        with open(f"{download_path}/{DATA_FILE}", "wb") as fout:
            fout.write(fin.read())
print("Downloaded demo data files into lakehouse.")
```

Start recording the runtime of this notebook.

Python

```
# Record the notebook running time
import time

ts = time.time()
```

Set up the MLflow experiment tracking

To extend the MLflow logging capabilities, autologging automatically captures the values of input parameters and output metrics of a machine learning model during its training. This information is then logged to the workspace, where the MLflow APIs or the corresponding experiment in the workspace can access and visualize it. Visit this resource for more information about autologging.

Python

```
# Set up the MLflow experiment
import mlflow

mlflow.set_experiment(EXPERIMENT_NAME)
mlflow.autolog(disable=True) # Disable MLflow autologging
```

 Note

To disable Microsoft Fabric autologging in a notebook session, call `mlflow.autolog()` and set `disable=True`.

Read data from the lakehouse

Read raw data from the lakehouse **Files** section and add more columns for different date parts. The same information is used to create a partitioned delta table.

Python

```
raw_df = spark.read.csv(f"{DATA_FOLDER}/raw/{DATA_FILE}", header=True,  
inferSchema=True).cache()
```

Step 2: Exploratory data analysis

Use the `display` command to view high-level statistics about the dataset. You can also show the Chart views to easily visualize subsets of the dataset.

Python

```
display(raw_df.limit(20))
```

Examine the percentage of the users that visit, the percentage of users that convert, and the percentage of the visitors that convert.

Python

```
raw_df.select(  
    F.mean("visit").alias("Percentage of users that visit"),  
    F.mean("conversion").alias("Percentage of users that convert"),  
    (F.sum("conversion") / F.sum("visit")).alias("Percentage of visitors  
that convert"),  
).show()
```

The analysis indicates that **4.9%** of users from the treatment group - users that received the treatment, or advertising - visited the online store. Only **3.8%** of users from the control group - users that never received the treatment, or were never offered or exposed to advertising - did the same. Additionally, **0.31%** of all users from the treatment group converted, or made a purchase - while only **0.19%** of users from the control group did so. As a result, the conversion rate of visitors that made a purchase, who were also members of treatment group, is **6.36%**, compared to only **5.07%**** for users of the control group. Based on these results, the treatment can potentially

improve the visit rate by about 1%, and the conversion rate of visitors by about 1.3%. The treatment leads to a significant improvement.

Step 3: Define the model for training

Prepare the training and test the datasets

Here, you fit a `Featurize` transformer to the `raw_df` DataFrame, to extract features from the specified input columns and output those features to a new column named `features`.

The resulting DataFrame is stored in a new DataFrame named `df`.

Python

```
transformer =  
    Featurize().setOutputCol("features").setInputCols(FEATURE_COLUMNS).fit(raw_d  
f)  
df = transformer.transform(raw_df)
```

Python

```
# Split the DataFrame into training and test sets, with a 80/20 ratio and a  
# seed of 42  
train_df, test_df = df.randomSplit([0.8, 0.2], seed=42)  
  
# Print the training and test dataset sizes  
print("Size of train dataset: %d" % train_df.count())  
print("Size of test dataset: %d" % test_df.count())  
  
# Group the training dataset by the treatment column, and count the number  
# of occurrences of each value  
train_df.groupby(TREATMENT_COLUMN).count().show()
```

Prepare the treatment and control datasets

After you create the training and test datasets, you must also form the treatment and control datasets, to train the machine learning models to measure the uplift.

Python

```
# Extract the treatment and control DataFrames  
treatment_train_df = train_df.where(f"{TREATMENT_COLUMN} > 0")  
control_train_df = train_df.where(f"{TREATMENT_COLUMN} = 0")
```

Now that you prepared your data, you can proceed to train a model with LightGBM.

Uplift modeling: T-Learner with LightGBM

Meta-learners are a set of algorithms, built on top of machine learning algorithms like LightGBM, Xgboost, etc. They help estimate conditional average treatment effect, or **CATE**. T-learner is a meta-learner that doesn't use a single model. Instead, T-learner uses one model per treatment variable. Therefore, two models are developed and we refer to the meta-learner as T-learner. T-learner uses multiple machine learning models to overcome the problem of entirely discarding the treatment, by forcing the learner to first split on it.

Python

```
mlflow.autolog(exclusive=False)
```

Python

```
classifier = (
    LightGBMClassifier(dataTransferMode="bulk")
    .setFeaturesCol("features") # Set the column name for features
    .setNumLeaves(10) # Set the number of leaves in each decision tree
    .setNumIterations(100) # Set the number of boosting iterations
    .setObjective("binary") # Set the objective function for binary
    classification
    .setLabelCol(LABEL_COLUMN) # Set the column name for the label
)

# Start a new MLflow run with the name "uplift"
active_run = mlflow.start_run(run_name="uplift")

# Start a new nested MLflow run with the name "treatment"
with mlflow.start_run(run_name="treatment", nested=True) as treatment_run:
    treatment_run_id = treatment_run.info.run_id # Get the ID of the
    treatment run
    treatment_model = classifier.fit(treatment_train_df) # Fit the
    classifier on the treatment training data

# Start a new nested MLflow run with the name "control"
with mlflow.start_run(run_name="control", nested=True) as control_run:
    control_run_id = control_run.info.run_id # Get the ID of the control
    run
    control_model = classifier.fit(control_train_df) # Fit the classifier
    on the control training data
```

Use the test dataset for a prediction

Here, you use the `treatment_model` and `control_model`, both defined earlier, to transform the `test_df` test dataset. Then, you calculate the predicted uplift. You define the predicted uplift as the difference between the predicted treatment outcome and the predicted control outcome. The greater this predicted uplift difference, the greater the effectiveness of the treatment (for example, advertising) on an individual or a subgroup.

Python

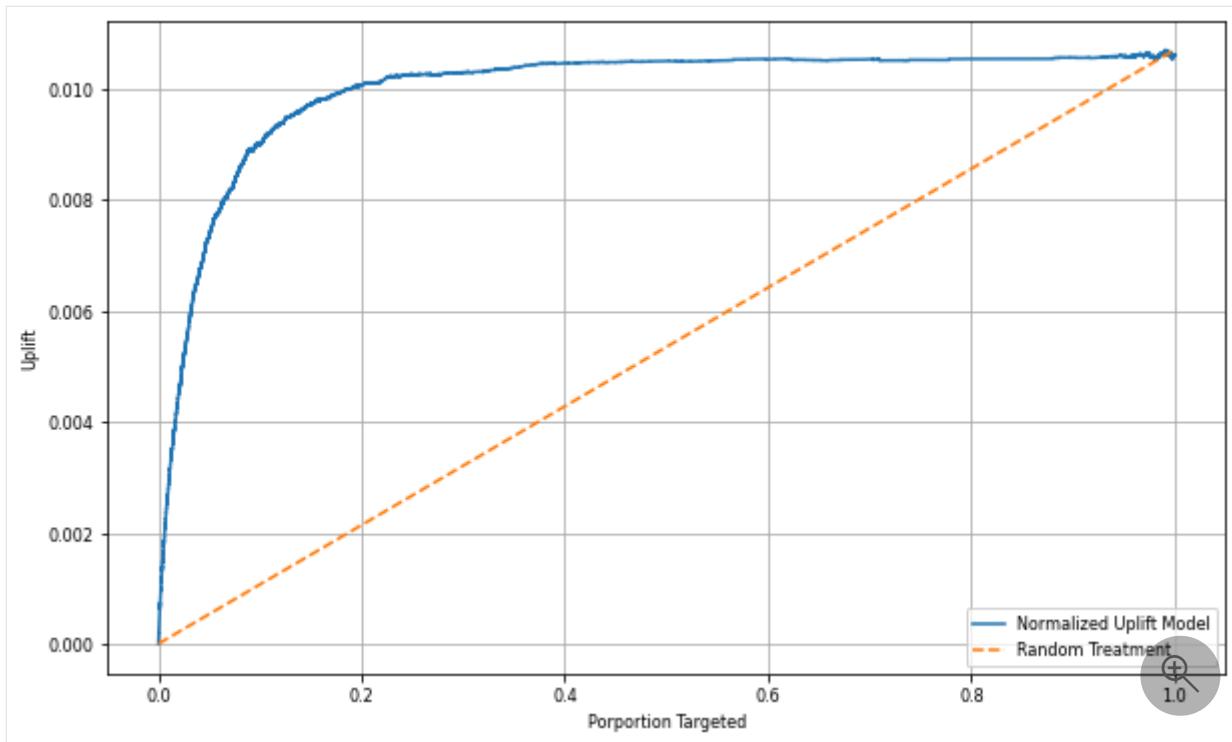
```
getPred = F.udf(lambda v: float(v[1]), FloatType())

# Cache the resulting DataFrame for easier access
test_pred_df = (
    test_df.mlTransform(treatment_model)
    .withColumn("treatment_pred", getPred("probability"))
    .drop("rawPrediction", "probability", "prediction")
    .mlTransform(control_model)
    .withColumn("control_pred", getPred("probability"))
    .drop("rawPrediction", "probability", "prediction")
    .withColumn("pred_uplift", F.col("treatment_pred") -
    F.col("control_pred"))
    .select(TREATMENT_COLUMN, LABEL_COLUMN, "treatment_pred",
    "control_pred", "pred_uplift")
    .cache()
)

# Display the first twenty rows of the resulting DataFrame
display(test_pred_df.limit(20))
```

Perform model evaluation

Since actual uplift can't be observed for each individual, you need to measure the uplift over a group of individuals. You use an Uplift Curve that plots the real, cumulative uplift across the population.



The x-axis represents the ratio of the population selected for the treatment. A value of 0 suggests no treatment group - no one is exposed to, or offered, the treatment. A value of 1 suggests a full treatment group - everyone is exposed to, or offered, the treatment. The y-axis shows the uplift measure. The aim is to find the size of the treatment group, or the percentage of the population that would be offered or exposed to the treatment (for example, advertising). This approach optimizes the target selection, to optimize the outcome.

First, rank the test DataFrame order by the predicted uplift. The predicted uplift is the difference between the predicted treatment outcome and the predicted control outcome.

Python

```
# Compute the percentage rank of the predicted uplift values in descending
# order, and display the top twenty rows
test_ranked_df = test_pred_df.withColumn("percent_rank",
F.percent_rank().over(Window.orderBy(F.desc("pred_uplift"))))

display(test_ranked_df.limit(20))
```

Next, calculate the cumulative percentage of visits in both the treatment and control groups.

Python

```
# Calculate the number of control and treatment samples
C = test_ranked_df.where(F"{TREATMENT_COLUMN} == 0").count()
```

```

T = test_ranked_df.where(F"{TREATMENT_COLUMN} != 0").count()

# Add columns to the DataFrame to calculate the control and treatment
# cumulative sum
test_ranked_df = (
    test_ranked_df.withColumn(
        "control_label",
        F.when(F.col(TREATMENT_COLUMN) == 0,
F.col(LABEL_COLUMN)).otherwise(0),
    )
    .withColumn(
        "treatment_label",
        F.when(F.col(TREATMENT_COLUMN) != 0,
F.col(LABEL_COLUMN)).otherwise(0),
    )
    .withColumn(
        "control_cumsum",
        F.sum("control_label").over(Window.orderBy("percent_rank")) / C,
    )
    .withColumn(
        "treatment_cumsum",
        F.sum("treatment_label").over(Window.orderBy("percent_rank")) / T,
    )
)

# Display the first 20 rows of the dataframe
display(test_ranked_df.limit(20))

```

Finally, at each percentage, calculate the uplift of the group as the difference between the cumulative percentage of visits between the treatment and control groups.

Python

```

test_ranked_df = test_ranked_df.withColumn("group_uplift",
F.col("treatment_cumsum") - F.col("control_cumsum")).cache()
display(test_ranked_df.limit(20))

```

Now, plot the uplift curve for the test dataset prediction. You must convert the PySpark DataFrame to a Pandas DataFrame before plotting.

Python

```

def uplift_plot(uplift_df):
    """
    Plot the uplift curve
    """
    gain_x = uplift_df.percent_rank
    gain_y = uplift_df.group_uplift
    # Plot the data
    fig = plt.figure(figsize=(10, 6))
    mpl.rcParams["font.size"] = 8

```

```

        ax = plt.plot(gain_x, gain_y, color="#2077B4", label="Normalized Uplift Model")

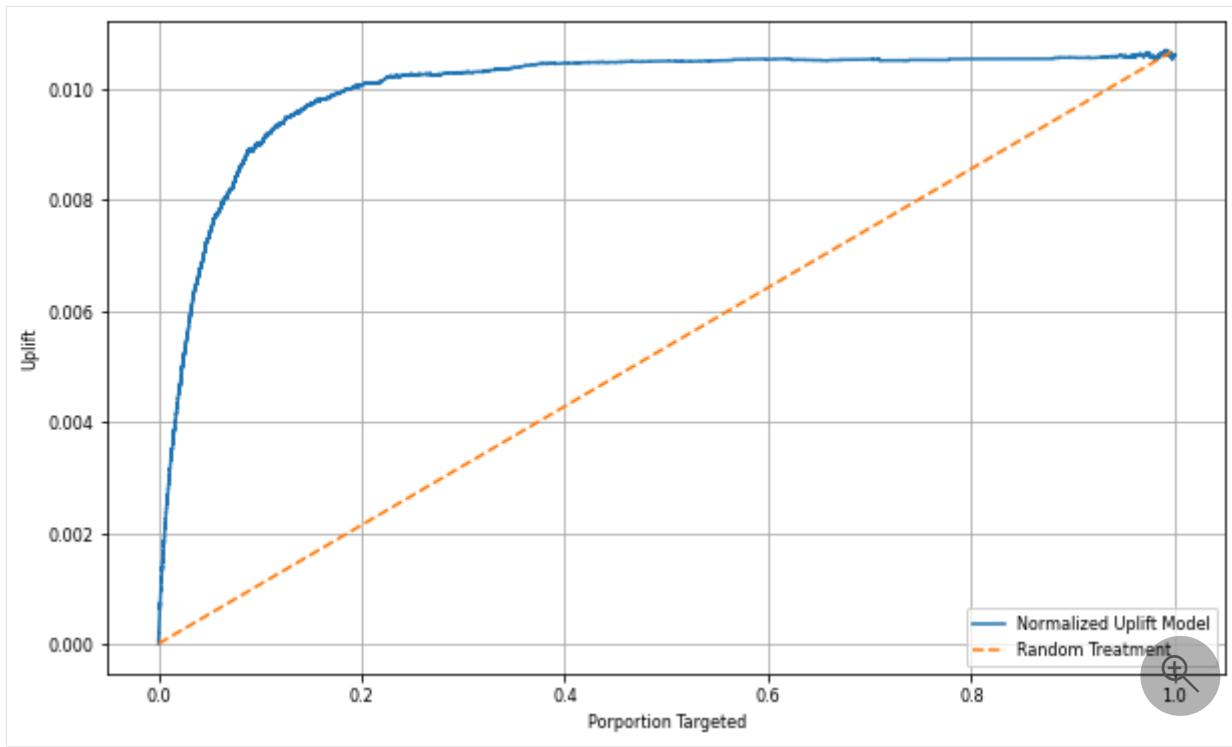
    plt.plot(
        [0, gain_x.max()],
        [0, gain_y.max()],
        "--",
        color="tab:orange",
        label="Random Treatment",
    )
    plt.legend()
    plt.xlabel("Porportion Targeted")
    plt.ylabel("Uplift")
    plt.grid()

    return fig, ax

test_ranked_pd_df = test_ranked_df.select(["pred_uplift", "percent_rank",
"group_uplift"]).toPandas()
fig, ax = uplift_plot(test_ranked_pd_df)

mlflow.log_figure(fig, "UpliftCurve.png")

```



The x-axis represents the ratio of the population selected for the treatment. A value of 0 suggests no treatment group - no one is exposed to, or offered, the treatment. A value of 1 suggests a full treatment group - everyone is exposed to, or offered, the treatment. The y-axis shows the uplift measure. The aim is to find the size of the treatment group, or the percentage of the population that would be offered or exposed to the treatment

(for example, advertising). This approach optimizes the target selection, to optimize the outcome.

First, rank the test DataFrame order by the predicted uplift. The predicted uplift is the difference between the predicted treatment outcome and the predicted control outcome.

Python

```
# Compute the percentage rank of the predicted uplift values in descending
# order, and display the top twenty rows
test_ranked_df = test_pred_df.withColumn("percent_rank",
F.percent_rank().over(Window.orderBy(F.desc("pred_uplift"))))

display(test_ranked_df.limit(20))
```

Next, calculate the cumulative percentage of visits in both the treatment and control groups.

Python

```
# Calculate the number of control and treatment samples
C = test_ranked_df.where(f"{TREATMENT_COLUMN} == 0").count()
T = test_ranked_df.where(f"{TREATMENT_COLUMN} != 0").count()

# Add columns to the DataFrame to calculate the control and treatment
# cumulative sum
test_ranked_df = (
    test_ranked_df.withColumn(
        "control_label",
        F.when(F.col(TREATMENT_COLUMN) == 0,
F.col(LABEL_COLUMN)).otherwise(0),
    )
    .withColumn(
        "treatment_label",
        F.when(F.col(TREATMENT_COLUMN) != 0,
F.col(LABEL_COLUMN)).otherwise(0),
    )
    .withColumn(
        "control_cumsum",
        F.sum("control_label").over(Window.orderBy("percent_rank")) / C,
    )
    .withColumn(
        "treatment_cumsum",
        F.sum("treatment_label").over(Window.orderBy("percent_rank")) / T,
    )
)

# Display the first 20 rows of the dataframe
display(test_ranked_df.limit(20))
```

Finally, at each percentage, calculate the uplift of the group as the difference between the cumulative percentage of visits between the treatment and control groups.

```
Python
```

```
test_ranked_df = test_ranked_df.withColumn("group_uplift",
F.col("treatment_cumsum") - F.col("control_cumsum")).cache()
display(test_ranked_df.limit(20))
```

Now, plot the uplift curve for the test dataset prediction. You must convert the PySpark DataFrame to a Pandas DataFrame before plotting.

```
Python
```

```
def uplift_plot(uplift_df):
    """
    Plot the uplift curve
    """
    gain_x = uplift_df.percent_rank
    gain_y = uplift_df.group_uplift
    # Plot the data
    fig = plt.figure(figsize=(10, 6))
    mpl.rcParams["font.size"] = 8

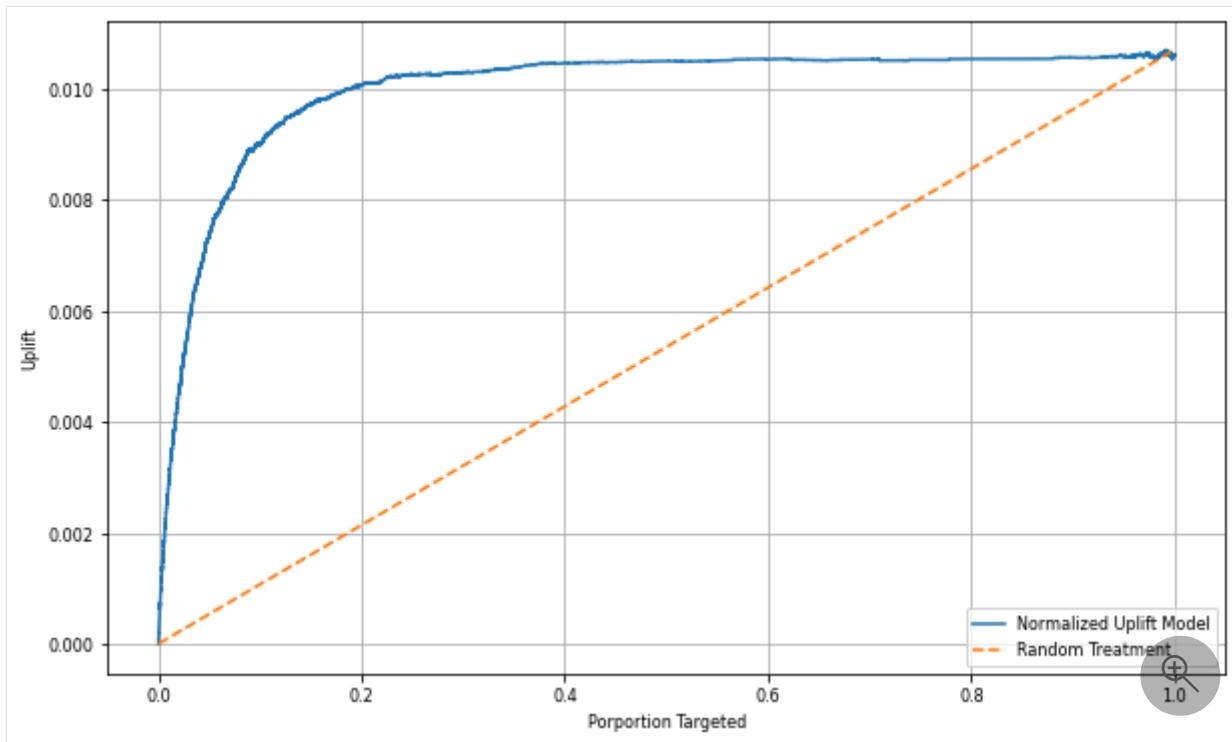
    ax = plt.plot(gain_x, gain_y, color="#2077B4", label="Normalized Uplift Model")

    plt.plot(
        [0, gain_x.max()],
        [0, gain_y.max()],
        "--",
        color="tab:orange",
        label="Random Treatment",
    )
    plt.legend()
    plt.xlabel("Porportion Targeted")
    plt.ylabel("Uplift")
    plt.grid()

    return fig, ax

test_ranked_pd_df = test_ranked_df.select(["pred_uplift", "percent_rank",
"group_uplift"]).toPandas()
fig, ax = uplift_plot(test_ranked_pd_df)

mlflow.log_figure(fig, "UpliftCurve.png")
```



The analysis and the uplift curve both show that the top 20% population, as ranked by the prediction, would have a large gain if they received the treatment. This means that the top 20% of the population represents the persuadables group. Therefore, you can then set the cutoff score for the desired size of treatment group at 20%, to identify the target selection customers for the greatest impact.

Python

```
cutoff_percentage = 0.2
cutoff_score = test_ranked_pd_df.iloc[int(len(test_ranked_pd_df) * 
cutoff_percentage)][
    "pred_uplift"]
]

print("Uplift scores that exceed {:.4f} map to
Persuadables.".format(cutoff_score))
mlflow.log_metrics(
    {"cutoff_score": cutoff_score, "cutoff_percentage": cutoff_percentage}
)
```

Step 4: Register the final ML Model

You use MLflow to track and log all experiments for both treatment and control groups. This tracking and logging include the corresponding parameters, metrics, and the models. This information is logged under the experiment name, in the workspace, for later use.

Python

```

# Register the model
treatment_model_uri = "runs:/{}model".format(treatment_run_id)
mlflow.register_model(treatment_model_uri, f"{EXPERIMENT_NAME}-treatmentmodel")

control_model_uri = "runs:/{}model".format(control_run_id)
mlflow.register_model(control_model_uri, f"{EXPERIMENT_NAME}-controlmodel")

mlflow.end_run()

```

To view your experiments:

1. On the left panel, select your workspace.
2. Find and select the experiment name, in this case *aisample-upliftmodelling*.

Step 5: Save the prediction results

Microsoft Fabric offers PREDICT - a scalable function that supports batch scoring in any compute engine. It enables customers to operationalize machine learning models. Users can create batch predictions straight from a notebook or the item page for a specific model. Visit this resource to learn more about PREDICT, and to learn how to use PREDICT in Microsoft Fabric.

Python

```

# Load the model back
loaded_treatmentmodel = mlflow.spark.load_model(treatment_model_uri,
dfs_tmpdir="Files/spark")

```

```
loaded_controlmodel = mlflow.spark.load_model(control_model_uri,
dfs_ttmpdir="Files/spark")

# Make predictions
batch_predictions_treatment = loaded_treatmentmodel.transform(test_df)
batch_predictions_control = loaded_controlmodel.transform(test_df)
batch_predictions_treatment.show(5)
```

Python

```
# Save the predictions in the lakehouse
batch_predictions_treatment.write.format("delta").mode("overwrite").save(
    f"{DATA_FOLDER}/predictions/batch_predictions_treatment"
)
batch_predictions_control.write.format("delta").mode("overwrite").save(
    f"{DATA_FOLDER}/predictions/batch_predictions_control"
)
```

Python

```
# Determine the entire runtime
print(f"Full run cost {int(time.time() - ts)} seconds.")
```

Related content

- Machine learning model in Microsoft Fabric
- Train machine learning models
- Machine learning experiments in Microsoft Fabric

Feedback

Was this page helpful?



Yes



No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Tutorial: Use R to create, evaluate, and score a churn prediction model

Article • 02/07/2024

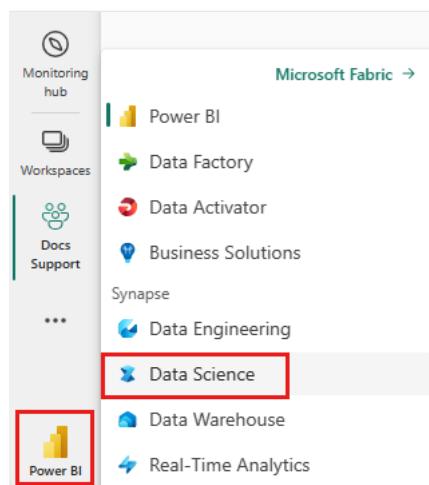
This tutorial presents an end-to-end example of a Synapse Data Science workflow in Microsoft Fabric. The scenario builds a model to predict whether or not bank customers churn. The churn rate, or the rate of attrition, involves the rate at which bank customers end their business with the bank.

This tutorial covers these steps:

- ✓ Install custom libraries
- ✓ Load the data
- ✓ Understand and process the data through exploratory data analysis
- ✓ Use scikit-learn and LightGBM to train machine learning models
- ✓ Evaluate and save the final machine learning model
- ✓ Show the model performance with Power BI visualizations

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



- If necessary, create a Microsoft Fabric lakehouse as described in [Create a lakehouse in Microsoft Fabric](#).

Follow along in a notebook

You can choose one of these options to follow along in a notebook:

- Open and run the built-in notebook in the Synapse Data Science experience.
- Upload your notebook from GitHub to the Synapse Data Science experience.

Open the built-in notebook

The sample **Customer churn** notebook accompanies this tutorial.

To open the tutorial's built-in sample notebook in the Synapse Data Science experience:

1. Go to the Synapse Data Science home page.
2. Select **Use a sample**.
3. Select the corresponding sample:
 - From the default **End-to-end workflows (Python)** tab, if the sample is for a Python tutorial.
 - From the **End-to-end workflows (R)** tab, if the sample is for an R tutorial.

- From the **Quick tutorials** tab, if the sample is for a quick tutorial.

4. [Attach a lakehouse to the notebook](#) before you start running code.

Import the notebook from GitHub

The [Alsample - R Bank Customer Churn.ipynb](#) notebook accompanies this tutorial.

To open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#), to import the notebook to your workspace.

If you'd rather copy and paste the code from this page, you can [create a new notebook](#).

Be sure to [attach a lakehouse to the notebook](#) before you start running code.

Step 1: Install custom libraries

For machine learning model development or ad-hoc data analysis, you might need to quickly install a custom library for your Apache Spark session. You have two options to install libraries.

- Use inline installation resources, for example `install.packages` and `devtools::install_version`, to install in your current notebook only.
- Alternatively, you can create a Fabric environment, install libraries from public sources or upload custom libraries to it, and then your workspace admin can attach the environment as the default for the workspace. All the libraries in the environment will then become available for use in any notebooks and Spark job definitions in the workspace. For more information on environments, see [create, configure, and use an environment in Microsoft Fabric](#).

In this tutorial, use `install.packages()` to install the `imbalance` and `randomForest` libraries. Set `quiet` to `TRUE` to make the output more concise:

```
R

# Install imbalance for SMOTE
install.packages("imbalance", quiet = TRUE)
# Install the random forest algorithm
install.packages("randomForest", quiet=TRUE)
```

Step 2: Load the data

The dataset in `churn.csv` contains the churn status of 10,000 customers, along with 14 attributes that include:

- Credit score
- Geographical location (Germany, France, Spain)
- Gender (male, female)
- Age
- Tenure (number of years the person was a customer at that bank)
- Account balance
- Estimated salary
- Number of products that a customer purchased through the bank
- Credit card status (whether or not a customer has a credit card)
- Active member status (whether or not the person is an active bank customer)

The dataset also includes row number, customer ID, and customer surname columns. Values in these columns shouldn't influence a customer's decision to leave the bank.

A customer bank account closure event defines the churn for that customer. The dataset `Exited` column refers to the customer's abandonment. Since we have little context about these attributes, we don't need background information about the dataset. We want to understand how these attributes contribute to the `Exited` status.

Out of the 10,000 customers, only 2037 customers (around 20%) left the bank. Because of the class imbalance ratio, we recommend generating synthetic data generation.

This table shows a preview sample of the `churn.csv` data:

 Expand table

CustomerID	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary
15634602	Hargrave	619	France	Female	42	2	0.00	1	1	1	101348.88
15647311	Hill	608	Spain	Female	41	1	83807.86	1	0	1	112542.58

Download the dataset and upload to the lakehouse

Important

Add a lakehouse  to the notebook before you run it. Failure to do so will result in an error.

This code downloads a publicly available version of the dataset, and then stores that data in a Fabric lakehouse:

```
R

library(fs)
library(httr)

remote_url <- "https://sdkstorerta.blob.core.windows.net/churnblob"
file_list <- c("churn.csv")
download_path <- "/lakehouse/default/Files/churn/raw"

if (!dir_exists("/lakehouse/default")) {
  stop("Default lakehouse not found, please add a lakehouse and restart the session.")
}
dir_create(download_path, recurse= TRUE)
for (fname in file_list) {
  if (!file_exists(paste0(download_path, "/", fname))) {
    r <- GET(paste0(remote_url, "/", fname), timeout(30))
    writeBin(content(r, "raw"), paste0(download_path, "/", fname))
  }
}
print("Downloaded demo data files into lakehouse.")
```

Start recording the time needed to run this notebook:

```
R

# Record the notebook running time
ts <- as.numeric(Sys.time())
```

Read raw date data from the lakehouse

This code reads raw data from the **Files** section of the lakehouse:

```
R

fname <- "churn.csv"
download_path <- "/lakehouse/default/Files/churn/raw"
rdf <- readr::read_csv(paste0(download_path, "/", fname))
```

Step 3: Perform exploratory data analysis

Display raw data

Use the `head()` or `str()` commands to perform a preliminary exploration of the raw data:

```
R

head(rdf)
```

Perform initial data cleaning

You must convert the R DataFrame to a Spark DataFrame. These operations on the Spark DataFrame clean the raw dataset:

- Drop the rows that have missing data across all columns
- Drop the duplicate rows across the columns `RowNumber` and `CustomerId`
- Drop the columns `RowNumber`, `CustomerId`, and `Surname`

R

```
# Transform the R DataFrame to a Spark DataFrame
df <- as.DataFrame(rdf)

clean_data <- function(df) {
  sdf <- df %>%
    # Drop rows that have missing data across all columns
    na.omit() %>%
    # Drop duplicate rows in columns: 'RowNumber', 'CustomerId'
    dropDuplicates(c("RowNumber", "CustomerId")) %>%
    # Drop columns: 'RowNumber', 'CustomerId', 'Surname'
    SparkR::select("CreditScore", "Geography", "Gender", "Age", "Tenure", "Balance", "NumOfProducts", "HasCrCard",
    "IsActiveMember", "EstimatedSalary", "Exited")
  return(sdf)
}

df_clean <- clean_data(df)
```

Explore the Spark DataFrame with the `display` command:

R

```
display(df_clean)
```

This code determines the categorical, numerical, and target attributes:

R

```
# Determine the dependent (target) attribute
dependent_variable_name <- "Exited"
print(dependent_variable_name)

# Obtain the distinct values for each column
exprs = lapply(names(df_clean), function(x) alias(countDistinct(df_clean[[x]]), x))
# Use do.call to splice the aggregation expressions to aggregate function
distinct_value_number <- SparkR::collect(do.call(agg, c(x = df_clean, exprs)))

# Determine the categorical attributes
categorical_variables <- names(df_clean)[sapply(names(df_clean), function(col) col %in% c("0") ||
  distinct_value_number[[col]] <= 5 && !(col %in% c(dependent_variable_name)))]
print(categorical_variables)

# Determine the numerical attributes
numeric_variables <- names(df_clean)[sapply(names(df_clean), function(col) coltypes(SparkR::select(df_clean, col)) ==
  "numeric" && distinct_value_number[[col]] > 5)]
print(numeric_variables)
```

For easier processing and visualization, convert the cleaned Spark DataFrame to an R DataFrame:

R

```
# Transform the Spark DataFrame to an R DataFrame
rdf_clean <- SparkR::collect(df_clean)
```

Show the five-number summary

Use box plots to show the five-number summary (minimum score, first quartile, median, third quartile, maximum score) for the numerical attributes:

R

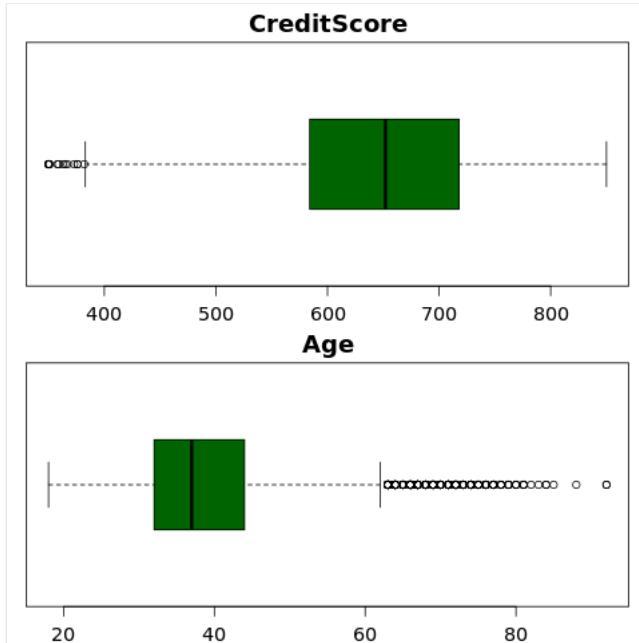
```
# Set the overall layout of the graphics window
par(mfrow = c(2, 1),
  mar = c(2, 1, 2, 1)) # Margin size

for(item in numeric_variables[1:2]) {
```

```

# Create a box plot
boxplot(rdf_clean[, item],
        main = item,
        col = "darkgreen",
        cex.main = 1.5, # Title size
        cex.lab = 1.3, # Axis label size
        cex.axis = 1.2,
        horizontal = TRUE) # Axis size
}

```



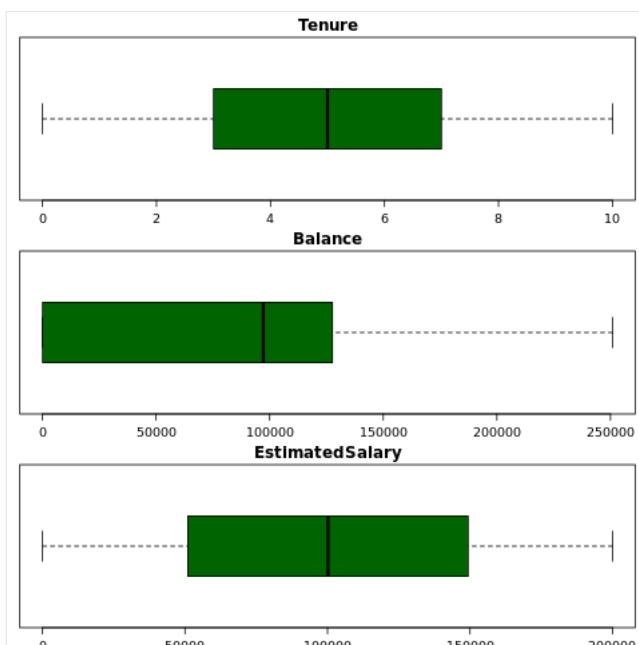
R

```

# Set the overall layout of the graphics window
par(mfrow = c(3, 1),
    mar = c(2, 1, 2, 1)) # Margin size

for(item in numeric_variables[3:5]){
  # Create a box plot
  boxplot(rdf_clean[, item],
          main = item,
          col = "darkgreen",
          cex.main = 1.5, # Title size
          cex.lab = 1.3, # Axis label size
          cex.axis = 1.2,
          horizontal = TRUE) # Axis size
}

```

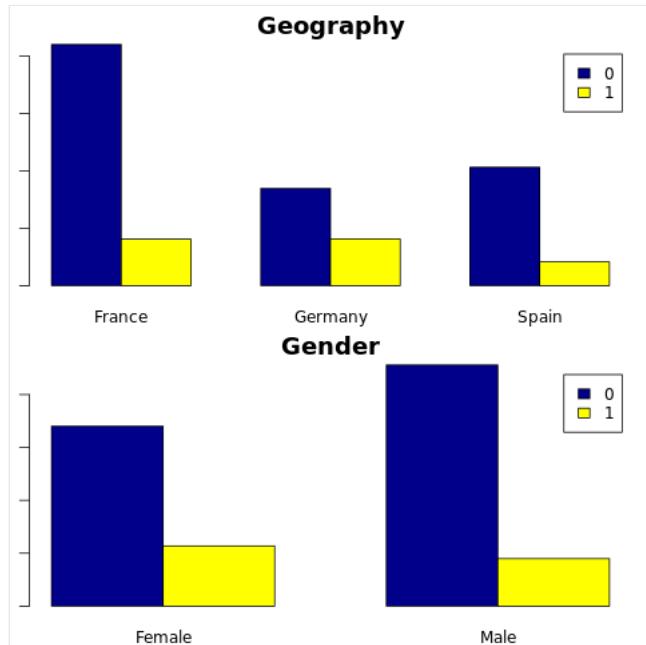


Show the distribution of exited and non-exited customers

Show the distribution of exited versus non-exited customers, across the categorical attributes:

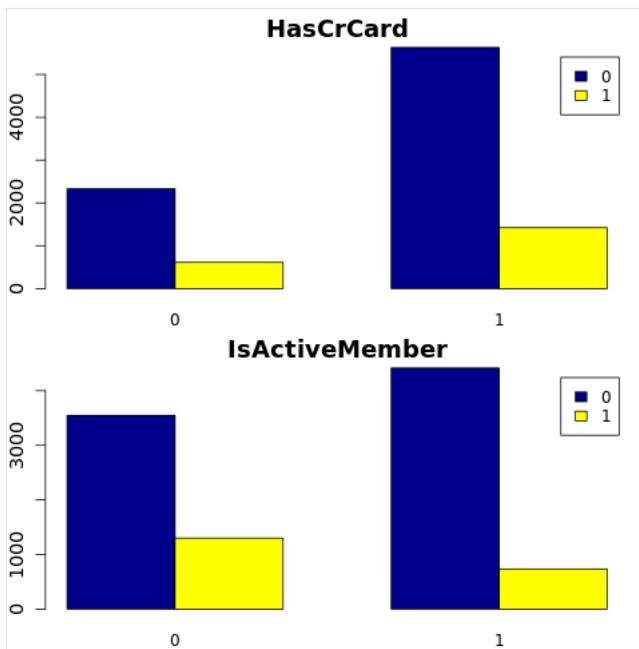
R

```
attr_list <- c('Geography', 'Gender', 'HasCrCard', 'IsActiveMember', 'NumOfProducts', 'Tenure')
par(mfrow = c(2, 1),
  mar = c(2, 1, 2, 1)) # Margin size
for (item in attr_list[1:2]) {
  counts <- table(rdf_clean$Exited, rdf_clean[,item])
  barplot(counts, main=item, col=c("darkblue","yellow"),
          cex.main = 1.5, # Title size
          cex.axis = 1.2,
          legend = rownames(counts), beside=TRUE)
}
```



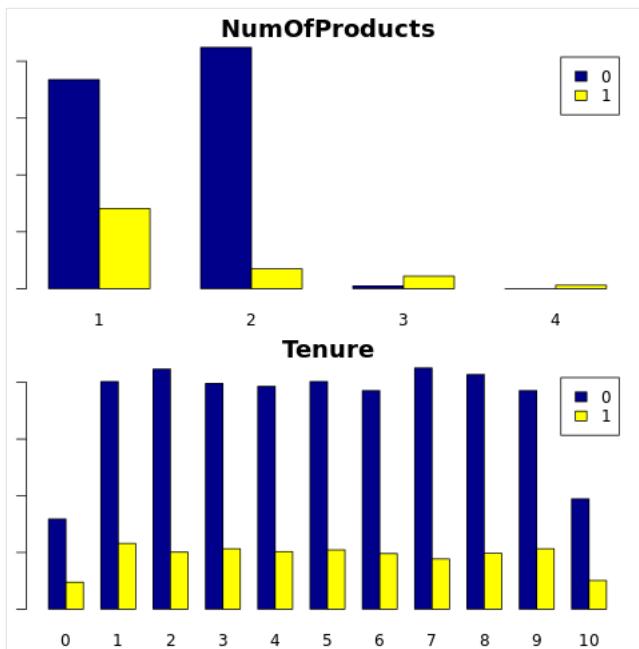
R

```
par(mfrow = c(2, 1),
  mar = c(2, 2, 2, 1)) # Margin size
for (item in attr_list[3:4]) {
  counts <- table(rdf_clean$Exited, rdf_clean[,item])
  barplot(counts, main=item, col=c("darkblue","yellow"),
          cex.main = 1.5, # Title size
          cex.axis = 1.2,
          legend = rownames(counts), beside=TRUE)
}
```



R

```
par(mfrow = c(2, 1),
  mar = c(2, 1, 2, 1)) # Margin size
for (item in attr_list[5:6]) {
  counts <- table(rdf_clean$Exited, rdf_clean[,item])
  barplot(counts, main=item, col=c("darkblue","yellow"),
    cex.main = 1.5, # Title size
    cex.axis = 1.2,
    legend = rownames(counts), beside=TRUE)
}
```



Show the distribution of numerical attributes

Use a histogram to show the frequency distribution of numerical attributes:

```
R

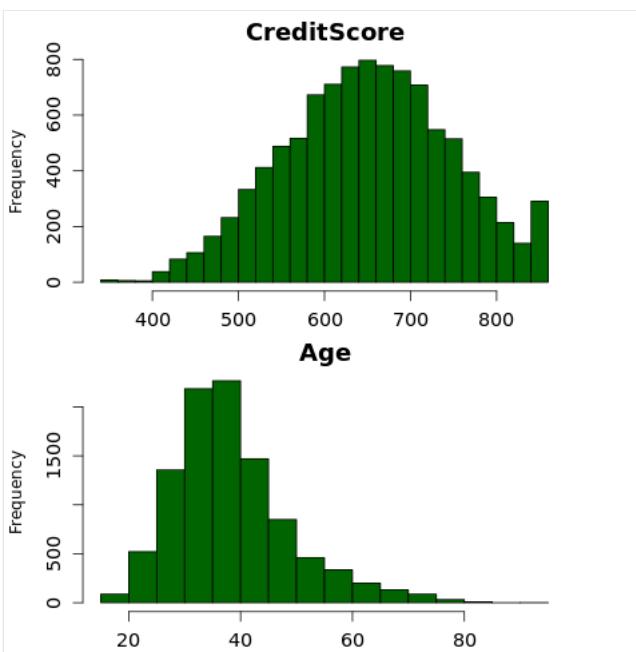
# Set the overall layout of the graphics window
par(mfrow = c(2, 1),
  mar = c(2, 4, 2, 4) + 0.1) # Margin size

# Create a histogram
for (item in numeric_variables[1:2]) {
  hist(rdf_clean[, item],
    main = item,
```

```

        col = "darkgreen",
        xlab = item,
        cex.main = 1.5, # Title size
        cex.axis = 1.2,
        breaks = 20) # Number of bins
    }
}

```



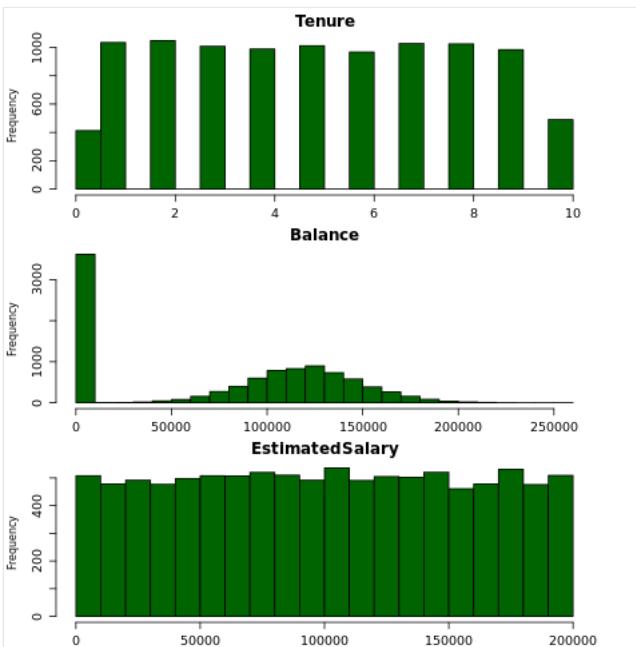
R

```

# Set the overall layout of the graphics window
par(mfrow = c(3, 1),
     mar = c(2, 4, 2, 4) + 0.1) # Margin size

# Create a histogram
for (item in numeric_variables[3:5]) {
  hist(rdf_clean[, item],
       main = item,
       col = "darkgreen",
       xlab = item,
       cex.main = 1.5, # Title size
       cex.axis = 1.2,
       breaks = 20) # Number of bins
}

```



Perform feature engineering

This feature engineering generates new attributes based on current attributes:

```
R

rdf_clean$NewTenure <- rdf_clean$Tenure / rdf_clean$Age
rdf_clean$NewCreditsScore <- as.numeric(cut(rdf_clean$CreditScore, breaks=quantile(rdf_clean$CreditScore, probs=seq(0, 1, by=1/6)), include.lowest=TRUE, labels=c(1, 2, 3, 4, 5, 6)))
rdf_clean$NewAgeScore <- as.numeric(cut(rdf_clean$Age, breaks=quantile(rdf_clean$Age, probs=seq(0, 1, by=1/8)), include.lowest=TRUE, labels=c(1, 2, 3, 4, 5, 6, 7, 8)))
rdf_clean$NewBalanceScore <- as.numeric(cut(rank(rdf_clean$Balance), breaks=quantile(rank(rdf_clean$Balance, ties.method = "first"), probs=seq(0, 1, by=1/5)), include.lowest=TRUE, labels=c(1, 2, 3, 4, 5)))
rdf_clean$NewEstSalaryScore <- as.numeric(cut(rdf_clean$EstimatedSalary, breaks=quantile(rdf_clean$EstimatedSalary, probs=seq(0, 1, by=1/10)), include.lowest=TRUE, labels=c(1:10)))
```

Perform one-hot encoding

Use one-hot encoding to convert the categorical attributes to numerical attributes, to feed them into the machine learning model:

```
R

rdf_clean <- cbind(rdf_clean, model.matrix(~Geography+Gender-1, data=rdf_clean))
rdf_clean <- subset(rdf_clean, select = - c(Geography, Gender))
```

Create a delta table to generate the Power BI report

```
R

table_name <- "rdf_clean"
# Create a Spark DataFrame from an R DataFrame
sparkDF <- as.DataFrame(rdf_clean)
write.df(sparkDF, paste0("Tables/", table_name), source = "delta", mode = "overwrite")
cat(paste0("Spark DataFrame saved to delta table: ", table_name))
```

Summary of observations from the exploratory data analysis

- Most of the customers are from France. Spain has the lowest churn rate, compared to France and Germany.
- Most customers have credit cards
- Some customers are both over the age of 60 and have credit scores below 400. However, they can't be considered as outliers
- Few customers have more than two bank products
- Inactive customers have a higher churn rate
- Gender and tenure years have little impact on a customer's decision to close a bank account

Step 4: Perform model training

With the data in place, you can now define the model. Apply Random Forest and LightGBM models. Use randomForest and LightGBM to implement the models with a few lines of code.

Load the delta table from the lakehouse. You can use other delta tables that consider the lakehouse as the source.

```
R

SEED <- 12345
rdf_clean <- read.df("Tables/rdf_clean", source = "delta")
df_clean <- as.data.frame(rdf_clean)
```

Import randomForest and LightGBM:

```
R

library(randomForest)
library(lightgbm)
```

Prepare the training and test datasets:

```
R
```

```

set.seed(SEED)
y <- factor(df_clean$Exited)
X <- df_clean[, !(colnames(df_clean) %in% c("Exited"))]
split <- base::sample(c(TRUE, FALSE), nrow(df_clean), replace = TRUE, prob = c(0.8, 0.2))
X_train <- X[split,]
X_test <- X[!split,]
y_train <- y[split]
y_test <- y[!split]
train_df <- cbind(X_train, y_train)

```

Apply SMOTE to the training dataset

Imbalanced classification has a problem, because it has too few examples of the minority class for a model to effectively learn the decision boundary. To handle this, Synthetic Minority Oversampling Technique (SMOTE) is the most widely used technique to synthesize new samples for the minority class. Access SMOTE with the `imblearn` library that you installed in step 1.

Apply SMOTE only to the training dataset. You must leave the test dataset in its original imbalanced distribution, to get a valid approximation of model performance on the original data. This experiment represents the situation in production.

First, show the distribution of classes in the dataset, to learn which class is the minority class. The ratio of minority class to majority class is defined as `imbalance Ratio` in the `imbalance` library.

```

R

original_ratio <- imbalance::imbalanceRatio(train_df, classAttr = "y_train")
message(sprintf("Original imbalance ratio is %.2f%% as {Size of minority class}/{Size of majority class}.", original_ratio * 100))
message(sprintf("Positive class(Exited) takes %.2f% of the dataset.", round(sum(train_df$y_train == 1)/nrow(train_df) * 100, 2)))
message(sprintf("Negative class(Non-Exited) takes %.2f% of the dataset.", round(sum(train_df$y_train == 0)/nrow(train_df) * 100, 2)))

```

In the training dataset:

- `Positive class(Exited)` refers to the minority class, which takes 20.34% of the dataset.
- `Negative class(Non-Exited)` refers to the majority class, which takes 79.66% of the dataset.

The next cell rewrites the oversample function of the `imbalance` library, to generate a balanced dataset:

```

R

binary_oversample <- function(train_df, X_train, y_train, class_Attr = "Class"){
  negative_num <- sum(y_train == 0) # Compute the number of the negative class
  positive_num <- sum(y_train == 1) # Compute the number of the positive class
  difference_num <- abs(negative_num - positive_num) # Compute the difference between the negative and positive classes
  originalShape <- imbalance::datasetStructure(train_df, class_Attr) # Get the original dataset schema
  new_samples <- smotefamily::SMOTE(X_train, y_train, dup_size = ceiling(max(negative_num, positive_num)/min(negative_num, positive_num))) # Use SMOTE to oversample
  new_samples <- new_samples$syn_data # Get the synthetic data
  new_samples <- new_samples[base::sample(1:nrow(new_samples), size = difference_num), ] # Sample and shuffle the synthetic data
  new_samples <- new_samples[, -ncol(new_samples)] # Remove the class column
  new_samples <- imbalance::normalizeNewSamples(originalShape, new_samples) # Normalize the synthetic data
  new_train_df <- rbind(train_df, new_samples) # Concatenate original and synthetic data by row
  new_train_df <- new_train_df[base::sample(nrow(new_train_df)), ] # Shuffle the training dataset
  new_train_df
}

```

To learn more about SMOTE, see the [Package `imbalance`](#) and [Working with imbalanced datasets](#) resources on the CRAN website.

Oversample the training dataset

Use the newly defined oversample function to perform oversampling on the training dataset:

```

R

library(dplyr)
new_train_df <- binary_oversample(train_df, X_train, y_train, class_Attr="y_train")
smote_ratio <- imbalance::imbalanceRatio(new_train_df, classAttr = "y_train")
message(sprintf("Imbalance ratio after using smote is %.2f%\n", smote_ratio * 100))

```

Train the model

Use random forest to train the model, with four features:

```
R

set.seed(1)
rfc1_sm <- randomForest(y_train ~ ., data = new_train_df, ntree = 500, mtry = 4, nodesize = 3)
y_pred <- predict(rfc1_sm, X_test, type = "response")
cr_rfc1_sm <- caret::confusionMatrix(y_pred, y_test)
cm_rfc1_sm <- table(y_pred, y_test)
roc_auc_rfc1_sm <- pROC::auc(pROC::roc(as.numeric(y_test), as.numeric(y_pred)))
print(paste0("The auc is ", roc_auc_rfc1_sm))
```

Use random forest to train the model, with six features:

```
R

rfc2_sm <- randomForest(y_train ~ ., data = new_train_df, ntree = 500, mtry = 6, nodesize = 3)
y_pred <- predict(rfc2_sm, X_test, type = "response")
cr_rfc2_sm <- caret::confusionMatrix(y_pred, y_test)
cm_rfc2_sm <- table(y_pred, y_test)
roc_auc_rfc2_sm <- pROC::auc(pROC::roc(as.numeric(y_test), as.numeric(y_pred)))
print(paste0("The auc is ", roc_auc_rfc2_sm))
```

Train the model with LightGBM:

```
R

set.seed(42)
X_train <- new_train_df[, !(colnames(new_train_df) %in% c("y_train"))]
y_train <- as.numeric(as.character(new_train_df$y_train))
y_test <- as.numeric(as.character(y_test))
lgbm_sm_model <- lgb.train(list(objective = "binary", learning_rate = 0.1, max_delta_step = 2, nrounds = 100, max_depth = 10, eval_metric = "logloss"), lgb.Dataset(as.matrix(X_train), label = as.vector(y_train)), valids = list(test = lgb.Dataset(as.matrix(X_test), label = as.vector(as.numeric(y_test)))))
y_pred <- as.numeric(predict(lgbm_sm_model, as.matrix(X_test)) > 0.5)
accuracy <- mean(y_pred == as.vector(y_test))
cr_lgbm_sm <- caret::confusionMatrix(as.factor(y_pred), as.factor(as.vector(y_test)))
cm_lgbm_sm <- table(y_pred, as.vector(y_test))
roc_auc_lgbm_sm <- pROC::auc(pROC::roc(as.vector(y_test), y_pred))
print(paste0("The auc is ", roc_auc_lgbm_sm))
```

Step 5: Evaluate and save the final machine learning model

Assess the performance of the saved models on the test dataset:

```
R

ypred_rfc1_sm <- predict(rfc1_sm, X_test, type = "response")
ypred_rfc2_sm <- predict(rfc2_sm, X_test, type = "response")
ypred_lgbm1_sm <- as.numeric(predict(lgbm_sm_model, as.matrix(X_test)) > 0.5)
```

Show true/false positives/negatives with a confusion matrix. Develop a script to plot the confusion matrix, to evaluate the classification accuracy:

```
R

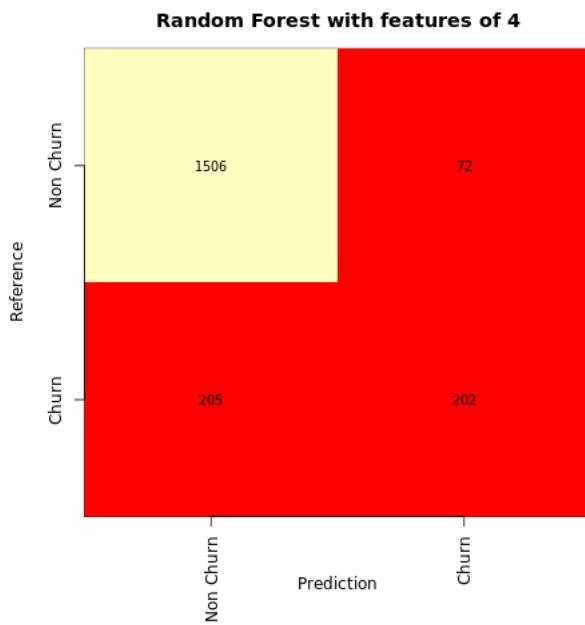
plot_confusion_matrix <- function(cm, classes, normalize=FALSE, title='Confusion matrix', cmap=heat.colors(10)) {
  if (normalize) {
    cm <- cm / rowSums(cm)
  }
  op <- par(mar = c(6,6,3,1))
  image(1:nrow(cm), 1:ncol(cm), t(cm[nrow(cm):1,]), col = cmap, xaxt = 'n', yaxt = 'n', main = title, xlab = "Prediction",
  ylab = "Reference")
  axis(1, at = 1:nrow(cm), labels = classes, las = 2)
  axis(2, at = 1:ncol(cm), labels = rev(classes))
  for (i in seq_len(nrow(cm))) {
    for (j in seq_len(ncol(cm))) {
      text(i, ncol(cm) - j + 1, cm[j,i], cex = 0.8)
    }
  }
}
```

```
    par(op)
}
```

Create a confusion matrix for the random forest classifier, with four features:

```
R
```

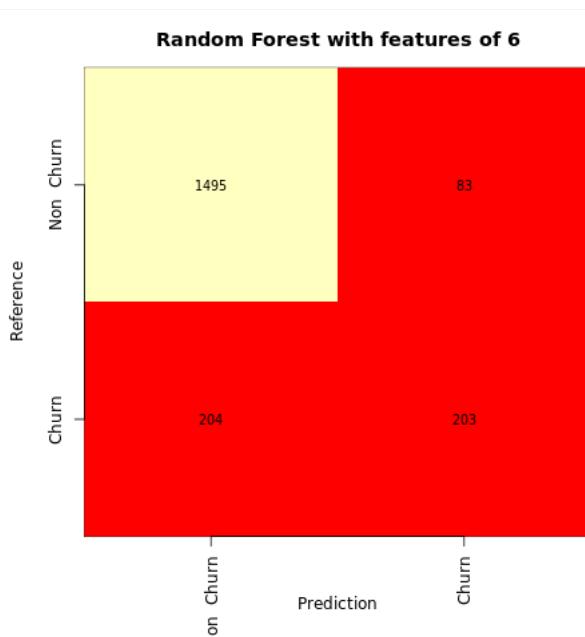
```
cfm <- table(y_test, ypred_rfc1_sm)
plot_confusion_matrix(cfm, classes=c('Non Churn', 'Churn'), title='Random Forest with features of 4')
tn <- cfm[1,1]
fp <- cfm[1,2]
fn <- cfm[2,1]
tp <- cfm[2,2]
```



Create a confusion matrix for the random forest classifier, with six features:

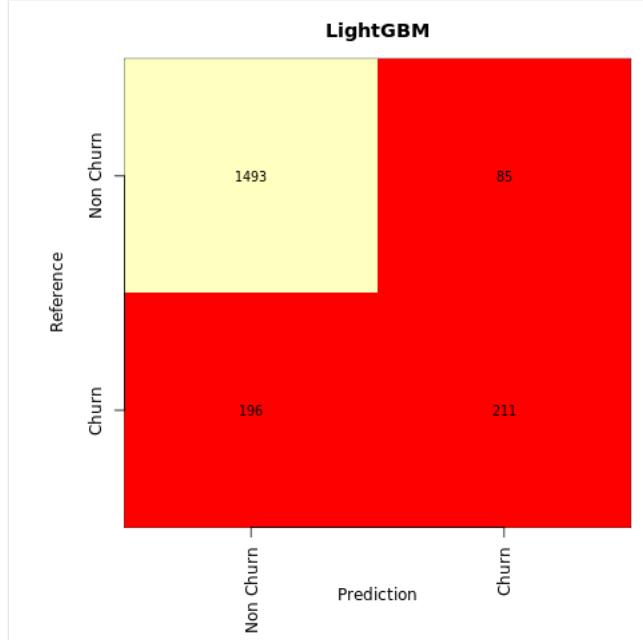
```
R
```

```
cfm <- table(y_test, ypred_rfc2_sm)
plot_confusion_matrix(cfm, classes=c('Non Churn', 'Churn'), title='Random Forest with features of 6')
tn <- cfm[1,1]
fp <- cfm[1,2]
fn <- cfm[2,1]
tp <- cfm[2,2]
```



Create a confusion matrix for LightGBM:

```
R  
  
cfm <- table(y_test, ypred_lgbm1_sm)  
plot_confusion_matrix(cfm, classes=c('Non Churn', 'Churn'), title='LightGBM')  
tn <- cfm[1,1]  
fp <- cfm[1,2]  
fn <- cfm[2,1]  
tp <- cfm[2,2]
```



Save results for Power BI

Save the delta frame to the lakehouse, to move the model prediction results to a Power BI visualization:

```
R  
  
df_pred <- X_test  
df_pred$y_test <- y_test  
df_pred$ypred_rfc1_sm <- ypred_rfc1_sm  
df_pred$ypred_rfc2_sm <- ypred_rfc2_sm  
df_pred$ypred_lgbm1_sm <- ypred_lgbm1_sm  
  
table_name <- "df_pred_results"  
sparkDF <- as.DataFrame(df_pred)  
write.df(sparkDF, paste0("Tables/", table_name), source = "delta", mode = "overwrite", overwriteSchema = "true")  
  
cat(paste0("Spark DataFrame saved to delta table: ", table_name))
```

Step 6: Access visualizations in Power BI

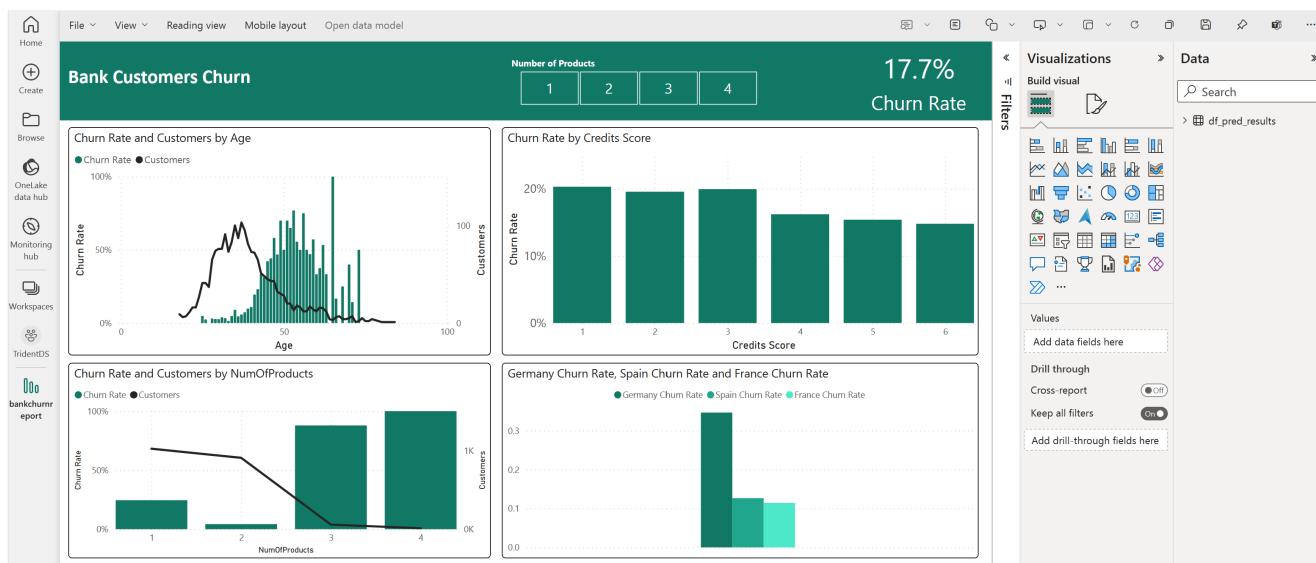
Access your saved table in Power BI:

1. On the left, select **OneLake data hub**
2. Select the lakehouse that you added to this notebook
3. In the **Open this Lakehouse** section, select **Open**
4. On the ribbon, select **New semantic model**. Select `df_pred_results`, and then select **Continue** to create a new Power BI semantic model linked to the predictions
5. On the tools at the top of the dataset page, select **New report** to open the Power BI report authoring page.

The following screenshot shows some example visualizations. The data panel shows the delta tables and columns to select from a table. After selection of appropriate category (x) axis and value (y) axis, you can choose the filters and functions. For example, you can choose a sum or average of the table column.

ⓘ Note

The screenshot is an illustrated example that shows the analysis of the saved prediction results in Power BI. For a real use case of customer churn, platform users might need a more thorough ideation of the visualizations to create, based on both subject matter expertise and what the organization and business analytics team and firm have standardized as metrics.



The Power BI report shows that customers who use more than two of the bank products have a higher churn rate. However, few customers had more than two products. (See the plot in the lower-left panel.) The bank should collect more data but also investigate other features that correlate with more products.

Bank customers in Germany have a higher churn rate compared to customers in France and Spain. (See the plot in the lower-right panel.) Based on the report results, an investigation into the factors that encouraged customers to leave might help.

There are more middle-aged customers (between 25 and 45). Customers between 45 and 60 tend to exit more.

Finally, customers with lower credit scores would most likely leave the bank for other financial institutions. The bank should explore ways to encourage customers with lower credit scores and account balances to stay with the bank.

```
R
# Determine the entire runtime
cat(paste0("Full run cost ", as.integer(Sys.time() - ts), " seconds.\n"))
```

Related content

- [Machine learning model in Microsoft Fabric](#)
- [Train machine learning models](#)
- [Machine learning experiments in Microsoft Fabric](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback](#) | [Ask the community](#)

Tutorial: Use R to create, evaluate, and score a fraud detection model

Article • 04/14/2025

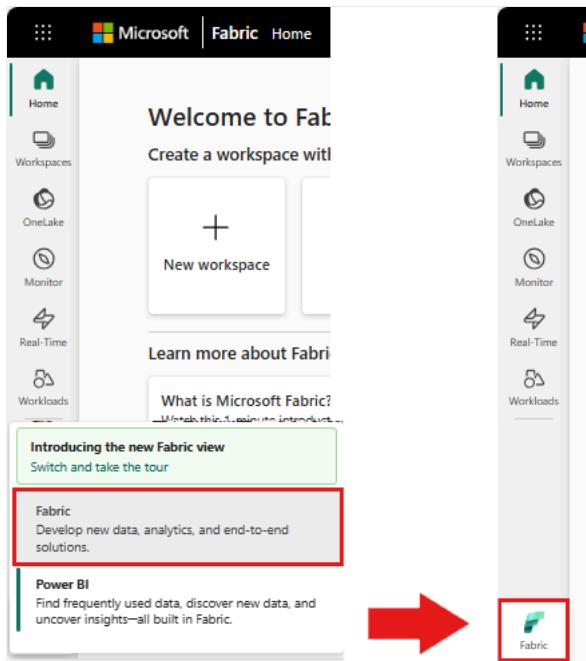
This tutorial presents an end-to-end example of a Synapse Data Science workflow in Microsoft Fabric. This scenario builds an R language fraud detection model, with machine learning algorithms trained on historical data. The scenario then uses the model to detect future fraudulent transactions.

This tutorial covers these steps:

- ✓ Install custom libraries
- ✓ Load the data
- ✓ Understand and process the data with exploratory data analysis, and show the use of the Fabric Data Wrangler feature
- ✓ Train machine learning models with LightGBM
- ✓ Use the machine learning models for scoring and predictions

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- If necessary, create a Microsoft Fabric lakehouse as described in [Create a lakehouse in Microsoft Fabric](#).

Follow along in a notebook

To follow along in a notebook, you have these options:

- Open and run the built-in notebook in the Synapse Data Science experience
- Upload your notebook from GitHub to the Synapse Data Science experience

Open the built-in notebook

The sample Fraud detection notebook accompanies this tutorial.

1. To open the sample notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#).
2. Make sure to [attach a lakehouse to the notebook](#) before you start running code.

Import the notebook from GitHub

The [Alsample - R Fraud Detection.ipynb](#) notebook accompanies this tutorial.

- To open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#) to import the notebook to your workspace.
- If you'd rather copy and paste the code from this page, you can [create a new notebook](#).
- Be sure to [attach a lakehouse to the notebook](#) before you start running code.

Step 1: Install custom libraries

For machine learning model development or ad-hoc data analysis, you might need to quickly install a custom library for your Apache Spark session. You have two options to install libraries.

- Use inline installation resources - for example `install.packages` and `devtools::install_version` - to install in your current notebook only.
- As an alternative, you can create a Fabric environment, and install libraries from public sources or upload custom libraries to it. Then, your workspace admin can attach the environment as the default for the workspace. All the libraries in the environment then become available for use in any notebooks and Spark job definitions in the workspace. For more information about environments, visit [create, configure, and use an environment in Microsoft Fabric](#).

In this tutorial, use `install.version()` to install the imbalanced-learn library:

```
R

# Install dependencies
devtools::install_version("bnlearn", version = "4.8")
# Install imbalance for SMOTE
devtools::install_version("imbalance", version = "1.0.2.1")
```

This installation step might need 8 to 10 minutes to complete.

Step 2: Load the data

The fraud detection dataset contains credit card transactions from September 2013, that European cardholders made over the course of two days. The dataset contains only numerical features, because of a Principal Component Analysis (PCA) transformation applied to the original features. The PCA transformed all features except for `Time` and `Amount`. To protect confidentiality, the original features or more background information about the dataset are not available.

These details describe the dataset:

- The `V1`, `V2`, `V3`, ..., `V28` features are the principal components obtained with PCA
- The `Time` feature contains the elapsed seconds between a transaction and the first transaction in the dataset
- The `Amount` feature is the transaction amount. You can use this feature for example-dependent, cost-sensitive learning
- The `Class` column is the response (target) variable. It has the value `1` for fraud, and `0` otherwise

Only 492 transactions, out of 284,807 transactions total, are fraudulent. The dataset is highly imbalanced, because the minority (fraudulent) class accounts for only about 0.172% of the data.

This table shows a preview of the `creditcard.csv` data:

[Expand table](#)

Time	V1	V2	V3	V4	V5	V6	V7
0	-1.3598071336738	-0.0727811733098497	2.53634673796914	1.37815522427443	-0.338320769942518	0.46238777762292	0.239598554061257
0	1.19185711131486	0.26615071205963	0.16648011335321	0.448154078460911	0.0600176492822243	-0.0823608088155687	-0.0788029833323113

Download the dataset and upload to the lakehouse

Define these parameters, so that you can use this notebook with different datasets:

```
R
```

```

IS_CUSTOM_DATA <- FALSE # If TRUE, the dataset has to be uploaded manually

IS_SAMPLE <- FALSE # If TRUE, use only rows of data for training; otherwise, use all data
SAMPLE_ROWS <- 5000 # If IS_SAMPLE is True, use only this number of rows for training

DATA_ROOT <- "/lakehouse/default"
DATA_FOLDER <- "Files/fraud-detection" # Folder with data files
DATA_FILE <- "creditcard.csv" # Data file name

```

This code downloads a publicly available version of the dataset, and then stores that data in a Fabric lakehouse:

① Important

Be sure to [add a lakehouse](#) to the notebook before you run it. Otherwise, you get an error.

R

```

if (!IS_CUSTOM_DATA) {
    # Download data files into a lakehouse if they don't exist
    library(httr)

    remote_url <- "https://synapseaisolutionsa.blob.core.windows.net/public/Credit_Card_Fraud_Detection"
    fname <- "creditcard.csv"
    download_path <- file.path(DATA_ROOT, DATA_FOLDER, "raw")

    dir.create(download_path, showWarnings = FALSE, recursive = TRUE)
    if (!file.exists(file.path(download_path, fname))) {
        r <- GET(file.path(remote_url, fname), timeout(30))
        writeBin(content(r, "raw"), file.path(download_path, fname))
    }
    message("Downloaded demo data files into lakehouse.")
}

```

Read raw date data from the lakehouse

This code reads raw data from the **Files** section of the lakehouse:

R

```
data_df <- read.csv(file.path(DATA_ROOT, DATA_FOLDER, "raw", DATA_FILE))
```

Step 3: Perform exploratory data analysis

Use the `display` command to view the high-level statistics of the dataset:

R

```
display(as.DataFrame(data_df, numPartitions = 3L))
```

R

```
# Print dataset basic information
message(sprintf("records read: %d", nrow(data_df)))
message("Schema:")
str(data_df)
```

R

```
# If IS_SAMPLE is True, use only SAMPLE_ROWS of rows for training
if (IS_SAMPLE) {
    data_df = sample_n(data_df, SAMPLE_ROWS)
}
```

Print the dataset distribution of the classes:

R

```
# The distribution of classes in the dataset
message(sprintf("No Frauds %.2f%% of the dataset\n", round(sum(data_df$Class == 0)/nrow(data_df) * 100, 2)))
message(sprintf("Frauds %.2f%% of the dataset\n", round(sum(data_df$Class == 1)/nrow(data_df) * 100, 2)))
```

This class distribution shows that most of the transactions are nonfraudulent. Therefore, data preprocessing is required before model training, to avoid overfitting.

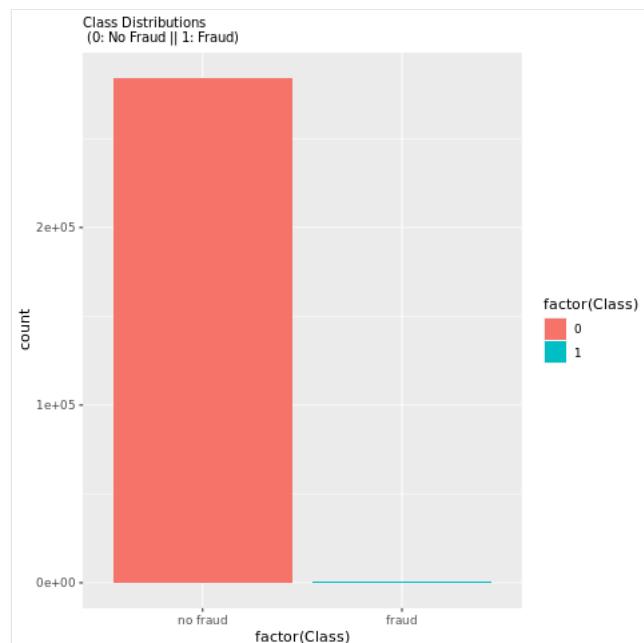
View the distribution of fraudulent versus nonfraudulent transactions

To show the class imbalance in the dataset, view the distribution of fraudulent versus nonfraudulent transactions with a plot:

```
R

library(ggplot2)

ggplot(data_df, aes(x = factor(Class), fill = factor(Class))) +
  geom_bar(stat = "count") +
  scale_x_discrete(labels = c("no fraud", "fraud")) +
  ggtitle("Class Distributions \n (0: No Fraud || 1: Fraud)") +
  theme(plot.title = element_text(size = 10))
```



The plot clearly shows the dataset imbalance.

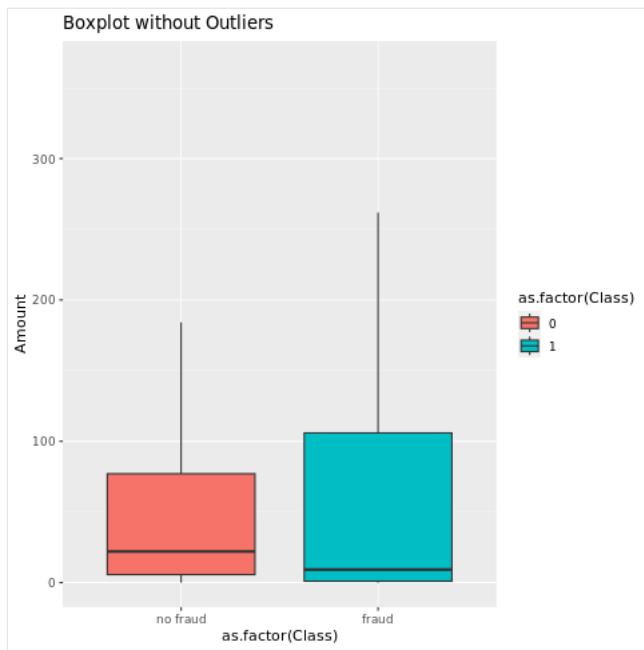
Show the five-number summary

Show the five-number summary (minimum score, first quartile, median, third quartile, and maximum score) for the transaction amount, with box plots:

```
R

library(ggplot2)
library(dplyr)

ggplot(data_df, aes(x = as.factor(Class), y = Amount, fill = as.factor(Class))) +
  geom_boxplot(outlier.shape = NA) +
  scale_x_discrete(labels = c("no fraud", "fraud")) +
  ggtitle("Boxplot without Outliers") +
  coord_cartesian(ylim = quantile(data_df$Amount, c(0.05, 0.95)))
```



For highly imbalanced data, box plots might not show accurate insights. However, you can address the `Class` imbalance problem first, and then create the same plots for more accurate insights.

Step 4: Train and evaluate the models

Train a LightGBM model to classify the fraud transactions. Here, you train a LightGBM model on both the imbalanced dataset and the balanced dataset. Then, you compare the performance of both models.

Prepare training and test datasets

Before training, split the data to the training and test datasets:

```
R

# Split the dataset into training and test datasets
set.seed(42)
train_sample_ids <- base::sample(seq_len(nrow(data_df)), size = floor(0.85 * nrow(data_df)))

train_df <- data_df[train_sample_ids, ]
test_df <- data_df[-train_sample_ids, ]
```

Apply SMOTE to the training dataset

Imbalanced classification has a problem. It has too few minority class examples for a model to effectively learn the decision boundary. The Synthetic Minority Oversampling Technique (SMOTE) can handle this problem. SMOTE is the most widely used approach to synthesize new samples for the minority class. You can access SMOTE with the `imbalance` library that you installed in Step 1.

Apply SMOTE only to the training dataset, instead of the test dataset. When you score the model with the test data, you need an approximation of the model performance on unseen data in production. For a valid approximation, your test data relies on the original imbalanced distribution to represent production data as closely as possible.

```
R

# Apply SMOTE to the training dataset
library(imbalance)

# Print the shape of the original (imbalanced) training dataset
train_y_categ <- train_df %>% select(Class) %>% table
message(
  paste0(
    "Original dataset shape ",
    paste(names(train_y_categ), train_y_categ, sep = ": ", collapse = ", ")
  )
)
```

```

# Resample the training dataset by using SMOTE
smote_train_df <- train_df %>%
  mutate(Class = factor(Class)) %>%
  oversample(ratio = 0.99, method = "SMOTE", classAttr = "Class") %>%
  mutate(Class = as.integer(as.character(Class)))

# Print the shape of the resampled (balanced) training dataset
smote_train_y_categ <- smote_train_df %>% select(Class) %>% table
message(
  paste0(
    "Resampled dataset shape ",
    paste(names(smote_train_y_categ), smote_train_y_categ, sep = ":", collapse = ", ")
  )
)

```

For more information about SMOTE, visit the [Package 'imbalance'](#) and [Working with imbalanced datasets](#) resources at the CRAN website.

Train the model with LightGBM

Train the LightGBM model with both the imbalanced dataset and the balanced (via SMOTE) dataset. Then, compare their performance:

```

R

# Train LightGBM for both imbalanced and balanced datasets and define the evaluation metrics
library(lightgbm)

# Get the ID of the label column
label_col <- which(names(train_df) == "Class")

# Convert the test dataset for the model
test_mtx <- as.matrix(test_df)
test_x <- test_mtx[, -label_col]
test_y <- test_mtx[, label_col]

# Set up the parameters for training
params <- list(
  objective = "binary",
  learning_rate = 0.05,
  first_metric_only = TRUE
)

# Train for the imbalanced dataset
message("Start training with imbalanced data:")
train_mtx <- as.matrix(train_df)
train_x <- train_mtx[, -label_col]
train_y <- train_mtx[, label_col]
train_data <- lgb.Dataset(train_x, label = train_y)
valid_data <- lgb.Dataset.create.valid(train_data, test_x, label = test_y)
model <- lgb.train(
  data = train_data,
  params = params,
  eval = list("binary_logloss", "auc"),
  valids = list(valid = valid_data),
  nrounds = 300L
)

# Train for the balanced (via SMOTE) dataset
message("\n\nStart training with balanced data:")
smote_train_mtx <- as.matrix(smote_train_df)
smote_train_x <- smote_train_mtx[, -label_col]
smote_train_y <- smote_train_mtx[, label_col]
smote_train_data <- lgb.Dataset(smote_train_x, label = smote_train_y)
smote_valid_data <- lgb.Dataset.create.valid(smote_train_data, test_x, label = test_y)
smote_model <- lgb.train(
  data = smote_train_data,
  params = params,
  eval = list("binary_logloss", "auc"),
  valids = list(valid = smote_valid_data),
  nrounds = 300L
)

```

Determine feature importance

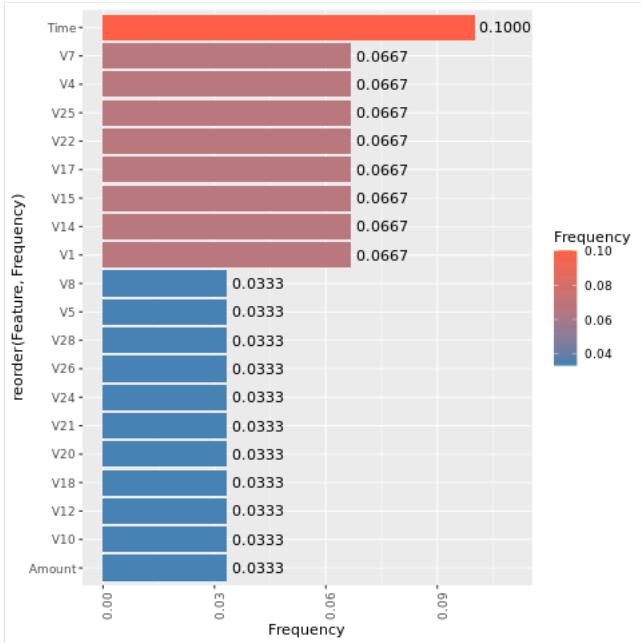
Determine feature importance for the model that you trained on the imbalanced dataset:

```
R
```

```

imp <- lgb.importance(model, percentage = TRUE)
ggplot(imp, aes(x = Frequency, y = reorder(Feature, Frequency), fill = Frequency)) +
  scale_fill_gradient(low="steelblue", high="tomato") +
  geom_bar(stat = "identity") +
  geom_text(aes(label = sprintf("%.4f", Frequency)), hjust = -0.1) +
  theme(axis.text.x = element_text(angle = 90)) +
  xlim(0, max(imp$Frequency) * 1.1)

```



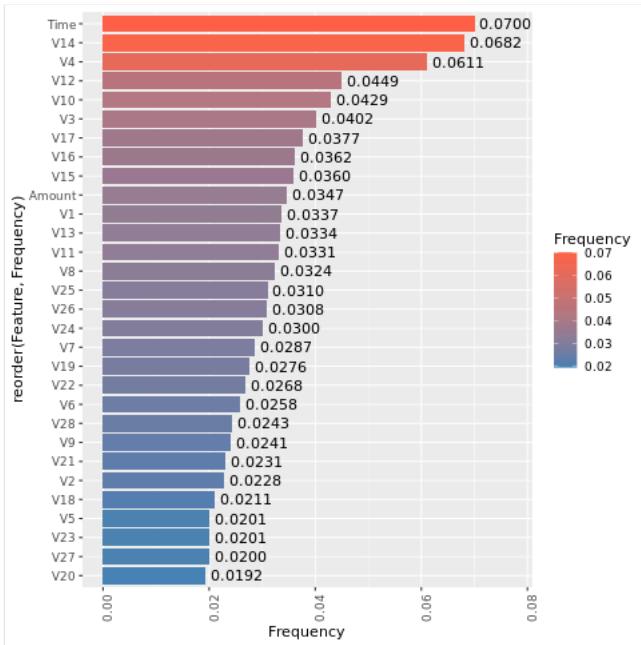
For the model that you trained on the balanced (via SMOTE) dataset, calculate the feature importance:

```

R

smote_imp <- lgb.importance(smote_model, percentage = TRUE)
ggplot(smote_imp, aes(x = Frequency, y = reorder(Feature, Frequency), fill = Frequency)) +
  geom_bar(stat = "identity") +
  scale_fill_gradient(low="steelblue", high="tomato") +
  geom_text(aes(label = sprintf("%.4f", Frequency)), hjust = -0.1) +
  theme(axis.text.x = element_text(angle = 90)) +
  xlim(0, max(smote_imp$Frequency) * 1.1)

```



A comparison of these plots clearly shows that balanced and imbalanced training datasets have large feature importance differences.

Evaluate the models

Evaluate the two trained models:

- `model` trained on raw, imbalanced data
- `smote_model` trained on balanced data

R

```
preds <- predict(model, test_mtx[, -label_col])
smote_preds <- predict(smote_model, test_mtx[, -label_col])
```

Evaluate model performance with a confusion matrix

A *confusion matrix* displays the number of

- true positives (TP)
- true negatives (TN)
- false positives (FP)
- false negatives (FN)

that a model produces when scored with test data. For binary classification, the model returns a `2x2` confusion matrix. For multiclass classification, the model returns an `nxn` confusion matrix, where `n` is the number of classes.

1. Use a confusion matrix to summarize the performance of the trained machine learning models on the test data:

R

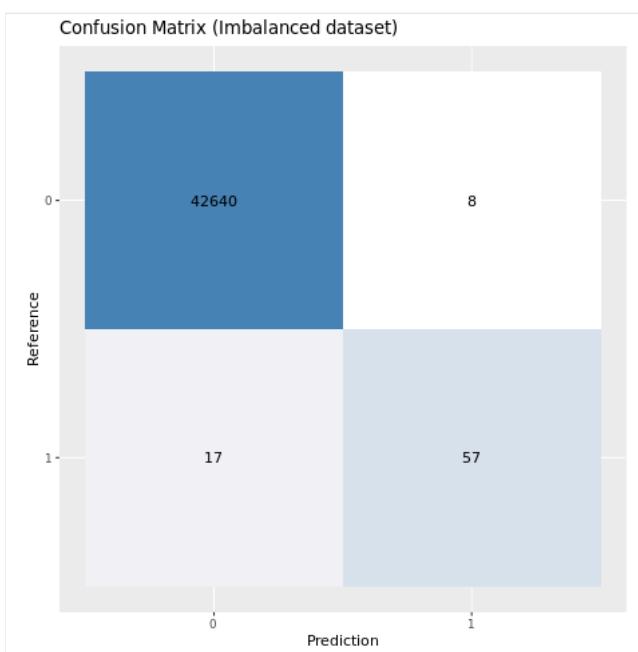
```
plot_cm <- function(preds, refs, title) {
  library(caret)
  cm <- confusionMatrix(factor(refs), factor(preds))
  cm_table <- as.data.frame(cm$table)
  cm_table$Prediction <- factor(cm_table$Prediction, levels=rev(levels(cm_table$Prediction)))

  ggplot(cm_table, aes(Reference, Prediction, fill = Freq)) +
    geom_tile() +
    geom_text(aes(label = Freq)) +
    scale_fill_gradient(low = "white", high = "steelblue", trans = "log") +
    labs(x = "Prediction", y = "Reference", title = title) +
    scale_x_discrete(labels=c("0", "1")) +
    scale_y_discrete(labels=c("1", "0")) +
    coord_equal() +
    theme(legend.position = "none")
}
```

2. Plot the confusion matrix for the model trained on the imbalanced dataset:

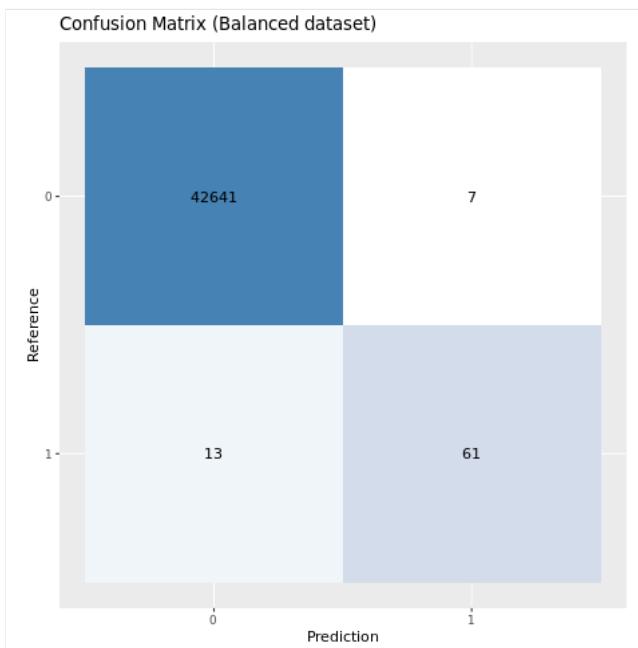
R

```
# The value of the prediction indicates the probability that a transaction is fraud
# Use 0.5 as the threshold for fraud/no-fraud transactions
plot_cm(ifelse(preds > 0.5, 1, 0), test_df$Class, "Confusion Matrix (Imbalanced dataset)")
```



3. Plot the confusion matrix for the model trained on the balanced dataset:

```
R
plot_cm(ifelse(smote_preds > 0.5, 1, 0), test_df$Class, "Confusion Matrix (Balanced dataset)")
```



Evaluate model performance with AUC-ROC and AUPRC measures

The **Area Under the Curve Receiver Operating Characteristic (AUC-ROC)** measure assesses the performance of binary classifiers. The AUC-ROC chart visualizes the trade-off between the true positive rate (TPR) and the false positive rate (FPR).

In some cases, it's more appropriate to evaluate your classifier based on the **Area Under the Precision-Recall Curve (AUPRC)** measure. The AUPRC curve combines these rates:

- The precision, or the positive predictive value (PPV)
- The recall, or TPR

```
R
# Use the PRROC package to help calculate and plot AUC-ROC and AUPRC
install.packages("PRROC", quiet = TRUE)
library(PRROC)
```

Calculate the AUC-ROC and AUPRC metrics

Calculate and plot the AUC-ROC and AUPRC metrics for the two models.

Imbalanced dataset

Calculate the predictions:

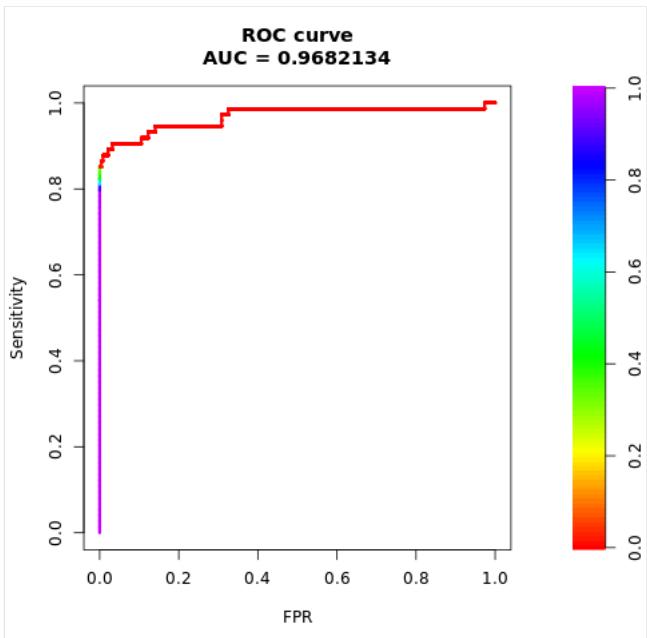
```
R  
  
fg <- preds[test_df$Class == 1]  
bg <- preds[test_df$Class == 0]
```

Print the area under the AUC-ROC curve:

```
R  
  
# Compute AUC-ROC  
roc <- roc.curve(scores.class0 = fg, scores.class1 = bg, curve = TRUE)  
print(roc)
```

Plot the AUC-ROC curve:

```
R  
  
# Plot AUC-ROC  
plot(roc)
```

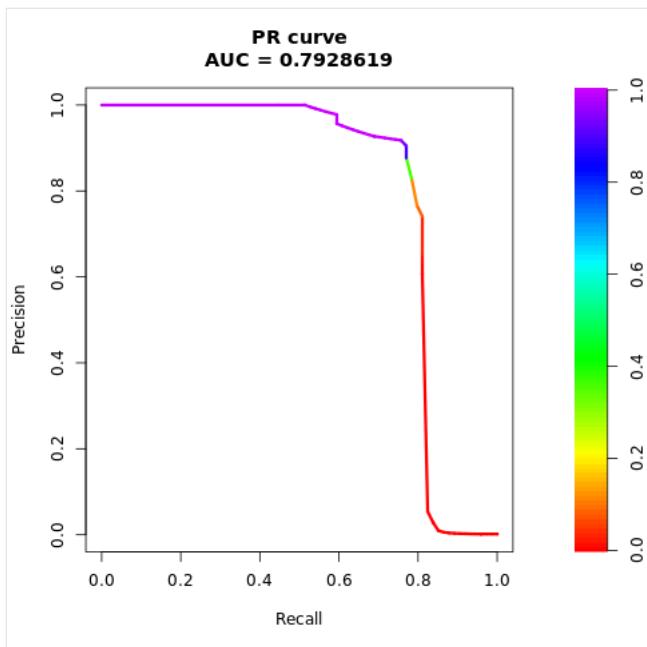


Print the AUPRC curve:

```
R  
  
# Compute AUPRC  
pr <- pr.curve(scores.class0 = fg, scores.class1 = bg, curve = TRUE)  
print(pr)
```

Plot the AUPRC curve:

```
R  
  
# Plot AUPRC  
plot(pr)
```



Balanced (via SMOTE) dataset

Calculate the predictions:

```
R

smote_fg <- smote_preds[test_df$Class == 1]
smote_bg <- smote_preds[test_df$Class == 0]
```

Print the AUC-ROC curve:

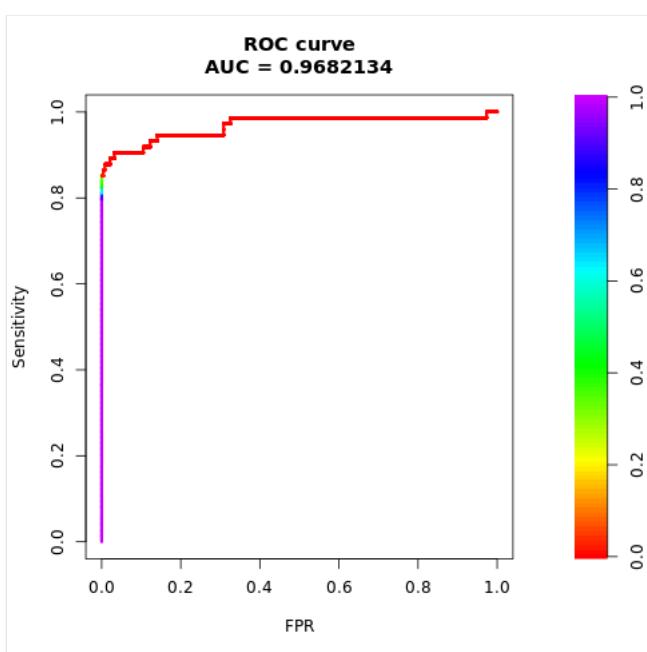
```
R

# Compute AUC-ROC
smote_roc <- roc.curve(scores.class0 = smote_fg, scores.class1 = smote_bg, curve = TRUE)
print(smote_roc)
```

Plot the AUC-ROC curve:

```
R

# Plot AUC-ROC
plot(smote_roc)
```

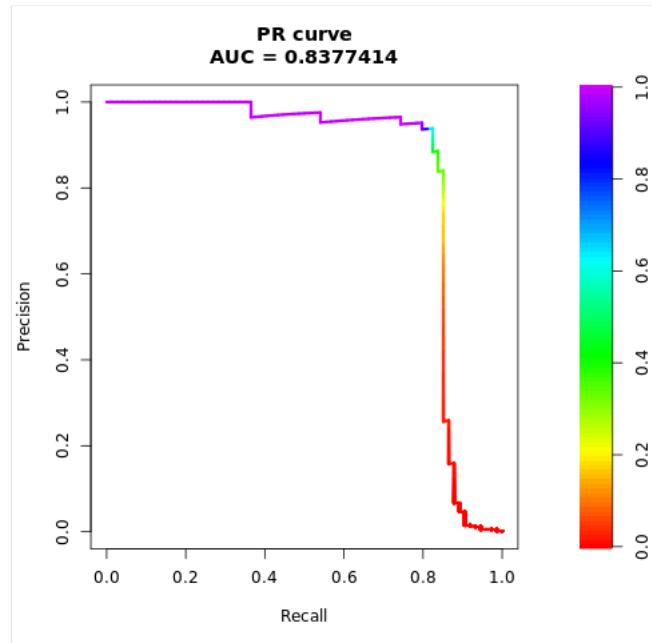


Print the AUPRC curve:

```
R  
  
# Compute AUPRC  
smote_pr <- pr.curve(scores.class0 = smote_fg, scores.class1 = smote_bg, curve = TRUE)  
print(smote_pr)
```

Plot the AUPRC curve:

```
R  
  
# Plot AUPRC  
plot(smote_pr)
```



The earlier figures clearly show that the model trained on the balanced dataset outperforms the model trained on the imbalanced dataset, for both AUC-ROC and AUPRC scores. This result suggests that SMOTE effectively improves model performance when working with highly imbalanced data.

Related content

- [Machine learning model in Microsoft Fabric](#)
- [Train machine learning models](#)
- [Machine learning experiments in Microsoft Fabric](#)

Tutorial: use R to predict avocado prices

Article • 11/20/2024

This tutorial presents an end-to-end example of a Synapse Data Science workflow in Microsoft Fabric. It uses R to analyze and visualize avocado prices in the United States, to build a machine learning model that predicts future avocado prices.

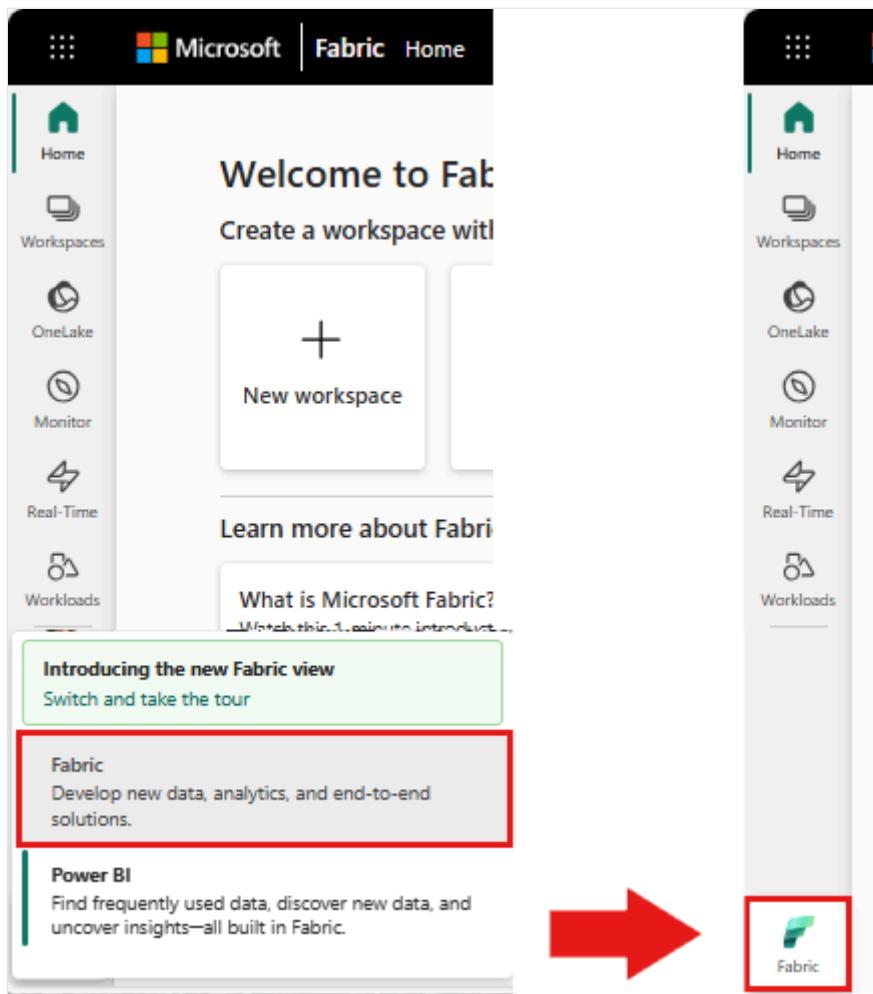
This tutorial covers these steps:

- ✓ Load default libraries
- ✓ Load the data
- ✓ Customize the data
- ✓ Add new packages to the session
- ✓ Analyze and visualize the data
- ✓ Train the model



Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- Open or create a notebook. To learn how, see [How to use Microsoft Fabric notebooks](#).
- Set the language option to **SparkR (R)** to change the primary language.
- Attach your notebook to a lakehouse. On the left side, select **Add** to add an existing lakehouse or to create a lakehouse.

Load libraries

Use libraries from the default R runtime:

```
R

library(tidyverse)
library(lubridate)
library(hms)
```

Load the data

Read avocado prices from a .CSV file, downloaded from the internet:

R

```
df <-  
read.csv('https://synapseaisolutionsa.blob.core.windows.net/public/AvocadoPr  
ice/avocado.csv', header = TRUE)  
head(df,5)
```

Manipulate the data

First, give the columns friendlier names.

R

```
# To use lowercase  
names(df) <- tolower(names(df))  
  
# To use snake case  
avocado <- df %>%  
  rename("av_index" = "x",  
         "average_price" = "averageprice",  
         "total_volume" = "total.volume",  
         "total_bags" = "total.bags",  
         "amount_from_small_bags" = "small.bags",  
         "amount_from_large_bags" = "large.bags",  
         "amount_from_xlarge_bags" = "xlarge.bags")  
  
# Rename codes  
avocado2 <- avocado %>%  
  rename("PLU4046" = "x4046",  
         "PLU4225" = "x4225",  
         "PLU4770" = "x4770")  
  
head(avocado2,5)
```

Change the data types, remove unwanted columns, and add total consumption:

R

```
# Convert data  
avocado2$year = as.factor(avocado2$year)  
avocado2$date = as.Date(avocado2$date)  
avocado2$month = factor(months(avocado2$date), levels = month.name)  
avocado2$average_price = as.numeric(avocado2$average_price)  
avocado2$PLU4046 = as.double(avocado2$PLU4046)  
avocado2$PLU4225 = as.double(avocado2$PLU4225)  
avocado2$PLU4770 = as.double(avocado2$PLU4770)  
avocado2$amount_from_small_bags =  
  as.numeric(avocado2$amount_from_small_bags)  
avocado2$amount_from_large_bags =
```

```
as.numeric(avocado2$amount_from_large_bags)
avocado2$amount_from_xlarge_bags =
as.numeric(avocado2$amount_from_xlarge_bags)

# Remove unwanted columns
avocado2 <- avocado2 %>%
  select(-av_index, -total_volume, -total_bags)

# Calculate total consumption
avocado2 <- avocado2 %>%
  mutate(total_consumption = PLU4046 + PLU4225 + PLU4770 +
amount_from_small_bags + amount_from_large_bags + amount_from_xlarge_bags)
```

Install new packages

Use the inline package installation to add new packages to the session:

```
R
install.packages(c("repr", "gridExtra", "fpp2"))
```

Load the needed libraries.

```
R
library(tidyverse)
library(knitr)
library(repr)
library(gridExtra)
library(data.table)
```

Analyze and visualize the data

Compare conventional (nonorganic) avocado prices by region:

```
R
options(repr.plot.width = 10, repr.plot.height = 10)
# filter(mydata, gear %in% c(4,5))
avocado2 %>%
  filter(region %in%
c("PhoenixTucson", "Houston", "WestTexNewMexico", "DallasFtWorth", "LosAngeles",
"Denver", "Roanoke", "Seattle", "Spokane", "NewYork")) %>%
  filter(type == "conventional") %>%
  select(date, region, average_price) %>%
  ggplot(aes(x = reorder(region, -average_price, na.rm = T), y =
```

```

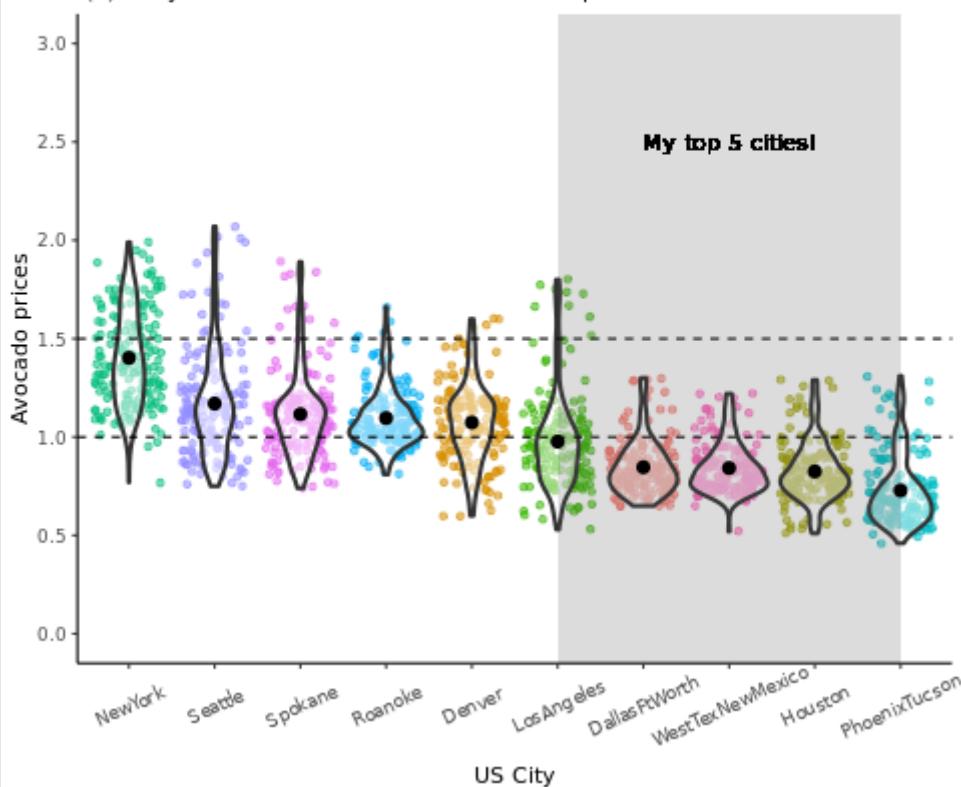
average_price)) +
  geom_jitter(aes(colour = region, alpha = 0.5)) +
  geom_violin(outlier.shape = NA, alpha = 0.5, size = 1) +
  geom_hline(yintercept = 1.5, linetype = 2) +
  geom_hline(yintercept = 1, linetype = 2) +
  annotate("rect", xmin = "LosAngeles", xmax = "PhoenixTucson", ymin = -Inf,
  ymax = Inf, alpha = 0.2) +
  geom_text(x = "WestTexNewMexico", y = 2.5, label = "My top 5 cities!", hjust = 0.5) +
  stat_summary(fun = "mean") +
  labs(x = "US city",
       y = "Avocado prices",
       title = "Figure 1. Violin plot of nonorganic avocado prices",
       subtitle = "Visual aids: \n(1) Black dots are average prices of individual avocados by city \n      between January 2015 and March 2018.\n(2) The plot is ordered descendingly.\n(3) The body of the violin becomes fatter when data points increase.") +
  theme_classic() +
  theme(legend.position = "none",
        axis.text.x = element_text(angle = 25, vjust = 0.65),
        plot.title = element_text(face = "bold", size = 15)) +
  scale_y_continuous(lim = c(0, 3), breaks = seq(0, 3, 0.5))

```

Figure 1. Violin plot of Non-organic Avocado Prices

Visual aids:

- (1) Black dots are average price of individual avocado by city between Jan-2015 to Mar-2018,
- (2) the plot has been ordered descendingly,
- (3) Body of violin become fatter when data points increase.



Focus on the Houston region.

```

library(fpp2)
conv_houston <- avocado2 %>%
  filter(region == "Houston",
         type == "conventional") %>%
  group_by(date) %>%
  summarise(average_price = mean(average_price))

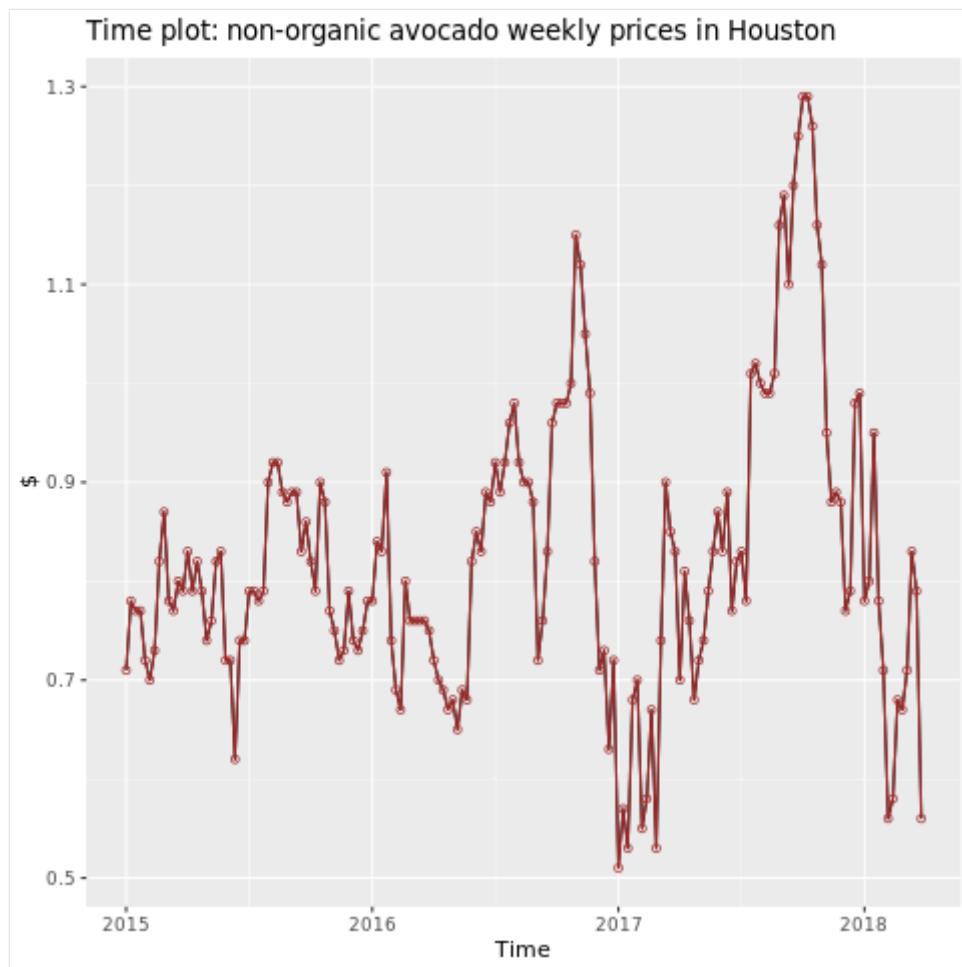
# Set up ts

conv_houston_ts <- ts(conv_houston$average_price,
                        start = c(2015, 1),
                        frequency = 52)

# Plot

autoplot(conv_houston_ts) +
  labs(title = "Time plot: nonorganic avocado weekly prices in Houston",
       y = "$") +
  geom_point(colour = "brown", shape = 21) +
  geom_path(colour = "brown")

```



Train a machine learning model

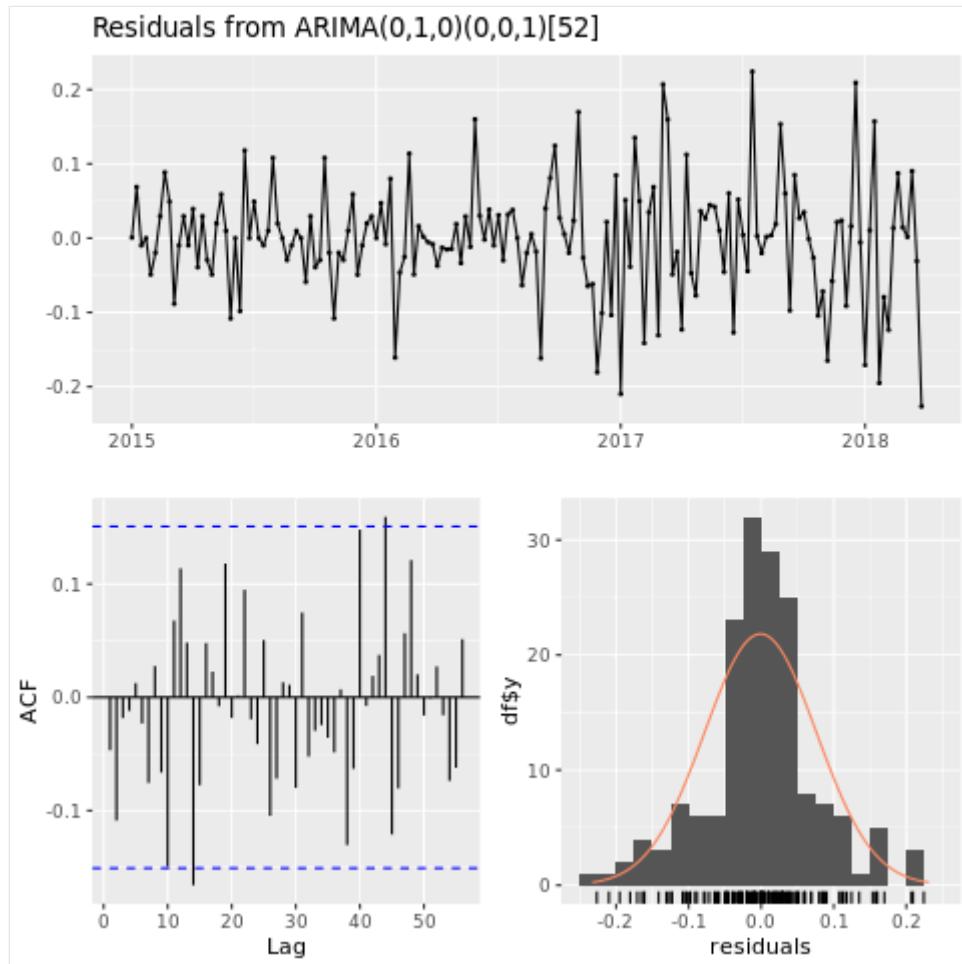
Build a price prediction model for the Houston area, based on AutoRegressive Integrated Moving Average (ARIMA):

R

```
conv_houston_ts_arima <- auto.arima(conv_houston_ts,
                                         d = 1,
                                         approximation = F,
                                         stepwise = F,
                                         trace = T)
```

R

```
checkresiduals(conv_houston_ts_arima)
```

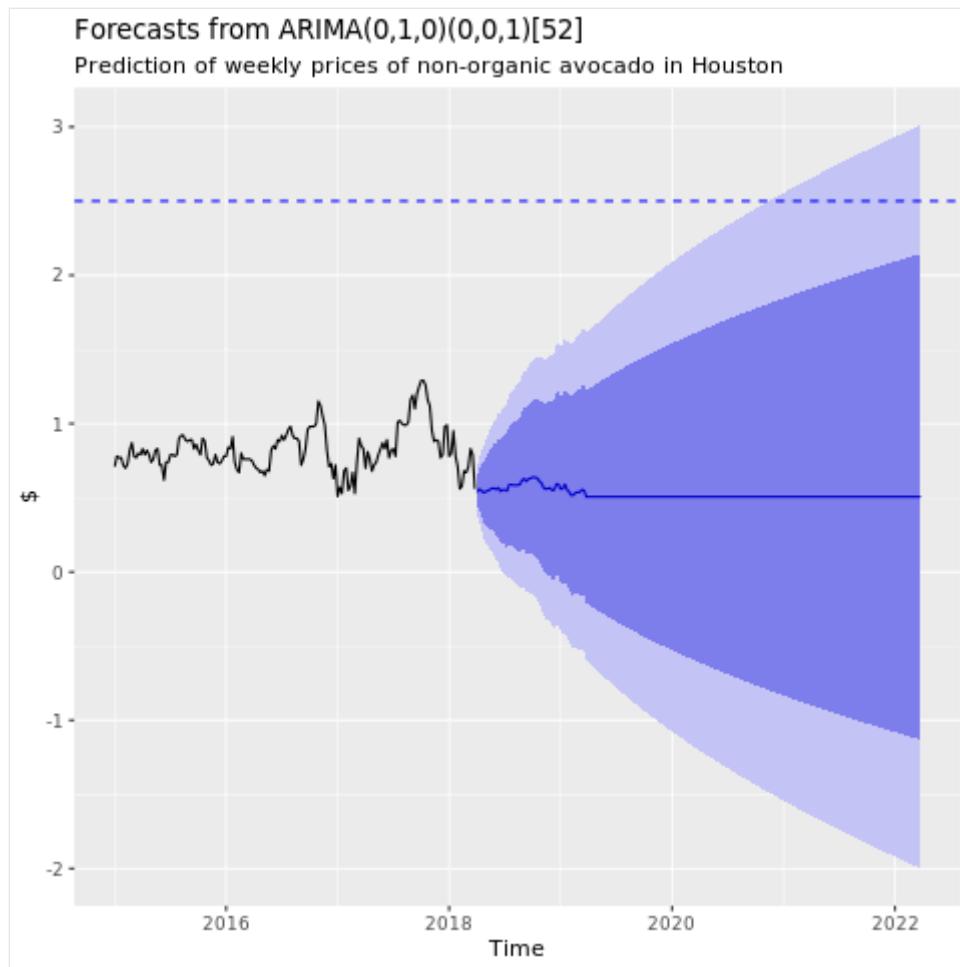


Show a graph of forecasts from the Houston ARIMA model:

R

```
conv_houston_ts_arima_fc <- forecast(conv_houston_ts_arima, h = 208)

autoplot(conv_houston_ts_arima_fc) + labs(subtitle = "Prediction of weekly
prices of nonorganic avocados in Houston",
y = "$") +
geom_hline(yintercept = 2.5, linetype = 2, colour = "blue")
```



Related content

- [How to use SparkR](#)
- [How to use sparklyr](#)
- [How to use Tidyverse](#)
- [R library management](#)
- [Visualize data in R](#)
- [Tutorial: Use R to predict flight delay](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Tutorial: Use R to predict flight delay

Article • 04/10/2025

This tutorial presents an end-to-end example of a Synapse Data Science workflow in Microsoft Fabric. It uses both the [nycflights13](#) data resource, and R, to predict whether or not a plane arrives more than 30 minutes late. It then uses the prediction results to build an interactive Power BI dashboard.

In this tutorial, you learn how to:

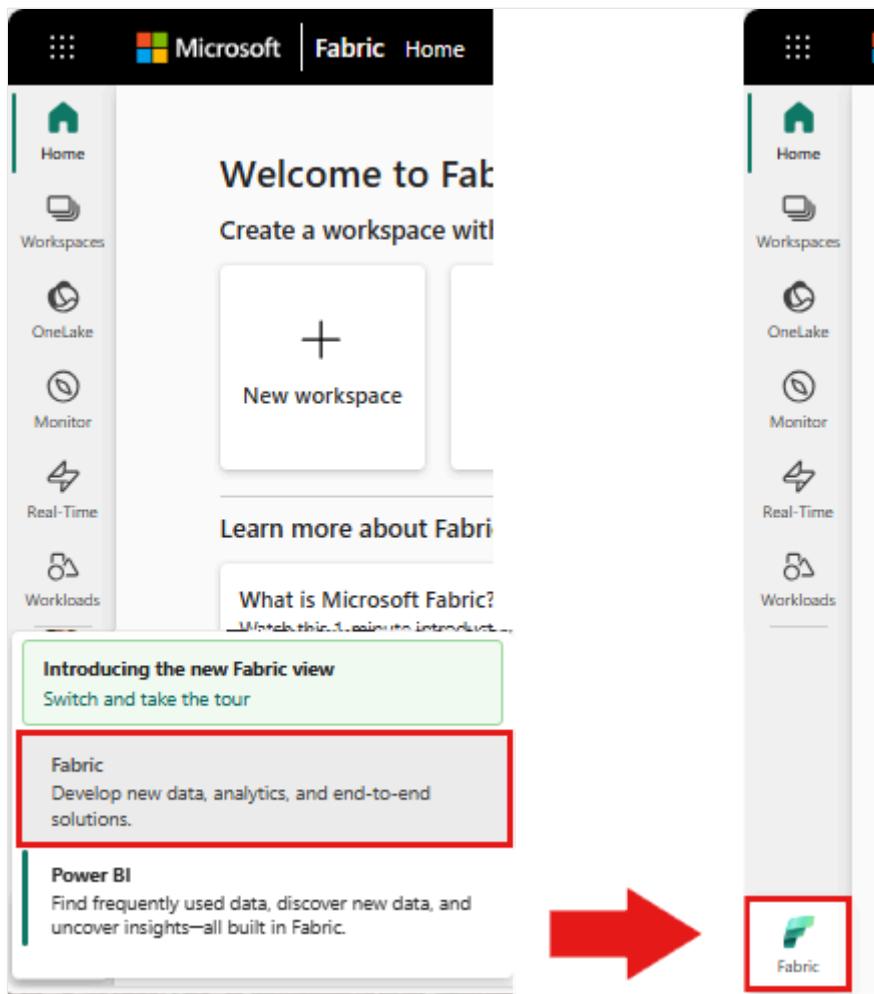
- Use [tidymodels](#) packages
 - [recipes](#)
 - [parsnip](#)
 - [rsample](#)
 - [workflows](#)

to process data and train a machine learning model

- Write the output data to a lakehouse as a delta table
- Build a Power BI visual report to directly access data in that lakehouse

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- Open or create a notebook. To learn how, see [How to use Microsoft Fabric notebooks](#).
- Set the language option to **SparkR (R)** to change the primary language.
- Attach your notebook to a lakehouse. On the left side, select **Add** to add an existing lakehouse or to create a lakehouse.

Install packages

Install the `nycflights13` package to use the code in this tutorial.

```
R
install.packages("nycflights13")
```

```
R
# Load the packages
library(tidymodels)      # For tidymodels packages
library(nycflights13)     # For flight data
```

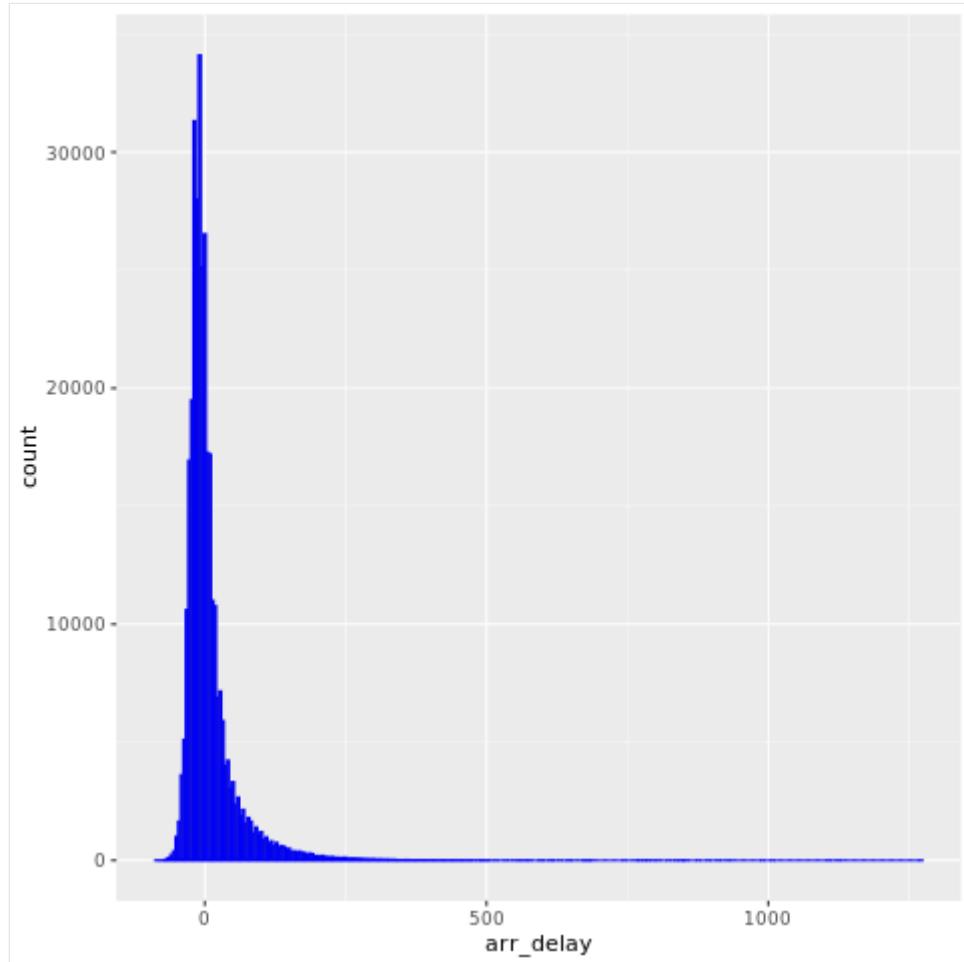
Explore the data

The `nycflights13` data has information about 325,819 flights that arrived near New York City in 2013. First, examine the distribution of flight delays. The following code cell generates a graph showing that the arrival delay distribution is right skewed:

```
R
```

```
ggplot(flights, aes(arr_delay)) + geom_histogram(color="blue", bins = 300)
```

It has a long tail in the high values, as shown in the following image:



Load the data, and make a few changes to the variables:

```
R
```

```
set.seed(123)

flight_data <-
  flights %>%
  mutate(
    # Convert the arrival delay to a factor
    arr_delay = ifelse(arr_delay >= 30, "late", "on_time"),
    arr_delay = factor(arr_delay),
```

```
# You'll use the date (not date-time) for the recipe that you'll create
date = lubridate::as_date(time_hour)
) %>%
# Include weather data
inner_join(weather, by = c("origin", "time_hour")) %>%
# Retain only the specific columns that you'll use
select(dep_time, flight, origin, dest, air_time, distance,
      carrier, date, arr_delay, time_hour) %>%
# Exclude missing data
na.omit() %>%
# For creating models, it's better to have qualitative columns
# encoded as factors (instead of character strings)
mutate_if(is.character, as.factor)
```

Before we build the model, consider a few specific variables that have importance for both preprocessing and modeling.

The `arr_delay` variable is a factor variable. For logistic regression model training, it's important that the outcome variable is a factor variable.

R

```
glimpse(flight_data)
```

About 16% of the flights in this dataset arrived more than 30 minutes late:

R

```
flight_data %>%
  count(arr_delay) %>%
  mutate(prop = n/sum(n))
```

The `dest` feature has 104 flight destinations:

R

```
unique(flight_data$dest)
```

There are 16 distinct carriers:

R

```
unique(flight_data$carrier)
```

Split the data

Split the single dataset into two sets: a *training* set and a *testing* set. Keep most of the rows in the original dataset (as a randomly chosen subset) in the training dataset. Use the training dataset to fit the model, and use the test dataset to measure model performance.

Use the `rsample` package to create an object that contains information about how to split the data. Then, use two more `rsample` functions to create DataFrames for the training and testing sets:

```
R

set.seed(123)
# Keep most of the data in the training set
data_split <- initial_split(flight_data, prop = 0.75)

# Create DataFrames for the two sets:
train_data <- training(data_split)
test_data <- testing(data_split)
```

Create a recipe and roles

Create a recipe for a simple logistic regression model. Before training the model, use a recipe to create new predictors, and conduct the preprocessing that the model requires.

Use the `update_role()` function, with a custom role named `ID`, so that the recipes know that `flight` and `time_hour` are variables. A role can have any character value. The formula includes all variables in the training set as predictors, except for `arr_delay`. The recipe keeps these two ID variables but doesn't use them as either outcomes or predictors:

```
R

flights_rec <-
  recipe(arr_delay ~ ., data = train_data) %>%
  update_role(flight, time_hour, new_role = "ID")
```

To view the current set of variables and roles, use the `summary()` function:

```
R

summary(flights_rec)
```

Create features

Feature engineering can improve your model. The flight date might have a reasonable effect on the likelihood of a late arrival:

```
R  
  
flight_data %>%  
  distinct(date) %>%  
  mutate(numeric_date = as.numeric(date))
```

It might help to add model terms, derived from the date, that have potential importance for the model. Derive the following meaningful features from the single date variable:

- Day of the week
- Month
- Whether or not the date corresponds to a holiday

Add the three steps to your recipe:

```
R  
  
flights_rec <-  
  recipe(arr_delay ~ ., data = train_data) %>%  
  update_role(flight, time_hour, new_role = "ID") %>%  
  step_date(date, features = c("dow", "month")) %>%  
  step_holiday(date,  
    holidays = timeDate::listHolidays("US"),  
    keep_original_cols = FALSE) %>%  
  step_dummy(all_nominal_predictors()) %>%  
  step_zv(all_predictors())
```

Fit a model with a recipe

Use logistic regression to model the flight data. First, build a model specification with the `parsnip` package:

```
R  
  
lr_mod <-  
  logistic_reg() %>%  
  set_engine("glm")
```

Use the `workflows` package to bundle your `parsnip` model (`lr_mod`) with your recipe (`flights_rec`):

```
R
```

```
flights_wflow <-
  workflow() %>%
  add_model(lr_mod) %>%
  add_recipe(flights_rec)

flights_wflow
```

Train the model

This function can prepare the recipe, and train the model from the resulting predictors:

R

```
flights_fit <-
  flights_wflow %>%
  fit(data = train_data)
```

Use the helper functions `xtract_fit_parsnip()` and `extract_recipe()` to extract the model or recipe objects from the workflow. In this example, pull the fitted model object, then use the `broom::tidy()` function to get a tidy [tibble](#) of model coefficients:

R

```
flights_fit %>%
  extract_fit_parsnip() %>%
  tidy()
```

Predict results

A single call to `predict()` uses the trained workflow (`flights_fit`) to make predictions with the unseen test data. The `predict()` method applies the recipe to the new data, then passes the results to the fitted model.

R

```
predict(flights_fit, test_data)
```

Get the output from `predict()` to return the predicted class: `late` versus `on_time`. However, for the predicted class probabilities for each flight, use `augment()` with the model, combined with test data, to save them together:

R

```
flights_aug <-  
  augment(flights_fit, test_data)
```

Review the data:

```
R
```

```
glimpse(flights_aug)
```

Evaluate the model

We now have a tibble with the predicted class probabilities. In the first few rows, the model correctly predicted five on-time flights (values of `.pred_on_time` are `p > 0.50`). However, we need predictions for a total of 81,455 rows.

We need a metric that tells how well the model predicted late arrivals, compared to the true status of the `arr_delay` outcome variable.

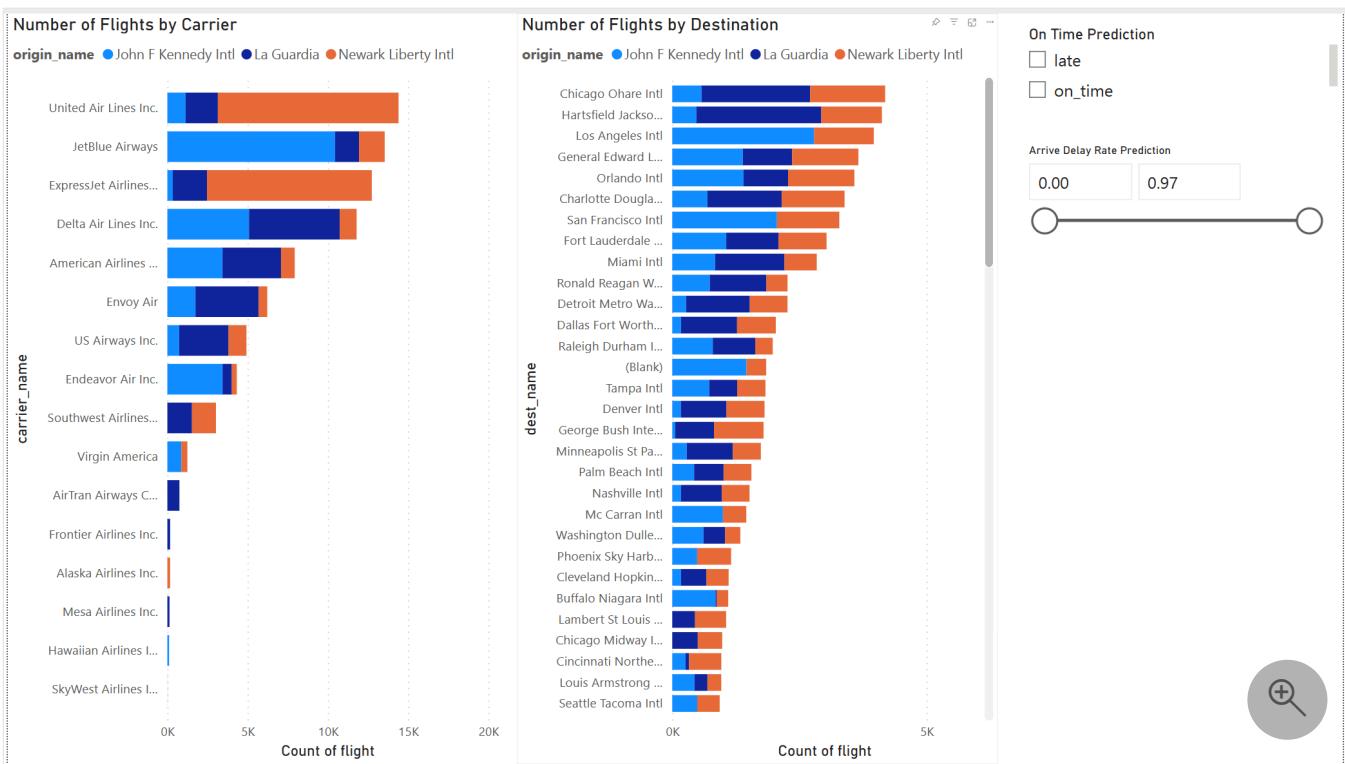
Use the Area Under the Curve Receiver Operating Characteristic (AUC-ROC) as the metric. Compute it with `roc_curve()` and `roc_auc()`, from the `yardstick` package:

```
R
```

```
flights_aug %>%  
  roc_curve(truth = arr_delay, .pred_late) %>%  
  autoplot()
```

Build a Power BI report

The model result looks good. Use the flight delay prediction results to build an interactive Power BI dashboard. The dashboard shows the number of flights by carrier, and the number of flights by destination. The dashboard can filter by the delay prediction results.



Include the carrier name and airport name in the prediction result dataset:

R

```
flights_clean <- flights_aug %>%
# Include the airline data
left_join(airlines, c("carrier"="carrier"))%>%
rename("carrier_name"="name") %>%
# Include the airport data for origin
left_join(airports, c("origin"="faa")) %>%
rename("origin_name"="name") %>%
# Include the airport data for destination
left_join(airports, c("dest"="faa")) %>%
rename("dest_name"="name") %>%
# Retain only the specific columns you'll use
select(flight, origin, origin_name, dest,dest_name, air_time,distance, carrier,
carrier_name, date, arr_delay, time_hour, .pred_class, .pred_late, .pred_on_time)
```

Review the data:

R

```
glimpse(flights_clean)
```

Convert the data to a Spark DataFrame:

R

```
sparkdf <- as.DataFrame(flights_clean)
```

```
display(sparkdf)
```

Write the data into a delta table in your lakehouse:

R

```
# Write data into a delta table
temp_delta<- "Tables/nycflight13"
write.df(sparkdf, temp_delta ,source="delta", mode = "overwrite", header = "true")
```

Use the delta table to create a semantic model.

1. In the left nav, select your workspace, and in the upper right textbox, enter the name of the lakehouse you attached to your notebook. The following screenshot shows that we selected **My Workspace**:

The screenshot shows the Microsoft Fabric workspace interface. On the left, there is a sidebar with icons for Home, Workspaces (with 'OneLake catalog' selected), Monitor, Real-Time, Workloads, and My workspace (which is highlighted with a red box). The main area is titled 'My workspace' and contains a large circular placeholder icon with two overlapping squares. Below it, a message says 'Choose from predesigned task flows or add a task to build one (preview)'. There are buttons for 'Select a predesigned task flow' and 'Add a task'. At the top right, there is a 'Filter by keyword' input field, a 'Workspace settings' button, and a 'Filter' dropdown. A magnifying glass icon is located in the bottom right corner of the main workspace area.

2. Enter the name of the lakehouse that you attached to your notebook. We enter **test_lakehouse1**, as shown in the following screenshot:

The screenshot shows the Microsoft Fabric workspace interface. The top navigation bar includes 'My workspace', 'Home', 'New item', 'New folder', 'Import', 'Migrate', 'Workspace settings', 'Filter', and a search bar containing 'test_lakehouse1'. The left sidebar has links for 'Home', 'Workspaces', 'OneLake catalog', 'Monitor', 'Real-Time', 'Workloads', and 'My workspace'. The main area displays a message 'Choose from predesigned task flows or add a task to build one (preview)' and a table titled 'Filtered results'.

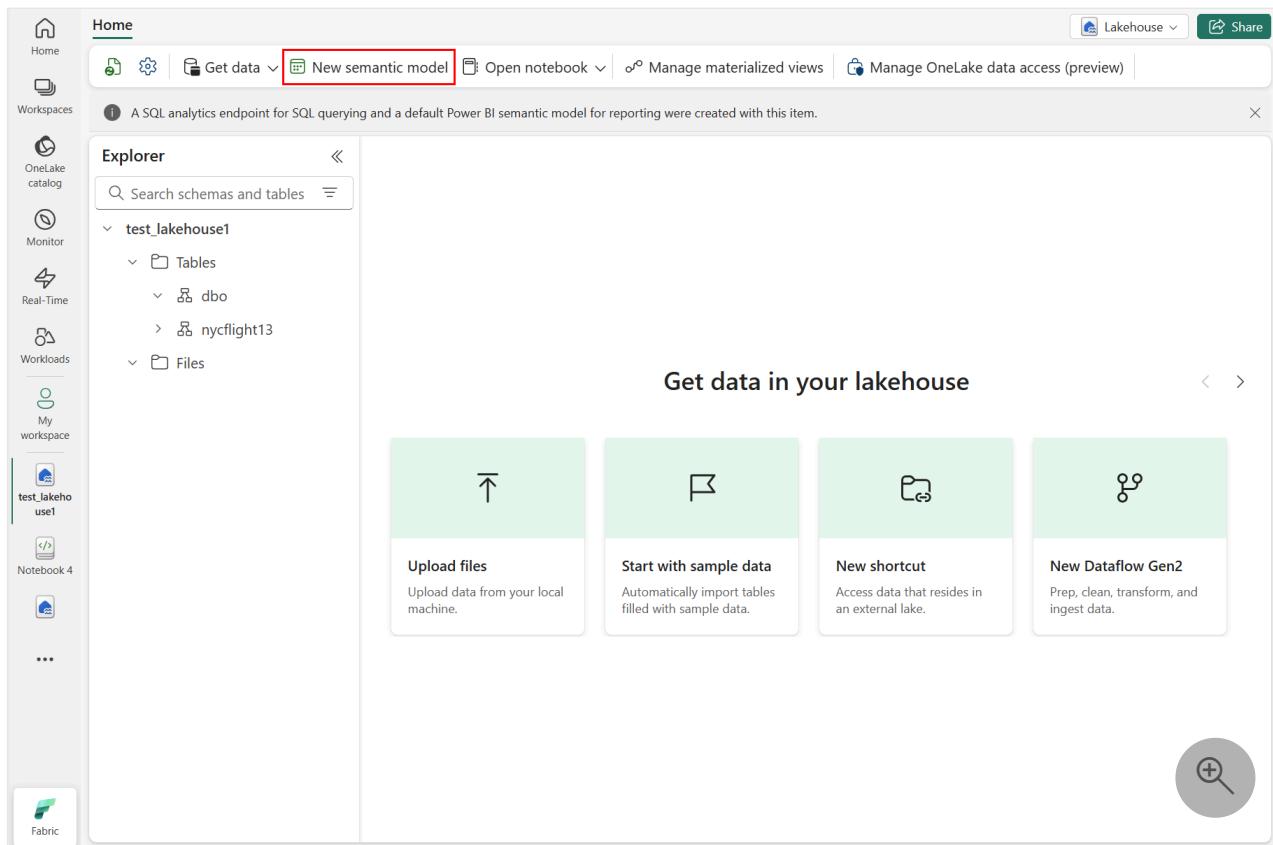
	Name	Location	Type	Task	Owner
...	test_lakehouse1	My workspace	Lakehouse	—	
	test_lakehouse1	My workspace	Semantic model (...)	—	
	test_lakehouse1	My workspace	SQL analytics end...	—	

3. In the filtered results area, select the lakehouse, as shown in the following screenshot:

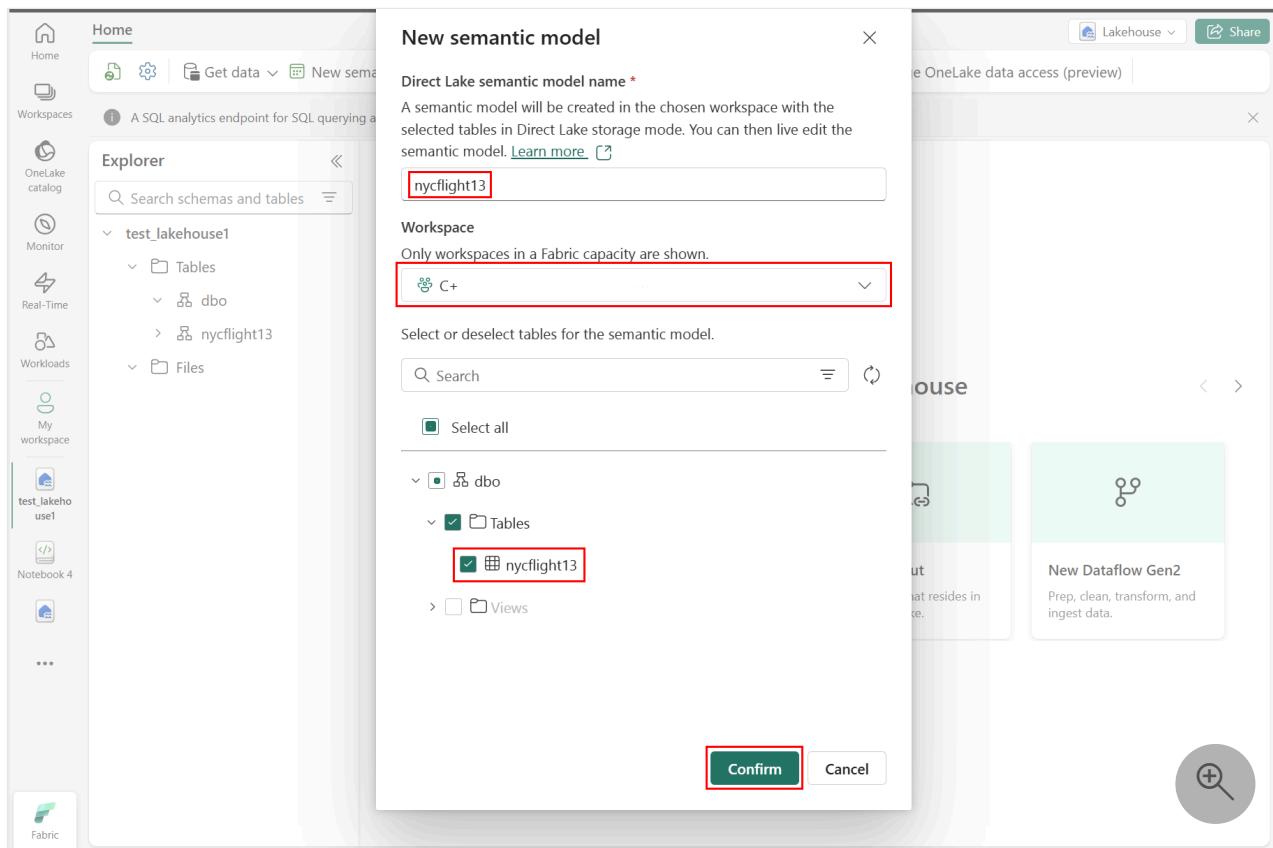
The screenshot shows the Microsoft Fabric workspace interface, identical to the previous one but with a red box highlighting the first row in the 'Filtered results' table, indicating it is selected.

	Name	Location	Type	Task	Owner
...	test_lakehouse1	My workspace	Lakehouse	—	
	test_lakehouse1	My workspace	Semantic model (...)	—	
	test_lakehouse1	My workspace	SQL analytics end...	—	

4. Select **New semantic model** as shown in the following screenshot:



5. At the New semantic model pane, enter a name for the new semantic model, select a workspace, and select the tables to use for that new model, then select **Confirm**, as shown in the following screenshot:



6. To create a new report, select **Create new report**, as shown in the following screenshot:

The screenshot shows the Power BI desktop application. In the top navigation bar, 'File', 'Home', and 'Help' are selected. The 'Create new report' button is highlighted with a red box. The 'Data' pane on the right has 'Tables' selected, showing a search bar and a list of tables, with 'nycflight13' being the current selection. The 'Properties' pane on the far right displays settings for the current table, such as 'Cards' and 'Pin related fields to top of card'. The 'Visualizations' pane at the bottom is partially visible.

7. Select or drag fields from the **Data** and **Visualizations** panes onto the report canvas to build your report

The screenshot shows the Power BI report canvas. On the left, the 'Filters' pane contains sections for 'Filters on this page' and 'Filters on all pages', each with a 'Search' input field and a 'Add data fields here' button. In the center, the 'Visualizations' pane features a 'Build visual' section with a grid of visualization icons (e.g., bar charts, line graphs, pie charts) and a 'Values' section with a 'Add data fields here' button. On the right, the 'Data' pane lists various fields from the 'nycflight13' table, each preceded by a checkbox. Fields include '_pred_class', '_pred_late', '_pred_on_time', 'air_time', 'arr_delay', 'carrier', 'carrier_name', 'date', 'dest', 'dest_name', 'distance', 'flight', 'origin', 'origin_name', and 'time_hour'. A 'Search' input field is also present in the Data pane.

To create the report shown at the beginning of this section, use these visualizations and data:

1.  Stacked bar chart with:
 - a. Y-axis: **carrier_name**
 - b. X-axis: **flight**. Select **Count** for the aggregation
 - c. Legend: **origin_name**
2.  Stacked bar chart with:
 - a. Y-axis: **dest_name**
 - b. X-axis: **flight**. Select **Count** for the aggregation
 - c. Legend: **origin_name**
3.  Slicer with:
 - a. Field: **_pred_class**
4.  Slicer with:
 - a. Field: **_pred_late**

Related content

- [How to use SparkR](#)
- [How to use sparklyr](#)
- [How to use Tidyverse](#)
- [R library management](#)
- [Visualize data in R](#)
- [Tutorial: Use R to predict avocado prices](#)

Lineage for models and experiments

Article • 01/10/2025

In modern business intelligence (BI) projects, understanding the flow of data from the data source to its destination can be a challenge. The challenge is even bigger if you build advanced analytical projects that span multiple data sources, items, and dependencies.

Questions like "What happens if I change this data?" or "Why isn't this report up to date?" can be hard to answer. Such questions might require a team of experts or deep investigation to understand. The Microsoft Fabric lineage view helps you answer these questions.

Lineage and machine learning

There are several reasons why lineage is important in your machine learning workflow:

- **Reproducibility:** Knowing the lineage of a model makes it easier to reproduce the model and its results. If someone else wants to replicate the model, they can follow the same steps that you used to create it, and use the same data and parameters.
- **Transparency:** Understanding the lineage of a model helps to increase its transparency. Stakeholders, such as regulators or users, can understand how the model was created, and how it works. This factor can be important for ensuring fairness, accountability, and ethical considerations.
- **Debugging:** If a model doesn't perform as expected, knowing its lineage can help to identify the source of the problem. By examining the training data, parameters, and decisions that were made during the training process, users might be able to identify issues that affect the model's performance.
- **Improvement:** Knowing the lineage of a model can also help to improve it. By understanding how the model was created and trained, users might be able to make changes to the training data, parameters, or process that can improve the model's accuracy or other performance metrics.

Data science item types

Microsoft Fabric integrates machine learning models and experiments into a unified platform. As part of this approach, users can browse the relationship between Fabric Data Science items and other Fabric items.

Machine learning models

In Fabric, users can create and manage machine learning models. A machine learning model item represents a versioned list of models, which allows users to browse the various iterations of the model.

In the lineage view, users can browse the relationship between a machine learning model and other Fabric items to answer the following questions:

- What is the relationship between machine learning models and experiments in my workspace?
- Which machine learning models exist in my workspace?
- How can I trace back the lineage to see which Lakehouse items were related to this model?

Machine learning experiments

A machine learning *experiment* is the primary unit of organization and control for all related machine learning runs.

In the lineage view, users can browse the relationship between a machine learning experiment and other Fabric items to answer the following questions:

- What is the relationship between machine learning experiments and code items in my workspace? For example, what's the relationship between notebooks and Spark Job Definitions?
- Which machine learning experiments exist in my workspace?
- How can I trace back the lineage to see which Lakehouse items were related to this experiment?

Explore lineage view

Every Fabric workspace has a built-in lineage view. To access this view, you must have at least the **Contributor** role within the workspace. To learn more about permissions in Fabric, see [Data science roles and permissions](#).

To access the lineage view:

1. Select your Fabric workspace and then navigate to the workspace list.

The screenshot shows the 'DS Fraud Detection' workspace in the Synapse Data Science interface. On the left, there's a sidebar with icons for Home, Workspaces, OneLake, Monitor, Real-Time, Workloads, and My workspace (which is selected). The main area displays a table of items:

	Name	Type	Owner	Refreshed
1	DSF2	Lakehouse	DFS1	—
2	DSF2	Semantic model (...)	DS Fraud Detection	4/5/24, 8:23:26 AM
3	DSF2	SQL analytics endpoint	DS Fraud Detection	—
4	DSFLakehouse	Lakehouse	DFS1	—
5	DSFLakehouse	Semantic model (...)	DS Fraud Detection	4/5/24, 8:11:14 AM
6	DSFraudDetectionDemo	Experiment	DFS1	—
7	Ir-experiment-run-8630	Notebook	DFS1	—

At the top right, there are buttons for 'Filter by keyword', 'Filter', and a search bar. A red box highlights the 'Filter' button.

2. Switch from the workspace **List** view to the **Workspace Lineage** view.

The screenshot shows the same 'DS Fraud Detection' workspace, but the interface has switched to the 'Lineage' view. The workspace items are now represented as nodes in a graph, connected by arrows indicating their relationships:

- DSF2 Lakehouse** connects to **Ir-experiment-run-8630 Notebook**.
- DSF2 Lakehouse** connects to **DSF2 SQL analytics endpoint**.
- DSF2 SQL analytics endpoint** connects to **DSF2 Semantic model (default)**.
- DSFLakehouse Lakehouse** connects to **DSFLakehouse SQL analytics endpoint**.
- DSFLakehouse SQL analytics endpoint** connects to **DSFLakehouse Semantic model (default)**.
- DSFraudDetectionDemo Experiment** is shown at the bottom left.

A red box highlights the 'Lineage' icon in the top right corner of the toolbar.

3. You can also navigate to **Lineage** view for a specific item by opening the context menu and selecting to view the workspace or item lineage.

The screenshot shows the Synapse Data Science interface. On the left, there's a sidebar with icons for Home, Workspaces, OneLake, Monitor, Real-Time, Workloads, My workspace, and Fabric. The main area is titled 'DS Fraud Detection'. It lists datasets: DSF2, DSF2 (under DSF2), DSFLakehouse, DSFLakehouse (under DSFLakehouse), DSFraudDetectionDemo, and lr-experiment-run-8630. A context menu is open over the first DSF2 entry, with the 'View workspace lineage' and 'View item lineage' options highlighted by a red box.

Related content

- Learn about machine learning models: [Machine learning models](#)
- Learn about machine learning experiments: [Machine learning experiments](#)

Feedback

Was this page helpful?

[Yes](#)

[No](#)

[Provide product feedback ↗](#) | [Ask the community ↗](#)

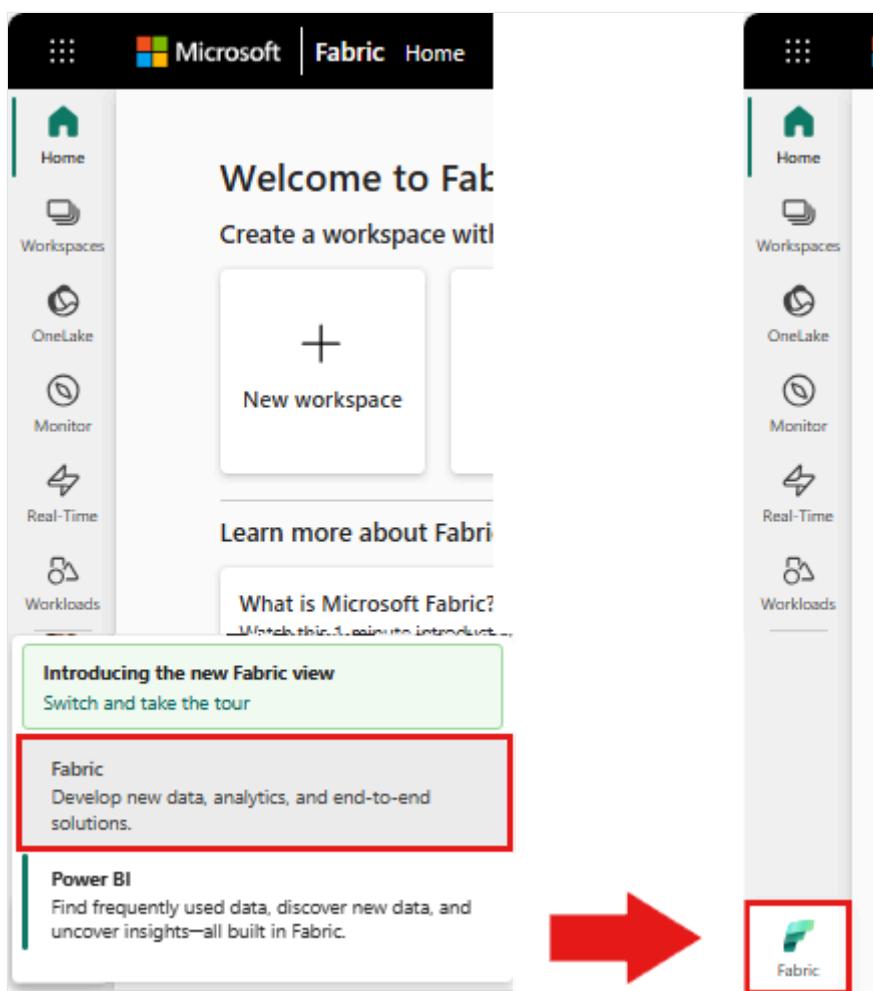
How to read and write data with Pandas in Microsoft Fabric

Article • 11/05/2024

Microsoft Fabric notebooks support seamless interaction with Lakehouse data using Pandas, the most popular Python library for data exploration and processing. Within a notebook, you can quickly read data from, and write data back to, their Lakehouse resources in various file formats. This guide provides code samples to help you get started in your own notebook.

Prerequisites

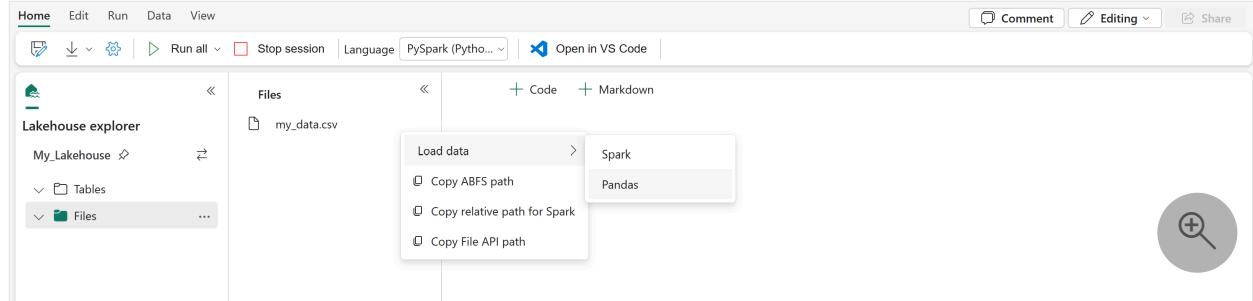
- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



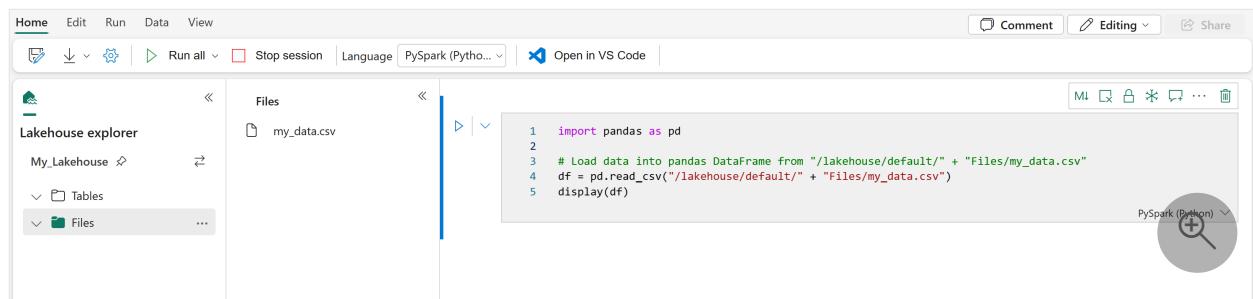
Load Lakehouse data into a notebook

Once you attach a Lakehouse to your Microsoft Fabric notebook, you can explore stored data without leaving the page, and read it into your notebook, all with a few steps.

Selection of any Lakehouse file surfaces options to "Load data" into a Spark or a Pandas DataFrame. You can also copy the file's full ABFS path or a friendly relative path.



Selection of one of the "Load data" prompts generates a code cell to load that file into a DataFrame in your notebook.



Converting a Spark DataFrame into a Pandas DataFrame

For reference, this command shows how to convert a Spark DataFrame into a Pandas DataFrame:

```
Python

# Replace "spark_df" with the name of your own Spark DataFrame
pandas_df = spark_df.toPandas()
```

Reading and writing various file formats

Note

Modifying the version of a specific package could potentially break other packages that depend on it. For instance, downgrading `azure-storage-blob` might cause problems with `Pandas` and various other libraries that rely on `Pandas`, including

`mssparkutils`, `fsspec_wrapper`, and `notebookutils`. You can view the list of preinstalled packages and their versions for each runtime [here](#).

These code samples describe the Pandas operations to read and write various file formats.

ⓘ Note

You must replace the file paths in these code samples. Pandas supports both relative paths, as shown here, and full ABFS paths. Paths of either type can be retrieved and copied from the interface according to the previous step.

Read data from a CSV file

Python

```
import pandas as pd

# Read a CSV file from your Lakehouse into a Pandas DataFrame
# Replace LAKEHOUSE_PATH and FILENAME with your own values
df = pd.read_csv("/LAKEHOUSE_PATH/Files/Filename.csv")
display(df)
```

Write data as a CSV file

Python

```
import pandas as pd

# Write a Pandas DataFrame into a CSV file in your Lakehouse
# Replace LAKEHOUSE_PATH and FILENAME with your own values
df.to_csv("/LAKEHOUSE_PATH/Files/Filename.csv")
```

Read data from a Parquet file

Python

```
import pandas as pd

# Read a Parquet file from your Lakehouse into a Pandas DataFrame
# Replace LAKEHOUSE_PATH and FILENAME with your own values
```

```
df = pandas.read_parquet("/LAKEHOUSE_PATH/Files/Filename.parquet")
display(df)
```

Write data as a Parquet file

Python

```
import pandas as pd

# Write a Pandas DataFrame into a Parquet file in your Lakehouse
# Replace LAKEHOUSE_PATH and FILENAME with your own values
df.to_parquet("/LAKEHOUSE_PATH/Files/Filename.parquet")
```

Read data from an Excel file

Python

```
import pandas as pd

# Read an Excel file from your Lakehouse into a Pandas DataFrame
# Replace LAKEHOUSE_PATH and FILENAME with your own values. Also need to add
correct filepath after Files/ if file is placed in different folders
# if using default lakehouse that attached to the notebook use the code to
replace below: df =
pandas.read_excel("/lakehouse/default/Files/Filename.xlsx")
df = pandas.read_excel("/LAKEHOUSE_PATH/Files/Filename.xlsx")
display(df)
```

Write data as an Excel file

Python

```
import pandas as pd

# Write a Pandas DataFrame into an Excel file in your Lakehouse
# Replace LAKEHOUSE_PATH and FILENAME with your own values
df.to_excel("/LAKEHOUSE_PATH/Files/Filename.xlsx")
```

Read data from a JSON file

Python

```
import pandas as pd
```

```
# Read a JSON file from your Lakehouse into a Pandas DataFrame
# Replace LAKEHOUSE_PATH and FILENAME with your own values
df = pandas.read_json("/LAKEHOUSE_PATH/Files/Filename.json")
display(df)
```

Write data as a JSON file

Python

```
import pandas as pd

# Write a Pandas DataFrame into a JSON file in your Lakehouse
# Replace LAKEHOUSE_PATH and FILENAME with your own values
df.to_json("/LAKEHOUSE_PATH/Files/Filename.json")
```

Related content

- Use Data Wrangler to [clean and prepare your data](#)
- Start [training ML models](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Transform and enrich data seamlessly with AI functions (Preview)

Article • 03/05/2025

ⓘ Important

This feature is in [preview](#).

With Microsoft Fabric, all business professionals—from developers to analysts—can derive more value from their enterprise data through Generative AI, using experiences like [Copilot](#) and [Fabric data agents](#). Thanks to a new set of AI functions for data engineering, Fabric users can now harness the power of industry-leading large language models (LLMs) to transform and enrich data seamlessly.

AI functions harness the power of GenAI for summarization, classification, text generation, and so much more—all with a single line of code:

- **Calculate similarity with `ai.similarity`**: Compare the meaning of input text with a single common text value, or with corresponding text values in another column.
- **Categorize text with `ai.classify`**: Classify input text values according to labels you choose.
- **Detect sentiment with `ai.analyze_sentiment`**: Identify the emotional state expressed by input text.
- **Extract entities with `ai.extract`**: Find and extract specific types of information from input text, for example locations or names.
- **Fix grammar with `ai.fix_grammar`**: Correct the spelling, grammar, and punctuation of input text.
- **Summarize text with `ai.summarize`**: Get summaries of input text.
- **Translate text with `ai.translate`**: Translate input text into another language.
- **Answer custom user prompts with `ai.generate_response`**: Generate responses based on your own instructions.

It's seamless to incorporate these functions as part of data-science and data-engineering workflows, whether you're working with pandas or Spark. There is no detailed configuration, no complex infrastructure management, and no specific technical expertise needed.

Prerequisites

- To use AI functions with Fabric's built-in AI endpoint, your administrator needs to enable [the tenant switch for Copilot and other features powered by Azure OpenAI](#).
- You also need an F64 or higher SKU or a P SKU. With a smaller capacity resource, you need to provide AI functions with your own Azure OpenAI resource [using custom configurations](#).
- Depending on your location, you may need to enable a tenant setting for cross-geo processing. Learn more [here](#).

ⓘ Note

- AI functions are supported in the [Fabric 1.3 runtime](#) and higher.
- By default, AI functions are currently powered by the **gpt-3.5-turbo (0125)** model. To learn more about billing and consumption rates, visit [this article](#).
- Although the underlying model can handle several languages, most of the AI functions are optimized for use on English-language texts.
- During the initial rollout of AI functions, users are temporarily limited to 1,000 requests per minute with Fabric's built-in AI endpoint.

Getting started with AI functions

Use of the AI functions library in a Fabric notebook currently requires certain custom packages. The following code installs and imports those packages. Afterward, you can use AI functions with pandas or PySpark, depending on your preference.

This code cell installs the AI functions library and its dependencies.

⚠ Warning

The PySpark configuration cell takes a few minutes to finish executing. We appreciate your patience.

pandas

```
# Install fixed version of packages
%pip install -q openai==1.30
%pip install -q --force-reinstall httpx==0.27.0

# Install latest version of SynapseML-core
%pip install -q --force-reinstall
https://mmlspark.blob.core.windows.net/pip/1.0.9/synapseml_core-1.0.9-
```

```
py2.py3-none-any.whl
```

```
# Install SynapseML-Internal .whl with AI functions library from blob storage:  
%pip install -q --force-reinstall  
https://mmlspark.blob.core.windows.net/pip/1.0.10.0-spark3.4-6-0cb46b55-SNAPSHOT/synapseml_internal-1.0.10.0.dev1-py2.py3-none-any.whl
```

This code cell imports the AI functions library and its dependencies. The pandas cell also imports an optional Python library to display progress bars that track the status of every AI function call.

```
pandas
```

```
# Required imports  
import synapse.ml.aifunc as aifunc  
import pandas as pd  
import openai  
  
# Optional import for progress bars  
from tqdm.auto import tqdm  
tqdm.pandas()
```

Applying AI functions

Each of the following functions allows you to invoke Fabric's built-in AI endpoint to transform and enrich data with a single line of code. You can use AI functions to analyze pandas DataFrames or Spark DataFrames.

Tip

To learn about customizing the configuration of AI functions, visit [this article](#).

Calculate similarity with `ai.similarity`

The `ai.similarity` function invokes AI to compare input text values with a single common text value, or with pairwise text values in another column. The output similarity scores are relative, and they can range from -1 (opposites) to 1 (identical). A score of 0 indicates that the values are completely unrelated in meaning. For more detailed instructions about the use of `ai.similarity`, visit [this article](#).

Sample usage

```
pandas
```

```
# This code uses AI. Always review output for mistakes.  
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/  
  
df = pd.DataFrame([  
    ("Bill Gates", "Microsoft"),  
    ("Satya Nadella", "Toyota"),  
    ("Joan of Arc", "Nike")  
], columns=["names", "companies"])  
  
df["similarity"] = df["names"].ai.similarity(df["companies"])  
display(df)
```

Categorize text with `ai.classify`

The `ai.classify` function invokes AI to categorize input text according to custom labels you choose. For more information about the use of `ai.classify`, visit [this article](#).

Sample usage

```
pandas
```

```
# This code uses AI. Always review output for mistakes.  
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/  
  
df = pd.DataFrame([  
    "This duvet, lovingly hand-crafted from all-natural fabric, is perfect for a good night's sleep.",  
    "Tired of friends judging your baking? With these handy-dandy measuring cups, you'll create culinary delights.",  
    "Enjoy this *BRAND NEW CAR!* A compact SUV perfect for the professional commuter!"  
], columns=["descriptions"])  
  
df["category"] = df['descriptions'].ai.classify("kitchen", "bedroom", "garage", "other")  
display(df)
```

Detect sentiment with `ai.analyze_sentiment`

The `ai.analyze_sentiment` function invokes AI to identify whether the emotional state expressed by input text is positive, negative, mixed, or neutral. If AI can't make this determination, the output is left blank. For more detailed instructions about the use of `ai.analyze_sentiment`, visit [this article](#).

Sample usage

pandas

```
# This code uses AI. Always review output for mistakes.  
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/  
  
df = pd.DataFrame([  
    "The cleaning spray permanently stained my beautiful kitchen counter. Never again!",  
    "I used this sunscreen on my vacation to Florida, and I didn't get burned at all. Would recommend.",  
    "I'm torn about this speaker system. The sound was high quality, though it didn't connect to my roommate's phone.",  
    "The umbrella is OK, I guess."  
], columns=["reviews"])  
  
df["sentiment"] = df["reviews"].ai.analyze_sentiment()  
display(df)
```

Extract entities with `ai.extract`

The `ai.extract` function invokes AI to scan input text and extract specific types of information designated by labels you choose—for example, locations or names. For more detailed instructions about the use of `ai.extract`, visit [this article](#).

Sample usage

pandas

```
# This code uses AI. Always review output for mistakes.  
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/
```

```
df = pd.DataFrame([
    "MJ Lee lives in Tuscon, AZ, and works as a software engineer
for Microsoft.",
    "Kris Turner, a nurse at NYU Langone, is a resident of Jersey
City, New Jersey."
], columns=["descriptions"])

df_entities = df["descriptions"].ai.extract("name", "profession",
"city")
display(df_entities)
```

Fix grammar with `ai.fix_grammar`

The `ai.fix_grammar` function invokes AI to correct the spelling, grammar, and punctuation of input text. For more detailed instructions about the use of `ai.fix_grammar`, visit [this article](#).

Sample usage

pandas

```
# This code uses AI. Always review output for mistakes.
# Read terms: https://azure.microsoft.com/support/legal/preview-
supplemental-terms/

df = pd.DataFrame([
    "There are an error here.",
    "She and me go weigh back. We used to hang out every weeks.",
    "The big picture are right, but you're details is all wrong."
], columns=["text"])

df["corrections"] = df["text"].ai.fix_grammar()
display(df)
```

Summarize text with `ai.summarize`

The `ai.summarize` function invokes AI to generate summaries of input text (either values from a single column of a DataFrame, or row values across all the columns). For more detailed instructions about the use of `ai.summarize`, visit [this dedicated article](#).

Sample usage

pandas

```
# This code uses AI. Always review output for mistakes.
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/

df= pd.DataFrame([
    ("Microsoft Teams", "2017",
     """
        The ultimate messaging app for your organization—a workspace for
        real-time
        collaboration and communication, meetings, file and app sharing,
        and even the
        occasional emoji! All in one place, all in the open, all
        accessible to everyone.
    """),
    ("Microsoft Fabric", "2023",
     """
        An enterprise-ready, end-to-end analytics platform that unifies
        data movement,
        data processing, ingestion, transformation, and report building
        into a seamless,
        user-friendly SaaS experience. Transform raw data into
        actionable insights.
    """)
], columns=["product", "release_year", "description"])

df[\"summaries\"] = df[\"description\"].ai.summarize()
display(df)
```

Translate text with `ai.translate`

The `ai.translate` function invokes AI to translate input text to a new language of your choice. For more detailed instructions about the use of `ai.translate`, visit [this article](#).

Sample usage

pandas

```
# This code uses AI. Always review output for mistakes.
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/

df = pd.DataFrame([
    "Hello! How are you doing today?",
    "Tell me what you'd like to know, and I'll do my best to help.",
```

```
"The only thing we have to fear is fear itself."  
], columns=["text"])  
  
df["translations"] = df["text"].ai.translate("spanish")  
display(df)
```

Answer custom user prompts with `ai.generate_response`

The `ai.generate_response` function invokes AI to generate custom text based on your own instructions. For more detailed instructions about the use of `ai.generate_response`, visit [this article](#).

Sample usage

pandas

```
# This code uses AI. Always review output for mistakes.  
# Read terms: https://azure.microsoft.com/support/legal/preview-supplemental-terms/  
  
df = pd.DataFrame([  
    ("Scarves"),  
    ("Snow pants"),  
    ("Ski goggles")  
], columns=["product"])  
  
df["response"] = df.ai.generate_response("Write a short, punchy email  
subject line for a winter sale.")  
display(df)
```

Related content

- Calculate similarity with [ai.similarity](#).
- Detect sentiment with [ai.analyze_sentiment](#).
- Categorize text with [ai.classify](#).
- Extract entities with [ai_extract](#).
- Fix grammar with [ai.fix_grammar](#).
- Summarize text with [ai.summarize](#).
- Translate text with [ai.translate](#).
- Answer custom user prompts with [ai.generate_response](#).
- Learn how to [customize the configuration of AI functions](#).

- Did we miss a feature you need? Suggest it on the [Fabric Ideas forum](#).
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

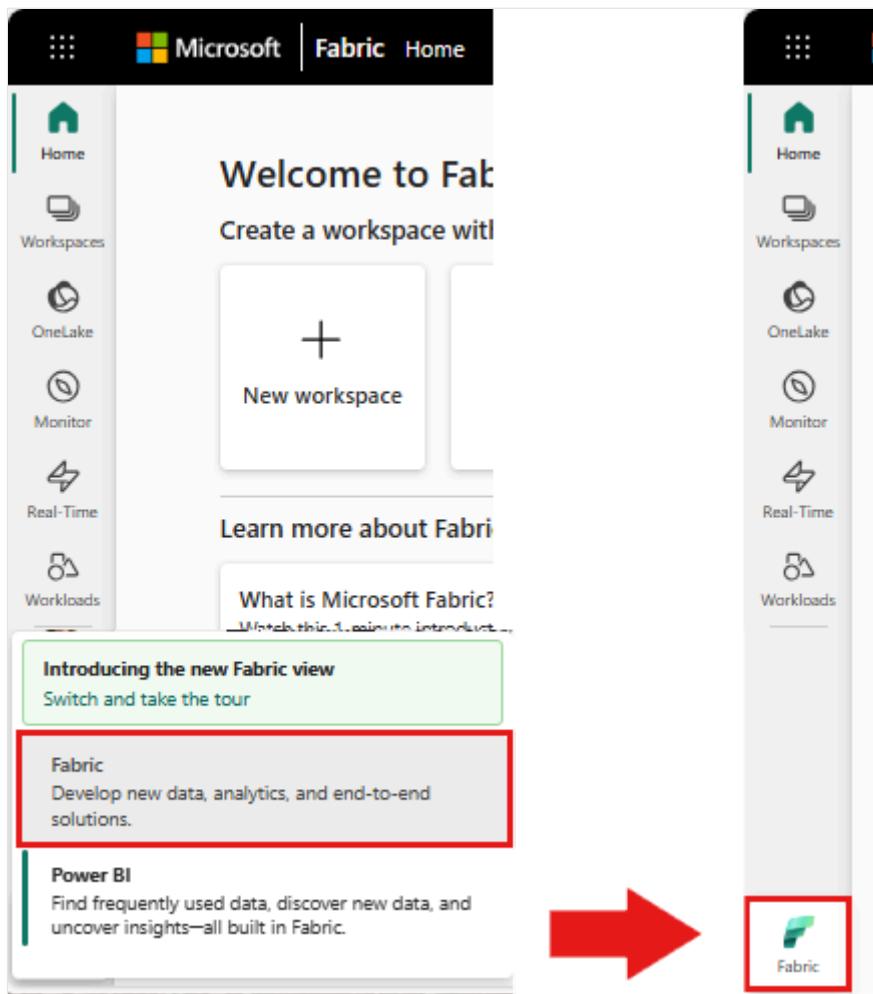
How to accelerate data prep with Data Wrangler in Microsoft Fabric

Article • 03/03/2025

The Data Wrangler tool is a notebook-based resource that provides an immersive interface for exploratory data analysis. It combines a grid-like data display with dynamic summary statistics, built-in visualizations, and a library of common data-cleaning operations. You can apply each operation with a few steps. You can update the data display in real time, and generate code in pandas or PySpark that you can save back to the notebook as a reusable function. This article focuses on exploration and transformation of pandas DataFrames. For more information about using Data Wrangler on Spark DataFrames, visit [this resource](#).

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



Limitations

- Custom code operations are currently supported only for pandas DataFrames.
- The Data Wrangler display works best on large monitors, although you can minimize or hide different portions of the interface, to accommodate smaller screens.

Launching Data Wrangler

You can launch Data Wrangler directly from a Microsoft Fabric notebook to explore and transform any pandas or Spark DataFrame. For more information about using Data Wrangler with Spark DataFrames, visit [this companion article](#). This code snippet shows how to read sample data into a pandas DataFrame:

Python

```
import pandas as pd

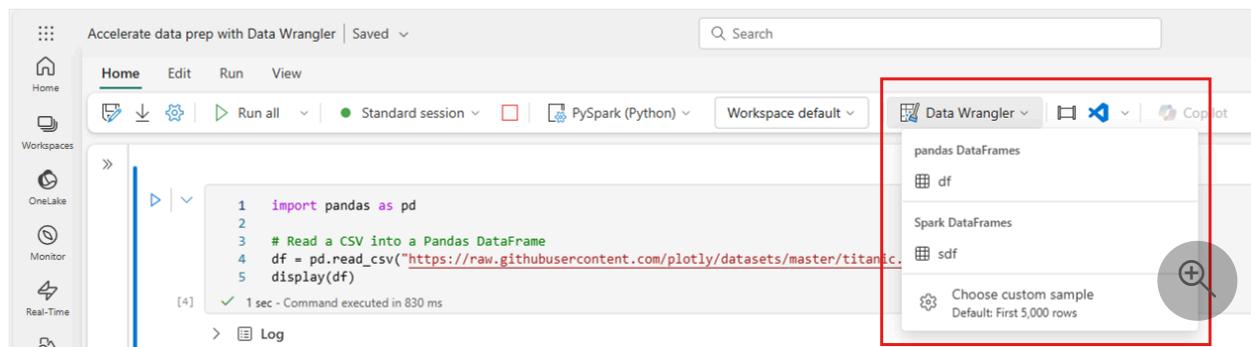
# Read a CSV into a Pandas DataFrame
df =
pd.read_csv("https://raw.githubusercontent.com/plotly/datasets/master/titanic")
```

```
c.csv")
display(df)
```

In the notebook ribbon "Home" tab, use the Data Wrangler dropdown prompt to browse the active DataFrames available for editing. Select the one you want to open in Data Wrangler.

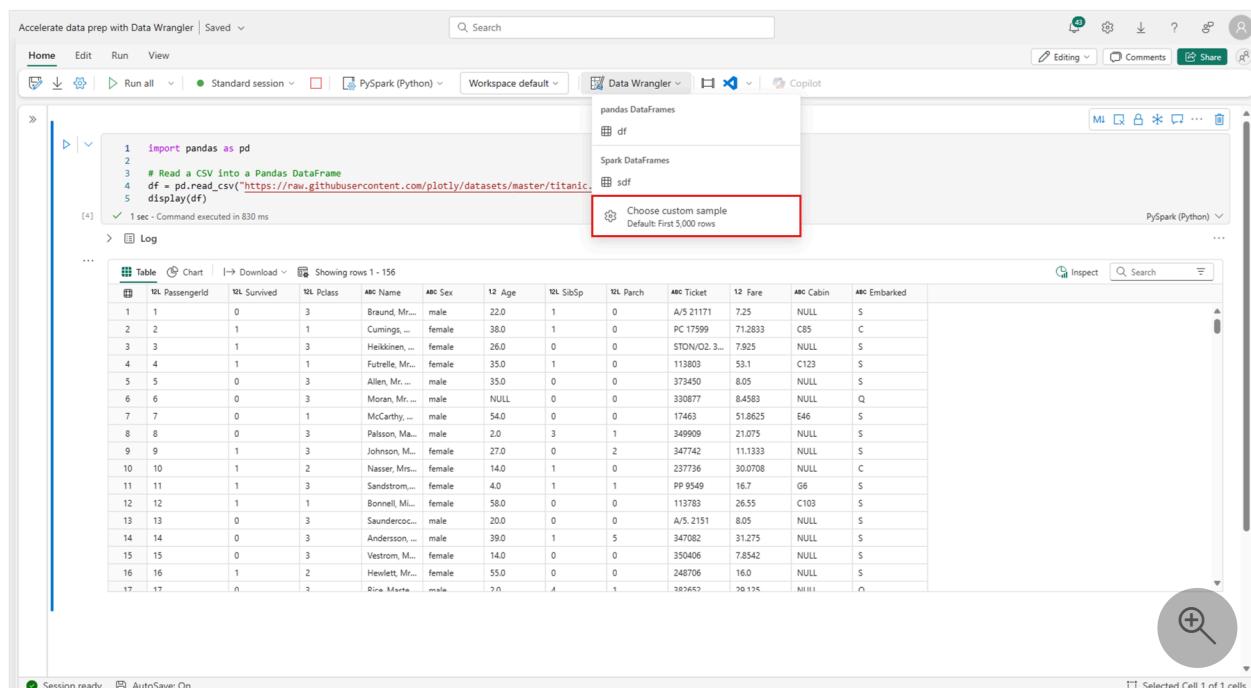
Tip

Data Wrangler cannot be opened while the notebook kernel is busy. An executing cell must finish its execution before Data Wrangler can launch, as shown in this screenshot:

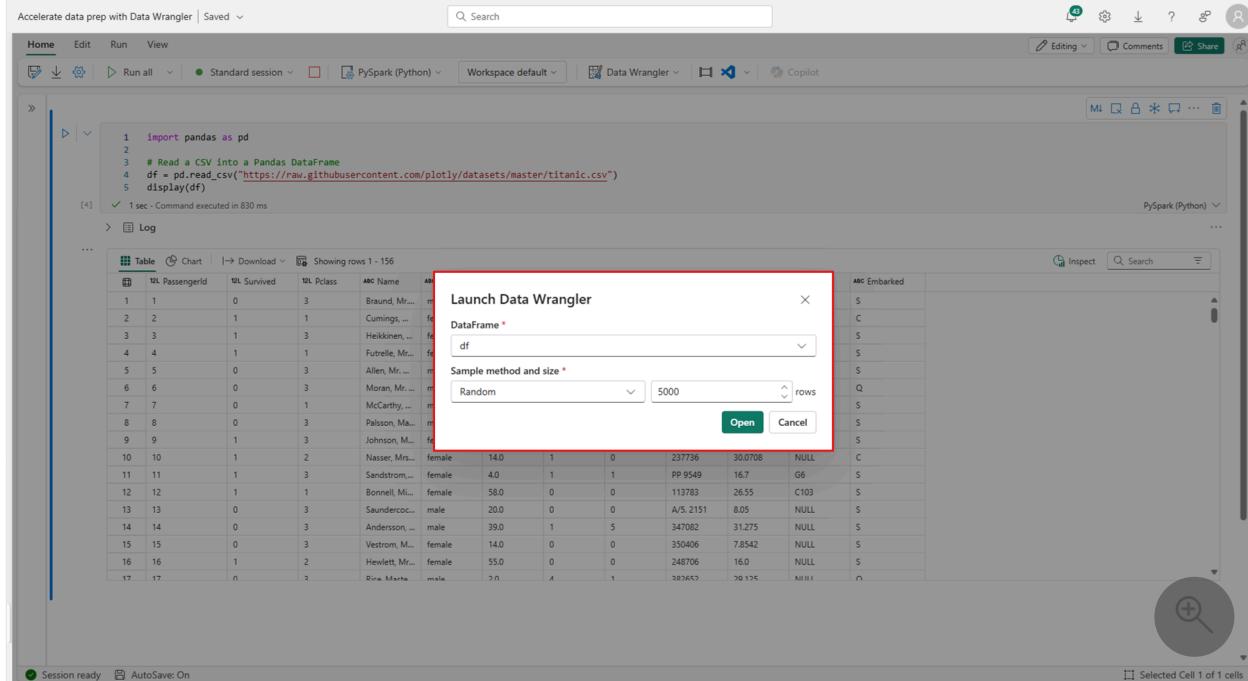


Choosing custom samples

To open a custom sample of any active DataFrame with Data Wrangler, select "Choose custom sample" from the dropdown, as shown in this screenshot:



This launches a pop-up with options to specify the size of the desired sample (number of rows) and the sampling method (first records, last records, or a random set). The first 5,000 rows of the DataFrame serve as the default sample size, as shown in this screenshot:

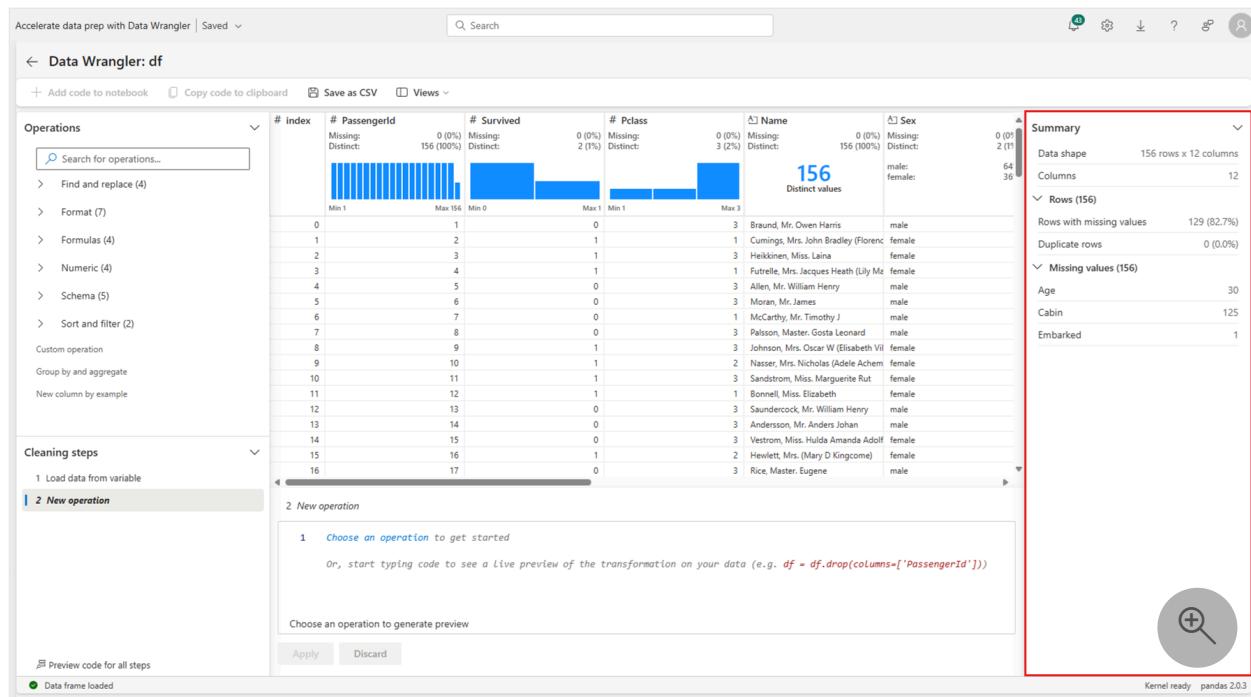


Viewing summary statistics

When Data Wrangler loads, it displays a descriptive overview of the chosen DataFrame in the "Summary" panel. This overview includes information about the DataFrame dimensions, its missing values, and more. Selection of any column in the Data Wrangler grid prompts the "Summary" panel to update and display descriptive statistics about that specific column. Quick insights about every column are also available in its header.

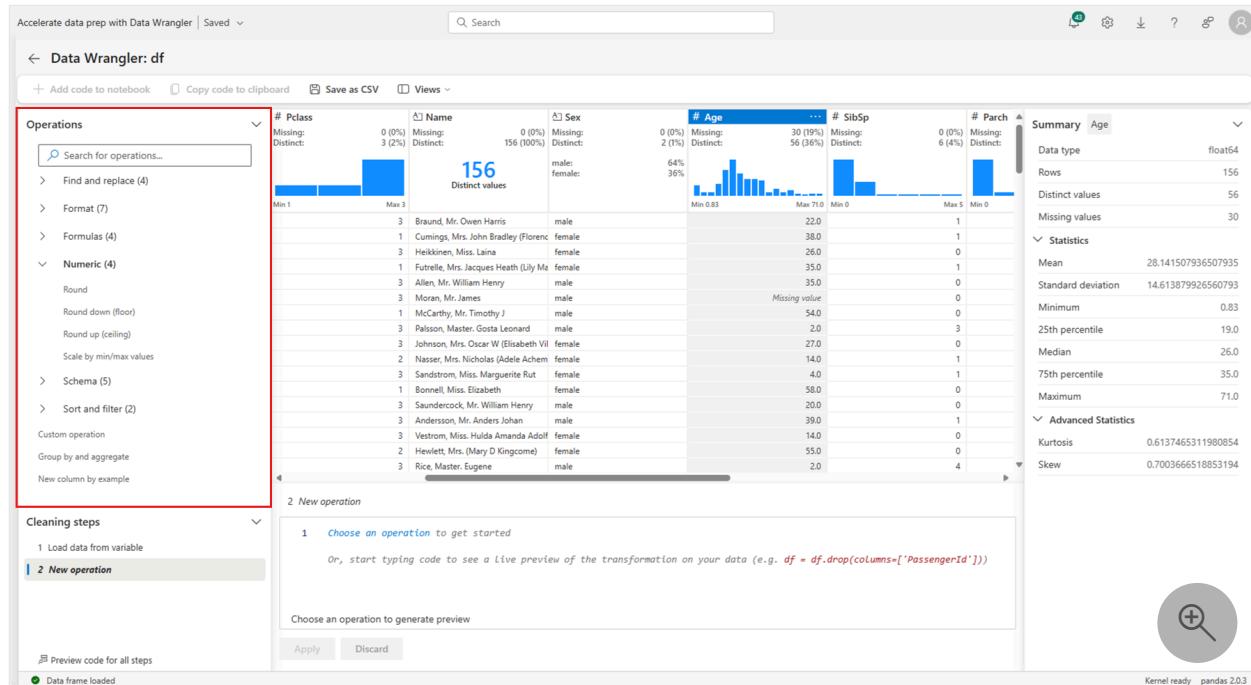
Tip

Column-specific statistics and visuals (both in the "Summary" panel and in the column headers) depend on the column datatype. For instance, a binned histogram of a numeric column appears in the column header only if the column is cast as a numeric type, as shown in this screenshot:



Browsing data-cleaning operations

A searchable list of data-cleaning steps can be found in the "Operations" panel. From the "Operations" panel, selection of a data-cleaning step prompts you to provide a target column or columns, along with any necessary parameters to complete the step. For example, the prompt to numerically scale a column requires a new range of values, as shown in this screenshot:



Tip

You can apply a smaller selection of operations from the menu of each column header, as shown in this screenshot:

The screenshot shows the Data Wrangler interface with the 'df' dataset loaded. A context menu is open over the 'Sex' column header, listing options such as 'Sort ascending', 'Sort descending', 'Filter', 'Rename column', 'Drop columns', 'Change column type', and 'Hide column insights'. The main workspace displays various data visualizations and summary statistics for the dataset.

Previewing and applying operations

The Data Wrangler display grid automatically previews the results of a selected operation, and the corresponding code automatically appears in the panel below the grid. To commit the previewed code, select "Apply" in either place. To delete the previewed code and try a new operation, select "Discard" as shown in this screenshot:

The screenshot shows the Data Wrangler interface with the 'df' dataset loaded. The 'Operations' panel shows a 'Scale by min/max values' operation applied to the 'Age' column, with new minimum and maximum values set to 0 and 1 respectively. The main workspace displays various data visualizations and summary statistics for the dataset. A preview of the transformation code is shown in the preview panel at the bottom.

```

1 # Scale column 'Age' between 0 and 1
2 new_min, new_max = 0, 1
3 old_min, old_max = df['Age'].min(), df['Age'].max()
4 df['Age'] = (df['Age'] - old_min) / (old_max - old_min) * (new_max - new_min) + new_min
    
```

Once an operation is applied, the Data Wrangler display grid and summary statistics update to reflect the results. The code appears in the running list of committed operations, located in the "Cleaning steps" panel, as shown in this screenshot:

Tip

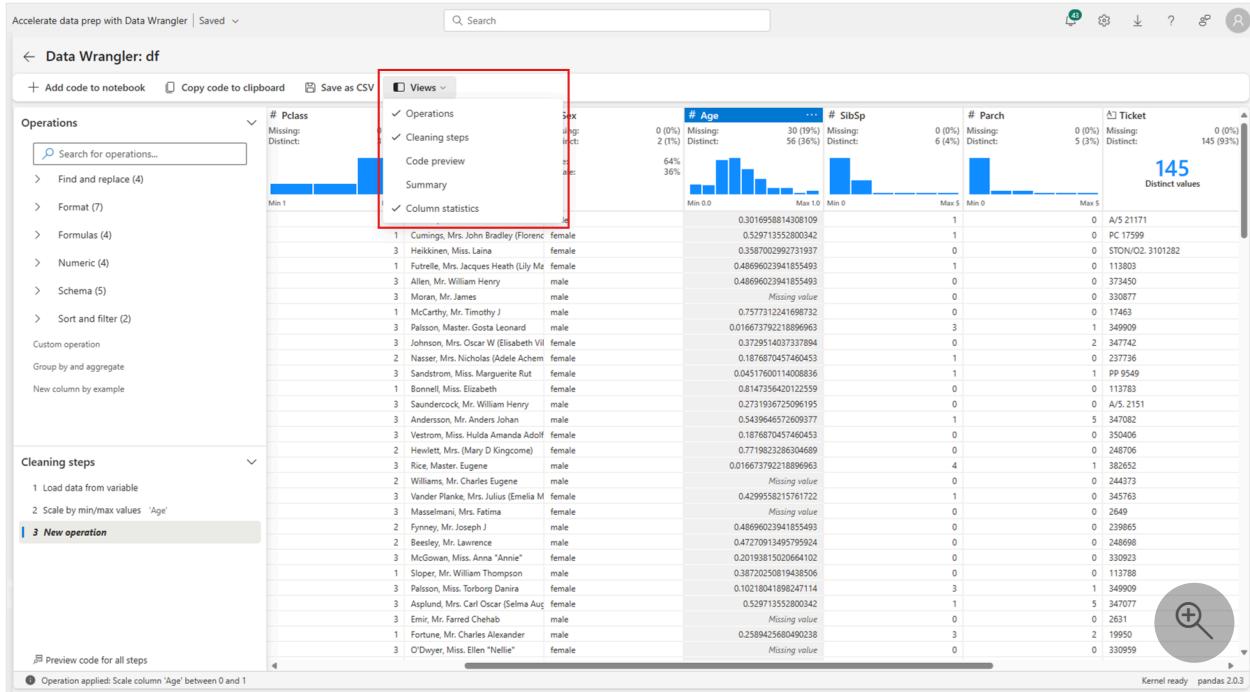
You can always undo the most recently applied step. In the "Cleaning steps" panel, a trash can icon will appear if you hover your cursor over that most recently applied step, as shown in this screenshot:

This table summarizes the operations that Data Wrangler currently supports:

Operation	Description
Sort	Sort a column in ascending or descending order
Filter	Filter rows based on one or more conditions
One-hot encode	Create new columns for each unique value in an existing column, indicating the presence or absence of those values per row
Multi-label binarizer	Split data using a separator and create new columns for each category, marking 1 if a row has that category and 0 if it doesn't.
Change column type	Change the data type of a column
Drop column	Delete one or more columns
Select column	Choose one or more columns to keep, and delete the rest
Rename column	Rename a column
Drop missing values	Remove rows with missing values
Drop duplicate rows	Drop all rows that have duplicate values in one or more columns
Fill missing values	Replace cells with missing values with a new value
Find and replace	Replace cells with an exact matching pattern
Group by column and aggregate	Group by column values and aggregate results
Strip whitespace	Remove whitespace from the beginning and end of text
Split text	Split a column into several columns based on a user-defined delimiter
Convert text to lowercase	Convert text to lowercase
Convert text to uppercase	Convert text to UPPERCASE
Scale min/max values	Scale a numerical column between a minimum and maximum value
Flash Fill	Automatically create a new column based on examples derived from an existing column

Modify your display

At any time, you can customize the interface with the "Views" tab in the toolbar located above the Data Wrangler display grid. This can hide or show different panes based on your preferences and screen size, as shown in this screenshot:

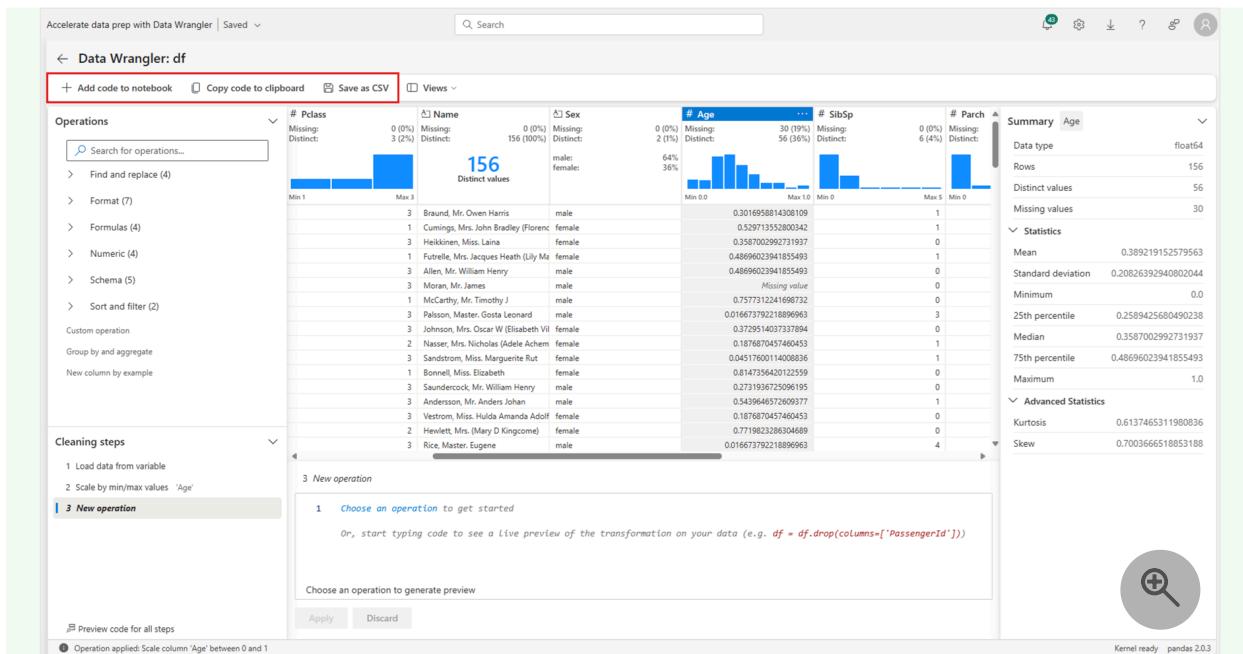


Saving and exporting code

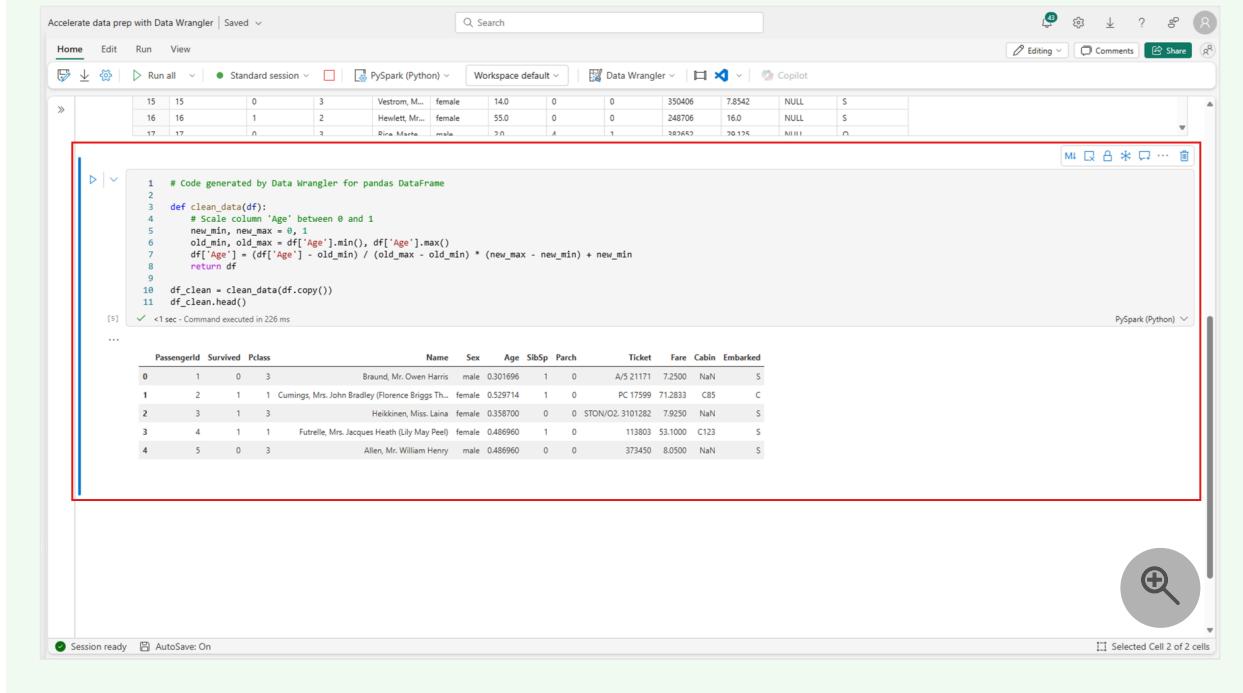
The toolbar above the Data Wrangler display grid provides options to save the generated code. You can copy the code to the clipboard, or export it to the notebook as a function. Exporting the code closes Data Wrangler and adds the new function to a code cell in the notebook. You can also download the cleaned DataFrame as a csv file.

Tip

Data Wrangler generates code that is applied only when you manually run the new cell, and it won't overwrite your original DataFrame, as shown in this screenshot:



You can then run that exported code, as shown in this screenshot:



Related content

- To try out Data Wrangler on Spark DataFrames, visit [this companion article](#)
- For a live-action demo of Data Wrangler in Fabric, check out [this video from our friends at Guy in a Cube](#) ↗
- To try out Data Wrangler in Visual Studio Code, head to [Data Wrangler in VS Code](#) ↗
- Did we miss a feature you need? Let us know! Suggest it at the [Fabric Ideas forum](#) ↗

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

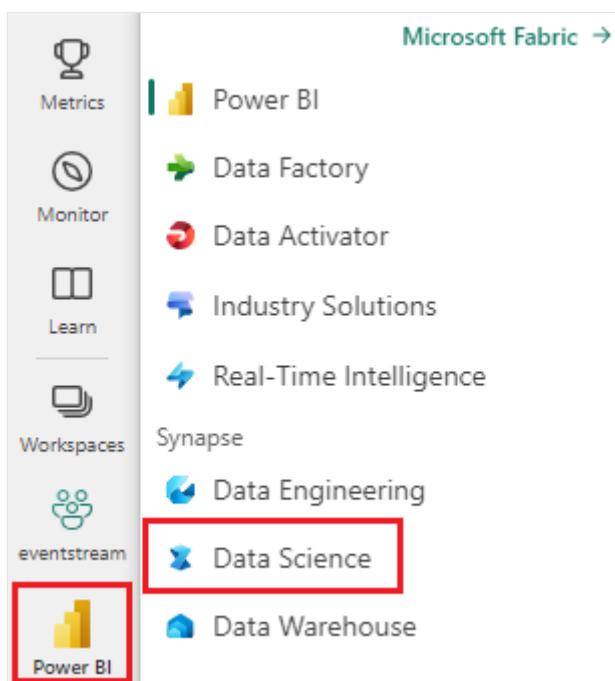
How to use Data Wrangler on Spark DataFrames

Article • 08/12/2024

[Data Wrangler](#), a notebook-based tool for exploratory data analysis, now supports both Spark DataFrames and pandas DataFrames, generating PySpark code in addition to Python code. For a general overview of Data Wrangler, which covers how to explore and transform pandas DataFrames, see the [the main tutorial](#). The following tutorial shows how to use Data Wrangler to explore and transform Spark DataFrames.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



Limitations

- Custom code operations are currently supported only for pandas DataFrames.
- Data Wrangler's display works best on large monitors, although different portions of the interface can be minimized or hidden to accommodate smaller screens.

Launching Data Wrangler with a Spark DataFrame

Users can open Spark DataFrames in Data Wrangler directly from a Microsoft Fabric notebook, by navigating to the same dropdown prompt where pandas DataFrames are displayed. A list of active Spark DataFrames appears in the dropdown beneath the list of active pandas variables.

The next code snippet creates a Spark DataFrame with the same sample data used in the [pandas Data Wrangler tutorial](#):

Python

```
import pandas as pd

# Read a CSV into a Spark DataFrame
sdf =
spark.createDataFrame(pd.read_csv("https://raw.githubusercontent.com/plotly/
datasets/master/titanic.csv"))
display(sdf)
```

In the notebook ribbon "Home" tab, use the Data Wrangler dropdown prompt to browse active DataFrames available for editing. Select the one you wish to open in Data Wrangler.

💡 Tip

Data Wrangler cannot be opened while the notebook kernel is busy. An executing cell must finish its execution before Data Wrangler can be launched.

```

1 import pandas as pd
2
3 # Read a CSV Into a Spark DataFrame
4 sdf = spark.createDataFrame(pd.read_csv("https://raw.githubusercontent.com/plotly/datasets/master/titanic.csv"))
5 display(sdf)

```

1 sec - Command executed in 925 ms

Log

Table Chart Download Showing rows 1 - 156

#	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	1	0	3	Braund, Mr. ...	male	22.0	1	0	A/5 21171	7.25	NULL	S
2	2	1	1	Cumings, ...	female	38.0	1	0	PC 17599	71.2833	C85	C
3	3	1	3	Heikkinen, ...	female	26.0	0	0	STON/O2. 3...	7.925	NULL	S
4	4	1	1	Futrelle, Mr...	female	35.0	1	0	113803	53.1	C123	S
5	5	0	3	Allan, Mr. ...	male	35.0	0	0	373450	8.05	NULL	S
6	6	0	3	Moran, Mr. ...	male	NULL	0	0	330877	8.4583	NULL	Q
7	7	0	1	McCarthy, ...	male	54.0	0	0	17463	51.8625	E46	S
8	8	0	3	Palsson, Ma...	male	2.0	3	1	349909	21.075	NULL	S
9	9	1	3	Johnson, M...	female	27.0	0	2	347742	11.1333	NULL	S
10	10	1	2	Nasser, Mrs. ...	female	14.0	1	0	237736	30.0708	NULL	C
11	11	1	3	Sandstrom, ...	female	4.0	1	1	PP 9549	16.7	G6	S
12	12	1	1	Bonnell, M...	female	58.0	0	0	113783	26.55	C103	S
13	13	0	3	Saundercock...	male	20.0	0	0	A/5. 2151	8.05	NULL	S
14	14	0	3	Anderson, ...	male	39.0	1	5	347082	31.275	NULL	S
15	15	0	3	Vestrom, M...	female	14.0	0	0	350406	7.8542	NULL	S
16	16	1	2	Hewlett, Mr...	female	55.0	0	0	248706	16.0	NULL	S
17	17	0	2	Ripa, Marte...	male	30.0	1	1	387642	30.154	NULL	Q

Choosing custom samples

Data Wrangler automatically converts Spark DataFrames to pandas samples for performance reasons. However, all the code generated by the tool is ultimately translated to PySpark when it exports back to the notebook. As with any pandas DataFrame, you can customize the default sample by selecting "Choose custom sample" from the Data Wrangler dropdown menu. Doing so launches a pop-up with options to specify the size of the desired sample (number of rows) and the sampling method (first records, last records, or a random set).

```

1 import pandas as pd
2
3 # Read a CSV Into a Spark DataFrame
4 sdf = spark.createDataFrame(pd.read_csv("https://raw.githubusercontent.com/plotly/datasets/master/titanic.csv"))
5 display(sdf)

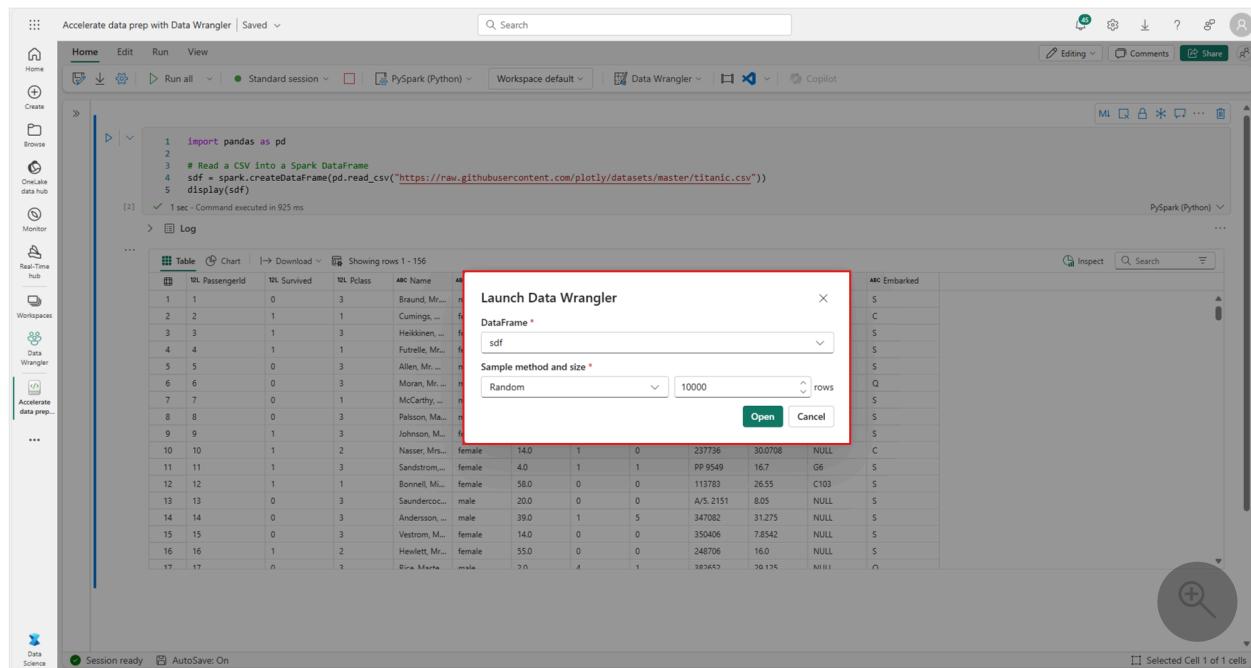
```

1 sec - Command executed in 925 ms

Log

Table Chart Download Showing rows 1 - 156

#	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	1	0	3	Braund, Mr. ...	male	22.0	1	0	A/5 21171	7.25	NULL	S
2	2	1	1	Cumings, ...	female	38.0	1	0	PC 17599	71.2833	C85	C
3	3	1	3	Heikkinen, ...	female	26.0	0	0	STON/O2. 3...	7.925	NULL	S
4	4	1	1	Futrelle, Mr...	female	35.0	1	0	113803	53.1	C123	S
5	5	0	3	Allan, Mr. ...	male	35.0	0	0	373450	8.05	NULL	S
6	6	0	3	Moran, Mr. ...	male	NULL	0	0	330877	8.4583	NULL	Q
7	7	0	1	McCarthy, ...	male	54.0	0	0	17463	51.8625	E46	S
8	8	0	3	Palsson, Ma...	male	2.0	3	1	349909	21.075	NULL	S
9	9	1	3	Johnson, M...	female	27.0	0	2	347742	11.1333	NULL	S
10	10	1	2	Nasser, Mrs. ...	female	14.0	1	0	237736	30.0708	NULL	C
11	11	1	3	Sandstrom, ...	female	4.0	1	1	PP 9549	16.7	G6	S
12	12	1	1	Bonnell, M...	female	58.0	0	0	113783	26.55	C103	S
13	13	0	3	Saundercock...	male	20.0	0	0	A/5. 2151	8.05	NULL	S
14	14	0	3	Anderson, ...	male	39.0	1	5	347082	31.275	NULL	S
15	15	0	3	Vestrom, M...	female	14.0	0	0	350406	7.8542	NULL	S
16	16	1	2	Hewlett, Mr...	female	55.0	0	0	248706	16.0	NULL	S
17	17	0	2	Ripa, Marte...	male	30.0	1	1	387642	30.154	NULL	Q

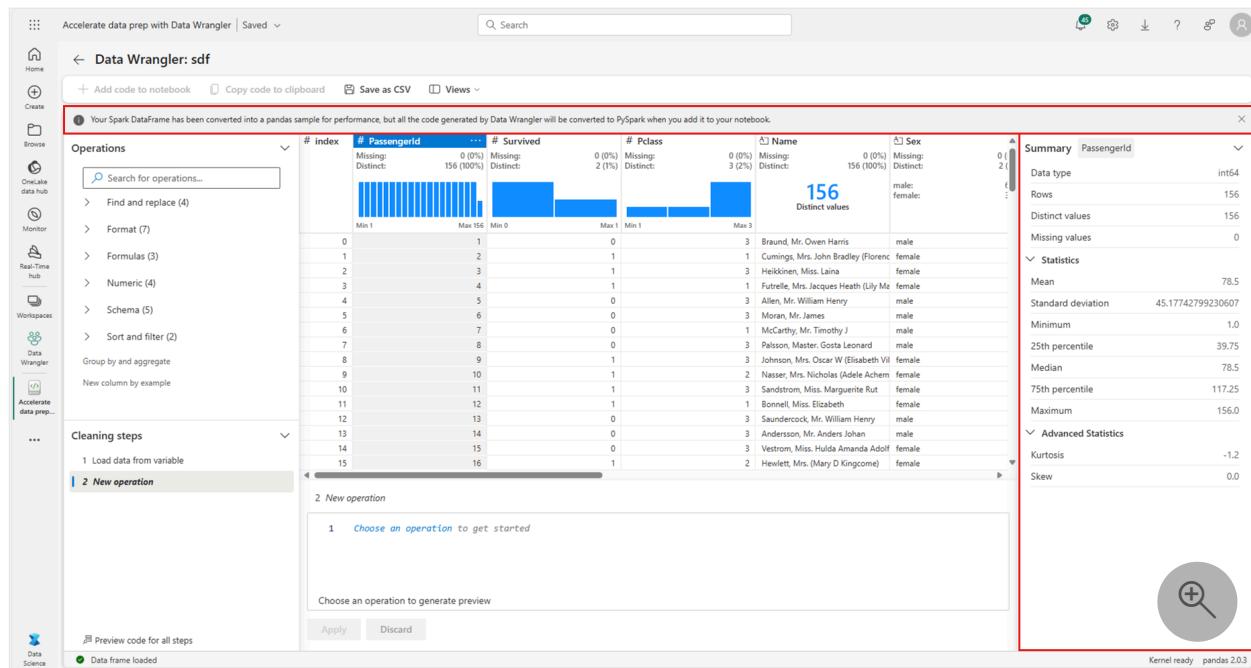


Viewing summary statistics

When Data Wrangler loads, an informational banner above the preview grid reminds you that Spark DataFrames are temporarily converted to pandas samples, but all generated code is ultimately converted to PySpark. Using Data Wrangler on Spark DataFrames is otherwise no different from using it on pandas DataFrames. A descriptive overview in the "Summary" panel displays information about the sample's dimensions, missing values, and more. Selecting any column in the Data Wrangler grid prompts the "Summary" panel to update and display descriptive statistics about that specific column. Quick insights about every column are also available in its header.

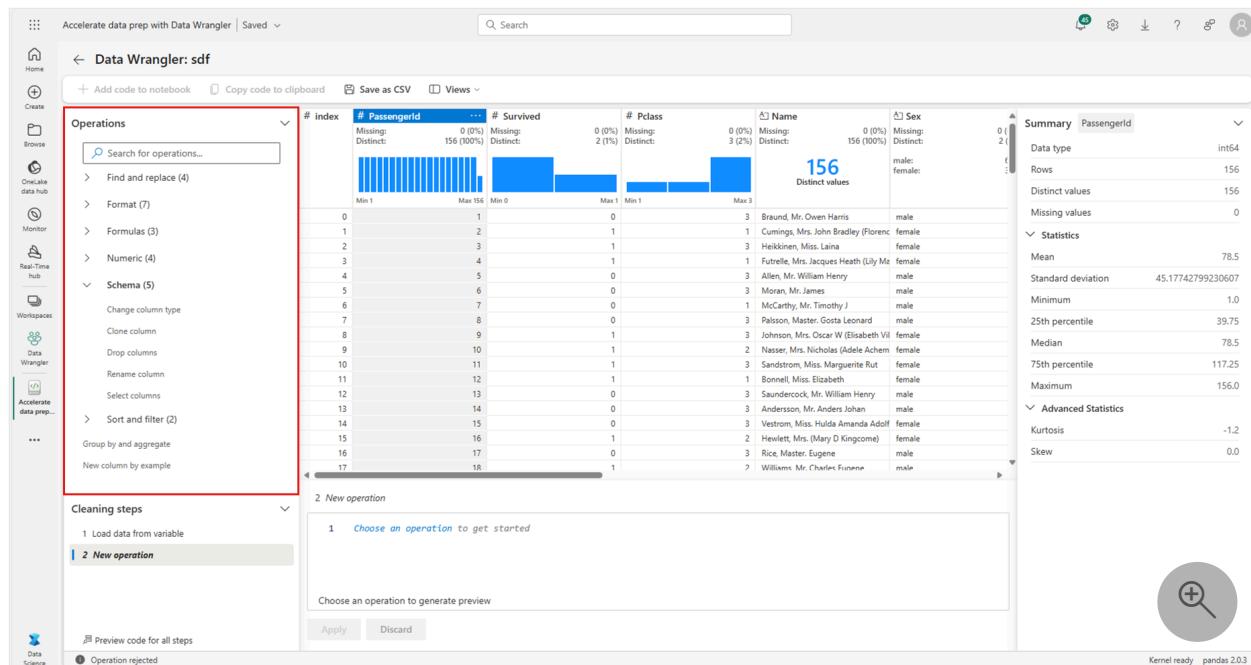
Tip

Column-specific statistics and visuals (both in the "Summary" panel and in the column headers) depend on the column datatype. For instance, a binned histogram of a numeric column will appear in the column header only if the column is cast as a numeric type.



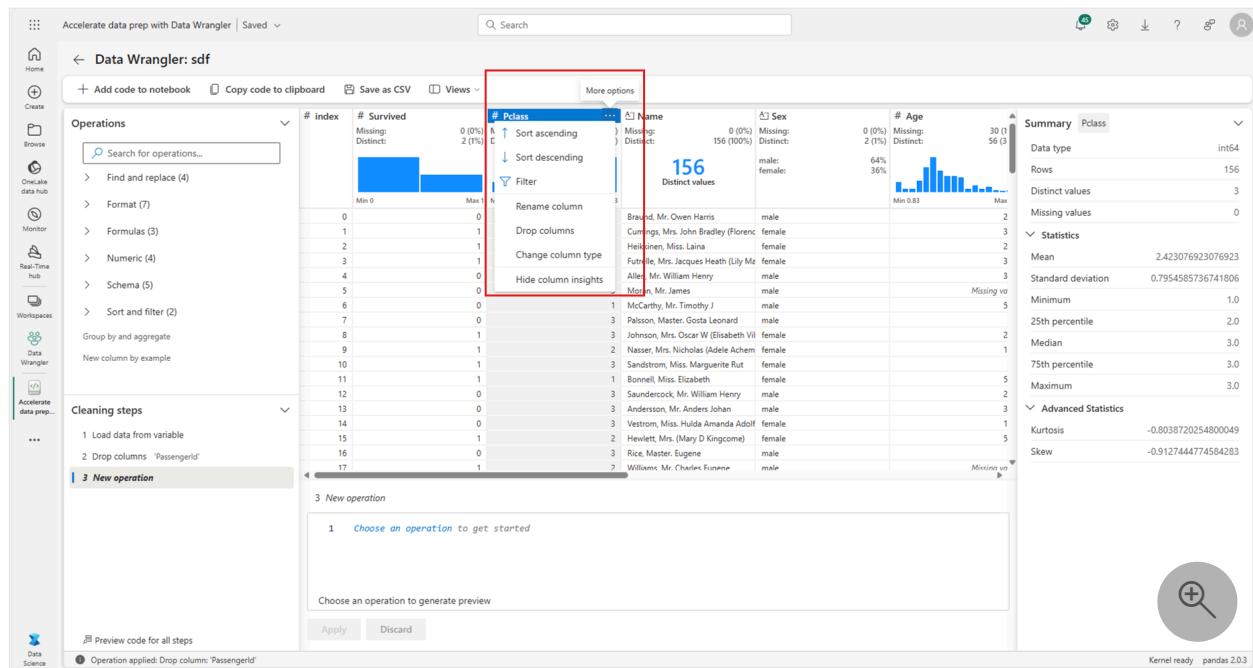
Browsing data-cleaning operations

A searchable list of data-cleaning steps can be found in the "Operations" panel. From the "Operations" panel, selecting a data-cleaning step prompts you to provide a target column or columns, along with any necessary parameters to complete the step. For example, the prompt for scaling a column numerically requires a new range of values.



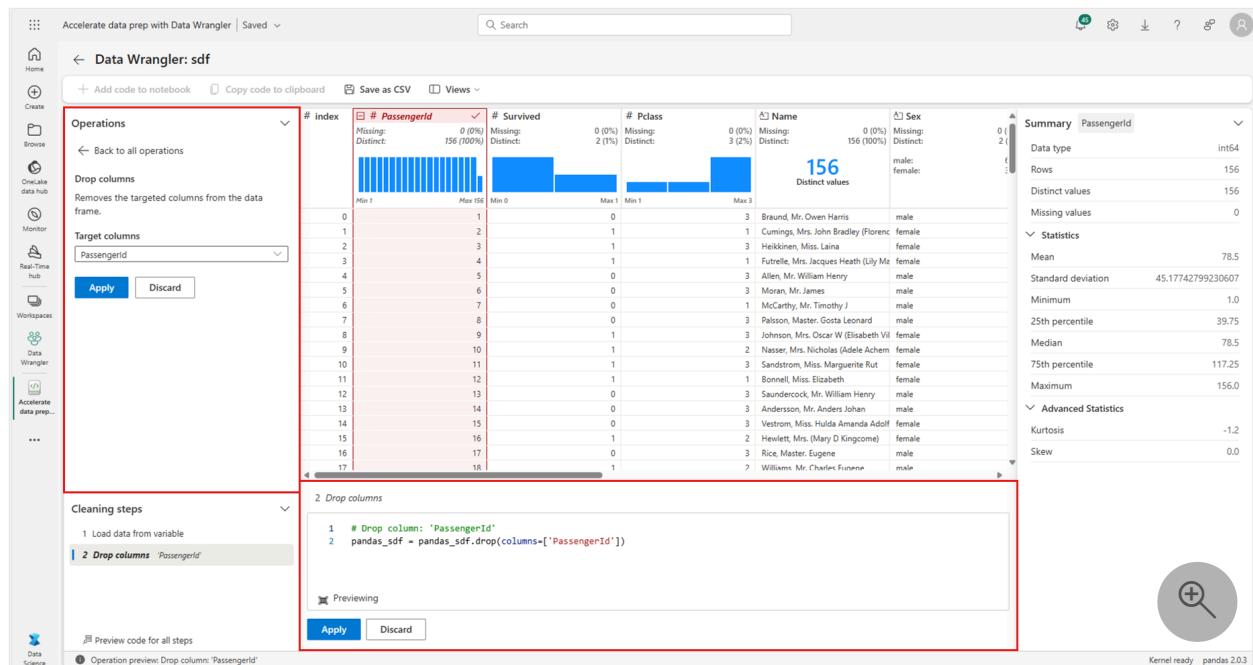
Tip

A smaller selection of operations can be applied from the menu of each column header.

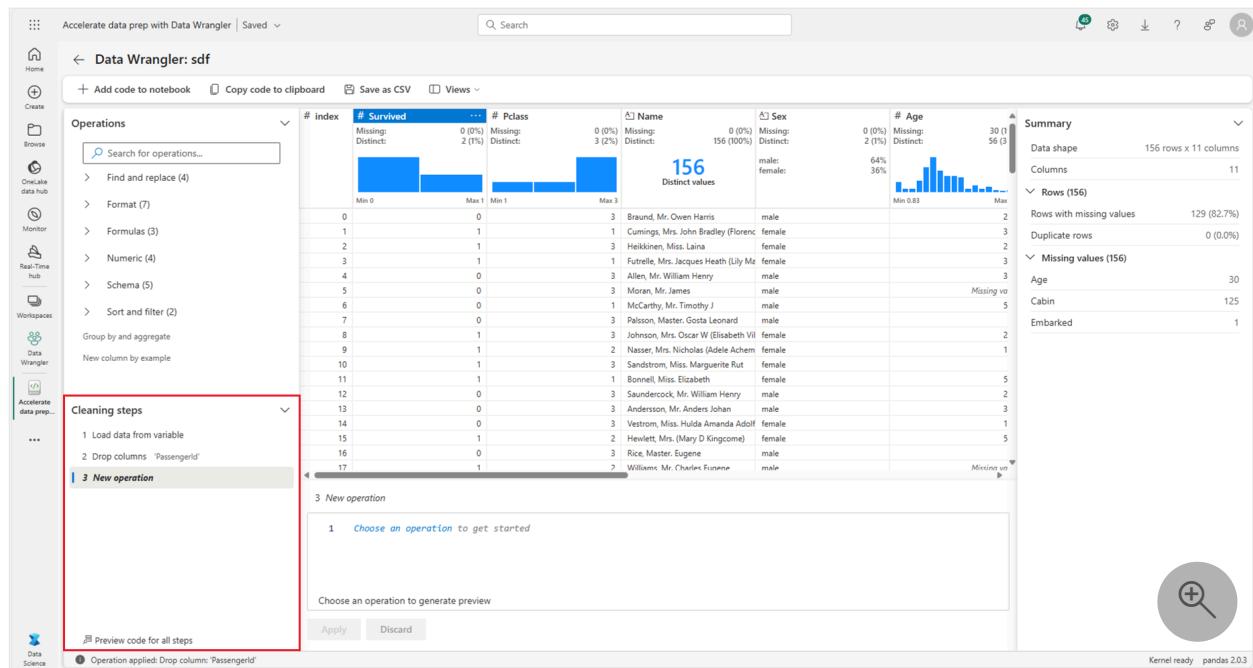


Previewing and applying operations

The results of a selected operation are automatically previewed in the Data Wrangler display grid, and the corresponding code automatically appears in the panel below the grid. To commit the previewed code, select "Apply" in either place. To get rid of the previewed code and try a new operation, select "Discard."

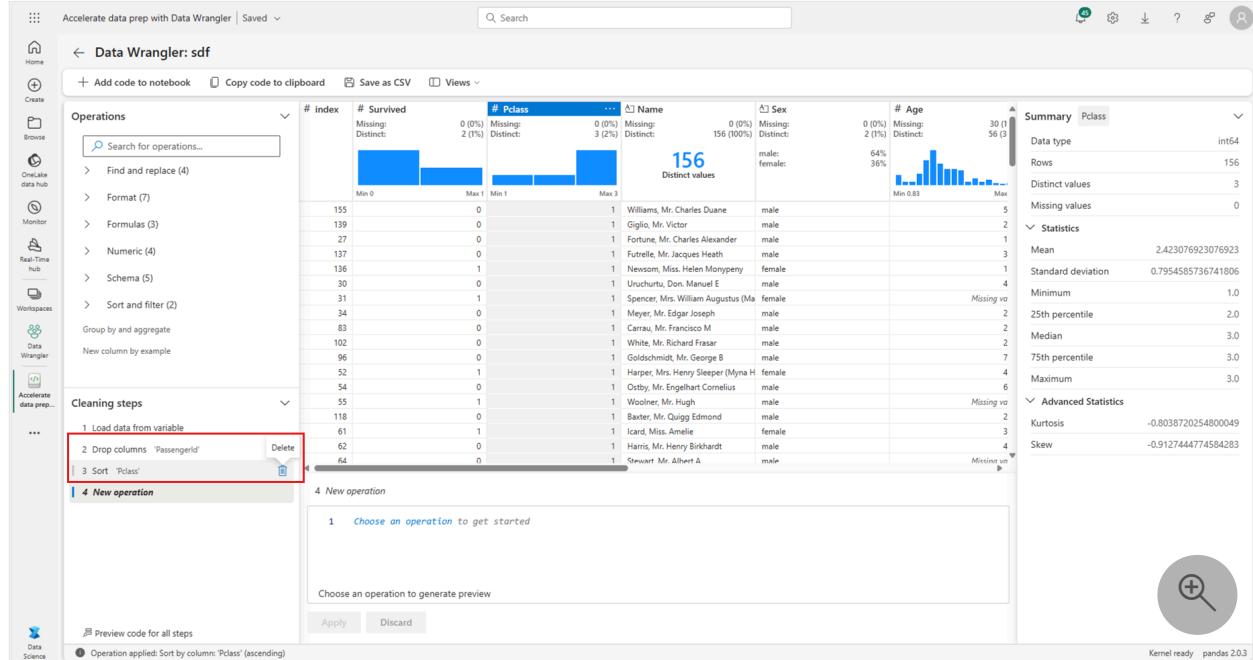


Once an operation is applied, the Data Wrangler display grid and summary statistics update to reflect the results. The code appears in the running list of committed operations, located in the "Cleaning steps" panel.



Tip

You can always undo the most recently applied step with the trash icon beside it, which appears if you hover your cursor over that step in the "Cleaning steps" panel.



The following table summarizes the operations that Data Wrangler currently supports for Spark DataFrames:

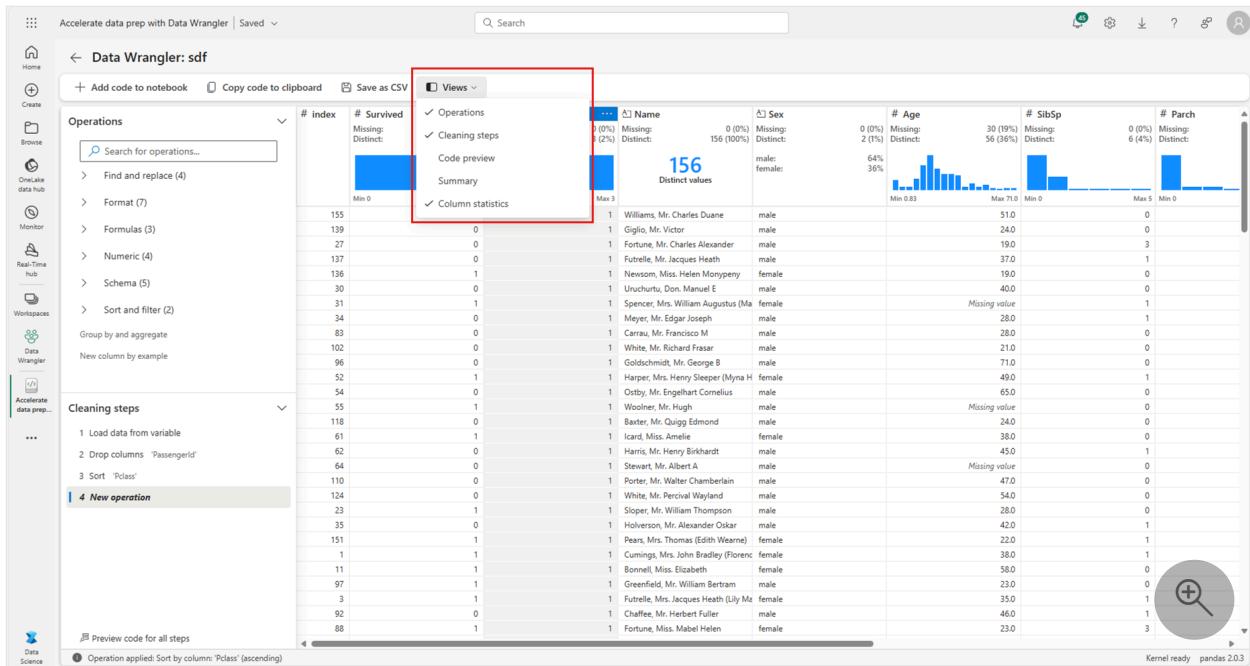
[Expand table](#)

Operation	Description
Sort	Sort a column in ascending or descending order

Operation	Description
Filter	Filter rows based on one or more conditions
One-hot encode	Create new columns for each unique value in an existing column, indicating the presence or absence of those values per row
One-hot encode with delimiter	Split and one-hot encode categorical data using a delimiter
Change column type	Change the data type of a column
Drop column	Delete one or more columns
Select column	Choose one or more columns to keep, and delete the rest
Rename column	Rename a column
Drop missing values	Remove rows with missing values
Drop duplicate rows	Drop all rows that have duplicate values in one or more columns
Fill missing values	Replace cells with missing values with a new value
Find and replace	Replace cells with an exact matching pattern
Group by column and aggregate	Group by column values and aggregate results
Strip whitespace	Remove whitespace from the beginning and end of text
Split text	Split a column into several columns based on a user-defined delimiter
Convert text to lowercase	Convert text to lowercase
Convert text to uppercase	Convert text to UPPERCASE
Scale min/max values	Scale a numerical column between a minimum and maximum value
Flash Fill	Automatically create a new column based on examples derived from an existing column

Modifying your display

At any point, you can customize the interface using the "Views" tab in the toolbar above the Data Wrangler display grid, hiding or showing different panes based on your preferences and screen size.



Saving and exporting code

The toolbar above the Data Wrangler display grid provides options to save the generated code. You can copy the code to the clipboard or export it to the notebook as a function. For Spark DataFrames, all the code generated on the pandas sample is translated to PySpark before it lands back in the notebook. Before Data Wrangler closes, the tool displays a preview of the translated PySpark code and provide an option to export the intermediate pandas code as well.

Tip

The code generated by Data Wrangler won't be applied until you manually run the new cell, and it will not overwrite your original DataFrame.

Accelerate data prep with Data Wrangler | Saved ▾

Q Search

Add code to notebook Copy code to clipboard Save as CSV Views

Operations

Search for operations...

Find and replace (4)

Format (7)

Formulas (3)

Numeric (4)

Schema (5)

Sort and filter (2)

Group by and aggregate

New column by example

Cleaning steps

Load data from variable

Drop columns 'PassengerId'

Sort 'Pclass'

4 New operation

Choose an operation to get started

Choose an operation to generate preview

Apply Discard

Preview code for all steps

Operation applied: Sort by column: 'Pclass' (ascending)

Kernel ready pandas 2.0.3

This screenshot shows the Data Wrangler interface for a saved workspace. The main area displays a preview of the 'titanic' dataset with various summary statistics and histograms for columns like Pclass, Age, and Sex. On the left, a sidebar lists 'Cleaning steps' such as loading data from a variable, dropping columns, and sorting. A modal window titled '4 New operation' is open, prompting the user to 'Choose an operation to get started'. Below it, another modal titled 'Add code to notebook' shows generated PySpark code for cleaning the data.

Accelerate data prep with Data Wrangler | Saved ▾

Q Search

Add code to notebook Copy code to clipboard Save as CSV Views

Operations

Search for operations...

Find and replace (4)

Format (7)

Formulas (3)

Numeric (4)

Schema (5)

Sort and filter (2)

Group by and aggregate

New column by example

Cleaning steps

Load data from variable

Drop columns 'PassengerId'

Sort 'Pclass'

4 New operation

Choose an operation to get started

Choose an operation to generate preview

Apply Discard

Preview code for all steps

Operation applied: Sort by column: 'Pclass' (ascending)

Kernel ready pandas 2.0.3

This screenshot is identical to the one above, showing the Data Wrangler interface with the same workspace and data preview. The 'Add code to notebook' modal is now open, displaying the generated PySpark code for cleaning the data. The code includes dropping the 'PassengerId' column, sorting by 'Pclass', and returning the modified DataFrame.

The screenshot shows the Data Wrangler interface within Azure Data Studio. On the left, there's a sidebar with icons for Home, Create, Browse, OneLake data hub, Monitor, Real-Time hub, Workspaces, Data Wrangler, Accelerate data prep..., and Data Science. The main area has a title bar with 'Accelerate data prep with Data Wrangler | Saved' and a search bar. Below the title bar, there are tabs for Home, Edit, Run, View, and a workspace dropdown. The central part of the interface shows a code editor with PySpark Python code:

```

1 # Code generated by Data Wrangler for PySpark DataFrame
2
3 def clean_data(sdf):
4     # Drop column: 'PassengerId'
5     sdf = sdf.drop('PassengerId')
6     # Sort by column: 'Pclass'(ascending)
7     sdf = sdf.sort(sdf['Pclass'].asc())
8     return sdf
9
10 sdf_clean = clean_data(sdf)
11 display(sdf_clean)

```

Below the code editor is a log message: "[4] ✓ 2 sec - Command executed in 1 sec: 572 ms". To the right of the code editor is a data preview table titled "Table" showing rows 1-156 of the Titanic dataset. The columns are: Survived, Pclass, Name, Sex, Age, SibSp, Parch, Ticket, Fare, Cabin, Embarked. The data preview shows various passenger details like name, sex, age, and survival status.

Related content

- To get an overview of Data Wrangler, see [this companion article](#).
- To try out Data Wrangler in Visual Studio Code, head to [Data Wrangler in VS Code](#).
- Are we missing a feature you need? Suggest it on the [Fabric Ideas forum](#).

Feedback

Was this page helpful?

Yes

No

[Provide product feedback](#) | [Ask the community](#)

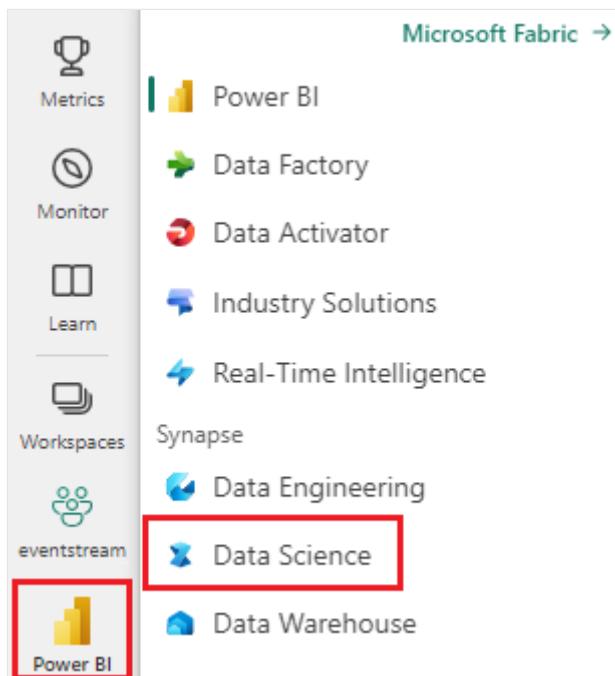
Read from semantic models and write data consumable by Power BI using python

Article • 11/19/2024

In this article, you learn how to read data and metadata and evaluate measures in semantic models using the SemPy python library in Microsoft Fabric. You also learn how to write data that semantic models can consume.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



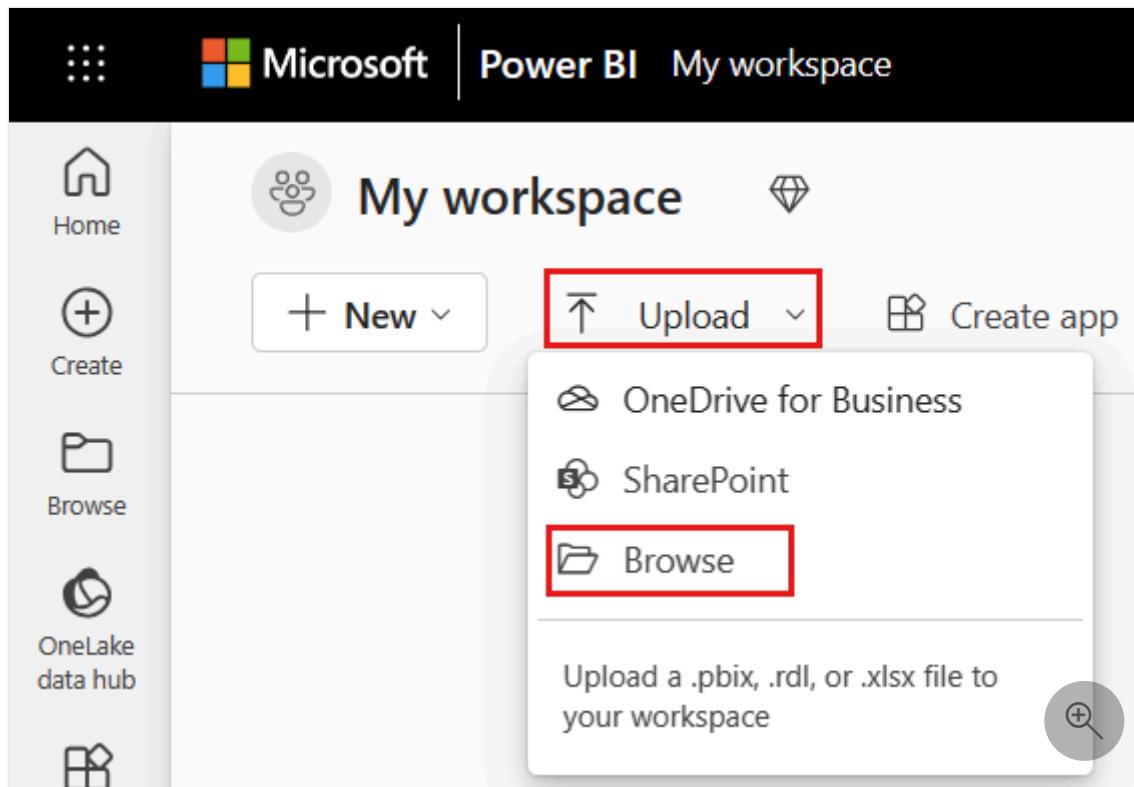
- Visit the Data Science experience in Microsoft Fabric.
- Create a [new notebook](#), to copy/paste code into cells
- For Spark 3.4 and above, Semantic link is available in the default runtime when using Fabric, and there's no need to install it. If you're using Spark 3.3 or below, or if you want to update to the most recent version of Semantic Link, you can run the command: `python %pip install -U semantic-link`
- [Add a Lakehouse to your notebook](#)

- Download the *Customer Profitability Sample.pbix* semantic model from the [datasets folder](#) of the fabric-samples repository, and save the semantic model locally

Upload the semantic model into your workspace

This article uses the *Customer Profitability Sample.pbix* semantic model. This semantic model references a company manufacturing marketing materials. It contains product, customer, and corresponding revenue data for various business units.

1. Open your [workspace](#) in Fabric Data Science
2. Select **Upload > Browse**, and select the *Customer Profitability Sample.pbix* semantic model.



When the upload is complete, your workspace has three new artifacts: a Power BI report, a dashboard, and a semantic model named *Customer Profitability Sample*. The steps in this article rely on that semantic model.

The screenshot shows the Microsoft Fabric workspace interface. At the top, there's a navigation bar with icons for 'My workspace' (a person icon), 'New', 'Upload', 'Create deployment pipeline', 'Create app', and a three-dot menu. Below the navigation bar is a table listing semantic models:

	Name	Type	Owner
Customer Profitability Sample	Report	My workspace	
Customer Profitability Sample	Dataset	My workspace	
Customer Profitability Sample.pbix	Dashboard	My workspace	

A magnifying glass icon is located in the bottom right corner of the table area.

Use Python to read data from semantic models

The SemPy Python API can retrieve data and metadata from semantic models located in a Microsoft Fabric workspace. The API can also execute queries on them.

Your notebook, Power BI dataset semantic model, and [lakehouse](#) can be located in the same workspace or in different workspaces. By default, SemPy tries to access your semantic model from:

- The workspace of your lakehouse, if you attached a lakehouse to your notebook.
- The workspace of your notebook, if there's no lakehouse attached.

If your semantic model isn't located in either of these workspaces, you must specify the workspace of your semantic model when you call a SemPy method.

To read data from semantic models:

1. List the available semantic models in your workspace.

```
Python

import sempy.fabric as fabric

df_datasets = fabric.list_datasets()
df_datasets
```

2. List the tables available in the *Customer Profitability Sample* semantic model.

```
Python

df_tables = fabric.list_tables("Customer Profitability Sample",
                               include_columns=True)
```

```
df_tables
```

3. List the measures defined in the *Customer Profitability Sample* semantic model.

💡 Tip

In the following code sample, we specified the workspace for SemPy to use for accessing the semantic model. You can replace `Your Workspace` with the name of the workspace where you uploaded the semantic model (from the [Upload the semantic model into your workspace](#) section).

Python

```
df_measures = fabric.list_measures("Customer Profitability Sample",
                                    workspace="Your Workspace")
df_measures
```

Here, we determined that the *Customer* table is the table of interest.

4. Read the *Customer* table from the *Customer Profitability Sample* semantic model.

Python

```
df_table = fabric.read_table("Customer Profitability Sample",
                             "Customer")
df_table
```

⚠ Note

- Data is retrieved using XMLA. This requires at least [XMLA read-only](#) to be enabled.
- The amount of retrievable data is limited by - the [maximum memory per query](#) of the capacity SKU that hosts the semantic model - the Spark driver node (visit [node sizes](#) for more information) that runs the notebook
- All requests use low priority to minimize the impact on Microsoft Azure Analysis Services performance, and are billed as [interactive requests](#).

5. Evaluate the *Total Revenue* measure for the state and date of each customer.

Python

```
df_measure = fabric.evaluate_measure(  
    "Customer Profitability Sample",  
    "Total Revenue",  
    ["'Customer'[State]", "Calendar[Date]"])  
df_measure
```

① Note

- By default, data is **not** retrieved using XMLA and therefore doesn't require XMLA read-only to be enabled.
- The data is **not** subject to [Power BI backend limitations](#).
- The amount of retrievable data is limited by - the [maximum memory per query](#) of the capacity SKU hosting the semantic model - the Spark driver node (visit [node sizes](#) for more information) that runs the notebook
- All requests are billed as [interactive requests](#)

6. To add filters to the measure calculation, specify a list of permissible values for a particular column.

Python

```
filters = {  
    "State[Region]": ["East", "Central"],  
    "State[State]": ["FLORIDA", "NEW YORK"]  
}  
df_measure = fabric.evaluate_measure(  
    "Customer Profitability Sample",  
    "Total Revenue",  
    ["Customer[State]", "Calendar[Date]"],  
    filters=filters)  
df_measure
```

7. You can also evaluate the *Total Revenue* measure per customer's state and date with a [DAX query](#).

Python

```
df_dax = fabric.evaluate_dax(  
    "Customer Profitability Sample",  
    """  
    EVALUATE SUMMARIZECOLUMNS(  
        'State'[Region],  
        'Calendar'[Date].[Year],
```

```
'Calendar'[Date].[Month],  
"Total Revenue",  
CALCULATE([Total Revenue]))  
"""")
```

ⓘ Note

- Data is retrieved using XMLA and therefore requires at least [XMLA read-only](#) to be enabled
- The amount of retrievable data is limited by the available memory in Microsoft Azure Analysis Services and the Spark driver node (visit [node sizes](#) for more information)
- All requests use low priority to minimize the impact on Analysis Services performance and are billed as interactive requests

8. Use the `%%dax` cell magic to evaluate the same DAX query, without the need to import the library. Run this cell to load `%%dax` cell magic:

```
%load_ext sempy
```

The workspace parameter is optional. It follows the same rules as the workspace parameter of the `evaluate_dax` function.

The cell magic also supports access of Python variables with the `{variable_name}` syntax. To use a curly brace in the DAX query, escape it with another curly brace (example: `EVALUATE {{1}}`).

DAX

```
%%dax "Customer Profitability Sample" -w "Your Workspace"  
EVALUATE SUMMARIZECOLUMNS(  
    'State'[Region],  
    'Calendar'[Date].[Year],  
    'Calendar'[Date].[Month],  
    "Total Revenue",  
    CALCULATE([Total Revenue]))
```

The resulting FabricDataFrame is available via the `_` variable. That variable captures the output of the last executed cell.

```
Python
```

```
df_dax = _  
df_dax.head()
```

9. You can add measures to data retrieved from external sources. This approach combines three tasks:

- It resolves column names to Power BI dimensions
- It defines group by columns
- It filters the measure Any column names that can't be resolved within the given semantic model are ignored (visit the supported [DAX syntax](#) resource for more information).

```
Python
```

```
from sempy.fabric import FabricDataFrame  
  
df = FabricDataFrame({  
    "Sales Agent": ["Agent 1", "Agent 1", "Agent 2"],  
    "Customer[Country/Region)": ["US", "GB", "US"],  
    "Industry[Industry)": ["Services", "CPG", "Manufacturing"],  
}  
)  
  
joined_df = df.add_measure("Total Revenue", dataset="Customer  
Profitability Sample")  
joined_df
```

Special parameters

The SemPy `read_table` and `evaluate_measure` methods have more parameters that are useful for manipulating the output. These parameters include:

- `fully_qualified_columns`: For a "True" value, the methods return column names in the form `TableName[ColumnName]`
- `num_rows`: The number of rows to output in the result
- `pandas_convert_dtypes`: For a "True" value, pandas cast the resulting DataFrame columns to the best possible *dtype* [convert_dtypes ↗](#). If this parameter is turned off, type incompatibility issues between columns of related tables can result; the Power BI model might not detect those issues because of [DAX implicit type conversion](#)

SemPy `read_table` also uses the model information that Power BI provides.

- `multiindex_hierarchies`: If "True", it converts [Power BI Hierarchies](#) to a pandas MultiIndex structure

Write data consumable by semantic models

Spark tables added to a Lakehouse are automatically added to the corresponding [default semantic model](#). This example demonstrates how to write data to the attached Lakehouse. The FabricDataFrame accepts the same input data as Pandas dataframes.

Python

```
from sempy.fabric import FabricDataFrame

df_forecast = FabricDataFrame({'ForecastedRevenue': [1, 2, 3]})

df_forecast.to_lakehouse_table("ForecastTable")
```

With Power BI, the *ForecastTable* table can be added to a composite semantic model with the Lakehouse semantic model.

Related content

- See [sempy.functions](#) to learn about usage of semantic functions
- Tutorial: Extract and calculate Power BI measures from a Jupyter notebook
- Explore and validate relationships in semantic models
- How to validate data with semantic link

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Detect, explore, and validate functional dependencies in your data, using semantic link

Article • 11/19/2024

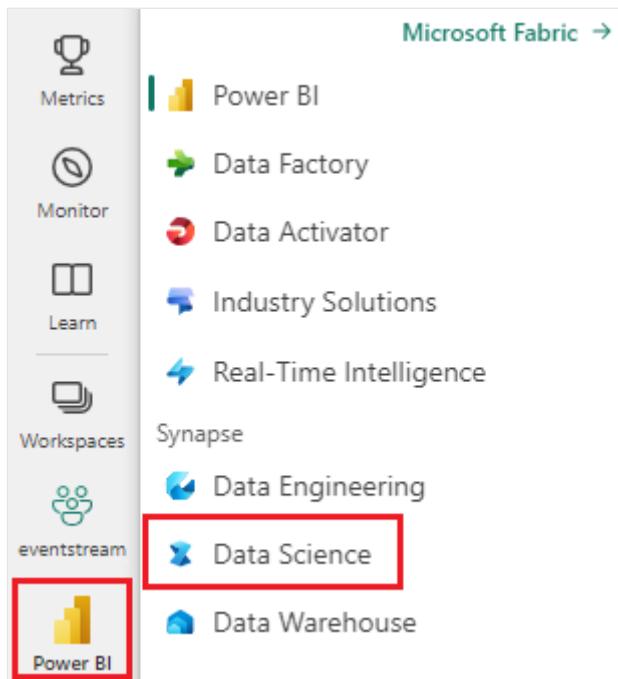
Functional dependencies are relationships between columns in a table, where the values in one column are used to determine the values in another column. Understanding these dependencies can help you uncover patterns and relationships in your data, which in turn can help with feature engineering, data cleaning, and model building tasks. Functional dependencies act as an effective invariant that allows you to find and fix data quality issues that might be hard to detect otherwise.

In this article, you use semantic link to:

- ✓ Find dependencies between columns of a FabricDataFrame
- ✓ Visualize dependencies
- ✓ Identify data quality issues
- ✓ Visualize data quality issues
- ✓ Enforce functional constraints between columns in a dataset

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



- Go to the Data Science experience found in Microsoft Fabric.
- Create a new notebook to copy/paste code into cells.
- For Spark 3.4 and above, Semantic link is available in the default runtime when using Fabric, and there's no need to install it. If you're using Spark 3.3 or below, or if you want to update to the most recent version of Semantic Link, you can run the command: `python %pip install -U semantic-link`
- Add a Lakehouse to your notebook.

For Spark 3.4 and above, Semantic link is available in the default runtime when using Fabric, and there's no need to install it. If you use Spark 3.3 or below, or if you want to update to the most recent version of Semantic Link, run this command:

Python

```
%pip install -U semantic-link
```
Find functional dependencies in data
```

The SemPy `find\_dependencies` function detects functional dependencies between the columns of a FabricDataFrame. The function uses a threshold on conditional entropy to discover approximate functional dependencies, where low conditional entropy indicates strong dependence between columns. To make the `find\_dependencies` function more selective, you can set a lower threshold on conditional entropy. The lower threshold means that only stronger dependencies will be detected.

This Python code snippet demonstrates how to use `find\_dependencies`:

```
```python
from sempy.fabric import FabricDataFrame
from sempy.dependencies import plot_dependency_metadata
```

```

import pandas as pd

df = FabricDataFrame(pd.read_csv("your_data.csv"))

deps = df.find_dependencies()

```

The `find_dependencies` function returns a FabricDataFrame with detected dependencies between columns. A list represents columns that have a 1:1 mapping. The function also removes [transitive edges](#), to try to prune the potential dependencies.

When you specify the `dropna=True` option, rows that have a NaN value in either column are eliminated from evaluation. This can result in nontransitive dependencies, as shown in this example:

[\[+\] Expand table](#)

A	B	C
1	1	1
1	1	1
1	NaN	9
2	NaN	2
2	2	2

In some cases, the dependency chain can form cycles when you specify the `dropna=True` option, as shown in this example:

[\[+\] Expand table](#)

A	B	C
1	1	NaN
2	1	NaN
NaN	1	1
NaN	2	1
1	NaN	1
1	NaN	2

Visualize dependencies in data

After you find functional dependencies in a dataset (using `find_dependencies`), you can visualize the dependencies with the `plot_dependency_metadata` function. This function takes the resulting FabricDataFrame from `find_dependencies` and creates a visual representation of the dependencies between columns and groups of columns.

This Python code snippet shows how to use `plot_dependencies`:

Python

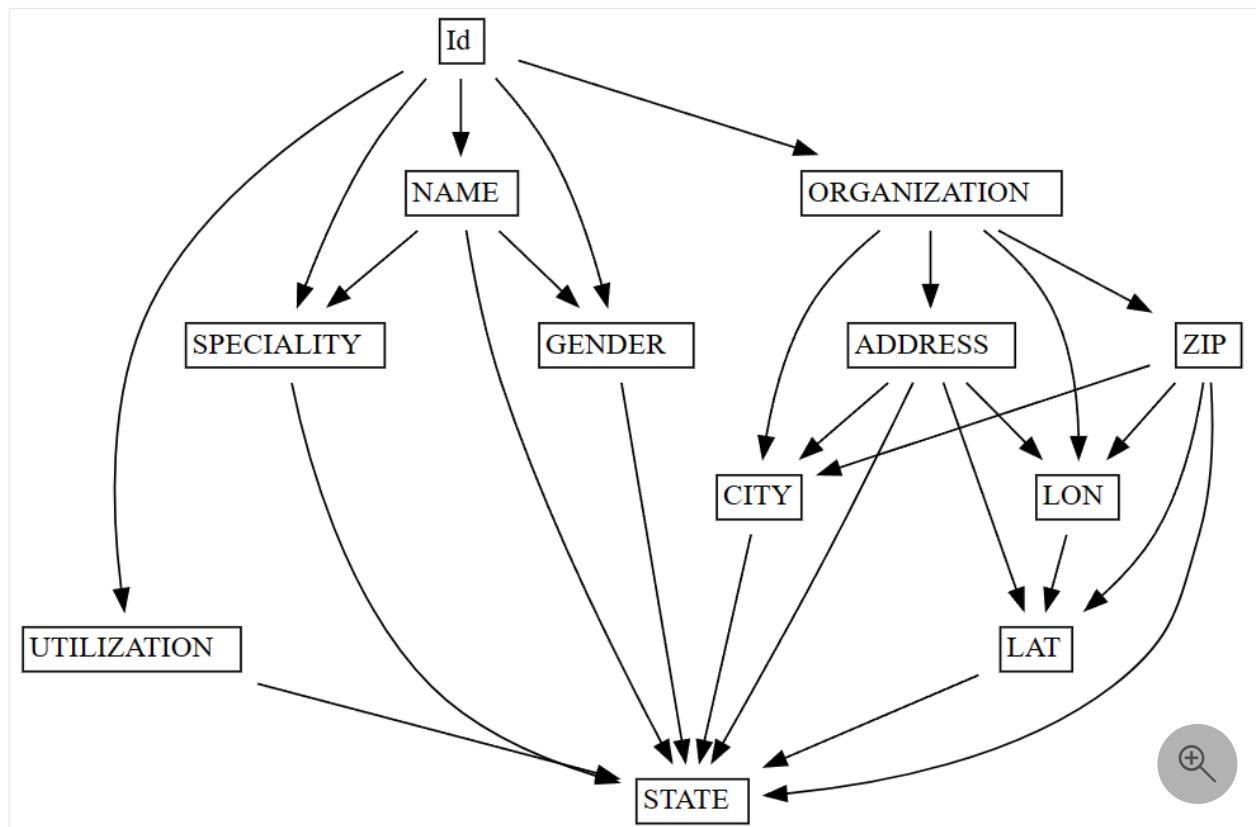
```
from sempy.fabric import FabricDataFrame
from sempy.dependencies import plot_dependency_metadata
from sempy.samples import download_synthea

download_synthea(which='small')

df = FabricDataFrame(pd.read_csv("synthea/csv/providers.csv"))

deps = df.find_dependencies()
plot_dependency_metadata(deps)
```

The `plot_dependency_metadata` function generates a visualization that shows the 1:1 groupings of columns. Columns that belong to a single group are placed in a single cell. If no suitable candidates are found, an empty FabricDataFrame is returned.



Identify data quality issues

Data quality issues can have various forms - for example, missing values, inconsistencies, or inaccuracies. Identifying and addressing these issues is important to ensure the reliability and validity of any analysis or model built on the data. One way to detect data quality issues is to examine violations of functional dependencies between columns in a dataset.

The `list_dependency_violations` function can help identify violations of functional dependencies between dataset columns. Given a determinant column and a dependent column, this function shows values that violate the functional dependency, along with the count of their respective occurrences. This can help inspect approximate dependencies and identify data quality issues.

This code snippet shows how to use the `list_dependency_violations` function:

Python

```
from sempy.fabric import FabricDataFrame
from sempy.samples import download_synthea

download_synthea(which='small')

df = FabricDataFrame(pd.read_csv("synthea/csv/providers.csv"))

violations = df.list_dependencyViolations(determinant_col="ZIP",
dependent_col="CITY")
```

In this example, the function assumes a functional dependency between the ZIP (determinant) and CITY (dependent) columns. If the dataset has data quality issues - for example, the same ZIP Code assigned to multiple cities - the function outputs the data with the problems:

[\[\] Expand table](#)

ZIP	CITY	count
12345	Boston	2
12345	Seattle	1

This output indicates that two different cities (Boston and Seattle) have the same ZIP Code value (12345). This suggests a data quality issue within the dataset.

The `list_dependency_violations` function provides more options that can handle missing values, show values mapped to violating values, limit the number of violations returned, and sort the results by count or determinant column.

The `list_dependency_violations` output can help identify dataset data quality issues. However, you should carefully examine the results and consider the context of your data, to determine the most appropriate course of action to address the identified issues. This approach might involve more data cleaning, validation, or exploration to ensure the reliability and validity of your analysis or model.

Visualize data quality issues

Data quality issues can damage the reliability and validity of any analysis or model built on that data. Identifying and addressing these issues is important to ensure the accuracy of your results. To detect data quality issues, you can examine violations of functional dependencies between columns in a dataset. Visualizing these violations can show the problems more clearly, and help you address them more effectively.

The `plot_dependency_violations` function can help visualize violations of functional dependencies between columns in a dataset. Given a determinant column and a dependent column, this function shows the violating values in a graphical format, to make it easier to understand the nature and extent of the data quality issues.

This code snippet shows how to use the `plot_dependency_violations` function:

Python

```
from sempy.fabric import FabricDataFrame
from sempy.dependencies import plot_dependencyViolations
from sempy.samples import download_synthea

download_synthea(which='small')

df = FabricDataFrame(pd.read_csv("synthea/csv/providers.csv"))

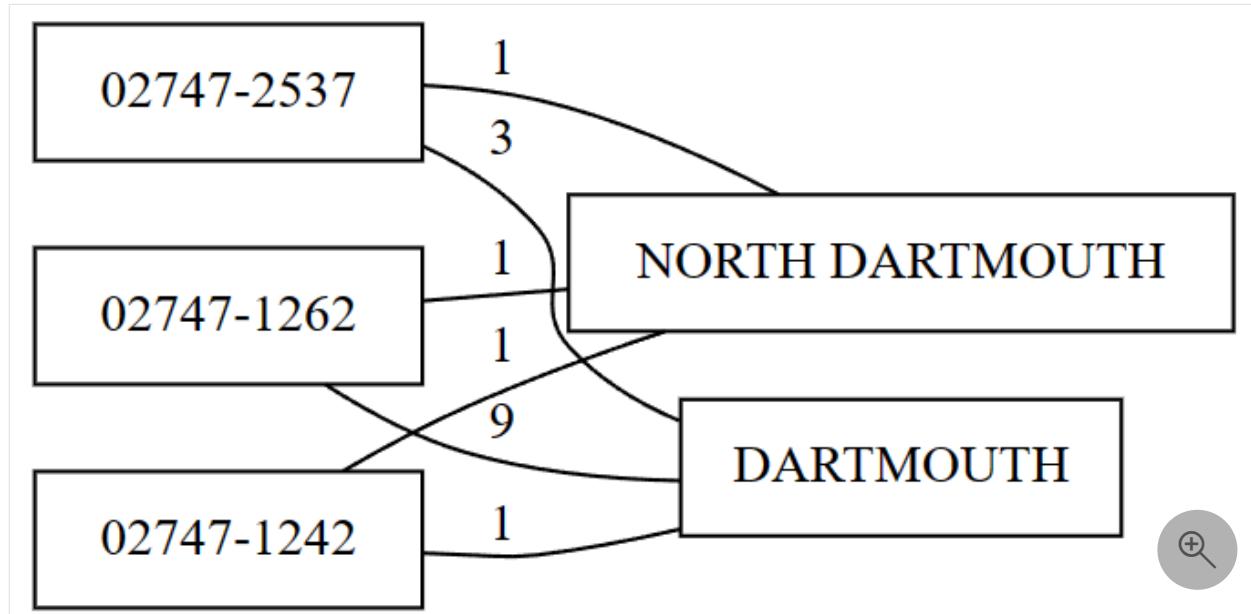
df.plot_dependencyViolations(determinant_col="ZIP", dependent_col="CITY")
```

In this example, the function assumes an existing functional dependency between the ZIP (determinant) and CITY (dependent) columns. If the dataset has data quality issues - for example, the same ZIP code assigned to multiple cities - the function generates a graph of the violating values.

The `plot_dependency_violations` function provides more options that can handle missing values, show values mapped to violating values, limit the number of violations

returned, and sort the results by count or determinant column.

The `plot_dependencyViolations` function generates a visualization that can help identify dataset data quality issues. However, you should carefully examine the results and consider the context of your data, to determine the most appropriate course of action to address the identified issues. This approach might involve more data cleaning, validation, or exploration to ensure the reliability and validity of your analysis or model.



Enforce functional constraints

Data quality is crucial for ensuring the reliability and validity of any analysis or model built on a dataset. Enforcement of functional constraints between columns in a dataset can help improve data quality. Functional constraints can help ensure that the relationships between columns have accuracy and consistency, which can lead to more accurate analysis or model results.

The `drop_dependencyViolations` function can help enforce functional constraints between columns in a dataset. It removes rows that violate a given constraint. Given a determinant column and a dependent column, this function removes rows with values that don't conform to the functional constraint between the two columns.

This code snippet shows how to use the `drop_dependencyViolations` function:

Python

```
from sempy.fabric import FabricDataFrame
from sempy.samples import download_synthea

download_synthea(which='small')
```

```
df = FabricDataFrame(pd.read_csv("synthea/csv/providers.csv"))

cleaned_df = df.drop_dependency_violations(determinant_col="ZIP",
                                            dependent_col="CITY")
```

Here, the function enforces a functional constraint between the ZIP (determinant) and CITY (dependent) columns. For each value of the determinant, the most common value of the dependent is picked, and all rows with other values are dropped. For example, given this dataset, the row with **CITY=Seattle** would be dropped, and the functional dependency **ZIP -> CITY** holds in the output:

[+] Expand table

ZIP	CITY
12345	Seattle
12345	Boston
12345	Boston
98765	Baltimore
00000	San Francisco

The `drop_dependency_violations` function provides the `verbose` option to control the output verbosity. By setting `verbose=1`, you can see the number of dropped rows. A `verbose=2` value shows the entire row content of the dropped rows.

The `drop_dependency_violations` function can enforce functional constraints between columns in your dataset, which can help improve data quality and lead to more accurate results in your analysis or model. However, you must carefully consider the context of your data and the functional constraints you choose to enforce, to ensure that you don't accidentally remove valuable information from your dataset.

Related content

- See the SemPy reference documentation for the `FabricDataFrame` class
- Tutorial: Clean data with functional dependencies
- Explore and validate relationships in semantic models
- Accelerate data science using semantic functions

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Explore and validate relationships in semantic models and DataFrames

Article • 06/18/2024

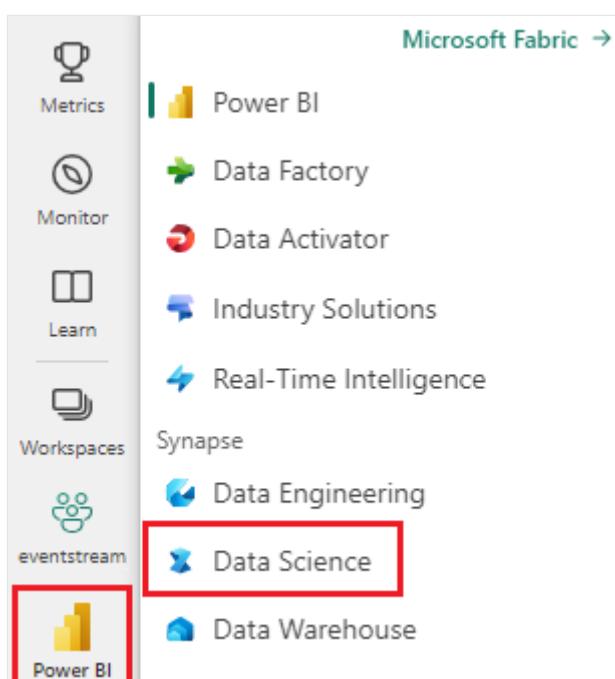
In this article, you learn to use the SemPy semantic link functions to discover and validate relationships in your Power BI semantic models and pandas DataFrames.

In data science and machine learning, it's important to understand the structure and relationships within your data. Power BI is a powerful tool that allows you to model and visualize these structures and relationships. To gain more insights or build machine learning models, you can dive deeper by using the semantic link functions in the SemPy library modules.

Data scientists and business analysts can use SemPy functions to list, visualize, and validate relationships in Power BI semantic models, or find and validate relationships in pandas DataFrames.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



- Create a [new notebook](#) to copy/paste code into cells.
- For Spark 3.4 and above, semantic link is available in the default runtime when using Fabric, and there's no need to install it. For Spark 3.3 or below, or to update to the latest version of semantic link, run the following command:

```
Python
```

```
%pip install -U semantic-link
```

- Add a [lakehouse](#) to your notebook.

List relationships in semantic models

The `list_relationships` function in the `sempy.fabric` module returns a list of all relationships found in a Power BI semantic model. The list helps you understand the structure of your data and how different tables and columns are connected.

This function works by using semantic link to provide annotated DataFrames. The DataFrames include the necessary metadata to understand the relationships within the semantic model. The annotated DataFrames make it easy to analyze the semantic model's structure and use it in machine learning models or other data analysis tasks.

To use the `list_relationships` function, you first import the `sempy.fabric` module. Then you call the function by using the name or UUID of your Power BI semantic model, as shown in the following example:

```
Python
```

```
import sempy.fabric as fabric
fabric.list_relationships("my_dataset")
```

The preceding code calls the `list_relationships` function with a Power BI semantic model called *my_dataset*. The function returns a pandas DataFrame with one row per relationship, allowing you to easily explore and analyze the relationships within the semantic model.

ⓘ Note

Your notebook, Power BI dataset semantic model, and [lakehouse](#) can be located in the same workspace or in different workspaces. By default, SemPy tries to access

your semantic model from:

- The workspace of your lakehouse, if you attached a lakehouse to your notebook.
- The workspace of your notebook, if there's no lakehouse attached.

If your semantic model isn't located in either of these workspaces, you must specify the workspace of your semantic model when you call a SemPy method.

Visualize relationships in semantic models

The `plot_relationship_metadata` function helps you visualize relationships in a semantic model so you can gain a better understanding of the model's structure. This function creates a graph that displays the connections between tables and columns. The graph makes it easier to understand the semantic model's structure and how different elements are related.

The following example shows how to use the `plot_relationship_metadata` function:

Python

```
import sempy.fabric as fabric
from sempy.relationships import plot_relationship_metadata

relationships = fabric.list_relationships("my_dataset")
plot_relationship_metadata(relationships)
```

In the preceding code, the `list_relationships` function retrieves the relationships in the `my_dataset` semantic model, and the `plot_relationship_metadata` function creates a graph to visualize the relationships.

You can customize the graph by defining which columns to include, specifying how to handle missing keys, and providing more [graphviz](#) attributes.

Validate relationships in semantic models

Now that you have a better understanding of the relationships in your semantic model, you can use the `list_relationshipViolations` function to validate these relationships and identify any potential issues or inconsistencies. The `list_relationshipViolations` function helps you validate the content of your tables to ensure that they match the relationships defined in your semantic model.

By using this function, you can identify inconsistencies with the specified relationship multiplicity and address any issues before they impact your data analysis or machine learning models.

To use the `list_relationshipViolations` function, first you import the `sempy.fabric` module and read the tables from your semantic model. Then, you call the function with a dictionary that maps table names to the DataFrames with table content.

The following example code shows how to list relationship violations:

Python

```
import sempy.fabric as fabric

tables = {
    "Sales": fabric.read_table("my_dataset", "Sales"),
    "Products": fabric.read_table("my_dataset", "Products"),
    "Customers": fabric.read_table("my_dataset", "Customers"),
}

fabric.list_relationshipViolations(tables)
```

The preceding code calls the `list_relationshipViolations` function with a dictionary that contains the *Sales*, *Products*, and *Customers* tables from the *my_dataset* semantic model. You can customize the function by setting a coverage threshold, specifying how to handle missing keys, and defining the number of missing keys to report.

The function returns a pandas DataFrame with one row per relationship violation, allowing you to easily identify and address any issues within your semantic model. By using the `list_relationshipViolations` function, you can ensure that your semantic model is consistent and accurate, allowing you to build more reliable machine learning models and gain deeper insights into your data.

Find relationships in pandas DataFrames

While the `list_relationships`, `plot_relationships_df` and `list_relationshipViolations` functions in the Fabric module are powerful tools for exploring relationships within semantic models, you might also need to discover relationships within other data sources imported as pandas DataFrames.

This is where the `find_relationships` function in the `sempy.relationship` module comes into play.

The `find_relationships` function in the `sempy.relationships` module helps data scientists and business analysts discover potential relationships within a list of pandas DataFrames. By using this function, you can identify possible connections between tables and columns, allowing you to better understand the structure of your data and how different elements are related.

The following example code shows how to find relationships in pandas DataFrames:

Python

```
from sempy.relationships import find_relationships

tables = [df_sales, df_products, df_customers]

find_relationships(tables)
```

The preceding code calls the `find_relationships` function with a list of three Pandas DataFrames: `df_sales`, `df_products`, and `df_customers`. The function returns a pandas DataFrame with one row per potential relationship, allowing you to easily explore and analyze the relationships within your data.

You can customize the function by specifying a coverage threshold, a name similarity threshold, a list of relationships to exclude, and whether to include many-to-many relationships.

Validate relationships in pandas DataFrames

After you discover potential relationships in your pandas DataFrames by using the `find_relationships` function, you can use the `list_relationshipViolations` function to validate these relationships and identify any potential issues or inconsistencies.

The `list_relationshipViolations` function validates the content of your tables to ensure that they match the discovered relationships. By using this function to identify inconsistencies with the specified relationship multiplicity, you can address any issues before they impact your data analysis or machine learning models.

The following example code shows how to find relationship violations in pandas DataFrames:

Python

```
from sempy.relationships import find_relationships,
list_relationshipViolations
```

```
tables = [df_sales, df_products, df_customers]
relationships = find_relationships(tables)

list_relationshipViolations(tables, relationships)
```

The preceding code calls the `list_relationship_violations` function with a list of three pandas DataFrames, `df_sales`, `df_products`, and `df_customers`, plus the relationships DataFrame from the `find_relationships` function. The `list_relationship_violations` function returns a pandas DataFrame with one row per relationship violation, allowing you to easily identify and address any issues within your data.

You can customize the function by setting a coverage threshold, specifying how to handle missing keys, and defining the number of missing keys to report.

By using the `list_relationship_violations` function with pandas DataFrames, you can ensure that your data is consistent and accurate, allowing you to build more reliable machine learning models and gain deeper insights into your data.

Related content

- [Learn about semantic functions](#)
- [Get started with the SemPy reference documentation](#)
- [Tutorial: Discover relationships in a semantic model by using semantic link](#)
- [Tutorial: Discover relationships in the Synthea dataset by using semantic link](#)
- [Detect, explore, and validate functional dependencies in your data](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Train machine learning models

Article • 06/14/2024

Apache Spark - a part of Microsoft Fabric - enables machine learning with big data. With Apache Spark, you can build valuable insights into large masses of structured, unstructured, and fast-moving data. You have several available open-source library options when you train machine learning models with Apache Spark in Microsoft Fabric: Apache Spark MLlib, SynapseML, and others.

Apache SparkML and MLlib

Apache Spark - a part of Microsoft Fabric - provides a unified, open-source, parallel data processing framework. This framework supports in-memory processing that boosts big data analytics. The Spark processing engine is built for speed, ease of use, and sophisticated analytics. Spark's in-memory distributed computation capabilities make it a good choice for the iterative algorithms that machine learning and graph computations use.

The **MLlib** and **SparkML** scalable machine learning libraries bring algorithmic modeling capabilities to this distributed environment. MLlib contains the original API, built on top of RDDs. SparkML is a newer package. It provides a higher-level API built on top of DataFrames for construction of ML pipelines. SparkML doesn't yet support all of the features of MLlib, but is replacing MLlib as the standard Spark machine learning library.

ⓘ Note

For more information about SparkML model creation, visit the [Train models with Apache Spark MLlib](#) resource.

Popular libraries

The Microsoft Fabric runtime for Apache Spark includes several popular, open-source packages for training machine learning models. These libraries provide reusable code that you can include in your programs or projects. The runtime includes these relevant machine learning libraries, and others:

- [Scikit-learn](#) - one of the most popular single-node machine learning libraries for classical ML algorithms. Scikit-learn supports most supervised and unsupervised learning algorithms, and can handle data-mining and data-analysis.

- [XGBoost](#) - a popular machine learning library that contains optimized algorithms for training decision trees and random forests.
- [PyTorch](#) and [Tensorflow](#) are powerful Python deep learning libraries. With these libraries, you can set the number of executors on your pool to zero, to build single-machine models. Although that configuration doesn't support Apache Spark, it's a simple, cost-effective way to create single-machine models.

SynapseML

The [SynapseML](#) open-source library (previously known as MMLSpark) simplifies the creation of massively scalable machine learning (ML) pipelines. With it, data scientist use of Spark becomes more productive because that library increases the rate of experimentation and applies cutting-edge machine learning techniques - including deep learning - on large datasets.

SynapseML provides a layer above the SparkML low-level APIs when building scalable ML models. These APIs cover string indexing, feature vector assembly, coercion of data into layouts appropriate for machine learning algorithms, and more. The SynapseML library simplifies these and other common tasks for building models in PySpark.

Related content

This article provides an overview of the various options available to train machine learning models within Apache Spark in Microsoft Fabric. For more information about model training, visit these resources:

- Use AI samples to build machine learning models: [Use AI samples](#)
- Track machine learning runs using Experiments: [Machine learning experiments](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Hyperparameter tuning (preview)

Article • 03/26/2024

Hyperparameter tuning is the process of finding the optimal values for the parameters that are not learned by the machine learning model during training, but rather set by the user before the training process begins. These parameters are commonly referred to as hyperparameters, and examples include the learning rate, the number of hidden layers in a neural network, the regularization strength, and the batch size.

The performance of a machine learning model can be highly sensitive to the choice of hyperparameters, and the optimal set of hyperparameters can vary greatly depending on the specific problem and dataset. Hyperparameter tuning is therefore a critical step in the machine learning pipeline, as it can have a significant impact on the model's accuracy and generalization performance.

In Fabric, data scientists can leverage `FLAML`, a lightweight Python library for efficient automation of machine learning and AI operations, for their hyperparameter tuning requirements. Within Fabric notebooks, users can call `f1aml.tune` for economical hyperparameter tuning.

 **Important**

This feature is in [preview](#).

Tuning workflow

There are three essential steps to use `f1aml.tune` to finish a basic tuning task:

1. Specify the tuning objective with respect to the hyperparameters.
2. Specify a search space of the hyperparameters.
3. Specify tuning constraints, including constraints on the resource budget to do the tuning, constraints on the configurations, or/and constraints on a (or multiple) particular metric(s).

Tuning objective

The first step is to specify your tuning objective. To do this, you should first specify your evaluation procedure with respect to the hyperparameters in a user-defined function `evaluation_function`. The function requires a hyperparameter configuration as input. It

can simply return a metric value in a scalar or return a dictionary of metric name and metric value pairs.

In the example below, we can define an evaluation function with respect to 2 hyperparameters named `x` and `y`.

Python

```
import time

def evaluate_config(config: dict):
    """evaluate a hyperparameter configuration"""
    score = (config["x"] - 85000) ** 2 - config["x"] / config["y"]

    faked_evaluation_cost = config["x"] / 100000
    time.sleep(faked_evaluation_cost)
    # we can return a single float as a score on the input config:
    # return score
    # or, we can return a dictionary that maps metric name to metric value:
    return {"score": score, "evaluation_cost": faked_evaluation_cost,
"constraint_metric": config["x"] * config["y"]}
```

Search space

Next, we will specify the search space of hyperparameters. In the search space, you need to specify valid values for your hyperparameters and how these values are sampled (e.g., from a uniform distribution or a log-uniform distribution). In the example below, we can provide the search space for the hyperparameters `x` and `y`. The valid values for both are integers ranging from [1, 100,000]. These hyperparameters are sampled uniformly in the specified ranges.

Python

```
from flaml import tune

# construct a search space for the hyperparameters x and y.
config_search_space = {
    "x": tune.lograndint(lower=1, upper=100000),
    "y": tune.randint(lower=1, upper=100000)
}

# provide the search space to tune.run
tune.run(..., config=config_search_space, ...)
```

With FLAML, users can customize the domain for a particular hyperparameter. This allows users to specify a *type* and *valid range* to sample parameters from. FLAML

supports the following hyperparameter types: float, integer, and categorical. You can see this example below for commonly used domains:

Python

```
config = {
    # Sample a float uniformly between -5.0 and -1.0
    "uniform": tune.uniform(-5, -1),

    # Sample a float uniformly between 3.2 and 5.4,
    # rounding to increments of 0.2
    "quniform": tune.quniform(3.2, 5.4, 0.2),

    # Sample a float uniformly between 0.0001 and 0.01, while
    # sampling in log space
    "loguniform": tune.loguniform(1e-4, 1e-2),

    # Sample a float uniformly between 0.0001 and 0.1, while
    # sampling in log space and rounding to increments of 0.00005
    "qloguniform": tune.qloguniform(1e-4, 1e-1, 5e-5),

    # Sample a random float from a normal distribution with
    # mean=10 and sd=2
    "randn": tune.randn(10, 2),

    # Sample a random float from a normal distribution with
    # mean=10 and sd=2, rounding to increments of 0.2
    "qrandsn": tune.qrandn(10, 2, 0.2),

    # Sample a integer uniformly between -9 (inclusive) and 15 (exclusive)
    "randint": tune.randint(-9, 15),

    # Sample a random uniformly between -21 (inclusive) and 12 (inclusive
    (!))
    # rounding to increments of 3 (includes 12)
    "qrandint": tune.qrandint(-21, 12, 3),

    # Sample a integer uniformly between 1 (inclusive) and 10 (exclusive),
    # while sampling in log space
    "lograndint": tune.lograndint(1, 10),

    # Sample a integer uniformly between 2 (inclusive) and 10 (inclusive
    (!)),
    # while sampling in log space and rounding to increments of 2
    "qlograndint": tune.qlograndint(2, 10, 2),

    # Sample an option uniformly from the specified choices
    "choice": tune.choice(["a", "b", "c"]),
}
```

To learn more about how you can customize domains within your search space, visit [the FLAML documentation on customizing search spaces](#).

Tuning constraints

The last step is to specify constraints of the tuning task. One notable property of `flaml.tune` is that it is able to complete the tuning process within a required resource constraint. To do this, a user can provide resource constraints in terms of wall-clock time (in seconds) using the `time_budget_s` argument or in terms of the number of trials using the `num_samples` argument.

Python

```
# Set a resource constraint of 60 seconds wall-clock time for the tuning.  
flaml.tune.run(..., time_budget_s=60, ...)  
  
# Set a resource constraint of 100 trials for the tuning.  
flaml.tune.run(..., num_samples=100, ...)  
  
# Use at most 60 seconds and at most 100 trials for the tuning.  
flaml.tune.run(..., time_budget_s=60, num_samples=100, ...)
```

To learn more about addition configuration constraints, you can visit [the FLAML documentation for advanced tuning options](#).

Putting it together

Once we've defined our tuning criteria, we can execute the tuning trial. To track the results of our trial, we can leverage [MLFlow autologging](#) to capture the metrics and parameters for each of these runs. This code will capture the entire hyperparameter tuning trial, highlighting each of the hyperparameter combinations that were explored by FLAML.

Python

```
import mlflow  
mlflow.set_experiment("flaml_tune_experiment")  
mlflow.autolog(exclusive=False)  
  
with mlflow.start_run(nested=True, run_name="Child Run: "):  
    analysis = tune.run(  
        evaluate_config, # the function to evaluate a config  
        config=config_search_space, # the search space defined  
        metric="score",  
        mode="min", # the optimization mode, "min" or "max"  
        num_samples=-1, # the maximal number of configs to try, -1 means
```

```
    infinite
        time_budget_s=10, # the time budget in seconds
    )
```

ⓘ Note

When MLflow autologging is enabled, metrics, parameters and models should be logged automatically as MLFlow runs. However, this varies by the framework.

Metrics and parameters for specific models may not be logged. For example, no metrics will be logged for XGBoost , LightGBM , Spark and SynapseML models. You can learn more about what metrics and parameters are captured from each framework using the [MLFlow autologging documentation](#) ↗ .

Parallel tuning with Apache Spark

The `flaml.tune` functionality supports tuning both Apache Spark and single-node learners. In addition, when tuning single-node learners (e.g. Scikit-Learn learners), you can also parallelize the tuning to speed up your tuning process by setting `use_spark = True`. For Spark clusters, by default, FLAML will launch one trial per executor. You can also customize the number of concurrent trials by using the `n_concurrent_trials` argument.

Python

```
analysis = tune.run(
    evaluate_config, # the function to evaluate a config
    config=config_search_space, # the search space defined
    metric="score",
    mode="min", # the optimization mode, "min" or "max"
    num_samples=-1, # the maximal number of configs to try, -1 means
    infinite
    time_budget_s=10, # the time budget in seconds
    use_spark=True,
)
print(analysis.best_trial.last_result) # the best trial's result
print(analysis.best_config) # the best config
```

To learn more about how to parallelize your tuning trails, you can visit the [FLAML documentation for parallel Spark jobs](#) ↗ .

Visualize results

The `flaml.visualization` module provides utility functions for plotting the optimization process using Plotly. By leveraging Plotly, users can interactively explore their AutoML experiment results. To use these plotting functions, simply provide your optimized `flaml.AutoML` or `flaml.tune.tune.ExperimentAnalysis` object as an input.

You can use the following functions within your notebook:

- `plot_optimization_history`: Plot optimization history of all trials in the experiment.
- `plot_feature_importance`: Plot importance for each feature in the dataset.
- `plot_parallel_coordinate`: Plot the high-dimensional parameter relationships in the experiment.
- `plot_contour`: Plot the parameter relationship as contour plot in the experiment.
- `plot_edf`: Plot the objective value EDF (empirical distribution function) of the experiment.
- `plot_timeline`: Plot the timeline of the experiment.
- `plot_slice`: Plot the parameter relationship as slice plot in a study.
- `plot_param_importance`: Plot the hyperparameter importance of the experiment.

Next steps

- [Tune a SynapseML Spark LightGBM model](#)
 - [Visualize AutoML results](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Build a machine learning model with Apache Spark MLlib

Article • 06/19/2024

In this article, you learn how to use Apache Spark [MLlib](#) to create a machine learning application that handles simple predictive analysis on an Azure open dataset. Spark provides built-in machine learning libraries. This example uses *classification* through logistic regression.

The core SparkML and MLlib Spark libraries provide many utilities that are useful for machine learning tasks. These utilities are suitable for:

- Classification
- Clustering
- Hypothesis testing and calculating sample statistics
- Regression
- Singular value decomposition (SVD) and principal component analysis (PCA)
- Topic modeling

Understand classification and logistic regression

Classification, a popular machine learning task, involves sorting input data into categories. A classification algorithm should figure out how to assign *labels* to the supplied input data. For example, a machine learning algorithm could accept stock information as input, and divide the stock into two categories: stocks that you should sell and stocks that you should keep.

The *Logistic regression* algorithm is useful for classification. The Spark logistic regression API is useful for *binary classification* of input data into one of two groups. For more information about logistic regression, see [Wikipedia](#).

Logistic regression produces a *logistic function* that can predict the probability that an input vector belongs to one group or the other.

Predictive analysis example of NYC taxi data

First, install `azureml-opendatasets`. The data is available through the [Azure Open Datasets](#) resource. This dataset subset hosts information about yellow taxi trips,

including the start times, end times, start locations, end locations, trip costs, and other attributes.

Python

```
%pip install azureml-opendatasets
```

The rest of this article relies on Apache Spark to first perform some analysis on the NYC taxi-trip tip data and then develop a model to predict whether a particular trip includes a tip or not.

Create an Apache Spark machine learning model

1. Create a PySpark notebook. For more information, visit [Create a notebook](#).
2. Import the types required for this notebook.

Python

```
import matplotlib.pyplot as plt
from datetime import datetime
from dateutil import parser
from pyspark.sql.functions import unix_timestamp, date_format, col,
when
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel
from pyspark.ml.feature import RFormula
from pyspark.ml.feature import OneHotEncoder, StringIndexer,
VectorIndexer
from pyspark.ml.classification import LogisticRegression
from pyspark.mllib.evaluation import BinaryClassificationMetrics
from pyspark.ml.evaluation import BinaryClassificationEvaluator
```

3. We'll use [MLflow](#) to track our machine learning experiments and corresponding runs. If Microsoft Fabric Autologging is enabled, the corresponding metrics and parameters are automatically captured.

Python

```
import mlflow
```

Construct the input DataFrame

This example loads the data into a Pandas dataframe, and then converts it into an Apache Spark dataframe. In that format, we can apply other Apache Spark operations to clean and filter the dataset.

1. Paste these lines into a new cell, and run them to create a Spark DataFrame. This step retrieves the data via the Open Datasets API. We can filter this data down to examine a specific window of data. The code example uses `start_date` and `end_date` to apply a filter that returns a single month of data.

Python

```
from azureml.opendatasets import NycTlcYellow

end_date = parser.parse('2018-06-06')
start_date = parser.parse('2018-05-01')
nyc_tlc = NycTlcYellow(start_date=start_date, end_date=end_date)
nyc_tlc_pd = nyc_tlc.to_pandas_dataframe()

nyc_tlc_df = spark.createDataFrame(nyc_tlc_pd).repartition(20)
```

2. This code reduces the dataset to about 10,000 rows. To speed up the development and training, the code samples down our dataset for now.

Python

```
# To make development easier, faster, and less expensive, sample down
# for now
sampled_taxi_df = nyc_tlc_df.sample(True, 0.001, seed=1234)
```

3. We want to look at our data using the built-in `display()` command. With this command, we can easily view a data sample, or graphically explore trends in the data.

Python

```
#sampled_taxi_df.show(10)
display(sampled_taxi_df.limit(10))
```

Prepare the data

Data preparation is a crucial step in the machine learning process. It involves cleaning, transformation, and organization of raw data, to make it suitable for analysis and modeling. In this code sample, you perform several data preparation steps:

- Filter the dataset to remove outliers and incorrect values
- Remove columns that aren't needed for model training
- Create new columns from the raw data
- Generate a label to determine whether or not a given Taxi trip involves a tip

Python

```
taxi_df = sampled_taxi_df.select('totalAmount', 'fareAmount', 'tipAmount',
'paymentType', 'rateCodeId', 'passengerCount' \
, 'tripDistance', 'tpepPickupDateTime',
'tpepDropoffDateTime' \
, date_format('tpepPickupDateTime',
'hh').alias('pickupHour') \
, date_format('tpepPickupDateTime',
'EEEE').alias('weekdayString') \
, (unix_timestamp(col('tpepDropoffDateTime')) -
unix_timestamp(col('tpepPickupDateTime'))).alias('tripTimeSecs') \
, (when(col('tipAmount') > 0,
1).otherwise(0)).alias('tipped') \
) \
.filter((sampled_taxi_df.passengerCount > 0) &
(sampled_taxi_df.passengerCount < 8) \
& (sampled_taxi_df.tipAmount >= 0) &
(sampled_taxi_df.tipAmount <= 25) \
& (sampled_taxi_df.fareAmount >= 1) &
(sampled_taxi_df.fareAmount <= 250) \
& (sampled_taxi_df.tipAmount <
sampled_taxi_df.fareAmount) \
& (sampled_taxi_df.tripDistance > 0) &
(sampled_taxi_df.tripDistance <= 100) \
& (sampled_taxi_df.rateCodeId <= 5) \
& (sampled_taxi_df.paymentType.isin({"1", "2"})) \
)
```

Next, make a second pass over the data to add the final features.

Python

```
taxi_featurised_df = taxi_df.select('totalAmount', 'fareAmount',
'tipAmount', 'paymentType', 'passengerCount' \
, 'tripDistance',
'weekdayString', 'pickupHour', 'tripTimeSecs', 'tipped' \
, when((taxi_df.pickupHour \
<= 6) | (taxi_df.pickupHour >= 20), "Night") \
.when((taxi_df.pickupHour >= 7) & (taxi_df.pickupHour <= 10), "AMRush") \
.when((taxi_df.pickupHour >= 11) & (taxi_df.pickupHour <= 15), "Afternoon") \
.when((taxi_df.pickupHour >= 16) & (taxi_df.pickupHour <= 19), "PMRush") \
```

```
.otherwise(0).alias('trafficTimeBins')
    )\filter((taxi_df.tripTimeSecs >= 30)
& (taxi_df.tripTimeSecs <= 7200))
```

Create a logistic regression model

The final task converts the labeled data into a format that logistic regression can handle. The input to a logistic regression algorithm must have a *label/feature vector pairs* structure, where the *feature vector* is a vector of numbers that represent the input point.

Based on the final task requirements, we must convert the categorical columns into numbers. Specifically, we must convert the `trafficTimeBins` and `weekdayString` columns into integer representations. We have many options available to handle this requirement. This example involves the `OneHotEncoder` approach:

Python

```
# Because the sample uses an algorithm that works only with numeric
# features, convert them so they can be consumed
sI1 = StringIndexer(inputCol="trafficTimeBins",
outputCol="trafficTimeBinsIndex")
en1 = OneHotEncoder(dropLast=False, inputCol="trafficTimeBinsIndex",
outputCol="trafficTimeBinsVec")
sI2 = StringIndexer(inputCol="weekdayString", outputCol="weekdayIndex")
en2 = OneHotEncoder(dropLast=False, inputCol="weekdayIndex",
outputCol="weekdayVec")

# Create a new DataFrame that has had the encodings applied
encoded_final_df = Pipeline(stages=[sI1, en1, sI2,
en2]).fit(taxi_featurised_df).transform(taxi_featurised_df)
```

This action results in a new DataFrame with all columns in the proper format to train a model.

Train a logistic regression model

The first task splits the dataset into a training set, and a testing or validation set.

Python

```
# Decide on the split between training and test data from the DataFrame
trainingFraction = 0.7
testingFraction = (1-trainingFraction)
seed = 1234
```

```
# Split the DataFrame into test and training DataFrames
train_data_df, test_data_df =
encoded_final_df.randomSplit([trainingFraction, testingFraction], seed=seed)
```

Once we have two DataFrames, we must create the model formula and run it against the training DataFrame. Then we can validate against the test DataFrame. Experiment with different versions of the model formula to see the effects of different combinations.

Python

```
## Create a new logistic regression object for the model
logReg = LogisticRegression(maxIter=10, regParam=0.3, labelCol = 'tipped')

## The formula for the model
classFormula = RFormula(formula="tipped ~ pickupHour + weekdayVec +
passengerCount + tripTimeSecs + tripDistance + fareAmount + paymentType+
trafficTimeBinsVec")

## Undertake training and create a logistic regression model
lrModel = Pipeline(stages=[classFormula, logReg]).fit(train_data_df)

## Predict tip 1/0 (yes/no) on the test dataset; evaluation using area under
ROC
predictions = lrModel.transform(test_data_df)
predictionAndLabels = predictions.select("label","prediction").rdd
metrics = BinaryClassificationMetrics(predictionAndLabels)
print("Area under ROC = %s" % metrics.areaUnderROC)
```

The cell outputs:

shell

```
Area under ROC = 0.9749430523917996
```

Create a visual representation of the prediction

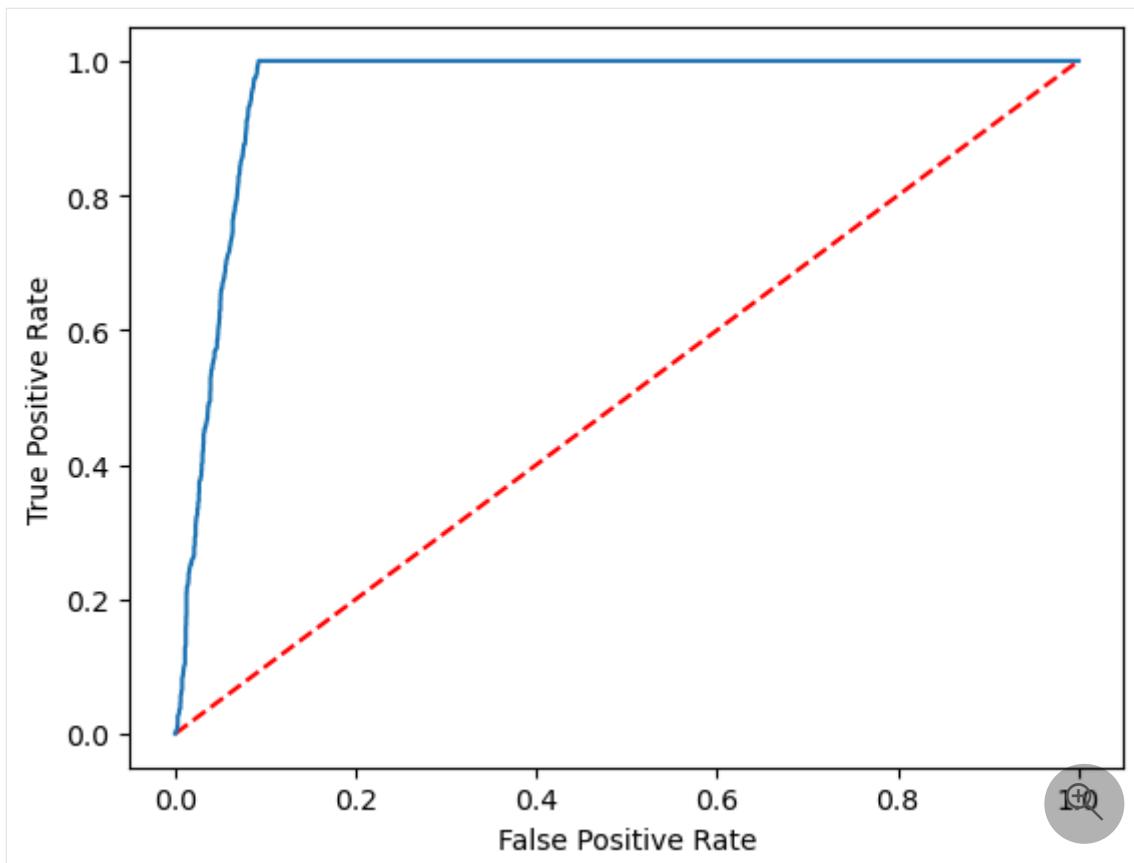
We can now build a final visualization to interpret the model results. A [ROC curve](#) can certainly present the result.

Python

```
## Plot the ROC curve; no need for pandas, because this uses the
modelSummary object
modelSummary = lrModel.stages[-1].summary

plt.plot([0, 1], [0, 1], 'r--')
plt.plot(modelSummary.roc.select('FPR').collect(),
```

```
modelSummary.roc.select('TPR').collect())
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```



Related content

- Use AI samples to build machine learning models: [Use AI samples](#)
- Track machine learning runs using Experiments: [Machine learning experiments](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback](#) | [Ask the community](#)

Train models with scikit-learn in Microsoft Fabric

Article • 07/19/2024

This article describes how to train and track the iterations of a scikit-learn model. [Scikit-learn](#) is a popular open-source machine learning framework frequently used for supervised and unsupervised learning. The framework provides tools for model fitting, data preprocessing, model selection, model evaluation, and more.

Prerequisites

Install scikit-learn within your notebook. You can install or upgrade the version of scikit-learn on your environment by using the following command:

shell

```
pip install scikit-learn
```

Set up the machine learning experiment

You can create a machine learning experiment by using the MLflow API. The MLflow `set_experiment()` function creates a new machine learning experiment named *sample-sklearn*, if it doesn't already exist.

Run the following code in your notebook and create the experiment:

Python

```
import mlflow

mlflow.set_experiment("sample-sklearn")
```

Train a scikit-learn model

After you set up the experiment, you create a sample dataset and a logistic regression model. The following code starts an MLflow run, and tracks the metrics, parameters, and final logistic regression model. After you generate the final model, you can save the resulting model for more tracking.

Run the following code in your notebook and create the sample dataset and logistic regression model:

```
Python

import mlflow.sklearn
import numpy as np
from sklearn.linear_model import LogisticRegression
from mlflow.models.signature import infer_signature

with mlflow.start_run() as run:

    lr = LogisticRegression()
    X = np.array([-2, -1, 0, 1, 2, 1]).reshape(-1, 1)
    y = np.array([0, 0, 1, 1, 1, 0])
    lr.fit(X, y)
    score = lr.score(X, y)
    signature = infer_signature(X, y)

    print("log_metric.")
    mlflow.log_metric("score", score)

    print("log_params.")
    mlflow.log_param("alpha", "alpha")

    print("log_model.")
    mlflow.sklearn.log_model(lr, "sklearn-model", signature=signature)
    print("Model saved in run_id=%s" % run.info.run_id)

    print("register_model.")
    mlflow.register_model(
        "runs:/{}sklearn-model".format(run.info.run_id), "sample-sklearn"
    )
    print("All done")
```

Load and evaluate the model on a sample dataset

After you save the model, you can load it for inferencing.

Run the following code in your notebook and load the model, and then run the inference on a sample dataset:

```
Python

# Inference with loading the logged model
from synapse.ml.predict import MLflowTransformer
```

```
spark.conf.set("spark.synapse.ml.predict.enabled", "true")

model = MLflowTransformer(
    inputCols=["x"],
    outputCol="prediction",
    modelName="sample-sklearn",
    modelVersion=1,
)

test_spark = spark.createDataFrame(
    data=np.array([-2, -1, 0, 1, 2, 1]).reshape(-1, 1).tolist(), schema=
    ["x"]
)

batch_predictions = model.transform(test_spark)

batch_predictions.show()
```

Related content

- Explore [machine learning models](#)
- Create [machine learning experiments](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

How to train models with SynapseML

Article • 04/14/2025

The [SynapseML](#) tool ecosystem expands the Apache Spark distributed computing framework in several new spaces. SynapseML adds many deep learning and data science tools to the Spark ecosystem:

- Seamless integration of Spark Machine Learning pipelines with Microsoft Cognitive Toolkit (CNTK)
- LightGBM
- OpenCV

These tools make possible powerful and highly scalable predictive and analytical models, for many types of datasources.

This section describes how to train your SynapseML model.

Prerequisites

Import numpy and pandas:

Python

```
import numpy as np
import pandas as pd
```

Read in data

A typical Spark application involves huge datasets stored on a distributed file system - for example, HDFS. However, to simplify things here, copy over a small dataset from a URL. Then, read this data into memory with the Pandas CSV reader, and distribute the data as a Spark DataFrame. Finally, show the first five rows of the dataset:

Python

```
dataFile = "AdultCensusIncome.csv"
import os, urllib
if not os.path.isfile(dataFile):
    urllib.request.urlretrieve("https://mmlspark.azureedge.net/datasets/" +
dataFile, dataFile)
data = spark.createDataFrame(pd.read_csv(dataFile, dtype={"hours-per-week": np.float64}))
data.show(5)
```

Select features and split data, to train and test sets

Select some features to use in our model. You can try out different features, but you should include `" income"` because it's the label column the model tries to predict. Then split the data into `train` and `test` sets:

Python

```
data = data.select([" education", " marital-status", " hours-per-week", "  
income"])  
train, test = data.randomSplit([0.75, 0.25], seed=123)
```

Train a model

To train the classifier model, use the `synapse.ml.TrainClassifier` class. It takes in training data and a base SparkML classifier, maps the data into the format the base classifier algorithm expects, and fits a model:

Python

```
from synapse.ml.train import TrainClassifier  
from pyspark.ml.classification import LogisticRegression  
model = TrainClassifier(model=LogisticRegression(), labelCol=" income").fit(train)
```

`TrainClassifier` implicitly handles string-valued columns and binarizes the label column.

Score and evaluate the model

Finally, score the model against the test set, and use the `synapse.ml.ComputeModelStatistics` class to compute:

- Accuracy
- AUC
- Precision
- Recall

Metrics from the scored data:

Python

```
from synapse.ml.train import ComputeModelStatistics  
prediction = model.transform(test)
```

```
metrics = ComputeModelStatistics().transform(prediction)
metrics.select('accuracy').show()
```

That's it! You built your first machine learning model with the SynapseML package. Use the Python `help()` function for more information about SynapseML classes and methods:

Python

```
import synapse.ml.train.TrainClassifier
help(synapse.ml.train.TrainClassifier)
```

Related content

- [Explore and validate relationships in semantic models \(preview\)](#)
- [Track models with MLflow](#)

Train models with PyTorch in Microsoft Fabric

Article • 07/18/2024

This article describes how to train and track the iterations of a PyTorch model. The [PyTorch](#) machine learning framework is based on the Torch library. PyTorch is often used for computer vision and natural language processing applications.

Prerequisites

Install PyTorch and torchvision within your notebook. You can install or upgrade the version of these libraries on your environment by using the following command:

shell

```
pip install torch torchvision
```

Set up the machine learning experiment

You can create a machine learning experiment by using the MLflow API. The MLflow `set_experiment()` function creates a new machine learning experiment named *sample-pytorch*, if it doesn't already exist.

Run the following code in your notebook and create the experiment:

Python

```
import mlflow

mlflow.set_experiment("sample-pytorch")
```

Train and evaluate a Pytorch model

After you set up the experiment, you load the Modified National Institute of Standards and Technology (MNIST) dataset. You generate the test and training datasets, and then create a training function.

Run the following code in your notebook and train the Pytorch model:

Python

```
import os
import torch
import torch.nn as nn
from torch.autograd import Variable
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torch.nn.functional as F
import torch.optim as optim

# Load the MNIST dataset
root = "/tmp/mnist"
if not os.path.exists(root):
    os.mkdir(root)

trans = transforms.Compose(
    [transforms.ToTensor(), transforms.Normalize((0.5,), (1.0,))])
)

# If the data doesn't exist, download the MNIST dataset
train_set = dset.MNIST(root=root, train=True, transform=trans,
download=True)
test_set = dset.MNIST(root=root, train=False, transform=trans,
download=True)

batch_size = 100

train_loader = torch.utils.data.DataLoader(
    dataset=train_set, batch_size=batch_size, shuffle=True
)
test_loader = torch.utils.data.DataLoader(
    dataset=test_set, batch_size=batch_size, shuffle=False
)

print("==>> total training batch number: {}".format(len(train_loader)))
print("==>> total testing batch number: {}".format(len(test_loader)))

# Define the network
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5, 1)
        self.conv2 = nn.Conv2d(20, 50, 5, 1)
        self.fc1 = nn.Linear(4 * 4 * 50, 500)
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 4 * 4 * 50)
        x = F.relu(self.fc1(x))
```

```

        x = self.fc2(x)
        return x

    def name(self):
        return "LeNet"

# Train the model
model = LeNet()

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

criterion = nn.CrossEntropyLoss()

for epoch in range(1):
    # Model training
    ave_loss = 0
    for batch_idx, (x, target) in enumerate(train_loader):
        optimizer.zero_grad()
        x, target = Variable(x), Variable(target)
        out = model(x)
        loss = criterion(out, target)
        ave_loss = (ave_loss * batch_idx + loss.item()) / (batch_idx + 1)
        loss.backward()
        optimizer.step()
        if (batch_idx + 1) % 100 == 0 or (batch_idx + 1) ==
len(train_loader):
            print(
                "=>> epoch: {}, batch index: {}, train loss: {:.6f}"
                .format(
                    epoch, batch_idx + 1, ave_loss
                )
            )
    # Model testing
    correct_cnt, total_cnt, ave_loss = 0, 0, 0
    for batch_idx, (x, target) in enumerate(test_loader):
        x, target = Variable(x, volatile=True), Variable(target,
volatile=True)
        out = model(x)
        loss = criterion(out, target)
        _, pred_label = torch.max(out.data, 1)
        total_cnt += x.data.size()[0]
        correct_cnt += (pred_label == target.data).sum()
        ave_loss = (ave_loss * batch_idx + loss.item()) / (batch_idx + 1)

        if (batch_idx + 1) % 100 == 0 or (batch_idx + 1) ==
len(test_loader):
            print(
                "=>> epoch: {}, batch index: {}, test loss: {:.6f}, acc: {:.3f}"
                .format(
                    epoch, batch_idx + 1, ave_loss, correct_cnt * 1.0 /
total_cnt
                )
            )
torch.save(model.state_dict(), model.name())

```

Log model with MLflow

The next task starts an MLflow run and tracks the results within the machine learning experiment. The sample code creates a new model named `sample-pytorch`. It creates a run with the specified parameters, and logs the run within the `sample-pytorch` experiment.

Run the following code in your notebook and log the model:

Python

```
with mlflow.start_run() as run:
    print("log pytorch model:")
    mlflow.pytorch.log_model(
        model, "pytorch-model", registered_model_name="sample-pytorch"
    )

    model_uri = "runs:/{}{}".format(run.info.run_id)
    print("Model saved in run {}".format(run.info.run_id))
    print(f"Model URI: {model_uri}")
```

Load and evaluate the model

After you save the model, you can load it for inferencing.

Run the following code in your notebook and load the model for inferencing:

Python

```
# Inference with loading the logged model
loaded_model = mlflow.pytorch.load_model(model_uri)
print(type(loaded_model))

correct_cnt, total_cnt, ave_loss = 0, 0, 0
for batch_idx, (x, target) in enumerate(test_loader):
    x, target = Variable(x, volatile=True), Variable(target, volatile=True)
    out = loaded_model(x)
    loss = criterion(out, target)
    _, pred_label = torch.max(out.data, 1)
    total_cnt += x.data.size()[0]
    correct_cnt += (pred_label == target.data).sum()
    ave_loss = (ave_loss * batch_idx + loss.item()) / (batch_idx + 1)

    if (batch_idx + 1) % 100 == 0 or (batch_idx + 1) == len(test_loader):
        print(
            "====>> epoch: {}, batch index: {}, test loss: {:.6f}, acc: {}"
        )
```

```
{:.3f}".format(  
    epoch, batch_idx + 1, ave_loss, correct_cnt * 1.0 /  
    total_cnt  
)  
)
```

Related content

- Explore [machine learning models](#)
- Create [machine learning experiments](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Train explainable boosting machines - classification (preview)

Article • 11/15/2023

In this article, you learn how to train classification models using explainable boosting machines (EBM). An explainable boosting machine is a machine learning technique that combines the power of gradient boosting with an emphasis on model interpretability. It creates an ensemble of decision trees, similar to gradient boosting, but with a unique focus on generating human-readable models. EBMs not only provide accurate predictions but also offer clear and intuitive explanations for those predictions. They're well-suited for applications where understanding the underlying factors driving model decisions is essential, such as healthcare, finance, and regulatory compliance.

In SynapseML, you can use a scalable implementation of an EBM, powered by Apache Spark, for training new models. This article guides you through the process of applying the scalability and interpretability of EBMs within Microsoft Fabric by utilizing Apache Spark.

Important

This feature is in [preview](#).

In this article, you walk through the process of acquiring and preprocessing data from Azure Open Datasets that records NYC Yellow Taxi trips. Then you train a predictive model with the ultimate objective of determining whether a given trip will occur or not.

Benefits of explainable boosting machines

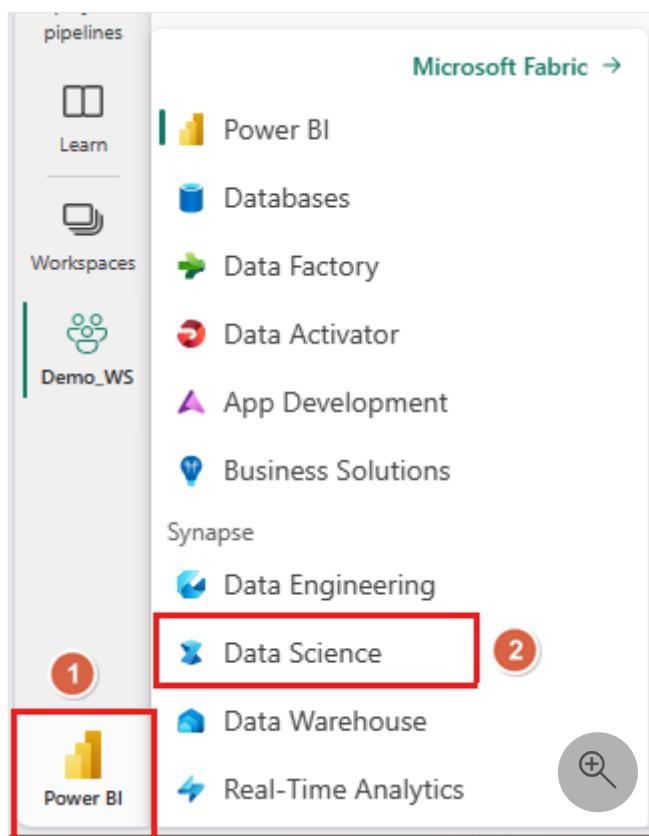
EBMs offer a unique blend of interpretability and predictive power, making them an ideal choice when transparency and comprehensibility of machine learning models are crucial. With EBMs, users can gain valuable insights into the underlying factors driving predictions, enabling them to understand why a model makes specific decisions or predictions, which is essential for building trust in AI systems.

Their ability to uncover complex relationships within the data while providing clear and interpretable results makes them invaluable in fields like finance, healthcare, and fraud detection. In these fields, model explainability isn't only desirable but often a regulatory requirement. Ultimately, users opting for EBMs can strike a balance between model

performance and transparency, ensuring that AI solutions are accurate, easily understandable, and accountable.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Switch to the Data Science experience by using the experience switcher icon on the left side of your home page.



- Create a new notebook in your workspace by selecting + and then **Notebook**.

Install libraries

To begin, install the Azure Machine Learning Open Datasets library, which grants access to the dataset. This installation step is essential for accessing and utilizing the dataset effectively.

```
Python
```

```
%pip install azureml-opendatasets
```

Import MLflow

MLflow allows you to track the model's parameters and outcomes. The following code snippet demonstrates how to use MLflow for experimentation and tracking purposes. The `ebm_classification_nyc_taxi` value is the name of the experiment where the information is logged.

Python

```
import mlflow

# Set up the experiment name
mlflow.set_experiment("ebm_classification_nyc_taxi")
```

Import libraries

Next, import the essential packages used for analysis:

Python

```
# Import necessary packages
import interpret
from pyspark.ml.feature import StringIndexer, VectorAssembler
from pyspark.ml import Pipeline
from pyspark.mllib.evaluation import BinaryClassificationMetrics
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import DoubleType
from synapse.ml.ebm import EbmClassification
```

These imported packages provide the fundamental tools and functionalities needed for your analysis and modeling tasks.

Load data

Next, retrieve the NYC Yellow Taxi data from Azure Machine Learning Open Datasets and optionally down-sample the dataset to expedite development:

Python

```
# Import NYC Yellow Taxi data from Azure Open Datasets
from azureml.opendatasets import NycTlcYellow
```

```
from datetime import datetime
from dateutil import parser

# Define the start and end dates for the dataset
end_date = parser.parse('2018-05-08 00:00:00')
start_date = parser.parse('2018-05-01 00:00:00')

# Load the NYC Yellow Taxi dataset within the specified date range
nyc_tlc = NycTlcYellow(start_date=start_date, end_date=end_date)
nyc_pd = nyc_tlc.to_pandas_dataframe()
nyc_tlc_df = spark.createDataFrame(nyc_pd)
```

To facilitate development and reduce computational overhead, you might down-sample the dataset:

Python

```
# For development convenience, consider down-sampling the dataset
sampled_taxi_df = nyc_tlc_df.sample(True, 0.001, seed=1234)
```

Down-sampling allows for a faster and more cost-effective development experience, particularly when working with large datasets.

Prepare the data

In this section, you prepare the dataset by filtering out rows with outliers and irrelevant variables.

Generate features

Create new derived features that are expected to enhance model performance:

Python

```
taxi_df = sampled_taxi_df.select('totalAmount', 'fareAmount', 'tipAmount',
'paymentType', 'rateCodeId', 'passengerCount',
'tripDistance', 'tpepPickupDateTime',
'tpepDropoffDateTime',
date_format('tpepPickupDateTime',
'hh').alias('pickupHour'),
date_format('tpepPickupDateTime',
'EEEE').alias('weekdayString'),
(unix_timestamp(col('tpepDropoffDateTime')) -
 unix_timestamp(col('tpepPickupDateTime'))).alias('tripTimeSecs'),
(when(col('tipAmount') > 0,
```

```

1).otherwise(0)).alias('tipped')
    )\

        .filter((sampled_taxi_df.passengerCount > 0) &
(sampled_taxi_df.passengerCount < 8)
            & (sampled_taxi_df.tipAmount >= 0) &
(sampled_taxi_df.tipAmount <= 25)
            & (sampled_taxi_df.fareAmount >= 1) &
(sampled_taxi_df.fareAmount <= 250)
            & (sampled_taxi_df.tipAmount <
sampled_taxi_df.fareAmount)
            & (sampled_taxi_df.tripDistance > 0) &
(sampled_taxi_df.tripDistance <= 100)
            & (sampled_taxi_df.rateCodeId <= 5)
            & (sampled_taxi_df.paymentType.isin({"1",
"2"})))
    )

```

Now that you've created new variables, drop the columns from which they were derived. The dataframe is now more streamlined for model input. Additionally, you can generate further features based on the newly created columns:

Python

```

taxi_featurised_df = taxi_df.select('totalAmount', 'fareAmount',
'tipAmount', 'paymentType', 'passengerCount',
                    'tripDistance', 'weekdayString',
'pickupHour', 'tripTimeSecs', 'tipped',
                    when((taxi_df.pickupHour <= 6) |
(taxi_df.pickupHour >= 20), "Night")
                    .when((taxi_df.pickupHour >= 7) &
(taxi_df.pickupHour <= 10), "AMRush")
                    .when((taxi_df.pickupHour >= 11) &
(taxi_df.pickupHour <= 15), "Afternoon")
                    .when((taxi_df.pickupHour >= 16) &
(taxi_df.pickupHour <= 19), "PMRush")
                    .otherwise(0).alias('trafficTimeBins'))\
.filter((taxi_df.tripTimeSecs >= 30) &
(taxi_df.tripTimeSecs <= 7200))

```

These data preparation steps ensure that your dataset is both refined and optimized for subsequent modeling processes.

Encode the data

In the context of SynapseML EBMs, the `Estimator` expects input data to be in the form of an `org.apache.spark.mllib.linalg.Vector`, essentially a vector of `Doubles`. So, it's necessary to convert any categorical (string) variables into numerical representations,

and any variables that possess numeric values but non-numeric data types must be cast into numeric data types.

Presently, SynapseML EBMs employ the `StringIndexer` approach to manage categorical variables. Currently, SynapseML EBMs don't offer specialized handling for categorical features.

Here's a series of steps to prepare the data for use with SynapseML EBMs:

Python

```
# Convert categorical features into numerical representations using
StringIndexer
sI1 = StringIndexer(inputCol="trafficTimeBins",
outputCol="trafficTimeBinsIndex")
sI2 = StringIndexer(inputCol="weekdayString", outputCol="weekdayIndex")

# Apply the encodings to create a new dataframe
encoded_df = Pipeline(stages=[sI1,
sI2]).fit(taxi_featurised_df).transform(taxi_featurised_df)
final_df = encoded_df.select('fareAmount', 'paymentType', 'passengerCount',
'tripDistance',
'pickupHour', 'tripTimeSecs',
'trafficTimeBinsIndex', 'weekdayIndex',
'tipped')

# Ensure that any string representations of numbers in the data are cast to
numeric data types
final_df = final_df.withColumn('paymentType',
final_df['paymentType'].cast(DoubleType()))
final_df = final_df.withColumn('pickupHour',
final_df['pickupHour'].cast(DoubleType()))

# Define the label column name
labelColumnName = 'tipped'

# Assemble the features into a vector
assembler = VectorAssembler(outputCol='features')
assembler.setInputCols([c for c in final_df.columns if c !=
labelColumnName])
vectorized_final_df = assembler.transform(final_df)
```

These data preparation steps are crucial for aligning the data format with the requirements of SynapseML EBMs, ensuring accurate and effective model training.

Generate test and training datasets

Now divide the dataset into training and testing sets using a straightforward split. 70% of the data is allocated for training and 30% for testing the model:

Python

```
# Decide on the split between training and testing data from the dataframe
trainingFraction = 0.7
testingFraction = (1-trainingFraction)
seed = 1234

# Split the dataframe into test and training dataframes
train_data_df, test_data_df =
vectorized_final_df.randomSplit([trainingFraction, testingFraction],
seed=seed)
```

This division of data allows us to train the model on a substantial portion while reserving another portion to assess its performance effectively.

Train the model

Now, train the EBM model and then evaluate its performance using the Area under the Receiver Operating Characteristic (ROC) curve (AUROC) as the metric:

Python

```
# Create an instance of the EBMClassifier
estimator = EbmClassification()
estimator.setLabelCol(labelColumnName)

# Fit the EBM model to the training data
model = estimator.fit(train_data_df)

# Make predictions for tip (1/0 - yes/no) on the test dataset and evaluate
# using AUROC
predictions = model.transform(test_data_df)
predictionsAndLabels = predictions.select("prediction", labelColumnName)
predictionsAndLabels = predictionsAndLabels.withColumn(labelColumnName,
predictionsAndLabels[labelColumnName].cast(DoubleType()))

# Calculate AUROC using BinaryClassificationMetrics
metrics = BinaryClassificationMetrics(predictionsAndLabels.rdd)
print("Area under ROC = %s" % metrics.areaUnderROC)
```

This process entails training the EBM model on the training dataset and then utilizing it to make predictions on the test dataset, followed by assessing the model's performance using AUROC as a key metric.

View global explanations

To visualize the model's overall explanation, you can obtain the visualization wrapper and utilize the `interpret` library's `show` method. The visualization wrapper acts as a bridge to facilitate the visualization experience of the model. Here's how you can do it:

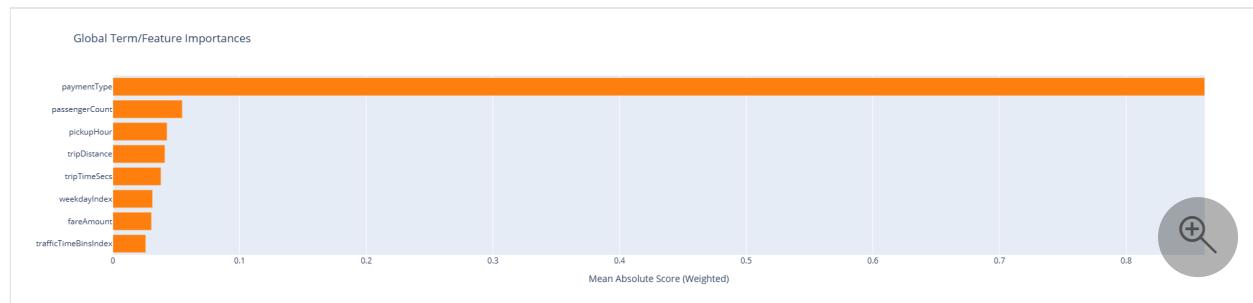
Python

```
wrapper = model.getVizWrapper()  
explanation = wrapper.explain_global()
```

Next, import the `interpret` library and use the `show` method to display the explanation:

Python

```
import interpret  
interpret.show(explanation)
```



The term "importances" represents the mean absolute contribution (score) of each term (feature or interaction) towards predictions. These contributions are averaged across the training dataset, taking into account the number of samples in each bin and sample weights (if applicable). The top 15 most important terms are displayed in the explanation.

View local explanations

The provided explanations are at a global level, but there are scenarios where per-feature outputs are also valuable. Both the trainer and the model offer the capability to set the `featurescores`, which, when populated, introduces another vector-valued column. Each vector within this column matches the length of the feature column, with each value corresponding to the feature at the same index. These values represent the contribution of each feature's value to the final output of the model.

Unlike global explanations, there's currently no direct integration with the `interpret` visualization for per-feature outputs. This is because global visualizations scale primarily

with the number of features (which is typically small), while local explanations scale with the number of rows (which, if a Spark dataframe, can be substantial).

Here's how to set up and use the `featurescores`:

Python

```
prediction2 =  
model.setFeatureScoresCol("featurescores").transform(train_data_df)
```

For illustration purposes, let's print the details of the first example:

Python

```
# Convert to Pandas for easier inspection  
predictions_pandas = prediction2.toPandas()  
predictions_list = prediction2.collect()  
  
# Extract the first example from the collected predictions.  
first = predictions_list[0]  
  
# Print the lengths of the features and feature scores.  
print('Length of the features is', len(first['features']), 'while the  
feature scores have length', len(first['featurescores']))  
  
# Print the values of the features and feature scores.  
print('Features are', first['features'])  
print('Feature scores are', first['featurescores'])
```

Next steps

- InterpretML explainable boosting machine: How it Works ↗
- Track models with MLflow

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Train explainable boosting machines - regression (preview)

Article • 04/17/2025

This article explains how to use explainable boosting machines (EBM) in Microsoft Fabric to train regression models. An EBM is a machine learning technique that combines the power of gradient boosting with an emphasis on model interpretability. An EBM creates a blend of decision trees, similar to gradient boosting, but with a unique focus on generation of human-readable models. EBMs provide both accurate predictions and clear, intuitive explanations for those predictions. EBMs are well-suited for applications that involve healthcare, finance, and regulatory compliance, where understanding the underlying factors that drive model decisions is essential.

In SynapseML, you can use a scalable implementation of explainable boosting machines - powered by Apache Spark - to train new models. This tutorial describes how to use Apache Spar to apply the scalability and interpretability of explainable boosting machines within Microsoft Fabric. Use of explainable boosting machines with Microsoft Fabric is currently in preview.

 **Important**

This feature is in [preview](#).

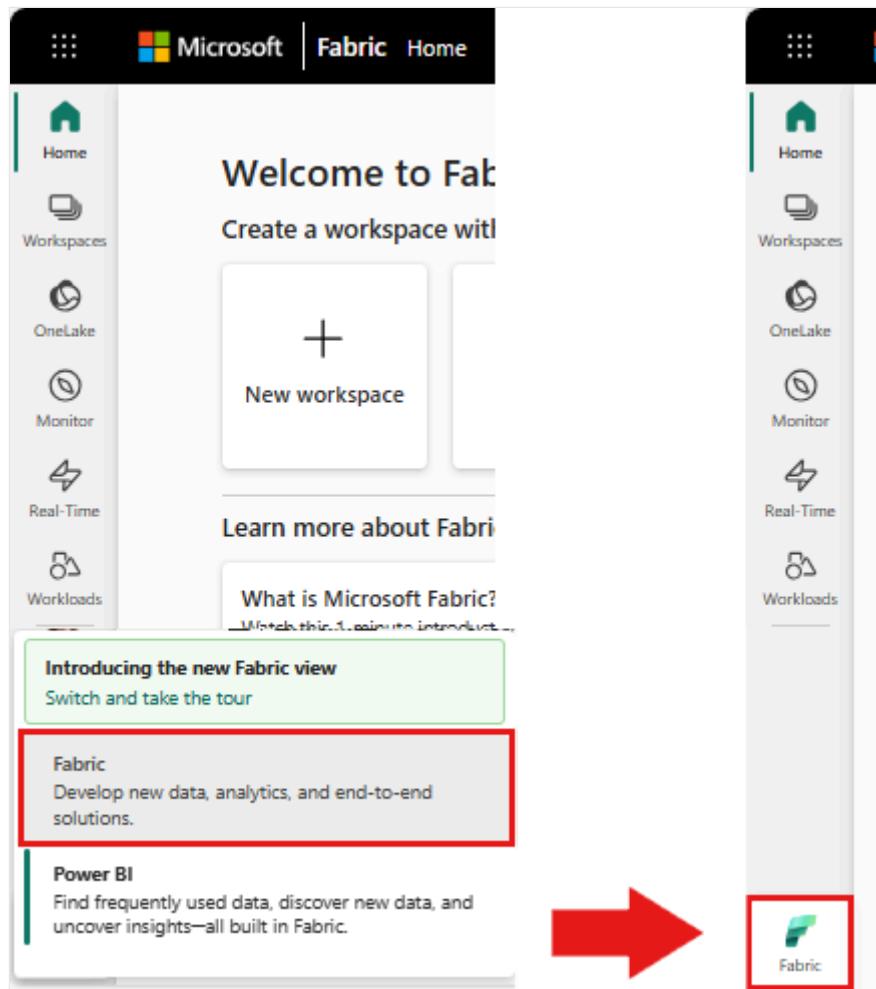
Benefits of explainable boosting machines

An EBM offers a unique blend of interpretability and predictive power, which makes it an ideal choice when transparency and comprehensibility of machine learning models are crucial. With EBMs, users can build valuable insights into the underlying factors that drive predictions, and they can then understand why a model makes specific decisions or predictions. This is essential to build trust in AI systems.

Their ability to uncover complex relationships within the data, while providing clear and interpretable results, makes EBMs highly useful in finance, healthcare, fraud detection, etc. In these areas, model explainability is not only useful but often a regulatory requirement. Ultimately, an EBM can balance between model performance and transparency, ensuring accurate, understandable, and accountable AI solutions.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free Microsoft Fabric trial.
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- Select + and then **Notebook** to create a new notebook in your workspace.

Import MLflow

MLflow allows you to track the parameters and outcomes of the model. The following code snippet shows how to use MLflow for experimentation and tracking. The `ebm-wine-quality` value is the name of the experiment that logs the information.

Python

```
# Import MLflow
import mlflow

# Create a new experiment for EBM Wine Quality
```

```
mlflow.set_experiment("ebm-wine-quality")
```

Load data

The following code snippet loads and prepares the standard **Wine Quality Data Set**, which serves as a useful regression task dataset. It shows how to load, manipulate, and convert the dataset for use with Spark-based machine learning tasks. The conversion process involves conversion of a Pandas dataframe (returned by Sklearn when using the `as_frame` argument) to a Spark dataframe, as the Spark ML style trainers require:

Python

```
import sklearn

# Load the Wine Quality Data Set using the as_frame argument for Pandas
# compatibility.
bunch = sklearn.datasets.load_wine(as_frame=True)

# Extract the data into a Pandas dataframe.
pandas_df = bunch[ 'data' ]

# Add the target variable to the Pandas dataframe.
pandas_df[ 'target' ] = bunch[ 'target' ]

# Convert the Pandas dataframe to a Spark dataframe.
df = spark.createDataFrame(pandas_df)

# Display the resulting Spark dataframe.
display(df)
```

Prepare data

For Spark ML-style learners, it's essential to organize the features within a vector-valued column. In our case here, refer to this column as "**features**." This column encompasses all the columns from the loaded dataframe, except for the target variable. This code snippet shows how to use the `VectorAssembler` resource to structure the features correctly, for subsequent Spark ML-based modeling:

Python

```
from pyspark.ml.feature import VectorAssembler

# Define the name of the target variable column.
labelColumnName = 'target'
```

```
# Create a VectorAssembler to consolidate features.  
assembler = VectorAssembler(outputCol='features')  
  
# Specify the input columns, excluding the target column.  
assembler.setInputCols([c for c in df.columns if c != labelColumnName])  
  
# Transform the dataframe to include the 'features' column.  
df_with_features = assembler.transform(df)
```

Train the model

The following code snippet uses the Synapse ML library to start the EBM regression model creation process. It first initializes the EBM regression estimator, specifying that a regression task needs it. It then sets the label column name to ensure the model knows which column to predict. Finally, it fits the model to the preprocessed dataset:

Python

```
# Import the EBMRegression estimator from Synapse ML.  
from synapse.ml.ebm import EbmRegression  
  
# Create an instance of the EBMRegression estimator.  
estimator = EbmRegression()  
  
# Set the label column for the regression task.  
estimator.setLabelCol(labelColumnName)  
  
# Fit the EBM regression model to the prepared dataset.  
model = estimator.fit(df_with_features)
```

View global explanations

You can obtain the visualization wrapper, and utilize the `interpret` library's `show` method, to visualize the overall explanation of the model. The visualization wrapper acts as a bridge to facilitate the visualization experience of the model. The following code snippet shows how to do it:

Python

```
# Get the visualization wrapper for the model.  
wrap = model.getVizWrapper()  
  
# Generate the global explanation.
```

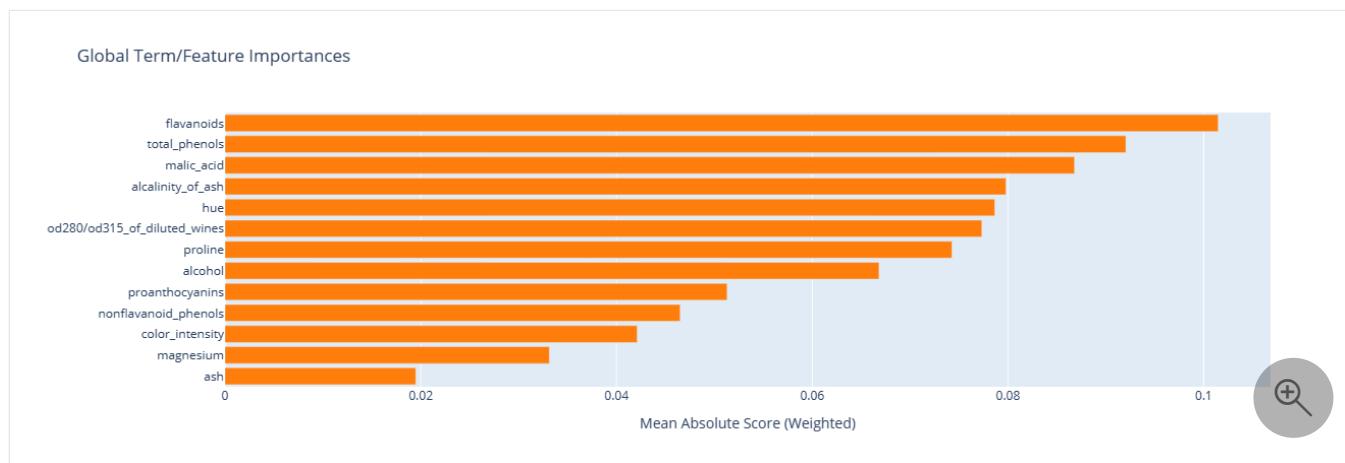
```
explanation = wrap.explain_global()
```

Next, import the `interpret` library, and use the `show` method to display the explanation:

Python

```
import interpret  
interpret.show(explanation)
```

The term "**importances**" shown in the following image represents the mean absolute contribution (score) of each term (feature or interaction) towards predictions. These contributions are averaged across the training dataset, to account for the number of samples in each bin and the sample weights (if applicable). The explanation displays the top 15 most-important terms.



View local explanations

The provided explanations operate at a global level, but in some scenarios, per-feature outputs are also valuable. Both the trainer and the model offer the capability to set the `FeaturesScoresCol` column, which, when populated, introduces another vector-valued column. Each vector in this column matches the length of the feature column, and each value corresponds to the feature at the same index. These values represent the contribution of each feature value to the final output of the model.

Unlike global explanations, there's currently no direct integration with the `interpret` visualization for per-feature outputs. This is because global visualizations scale primarily with the number of features (a typically small value), while local explanations scale with the row count (which, for a Spark dataframe, can be substantial).

The following code snippet shows how to set up and use the `FeaturesScoresCol` column:

Python

```
# Set the FeaturesScoresCol to include per-feature outputs.
prediction =
model.setFeatureScoresCol("featurescores").transform(df_with_features)

# For small datasets, you can collect the results to a single machine without
issues.
# However, for larger datasets, caution should be exercised when collecting all
rows locally.
# In this example, we convert to Pandas for easy local inspection.
predictions_pandas = prediction.toPandas()
predictions_list = prediction.collect()
```

Print the first example details:

Python

```
# Extract the first example from the collected predictions.

first = predictions_list[0]

# Print the lengths of the features and feature scores.
print('Length of the features is', len(first['features']), 'while the feature
scores have length', len(first['featurescores']))

# Print the values of the features and feature scores.
print('Features are', first['features'])
print('Feature scores are', first['featurescores'])
```

The code snippet showed how to access and print the feature , and corresponding feature scores, for the first example in the predictions. This code produces the following output:

HTML

```
Length of the features is 13 while the feature scores have length 13
Features are [14.23, 1.71, 2.43, 15.6, 127.0, 2.8, 3.06, 0.28, 2.29, 5.64, 1.04,
3.92, 1065.0]
Feature scores are
[-0.05929027436479602,-0.06788488062509922,-0.0385850430666259,-0.2761907140329337
,-0.0423377816119861,0.03582834632321236,0.07759833436021146,-0.08428610897153033
,-0.01322508472067107,-0.05477604157900576,0.08087667928468423,0.09010794901713073
,-0.09521961842295387]
```

Related content

- InterpretML explainable boosting machine: How it works ↗
- Track models with MLflow

Perform hyperparameter tuning in Fabric (preview)

Article • 04/07/2025

Hyperparameter tuning involves finding the optimal machine learning model parameter values that affect its performance. This can become challenging and time-consuming, especially for complex models and large datasets. This article shows how to perform Fabric hyperparameter tuning.

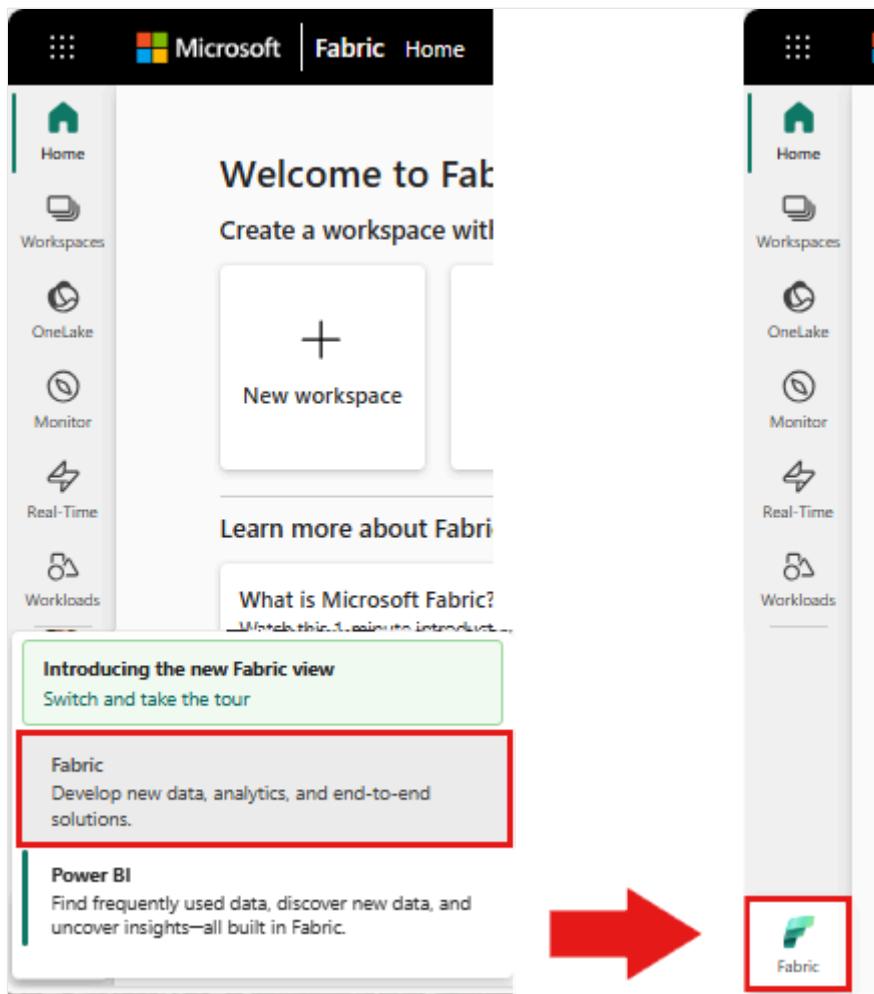
This tutorial uses the California housing dataset. That resource contains information about the median house value and other features for different census blocks in California. Once we prep the data, we train a SynapseML LightGBM model to predict the house value based on the features. Next, we use FLAML - a fast and lightweight AutoML library - to find the best hyperparameters for the LightGBM model. Finally, we compare the results of the tuned model with the baseline model that uses the default parameters.

Important

This feature is in [preview](#).

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- Create a new [Fabric environment](#), or ensure that you run on the Fabric Runtime 1.2 (Spark 3.4 (or higher) and Delta 2.4)
- Create [a new notebook](#).
- Attach your notebook to a [lakehouse](#). On the left side of your notebook, select **Add** to add an existing lakehouse or create a new one.

Prepare the training and test datasets

This section prepares the training and test datasets for the LightGBM model. We use the California housing dataset from Sklearn. We create a Spark dataframe from the data, and use a VectorAssembler to combine the features into a single vector column.

```
Python

from sklearn.datasets import fetch_california_housing
from pyspark.sql import SparkSession

# Load the Scikit-learn California Housing dataset
sklearn_dataset = fetch_california_housing()

# Convert the Scikit-learn dataset to a Pandas DataFrame
import pandas as pd
pandas_df = pd.DataFrame(sklearn_dataset.data,
```

```
columns=sklearn_dataset.feature_names)
pandas_df['target'] = sklearn_dataset.target

# Create a Spark DataFrame from the Pandas DataFrame
spark_df = spark.createDataFrame(pandas_df)

# Display the data
display(spark_df)
```

Next, randomly split the data into three subsets: training, validation, and test, with 85%, 12.75%, and 2.25% of the data respectively. Use the training and validation sets for hyperparameter tuning, and the test set for model evaluation.

Python

```
from pyspark.ml.feature import VectorAssembler

# Combine features into a single vector column
featurizer = VectorAssembler(inputCols=sklearn_dataset.feature_names,
outputCol="features")
data = featurizer.transform(spark_df)[ "target", "features"]

# Split the data into training, validation, and test sets
train_data, test_data = data.randomSplit([0.85, 0.15], seed=41)
train_data_sub, val_data_sub = train_data.randomSplit([0.85, 0.15], seed=41)
```

Set up the ML experiment

Configure MLflow

Before we perform hyperparameter tuning, we need to define a training function that can take different values of hyperparameters and train a LightGBM model on the training data. We also need to evaluate the model performance on the validation data using the R2 score, which measures how well the model fits the data.

To do this, first import the necessary modules and set up the MLflow experiment. The open source MLflow platform manages the end-to-end machine learning lifecycle. It helps track and compare the results of different models and hyperparameters.

Python

```
# Import MLflow and set up the experiment name
import mlflow
```

```
mlflow.set_experiment("flaml_tune_sample")

# Enable automatic logging of parameters, metrics, and models
mlflow.autolog(exclusive=False)
```

Set the logging level

Configure the logging level to suppress unnecessary output from the Synapse.ml library. This step keeps the logs cleaner.

Python

```
import logging

logging.getLogger('synapse.ml').setLevel(logging.ERROR)
```

Train the baseline model

Define the training function. This function takes four hyperparameters

- alpha
- learningRate
- numLeaves
- numIterations

as inputs. We want to tune these hyperparameters later on, using FLAML.

The train function also takes two dataframes as inputs: train_data and val_data - the training and validation datasets respectively. The train function returns two outputs: the trained model and the R2 score on the validation data.

Python

```
# Import LightGBM and RegressionEvaluator
from synapse.ml.lightgbm import LightGBMRegressor
from pyspark.ml.evaluation import RegressionEvaluator

def train(alpha, learningRate, numLeaves, numIterations,
train_data=train_data_sub, val_data=val_data_sub):
    """
    This train() function:
        - takes hyperparameters as inputs (for tuning later)
        - returns the R2 score on the validation dataset

    Wrapping code as a function makes it easier to reuse the code later for
```

```

tuning.

"""

with mlflow.start_run() as run:

    # Capture run_id for prediction later
    run_details = run.info.run_id

    # Create a LightGBM regressor with the given hyperparameters and target
    column
    lgr = LightGBMRegressor(
        objective="quantile",
        alpha=alpha,
        learningRate=learningRate,
        numLeaves=numLeaves,
        labelCol="target",
        numIterations=numIterations,
        dataTransferMode="bulk"
    )

    # Train the model on the training data
    model = lgr.fit(train_data)

    # Make predictions on the validation data
    predictions = model.transform(val_data)
    # Define an evaluator with R2 metric and target column
    evaluator = RegressionEvaluator(predictionCol="prediction",
labelCol="target", metricName="r2")
    # Compute the R2 score on the validation data
    eval_metric = evaluator.evaluate(predictions)

    mlflow.log_metric("r2_score", eval_metric)

    # Return the model and the R2 score
    return model, eval_metric, run_details

```

Finally, use the train function to train a baseline model with the default hyperparameter values. Also, evaluate the baseline model on the test data and print the R2 score.

Python

```

# Train the baseline model with the default hyperparameters
init_model, init_eval_metric, init_run_id = train(alpha=0.2, learningRate=0.3,
numLeaves=31, numIterations=100, train_data=train_data, val_data=test_data)
# Print the R2 score of the baseline model on the test data
print("R2 of initial model on test dataset is: ", init_eval_metric)

```

Perform hyperparameter tuning with FLAML

The FLAML AutoML library is a fast and lightweight library resource that can automatically find the best hyperparameters for a given model and dataset. It uses a low-cost search strategy that adapts to the feedback from the evaluation metric. In this section, we use FLAML to tune the hyperparameters of the LightGBM model that we defined in the previous section.

Define the tune function

To use FLAML, we must define a tune function that takes a config dictionary as input, and returns a dictionary. That dictionary has the evaluation metric as the key, and the metric value as the value.

The config dictionary contains the hyperparameters that we want to tune. It also contains their values. The tune function uses the train function that we defined earlier, to train and evaluate the model with the given config.

Python

```
# Import FLAML
import flaml

# Define the tune function
def flaml_tune(config):
    # Train and evaluate the model with the given config
    _, metric, run_id = train(**config)
    # Return the evaluation metric and its value
    return {"r2": metric}
```

Define the search space

We must define the search space for the hyperparameters that we want to tune. The search space is a dictionary that maps the hyperparameter names to the ranges of values that we want to explore. FLAML provides some convenient functions to define different types of ranges - for example, uniform, loguniform, and randint.

Here, we want to tune the following four hyperparameters: alpha, learningRate, numLeaves, and numIterations.

Python

```
# Define the search space
params = {
    # Alpha is a continuous value between 0 and 1
    "alpha": flaml.tune.uniform(0, 1),
    # Learning rate is a continuous value between 0.001 and 1
    "learning_rate": flaml.tune.uniform(0.001, 1),
    # Number of leaves is an integer between 10 and 50
    "num_leaves": flaml.tune.randint(10, 50),
    # Number of iterations is an integer between 100 and 500
    "num_iterations": flaml.tune.randint(100, 500)}
```

```
"learningRate": flaml.tune.uniform(0.001, 1),  
    # Number of leaves is an integer value between 30 and 100  
    "numLeaves": flaml.tune.randint(30, 100),  
    # Number of iterations is an integer value between 100 and 300  
    "numIterations": flaml.tune.randint(100, 300),  
}
```

Define the hyperparameter trial

Finally, we must define a hyperparameter trial that uses FLAML to optimize the hyperparameters. We must pass the tune function, the search space, the time budget, the number of samples, the metric name, the mode, and the verbosity level to the `flaml.tune.run` function. We must also start a nested MLflow run to track the results of the trial.

The `flaml.tune.run` function returns an analysis object that contains the best config and the best metric value.

Python

```
# Start a nested MLflow run  
with mlflow.start_run(nested=True, run_name="Child Run: "):  
    # Run the hyperparameter trial with FLAML  
    analysis = flaml.tune.run(  
        # Pass the tune function  
        flaml_tune,  
        # Pass the search space  
        params,  
        # Set the time budget to 120 seconds  
        time_budget_s=120,  
        # Set the number of samples to 100  
        num_samples=100,  
        # Set the metric name to r2  
        metric="r2",  
        # Set the mode to max (we want to maximize the r2 score)  
        mode="max",  
        # Set the verbosity level to 5  
        verbose=5,  
    )
```

After the trial finishes, view the best configuration and the best metric value from the analysis object.

Python

```
# Get the best config from the analysis object  
flaml_config = analysis.best_config
```

```
# Print the best config
print("Best config: ", flaml_config)
print("Best score on validation data: ", analysis.best_result["r2"])
```

Compare the results

After we find the best hyperparameters with FLAML, we must evaluate how much those hyperparameters improve the model performance. To do this, use the train function to create a new model with the best hyperparameters on the full training dataset. Then, use the test dataset to calculate the R2 score for both the new model and the baseline model.

Python

```
# Train a new model with the best hyperparameters
flaml_model, flaml_metric, flaml_run_id = train(train_data=train_data,
val_data=test_data, **flaml_config)

# Print the R2 score of the baseline model on the test dataset
print("On the test dataset, the initial (untuned) model achieved R^2: ",
init_eval_metric)
# Print the R2 score of the new model on the test dataset
print("On the test dataset, the final flaml (tuned) model achieved R^2: ",
flaml_metric)
```

Save the final model

After we complete our hyperparameter trial, we can save the final, tuned model as an ML model in Fabric.

Python

```
# Specify the model name and the path where you want to save it in the registry
model_name = "housing_model" # Replace with your desired model name
model_path = f"runs:{flaml_run_id}/model"

# Register the model to the MLflow registry
registered_model = mlflow.register_model(model_uri=model_path, name=model_name)

# Print the registered model's name and version
print(f"Model '{registered_model.name}' version {registered_model.version} registered successfully.")
```

Related content

- [Visualize results](#)
- [Hyperparameter tuning in Fabric](#)

Automated ML in Fabric (preview)

Article • 11/19/2024

Automated Machine Learning (AutoML) enables users to build and deploy machine learning models by automating the most time-consuming and complex parts of the model development process. Traditionally, building a machine learning model requires expertise in data science, model selection, hyperparameter tuning, and evaluation—a process that can be resource-intensive and prone to trial-and-error. AutoML simplifies this by automatically selecting the best algorithms, tuning hyperparameters, and generating optimized models based on the input data and desired outcomes.

In Microsoft Fabric, AutoML becomes even more powerful by integrating seamlessly with the platform's data ecosystem, allowing users to build, train, and deploy models directly on their lakehouses. With AutoML, both technical and non-technical users can create predictive models quickly, making machine learning accessible to a broader audience. From forecasting demand to detecting anomalies and optimizing business operations, AutoML in Fabric accelerates the path from raw data to actionable insights, empowering users to leverage AI with minimal effort and maximum impact.

Important

This feature is in [preview](#).

How does AutoML work?

[FLAML \(Fast and Lightweight AutoML\)](#) powers the AutoML capabilities in Fabric, enabling users to build, optimize, and deploy machine learning models seamlessly within the platform's data ecosystem.

FLAML is an open-source AutoML library designed to deliver accurate models quickly by focusing on efficiency, minimizing computational costs, and dynamically tuning hyperparameters. Behind the scenes, FLAML automates model selection and optimization using a resource-aware search strategy, balancing exploration and exploitation to identify the best models without exhaustive trial-and-error. Its adaptive search space and lightweight algorithms make it ideal for large datasets and constrained environments, ensuring scalable and fast performance. This integration with Fabric makes machine learning accessible to both technical and non-technical users, accelerating the path from raw data to actionable insights.

Machine learning tasks

AutoML in Fabric supports a wide range of machine learning tasks, including classification, regression, and forecasting, making it versatile for various data-driven applications.

Binary classification

Binary classification is a type of supervised machine learning task where the goal is to categorize data points into one of two distinct classes. It involves training a model on labeled data, where each instance is assigned to one of two possible categories, and the model learns to predict the correct class for new, unseen data. Examples include:

- **Spam Detection:** Classifying emails as either spam or not spam.
- **Fraud Detection:** Flagging financial transactions as fraudulent or legitimate.
- **Disease Screening:** Predicting whether a patient has a condition (positive) or not (negative).

Multi-class classification

Multi-Class Classification for tabular data involves assigning one of several possible labels to each row of structured data based on the features in that dataset. Here are a few examples relevant to real-world tabular datasets:

- **Customer Segmentation:** Classifying customers into segments such as "High-value," "Moderate-value," or "Low-value" based on demographic, purchase, and behavioral data.
- **Loan Risk Assessment:** Predicting the risk level of a loan application as "Low," "Medium," or "High" using applicant data like income, credit score, and employment status.
- **Product Category Prediction:** Assigning an appropriate product category, such as "Electronics," "Clothing," or "Furniture," based on attributes like price, brand, and product specifications.
- **Disease Diagnosis:** Identifying the type of disease a patient might have, such as "Diabetes Type 1," "Diabetes Type 2," or "Gestational Diabetes," based on clinical metrics and test results.

These examples highlight how multi-class classification can support decision-making in various industries, where the outcome can take one of several mutually exclusive categories.

Regression

Regression is a type of machine learning used to predict a number based on other related data. It's helpful when we want to estimate a specific value, like a price, temperature, or time, based on different factors that could affect it. Here are some example scenarios:

- Predicting **house prices** using information like square footage, number of rooms, and location.
- Estimating **monthly sales** based on marketing spend, seasonality, and past sales trends.

Forecasting

Forecasting is a machine learning technique used to predict future values based on historical data. It's especially useful for planning and decision-making in situations where past trends and patterns can inform what's likely to happen next. Forecasting takes time-based data—also called **time series data**—and analyzes patterns like seasonality, trends, and cycles to make accurate predictions. Here are some example scenarios:

- **Sales Forecasting:** Predicting future sales figures based on past sales, seasonality, and market trends.
- **Inventory Forecasting:** Determining the future demand for products using previous purchasing data and seasonal cycles.

Forecasting helps organizations make informed decisions, whether it's ensuring enough stock, planning resources, or preparing for market changes.

Training and test datasets

Creating **training and test datasets** is an essential step in building machine learning models. The **training dataset** is used to teach the model, allowing it to learn patterns from labeled data, while the **test dataset** evaluates the model's performance on new, unseen data, helping to check its accuracy and generalizability. Splitting data this way ensures the model isn't simply memorizing but can generalize to other data.

In Fabric, AutoML tools simplify this process by automatically splitting data into training and test sets, customizing the split based on best practices for the specific task, such as classification, regression, or forecasting.

Feature engineering

Feature engineering is the process of transforming raw data into meaningful features that improve a machine learning model's performance. It's a critical step because the right features help the model learn the important patterns and relationships in the data, leading to better predictions. For instance, in a dataset of dates, creating features like "is holiday" can reveal trends that improve forecasting models.

In Fabric, users can leverage the `auto_featurize` functionality to automate parts of this process. `auto_featurize` analyzes the data and suggests or generates relevant features, such as aggregations, categorical encodings, or transformations, that may enhance the model's predictive power. This functionality saves time and brings feature engineering within reach for users with varied experience levels, enabling them to build more accurate and robust models.

Next steps

- [Use the AutoML interface](#)
 - [Experiment with the AutoML Python APIs](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Use the low-code AutoML interface in Fabric (preview)

Article • 11/19/2024

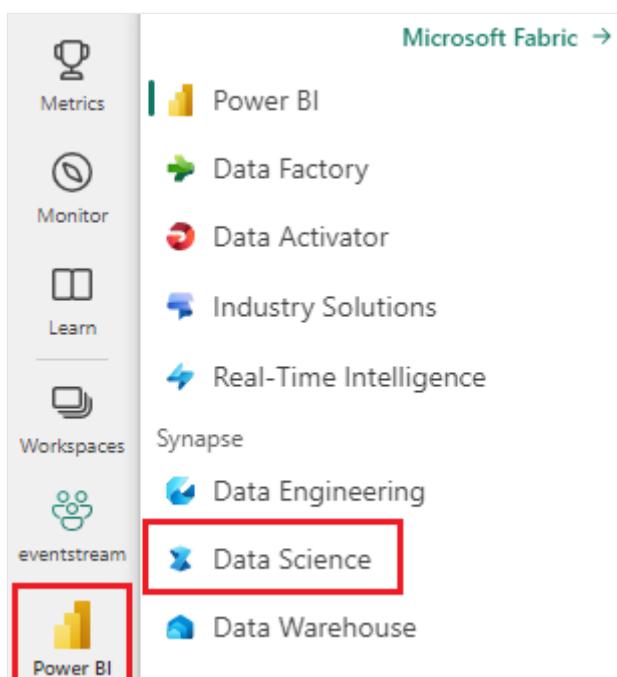
The low-code AutoML interface in Fabric makes it easy for users to get started with machine learning by specifying their ML task and a few basic configurations. Based on these selections, the AutoML UI generates a pre-configured notebook tailored to the user's inputs. Once executed, all model metrics and iterations are automatically logged and tracked within existing ML experiments and model items, providing an organized and efficient way to manage and evaluate model performance.

ⓘ Important

This feature is in [preview](#).

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



Setup an Automated ML trial

The AutoML wizard in Fabric can be conveniently launched directly from an existing experiment, model, or notebook item.

Customer churn | Data updated 12/12/20

Home Label

Experiment runs 25

Properties

Description: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed eusmod.

Run name: run_diabetes_n21 Start date: 07.12.2022, 10:00:05 PM

Created by: Daisy Source: tamil-synapse-regression

Train an ML model without writing any code

Follow a few steps to set up your training data and goals, and get a completed notebook to train an ML model.

Try now Got it

Name Value

Model version metrics (6)

Model type: Regression

Accuracy: 0.862

F1 Score: 0.87

Training time: 0.855

Loss: 0.257

Recall: 0.87

Model hyperparameters (5)

Model training and test size (2)

Input schema (100)

Output schema (10)

Save

Compare runs in a list

View run list

Customer Churn experiment

Customer Churn experiment

Data Science

Choose data source

AutoML users in Fabric have the option to select from their available lakehouses, making it easy to access and analyze data stored within the platform. Once a lakehouse is selected, users can choose a specific table or file to use for their AutoML tasks.

CustomerChurn-experiment | Data updated 12/12/20

Home Label

Experiment runs 25

Choose data source

All data My data Endorsed in your org Favorites

Find a workspace

All

Customer360

Client Logs Db

Top Campaigns

Dataflow for triggers

Daily Sales

Contoso DB

Test datamart

Contoso data

Primary content

Primary content 2024

Contoso data

Primary content

Refreshed Owner Location Endorsement Sensitivity

Customer360 May 23 at 3:00 P Tim Deboar Contoso workspace Certified Confidential

Client Logs Db May 23 at 3:00 P Daichi Fukuda Contoso workspace Certified Public

Top Campaigns May 23 at 3:00 P Emiliano Ceballo Contoso workspace Certified -

Dataflow for triggers May 23 at 3:00 P Mikhail Kotov Contoso workspace Certified -

Daily Sales May 23 at 3:00 P Marie Mikhail Kc Contoso workspace Certified Public

Contoso DB May 23 at 3:00 P Oscar Krogh Contoso workspace Promoted Confidential

Test datamart May 23 at 3:00 P Marie Beaudoir Contoso workspace Promoted Public

Contoso data May 23 at 3:00 P Tim Deboar Contoso workspace Certified Public

Primary content May 23 at 3:00 P Ruth Bengtsson Contoso workspace Certified -

Primary content 2024 May 23 at 3:00 P Ruth Bengtsson Contoso workspace Certified -

Contoso data May 23 at 3:00 P Ruth Bengtsson Contoso workspace Certified -

Primary content May 23 at 3:00 P Tim Deboar Contoso workspace Confidential/Certified

Next

Customer Churn experiment

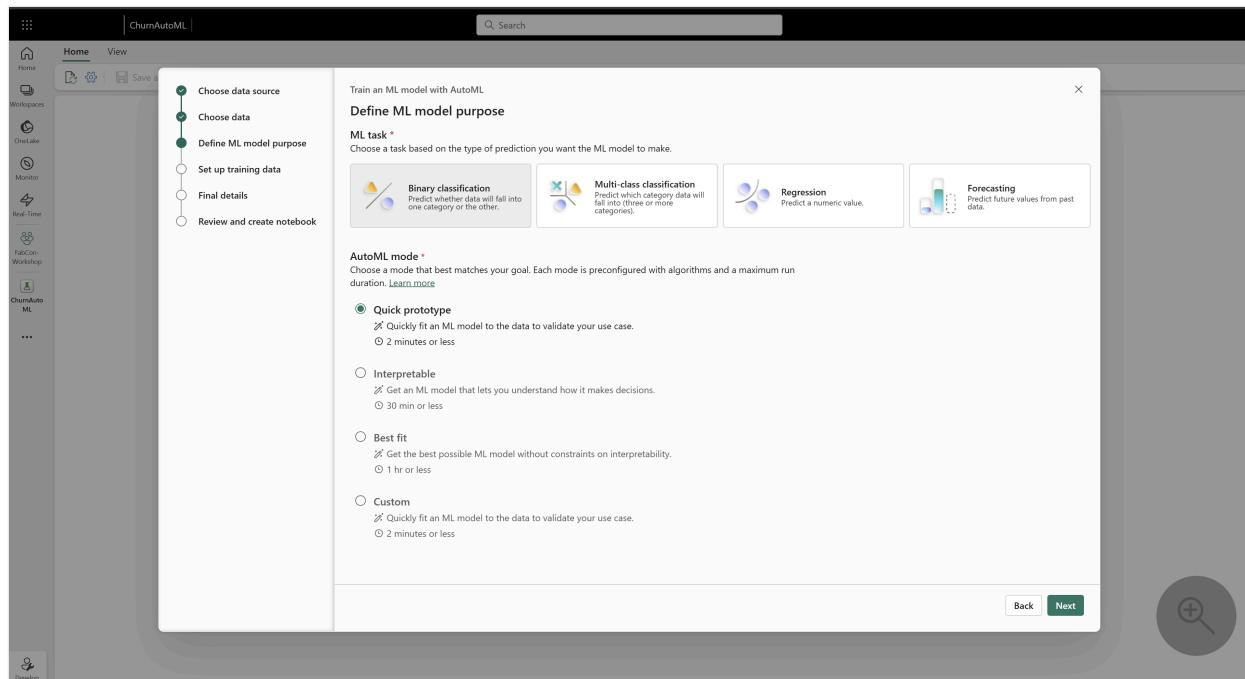
Data Science

💡 Tip

When selecting a lakehouse, users can pick a **table** or a **file** to use with AutoML. Supported file types include **CSV**, **XLS**, **XLSX**, and **JSON**.

Define ML model purpose

In this step, users define the purpose of their model by selecting the **ML task** that best fits their data and goals.



Fabric's AutoML wizard offers the following ML tasks:

- **Regression:** For predicting continuous numerical values.
- **Binary Classification:** For categorizing data into one of two classes.
- **Multi-Class Classification:** For categorizing data into one of multiple classes.
- **Forecasting:** For making predictions over time series data.

Once you've selected your ML task, you can then choose an **AutoML Mode**. Each mode sets default configurations for the AutoML trial, such as which models to explore and the time allocated to find the best model. The available modes are:

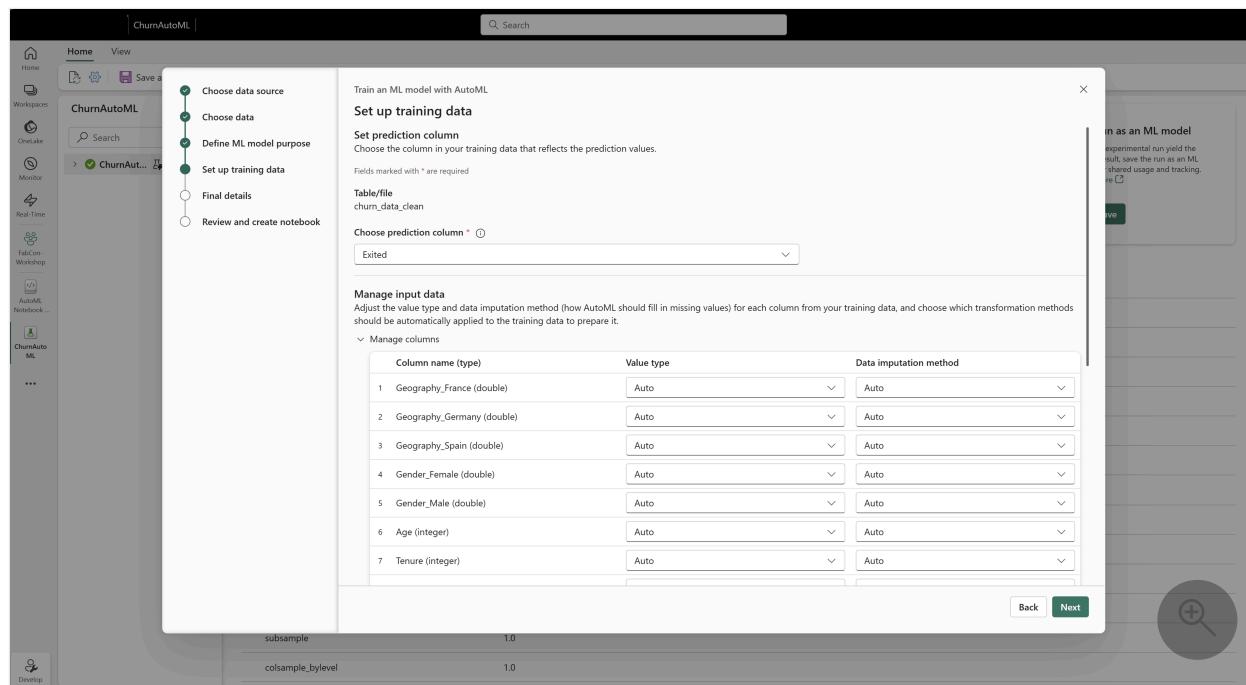
- **Quick Prototype:** Delivers rapid results, ideal for testing and iterating quickly.
- **Interpretable Mode:** Runs a bit longer and focuses on models that are inherently easier to interpret.
- **Best Fit:** Conducts a more comprehensive search with an extended runtime, aiming to find the best possible model.

- **Custom:** Allows you to manually adjust some settings in your AutoML trial for a tailored configuration.

Selecting the right ML task and AutoML mode ensures that the AutoML wizard aligns with your objectives, balancing speed, interpretability, and performance based on your chosen configuration.

Set up training data

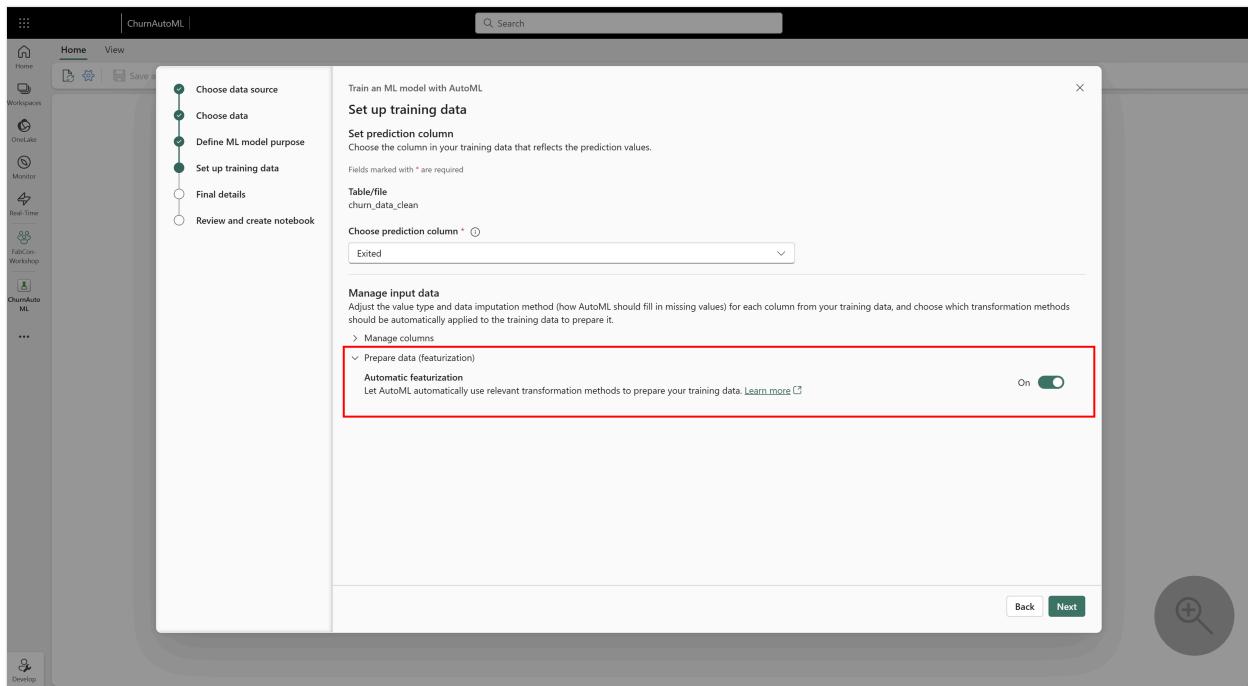
In this step, you'll configure the training data that AutoML will use to build your model. Start by selecting the **prediction column**—this is the target column that your model will be trained to predict.



After selecting your prediction column, you can further customize how your input data is handled:

- **Data Types:** Review and adjust the data types for each input column to ensure compatibility and optimize the model's performance.
- **Imputation Method:** Choose how to handle missing values in your dataset by selecting an imputation method, which will fill gaps in the data based on your preferences.

You can also enable or disable the **auto featurize** setting. When enabled, auto featurize generates additional features for training, potentially enhancing model performance by extracting extra insights from your data. Defining these data settings helps the AutoML wizard accurately interpret and process your dataset, improving the quality of your trial results.



Provide final details

Now, you'll decide how you want your AutoML trial to be executed, along with naming conventions for your experiment and output. You have two options for executing your AutoML trial:

- 1. Train Multiple Models Simultaneously:** This option is ideal if your data can be loaded into a pandas DataFrame, allowing you to leverage your Spark cluster to run multiple models in parallel. This approach accelerates the trial process by training several models at once.
- 2. Train Models Sequentially Using Spark:** This option is suited for larger datasets or those that benefit from distributed training. It uses Spark and SynapseML to explore distributed models, training one model at a time with the scalability that Spark provides.

! Note

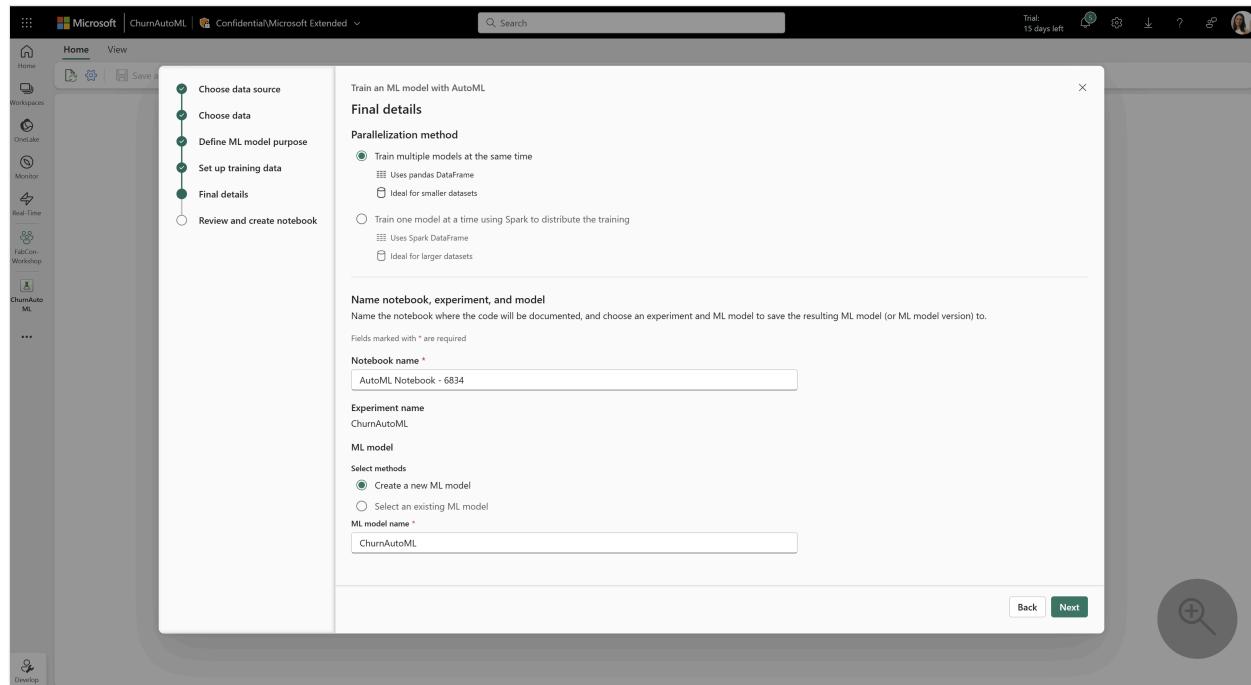
Currently, the **Spark mode** does not support logging the input and output schema for Spark-based models. This schema is a required field for the **SynapseML PREDICT** function. As a workaround, you can load the model directly with **MLFlow** and perform inferencing within your notebook, bypassing the schema requirement for prediction.

After selecting your execution mode, finalize your setup by specifying names for your **Notebook**, **Experiment**, and **Model**. These naming conventions will help organize your

AutoML assets within Fabric and make it easy to track and manage your trials. Once complete, a notebook will be generated based on your selections, ready to execute and customize as needed.

Review and create notebook

In the final step, you'll have the chance to review all your AutoML settings and preview the generated code that aligns with your selections. This is your opportunity to ensure that the chosen ML task, mode, data setup, and other configurations meet your objectives.



Once you're satisfied, you can finalize this step to generate a notebook that includes all the components of your AutoML trial. This notebook allows you to track each stage of the process, from data preparation to model evaluation, and serves as a comprehensive record of your work. You can also further customize this notebook as needed, adjusting code and settings to refine your AutoML trial results.

Track Your AutoML Runs

Once you execute your notebook, the AutoML code will utilize **MLFlow logging** to automatically track key metrics and parameters for each model tested during the trial. This seamless integration allows you to monitor and review each iteration of your AutoML run without needing additional setup.

To explore the results of your AutoML trial:

- 1. Navigate to your ML Experiment item:** In a [ML experiment](#), you can track all the different runs created by your AutoML process. Each run logs valuable details such as model performance metrics, parameters, and configurations, making it easy to analyze and compare results.
- 2. Review AutoML Configurations:** For each AutoML trial, you'll find the AutoML configurations used, providing insights into how each model was set up and which settings led to optimal results.
- 3. Locate the Best Model:** Open your [ML model](#) to access the final, best-performing model from your AutoML trial.

This tracking workflow helps you organize, evaluate, and manage your models, ensuring you have full visibility into the performance and settings of each model tested in your AutoML trial. From here, you can leverage the [SynapseML PREDICT interface](#) or generate predictions directly from your notebooks.

Next steps

- [Learn about AutoML](#)
- [Experiment with the AutoML Python APIs](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Code-first AutoML in Fabric (preview)

Article • 11/19/2024

AutoML (Automated Machine Learning) is a collection of methods and tools that automate machine learning model training and optimization with little human involvement. The aim of AutoML is to simplify and speed up the process of choosing the best machine learning model and hyperparameters for a given dataset, which usually demands much skill and computing power.

ⓘ Important

This feature is in [preview](#).

In Fabric, data scientists can use `f1aml.AutoML` to automate their machine learning tasks.

AutoML can help ML professionals and developers from different sectors to:

- Build ML solutions with minimal coding
- Reduce time and cost
- Apply data science best practices
- Solve problems quickly and efficiently

AutoML workflow

`f1aml.AutoML` is a class for AutoML based on the task. It can be used as a Scikit-learn style estimator with the usual fit and predict methods.

To start an AutoML trial, users only need to provide the training data and the task type. With the integrated MLflow experiences in Fabric, users can also examine the different runs that were attempted in the trial to see how the final model was chosen.

Training data

In Fabric, users can pass the following input types to the AutoML `fit` function:

- Numpy Array: When the input data is stored in a Numpy array, it's passed to `fit()` as `X_train` and `y_train`.
- Pandas dataframe: When the input data is stored in a Pandas dataframe, it's passed to `fit()` either as `X_train` and `y_train`, or as `dataframe` and `label`.

- Pandas on Spark dataframe: When the input data is stored as a Spark dataframe, it can be converted into a `Pandas` on `Spark` dataframe using `to_pandas_on_spark()` and then passed to `fit()` as a dataframe and label.

Python

```
from flaml.automl.spark.utils import to_pandas_on_spark
psdf = to_pandas_on_spark(sdf)
automl.fit(dataframe=psdf, label='Bankrupt?', isUnbalance=True,
**settings)
```

Machine learning problem

Users can specify the machine learning task using the `task` argument. There are various supported machine learning tasks, including:

- Classification: The main goal of classification models is to predict which categories new data fall into based on learnings from its training data. Common classification examples include fraud detection, handwriting recognition, and object detection.
- Regression: Regression models predict numerical output values based on independent predictors. In regression, the objective is to help establish the relationship among those independent predictor variables by estimating how one variable impacts the others. For example, automobile prices based on features like, gas mileage, safety rating, etc.
- Time Series Forecasting: This is used to predict future values based on historical data points ordered by time. In a time series, data is collected and recorded at regular intervals over a specific period, such as daily, weekly, monthly, or yearly. The objective of time series forecasting is to identify patterns, trends, and seasonality in the data and then use this information to make predictions about future value.

To learn more about the other tasks supported in FLAML, you can visit the [documentation on AutoML tasks in FLAML](#).

Optional inputs

Provide various constraints and inputs to configure your AutoML trial.

Constraints

When creating an AutoML trial, users can also configure constraints on the AutoML process, constructor arguments of potential estimators, types of models tried in AutoML, and even constraints on the metrics of the AutoML trial.

For example, the code below allows users to specify a metrics constraint on the AutoML trial.

Python

```
metric_constraints = [("train_loss", "<=", 0.1), ("val_loss", "<=", 0.1)]  
automl.fit(X_train, y_train, max_iter=100, train_time_limit=1,  
metric_constraints=metric_constraints)
```

To learn more about these configurations, you can visit the [documentation on configurations in FLAML](#).

Optimization metric

During training, the AutoML function creates many trials, which try different algorithms and parameters. The AutoML tool iterates through ML algorithms and hyperparameters. In this process, each iteration creates a model with a training score. The better the score for the metric you want to optimize for, the better the model is considered to "fit" your data. The optimization metric is specified via the `metric` argument. It can be either a string, which refers to a built-in metric, or a user-defined function.

[AutoML optimization metrics](#)

Parallel tuning

In some cases, you might want to expedite your AutoML trial by using Apache Spark to parallelize your training. For Spark clusters, by default, FLAML launches one trial per executor. You can also customize the number of concurrent trials by using the `n_concurrent_trials` argument.

Python

```
automl.fit(X_train, y_train, n_concurrent_trials=4, use_spark=True)
```

To learn more about how to parallelize your AutoML trails, you can visit the [FLAML documentation for parallel Spark jobs](#).

Track with MLflow

You can also use the Fabric MLflow integration to capture the metrics, parameters, and metrics of the explored trails.

Python

```
import mlflow
mlflow.autolog()

with mlflow.start_run(nested=True):
    automl.fit(dataframe=pandas_df, label='Bankrupt?', mlflow_exp_name =
"automl_spark_demo")

# You can also provide a run_name pre-fix for the child runs

automl_experiment = flaml.AutoML()
automl_settings = {
    "metric": "r2",
    "task": "regression",
    "use_spark": True,
    "mlflow_exp_name": "test_doc",
    "estimator_list": [
        "lgbm",
        "rf",
        "xgboost",
        "extra_tree",
        "xgb_limitdepth",
    ], # catboost does not yet support mlflow autologging
}
with mlflow.start_run(run_name=f"automl_spark_trials"):
    automl_experiment.fit(X_train=train_x, y_train=train_y,
**automl_settings)
```

Supported models

AutoML in Fabric supports the following models:

[+] Expand table

Classification	Regression	Time Series Forecasting
(PySpark) Gradient-Boosted Trees (GBT) Classifier	(PySpark) Accelerated Failure Time (AFT) Survival Regression	Arimax
(PySpark) Linear SVM	(PySpark) Generalized Linear Regression	AutoARIMA
(PySpark) Naive Bayes	(PySpark) Gradient-Boosted Trees (GBT) Regression	Average

Classification	Regression	Time Series Forecasting
(Synapse) LightGBM	(PySpark) Linear Regression	CatBoost
CatBoost	(Synapse) LightGBM	Decision Tree
Decision Tree	CatBoost	Exponential Smoothing
Extremely Randomized Trees	Decision Tree	Extremely Randomized Trees
Gradient Boosting	Elastic Net	ForecastTCN
K Nearest Neighbors	Extremely Randomized Trees	Gradient Boosting
Light GBM	Gradient Boosting	Holt-Winters Exponential Smoothing
Linear SVC	K Nearest Neighbors	K Nearest Neighbors
Logistic Regression	LARS Lasso	LARS Lasso
Logistic Regression with L1/L2 Regularization	Light GBM	Light GBM
Naive Bayes	Logistic Regression with L1/L2 Regularization	Naive
Random Forest	Random Forest	Orbit
Random Forest on Spark	Random Forest on Spark	Prophet
Stochastic Gradient Descent (SGD)	Stochastic Gradient Descent (SGD)	Random Forest
Support Vector Classification (SVC)	XGBoost	SARIMAX
XGboost	XGBoost with Limited Depth	SeasonalAverage
XGBoost with Limited Depth		SeasonalNaive
		Temporal Fusion Transformer
		XGBoost
		XGBoost for Time Series
		XGBoost with Limited Depth for Time Series
		ElasticNet

Visualize results

The `flaml.visualization` module provides utility functions for plotting the optimization process using Plotly. By leveraging Plotly, users can interactively explore their AutoML experiment results. To use these plotting functions, provide your optimized `flaml.AutoML` or `flaml.tune.tune.ExperimentAnalysis` object as an input.

You can use the following functions within your notebook:

- `plot_optimization_history`: Plot optimization history of all trials in the experiment.
- `plot_feature_importance`: Plot importance for each feature in the dataset.
- `plot_parallel_coordinate`: Plot the high-dimensional parameter relationships in the experiment.
- `plot_contour`: Plot the parameter relationship as contour plot in the experiment.
- `plot_edf`: Plot the objective value EDF (empirical distribution function) of the experiment.
- `plot_timeline`: Plot the timeline of the experiment.
- `plot_slice`: Plot the parameter relationship as slice plot in a study.
- `plot_param_importance`: Plot the hyperparameter importance of the experiment.

Related content

- [Visualize AutoML results](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Create models with Automated ML (preview)

Article • 03/26/2024

Automated Machine Learning (AutoML) encompasses a set of techniques and tools designed to streamline the process of training and optimizing machine learning models with minimal human intervention. The primary objective of AutoML is to simplify and accelerate the selection of the most suitable machine learning model and hyperparameters for a given dataset, a task that typically demands considerable expertise and computational resources. Within the Fabric framework, data scientists can leverage the `fml AutoML` module to automate various aspects of their machine learning workflows.

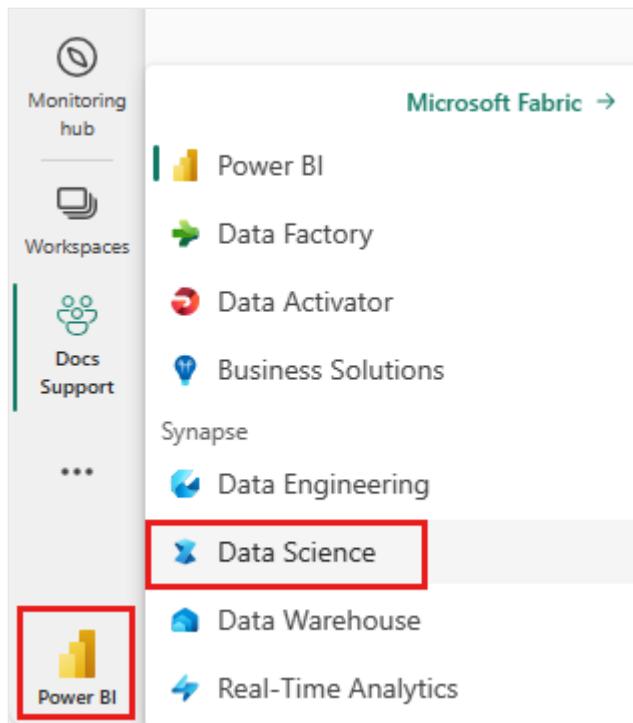
In this article, we will delve into the process of generating AutoML trials directly from code using a Spark dataset. Additionally, we will explore methods for converting this data into a Pandas dataframe and discuss techniques for parallelizing your experimentation trials.

ⓘ Important

This feature is in [preview](#).

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



- Create a new [Fabric environment](#) or ensure you are running on the Fabric Runtime 1.2 (Spark 3.4 (or higher) and Delta 2.4)
- Create a [new notebook](#).
- Attach your notebook to a lakehouse. On the left side of your notebook, select [Add](#) to add an existing lakehouse or create a new one.

Load and prepare data

In this section, we'll specify the download settings for the data and then save it to the lakehouse.

Download data

This code block downloads the data from a remote source and saves it to the lakehouse

Python

```
import os
import requests

IS_CUSTOM_DATA = False # if TRUE, dataset has to be uploaded manually

if not IS_CUSTOM_DATA:
    # Specify the remote URL where the data is hosted
    remote_url =
"https://synapseaisolutionsa.blob.core.windows.net/public/bankcustomerchurn"

    # List of data files to download
    file_list = ["churn.csv"]
```

```
# Define the download path within the lakehouse
download_path = "/lakehouse/default/Files/churn/raw"

# Check if the lakehouse directory exists; if not, raise an error
if not os.path.exists("/lakehouse/default"):
    raise FileNotFoundError("Default lakehouse not found. Please add a
lakehouse and restart the session.")

# Create the download directory if it doesn't exist
os.makedirs(download_path, exist_ok=True)

# Download each data file if it doesn't already exist in the lakehouse
for fname in file_list:
    if not os.path.exists(f"{download_path}/{fname}"):
        r = requests.get(f"{remote_url}/{fname}", timeout=30)
        with open(f"{download_path}/{fname}", "wb") as f:
            f.write(r.content)

print("Downloaded demo data files into lakehouse.")
```

Load data into a Spark dataframe

The following code block loads the data from the CSV file into a Spark DataFrame and caches it for efficient processing.

Python

```
df = (
    spark.read.option("header", True)
    .option("inferSchema", True)
    .csv("Files/churn/raw/churn.csv")
    .cache()
)
```

This code assumes that the data file has been downloaded and is located in the specified path. It reads the CSV file into a Spark DataFrame, infers the schema, and caches it for faster access during subsequent operations.

Prepare the data

In this section, we'll perform data cleaning and feature engineering on the dataset.

Clean data

First, we define a function to clean the data, which includes dropping rows with missing data, removing duplicate rows based on specific columns, and dropping unnecessary columns.

Python

```
# Define a function to clean the data
def clean_data(df):
    # Drop rows with missing data across all columns
    df = df.dropna(how="all")
    # Drop duplicate rows based on 'RowNumber' and 'CustomerId'
    df = df.dropDuplicates(subset=['RowNumber', 'CustomerId'])
    # Drop columns: 'RowNumber', 'CustomerId', 'Surname'
    df = df.drop('RowNumber', 'CustomerId', 'Surname')
    return df

# Create a copy of the original dataframe by selecting all the columns
df_copy = df.select("*")

# Apply the clean_data function to the copy
df_clean = clean_data(df_copy)
```

The `clean_data` function helps ensure the dataset is free of missing values and duplicates while removing unnecessary columns.

Feature engineering

Next, we perform feature engineering by creating dummy columns for the 'Geography' and 'Gender' columns using one-hot encoding.

Python

```
# Import PySpark functions
from pyspark.sql import functions as F

# Create dummy columns for 'Geography' and 'Gender' using one-hot encoding
df_clean = df_clean.select(
    "*",
    F.when(F.col("Geography") == "France",
1).otherwise(0).alias("Geography_France"),
    F.when(F.col("Geography") == "Germany",
1).otherwise(0).alias("Geography_Germany"),
    F.when(F.col("Geography") == "Spain",
1).otherwise(0).alias("Geography_Spain"),
    F.when(F.col("Gender") == "Female",
1).otherwise(0).alias("Gender_Female"),
    F.when(F.col("Gender") == "Male", 1).otherwise(0).alias("Gender_Male")
)
```

```
# Drop the original 'Geography' and 'Gender' columns
df_clean = df_clean.drop("Geography", "Gender")
```

Here, we use one-hot encoding to convert categorical columns into binary dummy columns, making them suitable for machine learning algorithms.

Display cleaned data

Finally, we display the cleaned and feature-engineered dataset using the display function.

Python

```
display(df_clean)
```

This step allows you to inspect the resulting DataFrame with the applied transformations.

Save to lakehouse

Now, we will save the cleaned and feature-engineered dataset to the lakehouse.

Python

```
# Create PySpark DataFrame from Pandas
df_clean.write.mode("overwrite").format("delta").save(f"Tables/churn_data_clean")
print(f"Spark dataframe saved to delta table: churn_data_clean")
```

Here, we take the cleaned and transformed PySpark DataFrame, `df_clean`, and save it as a Delta table named "churn_data_clean" in the lakehouse. We use the Delta format for efficient versioning and management of the dataset. The `mode("overwrite")` ensures that any existing table with the same name is overwritten, and a new version of the table is created.

Create test and training datasets

Next, we will create the test and training datasets from the cleaned and feature-engineered data.

In the provided code section, we load a cleaned and feature-engineered dataset from the lakehouse using Delta format, split it into training and testing sets with an 80-20 ratio, and prepare the data for machine learning. This preparation involves importing the `VectorAssembler` from PySpark ML to combine feature columns into a single "features" column. Subsequently, we use the `VectorAssembler` to transform the training and testing datasets, resulting in `train_data` and `test_data` DataFrames that contain the target variable "Exited" and the feature vectors. These datasets are now ready for use in building and evaluating machine learning models.

Python

```
# Import the necessary library for feature vectorization
from pyspark.ml.feature import VectorAssembler

# Load the cleaned and feature-engineered dataset from the lakehouse
df_final = spark.read.format("delta").load("Tables/churn_data_clean")

# Train-Test Separation
train_raw, test_raw = df_final.randomSplit([0.8, 0.2], seed=41)

# Define the feature columns (excluding the target variable 'Exited')
feature_cols = [col for col in df_final.columns if col != "Exited"]

# Create a VectorAssembler to combine feature columns into a single
# 'features' column
featurizer = VectorAssembler(inputCols=feature_cols, outputCol="features")

# Transform the training and testing datasets using the VectorAssembler
train_data = featurizer.transform(train_raw)[["Exited", "features"]]
test_data = featurizer.transform(test_raw)[["Exited", "features"]]
```

Train baseline model

Using the featurized data, we'll train a baseline machine learning model, configure MLflow for experiment tracking, define a prediction function for metrics calculation, and finally, view and log the resulting ROC AUC score.

Set logging level

Here, we configure the logging level to suppress unnecessary output from the Synapse.ml library, keeping the logs cleaner.

Python

```
import logging

logging.getLogger('synapse.ml').setLevel(logging.ERROR)
```

Configure MLflow

In this section, we configure MLflow for experiment tracking. We set the experiment name to "automl_sample" to organize the runs. Additionally, we enable automatic logging, ensuring that model parameters, metrics, and artifacts are automatically logged to MLflow.

Python

```
import mlflow

# Set the MLflow experiment to "automl_sample" and enable automatic logging
mlflow.set_experiment("automl_sample")
mlflow.autolog(exclusive=False)
```

Train and evaluate the model

Finally, we train a LightGBMClassifier model on the provided training data. The model is configured with the necessary settings for binary classification and imbalance handling. We then use this trained model to make predictions on the test data. We extract the predicted probabilities for the positive class and the true labels from the test data. Afterward, we calculate the ROC AUC score using sklearn's `roc_auc_score` function.

Python

```
from synapse.ml.lightgbm import LightGBMClassifier
from sklearn.metrics import roc_auc_score

# Assuming you have already defined 'train_data' and 'test_data'

with mlflow.start_run(run_name="default") as run:
    # Create a LightGBMClassifier model with specified settings
    model = LightGBMClassifier(objective="binary", featuresCol="features",
                                labelCol="Exited")

    # Fit the model to the training data
    model = model.fit(train_data)

    # Get the predictions
    predictions = model.transform(test_data)
```

```

# Extract the predicted probabilities for the positive class
y_pred = predictions.select("probability").rdd.map(lambda x: x[0])
[1]).collect()

# Extract the true labels from the 'test_data' DataFrame
y_true = test_data.select("Exited").rdd.map(lambda x: x[0]).collect()

# Compute the ROC AUC score
roc_auc = roc_auc_score(y_true, y_pred)

# Log the ROC AUC score with MLflow
mlflow.log_metric("ROC_AUC", roc_auc)

# Print or log the ROC AUC score
print("ROC AUC Score:", roc_auc)

```

From here, we can see that our resulting model achieves a ROC AUC score of 84%.

Create an AutoML trial with FLAML

In this section, we'll create an AutoML trial using the FLAML package, configure the trial settings, convert the Spark dataset to a Pandas on Spark dataset, run the AutoML trial, and view the resulting metrics.

Configure the AutoML trial

Here, we import the necessary classes and modules from the FLAML package and create an instance of AutoML, which will be used to automate the machine learning pipeline.

Python

```

# Import the AutoML class from the FLAML package
from flaml import AutoML
from flaml.automl.spark.utils import to_pandas_on_spark

# Create an AutoML instance
automl = AutoML()

```

Configure settings

In this section, we define the configuration settings for the AutoML trial.

Python

```
# Define AutoML settings
settings = {
    "time_budget": 250,           # Total running time in seconds
    "metric": 'roc_auc',         # Optimization metric (ROC AUC in this case)
    "task": 'classification',   # Task type (classification)
    "log_file_name": 'flaml_experiment.log', # FLAML log file
    "seed": 41,                  # Random seed
    "force_cancel": True,        # Force stop training once time_budget is
                                 used up
    "mlflow_exp_name": "automl_sample"      # MLflow experiment name
}
```

Convert to Pandas on Spark

To run AutoML with a Spark-based dataset, we need to convert it to a Pandas on Spark dataset using the `to_pandas_on_spark` function. This enables FLAML to work with the data efficiently.

Python

```
# Convert the Spark training dataset to a Pandas on Spark dataset
df_automl = to_pandas_on_spark(train_data)
```

Run the AutoML trial

Now, we execute the AutoML trial. We use a nested MLflow run to track the experiment within the existing MLflow run context. The AutoML trial is performed on the Pandas on Spark dataset (`df_automl`) with the target variable "`Exited`" and the defined settings are passed to the `fit` function for configuration.

Python

```
'''The main flaml automl API'''

with mlflow.start_run(nested=True):
    automl.fit(dataframe=df_automl, label='Exited', isUnbalance=True,
**settings)
```

View resulting metrics

In this final section, we retrieve and display the results of the AutoML trial. These metrics provide insights into the performance and configuration of the AutoML model on the given dataset.

Python

```
# Retrieve and display the best hyperparameter configuration and metrics
print('Best hyperparameter config:', automl.best_config)
print('Best ROC AUC on validation data: {:.4g}'.format(1 -
automl.best_loss))
print('Training duration of the best run: {:.4g} s'.format(automl.best_config_train_time))
```

Parallelize your AutoML trial with Apache Spark

In scenarios where your dataset can fit into a single node and you want to leverage the power of Spark for running multiple parallel AutoML trials simultaneously, you can follow these steps:

Convert to Pandas dataframe

To enable parallelization, your data must first be converted into a Pandas DataFrame.

Python

```
pandas_df = train_raw.toPandas()
```

Here, we convert the `train_raw` Spark DataFrame into a Pandas DataFrame named `pandas_df` to make it suitable for parallel processing.

Configure parallelization settings

Set `use_spark` to `True` to enable Spark-based parallelism. By default, FLAML will launch one trial per executor. You can customize the number of concurrent trials by using the `n_concurrent_trials` argument.

Python

```
settings = {
    "time_budget": 250,           # Total running time in seconds
    "metric": 'roc_auc',         # Optimization metric (ROC AUC in this
```

```
        case):
            "task": 'classification',      # Task type (classification)
            "seed": 41,                   # Random seed
            "use_spark": True,           # Enable Spark-based parallelism
            "n_concurrent_trials": 3,    # Number of concurrent trials to run
            "force_cancel": True,        # Force stop training once time_budget is
            used up
            "mlflow_exp_name": "automl_sample" # MLflow experiment name
        }
```

In these settings, we specify that we want to utilize Spark for parallelism by setting `use_spark` to `True`. We also set the number of concurrent trials to 3, meaning that three trials will run in parallel on Spark.

To learn more about how to parallelize your AutoML trials, you can visit the [FLAML documentation for parallel Spark jobs](#).

Run the AutoML trial in parallel

Now, we will run the AutoML trial in parallel with the specified settings. We will use a nested MLflow run to track the experiment within the existing MLflow run context.

Python

```
'''The main FLAML AutoML API'''
with mlflow.start_run(nested=True, run_name="parallel_trial"):
    automl.fit(dataframe=pandas_df, label='Exited', **settings)
```

This will now execute the AutoML trial with parallelization enabled. The `dataframe` argument is set to the Pandas DataFrame `pandas_df`, and other settings are passed to the `fit` function for parallel execution.

View metrics

After running the parallel AutoML trial, retrieve and display the results, including the best hyperparameter configuration, ROC AUC on the validation data, and the training duration of the best-performing run.

Python

```
''' retrieve best config'''
print('Best hyperparameter config:', automl.best_config)
print('Best roc_auc on validation data: {:.4g}'.format(1-automl.best_loss))
```

```
print('Training duration of best run: {:.4g}\n'.format(automl.best_config_train_time))
```

Next steps

- Learn about AutoML in Fabric
 - Visualize the results of your AutoML trial
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Training visualizations (preview)

Article • 03/26/2024

A hyperparameter trial or AutoML trial searches for the optimal parameters for a machine learning model. Each trial consists of multiple runs, where each run evaluates a specific parameter combination. Users can monitor these runs using ML experiment items in Fabric.

The `flaml.visualization` module offers functions to plot and compare the runs in FLAML. Users can use Plotly to interact with their AutoML experiment plots. To use these functions, users need to input their optimized `flaml.AutoML` or `flaml.tune.tune.ExperimentAnalysis` object.

This article teaches you how to use the `flaml.visualization` module to analyze and explore your AutoML trial results. You can follow the same steps for your hyperparameter trial as well.

ⓘ Important

This feature is in [preview](#).

Create an AutoML trial

AutoML offers a suite of automated processes that can identify the best machine learning pipeline for your dataset, making the entire modeling process more straightforward and often more accurate. In essence, it saves you the trouble of hand-tuning different models and hyperparameters.

In the code cell below, we will:

1. Load the Iris dataset.
2. Split the data into training and test sets.
3. Initiate an AutoML trial to fit our training data.
4. Explore the results of our AutoML trial with the visualizations from `flaml.visualization`.

Python

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from flaml import AutoML
```

```
# Load the Iris data and split it into train and test sets
x, y = load_iris(return_X_y=True, as_frame=True)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,
random_state=7654321)

# Create an AutoML instance and set the parameters
automl = AutoML()
automl_settings = {
    "time_budget": 10, # Time limit in seconds
    "task": "classification", # Type of machine learning task
    "log_file_name": "aml_iris.log", # Name of the log file
    "metric": "accuracy", # Evaluation metric
    "log_type": "all", # Level of logging
}
# Fit the AutoML instance on the training data
automl.fit(X_train=x_train, y_train=y_train, **automl_settings)
```

Visualize the experiment results

Once you run an AutoML trial, you need to visualize the outcomes to analyze how well the models performed and how they behaved. In this part of our documentation, we show you how to use the built-in utilities in the FLAML library for this purpose.

Import visualization module

To access these visualization utilities, we run the following import command:

Python

```
import flaml.visualization as fviz
```

Optimization history

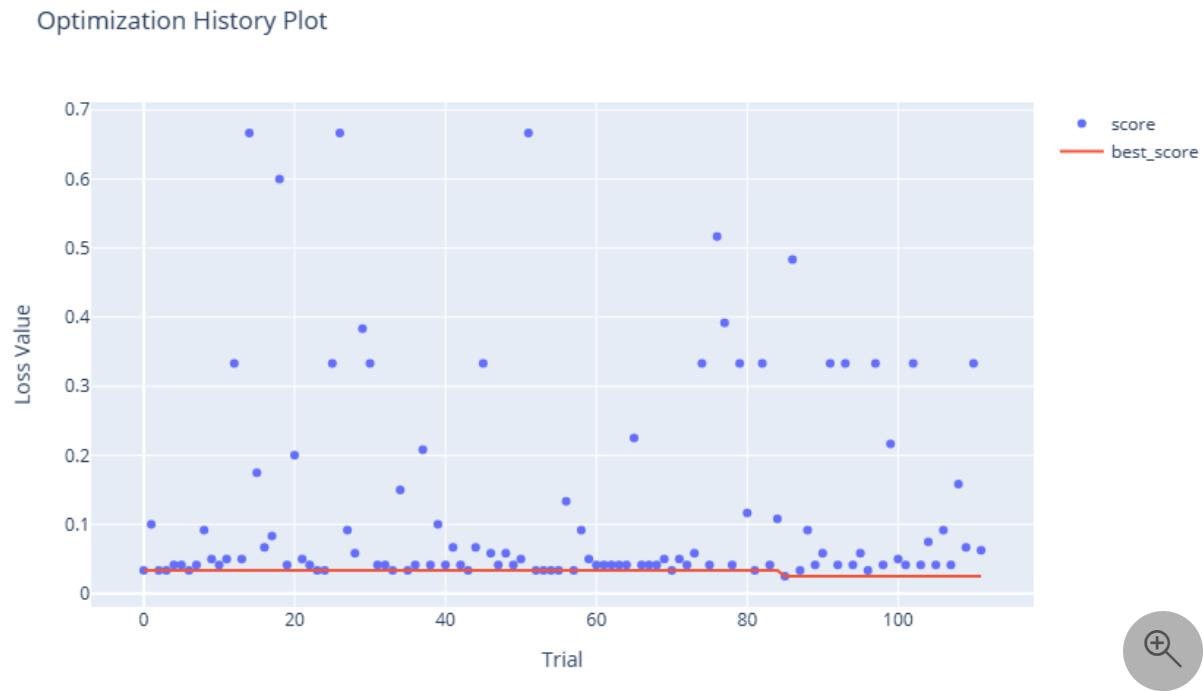
An optimization history plot typically has the number of trials/iterations on the x-axis and a performance metric (like accuracy, RMSE, etc.) on the y-axis. As the number of trials increases, you would see a line or scatter plot indicating the performance of each trial.

Python

```
fig = fviz.plot_optimization_history(automl)
# or
```

```
fig = fviz.plot(automl, "optimization_history")
fig.show()
```

Here is the resulting plot:



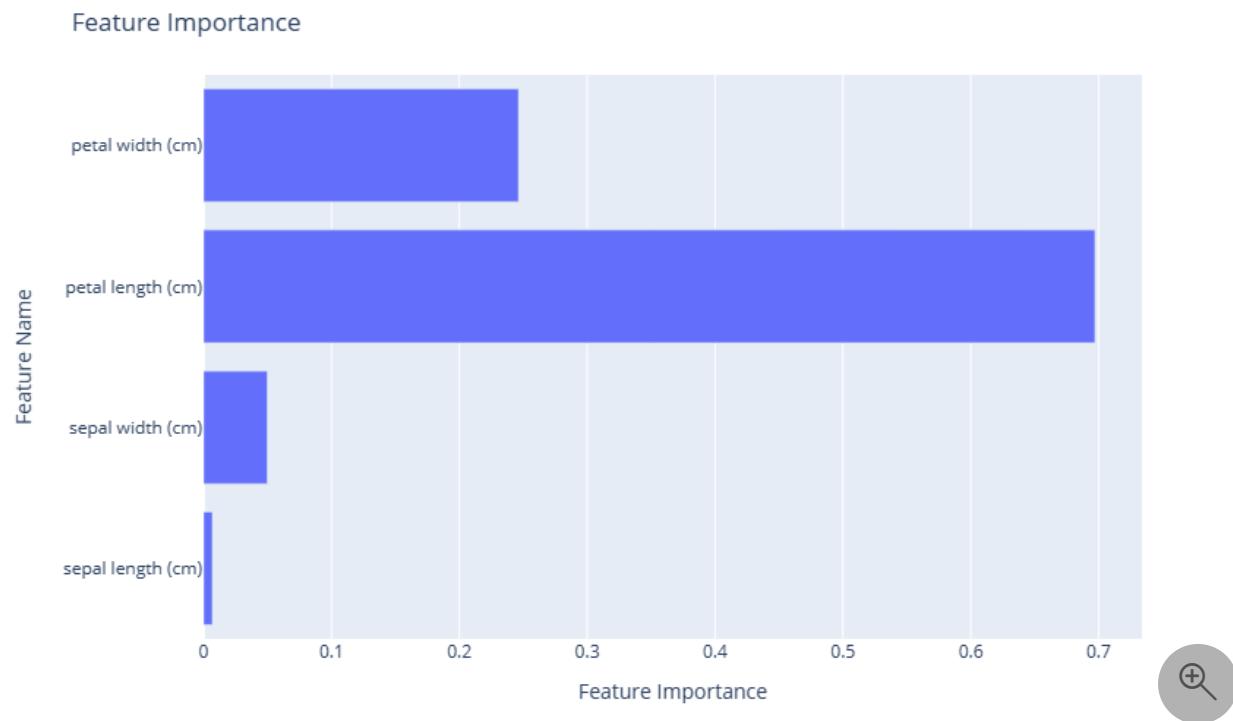
Feature importance

A feature importance plot is a powerful visualization tool that allows you to understand the significance of different input features in determining the predictions of a model.

Python

```
fig = fviz.plot_feature_importance(automl)
# or
fig = fviz.plot(automl, "feature_importance")
fig.show()
```

Here is the resulting plot:



Parallel coordinate plot

A parallel coordinate plot is a visualization tool that represents multi-dimensional data by drawing multiple vertical lines (axes) corresponding to variables or hyperparameters, with data points plotted as connected lines across these axes. In the context of an AutoML or tuning experiment, it's instrumental in visualizing and analyzing the performance of different hyperparameter combinations. By tracing the paths of high-performing configurations, one can discern patterns or trends in hyperparameter choices and their interactions. This plot aids in understanding which combinations lead to optimal performance, pinpointing potential areas for further exploration, and identifying any trade-offs between different hyperparameters.

This utility takes the following other arguments:

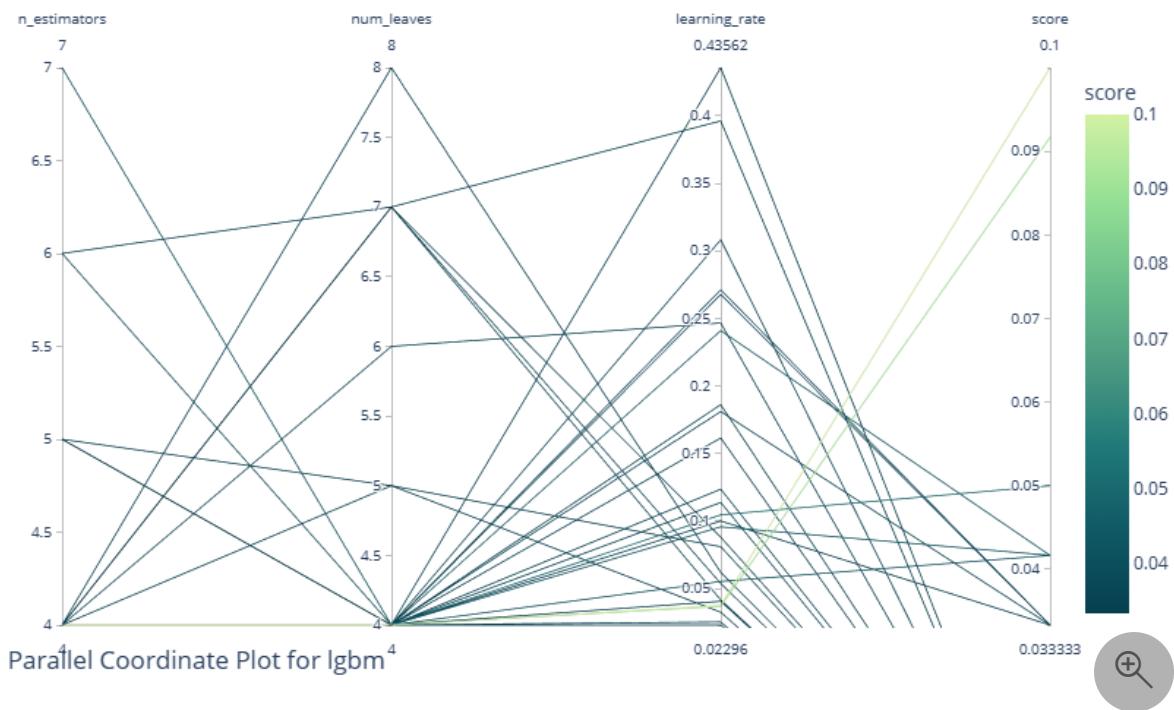
- `learner`: Specify the learner you intend to study in the experiment. This parameter is only applicable for AutoML experiment results. By leaving this blank, the system chooses the best learner in the whole experiment.
- `params`: A list to specify which hyperparameter to display. By leaving this blank, the system displays all the available hyperparameters.

Python

```
fig = fviz.plot_parallel_coordinate(automl, learner="lgbm", params=
["n_estimators", "num_leaves", "learning_rate"])
# or
fig = fviz.plot(automl, "parallel_coordinate", learner="lgbm", params=
```

```
["n_estimators", "num_leaves", "learning_rate"])
fig.show()
```

Here is the resulting plot:



Contour plot

A contour plot visualizes three-dimensional data in two dimensions, where the x and y axes represent two hyperparameters, and the contour lines or filled contours depict levels of a performance metric (for example, accuracy or loss). In the context of an AutoML or tuning experiment, a contour plot is beneficial for understanding the relationship between two hyperparameters and their combined effect on model performance.

By examining the density and positioning of the contour lines, one can identify regions of hyperparameter space where performance is optimized, ascertain potential trade-offs between hyperparameters, and gain insights into their interactions. This visualization helps refine the search space and tuning process.

This utility also takes the following arguments:

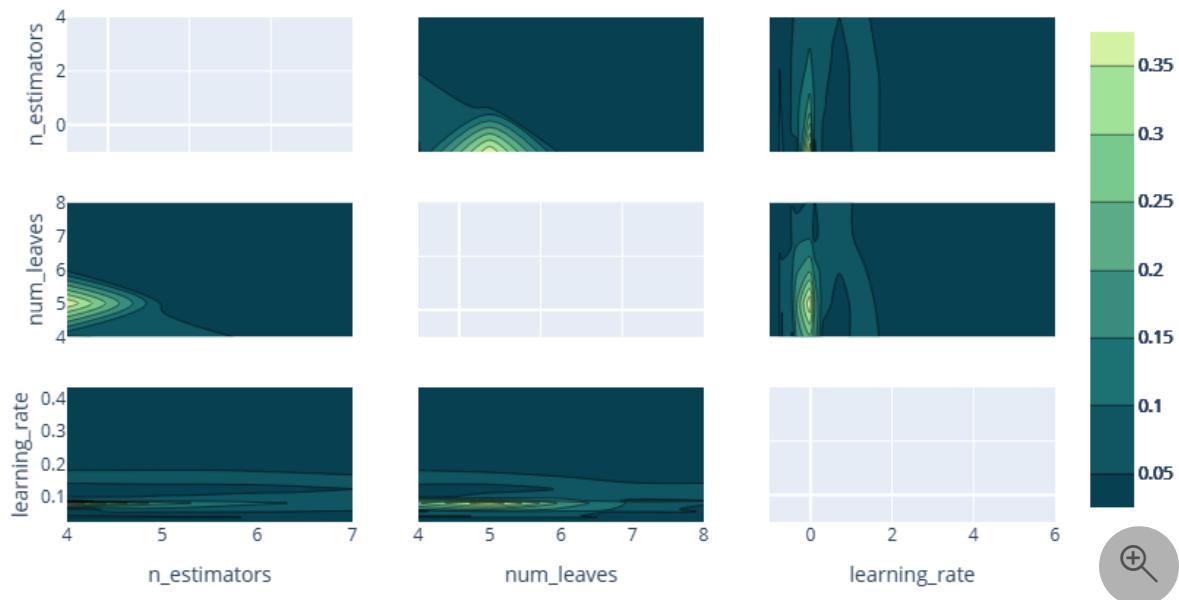
- `learner`: Specify the learner you intend to study in the experiment. This parameter is only applicable for AutoML experiment results. By leaving this blank, the system chooses the best learner in the whole experiment.

- `params`: A list to specify which hyperparameter to display. By leaving this blank, the system displays all the available hyperparameters.

Python

```
fig = fviz.plot_contour(automl, learner="lgbm", params=["n_estimators",
"num_leaves", "learning_rate"])
# or
fig = fviz.plot(automl, "contour", learner="lgbm", params=["n_estimators",
"num_leaves", "learning_rate"])
fig.show()
```

Here is the resulting plot:



Empirical distribution function

An empirical distribution function (EDF) plot, often visualized as a step function, represents the cumulative probability of data points being less than or equal to a particular value. Within an AutoML or tuning experiment, an EDF plot can be employed to visualize the distribution of model performances across different hyperparameter configurations.

By observing the steepness or flatness of the curve at various points, one can understand the concentration of good or poor model performances, respectively. This visualization offers insights into the overall efficacy of the tuning process, highlighting

whether most of the attempted configurations are yielding satisfactory results or if only a few configurations stand out.

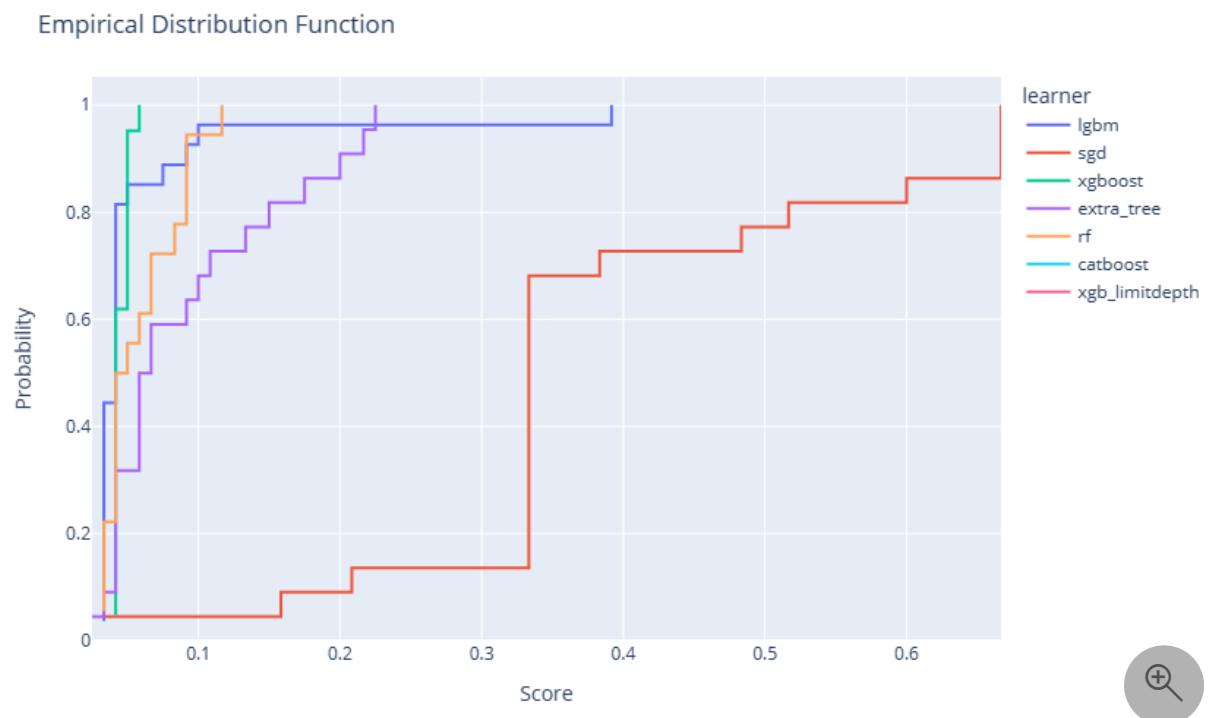
ⓘ Note

For AutoML experiments, multiple models will be applied during training. The trials of each learner are represented as an optimization series. For hyperparameter tuning experiments, there will be only a single learner that is evaluated. However, you can provide additional tuning experiments to see the trends across each learner.

Python

```
fig = fviz.plot_edf(automl)
# or
fig = fviz.plot(automl, "edf")
fig.show()
```

Here is the resulting plot:



Timeline plot

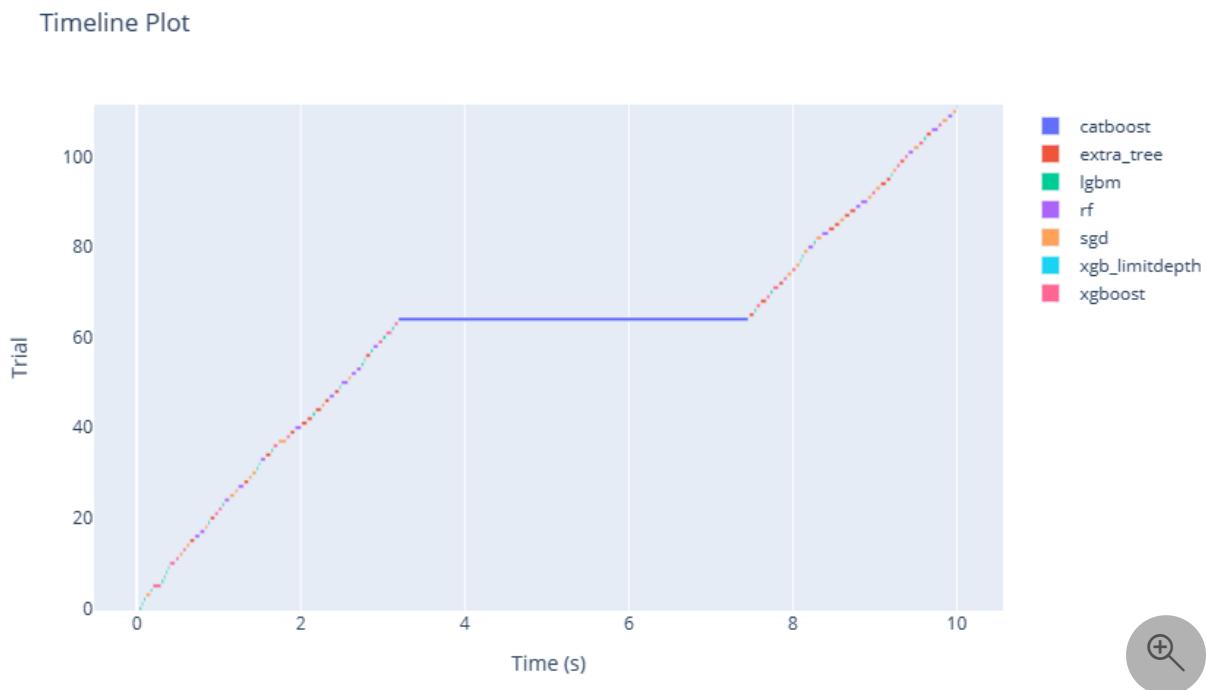
A timeline plot, often represented as a Gantt chart or a sequence of bars, visualizes the start, duration, and completion of tasks over time. In the context of an AutoML or tuning experiment, a timeline plot can showcase the progression of various model evaluations

and their respective durations, plotted against time. By observing this plot, users can grasp the efficiency of the search process, identify any potential bottlenecks or idle periods, and understand the temporal dynamics of different hyperparameter evaluations.

Python

```
fig = fviz.plot_timeline(automl)
# or
fig = fviz.plot(automl, "timeline")
fig.show()
```

Here is the resulting plot:



Slice plot

Plot the parameter relationship as slice plot in a study.

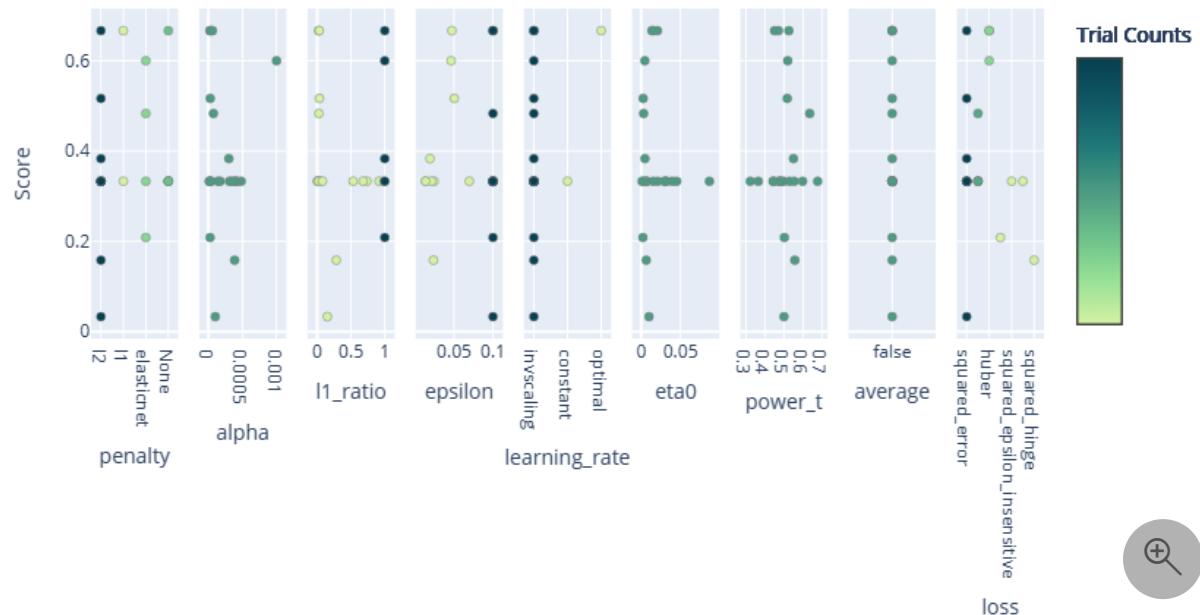
This utility also takes the following arguments:

- `learner`: Specify the learner you intend to study in the experiment. This parameter is only applicable for AutoML experiment results. By leaving this blank, the system chooses the best learner in the whole experiment.
- `params`: A list to specify which hyperparameter to display. By leaving this blank, the system displays all the available hyperparameters.

Python

```
fig = fviz.plot_slice(automl, learner="sgd")
# or
fig = fviz.plot(automl, "slice", learner="sgd")
fig.show()
```

Here is the resulting plot:



Hyperparameter importance

A hyperparameter importance plot visually ranks hyperparameters based on their influence on model performance in an AutoML or tuning experiment. Displayed typically as a bar chart, it quantifies the impact of each hyperparameter on the target metric. By examining this plot, practitioners can discern which hyperparameters are pivotal in determining model outcomes and which ones have minimal effect.

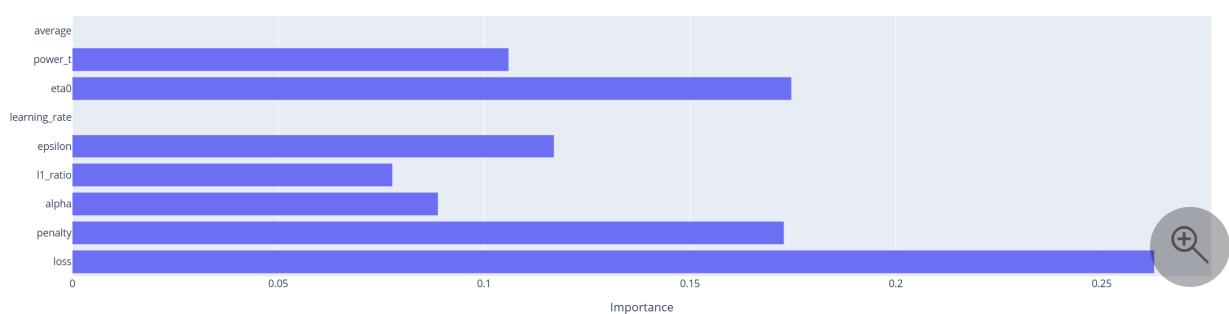
This utility also takes the following arguments:

- `learner`: Specify the learner you intend to study in the experiment. This parameter is only applicable for AutoML experiment results. By leaving this blank, the system chooses the best learner in the whole experiment.
- `params`: A list to specify which hyperparameter to display. By leaving this blank, the system displays all the available hyperparameters.

Python

```
fig = fviz.plot_param_importance(automl, learner="sgd")
# or
fig = fviz.plot(automl, "param_importance", learner="sgd")
fig.show()
```

Here is the resulting plot:



Next steps

- Tune a SynapseML Spark LightGBM model

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Machine learning experiments in Microsoft Fabric

Article • 01/17/2025

A machine learning *experiment* is the primary unit of organization and control for all related machine learning runs. A *run* corresponds to a single execution of model code. In [MLflow](#), tracking is based on experiments and runs.

Machine learning experiments allow data scientists to log parameters, code versions, metrics, and output files when running their machine learning code. Experiments also let you visualize, search for, and compare runs, as well as download run files and metadata for analysis in other tools.

In this article, you learn more about how data scientists can interact with and use machine learning experiments to organize their development process and to track multiple runs.

Prerequisites

- A Power BI Premium subscription. If you don't have one, see [How to purchase Power BI Premium](#).
- A Power BI Workspace with assigned premium capacity.

Create an experiment

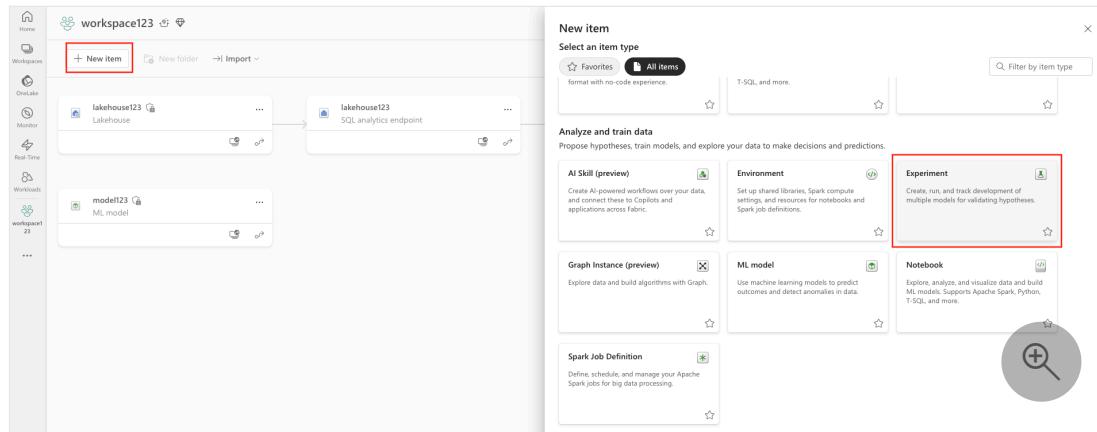
You can create a machine learning experiment directly from the fabric user interface (UI) or by writing code that uses the MLflow API.

Create an experiment using the UI

To create a machine learning experiment from the UI:

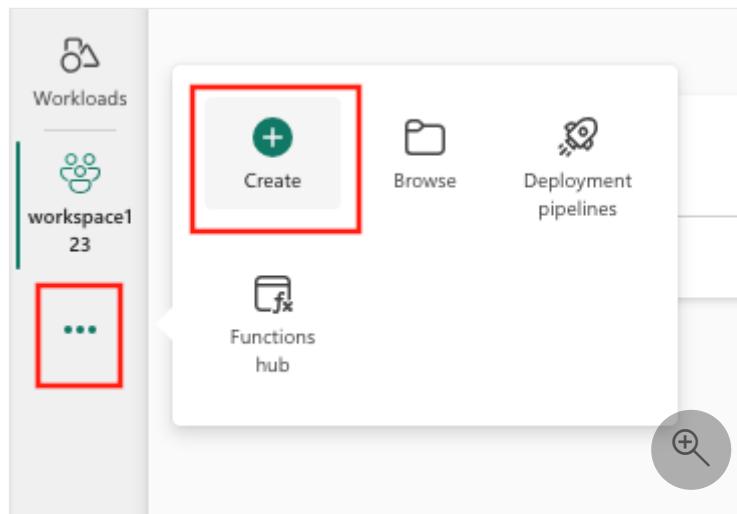
1. Create a new workspace or select an existing one.
2. You can create a new item through the workspace or by using Create.
 - a. Workspace:
 - i. Select your workspace.
 - ii. Select **New item**.

iii. Select Experiment under Analyze and train data.

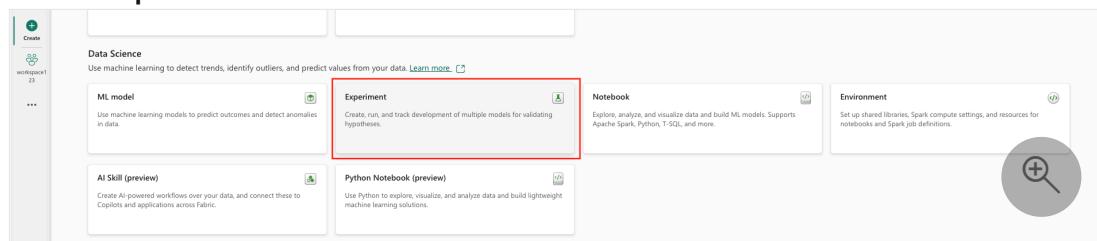


b. Create button:

i. Select Create, which can be found in ... from the vertical menu.



ii. Select Experiment under Data Science.



3. Provide an experiment name and select **Create**. This action creates an empty experiment within your workspace.

After creating the experiment, you can start adding runs to track run metrics and parameters.

Create an experiment using the MLflow API

You can also create a machine learning experiment directly from your authoring experience using the `mlflow.create_experiment()` or `mlflow.set_experiment()` APIs. In the following code, replace `<EXPERIMENT_NAME>` with your experiment's name.

Python

```

import mlflow

# This will create a new experiment with the provided name.
mlflow.create_experiment("<EXPERIMENT_NAME>")

# This will set the given experiment as the active experiment.
# If an experiment with this name does not exist, a new experiment with this
# name is created.
mlflow.set_experiment("<EXPERIMENT_NAME>")

```

Manage runs within an experiment

A machine learning experiment contains a collection of runs for simplified tracking and comparison. Within an experiment, a data scientist can navigate across various runs and explore the underlying parameters and metrics. Data scientists can also compare runs within a machine learning experiment to identify which subset of parameters yield a desired model performance.

Track runs

A machine learning run corresponds to a single execution of model code.

Each run includes the following information:

- **Source:** Name of the notebook that created the run.
- **Registered Version:** Indicates if the run was saved as a machine learning model.
- **Start date:** Start time of the run.
- **Status:** Progress of the run.

- **Hyperparameters:** Hyperparameters saved as key-value pairs. Both keys and values are strings.
- **Metrics:** Run metrics saved as key-value pairs. The value is numeric.
- **Output files:** Output files in any format. For example, you can record images, environment, models, and data files.
- **Tags:** Metadata as key-value pairs to runs.

View recent runs

You can also view recent runs for an experiment by selecting **Run list**. This view allows you to keep track of recent activity, quickly jump to the related Spark application, and apply filters based on the run status.

Run name	Start time	Duration	Status	accuracy_score	training_accuracy	training_f1_score	boosting_type	colsample_bytree	leam
lgbm_sm	1/14/2025 4:04 PM	15s	Completed	0.8675	—	—	gbdt	1.0	0.07
rfc2_sm	1/14/2025 4:04 PM	19s	Completed	—	0.8818897637795...	0.8817979254065...	—	—	—
rfc1_sm	1/14/2025 4:04 PM	26s	Completed	—	0.8256692913385...	0.8255585417240...	—	—	—
sad_spoon_0gvhpwy4j	1/14/2025 4:03 PM	10s	Completed	—	—	—	—	—	—

Metric comparison Performance Training

Select runs to compare their metrics
Select two or more runs from the list above to get a visual comparison of their metrics here. [Learn more](#)

Compare and filter runs

To compare and evaluate the quality of your machine learning runs, you can compare the parameters, metrics, and metadata between selected runs within an experiment.

Apply tags to runs

MLflow tagging for experiment runs allows users to add custom metadata in the form of key-value pairs to their runs. These tags help categorize, filter, and search for runs based on specific attributes, making it easier to manage and analyze experiments within the MLflow platform. Users can utilize tags to label runs with information such as model types, parameters, or any relevant identifiers, enhancing the overall organization and traceability of experiments.

This code snippet starts an MLflow run, logs some parameters and metrics, and adds tags to categorize and provide additional context for the run.

Python

```
import mlflow
import mlflow.sklearn
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.datasets import fetch_california_housing

# Autologging
mlflow.autolog()

# Load the California housing dataset
data = fetch_california_housing(as_frame=True)
X = data.data
y = data.target

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Start an MLflow run
with mlflow.start_run() as run:

    # Train the model
    model = LinearRegression()
    model.fit(X_train, y_train)

    # Predict and evaluate
    y_pred = model.predict(X_test)

    # Add tags
    mlflow.set_tag("model_type", "Linear Regression")
    mlflow.set_tag("dataset", "California Housing")
    mlflow.set_tag("developer", "Bob")
```

Once the tags are applied, you can then view the results directly from the inline MLflow widget or from the run details page.

The screenshot shows the MLflow UI interface. On the left, there's a sidebar with 'Tagging-Demo' selected. The main area displays a table of run parameters and tags. The table includes columns for parameter names and values. A separate 'Add or edit tags' dialog is open on the right, listing five tags with their corresponding values. A magnifying glass icon is at the bottom right of the dialog.

Name	Value
estimator_name	LinearRegression
estimator_class	sklearn.linear_model.base.LinearR egression
model_type	Linear Regression
dataset	California Housing
developer	Your Name

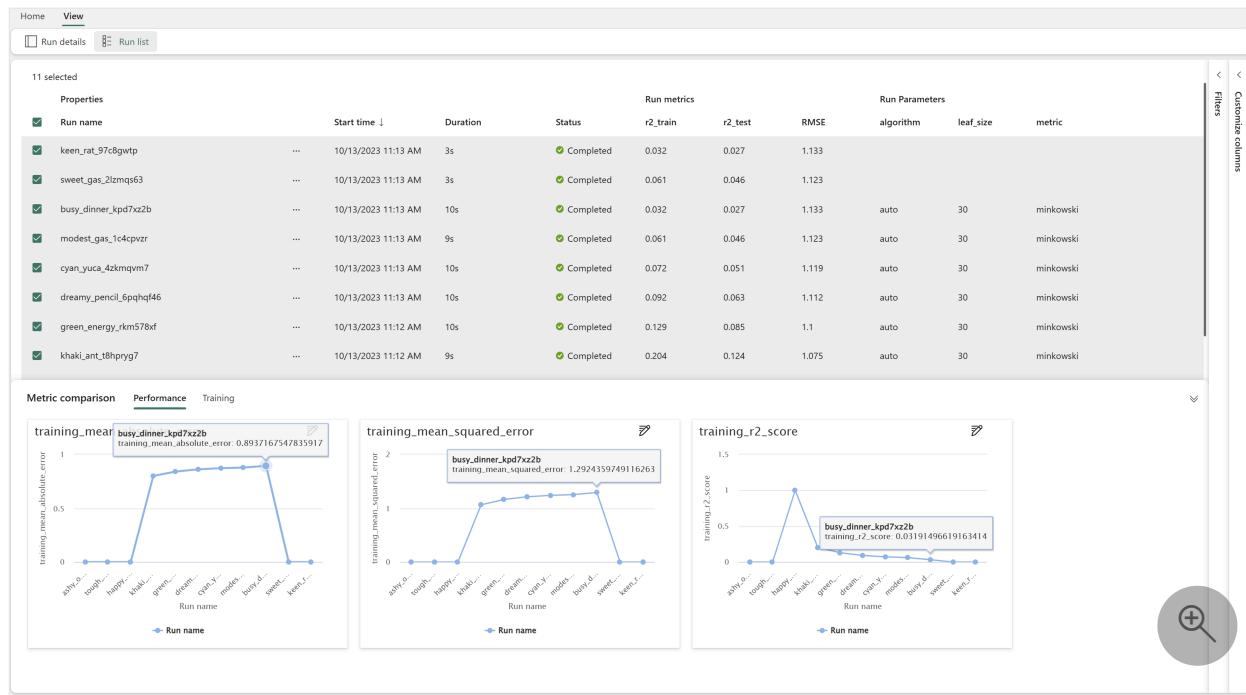
⚠️ Warning

Warning: Limitations on Applying Tags to MLflow Experiment Runs in Fabric

- Non-Empty Tags:** Tag names or values cannot be empty. If you attempt to apply a tag with an empty name or value, the operation will fail.
- Tag Names:** Tag names can be up to 250 characters in length.
- Tag Values:** Tag values can be up to 5000 characters in length.
- Restricted Tag Names:** Tag names that start with certain prefixes are not supported. Specifically, tag names beginning with `synapseml`, `mlflow`, or `trident` are restricted and will not be accepted.

Visually compare runs

You can visually compare and filter runs within an existing experiment. Visual comparison allows you to easily navigate between multiple runs and sort across them.



To compare runs:

1. Select an existing machine learning experiment that contains multiple runs.
2. Select the **View** tab and then go to the **Run list** view. Alternatively, you could select the option to **View run list** directly from the **Run details** view.
3. Customize the columns within the table by expanding the **Customize columns** pane. Here, you can select the properties, metrics, tags, and hyperparameters that you would like to see.
4. Expand the **Filter** pane to narrow your results based on certain selected criteria.
5. Select multiple runs to compare their results in the metrics comparison pane. From this pane, you can customize the charts by changing the chart title, visualization type, X-axis, Y-axis, and more.

Compare runs using the MLflow API

Data scientists can also use MLflow to query and search among runs within an experiment. You can explore more MLflow APIs for searching, filtering, and comparing runs by visiting the [MLflow documentation](#).

Get all runs

You can use the MLflow search API `mlflow.search_runs()` to get all runs in an experiment by replacing `<EXPERIMENT_NAME>` with your experiment name or `<EXPERIMENT_ID>` with your experiment ID in the following code:

Python

```
import mlflow

# Get runs by experiment name:
mlflow.search_runs(experiment_names=[ "<EXPERIMENT_NAME>" ])

# Get runs by experiment ID:
mlflow.search_runs(experiment_ids=[ "<EXPERIMENT_ID>" ])
```

💡 Tip

You can search across multiple experiments by providing a list of experiment IDs to the `experiment_ids` parameter. Similarly, providing a list of experiment names to the `experiment_names` parameter will allow MLflow to search across multiple experiments. This can be useful if you want to compare across runs within different experiments.

Order and limit runs

Use the `max_results` parameter from `search_runs` to limit the number of runs returned. The `order_by` parameter allows you to list the columns to order by and can contain an optional `DESC` or `ASC` value. For instance, the following example returns the last run of an experiment.

Python

```
mlflow.search_runs(experiment_ids=[ "1234-5678-90AB-CDEFG" ], max_results=1,
order_by=[ "start_time DESC" ])
```

Compare runs within a Fabric notebook

You can use the MLFlow authoring widget within Fabric notebooks to track MLflow runs generated within each notebook cell. The widget allows you to track your runs, associated metrics, parameters, and properties right down to the individual cell level.

To obtain a visual comparison, you can also switch to the **Run comparison** view. This view presents the data graphically, aiding in the quick identification of patterns or deviations across different runs.

```

1  from sklearn.neighbors import KNeighborsRegressor
2  from sklearn.metrics import accuracy_score
3  from sklearn.metrics import mean_squared_error
4  from math import sqrt
5  from mlflow.models import infer_signature
6
7  for k in range(1,100,20):
8      with mlflow.start_run() as run:
9          # Train
10         knn = KNeighborsRegressor(n_neighbors=k)
11         knn.fit(X_train, y_train)
12         k1 = knn.score(X_train, y_train)
13         k2 = knn.score(X_test, y_test)
14         # R2 value
15         mlflow.log_metric("r2_train", round(k1,3))
16         mlflow.log_metric("r2_test", round(k2,3))
17         # RMSE
18         y_pred = knn.predict(X_test)
19         mlflow.log_metric("RMSE", round(sqrt(mean_squared_error(y_test,y_pred)),3))
20

```

52 sec -Command executed in 53 sec 293 ms by on 11/13/2023 AM 10/13/23 PySpark (Python) ✓

Run list Run comparison

Run name	Start time	Duration	Status	Experiment name	r2_test	mean_squared_error...	RMSE	n_neighbors
happy_truck_cgg3lw2	10/13/2023 11:12 AM	9s	Finished	housing-experiment	-0.237	1.633830019184593	1.278	1
khaki_ant_t8hpnyg7	10/13/2023 11:12 AM	9s	Finished	housing-experiment	0.124	1.1566989824095744	1.075	21
green_energy_rkm578xf	10/13/2023 11:12 AM	9s	Finished	housing-experiment	0.085	1.2090210667016414	1.1	41
dreamy_pencil_6pqhqf...	10/13/2023 11:13 AM	9s	Finished	housing-experiment	0.063	1.237159849843777	1.112	61
cyan_yuca_d2kmqv7	10/13/2023 11:13 AM	10s	Finished	housing-experiment	0.051	1.2530128905403495	1.119	81

Session ready Save option: Automatic Selected Cell 12 of 18 cells

Save run as a machine learning model

Once a run yields the desired result, you can save the run as a model for enhanced model tracking and for model deployment by selecting **Save as a ML model**.

Properties

Description

Run name: lgbm_sm

Start date: 1/14/2023 4:04 PM

Duration: 15s

Status: Completed

Run details

- Run metrics (1): accuracy_score_X_test: 0.86
- Run parameters (23): boosting_type: gbdt, colsample_bytree: 1.0, learning_rate: 0.07, max_depth: 10, min_child_samples: 20

Save as ML model

Select methods

Create a new ML model

Select an existing ML model

Select folder

ML model name

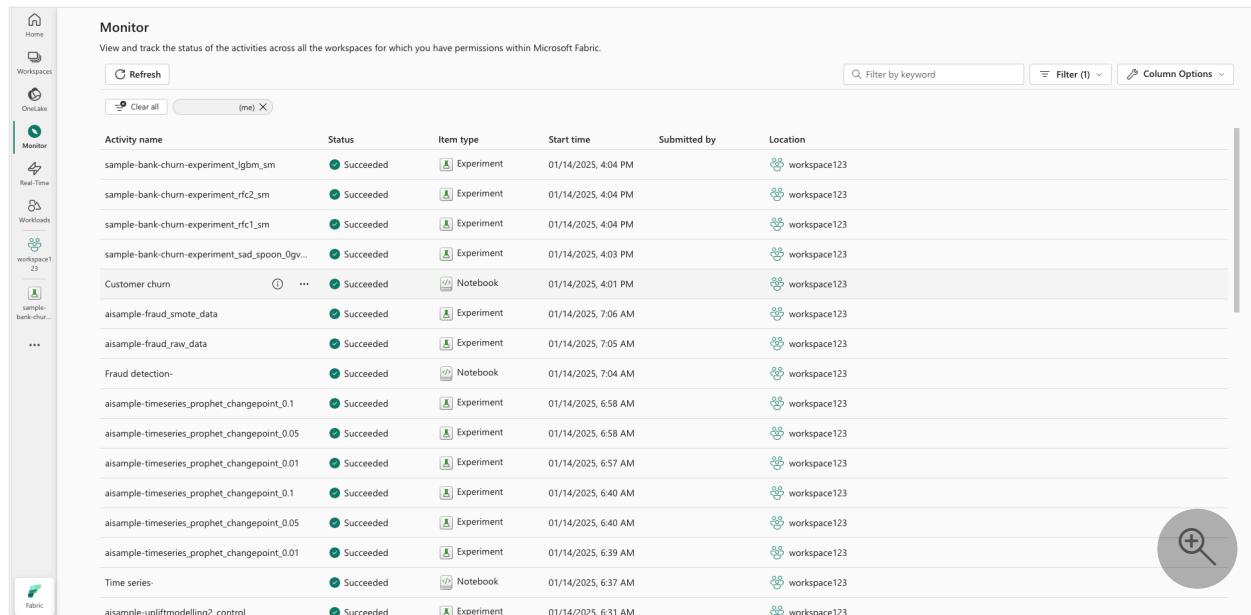
Save Cancel

Monitor ML Experiments (preview)

ML experiments are integrated directly into Monitor. This functionality is designed to provide more insight into your Spark applications and the ML experiments they generate, making it easier to manage and debug these processes.

Track runs from monitor

Users can track experiment runs directly from monitor, providing a unified view of all their activities. This integration includes filtering options, enabling users to focus on experiments or runs created within the last 30 days or other specified periods.



The screenshot shows the Microsoft Fabric Monitor interface. On the left is a sidebar with icons for Home, Workspaces, OneLake, Monitor (which is selected), Real-Time, Workloads, and a workspace named 'workspace1 23'. The main area is titled 'Monitor' and has a sub-instruction 'View and track the status of the activities across all the workspaces for which you have permissions within Microsoft Fabric.' Below this are buttons for Refresh, Clear all, and a search bar '(me) X'. To the right are buttons for Filter by keyword, Filter (1), and Column Options. The main content is a table with columns: Activity name, Status, Item type, Start time, Submitted by, and Location. The table lists various experiment runs, all of which have a status of 'Succeeded'. The 'Item type' column shows mostly 'Experiment' with one 'Notebook'. The 'Start time' column shows dates ranging from 01/14/2025 at 4:03 PM to 6:31 AM. The 'Submitted by' column shows '(me)' and the 'Location' column shows 'workspace123' for all entries. A magnifying glass icon is in the bottom right corner of the table area.

Activity name	Status	Item type	Start time	Submitted by	Location
sample-bank-churn-experiment_lgbm_sm	Succeeded	Experiment	01/14/2025, 4:04 PM	(me)	workspace123
sample-bank-churn-experiment_rf2_sm	Succeeded	Experiment	01/14/2025, 4:04 PM	(me)	workspace123
sample-bank-churn-experiment_rf1_sm	Succeeded	Experiment	01/14/2025, 4:04 PM	(me)	workspace123
sample-bank-churn-experiment_sad_spoon_0gv...	Succeeded	Experiment	01/14/2025, 4:03 PM	(me)	workspace123
Customer churn	...	Succeeded	01/14/2025, 4:01 PM	(me)	workspace123
aisample-fraud_smote_data	Succeeded	Experiment	01/14/2025, 7:06 AM	(me)	workspace123
aisample-fraud_raw_data	Succeeded	Experiment	01/14/2025, 7:05 AM	(me)	workspace123
Fraud detection-	Succeeded	Notebook	01/14/2025, 7:04 AM	(me)	workspace123
aisample-timeseries_prophet_changepoint_0.1	Succeeded	Experiment	01/14/2025, 6:58 AM	(me)	workspace123
aisample-timeseries_prophet_changepoint_0.05	Succeeded	Experiment	01/14/2025, 6:58 AM	(me)	workspace123
aisample-timeseries_prophet_changepoint_0.01	Succeeded	Experiment	01/14/2025, 6:57 AM	(me)	workspace123
aisample-timeseries_prophet_changepoint_0.1	Succeeded	Experiment	01/14/2025, 6:40 AM	(me)	workspace123
aisample-timeseries_prophet_changepoint_0.05	Succeeded	Experiment	01/14/2025, 6:40 AM	(me)	workspace123
aisample-timeseries_prophet_changepoint_0.01	Succeeded	Experiment	01/14/2025, 6:39 AM	(me)	workspace123
Time series	Succeeded	Notebook	01/14/2025, 6:37 AM	(me)	workspace123
aisample-upliftmodelling2_control	Succeeded	Experiment	01/14/2025, 6:31 AM	(me)	workspace123

Track related ML Experiment runs from your Spark application

ML Experiment are integrated directly into Monitor, where you can select a specific Spark application and access Item Snapshots. Here, you'll find a list of all the experiments and runs generated by that application.

Related content

- [Learn about MLflow Experiment APIs ↗](#)
- [Track and manage machine learning models](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Machine learning model in Microsoft Fabric

Article • 01/17/2025

A machine learning model is a file trained to recognize certain types of patterns. You train a model over a set of data, and you provide it with an algorithm that uses to reason over and learn from that data set. After you train the model, you can use it to reason over data that it never saw before, and make predictions about that data.

In [MLflow](#), a machine learning model can include multiple model versions. Here, each version can represent a model iteration. In this article, you learn how to interact with ML models to track and compare model versions.

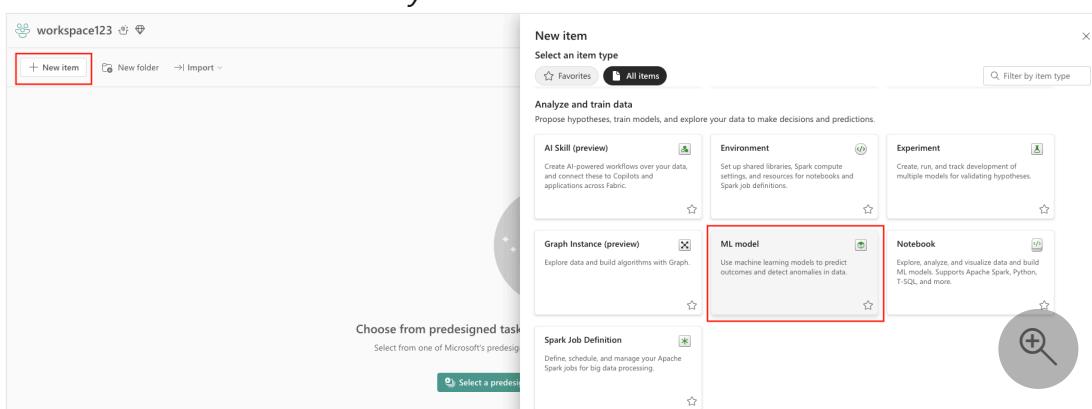
Create a machine learning model

In MLflow, machine learning models include a standard packaging format. This format allows use of those models in various downstream tools, including batch inferencing on Apache Spark. The format defines a convention to save a model in different "flavors" that different downstream tools can understand.

You can directly create a machine learning model from the Fabric UI. The MLflow API can also directly create the model.

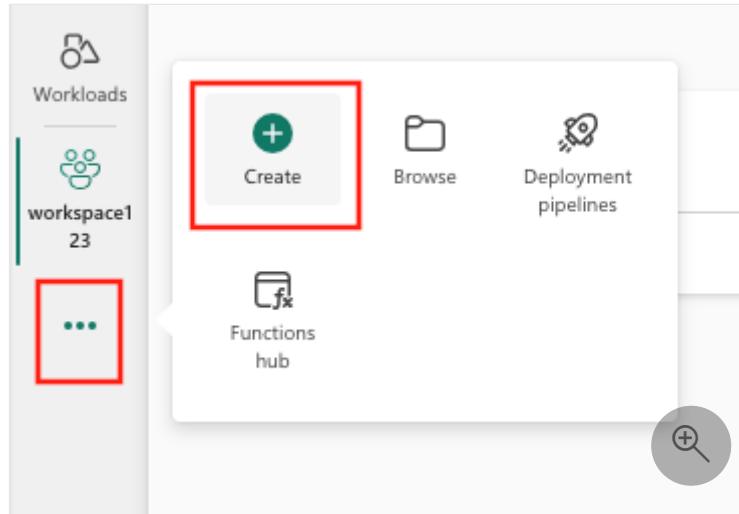
To create a machine learning model from the UI, you can:

1. Create a new data science workspace, or select an existing data science workspace.
2. Create a new workspace or select an existing one.
3. You can create a new item through the workspace or by using Create.
 - a. Workspace:
 - i. Select your workspace.
 - ii. Select **New item**.
 - iii. Select **ML Model** under *Analyze and train data*.

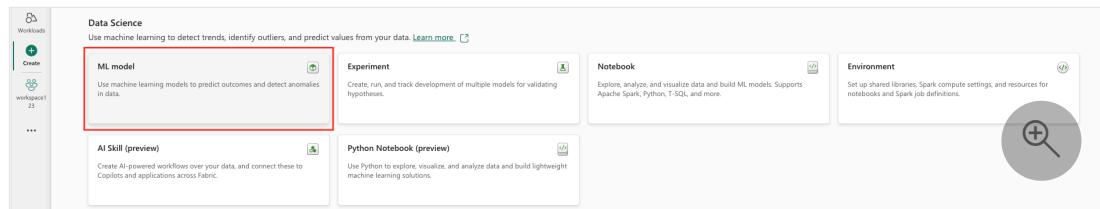


b. Create button:

i. Select **Create**, which can be found in ... from the vertical menu.



ii. Select **ML Model** under *Data Science*.



4. After model creation, you can start adding model versions to track run metrics and parameters. Register or save experiment runs to an existing model.

You can also create a machine learning model directly from your authoring experience with the `mlflow.register_model()` API. If a registered machine learning model with the given name doesn't exist, the API creates it automatically.

```
Python

import mlflow

model_uri = "runs:/{}//model-uri-name".format(run.info.run_id)
mv = mlflow.register_model(model_uri, "model-name")

print("Name: {}".format(mv.name))
print("Version: {}".format(mv.version))
```

Manage versions within a machine learning model

A machine learning model contains a collection of model versions for simplified tracking and comparison. Within a model, a data scientist can navigate across various model versions to explore the underlying parameters and metrics. Data scientists can also make

comparisons across model versions to identify whether or not newer models might yield better results.

Track machine learning models

A machine learning model version represents an individual model that is registered for tracking.

The screenshot shows the MLflow Model Registry interface. At the top, there's a navigation bar with 'Home' and 'View' options, followed by buttons for 'Apply this version', 'Download ML model version', and 'Delete version'. Below this is a list of registered models: 'RandomForestRegression...'. Under each model, a list of versions is shown with columns for 'Version', 'Created time', and 'Last modified'. For 'Version 4', the details are: Version name 'Version 4', Created time '1/15/2025 12:34 AM', Last modified '1/15/2025 12:34 AM', Experiment name 'lr-model-version...', and Run name 'bubbly_cloud_6rx...'. To the right of the list are two main sections: 'Apply this version' (with a green button) and 'Compare ML model versions in a list' (with a green button). Below these sections is a 'View ML model list' button. The central area is titled 'Version details' and contains sections for 'ML model version metrics (0)', 'ML model version parameters (2)', and 'ML model version tags (4)'. The 'ML model version parameters' section lists 'n_estimators' (3), 'random_state' (42). The 'ML model version tags' section lists 'project_name' (grocery-forecasting), 'store_dept' (produce), 'team' (stores-ml), and 'project_quarter' (Q3-2023). The 'Input schema' section shows a single entry: 'Tensor (dtype: float64, shape: [-1,4])'. A large circular icon with a plus sign and a magnifying glass is located in the bottom right corner of the central area.

Each model version includes the following information:

- **Time Created:** Date and time of model creation.
- **Run Name:** The identifier for the experiment runs used to create this specific model version.
- **Hyperparameters:** Hyperparameters are saved as key-value pairs. Both keys and values are strings.
- **Metrics:** Run metrics saved as key-value pairs. The value is numeric.
- **Model Schema/Signature:** A description of the model inputs and outputs.
- **Logged files:** Logged files in any format. For example, you can record images, environment, models, and data files.
- **Tags:** Metadata as key-value pairs to runs.

Apply tags to machine learning models

MLflow tagging for model versions enables users to attach custom metadata to specific versions of a registered model in the MLflow Model Registry. These tags, stored as key-value pairs, help organize, track, and differentiate between model versions, making it easier to manage model lifecycles. Tags can be used to denote the model's purpose,

deployment environment, or any other relevant information, facilitating more efficient model management and decision-making within teams.

This code demonstrates how to train a RandomForestRegressor model using Scikit-learn, log the model and parameters with MLflow, and then register the model in the MLflow Model Registry with custom tags. These tags provide useful metadata, such as project name, department, team, and project quarter, making it easier to manage and track the model version.

Python

```
import mlflow.sklearn
from mlflow.models import infer_signature
from sklearn.datasets import make_regression
from sklearn.ensemble import RandomForestRegressor

# Generate synthetic regression data
X, y = make_regression(n_features=4, n_informative=2, random_state=0,
shuffle=False)

# Model parameters
params = {"n_estimators": 3, "random_state": 42}

# Model tags for MLflow
model_tags = {
    "project_name": "grocery-forecasting",
    "store_dept": "produce",
    "team": "stores-ml",
    "project_quarter": "Q3-2023"
}

# Log MLflow entities
with mlflow.start_run() as run:
    # Train the model
    model = RandomForestRegressor(**params).fit(X, y)

    # Infer the model signature
    signature = infer_signature(X, model.predict(X))

    # Log parameters and the model
    mlflow.log_params(params)
    mlflow.sklearn.log_model(model, artifact_path="sklearn-model",
signature=signature)

    # Register the model with tags
    model_uri = f"runs:{run.info.run_id}/sklearn-model"
    model_version = mlflow.register_model(model_uri,
    "RandomForestRegressionModel", tags=model_tags)

    # Output model registration details
    print(f"Model Name: {model_version.name}")
```

```
print(f"Model Version: {model_version.version}")
```

After applying the tags, you can view them directly on the model version details page. Additionally, tags can be added, updated, or removed from this page at any time.

The screenshot shows the 'ML Model Details' page for a 'RandomForestRegressor...' model. On the left, there's a sidebar with a tree view showing 'Version 4' (12:34:51 AM), 'Version 3' (12:33:50 AM), 'Version 2' (12:32:44 AM), and 'Version 1' (12:29:41 AM). The main area displays 'Properties' like 'Version name: Version 4', 'Created time: 1/15/2025 12:34 AM', 'Last modified: 1/15/2025 12:34 AM', and 'Run name: bubbly_cloud_6rx...'. Below this is the 'Version details' section, which includes 'ML model version metrics (0)', 'ML model version parameters (2)' (n_estimators: 3, random_state: 42), 'ML model version tags (4)' (project_name: grocery-forecasting, store_dept: produce, team: stores-ml, project_quarter: Q3-2023), and 'Input schema (1)' (Tensor (dtype: float64, shape: [-1,4])). A modal window titled 'Add or edit tags' is open on the right, containing a table with four rows: 'project_name' (value: grocery-forecasting), 'store_dept' (value: produce), 'team' (value: stores-ml), and 'project_quarter' (value: Q3-2023). There are 'Add' and 'Apply this' buttons at the bottom of the modal.

Compare and filter machine learning models

To compare and evaluate the quality of machine learning model versions, you can compare the parameters, metrics, and metadata between selected versions.

Visually compare machine learning models

You can visually compare runs within an existing model. Visual comparison allows easy navigation between, and sorts across, multiple versions.

The screenshot shows the 'ML Model List' page. At the top, there are tabs for 'ML model details' and 'ML model list'. The main area displays a table with 6 selected rows, each representing a different model version. The columns are: Version name, Created time, Created by, Experiment, n_estimators, random_state, project_name, store_dept, and team. The table shows the following data:

Version name	Created time	Created by	Experiment	n_estimators	random_state	project_name	store_dept	team
Version 6	1/15/2025 12:40 AM	Ir-model-version-7118	3	42	grocery-forec...	produce	stores-ml	
Version 5	1/15/2025 12:38 AM	Ir-model-version-7118	3	42	grocery-forec...	produce	stores-ml	
Version 4	1/15/2025 12:34 AM	Ir-model-version-7118	3	42	grocery-forec...	produce	stores-ml	
Version 3	1/15/2025 12:33 AM	Ir-model-version-7118	3	42	grocery-forec...	produce	stores-ml	
Version 2	1/15/2025 12:32 AM	Ir-model-version-7118	3	42	grocery-forec...	produce	stores-ml	
Version 1	1/15/2025 12:29 AM	Ir-model-version-7118	3	42	grocery-forec...	produce	stores-ml	

On the right side, there's a 'Customize columns' section with a search bar and a list of checked filters: 'Properties' (Created time, Created by, Experiment), 'Metrics' (No metrics have been logged), 'Parameters' (n_estimators, random_state), and 'Tags' (project_name, store_dept, team, project_quarter). A magnifying glass icon is also present in this section.

To compare runs, you can:

1. Select an existing machine learning model that contains multiple versions.
2. Select the **View** tab, and then navigate to the **Model list** view. You can also select the option to **View model list** directly from the details view.
3. You can customize the columns within the table. Expand the **Customize columns** pane. From there, you can select the properties, metrics, tags, and hyperparameters that you want to see.
4. Lastly, you can select multiple versions, to compare their results, in the metrics comparison pane. From this pane, you can customize the charts with changes to the chart title, visualization type, X-axis, Y-axis, and more.

Compare machine learning models using the MLflow API

Data scientists can also use MLflow to search among multiple models saved within the workspace. Visit the [MLflow documentation](#) to explore other MLflow APIs for model interaction.

Python

```
from pprint import pprint

client = MlflowClient()
for rm in client.list_registered_models():
    pprint(dict(rm), indent=4)
```

Apply machine learning models

Once you train a model on a data set, you can apply that model to data it never saw to generate predictions. We call this model use technique **scoring** or **inferencing**. For more information about Microsoft Fabric model scoring, see the next section.

Related content

- [Learn about MLflow Experiment APIs](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Autologging in Microsoft Fabric

Article • 01/14/2025

Synapse Data Science in Microsoft Fabric includes autologging, which significantly reduces the amount of code required to automatically log the parameters, metrics, and items of a machine learning model during training. This article describes autologging for Synapse Data Science in Microsoft Fabric.

Autologging extends [MLflow Tracking](#) capabilities and is deeply integrated into the Synapse Data Science in Microsoft Fabric experience. Autologging can capture various metrics, including accuracy, loss, F1 score, and custom metrics you define. By using autologging, developers and data scientists can easily track and compare the performance of different models and experiments without manual tracking.

Supported frameworks

Autologging supports a wide range of machine learning frameworks, including TensorFlow, PyTorch, Scikit-learn, and XGBoost. To learn more about the framework-specific properties that autologging captures, see the [MLflow documentation](#).

Configuration

Autologging works by automatically capturing values of input parameters, output metrics, and output items of a machine learning model as it's being trained. This information is logged to your Microsoft Fabric workspace, where you can access and visualize it by using the MLflow APIs or the corresponding experiment and model items in your Microsoft Fabric workspace.

When you launch a Synapse Data Science notebook, Microsoft Fabric calls `mlflow.autolog()` to instantly enable tracking and load the corresponding dependencies. As you train models in your notebook, MLflow automatically tracks this model information.

The configuration happens automatically behind the scenes when you run `import mlflow`. The default configuration for the notebook `mlflow.autolog()` hook is:

Python

```
mlflow.autolog(  
    log_input_examples=False,
```

```
    log_model_signatures=True,
    log_models=True,
    disable=False,
    exclusive=True,
    disable_for_unsupported_versions=True,
    silent=True
)
```

Customization

To customize logging behavior, you can use the [mlflow.autolog\(\)](#) configuration. This configuration provides parameters to enable model logging, collect input samples, configure warnings, or enable logging for added content that you specify.

Track more metrics, parameters, and properties

For runs created with MLflow, update the MLflow autologging configuration to track additional metrics, parameters, files, and metadata as follows:

1. Update the [mlflow.autolog\(\)](#) call to set `exclusive=False`.

Python

```
mlflow.autolog(
    log_input_examples=False,
    log_model_signatures=True,
    log_models=True,
    disable=False,
    exclusive=False, # Update this property to enable custom logging
    disable_for_unsupported_versions=True,
    silent=True
)
```

2. Use the MLflow tracking APIs to log additional [parameters](#) and [metrics](#). The following example code enables you to log your custom metrics and parameters alongside additional properties.

Python

```
import mlflow
mlflow.autolog(exclusive=False)

with mlflow.start_run():
    mlflow.log_param("parameter name", "example value")
```

```
# <add model training code here>
mlflow.log_metric("metric name", 20)
```

Disable Microsoft Fabric autologging

You can disable Microsoft Fabric autologging for a specific notebook session. You can also disable autologging across all notebooks by using the workspace setting.

ⓘ Note

If autologging is disabled, you must manually log your [parameters](#) and [metrics](#) by using the MLflow APIs.

Disable autologging for a notebook session

To disable Microsoft Fabric autologging for a specific notebook session, call `mlflow.autolog()` and set `disable=True`.

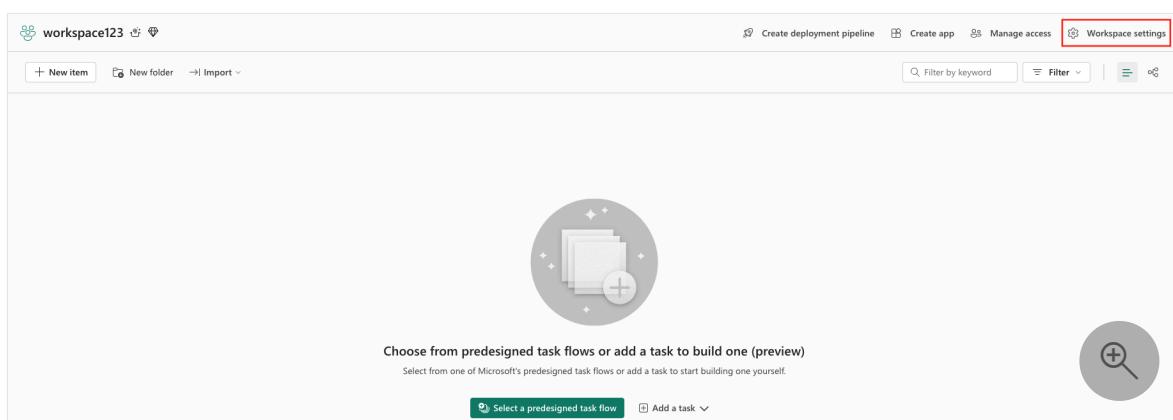
Python

```
import mlflow
mlflow.autolog(disable=True)
```

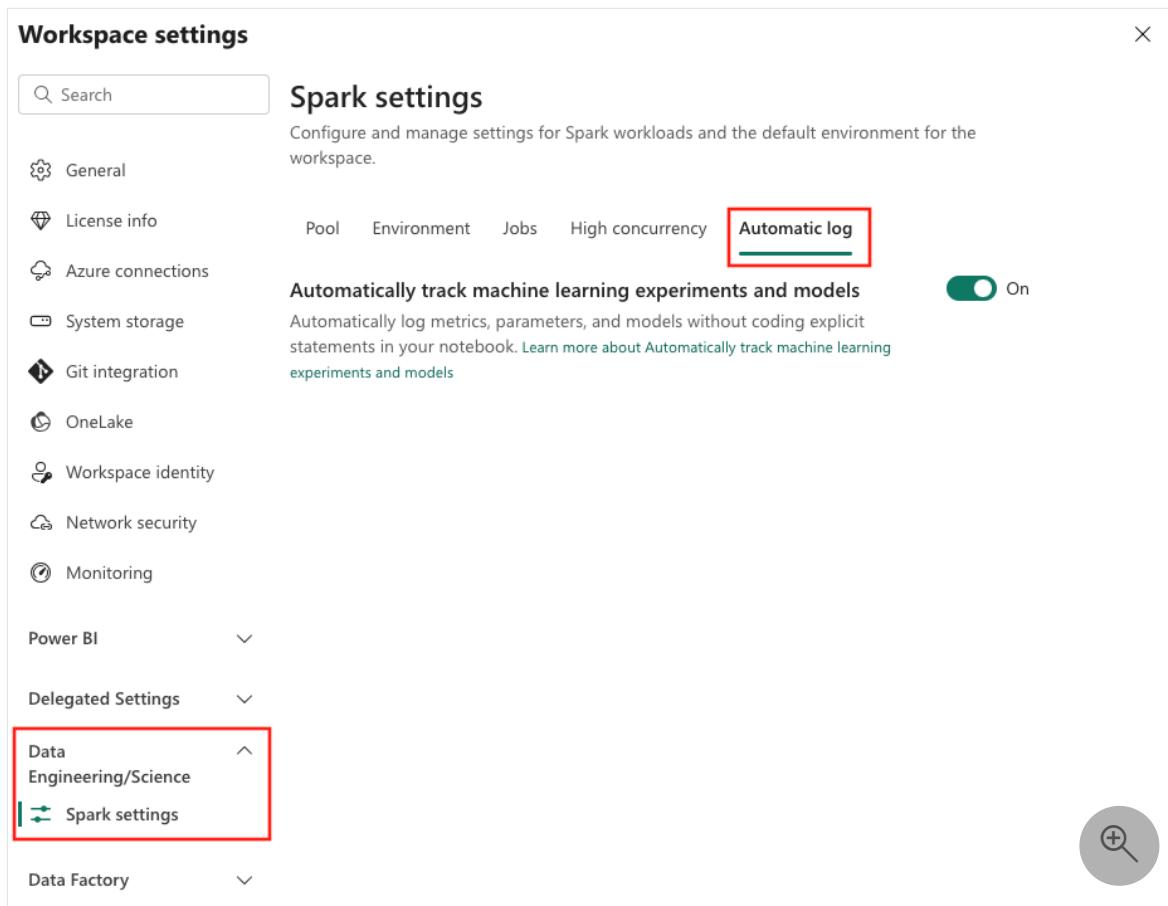
Disable autologging for all notebooks and sessions

Workspace administrators can enable or disable Microsoft Fabric autologging for all notebooks and sessions in their workspace by using the workspace settings. To enable or disable Synapse Data Science autologging:

1. In your workspace, select **Workspace settings**.



2. In *Workspace settings*, expand **Data Engineering/Science** on the left navigation bar and select **Spark settings**.
3. In *Spark settings*, select the **Automatic log** tab.
4. Set **Automatically track machine learning experiments and models** to **On** or **Off**.
5. Select **Save**.



Related content

- [Build a machine learning model with Apache Spark MLlib](#)
- [Machine learning experiments in Microsoft Fabric](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Machine learning model scoring with PREDICT in Microsoft Fabric

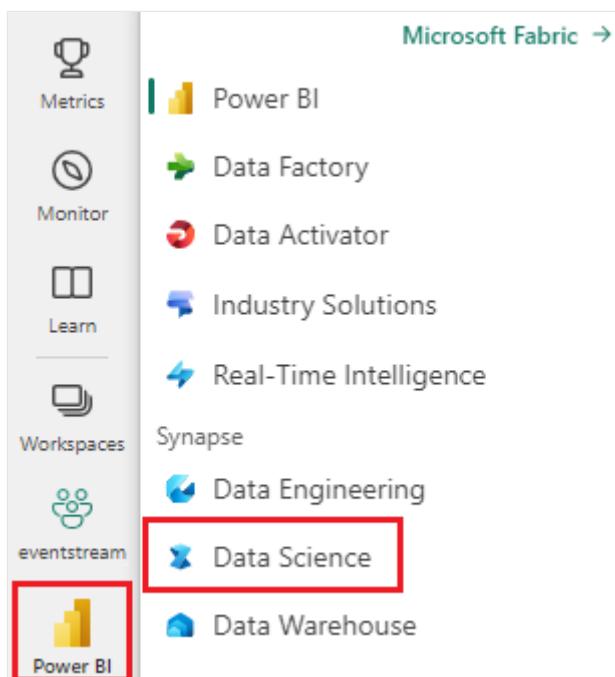
Article • 11/19/2024

Microsoft Fabric allows users to operationalize machine learning models with the scalable PREDICT function. This function supports batch scoring in any compute engine. Users can generate batch predictions directly from a Microsoft Fabric notebook or from the item page of a given ML model.

In this article, you learn how to apply PREDICT by writing code yourself or through use of a guided UI experience that handles batch scoring for you.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



Limitations

- The PREDICT function is currently supported for this limited set of ML model flavors:

- CatBoost
- Keras
- LightGBM
- ONNX
- Prophet
- PyTorch
- Sklearn
- Spark
- Statsmodels
- TensorFlow
- XGBoost
- PREDICT **requires** that you save ML models in the MLflow format, with their signatures populated
- PREDICT **does not** support ML models with multi-tensor inputs or outputs

Call PREDICT from a notebook

PREDICT supports MLflow-packaged models in the Microsoft Fabric registry. If an already trained and registered ML model exists in your workspace, you can skip to step 2. If not, step 1 provides sample code to guide you through training a sample logistic regression model. You can use this model to generate batch predictions at the end of the procedure.

1. **Train an ML model and register it with MLflow.** The next code sample uses the MLflow API to create a machine learning experiment, and then start an MLflow run for a scikit-learn logistic regression model. The model version is then stored and registered in the Microsoft Fabric registry. Visit the [how to train ML models with scikit-learn](#) resource for more information about training models and tracking your own experiments.

Python

```
import mlflow
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_diabetes
from mlflow.models.signature import infer_signature

mlflow.set_experiment("diabetes-demo")
with mlflow.start_run() as run:
    lr = LogisticRegression()
    data = load_diabetes(as_frame=True)
    lr.fit(data.data, data.target)
    signature = infer_signature(data.data, data.target)
```

```
mlflow.sklearn.log_model(  
    lr,  
    "diabetes-model",  
    signature=signature,  
    registered_model_name="diabetes-model"  
)
```

2. Load in test data as a Spark DataFrame. To generate batch predictions with the ML model trained in the previous step, you need test data in the form of a Spark DataFrame. In the following code, substitute the `test` variable value with your own data.

Python

```
# You can substitute "test" below with your own data  
test = spark.createDataFrame(data.frame.drop(['target'], axis=1))
```

3. Create an `MLFlowTransformer` object to load the ML model for inferencing. To create an `MLFlowTransformer` object to generate batch predictions, you must perform these actions:

- specify the `test` DataFrame columns you need as model inputs (in this case, all of them)
- choose a name for the new output column (in this case, `predictions`)
- provide the correct model name and model version for generation of those predictions.

If you use your own ML model, substitute the values for the input columns, output column name, model name, and model version.

Python

```
from synapse.ml.predict import MLFlowTransformer  
  
# You can substitute values below for your own input columns,  
# output column name, model name, and model version  
model = MLFlowTransformer(  
    inputCols=test.columns,  
    outputCol='predictions',  
    modelName='diabetes-model',  
    modelVersion=1  
)
```

4. Generate predictions using the PREDICT function. To invoke the PREDICT function, use the Transformer API, the Spark SQL API, or a PySpark user-defined

function (UDF). The following sections show how to generate batch predictions with the test data and ML model defined in the previous steps, using the different methods to invoke the PREDICT function.

PREDICT with the Transformer API

This code invokes the PREDICT function with the Transformer API. If you use your own ML model, substitute the values for the model and test data.

Python

```
# You can substitute "model" and "test" below with values  
# for your own model and test data  
model.transform(test).show()
```

PREDICT with the Spark SQL API

This code invokes the PREDICT function with the Spark SQL API. If you use your own ML model, substitute the values for `model_name`, `model_version`, and `features` with your model name, model version, and feature columns.

 **Note**

Use of the Spark SQL API for prediction generation still requires creation of an `MLflowTransformer` object (as shown in step 3).

Python

```
from pyspark.ml.feature import SQLTransformer  
  
# You can substitute "model_name," "model_version," and "features"  
# with values for your own model name, model version, and feature columns  
model_name = 'diabetes-model'  
model_version = 1  
features = test.columns  
  
sqlt = SQLTransformer().setStatement(  
    f"SELECT PREDICT('{model_name}/{model_version}', {','.join(features)})  
    as predictions FROM __THIS__")  
  
# You can substitute "test" below with your own test data  
sqlt.transform(test).show()
```

PREDICT with a user-defined function

This code invokes the PREDICT function with a PySpark UDF. If you use your own ML model, substitute the values for the model and features.

Python

```
from pyspark.sql.functions import col, pandas_udf, udf, lit  
  
# You can substitute "model" and "features" below with your own values  
my_udf = model.to_udf()  
features = test.columns  
  
test.withColumn("PREDICT", my_udf(*[col(f) for f in features])).show()
```

Generate PREDICT code from an ML model's item page

From the item page of any ML model, you can choose one of these options to start batch prediction generation for a specific model version, with the PREDICT function:

- Copy a code template into a notebook, and customize the parameters yourself
- Use a guided UI experience to generate PREDICT code

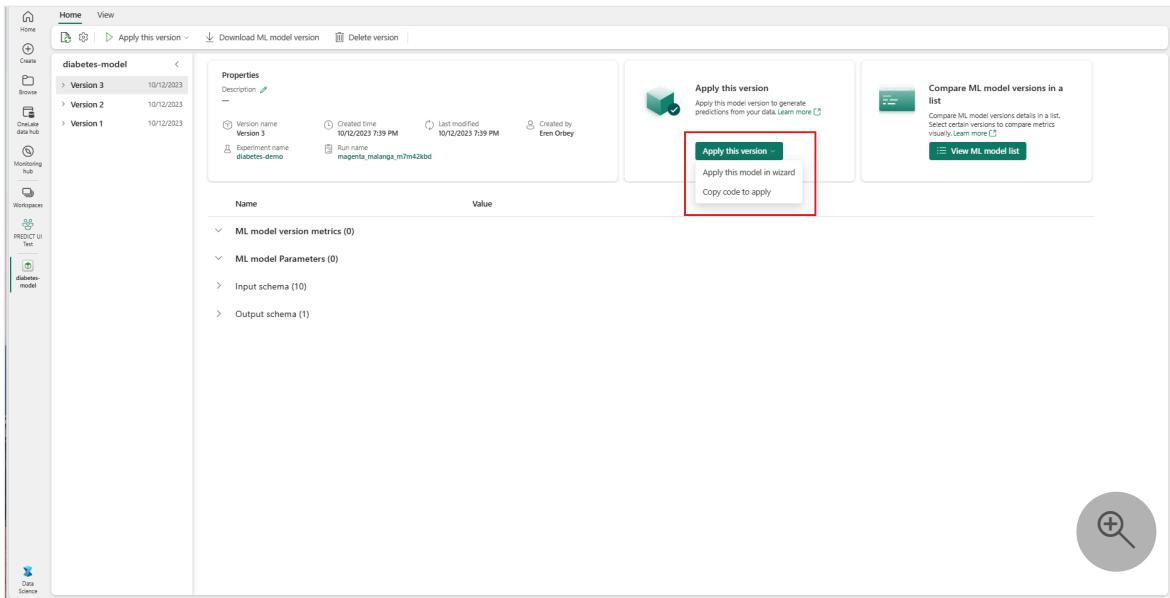
Use a guided UI experience

The guided UI experience walks you through these steps:

- Select the source data for scoring
- Map the data correctly to your ML model inputs
- Specify the destination for your model outputs
- Create a notebook that uses PREDICT to generate and store prediction results

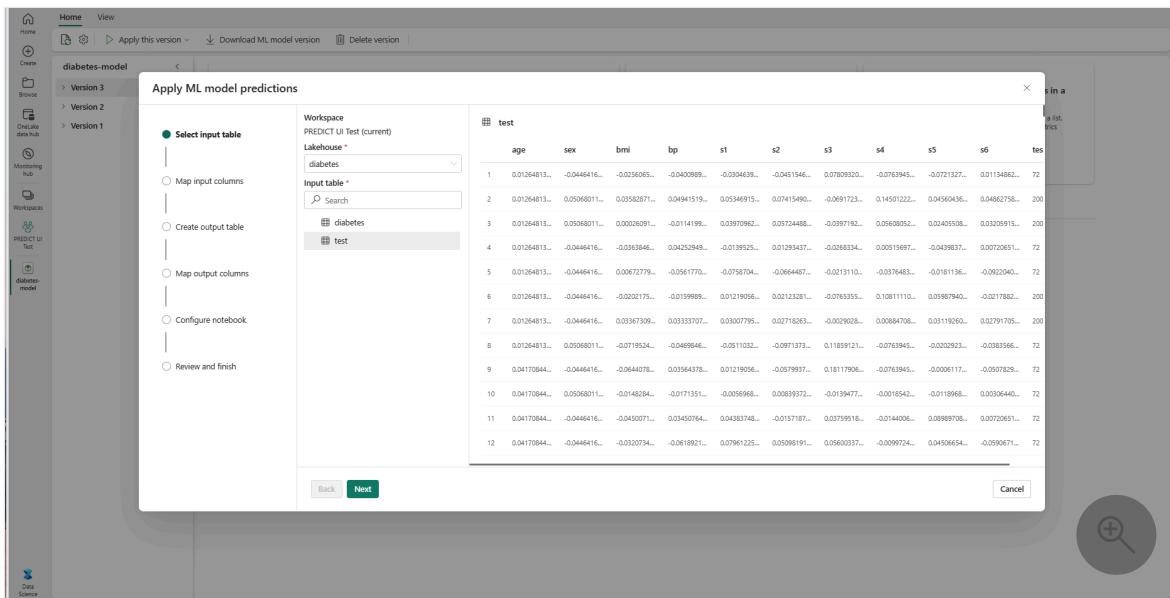
To use the guided experience,

1. Navigate to the item page for a given ML model version.
2. From the **Apply this version** dropdown, select **Apply this model in wizard**.



At the "Select input table" step, the "Apply ML model predictions" window opens.

3. Select an input table from a lakehouse in your current workspace.

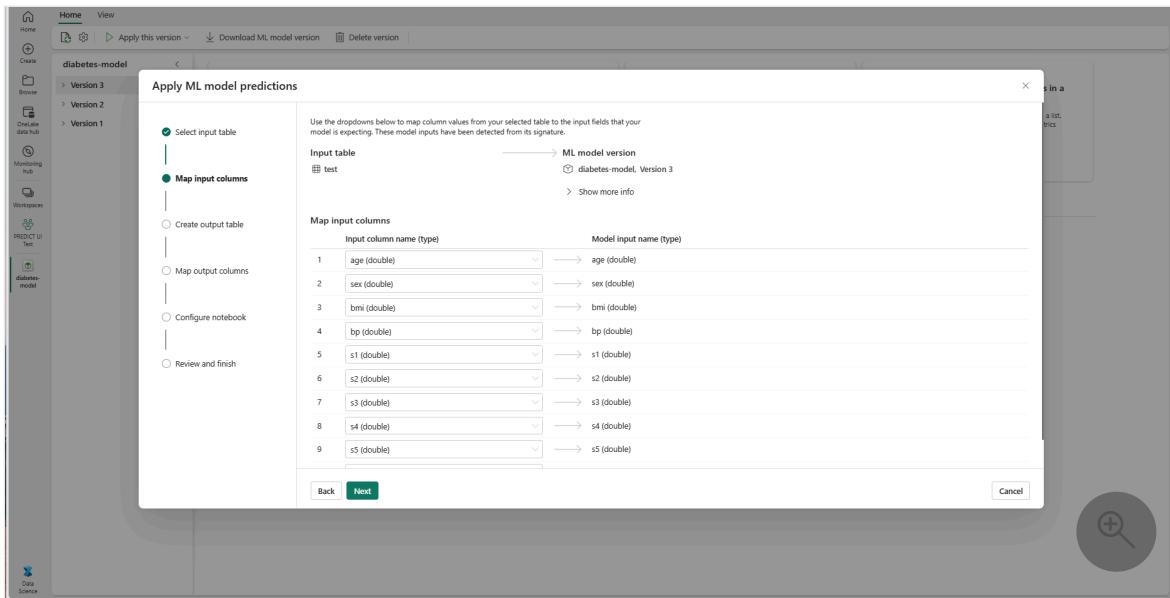


4. Select Next to go to the "Map input columns" step.

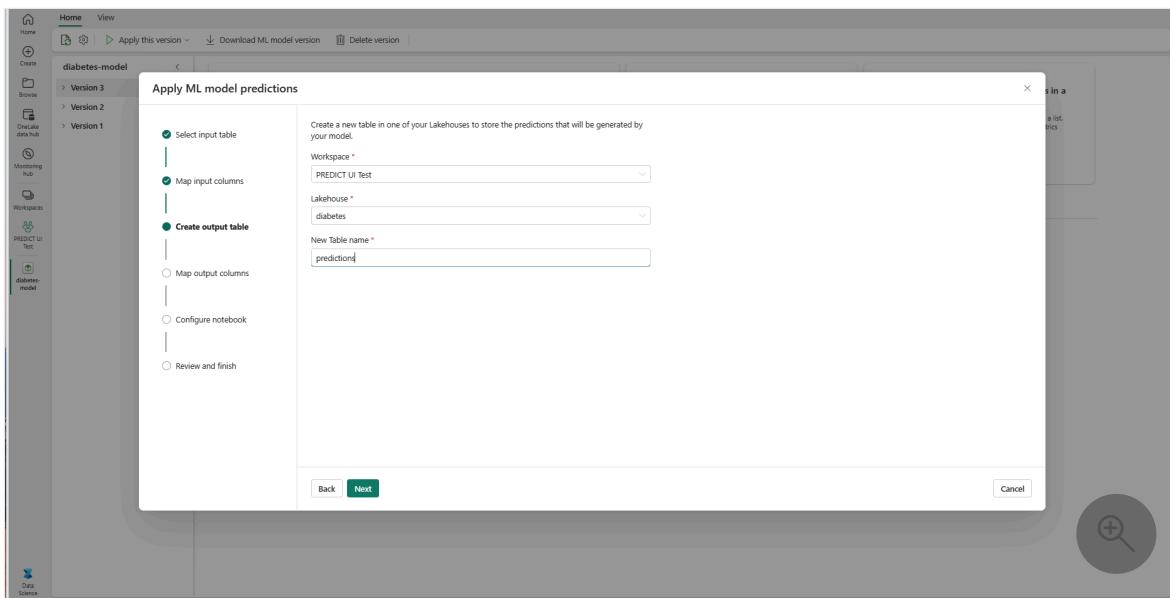
5. Map column names from the source table to the ML model's input fields, which are pulled from the signature of the model. You must provide an input column for all of the required fields of the model. Additionally, the source column data types must match the expected data types of the model.

💡 Tip

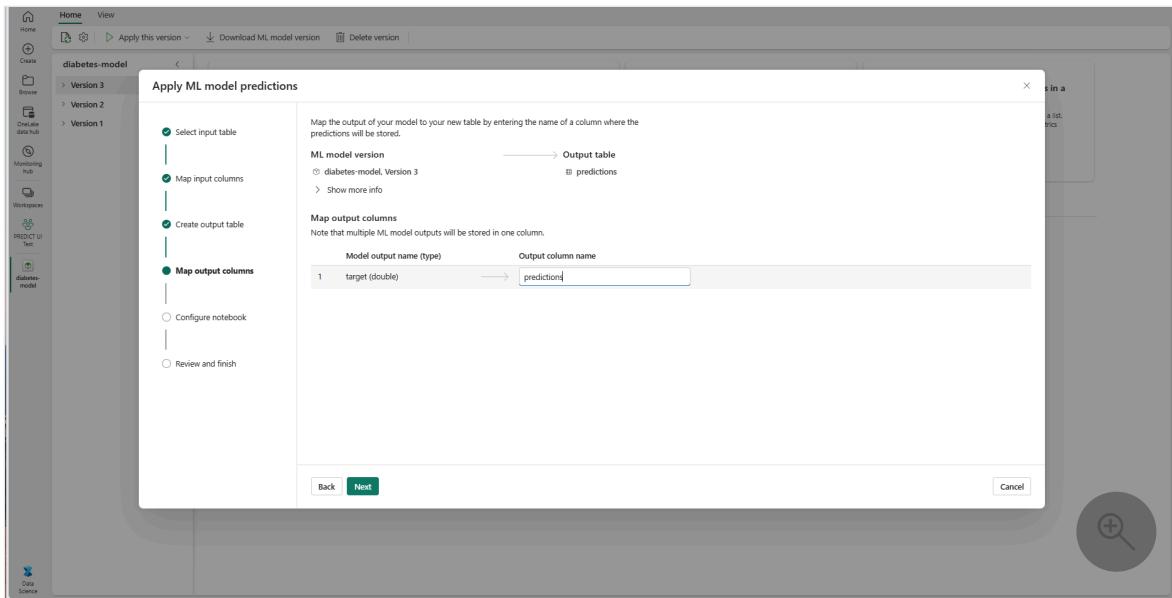
The wizard prepopulates this mapping if the names of the input table columns match the column names logged in the ML model signature.



6. Select **Next** to go to the "Create output table" step.
7. Provide a name for a new table within the selected lakehouse of your current workspace. This output table stores your ML model's input values, and it appends the prediction values to that table. By default, the output table is created in the same lakehouse as the input table. You can change the destination lakehouse.

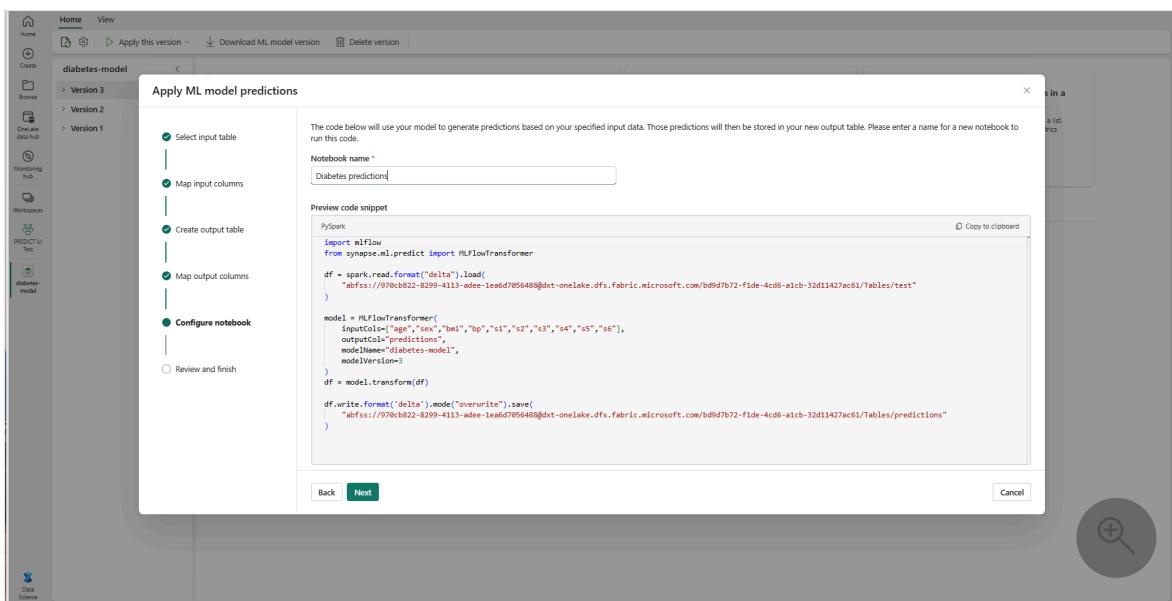


8. Select **Next** to go to the "Map output columns" step.
9. Use the provided text fields to name the columns of the output table that stores the ML model predictions.



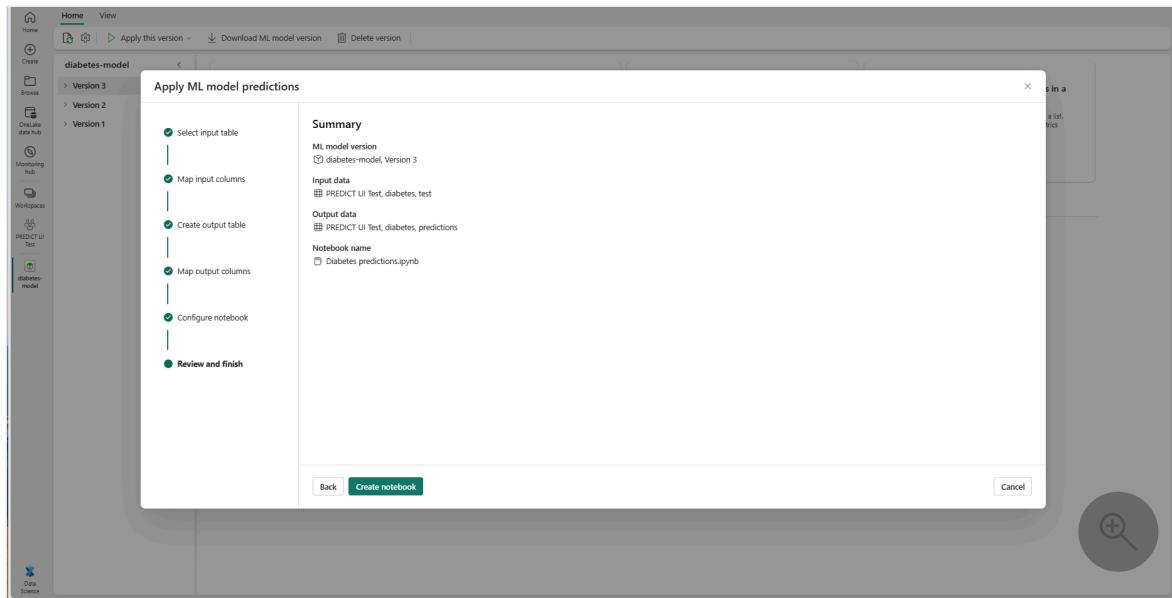
10. Select **Next** to go to the "Configure notebook" step.

11. Provide a name for a new notebook that runs the generated PREDICT code. The wizard displays a preview of the generated code at this step. If you want, you can copy the code to your clipboard and paste it into an existing notebook.



12. Select **Next** to go to the "Review and finish" step.

13. Review the details on the summary page, and select **Create notebook** to add the new notebook with its generated code to your workspace. You're taken directly to that notebook, where you can run the code to generate and store predictions.



Use a customizable code template

To use a code template for generating batch predictions:

1. Go to the item page for a given ML model version.
2. Select **Copy code to apply** from the **Apply this version** dropdown. The selection allows you to copy a customizable code template.

You can paste this code template into a notebook to generate batch predictions with your ML model. To successfully run the code template, you must manually replace the following values:

- <INPUT_TABLE>: The file path for the table that provides inputs to the ML model
- <INPUT_COLS>: An array of column names from the input table to feed to the ML model
- <OUTPUT_COLS>: A name for a new column in the output table that stores predictions
- <MODEL_NAME>: The name of the ML model to use for generating predictions
- <MODEL_VERSION>: The version of the ML model to use for generating predictions
- <OUTPUT_TABLE>: The file path for the table that stores the predictions

The screenshot shows the Microsoft Fabric interface for managing machine learning models. On the left, there's a sidebar with various icons for Home, Create, Browse, Monitor, Workspaces, and Predict UI. The main area shows a list of ML models, with 'diabetes-model' selected. This model has three versions listed: Version 3 (10/12/2023), Version 2 (10/12/2023), and Version 1 (10/12/2023). Below the list, there are sections for Properties (Description, Version name: Version 3, Experiment name: diabetes-demo, Created time: 10/12/2023 7:38 AM, Run name: magenta_malar), Name (ML model version metrics (0), ML model Parameters (0)), Input schema (10), and Output schema (1). A central modal window titled 'Copy code to apply ML model predictions' contains PySpark code:

```
PySpark
import mlflow
from synapse.ml.predict import MLFlowTransformer

df = spark.read.format("delta").load(
    <INPUT_TABLE> # Your input table filepath here
)

model = MLFlowTransformer(
    inputCols=<INPUT_COLS>, # Your input columns here
    outputCol=<OUTPUT_COLS>, # Your new column name here
    modelName=<MODEL_NAME>, # Your ML model name here
    modelVersion=<MODEL_VERSION> # Your ML model version here
)
df = model.transform(df)

df.write.format('delta').mode("overwrite").save(
    <OUTPUT_TABLE> # Your output table filepath here
)
```

Python

```
import mlflow
from synapse.ml.predict import MLFlowTransformer

df = spark.read.format("delta").load(
    <INPUT_TABLE> # Your input table filepath here
)

model = MLFlowTransformer(
    inputCols=<INPUT_COLS>, # Your input columns here
    outputCol=<OUTPUT_COLS>, # Your new column name here
    modelName=<MODEL_NAME>, # Your ML model name here
    modelVersion=<MODEL_VERSION> # Your ML model version here
)
df = model.transform(df)

df.write.format('delta').mode("overwrite").save(
    <OUTPUT_TABLE> # Your output table filepath here
)
```

Related content

- End-to-end prediction example using a fraud detection model
- How to train ML models with scikit-learn in Microsoft Fabric

Feedback

Was this page helpful?

Yes

No

[Provide product feedback](#) | [Ask the community](#)

Disaster recovery guidance for Fabric Data Science

Article • 04/10/2025

In Microsoft Fabric, machine learning experiments and models consist of files and metadata. This article provides guidance about how to protect your data from rare, region-wide outages.

Disaster recovery

To protect your data from rare region-wide outages, we recommend that you copy your critical data to another region, with a frequency that matches the needs of your disaster recovery plan.

To store your machine learning experiments and models in two different regions, you must create these resources in two different workspaces. When you select workspaces, you must choose workspaces associated with capacities in two different regions.

Next, export and copy your Fabric notebooks into your secondary workspace, and rerun the notebooks to create the relevant machine learning items in the secondary workspace.

If a regional outage occurs, you can then access your machine learning items in the second region where you copied the items.

Related content

- [OneLake Disaster Recovery](#)

Data science roles and permissions

Article • 01/10/2025

This article describes machine learning model and experiment permissions in Microsoft Fabric and how these permissions are acquired by users.

ⓘ Note

After you create a workspace in Microsoft Fabric, or if you have an admin role in a workspace, you can give others access to it by assigning them a different role. To learn more about workspaces and how to grant users access to your workspace, review the articles [Workspaces](#) and [Giving users access to workspaces](#).

Permissions for machine learning experiments

The table below describes the levels of permission that control access to machine learning experiments in Microsoft Fabric.

expand Expand table

Permission	Description
Read	<p>Allows user to read machine learning experiments.</p> <p>Allows user to view runs within machine learning experiments.</p> <p>Allows user to view run metrics and parameters.</p> <p>Allows user to view and download run files.</p>
Write	<p>Allows user to create machine learning experiments.</p> <p>Allows user to modify or delete machine learning experiments.</p> <p>Allows user to add runs to machine learning experiments.</p> <p>Allows user to save an experiment run as a model.</p>

Permissions for machine learning models

The table below describes the levels of permission that control access to machine learning models in Microsoft Fabric.

expand Expand table

Permission	Description
Read	Allows user to read machine learning models. Allows user to view versions within machine learning models. Allows user to view model version metrics and parameters. Allows user to view and download model version files.
Write	Allows user to create machine learning models. Allows user to modify or delete machine learning models. Allows user to add model versions to machine learning models.

Permissions acquired by workspace role

A user's role in a workspace implicitly grants them permissions on the datasets in the workspace, as described in the following table.

[Expand table](#)

	Admin	Member	Contributor	Viewer
Read	✓	✓	✓	✓
Write	✓	✓	✓	✗

① Note

You can either assign roles to individuals or to security groups, Microsoft 365 groups, and distribution lists. To learn more about workspace roles in Microsoft Fabric, see [Roles in workspaces](#)

Related content

- Learn about roles in workspaces: [Roles in Microsoft Fabric workspaces](#)
- Give users access to workspaces: [Granting access to users](#)

Feedback

Was this page helpful?

 Yes

 No

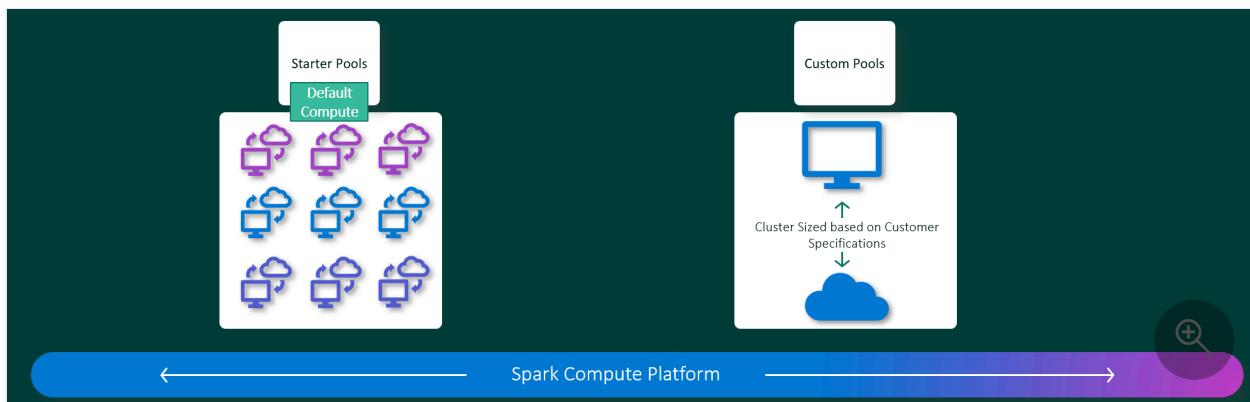
[Provide product feedback ↗](#) | [Ask the community ↗](#)

What is Apache Spark compute in Microsoft Fabric?

Article • 03/13/2025

Applies to: Data Engineering and Data Science in Microsoft Fabric

Microsoft Fabric Data Engineering and Data Science experiences operate on a fully managed Apache Spark compute platform. This platform is designed to deliver unparalleled speed and efficiency. With starter pools, you can expect rapid Apache Spark session initialization, typically within 5 to 10 seconds, with no need for manual setup. You also get the flexibility to customize Apache Spark pools according to your specific data engineering and data science requirements. The platform enables an optimized and tailored analytics experience. In short a starter pool is a quick way to use pre-configured Spark, while a Spark pool offers customization and flexibility.



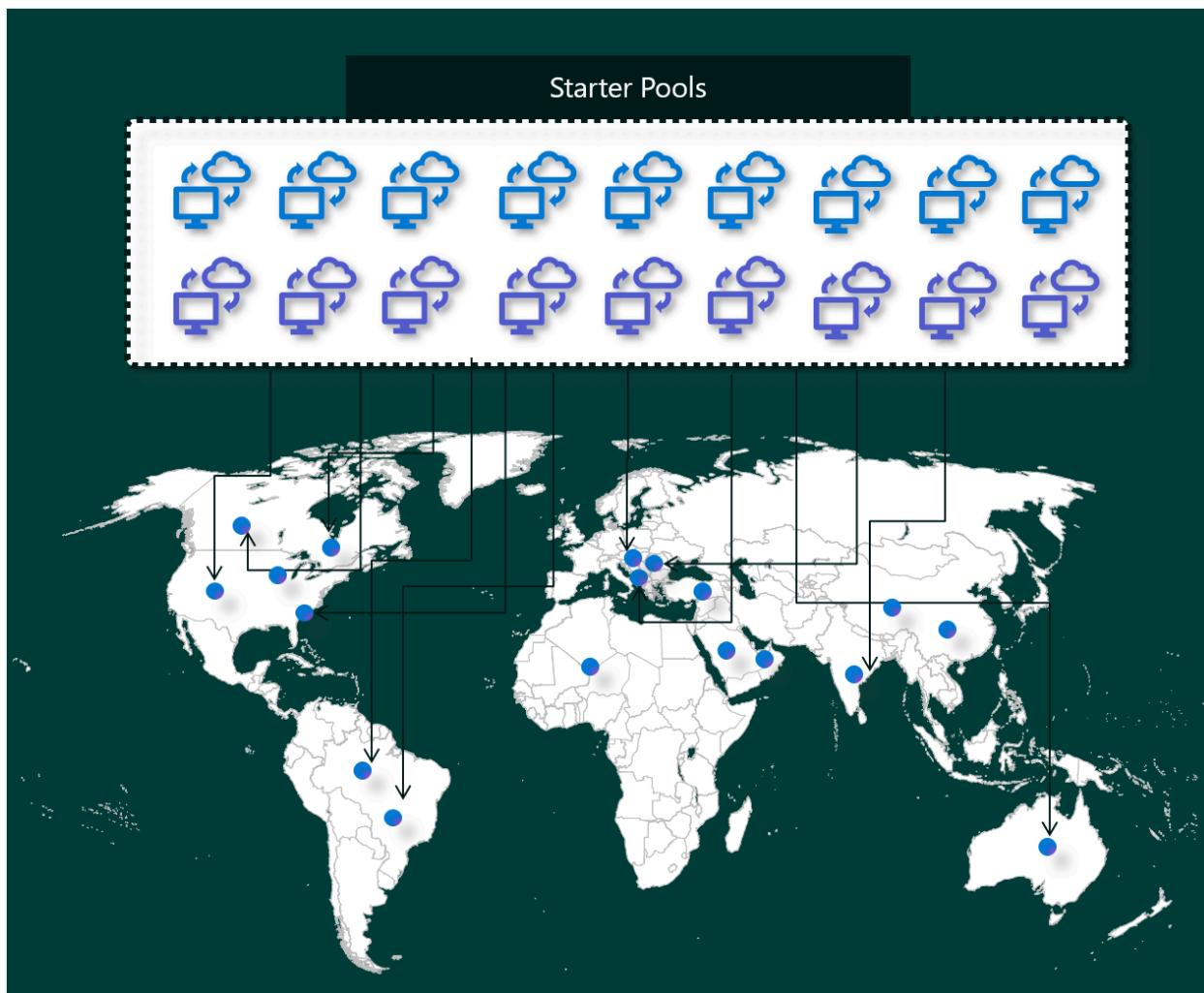
Starter pools

Starter pools are a fast and easy way to use Spark on the Microsoft Fabric platform within seconds. You can use Spark sessions right away, instead of waiting for Spark to set up the nodes for you, which helps you do more with data and get insights quicker.

Starter Pool Configuration

Node family	Memory optimized
Node Size	Medium
Min and Max Nodes	[1, 10]
Autoscale	On
Dynamic Allocation	On

Starter pools have Apache Spark clusters that are always on and ready for your requests. They use medium nodes that dynamically scale up based on your Spark job needs.



When you use a Starter Pool **without any extra library dependencies or custom Spark properties**, your session typically starts in **5 to 10 seconds**. This fast startup is possible because the cluster is already running and doesn't require provisioning time.

However, there are several scenarios where your session might take longer to start:

1. You have custom libraries or Spark properties

If you've configured libraries or custom settings in your environment, Spark has to personalize the session once it's created. This process can add around **30 seconds to 5 minutes** to your startup time, depending on the number and size of your library dependencies.

2. Starter Pools in your region are fully used

In rare cases, a region's Starter Pools might be temporarily exhausted due to high traffic. When that happens, Fabric spins up a **new cluster** to accommodate your request, which takes about **2 to 5 minutes**. Once the new cluster is available, your session starts. If you also have custom libraries to install, you need to add the additional **30 seconds to 5 minutes** required for personalization.

3. Advanced networking or security features (Private Links or Managed VNets)

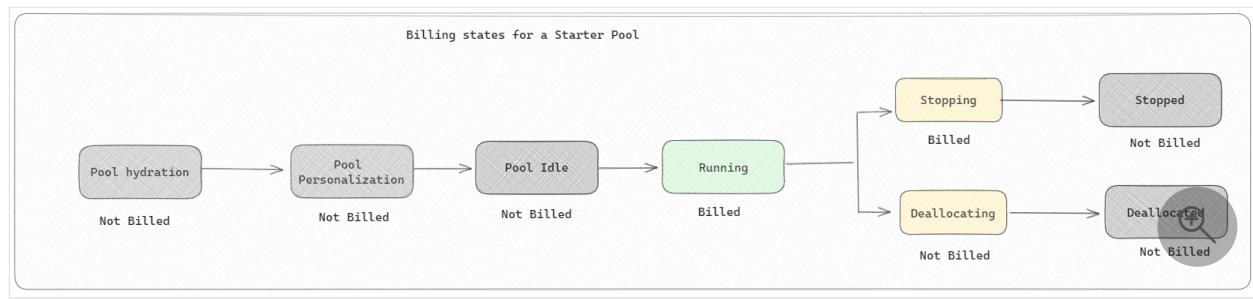
When your workspace has networking features such as **Tenant Private Links** or **Managed VNets**, Starter Pools aren't supported. In this situation, Fabric must create a cluster on demand, which adds **2 to 5 minutes** to your session start time. If you also have library dependencies, that personalization step can again add another **30 seconds to 5 minutes**.

Here are a few example scenarios to illustrate potential start times:

[+] Expand table

Scenario	Typical Startup Time
Default settings, no libraries	5 – 10 seconds
Default settings + library dependencies	5 – 10 seconds + 30 seconds – 5 min (for library setup)
High traffic in region, no libraries	2 – 5 minutes
High traffic + library dependencies	2 – 5 minutes + 30 seconds – 5 min (for libraries)
Network security (Private Links/VNet), no libraries	2 – 5 minutes
Network security + library dependencies	2 – 5 minutes + 30 seconds – 5 min (for libraries)

When it comes to billing and capacity consumption, you're charged for the capacity consumption when you start executing your notebook or Apache Spark job definition. You aren't charged for the time the clusters are idle in the pool.



For example, if you submit a notebook job to a starter pool, you're billed only for the time period where the notebook session is active. The billed time doesn't include the idle time or the time taken to personalize the session with the Spark context.

Spark pools

A Spark pool is a way of telling Spark what kind of resources you need for your data analysis tasks. You can give your Spark pool a name, and choose how many and how large the nodes (the machines that do the work) are. You can also tell Spark how to adjust the number of nodes depending on how much work you have. Creating a Spark pool is free; you only pay when you run a Spark job on the pool, and then Spark sets up the nodes for you.

If you don't use your Spark pool for 2 minutes after your session expires, your Spark pool will be deallocated. This default session expiration time period is set to 20 minutes, and you can change it if you want. If you're a workspace admin, you can also create custom Spark pools for your workspace, and make them the default option for other users. This way, you can save time and avoid setting up a new Spark pool every time you run a notebook or a Spark job. Custom Spark pools take about three minutes to start, because Spark must get the nodes from Azure.

You can even create single node Spark pools, by setting the minimum number of nodes to one, so the driver and executor run in a single node that comes with restorable HA and is suited for small workloads.

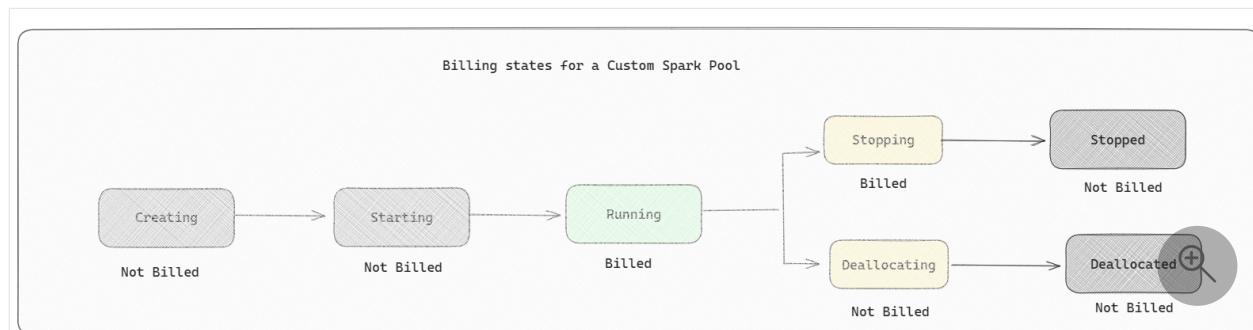
The size and number of nodes you can have in your custom Spark pool depends on your Microsoft Fabric capacity. Capacity is a measure of how much computing power you can use in Azure. One way to think of it is that two Apache Spark Vcores (a unit of computing power for Spark) equals one capacity unit.

Note

In Apache Spark, users get two Apache Spark VCores for every capacity unit they reserve as part of their SKU. One Capacity Unit = Two Spark VCores So F64 => 128 Spark Vcores and on which a 3x Burst Multiplier is applied which gives a total of 384 Spark VCores

For example, a Fabric capacity SKU F64 has 64 capacity units, which is equivalent to 384 Spark VCores ($64 * 2 * 3X$ Burst Multiplier). You can use these Spark VCores to create nodes of different sizes for your custom Spark pool, as long as the total number of Spark VCores doesn't exceed 384.

Spark pools are billed like starter pools; you don't pay for the custom Spark pools that you have created unless you have an active Spark session created for running a notebook or Spark job definition. You're only billed for the duration of your job runs. You aren't billed for stages like the cluster creation and deallocation after the job is complete.



For example, if you submit a notebook job to a custom Spark pool, you're only charged for the time period when the session is active. The billing for that notebook session stops once the Spark session has stopped or expired. You aren't charged for the time taken to acquire cluster instances from the cloud or for the time taken for initializing the Spark context.

Possible custom pool configurations for F64 based on the previous example:

[] Expand table

Fabric capacity SKU	Capacity units	Max Spark VCores with Burst Factor	Node size	Max number of nodes
F64	64	384	Small	96
F64	64	384	Medium	48
F64	64	384	Large	24
F64	64	384	X-Large	12

Fabric capacity SKU	Capacity units	Max Spark VCores with Burst Factor	Node size	Max number of nodes
F64	64	384	XX-Large	6

! Note

To create custom pools, you need **admin** permissions for the workspace. And the Microsoft Fabric capacity admin must grant permissions to allow workspace admins to size their custom Spark pools. To learn more, see [Get started with custom Spark pools in Fabric](#)

Nodes

An Apache Spark pool instance consists of one head node and worker nodes, could start a minimum of one node in a Spark instance. The head node runs extra management services such as Livy, Yarn Resource Manager, Zookeeper, and the Apache Spark driver. All nodes run services such as Node Agent and Yarn Node Manager. All worker nodes run the Apache Spark Executor service.

Node sizes

A Spark pool can be defined with node sizes that range from a small compute node (with 4 vCore and 32 GB of memory) to a double extra large compute node (with 64 vCore and 512 GB of memory per node). Node sizes can be altered after pool creation, although the active session would have to be restarted.

[] Expand table

Size	vCore	Memory
Small	4	32 GB
Medium	8	64 GB
Large	16	128 GB
X-Large	32	256 GB
XX-Large	64	512 GB

! Note

Node sizes X-Large and XX-Large are only allowed for non-trial Fabric SKUs.

Autoscale

Autoscale for Apache Spark pools allows automatic scale up and down of compute resources based on the amount of activity. When you enable the autoscale feature, you set the minimum and maximum number of nodes to scale. When you disable the autoscale feature, the number of nodes set remains fixed. You can alter this setting after pool creation, although you might need to restart the instance.

 **Note**

By default, `spark.yarn.executor.decommission.enabled` is set to true, enabling the automatic shutdown of underutilized nodes to optimize compute efficiency. If less aggressive scaling down is preferred, this configuration can be set to false

Dynamic allocation

Dynamic allocation allows the Apache Spark application to request more executors if the tasks exceed the load that current executors can bear. It also releases the executors when the jobs are completed, and if the Spark application is moving to idle state.

Enterprise users often find it hard to tune the executor configurations because they're vastly different across different stages of a Spark job execution process. These configurations are also dependent on the volume of data processed, which changes from time to time. You can enable dynamic allocation of executors option as part of the pool configuration, which enables automatic allocation of executors to the Spark application based on the nodes available in the Spark pool.

When you enable the dynamic allocation option for every Spark application submitted, the system reserves executors during the job submission step based on the minimum nodes. You specify maximum nodes to support successful automatic scale scenarios.

Related content

- [Get Started with Data Engineering/Science Admin Settings for your Fabric Capacity](#)
- [Apache Spark workspace administration settings in Microsoft Fabric](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Configuring starter pools in Microsoft Fabric

Article • 04/10/2024

In this article, we explain how to customize starter pools in Microsoft Fabric for your analytics workloads. Starter pools are a fast and easy way to use Spark on the Microsoft Fabric platform within seconds. You can use Spark sessions right away, instead of waiting for Spark to set up the nodes for you, which helps you do more with data and get insights quicker.

Starter pools have Spark clusters that are always on and ready for your requests. They use medium-sized nodes and can be scaled up based on your workload requirements.

You can specify the maximum nodes for autoscaling based on the data engineering or data science workload requirements. Based on the max nodes you configure, the system dynamically acquires and retires nodes as the job's compute requirements change, which results in efficient scaling and improved performance.

You can also set the maximum limit for executors in starter pools and with Dynamic Allocation enabled, the system adjusts the number of executors depending on the data volume and job-level compute needs. This process enables you to focus on your workloads without worrying about performance optimization and resource management.

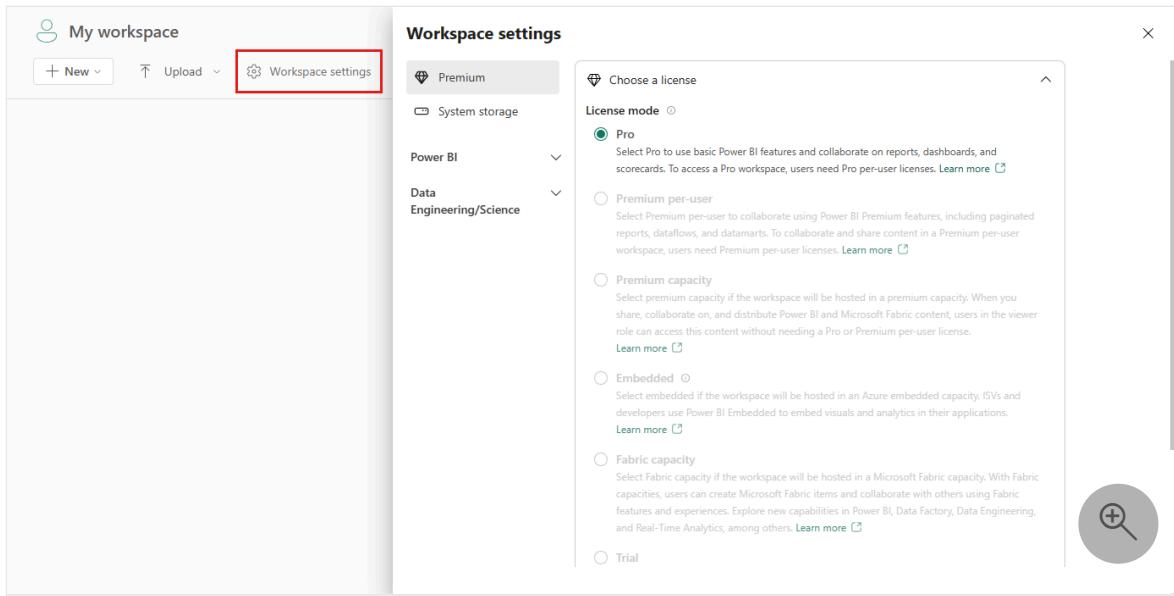
ⓘ Note

To customize a starter pool, you need admin access to the workspace.

Configure starter pools

To manage the starter pool associated with your workspace:

1. Go to your workspace and choose the **Workspace settings**.



2. Then, select the **Data Engineering/Science** option to expand the menu.

Workspace settings

Search

Premium

System storage

Power BI

Data Engineering/Science

Spark settings

3. Select the **StarterPool** option.

Workspace settings

Search

Spark settings

Configure and manage settings for Spark workloads and the default environment for the workspace.

Premium

System storage

Pool Environment High concurrency Automatic log

Power BI

Data Engineering/Science

Spark settings

Default pool for workspace

Use the automatically created starter pool or create custom pools for workspaces and items in the capacity. If the setting Customize compute configurations for items is turned off, this pool will be used for all environments in this workspace.

StarterPool

Pool details

Node family	Node size	Number of nodes
Memory optimized	Medium	1 - 10

4. You can set the maximum node configuration for your starter pools to an allowed number based on the purchased capacity or reduce the default max node configuration to a smaller value when running smaller workloads.

[←](#)

Edit pool

Spark pool name *

Node family

Node size

Autoscale

If enabled, your Apache Spark pool will automatically scale up and down based on the amount of activity.

Enable autoscale

1  16

Dynamically allocate executors

Enable allocate

1  15

Save **Discard**

The following section lists various default configurations and the max node limits supported for starter pools based on Microsoft Fabric capacity SKUs:

[Expand table](#)

SKU name	Capacity units	Spark VCores	Node size	Default max nodes	Max number of nodes
F2	2	4	Medium	1	1
F4	4	8	Medium	1	1
F8	8	16	Medium	2	2
F16	16	32	Medium	3	4
F32	32	64	Medium	8	8
F64	64	128	Medium	10	16
(Trial Capacity)	64	128	Medium	10	16
F128	128	256	Medium	10	32
F256	256	512	Medium	10	64
F512	512	1024	Medium	10	128
F1024	1024	2048	Medium	10	200
F2048	2048	4096	Medium	10	200

ⓘ Note

To customize a starter pool, you need admin access to the workspace.

Related content

- Learn more from the Apache Spark [public documentation](#).
- Get started with [Spark workspace administration settings in Microsoft Fabric](#).

Feedback

Was this page helpful?

[Yes](#)[No](#)

[Provide product feedback](#) | [Ask the community](#)

Billing and utilization reporting for Apache Spark in Microsoft Fabric

Article • 04/09/2025

Applies to:  Data Engineering and Data Science in Microsoft Fabric

This article explains the compute utilization and reporting for ApacheSpark which powers the Fabric Data Engineering and Science workloads in Microsoft Fabric. The compute utilization includes lakehouse operations like table preview, load to delta, notebook runs from the interface, scheduled runs, runs triggered by notebook steps in the pipelines, and Apache Spark job definition runs.

Like other experiences in Microsoft Fabric, Data Engineering also uses the capacity associated with a workspace to run these job and your overall capacity charges appear in the Azure portal under your [Microsoft Cost Management](#) subscription. To learn more about Fabric billing, see [Understand your Azure bill on a Fabric capacity](#).

Fabric capacity

You as a user could purchase a Fabric capacity from Azure by specifying using an Azure subscription. The size of the capacity determines the amount of computation power available. For Apache Spark for Fabric, every CU purchased translates to 2 Apache Spark Vcores. For example if you purchase a Fabric capacity F128, this translates to 256 SparkVcores. A Fabric capacity is shared across all the workspaces added to it and in which the total Apache Spark compute allowed gets shared across all the jobs submitted from all the workspaces associated to a capacity. To understand about the different SKUs, cores allocation and throttling on Spark, see [Concurrency limits and queueing in Apache Spark for Microsoft Fabric](#).

Autoscale Billing for Spark

Autoscale Billing for Spark introduces a flexible, pay-as-you-go billing model for Spark workloads in Microsoft Fabric. With this model enabled, Spark jobs use dedicated serverless resources instead of consuming compute from Fabric capacity. This serverless option optimizes cost and provides scalability without resource contention.

When enabled, **Autoscale Billing** allows you to set a maximum Capacity Unit (CU) limit, which controls your budget and resource allocation. The billing for Spark jobs is based solely on the compute used during job execution, with no idle compute costs. The cost per Spark job remains the same (0.5 CU Hour), and you are only charged for the runtime of active jobs.

Key Benefits of Autoscale Billing:

- **Cost Efficiency:** Pay only for the Spark job runtime.
- **Independent Scaling:** Spark workloads scale independently of other workload demands.
- **Enterprise-Ready:** Integrates with Azure Quota Management for flexible scaling.

How Autoscale Billing Works:

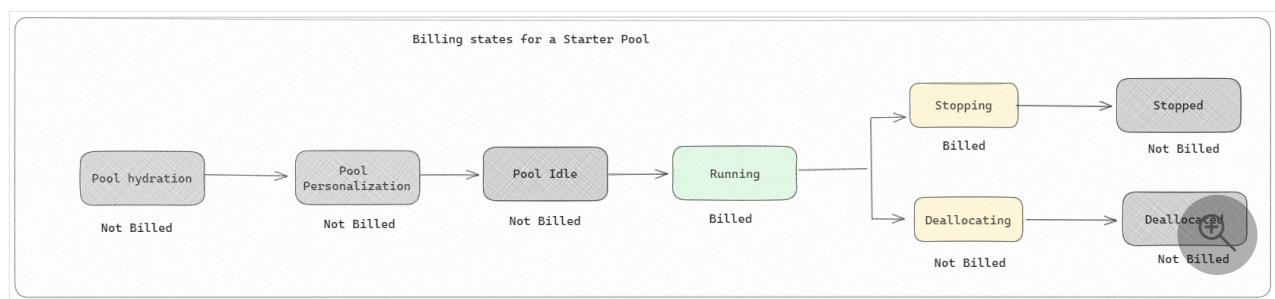
- Spark jobs no longer consume CU from the Fabric capacity but instead use serverless resources.
- A max CU limit can be set to align with budget or governance policies, ensuring predictable costs.
- Once the CU limit is reached, Spark jobs will either queue (for batch jobs) or throttle (for interactive jobs).
- There is no idle compute cost, and only active job compute usage is billed.

For more detailed information, see the [Autoscale Billing for Spark Overview](#).

Spark compute configuration and purchased capacity

Apache Spark compute for Fabric offers two options when it comes to compute configuration.

1. **Starter pools:** These default pools are fast and easy way to use Spark on the Microsoft Fabric platform within seconds. You can use Spark sessions right away, instead of waiting for Spark to set up the nodes for you, which helps you do more with data and get insights quicker. When it comes to billing and capacity consumption, you're charged when you start executing your notebook or Spark job definition or lakehouse operation. You aren't charged for the time the clusters are idle in the pool.

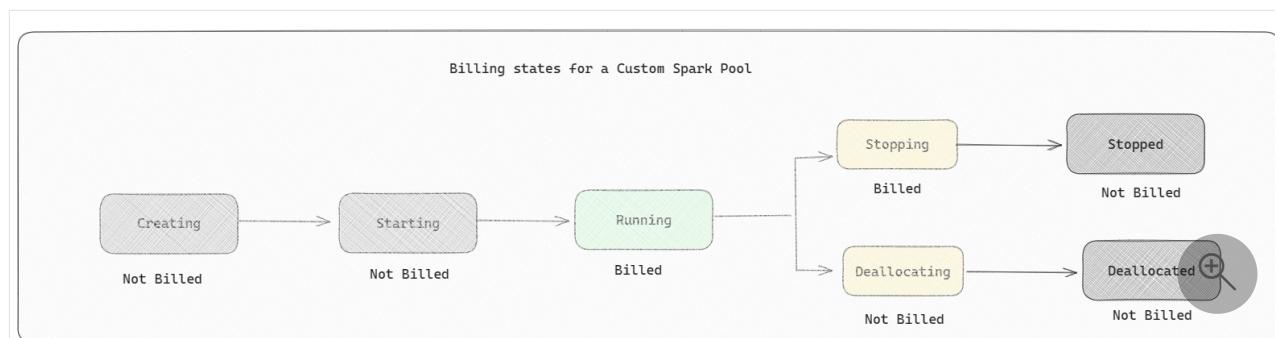


For example, if you submit a notebook job to a starter pool, you're billed only for the time period where the notebook session is active. The billed time doesn't include the idle time or the time taken to personalize the session with the Spark context. To understand more

about configuring Starter pools based on the purchased Fabric Capacity SKU, visit [Configuring Starter Pools based on Fabric Capacity](#)

2. Spark pools: These are custom pools, where you get to customize on what size of resources you need for your data analysis tasks. You can give your Spark pool a name, and choose how many and how large the nodes (the machines that do the work) are. You can also tell Spark how to adjust the number of nodes depending on how much work you have. Creating a Spark pool is free; you only pay when you run a Spark job on the pool, and then Spark sets up the nodes for you.

- The size and number of nodes you can have in your custom Spark pool depends on your Microsoft Fabric capacity. You can use these Spark VCores to create nodes of different sizes for your custom Spark pool, as long as the total number of Spark VCores doesn't exceed 128.
- Spark pools are billed like starter pools; you don't pay for the custom Spark pools that you have created unless you have an active Spark session created for running a notebook or Spark job definition. You're only billed for the duration of your job runs. You aren't billed for stages like the cluster creation and deallocation after the job is complete.



For example, if you submit a notebook job to a custom Spark pool, you're only charged for the time period when the session is active. The billing for that notebook session stops once the Spark session has stopped or expired. You aren't charged for the time taken to acquire cluster instances from the cloud or for the time taken for initializing the Spark context. To understand more about configuring Spark pools based on the purchased Fabric Capacity SKU, visit [Configuring Pools based on Fabric Capacity](#)

Note

The default session expiration time period for the Starter Pools and Spark Pools that you create is set to 20 minutes. If you don't use your Spark pool for 2 minutes after your session expires, your Spark pool will be deallocated. To stop the session and the billing after completing your notebook execution before the session expiry time period, you can

either click the stop session button from the notebooks Home menu or go to the monitoring hub page and stop the session there.

Spark compute usage reporting

The [Microsoft Fabric Capacity Metrics app](#) provides visibility into capacity usage for all Fabric workloads in one place. It's used by capacity administrators to monitor the performance of workloads and their usage, compared to purchased capacity.

Once you have installed the app, select the item type **Notebook,Lakehouse,Spark Job Definition** from the **Select item kind:** dropdown list. The **Multi metric ribbon chart** chart can now be adjusted to a desired timeframe to understand the usage from all these selected items.

All Spark related operations are classified as [background operations](#). Capacity consumption from Spark is displayed under a notebook, a Spark job definition, or a lakehouse, and is aggregated by operation name and item. For example: If you run a notebook job, you can see the notebook run, the CUs used by the notebook (Total Spark VCores/2 as 1 CU gives 2 Spark VCores), duration the job has taken in the report.

Item	CU(s)	Duration (s)	Users	Item Size (GB)	Overloaded minutes	Performance delta	Preview Status	Status
housing-study-notebook \ SynapseNotebook \ Spark PM Team	0.00	0.00	0				True	
Notebook 1 \ SynapseNotebook \ bw workspace	5096.05	1274.01	1				True	
Notebook 1 \ SynapseNotebook \ dbrowne_Trident	20610.67	5152.66	1				True	
Notebook 2 \ SynapseNotebook \ bw workspace	5728.82	1432.19	1				True	
Notebook 3 \ SynapseNotebook \ umaws01	5658.31	1414.58	1				True	
Notebook 4 \ SynapseNotebook \ umaws01	5416.29	1354.07	1				True	
Notebook-DeletionTest \ SynapseNotebook \ umaws1	6736.91	1684.23	1				True	
Notebook-DeletionTest2 \ SynapseNotebook \ umaws1	9511.99	2378.00	1				True	
Notebook-DeletionTest3 \ SynapseNotebook \ umaws1	9409.55	2352.39	1				True	
NotebookSample \ SynapseNotebook \ CapacityBugBash	11302.38	2825.60	1				True	
Uma BugBash NB1 \ SynapseNotebook \ CapacityBugBash	84094409.09	21023476....	1				True	
Notebook Interactive Run	2887999.14	721997.64	1				True	
Notebook Scheduled Run	81206409.94	20301478.53	1				True	
Total	84196562.99	21047589....	1				True	

To understand more about Spark capacity usage reporting, see [Monitor Apache Spark capacity consumption](#)

Billing example

Consider the following scenario:

- There is a Capacity C1 which hosts a Fabric Workspace W1 and this Workspace contains Lakehouse LH1 and Notebook NB1.
 - Any Spark operation that the notebook(NB1) or lakehouse(LH1) performs is reported against the capacity C1.

- Extending this example to a scenario where there is another Capacity C2 which hosts a Fabric Workspace W2 and lets say that this Workspace contains a Spark job definition (SJD1) and Lakehouse (LH2).
 - If the Spark Job Definition (SDJ2) from Workspace (W2) reads data from lakehouse (LH1) the usage is reported against the Capacity C2 which is associated with the Workspace (W2) hosting the item.
 - If the Notebook (NB1) performs a read operation from Lakehouse(LH2), the capacity consumption is reported against the Capacity C1 which is powering the workspace W1 that hosts the notebook item.

When **Autoscale Billing** is enabled for Spark, the usage is reported against the **Autoscale for Spark Capacity CU** meter. This separate meter tracks the compute usage directly and is reflected in the **Cost Analysis** section of your Azure subscription, allowing administrators to monitor costs specifically associated with Spark workloads using Autoscale Billing.

Tracking Autoscale Billing in Azure Cost Analysis:

After enabling Autoscale Billing, use Azure's built-in cost management tools to track the spend:

1. Navigate to the [Azure portal](#).
2. Select the **Subscription** linked to your Fabric capacity.
3. In the subscription page, go to **Cost Analysis**.
4. Filter by the resource (Fabric capacity) and use the meter: **Autoscale for Spark Capacity Usage CU**.
5. View real-time compute spend for Spark workloads using Autoscale Billing.

Related content

- [Get Started with Data Engineering/Science Admin Settings for your Fabric Capacity](#)
- [Apache Spark workspace administration settings in Microsoft Fabric](#)
- [Autoscale Billing for Spark Overview](#)
- [Install the Premium metrics app](#)
- [Use the Premium metrics app](#)

Apache Spark Runtimes in Fabric

Article • 01/08/2025

Microsoft Fabric Runtime is an Azure-integrated platform based on Apache Spark that enables the execution and management of data engineering and data science experiences. It combines key components from both internal and open-source sources, providing customers with a comprehensive solution. For simplicity, we refer to Microsoft Fabric Runtime powered by Apache Spark as Fabric Runtime.

Major components of Fabric Runtime:

- **Apache Spark** - a powerful open-source distributed computing library that enables large-scale data processing and analytics tasks. Apache Spark provides a versatile and high-performance platform for data engineering and data science experiences.
- **Delta Lake** - an open-source storage layer that brings ACID transactions and other data reliability features to Apache Spark. Integrated within Fabric Runtime, Delta Lake enhances data processing capabilities and ensures data consistency across multiple concurrent operations.
- **The Native Execution Engine** - is a transformative enhancement for Apache Spark workloads, offering significant performance gains by directly executing Spark queries on lakehouse infrastructure. Integrated seamlessly, it requires no code changes and avoids vendor lock-in, supporting both Parquet and Delta formats across Apache Spark APIs in Runtime 1.3 (Spark 3.5). This engine boosts query speeds up to four times faster than traditional OSS Spark, as shown by the TPC-DS 1TB benchmark, reducing operational costs and improving efficiency across various data tasks—including data ingestion, ETL, analytics, and interactive queries. Built on Meta's Velox and Intel's Apache Gluten, it optimizes resource use while handling diverse data processing scenarios.
- **Default-level packages for Java/Scala, Python, and R** - packages that support diverse programming languages and environments. These packages are automatically installed and configured, allowing developers to apply their preferred programming languages for data processing tasks.
- The Microsoft Fabric Runtime is built upon a **robust open-source operating system**, ensuring compatibility with various hardware configurations and system requirements.

Below, you find a comprehensive comparison of key components, including Apache Spark versions, supported operating systems, Java, Scala, Python, Delta Lake, and R, for

Apache Spark-based runtimes within the Microsoft Fabric platform.

💡 Tip

Always use the most recent, GA runtime version for your production workload, which currently is [Runtime 1.3](#).

[+] Expand table

	Runtime 1.1	Runtime 1.2	Runtime 1.3
Release Stage	EOSA	GA	GA
Apache Spark	3.3.1	3.4.1	3.5.0
Operating System	Ubuntu 18.04	Mariner 2.0	Mariner 2.0
Java	8	11	11
Scala	2.12.15	2.12.17	2.12.17
Python	3.10	3.10	3.11
Delta Lake	2.2.0	2.4.0	3.2
R	4.2.2	4.2.2	4.4.1

Visit [Runtime 1.1](#), [Runtime 1.2](#) or [Runtime 1.3](#) to explore details, new features, improvements, and migration scenarios for the specific runtime version.

Fabric optimizations

In Microsoft Fabric, both the Spark engine and the Delta Lake implementations incorporate platform-specific optimizations and features. These features are designed to use native integrations within the platform. It's important to note that all these features can be disabled to achieve standard Spark and Delta Lake functionality. The Fabric Runtimes for Apache Spark encompass:

- The complete open-source version of Apache Spark.
- A collection of nearly 100 built-in, distinct query performance enhancements. These enhancements include features like partition caching (enabling the FileSystem partition cache to reduce metastore calls) and Cross Join to Projection of Scalar Subquery.
- Built-in intelligent cache.

Within the Fabric Runtime for Apache Spark and Delta Lake, there are native writer capabilities that serve two key purposes:

1. They offer differentiated performance for writing workloads, optimizing the writing process.
2. They default to V-Order optimization of Delta Parquet files. The Delta Lake V-Order optimization is crucial for delivering superior read performance across all Fabric engines. To gain a deeper understanding of how it operates and how to manage it, refer to the dedicated article on [Delta Lake table optimization and V-Order](#).

Multiple runtimes support

Fabric supports multiple runtimes, offering users the flexibility to seamlessly switch between them, minimizing the risk of incompatibilities or disruptions.

By default, all new workspaces use the latest runtime version, which is currently [Runtime 1.3](#).

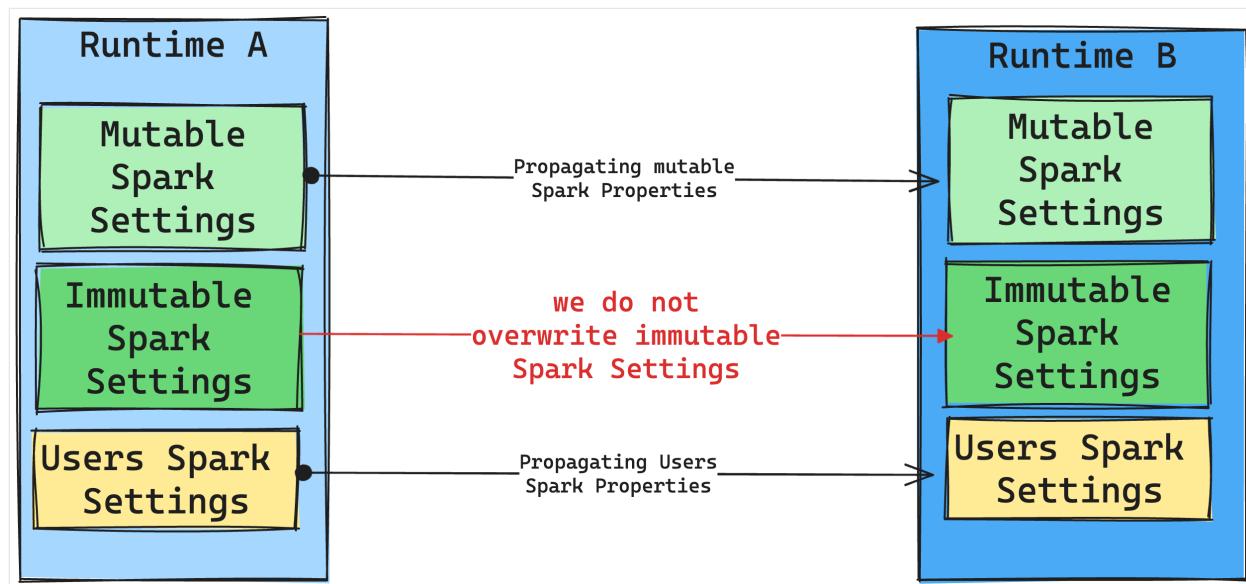
To change the runtime version at the workspace level, go to **Workspace Settings > Data Engineering/Science > Spark settings**. From the **Environment** tab, select your desired runtime version from the available options. Select **Save** to confirm your selection.

The screenshot shows the 'Spark settings' page within the 'Data Engineering/Science' section of the 'Workspace settings' dialog. The 'Environment' tab is active. A dropdown menu lists runtime versions: 1.1 (Spark 3.3, Delta 2.2) with a warning icon, 1.2 (Spark 3.4, Delta 2.4), and 1.3 (Spark 3.5, Delta 3.2), which is highlighted with a red box. A note below states: 'Existing items will use Runtime 1.1 the next time they're opened or the next time a session is started. When you create new items in this workspace, they'll use Runtime 1.1 by default.' Buttons at the bottom are 'Save' (highlighted with a red box) and 'Discard'.

Once you make this change, all system-created items within the workspace, including Lakehouses, SJDs, and Notebooks, will operate using the newly selected workspace-level runtime version starting from the next Spark Session. If you're currently using a notebook with an existing session for a job or any lakehouse-related activity, that Spark session continue as is. However, starting from the next session or job, the selected runtime version will be applied.

Consequences of runtime changes on Spark Settings

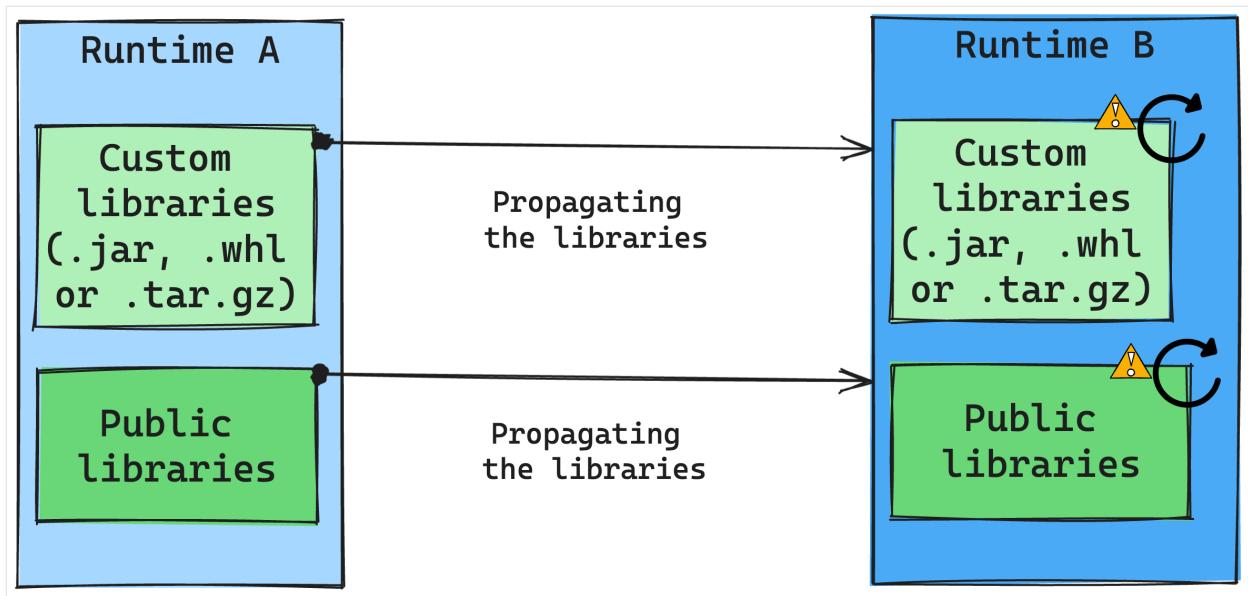
In general, we aim to migrate all Spark settings. However, if we identify that the Spark setting isn't compatible with Runtime B, we issue a warning message and refrain from implementing the setting.



Consequences of runtime changes on library management

In general, our approach is to migrate all libraries from Runtime A to Runtime B, including both Public and Custom Runtimes. If the Python and R versions remain unchanged, the libraries should function properly. However, for Jars, there's a significant likelihood that they may not work due to alterations in dependencies, and other factors such as changes in Scala, Java, Spark, and the operating system.

The user is responsible for updating or replacing any libraries that don't work with Runtime B. If there's a conflict, which means that Runtime B includes a library originally defined in Runtime A, our library management system will try to create the necessary dependency for Runtime B based on the user's settings. However, the building process will fail if a conflict occurs. In the error log, users can see which libraries are causing conflicts and make adjustments to their versions or specifications.



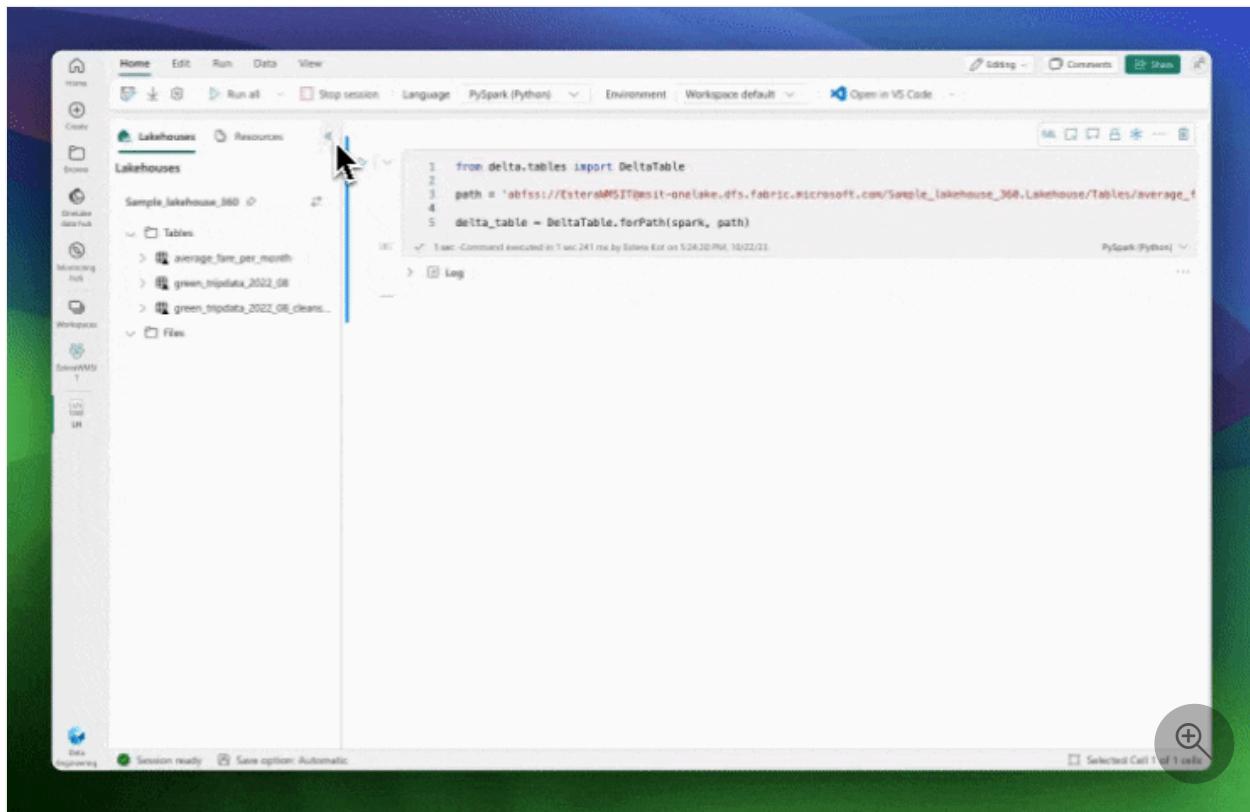
Upgrade Delta Lake protocol

Delta Lake features are always backwards compatible, ensuring tables created in a lower Delta Lake version can seamlessly interact with higher versions. However, when certain features are enabled (for example, by using

`delta.upgradeTableProtocol(minReaderVersion, minWriterVersion)` method, forward compatibility with lower Delta Lake versions may be compromised. In such instances, it's essential to modify workloads referencing the upgraded tables to align with a Delta Lake version that maintains compatibility.

Each Delta table is associated with a protocol specification, defining the features it supports. Applications that interact with the table, either for reading or writing, rely on this protocol specification to determine if they are compatible with the table's feature set. If an application lacks the capability to handle a feature listed as supported in the table's protocol, it's unable to read from or write to that table.

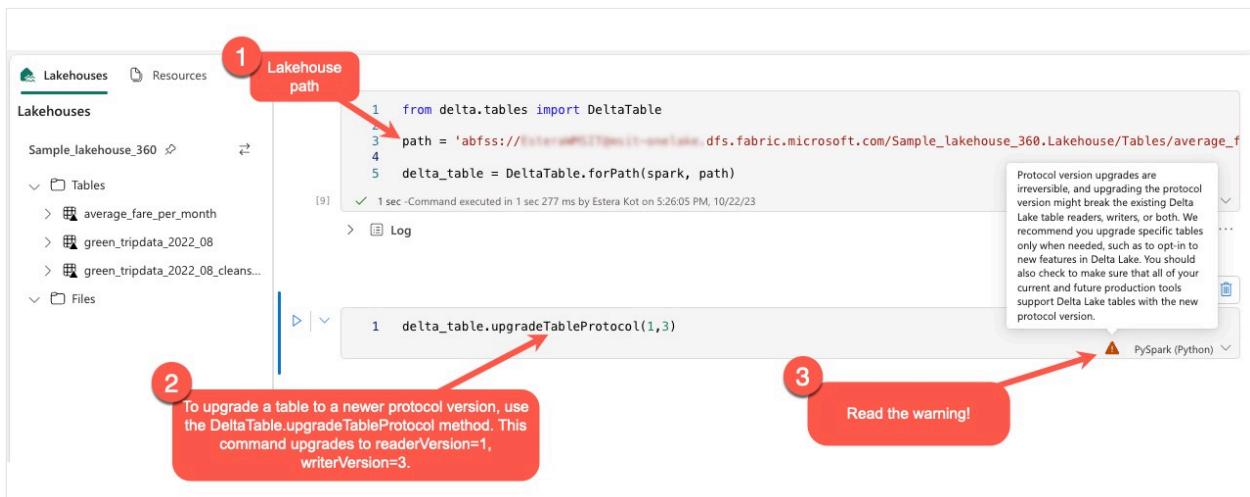
The protocol specification is divided into two distinct components: the read protocol and the write protocol. Visit the page "[How does Delta Lake manage feature compatibility?](#)" to read details about it.



Users can execute the command `delta.upgradeTableProtocol(minReaderVersion, minWriterVersion)` within the PySpark environment, and in Spark SQL and Scala. This command allows them to initiate an update on the Delta table.

It's essential to note that when performing this upgrade, users receive a warning indicating that upgrading the Delta protocol version is a nonreversible process. This means that once the update is executed, it cannot be undone.

Protocol version upgrades can potentially impact the compatibility of existing Delta Lake table readers, writers, or both. Therefore, it's advisable to proceed with caution and upgrade the protocol version only when necessary, such as when adopting new features in Delta Lake.



Additionally, users should verify that all current and future production workloads and processes are compatible with Delta Lake tables using the new protocol version to ensure a seamless transition and prevent any potential disruptions.

Delta 2.2 vs Delta 2.4 changes

In the latest [Fabric Runtime, version 1.3](#) and [Fabric Runtime, version 1.2](#), the default table format (`spark.sql.sources.default`) is now `delta`. In previous versions of [Fabric Runtime, version 1.1](#) and on all Synapse Runtime for Apache Spark containing Spark 3.3 or below, the default table format was defined as `parquet`. Check [the table with Apache Spark configuration details](#) for differences between Azure Synapse Analytics and Microsoft Fabric.

All tables created using Spark SQL, PySpark, Scala Spark, and Spark R, whenever the table type is omitted, will create the table as `delta` by default. If scripts explicitly set the table format, that will be respected. The command `USING DELTA` in Spark create table commands becomes redundant.

Scripts that expect or assume parquet table format should be revised. The following commands are not supported in Delta tables:

- `ANALYZE TABLE $partitionedTableName PARTITION (p1) COMPUTE STATISTICS`
- `ALTER TABLE $partitionedTableName ADD PARTITION (p1=3)`
- `ALTER TABLE DROP PARTITION`
- `ALTER TABLE RECOVER PARTITIONS`
- `ALTER TABLE SET SERDEPROPERTIES`
- `LOAD DATA`
- `INSERT OVERWRITE DIRECTORY`
- `SHOW CREATE TABLE`
- `CREATE TABLE LIKE`

Related content

- [Runtime 1.3 \(Spark 3.5, Java 11, Python 3.11, Delta Lake 3.2\)](#)
- [Runtime 1.2 \(Spark 3.4, Java 11, Python 3.10, Delta Lake 2.4\)](#)
- [Runtime 1.1 \(Spark 3.3, Java 8, Python 3.10, Delta Lake 2.2\)](#)

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Ask the community 

Fabric Runtime 1.3 (GA)

Article • 11/07/2024

Fabric runtime offers a seamless integration with Azure. It provides a sophisticated environment for both data engineering and data science projects that use Apache Spark. This article provides an overview of the essential features and components of Fabric Runtime 1.3, the newest runtime for big data computations.

Microsoft Fabric Runtime 1.3 is the latest GA runtime version and incorporates the following components and upgrades designed to enhance your data processing capabilities:

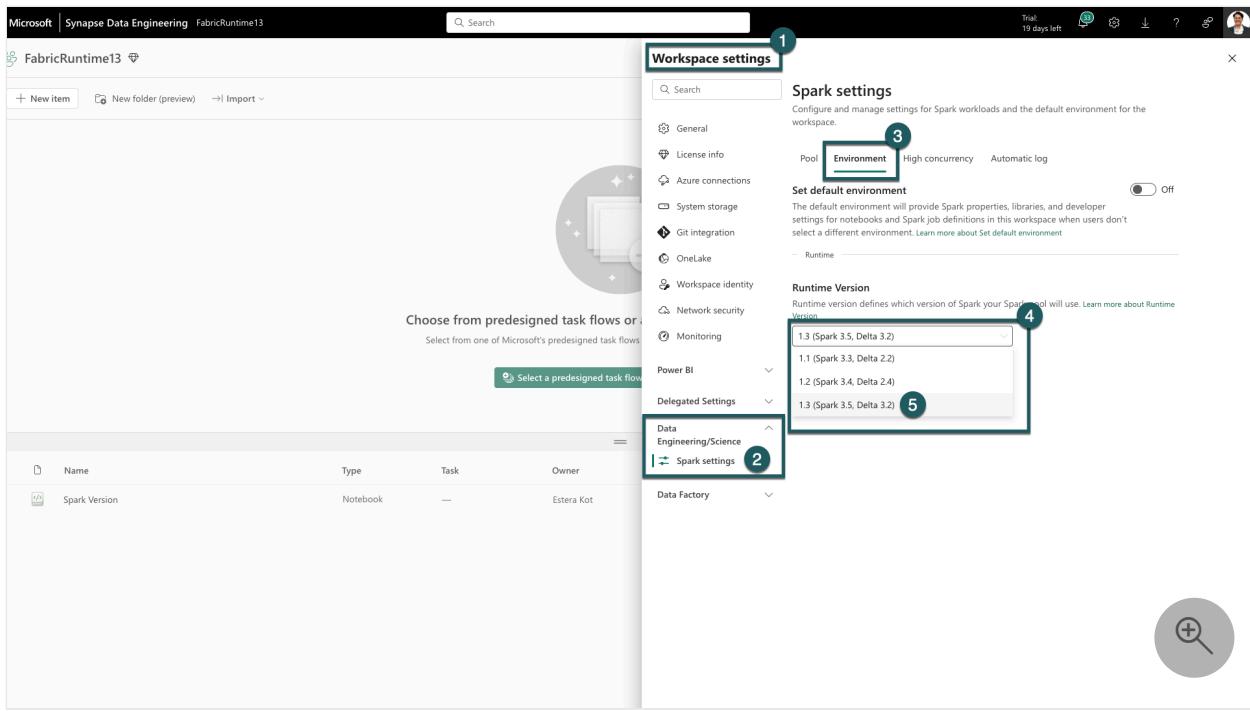
- Apache Spark 3.5
- Operating System: Mariner 2.0
- Java: 11
- Scala: 2.12.17
- Python: 3.11
- Delta Lake: 3.2
- R: 4.4.1

💡 Tip

Fabric Runtime 1.3 includes support for [the Native Execution Engine](#), which can significantly enhance performance without additional costs. To enable the native execution engine across all jobs and notebooks in your environment, navigate to your environment settings, select Spark compute, go to the Acceleration tab, and check Enable native execution engine. After saving and publishing, this setting is applied across the environment, so all new jobs and notebooks automatically inherit and benefit from the enhanced performance capabilities.

Use the following instructions to integrate runtime 1.3 into your workspace and use its new features:

1. Navigate to the **Workspace settings** tab within your Fabric workspace.
2. Go to **Data Engineering/Science** tab and select **Spark Settings**.
3. Select the **Environment** tab.
4. Under the **Runtime Versions** expand the dropdown.
5. Select **1.3 (Spark 3.5, Delta 3.2)** and save your changes. This action sets 1.3 as the default runtime for your workspace.



You can now start working with the newest improvements and functionalities introduced in Fabric runtime 1.3 (Spark 3.5 and Delta Lake 3.2).

Key highlights

Apache Spark 3.5

[Apache Spark 3.5.0 ↗](#) is the sixth version in the 3.x series. This version is a product of extensive collaboration within the open-source community, addressing more than 1,300 issues as recorded in Jira.

In this version, there's an upgrade in compatibility for structured streaming. Additionally, this release broadens the functionality within PySpark and SQL. It adds features such as the SQL identifier clause, named arguments in SQL function calls, and the inclusion of SQL functions for HyperLogLog approximate aggregations. Other new capabilities also include Python user-defined table functions, the simplification of distributed training via *DeepSpeed*, and new structured streaming capabilities like watermark propagation and the *dropDuplicatesWithinWatermark* operation.

You can check the full list and detailed changes here:

<https://spark.apache.org/releases/spark-release-3-5-0.html ↗>.

Delta Spark

Delta Lake 3.2 marks a collective commitment to making Delta Lake interoperable across formats, easier to work with, and more performant. Delta Spark 3.2 is built on top of

Apache Spark™ 3.5 [↗](#). The Delta Spark maven artifact has been renamed from **delta-core** to **delta-spark**.

You can check the full list and detailed changes here:
<https://docs.delta.io/3.2.0/index.html> [↗](#).

Tip

For up-to-date information, a detailed list of changes, and specific release notes for Fabric runtimes, check and subscribe [Spark Runtimes Releases and Updates](#) [↗](#).

Related content

- Read about [Apache Spark Runtimes in Fabric - Overview, Versioning, Multiple Runtimes Support and Upgrading Delta Lake Protocol](#)
- [Spark Core migration guide](#) [↗](#)
- [SQL, Datasets, and DataFrame migration guides](#) [↗](#)
- [Structured Streaming migration guide](#) [↗](#)
- [MLlib \(Machine Learning\) migration guide](#) [↗](#)
- [PySpark \(Python on Spark\) migration guide](#) [↗](#)
- [SparkR \(R on Spark\) migration guide](#) [↗](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) [↗](#) | [Ask the community](#) [↗](#)

Fabric Runtime 1.2 (GA)

Article • 10/14/2024

The Microsoft Fabric Runtime is an Azure-integrated platform based on Apache Spark that enables the execution and management of data engineering and data science experiences. This document covers the Runtime 1.2 components and versions.

The major components of Runtime 1.2 include:

- Apache Spark 3.4.1
- Operating System: Mariner 2.0
- Java: 11
- Scala: 2.12.17
- Python: 3.10
- Delta Lake: 2.4.0
- R: 4.2.2

Tip

Always use the most recent, GA runtime version for your production workload, which currently is [Runtime 1.3](#).

— Runtime —

Runtime Version

Runtime version defines which version of Spark your Spark pool will use.

Learn more about Runtime Version [↗](#)

1.2 (Spark 3.4, Delta 2.4)

1.1 (Spark 3.3, Delta 2.2)

1.2 (Spark 3.4, Delta 2.4)

Microsoft Fabric Runtime 1.2 comes with a collection of default level packages, including a full Anaconda installation and commonly used libraries for Java/Scala, Python, and R. These libraries are automatically included when using notebooks or jobs in the Microsoft Fabric platform. Refer to the documentation for a complete list of libraries. Microsoft Fabric periodically rolls out maintenance updates for Runtime 1.2, providing bug fixes, performance enhancements, and security patches. *Staying up to date ensures optimal performance and reliability for your data processing tasks.*

New features and improvements of Spark Release 3.4.1

Apache Spark 3.4.0 is the fifth release in the 3.x line. This release, driven by the open-source community, resolved over 2,600 Jira tickets. It introduces a Python client for Spark Connect, enhances Structured Streaming with async progress tracking and Python stateful processing. It expands Pandas API coverage with NumPy input support, simplifies migration from traditional data warehouses through ANSI compliance and new built-in functions. It also improves development productivity and debuggability with memory profiling. Additionally, Runtime 1.2 is based on Apache Spark 3.4.1, a maintenance release focused on stability fixes.

Key highlights

Read the full version of the release notes for a specific Apache Spark version by visiting both [Spark 3.4.0](#) and [Spark 3.4.1](#).

New custom query optimizations

Concurrent Writes Support in Spark

Encountering a 404 error with the message 'Operation failed: The specified path doesn't exist' is a common issue when performing parallel data insertions into the same table using an SQL INSERT INTO query. This error can result in data loss. Our new feature, the File Output Committer Algorithm, resolves this issue, allowing customers to perform parallel data insertion seamlessly.

To access this feature, enable the `spark.sql.enable.concurrentWrites` feature flag, which is enabled by default starting from Runtime 1.2 (Spark 3.4). While this feature is also available in other Spark 3 versions, it isn't enabled by default. This feature doesn't support parallel execution of INSERT OVERWRITE queries where each concurrent job overwrites data on different partitions of the same table dynamically. For this purpose, Spark offers an alternative feature, which can be activated by configuring the `spark.sql.sources.partitionOverwriteMode` setting to [dynamic](#).

Smart reads, which skip files from failed jobs

In the current Spark committer system, when an insert into a table job fails but some tasks succeed, the files generated by the successful tasks coexist with files from the failed job. This coexistence can cause confusion for users as it becomes challenging to

distinguish between files belonging to successful and unsuccessful jobs. Moreover, when one job reads from a table while another is inserting data concurrently into the same table, the reading job might access uncommitted data. If a write job fails, the reading job could process incorrect data.

The `spark.sql.auto.cleanup.enabled` flag controls our new feature, addressing this issue. When enabled, Spark automatically skips reading files that haven't been committed when it performs `spark.read` or selects queries from a table. Files written before enabling this feature continue to be read as usual.

Here are the visible changes:

- All files now include a `tid-{jobID}` identifier in their filenames.
- Instead of the `_success` marker typically created in the output location upon successful job completion, a new `_committed_{jobID}` marker is generated. This marker associates successful Job IDs with specific filenames.
- We introduced a new SQL command that users can run periodically to manage storage and clean up uncommitted files. The syntax for this command is as follows:
 - To clean up a specific directory: `CLEANUP ('/path/to/dir') [RETAIN number HOURS];`
 - To clean up a specific table: `CLEANUP [db_name.]table_name [RETAIN number HOURS];` In this syntax, `path/to/dir` represents the location URI where cleanup is required, and `number` is a double type value representing the retention period. The default retention period is set to seven days.
- We introduced a new configuration option called `spark.sql.deleteUncommittedFilesWhileListing`, which is set to `false` by default. Enabling this option results in the automatic deletion of uncommitted files during reads, but this scenario might slow down read operations. It's recommended to manually run the cleanup command when the cluster is idle instead of enabling this flag.

Migration guide from Runtime 1.1 to Runtime 1.2

When migrating from Runtime 1.1, powered by Apache Spark 3.3, to Runtime 1.2, powered by Apache Spark 3.4, review [the official migration guide ↗](#).

New features and improvements of Delta Lake 2.4

[Delta Lake](#) is an [open source project](#) that enables building a lakehouse architecture on top of data lakes. Delta Lake provides [ACID transactions](#), scalable metadata handling, and unifies [streaming](#) and [batch](#) data processing on top of existing data lakes.

Specifically, Delta Lake offers:

- [ACID transactions](#) on Spark: Serializable isolation levels ensure that readers never see inconsistent data.
- Scalable metadata handling: Uses Spark distributed processing power to handle all the metadata for petabyte-scale tables with billions of files at ease.
- [Streaming](#) and [batch](#) unification: A table in Delta Lake is a batch table and a streaming source and sink. Streaming data ingest, batch historic backfill, interactive queries all just work out of the box.
- Schema enforcement: Automatically handles schema variations to prevent insertion of bad records during ingestion.
- [Time travel](#): Data versioning enables rollbacks, full historical audit trails, and reproducible machine learning experiments.
- [Upserts](#) and [deletes](#): Supports merge, update, and delete operations to enable complex use cases like change-data-capture, slowly changing dimension (SCD) operations, streaming upserts, and so on.

Read the full version of the release notes for [Delta Lake 2.4](#).

Default level packages for Java, Scala, Python libraries

For a list of all the default level packages for Java, Scala, Python and their respective versions see the [release notes](#).

Related content

- Read about [Apache Spark Runtimes in Fabric - Overview, Versioning, Multiple Runtimes Support and Upgrading Delta Lake Protocol](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Fabric Runtime 1.1 (deprecated)

Article • 04/01/2025

Microsoft Fabric Runtime is an Azure-integrated platform based on Apache Spark that enables the execution and management of the Data Engineering and Data Science experiences in Fabric. This document covers the Fabric Runtime 1.1 components and versions.

✖ Caution

Deprecation and disablement notification for Microsoft Fabric Runtime 1.1 Runtime 1.1, based on Apache Spark 3.3, will be **deprecated and disabled as of March 31, 2025**. The end of support date for Runtime 1.1 has been announced as July 12, 2024. [Upgrade your Fabric workspace](#) and environments to use [Runtime 1.3 \(Apache Spark 3.5 and Delta Lake 3.2\)](#). For the complete lifecycle and support policies of Apache Spark runtimes in Fabric, refer to [Lifecycle of Apache Spark runtimes in Fabric](#).

Microsoft Fabric Runtime 1.1 is one of the runtimes offered within the Microsoft Fabric platform. The Runtime 1.1 major components are:

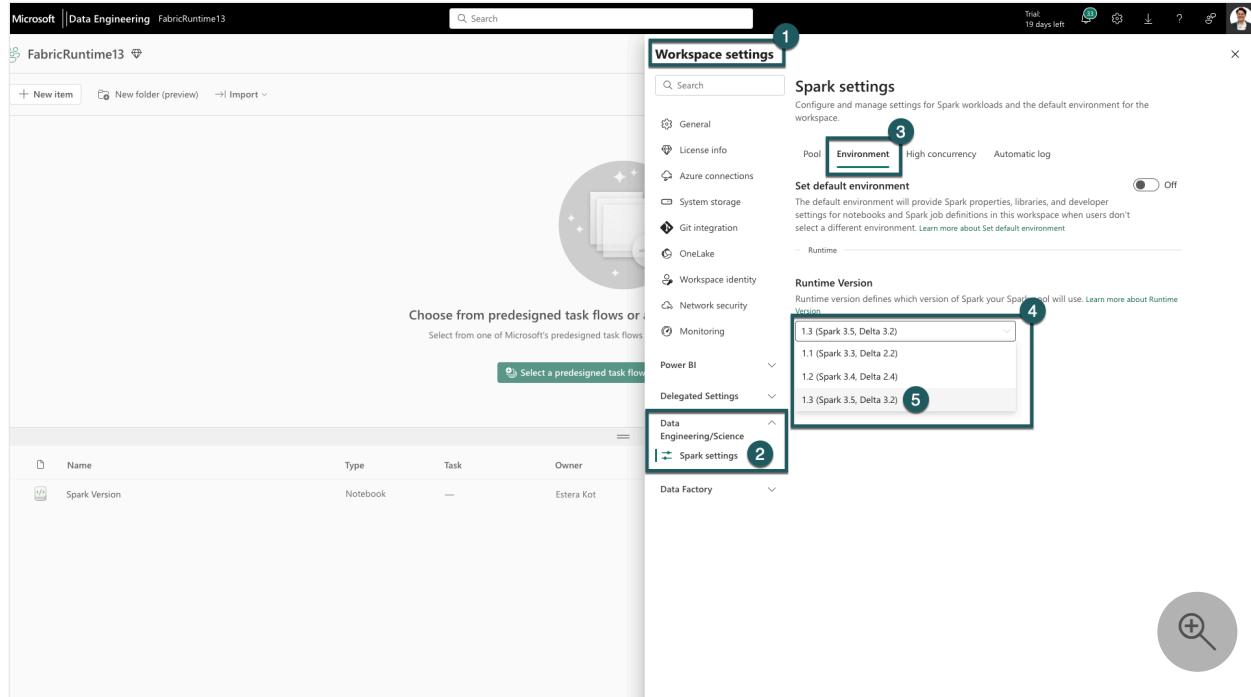
- Apache Spark 3.3
- Operating System: Ubuntu 18.04
- Java: 1.8.0_282
- Scala: 2.12.15
- Python: 3.10
- Delta Lake: 2.2
- R: 4.2.2

💡 Tip

Always use the most recent, GA runtime version for your production workload, which currently is [Runtime 1.3](#).

Microsoft Fabric Runtime 1.1 comes with a collection of default level packages, including a full Anaconda installation and commonly used libraries for Java/Scala, Python, and R. These libraries are automatically included when using notebooks or jobs in the Microsoft Fabric platform. Refer to the documentation for a complete list of libraries.

Microsoft Fabric periodically releases maintenance updates for Runtime 1.1, delivering bug fixes, performance enhancements, and security patches. Ensuring you stay up to date with these updates guarantees optimal performance and reliability for your data processing tasks. If you are currently using Runtime 1.1, you can upgrade to Runtime 1.3 or to Runtime 1.2 by navigating to **Workspace Settings > Data Engineering / Science > Spark Settings > Environment**.



New features and improvements - Apache Spark 3.3.1

Read the full version of the release notes for a specific Apache Spark version by visiting both [Spark 3.3.0](#) and [Spark 3.3.1](#).

New features and improvements - Delta Lake 2.2

Check the source and full release notes at [Delta Lake 2.2.0](#).

Default-level packages for Java/Scala

For a list of all the default level packages for Java, Scala, Python and their respective versions see the [release notes](#).

Migration between different Apache Spark versions

Migrating your workloads to Fabric Runtime 1.1 (Apache Spark 3.3) from an older version of Apache Spark involves a series of steps to ensure a smooth migration. This guide outlines the necessary steps to help you migrate efficiently and effectively.

1. Review Fabric Runtime 1.1 release notes, including checking the components and default-level packages included into the runtime, to understand the new features and improvements.
2. Check compatibility of your current setup and all related libraries, including dependencies and integrations. Review the migration guides to identify potential breaking changes:
 - Review the [Spark Core migration guide ↗](#).
 - Review the [SQL, Datasets and DataFrame migration guide ↗](#).
 - If your solution is Apache Spark Structure Streaming related, review the [Structured Streaming migration guide ↗](#).
 - If you use PySpark, review the [Pyspark migration guide ↗](#).
 - If you migrate code from Koalas to PySpark, review the [Koalas to pandas API on Spark migration guide ↗](#).
3. Move your workloads to Fabric and ensure that you have backups of your data and configuration files in case you need to revert to the previous version.
4. Update any dependencies that the new version of Apache Spark or other Fabric Runtime 1.1 related components might impact, including third-party libraries or connectors. Make sure to test the updated dependencies in a staging environment before deploying to production.
5. Update the Apache Spark configuration on your workload, including updating configuration settings, adjusting memory allocations, and modifying any deprecated configurations.
6. Modify your Apache Spark applications (notebooks and Apache Spark job definitions) to use the new APIs and features introduced in Fabric Runtime 1.1 and Apache Spark 3.3. You might need to update your code to accommodate any deprecated or removed APIs, and refactor your applications to take advantage of performance improvements and new functionalities.
7. Thoroughly test your updated applications in a staging environment to ensure compatibility and stability with Apache Spark 3.3. Perform performance testing,

functional testing, and regression testing to identify and resolve any issues that might arise during the migration process.

8. After validating your applications in a staging environment, deploy the updated applications to your production environment. Monitor the performance and stability of your applications after the migration to identify any issues that need to be addressed.
9. Update your internal documentation and training materials to reflect the changes introduced in Fabric Runtime 1.1. Ensure that your team members are familiar with the new features and improvements to maximize the benefits of the migration.

Related content

- Read about [Apache Spark Runtimes in Fabric - Overview, Versioning, Multiple Runtimes Support and Upgrading Delta Lake Protocol](#)
 - [Runtime 1.2 \(Spark 3.4, Java 11, Python 3.10, Delta Lake 2.4\)](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

How to create custom Spark pools in Microsoft Fabric

Article • 10/25/2023

In this document, we explain how to create custom Apache Spark pools in Microsoft Fabric for your analytics workloads. Apache Spark pools enable users to create tailored compute environments based on their specific requirements, ensuring optimal performance and resource utilization.

ⓘ Important

Microsoft Fabric is in [preview](#).

You specify the minimum and maximum nodes for autoscaling. Based on those values, the system dynamically acquires and retires nodes as the job's compute requirements change, which results in efficient scaling and improved performance. The dynamic allocation of executors in Spark pools also alleviates the need for manual executor configuration. Instead, the system adjusts the number of executors depending on the data volume and job-level compute needs. This process enables you to focus on your workloads without worrying about performance optimization and resource management.

ⓘ Note

To create a custom Spark pool, you need admin access to the workspace. The capacity admin must enable the **Customized workspace pools** option in the **Spark Compute** section of **Capacity Admin settings**. To learn more, see [Spark Compute Settings for Fabric Capacities](#).

Create custom Spark pools

To create or manage the Spark pool associated with your workspace:

1. Go to your workspace and select **Workspace settings**.

Fabric Testing

+ New Create a pipeline Create app Manage access Workspace settings

Name	Type	Owner
HollyTest	Lakehouse	Remy Morris
Notebook 1	Notebook	Remy Morris
Notebook 2	Notebook	Remy Morris
Notebook 3	Notebook	Remy Morris
SampleLakeHouse	Lakehouse	Remy Morris

Workspace settings

- About
- Premium
- Azure connections
- System Storage
- Other
- Power BI
- Extension API playground
- Data Engineering

2. Select the **Data Engineering/Science** option to expand the menu and then select **Spark Compute**.

Product name: <Name of workspace>

+ New Manage access Workspace settings Create app ...

Name Type Owner Refreshed

Cool data mart	Datamart	Tim Deboar	6/29/21
Cool data mart	Datamart	Tim Deboar	6/29/21
Data flow for triggers	Data flow	Tim Deboar	6/29/21
Data flow for triggers	Data flow	Tim Deboar	6/29/21
User data	Datamart	Tim Deboar	6/29/21
Copy data pipeline	Pipeline	Tim Deboar	6/29/21
Copy data pipeline	Pipeline	Tim Deboar	6/29/21
Ingestion flow	Data flow	Tim Deboar	6/29/21
Ingestion flow	Data flow	Tim Deboar	6/29/21
Contoso pipeline	Pipeline	Tim Deboar	6/29/21
Contoso pipeline	Pipeline	Tim Deboar	6/29/21
My first data flow	Data flow	Tim Deboar	6/29/21

Workspace settings

- General
- OneLake storage
- Azure connections
- Premium

Spark compute

Configure and manage settings for Spark workloads in the workspace.

Default pool for workspace

Select Default or Create Custom Pools which becomes the default pool option for workspaces and items within the capacity.

Starter pool

A starter pool with 10 nodes is provided for evaluation purposes. In the coming months, the starter pool will automatically be resized based on your purchased capacity.

Pool details

Node family	Node size	Number of nodes
Auto (Memory optimized)	Medium	3-10

Runtime version

Runtime version defines which version of Spark your Spark pool will use. Learn more

Configurations

Spark properties

3. Select the **New Pool** option. In the **Create Pool** screen, name your Spark pool. Also choose the **Node family**, and select a **Node size** from the available sizes (**Small**, **Medium**, **Large**, **X-Large**, and **XX-Large**) based on compute requirements for your workloads.



Create pool

Spark pool name *

Node family

Node size

Small

Medium

Large

X-Large

XX-Large

Enable autoscale



4. You can set the minimum node configuration for your custom pools to 1. Because Fabric Spark provides restorable availability for clusters with a single node, you don't have to worry about job failures, loss of session during failures, or over paying on compute for smaller Spark jobs.
5. You can enable or disable autoscaling for your custom Spark pools. When autoscaling is enabled, the pool will dynamically acquire new nodes up to the maximum node limit specified by the user, and then retire them after job execution. This feature ensures better performance by adjusting resources based on the job requirements. You're allowed to size the nodes, which fit within the capacity units purchased as part of the Fabric capacity SKU.

Edit pool

Spark pool name *

customlargepool

Node family

Memory optimized

Node size

Large

Autoscale

If enabled, your Apache Spark pool will automatically scale up and down based on the amount of activity.

Enable autoscale



Dynamically allocate executors

Enable allocate



6. You can also choose to enable dynamic executor allocation for your Spark pool, which automatically determines the optimal number of executors within the user-specified maximum bound. This feature adjusts the number of executors based on data volume, resulting in improved performance and resource utilization.

These custom pools have a default autopause duration of 2 minutes. Once the autopause duration is reached, the session expires and the clusters are unallocated. You're charged based on the number of nodes and the duration for which the custom Spark pools are used.

Next steps

- Learn more from the Apache Spark [public documentation](#).
- Get started with [Spark workspace administration settings in Microsoft Fabric](#).

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

High concurrency mode in Apache Spark for Fabric

Article • 04/09/2025

High concurrency mode allows users to share the same Spark sessions in Spark for Fabric for data engineering and data science workloads. An item like a notebook uses a standard Spark session for its execution. In high concurrency mode, the Spark session can support independent execution of multiple items within individual read-eval-print loop (REPL) cores that exist within the Spark application. These REPL cores provide isolation for each item, and prevent local notebook variables from being overwritten by variables with the same name from other notebooks sharing the same session.

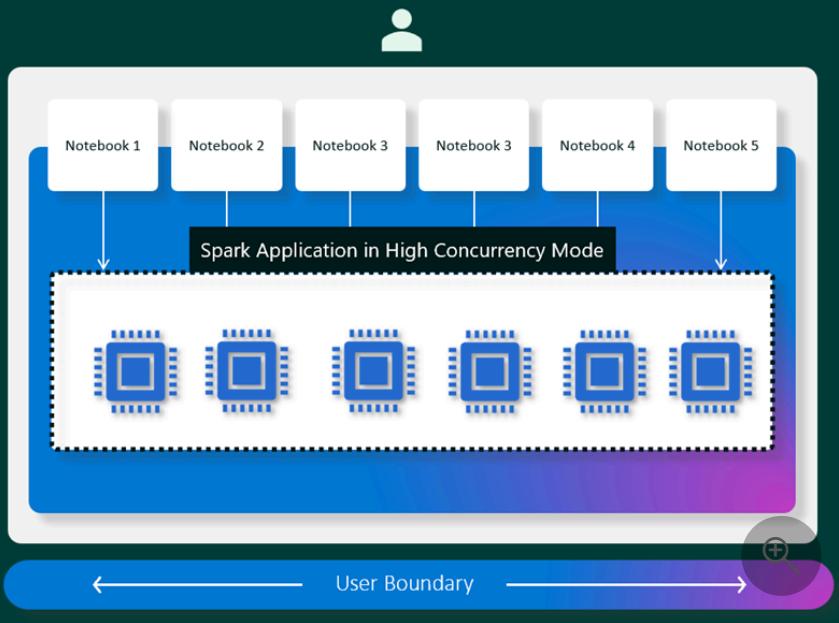
As the session is already running, this provides users with an instant run experience when reusing the session across multiple notebooks.

ⓘ Note

In the case of custom pools with high concurrency mode, users get 36X faster session start experience compared to a standard Spark session.

How does it work?

- **Security**: Session sharing is always within a single user boundary
- **Multitask**: Users can seamlessly switch between notebooks and continue their work without experiencing any delays due to session creation or initialization.
- **Cost-effective**: Allows users to achieve better resource utilization and cost savings for their data engineering / science workloads.



ⓘ Important

Session sharing conditions include:

- Sessions should be within a single user boundary.
- Sessions should have the same default lakehouse configuration.

- Sessions should have the same Spark compute properties.

As part of Spark session initialization, a REPL core is created. Every time a new item starts sharing the same session and the executors are allocated in FAIR based manner to these notebooks running in these REPL cores inside the Spark application preventing starvation scenarios.

Billing of High Concurrency Sessions

When using high concurrency mode, **only the initiating session** that starts the shared Spark application is billed. All subsequent sessions that share the same Spark session **do not incur additional billing**. This approach enables cost optimization for teams and users running multiple concurrent workloads in a shared context.

Example:

- A user starts **Notebook 1**, which initiates a Spark session in high concurrency mode.
- The same session is then shared by **Notebook 2**, **Notebook 3**, **Notebook 4**, and **Notebook 5**.
- In this case, **only Notebook 1 will be billed** for the Spark compute usage.
- The shared notebooks (2 to 5) **will not be billed** individually.

This billing behavior is also reflected in **Capacity Metrics** — usage will only be reported against the initiating notebook (Notebook 1 in this case).

Note

The same billing behavior applies when high concurrency mode is used within **pipeline activities** — only the notebook or activity that initiates the Spark session is charged.

Related content

- To get started with high concurrency mode in notebooks, see [Configure high concurrency mode for Fabric notebooks](#).

What is autotune for Apache Spark configurations in Fabric?

Article • 06/11/2024

Autotune automatically adjusts Apache Spark configuration to speed up workload execution and to optimize overall performance. Autotune saves time and resources compared to manual tuning which, requires extensive effort, resources, time, and experimentation. Autotune uses historical execution data from your workloads to iteratively discover and apply the most effective configurations for a specific workload.

ⓘ Note

The autotune query tuning feature in Microsoft Fabric is currently in preview. Autotune is available across all production regions but is disabled by default. You can activate it through the Spark configuration setting within the environment or within a single session by including the respective Spark setting in your Spark notebook or Spark Job Definition code.

Query tuning

Autotune configures three Apache Spark settings for each of your queries separately:

- `spark.sql.shuffle.partitions` - Sets the partition count for data shuffling during joins or aggregations. The default value is 200.
- `spark.sql.autoBroadcastJoinThreshold` - Sets the maximum table size in bytes that is broadcasted to all worker nodes when join operation is executed. The default value is 10 MB.
- `spark.sql.files.maxPartitionBytes` - Defines the maximum number of bytes to pack into a single partition when reading files. Works for Parquet, JSON, and ORC file-based sources. Default is 128 MB.

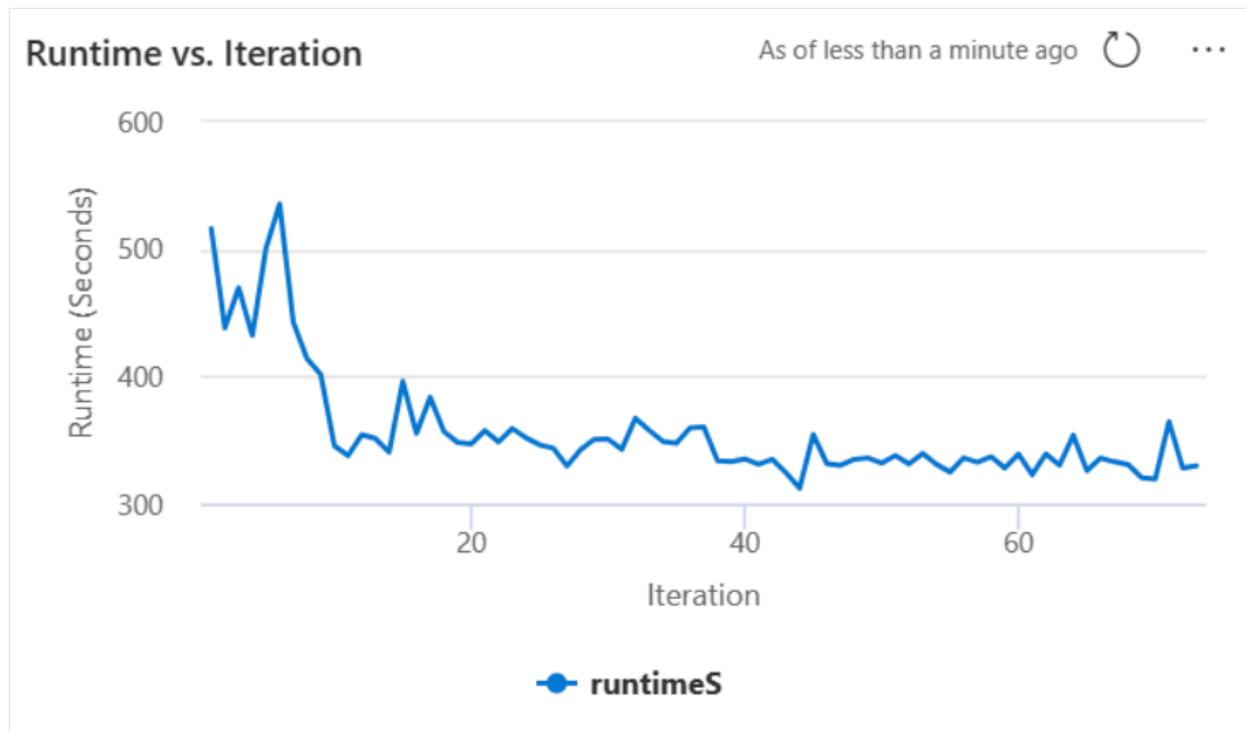
💡 Tip

Autotune query tuning examines individual queries and builds a distinct ML model for each query. It specifically targets:

- Repetitive queries
- Long-running queries (those with more than 15 seconds of execution)

- Apache Spark SQL API queries (excluding those written in the RDD API, which are very rare), but we optimize all queries regardless the language (Scala, PySpark, R, Spark SQL)

This feature is compatible with notebooks, Apache Spark job definitions, and pipelines. The benefits vary based on the complexity of the query, the methods used, and the structure. Extensive testing has shown that the greatest advantages are realized with queries related to exploratory data analysis, such as reading data, running joins, aggregations, and sorting.



AI-based intuition behind the Autotune

The autotune feature utilizes an iterative process to optimize query performance. It begins with a default configuration and employs a machine learning model to evaluate effectiveness. When a user submits a query, the system retrieves the stored models based on the previous interactions. It generates potential configurations around a default setting named *centroid*. The best candidate predicted by the model, is applied. After query execution, the performance data is sent back to the system to refine the model.

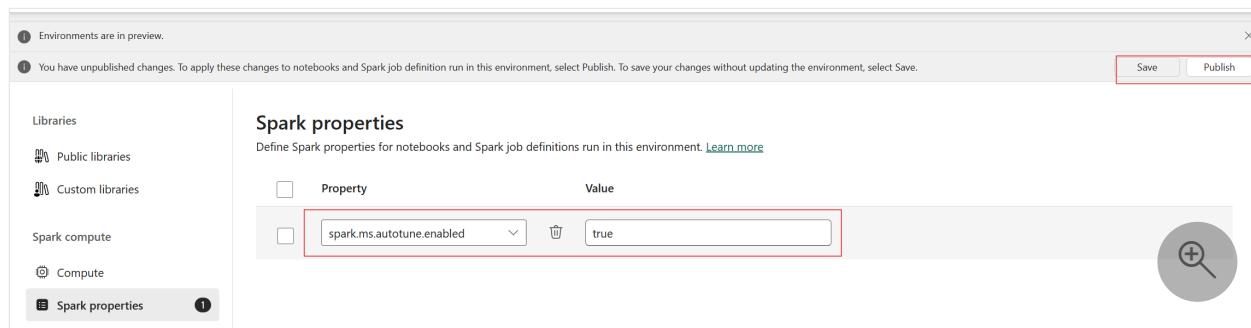
The feedback loop gradually shifts the centroid towards optimal settings. It refines performance over time while minimizing the risk of regression. Continuous updates based on user queries enable refinement of performance benchmarks. Moreover, the process updates the *centroid* configurations to ensure the model moves towards more efficient settings incrementally. This is achieved by evaluating past performances and

using them to guide future adjustments. It uses all the data points to mitigate the impact of anomalies.

From a responsible AI perspective, the Autotune feature includes transparency mechanisms designed to keep you informed about your data usage and benefits. The security and privacy align with Microsoft's standards. Ongoing monitoring maintains performance and system integrity post-launch.

Enable autotune

Autotune is available across all production regions but is disabled by default. You can activate it through the Spark configuration setting within the environment. To enable Autotune, either create a new environment or, for the existing environment, set the Spark property 'spark.ms.autotune.enabled = true' as shown in the screenshot below. This setting is then inherited by all notebooks and jobs running in that environment, automatically tuning them.



Autotune includes a built-in mechanism for monitoring performance and detecting performance regressions. For instance, if a query processes an unusually large amount of data, Autotune will automatically deactivate. It typically requires 20 to 25 iterations to learn and identify the optimal configuration.

Note

The Autotune is compatible with [Fabric Runtime 1.1](#) and [Runtime 1.2](#). Autotune doesn't function when [the high concurrency mode](#) or when the [private endpoint](#) is enabled. However, autotune seamlessly integrates with autoscaling, regardless of its configuration.

You can enable autotune within a single session by including the respective Spark setting in your Spark notebook or Spark Job Definition code.

Spark SQL

SQL

```
%%sql  
SET spark.ms.autotune.enabled=TRUE
```

You can control Autotune through Spark settings for your respective Spark notebook or Spark job definition code. To disable Autotune, execute the following commands as the first cell (notebook) or line of the code (SJD).

Spark SQL

SQL

```
%%sql  
SET spark.ms.autotune.enabled=FALSE
```

Case study

When executing an Apache Spark query, autotune creates a customized ML model dedicated to optimizing the query's execution. It analyzes query patterns and resource needs. Consider an initial query filtering a dataset based on a specific attribute, such as a country. While this example uses geographic filtering, the principle applies universally to any attribute or operation within the query:

Python

```
%%pyspark  
df.filter(df.country == "country-A")
```

Autotune learns from this query, optimizing subsequent executions. When the query changes, for instance, by altering the filter value or applying a different data transformation, the structural essence of the query often remains consistent:

Python

```
%%pyspark  
df.filter(df.country == "country-B")
```

Despite alterations, autotune identifies the fundamental structure of the new query, implementing previously learned optimizations. This capability ensures sustained high efficiency without the need for manual reconfiguration for each new query iteration.

Logs

For each of your queries, autotune determines the most optimal settings for three Spark configurations. You can view the suggested settings by navigating to the logs. The configurations recommended by autotune are located in the driver logs, specifically those entries starting with *[Autotune]*.

The screenshot shows the Databricks interface with the 'Logs' tab selected. In the logs pane, there are several log entries. One entry from March 13, 2024, at 06:43:27,224 is highlighted with a callout labeled [Autotune]. This entry indicates that Autotune query tuning is enabled. Another callout points to the 'Run details' sidebar, which provides runtime information such as the status being stopped and the total duration being 4 minutes and 36 seconds.

You can find various types of entries in your logs. The following include the key ones:

Expand table

Status	Description
AUTOTUNE_DISABLED	Skipped. Autotune is disabled; preventing telemetry data retrieval and query optimization. Enable Autotune to fully use its capabilities while respecting customer privacy.".
QUERY_TUNING_DISABLED	Skipped. Autotune query tuning is disabled. Enable it to fine-tune settings for your Spark SQL queries.
QUERY_PATTERN_NOT_MATCH	Skipped. Query pattern did not match. Autotune is effective for read-only queries.

Status	Description
QUERY_DURATION_TOO_SHORT	Skipped. Your query duration too short to optimize. Autotune requires longer queries for effective tuning. Queries should run for at least 15 seconds.
QUERY_TUNING_SUCCEED	Success. Query tuning completed. Optimal spark settings applied.

Transparency note

In adherence to the Responsible AI Standard, this section aims to clarify the uses and validation of the Autotune feature, promoting transparency and enabling informed decision-making.

Purpose of Autotune

Autotune is developed to enhance Apache Spark workload efficiency, primarily for data professionals. Its key functions include:

- Automating Apache Spark configuration tuning to reduce execution times.
- Minimizing manual tuning efforts.
- Utilizing historical workload data to refine configurations iteratively.

Validation of Autotune

Autotune has undergone extensive testing to ensure its effectiveness and safety:

- Rigorous tests with diverse Spark workloads to verify tuning algorithm efficacy.
- Benchmarking against standard Spark optimization methods to demonstrate performance benefits.
- Real-world case studies highlighting Autotune's practical value.
- Adherence to strict security and privacy standards to safeguard user data.

User data is exclusively used to enhance your workload's performance, with robust protections to prevent misuse or exposure of sensitive information.

Related content

- [Concurrency limits and queueing in Apache Spark for Microsoft Fabric](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Concurrency limits and queueing in Apache Spark for Microsoft Fabric

Article • 03/31/2025

Applies to:  Data Engineering and Data Science in Microsoft Fabric

Microsoft Fabric allows allocation of compute units through capacity, which is a dedicated set of resources that is available at a given time to be used. Capacity defines the ability of a resource to perform an activity or to produce output. Different items consume different capacity at a certain time. Microsoft Fabric offers capacity through the Fabric SKUs and trials. For more information, see [What is capacity?](#).

When users create a Microsoft Fabric capacity on Azure, they choose a capacity size based on their analytics workload size. In Apache Spark, users get two Apache Spark Vcores for every capacity unit they reserve as part of their SKU.

One Capacity Unit = Two Spark Vcores

Once they have purchased the capacity, admins can create workspaces within the capacity in Microsoft Fabric. The Spark Vcores associated with the capacity are shared among all the Apache Spark-based items like notebooks, Apache Spark job definitions, and lakehouses created in these workspaces.

Concurrency throttling and queueing

Spark for Fabric enforces a cores-based throttling and queueing mechanism, where users can submit jobs based on the purchased Fabric capacity SKUs. The queueing mechanism is a simple FIFO-based queue, which checks for available job slots and automatically retries the jobs once the capacity has become available. When users submit notebook or lakehouse jobs like Load to Table when their capacity is at its maximum utilization due to concurrent running jobs using all the Spark Vcores available for their purchased Fabric capacity SKU, they're throttled with the message

HTTP Response code 430: This Spark job can't be run because you have hit a Spark compute or API rate limit. To run this Spark job, cancel an active Spark job through the Monitoring hub, or choose a larger capacity SKU or try again later.

With queueing enabled, notebook jobs triggered from pipelines and job scheduler and Spark job definitions are added to the queue and automatically retried when the capacity is freed up. The queue expiration is set to 24 hours from the job submission time. After this period, the jobs will need to be resubmitted.

Fabric capacities are enabled with bursting which allows you to consume extra compute cores beyond what have been purchased to speed the execution of a workload. For Apache Spark workloads bursting allows users to submit jobs with a total of 3X the Spark VCores purchased.

 **Note**

The bursting factor only increases the total number of Spark VCores to help with the concurrency but doesn't increase the max cores per job. Users can't submit a job that requires more cores than what their Fabric capacity offers.

The following section lists various cores-based limits for Spark workloads based on Microsoft Fabric capacity SKUs:

 [Expand table](#)

Fabric capacity SKU	Equivalent Power BI SKU	Spark VCores	Max Spark VCores with Burst Factor	Queue limit
F2	-	4	20	4
F4	-	8	24	4
F8	-	16	48	8
F16	-	32	96	16
F32	-	64	192	32
F64	P1	128	384	64
F128	P2	256	768	128
F256	P3	512	1536	256
F512	P4	1024	3072	512
F1024	-	2048	6144	1024
F2048	-	4096	12288	2048
Trial Capacity	P1	128	128	NA

 **Important**

The table applies only to Spark jobs running on Fabric Capacity. With autoscale billing enabled, Spark jobs run separately from Fabric capacity, avoiding bursting or

smoothing. The total Spark vCores will be twice the maximum capacity Units set in autoscale settings.

Example calculation: *F64 SKU* offers *128 Spark VCores*. The burst factor applied for a F64 SKU is 3, which gives a total of 384 Spark Vcores. The burst factor is only applied to help with concurrency and doesn't increase the max cores available for a single Spark job. That means *a single Notebook or Spark job definition or lakehouse job* can use a pool configuration of max 128 vCores and 3 jobs with the same configuration can be run concurrently. If notebooks are using a smaller compute configuration, they can be run concurrently till the max utilization reaches the 384 SparkVcore limit.

ⓘ Note

The jobs have a queue expiration period of 24 hours, after which they're canceled, and users must resubmit them for job execution.

Spark for Fabric throttling doesn't have enforced arbitrary jobs-based limits, and the throttling is only based on the number of cores allowed for the purchased Fabric capacity SKU. The job admission by default will be an optimistic admission control, where the jobs are admitted based on their minimum cores requirement. Learn more about the optimistic job admission [Job Admission and Management](#) If the default pool (Starter Pool) option is selected for the workspace, the following table lists the max concurrency job limits.

Learn more about the default starter pool configurations based on the Fabric Capacity SKU [Configuring Starter Pools](#).

Job level bursting

Admins can configure their Apache Spark pools to utilize the max Spark cores with burst factor available for the entire capacity. For example a workspace admin having their workspace attached to a F64 Fabric capacity can now configure their Spark pool (Starter pool or Custom pool) to 384 Spark VCores, where the max nodes of Starter pools can be set to 48 or admins can set up an XX Large node size pool with six max nodes.

Related content

- Get started with [Apache Spark workspace administration settings in Microsoft Fabric](#).

- Learn about the [Apache Spark compute for Fabric](#) data engineering and data science experiences.
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

What is an Apache Spark job definition?

Article • 12/01/2023

An Apache Spark job definition is a Microsoft Fabric code item that allows you to submit batch/streaming jobs to Spark clusters. By uploading the binary files from the compilation output of different languages (for example, .jar from Java), you can apply different transformation logic to the data hosted on a lakehouse. Besides the binary file, you can further customize the behavior of the job by uploading more libraries and command line arguments.

To run a Spark job definition, you must have at least one lakehouse associated with it. This default lakehouse context serves as the default file system for Spark runtime. For any Spark code using a relative path to read/write data, the data is served from the default lakehouse.

Tip

To run a Spark job definition item, you must have a main definition file and default lakehouse context. If you don't have a lakehouse, create one by following the steps in [Create a lakehouse](#).

Related content

- [How to create an Apache Spark job definition in Fabric](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

How to create an Apache Spark job definition in Fabric

Article • 01/17/2025

In this tutorial, learn how to create a Spark job definition in Microsoft Fabric.

Prerequisites

Before you get started, you need:

- A Fabric tenant account with an active subscription. [Create an account for free.](#)

Tip

To run the Spark job definition item, you must have a main definition file and default lakehouse context. If you don't have a lakehouse, you can create one by following the steps in [Create a lakehouse](#).

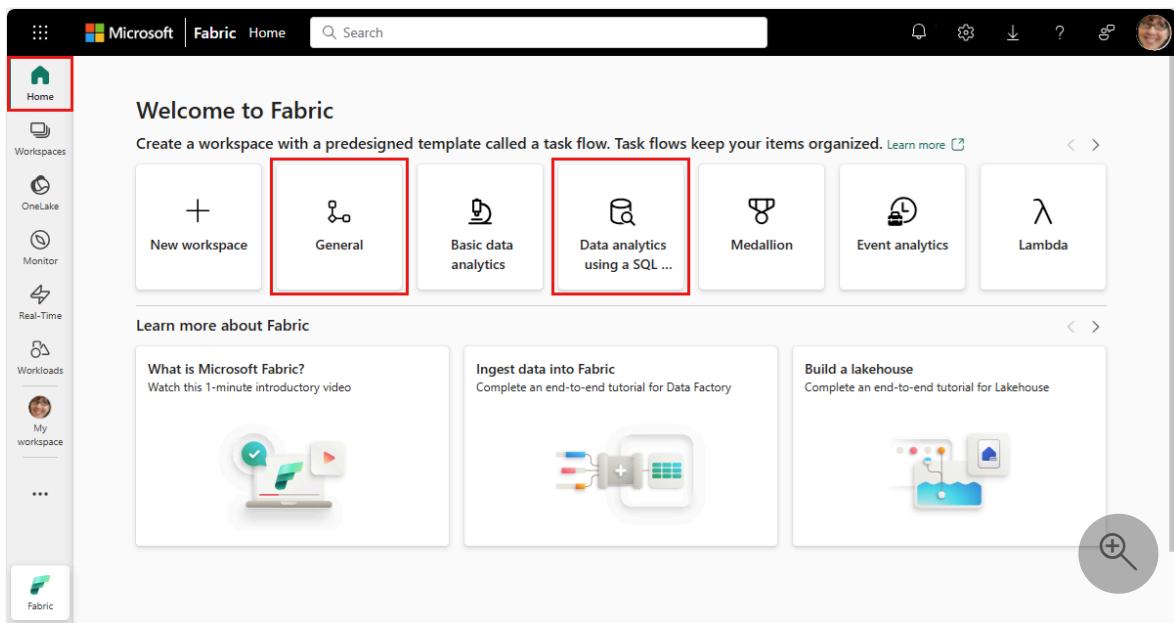
Create a Spark job definition

The Spark job definition creation process is quick and simple; there are several ways to get started.

Options to create a Spark job definition

There are two ways you can get started with the creation process:

- **Workspace view:** You can easily create a Spark job definition through the **Fabric workspace** by selecting **New item > Spark Job Definition**.
- **Fabric Home:** Another entry point to create a Spark job definition is the **Data analytics using a SQL ... tile** on the Fabric home page. You can find the same option by selecting the **General** tile.



You need to give your Spark job definition a name when you create it. The name must be unique within the current workspace. The new Spark job definition is created in your current workspace.

Create a Spark job definition for PySpark (Python)

To create a Spark job definition for PySpark:

1. Download the sample Parquet file [yellow_tripdata_2022-01.parquet](#) and upload it to the files section of the lakehouse.
2. Create a new Spark job definition.
3. Select **PySpark (Python)** from the **Language** dropdown.
4. Download the [createTablefromParquet.py](#) sample and upload it as the main definition file. The main definition file (*job.Main*) is the file that contains the application logic and is mandatory to run a Spark job. For each Spark job definition, you can only upload one main definition file.

You can upload the main definition file from your local desktop, or you can upload from an existing Azure Data Lake Storage (ADLS) Gen2 by providing the full ABFSS path of the file. For example, `abfss://your-storage-account-name.dfs.core.windows.net/your-file-path`.

5. Upload reference files as *.py* files. The reference files are the python modules that are imported by the main definition file. Just like the main definition file, you can upload from your desktop or an existing ADLS Gen2. Multiple reference files are supported.

💡 Tip

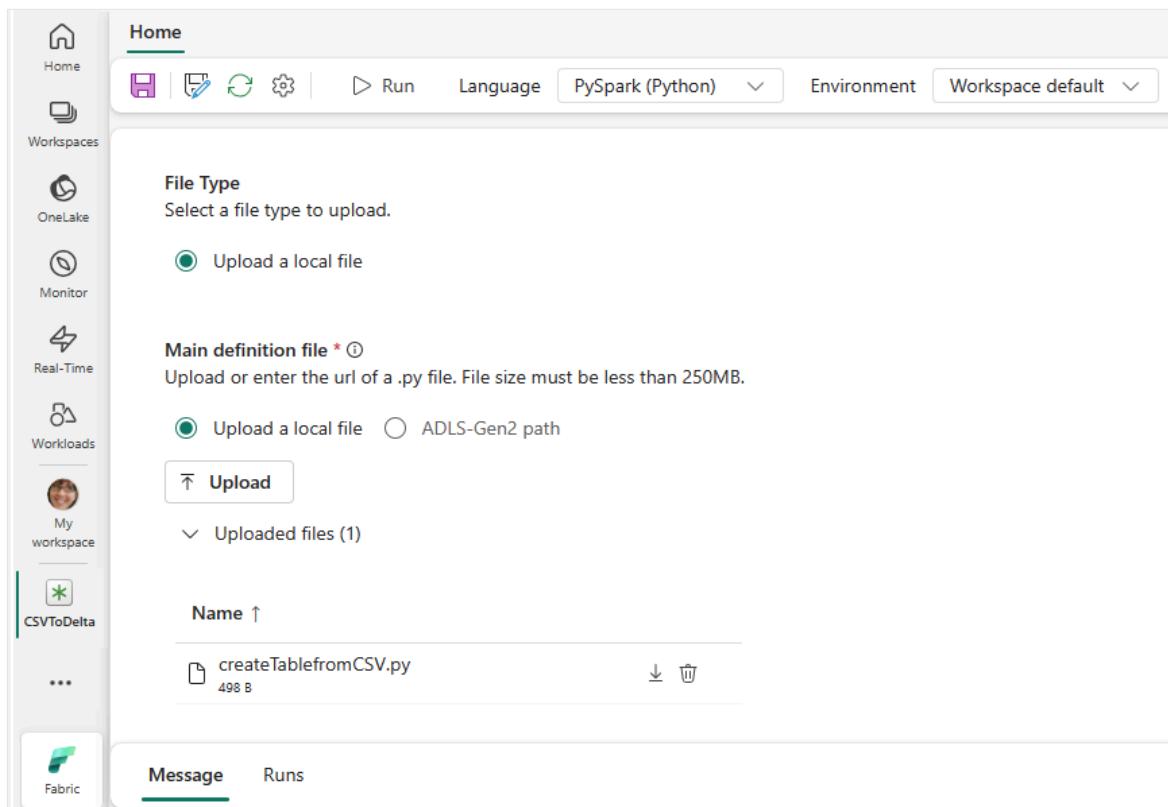
If you use an ADLS Gen2 path, to make sure the file is accessible, you must give the user account that runs the job the proper permission to the storage account. We suggest two different ways to do this:

- Assign the user account a Contributor role for the storage account.
- Grant Read and Execution permission to the user account for the file via the ADLS Gen2 Access Control List (ACL).

For a manual run, the account of the current login user is used to run the job.

6. Provide command line arguments for the job, if needed. Use a space as a splitter to separate the arguments.
7. Add the lakehouse reference to the job. You must have at least one lakehouse reference added to the job. This lakehouse is the default lakehouse context for the job.

Multiple lakehouse references are supported. Find the non-default lakehouse name and full OneLake URL in the **Spark Settings** page.



Create a Spark job definition for Scala/Java

To create a Spark job definition for Scala/Java:

1. Create a new Spark job definition.
2. Select **Spark(Scala/Java)** from the **Language** dropdown.
3. Upload the main definition file as a *.jar* file. The main definition file is the file that contains the application logic of this job and is mandatory to run a Spark job. For each Spark job definition, you can only upload one main definition file. Provide the Main class name.
4. Upload reference files as *.jar* files. The reference files are the files that are referenced/imported by the main definition file.
5. Provide command line arguments for the job, if needed.
6. Add the lakehouse reference to the job. You must have at least one lakehouse reference added to the job. This lakehouse is the default lakehouse context for the job.

Create a Spark job definition for R

To create a Spark job definition for SparkR(R):

1. Create a new Spark job definition.
2. Select **SparkR(R)** from the **Language** dropdown.
3. Upload the main definition file as an *.R* file. The main definition file is the file that contains the application logic of this job and is mandatory to run a Spark job. For each Spark job definition, you can only upload one main definition file.
4. Upload reference files as *.R* files. The reference files are the files that are referenced/imported by the main definition file.
5. Provide command line arguments for the job, if needed.
6. Add the lakehouse reference to the job. You must have at least one lakehouse reference added to the job. This lakehouse is the default lakehouse context for the job.

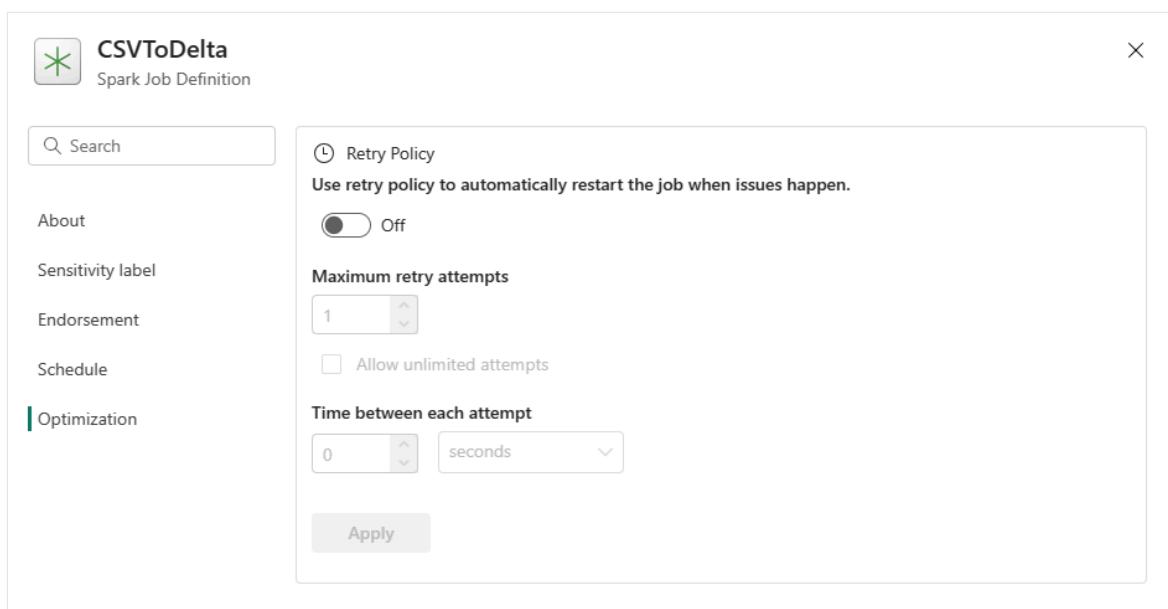
Note

The Spark job definition will be created in your current workspace.

Options to customize Spark job definitions

There are a few options to further customize the execution of Spark job definitions.

- **Spark Compute:** Within the **Spark Compute** tab, you can see [the Runtime Version](#) which is the version of Spark that will be used to run the job. You can also see the Spark configuration settings that will be used to run the job. You can customize the Spark configuration settings by clicking on the **Add** button.
- **Optimization:** On the **Optimization** tab, you can enable and set up the **Retry Policy** for the job. When enabled, the job is retried if it fails. You can also set the maximum number of retries and the interval between retries. For each retry attempt, the job is restarted. Make sure the job is **idempotent**.



Related content

- [Run an Apache Spark job definition](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Schedule and run an Apache Spark job definition

Article • 11/29/2023

Learn how to run a Microsoft Fabric Apache Spark job definition and find the job definition status and details.

Prerequisites

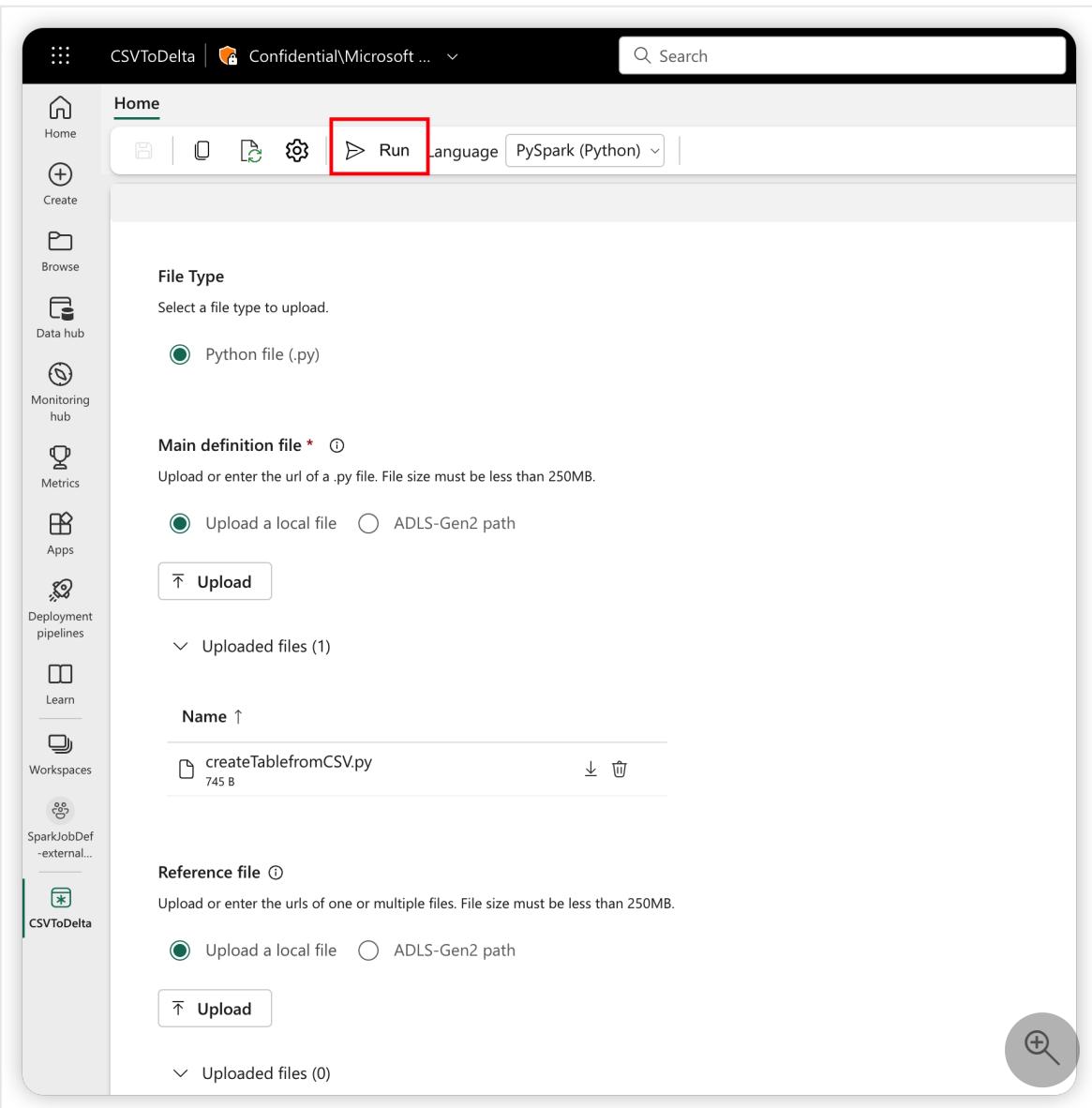
Before you get started, you must:

- Create a Microsoft Fabric tenant account with an active subscription. [Create an account for free](#).
- Understand the Spark job definition: see [What is an Apache Spark job definition?](#).
- Create a Spark job definition: see [How to create an Apache Spark job definition in Fabric](#).

How to run a Spark job definition

There are two ways you can run a Spark job definition:

- Run a Spark job definition manually by selecting **Run** from the Spark job definition item in the job list.



- Schedule a Spark job definition by setting up a schedule plan on the **Settings** tab. Select **Settings** on the toolbar, then select **Schedule**.

The screenshot shows the 'CSVToDelta' Spark Job Definition page. On the left, there's a sidebar with links: About, Sensitivity label, Endorsement, Schedule (which is selected and highlighted with a red box), Spark compute, and Optimization. The main area displays the last success information ('Last success is in April 19, 2023 at 1:57:22 AM (UTC) Coordinated Universal Time') and a note that the scheduled refresh is turned off. Below this is a large 'Schedule' configuration panel. It includes a 'Scheduled run' section with an 'On' radio button selected, a 'Repeat' section with a dropdown menu 'Select a frequency', and two date pickers for 'Start' and 'End'. The 'Time zone' dropdown is set to '(UTC+08:00) Beijing, Chongqing, Hong Kong...'. At the bottom of the panel are 'Apply' and search buttons.

ⓘ Important

To run, a Spark job definition must have a main definition file and a default lakehouse context.

💡 Tip

For a manual run, the account of the currently logged in user is used to submit the job. For a run triggered by a schedule, the account of the user who created the schedule plan is used to submit the job.

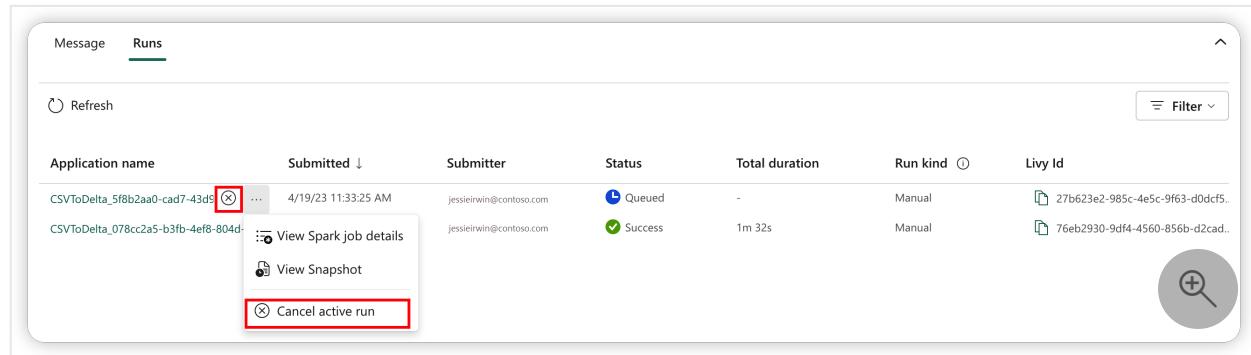
Three to five seconds after you've submitted the run, a new row appears under the **Runs** tab. The row shows details about your new run. The **Status** column shows the near real-time status of the job, and the **Run kind** column shows if the job is manual or scheduled.

The screenshot shows the 'Runs' tab of the Azure Data Factory interface. The table has columns: Application name, Submitted (with a sort arrow), Submitter, Status, Total duration, Run kind, and Livy Id. One row is present: 'CSVToDelta_078cc2a5-b3fb-4ef8-804d...' with a timestamp of '4/19/23 9:57:25 AM', 'jessieirwin@contoso.com' as the submitter, 'Success' status, '1m 32s' duration, 'Manual' run kind, and a Livy ID starting with '76eb2930-9df4-4560-856b-d2cad...'. A 'Refresh' button and a 'Filter' dropdown are at the top, and a search icon is on the right.

For more information on how to monitor a job, see [Monitor your Apache Spark job definition](#).

How to cancel a running job

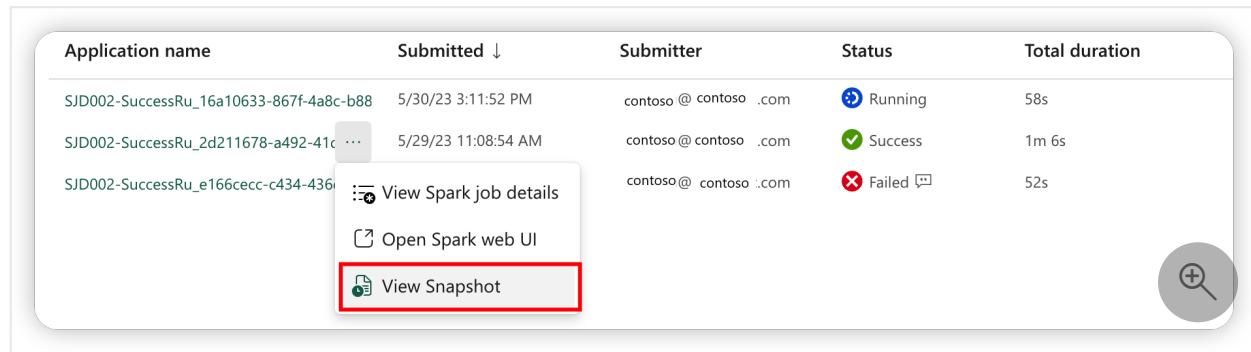
Once the job is submitted, you can cancel the job by selecting **Cancel active run** from the Spark job definition item in the job list.



Application name	Submitted ↓	Submitter	Status	Total duration	Run kind	Livy Id
CSVToDelta_5f8b2aa0-cad7-43d9...	4/19/23 11:33:25 AM	jessieirwin@contoso.com	Queued	-	Manual	27b623e2-985c-4e5c-9f63-d0dcf5...
CSVToDelta_078cc2a5-b3fb-4ef8-804d...	4/19/23 11:33:25 AM	jessieirwin@contoso.com	Success	1m 32s	Manual	76eb2930-9df4-4560-856b-d2cad...

Spark job definition snapshot

The Spark job definition stores its latest state. To view the snapshot of the history run, select **View Snapshot** from the Spark job definition item in the job list. The snapshot shows the state of the job definition when the job is submitted, including the main definition file, the reference file, the command line arguments, the referenced lakehouse, and the Spark properties.



Application name	Submitted ↓	Submitter	Status	Total duration
SJD002-SuccessRu_16a10633-867f-4a8c-b88...	5/30/23 3:11:52 PM	contoso @ contoso .com	Running	58s
SJD002-SuccessRu_2d211678-a492-41c...	5/29/23 11:08:54 AM	contoso @ contoso .com	Success	1m 6s
SJD002-SuccessRu_e166cecc-c434-436...	5/29/23 11:08:54 AM	contoso @ contoso .com	Failed	52s

From a snapshot, you can take three actions:

- **Save as a Spark job definition:** Save the snapshot as a new Spark job definition.
- **Open Spark job definition:** Open the current Spark job definition.
- **Restore:** Restore the job definition with the snapshot. The job definition is restored to the state when the job was submitted.

The screenshot shows the Microsoft Fabric interface for a successful Spark job run. The top navigation bar includes 'Refresh', 'Cancel', and 'Spark history server'. Below it, tabs for 'Jobs', 'Logs', 'Data', and 'Related items' are present, with 'Related items' being the active tab. A breadcrumb trail indicates the path: 'SparkJobDefinitionUserResearch > SJD002-SuccessRun > SJD002-SuccessRun_2d211678-a492-41dd-934a-f930da36d98f'. The main content area displays 'Snapshot views: SJD002-SuccessRun' and details about the job, such as 'Language: PySpark (Python)'. A context menu is open over the job name 'SJD002-SuccessRun', with options: 'Save as a Spark job definition' (selected), 'Open Spark job definition', and 'Restore'. A red box highlights the first two options. At the bottom of the menu, it says 'Main definition file' and shows 'createTablefromCSVwithdependnecy.py'. A search icon is located in the bottom right corner.

Related content

- [Microsoft Spark Utilities \(MSSparkUtils\) for Fabric](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

How to create and update a Spark Job Definition with Microsoft Fabric Rest API

Article • 11/29/2023

The Microsoft Fabric Rest API provides a service endpoint for CRUD operations of Fabric items. In this tutorial, we walk through an end-to-end scenario of how to create and update a Spark Job Definition artifact. Three high-level steps are involved:

1. create a Spark Job Definition item with some initial state
2. upload the main definition file and other lib files
3. update the Spark Job Definition item with the OneLake URL of the main definition file and other lib files

Prerequisites

1. An Entra token is required to access the Fabric Rest API. The MSAL library is recommended to get the token. For more information, see [Authentication flow support in MSAL](#).
2. A storage token is required to access the OneLake API. For more information, see [MSAL for Python](#).

Create a Spark Job Definition item with the initial state

The Microsoft Fabric Rest API defines a unified endpoint for CRUD operations of Fabric items. The endpoint is

`https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/items`

The item detail is specified inside the request body. Here's an example of the request body for creating a Spark Job Definition item:

JSON

```
{  
  "displayName": "SJDHelloWorld",  
  "type": "SparkJobDefinition",  
  "definition": {  
    "format": "SparkJobDefinitionV1",  
    "parts": [  
      {  
        "path": "SparkJobDefinitionV1.json",  
        "type": "File",  
        "size": 1024  
      }  
    ]  
  }  
}
```

```

    "payload": "eyJleGVjdXRhYmxlRmlsZSI6bnVsbCwiZGVmYXVsdExha2Vob3VzZUFydGlmYWN0S
WQiOiiLCJtYWluQ2xhc3MiOiiLCJhZGRpdGlvbmFsTGFrZWhvdXN1SWRzIjpbXSwickmV0cn1Qb
2xpY3kiOm51bGwsImNvbW1hbmRMaW51QXJndW1lbnRzIjoiIiwiYWRkaXRpb25hbExpYnJhcn1Vc
mlzIjpbXSwibGFuZ3VhZ2UiOiiLCJlbnZpcm9ubWVudEFydGlmYWN0SWQiOm51bGx9",
    "payloadType": "InlineBase64"
}
]
}
}

```

In this example, the Spark Job Definition item is named as `SJDHelloWorld`. The `payload` field is the base64 encoded content of the detail setup, after decoding, the content is:

JSON

```
{
  "executableFile": null,
  "defaultLakehouseArtifactId": "",
  "mainClass": "",
  "additionalLakehouseIds": [],
  "retryPolicy": null,
  "commandLineArguments": "",
  "additionalLibraryUris": [],
  "language": "",
  "environmentArtifactId": null
}
```

Here are two helper functions to encode and decode the detailed setup:

Python

```

import base64

def json_to_base64(json_data):
    # Serialize the JSON data to a string
    json_string = json.dumps(json_data)

    # Encode the JSON string as bytes
    json_bytes = json_string.encode('utf-8')

    # Encode the bytes as Base64
    base64_encoded = base64.b64encode(json_bytes).decode('utf-8')

    return base64_encoded

def base64_to_json(base64_data):
    # Decode the Base64-encoded string to bytes
    base64_bytes = base64_data.encode('utf-8')

    # Decode the bytes to a JSON string

```

```

json_string = base64.b64decode(base64_bytes).decode('utf-8')

# Deserialize the JSON string to a Python dictionary
json_data = json.loads(json_string)

return json_data

```

Here's the code snippet to create a Spark Job Definition item:

Python

```

import requests

bearerToken = "breadcrumb"; # replace this token with the real AAD token

headers = {
    "Authorization": f"Bearer {bearerToken}",
    "Content-Type": "application/json" # Set the content type based on your
request
}

payload =
"eyJleGVjdXRhYmxlRmlsZSI6bnVsbCwiZGVmYXVsdExha2Vob3VzZUFydGlmYWN0SWQiOiiLCJ
tYwluQ2xhc3MiOiiLCJhZGRpdGvbmfTsTGFrdWhvdXN1SWRzIjpbXSwicmV0cn1Qb2xpY3kiOm5
1bGwsImNvbW1hbRMaw5lQXJndW1bnRzIjoiIiwiYWRkaXRpb25hbExpYnJhcnlVcmLzIjpbXSw
ibGFuZ3VhZ2UiOiiLCJlbnZpcm9ubWVudEFydGlmYWN0SWQiOm51bGx9"

# Define the payload data for the POST request
payload_data = {
    "displayName": "SJDHelloWorld",
    "Type": "SparkJobDefinition",
    "definition": {
        "format": "SparkJobDefinitionV1",
        "parts": [
            {
                "path": "SparkJobDefinitionV1.json",
                "payload": payload,
                "payloadType": "InlineBase64"
            }
        ]
    }
}

# Make the POST request with Bearer authentication
response = requests.post(sjdCreateUrl, json=payload_data, headers=headers)

```

Upload the main definition file and other lib files

A storage token is required to upload the file to OneLake. Here's a helper function to get the storage token:

Python

```
import msal

def getOnelakeStorageToken():
    app = msal.PublicClientApplication(
        "{client id}", # this field should be the client id
        authority="https://login.microsoftonline.com/microsoft.com")

    result = app.acquire_token_interactive(scopes=
        ["https://storage.azure.com/.default"])

    print(f"Successfully acquired AAD token with storage audience:
{result['access_token']}")

    return result['access_token']
```

Now we have a Spark Job Definition item created, to make it runnable, we need to set up the main definition file and required properties. The endpoint for uploading the file for this SJD item is

<https://onelake.dfs.fabric.microsoft.com/{workspaceId}/{sjdartifactid}>. The same "workspaceId" from the previous step should be used, the value of "sjdartifactid" could be found in the response body of the previous step. Here's the code snippet to set up the main definition file:

Python

```
import requests

# three steps are required: create file, append file, flush file

onelakeEndPoint =
"https://onelake.dfs.fabric.microsoft.com/workspaceId/sjdartifactid"; #
replace the id of workspace and artifact with the right one
mainExecutableFile = "main.py"; # the name of the main executable file
mainSubFolder = "Main"; # the sub folder name of the main executable file.
Don't change this value

onelakeRequestMainFileCreateUrl = f"
{onelakeEndPoint}/{mainSubFolder}/{mainExecutableFile}?resource=file" # the
url for creating the main executable file via the 'file' resource type
onelakePutRequestHeaders = {
    "Authorization": f"Bearer {onelakeStorageToken}", # the storage token
    can be achieved from the helper function above
```

```

}

onelakeCreateMainFileResponse =
requests.put(onelakeRequestMainFileCreateUrl,
headers=onelakePutRequestHeaders)
if onelakeCreateMainFileResponse.status_code == 201:
    # Request was successful
    print(f"Main File '{mainExecutableFile}' was successfully created in
onelake.")

# with previous step, the main executable file is created in OneLake, now we
need to append the content of the main executable file

appendPosition = 0;
appendAction = "append";

### Main File Append.
mainExecutableFileSizeInBytes = 83; # the size of the main executable file
in bytes
onelakeRequestMainFileAppendUrl = f"
{onelakeEndPoint}/{mainSubFolder}/{mainExecutableFile}?position=
{appendPosition}&action={appendAction}";
mainFileContents = "filename = 'Files/' + Constant.filename; tablename =
'Tables/' + Constant.tablename"; # the content of the main executable file,
please replace this with the real content of the main executable file
mainExecutableFileSizeInBytes = 83; # the size of the main executable file
in bytes, this value should match the size of the mainFileContents

onelakePatchRequestHeaders = {
    "Authorization": f"Bearer {onelakeStorageToken}",
    "Content-Type" : "text/plain"
}

onelakeAppendMainFileResponse =
requests.patch(onelakeRequestMainFileAppendUrl, data = mainFileContents,
headers=onelakePatchRequestHeaders)
if onelakeAppendMainFileResponse.status_code == 202:
    # Request was successful
    print(f"Successfully Accepted Main File '{mainExecutableFile}' append
data.")

# with previous step, the content of the main executable file is appended to
the file in OneLake, now we need to flush the file

flushAction = "flush";

### Main File flush
onelakeRequestMainFileFlushUrl = f"
{onelakeEndPoint}/{mainSubFolder}/{mainExecutableFile}?position=
{mainExecutableFileSizeInBytes}&action={flushAction}"
print(onelakeRequestMainFileFlushUrl)
onelakeFlushMainFileResponse =
requests.patch(onelakeRequestMainFileFlushUrl,
headers=onelakePatchRequestHeaders)
if onelakeFlushMainFileResponse.status_code == 200:

```

```

        print(f"Successfully Flushed Main File '{mainExecutableFile}'"
contents.")
else:
    print(onelakeFlushMainFileResponse.json())

```

Follow the same process to upload the other lib files if needed.

Update the Spark Job Definition item with the OneLake URL of the main definition file and other lib files

Until now, we have created a Spark Job Definition item with some initial state, uploaded the main definition file and other lib files, The last step is to update the Spark Job Definition item to set the URL properties of the main definition file and other lib files.

The endpoint for updating the Spark Job Definition item is

<https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/items/{sjdartifactid}>.

The same "workspaceId" and "sjdartifactid" from previous steps should be used. Here's the code snippet to update the Spark Job Definition item:

Python

```

mainAbfssPath =
f"abfss://{workspaceId}@onelake.dfs.fabric.microsoft.com/{sjdartifactid}/Mai
n/{mainExecutableFile}" # the workspaceId and sjdartifactid are the same as
previous steps, the mainExecutableFile is the name of the main executable
file
libsAbfssPath =
f"abfss://{workspaceId}@onelake.dfs.fabric.microsoft.com/{sjdartifactid}/Lib
s/{libsFile}" # the workspaceId and sjdartifactid are the same as previous
steps, the libsFile is the name of the libs file
defaultLakehouseId = 'defaultLakehouseid'; # replace this with the real
default lakehouse id

updateRequestBodyJson = {
    "executableFile":mainAbfssPath,
    "defaultLakehouseArtifactId":defaultLakehouseId,
    "mainClass":"",
    "additionalLakehouseIds":[],
    "retryPolicy":None,
    "commandLineArguments":"",
    "additionalLibraryUris": [libsAbfssPath],
    "language":"Python",
    "environmentArtifactId":None}

# Encode the bytes as a Base64-encoded string

```

```

base64EncodedUpdateSJDPayload = json_to_base64(updateRequestBodyJson)

# Print the Base64-encoded string
print("Base64-encoded JSON payload for SJD Update:")
print(base64EncodedUpdateSJDPayload)

# Define the API URL
updateSjdUrl =
f"https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/items/{sjdartifactid}/updateDefinition"

updatePayload = base64EncodedUpdateSJDPayload
payloadType = "InlineBase64"
path = "SparkJobDefinitionV1.json"
format = "SparkJobDefinitionV1"
Type = "SparkJobDefinition"

# Define the headers with Bearer authentication
bearerToken = "breadcrumb"; # replace this token with the real AAD token

headers = {
    "Authorization": f"Bearer {bearerToken}",
    "Content-Type": "application/json" # Set the content type based on your
request
}

# Define the payload data for the POST request
payload_data = {
    "displayName": "sjdCreateTest11",
    "Type": Type,
    "definition": {
        "format": format,
        "parts": [
            {
                "path": path,
                "payload": updatePayload,
                "payloadType": payloadType
            }
        ]
    }
}

# Make the POST request with Bearer authentication
response = requests.post(updateSjdUrl, json=payload_data, headers=headers)
if response.status_code == 200:
    print("Successfully updated SJD.")
else:
    print(response.json())
    print(response.status_code)

```

To recap the whole process, both Fabric REST API and OneLake API are needed to create and update a Spark Job Definition item. The Fabric REST API is used to create and update the Spark Job Definition item, the OneLake API is used to upload the main definition file and other lib files. The main definition file and other lib files are uploaded to OneLake first. Then the URL properties of the main definition file and other lib files are set in the Spark Job Definition item.

Related content

- [Schedule and run an Apache Spark job definition](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Microsoft Spark Utilities (MSSparkUtils) for Fabric

Article • 11/19/2024

Microsoft Spark Utilities (MSSparkUtils) is a built-in package to help you easily perform common tasks. You can use MSSparkUtils to work with file systems, to get environment variables, to chain notebooks together, and to work with secrets. The MSSparkUtils package is available in PySpark (Python) Scala, SparkR notebooks, and Fabric pipelines.

⚠ Note

- MsSparkUtils has been officially renamed to [NotebookUtils](#). The existing code will remain **backward compatible** and won't cause any breaking changes. It is **strongly recommend** upgrading to notebookutils to ensure continued support and access to new features. The mssparkutils namespace will be retired in the future.
- NotebookUtils is designed to work with **Spark 3.4(Runtime v1.2) and above**. All new features and updates will be exclusively supported with notebookutils namespace going forward.

File system utilities

`mssparkutils.fs` provides utilities for working with various file systems, including Azure Data Lake Storage (ADLS) Gen2 and Azure Blob Storage. Make sure you configure access to [Azure Data Lake Storage Gen2](#) and [Azure Blob Storage](#) appropriately.

Run the following commands for an overview of the available methods:

Python

```
from notebookutils import mssparkutils
mssparkutils.fs.help()
```

Output

Console

`mssparkutils.fs` provides utilities for working with various FileSystems.

Below is overview about the available methods:

```
cp(from: String, to: String, recurse: Boolean = false): Boolean -> Copies a file or directory, possibly across FileSystems
mv(from: String, to: String, recurse: Boolean = false): Boolean -> Moves a file or directory, possibly across FileSystems
ls(dir: String): Array -> Lists the contents of a directory
mkdirs(dir: String): Boolean -> Creates the given directory if it does not exist, also creating any necessary parent
directories
put(file: String, contents: String, overwrite: Boolean = false): Boolean -> Writes the given String out to a file, encoded
in UTF-8
head(file: String, maxBytes: int = 1024 * 100): String -> Returns up to the first 'maxBytes' bytes of the given file as a
String encoded in UTF-8
append(file: String, content: String, createFileIfNotExists: Boolean): Boolean -> Append the content to a file
rm(dir: String, recurse: Boolean = false): Boolean -> Removes a file or directory
exists(file: String): Boolean -> Check if a file or directory exists
mount(source: String, mountPoint: String, extraConfigs: Map[String, Any]): Boolean -> Mounts the given remote storage
directory at the given mount point
unmount(mountPoint: String): Boolean -> Deletes a mount point
mounts(): Array[MountPointInfo] -> Show information about what is mounted
getMountPath(mountPoint: String, scope: String = ""): String -> Gets the local path of the mount point
```

Use `mssparkutils.fs.help("methodName")` for more info about a method.

MSSparkUtils works with the file system in the same way as Spark APIs. Take `mssparkutils.fs.mkdirs()` and Fabric lakehouse usage for example:

Expand table

Usage	Relative path from HDFS root	Absolute path for ABFS file system
Nondefault lakehouse	Not supported	<code>mssparkutils.fs.mkdirs("abfss://<container_name>@<storage_account_name>.dfs.core.windows.net/<new_dir>"</code>

Usage	Relative path from HDFS root	Absolute path for ABFS file system
Default lakehouse	Directory under "Files" or "Tables": <code>mssparkutils.fs.mkdirs("abfss://<container_name>@<storage_account_name>.dfs.core.windows.net/<new_dir>"</code>	

List files

To list the content of a directory, use `mssparkutils.fs.ls('Your directory path')`. For example:

Python

```
mssparkutils.fs.ls("Files/tmp") # works with the default lakehouse files using relative path
mssparkutils.fs.ls("abfss://<container_name>@<storage_account_name>.dfs.core.windows.net/<path>") # based on ABFS file system
mssparkutils.fs.ls("file:/tmp") # based on local file system of driver node
```

View file properties

This method returns file properties including file name, file path, file size, and whether it's a directory and a file.

Python

```
files = mssparkutils.fs.ls('Your directory path')
for file in files:
    print(file.name, file.isDir, file.isFile, file.path, file.size)
```

Create new directory

This method creates the given directory if it doesn't exist, and creates any necessary parent directories.

Python

```
mssparkutils.fs.mkdirs('new directory name')
mssparkutils.fs.mkdirs("Files/<new_dir>") # works with the default lakehouse files using relative path
mssparkutils.fs.ls("abfss://<container_name>@<storage_account_name>.dfs.core.windows.net/<new_dir>") # based on ABFS file system
mssparkutils.fs.ls("file:/<new_dir>") # based on local file system of driver node
```

Copy file

This method copies a file or directory, and supports copy activity across file systems.

Python

```
mssparkutils.fs.cp('source file or directory', 'destination file or directory', True)# Set the third parameter as True to copy all files and directories recursively
```

Performant copy file

This method provides a faster way of copying or moving files, especially large volumes of data.

Python

```
mssparkutils.fs.fastcp('source file or directory', 'destination file or directory', True)# Set the third parameter as True to copy all files and directories recursively
```

Preview file content

This method returns up to the first 'maxBytes' bytes of the given file as a String encoded in UTF-8.

Python

```
mssparkutils.fs.head('file path', maxBytes to read)
```

Move file

This method moves a file or directory, and supports moves across file systems.

Python

```
mssparkutils.fs.mv('source file or directory', 'destination directory', True) # Set the last parameter as True to firstly  
create the parent directory if it does not exist  
mssparkutils.fs.mv('source file or directory', 'destination directory', True, True) # Set the third parameter to True to  
firstly create the parent directory if it does not exist. Set the last parameter to True to overwrite the updates.
```

Write file

This method writes the given string out to a file, encoded in UTF-8.

Python

```
mssparkutils.fs.put("file path", "content to write", True) # Set the last parameter as True to overwrite the file if it  
existed already
```

Append content to a file

This method appends the given string to a file, encoded in UTF-8.

Python

```
mssparkutils.fs.append("file path", "content to append", True) # Set the last parameter as True to create the file if it  
does not exist
```

ⓘ Note

When using the `mssparkutils.fs.append` API in a `for` loop to write to the same file, we recommend to add a `sleep` statement around 0.5s~1s between the recurring writes. This is because the `mssparkutils.fs.append` API's internal `flush` operation is asynchronous, so a short delay helps ensure data integrity.

Delete file or directory

This method removes a file or directory.

Python

```
mssparkutils.fs.rm('file path', True) # Set the last parameter as True to remove all files and directories recursively
```

Mount/unmount directory

Find more information about detailed usage in [File mount and unmount](#).

Notebook utilities

Use the MSSparkUtils Notebook Utilities to run a notebook or exit a notebook with a value. Run the following command to get an overview of the available methods:

Python

```
mssparkutils.notebook.help()
```

Output:

Console

```
exit(value: String): void -> This method lets you exit a notebook with a value.  
run(path: String, timeoutSeconds: int, arguments: Map): String -> This method runs a notebook and returns its exit value.
```

⚠ Note

Notebook utilities aren't applicable for Apache Spark job definitions (SJD).

Reference a notebook

This method references a notebook and returns its exit value. You can run nesting function calls in a notebook interactively or in a pipeline. The notebook being referenced runs on the Spark pool of the notebook that calls this function.

Python

```
mssparkutils.notebook.run("notebook_name", <timeoutSeconds>, <parameterMap>, <workspaceId>)
```

For example:

Python

```
mssparkutils.notebook.run("Sample1", 90, {"input": 20 })
```

Fabric notebook also supports referencing notebooks across multiple workspaces by specifying the *workspace ID*.

Python

```
mssparkutils.notebook.run("Sample1", 90, {"input": 20 }, "fe0a6e2a-a909-4aa3-a698-0a651de790aa")
```

You can open the snapshot link of the reference run in the cell output. The snapshot captures the code run results and allows you to easily debug a reference run.

The screenshot shows the Fabric notebook interface. At the top, there's a section titled "Notebook Reference run" with a dropdown arrow. Below it, a command is shown in a code editor:

```
1 mssparkutils.notebook.run("Notebook 2", 90, {"input": 50})
```

Output from this command is displayed:

[3] ✓ 11 sec - Command executed in 10 sec 568 ms by on 9:43:21 PM, 5/05/23

Details:

- PySpark (Python) ↗
- View notebook run: Notebook 2
- 'Hello, Notebook 2 executed successfully'

Below this, there's a "Code" and "Markdown" button. The main area shows a "Snapshot(s): Notebook 2" section with a "Save as copy" button. It displays three code cells:

```
1 input = "10"
```

```
1 # This cell is generated from runtime parameters. Learn more: https://go.microsoft.com/fwlink/?linkid=2161015  
2 input = 50  
3
```

```
1 print(input)
```

Output for the first cell is "50". On the right side, there's a "Details" panel with the following information:

Snapshot ID	...
Livy ID	...
Job end time	5/5/23 9:43:19 PM
Duration	9 sec
Submitter	
Default lakehouse	

⚠ Note

- The cross-workspace reference notebook is supported by **runtime version 1.2 and above**.
- If you use the files under **Notebook Resource**, use `mssparkutils.nbResPath` in the referenced notebook to make sure it points to the same folder as the interactive run.

Reference run multiple notebooks in parallel

ⓘ Important

This feature is in [preview](#).

The method `mssparkutils.notebook.runMultiple()` allows you to run multiple notebooks in parallel or with a predefined topological structure. The API is using a multi-thread implementation mechanism within a spark session, which means the compute resources are shared by the reference notebook runs.

With `mssparkutils.notebook.runMultiple()`, you can:

- Execute multiple notebooks simultaneously, without waiting for each one to finish.
- Specify the dependencies and order of execution for your notebooks, using a simple JSON format.
- Optimize the use of Spark compute resources and reduce the cost of your Fabric projects.
- View the Snapshots of each notebook run record in the output, and debug/monitor your notebook tasks conveniently.
- Get the exit value of each executive activity and use them in downstream tasks.

You can also try to run the `mssparkutils.notebook.help("runMultiple")` to find the example and detailed usage.

Here's a simple example of running a list of notebooks in parallel using this method:

Python

```
mssparkutils.notebook.runMultiple(["NotebookSimple", "NotebookSimple2"])
```

The execution result from the root notebook is as follows:

[4]	<pre>1 # run multi notebooks 2 mssparkutils.notebook.runMultiple(["NotebookSimple", "NotebookSimple2"]) ✓ - Command executed in 10 sec 1 ms by</pre>	PySpark (Python) ▾																					
<hr/>																							
	<table border="1"> <thead> <tr> <th>Activity name</th><th>Snapshot</th><th>Status</th><th>Progress</th><th>Duration</th><th>Exit value</th><th>Exception</th></tr> </thead> <tbody> <tr> <td>0</td><td>NotebookSimple</td><td>✓ Succeeded</td><td><div style="width: 100%;">100%</div></td><td>8 sec 889 ms</td><td>goodbye: default value</td><td>-</td></tr> <tr> <td>1</td><td>NotebookSimple2</td><td>✓ Succeeded</td><td><div style="width: 100%;">100%</div></td><td>8 sec 887 ms</td><td>hello: default value</td><td>-</td></tr> </tbody> </table>	Activity name	Snapshot	Status	Progress	Duration	Exit value	Exception	0	NotebookSimple	✓ Succeeded	<div style="width: 100%;">100%</div>	8 sec 889 ms	goodbye: default value	-	1	NotebookSimple2	✓ Succeeded	<div style="width: 100%;">100%</div>	8 sec 887 ms	hello: default value	-	
Activity name	Snapshot	Status	Progress	Duration	Exit value	Exception																	
0	NotebookSimple	✓ Succeeded	<div style="width: 100%;">100%</div>	8 sec 889 ms	goodbye: default value	-																	
1	NotebookSimple2	✓ Succeeded	<div style="width: 100%;">100%</div>	8 sec 887 ms	hello: default value	-																	
	<pre>{"0": {"exitVal": "goodbye: default value", "exception": None}, "1": {"exitVal": "hello: default value", "exception": None}}</pre>																						

The following is an example of running notebooks with topological structure using `mssparkutils.notebook.runMultiple()`. Use this method to easily orchestrate notebooks through a code experience.

Python

```
# run multiple notebooks with parameters
DAG = {
    "activities": [
        {
            "name": "NotebookSimple", # activity name, must be unique
            "path": "NotebookSimple", # notebook path
            "timeoutPerCellInSeconds": 90, # max timeout for each cell, default to 90 seconds
            "args": {"p1": "changed value", "p2": 100}, # notebook parameters
        },
        {
            "name": "NotebookSimple2",
            "path": "NotebookSimple2",
            "timeoutPerCellInSeconds": 120,
            "args": {"p1": "changed value 2", "p2": 200}
        },
    ],
}
```

```

        "name": "NotebookSimple2.2",
        "path": "NotebookSimple2",
        "timeoutPerCellInSeconds": 120,
        "args": {"p1": "changed value 3", "p2": 300},
        "retry": 1,
        "retryIntervalInSeconds": 10,
        "dependencies": ["NotebookSimple"] # list of activity names that this activity depends on
    }
],
"timeoutInSeconds": 43200, # max timeout for the entire DAG, default to 12 hours
"concurrency": 50 # max number of notebooks to run concurrently, default to 50
}
mssqlutils.notebook.runMultiple(DAG, {"displayDAGViaGraphviz": False})

```

The execution result from the root notebook is as follows:

The screenshot shows a PySpark (Python) notebook interface. The code cell contains the DAG definition, which runs three notebooks sequentially: NotebookSimple, NotebookSimple2, and NotebookSimple2.2. Each notebook has specific parameters like path, timeout, and args. The log section shows the execution results for each activity:

Activity name	Snapshot	Status	Progress	Duration	Exit value	Exception
NotebookSimple	NotebookSimple	✓ Succeeded	<div style="width: 100%;">100%</div>	9 sec 804 ms	goodbye: changed value	-
NotebookSimple2	NotebookSimple2	✓ Succeeded	<div style="width: 100%;">100%</div>	7 sec 985 ms	hello: changed value 2	-
NotebookSimple2.2	NotebookSimple2	✓ Succeeded	<div style="width: 100%;">100%</div>	9 sec 539 ms	hello: changed value 3	-

At the bottom of the log, there is a JSON object showing the exit values for each activity: {'NotebookSimple': {'exitVal': 'goodbye: changed value', 'exception': None}, 'NotebookSimple2': {'exitVal': 'hello: changed value 2', 'exception': None}, 'NotebookSimple2.2': {'exitVal': 'hello: changed value 3', 'exception': None}}.

① Note

- The parallelism degree of the multiple notebook run is restricted to the total available compute resource of a Spark session.
- The upper limit for notebook activities or concurrent notebooks is 50. Exceeding this limit may lead to stability and performance issues due to high compute resource usage. If issues arise, consider separating notebooks into multiple `runMultiple` calls or reducing the concurrency by adjusting the `concurrency` field in the DAG parameter.
- The default timeout for entire DAG is 12 hours, and the default timeout for each cell in child notebook is 90 seconds. You can change the timeout by setting the `timeoutInSeconds` and `timeoutPerCellInSeconds` fields in the DAG parameter.

Exit a notebook

This method exits a notebook with a value. You can run nesting function calls in a notebook interactively or in a pipeline.

- When you call an `exit()` function from a notebook interactively, the Fabric notebook throws an exception, skips running subsequent cells, and keeps the Spark session alive.
- When you orchestrate a notebook in a pipeline that calls an `exit()` function, the notebook activity returns with an exit value, completes the pipeline run, and stops the Spark session.
- When you call an `exit()` function in a notebook that is being referenced, Fabric Spark will stop the further execution of the referenced notebook, and continue to run the next cells in the main notebook that calls the `run()` function. For example: Notebook1 has three cells and calls an `exit()` function in the second cell. Notebook2 has five cells and calls `run(notebook1)` in the third cell. When you run

Notebook2, Notebook1 stops at the second cell when hitting the `exit()` function. Notebook2 continues to run its fourth cell and fifth cell.

Python

```
mssparkutils.notebook.exit("value string")
```

For example:

Sample1 notebook with following two cells:

- Cell 1 defines an `input` parameter with default value set to 10.
- Cell 2 exits the notebook with `input` as exit value.

```
1 input = "10"
[ ] Press shift + enter to run
```



```
1 mssparkutils.notebook.exit("Notebook executed successfully with exit value"+ str(input))
[ ] Press shift + enter to run
```

+ Code + Markdown

You can run the Sample1 in another notebook with default values:

Python

```
exitVal = mssparkutils.notebook.run("Sample1")
print (exitVal)
```

Output:

Console

```
Notebook executed successfully with exit value 10
```

You can run the Sample1 in another notebook and set the `input` value as 20:

Python

```
exitVal = mssparkutils.notebook.run("Sample1", 90, {"input": 20 })
print (exitVal)
```

Output:

Console

```
Notebook executed successfully with exit value 20
```

Credentials utilities

You can use the MSSparkUtils Credentials Utilities to get access tokens and manage secrets in an Azure Key Vault.

Run the following command to get an overview of the available methods:

Python

```
mssparkutils.credentials.help()
```

Output:

Console

```
getToken(audience, name): returns AAD token for a given audience, name (optional)
getSecret(keyvault_endpoint, secret_name): returns secret for a given Key Vault and secret name
```

Get token

getToken returns a Microsoft Entra token for a given audience and name (optional). The following list shows the currently available audience keys:

- Storage Audience Resource: "storage"
- Power BI Resource: "pbi"
- Azure Key Vault Resource: "keyvault"
- Synapse RTA KQL DB Resource: "kusto"

Run the following command to get the token:

Python

```
mssparkutils.credentials.getToken('audience Key')
```

Get secret using user credentials

getSecret returns an Azure Key Vault secret for a given Azure Key Vault endpoint and secret name using user credentials.

Python

```
mssparkutils.credentials.getSecret('https://<name>.vault.azure.net/', 'secret name')
```

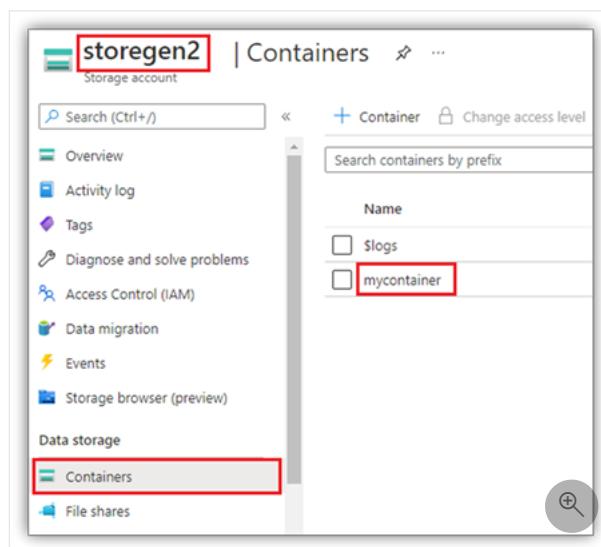
File mount and unmount

Fabric supports the following mount scenarios in the Microsoft Spark Utilities package. You can use the *mount*, *unmount*, *getMountPath()*, and *mounts()* APIs to attach remote storage (ADLS Gen2) to all working nodes (driver node and worker nodes). After the storage mount point is in place, use the local file API to access data as if it's stored in the local file system.

How to mount an ADLS Gen2 account

The following example illustrates how to mount Azure Data Lake Storage Gen2. Mounting Blob Storage works similarly.

This example assumes that you have one Data Lake Storage Gen2 account named *storegen2*, and the account has one container named *mycontainer* that you want to mount to */test* into your notebook Spark session.

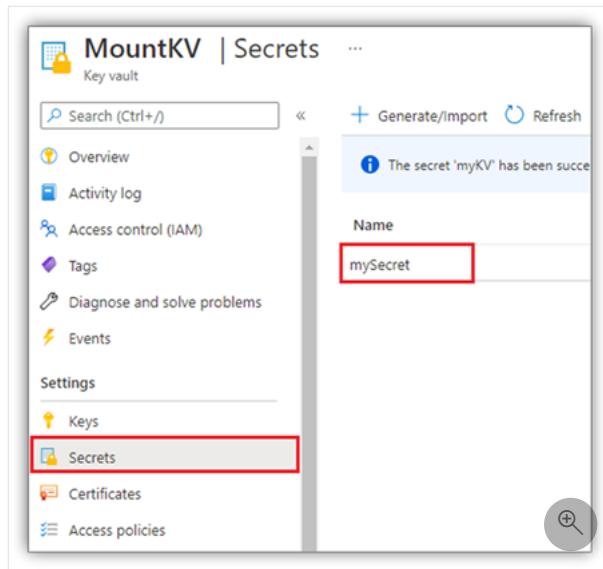


To mount the container called *mycontainer*, *mssparkutils* first needs to check whether you have the permission to access the container. Currently, Fabric supports two authentication methods for the trigger mount operation: *accountKey* and *sastoken*.

Mount via shared access signature token or account key

MSSparkUtils supports explicitly passing an account key or [Shared access signature \(SAS\)](#) token as a parameter to mount the target.

For security reasons, we recommend that you store account keys or SAS tokens in Azure Key Vault (as the following screenshot shows). You can then retrieve them by using the `mssparkutils.credentials.getSecret` API. For more information about Azure Key Vault, see [About Azure Key Vault managed storage account keys](#).



Sample code for the `accountKey` method:

Python

```
from notebookutils import mssparkutils
# get access token for keyvault resource
# you can also use full audience here like https://vault.azure.net
accountKey = mssparkutils.credentials.getSecret("<vaultURI>", "<secretName>")
mssparkutils.fs.mount(
    "abfss://mycontainer@<accountname>.dfs.core.windows.net",
    "/test",
    {"accountKey":accountKey}
)
```

Sample code for `sastoken`:

Python

```
from notebookutils import mssparkutils
# get access token for keyvault resource
# you can also use full audience here like https://vault.azure.net
sasToken = mssparkutils.credentials.getSecret("<vaultURI>", "<secretName>")
mssparkutils.fs.mount(
    "abfss://mycontainer@<accountname>.dfs.core.windows.net",
    "/test",
    {"sasToken":sasToken}
)
```

Note

You might need to import `mssparkutils` if it's not available:

Python

```
from notebookutils import mssparkutils
```

Mount parameters:

- `fileCacheTimeout`: Blobs will be cached in the local temp folder for 120 seconds by default. During this time, blobfuse will not check whether the file is up to date or not. The parameter could be set to change the default timeout time. When multiple clients modify

files at the same time, in order to avoid inconsistencies between local and remote files, we recommend shortening the cache time, or even changing it to 0, and always getting the latest files from the server.

- **timeout:** The mount operation timeout is 120 seconds by default. The parameter could be set to change the default timeout time. When there are too many executors or when mount times out, we recommend increasing the value.

You can use these parameters like this:

Python

```
mssparkutils.fs.mount(  
    "abfss://mycontainer@<accountname>.dfs.core.windows.net",  
    "/test",  
    {"fileCacheTimeout": 120, "timeout": 120}  
)
```

① Note

For security reasons, we recommended you don't store credentials in code. To further protect your credentials, we will redact your secret in notebook output. For more information, see [Secret redaction](#).

How to mount a lakehouse

Sample code for mounting a lakehouse to `/test`:

Python

```
from notebookutils import mssparkutils  
mssparkutils.fs.mount(  
    "abfss://<workspace_id>@onelake.dfs.fabric.microsoft.com/<lakehouse_id>",  
    "/test"  
)
```

① Note

Mounting a regional endpoint is not supported. Fabric only supports mounting the global endpoint, `onelake.dfs.fabric.microsoft.com`.

Access files under the mount point by using the *mssparkutils fs* API

The main purpose of the mount operation is to let customers access the data stored in a remote storage account with a local file system API. You can also access the data by using the *mssparkutils fs* API with a mounted path as a parameter. This path format is a little different.

Assume that you mounted the Data Lake Storage Gen2 container *mycontainer* to `/test` by using the mount API. When you access the data with a local file system API, the path format is like this:

Python

```
/synfs/notebook/{sessionId}/test/{filename}
```

When you want to access the data by using the *mssparkutils fs* API, we recommend using `getMountPath()` to get the accurate path:

Python

```
path = mssparkutils.fs.getMountPath("/test")
```

- List directories:

Python

```
mssparkutils.fs.ls(f"file://{mssparkutils.fs.getMountPath('/test')}")
```

- Read file content:

Python

```
mssparkutils.fs.head("file://{mssparkutils.fs.getMountPath('/test')}/myFile.txt")
```

- Create a directory:

Python

```
mssparkutils.fs.mkdirs("file://{mssparkutils.fs.getMountPath('/test')}/newdir")
```

Access files under the mount point via local path

You can easily read and write the files in mount point using the standard file system. Here's a Python example:

Python

```
#File read
with open(mssparkutils.fs.getMountPath('/test2') + "/myFile.txt", "r") as f:
    print(f.read())
#File write
with open(mssparkutils.fs.getMountPath('/test2') + "/myFile.txt", "w") as f:
    print(f.write("dummy data"))
```

How to check existing mount points

You can use `mssparkutils.fs.mounts()` API to check all existing mount point info:

Python

```
mssparkutils.fs.mounts()
```

How to unmount the mount point

Use the following code to unmount your mount point (`/test` in this example):

Python

```
mssparkutils.fs.unmount("/test")
```

Known limitations

- The current mount is a job level configuration; we recommend you use the `mounts` API to check if a mount point exists or isn't available.
- The unmount mechanism isn't automatic. When the application run finishes, to unmount the mount point and release the disk space, you need to explicitly call an unmount API in your code. Otherwise, the mount point will still exist in the node after the application run finishes.
- Mounting an ADLS Gen1 storage account isn't supported.

Lakehouse utilities

`mssparkutils.lakehouse` provides utilities specifically tailored for managing Lakehouse artifacts. These utilities empower users to create, retrieve, and delete Lakehouse artifacts effortlessly.

Note

Lakehouse APIs are only supported on Runtime version 1.2+.

Overview of methods

Below is an overview of the available methods provided by `mssparkutils.lakehouse`:

Python

```
# Create a new Lakehouse artifact
create(name: String, description: String = "", workspaceId: String = ""): Artifact

# Retrieve a Lakehouse artifact
get(name: String, workspaceId: String = ""): Artifact

# Update an existing Lakehouse artifact
update(name: String, newName: String, description: String = "", workspaceId: String = ""): Artifact

# Delete a Lakehouse artifact
delete(name: String, workspaceId: String = ""): Boolean

# List all Lakehouse artifacts
list(workspaceId: String = ""): Array[Artifact]
```

Usage examples

To utilize these methods effectively, consider the following usage examples:

Creating a Lakehouse artifact

Python

```
artifact = mssparkutils.lakehouse.create("artifact_name", "Description of the artifact", "optional_workspace_id")
```

Retrieving a Lakehouse Artifact

Python

```
artifact = mssparkutils.lakehouse.get("artifact_name", "optional_workspace_id")
```

Updating a Lakehouse artifact

Python

```
updated_artifact = mssparkutils.lakehouse.update("old_name", "new_name", "Updated description", "optional_workspace_id")
```

Deleting a Lakehouse artifact

Python

```
is_deleted = mssparkutils.lakehouse.delete("artifact_name", "optional_workspace_id")
```

Listing Lakehouse artifacts

Python

```
artifacts_list = mssparkutils.lakehouse.list("optional_workspace_id")
```

Additional information

For more detailed information about each method and its parameters, utilize the `mssparkutils.lakehouse.help("methodName")` function.

With MSSparkUtils' Lakehouse utilities, managing your Lakehouse artifacts becomes more efficient and integrated into your Fabric pipelines, enhancing your overall data management experience.

Feel free to explore these utilities and incorporate them into your Fabric workflows for seamless Lakehouse artifact management.

Runtime utilities

Show the session context info

With `mssparkutils.runtime.context` you can get the context information of the current live session, including the notebook name, default lakehouse, workspace info, if it's a pipeline run, etc.

Python

```
mssparkutils.runtime.context
```

Known issue

When using runtime version above 1.2 and run `mssparkutils.help()`, the listed `fabricClient`, `warehouse`, and `workspace` APIs are not supported for now, will be available in the further.

Related content

- [Library management](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Manage Apache Spark libraries in Microsoft Fabric

Article • 11/08/2024

A library is a collection of prewritten code that developers can import to provide functionality. By using libraries, you can save time and effort by not having to write code from scratch to do common tasks. Instead, import the library and use its functions and classes to achieve the desired functionality. Microsoft Fabric provides multiple mechanisms to help you manage and use libraries.

- **Built-in libraries:** Each Fabric Spark runtime provides a rich set of popular preinstalled libraries. You can find the full built-in library list in [Fabric Spark Runtime](#).
- **Public libraries:** Public libraries are sourced from repositories such as PyPI and Conda, which are currently supported.
- **Custom libraries:** Custom libraries refer to code that you or your organization build. Fabric supports them in the `.whl`, `.jar`, and `.tar.gz` formats. Fabric supports `.tar.gz` only for the R language. For Python custom libraries, use the `.whl` format.

Summary of library management best practices

The following scenarios describe best practices when using libraries in Microsoft Fabric.

Scenario 1: Admin sets default libraries for the workspace

To set default libraries, you have to be the administrator of the workspace. As admin, you can perform these tasks:

1. [Create a new environment](#)
2. [Install the required libraries in the environment](#)
3. [Attach this environment as the workspace default](#)

When your notebooks and Spark job definitions are attached to the [Workspace settings](#), they start sessions with the libraries installed in the workspace's default environment.

Scenario 2: Persist library specifications for one or multiple code items

If you have common libraries for different code items and don't require frequent update, [install the libraries in an environment](#) and [attach it to the code items](#) is a good choice.

It will take some time to make the libraries in environments become effective when publishing. It normally takes 5-15 minutes, depending on the complexity of the libraries. During this process, the system will help to resolve the potential conflicts and download required dependencies.

One benefit of this approach is that the successfully installed libraries are guaranteed to be available when the Spark session is started with environment attached. It saves effort of maintaining common libraries for your projects.

It's highly recommended for pipeline scenarios with its stability.

Scenario 3: Inline installation in interactive run

If you are using the notebooks to write code interactively, using [inline installation](#) to add extra new PyPI/conda libraries or validate your custom libraries for one-time use is the best practice. Inline commands in Fabric allow you to have the library effective in the current notebook Spark session. It allows the quick installation but the installed library doesn't persist across different sessions.

Since `%pip install` generating different dependency trees from time to time, which might lead to library conflicts, inline commands are turned off by default in the pipeline runs and NOT recommended to be used in your pipelines.

Summary of supported library types

[] [Expand table](#)

Library type	Environment library management	Inline installation
Python Public (PyPI & Conda)	Supported	Supported
Python Custom (.whl)	Supported	Supported
R Public (CRAN)	Not supported	Supported
R custom (.tar.gz)	Supported as custom library	Supported
Jar	Supported as custom library	Supported

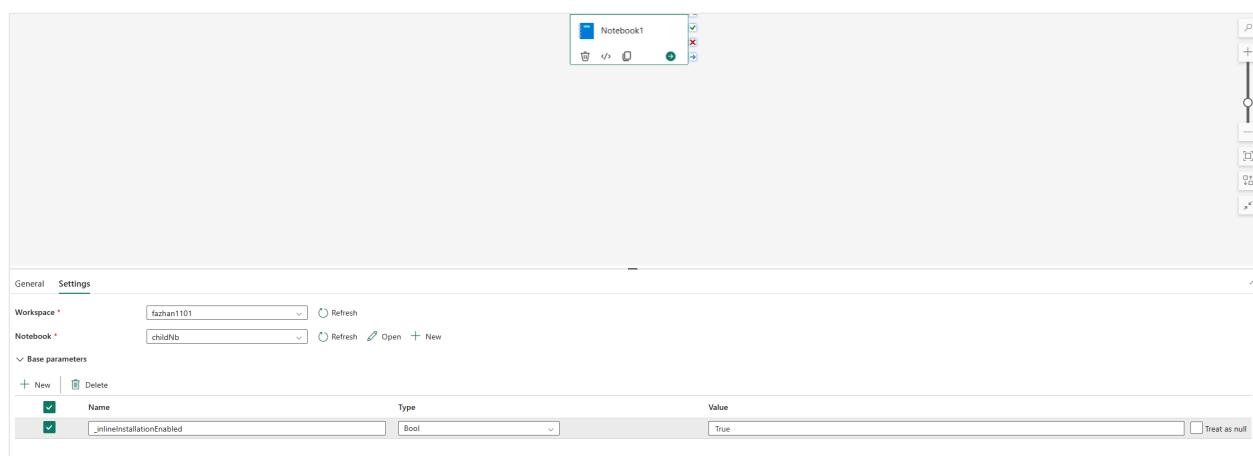
Inline installation

Inline commands support managing libraries in each notebook sessions.

Python inline installation

The system restarts the Python interpreter to apply the change of libraries. Any variables defined before you run the command cell are lost. We strongly recommend that you put all the commands for adding, deleting, or updating Python packages **at the beginning of your notebook**.

The inline commands for managing Python libraries are disabled in notebook pipeline run by default. If you want to enable `%pip install` for pipeline, add "`_inlineInstallationEnabled`" as bool parameter equals True in the notebook activity parameters.



ⓘ Note

The `%pip install` may lead to inconsistent results from time to time. It's recommended to install library in an environment and use it in the pipeline. In notebook reference runs, inline commands for managing Python libraries are not supported. To ensure the correctness of execution, it is recommended to remove these inline commands from the referenced notebook.

We recommend `%pip` instead of `!pip`. `!pip` is an IPython built-in shell command, which has the following limitations:

- `!pip` only installs a package on the driver node, not executor nodes.
- Packages that install through `!pip` don't affect conflicts with built-in packages or whether packages are already imported in a notebook.

However, `%pip` handles these scenarios. Libraries installed through `%pip` are available on both driver and executor nodes and are still effective even the library is already

imported.

Tip

The `%conda install` command usually takes longer than the `%pip install` command to install new Python libraries. It checks the full dependencies and resolves conflicts.

You might want to use `%conda install` for more reliability and stability. You can use `%pip install` if you are sure that the library you want to install doesn't conflict with the preinstalled libraries in the runtime environment.

For all available Python inline commands and clarifications, see [%pip commands ↗](#) and [%conda commands ↗](#).

Manage Python public libraries through inline installation

In this example, see how to use inline commands to manage libraries. Suppose you want to use *altair*, a powerful visualization library for Python, for a one-time data exploration. Suppose the library isn't installed in your workspace. The following example uses conda commands to illustrate the steps.

You can use inline commands to enable *altair* on your notebook session without affecting other sessions of the notebook or other items.

1. Run the following commands in a notebook code cell. The first command installs the *altair* library. Also, install *vega_datasets*, which contains a semantic model you can use to visualize.

Python

```
%conda install altair          # install latest version through conda  
command  
%conda install vega_datasets    # install latest version through conda  
command
```

The output of the cell indicates the result of the installation.

2. Import the package and semantic model by running the following code in another notebook cell.

Python

```
import altair as alt
from vega_datasets import data
```

3. Now you can play around with the session-scoped *altair* library.

Python

```
# load a simple dataset as a pandas DataFrame
cars = data.cars()
alt.Chart(cars).mark_point().encode(
    x='Horsepower',
    y='Miles_per_Gallon',
    color='Origin',
).interactive()
```

Manage Python custom libraries through inline installation

You can upload your Python custom libraries to the resources folder of your notebook or the attached environment. The resources folders are the built-in file system provided by each notebook and environments. See [Notebook resources](#) for more details. After your upload, you can drag-and-drop the custom library to a code cell, the inline command to install the library is automatically generated. Or you can use the following command to install.

Python

```
# install the .whl through pip command from the notebook built-in folder
%pip install "builtin/wheel_file_name.whl"
```

R inline installation

To manage R libraries, Fabric supports the `install.packages()`, `remove.packages()`, and `devtools::` commands. For all available R inline commands and clarifications, see [install.packages command](#) and [remove.package command](#).

Manage R public libraries through inline installation

Follow this example to walk through the steps of installing an R public library.

To install an R feed library:

1. Switch the working language to **SparkR (R)** in the notebook ribbon.

2. Install the *caesar* library by running the following command in a notebook cell.

```
Python
```

```
install.packages("caesar")
```

3. Now you can play around with the session-scoped *caesar* library with a Spark job.

```
Python
```

```
library(SparkR)
sparkR.session()

hello <- function(x) {
  library(caesar)
  caesar(x)
}
spark.lapply(c("hello world", "good morning", "good evening"), hello)
```

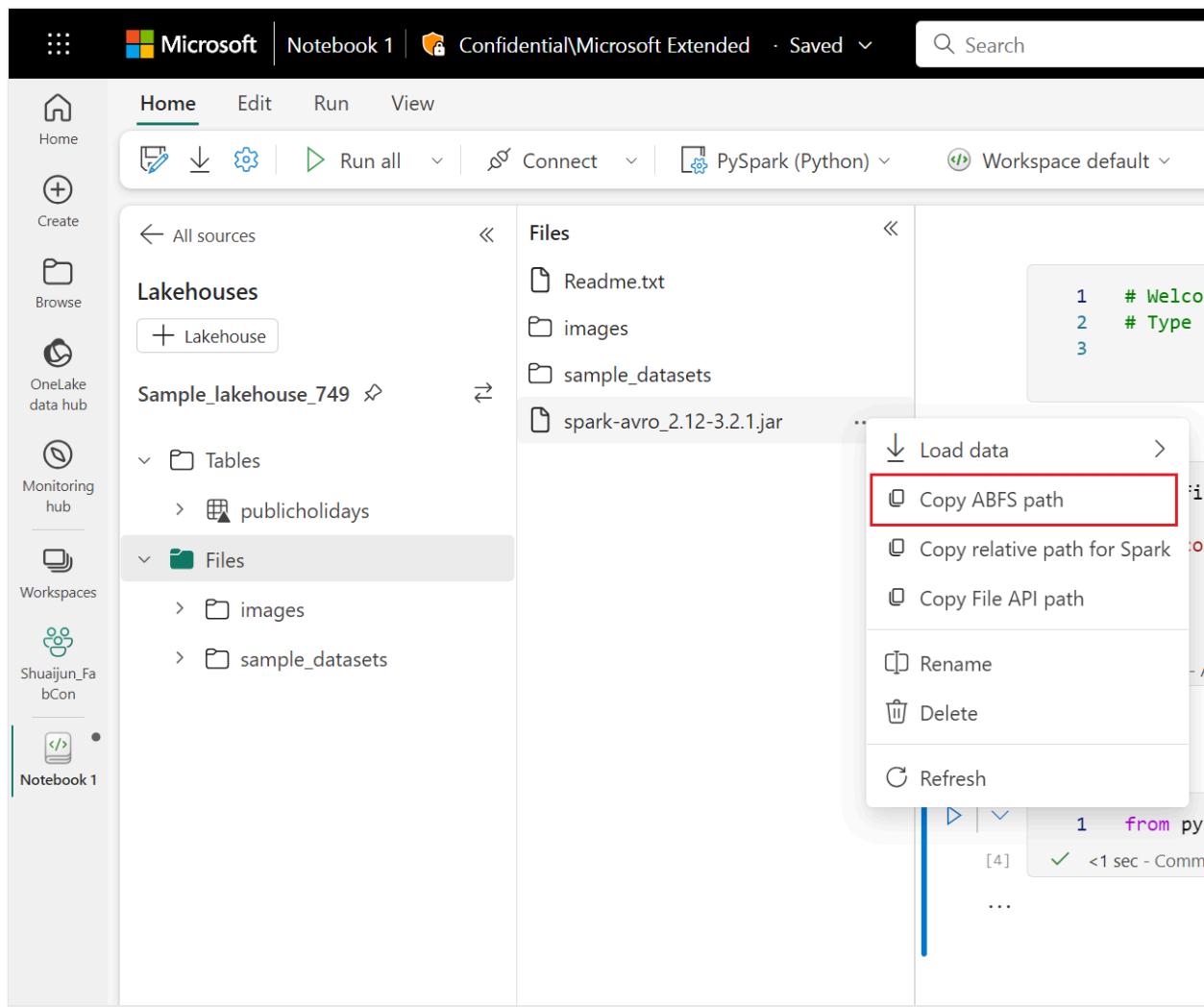
Manage Jar libraries through inline installation

The *.jar* files are support at notebook sessions with following command.

```
Scala
```

```
%%configure -f
{
  "conf": {
    "spark.jars": "abfss://<<Lakehouse
prefix>>.dfs.fabric.microsoft.com/<<path to JAR file>>/<<JAR file
name>>.jar",
  }
}
```

The code cell is using Lakehouse's storage as an example. At the notebook explorer, you can copy the full file ABFS path and replace in the code.



Related content

- [Create, configure, and use an environment in Microsoft Fabric](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback](#) | [Ask the community](#)

Capacity administration settings for Data Engineering and Data Science

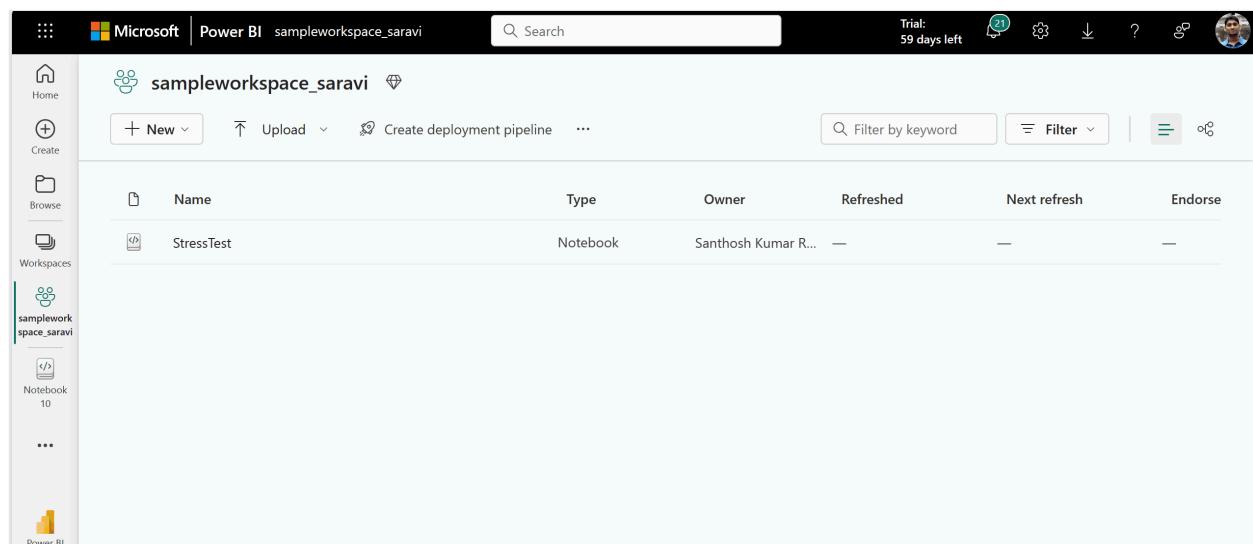
Article • 06/28/2024

Applies to: Data Engineering and Data Science in Microsoft Fabric

Admins purchase Microsoft Fabric capacities based on the compute and scale requirements of their enterprise's analytics needs. Admins are responsible to manage the capacity and governance. They must govern and manage the compute properties for data engineering and science analytics applications.

Microsoft Fabric capacity admins can now manage and govern their Data Engineering and Data Science settings from the admin settings portal. Admins can configure Spark environment for their users by enabling workspace level compute, choose a default runtime, and also create or manage spark properties for their capacities.

From the Admin portal, navigate to the **Data Engineering/Science Settings** section and select a specific capacity as shown in the following animation:



The screenshot shows the Microsoft Admin portal interface. The top navigation bar includes the Microsoft logo, Power BI icon, workspace name "sampleworkspace_saravi", a search bar, and various administrative icons like trial status (59 days left), notifications (21), and user profile. The left sidebar has navigation links for Home, Create, Browse, Workspaces, and a specific workspace named "sampleworkspace_saravi" which is currently selected. The main content area displays a table with columns: Name, Type, Owner, Refreshed, Next refresh, and Endorse. One row is visible: "StressTest" (Notebook, Santhosh Kumar R..., —, —, —). There are also buttons for "+ New", "Upload", "Create deployment pipeline", and "Filter".

Related content

- [Get Started with Data Engineering/Science Admin Settings for your Fabric Capacity](#)

Feedback

Was this page helpful?

[Provide product feedback](#) | [Ask the community](#)

Configure and manage data engineering and data science settings for Fabric capacities

Article • 06/26/2024

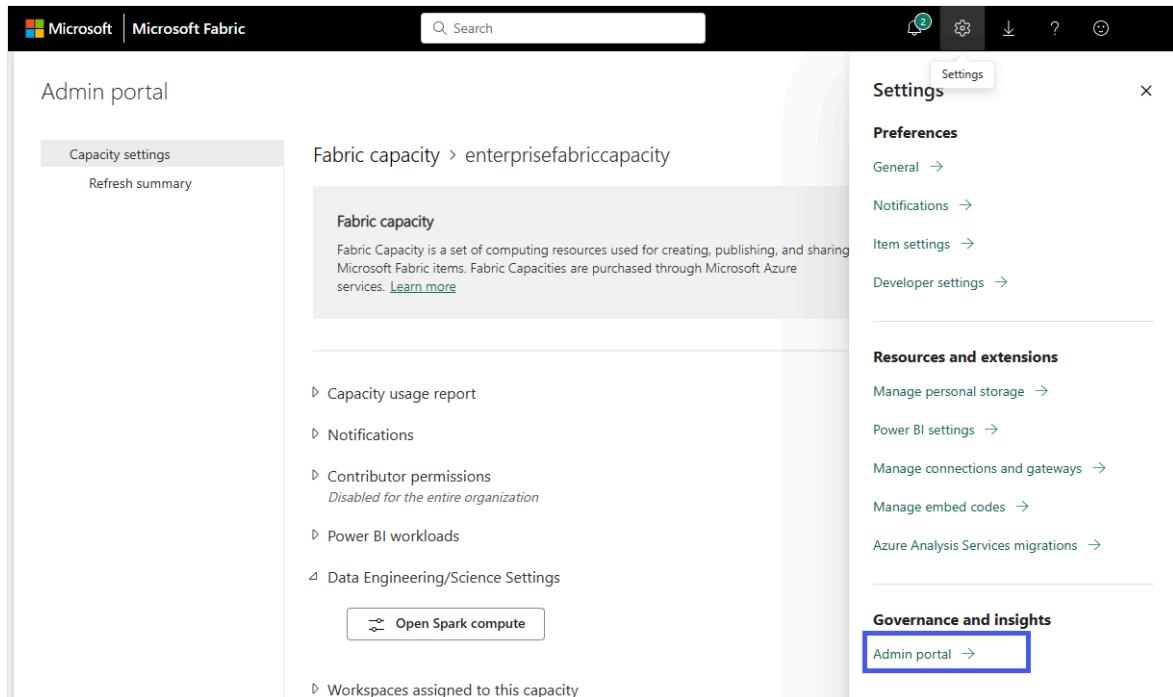
Applies to:  Data Engineering and Data Science in Microsoft Fabric

When you create Microsoft Fabric from the Azure portal, it is automatically added to the Fabric tenant that's associated with the subscription used to create the capacity. With the simplified setup in Microsoft Fabric, there's no need to link the capacity to the Fabric tenant. Because the newly created capacity will be listed in the admin settings pane. This configuration provides a faster experience for admins to start setting up the capacity for their enterprise analytics teams.

To make changes to the Data Engineering/Science settings in a capacity, you must have admin role for that capacity. To learn more about the roles that you can assign to users in a capacity, see [Roles in capacities](#).

Use the following steps to manage the Data Engineering/Science settings for Microsoft Fabric capacity:

1. Select the **Settings** option to open the setting pane for your Fabric account. Select **Admin portal** under Governance and insights section



2. Choose the **Capacity settings** option to expand the menu and select **Fabric capacity** tab. Here you should see the capacities that you have created in your tenant. Choose the capacity that you want to configure.

The screenshot shows the Microsoft Fabric Admin portal interface. On the left, there's a sidebar with various navigation options like Home, Create, Browse, Data Hub, Apps, Metrics, Monitor, Learn, Real-Time hub, Workspaces, and My workspace. The 'Capacity settings' option is selected and highlighted with a blue box. The main content area shows a breadcrumb path: Fabric capacity > enterprisefabriccapacity. Below this, there's a summary card for 'Fabric capacity' with a 'Learn more' link. The main pane is titled 'Details' and shows 'Delegated tenant settings'. Under this, there are several sections with arrows: Disaster Recovery, Capacity usage report, Notifications, Contributor permissions (disabled), Admin permissions, Power BI workloads, and Data Engineering/Science Settings. The 'Data Engineering/Science Settings' section is also highlighted with a blue box. At the bottom of the main pane, it says 'Workspaces assigned to this capacity'. A search bar is at the top right, and a magnifying glass icon is in the bottom right corner of the main content area.

3. You are navigated to the capacities detail pane, where you can view the usage and other admin controls for your capacity. Navigate to the **Data Engineering/Science Settings** section and select **Open Spark Compute**. Configure the following parameters:

! Note

Atleast one workspace should be attached to the Fabric Capacity to explore the Data Engineering/Science Settings from the Fabric Capacity Admin Portal.

- **Customized workspace pools:** You can restrict or democratize compute customization to workspace admins by enabling or disabling this option. Enabling this option allows workspace admins to create, update, or delete workspace level custom spark pools. Additionally, it allows you to resize them based on the compute requirements within the maximum cores limit of a capacity.

Capacity Pools for Data Engineering and Data Science in Microsoft Fabric (Public Preview)

In the **Pool List** section of Spark Settings, by clicking on the **Add** option, you can create a Custom pool for your Fabric Capacity.

The screenshot shows the Microsoft Fabric Admin portal. On the left, there's a sidebar with various icons for Home, Create, Browse, Data Lake, Data Hub, My workspaces, and more. The main area is titled 'Fabric capacity > enterprisefabriccapacity'. It includes sections for Disaster Recovery, Capacity usage report, Notifications, Contributor permissions (disabled), Admin permissions, and Power BI workloads. A blue box highlights the 'Data Engineering/Science Settings' section, which contains a 'Open Spark settings' button. To the right, there's a 'Spark settings' panel with a 'Customized workspace pools' section where the 'On' toggle is turned on, and buttons for Pool list, Add, and Delete.

5. You are navigated to the Pool creation section, where you specify the Pool name, Node family, select the Node size and set the Min and Max nodes for your custom pool, enable/disable autoscale, and dynamic allocation of executors.

The screenshot shows the 'Create new pool' dialog. At the top, there's a header with a back arrow, user info (Trial, 17 days left), notifications, settings, help, and profile. The main form starts with 'Spark pool name *' (capacitypool). Under 'Node family', it says 'Memory optimized'. The 'Node size' dropdown is open, showing 'Small' (selected), 'Medium', and 'Large'. A blue box highlights 'Small'. Below is a slider for 'Min' with values 1 and 6. To its right is a tooltip: 'd down based on the number of nodes in the cluster'. Below is a slider for 'Max' with values 1 and 5. Under 'Dynamically allocate executors', there's a checked checkbox for 'Enable allocate'. At the bottom are 'Create' and 'Cancel' buttons.

6. Click Create and Save the settings.



Spark settings

— Pool

✓ Spark settings updated
successfully!



Customized workspace pools

Permit workspace admins to size their own custom Spark pools based on workspace compute requirements.



On

▼ Pool list



Add



Delete



Pool name



capacitypool



ⓘ Note

The custom pools created in the capacity settings, will have a 2 to 3 minute session start latency as these are on-demand sessions unlike the sessions served through Starter Pools.

- Now the newly created Capacity pool is available as a Compute option in the Pool Selection menu in all the workspaces attached to this Fabric capacity.

Spark settings

Configure and manage settings for Spark workloads and the default environment for the workspace.

Pool Environment Jobs High concurrency Automatic log

Default pool for workspace

Use the automatically created starter pool or create custom pools for workspaces and items in the capacity. If the setting Customize compute configurations for items is turned off, this pool will be used for all environments in this workspace.

StarterPool

Starter pool

✓ StarterPool
Node family: Memory optimized; Node size: Medium

Capacity pools

samplecapacitypool
Node family: Memory optimized; Node size: Small

Workspace pools

samplepool
Node family: Memory optimized; Node size: Medium

New pool

Number of nodes
1 - 3

On

8. You can also view the created capacity pool as a compute option in the environment item within the workspaces.

Microsoft | env7 | Confidential/Microsoft Extended

Home

Libraries

Public libraries

Custom libraries

Spark compute

Compute

Spark properties

Storage

Resources

Spark compute configuration

This configuration applies to all notebooks and Spark job definitions in this environment. When you select a pool, the settings in that pool serve as limits for any environment-level pool details you customize. [learn more](#)

Environment pool

Default pool

Starter pool

✓ Default pool
Node family: Auto (Memory optimized); Node size: Medium

Capacity pools

samplecapacitypool
Node family: Auto (Memory optimized); Node size: Small

Workspace pools

samplepool
Node family: Auto (Memory optimized); Node size: Medium

8

Spark executor memory

56GB

Dynamically allocate executors

Enable dynamic allocation

Spark executor instances

1

9. This provides additional administrative controls to manage compute governance for your Spark compute in Microsoft Fabric. As a capacity admin, you can create Pools for workspaces and disable workspace-level customization, which would prevent workspace admins from creating custom pools.

Related content

- Get Started with Data Engineering/Science Admin Settings for your Fabric Workspace
 - Learn about the Spark Compute for Fabric Data Engineering/Science experiences
-

Feedback

Was this page helpful?



[Provide product feedback](#) | [Ask the community](#)

Data Engineering workspace administration settings in Microsoft Fabric

Article • 09/24/2024

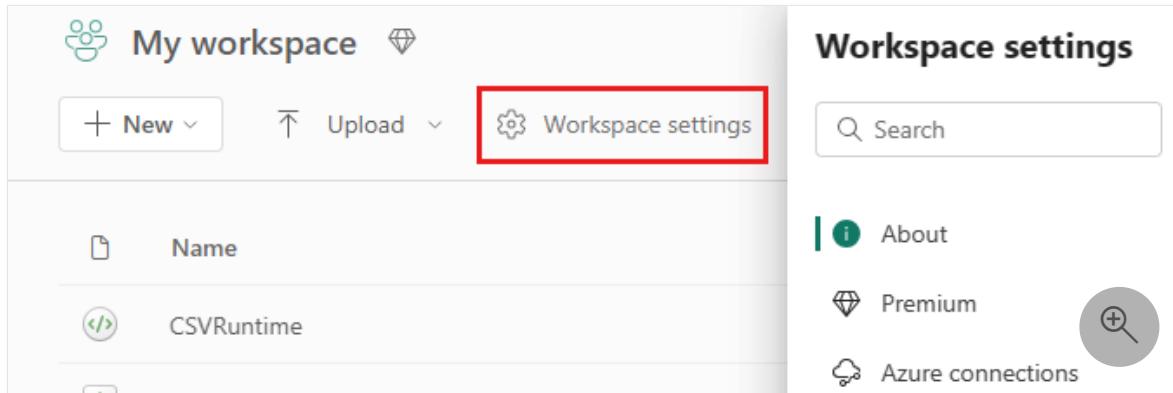
Applies to: Data Engineering and Data Science in Microsoft Fabric

When you create a workspace in Microsoft Fabric, a [starter pool](#) that is associated with that workspace is automatically created. With the simplified setup in Microsoft Fabric, there's no need to choose the node or machine sizes, as these options are handled for you behind the scenes. This configuration provides a faster (5-10 seconds) Apache Spark session start experience for users to get started and run your Apache Spark jobs in many common scenarios without having to worry about setting up the compute. For advanced scenarios with specific compute requirements, users can create a custom Apache Spark pool and size the nodes based on their performance needs.

To make changes to the Apache Spark settings in a workspace, you should have the admin role for that workspace. To learn more, see [Roles in workspaces](#).

To manage the Spark settings for the pool associated with your workspace:

1. Go to the **Workspace settings** in your workspace and choose the **Data Engineering/Science** option to expand the menu:



2. You see the **Spark Compute** option in your left-hand menu:

The screenshot shows the Power BI interface for the 'Enterprise_Fabric_Workspace'. The left sidebar includes options like Home, Create, Browse, Data hub, Apps, Metrics, Monitoring hub, Deployment pipelines, Team, and Workspaces. The main area displays a table with one item:

Name	Type	Owner	Refreshed	Next refresh	Endorsement	Sensitivity	Included in app
Notebook 1	Notebook	Santhosh Kumar R...	—	—	—	Confidential/Micro...	○

⚠ Note

If you change the default pool from Starter Pool to a Custom Spark pool you may see longer session start (~3 minutes).

Pool

Default pool for the workspace

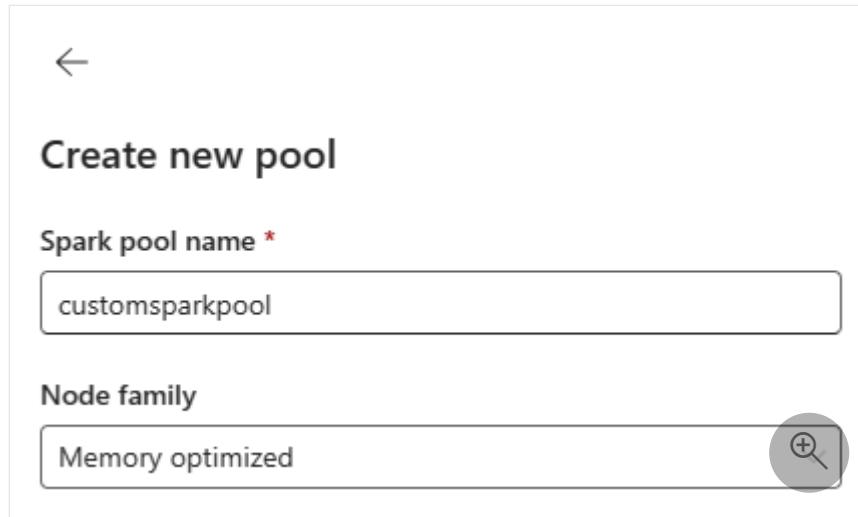
You can use the automatically created starter pool or create custom pools for the workspace.

- **Starter Pool:** Prehydrated live pools automatically created for your faster experience. These clusters are medium size. The starter pool is set to a default configuration based on the Fabric capacity SKU purchased. Admins can customize the max nodes and executors based on their Spark workload scale requirements.
To learn more, see [Configure Starter Pools](#)
- **Custom Spark Pool:** You can size the nodes, autoscale, and dynamically allocate executors based on your Spark job requirements. To create a custom Spark pool, the capacity admin should enable the **Customized workspace pools** option in the **Spark Compute** section of **Capacity Admin** settings.

⚠ Note

The capacity level control for Customized workspace pools is enabled by default. To learn more, see [Configure and manage data engineering and data science settings for Fabric capacities](#).

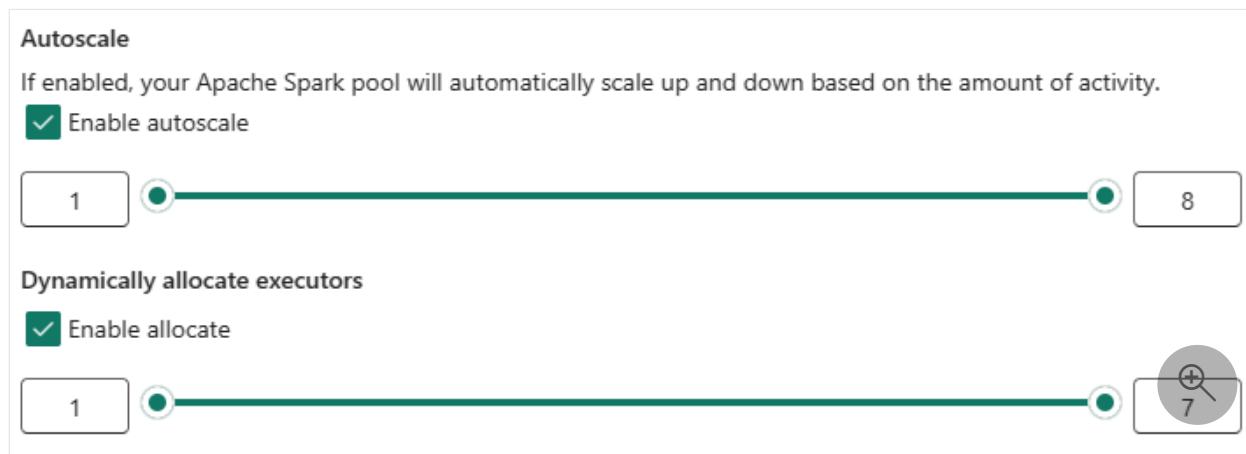
Admins can create custom Spark pools based on their compute requirements by selecting the **New Pool** option.



The screenshot shows the 'Create new pool' page. At the top left is a back arrow. The title 'Create new pool' is centered above a form area. The first field is 'Spark pool name *' with a red asterisk, containing the value 'customsparkpool'. Below it is a 'Node family' section with a dropdown menu showing 'Memory optimized' and a magnifying glass icon for searching other options.

Apache Spark for Microsoft Fabric supports single node clusters, which allows users to select a minimum node configuration of 1 in which case the driver and executor run in a single node. These single node clusters offer restorable high-availability during node failures and better job reliability for workloads with smaller compute requirements. You can also enable or disable autoscaling option for your custom Spark pools. When enabled with autoscale, the pool would acquire new nodes within the max node limit specified by the user and retire them after the job execution for better performance.

You can also select the option to dynamically allocate executors to pool automatically optimal number of executors within the max bound specified based on the data volume for better performance.



The screenshot shows two configuration sections. The first section is 'Autoscale' with the sub-instruction: 'If enabled, your Apache Spark pool will automatically scale up and down based on the amount of activity.' A checked checkbox labeled 'Enable autoscale' is followed by a horizontal slider with endpoints at 1 and 8. The second section is 'Dynamically allocate executors' with the sub-instruction: 'Enable allocate' (also checked). It has a similar slider with endpoints at 1 and 7, where the number 7 is highlighted with a gray circle and a plus sign.

Learn more about [Apache Spark compute for Fabric](#).

- **Customize compute configuration for items:** As a workspace admin, you can allow users to adjust compute configurations (session level properties which include Driver/Executor Core, Driver/Executor Memory) for individual items such as notebooks, Spark job definitions using Environment.

Customize compute configurations for items

 On

When turned on, users can adjust compute configuration for individual items such as notebooks and Spark job definitions.

[Learn more about Customize compute configurations for items](#) 

If the setting is turned off by the workspace admin, the Default pool and its compute configurations are used for all environments in the workspace.

Environment

Environment provides flexible configurations for running your Spark jobs (notebooks, Spark job definitions). In an Environment you can configure compute properties, select different runtime, setup library package dependencies based on your workload requirements.

In the environment tab, you have the option to set the default environment. You may choose which version of Spark you'd like to use for the workspace.

As a Fabric workspace admin, you can select an Environment as workspace default Environment.

You can also create a new one through the **Environment** dropdown.

Set default environment

The default environment will provide Spark properties settings for notebooks and Spark job definitions. Select a different environment.

[Learn more about Set default environment](#) 

Workspace default



If you disable the option to have a default environment, you have the option to select the Fabric runtime version from the available runtime versions listed in the dropdown selection.

Runtime Version

Runtime version defines which version of Spark your Spark pool will use.

[Learn more about Runtime Version](#) 

1.2 (Spark 3.4, Delta 2.4)

1.1 (Spark 3.3, Delta 2.2)

1.2 (Spark 3.4, Delta 2.4)



Learn more about [Apache Spark runtimes](#).

Jobs

Jobs settings allow admins to control the job admission logic for all the Spark jobs in the workspace.

Pool

Environment

Jobs

High concurrency

Automatic log

Reserve maximum cores for active Spark jobs

 On

When this setting is on, your Fabric capacity reserves the maximum number of cores needed for active Spark jobs, ensuring job reliability by making sure that cores are available if a job scales up. When this setting is off, jobs are started based on the minimum number of cores needed, letting more jobs run at the same time. [Learn more about reserving maximum cores](#)

- To reduce Spark session start times for individual notebooks, turn on high concurrency settings for notebooks and pipelines in the **High concurrency** tab.

Set Spark session timeout

Specify a time to terminate inactive Spark sessions. [Learn more about session expiry](#)

70



minutes



 Reset to default time



By default all workspaces are enabled with Optimistic Job Admission. Learn more about [Job admission for Spark in Microsoft Fabric](#).

You can enable the **Reserve maximum cores for active Spark jobs** to turn of Optimistic job admission based approach and reserve max cores for their Spark jobs.

You can also set the Spark session timeout to customize the session expiry for all the notebook interactive sessions.

 **Note**

The default session expiry is set to 20 minutes for the interactive Spark sessions.

High concurrency

High concurrency mode allows users to share the same Spark sessions in Apache Spark for Fabric data engineering and data science workloads. An item like a notebook uses a Spark session for its execution and when enabled allows users to share a single Spark session across multiple notebooks.

High concurrency

 On

When high concurrency is on, multiple notebooks can use the same Spark application to reduce the start time for each session.

[Learn more about High concurrency](#) 



Learn more about [High concurrency in Apache Spark for Fabric](#).

Automatic logging for Machine Learning models and experiments

Admins can now enable autologging for their machine learning models and experiments. This option automatically captures the values of input parameters, output metrics, and output items of a machine learning model as it is being trained. [Learn more about autologging](#) .

Automatically track machine learning experiments and models

 On

Automatically log metrics, parameters, and models without coding explicit statements in your notebook.

[Learn more about Automatically track machine learning experiments and models](#) 



Related content

- Read about [Apache Spark Runtimes in Fabric - Overview, Versioning, Multiple Runtimes Support and Upgrading Delta Lake Protocol](#).
- Learn more from the Apache Spark [public documentation](#) .

- Find answers to frequently asked questions: [Apache Spark workspace administration settings FAQ](#).
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Apache Spark workspace administration settings FAQ

FAQ

This article lists answers to frequently asked questions about Apache Spark workspace administration settings.

How do I use the RBAC roles to configure my Spark workspace settings?

Use the **Manage Access** menu to add **Admin** permissions for specific users, distribution groups, or security groups. You can also use this menu to make changes to the workspace and to grant access to add, modify, or delete the Spark workspace settings.

Are the changes made to the Spark properties at the environment level apply to the active notebook sessions or scheduled Spark jobs?

When you make a configuration change at the workspace level, it's not applied to active Spark sessions. This includes batch or notebook based sessions. You must start a new notebook or a batch session after saving the new configuration settings for the settings to take effect.

Can I configure the node family, Spark runtime, and Spark properties at a capacity level?

Yes, you can change the runtime, or manage the spark properties using the Data Engineering/Science settings as part of the capacity admin settings page. You need the capacity admin access to view and change these capacity settings.

Can I choose different node families for different notebooks and Spark job definitions in my workspace?

Currently, you can only select Memory Optimized based node family for the entire workspace.

Can I configure these settings at a notebook level?

Yes, you can use %%configure to customize properties at the Spark session level in Notebooks

Can I configure the minimum and maximum number of nodes for the selected node family?

Yes, you can choose the min and max nodes based on the allowed max burst limits of the Fabric capacity linked to the Fabric workspace.

Can I enable Autoscaling for the Spark Pools in a memory optimized or hardware accelerated GPU based node family?

Autoscaling is available for Spark pools and enabling that allows the system to automatically scale up the compute based on the job stages during runtime. GPUs are currently unavailable. This capability will be enabled in future releases.

Is Intelligent Caching for the Spark Pools supported or enabled by default

for a workspace?

Intelligent Caching is enabled by default for the Spark pools for all workspaces.

Feedback

Was this page helpful?



Yes



No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Apache Spark monitoring overview

Article • 09/15/2023

Microsoft Fabric Spark monitoring is designed to offer a web-UI based experience with built-in rich capabilities for monitoring the progress and status of Spark applications in progress, browsing past Spark activities, analyzing and optimizing performance, and facilitating troubleshooting of failures. Multiple entry points are available for browsing, monitoring, and viewing Spark application details.

ⓘ Important

Microsoft Fabric is in [preview](#).

Monitoring hub

The Monitoring Hub serves as a centralized portal for browsing Spark activities across items. At a glance, you can view in-progress Spark applications triggered from Notebooks, Spark Job Definitions, and Pipelines. You can also search and filter Spark applications based on different criteria and drill down to view more Spark execution details of a Spark application.

Item recent runs

When working on specific items, the item Recent Runs feature allows you to browse the item's current and recent activities and gain insights on the submitter, status, duration, and other information for activities submitted by you or others.

Notebook contextual monitoring

Notebook Contextual Monitoring offers the capability of authoring, monitoring, and debugging Spark jobs within a single place. You can monitor Spark job progress, view Spark execution tasks and executors, and access Spark logs within a Notebook at the Notebook cell level. The Spark advisor is also built into Notebook to offer real-time advice on code and cell Spark execution and perform error analysis.

Spark job definition inline monitoring

The Spark job definition Inline Monitoring feature allows you to view Spark job definition submission and run status in real-time, as well as view the Spark job definition's past runs and configurations. You can navigate to the Spark application detail page to view more details.

Pipeline Spark activity inline monitoring

For Pipeline Spark Activity Inline Monitoring, deep links have been built into the Notebook and Spark job definition activities within the Pipeline. You can view Spark application execution details, the respective Notebook and Spark job definition snapshot, and access Spark logs for troubleshooting. If the Spark activities fail, the inline error message is also available within Pipeline Spark activities.

Next steps

- [Apache Spark advisor for real-time advice on notebooks](#)
- [Browse the Apache Spark applications in the Fabric monitoring hub](#)
- [Browse item's recent runs](#)
- [Monitor Spark jobs within a notebook](#)
- [Monitor your Apache Spark job definition](#)
- [Apache Spark application detail monitoring](#)
- [Use extended Apache Spark history server to debug and diagnose Apache Spark applications](#)
- [Monitor Spark capacity consumption](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Ask the community ↗](#)

Apache Spark advisor for real-time advice on notebooks

Article • 03/19/2024

The Apache Spark advisor analyzes commands and code run by Apache Spark and displays real-time advice for Notebook runs. The Apache Spark advisor has built-in patterns to help users avoid common mistakes. It offers recommendations for code optimization, performs error analysis, and locates the root cause of failures.

Built-in advice

The Spark advisor, a tool integrated with Impulse, provides built-in patterns for detecting and resolving issues in Apache Spark applications. This article explains some of the patterns included in the tool.

You can open the **Recent runs** pane based on the type of advice you need.

May return inconsistent results when using 'randomSplit'

Inconsistent or inaccurate results may be returned when working with the *randomSplit* method. Use Apache Spark (RDD) caching before using the randomSplit() method.

Method randomSplit() is equivalent to performing sample() on your data frame multiple times. Where each sample refetches, partitions, and sorts your data frame within partitions. The data distribution across partitions and sorting order is important for both randomSplit() and sample(). If either changes upon data refetch, there may be duplicates or missing values across splits. And the same sample using the same seed may produce different results.

These inconsistencies may not happen on every run, but to eliminate them completely, cache your data frame, repartition on a column(s), or apply aggregate functions such as *groupBy*.

Table/view name is already in use

A view already exists with the same name as the created table, or a table already exists with the same name as the created view. When this name is used in queries or applications, only the view will be returned no matter which one created first. To avoid conflicts, rename either the table or the view.

Unable to recognize a hint

Scala

```
spark.sql("SELECT /*+ unknownHint */ * FROM t1")
```

Unable to find a specified relation name(s)

Unable to find the relation(s) specified in the hint. Verify that the relation(s) are spelled correctly and accessible within the scope of the hint.

Scala

```
spark.sql("SELECT /*+ BROADCAST(unknownTable) */ * FROM t1 INNER JOIN t2 ON  
t1.str = t2.str")
```

A hint in the query prevents another hint from being applied

The selected query contains a hint that prevents another hint from being applied.

Scala

```
spark.sql("SELECT /*+ BROADCAST(t1), MERGE(t1, t2) */ * FROM t1 INNER JOIN  
t2 ON t1.str = t2.str")
```

Enable 'spark.adviser.divisionExprConvertRule.enable' to reduce rounding error propagation

This query contains the expression with Double type. We recommend that you enable the configuration 'spark.adviser.divisionExprConvertRule.enable', which can help reduce the division expressions and to reduce the rounding error propagation.

Console

```
"t.a/t.b/t.c" convert into "t.a/(t.b * t.c)"
```

Enable 'spark.adviser.nonEqJoinConvertRule.enable' to improve query performance

This query contains time consuming join due to "Or" condition within query. We recommend that you enable the configuration 'spark.advisor.nonEqJoinConvertRule.enable', which can help to convert the join triggered by "Or" condition to SMJ or BHJ to accelerate this query.

User experience

The Apache Spark advisor displays the advice, including info, warnings, and errors, at Notebook cell output in real-time.

- Info

```

1 val rdd = sc.parallelize(1 to 1000000)
2 val rdd2 = rdd.repartition(64)
3 val Array(train, test) = rdd2.randomSplit(Array(70, 30), 1)
4 train.takeOrdered(10)

[4] ✓ 7 sec - Command executed in 5 sec 367 ms by Dakota Sanchez on 2:30:22 PM, 2/28/23
    ...
    > Spark jobs (1 of 1 succeeded)
    > Diagnostics 1
        > May return inconsistent results when using 'randomSplit'
...
    rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[44] at parallelize at <console>:31
    rdd2: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[48] at repartition at <console>:31
    train: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[49] at randomSplit at <console>:32
    test: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[50] at randomSplit at <console>:32
    res10: Array[Int] = Array(1, 3, 4, 5, 6, 7, 8, 10, 11, 13)

```

- Warning

```

1 %%spark
2 import org.apache.spark.SparkContext
3 import java.util.Random
4 def testDataSkew(sc: SparkContext): Unit = {
5     val numMappers = 400
6     val numKVPairs = 100000
7     val valSize = 256
8     val numReducers = 200
9     val biasPct = 0.4
10    val biasCount = numKVPairs * biasPct
11    for (i < 1 to 2) {
12        val query = sc.parallelize(0 until numMappers, numMappers).flatMap { p =>
13            val ranGen = new Random
14            val arr1 = new Array[(Int, Array[Byte])](numKVPairs)
15            for (i < 0 until numKVPairs) {
16                val byteArr = new Array[Byte](valSize)
17                ranGen.nextBytes(byteArr)
18                var key = ranGen.nextInt(Int.MaxValue)
19                if(i <= biasCount) {
20                    key = 1
21                }
22                arr1(i) = (key, byteArr)
23            }
24        }
25        arr1
26    }.groupByKey(numReducers)
27    println(query.count())
28 }
29 }
30 testDataSkew(sc)

```

✓ 3 min 52 sec - Dakota Sanchez is running the cell.

ID	Description	Status	Stages	Tasks	Duration	Rows	Data read	Data written
Job 7	count at <console>:53	✓ Succeeded	2/2	600/600 succeeded	56 sec	8000000	9.83 GB	9.83 GB
Job 8	count at <console>:53	✓ Succeeded	2/2	600/600 succeeded	2 min 38 sec	8000000	9.83 GB	9.83 GB

> Diagnostics 3

- ⚠ Data skew for job 7
- ⚠ Data skew for job 8
- ⚠ Time skew for job 8

```

23865871
23866411
import org.apache.spark.SparkContext
import java.util.Random
testDataSkew: (sc: org.apache.spark.SparkContext)Unit

```

- Error

```
1 # display(pandas.DataFrame) sample:
2 import numpy as np
3 import pandas as pd
4 from pyspark.sql import *
5 d = {'col1': [1, 2, 3, 4, 5, 6, 7, 8], 'col2': [3, 4, 5, 6, 8, 3, 16, 20]}
6 pdf = pd.DataFrame(data=d)
7
8 display(pdf)
9
10 1 sec - Dakota Sanchez is running the cell.
```

Spark (Scala) ▾

▼ Diagnostics ▾ 1

> ✖ Spark_User_AutoClassification_pandas_DataFrame

<console>;1: error: illegal start of definition
display(pandas.DataFrame) sample:

~

Spark Advisor Setting

The Spark advisor setting allows you to choose whether to show or hide specific types of Spark advice according to your needs. Additionally, you have the flexibility to enable or disable the Spark Advisor for your Notebooks within a workspace, based on your preferences.

You can access the Spark Advisor settings at the Fabric Notebook level to enjoy its benefits and ensure a productive notebook authoring experience.

Spark diagnostics settings

Spark diagnostics

Show errors, warnings, and performance suggestions for your Spark code in all notebooks in this workspace. [Learn more](#)

On

Manage diagnostic messages

Selecting **Hide this message** means you won't see this specific diagnostic message for any Spark runs in this workspace. Select the **Show** button in the table below if you want to display the message again.

Type	Message	Action
×	Job level error summary	Show
⚠	May include corrupted records within your file(s).	Show
⚠	May return inconsistent results when using 'randomSplit'.	Show
⚠	A hint in the query prevents another hint from being applied.	Show
💡	Cache data for better performance of query runs.	Show

Related content

- [Monitor Apache Spark jobs within notebooks](#)
- [Monitor Apache Spark job definition](#)
- [Monitor Apache Spark application details](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

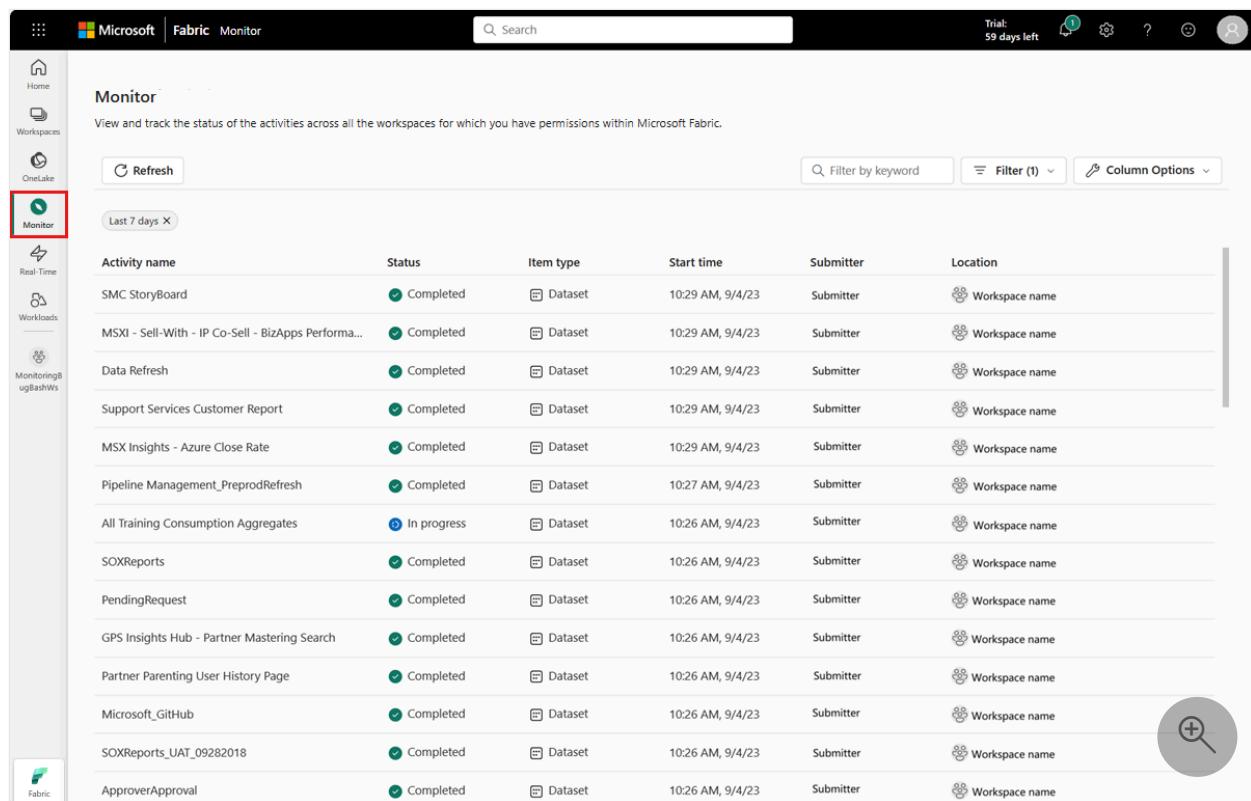
Browse the Apache Spark applications in Fabric Monitor

Article • 01/17/2025

The Monitor pane serves as a centralized portal for browsing Apache Spark activities across items. When you are in Data Engineering or Data Science, you can view in-progress Apache Spark applications triggered from notebooks, Apache Spark job definitions, and pipelines. You can also search and filter Apache Spark applications based on different criteria. Additionally, you can cancel your in-progress Apache Spark applications and drill down to view more execution details of an Apache Spark application.

Access the Monitor pane

You can access the Monitor pane to view various Apache Spark activities by selecting **Monitor** from the navigation bar.



The screenshot shows the Microsoft Fabric Monitor interface. On the left, there's a navigation sidebar with icons for Home, Workspaces, OneLake, Monitor (which is highlighted with a red box), Real Time, Workloads, and Monitoring. The main area is titled 'Monitor' and displays a table of Apache Spark activities. The table has columns for Activity name, Status, Item type, Start time, Submitter, and Location. Most entries show a green 'Completed' status and 'Dataset' type. A single entry for 'All Training Consumption Aggregates' is listed as 'In progress'. The table includes filters at the top right: 'Filter by keyword', 'Filter (1)', and 'Column Options'. A search bar is at the top center, and a large magnifying glass icon is in the bottom right corner of the main area.

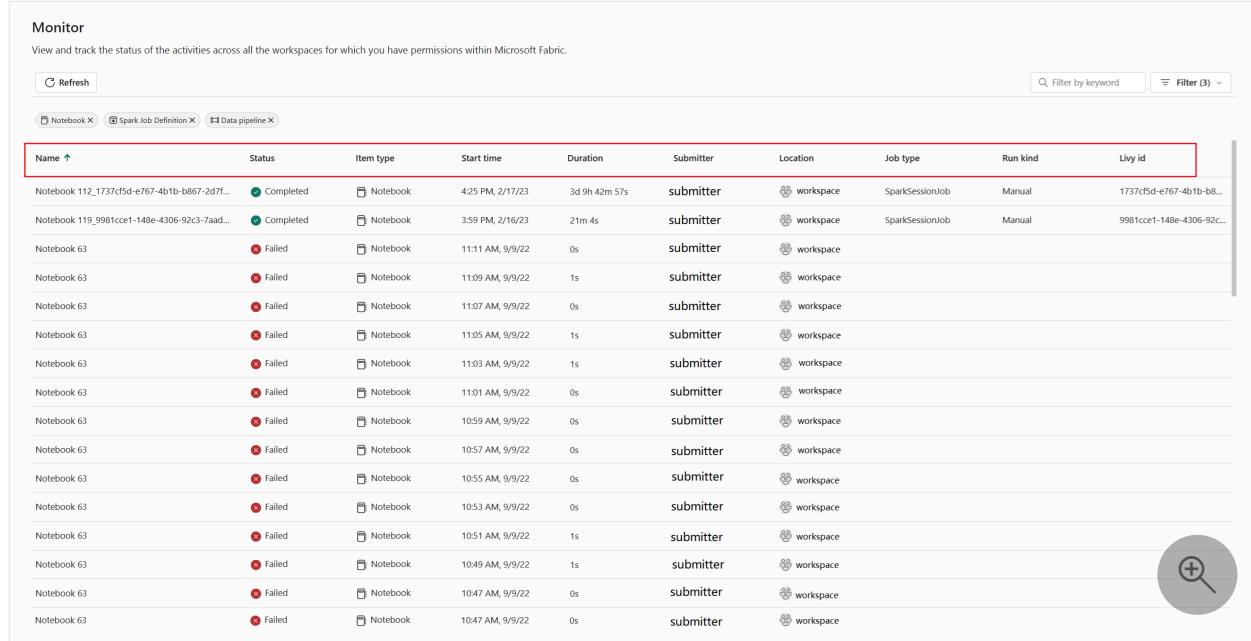
Activity name	Status	Item type	Start time	Submitter	Location
SMC StoryBoard	Completed	Dataset	10:29 AM, 9/4/23	Submitter	Workspace name
MSXI - Sell-With - IP Co-Sell - BizApps Performa...	Completed	Dataset	10:29 AM, 9/4/23	Submitter	Workspace name
Data Refresh	Completed	Dataset	10:29 AM, 9/4/23	Submitter	Workspace name
Support Services Customer Report	Completed	Dataset	10:29 AM, 9/4/23	Submitter	Workspace name
MSX Insights - Azure Close Rate	Completed	Dataset	10:29 AM, 9/4/23	Submitter	Workspace name
Pipeline Management_PrepodRefresh	Completed	Dataset	10:27 AM, 9/4/23	Submitter	Workspace name
All Training Consumption Aggregates	In progress	Dataset	10:26 AM, 9/4/23	Submitter	Workspace name
SOXRReports	Completed	Dataset	10:26 AM, 9/4/23	Submitter	Workspace name
PendingRequest	Completed	Dataset	10:26 AM, 9/4/23	Submitter	Workspace name
GPS Insights Hub - Partner Mastering Search	Completed	Dataset	10:26 AM, 9/4/23	Submitter	Workspace name
Partner Parenting User History Page	Completed	Dataset	10:26 AM, 9/4/23	Submitter	Workspace name
Microsoft_GitHub	Completed	Dataset	10:26 AM, 9/4/23	Submitter	Workspace name
SOXRReports_UAT_09282018	Completed	Dataset	10:26 AM, 9/4/23	Submitter	Workspace name
ApproverApproval	Completed	Dataset	10:26 AM, 9/4/23	Submitter	Workspace name

Sort, search, filter and column options Apache Spark applications

For better usability and discoverability, you can sort the Apache Spark applications by selecting different columns in the UI. You can also filter the applications based on different columns and search for specific applications. You can also adjust the display and sort order of the columns independently through the column options.

Sort Apache Spark applications

To sort Apache Spark applications, you can select on each column header, such as **Name, Status, Item Type, Start Time, Location**, and so on.

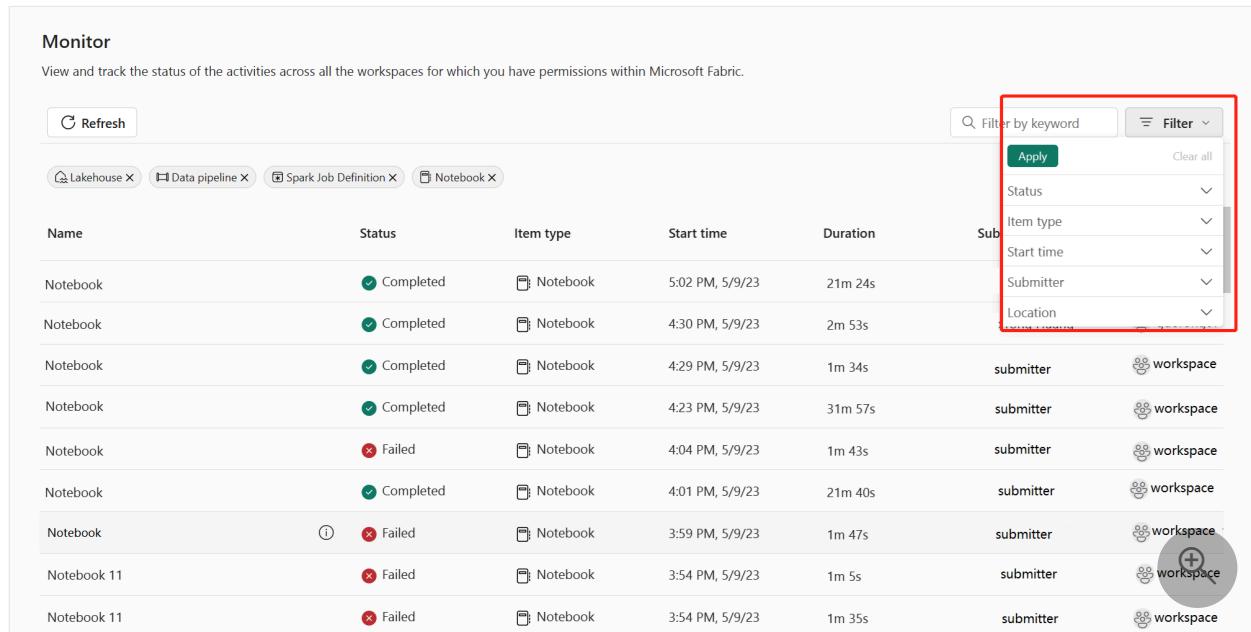


The screenshot shows the Microsoft Fabric Monitor interface. At the top, there's a navigation bar with tabs for Notebook, Data pipeline, Spark Job Definition, and Notebook. Below the navigation is a search bar labeled 'Filter by keyword' and a dropdown labeled 'Filter (3)'. The main area is a table with the following columns: Name, Status, Item type, Start time, Duration, Submitter, Location, Job type, Run kind, and Livy id. The 'Name' column header is highlighted with a red border. The table contains multiple rows of data, each representing an Apache Spark application. At the bottom right of the table is a circular icon with a magnifying glass and a plus sign.

Name	Status	Item type	Start time	Duration	Submitter	Location	Job type	Run kind	Livy id
Notebook 112_1737cf5d-e767-4b1b-b867-2d71...	Completed	Notebook	4:25 PM, 2/17/23	3d 9h 42m 57s	submitter	workspace	SparkSessionJob	Manual	1737cf5d-e767-4b1b-b8...
Notebook 119_9981cce1-148e-4306-92c3-7aad...	Completed	Notebook	3:59 PM, 2/16/23	21m 4s	submitter	workspace	SparkSessionJob	Manual	9981cce1-148e-4306-92c...
Notebook 63	Failed	Notebook	11:11 AM, 9/9/22	0s	submitter	workspace			
Notebook 63	Failed	Notebook	11:09 AM, 9/9/22	1s	submitter	workspace			
Notebook 63	Failed	Notebook	11:07 AM, 9/9/22	0s	submitter	workspace			
Notebook 63	Failed	Notebook	11:05 AM, 9/9/22	1s	submitter	workspace			
Notebook 63	Failed	Notebook	11:03 AM, 9/9/22	1s	submitter	workspace			
Notebook 63	Failed	Notebook	11:01 AM, 9/9/22	0s	submitter	workspace			
Notebook 63	Failed	Notebook	10:59 AM, 9/9/22	0s	submitter	workspace			
Notebook 63	Failed	Notebook	10:57 AM, 9/9/22	0s	submitter	workspace			
Notebook 63	Failed	Notebook	10:55 AM, 9/9/22	0s	submitter	workspace			
Notebook 63	Failed	Notebook	10:53 AM, 9/9/22	0s	submitter	workspace			
Notebook 63	Failed	Notebook	10:51 AM, 9/9/22	1s	submitter	workspace			
Notebook 63	Failed	Notebook	10:49 AM, 9/9/22	1s	submitter	workspace			
Notebook 63	Failed	Notebook	10:47 AM, 9/9/22	0s	submitter	workspace			
Notebook 63	Failed	Notebook	10:47 AM, 9/9/22	0s	submitter	workspace			

Filter Apache Spark applications

You can filter Apache Spark applications by **Status, Item Type, Start Time, Submitter, and Location** using the Filter pane in the upper-right corner.

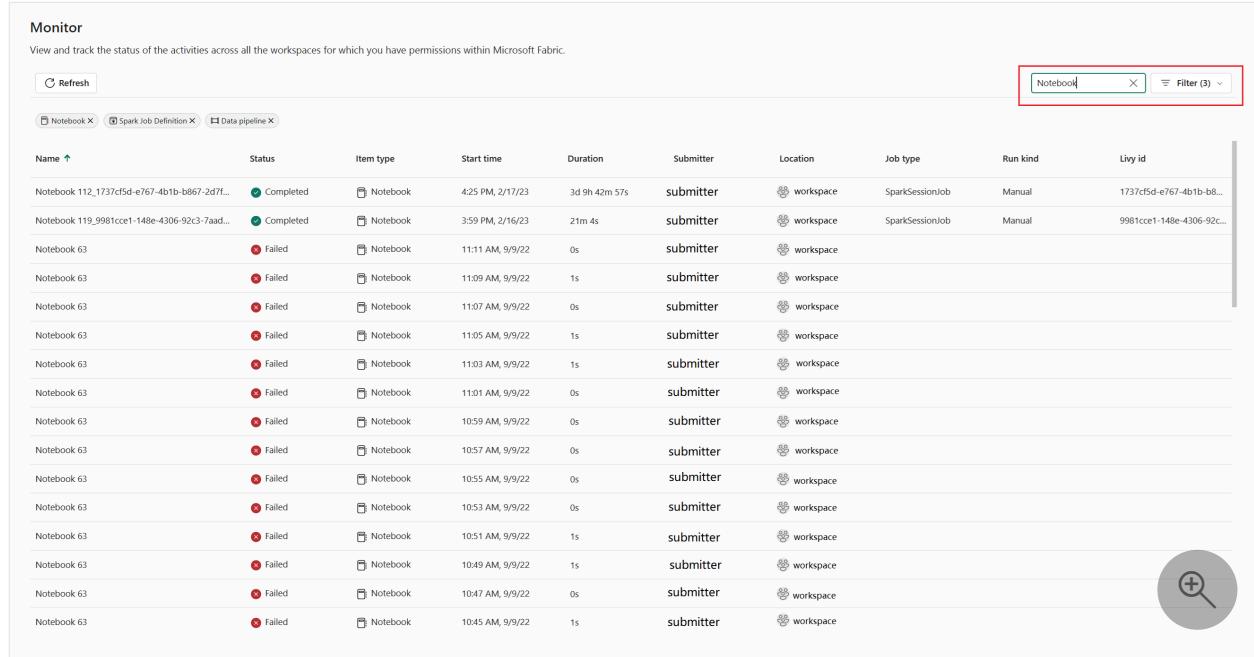


The screenshot shows the Microsoft Fabric Monitor interface with a filter pane open in the top right corner. The filter pane includes fields for Status, Item type, Start time, Submitter, and Location, each with a dropdown menu and a 'Clear all' link. The 'Status' field is highlighted with a red border. The main table below the filter pane lists several Apache Spark applications with columns for Name, Status, Item type, Start time, Duration, Submitter, and Location. Each row in the table represents an application with its details like status (Completed or Failed), item type (Notebook), start time, duration, submitter (workspace), and location (workspace).

Name	Status	Item type	Start time	Duration	Submitter	Location
Notebook	Completed	Notebook	5:02 PM, 5/9/23	21m 24s	submitter	workspace
Notebook	Completed	Notebook	4:30 PM, 5/9/23	2m 53s	submitter	workspace
Notebook	Completed	Notebook	4:29 PM, 5/9/23	1m 34s	submitter	workspace
Notebook	Completed	Notebook	4:23 PM, 5/9/23	31m 57s	submitter	workspace
Notebook	Failed	Notebook	4:04 PM, 5/9/23	1m 43s	submitter	workspace
Notebook	Completed	Notebook	4:01 PM, 5/9/23	21m 40s	submitter	workspace
Notebook	Failed	Notebook	3:59 PM, 5/9/23	1m 47s	submitter	workspace
Notebook 11	Failed	Notebook	3:54 PM, 5/9/23	1m 5s	submitter	workspace
Notebook 11	Failed	Notebook	3:54 PM, 5/9/23	1m 35s	submitter	workspace

Search Apache Spark applications

To search for specific Apache Spark applications, you can enter certain keywords in the search box located in the upper-right corner.

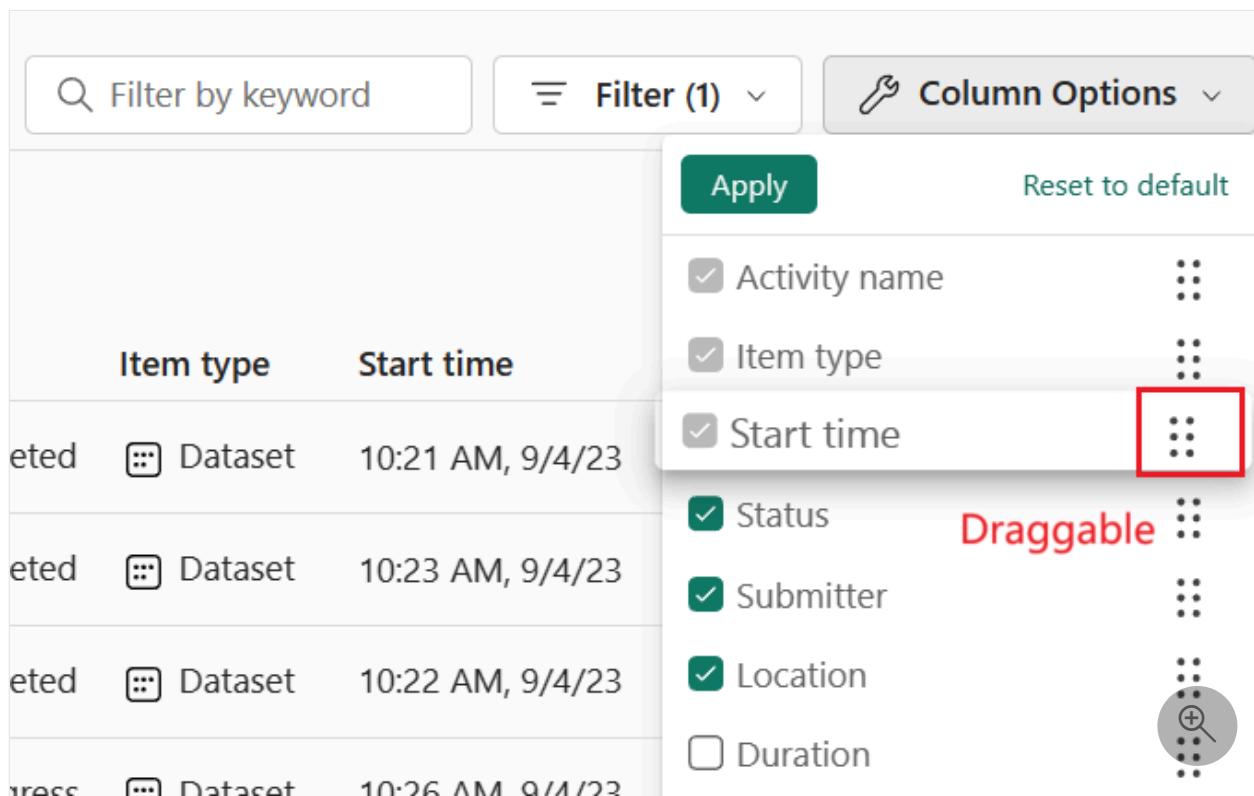


The screenshot shows the Microsoft Fabric Monitor interface. At the top, there's a search bar with the text "Notebook" and a filter button labeled "Filter (3)". Below the search bar is a table with columns: Name, Status, Item type, Start time, Duration, Submitter, Location, Job type, Run kind, and Livy id. The table lists multiple entries for "Notebook" items, some completed and some failed. A large circular button with a plus sign and a magnifying glass icon is visible on the right side of the table.

Name	Status	Item type	Start time	Duration	Submitter	Location	Job type	Run kind	Livy id
Notebook 112_1737cf5d-e767-4b1b-b867-2d7f...	Completed	Notebook	4:25 PM, 2/17/23	3d 9h 42m 57s	submitter	workspace	SparkSessionJob	Manual	1737cf5d-e767-4b1b-b8...
Notebook 119_9981cce1-148e-4306-92c3-7aad...	Completed	Notebook	3:59 PM, 2/16/23	21m 4s	submitter	workspace	SparkSessionJob	Manual	9981cce1-148e-4306-92c...
Notebook 63	Failed	Notebook	11:11 AM, 9/9/22	0s	submitter	workspace			
Notebook 63	Failed	Notebook	11:09 AM, 9/9/22	1s	submitter	workspace			
Notebook 63	Failed	Notebook	11:07 AM, 9/9/22	0s	submitter	workspace			
Notebook 63	Failed	Notebook	11:05 AM, 9/9/22	1s	submitter	workspace			
Notebook 63	Failed	Notebook	11:03 AM, 9/9/22	1s	submitter	workspace			
Notebook 63	Failed	Notebook	11:01 AM, 9/9/22	0s	submitter	workspace			
Notebook 63	Failed	Notebook	10:59 AM, 9/9/22	0s	submitter	workspace			
Notebook 63	Failed	Notebook	10:57 AM, 9/9/22	0s	submitter	workspace			
Notebook 63	Failed	Notebook	10:55 AM, 9/9/22	0s	submitter	workspace			
Notebook 63	Failed	Notebook	10:53 AM, 9/9/22	0s	submitter	workspace			
Notebook 63	Failed	Notebook	10:51 AM, 9/9/22	1s	submitter	workspace			
Notebook 63	Failed	Notebook	10:49 AM, 9/9/22	1s	submitter	workspace			
Notebook 63	Failed	Notebook	10:47 AM, 9/9/22	0s	submitter	workspace			
Notebook 63	Failed	Notebook	10:45 AM, 9/9/22	1s	submitter	workspace			

Column options Apache Spark applications

You can change the order in which the lists are displayed by selecting the list you want to display and then dragging the list options.



The screenshot shows the Microsoft Fabric Monitor interface with the "Column Options" dialog open. The dialog lists several columns: Activity name, Item type, Start time, Status, Submitter, Location, and Duration. The "Start time" column is currently selected and has a red box around its "Draggable" handle. The "Duration" column is unselected. The "Apply" and "Reset to default" buttons are at the top of the dialog.

Item type	Start time
eted	Dataset 10:21 AM, 9/4/23
eted	Dataset 10:23 AM, 9/4/23
eted	Dataset 10:22 AM, 9/4/23
ress	Dataset 10:26 AM, 9/4/23

Activity name

Item type

Start time

Status

Submitter

Location

Duration

Draggable

Reset to default

Enable upstream view for related Pipelines

If you have scheduled notebook and Spark job definitions to run in pipelines, you can view the Spark activities from these notebooks and Spark job definitions on the Monitor pane. Additionally, you can also see the corresponding parent pipeline and all its activities in the Monitor pane.

1. Select the **Upstream run** column option.

The screenshot shows a 'Column Options' dropdown menu with various filter options listed. The 'Upstream run' option is highlighted with a red box at the bottom of the list.

Column Option	Action
Activity name	<input checked="" type="checkbox"/>
Status	<input checked="" type="checkbox"/>
Item type	<input checked="" type="checkbox"/>
Start time	<input checked="" type="checkbox"/>
Submitter	<input checked="" type="checkbox"/>
Livy id	<input checked="" type="checkbox"/>
Run kind	<input checked="" type="checkbox"/>
Allocated Resource	<input checked="" type="checkbox"/>
Duration	<input checked="" type="checkbox"/>
Job type	<input checked="" type="checkbox"/>
Location	<input checked="" type="checkbox"/>
End time	<input type="checkbox"/>
Submitted by	<input type="checkbox"/>
Capacity	<input type="checkbox"/>
Average duration	<input type="checkbox"/>
Refresh type	<input type="checkbox"/>
Refreshes per day	<input type="checkbox"/>
Upstream run	<input checked="" type="checkbox"/>
Downstream runs	<input type="checkbox"/>

2. View the related parent pipeline run in the **Upstream run** column, and click the pipeline run to view all its activities.

Monitor

View and track the status of the activities across all the workspaces for which you have permissions within Microsoft Fabric.

Activity name	Status	Item type	Start time	Submitted by	Location	Upstream run
ResourceUsageDemo (2)_1321465a-eeff-4ff4-a0...	Succeeded	Notebook	2:58 PM, 2/29/24	submitter	workspace name	—
Book recommendation-181_e1e0de55-1a89-45b...	Succeeded	Notebook	2:54 PM, 2/29/24	submitter	workspace name	—
Notebook_b757be3f-cdaa-4cc8-8a7e-7da5cc22...	Succeeded	Notebook	2:46 PM, 2/29/24	submitter	workspace name	—
Notebook Run_7083e09d-50db-4091-ac90-c951...	Succeeded	Notebook	2:43 PM, 2/29/24	submitter	workspace name	—
Notebook Run_c665f182-d337-4b26-8ab7-063f...	Succeeded	Notebook	2:41 PM, 2/29/24	submitter	workspace name	pipeline1 ↗
pipeline1	Succeeded	Data pipeline	2:51 PM, 2/29/24	submitter	workspace name	—
HC session7_5ae41854-0642-48c1-963b-2e4807...	Succeeded	Notebook	2:48 PM, 2/29/24	submitter	workspace name	pipeline ↗
pipeline1024	Succeeded	Data pipeline	2:48 PM, 2/29/24	submitter	workspace name	—
pipeline1024	Failed	Data pipeline	2:45 PM, 2/29/24	submitter	workspace name	—
Book recommendation-181_931f731b-d5dc-4c0...	Succeeded	Notebook	2:34 PM, 2/29/24	submitter	workspace name	—
Book recommendation-11711111_34f6d34b-5c3...	Succeeded	Notebook	2:28 PM, 2/29/24	submitter	workspace name	—

Manage an Apache Spark application

When you hover over an Apache Spark application row, you can see various row-level actions that enable you to manage a particular Apache Spark application.

View Apache Spark application detail pane

You can hover over an Apache Spark application row and click the **View details** icon to open the **Detail** pane and view more details about an Apache Spark application.

Monitor

View and track the status of the activities across all the workspaces for which you have permissions within Microsoft Fabric.

Name	Status	Item type	Start time	Duration
Application name	Completed	Notebook	5:02 PM, 5/9/23	21m 24s
Application name	Completed	Notebook	4:30 PM, 5/9/23	2m 53s
Application name	Completed	Notebook	4:29 PM, 5/9/23	1m 34s
Application name	Completed	Notebook	4:23 PM, 5/9/23	31m 57s
Application name	Failed	Notebook	4:04 PM, 5/9/23	1m 43s
Application name	Completed	Notebook	4:01 PM, 5/9/23	21m 40s
Application name	Failed	Notebook	3:59 PM, 5/9/23	1m 47s
Application name	Failed	Notebook	3:54 PM, 5/9/23	1m 5s
Application name	Failed	Notebook	3:54 PM, 5/9/23	1m 35s

Details

Spark Application

Status: StoppedSessionTimedOut

Application name: Application name

Run kind: Manual

Livy Id: XXXXXXXXXXXXXXXXXXXX

Job Instance Id: XXXXXXXXXXXXXXXXXXXX

Submitted: 5/9/23 5:02:39 PM

Submitter: Submitter

Total duration: 21m 21s

Running Duration: -

Queued Duration: -

Cancel an Apache Spark application

If you need to cancel an in-progress Apache Spark application, hover over its row and click the **Cancel** icon.

Name	Status	Item type	Start time	Duration	Submitter	Location	Job type	Run kind	Livy id
Notebook1	In progress	Notebook	2:24 PM, 2/27/23		Submitter	workspace name	SparkSessionJob	Manual	xxxxxxxxxxxxxx
Notebook2	Failed	Notebook	1:31 AM, 2/25/23	2m 1s	Submitter	workspace name			xxxxxxxxxxxxxx
Notebook3	Failed	Notebook	12:31 AM, 2/25/23	2m 0s	Submitter	workspace name			xxxxxxxxxxxxxx

Navigate to Apache Spark application detail view

If you need more information about Apache Spark execution statistics, access Apache Spark logs, or check input and output data, you can click on the name of an Apache Spark application to navigate to its corresponding Apache Spark application detail page.

Related content

- [Apache Spark monitoring overview](#)
- [Browse item's recent runs](#)
- [Monitor Apache Spark jobs within notebooks](#)
- [Monitor Apache Spark job definition](#)
- [Monitor Apache Spark application details](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Browse item's recent runs

Article • 11/29/2023

With Microsoft Fabric, you can use Apache Spark to run notebooks, Apache Spark job definitions, jobs, and other types of applications in your workspace. This article explains how to view your running Apache Spark applications, making it easier to keep an eye on the latest running status.

View the recent runs pane

We can open **Recent runs** pane with the following steps:

1. Open the Microsoft Fabric homepage and select a workspace where you want to run the job.
2. Selecting **Spark job definition \ notebook item context \ pipeline item context** menu shows the recent run option.
3. Select **Recent runs**.
 - Open the recent run pane from the **Spark job definition \ notebook item context**.

The screenshot shows the Microsoft DXT Synapse Data Engineering workspace interface. On the left, there's a sidebar with icons for Home, Create, Browse, OneLake data hub, Monitoring hub, Workspaces, Workspace name (which is selected), and Notebookdemo. The main area displays a list of items under 'Workspace name', with 'Notebookdemo' selected. A context menu is open over 'Notebookdemo', listing options: Open, Delete, Settings, Add to Favorites, View lineage, View details, Schedule, Recent runs (which is highlighted with a red box and has a red arrow pointing down to it), Manage permissions, and Share.

Recent runs for Notebookdemo

Application name	Submitted	Submitter	Status	Total dur...	Run kind	Livy Id
Notebookdemo_6b220261-efb4-45af-bf95-3	9/11/23 4:46:20 P...	submitter	Running	2m 40s	Manual	

- Open the recent run pane from the pipeline item context.

The screenshot shows the Microsoft Fabric workspace interface. On the left, there's a sidebar with icons for Home, Create, Browse, Data hub, Monitoring hub, Workspaces, and Monitoring BugBashWs. The main area displays a list of pipelines: pipeline1, pipeline2, PipelineBugBash, PipelineTest, and PipelineTest2. For 'pipeline1', a context menu is open, with 'Recent runs' highlighted and surrounded by a red box. Below the menu, a section titled 'Recent runs for pipeline1' lists five completed runs, each with an activity name, start time, submitter, and status (Completed). A red box also surrounds this 'Recent runs' section. At the bottom right of the workspace area, there's a 'Go to Monitoring hub' button and a magnifying glass icon.

Activity name	Start time	Submitter	Status
pipeline1	7:12 AM, 4/28/23	Submitter	Completed
pipeline1	7:38 AM, 4/28/23	Submitter	Completed
pipeline1	7:41 AM, 4/28/23	Submitter	Completed
pipeline1	7:45 AM, 4/28/23	Submitter	Completed
pipeline1	7:51 AM, 4/28/23	Submitter	Completed

All runs within a Notebook

We can open **Recent runs** pane within a notebook by following steps:

1. Open the Microsoft Fabric homepage and select a workspace where you want to run the job.
2. Open a notebook in this workspace.
3. Selecting Run -> All runs

This screenshot shows the Databricks interface with the 'Run' tab selected in the top navigation bar. A red box highlights the 'Run' tab and the 'All runs' button in the top right of the main content area. The main content displays a notebook titled 'Chart and Graph Types with Python'. The notebook content includes a section about table views and a table view of recent runs.

Application name	Submitted	Submitter	Status	Total dur...	Run kind	Livy Id
Notebook	9/4/23 4:14:04 AM	Submitter	Stopped	26s	Scheduled	a4d26...
Notebook	9/3/23 4:14:05 AM	Submitter	Stopped	27s	Scheduled	a3ce3...
Notebook	9/2/23 4:14:04 AM	Submitter	Failed	8m 6s	Scheduled	6b841...
Notebook	9/1/23 4:15:05 AM	Submitter	Stopped	24s	Scheduled	e8416...
Notebook	8/31/23 4:14:02 AM	Submitter	Stopped	23s	Scheduled	cc191...
Notebook	8/31/23 4:01:25 AM	Submitter	Stopped	22m 35s	Manual	c31d9...

Detail for recent run pane

If the notebook or Spark job definition doesn't have any run operations, the **Recent runs** page shows **No jobs found**.

This screenshot shows the 'Recent runs' pane for 'Notebook 1'. It includes a 'Refresh' button and a message stating 'No jobs found.'

In the **Recent runs** pane, you can view a list of applications, including **Application name**, **Submitted time**, **Submitter**, **Status**, **Total duration**, **Run kind**, and **Livy Id**. You can filter applications by their status and submission time, which makes it easier for you to view applications.

This screenshot shows the 'Recent runs' pane for a workspace named 'MonitoringDemobs'. It displays a table of recent runs for an application named 'wordcountSJD'. The table includes columns for Application name, Submitted, Submitter, Status, Total duration, Run kind, and Livy Id. A red box highlights the 'Status' column. The pane also includes a 'Filter' dropdown and a 'Submit time' dropdown with options like 'All', 'Last 24 hours', 'Last 7 days', and 'Last 30 days'.

Application name	Submitted	Submitter	Status	Total dura...	Run kind	Livy Id
wordcountSJD_591199cf-1107-40b...	7/5/22 11:19:23 AM	Submitter	Success	5m 3s	Man	
wordcountSJD_cfc95a-d5ba-46bb...	7/5/22 10:16:58 AM	Submitter	Success	2m 2s	Man	
wordcountSJD_3f997ebe-3252-4e7f...	7/5/22 10:11:01 AM	Submitter	Failed	2m 32s	Man	
wordcountSJD_77eb2349-1ae8-47e...	7/5/22 9:58:51 AM	Submitter	Failed	7m 23s	Man	

Selecting the application name link navigates to spark application details where we can get to see the logs, data and skew details for the Spark run.

Related content

The next step after viewing the list of running Apache Spark applications is to view the application details. You can refer to:

- [Apache Spark application detail monitoring](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

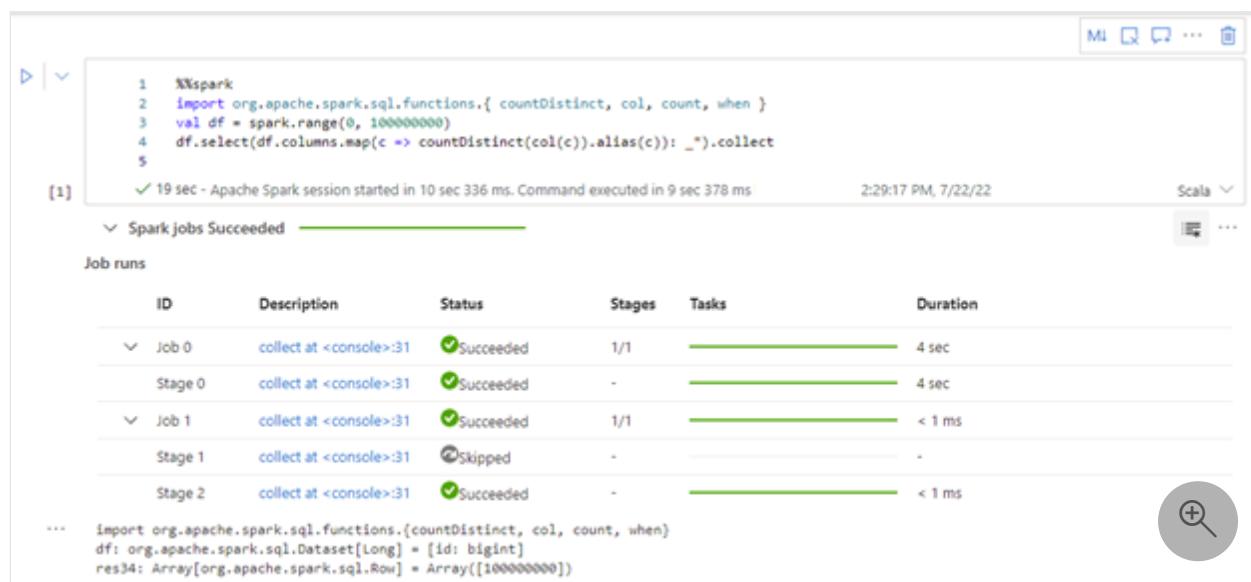
Monitor Spark jobs within a notebook

Article • 03/14/2024

The Microsoft Fabric notebook is a web-based interactive surface for developing Apache Spark jobs and conducting machine learning experiments. This article outlines how to monitor the progress of your Spark jobs, access Spark logs, receive advice within the notebook, and navigate to the Spark application detail view or Spark UI for more comprehensive monitoring information for the entire notebook.

Monitor Spark Job progress

A Spark job progress indicator is provided with a real-time progress bar that helps you monitor the job execution status for each notebook cell. You can view the status and tasks' progress across your Spark jobs and stages.



The screenshot shows a Microsoft Fabric notebook interface. At the top, there's a code cell containing Scala code to calculate the count of distinct values in a DataFrame. Below the code, a message indicates the session started and the command was executed. To the right, the current time is shown. A dropdown menu for "Scala" is open. The main area displays a "Spark jobs Succeeded" section with a table of completed tasks. The table has columns for ID, Description, Status, Stages, Tasks, and Duration. It lists two jobs: Job 0 and Job 1, each with one stage and one task that succeeded. Stage 1 of Job 1 was skipped. The duration for each task is indicated by a green progress bar. At the bottom, there's some additional Scala code showing the creation of a Dataset and a resulting array of Row objects.

ID	Description	Status	Stages	Tasks	Duration
Job 0	collect at <console>:31	SUCCEEDED	1/1	[Progress Bar]	4 sec
Stage 0	collect at <console>:31	SUCCEEDED	-	[Progress Bar]	4 sec
Job 1	collect at <console>:31	SUCCEEDED	1/1	[Progress Bar]	< 1 ms
Stage 1	collect at <console>:31	SKIPPED	-	-	-
Stage 2	collect at <console>:31	SUCCEEDED	-	[Progress Bar]	< 1 ms

Monitor Resource usage

The executor usage graph visually displays the allocation of Spark job executors and resource usage. Currently, only the runtime information of spark 3.4 and above will display this feature. Click on **Resources** tab, the line chart for the resource usage of code cell will be showing.

The screenshot shows a Jupyter Notebook cell with the following code:

```

1 %%spark
2
3 import org.apache.spark.sql.functions.{ countDistinct, col, count, when }
4
5 val df = spark.range(0, 100000000)
6
7 df.select(df.columns.map(c => countDistinct(col(c)).alias(c)): _*).collect

```

[1] ✓ 20 sec Apache Spark session ready in 10 sec 996 ms. Command executed in 9 sec 633 ms by ...

Spark resources: 1-3 executors 8-24 cores Jobs: Spark jobs (2 of 2 succeeded)

Legend: Running (blue arrow), Idle (grey arrow), Allocated (orange dashed line), Max instance (purple line).

The chart shows cores allocated over time from 4:49:03 PM to 4:49:12 PM. It starts at 25 cores, remains constant until 4:49:04 PM, then drops to 8 cores and stays constant until 4:49:10 PM, where it drops sharply to 1 core and then back to 0.

...
import org.apache.spark.sql.functions.{countDistinct, col, count, when}
df: org.apache.spark.sql.Dataset[Long] = [id: bigint]
res8: Array[org.apache.spark.sql.Row] = Array([100000000])

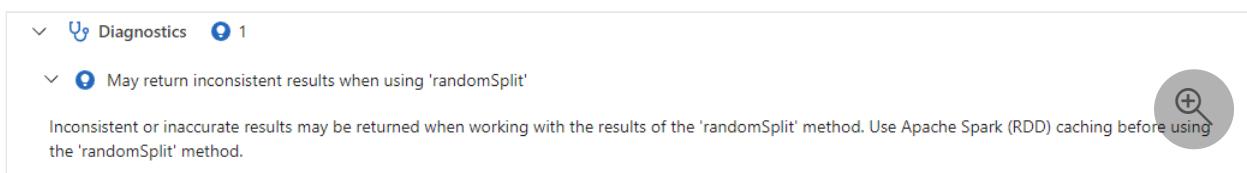
View Spark Advisor recommendations

A built-in Spark advisor analyzes your notebook code and Spark executions in real-time to help optimize the running performance of your notebook and assist in debugging failures. There are three types of built-in advice: Info, Warning, and Error. The icons with numbers indicate the respective count of advice in each category (Info, Warning, and Error) generated by the Spark advisor for a particular notebook cell.

To view the advice, click the arrow at the beginning to expand and reveal the details.



After expanding the advisor section, one or more pieces of advice become visible.



Spark Advisor Skew Detection

Data skew is a common issue users often encounter. The Spark advisor supports skew detection, and if skew is detected, a corresponding analysis is displayed below.

The screenshot shows a Jupyter Notebook dashboard with two main sections. Each section has a warning icon (triangle), a title ('Data skew for job 0' or 'Data skew for job 1'), a 'Data Skew Analysis' heading, and a table with columns: Name, Max Task Dat..., Mean Task Da..., and Task Data Read Skewness. Below the table is a search icon. The top right corner of the dashboard area is highlighted with a red box.

Name	Max Task Dat...	Mean Task Da...	Task Data Read Skewness
Stage 1	4003.12 MB	50.31 MB	0.21

Name	Max Task Dat...	Mean Task Da...	Task Data Read Skewness
Stage 3	4003.22 MB	50.31 MB	0.21

Access Spark Real-time logs

Spark logs are essential for locating exceptions and diagnosing performance or failures. The contextual monitoring feature in the notebook brings the logs directly to you for the specific cell you are running. You can search the logs or filter them by errors and warnings.

The screenshot shows a Jupyter Notebook cell with the title 'testDataSkew(Sc)'. The cell content includes a success message, a collapsible section for 'Spark jobs (2 of 2 succeeded)', and a 'Log' button. A red box highlights the 'Log' button and the search bar below it. Another red box highlights the log text area, which displays green-colored log entries from a Spark session. The log entries include various INFO messages from TaskSetManager and DAGScheduler, indicating task completion and stage finishing.

```

2023-04-06 08:48:54,889 INFO TaskSetManager [task-result-getter-1]: Finished task 192.0 in stage 13.0 (TID 1297) in 0.42 ms on vm-3cc46093 (executor)
2023-04-06 08:48:54,892 INFO TaskSetManager [task-result-getter-2]: Finished task 193.0 in stage 13.0 (TID 1298) in 646 ms on vm-3cc46093 (executor)
2023-04-06 08:48:54,925 INFO TaskSetManager [task-result-getter-0]: Finished task 194.0 in stage 13.0 (TID 1299) in 229 ms on vm-3cc46093 (executor)
2023-04-06 08:48:54,988 INFO TaskSetManager [task-result-getter-3]: Finished task 195.0 in stage 13.0 (TID 1300) in 173 ms on vm-de560261 (executor)
2023-04-06 08:48:55,015 INFO TaskSetManager [task-result-getter-1]: Finished task 198.0 in stage 13.0 (TID 1303) in 152 ms on vm-3cc46093 (executor)
2023-04-06 08:48:55,018 INFO TaskSetManager [task-result-getter-2]: Finished task 199.0 in stage 13.0 (TID 1304) in 134 ms on vm-3cc46093 (executor)
2023-04-06 08:48:55,028 INFO TaskSetManager [task-result-getter-0]: Finished task 196.0 in stage 13.0 (TID 1301) in 195 ms on vm-de560261 (executor)
2023-04-06 08:48:55,035 INFO TaskSetManager [task-result-getter-3]: Finished task 197.0 in stage 13.0 (TID 1302) in 194 ms on vm-de560261 (executor)
2023-04-06 08:49:02,830 INFO TaskSetManager [task-result-getter-1]: Finished task 1.0 in stage 13.0 (TID 1106) in 19350 ms on vm-de560261 (executor)
2023-04-06 08:49:02,830 INFO YarnClusterScheduler [task-result-getter-1]: Removed TaskSet 13.0, whose tasks have all completed, from pool
2023-04-06 08:49:02,830 INFO DAGScheduler [dag-scheduler-event-loop]: ResultStage 13 (count at <console>:53) finished in 19.358 s
2023-04-06 08:49:02,830 INFO DAGScheduler [dag-scheduler-event-loop]: Job 8 is finished. Cancelling potential speculative or zombie tasks for t
2023-04-06 08:49:02,830 INFO YarnClusterScheduler [dag-scheduler-event-loop]: Killing all running tasks in stage 13: Stage finished
2023-04-06 08:49:02,831 INFO DAGScheduler [Thread-38]: Job 8 finished: count at <console>:53, took 44.148951 s
23866440
23866306
2023-04-06 08:49:02,835 INFO KustoHandler [spark-listener-group-shared]: Logging DataSkew with appId: application_1680770743740_0001 to Kusto:
2023-04-06 08:49:02,837 INFO CreateAdviseEventHandler [spark-listener-group-shared]: Sending DataSkew to Advise Hub: Map(_source -> user, _jobId -> job)

```

Navigate to Spark monitoring detail and Spark UI

If you want to access additional information about the Spark execution at the notebook level, you can navigate to the Spark application details page or Spark UI through the options available in the context menu.

The screenshot shows the Apache Spark History Server interface. At the top, there's a navigation bar with icons for back, forward, and search. Below it, a section titled "Spark jobs Succeeded" is expanded, showing a table of job runs. The columns are ID, Description, Status, Stages, Tasks, and Duration. The first job, Job 0, has three stages: Stage 0, Stage 1, and Stage 2, all of which succeeded. Stage 0 and Stage 2 each have one task, while Stage 1 has none. The total duration for Stage 0 is 4 sec, Stage 1 is less than 1 ms, and Stage 2 is also less than 1 ms. The second job, Job 1, also has three stages: Stage 0, Stage 1, and Stage 2, all of which succeeded. Stage 0 and Stage 2 each have one task, while Stage 1 has none. The total duration for Stage 0 is 4 sec, Stage 1 is less than 1 ms, and Stage 2 is also less than 1 ms. Below the table, there is some Java code:

```
import org.apache.spark.sql.functions.{countDistinct, col, count, when}
df: org.apache.spark.sql.Dataset[Long] = [id: bigint]
res34: Array[org.apache.spark.sql.Row] = Array([100000000])
```

On the right side of the table, there are two buttons: "Spark application details" and "Spark web UI", both of which are highlighted with red boxes. A magnifying glass icon is located in the bottom right corner of the table area.

Related content

- Learn about [Spark advisor](#)
- [Apache Spark application detail monitoring](#)
- [Use the extended Spark history server to debug apps](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Monitor your Apache Spark job definition

Article • 01/17/2025

Using the Spark job definition item's inline monitoring, you can track the following:

- Monitor the progress and status of a running Spark job definition.
- View the status and duration of previous Spark job definition runs.

You can get this information from the **Recent Runs** contextual menu in the workspace or by browsing the Spark job definition activities in the Monitor pane.

Spark job definition inline monitoring

The Spark job definition inline monitoring feature allows you to view Spark job definition submission and run status in real-time. You can also view the Spark job definition's past runs and configurations and navigate to the **Spark application detail** page to view more details.

Application name	Submitted	Submitter	Status	Total duration	Run kind	Livy Id
GuorongSJD1_35292d9c-cd8c-4568-a	5/5/23 12:13:56 PM	Submitter	✗ Failed	1m 59s	Manual	732064c8-c389-4be0-b55c-1f...
GuorongSJD1_35292d9c-cd8c-4568-a	5/5/23 12:13:54 PM	Submitter	✗ Failed	1m 5s	Manual	35292d9c-cd8c-4568-a104-672...
GuorongSJD1_6ecb8bed-2d74-4134-9	5/5/23 12:13:40 PM	Submitter	✓ Success	1m 6s	Manual	1c2e11b4-2e6c-4a66-b2eb-b7c...
GuorongSJD1_d8295fc-04a7-462e-a5	5/5/23 12:08:07 PM	Submitter	✗ Cancelled	59s	Manual	58531dff-e4ad-42dc-a125-c306...
GuorongSJD1_973c2f43-cf47-4fda-99c	5/5/23 12:04:44 PM	Submitter	✗ Cancelled	54s	Manual	ec15caa0-dff6-4481-86f0-e043...
GuorongSJD1_960f0ffd-5142-4a0d-96	5/4/23 4:52:31 PM	Submitter	✓ Success	1m 6s	Manual	613b9735-c7cb-48ef-81a3-3195...
GuorongSJD1_fdc224e1-aa54-42d3-ac	5/4/23 4:43:04 PM	Submitter	✗ Failed	5m 7s	Manual	30955b3e-f9c6-4fbf-87a7-a543...
GuorongSJD1_f844d4d4-a903-4bdf-84	5/4/23 4:39:59 PM	Submitter	✓ Success	1m 6s	Manual	b2906ac3-7bd3-46ed-8d6c-0a2...

Spark job definition item view in workspace

You can access the job runs associated with specific Spark job definition items by using the **Recent runs** contextual menu on the workspace homepage.

Name	Type	Owner	Refreshed	Next refresh	Endorsement	Sensitivity
Artifact_20221013055638398	Test Artifact	Owner	—	—	—	Non-Business ⓘ
SDJtest	Spark Job Definition	Owner	—	—	—	General ⓘ
canceltest						
Notebook 2						
Notebook 3						
Notebook 4						
testnotebook						
asdfadfeawef	HomeOne	Owner	—	—	—	Public ⓘ

Spark job definition runs in the Monitor pane

To view all the Spark applications related to a Spark job definition, go to the **Monitor pane**. Sort or filter the **Item Type** column to view all the run activities associated with the Spark job definitions.

Name	Status	Item type	Start time	Duration	Submitter	Location
SDJtest_c7b7b4d9-e730-46b2-ae58-6d907d3bb...	Not started	Spark Job Definition	11:47 AM, 2/28/23		Submitter	workspace
SDJtest	Failed	Spark Job Definition	11:47 AM, 2/28/23	0s	Submitter	workspace
SDJtest_c612fe4f-7e42-426a-8627-92da6...	Failed	Spark Job Definition	11:11 AM, 2/28/23	14m 57s	Submitter	workspace
SDJtest_7558cbae-1179-4bd5-8edf-5df683590e4b	Failed	Spark Job Definition	11:11 AM, 2/28/23	13m 27s	Submitter	workspace
SDJtest_313d0c08-ef90-4ca2-bc3d-388ea13634b8	Failed	Spark Job Definition	11:11 AM, 2/28/23	13m 27s	Submitter	workspace
SDJtest_9e72242f-df2c-4034-b15b-1c14d4102b69	Failed	Spark Job Definition	11:09 AM, 2/28/23	13m 27s	Submitter	workspace
SDJtest	Failed	Spark Job Definition	11:09 AM, 2/28/23	0s	Submitter	workspace

Related content

The next step after viewing the details of an Apache Spark application is to view Spark job progress below the Notebook cell. You can refer to

- [Spark application detail monitoring](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Apache Spark application detail monitoring

Article • 01/27/2025

With Microsoft Fabric, you can use Apache Spark to run notebooks, jobs, and other kinds of applications in your workspace. This article explains how to monitor your Apache Spark application, allowing you to keep an eye on the recent run status, issues, and progress of your jobs.

View Apache Spark applications

You can view all Apache Spark applications from **Spark job definition**, or **notebook item context** menu shows the recent run option -> **Recent runs**.

The screenshot shows two views of the Microsoft Fabric workspace. The top view is a list of notebooks in the workspace, with one notebook selected. A context menu is open for this notebook, and the 'Recent runs' option is highlighted with a red box. The bottom view is a detailed list of recent runs for the selected notebook, titled 'Recent runs for Notebook 20'. This list includes columns for Application name, Submitted, Submitter, Status, Total dur..., Run kind, and Livy ID. Three recent runs are listed:

Application name	Submitted	Submitter	Status	Total dur...	Run kind	Livy ID
Notebook 20_7a559af8-2690-4314-9d02-3b...	9/11/23 5:00:00 P...	submitter	Running	31s	Manual	7a559af8-2690-4314-9d02-3b...
Notebook 20_15de81f2-68f4-4647-badd-5f8...	8/26/23 12:58:28 ...	submitter	Stopped (sessic...	20m 45s	Manual	15de81f2-68f4-4647-badd-5f8...
Notebook 20_d8da8dd5-bf6c-40b6-a73e-31...	8/13/23 6:16:29 ...	submitter	Stopped (sessic...	20m 45s	Manual	d8da8dd5-bf6c-40b6-a73e-31...

You can select the name of the application you want to view in the application list, in the application details page you can view the application details.

Monitor Apache Spark application status

Open the **Recent runs** page of the notebook or Spark job definition, you can view the status of the Apache application.

- Success

Application name	Submitted ↓	Submitter	Status	Total dura...	Run kind	Livy Id	
sparksession	7/29/22 11:02:36 AM	Submitter	Succeeded	7m 37s	-	33ecbc82-a892-...	

- Queued

Application name	Submitted ↓	Submitter	Status	Total dura...	Run kind	Livy Id	
sparksession	7/29/22 11:20:09 AM	Submitter	Queued	-	-	1213536sdva..	

- Stopped

Application name	Submitted ↓	Submitter	Status	Total dura...	Run kind	Livy Id	
sparksession	7/29/22 11:50:54 AM	Submitter	Stopped (session)	10m 30s	-	0ec4ce2e-98dd-...	

- Canceled

Application name	Submitted ↓	Submitter	Status	Total dura...	Run kind	Livy Id	
sparksession	7/29/22 11:02:36 AM	Submitter	Cancelled	7m 37s	-	33ecbc82-a892-...	

- Failed

Application name	Submitted ↓	Submitter	Status	Total dura...	Run kind	Livy Id	
sparksession	7/29/22 11:02:36 AM	Submitter	Failed	7m 37s	-	33ecbc82-a892-...	

Jobs

Open an Apache Spark application job from the **Spark job definition** or **notebook** item context menu shows the **Recent run** option -> **Recent runs** -> select a job in recent runs page.

In the Apache Spark application monitoring details page, the job runs list is displayed in the **Jobs** tab, you can view the details of each job here, including **Job ID, Description, Status, Stages, Tasks, Duration, Processed, Data read, Data written and Code snippet**.

- Clicking on Job ID can expand/collapse the job.
- Click on the job description, you can jump to job or stage page in spark UI.
- Click on the job Code snippet, you can check and copy the code related to this job.

Notebook 2-1_5e080055-ec0c-401a-85c2-4fa2effb4bb4											
 											
Jobs	Logs	Data	Related items	Status	Stages	Tasks	Duration	Processed	Data read	Data written	Code snippet
> Job 0	save at <console>:31			Succeeded	1/1	3/3 succeeded	12 sec	3 rows	0 B	3.88 KB	
> Job 1				Succeeded	0/0	0/0 succeeded	< 1 ms	0 rows	0 B	0 B	
> Job 2	\$anonfun\$recordDeltaOperation\$5 at SynapseLoggingShim.scala:95			Succeeded	1/1	1/1 succeeded	9 sec	12 rows	2.14 KB	2.33 KB	
> Job 3	\$anonfun\$recordDeltaOperation\$5 at SynapseLoggingShim.scala:95			Succeeded	1/1	50/50 succeeded	4 sec	56 rows	2.33 KB	4.24 KB	
> Job 4	\$anonfun\$recordDeltaOperation\$5 at SynapseLoggingShim.scala:95			Succeeded	1/1	1/1 succeeded	< 1 ms	50 rows	4.24 KB	0 B	
> Job 5	\$anonfun\$recordDeltaOperation\$5 at SynapseLoggingShim.scala:95			Succeeded	1/1	50/50 succeeded	1 sec	5 rows	2.29 KB	0 B	
> Job 6	takeAsList at <console>:36			Succeeded	1/1	1/1 succeeded	1 sec	1 row	2.17 KB	0 B	

Resources

The executor usage graph under the **Resources** tab visualizes the allocation and utilization of Spark executors for the current Spark application in near real-time during Spark execution. You can refer to: [Monitor Apache Spark Applications Resource Utilization](#).

Summary panel

In the Apache Spark application monitoring page, click the **Properties** button to open/collapse the summary panel. You can view the details for this application in **Details**.

- Status for this spark application.
- This Spark application's ID.
- Total duration.
- Running duration for this spark application.
- Queued duration for this spark application.
- Livy ID
- Submitter for this spark application.
- Submit time for this spark application.
- Number of executors.

<h2>Details</h2>	
Status	<input checked="" type="radio"/> Stopped (session timed out)
Application ID	11112222-bbbb-3333-cccc-4444ddd5555
Total duration	22m 0s
Running duration	0
Queued duration	0
Livy ID	0000aaaa-11bb-cccc-dd22-eeeeee333333
Submitter	Submitter
Submit time	3/31/23 4:21:39 PM
Number of executors	2

Logs

For the **Logs** tab, you can view the full log of **Livy**, **Prelaunch**, **Driver** log with different options selected in the left panel. And you can directly retrieve the required log information by searching keywords and view the logs by filtering the log status. Click Download Log to download the log information to the local.

Sometimes no logs are available, such as the status of the job is queued and cluster creation failed.

Live logs are only available when app submission fails, and driver logs are also provided.

Logs Log list. Select one << Driver (stderr) - stderr-active

Length: 229789 (102400 loaded)

Driver (stderr)

- Latest stderr
- All stderr

Driver (stdout)

- Latest stdout
- All stdout

Prelaunch (stdout)

Livy

Driver (stderr) - stderr-active

Length: 229789 (102400 loaded)

↑ Load older log Load older logs

```

2023-05-05 04:14:38,014 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint [dispatcher-0] : Successfully stopped SparkContext
2023-05-05 04:14:38,043 INFO SparkContext [shutdown-hook-0]: Unregistering ApplicationMaster
2023-05-05 04:14:38,059 INFO AMRMClientImpl [shutdown-hook-0]: Waiting for application to be succeeded
2023-05-05 04:14:38,160 INFO AMRMClientImpl [shutdown-hook-0]: Waiting for application to be succeeded
2023-05-05 04:14:38,262 INFO AMRMClientImpl [shutdown-hook-0]: Waiting for application to be succeeded
2023-05-05 04:14:38,365 INFO ApplicationMaster [shutdown-hook-0]: Deleting staging directory wasbs://1c50ae07-d66e-4822-b4c5-9184b563c61
2023-05-05 04:14:38,383 WARN AzureFileSystemThreadpoolExecutor [shutdown-hook-0]: Disabling threads for Delete operation as thread count
2023-05-05 04:14:38,396 INFO AzureFileSystemThreadpoolExecutor [shutdown-hook-0]: Time taken for Delete operation is: 13 ms with threads
2023-05-05 04:14:38,419 INFO RpcAppSparkContextServer [shutdown-hook-0]: Closing remote SparkContext service at 10.1.128.5:18883, remote address
2023-05-05 04:14:38,419 INFO ShutdownHookManager [shutdown-hook-0]: Shutdown hook called
2023-05-05 04:14:38,420 INFO ShutdownHookManager [shutdown-hook-0]: Deleting directory /mnt/var/hadoop/tmp/nm-secondary-local-dir/usercache
2023-05-05 04:14:38,424 INFO ShutdownHookManager [shutdown-hook-0]: Deleting directory /mnt/var/hadoop/tmp/nm-secondary-local-dir/usercache
2023-05-05 04:14:38,427 INFO RpcAppSender [shutdown-hook-0]: Closing RPC app sender
2023-05-05 04:14:38,428 INFO RpcAppSender [shutdown-hook-0]: Sending remaining events
2023-05-05 04:14:38,436 INFO RpcAppSender [shutdown-hook-0]: Sent RPC app end event of application_1683259812003_0001/1
2023-05-05 04:14:38,447 INFO RpcAppSender [shutdown-hook-0]: RPC app sender closed
2023-05-05 04:14:38,449 INFO RpcAppEventQueue [shutdown-hook-0]: Max number of events in queue: 160.
2023-05-05 04:14:38,516 INFO MetricsSystemImpl [shutdown-hook-0]: Stopping azure-file-system metrics system...
2023-05-05 04:14:38,516 INFO MetricsSystemImpl [shutdown-hook-0]: azure-file-system metrics system stopped.
2023-05-05 04:14:38,516 INFO MetricsSystemImpl [shutdown-hook-0]: azure-file-system metrics system shutdown complete.

```

End of LogType:stderr-active

Data

For the **Data** tab, you can copy the data list on clipboard, download the data list and single data, and check the properties for each data.

- The left panel can be expanded or collapse.
- The name, read format, size, source and path of the input and output files will be displayed in this list.
- The files in input and output can be downloaded, copy path and view properties.

relatedItems > Notebook > **Notebook_ID**

Refresh Cancel Spark history server

Jobs Logs Data Related items

Data << Showing 1 - 1 of 1 items

Input	File name	Read for...	Size	Source
Output	DummyDeltaTable			
	00000000000000000000.json	: delta	Failed to load ⓘ	Blob storage
	part-00002-8345602c-731d...	↓ Download	Failed to load ⓘ	Blob storage
		Copy path >		
		Properties		

Properties

File name
00000000000000000000.json

File Path
wasbs://84cd7bd4-4071-41c6-95f9-ca... ⓘ

Read format
delta

Size
Fail to load ⓘ

Modified
Fail to load ⓘ

Group
Fail to load ⓘ

Owner
Fail to load ⓘ

Permissions
Fail to load ⓘ

Close

Item snapshots

The **Item snapshots** tab allows you to browse and view items associated with the Apache Spark application, including Notebooks, Spark job definition, and/or Pipelines.

The item snapshots page displays the snapshot of the code and parameter values at the time of execution for Notebooks. It also shows the snapshot of all settings and parameters at the time of submission for Spark job definitions. If the Apache Spark application is associated with a pipeline, the related item page also presents the corresponding pipeline and the Spark activity.

In the Item snapshots screen, you can:

- Browse and navigate the related items in the hierarchical tree.
- Click the 'A list of more actions' ellipse icon for each item to take different actions.
- Click the snapshot item to view its content.
- View the Breadcrumb to see the path from the selected item to the root.

The screenshot shows the 'Item snapshots' section of the Apache Spark UI. The left sidebar lists 'Jobs', 'Resources', 'Logs', 'Data', and 'Item snapshots'. The 'Item snapshots' tab is active. The main area displays a hierarchical tree of notebooks under 'Notebook Root'. A specific notebook entry, 'Notebook 2-1', is selected and highlighted with a red box around its '... A list of more actions' button. A context menu is open over this button, listing options: 'Save as notebook', 'Open this notebook', and 'Restore'. To the right of the tree, there's a 'Breadcrumb, clickable' link showing the path 'Notebook 2 > Notebook 2-1'. Below the tree, a code snippet is shown:

```
1 print('Notebook 2-1')
```

 and

```
2-1
```

. Further down, another code snippet is shown:

```
1 mssparkutils.notebook.run('Notebook 2-1-1')
```

. At the bottom of the main area, it says 'View notebook run: Notebook 2-1-1'. On the far right, there's a 'Details' panel with fields like 'Snapshot ID', 'Livy ID', 'Job end time', 'Duration', 'Submitter', and 'Default lakehouse'. A magnifying glass icon is also present in the details panel.

ⓘ Note

The Notebook snapshots feature currently doesn't support notebooks that are in a running state or in a high-concurrency Spark session.

Diagnostics

The diagnostic panel provides users with real-time recommendations and error analysis, which are generated by the Spark Advisor through an analysis of the user's code. With built-in patterns, the Apache Spark Advisor helps users avoid common mistakes and analyzes failures to identify their root cause.

Click to check advisors

Drag to change the height of the panel

Expand/collapse diagnostic panel

Diagnostics ② ③

① Enable 'spark.advises.divisionExprConvertRule.enable' to reduce rounding error propagation
This query contains the expression 'CAST(id AS DOUBLE) / CAST(id AS DOUBLE) / CAST(id AS DOUBLE)' with Double type. We recommend that you enable the configuration 'spark.advises.divisionExprConvertRule.enable' which can help convert 'CAST(id AS DOUBLE) / CAST(id AS DOUBLE)' to 'CAST(id AS DOUBLE) / (CAST(id AS DOUBLE) * CAST(id AS DOUBLE))' to reduce the rounding error propagation.

① Enable 'spark.advises.divisionExprConvertRule.enable' to reduce rounding error propagation
This query contains the expression '(CAST(id AS DOUBLE) + 1.0D) / (CAST(id AS DOUBLE) + 2.0D) / (CAST(id AS DOUBLE) * 3.0D)' with Double type. We recommend that you enable the configuration 'spark.advises.divisionExprConvertRule.enable' which can help convert '(CAST(id AS DOUBLE) + 1.0D) / (CAST(id AS DOUBLE) + 2.0D) / (CAST(id AS DOUBLE) * 3.0D)' to '(CAST(id AS DOUBLE) + 1.0D) / ((CAST(id AS DOUBLE) + 2.0D) * (CAST(id AS DOUBLE) * 3.0D))' to reduce the rounding error propagation.

① Enable 'spark.advises.divisionExprConvertRule.enable' to reduce rounding error propagation

Expand/collapse to check advice content

Related content

The next step after viewing the details of an Apache Spark application is to view **Spark job progress** below the Notebook cell. You can refer to:

- [Notebook contextual monitoring and debugging](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Use extended Apache Spark history server to debug and diagnose Apache Spark applications

Article • 09/11/2024

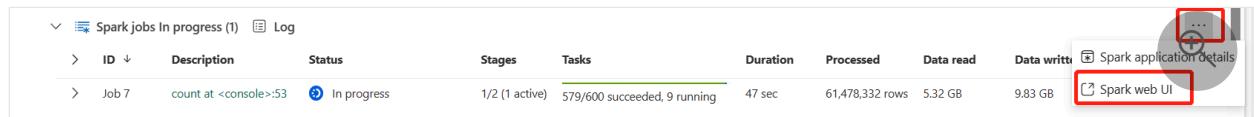
This article provides guidance on how to use the extended Apache Spark history server to debug and diagnose completed and running Apache Spark applications.

Access the Apache Spark history server

The Apache Spark history server is the web user interface for completed and running Spark applications. You can open the Apache Spark web user interface (UI) from the progress indicator notebook or the Apache Spark application detail page.

Open the Spark web UI from progress indicator notebook

When an Apache Spark job is triggered, the button to open **Spark web UI** is inside the **More action** option in the progress indicator. Select **Spark web UI** and wait for a few seconds, then the Spark UI page appears.



Open the Spark web UI from Apache Spark application detail page

The Spark web UI can also be opened through the Apache Spark application detail page. Select **Monitoring hub** on the left side of the page, and then select an Apache Spark application. The detail page of the application appears.

Monitoring hub

Monitoring hub is a station to view and track active activities across different products.

application_0001

Jobs Logs Data Related items

ID	Description	Status	Stages	Tasks	Duration	Processed	Data
Job 0	load at NativeMethodAccessImpl.java:0	Succeeded	1/1	1/1 succeeded	4 sec	1 row	64 KB
Job 1	load at NativeMethodAccessImpl.java:0	Succeeded	1/1	1/1 succeeded	1 sec	1,001 rows	89.03
Job 2	toString at String.java:2994	Succeeded	1/1	3/3 succeeded	1 sec	20 rows	8.11
Job 3	toString at String.java:2994	Succeeded	1/1	50/50 succeeded	2 sec	60 rows	6.33
Job 4	toString at String.java:2994	Succeeded	1/1	1/1 succeeded	< 1 ms	50 rows	4.51
Job 5	save at NativeMethodAccessImpl.java:0	Succeeded	1/1	1/1 succeeded	3 sec	2,000 rows	93.52
Job 6	\$anonfun\$recordDeltaOperationInternal\$1 at SynapseLoggingShim.scala:95	Succeeded	1/1	50/50 succeeded	< 1 ms	5 rows	2.77
Job 7	toString at String.java:2994	Succeeded	1/1	4/4 succeeded	< 1 ms	26 rows	10.28
Job 8	toString at String.java:2994	Succeeded	1/1	50/50 succeeded	1 sec	63 rows	8.38
Job 9	toString at String.java:2994	Succeeded	1/1	1/1 succeeded	< 1 ms	50 rows	4.51

Details

- Status: Success
- Application ID: 22223333-cccc-4444-dddd-5555eeee6666
- Total duration: 1m 6s
- Running duration: 1m 0s
- Queued duration: 6s
- Livy ID: 0000aaaa-11bb-cccc-dd22-eeeeee333333
- Job instance ID: 6666aaaa-77bb-cccc-dd88-eeeeee999999
- Submitter: submitter
- Submit time: 5/5/23 12:13:40 PM

For an Apache Spark application whose status is running, the button shows **Spark UI**. Select **Spark UI** and the Spark UI page appears.

Workspacename> notebook > Running application

Jobs Logs Data Related items

ID	Description	Status	Stages	Tasks	Duration	Processed	Data
Job 0	save at <console>:34	Succeeded	1/1	3/3 succeeded	6 sec	3 rows	0 B
Job 1		Succeeded	0/0	0/0 succeeded	< 1 ms	0 rows	0 B
Job 2	toString at String.java:2994	Succeeded	1/1	1/1 succeeded	1 sec	12 rows	2.14 KB
Job 3	toString at String.java:2994	Succeeded	1/1	50/50 succeeded	3 sec	56 rows	2.18 KB
Job 4	toString at String.java:2994	Succeeded	1/1	1/1 succeeded	< 1 ms	50 rows	4.24 KB
Job 5	\$anonfun\$recordDeltaOperationInternal\$1 at SynapseLoggingShim.scala:95	Succeeded	1/1	50/50 succeeded	1 sec	5 rows	2.22 KB
Job 6	takeAsList at <console>:39	Succeeded	1/1	1/1 succeeded	1 sec	1 row	2.2 KB
Job 7	collect at <console>:30	Succeeded	1/1	8/8 succeeded	< 1 ms	0 rows	0 B

Details

- Status: Running
- Application ID: 22223333-cccc-4444-dddd-5555eeee6666
- Total duration: 51s
- Running duration: 0
- Queued duration: 0
- Livy ID: 0000aaaa-11bb-cccc-dd22-eeeeee333333
- Submitter: submitter
- Submit time: 5/8/23 2:53:45 PM
- Number of executors: 1

For an Apache Spark application whose status is ended, the ended status can be **Stopped**, **Failed**, **Canceled**, or **Completed**. The button shows **Spark history server**. Select **Spark history server** and the Spark UI page appears.

Graph tab in Apache Spark history server

Select the Job ID for the job you want to view. Then, select **Graph** on the tool menu to get the job graph view.

Overview

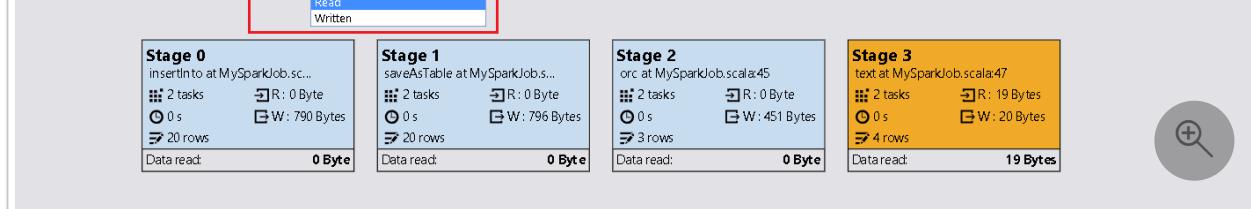
You can see an overview of your job in the generated job graph. By default, the graph shows all jobs. You can filter this view by **Job ID**.

Display

By default, the **Progress** display is selected. You can check the data flow by selecting **Read** or **Written** in the **Display** dropdown list.

Spark Application & Job Graph

Job ID: All jobs ▾ Display: Read Progress Read Written ▾ Playback ▶ 0 s / 5 s Zoom to fit 



The graph node displays the colors shown in the heatmap legend.

Stage 0: insert into at MySparkJob.scala...
2 tasks, 0 s, 20 rows, Data read: 0 Byte

Stage 1: saveAsTable at MySparkJob.scala...
2 tasks, 0 s, 20 rows, Data read: 0 Byte

Stage 2: orc at MySparkJob.scala:45
2 tasks, 0 s, 3 rows, Data read: 0 Byte

Stage 3: text at MySparkJob.scala:47
2 tasks, 0 s, 4 rows, Data read: 19 Bytes

Heatmap Legend:
 -2.5
 -1.5
 -0.5
 0.5
 1.5
 2.5
 Std. Dev.

Playback

To play back the job, select **Playback**. You can select **Stop** at any time to stop. The task colors show different statuses when playing back:

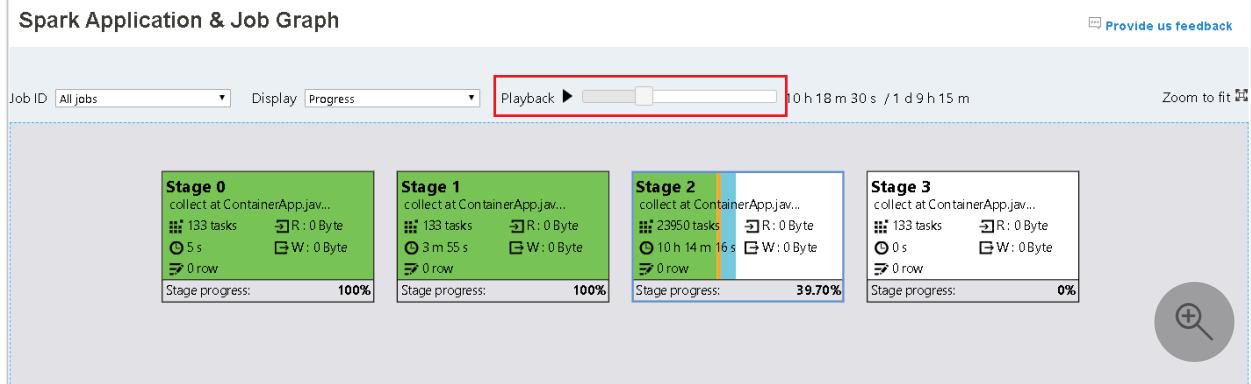
 Expand table

Color	Meaning
Green	Succeeded: The job has completed successfully.
Orange	Retried: Instances of tasks that failed but don't affect the final result of the job. These tasks had duplicate or retry instances that may succeed later.
Blue	Running: The task is running.
White	Waiting or skipped: The task is waiting to run, or the stage has skipped.
Red	Failed: The task has failed.

The following image shows green, orange, and blue status colors.

Spark Application & Job Graph

Job ID: All jobs ▾ Display: Progress ▾ Playback ▶ 10 h 18 m 30 s / 1 d 9 h 15 m Zoom to fit 



The following image shows green and white status colors.

Stage 0: collect at ContainerApp.java...
133 tasks, 5 s, 0 row, Stage progress: 100%

Stage 1: collect at ContainerApp.java...
133 tasks, 3 m 55 s, 0 row, Stage progress: 100%

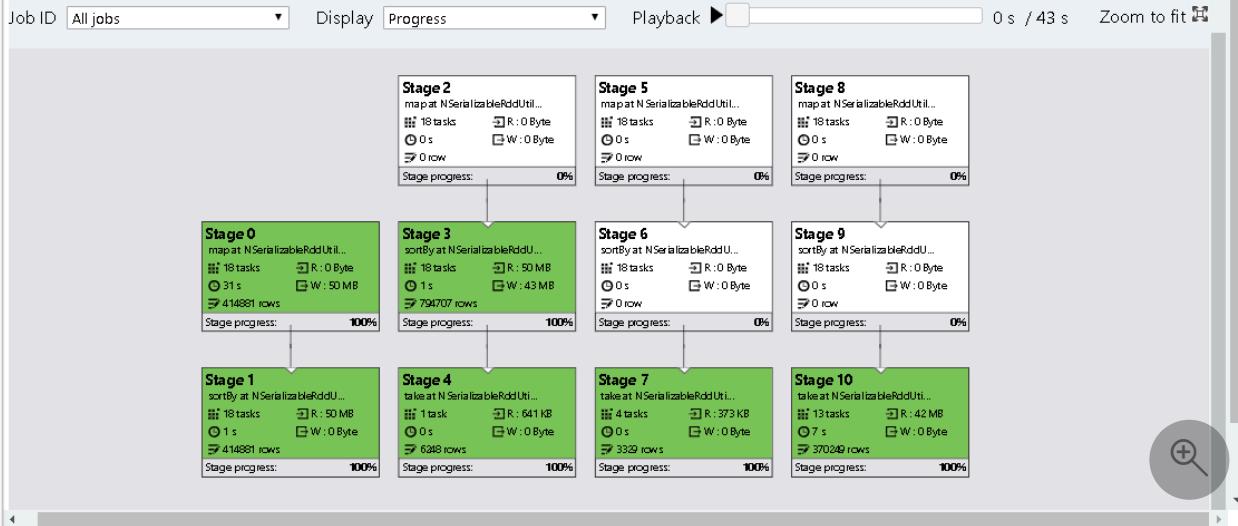
Stage 2: collect at ContainerApp.java...
23950 tasks, 10 h 14 m 16 s, 0 row, Stage progress: 39.70%

Stage 3: collect at ContainerApp.java...
133 tasks, 0 s, 0 row, Stage progress: 0%

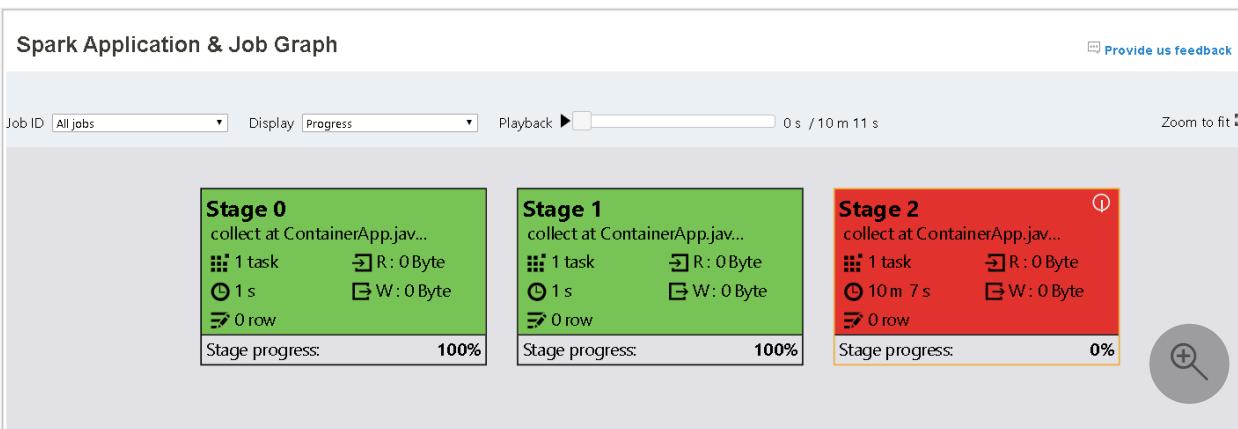
The following image shows green and white status colors.

Spark Application & Job Graph

[Provide us feedback](#)



The following image shows red and green status colors.

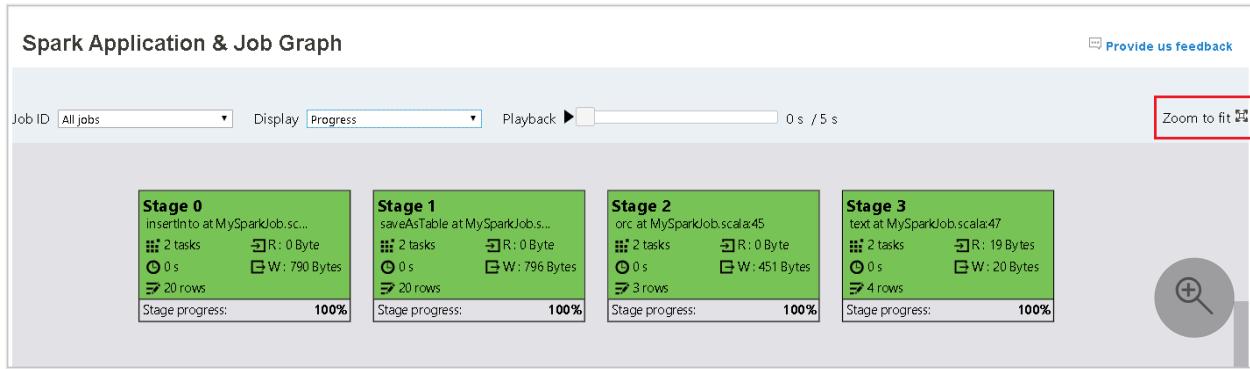


(!) Note

The Apache Spark history server allows playback for each completed job (but does not allow playback for incomplete jobs).

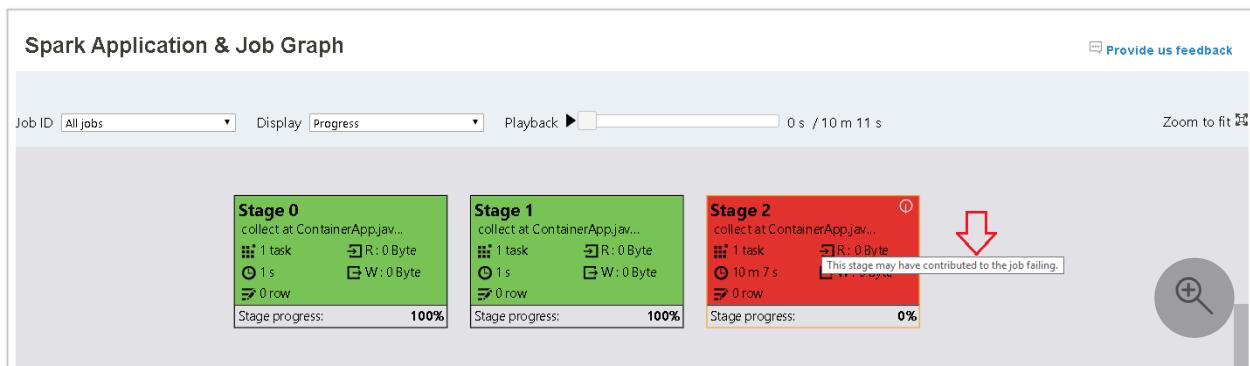
Zoom

Use your mouse scroll to zoom in and out on the job graph, or select **Zoom to fit** to make it fit to screen.



Tooltips

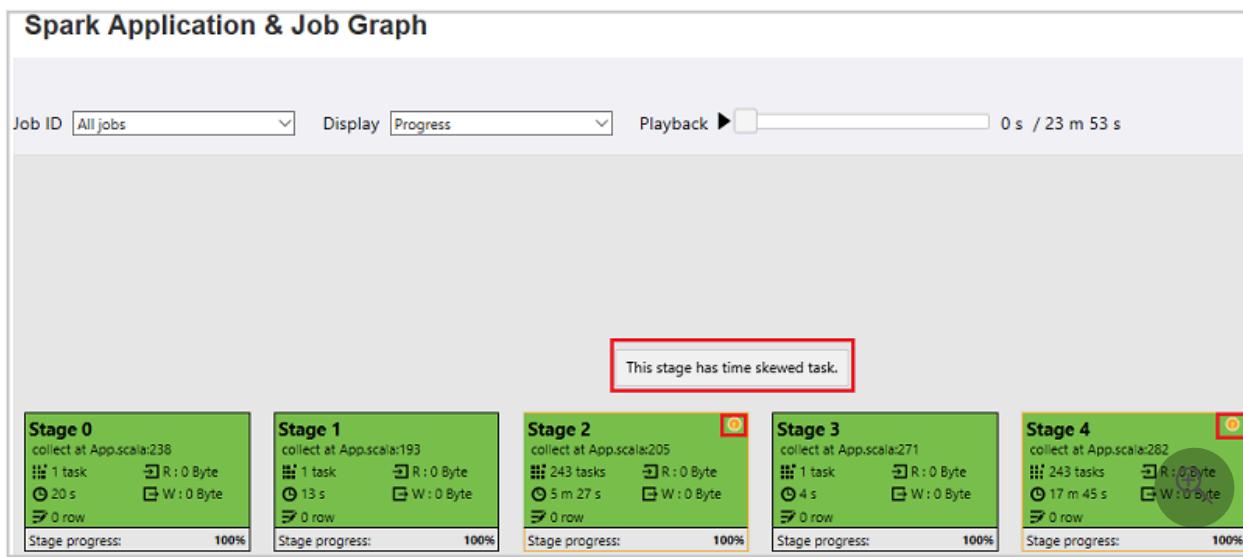
Hover on graph node to see the tooltip when there are failed tasks, and select a stage to open its stage page.



On the job graph tab, stages have a tooltip and a small icon displayed if they have tasks that meet the following conditions:

Expand table

Condition	Description
Data skew	Data read size > average data read size of all tasks inside this stage * 2 and data read size > 10 MB.
Time skew	Execution time > average execution time of all tasks inside this stage * 2 and execution time > 2 minutes.



Graph node description

The job graph node displays the following information of each stage:

- ID
- Name or description
- Total task number
- Data read: the sum of input size and shuffle read size
- Data write: the sum of output size and shuffle writes size
- Execution time: the time between start time of the first attempt and completion time of the last attempt
- Row count: the sum of input records, output records, shuffle read records and shuffle write records
- Progress

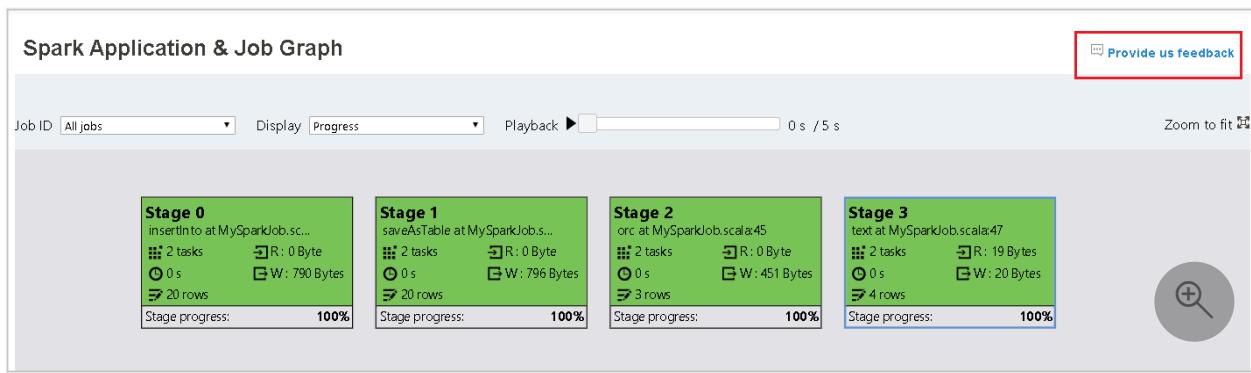
ⓘ Note

By default, the job graph node displays information from the last attempt of each stage (except for stage execution time). However, during playback, the graph node shows information of each attempt.

The data size of read and write is 1MB = 1000 KB = 1000 * 1000 bytes.

Provide feedback

Send feedback with issues by selecting **Provide us feedback**.



Stage number limit

For performance consideration, by default the graph is only available when the Spark application has less than 500 stages. If there are too many stages, it will fail with an error like this:

```
The number of stages in this application exceeds limit (500), graph page is
disabled in this case.
```

As a workaround, before starting a Spark application, please apply this Spark configuration to increase the limit:

```
spark.ui.enhancement.maxGraphStages 1000
```

But please notice that this may cause bad performance of the page and the API, because the content can be too large for browser to fetch and render.

Explore the Diagnosis tab in Apache Spark history server

To access the Diagnosis tab, select a job ID. Then select **Diagnosis** on the tool menu to get the job Diagnosis view. The diagnosis tab includes **Data Skew**, **Time Skew**, and **Executor Usage Analysis**.

Check the **Data Skew**, **Time Skew**, and **Executor Usage Analysis** by selecting the tabs respectively.

Diagnosis

Data Skew Time Skew Executor Usage Analysis

1. Specify Parameters:

Task data read > x Average.

Task data read > MB.



Data Skew

When you select the **Data Skew** tab, the corresponding skewed tasks are displayed based on the specified parameters.

- **Specify Parameters** - The first section displays the parameters, which are used to detect Data Skew. The default rule is: task data read is greater than three times of the average task data read, and the task data read is more than 10 MB. If you want to define your own rule for skewed tasks, you can choose your parameters. The **Skewed Stage** and **Skew Char** sections are refreshed accordingly.
- **Skewed Stage** - The second section displays stages, which have skewed tasks meeting the criteria previously specified. If there's more than one skewed task in a stage, the skewed stage table only displays the most skewed task (for example, the largest data for data skew).

Diagnosis

Data Skew Time Skew Executor Usage Analysis

1. Specify Parameters:

Task data read > x Average.

Task data read > MB.

2. Skewed Stage:

Select a Stage to view more details and solutions

Stage ID	Attempt ID	Stage name	Task ID	Data read	Average dat...	Execution time	Average exe...
10	0	take at NSer...	71	18.6MB	2.21MB	2 s	0 s



- **Skew Chart** - When a row in the skew stage table is selected, the skew chart displays more task distribution details based on data read and execution time. The skewed tasks are marked in red and the normal tasks are marked in blue. The chart displays up to 100 sample tasks, and the task details are displayed in right bottom panel.



Time Skew

The **Time Skew** tab displays skewed tasks based on task execution time.

- **Specify Parameters** - The first section displays the parameters, which are used to detect time skew. The default criteria to detect time skew is: task execution time is greater than three times of average execution time and task execution time is greater than 30 seconds. You can change the parameters based on your needs. The **Skewed Stage** and **Skew Chart** display the corresponding stages and tasks information just like the **Data Skew** tab described previously.
- Select **Time Skew**, then filtered result is displayed in **Skewed Stage** section according to the parameters set in section **Specify Parameters**. Select one item in **Skewed Stage** section, then the corresponding chart is drafted in section 3, and the task details are displayed in right bottom panel.

Diagnosis

Data Skew **Time Skew** Executor Usage Analysis

1. Specify Parameters:

Task execution time > **2** x Average.

Task execution time > **2** Min.

2. Skewed Stage:

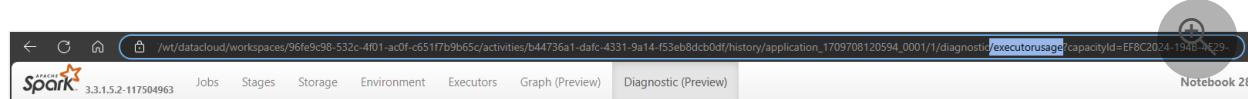
Select a Stage to view more details and solutions

Stage ID	Attempt ID	Stage name	Task ID	Data read	Average dat...	Execution time	Average exe...
4	0	collect at Ap...	417	0MB	0MB	6 m 33 s	2 m 5 s



Executor Usage Analysis

This feature has been deprecated in Fabric now. If you still want to use this as a workaround, please access the page by explicitly adding "/executorusage" behind path "/diagnostic" in the URL, like this:



Related content

- [Apache Spark monitoring overview](#)
- [Browse item recent runs](#)
- [Monitor Apache Spark jobs within notebooks](#)
- [Monitor Apache Spark job definition](#)
- [Monitor Apache Spark application details](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Monitor Apache Spark capacity consumption

Article • 11/19/2024

The purpose of this article is to offer guidance for admins who want to monitor activities in the capacities they manage. By utilizing the Apache Spark capacity consumption reports available in the [Microsoft Fabric Capacity Metrics app](#), admins can gain insights into the billable Spark capacity consumption for items, including Lakehouse, Notebook, and Apache Spark job definitions. Some Spark capacity consumption activities aren't reported in the app.

Spark capacity consumption reported

The following operations from lakehouses, notebooks, and Spark job definitions are treated as billable activities.

[+] Expand table

Operation name	Item	Comments
Lakehouse operations	Lakehouse	Users preview table in the Lakehouse explorer.
Lakehouse table load	Lakehouse	Users load delta table in the Lakehouse explorer.
Notebook run	Notebook	Notebook runs manually by users.
Notebook HC run	Notebook	Notebook runs under the high concurrency Apache Spark session.
Notebook scheduled run	Notebook	Notebook runs triggered by notebook scheduled events.
Notebook pipeline run	Notebook	Notebook runs triggered by pipeline.
Notebook VS Code run	Notebook	Notebook runs in VS Code.
Spark job run	Spark Job Definition	Spark batch job runs initiated by user submission.
Spark job scheduled run	Spark Job Definition	Batch job runs triggered by notebook scheduled events.

Operation name	Item	Comments
Spark job pipeline run	Spark Job Definition	Batch job runs triggered by pipeline.
Spark job VS Code run	Spark Job Definition	Spark job definition submitted from VS Code.

Spark capacity consumption that isn't reported

There are some Spark capacity consumption activities that aren't reported in the metrics app. These activities include system Spark jobs for library management and certain system Spark jobs for Spark Live pool or live sessions.

- **Library management** - The capacity consumption associated with library management at the workspace level isn't reported in the metrics app.
- **System Spark jobs** - Spark capacity consumption that isn't associated with a notebook, a Spark job definition, or a lakehouse, isn't included in the capacity reporting.

Capacity consumption reports

All Spark related operations are classified as [background operations](#). Capacity consumption from Spark is displayed under a notebook, a Spark job definition, or a lakehouse, and is aggregated by operation name and item.

KustoDatabase	Lakehouse	MLExperiment	MLModel	PaginatedReport	SparkJobDefinition	SynapseNotebook
Items (14 days)						
Item						
housing-study-notebook \ SynapseNotebook \ Spark PM Team				CU(s)	Duration (s)	Users
Notebook 1 \ SynapseNotebook \ bw workspace				0.00	0.00	0
Notebook 1 \ SynapseNotebook \ dbrowne_Trident				5096.05	1274.01	1
Notebook 2 \ SynapseNotebook \ bw workspace				20610.67	5152.66	1
Notebook 3 \ SynapseNotebook \ umaws01				5728.82	1432.19	1
Notebook 4 \ SynapseNotebook \ umaws01				5658.31	1414.58	1
Notebook-DeletionTest \ SynapseNotebook \ umaws1				5416.29	1354.07	1
Notebook-DeletionTest2 \ SynapseNotebook \ umaws1				6736.91	1684.23	1
Notebook-DeletionTest3 \ SynapseNotebook \ umaws1				9511.99	2378.00	1
NotebookSample \ SynapseNotebook \ CapacityBugBash				9409.55	2352.39	1
Uma BugBash NB1 \ SynapseNotebook \ CapacityBugBash				11302.38	2825.60	1
Notebook Interactive Run				84094409.09	21023476....	1
Notebook Scheduled Run				2887999.14	721997.64	1
Total				81206409.94	20301478.53	1
				84196562.99	21047589....	1

Background operations report

Background operations are displayed for a specific [timepoint](#). In the report's table, each row refers to a user operation. Review the **User** column to identify who performed a

specific operation. If you need more information about a specific operation, you can use its **Operation ID** to look it up in the Microsoft Fabric [monitoring hub](#).

Background Operations										
Item	Operation	Start	End	Status	User	Duration (s)	Total CU (s)	Timepoint CU (s)	% of Capacity	
Notebook-DeletionTest \ SynapseNotebook \ umaws1	Notebook Interact...	4/29/2023 6:11:58 A...	5/1/2023 4:57:52 AM	Failure	Power BI Service	1306	5224	1.81	0.05%	
Notebook-DeletionTest2 \ SynapseNotebook \ umaws1	Notebook Interact...	4/29/2023 6:18:28 A...	5/1/2023 5:04:55 AM	Cancelled	Power BI Service	39	157	0.05	0.00%	
Notebook-DeletionTest3 \ SynapseNotebook \ umaws1	Notebook Interact...	4/29/2023 6:19:05 A...	5/1/2023 4:56:52 AM	Failure	Power BI Service	2338	9354	3.25	0.08%	
Notebook-DeletionTest3 \ SynapseNotebook \ umaws1	Notebook Interact...	4/29/2023 6:41:01 A...	5/1/2023 5:01:55 AM	Failure	Power BI Service	2352	9409	3.27	0.09%	
NotebookSample \ SynapseNotebook \ CapacityBugBash	Notebook Interact...	4/26/2023 8:15:33 PM	5/1/2023 5:07:58 AM	Failure	Power BI Service	1417	5670	1.97	0.05%	
NotebookSample \ SynapseNotebook \ CapacityBugBash	Notebook Interact...	4/26/2023 8:43:42 PM	5/1/2023 4:40:35 AM	Failure	Power BI Service	1408	5632	1.96	0.05%	
Uma BugBash NB1 \ SynapseNotebook \ CapacityBugBash	Notebook Interact...	4/26/2023 8:47:54 PM	5/1/2023 3:19:40 PM	InProgress	Power BI Service	412281	1649126	572.61	14.91%	
Uma BugBash NB1 \ SynapseNotebook \ CapacityBugBash	Notebook Schedu...	4/26/2023 9:00:04 PM	5/1/2023 3:19:09 PM	InProgress	Power BI Service	86336	345346	119.91	3.12%	
Uma BugBash NB1 \ SynapseNotebook \ CapacityBugBash	Notebook Schedu...	4/26/2023 9:10:04 PM	5/1/2023 3:19:10 PM	InProgress	Power BI Service	86396	345589	120.00	3.12%	
Uma BugBash NB1 \ SynapseNotebook \ CapacityBugBash	Notebook Schedu...	4/26/2023 9:20:04 PM	5/1/2023 3:19:40 PM	InProgress	Power BI Service	86396	345586	120.00	3.12%	
Uma BugBash NB1 \ SynapseNotebook \ CapacityBugBash	Notebook Schedu...	4/26/2023 9:30:05 PM	5/1/2023 3:19:10 PM	InProgress	Power BI Service	86402	345610	120.00	3.13%	
Uma BugBash NB1 \ SynapseNotebook \ CapacityBugBash	Notebook Schedu...	4/26/2023 9:40:04 PM	5/1/2023 3:19:42 PM	InProgress	Power BI Service	86398	345597	120.00	3.12%	
Uma BugBash NB1 \ SynapseNotebook \ CapacityBugBash	Notebook Schedu...	4/26/2023 9:50:04 PM	5/1/2023 3:19:38 PM	InProgress	Power BI Service	86441	345767	120.06	3.13%	
Uma BugBash NB1 \ SynapseNotebook \ CapacityBugBash	Notebook Schedu...	4/26/2023 10:00:04 ...	5/1/2023 3:19:10 PM	InProgress	Power BI Service	86392	345572	119.99	3.12%	
Uma BugBash NB1 \ SynapseNotebook \ CapacityBugBash	Notebook Schedu...	4/26/2023 10:10:04 ...	5/1/2023 3:19:09 PM	InProgress	Power BI Service	86384	345540	119.98	3.12%	
Uma BugBash NB1 \ SynapseNotebook \ CapacityBugBash	Notebook Schedu...	4/26/2023 10:20:04 ...	5/1/2023 3:19:08 PM	InProgress	Power BI Service	86388	345557	119.99	3.12%	
Uma BugBash NB1 \ SynapseNotebook \ CapacityBugBash	Notebook Schedu...	4/26/2023 10:30:05 ...	5/1/2023 3:19:42 PM	InProgress	Power BI Service	86314	345370	110.02	3.12%	
Total						3215497	11847875	4,113.91	107.13%	

Related content

- [Install the Premium metrics app](#)
- [Use the Premium metrics app](#)

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Ask the community 

Intelligent cache in Microsoft Fabric

Article • 11/15/2023

The intelligent cache feature works seamlessly behind the scenes and caches data to help speed up the execution of Apache Spark jobs in Microsoft Fabric as it reads from your OneLake or Azure Data Lake Storage (ADLS) Gen2 storage via shortcuts. It also automatically detects changes to the underlying files and automatically refreshes the files in the cache, providing you with the most recent data. When the cache size reaches its limit, the cache automatically releases the least read data to make space for more recent data. This feature lowers the total cost of ownership by improving performance up to 60% on subsequent reads of the files that are stored in the available cache.

When the Apache Spark engine in Microsoft Fabric queries a file or table from your lakehouse, it makes a call to the remote storage to read the underlying files. With every query request to read the same data, the Spark engine must make a call to remote storage each time. This redundant process adds latency to your total processing time. Spark has a caching requirement that you must manually set and release the cache to minimize the latency and improve overall performance. However, this requirement can result in stale data if the underlying data changes.

Intelligent cache simplifies the process by automatically caching each read within the allocated cache storage space on each Spark node where data files are cached in SSD. Each request for a file checks to see if the file exists in the local node cache and compare the tag from the remote storage to determine if the file is stale. If the file doesn't exist or if the file is stale, Spark reads the file and store it in the cache. When the cache becomes full, the file with the oldest last access time is evicted from the cache to allow for more recent files.

Intelligent cache is a single cache per node. If you're using a medium-sized node and run with two small executors on that single node, the two executors share the same cache. Also, this data file level caching makes it possible for multiple queries to use the same cache if they're accessing the same data or data files.

How it works

In Microsoft Fabric (Runtime 1.1 and 1.2), intelligent caching is enabled by default for all the Spark pools for all workspaces with cache size with 50%. The actual size of the available storage and the cache size on each node depends on the node family and node size.

When to use intelligent cache

This feature benefits you if:

- Your workload requires reading the same file multiple times and the file size fits in the cache.
- Your workload uses Delta Lake tables, Parquet, or CSV file formats.

You don't see the benefit of intelligent cache if:

- You're reading a file that exceeds the cache size. If so, the beginning of the files could be evicted, and subsequent queries have to refetch the data from the remote storage. In this case, you don't see any benefits from the intelligent cache, and you might want to increase your cache size and/or node size.
- Your workload requires large amounts of shuffle. Disabling the intelligent cache frees up available space to prevent your job from failing due to insufficient storage space.

Enable and disable the intelligent cache

You can disable or enable the intelligent cache within a session by running the following code in your notebook or setting this configuration at the workspace or *Environment* item level.

Scala

```
spark.conf.set("spark.synapse.vegas.useCache", "false/true")
```

Next steps

- [What is Spark compute in Microsoft Fabric?](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

AI services in Fabric (preview)

Article • 12/01/2024

ⓘ Important

This feature is in [preview](#).

Azure AI services help developers and organizations rapidly create intelligent, cutting-edge, market-ready, and responsible applications with prebuilt and customizable APIs and models. Formerly named **Azure Cognitive Services**, Azure AI services empower developers even when they don't have direct AI or data science skills or knowledge. The goal of Azure AI services is to help developers create applications that can see, hear, speak, understand, and even begin to reason.

Fabric provides two options to use Azure AI services:

- **Pre-built AI models in Fabric (preview)**

Fabric seamlessly integrates with Azure AI services, allowing you to enrich your data with prebuilt AI models without any prerequisite. We recommend this option because you can use your Fabric authentication to access AI services, and all usages are billed against your Fabric capacity. This option is currently in public preview, with limited AI services available.

Fabric offers [Azure OpenAI Service](#), [Text Analytics](#), and [Azure AI Translator](#) by default, with support for both SynapseML and the RESTful API. You can also use the [OpenAI Python Library](#) to access Azure OpenAI service in Fabric. For more information about available models, visit [prebuilt AI models in Fabric](#).

- **Bring your own key (BYOK)**

You can provision your AI services on Azure, and bring your own key to use them from Fabric. If the prebuilt AI models don't yet support the desired AI services, you can still use BYOK (Bring your own key).

To learn more about how to use Azure AI services with BYOK, visit [Azure AI services in SynapseML with bring your own key](#).

Prebuilt AI models in Fabric (preview)

ⓘ Note

Prebuilt AI models are currently available in preview and offered for free, with a limit on the number of concurrent requests per user. For Open AI models, the limit is 20 requests per minute per user.

Azure OpenAI Service ↗

[REST API](#), [Python SDK](#), [SynapseML](#)

- GPT-35-turbo: GPT-3.5 models can understand and generate natural language or code. The most capable and cost effective model in the GPT-3.5 family is GPT-3. The `5 Turbo` option, which is optimized for chat, works well for traditional completion tasks as well. The `gpt-35-turbo-0125` model supports up to 16,385 input tokens and 4,096 output tokens.
- gpt-4 family: `gpt-4-32k` is supported.
- text-embedding-ada-002 (version 2), embedding model that can be used with embedding API requests. The maximum accepted request token is 8,191, and the returned vector has dimensions of 1,536.

Text Analytics ↗

[REST API](#), [SynapseML](#)

- Language detection: detects language of the input text
- Sentiment analysis: returns a score between 0 and 1, to indicate the sentiment in the input text
- Key phrase extraction: identifies the key talking points in the input text
- Personally Identifiable Information(PII) entity recognition: identify, categorize, and redact sensitive information in the input text
- Named entity recognition: identifies known entities and general named entities in the input text
- Entity linking: identifies and disambiguates the identity of entities found in text

Azure AI Translator ↗

[REST API](#), [SynapseML](#)

- Translate: Translates text
- Transliterate: Converts text in one language, in one script, to another script.

Available regions

Available regions for Azure OpenAI Service

For the list of Azure regions where prebuilt AI services in Fabric are now available, visit the [Available regions](#) section of the [Overview of Copilot in Fabric and Power BI \(preview\)](#) article.

Available regions for Text Analytics and Azure AI Translator

Prebuilt [Text Analytics](#) and the [Azure AI Translator](#) in Fabric are now available for public preview in the Azure regions listed in this article. If you don't find your Microsoft Fabric home region in this article, you can still create a Microsoft Fabric capacity in a supported region. For more information, visit [Buy a Microsoft Fabric subscription](#). To determine your Fabric home region, visit [Find your Fabric home region](#).

[] [Expand table](#)

Asia Pacific	Europe	Americas	Middle East and Africa
Australia East	North Europe	Brazil South	South Africa North
Australia Southeast	West Europe	Canada Central	UAE North
Central Indian	France Central	Canada East	
East Asia	Norway East	East US	
Japan East	Switzerland North	East US 2	
Korea Central	Switzerland West	North Central US	
Southeast Asia	UK South	South Central US	
South India	UK West	West US	
		West US 2	
		West US 3	

Consumption rate

! **Note**

The billing for prebuilt AI services in Fabric became effective on November 1st, 2024, as part of your existing Power BI Premium or Fabric Capacity.

A request for prebuilt AI services consumes Fabric Capacity Units. This table defines how many capacity units (CU) are consumed when an AI service is used.

Consumption rate for OpenAI language models

[\[+\] Expand table](#)

Models	Context	Input (Per 1,000 Tokens)	Output (Per 1,000 Tokens)
GPT-4o-2024-08-06 Global Deployment	128 K	84.03 CU seconds	336.13 CU seconds
GPT-4	32 K	2,016.81 CU seconds	4,033.61 CU seconds
GPT-3.5-Turbo-0125	16K	16.81 CU seconds	50.42 CU seconds

Consumption rate for OpenAI embedding models

[\[+\] Expand table](#)

Models	Operation Unit of Measure	Consumption rate
text-embedding-ada-002	1,000 Tokens	3.36 CU seconds

Consumption rate for Text Analytics

[\[+\] Expand table](#)

Operation	Operation Unit of Measure	Consumption rate
Language Detection	1,000 text records	33,613.45 CU seconds
Sentiment Analysis	1,000 text records	33,613.45 CU seconds
Key Phrase Extraction	1,000 text records	33,613.45 CU seconds

Operation	Operation Unit of Measure	Consumption rate
Personally Identifying Information Entity Recognition	1,000 text records	33,613.45 CU seconds
Named Entity Recognition	1,000 text records	33,613.45 CU seconds
Entity Linking	1,000 text records	33,613.45 CU seconds
Summarization	1,000 text records	67,226.89 CU seconds

Consumption rate for Text Translator

[\[+\] Expand table](#)

Operation	Operation Unit of Measure	Consumption rate
Translate	1M Characters	336,134.45 CU seconds
Transliterate	1M Characters	336,134.45 CU seconds

Changes to AI services in Fabric consumption rate

Consumption rates are subject to change at any time. Microsoft uses reasonable efforts to provide notice via email or through in-product notification. Changes shall be effective on the date stated in the Microsoft Release Notes or the Microsoft Fabric Blog. If any change to a AI service in Fabric Consumption Rate materially increases the Capacity Units (CU) required to use, customers can use the cancellation options available for the chosen payment method.

Monitor the Usage

The workload meter associated with the task determines the charges for prebuilt AI services in Fabric. For example, if AI service usage is derived from a Spark workload, the AI usage is grouped together and billed under the Spark billing meter on [Fabric Capacity Metrics app](#).

Example

An online shop owner uses SynapseML and Spark to categorize millions of products into relevant categories. Currently, the shop owner applies hard-coded logic to clean and map the raw "product type" to categories. However, the owner plans to switch to use of the new native Fabric OpenAI LLM (Large Language Model) endpoints. This iteratively processes the data against an LLM for each row, and then categorizes the products based on their "product name," "description," "technical details," and so on.

The expected cost for Spark usage is 1000 CUs. The expected cost for OpenAI usage is about 300 CUs.

To test the new logic, first iterate it in a Spark notebook interactive run. For the operation name of the run, use "Notebook Interactive Run." The owner expects to see an all-up usage of 1300 CUs under "Notebook Interactive Run," with the Spark billing meter accounting for the entire usage.

Once the shop owner validates the logic, the owner sets up the regular run and expects to see an all-up usage of 1300 CUs under the operation name "Spark Job Scheduled Run," with the Spark billing meter accounting for the entire usage.

Related content

- [Use prebuilt Azure OpenAI in Fabric](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Use Azure OpenAI in Fabric with REST API (preview)

Article • 02/14/2025

ⓘ Important

This feature is in [preview](#).

This document shows examples of how to use Azure OpenAI in Fabric using REST API.

Chat

ChatGPT and GPT-4 are language models optimized for conversational interfaces. To access Azure OpenAI chat endpoints in Fabric, you can send an API request using the following format:

```
POST <prebuilt_AI_base_url>/openai/deployments/<deployment_name>/chat/completions?  
api-version=2024-02-01
```

`deployment_name` could be one of:

- `gpt-35-turbo-0125`
- `gpt-4-32k`

Initialization

You can initialize the process by providing some necessary parameters. These parameters include the `Capacity ID`, `Workspace ID`, and User `MWC Token`, which can be obtained using the `TokenUtils` to create the base endpoint for Azure OpenAI models.

Python

```
from synapse.ml.mlflow import get_mlflow_env_config  
from synapse.ml.fabric.token_utils import TokenUtils  
  
mlflow_env_configs = get_mlflow_env_config()  
mwc_token = TokenUtils().get_openai_mwc_token()  
  
prebuilt_AI_base_url = mlflow_env_configs.workload_endpoint +  
    "cognitive/openai/"  
print("workload endpoint for OpenAI: \n" + prebuilt_AI_base_url)
```

```

deployment_name = "gpt-35-turbo-0125" # deployment_id could be one of {gpt-35-turbo-0125 or gpt-4-32k}
openai_url = prebuilt_AI_base_url +
f"openai/deployments/{deployment_name}/chat/completions?api-version=2024-02-01"
print("The full uri of ChatGPT is: ", openai_url)

post_headers = {
    "Content-Type" : "application/json",
    "Authorization" : "MwcToken {}".format(mwc_token)
}

```

Python

```

import requests

def printresult(openai_url:str, response_code:int, messages:list,
result:str):

    print("====")
    print("====")
    print(" | Post URI      | ", openai_url)
    print("-----")
    print(" | Response Status | ", response_code)
    print("-----")
    print(" | OpenAI Input    | \n")
    for msg in messages:
        if msg["role"] == "system":
            print("[System] ", msg["content"])
        elif msg["role"] == "user":
            print("Q: ", msg["content"])
        else:
            print("A: ", msg["content"])
    print("-----")
    print(" | OpenAI Output    | \n", result)

    print("====")
    print("====")

def ChatGPTRequest(system_msg:str, user_msg_box:list, bot_msg_box:list) ->
(int, dict, str):
    # change message type from string to dict
    system_msg = {"role":"system", "content":system_msg}
    user_msg_box = list(map(lambda x : {"role":"user", "content":x}, user_msg_box))
    bot_msg_box = list(map(lambda x : {"role":"assistant", "content":x}, bot_msg_box))

    # cross merge two lists

```

```

msgs = [msg for msgs in zip(user_msg_box, bot_msg_box) for msg in msgs]
if len(user_msg_box) > len(bot_msg_box):
    msgs.extend(user_msg_box[len(bot_msg_box):])
elif len(user_msg_box) < len(bot_msg_box):
    msgs.extend(bot_msg_box[len(user_msg_box):])

# add system msg in front of message box
msgs.insert(0, system_msg)

# request ChatGPT and analysis response
post_body = { "messages" : msgs }
response = requests.post(openai_url, headers=post_headers,
json=post_body)
if response.status_code == 200:
    result = response.json()["choices"][0]["message"]["content"]
else:
    result = response.content
return response.status_code, post_body, result

```

AI Assistant

OpenAI input:

Copilot prompt

[System] You are an AI assistant that helps people find information.
Q: Does Azure OpenAI support customer managed keys?

OpenAI output:

Console

A: Yes, Azure OpenAI supports customer managed keys. It enables you to use your own encryption keys to protect your data at rest in the OpenAI GPT-3 service. With customer managed keys, you have full control over the keys used to encrypt and decrypt your data, providing an additional layer of security.

Python

```

system_message = "You are an AI assistant that helps people find
information."
user_message_box = ["Does Azure OpenAI support customer managed keys?"]
bot_message_box = []

status_code, post_body, result = ChatGPTRequest(system_message,
user_message_box, bot_message_box)
printresult(openai_url, status_code, post_body[ "messages" ], result)

```

IT Architect helper

OpenAI input:

Copilot prompt

[System] You are an IT Architect. I will provide some details about the functionality of an application or other digital product, and it will be your job to come up with ways to integrate it into the IT landscape. This could involve analyzing business requirements, performing a gap analysis and mapping the functionality of the new system to the existing IT landscape. Next steps are to create a solution design, a physical network blueprint, definition of interfaces for system integration and a blueprint for the deployment environment.

A: I need help to integrate a CMS system.

OpenAI output:

Console

A: Sure, I can help you integrate a CMS system. Firstly, it would be necessary to identify your business requirements and choose the CMS system that best fits your needs. Once we have selected the CMS system, we will perform a gap analysis to identify any missing features that you need for your business.

Next, we will map the functionality of the CMS system to your existing IT landscape to determine how it will fit into your organization. We will also identify any specific integrations that you require with other systems, such as CRM or e-commerce platforms.

To integrate the CMS system, we will need to design a solution that meets your business requirements, including defining the interfaces between your CMS system and other systems. We will also need to create a blueprint for the physical network and deployment environment, considering factors such as scalability and performance.

Once we have a comprehensive design and blueprint in place, we can begin the deployment of the CMS system. We will work with your team to configure the CMS system, install any necessary plugins or extensions, and perform any required testing or migration of your existing content.

Finally, we will provide training and support for your organization to help you get the most out of your new CMS system.

Python

```
system_message = "You are an IT Architect. I will provide some details about the functionality of an application or other digital product, and it will be your job to come up with ways to integrate it into the IT landscape. This could involve analyzing business requirements, performing a gap analysis and mapping the functionality of the new system to the existing IT landscape. Next steps are to create a solution design, a physical network blueprint, definition of interfaces for system integration and a blueprint for the
```

```
deployment environment."  
user_message_box = ["I need help to integrate a CMS system."  
bot_message_box = []  
  
status_code, post_body, result = ChatGPTRequest(system_message,  
user_message_box, bot_message_box)  
printresult(openai_url, status_code, post_body["messages"], result)
```

Shakespeare Writing Assistant

OpenAI input:

Copilot prompt

[System] You are a Shakespearean writing assistant who speaks in a Shakespearean style. You help people come up with creative ideas and content like stories, poems, and songs that use Shakespearean style of writing style, including words like "thou" and "hath".

Here are some example of Shakespeare's style:

- Romeo, Romeo! Wherefore art thou Romeo?
- Love looks not with the eyes, but with the mind; and therefore is winged Cupid painted blind.
- Shall I compare thee to a summer's day? Thou art more lovely and more temperate.

Q: Please write a short text turning down an invitation to dinner.

A: Dearest,

Regretfully, I must decline thy invitation.

Prior engagements call me hence. Apologies.

Q: Add more details.

OpenAI output:

Console

A: Fairly well,
Thy gracious invitation to dine hath filled my heart with gladness.
Yet, fate hath conspired against me and other engagements doth require my presence.

Pray, accept my regrets and know that I am nigh as sad as thou must be. Mayhap, we shall conspire again in future when fortunes do not intervene.

With deepest respect,

Python

```
system_message = "You are a Shakespearean writing assistant who speaks in a  
Shakespearean style. You help people come up with creative ideas and content  
like stories, poems, and songs that use Shakespearean style of writing
```

```

style, including words like \"thou\" and \"hath\".\nHere are some example of
Shakespeare's style:\n - Romeo, Romeo! Wherefore art thou Romeo?\n - Love
looks not with the eyes, but with the mind; and therefore is winged Cupid
painted blind.\n - Shall I compare thee to a summer's day? Thou art more
lovely and more temperate."
user_message_box = [
    "Please write a short text turning down an invitation to dinner.",
    "Add more details."
]
bot_message_box = [
    "Dearest,\nRegretfully, I must decline thy invitation.\nPrior
engagements call me hence. Apologies."
]

status_code, post_body, result = ChatGPTRequest(system_message,
user_message_box, bot_message_box)
printresult(openai_url, status_code, post_body[ "messages" ], result)

```

Xbox Customer Support Agent

OpenAI input:

Copilot prompt

[System] You are an Xbox customer support agent whose primary goal is to help users with issues they are experiencing with their Xbox devices. You are friendly and concise. You only provide factual answers to queries, and do not provide answers that are not related to Xbox.

Q: How much is a PS5?

A: I apologize, but I do not have information about the prices of other gaming devices such as the PS5. My primary focus is to assist with issues regarding Xbox devices. Is there a specific issue you are having with your Xbox device that I may be able to help with?

Q: How can I get XGP tokens.

OpenAI output:

Console

A: I apologize, but XGP or Xbox Game Pass tokens are not a recognized form of currency or reward within the Xbox Game Pass program. Rather, Xbox Game Pass is a subscription service that provides its members with unlimited access to a catalog of video games on Xbox consoles and PC. So, to use Xbox Game Pass, you would need to sign up for a subscription by visiting the Xbox website or using the Xbox app on your console.

Python

```
system_message = "You are an Xbox customer support agent whose primary goal  
is to help users with issues they are experiencing with their Xbox devices.  
You are friendly and concise. You only provide factual answers to queries,  
and do not provide answers that are not related to Xbox."  
  
user_message_box = [  
    "How much is a PS5?",  
    "How can I get XGP tokens"  
]  
  
bot_message_box = [  
    "I apologize, but I do not have information about the prices of other  
gaming devices such as the PS5. My primary focus is to assist with issues  
regarding Xbox devices. Is there a specific issue you are having with your  
Xbox device that I may be able to help with?"  
]  
  
status_code, post_body, result = ChatGPTRequest(system_message,  
user_message_box, bot_message_box)  
printresult(openai_url, status_code, post_body["messages"], result)
```

Xbox Customer Support Agent

OpenAI input:

Copilot prompt

[System] You are a marketing writing assistant. You help come up with creative content ideas and content like marketing emails, blog posts, tweets, ad copy and product descriptions. You write in a friendly yet professional tone but can tailor your writing style that best works for a user-specified audience. If you do not know the answer to a question, respond by saying "I do not know the answer to your question."

Q: Help me promote the Windows system.

OpenAI output:

Console

A: Sure, here are a few creative content ideas to promote Windows system:

1. Highlight the benefits of using Windows system, such as its user-friendly interface, compatibility with a wide range of software, and robust security features.

E.g., "Discover the Power of Windows - Your Ultimate Companion for Seamless Computing Experience."

2. Share customer success stories and how they benefited from Windows solutions. For instance, some Windows users can share how it has enabled them to achieve their professional goals by increasing productivity.

E.g., "Windows Changed My Life - The Story of a Productivity Ninja."

3. Offer step-by-step guides and tutorials to help users better understand Windows features and make the most of the system.

E.g., "Mastering Windows - How to Customize and Personalize Your Operating System the Right Way."

4. Create short, visually appealing videos or graphics that showcase the unique features of Windows or compare it with other operating systems in a fun and friendly way.

E.g., "Windows vs. Mac - The Ultimate Showdown."

5. Promote and run special offers, deals, and discounts on Windows products to incentivize new customers to try or upgrade to Windows system.

E.g., "Get 50% Off on Windows 10 Pro - Limited Time Only."

I hope these ideas help you promote the Windows system. If you have any more questions, feel free to ask.

Python

```
system_message = '"You are a marketing writing assistant. You help come up with creative content ideas and content like marketing emails, blog posts, tweets, ad copy and product descriptions. You write in a friendly yet professional tone but can tailor your writing style that best works for a user-specified audience. If you do not know the answer to a question, respond by saying "I do not know the answer to your question."'
user_message_box = ["Help me promote the Windows system."]
bot_message_box = []

status_code, post_body, result = ChatGPTRequest(system_message,
user_message_box, bot_message_box)
printresult(openai_url, status_code, post_body[ "messages" ], result)
```

Embeddings

An embedding is a special data representation format that machine learning models and algorithms can easily utilize. It contains information-rich semantic meaning of a text, represented by a vector of floating point numbers. The distance between two embeddings in the vector space is related to the semantic similarity between two original inputs. For example, if two texts are similar, their vector representations should also be similar.

To access Azure OpenAI embeddings endpoint in Fabric, you can send an API request using the following format:

```
POST <url_prefix>/openai/deployments/<deployment_name>/embeddings?api-version=2024-
```

```
02-01
```

```
deployment_name could be text-embedding-ada-002.
```

Initialization

```
Python
```

```
from synapse.ml.mlflow import get_mlflow_env_config
from synapse.ml.fabric.token_utils import TokenUtils

mlflow_env_configs = get_mlflow_env_config()
mwc_token = TokenUtils().get_openai_mwc_token()

prebuilt_AI_base_url = mlflow_env_configs.workload_endpoint +
"cognitive/openai/"
print("workload endpoint for OpenAI: \n" + prebuilt_AI_base_url)

deployment_name = "text-embedding-ada-002"
openai_url = prebuilt_AI_base_url +
f"openai/deployments/{deployment_name}/embeddings?api-version=2024-02-01"
print("The full uri of Embeddings is: ", openai_url)

post_headers = {
    "Content-Type" : "application/json",
    "Authorization" : "MwcToken {}".format(mwc_token)
}

post_body = {
    "input": "empty prompt, need to fill in the content before the request",
}
```

```
Python
```

```
import json
import uuid
import requests
from pprint import pprint

def printresult(openai_url:str, response_code:int, prompt:str, result:str):
    print("====")
    print("====")
    print("| Post URI |", openai_url)
    print("-----")
    print("-----")
    print("| Response Status |", response_code)
```

```
print("-----")
print("| OpenAI Input    | \n", prompt)
print("-----")
print("| OpenAI Output   | \n", result)

print("=====")
```

Get Embeddings

OpenAI input:

Copilot prompt

John is good boy.

OpenAI output:

Console

```
[-0.0045386623, 0.0031397594, ..., 0.0006536394, -0.037461143, -0.033455864]
```

Python

```
input_words = "John is good boy."
post_body["input"] = input_words
response = requests.post(url=openai_url, headers=post_headers,
json=post_body)
printresult(openai_url=openai_url, response_code=response.status_code,
prompt=input_words, result=response.content)
```

Related content

- [Use prebuilt Text Analytics in Fabric with REST API](#)
- [Use prebuilt Text Analytics in Fabric with SynapseML](#)
- [Use prebuilt Azure AI Translator in Fabric with REST API](#)
- [Use prebuilt Azure AI Translator in Fabric with SynapseML](#)
- [Use prebuilt Azure OpenAI in Fabric with Python SDK](#)
- [Use prebuilt Azure OpenAI in Fabric with SynapseML](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Use Azure OpenAI in Fabric with Python SDK and Synapse ML (preview)

Article • 10/14/2024

ⓘ Important

This feature is in [preview](#).

This article shows examples of how to use Azure OpenAI in Fabric using [OpenAI Python SDK](#) and using SynapseML.

Prerequisites

Python SDK <1.0.0

[OpenAI Python SDK](#) isn't installed in default runtime, you need to first install it.

Python

```
%pip install openai==0.28.1
```

Chat

Python SDK <1.0.0

ChatGPT and GPT-4 are language models optimized for conversational interfaces.

The example presented here showcases simple chat completion operations and isn't intended to serve as a tutorial.

Python

```
import openai

response = openai.ChatCompletion.create(
    deployment_id='gpt-35-turbo-0125', # deployment_id could be one of
    {gpt-35-turbo-0125 or gpt-4-32k}
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Knock knock."},
```

```

        {"role": "assistant", "content": "Who's there?"},
        {"role": "user", "content": "Orange."},
    ],
    temperature=0,
)

print(f"{response.choices[0].message.role}:
{response.choices[0].message.content}")

```

Output

JSON

```
assistant: Orange who?
```

We can also stream the response

Python

```

response = openai.ChatCompletion.create(
    deployment_id='gpt-35-turbo-0125', # deployment_id could be one of
    {gpt-35-turbo-0125 or gpt-4-32k}
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Knock knock."},
        {"role": "assistant", "content": "Who's there?"},
        {"role": "user", "content": "Orange."},
    ],
    temperature=0,
    stream=True
)

for chunk in response:
    delta = chunk.choices[0].delta

    if "role" in delta.keys():
        print(delta.role + ": ", end="", flush=True)
    if "content" in delta.keys():
        print(delta.content, end="", flush=True)

```

Output

JSON

```
assistant: Orange who?
```

Embeddings

Python SDK <1.0.0

An embedding is a special data representation format that machine learning models and algorithms can easily utilize. It contains information-rich semantic meaning of a text, represented by a vector of floating point numbers. The distance between two embeddings in the vector space is related to the semantic similarity between two original inputs. For example, if two texts are similar, their vector representations should also be similar.

The example demonstrated here showcases how to obtain embeddings and isn't intended as a tutorial.

Python

```
deployment_id = "text-embedding-ada-002" # set deployment_name as text-embedding-ada-002
embeddings = openai.Embedding.create(deployment_id=deployment_id,
                                      input="The food was delicious and
the waiter...")
print(embeddings)
```

Output

```
{
  "object": "list",
  "data": [
    {
      "object": "embedding",
      "index": 0,
      "embedding": [
        0.002306425478309393,
        -0.009327292442321777,
        0.015797346830368042,
        ...
        0.014552861452102661,
        0.010463837534189224,
        -0.015327490866184235,
        -0.01937841810286045,
        -0.0028842221945524216
      ]
    }
  ],
}
```

```
"model": "ada",
"usage": {
    "prompt_tokens": 8,
    "total_tokens": 8
}
}
```

Related content

- [Use prebuilt Text Analytics in Fabric with REST API](#)
- [Use prebuilt Text Analytics in Fabric with SynapseML](#)
- [Use prebuilt Azure AI Translator in Fabric with REST API](#)
- [Use prebuilt Azure AI Translator in Fabric with SynapseML](#)
- [Use prebuilt Azure OpenAI in Fabric with REST API](#)
- [Use prebuilt Azure OpenAI in Fabric with SynapseML and Python SDK](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Use prebuilt Text Analytics in Fabric with REST API and SynapseML (preview)

Article • 11/15/2023

ⓘ Important

This feature is in **preview**.

Text Analytics is an [Azure AI services](#) that enables you to perform text mining and text analysis with Natural Language Processing (NLP) features.

This tutorial demonstrates using text analytics in Fabric with RESTful API to:

- Detect sentiment labels at the sentence or document level.
- Identify the language for a given text input.
- Extract key phrases from a text.
- Identify different entities in text and categorize them into predefined classes or types.

Prerequisites

Rest API

Python

```
# Get workload endpoints and access token

from synapse.ml.mlflow import get_mlflow_env_config
import json

mlflow_env_configs = get_mlflow_env_config()
access_token = access_token = mlflow_env_configs.driver_aad_token
prebuilt_AI_base_host = mlflow_env_configs.workload_endpoint +
"cognitive/textanalytics/"
print("Workload endpoint for AI service: \n" + prebuilt_AI_base_host)

service_url = prebuilt_AI_base_host + "language/:analyze-text?api-
version=2022-05-01"

# Make a RESful request to AI service

post_headers = {
    "Content-Type" : "application/json",
```

```

        "Authorization" : "Bearer {}".format(access_token)
    }

def printresponse(response):
    print(f"HTTP {response.status_code}")
    if response.status_code == 200:
        try:
            result = response.json()
            print(json.dumps(result, indent=2, ensure_ascii=False))
        except:
            print(f"parse error {response.content}")
    else:
        print(response.headers)
        print(f"error message: {response.content}")

```

Sentiment analysis

Rest API

The Sentiment Analysis feature provides a way for detecting the sentiment labels (such as "negative", "neutral" and "positive") and confidence scores at the sentence and document-level. This feature also returns confidence scores between 0 and 1 for each document and sentences within it for positive, neutral and negative sentiment. See the [Sentiment Analysis and Opinion Mining language support](#) for the list of enabled languages.

Python

```

import requests
from pprint import pprint
import uuid

post_body = {
    "kind": "SentimentAnalysis",
    "parameters": {
        "modelVersion": "latest",
        "opinionMining": "True"
    },
    "analysisInput": {
        "documents": [
            {
                "id": "1",
                "language": "en",
                "text": "The food and service were unacceptable. The concierge was nice, however."
            }
        ]
    }
}

```

```
}

post_headers["x-ms-workload-resource-moniker"] = str(uuid.uuid1())
response = requests.post(service_url, json=post_body,
headers=post_headers)

# Output all information of the request process
printresponse(response)
```

Output

```
HTTP 200
{
  "kind": "SentimentAnalysisResults",
  "results": [
    "documents": [
      {
        "id": "1",
        "sentiment": "mixed",
        "confidenceScores": {
          "positive": 0.43,
          "neutral": 0.04,
          "negative": 0.53
        },
        "sentences": [
          {
            "sentiment": "negative",
            "confidenceScores": {
              "positive": 0.0,
              "neutral": 0.01,
              "negative": 0.99
            },
            "offset": 0,
            "length": 40,
            "text": "The food and service were unacceptable. ",
            "targets": [
              {
                "sentiment": "negative",
                "confidenceScores": {
                  "positive": 0.01,
                  "negative": 0.99
                },
                "offset": 4,
                "length": 4,
                "text": "food",
                "relations": [
                  {
                    "relationType": "assessment",
                    "ref": "#/documents/0/sentences/0/assessments/0"
                  }
                ]
              }
            ]
          }
        ]
      }
    ]
  ]
}
```

```
        }
    ],
},
{
  "sentiment": "negative",
  "confidenceScores": {
    "positive": 0.01,
    "negative": 0.99
  },
  "offset": 13,
  "length": 7,
  "text": "service",
  "relations": [
    {
      "relationType": "assessment",
      "ref": "#/documents/0/sentences/0/assessments/0"
    }
  ]
},
],
"assessments": [
  {
    "sentiment": "negative",
    "confidenceScores": {
      "positive": 0.01,
      "negative": 0.99
    },
    "offset": 26,
    "length": 12,
    "text": "unacceptable",
    "isNegated": false
  }
]
},
{
  "sentiment": "positive",
  "confidenceScores": {
    "positive": 0.86,
    "neutral": 0.08,
    "negative": 0.07
  },
  "offset": 40,
  "length": 32,
  "text": "The concierge was nice, however.",
  "targets": [
    {
      "sentiment": "positive",
      "confidenceScores": {
        "positive": 1.0,
        "negative": 0.0
      },
      "offset": 44,
      "length": 9,
      "text": "concierge",
      "relations": [

```

```
        {
          "relationType": "assessment",
          "ref": "#/documents/0/sentences/1/assessments/0"
        }
      ]
    },
    "assessments": [
      {
        "sentiment": "positive",
        "confidenceScores": {
          "positive": 1.0,
          "negative": 0.0
        },
        "offset": 58,
        "length": 4,
        "text": "nice",
        "isNegated": false
      }
    ]
  },
  "warnings": []
},
],
"errors": [],
"modelVersion": "2022-11-01"
}
}
```

Language detector

Rest API

The Language Detector evaluates text input for each document and returns language identifiers with a score that indicates the strength of the analysis. This capability is useful for content stores that collect arbitrary text, where language is unknown. See the [Supported languages for language detection](#) for the list of enabled languages.

Python

```
post_body = {
  "kind": "LanguageDetection",
  "parameters": {
    "modelVersion": "latest"
  },
  "analysisInput":{
```

```

    "documents": [
        {
            "id": "1",
            "text": "This is a document written in English."
        }
    ]
}

post_headers["x-ms-workload-resource-moniker"] = str(uuid.uuid1())
response = requests.post(service_url, json=post_body,
headers=post_headers)

# Output all information of the request process
printresponse(response)

```

Output

```

HTTP 200
{
    "kind": "LanguageDetectionResults",
    "results": [
        "documents": [
            {
                "id": "1",
                "detectedLanguage": {
                    "name": "English",
                    "iso6391Name": "en",
                    "confidenceScore": 0.99
                },
                "warnings": []
            }
        ],
        "errors": [],
        "modelVersion": "2022-10-01"
    }
}

```

Key Phrase Extractor

Rest API

The Key Phrase Extraction evaluates unstructured text and returns a list of key phrases. This capability is useful if you need to quickly identify the main points in a

collection of documents. See the [Supported languages for key phrase extraction](#) for the list of enabled languages.

Python

```
post_body = {
    "kind": "KeyPhraseExtraction",
    "parameters": {
        "modelVersion": "latest"
    },
    "analysisInput": {
        "documents": [
            {
                "id": "1",
                "language": "en",
                "text": "Dr. Smith has a very modern medical office, and
she has great staff."
            }
        ]
    }
}

post_headers["x-ms-workload-resource-moniker"] = str(uuid.uuid1())
response = requests.post(service_url, json=post_body,
headers=post_headers)

# Output all information of the request process
printresponse(response)
```

Output

```
HTTP 200
{
    "kind": "KeyPhraseExtractionResults",
    "results": [
        "documents": [
            {
                "id": "1",
                "keyPhrases": [
                    "modern medical office",
                    "Dr. Smith",
                    "great staff"
                ],
                "warnings": []
            }
        ],
        "errors": [],
        "modelVersion": "2022-10-01"
    ]
}
```

```
    }
}
```

Named Entity Recognition (NER)

Rest API

Named Entity Recognition (NER) is the ability to identify different entities in text and categorize them into predefined classes or types such as: person, location, event, product, and organization. See the [NER language support](#) for the list of enabled languages.

Python

```
post_body = {
    "kind": "EntityRecognition",
    "parameters": {
        "modelVersion": "latest"
    },
    "analysisInput": {
        "documents": [
            {
                "id": "1",
                "language": "en",
                "text": "I had a wonderful trip to Seattle last week."
            }
        ]
    }
}

post_headers["x-ms-workload-resource-moniker"] = str(uuid.uuid1())
response = requests.post(service_url, json=post_body,
headers=post_headers)

# Output all information of the request process
printresponse(response)
```

Output

```
HTTP 200
{
    "kind": "EntityRecognitionResults",
    "results": {
        "documents": [
```

```
{  
    "id": "1",  
    "entities": [  
        {  
            "text": "trip",  
            "category": "Event",  
            "offset": 18,  
            "length": 4,  
            "confidenceScore": 0.74  
        },  
        {  
            "text": "Seattle",  
            "category": "Location",  
            "subcategory": "GPE",  
            "offset": 26,  
            "length": 7,  
            "confidenceScore": 1.0  
        },  
        {  
            "text": "last week",  
            "category": "DateTime",  
            "subcategory": "DateRange",  
            "offset": 34,  
            "length": 9,  
            "confidenceScore": 0.8  
        }  
    ],  
    "warnings": []  
},  
],  
"errors": [],  
"modelVersion": "2021-06-01"  
}  
}
```

Entity linking

Rest API

No steps for REST API in this section.

Next steps

- Use prebuilt Text Analytics in Fabric with SynapseML
- Use prebuilt Azure AI Translator in Fabric with REST API
- Use prebuilt Azure AI Translator in Fabric with SynapseML

- Use prebuilt Azure OpenAI in Fabric with REST API
 - Use prebuilt Azure OpenAI in Fabric with Python SDK
 - Use prebuilt Azure OpenAI in Fabric with SynapseML
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Use prebuilt Azure AI Translator in Fabric with REST API and SynapseML (preview)

Article • 02/10/2025

ⓘ Important

This feature is in [preview](#).

Azure AI Translator is an [Azure AI services](#) that enables you to perform language translation and other language-related operations.

This sample shows use, with RESTful APIs, of the prebuilt Azure AI translator, in Fabric:

- Translate text
- Transliterate text
- Get supported languages

Prerequisites

Rest API

Python

```
# Get workload endpoints and access token

from synapse.ml.mlflow import get_mlflow_env_config
import json

mlflow_env_configs = get_mlflow_env_config()
access_token = access_token = mlflow_env_configs.driver_aad_token
prebuilt_AI_base_host = mlflow_env_configs.workload_endpoint +
"cognitive/texttranslation/"
print("Workload endpoint for AI service: \n" + prebuilt_AI_base_host)

# Make a RESTful request to AI service

post_headers = {
    "Content-Type" : "application/json",
    "Authorization" : "Bearer {}".format(access_token),
}

def printresponse(response):
```

```
print(f"HTTP {response.status_code}")
if response.status_code == 200:
    try:
        result = response.json()
        print(json.dumps(result, indent=2, ensure_ascii=False))
    except:
        print(f"parse error {response.content}")
else:
    print(f"error message: {response.content}")
```

Text Translation

Rest API

Text translation is the core operation of the Translator service.

Python

```
import requests
import uuid

service_url = prebuilt_AI_base_host + "translate?api-version=3.0&to=fr"
post_body = [{'Text':'Hello, friend.'}]

post_headers["x-ms-workload-resource-moniker"] = str(uuid.uuid1())
response = requests.post(service_url, json=post_body,
headers=post_headers)

# Output all information of the request process
print(response)
```

Output

```
HTTP 200
[
  {
    "detectedLanguage": {
      "language": "en",
      "score": 1.0
    },
    "translations": [
      {
        "text": "Bonjour cher ami.",
        "to": "fr"
      }
    ]
  }
]
```

```
    ]
  }
]
```

Text Transliteration

Rest API

Transliteration converts a word or phrase from the script (alphabet) of one language to another, based on phonetic similarity.

Python

```
service_url = prebuilt_AI_base_host + "transliterate?api-version=3.0&language=ja&fromScript=Jpan&toScript=Latn"
post_body = [
    {"Text": "こんにちは"},
    {"Text": "さようなら"}
]

post_headers["x-ms-workload-resource-moniker"] = str(uuid.uuid1())
response = requests.post(service_url, json=post_body,
headers=post_headers)

# Output all information of the request process
printresponse(response)
```

Output

```
HTTP 200
[
  {
    "text": "Kon'nichiwa",
    "script": "Latn"
  },
  {
    "text": "sayonara",
    "script": "Latn"
  }
]
```

Supported Languages Retrieval

Rest API

Returns a list of languages that Translator operations support.

Python

```
service_url = prebuilt_AI_base_host + "languages?api-version=3.0"

post_headers["x-ms-workload-resource-moniker"] = str(uuid.uuid1())
response = requests.get(service_url, headers=post_headers)

# Output all information of the request process
printresponse(response)
```

Related content

- [Use prebuilt Text Analytics in Fabric with REST API](#)
- [Use prebuilt Text Analytics in Fabric with SynapseML](#)
- [Use prebuilt Azure AI Translator in Fabric with SynapseML](#)
- [Use prebuilt Azure OpenAI in Fabric with REST API](#)
- [Use prebuilt Azure OpenAI in Fabric with Python SDK](#)
- [Use prebuilt Azure OpenAI in Fabric with SynapseML](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Use Python for Apache Spark

Article • 04/16/2024

Microsoft Fabric provides built-in Python support for Apache Spark. Support includes [PySpark](#), which allows users to interact with Spark using familiar Spark or Python interfaces.

You can analyze data using Python through Spark batch job definitions or with interactive Fabric notebooks. This article provides an overview of developing Spark applications in Synapse using the Python language.

Create and run notebook sessions

Microsoft Fabric notebook is a web interface for you to create files that contain live code, visualizations, and narrative text. Notebooks are a good place to validate ideas and use quick experiments to get insights from your data. Notebooks are also widely used in data preparation, data visualization, machine learning, and other big data scenarios.

To get started with Python in Microsoft Fabric notebooks, change the primary **Language** at the top of your notebook by setting the language option to *PySpark (Python)*.

```
Python  
%%pyspark  
# Enter your Python code here
```

You can use multiple languages in one notebook by specifying the language magic command at the beginning of a cell.

To learn more about notebooks in Microsoft Fabric Analytics, see [How to use notebooks](#).

Install packages

Libraries provide reusable code that you can include in your programs or projects. To make partner code or locally built code available to your applications, install a library in-line into your notebook session. Alternatively, your workspace administrator can create an environment, install the library in it, and attach the environment as the workspace default in the workspace setting.

To learn more about library management in Microsoft Fabric, see [Manage Apache Spark libraries](#).

Notebook utilities

Microsoft Spark Utilities (MSSparkUtils) is a built-in package to help you easily perform common tasks. You can use MSSparkUtils to work with file systems, to get environment variables, to chain notebooks together, and to work with secrets. MSSparkUtils is supported for PySpark notebooks.

To get started, run the following commands:

Python

```
from notebookutils import mssparkutils  
mssparkutils.notebook.help()
```

For more information about the supported MSSparkUtils commands, see [Use Microsoft Spark Utilities](#).

Use Pandas on Spark

The [Pandas API on Spark](#) allows you to scale your Pandas workload to any size by running it distributed across multiple nodes. If you're already familiar with pandas and want to use Spark for big data, pandas API on Spark makes you immediately productive.

You can migrate your applications without modifying the code. You can have a single codebase that works both with pandas, for tests and smaller datasets, and with Spark, for production and distributed datasets. You can switch between the pandas API and the Pandas API on Spark easily and without overhead.

Python runtime

The Microsoft Fabric [Runtime](#) is a curated environment optimized for data science and machine learning. The Microsoft Fabric runtime offers a range of popular, Python open-source libraries, including libraries like Pandas, PyTorch, scikit-learn, and XGBoost.

Python visualization

The Python ecosystem offers multiple graphing libraries that come with many different features. By default, every Spark instance in Microsoft Fabric contains a set of curated and popular open-source libraries. You can also add or manage other libraries or versions. For more information on library management, see [Summary of library management best practices](#).

To learn more about how to create Python visualizations, see [Python visualization](#).

Related content

- Learn how to use the Pandas API on Apache Spark: [Pandas API on Apache Spark ↗](#)
 - [Manage Apache Spark libraries in Microsoft Fabric](#)
 - Visualize data in Python: [Visualize data in Python](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Analyze data with Apache Spark and Python

Article • 04/16/2024

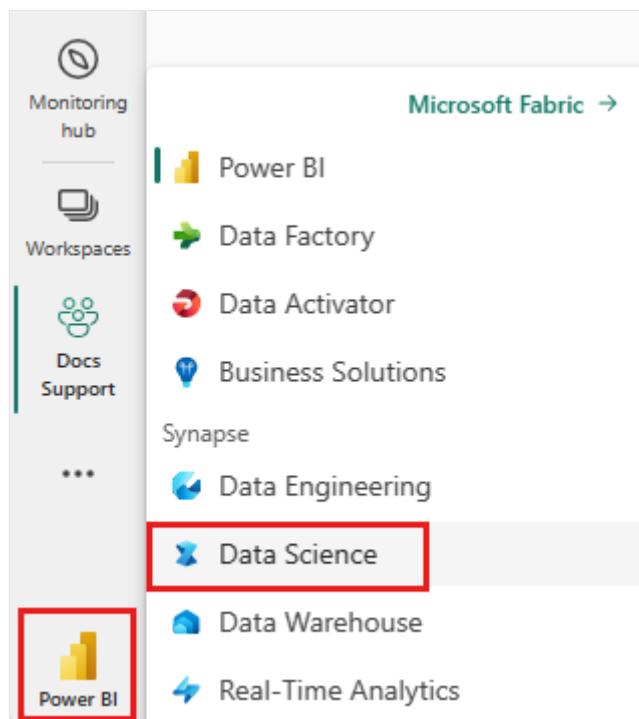
In this article, you learn how to perform exploratory data analysis by using Azure Open Datasets and Apache Spark. This article analyzes the New York City taxi dataset. The data is available through Azure Open Datasets. This subset of the dataset contains information about yellow taxi trips: information about each trip, the start and end time and locations, the cost, and other interesting attributes.

In this article, you:

- ✓ Download and prepare data
- ✓ Analyze data
- ✓ Visualize data

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



Download and prepare the data

To start, download the [New York City \(NYC\) Taxi](#) dataset and prepare the data.

1. Create a notebook by using PySpark. For instructions, see [Create a notebook](#).

➊ Note

Because of the PySpark kernel, you don't need to create any contexts explicitly. The Spark context is automatically created for you when you run the first code cell.

2. In this article, you use several different libraries to help visualize the dataset. To do this analysis, import the following libraries:

Python

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
```

3. Because the raw data is in Parquet format, you can use the Spark context to pull the file into memory as a DataFrame directly. Use the Open Datasets API to retrieve the data and create a Spark DataFrame. To infer the datatypes and schema, use the Spark DataFrame *schema on read* properties.

Python

```
from azureml.opendatasets import NycTlcYellow

end_date = parser.parse('2018-06-06')
start_date = parser.parse('2018-05-01')
nyc_tlc = NycTlcYellow(start_date=start_date, end_date=end_date)
nyc_tlc_pd = nyc_tlc.to_pandas_dataframe()

df = spark.createDataFrame(nyc_tlc_pd)
```

4. After the data is read, do some initial filtering to clean the dataset. You might remove unneeded columns and add columns that extract important information. In addition, you can filter out anomalies within the dataset.

Python

```
# Filter the dataset
from pyspark.sql.functions import *
```

```

filtered_df = df.select('vendorID', 'passengerCount',
    'tripDistance', 'paymentType', 'fareAmount', 'tipAmount' \
        , date_format('tpepPickupDateTime', \
    'hh').alias('hour_of_day')\

    ,
    dayofweek('tpepPickupDateTime').alias('day_of_week')\

    ,
    dayofmonth(col('tpepPickupDateTime')).alias('day_of_month'))\

        .filter((df.passengerCount > 0) \

            & (df.tipAmount >= 0) \

            & (df.fareAmount >= 1) & (df.fareAmount \

            <= 250) \

                & (df.tripDistance > 0) &

            (df.tripDistance <= 200))

filtered_df.createOrReplaceTempView("taxi_dataset")

```

Analyze data

As a data analyst, you have a wide range of tools available to help you extract insights from the data. In this part of the article, learn about a few useful tools available within Microsoft Fabric notebooks. In this analysis, you want to understand the factors that yield higher taxi tips for the selected period.

Apache Spark SQL Magic

First, do exploratory data analysis by using Apache Spark SQL and magic commands with the Microsoft Fabric notebook. After you have the query, visualize the results by using the built-in `chart options` capability.

1. In the notebook, create a new cell and copy the following code. By using this query, you can understand how the average tip amounts change over the period you select. This query also helps you identify other useful insights, including the minimum/maximum tip amount per day and the average fare amount.

SQL

```

%%sql
SELECT
    day_of_month
    , MIN(tipAmount) AS minTipAmount
    , MAX(tipAmount) AS maxTipAmount
    , AVG(tipAmount) AS avgTipAmount
    , AVG(fareAmount) as fareAmount
FROM taxi_dataset

```

```
GROUP BY day_of_month  
ORDER BY day_of_month ASC
```

- After your query finishes running, you can visualize the results by switching to the chart view. This example creates a line chart by specifying the `day_of_month` field as the key and `avgTipAmount` as the value. After you make the selections, select **Apply** to refresh your chart.

Visualize data

In addition to the built-in notebook charting options, you can use popular open-source libraries to create your own visualizations. In the following examples, use Seaborn and Matplotlib, which are commonly used Python libraries for data visualization.

- To make development easier and less expensive, downsample the dataset. Use the built-in Apache Spark sampling capability. In addition, both Seaborn and Matplotlib require a Pandas DataFrame or NumPy array. To get a Pandas DataFrame, use the `toPandas()` command to convert the DataFrame.

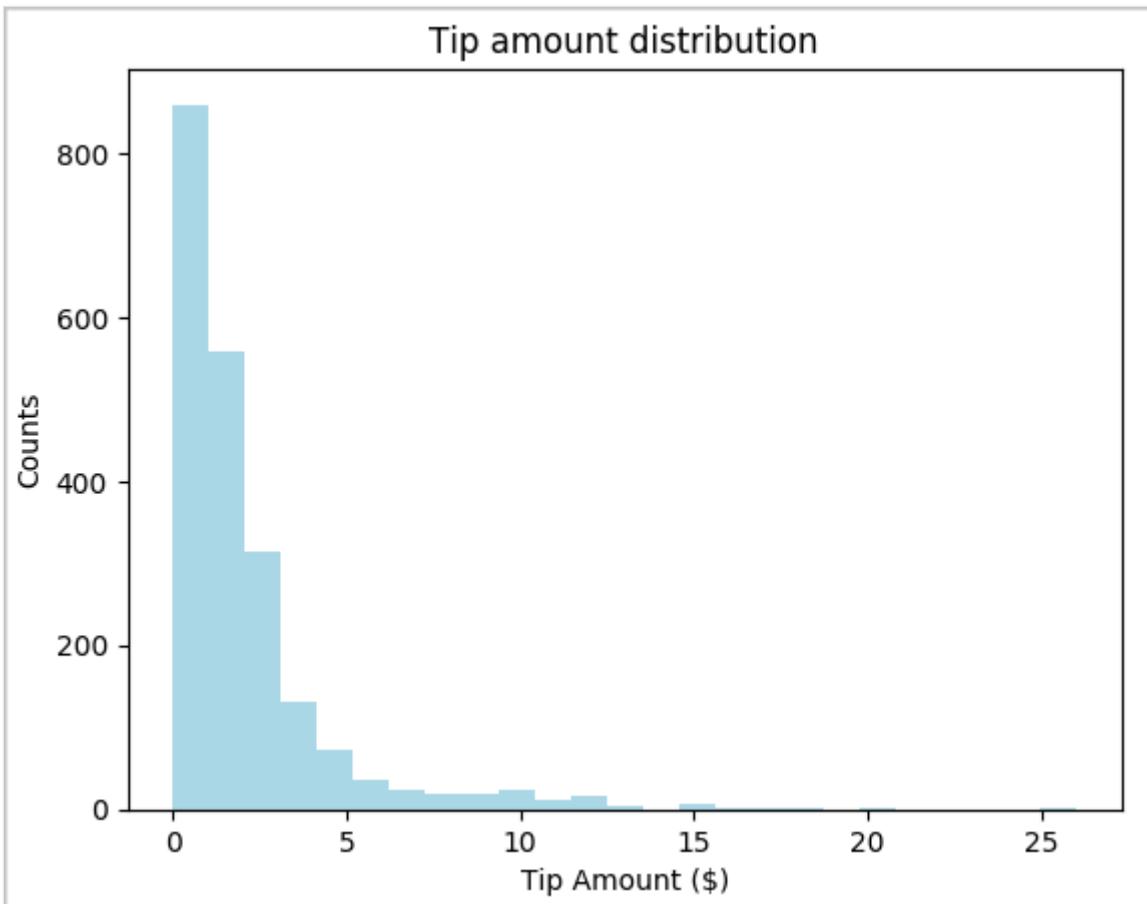
Python

```
# To make development easier, faster, and less expensive, downsample  
# for now  
sampled_taxi_df = filtered_df.sample(True, 0.001, seed=1234)  
  
# The charting package needs a Pandas DataFrame or NumPy array to do  
# the conversion  
sampled_taxi_pd_df = sampled_taxi_df.toPandas()
```

- You can understand the distribution of tips in the dataset. Use Matplotlib to create a histogram that shows the distribution of tip amount and count. Based on the distribution, you can see that tips are skewed toward amounts less than or equal to \$10.

Python

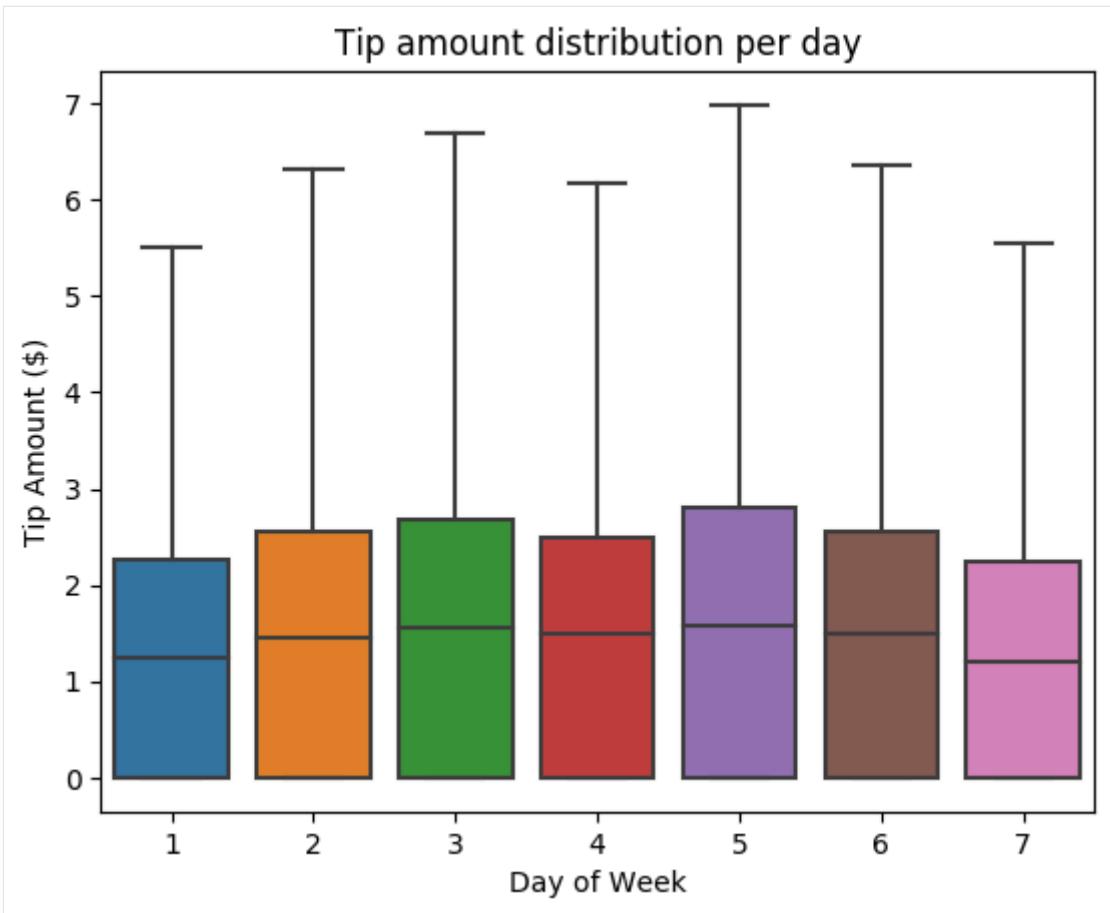
```
# Look at a histogram of tips by count by using Matplotlib  
  
ax1 = sampled_taxi_pd_df['tipAmount'].plot(kind='hist', bins=25,  
facecolor='lightblue')  
ax1.set_title('Tip amount distribution')  
ax1.set_xlabel('Tip Amount ($)')  
ax1.set_ylabel('Counts')  
plt.suptitle('')  
plt.show()
```



3. Next, try to understand the relationship between the tips for a given trip and the day of the week. Use Seaborn to create a box plot that summarizes the trends for each day of the week.

Python

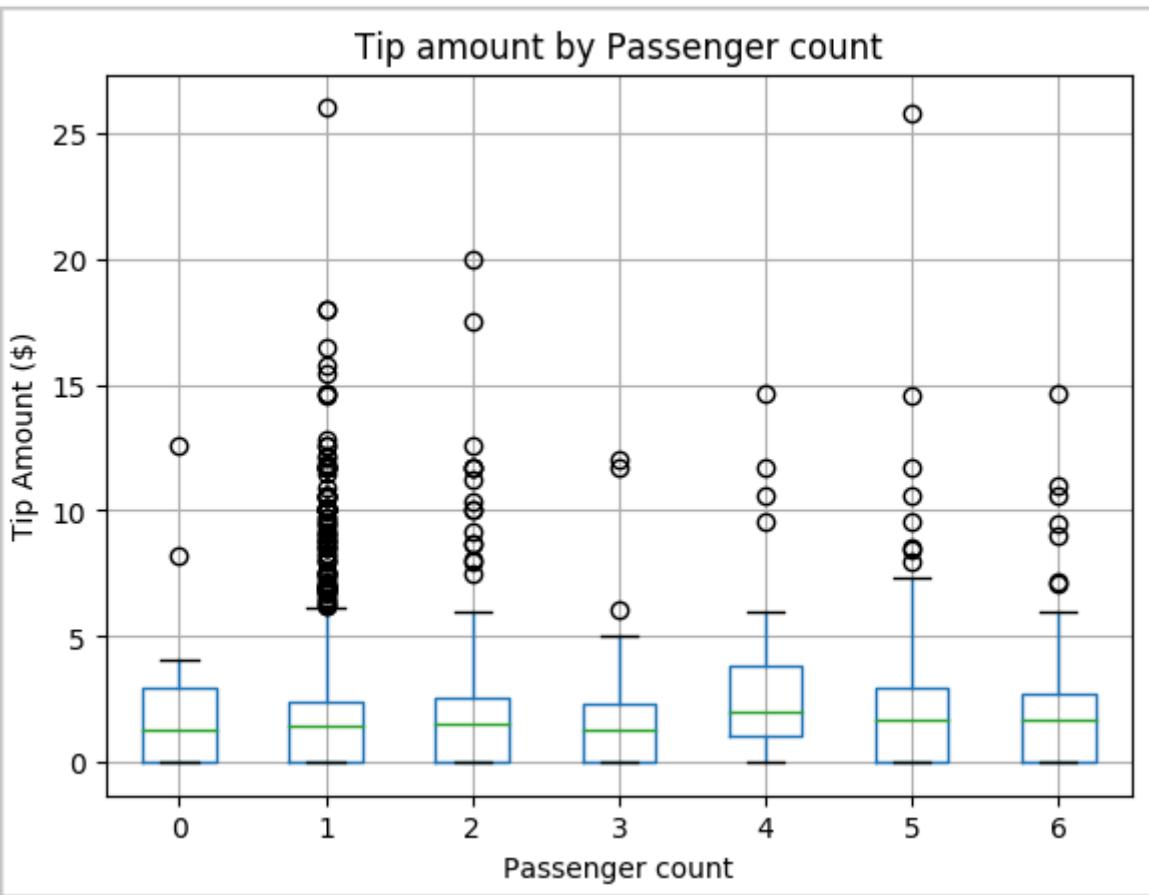
```
# View the distribution of tips by day of week using Seaborn
ax = sns.boxplot(x="day_of_week",
y="tipAmount",data=sampled_taxi_pd_df, showfliers = False)
ax.set_title('Tip amount distribution per day')
ax.set_xlabel('Day of Week')
ax.set_ylabel('Tip Amount ($)')
plt.show()
```



4. Another hypothesis might be that there's a positive relationship between the number of passengers and the total taxi tip amount. To verify this relationship, run the following code to generate a box plot that illustrates the distribution of tips for each passenger count.

Python

```
# How many passengers tipped by various amounts
ax2 = sampled_taxi_pd_df.boxplot(column=['tipAmount'], by=['passengerCount'])
ax2.set_title('Tip amount by Passenger count')
ax2.set_xlabel('Passenger count')
ax2.set_ylabel('Tip Amount ($)')
ax2.set_ylim(0,30)
plt.suptitle('')
plt.show()
```

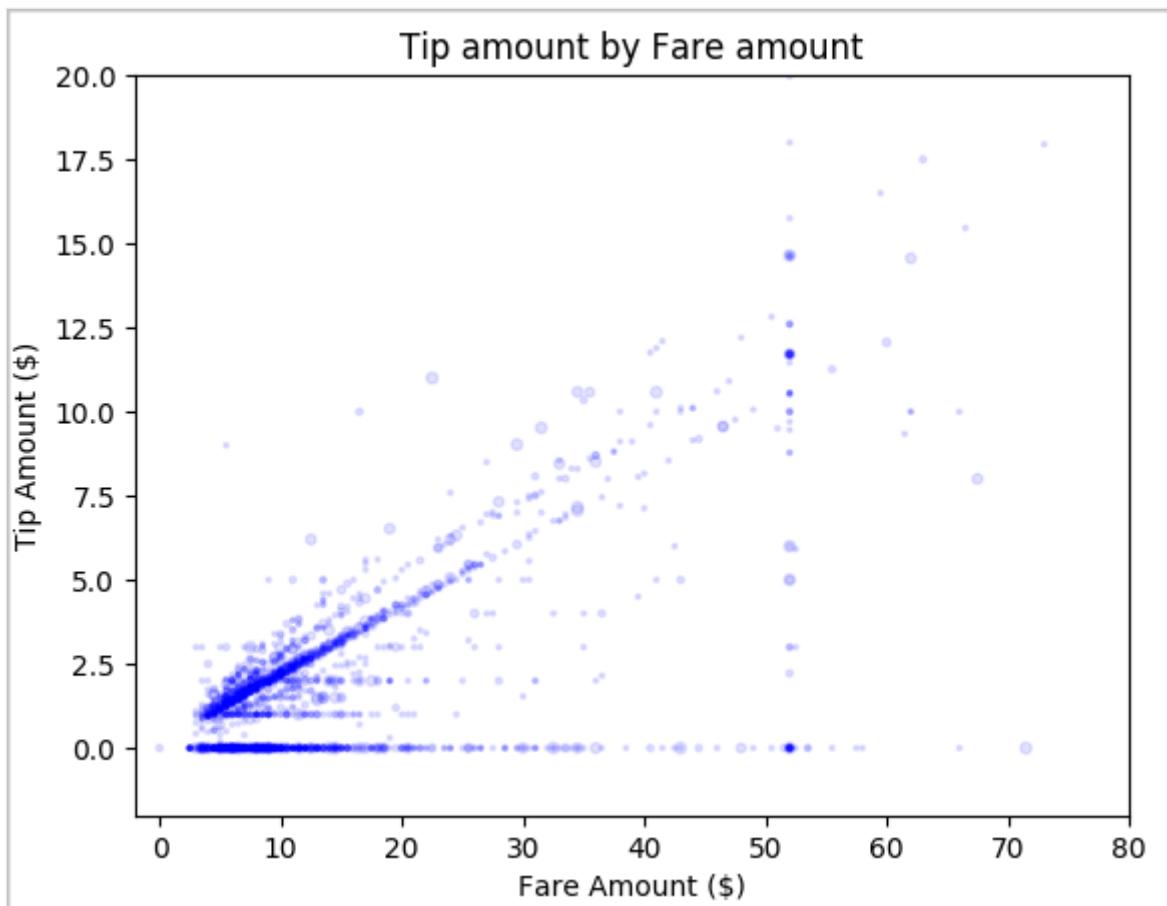


5. Last, explore the relationship between the fare amount and the tip amount. Based on the results, you can see that there are several observations where people don't tip. However, there's a positive relationship between the overall fare and tip amounts.

Python

```
# Look at the relationship between fare and tip amounts

ax = sampled_taxi_pd_df.plot(kind='scatter', x= 'fareAmount', y =
'tipAmount', c='blue', alpha = 0.10, s=2.5*
(sampled_taxi_pd_df[ 'passengerCount']))
ax.set_title('Tip amount by Fare amount')
ax.set_xlabel('Fare Amount ($)')
ax.set_ylabel('Tip Amount ($)')
plt.axis([-2, 80, -2, 20])
plt.suptitle('')
plt.show()
```



Related content

- [Pandas API on Apache Spark ↗](#)
- [Python in-line installation](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Use R for Apache Spark

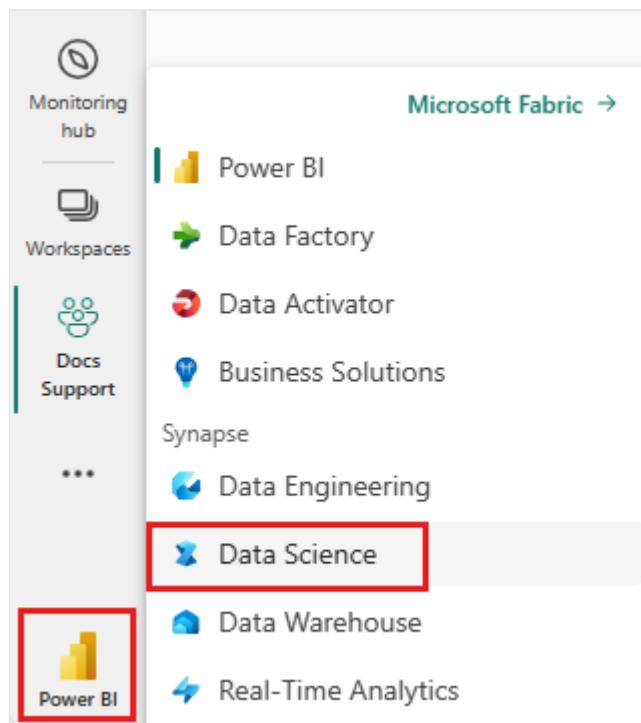
Article • 01/22/2024

Microsoft Fabric provides built-in R support for Apache Spark. This includes support for [SparkR](#) and [sparklyr](#), which allows users to interact with Spark using familiar Spark or R interfaces. You can analyze data using R through Spark batch job definitions or with interactive Microsoft Fabric notebooks.

This document provides an overview of developing Spark applications in Synapse using the R language.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



Create and run notebook sessions

Microsoft Fabric notebook is a web interface for you to create files that contain live code, visualizations, and narrative text. Notebooks are a good place to validate ideas and use quick experiments to get insights from your data. Notebooks are also widely

used in data preparation, data visualization, machine learning, and other big data scenarios.

To get started with R in Microsoft Fabric notebooks, change the primary **language** at the top of your notebook by setting the language option to *SparkR (R)*.

In addition, you can use multiple languages in one notebook by specifying the language magic command at the beginning of a cell.

```
R  
%%sparkr  
# Enter your R code here
```

To learn more about notebooks within Microsoft Fabric Analytics, see [How to use notebooks](#).

Install packages

Libraries provide reusable code that you might want to include in your programs or projects. To make third party or locally built code available to your applications, you can install a library onto one of your workspace or notebook session.

To learn more about how to manage R libraries, see [R library management](#).

Notebook utilities

Microsoft Spark Utilities (MSSparkUtils) is a built-in package to help you easily perform common tasks. You can use MSSparkUtils to work with file systems, to get environment variables, to chain notebooks together, and to work with secrets. MSSparkUtils is supported for R notebooks.

To get started, you can run the following commands:

```
SparkR  
library(notebookutils)  
mssparkutils.fs.help()
```

Learn more about the supported MSSparkUtils commands at [Use Microsoft Spark Utilities](#).

Use SparkR

[SparkR](#) is an R package that provides a light-weight frontend to use Apache Spark from R. SparkR provides a distributed data frame implementation that supports operations like selection, filtering, aggregation etc. SparkR also supports distributed machine learning using MLlib.

You can learn more about how to use SparkR by visiting [How to use SparkR](#).

Use sparklyr

[sparklyr](#) is an R interface to Apache Spark. It provides a mechanism to interact with Spark using familiar R interfaces. You can use sparklyr through Spark batch job definitions or with interactive Microsoft Fabric notebooks.

To learn more about how to use sparklyr, visit [How to use sparklyr](#).

Use Tidyverse

[Tidyverse](#) is a collection of R packages that data scientists commonly use in everyday data analyses. It includes packages for data import (`readr`), data visualization (`ggplot2`), data manipulation (`dplyr`, `tidyr`), functional programming (`purrr`), and model building (`tidymodels`) etc. The packages in `tidyverse` are designed to work together seamlessly and follow a consistent set of design principles. Microsoft Fabric distributes the latest stable version of `tidyverse` with every runtime release.

To learn more about how to use Tidyverse, visit [How to use Tidyverse](#).

R visualization

The R ecosystem offers multiple graphing libraries that come packed with many different features. By default, every Spark instance in Microsoft Fabric contains a set of curated and popular open-source libraries. You can also add or manage extra libraries or versions by using the Microsoft Fabric [library management capabilities](#).

Learn more about how to create R visualizations by visiting [R visualization](#).

Related content

- [How to use SparkR](#)

- How to use sparklyr
 - How to use Tidyverse
 - R library management
 - Visualize data in R
 - Tutorial: avocado price prediction
 - Tutorial: flight delay prediction
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

R library management

Article • 03/29/2024

Libraries provide reusable code that you might want to include in your programs or projects for Microsoft Fabric Spark.

Microsoft Fabric supports an R runtime with many popular open-source R packages, including TidyVerse, preinstalled. When a Spark instance starts, these libraries are included automatically and available to be used immediately in notebooks or Spark job definitions.

You might need to update your R libraries for various reasons. For example, one of your core dependencies released a new version, or your team has built a custom package that you need available in your Spark clusters.

There are two types of libraries you may want to include based on your scenario:

- **Feed libraries** refer to the ones residing in public sources or repositories, such as [CRAN](#) or GitHub.
- **Custom libraries** are the code built by you or your organization, .tar.gz can be managed through Library Management portals.

There are two levels of packages installed on Microsoft Fabric:

- **Environment**: Manage libraries through an [environment](#) to reuse the same set of libraries across multiple notebooks or jobs.
- **Session** : A session-level installation creates an environment for a specific notebook session. The change of session-level libraries isn't persisted between sessions.

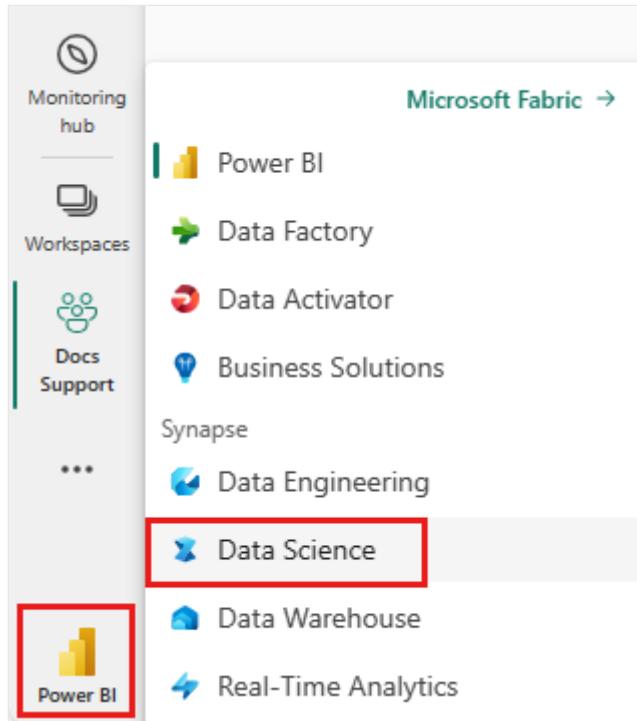
Summarizing the current available R library management behaviors:

Expand table

Library Type	Environment installation	Session-level installation
R Feed (CRAN)	Not Supported	Supported
R Custom	Supported	Supported

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



Session-level R libraries

When doing interactive data analysis or machine learning, you might try newer packages or you might need packages that are currently unavailable on your workspace. Instead of updating the workspace settings, you can use session-scoped packages to add, manage, and update session dependencies.

- When you install session-scoped libraries, only the current notebook has access to the specified libraries.
- These libraries don't impact other sessions or jobs using the same Spark pool.
- These libraries are installed on top of the base runtime and pool level libraries.
- Notebook libraries take the highest precedence.
- Session-scoped R libraries don't persist across sessions. These libraries are installed at the start of each session when the related installation commands are executed.
- Session-scoped R libraries are automatically installed across both the driver and worker nodes.

Note

The commands of managing R libraries are disabled when running pipeline jobs. If you want to install a package within a pipeline, you must use the library management capabilities at the workspace level.

Install R packages from CRAN

You can easily install an R library from [CRAN ↗](#).

```
R

# install a package from CRAN
install.packages(c("nycflights13", "Lahman"))
```

You can also use CRAN snapshots as the repository to ensure to download the same package version each time.

```
R

# install a package from CRAN snapshot
install.packages("highcharter", repos =
  "https://cran.microsoft.com/snapshot/2021-07-16/")
```

Install R packages using devtools

The `devtools` library simplifies package development to expedite common tasks. This library is installed within the default Microsoft Fabric runtime.

You can use `devtools` to specify a specific version of a library to install. These libraries are installed across all nodes within the cluster.

```
R

# Install a specific version.
install_version("caesar", version = "1.0.0")
```

Similarly, you can install a library directly from GitHub.

```
R

# Install a GitHub library.

install_github("jtilly/matchingR")
```

Currently, the following `devtools` functions are supported within Microsoft Fabric:

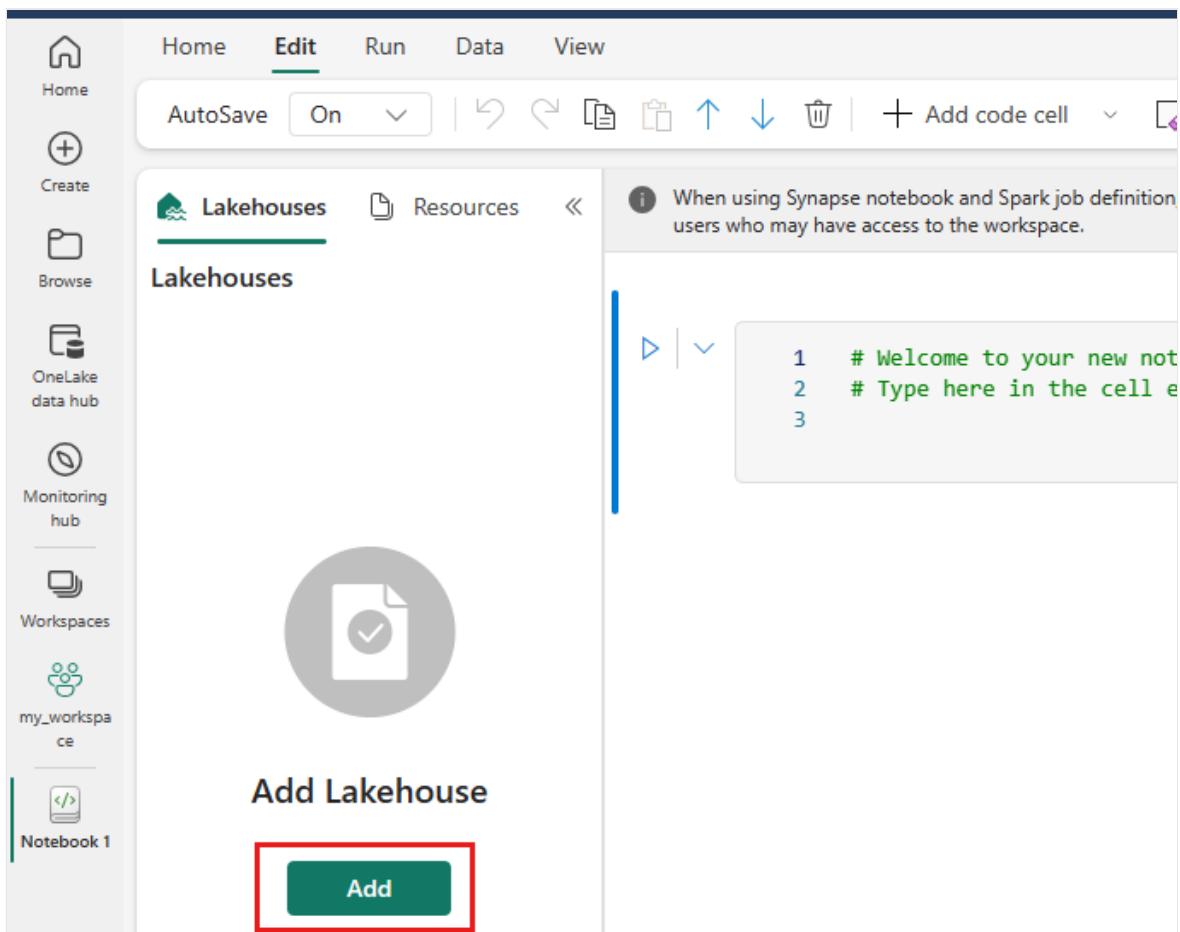
[Expand table](#)

Command	Description
<code>install_github()</code>	Installs an R package from GitHub
<code>install_gitlab()</code>	Installs an R package from GitLab
<code>install_bitbucket()</code>	Installs an R package from BitBucket
<code>install_url()</code>	Installs an R package from an arbitrary URL
<code>install_git()</code>	Installs from an arbitrary git repository
<code>install_local()</code>	Installs from a local file on disk
<code>install_version()</code>	Installs from a specific version on CRAN

Install R custom libraries

To use a session-level custom library, you must first upload it to an attached Lakehouse.

1. On the left side, select **Add** to add an existing lakehouse or create a lakehouse.



2. To add files to this lakehouse, select your workspace and then select the lakehouse.

The screenshot shows the OneLake Data Hub interface. On the left, a sidebar lists various options: Home, Create, Browse, OneLake data hub, Monitoring hub, Workspaces, SG_workspace (which is highlighted with a red box), and Notebook 1. The main area is titled "SG_workspace" and shows a table with one item:

Name	Type
SG_lakehouse	Lakehouse

A red box highlights the "SG_lakehouse" entry in the table.

3. Right click or select the "..." next to **Files** to upload your .tar.gz file.

The screenshot shows the SG_lakehouse workspace in the Explorer view. The sidebar on the left is identical to the previous screenshot. In the center, there is an "Explorer" section with a tree view showing "SG_lakehouse" expanded, with "Tables" and "Files" listed under it. A context menu is open over the "Files" node, with a red box highlighting the three-dot ellipsis button "...". The menu itself has a red box around the "Upload files" option, which is highlighted. Other options in the menu include "New shortcut", "New subfolder", "Upload", "Properties", and "Refresh".

4. After uploading, go back to your notebook. Use the following command to install the custom library to your session:

```
R  
  
install.packages("filepath/filename.tar.gz", repos = NULL, type =  
"source")
```

View installed libraries

Query all the libraries installed within your session using the `library` command.

```
R  
  
# query all the libraries installed in current session  
library()
```

Use the `packageVersion` function to check the version of the library:

```
R  
  
# check the package version  
packageVersion("caesar")
```

Remove an R package from a session

You can use the `detach` function to remove a library from the namespace. These libraries stay on disk until they're loaded again.

```
R  
  
# detach a library  
  
detach("package: caesar")
```

To remove a session-scoped package from a notebook, use the `remove.packages()` command. This library change has no impact on other sessions on the same cluster. Users can't uninstall or remove built-in libraries of the default Microsoft Fabric runtime.

Note

You can't remove core packages like SparkR, SparklyR, or R.

```
R
```

```
remove.packages("caesar")
```

Session-scoped R libraries and SparkR

Notebook-scoped libraries are available on SparkR workers.

```
R
```

```
install.packages("stringr")
library(SparkR)

str_length_function <- function(x) {
  library(stringr)
  str_length(x)
}

docs <- c("Wow, I really like the new light sabers!",
         "That book was excellent.",
         "R is a fantastic language.",
         "The service in this restaurant was miserable.",
         "This is neither positive or negative.")

spark.lapply(docs, str_length_function)
```

Session-scoped R libraries and sparklyr

With `spark_apply()` in sparklyr, you can use any R packages inside Spark. By default, in `sparklyr::spark_apply()`, the `packages` argument sets to FALSE. This copies libraries in the current libPaths to the workers, allowing you to import and use them on workers. For example, you can run the following to generate a caesar-encrypted message with `sparklyr::spark_apply()`:

```
R
```

```
install.packages("caesar", repos =
  "https://cran.microsoft.com/snapshot/2021-07-16/")

spark_version <- sparkR.version()
config <- spark_config()
sc <- spark_connect(master = "yarn", version = spark_version, spark_home =
  "/opt/spark", config = config)

apply_cases <- function(x) {
  library(caesar)
  caesar("hello world")
```

```
}
```

```
sdf_len(sc, 5) %>%
  spark_apply(apply_cases, packages=FALSE)
```

Related content

- [Create, configure, and use an environment in Microsoft Fabric](#)

Learn more about the R functionalities:

- [How to use SparkR](#)
- [How to use sparklyr](#)
- [How to use Tidyverse](#)
- [Create R visualization](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Ask the community ↗](#)

Use SparkR

Article • 09/18/2024

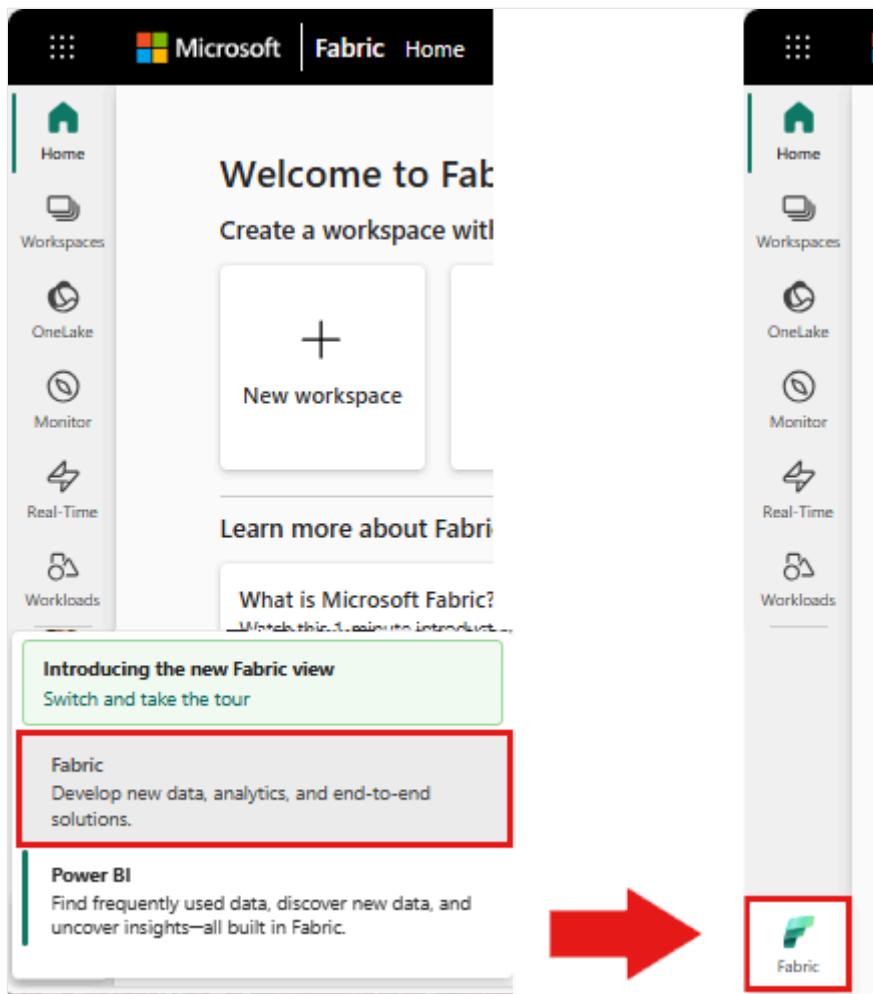
[SparkR](#) is an R package that provides a light-weight frontend to use Apache Spark from R. SparkR provides a distributed data frame implementation that supports operations like selection, filtering, aggregation etc. SparkR also supports distributed machine learning using MLlib.

Use SparkR through Spark batch job definitions or with interactive Microsoft Fabric notebooks.

R support is only available in Spark3.1 or above. R in Spark 2.4 is not supported.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- Open or create a notebook. To learn how, see [How to use Microsoft Fabric notebooks](#).
- Set the language option to **SparkR (R)** to change the primary language.
- Attach your notebook to a lakehouse. On the left side, select **Add** to add an existing lakehouse or to create a lakehouse.

Read and write SparkR DataFrames

Read a SparkR DataFrame from a local R data.frame

The simplest way to create a DataFrame is to convert a local R data.frame into a Spark DataFrame.

```
R

# load SparkR package
library(SparkR)

# read a SparkR DataFrame from a local R data.frame
df <- createDataFrame(faithful)
```

```
# displays the content of the DataFrame  
display(df)
```

Read and write SparkR DataFrame from Lakehouse

Data can be stored on the local filesystem of cluster nodes. The general methods to read and write a SparkR DataFrame from Lakehouse is `read.df` and `write.df`. These methods take the path for the file to load and the type of data source. SparkR supports reading CSV, JSON, text, and Parquet files natively.

To read and write to a Lakehouse, first add it to your session. On the left side of the notebook, select **Add** to add an existing Lakehouse or create a Lakehouse.

① Note

To access Lakehouse files using Spark packages, such as `read.df` or `write.df`, use its *ADFS path* or *relative path for Spark*. In the Lakehouse explorer, right click on the files or folder you want to access and copy its *ADFS path* or *relative path for Spark* from the contextual menu.

R

```
# write data in CSV using relative path for Spark  
temp_csv_spark<- "Files/data/faithful.csv"  
write.df(df, temp_csv_spark ,source="csv", mode = "overwrite", header =  
"true")  
  
# read data in CSV using relative path for Spark  
faithfulDF_csv <- read.df(temp_csv_spark, source= "csv", header = "true",  
inferSchema = "true")  
  
# displays the content of the DataFrame  
display(faithfulDF_csv)
```

R

```
# write data in parquet using ADFS path  
temp_parquet_spark<- "abfss://xxx/xxx/data/faithful.parquet"  
write.df(df, temp_parquet_spark ,source="parquet", mode = "overwrite",  
header = "true")  
  
# read data in parquet using ADFS path  
faithfulDF_pq <- read.df(temp_parquet_spark, source= "parquet", header =  
"true", inferSchema = "true")
```

```
# displays the content of the DataFrame  
display(faithfulDF_pq)
```

Microsoft Fabric has `tidyverse` preinstalled. You can access Lakehouse files in your familiar R packages, such as reading and writing Lakehouse files using `readr::read_csv()` and `readr::write_csv()`.

ⓘ Note

To access Lakehouse files using R packages, you need to use the *File API path*. In the Lakehouse explorer, right click on the file or folder that you want to access and copy its *File API path* from the contextual menu.

R

```
# read data in CSV using API path  
# To find the path, navigate to the csv file, right click, and Copy File  
# API path.  
temp_csv_api<- '/lakehouse/default/Files/data/faithful.csv/part-00000-  
d8e09a34-bd63-41bd-8cf8-f4ed2ef90e6c-c000.csv'  
faithfulDF_API <- readr::read_csv(temp_csv_api)  
  
# display the content of the R data.frame  
head(faithfulDF_API)
```

You can also read a SparkR Dataframe on your Lakehouse using SparkSQL queries.

R

```
# Register earlier df as temp view  
createOrReplaceTempView(df, "eruptions")  
  
# Create a df using a SparkSQL query  
waiting <- sql("SELECT * FROM eruptions")  
  
head(waiting)
```

DataFrame operations

SparkR DataFrames support many functions to do structured data processing. Here are some basic examples. A complete list can be found in the [SparkR API docs](#).

Select rows and columns

R

```
# Select only the "waiting" column  
head(select(df,df$waiting))
```

R

```
# Pass in column name as strings  
head(select(df, "waiting"))
```

R

```
# Filter to only retain rows with waiting times longer than 70 mins  
head(filter(df, df$waiting > 70))
```

Grouping and aggregation

SparkR data frames support many commonly used functions to aggregate data after grouping. For example, we can compute a histogram of the waiting time in the faithful dataset as shown below

R

```
# we use the `n` operator to count the number of times each waiting time  
appears  
head(summarize(groupBy(df, df$waiting), count = n(df$waiting)))
```

R

```
# we can also sort the output from the aggregation to get the most common  
waiting times  
waiting_counts <- summarize(groupBy(df, df$waiting), count = n(df$waiting))  
head(arrange(waiting_counts, desc(waiting_counts$count)))
```

Column operations

SparkR provides many functions that can be directly applied to columns for data processing and aggregation. The following example shows the use of basic arithmetic functions.

R

```
# convert waiting time from hours to seconds.  
# you can assign this to a new column in the same DataFrame  
df$waiting_secs <- df$waiting * 60  
head(df)
```

Apply user-defined function

SparkR supports several kinds of user-defined functions:

Run a function on a large dataset with `dapply` or `dapplyCollect`

`dapply`

Apply a function to each partition of a `SparkDataFrame`. The function to be applied to each partition of the `SparkDataFrame` and should have only one parameter, to which a `data.frame` corresponds to each partition will be passed. The output of function should be a `data.frame`. Schema specifies the row format of the resulting a `SparkDataFrame`. It must match to [data types](#) of returned value.

R

```
# convert waiting time from hours to seconds  
df <- createDataFrame(faithful)  
schema <- structType(structField("eruptions", "double"),  
                      structField("waiting", "double"),  
                      structField("waiting_secs", "double"))  
  
# apply UDF to DataFrame  
df1 <- dapply(df, function(x) { x <- cbind(x, x$waiting * 60) }, schema)  
head(collect(df1))
```

`dapplyCollect`

Like `dapply`, apply a function to each partition of a `SparkDataFrame` and collect the result back. The output of the function should be a `data.frame`. But, this time, schema isn't required to be passed. Note that `dapplyCollect` can fail if the outputs of the function run on all the partition can't be pulled to the driver and fit in driver memory.

R

```
# convert waiting time from hours to seconds  
# apply UDF to DataFrame and return a R's data.frame
```

```
ldf <- dapplyCollect(  
  df,  
  function(x) {  
    x <- cbind(x, "waiting_secs" = x$waiting * 60)  
  })  
head(ldf, 3)
```

Run a function on a large dataset grouping by input column(s) with `gapply` or `gapplyCollect`

`gapply`

Apply a function to each group of a `SparkDataFrame`. The function is to be applied to each group of the `SparkDataFrame` and should have only two parameters: grouping key and R `data.frame` corresponding to that key. The groups are chosen from `SparkDataFrames` column(s). The output of the function should be a `data.frame`. Schema specifies the row format of the resulting `SparkDataFrame`. It must represent R function's output schema from Spark [data types](#). The column names of the returned `data.frame` are set by user.

R

```
# determine six waiting times with the largest eruption time in minutes.  
schema <- structType(structField("waiting", "double"),  
  structField("max_eruption", "double"))  
result <- gapply(  
  df,  
  "waiting",  
  function(key, x) {  
    y <- data.frame(key, max(x$eruptions))  
  },  
  schema)  
head(collect(arrange(result, "max_eruption", decreasing = TRUE)))
```

`gapplyCollect`

Like `gapply`, applies a function to each group of a `SparkDataFrame` and collect the result back to R `data.frame`. The output of the function should be a `data.frame`. But, the schema isn't required to be passed. Note that `gapplyCollect` can fail if the outputs of the function run on all the partition can't be pulled to the driver and fit in driver memory.

R

```

# determine six waiting times with the largest eruption time in minutes.
result <- gapplyCollect(
  df,
  "waiting",
  function(key, x) {
    y <- data.frame(key, max(x$eruptions))
    colnames(y) <- c("waiting", "max_eruption")
    y
  })
head(result[order(result$max_eruption, decreasing = TRUE), ])

```

Run local R functions distributed with spark.lapply

spark.lapply

Similar to `lapply` in native R, `spark.lapply` runs a function over a list of elements and distributes the computations with Spark. Applies a function in a manner that is similar to `doParallel` or `lapply` to elements of a list. The results of all the computations should fit in a single machine. If that is not the case, they can do something like `df <- createDataFrame(list)` and then use `dapply`.

R

```

# perform distributed training of multiple models with spark.lapply. Here,
we pass
# a read-only list of arguments which specifies family the generalized
linear model should be.
families <- c("gaussian", "poisson")
train <- function(family) {
  model <- glm(Sepal.Length ~ Sepal.Width + Species, iris, family = family)
  summary(model)
}
# return a list of model's summaries
model.summaries <- spark.lapply(families, train)

# print the summary of each model
print(model.summaries)

```

Run SQL queries from SparkR

A SparkR DataFrame can also be registered as a temporary view that allows you to run SQL queries over its data. The `sql` function enables applications to run SQL queries programmatically and returns the result as a SparkR DataFrame.

R

```
# Register earlier df as temp view
createOrReplaceTempView(df, "eruptions")

# Create a df using a SparkSQL query
waiting <- sql("SELECT waiting FROM eruptions where waiting>70 ")

head(waiting)
```

Machine learning

SparkR exposes most of MLlib algorithms. Under the hood, SparkR uses MLlib to train the model.

The following example shows how to build a Gaussian GLM model using SparkR. To run linear regression, set family to "gaussian". To run logistic regression, set family to "binomial". When using SparkML `glm` SparkR automatically performs one-hot encoding of categorical features so that it doesn't need to be done manually. Beyond String and Double type features, it's also possible to fit over MLlib Vector features, for compatibility with other MLlib components.

To learn more about which machine learning algorithms are supported, you can visit the [documentation for SparkR and MLlib](#).

R

```
# create the DataFrame
cars <- cbind(model = rownames(mtcars), mtcars)
carsDF <- createDataFrame(cars)

# fit a linear model over the dataset.
model <- spark.glm(carsDF, mpg ~ wt + cyl, family = "gaussian")

# model coefficients are returned in a similar format to R's native glm().
summary(model)
```

Related content

- [How to use sparklyr](#)
- [How to use Tidyverse](#)
- [R library management](#)
- [Create R visualization](#)
- [Tutorial: avocado price prediction](#)

- Tutorial: flight delay prediction
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Use sparklyr

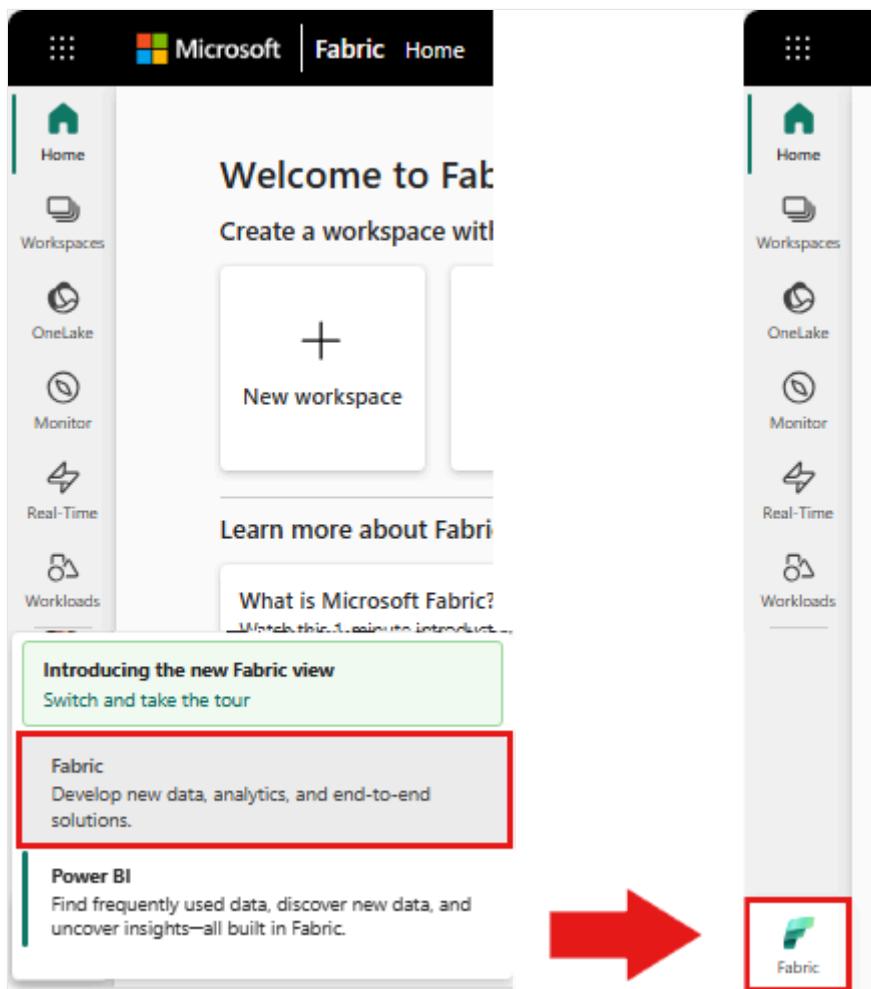
Article • 04/17/2025

The R language [sparklyr](#) resource serves as an interface to Apache Spark. The sparklyr resource provides a mechanism to interact with Spark with familiar R interfaces. Use sparklyr through Spark batch job definitions or with interactive Microsoft Fabric notebooks.

sparklyr is used with other [tidyverse](#) packages - for example, [dplyr](#). Microsoft Fabric distributes the latest stable version of both sparklyr and tidyverse with every runtime release. You can import these resources and start using the API.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- Open or create a notebook. To learn how, see [How to use Microsoft Fabric notebooks](#).

- Set the language option to **SparkR (R)** to change the primary language.
- Attach your notebook to a lakehouse. On the left side, select **Add** to add an existing lakehouse or to create a lakehouse.

Connect sparklyr to Synapse Spark cluster

The `spark_connect()` function connection method establishes a `sparklyr` connection. The function builds a new connection method named `synapse`, which connects to an existing Spark session. It dramatically reduces the `sparklyr` session start time. This connection method is available in the [open sourced sparklyr project](#). With `method = "synapse"`, you can use both `sparklyr` and `SparkR` in the same session, and easily [share data between them](#). The following notebook cell code sample uses the `spark_connect()` function:

```
R

# connect sparklyr to your spark cluster
spark_version <- sparkR.version()
config <- spark_config()
sc <- spark_connect(master = "yarn", version = spark_version, spark_home =
"/opt/spark", method = "synapse", config = config)
```

Use sparklyr to read data

A new Spark session contains no data. You must then either load data into your Spark session's memory, or point Spark to the location of the data so that the session can access the data on-demand:

```
R

# load the sparklyr package
library(sparklyr)

# copy data from R environment to the Spark session's memory
mtcars_tbl <- copy_to(sc, mtcars, "spark_mtcars", overwrite = TRUE)

head(mtcars_tbl)
```

With `sparklyr`, you can also `write` and `read` data from a Lakehouse file using an ABFS path value. To read and write to a Lakehouse, first add the Lakehouse to your session. On the left side of the notebook, select **Add** to add an existing Lakehouse. Additionally, you can create a Lakehouse.

To find your ABFS path, right-select the **Files** folder in your Lakehouse, and select **Copy ABFS path**. Paste your path to replace `abfss://xxxx@onelake.dfs.fabric.microsoft.com/xxxx/Files` in the following code sample:

```
R

temp_csv =
"abfss://xxxx@onelake.dfs.fabric.microsoft.com/xxxx/Files/data/mtcars.csv"

# write the table to your lakehouse using the ABFS path
spark_write_csv(mtcars_tbl, temp_csv, header = TRUE, mode = 'overwrite')

# read the data as CSV from lakehouse using the ABFS path
mtcarsDF <- spark_read_csv(sc, temp_csv)
head(mtcarsDF)
```

Use `sparklyr` to manipulate data

`sparklyr` provides different ways to process data inside Spark, with:

- `dplyr` commands
- SparkSQL
- Spark's feature transformers

Use `dplyr`

You can use familiar `dplyr` commands to prepare data inside Spark. The commands run inside Spark, preventing unnecessary data transfers between R and Spark.

```
R

# count cars by the number of cylinders the engine contains (cyl), order the
results descendingly
library(dplyr)

cargroup <- group_by(mtcars_tbl, cyl) %>%
  count() %>%
  arrange(desc(n))

cargroup
```

The [Manipulating Data with dplyr](#) resource offers more information about use of `dplyr` with Spark. `sparklyr` and `dplyr` translate the R commands into Spark SQL. Use `show_query()` to show the resulting query:

R

```
# show the dplyr commands that are to run against the Spark connection
dplyr::show_query(cargroup)
```

Use SQL

You can also execute SQL queries directly against tables within a Spark cluster. The `spark_connection()` object implements a [DBI](#) interface for Spark, so you can use `dbGetQuery()` to execute SQL and return the result as an R data frame:

R

```
library(DBI)
dbGetQuery(sc, "select cyl, count(*) as n from spark_mtcars
GROUP BY cyl
ORDER BY n DESC")
```

Use Feature Transformers

Both of the previous methods rely on SQL statements. Spark provides commands that make some data transformations more convenient, without the use of SQL. For example, the `ft_binarizer()` command simplifies the creation of a new column that indicates whether or not a value in another column exceeds a certain threshold:

R

```
mtcars_tbl %>%
  ft_binarizer("mpg", "over_20", threshold = 20) %>%
  select(mpg, over_20) %>%
  head(5)
```

The [Reference -FT](#) resource offers a full list of the Spark Feature Transformers available through `sparklyr`.

Share data between `sparklyr` and `SparkR`

When you [connect sparklyr to synapse spark cluster with method = "synapse"](#), both `sparklyr` and `SparkR` become available in the same session and can easily share data between themselves. You can create a spark table in `sparklyr`, and read it from `SparkR`:

R

```

# load the sparklyr package
library(sparklyr)

# Create table in `sparklyr`
mtcars_sparklyr <- copy_to(sc, df = mtcars, name = "mtcars_tbl", overwrite = TRUE,
repartition = 3L)

# Read table from `SparkR`
mtcars_sparklyr <- SparkR::sql("select cyl, count(*) as n
from mtcars_tbl
GROUP BY cyl
ORDER BY n DESC")

head(mtcars_sparklyr)

```

Machine learning

The following example uses `ml_linear_regression()` to fit a linear regression model. The model uses the built-in `mtcars` dataset to try to predict the fuel consumption (`mpg`) of a car based on the weight (`wt`) of the car and the number of cylinders (`cyl`) of the car engine. All cases here assume a linear relationship between `mpg` and each of our features.

Generate testing and training data sets

Use a split - 70% for training and 30% - to test the model. Changes to this ratio lead to different models:

```

R

# split the dataframe into test and training dataframes

partitions <- mtcars_tbl %>%
  select(mpg, wt, cyl) %>%
  sdf_random_split(training = 0.7, test = 0.3, seed = 2023)

```

Train the model

Train the Logistic Regression model.

```

R

fit <- partitions$training %>%
  ml_linear_regression(mpg ~ .)

```

```
fit
```

Use `summary()` to learn more about the quality of our model, and the statistical significance of each of our predictors:

```
R
```

```
summary(fit)
```

Use the model

Call `ml_predict()` to apply the model to the test dataset:

```
R
```

```
pred <- ml_predict(fit, partitions$test)  
head(pred)
```

Visit [Reference - ML](#) for a list of Spark ML models available through sparklyr.

Disconnect from Spark cluster

Call `spark_disconnect()`, or select the **Stop session** button on top of the notebook ribbon, to end your Spark session:

```
R
```

```
spark_disconnect(sc)
```

Related content

Learn more about the R functionalities:

- [How to use SparkR](#)
- [How to use Tidyverse](#)
- [R library management](#)
- [Create R visualization](#)
- [Tutorial: avocado price prediction](#)
- [Tutorial: flight delay prediction](#)

Use Tidyverse

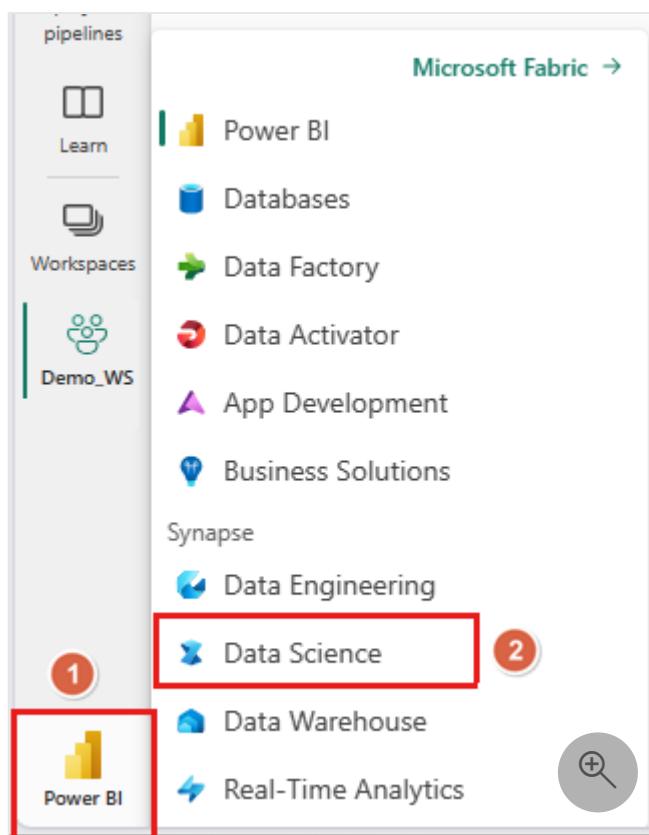
Article • 11/15/2023

Tidyverse [↗](#) is a collection of R packages that data scientists commonly use in everyday data analyses. It includes packages for data import (`readr`), data visualization (`ggplot2`), data manipulation (`dplyr`, `tidyverse`), functional programming (`purrr`), and model building (`tidymodels`) etc. The packages in `tidyverse` are designed to work together seamlessly and follow a consistent set of design principles.

Microsoft Fabric distributes the latest stable version of `tidyverse` with every runtime release. Import and start using your familiar R packages.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#) [↗](#).
- Switch to the Data Science experience by using the experience switcher icon on the left side of your home page.



- Open or create a notebook. To learn how, see [How to use Microsoft Fabric notebooks](#).

- Change the primary language by setting the `language` option to **SparkR (R)**.
- Attach your notebook to a lakehouse. On the left side, select **Add** to add an existing lakehouse or create a lakehouse.

Load tidyverse

R

```
# load tidyverse
library(tidyverse)
```

Data import

`readr` is an R package that provides tools for reading rectangular data files such as CSV, TSV, and fixed-width files. `readr` provides a fast and friendly way to read rectangular data files such as providing functions `read_csv()` and `read_tsv()` for reading CSV and TSV files respectively.

Let's first create an R `data.frame`, write it to lakehouse using `readr::write_csv()` and read it back with `readr::read_csv()`.

ⓘ Note

To access Lakehouse files using `readr`, you need to use the *File API path*. In the Lakehouse explorer, right click on the file or folder that you want to access and copy its *File API path* from the contextual menu.

R

```
# create an R data frame
set.seed(1)
stocks <- data.frame(
  time = as.Date('2009-01-01') + 0:9,
  X = rnorm(10, 20, 1),
  Y = rnorm(10, 20, 2),
  Z = rnorm(10, 20, 4)
)
stocks
```

Then let's write the data to lakehouse using the *File API path*.

R

```
# write data to lakehouse using the File API path  
temp_csv_api <- "/lakehouse/default/Files/stocks.csv"  
readr::write_csv(stocks,temp_csv_api)
```

Read the data from lakehouse.

R

```
# read data from lakehouse using the File API path  
stocks_readr <- readr::read_csv(temp_csv_api)  
  
# show the content of the R date.frame  
head(stocks_readr)
```

Data tidying

`tidyR` is an R package that provides tools for working with messy data. The main functions in `tidyR` are designed to help you reshape data into a tidy format. Tidy data has a specific structure where each variable is a column and each observation is a row, which makes it easier to work with data in R and other tools.

For example, the `gather()` function in `tidyR` can be used to convert wide data into long data. Here's an example:

R

```
# convert the stock data into longer data  
library(tidyR)  
stocksL <- gather(data = stocks, key = stock, value = price, X, Y, Z)  
stocksL
```

Functional programming

`purrr` is an R package that enhances R's functional programming toolkit by providing a complete and consistent set of tools for working with functions and vectors. The best place to start with `purrr` is the family of `map()` functions that allow you to replace many for loops with code that is both more succinct and easier to read. Here's an example of using `map()` to apply a function to each element of a list:

R

```
# double the stock values using purrr
library(purrr)
stocks_double = map(stocks %>% select_if(is.numeric), ~.x*2)
stocks_double
```

Data manipulation

`dplyr` is an R package that provides a consistent set of verbs that help you solve the most common data manipulation problems, such as selecting variables based on the names, picking cases based on the values, reducing multiple values down to a single summary, and changing the ordering of the rows etc. Here are some examples:

R

```
# pick variables based on their names using select()
stocks_value <- stocks %>% select(X:Z)
stocks_value
```

R

```
# pick cases based on their values using filter()
filter(stocks_value, X >20)
```

R

```
# add new variables that are functions of existing variables using mutate()
library(lubridate)

stocks_wday <- stocks %>%
  select(time:Z) %>%
  mutate(
    weekday = wday(time)
  )

stocks_wday
```

R

```
# change the ordering of the rows using arrange()
arrange(stocks_wday, weekday)
```

R

```
# reduce multiple values down to a single summary using summarise()
stocks_wday %>%
  group_by(weekday) %>%
  summarize(meanX = mean(X), n= n())
```

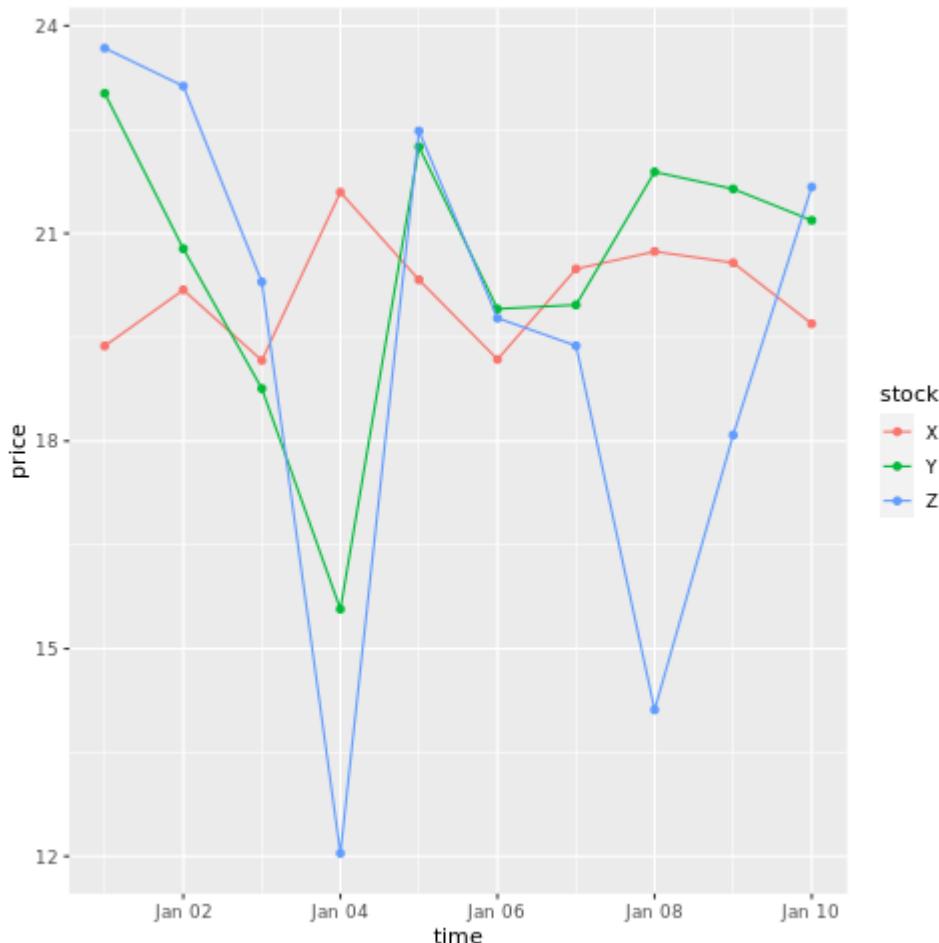
Data visualization

`ggplot2` is an R package for declaratively creating graphics, based on The Grammar of Graphics. You provide the data, tell `ggplot2` how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details. Here are some examples:

R

```
# draw a chart with points and lines all in one

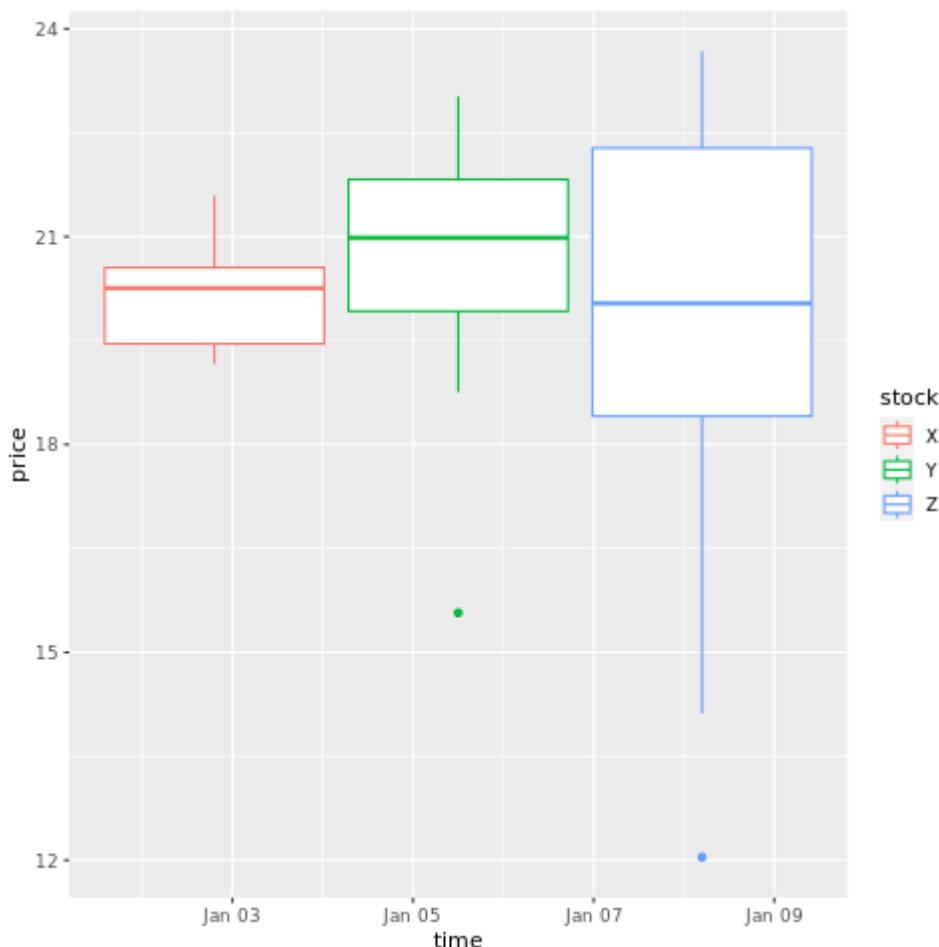
ggplot(stocksL, aes(x=time, y=price, colour = stock)) +
  geom_point()+
  geom_line()
```



R

```
# draw a boxplot

ggplot(stocksL, aes(x=time, y=price, colour = stock)) +
  geom_boxplot()
```



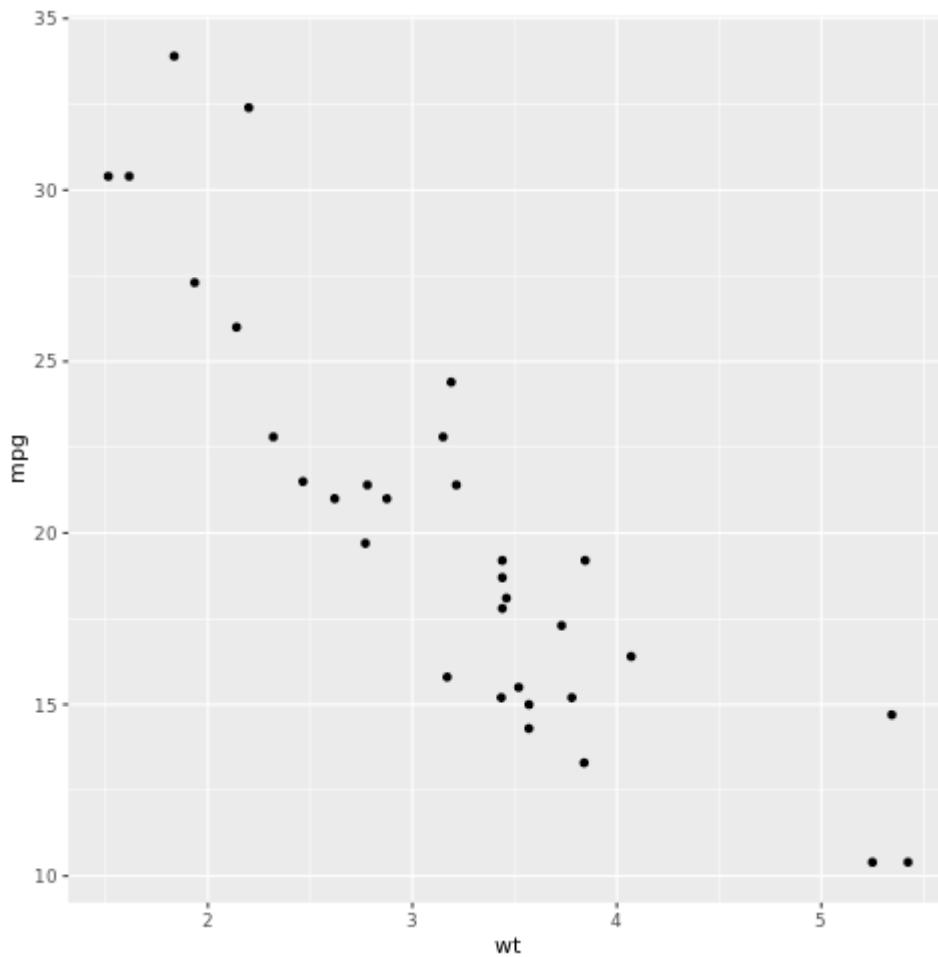
Model building

The `tidymodels` framework is a collection of packages for modeling and machine learning using `tidyverse` principles. It covers a list of core packages for a wide variety of model building tasks, such as `rsample` for train/test dataset sample splitting, `parsnip` for model specification, `recipes` for data preprocessing, `workflows` for modeling workflows, `tune` for hyperparameters tuning, `yardstick` for model evaluation, `broom` for tiding model outputs, and `dials` for managing tuning parameters. You can learn more about the packages by visiting [tidymodels website](#). Here's an example of building a linear regression model to predict the miles per gallon (mpg) of a car based on its weight (wt):

R

```
# look at the relationship between the miles per gallon (mpg) of a car and
# its weight (wt)
```

```
ggplot(mtcars, aes(wt,mpg))+  
  geom_point()
```



From the scatterplot, the relationship looks approximately linear and the variance looks constant. Let's try to model this using linear regression.

R

```
library(tidymodels)  
  
# split test and training dataset  
set.seed(123)  
split <- initial_split(mtcars, prop = 0.7, strata = "cyl")  
train <- training(split)  
test <- testing(split)  
  
  
# config the linear regression model  
lm_spec <- linear_reg() %>%  
  set_engine("lm") %>%  
  set_mode("regression")  
  
# build the model  
lm_fit <- lm_spec %>%  
  fit(mpg ~ wt, data = train)
```

```
tidy(lm_fit)
```

Apply the linear regression model to predict on test dataset.

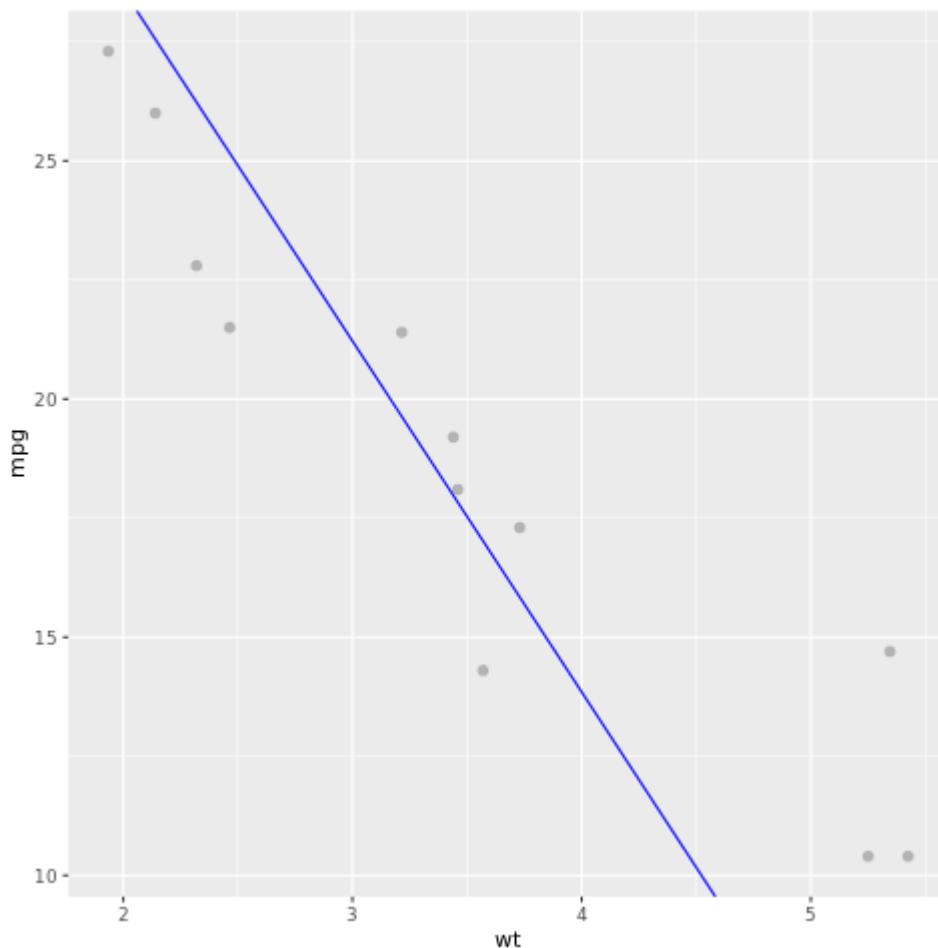
R

```
# using the lm model to predict on test dataset
predictions <- predict(lm_fit, test)
predictions
```

Let's take a look at the model result. We can draw the model as a line chart and the test ground truth data as points on the same chart. The model looks good.

R

```
# draw the model as a line chart and the test data groundtruth as points
lm_aug <- augment(lm_fit, test)
ggplot(lm_aug, aes(x = wt, y = mpg)) +
  geom_point(size=2,color="grey70") +
  geom_abline(intercept = lm_fit$fit$coefficients[1], slope =
  lm_fit$fit$coefficients[2], color = "blue")
```



Next steps

- [How to use SparkR](#)
 - [How to use sparklyr](#)
 - [R library management](#)
 - [Visualize data in R](#)
 - [Tutorial: avocado price prediction](#)
 - [Tutorial: flight delay prediction](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

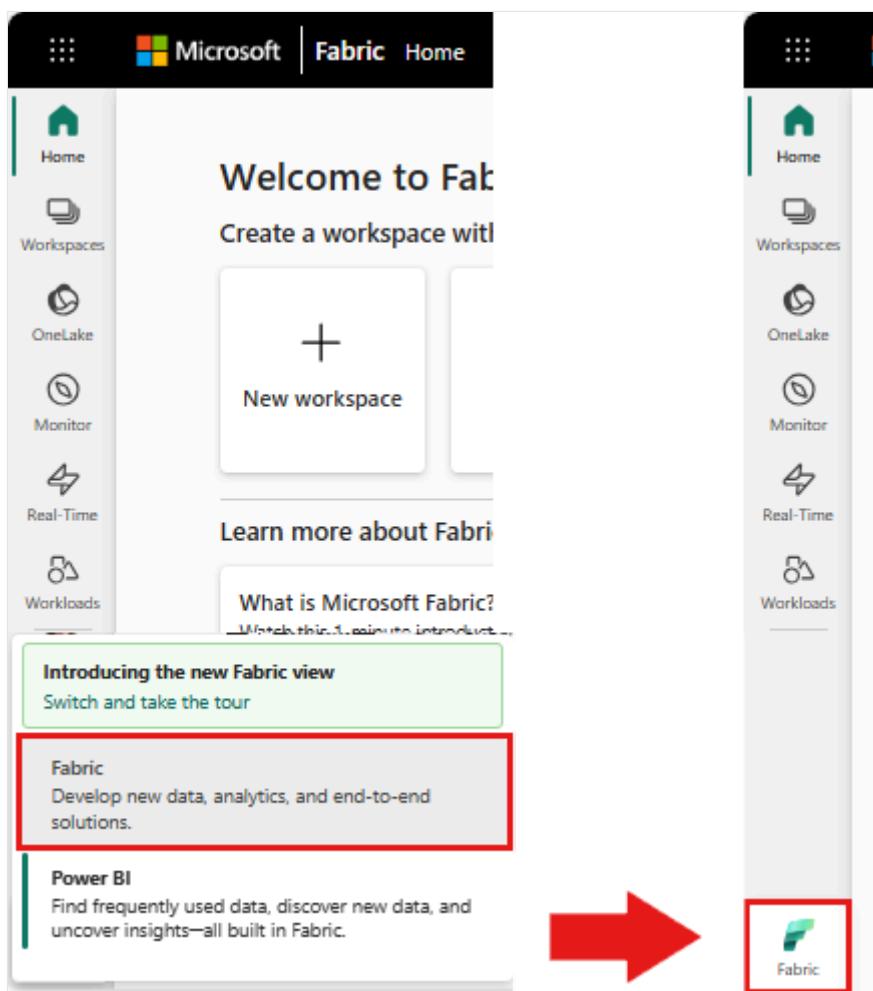
Visualize data in R

Article • 11/20/2024

The R ecosystem offers multiple graphing libraries that come packed with many different features. By default, every Apache Spark Pool in Microsoft Fabric contains a set of curated and popular open-source libraries. Add or manage extra libraries or versions by using the Microsoft Fabric [library management capabilities](#).

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.

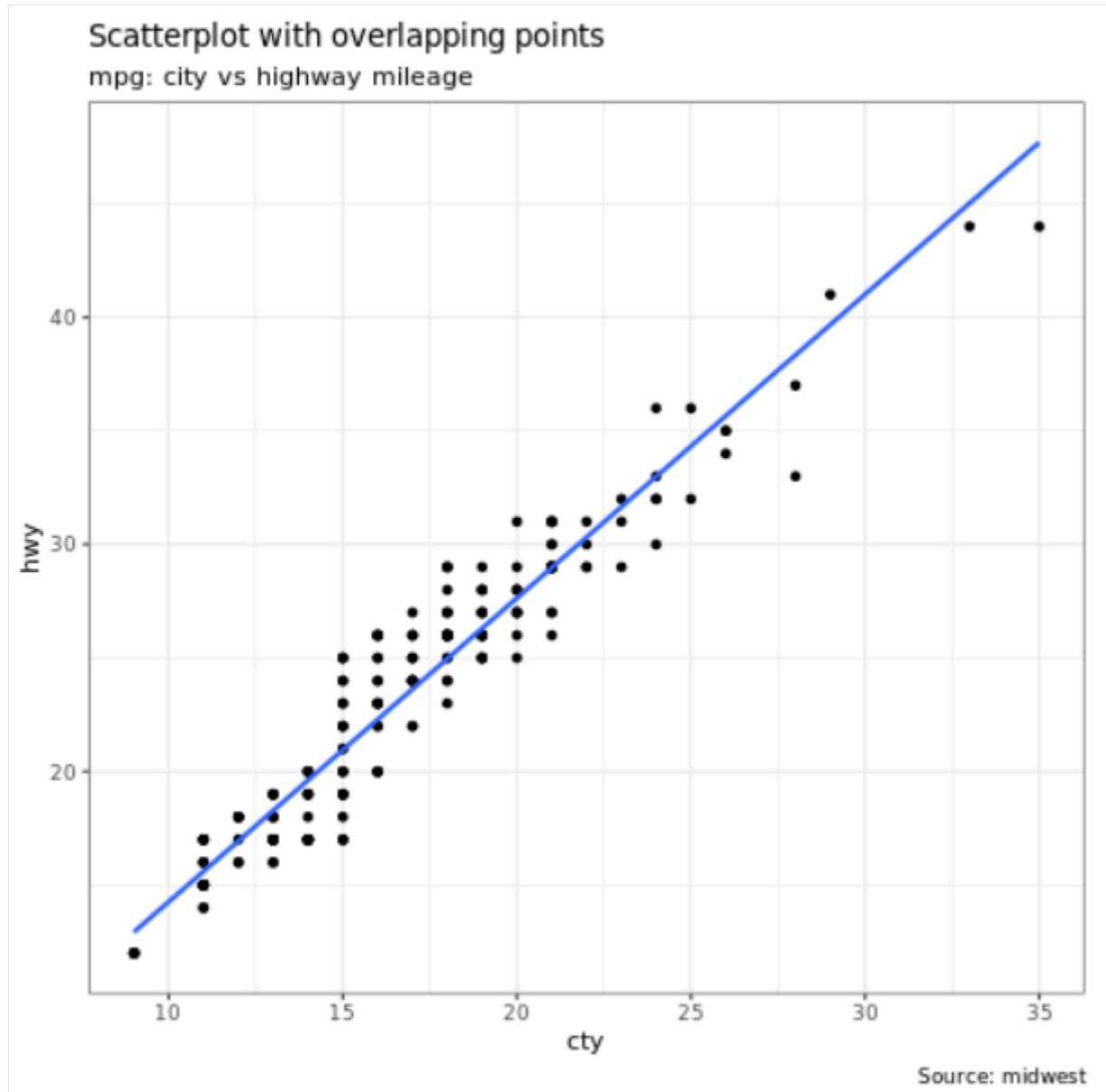


- Open or create a notebook. To learn how, see [How to use Microsoft Fabric notebooks](#).

- Set the language option to **SparkR (R)** to change the primary language.
- Attach your notebook to a lakehouse. On the left side, select **Add** to add an existing lakehouse or to create a lakehouse.

ggplot2

The [ggplot2](#) library is popular for data visualization and exploratory data analysis.



R

```
%%sparkr
library(ggplot2)
data(mpg, package="ggplot2")
theme_set(theme_bw())

g <- ggplot(mpg, aes(cty, hwy))

# Scatterplot
```

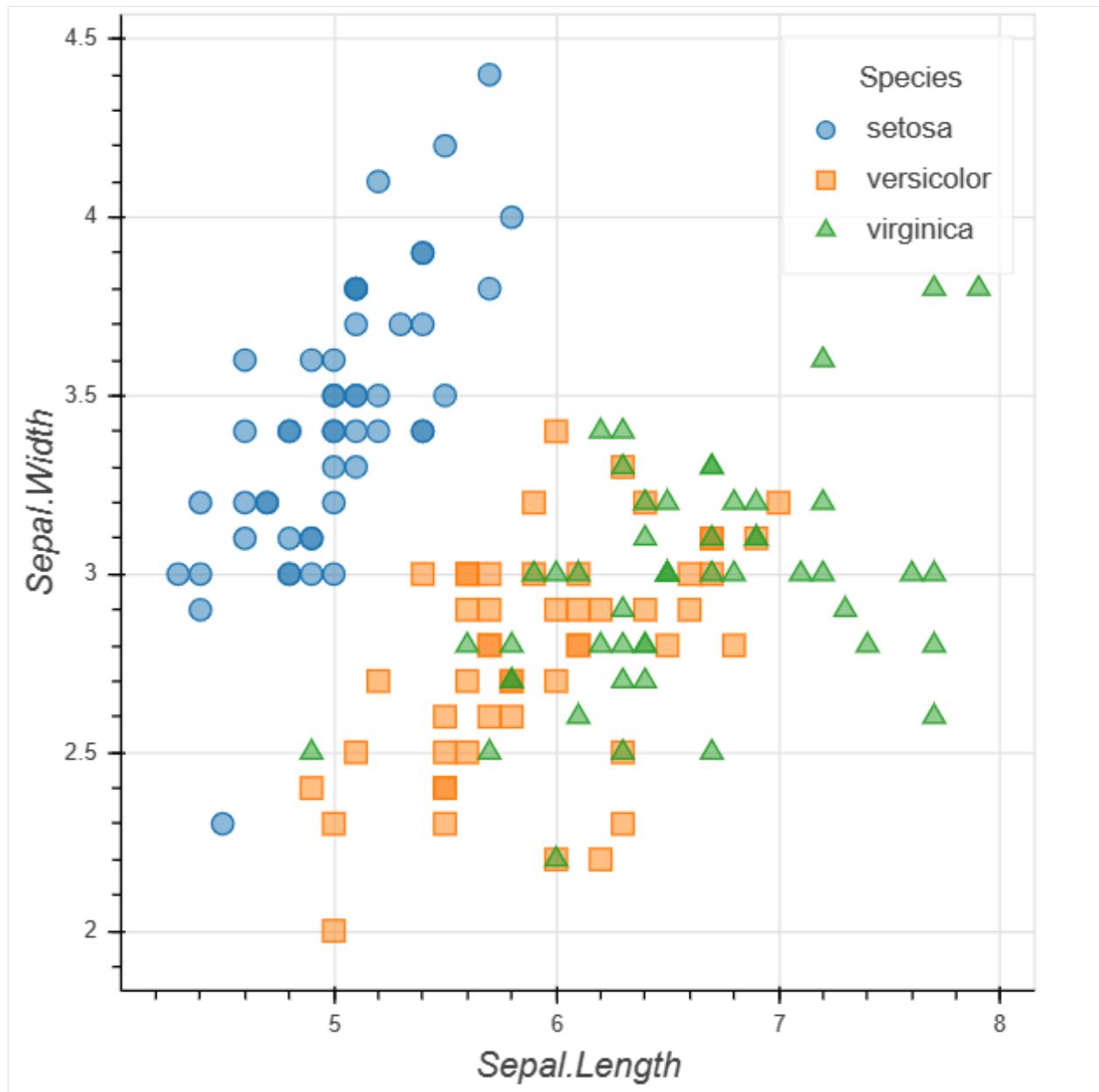
```

g + geom_point() +
  geom_smooth(method="lm", se=F) +
  labs(subtitle="mpg: city vs highway mileage",
       y="hwy",
       x="cty",
       title="Scatterplot with overlapping points",
       caption="Source: midwest")

```

rbokeh

rbokeh [↗](#) is a native R plotting library for creating interactive graphics.



R

```

library(rbokeh)
p <- figure() %>%
  ly_points(Sepal.Length, Sepal.Width, data = iris,
            color = Species, glyph = Species,

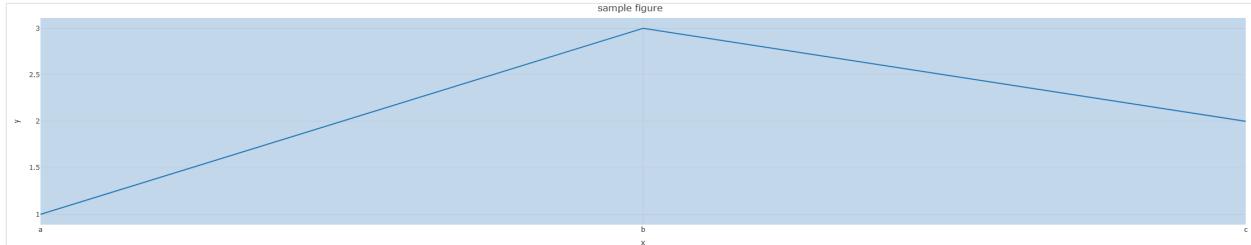
```

```
hover = list(Sepal.Length, Sepal.Width))
```

```
p
```

R Plotly

Plotly [↗](#) is an R graphing library that makes interactive, publication-quality graphs.



```
R
```

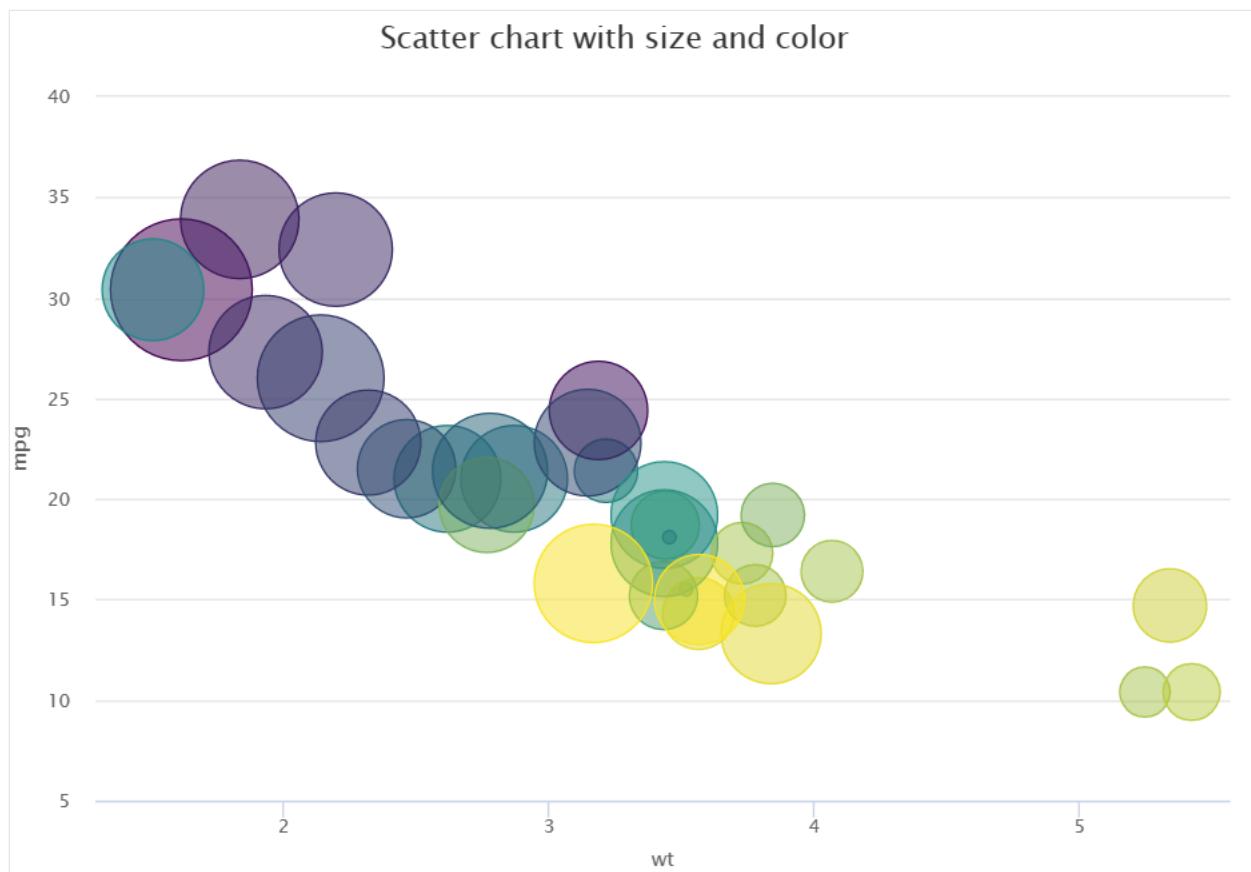
```
library(plotly)

fig <- plot_ly() %>%
  add_lines(x = c("a", "b", "c"), y = c(1,3,2))%>%
  layout(title="sample figure", xaxis = list(title = 'x'), yaxis =
  list(title = 'y'), plot_bgcolor = "#c7daec")

fig
```

Highcharter

Highcharter [↗](#) is an R wrapper for Highcharts JavaScript library and its modules.



R

```
library(magrittr)
library(highcharter)
hchart(mtcars, "scatter", hcaes(wt, mpg, z = drat, color = hp)) %>%
  hc_title(text = "Scatter chart with size and color")
```

Related content

- [How to use SparkR](#)
- [How to use sparklyr](#)
- [How to use Tidyverse](#)
- [R library management](#)
- [Tutorial: avocado price prediction](#)
- [Tutorial: flight delay prediction](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

What is semantic link?

Article • 06/09/2024

Semantic link is a feature that allows you to establish a connection between [semantic models](#) and Synapse Data Science in Microsoft Fabric. Use of semantic link is supported only in Microsoft Fabric.

- For Spark 3.4 and above, semantic link is available in the default runtime when using Fabric, and there's no need to install it.
- For Spark 3.3 or below, or to update to the latest version of semantic link, run the following command:

Python

```
%pip install -U semantic-link
```

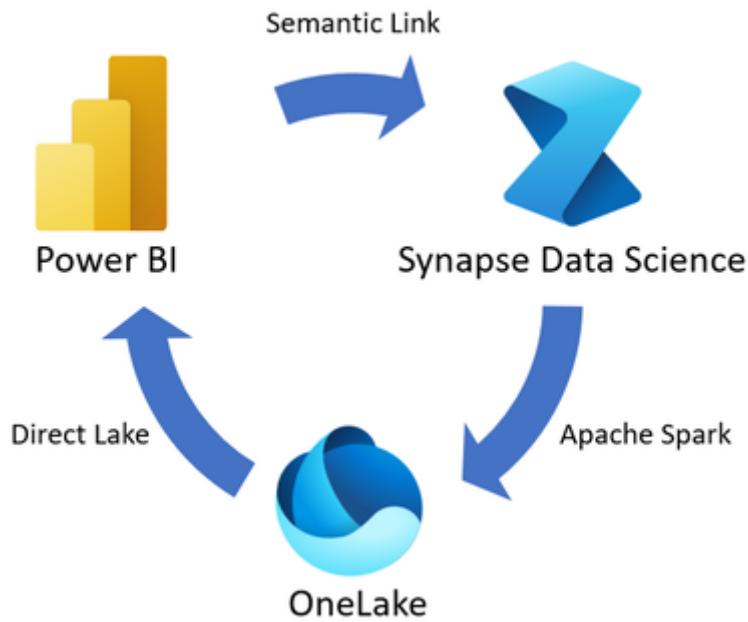
The primary goals of semantic link are to:

- Facilitate data connectivity.
- Enable the propagation of semantic information.
- Seamlessly integrate with established tools data scientists use, such as [notebooks](#).

Semantic link helps you preserve domain knowledge about data semantics in a standardized way that can speed up data analysis and reduce errors.

Semantic link data flow

The semantic link data flow starts with semantic models that contain data and semantic information. Semantic link bridges the gap between Power BI and the Synapse Data Science experience.



Semantic link allows you to use semantic models from Power BI in the Synapse Data Science experience to perform tasks such as in-depth statistical analysis and predictive modeling with machine learning techniques. You can store the output of your data science work into [OneLake](#) by using Apache Spark, and ingest the stored output into Power BI by using [Direct Lake](#).

Power BI connectivity

A semantic model serves as a single [tabular object model](#) that provides reliable sources for semantic definitions such as Power BI measures. Semantic link connects to semantic models in the following ecosystems, making it easy for data scientists to work in the system they're most familiar with.

- Python [pandas](#) ecosystem, through the [SemPy Python library](#).
- [Apache Spark](#) ecosystem, through the [Spark native connector](#). This implementation supports various languages, including PySpark, Spark SQL, R, and Scala.

Applications of semantic information

Semantic information in data includes Power BI [data categories](#) such as address and postal code, relationships between tables, and hierarchical information.

These data categories comprise metadata that semantic link propagates into the Synapse Data Science environment to enable new experiences and maintain data lineage.

Some example applications of semantic link include:

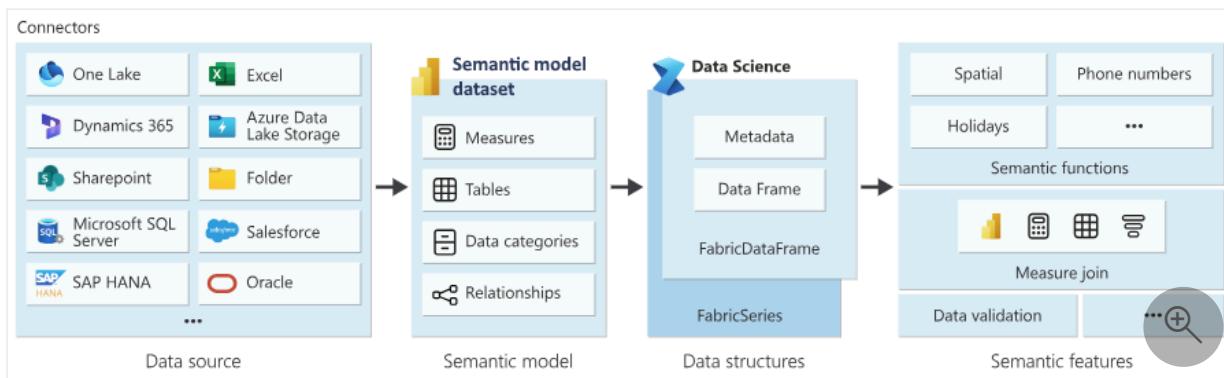
- Intelligent suggestions of built-in [semantic functions](#).
- Innovative integration for augmenting data with Power BI measures, by using [add-measures](#).
- Tools for [data quality validation](#) based on the relationships between tables and functional dependencies within tables.

Semantic link is a powerful tool that enables business analysts to use data effectively in a comprehensive data science environment.

Semantic link facilitates seamless collaboration between data scientists and business analysts by eliminating the need to reimplement business logic embedded in [Power BI measures](#). This approach ensures that both parties can work efficiently and productively, maximizing the potential of their data-driven insights.

FabricDataFrame data structure

[FabricDataFrame](#) is the primary data structure that semantic link uses to propagate semantic information from semantic models into the Synapse Data Science environment.



The `FabricDataFrame` class:

- Supports all pandas operations.
- Subclasses the [pandas DataFrame](#) and adds metadata, such as semantic information and lineage.
- Exposes semantic functions and the [add-measure](#) method that lets you use Power BI measures in data science work.

Related content

- Explore the reference documentation for the Python semantic link package (SemPy)
 - Tutorial: Clean data with functional dependencies
 - Power BI connectivity with semantic link and Microsoft Fabric
 - Explore and validate data by using semantic link
 - Explore and validate relationships in semantic models
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Tutorial: Clean data with functional dependencies (preview)

Article • 11/15/2023

In this tutorial, you use functional dependencies for data cleaning. A functional dependency exists when one column in a semantic model (a Power BI dataset) is a function of another column. For example, a *zip code* column might determine the values in a *city* column. A functional dependency manifests itself as a one-to-many relationship between the values in two or more columns within a DataFrame. This tutorial uses the *Synthea* dataset to show how functional relationships can help to detect data quality problems.

Important

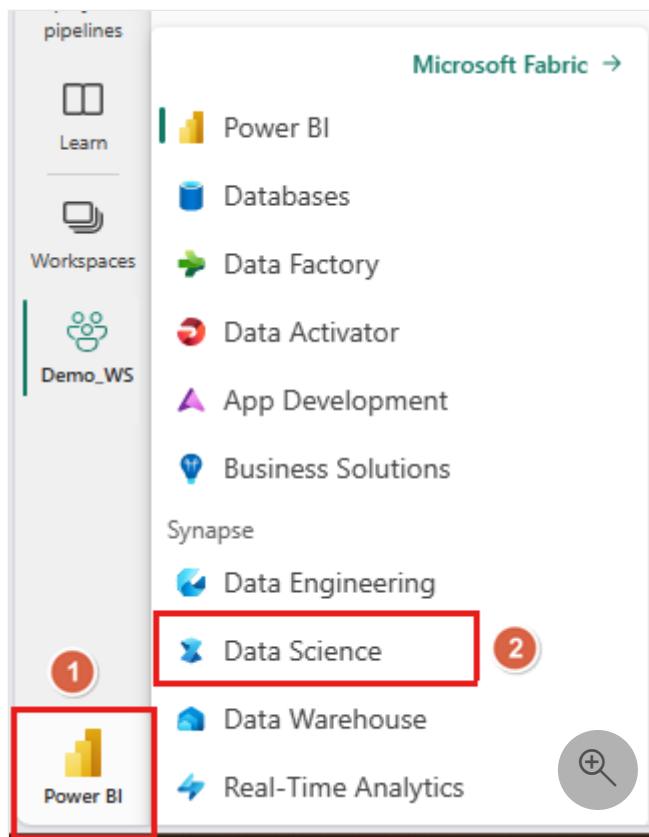
This feature is in **preview**.

In this tutorial, you learn how to:

- Apply domain knowledge to formulate hypotheses about functional dependencies in a semantic model.
- Get familiarized with components of semantic link's Python library ([SemPy](#)) that help to automate data quality analysis. These components include:
 - FabricDataFrame - a pandas-like structure enhanced with additional semantic information.
 - Useful functions that automate the evaluation of hypotheses about functional dependencies and that identify violations of relationships in your semantic models.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Switch to the Data Science experience by using the experience switcher icon on the left side of your home page.



- Select **Workspaces** from the left navigation pane to find and select your workspace. This workspace becomes your current workspace.

Follow along in the notebook

The [data_cleaning_functional_dependencies_tutorial.ipynb](#) notebook accompanies this tutorial.

If you want to open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science](#) to import the tutorial notebooks to your workspace.

Or, if you'd rather copy and paste the code from this page, you can [create a new notebook](#).

Be sure to [attach a lakehouse to the notebook](#) before you start running code.

Set up the notebook

In this section, you set up a notebook environment with the necessary modules and data.

1. Install `SemPy` from PyPI, using the `%pip` in-line installation capability within the notebook:

```
Python
```

```
%pip install semantic-link
```

2. Perform necessary imports of modules that you'll need later:

```
Python
```

```
import pandas as pd
import sempy.fabric as fabric
from sempy.fabric import FabricDataFrame
from sempy.dependencies import plot_dependency_metadata
from sempy.samples import download_synthea
```

3. Pull the sample data. For this tutorial, you use the *Synthea* dataset of synthetic medical records (small version for simplicity):

```
Python
```

```
download_synthea(which='small')
```

Explore the data

1. Initialize a `FabricDataFrame` with the content of the *providers.csv* file:

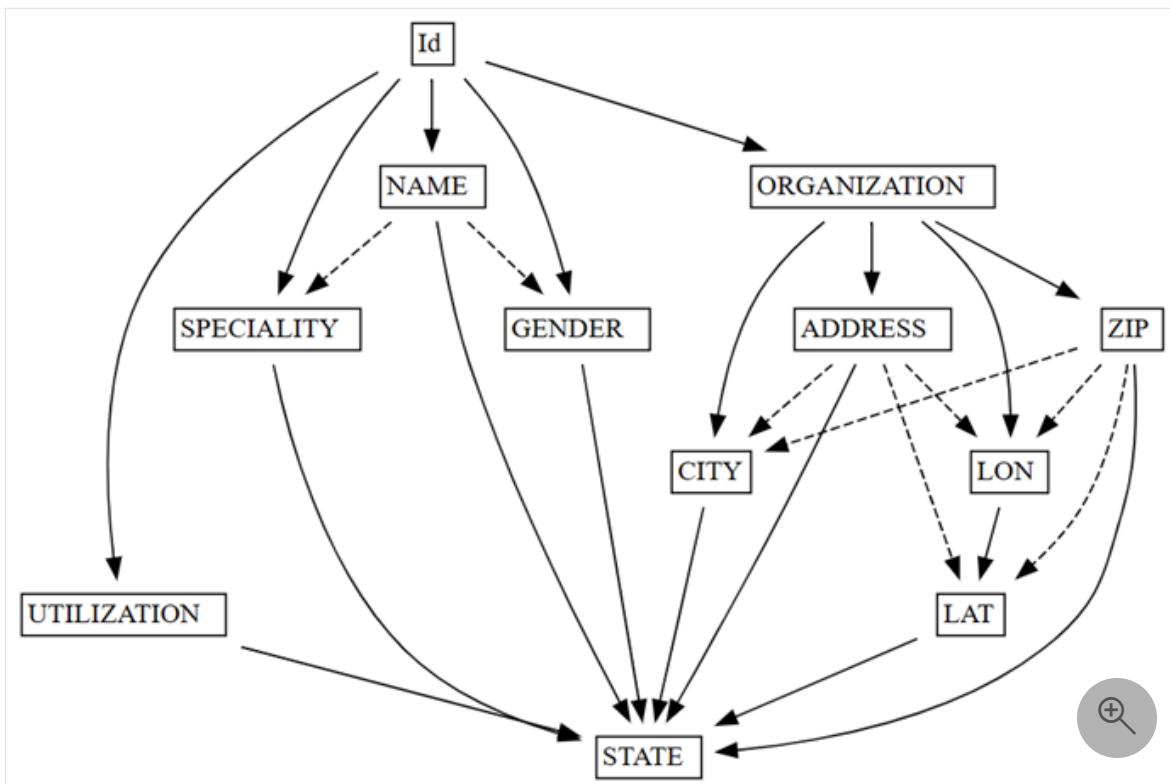
```
Python
```

```
providers = FabricDataFrame(pd.read_csv("synthea/csv/providers.csv"))
providers.head()
```

2. Check for data quality issues with SemPy's `find_dependencies` function by plotting a graph of autodetected functional dependencies:

```
Python
```

```
deps = providers.find_dependencies()
plot_dependency_metadata(deps)
```



The graph of functional dependencies shows that `Id` determines `NAME` and `ORGANIZATION` (indicated by the solid arrows), which is expected, since `Id` is unique:

3. Confirm that `Id` is unique:

```
Python
```

```
providers.Id.is_unique
```

The code returns `True` to confirm that `Id` is unique.

Analyze functional dependencies in depth

The functional dependencies graph also shows that `ORGANIZATION` determines `ADDRESS` and `ZIP`, as expected. However, you might expect `ZIP` to also determine `CITY`, but the dashed arrow indicates that the dependency is only approximate, pointing towards a data quality issue.

There are other peculiarities in the graph. For example, `NAME` doesn't determine `GENDER`, `Id`, `SPECIALITY`, or `ORGANIZATION`. Each of these peculiarities might be worth investigating.

1. Take a deeper look at the approximate relationship between `ZIP` and `CITY`, by using SemPy's `list_dependency_violations` function to see a tabular list of violations:

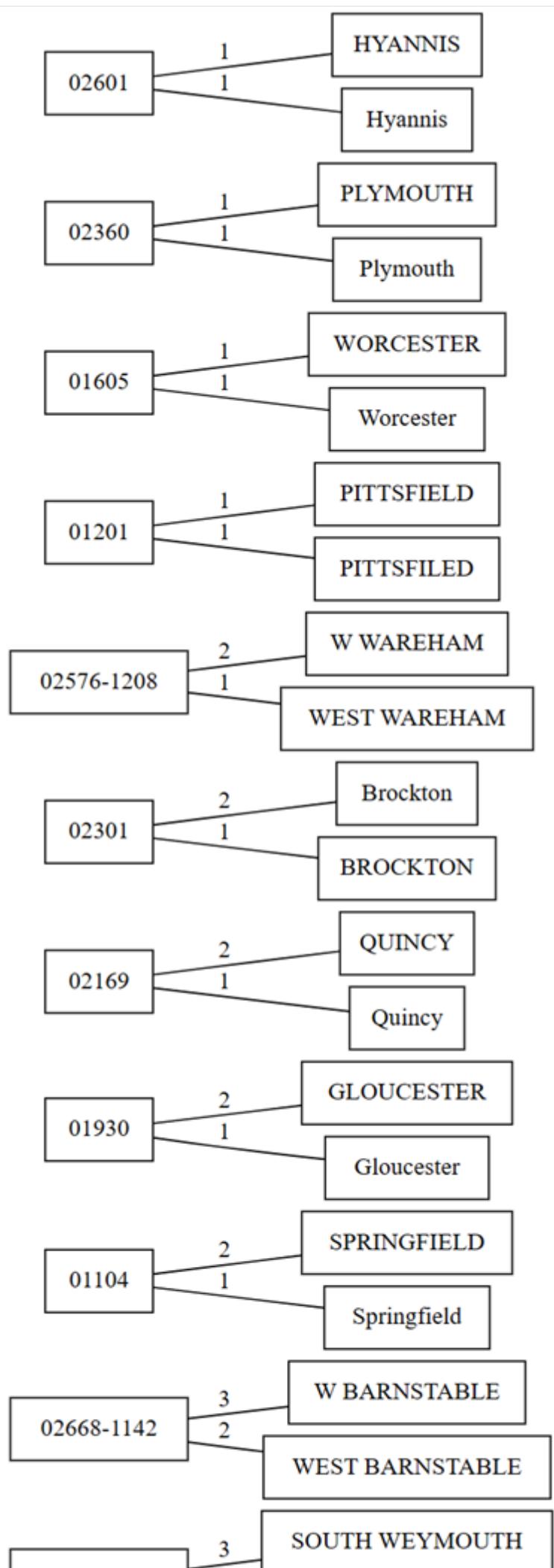
```
Python
```

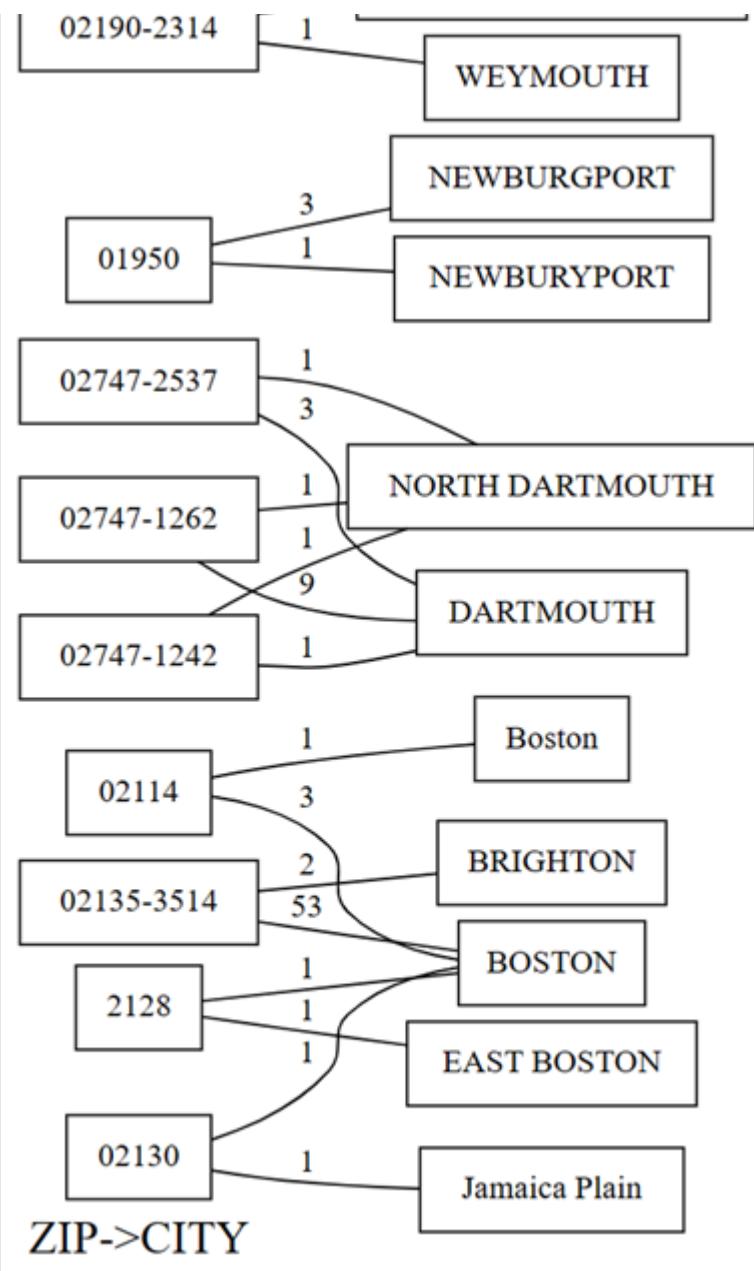
```
providers.list_dependencyViolations('ZIP', 'CITY')
```

2. Draw a graph with SemPy's `plot_dependency_violations` visualization function. This graph is helpful if the number of violations is small:

```
Python
```

```
providers.plot_dependencyViolations('ZIP', 'CITY')
```





The plot of dependency violations shows values for `ZIP` on the left hand side, and values for `CITY` on the right hand side. An edge connects a zip code on the left hand side of the plot with a city on the right hand side if there's a row that contains these two values. The edges are annotated with the count of such rows. For example, there are two rows with zip code 02747-1242, one row with city "NORTH DARTHMOUTH" and the other with city "DARTHMOUTH", as shown in the previous plot and the following code:

3. Confirm the previous observations you made with the plot of dependency violations by running the following code:

Python

```
providers[providers.ZIP == '02747-1242'].CITY.value_counts()
```

4. The plot also shows that among the rows that have `CITY` as "DARTMOUTH", nine rows have a `ZIP` of 02747-1262; one row has a `ZIP` of 02747-1242; and one row has a `ZIP` of 02747-2537. Confirms these observations with the following code:

```
Python
```

```
providers[providers.CITY == 'DARTMOUTH'].ZIP.value_counts()
```

5. There are other zip codes associated with "DARTMOUTH", but these zip codes aren't shown in the graph of dependency violations, as they don't hint at data quality issues. For example, the zip code "02747-4302" is uniquely associated to "DARTMOUTH" and doesn't show up in the graph of dependency violations. Confirm by running the following code:

```
Python
```

```
providers[providers.ZIP == '02747-4302'].CITY.value_counts()
```

Summarize data quality issues detected with SemPy

Going back to the graph of dependency violations, you can see that there are several interesting data quality issues present in this semantic model:

- Some city names are all uppercase. This issue is easy to fix using string methods.
- Some city names have qualifiers (or prefixes), such as "North" and "East". For example, the zip code "2128" maps to "EAST BOSTON" once and to "BOSTON" once. A similar issue occurs between "NORTH DARTMOUTH" and "DARTMOUTH". You could try to drop these qualifiers or map the zip codes to the city with the most common occurrence.
- There are typos in some cities, such as "PITTSFIELD" vs. "PITTSFILED" and "NEWBURGPORT" vs. "NEWBURYPORT". For "NEWBURGPORT" this typo could be fixed by using the most common occurrence. For "PITTSFIELD", having only one occurrence each makes it much harder for automatic disambiguation without external knowledge or the use of a language model.
- Sometimes, prefixes like "West" are abbreviated to a single letter "W". This issue could potentially be fixed with a simple replace, if all occurrences of "W" stand for "West".
- The zip code "02130" maps to "BOSTON" once and "Jamaica Plain" once. This issue isn't easy to fix, but if there was more data, mapping to the most common

occurrence could be a potential solution.

Clean the data

1. Fix the capitalization issues by changing all capitalization to title case:

```
Python
```

```
providers['CITY'] = providers.CITY.str.title()
```

2. Run the violation detection again to see that some of the ambiguities are gone (the number of violations is smaller):

```
Python
```

```
providers.list_dependencyViolations('ZIP', 'CITY')
```

At this point, you could refine your data more manually, but one potential data cleanup task is to drop rows that violate functional constraints between columns in the data, by using SemPy's `drop_dependency_violations` function.

For each value of the determinant variable, `drop_dependency_violations` works by picking the most common value of the dependent variable and dropping all rows with other values. You should apply this operation only if you're confident that this statistical heuristic would lead to the correct results for your data. Otherwise you should write your own code to handle the detected violations as needed.

3. Run the `drop_dependency_violations` function on the `ZIP` and `CITY` columns:

```
Python
```

```
providers_clean = providers.drop_dependencyViolations('ZIP', 'CITY')
```

4. List any dependency violations between `ZIP` and `CITY`:

```
Python
```

```
providers_clean.list_dependencyViolations('ZIP', 'CITY')
```

The code returns an empty list to indicate that there are no more violations of the functional constraint `CITY -> ZIP`.

Related content

Check out other tutorials for semantic link / SemPy:

- Tutorial: Analyze functional dependencies in a sample semantic model (preview)
 - Tutorial: Discover relationships in the *Synthea* dataset, using semantic link (preview)
 - Tutorial: Discover relationships in a semantic model, using semantic link (preview)
 - Tutorial: Extract and calculate Power BI measures from a Jupyter notebook (preview)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Tutorial: Analyze functional dependencies in a semantic model (preview)

Article • 11/15/2023

In this tutorial, you build upon prior work done by a Power BI analyst and stored in the form of semantic models (Power BI datasets). By using SemPy (preview) in the Synapse Data Science experience within Microsoft Fabric, you analyze functional dependencies that exist in columns of a DataFrame. This analysis helps to discover nontrivial data quality issues in order to gain more accurate insights.

Important

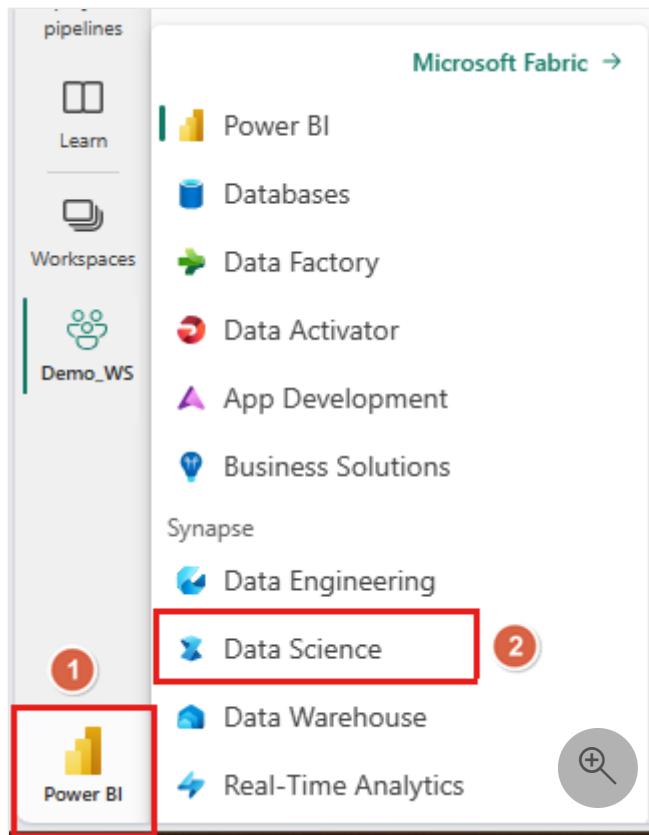
This feature is in **preview**.

In this tutorial, you learn how to:

- Apply domain knowledge to formulate hypotheses about functional dependencies in a semantic model.
- Get familiarized with components of semantic link's Python library ([SemPy](#)) that support integration with Power BI and help to automate data quality analysis. These components include:
 - `FabricDataFrame` - a pandas-like structure enhanced with additional semantic information.
 - Useful functions for pulling semantic models from a Fabric workspace into your notebook.
 - Useful functions that automate the evaluation of hypotheses about functional dependencies and that identify violations of relationships in your semantic models.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Switch to the Data Science experience by using the experience switcher icon on the left side of your home page.



- Select **Workspaces** from the left navigation pane to find and select your workspace. This workspace becomes your current workspace.
- Download the [*Customer Profitability Sample.pbix*](#) semantic model from the [fabric-samples GitHub repository](#) and upload it to your workspace.

Follow along in the notebook

The [`powerbi_dependencies_tutorial.ipynb`](#) notebook accompanies this tutorial.

If you want to open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science](#) to import the tutorial notebooks to your workspace.

Or, if you'd rather copy and paste the code from this page, you can [create a new notebook](#).

Be sure to [attach a lakehouse to the notebook](#) before you start running code.

Set up the notebook

In this section, you set up a notebook environment with the necessary modules and data.

1. Install `SemPy` from PyPI using the `%pip` in-line installation capability within the notebook:

```
Python
```

```
%pip install semantic-link
```

2. Perform necessary imports of modules that you'll need later:

```
Python
```

```
import sempy.fabric as fabric
from sempy.dependencies import plot_dependency_metadata
```

Load and preprocess the data

This tutorial uses a standard sample semantic model [Customer Profitability Sample.pbix](#). For a description of the semantic model, see [Customer Profitability sample for Power BI](#).

1. Load the Power BI data into FabricDataFrames, using SemPy's `read_table` function:

```
Python
```

```
dataset = "Customer Profitability Sample"
customer = fabric.read_table(dataset, "Customer")
customer.head()
```

2. Load the `State` table into a FabricDataFrame:

```
Python
```

```
state = fabric.read_table(dataset, "State")
state.head()
```

While the output of this code looks like a pandas DataFrame, you've actually initialized a data structure called a `FabricDataFrame` that supports some useful operations on top of pandas.

3. Check the data type of `customer`:

```
Python
```

```
type(customer)
```

The output confirms that `customer` is of type
`semopy.fabric._dataframe._fabric_dataframe.FabricDataFrame``

4. Join the `customer` and `state` DataFrames:

Python

```
customer_state_df = customer.merge(state, left_on="State",
right_on="StateCode", how='left')
customer_state_df.head()
```

Identify functional dependencies

A functional dependency manifests itself as a one-to-many relationship between the values in two (or more) columns within a DataFrame. These relationships can be used to automatically detect data quality problems.

1. Run SemPy's `find_dependencies` function on the merged DataFrame to identify any existing functional dependencies between values in the columns:

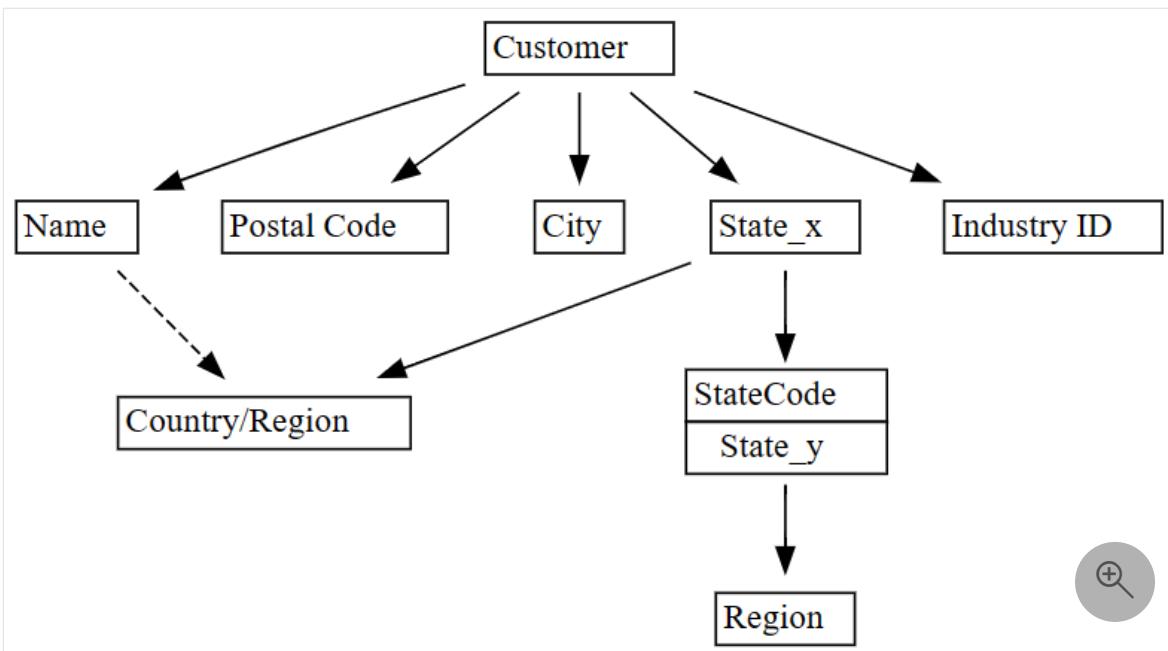
Python

```
dependencies = customer_state_df.find_dependencies()
dependencies
```

2. Visualize the identified dependencies by using SemPy's `plot_dependency_metadata` function:

Python

```
plot_dependency_metadata(dependencies)
```



As expected, the functional dependencies graph shows that the `Customer` column determines some columns like `City`, `Postal Code`, and `Name`.

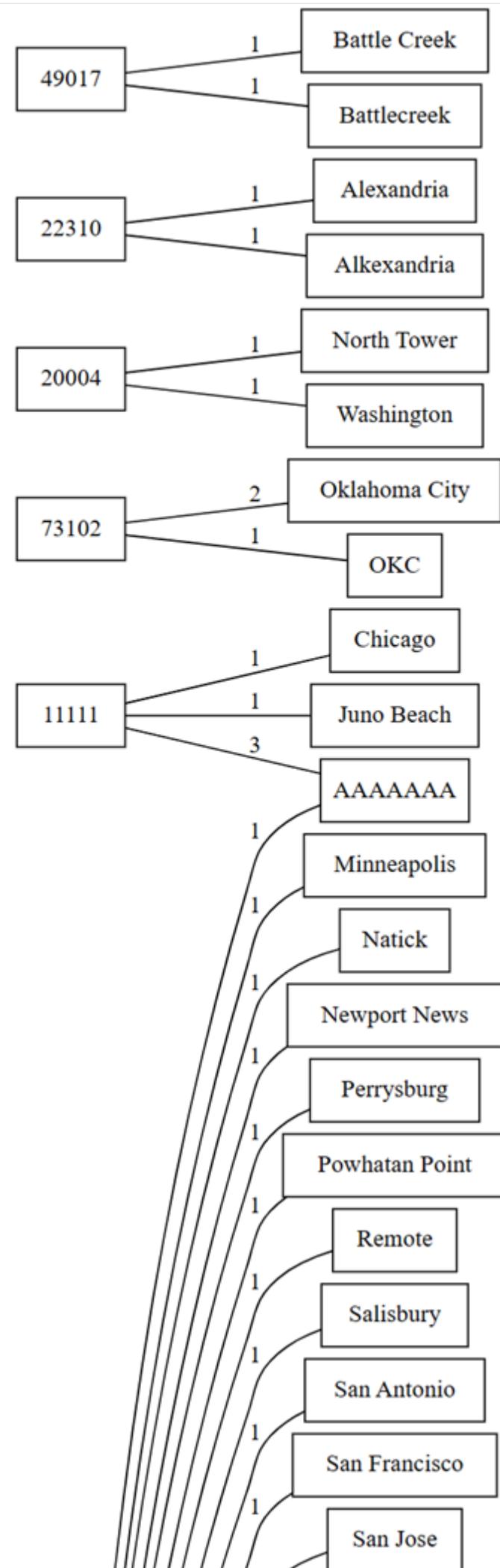
Surprisingly, the graph doesn't show a functional dependency between `City` and `Postal Code`, probably because there are many violations in the relationships between the columns. You can use SemPy's `plot_dependency_violations` function to visualize violations of dependencies between specific columns.

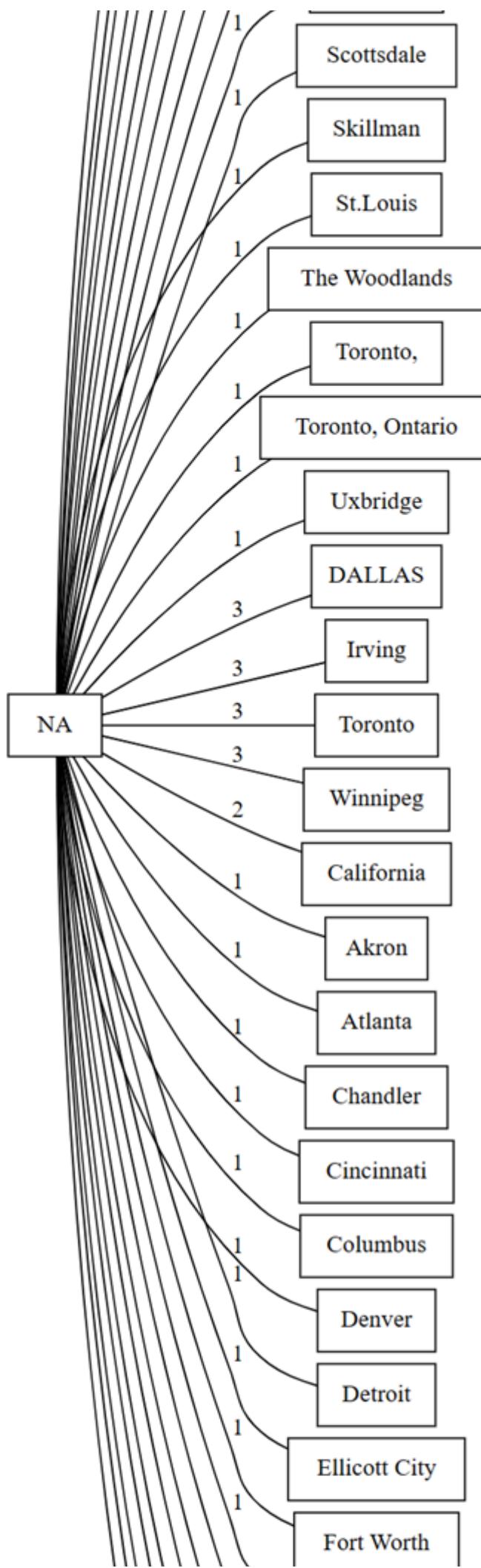
Explore the data for quality issues

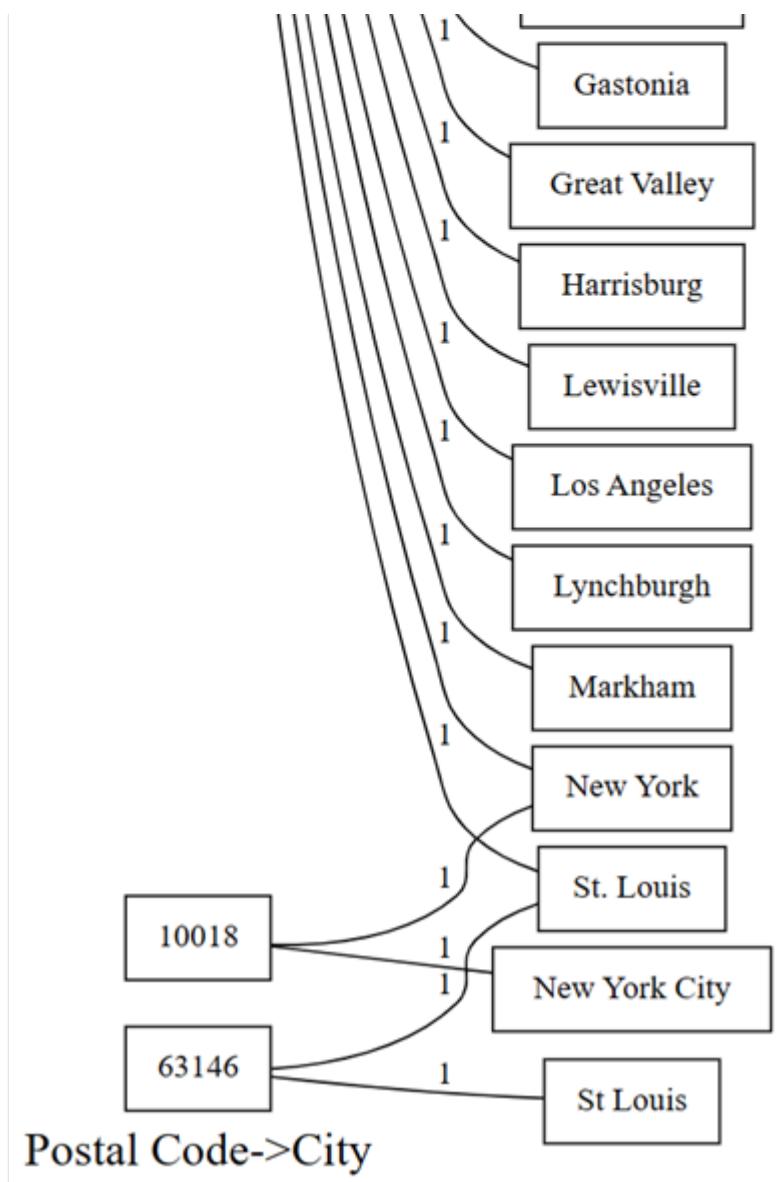
1. Draw a graph with SemPy's `plot_dependency_violations` visualization function.

Python

```
customer_state_df.plot_dependencyViolations('Postal Code', 'City')
```







The plot of dependency violations shows values for `Postal Code` on the left hand side, and values for `city` on the right hand side. An edge connects a `Postal Code` on the left hand side with a `city` on the right hand side if there's a row that contains these two values. The edges are annotated with the count of such rows. For example, there are two rows with postal code 20004, one with city "North Tower" and the other with city "Washington".

Moreover, the plot shows a few violations and many empty values.

2. Confirm the number of empty values for `Postal Code`:

Python

```
customer_state_df['Postal Code'].isna().sum()
```

50 rows have NA for postal code.

3. Drop rows with empty values. Then, find dependencies using the `find_dependencies` function. Notice the extra parameter `verbose=1` that offers a glimpse into the internal workings of SemPy:

Python

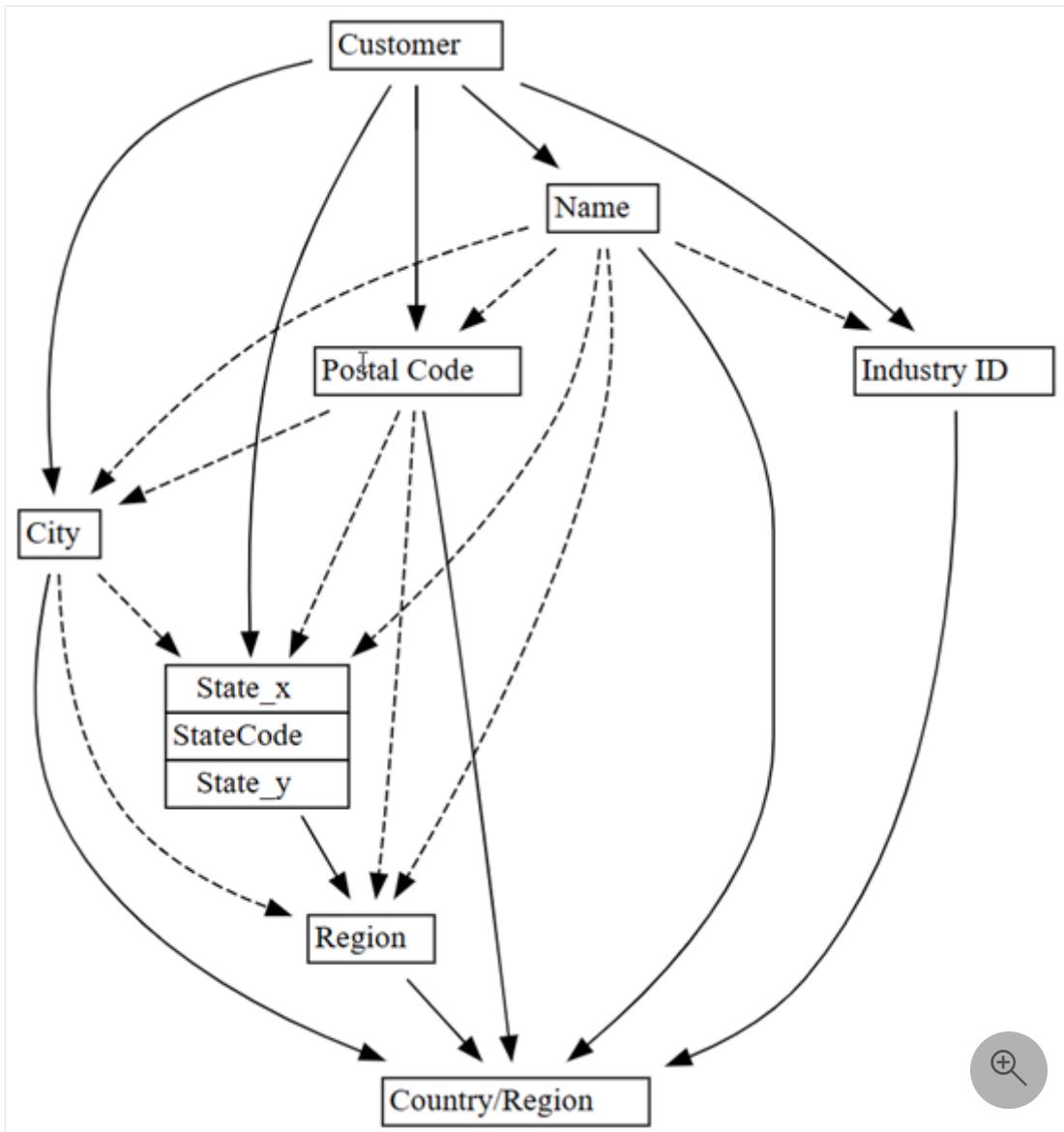
```
customer_state_df2=customer_state_df.dropna()  
customer_state_df2.find_dependencies(verbose=1)
```

The conditional entropy for `Postal Code` and `City` is 0.049. This value indicates that there are functional dependency violations. Before you fix the violations, raise the threshold on conditional entropy from the default value of `0.01` to `0.05`, just to see the dependencies. Lower thresholds result in fewer dependencies (or higher selectivity).

4. Raise the threshold on conditional entropy from the default value of `0.01` to `0.05`:

Python

```
plot_dependency_metadata(customer_state_df2.find_dependencies(threshold  
=0.05))
```



If you apply domain knowledge of which entity determines values of other entities, this dependencies graph seems accurate.

- Explore more data quality issues that were detected. For example, a dashed arrow joins `City` and `Region`, which indicates that the dependency is only approximate. This approximate relationship could imply that there's a partial functional dependency.

Python

```
customer_state_df.list_dependencyViolations('City', 'Region')
```

- Take a closer look at each of the cases where a nonempty `Region` value causes a violation:

Python

```
customer_state_df[customer_state_df.City=='Downers Grove']
```

The result shows Downers Grove city occurring in Illinois and Nebraska. However, Downer's Grove is a [city in Illinois](#), not Nebraska.

7. Take a look at the city of *Fremont*:

Python

```
customer_state_df[customer_state_df.City=='Fremont']
```

There's a city called [Fremont in California](#). However, for Texas, the search engine returns [Premont](#), not Fremont.

8. It's also suspicious to see violations of the dependency between `Name` and `Country/Region`, as signified by the dotted line in the original graph of dependency violations (before dropping the rows with empty values).

Python

```
customer_state_df.list_dependencyViolations('Name', 'Country/Region')
```

It appears that one customer, *SDI Design* is present in two regions - United States and Canada. This occurrence may not be a semantic violation, but may just be an uncommon case. Still, it's worth taking a close look:

9. Take a closer look at the customer *SDI Design*:

Python

```
customer_state_df[customer_state_df.Name=='SDI Design']
```

Further inspection shows that it's actually two different customers (from different industries) with the same name.

Exploratory data analysis is an exciting process, and so is data cleaning. There's always something that the data is hiding, depending on how you look at it, what you want to ask, and so on. Semantic link provides you with new tools that you can use to achieve more with your data.

Related content

Check out other tutorials for semantic link / SemPy:

- Tutorial: Clean data with functional dependencies (preview)
 - Tutorial: Discover relationships in the *Synthea* dataset using semantic link (preview)
 - Tutorial: Discover relationships in a semantic model using semantic link (preview)
 - Tutorial: Extract and calculate Power BI measures from a Jupyter notebook (preview)
-

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Ask the community ↗](#)

Tutorial: Extract and calculate Power BI measures from a Jupyter notebook

Article • 01/15/2025

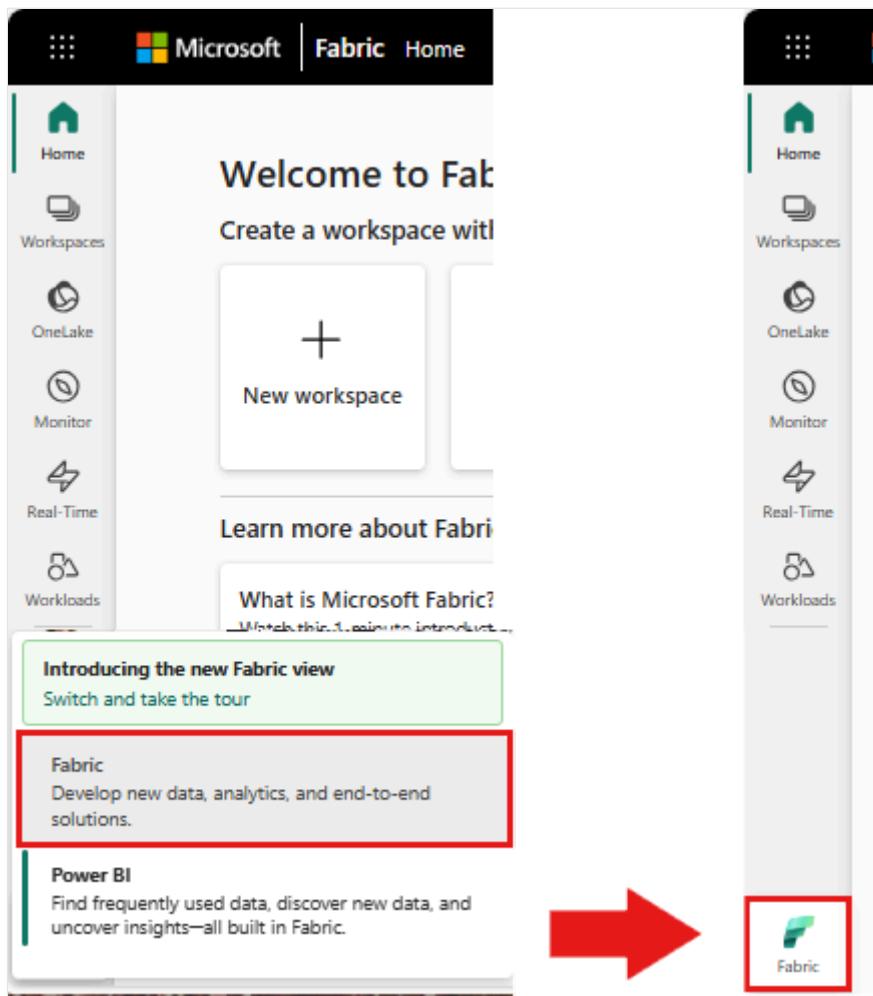
This tutorial illustrates how to use SemPy (preview) to calculate measures in semantic models (Power BI datasets).

In this tutorial, you learn how to:

- Evaluate Power BI measures programmatically via a Python interface of semantic link's Python library ([SemPy](#)).
- Get familiarized with components of SemPy that help to bridge the gap between AI and BI. These components include:
 - FabricDataFrame - a pandas-like structure enhanced with additional semantic information.
 - Useful functions that allow you to fetch semantic models, including raw data, configurations, and measures.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- Select **Workspaces** from the left navigation pane to find and select your workspace. This workspace becomes your current workspace.
- Download the [Retail Analysis Sample PBIX.pbix](#) semantic model and upload it to your workspace.

Follow along in the notebook

The [powerbi_measures_tutorial.ipynb](#) notebook accompanies this tutorial.

- To open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#) to import the notebook to your workspace.
- If you'd rather copy and paste the code from this page, you can [create a new notebook](#).
- Be sure to [attach a lakehouse to the notebook](#) before you start running code.

Set up the notebook

In this section, you set up a notebook environment with the necessary modules and data.

1. Install `SemPy` from PyPI using the `%pip` in-line installation capability within the notebook:

```
Python  
%pip install semantic-link
```

2. Perform necessary imports of modules that you'll need later:

```
Python  
import sempy.fabric as fabric
```

3. You can connect to the Power BI workspace. List the semantic models in the workspace:

```
Python  
fabric.list_datasets()
```

4. Load the semantic model. In this tutorial, you use the *Retail Analysis Sample PBIX* semantic model:

```
Python  
dataset = "Retail Analysis Sample PBIX"
```

List workspace measures

List measures in the semantic model, using SemPy's `list_measures` function as follows:

```
Python  
fabric.list_measures(dataset)
```

Evaluate measures

In this section, you evaluate measures in various ways, using SemPy's `evaluate_measure` function.

Evaluate a raw measure

In the following code, use SemPy's `evaluate_measure` function to calculate a preconfigured measure that is called "Average Selling Area Size". You can see the underlying formula for this measure in the output of the previous cell.

Python

```
fabric.evaluate_measure(dataset, measure="Average Selling Area Size")
```

Evaluate a measure with `groupby_columns`

You can group the measure output by certain columns by supplying the extra parameter `groupby_columns`:

Python

```
fabric.evaluate_measure(dataset, measure="Average Selling Area Size",
groupby_columns=["Store[Chain]", "Store[DistrictName]"])
```

In the previous code, you grouped by the columns `Chain` and `DistrictName` of the `Store` table in the semantic model.

Evaluate a measure with filters

You can also use the `filters` parameter to specify specific values that the result can contain for particular columns:

Python

```
fabric.evaluate_measure(dataset, \
    measure="Total Units Last Year", \
    groupby_columns=["Store[Territory]"], \
    filters={"Store[Territory)": ["PA", "TN", "VA"], \
    "Store[Chain)": ["Lindseys"]})
```

In the previous code, `Store` is the name of the table, `Territory` is the name of the column, and `PA` is one of the values that the filter allows.

Evaluate a measure across multiple tables

You can group the measure by columns that span across multiple tables in the semantic model.

Python

```
fabric.evaluate_measure(dataset, measure="Total Units Last Year",
groupby_columns=["Store[Territory]", "Sales[ItemID]"])
```

Evaluate multiple measures

The function `evaluate_measure` allows you to supply identifiers of multiple measures and output the calculated values in the same DataFrame:

Python

```
fabric.evaluate_measure(dataset, measure=[ "Average Selling Area Size",
"Total Stores"], groupby_columns=[ "Store[Chain]", "Store[DistrictName]"])
```

Use Power BI XMLA connector

The default semantic model client is backed by Power BI's REST APIs. If there are any issues running queries with this client, it's possible to switch the backend to Power BI's XMLA interface using `use_xmla=True`. The SemPy parameters remain the same for measure calculation with XMLA.

Python

```
fabric.evaluate_measure(dataset, \
                         measure=[ "Average Selling Area Size", "Total
Stores"], \
                         groupby_columns=[ "Store[Chain]", \
"Store[DistrictName]"], \
                         filters={"Store[Territory]": ["PA", "TN", "VA"], \
"Store[Chain]": ["Lindseys"]}, \
                         use_xmla=True)
```

Related content

Check out other tutorials for semantic link / SemPy:

- Tutorial: Clean data with functional dependencies
 - Tutorial: Analyze functional dependencies in a sample semantic model
 - Tutorial: Discover relationships in a semantic model, using semantic link
 - Tutorial: Discover relationships in the *Synthea* dataset, using semantic link
 - Tutorial: Validate data using SemPy and Great Expectations (GX)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Tutorial: Discover relationships in a semantic model, using semantic link

Article • 04/28/2024

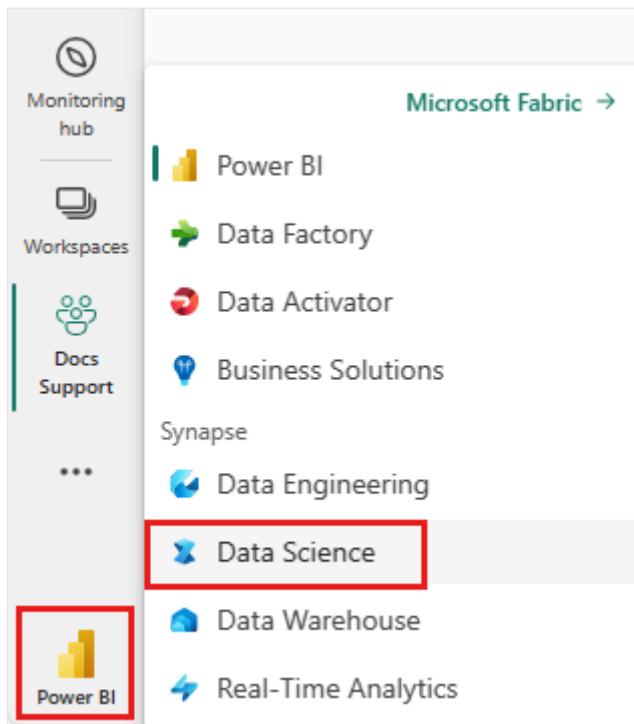
This tutorial illustrates how to interact with Power BI from a Jupyter notebook and detect relationships between tables with the help of the SemPy library.

In this tutorial, you learn how to:

- Discover relationships in a semantic model (Power BI dataset), using semantic link's Python library ([SemPy](#)).
- Use components of SemPy that support integration with Power BI and help to automate data quality analysis. These components include:
 - FabricDataFrame - a pandas-like structure enhanced with additional semantic information.
 - Functions for pulling semantic models from a Fabric workspace into your notebook.
 - Functions that automate the evaluation of hypotheses about functional dependencies and that identify violations of relationships in your semantic models.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



- Select **Workspaces** from the left navigation pane to find and select your workspace. This workspace becomes your current workspace.
- Download the *Customer Profitability Sample.pbix* and *Customer Profitability Sample (auto).pbix* semantic models from the [fabric-samples GitHub repository](#) and upload them to your workspace.

Follow along in the notebook

The [powerbi_relationships_tutorial.ipynb](#) notebook accompanies this tutorial.

To open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#), to import the notebook to your workspace.

If you'd rather copy and paste the code from this page, you can [create a new notebook](#).

Be sure to [attach a lakehouse to the notebook](#) before you start running code.

Set up the notebook

In this section, you set up a notebook environment with the necessary modules and data.

1. Install `SemPy` from PyPI using the `%pip` in-line installation capability within the notebook:

Python

```
%pip install semantic-link
```

2. Perform necessary imports of SemPy modules that you'll need later:

Python

```
import sempy.fabric as fabric

from sempy.relationships import plot_relationship_metadata
from sempy.relationships import find_relationships
from sempy.fabric import list_relationship_violations
```

3. Import pandas for enforcing a configuration option that helps with output formatting:

Python

```
import pandas as pd
pd.set_option('display.max_colwidth', None)
```

Explore semantic models

This tutorial uses a standard sample semantic model [Customer Profitability Sample.pbix](#). For a description of the semantic model, see [Customer Profitability sample for Power BI](#).

- Use SemPy's `list_datasets` function to explore semantic models in your current workspace:

Python

```
fabric.list_datasets()
```

For the rest of this notebook you use two versions of the Customer Profitability Sample semantic model:

- *Customer Profitability Sample*: the semantic model as it comes from Power BI samples with predefined table relationships
- *Customer Profitability Sample (auto)*: the same data, but relationships are limited to those ones that Power BI would autodetect.

Extract a sample semantic model with its predefined semantic model

1. Load relationships that are predefined and stored within the *Customer Profitability Sample* semantic model, using SemPy's `list_relationships` function. This function lists from the Tabular Object Model:

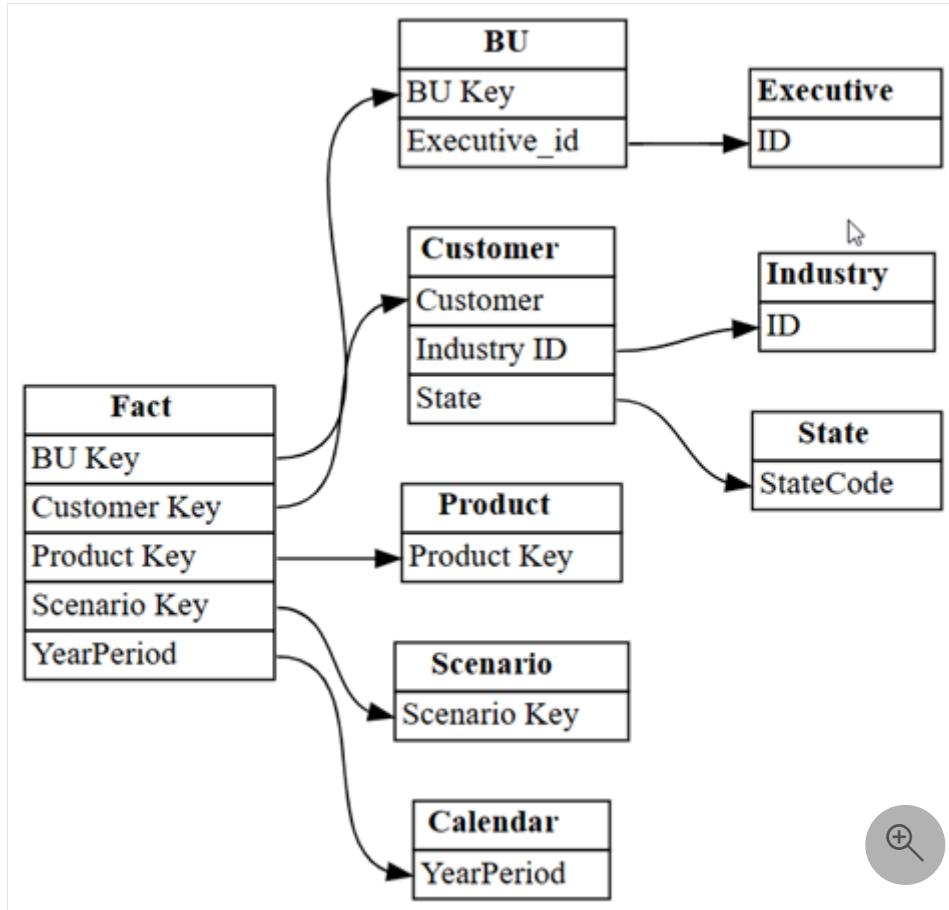
```
Python
```

```
dataset = "Customer Profitability Sample"  
relationships = fabric.list_relationships(dataset)  
relationships
```

2. Visualize the `relationships` DataFrame as a graph, using SemPy's `plot_relationship_metadata` function:

```
Python
```

```
plot_relationship_metadata(relationships)
```



This graph shows the "ground truth" for relationships between tables in this semantic model, as it reflects how they were defined in Power BI by a subject matter expert.

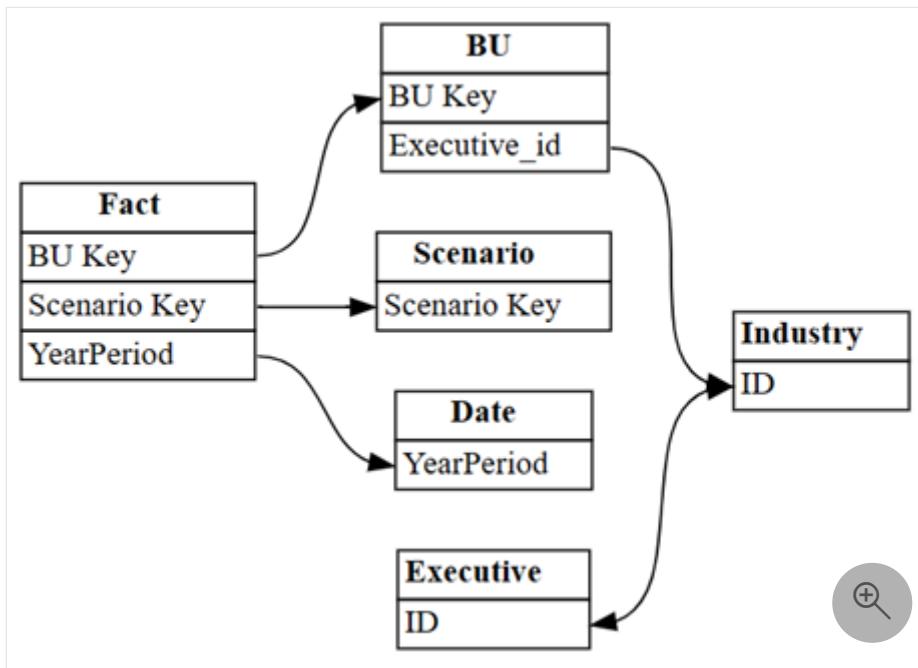
Complement relationships discovery

If you started with relationships that Power BI autodetected, you'd have a smaller set.

1. Visualize the relationships that Power BI autodetected in the semantic model:

Python

```
dataset = "Customer Profitability Sample (auto)"
autodetected = fabric.list_relationships(dataset)
plot_relationship_metadata(autodetected)
```



Power BI's autodetection missed many relationships. Moreover, two of the autodetected relationships are semantically incorrect:

- Executive[ID] -> Industry[ID]
- BU[Executive_id] -> Industry[ID]

2. Print the relationships as a table:

Python

```
autodetected
```

Incorrect relationships to the Industry table appear in rows with index 3 and 4. Use this information to remove these rows.

3. Discard the incorrectly identified relationships.

```
Python
```

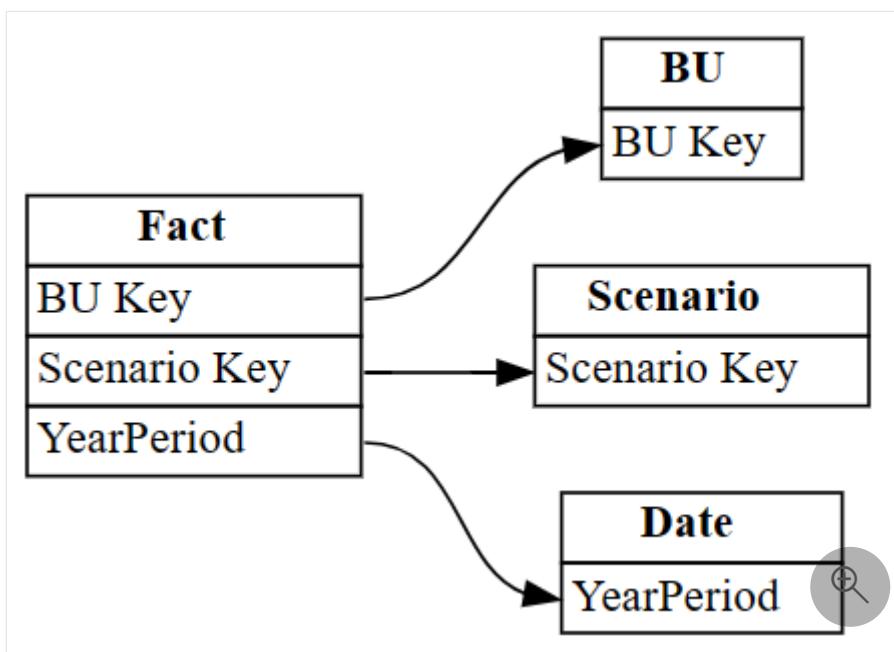
```
autodetected.drop(index=[3,4], inplace=True)  
autodetected
```

Now you have correct, but incomplete relationships.

4. Visualize these incomplete relationships, using `plot_relationship_metadata`:

```
Python
```

```
plot_relationship_metadata(autodetected)
```



5. Load all the tables from the semantic model, using SemPy's `list_tables` and `read_table` functions:

```
Python
```

```
tables = {table: fabric.read_table(dataset, table) for table in  
fabric.list_tables(dataset)[ 'Name' ]}  
  
tables.keys()
```

6. Find relationships between tables, using `find_relationships`, and review the log output to get some insights into how this function works:

```
Python
```

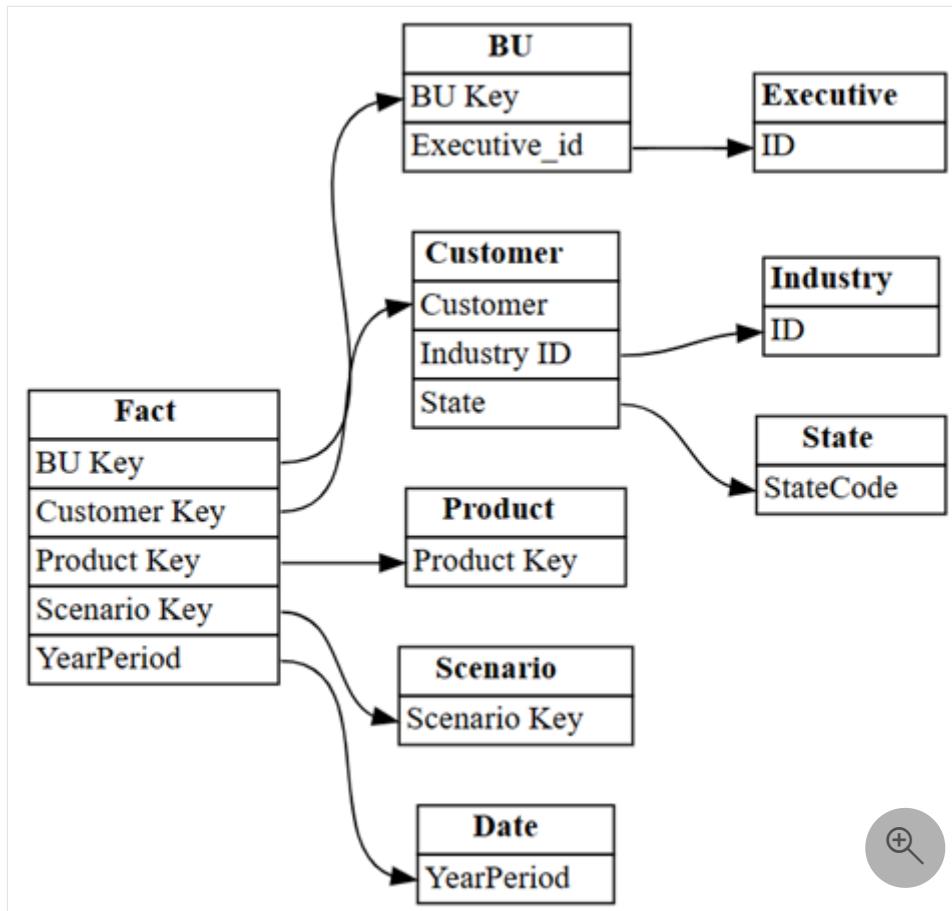
```
suggested_relationships_all = find_relationships(  
    tables,
```

```
        name_similarity_threshold=0.7,  
        coverage_threshold=0.7,  
        verbose=2  
)
```

7. Visualize newly discovered relationships:

Python

```
plot_relationship_metadata(suggested_relationships_all)
```



SemPy was able to detect all relationships.

8. Use the `exclude` parameter to limit the search to additional relationships that weren't identified previously:

Python

```
additional_relationships = find_relationships(  
    tables,  
    exclude=autodetected,  
    name_similarity_threshold=0.7,  
    coverage_threshold=0.7  
)
```

```
additional_relationships
```

Validate the relationships

1. First, load the data from the *Customer Profitability Sample* semantic model:

```
Python
```

```
dataset = "Customer Profitability Sample"
tables = {table: fabric.read_table(dataset, table) for table in
          fabric.list_tables(dataset)[ 'Name' ]}

tables.keys()
```

2. Check for overlap of primary and foreign key values by using the `list_relationshipViolations` function. Supply the output of the `listRelationships` function as input to `listRelationshipViolations`:

```
Python
```

```
listRelationshipViolations(tables,
                           fabric.listRelationships(dataset))
```

The relationship violations provide some interesting insights. For example, one out of seven values in `Fact[Product Key]` isn't present in `Product[Product Key]`, and this missing key is `50`.

Exploratory data analysis is an exciting process, and so is data cleaning. There's always something that the data is hiding, depending on how you look at it, what you want to ask, and so on. Semantic link provides you with new tools that you can use to achieve more with your data.

Related content

Check out other tutorials for semantic link / SemPy:

- [Tutorial: Clean data with functional dependencies](#)
- [Tutorial: Analyze functional dependencies in a sample semantic model](#)
- [Tutorial: Extract and calculate Power BI measures from a Jupyter notebook](#)
- [Tutorial: Discover relationships in the *Synthea* dataset, using semantic link](#)
- [Tutorial: Validate data using SemPy and Great Expectations \(GX\) \(preview\)](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Tutorial: Discover relationships in the *Synthea* dataset, using semantic link (preview)

Article • 11/15/2023

This tutorial illustrates how to detect relationships in the public *Synthea* dataset, using semantic link (preview).

Important

This feature is in **preview**.

When you're working with new data or working without an existing data model, it can be helpful to discover relationships automatically. This relationship detection can help you to:

- understand the model at a high level,
- gain more insights during exploratory data analysis,
- validate updated data or new, incoming data, and
- clean data.

Even if relationships are known in advance, a search for relationships can help with better understanding of the data model or identification of data quality issues.

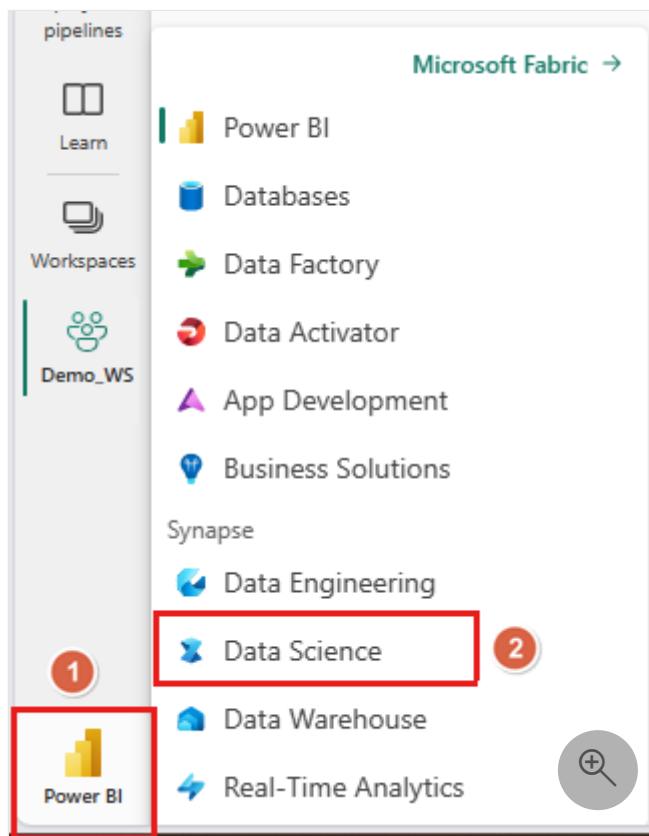
In this tutorial, you begin with a simple baseline example where you experiment with only three tables so that connections between them are easy to follow. Then, you show a more complex example with a larger table set.

In this tutorial, you learn how to:

- Use components of semantic link's Python library ([SemPy](#)) that support integration with Power BI and help to automate data analysis. These components include:
 - FabricDataFrame - a pandas-like structure enhanced with additional semantic information.
 - Functions for pulling semantic models from a Fabric workspace into your notebook.
 - Functions that automate the discovery and visualization of relationships in your semantic models.
- Troubleshoot the process of relationship discovery for semantic models with multiple tables and interdependencies.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Switch to the Data Science experience by using the experience switcher icon on the left side of your home page.



- Select **Workspaces** from the left navigation pane to find and select your workspace. This workspace becomes your current workspace.

Follow along in the notebook

The [relationships_detection_tutorial.ipynb](#) notebook accompanies this tutorial.

If you want to open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science](#) to import the tutorial notebooks to your workspace.

Or, if you'd rather copy and paste the code from this page, you can [create a new notebook](#).

Be sure to [attach a lakehouse to the notebook](#) before you start running code.

Set up the notebook

In this section, you set up a notebook environment with the necessary modules and data.

1. Install `SemPy` from PyPI using the `%pip` in-line installation capability within the notebook:

```
Python
```

```
%pip install semantic-link
```

2. Perform necessary imports of SemPy modules that you'll need later:

```
Python
```

```
import pandas as pd

from sempy.samples import download_synthea
from sempy.relationships import (
    find_relationships,
    list_relationshipViolations,
    plot_relationship_metadata
)
```

3. Import pandas for enforcing a configuration option that helps with output formatting:

```
Python
```

```
import pandas as pd
pd.set_option('display.max_colwidth', None)
```

4. Pull the sample data. For this tutorial, you use the *Synthea* dataset of synthetic medical records (small version for simplicity):

```
Python
```

```
download_synthea(which='small')
```

Detect relationships on a small subset of *Synthea* tables

1. Select three tables from a larger set:

- `patients` specifies patient information
- `encounters` specifies the patients that had medical encounters (for example, a medical appointment, procedure)
- `providers` specifies which medical providers attended to the patients

The `encounters` table resolves a many-to-many relationship between `patients` and `providers` and can be described as an [associative entity](#):

Python

```
patients = pd.read_csv('synthea/csv/patients.csv')
providers = pd.read_csv('synthea/csv/providers.csv')
encounters = pd.read_csv('synthea/csv/encounters.csv')
```

2. Find relationships between the tables using SemPy's `find_relationships` function:

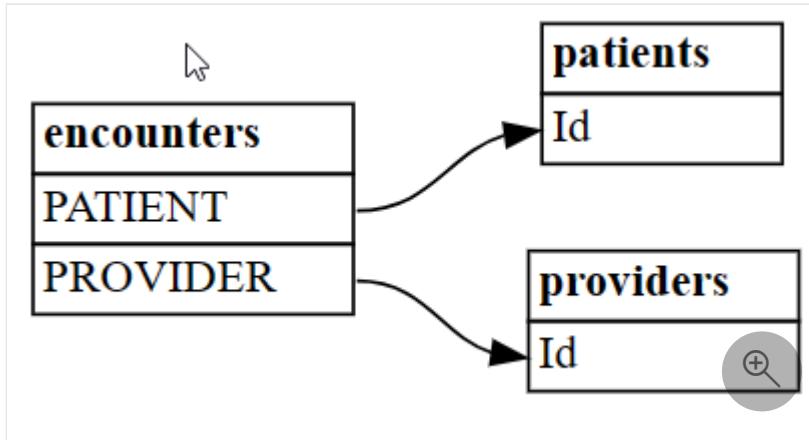
Python

```
suggested_relationships = find_relationships([patients, providers,
encounters])
suggested_relationships
```

3. Visualize the relationships DataFrame as a graph, using SemPy's `plot_relationship_metadata` function.

Python

```
plot_relationship_metadata(suggested_relationships)
```



The function lays out the relationship hierarchy from the left hand side to the right hand side, which corresponds to "from" and "to" tables in the output. In other

words, the independent "from" tables on the left hand side use their foreign keys to point to their "to" dependency tables on the right hand side. Each entity box shows columns that participate on either the "from" or "to" side of a relationship.

By default, relationships are generated as "m:1" (not as "1:m") or "1:1". The "1:1" relationships can be generated one or both ways, depending on if the ratio of mapped values to all values exceeds `coverage_threshold` in just one or both directions. Later in this tutorial, you cover the less frequent case of "m:m" relationships.

Troubleshoot relationship detection issues

The baseline example shows a successful relationship detection on clean *Synthea* data. In practice, the data is rarely clean, which prevents successful detection. There are several techniques that can be useful when the data isn't clean.

This section of this tutorial addresses relationship detection when the semantic model contains dirty data.

1. Begin by manipulating the original DataFrames to obtain "dirty" data, and print the size of the dirty data.

Python

```
# create a dirty 'patients' dataframe by dropping some rows using
# head() and duplicating some rows using concat()
patients_dirty = pd.concat([patients.head(1000), patients.head(50)],
axis=0)

# create a dirty 'providers' dataframe by dropping some rows using
# head()
providers_dirty = providers.head(5000)

# the dirty dataframes have fewer records than the clean ones
print(len(patients_dirty))
print(len(providers_dirty))
```

2. For comparison, print sizes of the original tables:

Python

```
print(len(patients))
print(len(providers))
```

3. Find relationships between the tables using SemPy's `find_relationships` function:

Python

```
find_relationships([patients_dirty, providers_dirty, encounters])
```

The output of the code shows that there's no relationships detected due to the errors that you introduced earlier to create the "dirty" semantic model.

Use validation

Validation is the best tool for troubleshooting relationship detection failures because:

- It reports clearly why a particular relationship doesn't follow the Foreign Key rules and therefore can't be detected.
- It runs fast with large semantic models because it focuses only on the declared relationships and doesn't perform a search.

Validation can use any DataFrame with columns similar to the one generated by `find_relationships`. In the following code, the `suggested_relationships` DataFrame refers to `patients` rather than `patients_dirty`, but you can alias the DataFrames with a dictionary:

Python

```
dirty_tables = {
    "patients": patients_dirty,
    "providers" : providers_dirty,
    "encounters": encounters
}

errors = list_relationshipViolations(dirty_tables, suggested_relationships)
errors
```

Loosen search criteria

In more murky scenarios, you can try loosening your search criteria. This method increases the possibility of false positives.

1. Set `include_many_to_many=True` and evaluate if it helps:

Python

```
find_relationships(dirty_tables, include_many_to_many=True,  
coverage_threshold=1)
```

The results show that the relationship from `encounters` to `patients` was detected, but there are two problems:

- The relationship indicates a direction from `patients` to `encounters`, which is an inverse of the expected relationship. This is because all `patients` happened to be covered by `encounters` (`Coverage From` is 1.0) while `encounters` are only partially covered by `patients` (`Coverage To` = 0.85), since `patients` rows are missing.
- There's an accidental match on a low cardinality `GENDER` column, which happens to match by name and value in both tables, but it isn't an "m:1" relationship of interest. The low cardinality is indicated by `Unique Count From` and `Unique Count To` columns.

2. Rerun `find_relationships` to look only for "m:1" relationships, but with a lower `coverage_threshold=0.5`:

Python

```
find_relationships(dirty_tables, include_many_to_many=False,  
coverage_threshold=0.5)
```

The result shows the correct direction of the relationships from `encounters` to `providers`. However, the relationship from `encounters` to `patients` isn't detected, because `patients` isn't unique, so it can't be on the "One" side of "m:1" relationship.

3. Loosen both `include_many_to_many=True` and `coverage_threshold=0.5`:

Python

```
find_relationships(dirty_tables, include_many_to_many=True,  
coverage_threshold=0.5)
```

Now both relationships of interest are visible, but there's a lot more noise:

- The low cardinality match on `GENDER` is present.
- A higher cardinality "m:m" match on `ORGANIZATION` appeared, making it apparent that `ORGANIZATION` is likely a column de-normalized to both tables.

Match column names

By default, SemPy considers as matches only attributes that show name similarity, taking advantage of the fact that database designers usually name related columns the same way. This behavior helps to avoid spurious relationships, which occur most frequently with low cardinality integer keys. For example, if there are `1,2,3,...,10` product categories and `1,2,3,...,10` order status code, they'll be confused with each other when only looking at value mappings without taking column names into account. Spurious relationships shouldn't be a problem with GUID-like keys.

SemPy looks at a similarity between column names and table names. The matching is approximate and case insensitive. It ignores the most frequently encountered "decorator" substrings such as "id", "code", "name", "key", "pk", "fk". As a result, the most typical match cases are:

- an attribute called 'column' in entity 'foo' matches with an attribute called 'column' (also 'COLUMN' or 'Column') in entity 'bar'.
- an attribute called 'column' in entity 'foo' matches with an attribute called 'column_id' in 'bar'.
- an attribute called 'bar' in entity 'foo' matches with an attribute called 'code' in 'bar'.

By matching column names first, the detection runs faster.

1. Match the column names:

- To understand which columns are selected for further evaluation, use the `verbose=2` option (`verbose=1` lists only the entities being processed).
- The `name_similarity_threshold` parameter determines how columns are compared. The threshold of 1 indicates that you're interested in 100% match only.

Python

```
find_relationships(dirty_tables, verbose=2,  
name_similarity_threshold=1.0);
```

Running at 100% similarity fails to account for small differences between names. In your example, the tables have a plural form with "s" suffix, which results in no exact match. This is handled well with the default `name_similarity_threshold=0.8`.

2. Rerun with the default `name_similarity_threshold=0.8`:

```
Python
```

```
find_relationships(dirty_tables, verbose=2,  
name_similarity_threshold=0.8);
```

Notice that the Id for plural form `patients` is now compared to singular `patient` without adding too many other spurious comparisons to the execution time.

3. Rerun with the default `name_similarity_threshold=0`:

```
Python
```

```
find_relationships(dirty_tables, verbose=2,  
name_similarity_threshold=0);
```

Changing `name_similarity_threshold` to 0 is the other extreme, and it indicates that you want to compare all columns. This is rarely necessary and results in increased execution time and spurious matches that need to be reviewed. Observe the number of comparisons in the verbose output.

Summary of troubleshooting tips

1. Start from exact match for "m:1" relationships (that is, the default `include_many_to_many=False` and `coverage_threshold=1.0`). This is usually what you want.
2. Use a narrow focus on smaller subsets of tables.
3. Use validation to detect data quality issues.
4. Use `verbose=2` if you want to understand which columns are considered for relationship. This can result in a large amount of output.
5. Be aware of trade-offs of search arguments. `include_many_to_many=True` and `coverage_threshold<1.0` may produce spurious relationships that may be harder to analyze and will need to be filtered.

Detect relationships on the full *Synthea* dataset

The simple baseline example was a convenient learning and troubleshooting tool. In practice, you may start from a semantic model such as the full *Synthea* dataset, which has a lot more tables. Explore the full *synthea* dataset as follows.

1. Read all files from the *synthea/csv* directory:

Python

```
all_tables = {
    "allergies": pd.read_csv('synthea/csv/allergies.csv'),
    "careplans": pd.read_csv('synthea/csv/careplans.csv'),
    "conditions": pd.read_csv('synthea/csv/conditions.csv'),
    "devices": pd.read_csv('synthea/csv/devices.csv'),
    "encounters": pd.read_csv('synthea/csv/encounters.csv'),
    "imaging_studies": pd.read_csv('synthea/csv/imaging_studies.csv'),
    "immunizations": pd.read_csv('synthea/csv/immunizations.csv'),
    "medications": pd.read_csv('synthea/csv/medications.csv'),
    "observations": pd.read_csv('synthea/csv/observations.csv'),
    "organizations": pd.read_csv('synthea/csv/organizations.csv'),
    "patients": pd.read_csv('synthea/csv/patients.csv'),
    "payer_transitions":
        pd.read_csv('synthea/csv/payer_transitions.csv'),
    "payers": pd.read_csv('synthea/csv/payers.csv'),
    "procedures": pd.read_csv('synthea/csv/procedures.csv'),
    "providers": pd.read_csv('synthea/csv/providers.csv'),
    "supplies": pd.read_csv('synthea/csv/supplies.csv'),
}
```

2. Find relationships between the tables, using SemPy's `find_relationships` function:

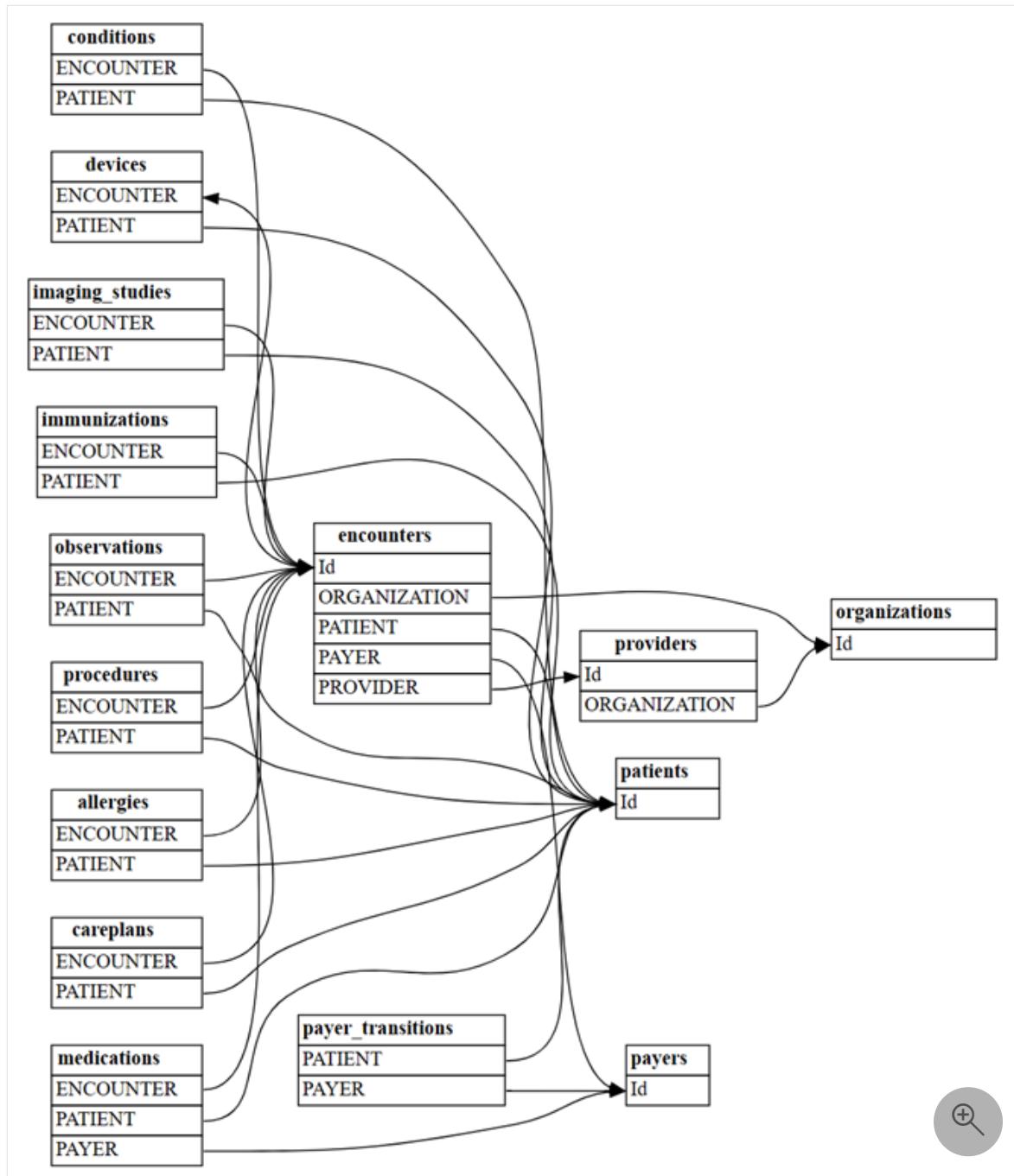
Python

```
suggested_relationships = find_relationships(all_tables)
suggested_relationships
```

3. Visualize relationships:

Python

```
plot_relationship_metadata(suggested_relationships)
```



- Count how many new "m:m" relationships will be discovered with `include_many_to_many=True`. These relationships are in addition to the previously shown "m:1" relationships; therefore, you have to filter on `multiplicity`:

Python

```
suggested_relationships = find_relationships(all_tables,
coverage_threshold=1.0, include_many_to_many=True)
suggested_relationships[suggested_relationships['Multiplicity']=='m:m']
```

- You can sort the relationship data by various columns to gain a deeper understanding of their nature. For example, you could choose to order the output by `Row Count From` and `Row Count To`, which help identify the largest tables.

Python

```
suggested_relationships.sort_values(['Row Count From', 'Row Count To'],  
ascending=False)
```

In a different semantic model, maybe it would be important to focus on number of nulls `Null Count From` or `Coverage To`.

This analysis can help you to understand if any of the relationships could be invalid, and if you need to remove them from the list of candidates.

Related content

Check out other tutorials for semantic link / SemPy:

- [Tutorial: Clean data with functional dependencies \(preview\)](#)
- [Tutorial: Analyze functional dependencies in a sample semantic model \(preview\)](#)
- [Tutorial: Discover relationships in a semantic model, using semantic link \(preview\)](#)
- [Tutorial: Extract and calculate Power BI measures from a Jupyter notebook \(preview\)](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Ask the community ↗](#)

Tutorial: Validate data using SemPy and Great Expectations (GX)

Article • 08/29/2024

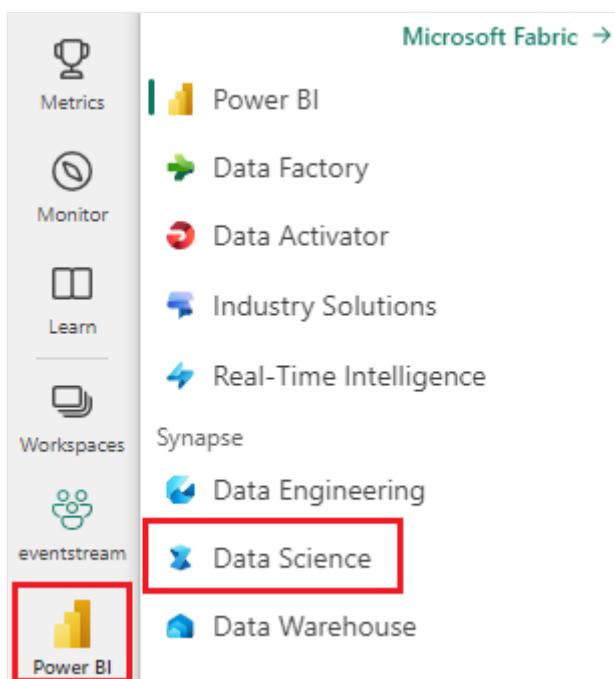
In this tutorial, you learn how to use SemPy together with [Great Expectations](#) (GX) to perform data validation on Power BI semantic models.

This tutorial shows you how to:

- ✓ Validate constraints on a dataset in your Fabric workspace with Great Expectation's Fabric Data Source (built on semantic link).
 - Configure a GX Data Context, Data Assets, and Expectations.
 - View validation results with a GX Checkpoint.
- ✓ Use semantic link to analyze raw data.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



- Select **Workspaces** from the left navigation pane to find and select your workspace. This workspace becomes your current workspace.

- Download the [Retail Analysis Sample PBIX.pbix](#) file.
- In your workspace, use the **Upload** button to upload the *Retail Analysis Sample PBIX.pbix* file to the workspace.

Follow along in notebook

[great_expectations_tutorial.ipynb](#) is the notebook that accompanies this tutorial.

To open the accompanying notebook for this tutorial, follow the instructions in [Prepare your system for data science tutorials](#), to import the notebook to your workspace.

If you'd rather copy and paste the code from this page, you can [create a new notebook](#).

Be sure to [attach a lakehouse to the notebook](#) before you start running code.

Set up the notebook

In this section, you set up a notebook environment with the necessary modules and data.

1. Install `SemPy` and the relevant `Great Expectations` libraries from PyPI using the `%pip` in-line installation capability within the notebook.

Python

```
# install libraries
%pip install semantic-link 'great-expectations<1.0'
great_expectations_experimental great_expectations_zipcode_expectations

# load %%dax cell magic
%load_ext sempy
```

2. Perform necessary imports of modules that you'll need later:

Python

```
import great_expectations as gx
from great_expectations.expectations.expectation import
ExpectationConfiguration
from great_expectations_zipcode_expectations.expectations import
expect_column_values_to_be_valid_zip5
```

Set up GX Data Context and Data Source

In order to get started with Great Expectations, you first have to set up a GX [Data Context](#). The context serves as an entry point for GX operations and holds all relevant configurations.

Python

```
context = gx.get_context()
```

You can now add your Fabric dataset to this context as a [Data Source](#) to start interacting with the data. This tutorial uses a standard Power BI sample semantic model [Retail Analysis Sample .pbix file](#).

Python

```
ds = context.sources.add_fabric_powerbi("Retail Analysis Data Source",
                                         dataset="Retail Analysis Sample PBIX")
```

Specify Data Assets

Define [Data Assets](#) to specify the subset of data you'd like to work with. The asset can be as simple as full tables, or be as complex as a custom Data Analysis Expressions (DAX) query.

Here, you'll add multiple assets:

- Power BI table
- Power BI measure
- Custom DAX query
- [Dynamic Management View](#) (DMV) query

Power BI table

Add a Power BI table as a data asset.

Python

```
ds.add_powerbi_table_asset("Store Asset", table="Store")
```

Power BI measure

If your dataset contains preconfigured measures, you add the measures as assets following a similar API to SemPy's `evaluate_measure`.

Python

```
ds.add_powerbi_measure_asset(  
    "Total Units Asset",  
    measure="TotalUnits",  
    groupby_columns=["Time[FiscalYear]", "Time[FiscalMonth]"]  
)
```

DAX

If you'd like to define your own measures or have more control over specific rows, you can add a DAX asset with a custom DAX query. Here, we define a `Total Units Ratio` measure by dividing two existing measures.

Python

```
ds.add_powerbi_dax_asset(  
    "Total Units YoY Asset",  
    dax_string=  
    """  
EVALUATE SUMMARIZECOLUMNS(  
    'Time'[FiscalYear],  
    'Time'[FiscalMonth],  
    "Total Units Ratio", DIVIDE([Total Units This Year], [Total Units  
Last Year]))  
    """  
)
```

DMV query

In some cases, it might be helpful to use [Dynamic Management View](#) (DMV) calculations as part of the data validation process. For example, you can keep track of the number of referential integrity violations within your dataset. For more information, see [Clean data = faster reports ↗](#).

Python

```
ds.add_powerbi_dax_asset(  
    "Referential Integrity Violation",  
    dax_string=  
    """  
SELECT
```

```
[Database_name],  
[Dimension_Name],  
[RIVIOLATION_COUNT]  
FROM $SYSTEM.DISCOVER_STORAGE_TABLES  
"""  
)
```

Expectations

To add specific constraints to the assets, you first have to configure [Expectation Suites](#). After adding individual [Expectations](#) to each suite, you can then update the Data Context set up in the beginning with the new suite. For a full list of available expectations, see the [GX Expectation Gallery](#).

Start by adding a "Retail Store Suite" with two expectations:

- a valid zip code
- a table with row count between 80 and 200

Python

```
suite_store = context.add_expectation_suite("Retail Store Suite")  
  
suite_store.add_expectation(ExpectationConfiguration("expect_column_values_to_be_valid_zip5", { "column": "PostalCode" }))  
suite_store.add_expectation(ExpectationConfiguration("expect_table_row_count_to_be_between", { "min_value": 80, "max_value": 200 }))  
  
context.add_or_update_expectation_suite(expectation_suite=suite_store)
```

TotalUnits Measure

Add a "Retail Measure Suite" with one expectation:

- Column values should be greater than 50,000

Python

```
suite_measure = context.add_expectation_suite("Retail Measure Suite")  
suite_measure.add_expectation(ExpectationConfiguration(  
    "expect_column_values_to_be_between",  
    {  
        "column": "TotalUnits",  
        "min_value": 50000  
    }  
))
```

```
context.add_or_update_expectation_suite(expectation_suite=suite_measure)
```

Total Units Ratio DAX

Add a "Retail DAX Suite" with one expectation:

- Column values for Total Units Ratio should be between 0.8 and 1.5

Python

```
suite_dax = context.add_expectation_suite("Retail DAX Suite")
suite_dax.add_expectation(ExpectationConfiguration(
    "expect_column_values_to_be_between",
    {
        "column": "[Total Units Ratio]",
        "min_value": 0.8,
        "max_value": 1.5
    }
))
context.add_or_update_expectation_suite(expectation_suite=suite_dax)
```

Referential Integrity Violations (DMV)

Add a "Retail DMV Suite" with one expectation:

- the RIVIOLATION_COUNT should be 0

Python

```
suite_dmv = context.add_expectation_suite("Retail DMV Suite")
# There should be no RI violations
suite_dmv.add_expectation(ExpectationConfiguration(
    "expect_column_values_to_be_in_set",
    {
        "column": "RIVIOLATION_COUNT",
        "value_set": [0]
    }
))
context.add_or_update_expectation_suite(expectation_suite=suite_dmv)
```

Validation

To actually run the specified expectations against the data, first create a [Checkpoint](#) and add it to the context. For more information on Checkpoint configuration, see [Data](#)

Validation workflow ↴.

Python

```
checkpoint_config = {
    "name": f"Retail Analysis Checkpoint",
    "validations": [
        {
            "expectation_suite_name": "Retail Store Suite",
            "batch_request": {
                "datasource_name": "Retail Analysis Data Source",
                "data_asset_name": "Store Asset",
            },
        },
        {
            "expectation_suite_name": "Retail Measure Suite",
            "batch_request": {
                "datasource_name": "Retail Analysis Data Source",
                "data_asset_name": "Total Units Asset",
            },
        },
        {
            "expectation_suite_name": "Retail DAX Suite",
            "batch_request": {
                "datasource_name": "Retail Analysis Data Source",
                "data_asset_name": "Total Units YoY Asset",
            },
        },
        {
            "expectation_suite_name": "Retail DMV Suite",
            "batch_request": {
                "datasource_name": "Retail Analysis Data Source",
                "data_asset_name": "Referential Integrity Violation",
            },
        },
    ],
}
checkpoint = context.add_checkpoint(
    **checkpoint_config
)
```

Now run the checkpoint and extract the results as a pandas DataFrame for simple formatting.

Python

```
result = checkpoint.run()
```

Process and print your results.

Python

```

import pandas as pd

data = []

for run_result in result.run_results:
    for validation_result in result.run_results[run_result]
        ["validation_result"]["results"]:
            row = {
                "Batch ID": run_result.batch_identifier,
                "type": validation_result.expectation_config.expectation_type,
                "success": validation_result.success
            }

            row.update(dict(validation_result.result))

            data.append(row)

result_df = pd.DataFrame.from_records(data)

result_df[["Batch ID", "type", "success", "element_count",
"unexpected_count", "partial_unexpected_list"]]

```

	Batch ID		type	success	element_count	unexpected_count	partial_unexpected_list
0	Retail Analysis Data Source-Store Asset	expect_column_values_to_be_valid_zip5		True	104.0	0.0	None
1	Retail Analysis Data Source-Store Asset	expect_table_row_count_to_be_between		True	NaN	NaN	None
2	Retail Analysis Data Source-Total Units Asset	expect_column_values_to_be_between		True	8.0	0.0	None
3	Retail Analysis Data Source-Total Units YoY Asset	expect_column_values_to_be_between		False	8.0	3.0	[0.7972605895837893, 0.7302630613281353, 0.73...
4	Retail Analysis Data Source-Referential Integr...	expect_column_values_to_be_in_set		True	84.0	0.0	None

From these results you can see that all your expectations passed the validation, except for the "Total Units YoY Asset" that you defined through a custom DAX query.

Diagnostics

Using semantic link, you can fetch the source data to understand which exact years are out of range. Semantic link provides an inline magic for executing DAX queries. Use semantic link to execute the same query you passed into the GX Data Asset and visualize the resulting values.

Python

```

%%dax "Retail Analysis Sample PBIX"

EVALUATE SUMMARIZECOLUMNS(
    'Time'[FiscalYear],
    'Time'[FiscalMonth],
    "Total Units Ratio", DIVIDE([Total Units This Year], [Total Units Last

```

```
Year])  
)
```

	Time[FiscalYear]	Time[FiscalMonth]	[Total Units Ratio]
0	2014	Jan	0.797261
1	2014	Feb	0.95912
2	2014	Mar	1.282908
3	2014	Apr	0.730263
4	2014	May	1.084064
5	2014	Jun	0.969717
6	2014	Jul	0.663352
7	2014	Aug	0.810861

Save these results in a DataFrame.

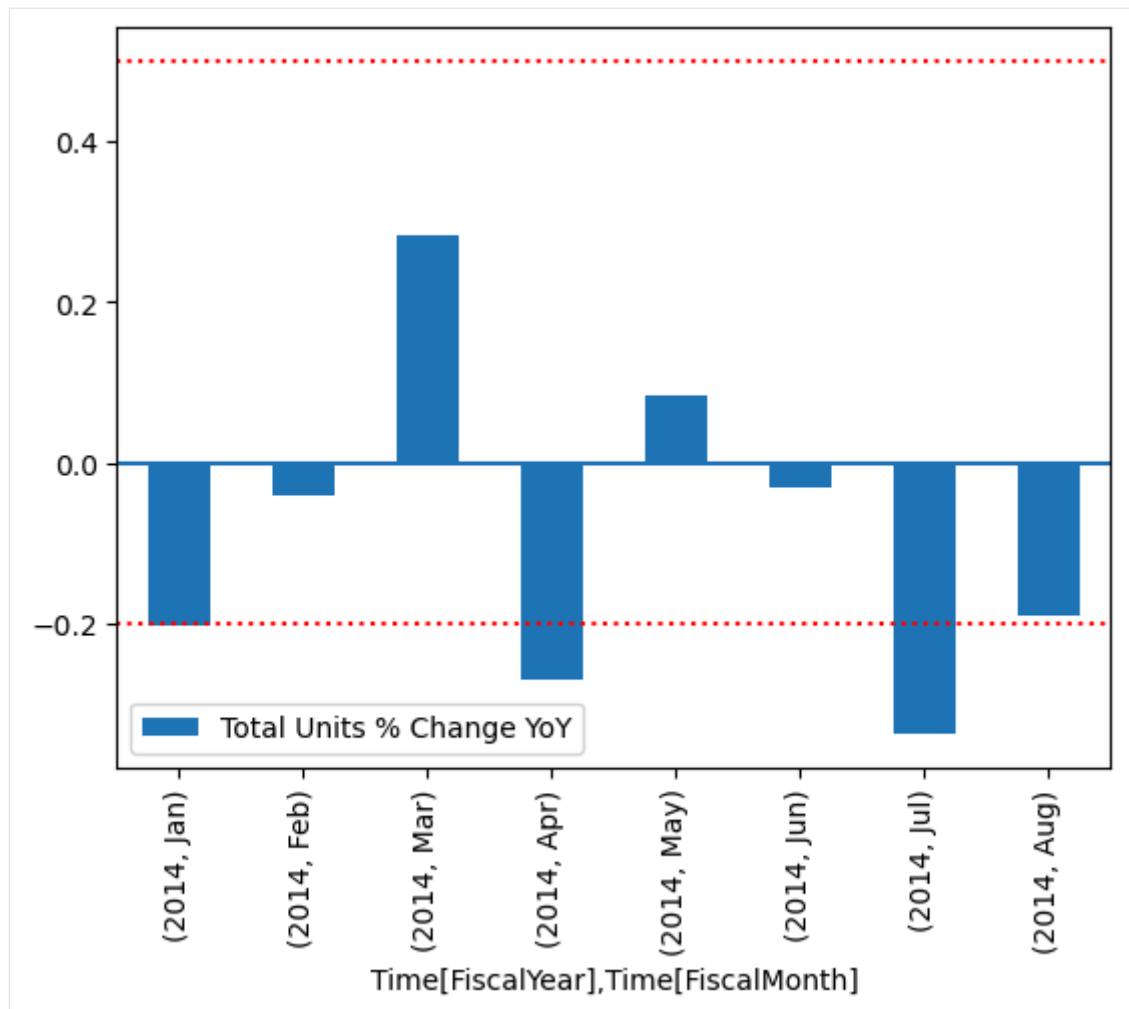
```
Python
```

```
df = _
```

Plot the results.

```
Python
```

```
import matplotlib.pyplot as plt  
  
df["Total Units % Change YoY"] = (df["[Total Units Ratio]"] - 1)  
  
df.set_index(["Time[FiscalYear]", "Time[FiscalMonth]").plot.bar(y="Total  
Units % Change YoY")  
  
plt.axhline(0)  
  
plt.axhline(-0.2, color="red", linestyle="dotted")  
plt.axhline( 0.5, color="red", linestyle="dotted")  
  
None
```



From the plot, you can see that April and July were slightly out of range and can then take further steps to investigate.

Storing GX configuration

As the data in your dataset changes over time, you might want to rerun the GX validations you just performed. Currently, the Data Context (containing the connected Data Assets, Expectation Suites, and Checkpoint) lives ephemerally, but it can be converted to a File Context for future use. Alternatively, you can instantiate a File Context (see [Instantiate a Data Context ↴](#)).

Python

```
context = context.convert_to_file_context()
```

Now that you saved the context, copy the `gx` directory to your lakehouse.

ⓘ **Important**

This cell assumes you [added a lakehouse](#) to the notebook. If there is no lakehouse attached, you won't see an error, but you also won't later be able to get the context. If you add a lakehouse now, the kernel will restart, so you'll have to re-run the entire notebook to get back to this point.

Python

```
# copy GX directory to attached lakehouse
!cp -r gx/ /lakehouse/default/Files/gx
```

Now, future contexts can be created with `context = gx.get_context(project_root_dir="
<your path here>")` to use all the configurations from this tutorial.

For example, in a new notebook, attach the same lakehouse and use `context =
gx.get_context(project_root_dir="/lakehouse/default/Files/gx")` to retrieve the context.

Related content

Check out other tutorials for semantic link / SemPy:

- [Tutorial: Clean data with functional dependencies](#)
- [Tutorial: Analyze functional dependencies in a sample semantic model](#)
- [Tutorial: Extract and calculate Power BI measures from a Jupyter notebook](#)
- [Tutorial: Discover relationships in a semantic model, using semantic link](#)
- [Tutorial: Discover relationships in the *Synthea* dataset, using semantic link](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback](#) | [Ask the community](#)

Power BI connectivity with semantic link

Article • 06/18/2024

Power BI connectivity is at the core of semantic link in Microsoft Fabric. This article describes the ways that semantic link provides connectivity to semantic models for users of the Python pandas and Apache Spark ecosystems.

A semantic model usually represents a high data standard that's the result of upstream data processing and refinement. Business analysts can:

- Encode domain knowledge and business logic into Power BI measures.
- Create Power BI reports by using semantic models.
- Use these reports to drive business decisions.

When data scientists working with the same semantic models try to duplicate business logic in different code environments or languages, critical errors can result. Semantic link bridges the gap between semantic models and the Synapse Data Science in Microsoft Fabric experience to provide a way for business analysts and data scientists to collaborate seamlessly and reduce data mismatch.

Semantic link offers connectivity to:

- The Python [pandas](#) ecosystem via the [SemPy Python library](#).
- Semantic models through the [Spark native connector](#) that supports PySpark, Spark SQL, R, and Scala.

Data connectivity through SemPy Python library for pandas users

The [SemPy Python library](#) is part of the semantic link feature and serves pandas users. SemPy functionality includes data retrieval from [tables](#), [computation of measures](#), and [execution of Data Analysis Expressions \(DAX\) queries](#) and metadata.

- For Spark 3.4 and above, semantic link is available in the default runtime when using Fabric, and there's no need to install it.
- For Spark 3.3 or below, or to update to the latest version of semantic link, run the following command:

Python

```
%pip install -U semantic-link
```

SemPy also extends pandas DataFrames with added metadata propagated from the Power BI data source. This metadata includes:

- Power BI data categories:
 - Geographic: Address, place, city
 - URL: Web URL, image URL
 - Barcode
- Relationships between tables
- Hierarchies

Data connectivity through semantic link Spark native connector

The semantic link Spark native connector lets Spark users access Power BI tables and measures. The connector is language-agnostic and supports PySpark, Spark SQL, R, and Scala.

To use the Spark native connector, you represent semantic models as Spark namespaces and transparently expose Power BI tables as Spark tables.

Spark SQL

The following command configures Spark to use the Power BI Spark native connector for Spark SQL:

Python

```
spark.conf.set("spark.sql.catalog.pbi",
"com.microsoft.azure.synapse.ml.powerbi.PowerBICatalog")

# Optionally, configure the workspace using its ID
# Resolve workspace name to ID using fabric.resolve_workspace_id("My
workspace")
# Replace 00000000-0000-0000-0000-000000000000 with your own workspace
ID
# spark.conf.set("spark.sql.catalog.pbi.workspace, "00000000-0000-0000-
0000-000000000000")
```

The following command lists all tables in a semantic model called `Sales Dataset`:

SQL

```
%%sql  
SHOW TABLES FROM pbi.`Sales Dataset`
```

The following command displays data from the `Customer` table in the semantic model `Sales Dataset`:

SQL

```
%%sql  
SELECT * FROM pbi.`Sales Dataset`.Customer
```

Power BI measures are accessible through the virtual `_Metrics` table to bridge relational Spark SQL with multidimensional Power BI. In the following example, `Total Revenue` and `Revenue Budget` are measures defined in the `Sales Dataset` semantic model, and the other columns are dimensions. Aggregation functions like `AVG` are ignored for measures and are present only to provide consistency with SQL.

The connector supports predicate push down of computations like `Customer[State] in ('CA', 'WA')` from Spark expressions into the Power BI engine to enable use of the Power BI optimized engine.

SQL

```
SELECT  
    `Customer[Country/Region]`,  
    `Industry[Industry]`,  
    AVG(`Total Revenue`),  
    AVG(`Revenue Budget`)  
FROM  
    pbi.`Sales Dataset`.`_Metrics`  
WHERE  
    `Customer[State]` in ('CA', 'WA')  
GROUP BY  
    `Customer[Country/Region]`,  
    `Industry[Industry]`
```

Data augmentation with Power BI measures

The `add_measure` operation is a powerful feature of semantic link that lets you augment data with measures from semantic models. This operation is available only in the SemPy Python library and isn't supported in the Spark native connector. For more information

on the `add_measure` method, see [add_measure](#) in the `FabricDataFrame` class documentation.

To use the SemPy Python library, install it in your notebook kernel by running the following code in a notebook cell:

Python

```
# %pip and import only needs to be done once per notebook
%pip install semantic-link
from sempy.fabric import FabricDataFrame
```

The following code example assumes you have an existing `FabricDataFrame` with data that you want to augment with measures from a semantic model.

Python

```
df = FabricDataFrame({
    "Sales Agent": ["Agent 1", "Agent 1", "Agent 2"],
    "Customer[Country/Region)": ["US", "GB", "US"],
    "Industry[Industry)": ["Services", "CPG", "Manufacturing"],
})
joined_df = df.add_measure(["Total Revenue", "Total Budget"], dataset="Sales
Dataset")
```

The `add_measure` method does the following steps:

1. Resolves column names in the `FabricDataFrame` to Power BI dimensions. The operation ignores any column names that can't be resolved within the given semantic model. For more information, see the supported [DAX syntax](#).
2. Defines `group by` columns, using the resolved column names.
3. Computes one or more measures at the `group by` level.
4. Filters the result by the existing rows in the `FabricDataFrame`.

Related content

- [SemPy add_measure reference documentation](#)
- [Tutorial: Extract and calculate Power BI measures from a Jupyter notebook](#)
- [Explore and validate data by using semantic link](#)
- [Explore and validate relationships in semantic models](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Semantic data propagation from semantic models

Article • 06/10/2024

When you read a [semantic model](#) into a [FabricDataFrame](#), semantic information such as metadata and annotations from the semantic model are automatically attached to the FabricDataFrame. In this article, you learn how the SemPy Python library preserves annotations that are attached to a semantic model's tables and columns.

Semantic propagation for pandas users

The [SemPy Python library](#) is part of the semantic link feature and serves [pandas](#) users. SemPy supports the operations that pandas allows you to perform on your data.

SemPy also lets you propagate semantic data from semantic models that you operate upon. By propagating semantic data, you can preserve annotations that are attached to tables and columns in the semantic model when you perform operations like slicing, merges, and concatenation.

You can create a [FabricDataFrame data structure](#) in either of two ways:

- You can read a table or the output of a [measure](#) from a semantic model into a FabricDataFrame.

When you read from a semantic model into a FabricDataFrame, the metadata from Power BI automatically *hydrates*, or populates, the FabricDataFrame. In other words, the FabricDataFrame preserves the semantic information from the model's tables or measures.

- You can use in-memory data to create the FabricDataFrame, just as you do for pandas DataFrames.

When you create a FabricDataFrame from in-memory data, you need to supply the name of a semantic model from which the FabricDataFrame can pull metadata information.

The way SemPy preserves semantic data varies depending on factors like the operations you do and the order of the FabricDataFrames you operate on.

Semantic propagation with merge

When you merge two FabricDataFrames, the order of the DataFrames determines how SemPy propagates semantic information.

- If both FabricDataFrames are annotated, the table-level metadata of the left FabricDataFrame takes precedence. The same rule applies to individual columns; the column annotations in the left FabricDataFrame take precedence over the column annotations in the right DataFrame.
- If only one FabricDataFrame is annotated, SemPy uses its metadata. The same rule applies to individual columns; SemPy uses the column annotations present in the annotated FabricDataFrame.

Semantic propagation with concatenation

When you concatenate multiple FabricDataFrame, for each column, SemPy copies the metadata from the first FabricDataFrame that matches the column name. If there are multiple matches and the metadata isn't the same, SemPy issues a warning.

You can also propagate concatenations of FabricDataFrames with regular pandas DataFrames by placing the FabricDataFrame first.

Semantic propagation for Spark users

The semantic link Spark native connector hydrates (or populates) the [metadata ↗](#) dictionary of a Spark column. Currently, support for semantic propagation is limited and subject to Spark's internal implementation of how schema information is propagated. For example, column aggregation strips the metadata.

Related content

- [Reference for SemPy's FabricDataFrame class](#)
- [Get started with Python semantic link \(SemPy\)](#)
- [Tutorial: Analyze functional dependencies in a sample semantic model](#)
- [Explore and validate data by using semantic link](#)
- [Explore and validate relationships in semantic models](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Semantic functions

Article • 06/18/2024

This article describes semantic functions and how they can help data scientists and data engineers discover functions that are relevant to the FabricDataFrame or FabricSeries they're working on. Semantic functions are part of the Microsoft Fabric semantic link feature.

For Spark 3.4 and above, the semantic link core package is available in the default Fabric runtime, but the semantic-link-functions package that includes the semantic function logic (such as `is_holiday`) needs to be installed manually. To update to the most recent version of the Python semantic link (SemPy) library, run the following command:

Python

```
%pip install -U semantic-link
```

A [FabricDataFrame](#) dynamically exposes semantic functions based on the logic each function defines. For example, the `is_holiday` function appears in the autocomplete suggestions when you work on a FabricDataFrame that contains both a datetime column and a country column.

Each semantic function uses information about the data, data types, and metadata (like Power BI data categories) in the FabricDataFrame or FabricSeries to determine its relevance to the particular data you're working on.

Semantic functions are automatically discovered when annotated with the `@semantic_function` decorator. You can think of semantic functions as being like [C# extension methods](#) applied to the DataFrame concept.

Semantic functions autocomplete suggestions

Semantic functions are available in the autocomplete suggestions when you work with a FabricDataFrame or FabricSeries. Use **Ctrl+Space** to trigger autocomplete.



The following code example manually specifies the metadata for a FabricDataFrame:

Python

```
from sempy.fabric import FabricDataFrame

df = FabricDataFrame(
    {"country": ["US", "AT"],
     "lat": [40.7128, 47.8095],
     "long": [-74.0060, 13.0550]},
    column_metadata={"lat": {"data_category": "Latitude"}, "long":
    {"data_category": "Longitude"}},
)

# Convert to GeoPandas dataframe
df_geo = df.to_geopandas(lat_col="lat", long_col="long")

# Use the explore function to visualize the data
df_geo.explore()
```

Alternatively, if you read from a semantic model into a FabricDataFrame, the metadata is autopopulated.

Python

```
from sempy.fabric import FabricDataFrame

# Read from semantic model
import sempy.fabric as fabric
df = fabric.read_table("my_dataset_name", "my_countries")

# Convert to GeoPandas dataframe
df_geo = df.to_geopandas(lat_col="lat", long_col="long")

# Use the explore function to visualize the data
df_geo.explore()
```

Built-in semantic functions

The SemPy Python library provides a set of built-in semantic functions that are available out of the box. These built-in functions include:

- `is_holiday(...)` uses the [holidays](#) Python package to return `true` if the date is a holiday in the given country.
- `to_geopandas(...)` converts a FabricDataFrame to a [GeoPandas](#) GeoDataFrame.
- `parse_phonenumber(...)` uses the [phone numbers](#) Python package to parse a phone number into its components.

- `validators` uses the [validators](#) Python package to validate common data types like email and credit card numbers.

Custom semantic functions

Semantic functions are designed for extensibility. You can define your own semantic functions within your notebook or as separate Python modules.

To use a semantic function outside of a notebook, declare the semantic function within the `sempy.functions` module. The following code example shows the definition of a semantic function `_is_capital` that returns `true` if a city is the capital of a country.

```
Python

from sempy.fabric import FabricDataFrame, FabricSeries
from sempy.fabric.matcher import CountryMatcher, CityMatcher
from sempy.functions import semantic_function, semantic_parameters

@semantic_function("is_capital")
@semantic_parameters(col_country=CountryMatcher, col_city=CityMatcher)
def _is_capital(df: FabricDataFrame, col_country: str, col_city: str) ->
FabricSeries:
    """Returns true if the city is the capital of the country"""
    capitals = {
        "US": ["Washington"],
        "AT": ["Vienna"],
        # ...
    }

    return df[[col_country, col_city]] \
        .apply(lambda row: row[1] in capitals[row[0]]), axis=1)
```

In the preceding code example:

- The `col_country` and `col_city` parameters are annotated with `CountryMatcher` and `CityMatcher`, respectively. This annotation allows the semantic function to be automatically discovered when working with a `FabricDataFrame` that has the corresponding metadata.
- Calling the function also supplies standard data types such as `str`, `int`, `float`, and `datetime` to define required input columns.
- The type annotation of the first parameter `df` shows that the function is applicable to a `FabricDataFrame` rather than a `FabricSeries`.

Related content

- SemPy functions package
 - Tutorial: Clean data with functional dependencies
 - Power BI connectivity with semantic link
 - Semantic data propagation from semantic models
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

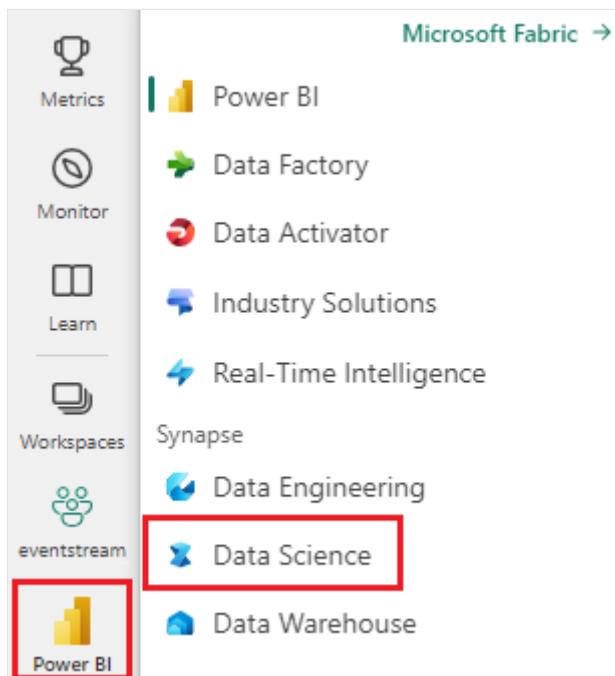
Read from semantic models and write data consumable by Power BI using python

Article • 11/19/2024

In this article, you learn how to read data and metadata and evaluate measures in semantic models using the SemPy python library in Microsoft Fabric. You also learn how to write data that semantic models can consume.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



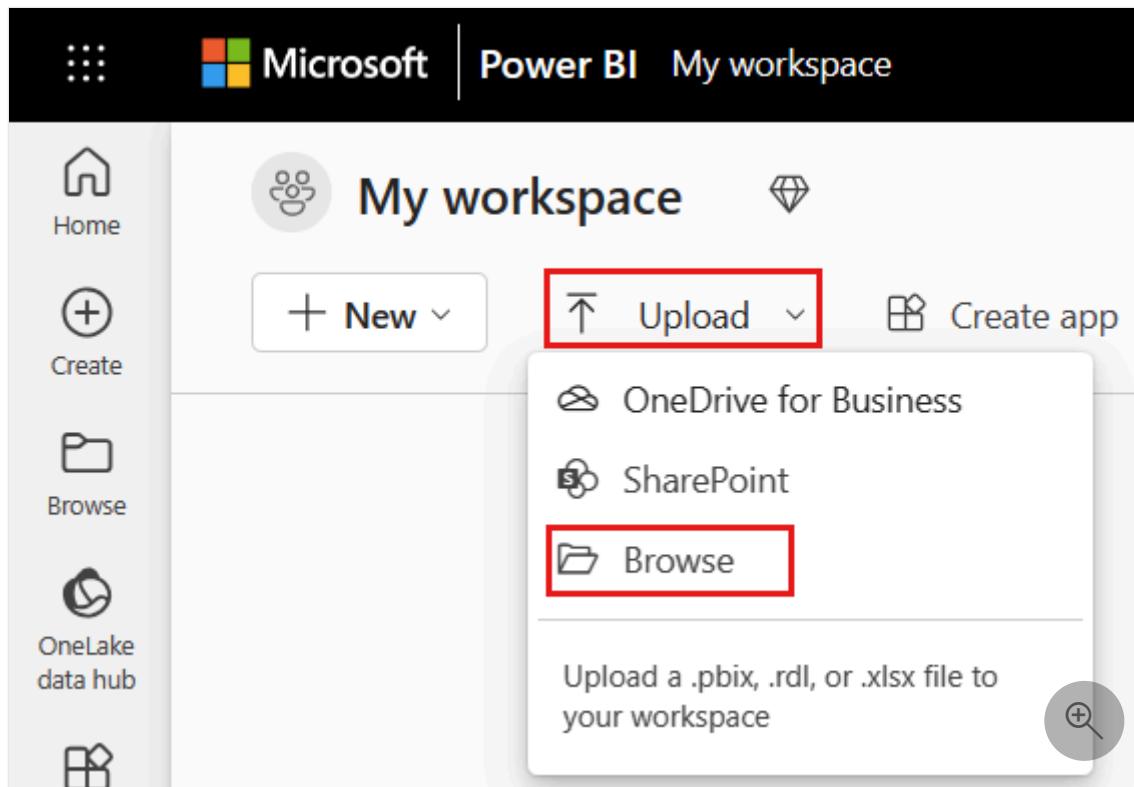
- Visit the Data Science experience in Microsoft Fabric.
- Create a [new notebook](#), to copy/paste code into cells
- For Spark 3.4 and above, Semantic link is available in the default runtime when using Fabric, and there's no need to install it. If you're using Spark 3.3 or below, or if you want to update to the most recent version of Semantic Link, you can run the command: `python %pip install -U semantic-link`
- [Add a Lakehouse to your notebook](#)

- Download the *Customer Profitability Sample.pbix* semantic model from the [datasets folder](#) of the fabric-samples repository, and save the semantic model locally

Upload the semantic model into your workspace

This article uses the *Customer Profitability Sample.pbix* semantic model. This semantic model references a company manufacturing marketing materials. It contains product, customer, and corresponding revenue data for various business units.

1. Open your [workspace](#) in Fabric Data Science
2. Select **Upload > Browse**, and select the *Customer Profitability Sample.pbix* semantic model.



When the upload is complete, your workspace has three new artifacts: a Power BI report, a dashboard, and a semantic model named *Customer Profitability Sample*. The steps in this article rely on that semantic model.

The screenshot shows the Microsoft Fabric workspace interface. At the top, there's a navigation bar with icons for 'My workspace' (a person icon), 'New', 'Upload', 'Create deployment pipeline', 'Create app', and a three-dot menu. Below the navigation bar is a table listing semantic models:

	Name	Type	Owner
Customer Profitability Sample	Report	My workspace	
Customer Profitability Sample	Dataset	My workspace	
Customer Profitability Sample.pbix	Dashboard	My workspace	

A magnifying glass icon is located in the bottom right corner of the table area.

Use Python to read data from semantic models

The SemPy Python API can retrieve data and metadata from semantic models located in a Microsoft Fabric workspace. The API can also execute queries on them.

Your notebook, Power BI dataset semantic model, and [lakehouse](#) can be located in the same workspace or in different workspaces. By default, SemPy tries to access your semantic model from:

- The workspace of your lakehouse, if you attached a lakehouse to your notebook.
- The workspace of your notebook, if there's no lakehouse attached.

If your semantic model isn't located in either of these workspaces, you must specify the workspace of your semantic model when you call a SemPy method.

To read data from semantic models:

1. List the available semantic models in your workspace.

```
Python

import sempy.fabric as fabric

df_datasets = fabric.list_datasets()
df_datasets
```

2. List the tables available in the *Customer Profitability Sample* semantic model.

```
Python

df_tables = fabric.list_tables("Customer Profitability Sample",
                               include_columns=True)
```

```
df_tables
```

3. List the measures defined in the *Customer Profitability Sample* semantic model.

💡 Tip

In the following code sample, we specified the workspace for SemPy to use for accessing the semantic model. You can replace `Your Workspace` with the name of the workspace where you uploaded the semantic model (from the [Upload the semantic model into your workspace](#) section).

Python

```
df_measures = fabric.list_measures("Customer Profitability Sample",
                                    workspace="Your Workspace")
df_measures
```

Here, we determined that the *Customer* table is the table of interest.

4. Read the *Customer* table from the *Customer Profitability Sample* semantic model.

Python

```
df_table = fabric.read_table("Customer Profitability Sample",
                             "Customer")
df_table
```

⚠ Note

- Data is retrieved using XMLA. This requires at least [XMLA read-only](#) to be enabled.
- The amount of retrievable data is limited by - the [maximum memory per query](#) of the capacity SKU that hosts the semantic model - the Spark driver node (visit [node sizes](#) for more information) that runs the notebook
- All requests use low priority to minimize the impact on Microsoft Azure Analysis Services performance, and are billed as [interactive requests](#).

5. Evaluate the *Total Revenue* measure for the state and date of each customer.

Python

```
df_measure = fabric.evaluate_measure(  
    "Customer Profitability Sample",  
    "Total Revenue",  
    ["'Customer'[State]", "Calendar[Date]"])  
df_measure
```

① Note

- By default, data is **not** retrieved using XMLA and therefore doesn't require XMLA read-only to be enabled.
- The data is **not** subject to [Power BI backend limitations](#).
- The amount of retrievable data is limited by - the [maximum memory per query](#) of the capacity SKU hosting the semantic model - the Spark driver node (visit [node sizes](#) for more information) that runs the notebook
- All requests are billed as [interactive requests](#)

6. To add filters to the measure calculation, specify a list of permissible values for a particular column.

Python

```
filters = {  
    "State[Region]": ["East", "Central"],  
    "State[State]": ["FLORIDA", "NEW YORK"]  
}  
df_measure = fabric.evaluate_measure(  
    "Customer Profitability Sample",  
    "Total Revenue",  
    ["Customer[State]", "Calendar[Date]"],  
    filters=filters)  
df_measure
```

7. You can also evaluate the *Total Revenue* measure per customer's state and date with a [DAX query](#).

Python

```
df_dax = fabric.evaluate_dax(  
    "Customer Profitability Sample",  
    """  
    EVALUATE SUMMARIZECOLUMNS(  
        'State'[Region],  
        'Calendar'[Date].[Year],
```

```
'Calendar'[Date].[Month],  
"Total Revenue",  
CALCULATE([Total Revenue]))  
"""")
```

ⓘ Note

- Data is retrieved using XMLA and therefore requires at least [XMLA read-only](#) to be enabled
- The amount of retrievable data is limited by the available memory in Microsoft Azure Analysis Services and the Spark driver node (visit [node sizes](#) for more information)
- All requests use low priority to minimize the impact on Analysis Services performance and are billed as interactive requests

8. Use the `%%dax` cell magic to evaluate the same DAX query, without the need to import the library. Run this cell to load `%%dax` cell magic:

```
%load_ext sempy
```

The workspace parameter is optional. It follows the same rules as the workspace parameter of the `evaluate_dax` function.

The cell magic also supports access of Python variables with the `{variable_name}` syntax. To use a curly brace in the DAX query, escape it with another curly brace (example: `EVALUATE {{1}}`).

DAX

```
%%dax "Customer Profitability Sample" -w "Your Workspace"  
EVALUATE SUMMARIZECOLUMNS(  
    'State'[Region],  
    'Calendar'[Date].[Year],  
    'Calendar'[Date].[Month],  
    "Total Revenue",  
    CALCULATE([Total Revenue]))
```

The resulting FabricDataFrame is available via the `_` variable. That variable captures the output of the last executed cell.

```
Python
```

```
df_dax = _  
df_dax.head()
```

9. You can add measures to data retrieved from external sources. This approach combines three tasks:

- It resolves column names to Power BI dimensions
- It defines group by columns
- It filters the measure Any column names that can't be resolved within the given semantic model are ignored (visit the supported [DAX syntax](#) resource for more information).

```
Python
```

```
from sempy.fabric import FabricDataFrame  
  
df = FabricDataFrame({  
    "Sales Agent": ["Agent 1", "Agent 1", "Agent 2"],  
    "Customer[Country/Region)": ["US", "GB", "US"],  
    "Industry[Industry)": ["Services", "CPG", "Manufacturing"],  
}  
)  
  
joined_df = df.add_measure("Total Revenue", dataset="Customer  
Profitability Sample")  
joined_df
```

Special parameters

The SemPy `read_table` and `evaluate_measure` methods have more parameters that are useful for manipulating the output. These parameters include:

- `fully_qualified_columns`: For a "True" value, the methods return column names in the form `TableName[ColumnName]`
- `num_rows`: The number of rows to output in the result
- `pandas_convert_dtypes`: For a "True" value, pandas cast the resulting DataFrame columns to the best possible *dtype* [convert_dtypes ↗](#). If this parameter is turned off, type incompatibility issues between columns of related tables can result; the Power BI model might not detect those issues because of [DAX implicit type conversion](#)

SemPy `read_table` also uses the model information that Power BI provides.

- `multiindex_hierarchies`: If "True", it converts [Power BI Hierarchies](#) to a pandas MultiIndex structure

Write data consumable by semantic models

Spark tables added to a Lakehouse are automatically added to the corresponding [default semantic model](#). This example demonstrates how to write data to the attached Lakehouse. The FabricDataFrame accepts the same input data as Pandas dataframes.

Python

```
from sempy.fabric import FabricDataFrame

df_forecast = FabricDataFrame({'ForecastedRevenue': [1, 2, 3]})

df_forecast.to_lakehouse_table("ForecastTable")
```

With Power BI, the *ForecastTable* table can be added to a composite semantic model with the Lakehouse semantic model.

Related content

- See [sempy.functions](#) to learn about usage of semantic functions
- Tutorial: Extract and calculate Power BI measures from a Jupyter notebook
- Explore and validate relationships in semantic models
- How to validate data with semantic link

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

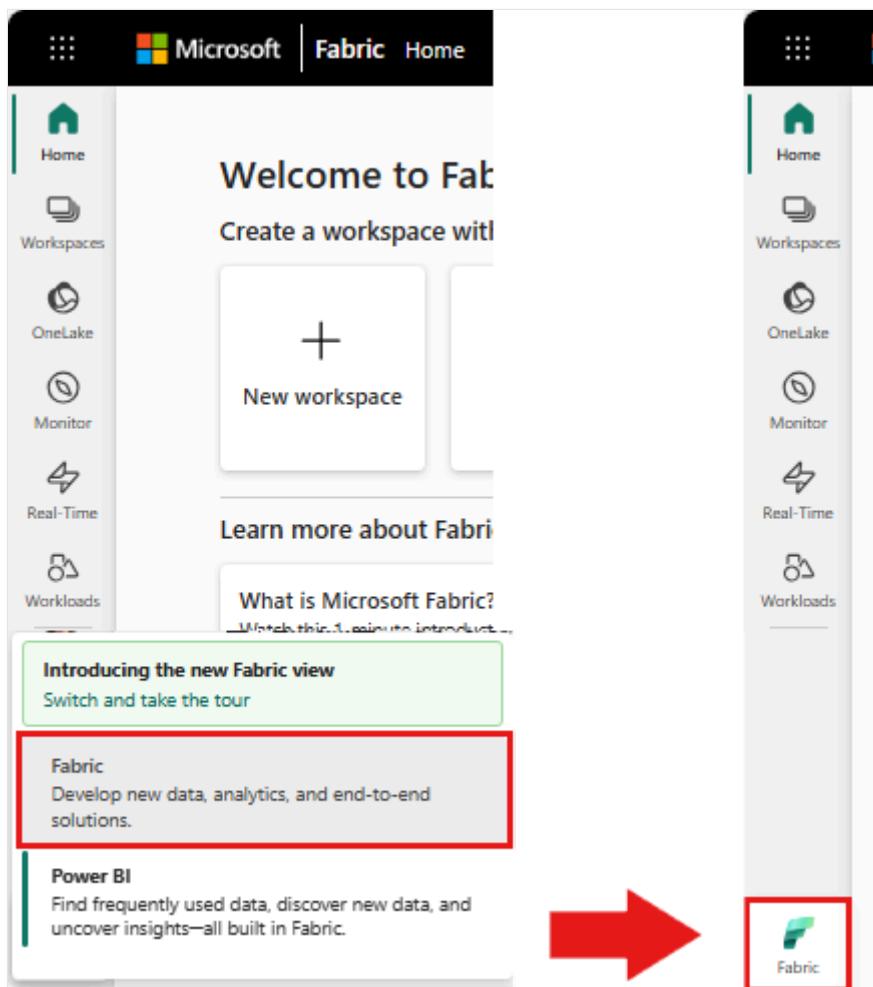
Read from semantic models and write data consumable by Power BI using Spark

Article • 03/23/2025

In this article, you can learn how to read data and metadata and evaluate measures in semantic models using the semantic link Spark native connector in Microsoft Fabric. You will also learn how to write data that semantic models can consume.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- Go to the Data Science experience in Microsoft Fabric.

- From the left pane, select **Workloads**.
- Select **Data Science**.
- Create a [new notebook](#) to copy/paste code into cells.
- For Spark 3.4 and above, Semantic link is available in the default runtime when using Fabric, and there's no need to install it. If you're using Spark 3.3 or below, or if you want to update to the most recent version of Semantic Link, you can run the command:

```
python %pip install -U semantic-link
```

- [Add a Lakehouse to your notebook](#).
- Download the *Customer Profitability Sample.pbix* semantic model from the [datasets folder](#) of the fabric-samples repository, and save the semantic model locally.

Upload the semantic model into your workspace

In this article, we use the *Customer Profitability Sample.pbix* semantic model. This semantic model references a company manufacturing marketing materials and contains data about products, customers, and corresponding revenue for various business units.

1. From the left pane, select **Workspaces** and then select the name of your [workspace](#) to open it.
2. Select **Import > Report or Paginated Report > From this computer** and select the *Customer Profitability Sample.pbix* semantic model.

The screenshot shows the Power BI 'My workspace' interface. On the left is a vertical sidebar with icons for Home, Create, Browse, OneLake, Apps, and More. The main area has a title 'My workspace' with a user icon. Below it are buttons for '+ New item', 'New folder', and 'Upload'. A dropdown menu for 'Upload' is open, showing options for OneDrive, SharePoint, and 'Browse'. The 'Browse' option is highlighted with a red box. A tooltip below the menu says 'Upload a .pbix or .rdl file to your workspace'. A magnifying glass icon is in the bottom right corner of the main area.

Once the upload is done, your workspace has three new artifacts: a Power BI report, a dashboard, and a semantic model named *Customer Profitability Sample*. You use this semantic model for the steps in this article.

Name	Type	Task	Owner	Refreshed
Customer Profitability Sample	Report	—	MA_ws	1/10/2025, 4:38:13 PM
Customer Profitability Sample	Semantic model	—	MA_ws	1/10/2025, 4:38:13 PM
Customer Profitability Sample.pbix	Dashboard	—	MA_ws	—

Read and write data, using Spark in Python, R, SQL, and Scala

By default, the workspace used to access semantic models is:

- the workspace of the attached [Lakehouse](#) or
- the workspace of the notebook, if no Lakehouse is attached.

Microsoft Fabric exposes all tables from all semantic models in the workspace as Spark tables. All Spark SQL commands can be executed in Python, R, and Scala. The semantic link Spark native connector supports push-down of Spark predicates to the Power BI engine.

Tip

Since Power BI tables and measures are exposed as regular Spark tables, they can be joined with other Spark data sources in a single query.

1. List tables of all semantic models in the workspace, using PySpark.

Python

```
df = spark.sql("SHOW TABLES FROM pbi")
display(df)
```

2. Retrieve the data from the *Customer* table in the *Customer Profitability Sample* semantic model, using SparkR.

① Note

Retrieving tables is subject to strict limitations (see [Read Limitations](#)) and the results might be incomplete. Use aggregate pushdown to reduce the amount of data transferred. The supported aggregates are: COUNT, SUM, AVG, MIN, and MAX.

R

```
%%sparkr
```

```
df = sql("SELECT * FROM pbi.`Customer Profitability Sample`.Customer")
display(df)
```

3. Power BI measures are available through the virtual table *_Metrics*. The following query computes the *total revenue* and *revenue budget* by *region* and *industry*.

SQL

```
%%sql
```

```
SELECT
    `Customer[Country/Region]`,
    `Industry[Industry]`,
    AVG(`Total Revenue`),
    AVG(`Revenue Budget`)
FROM
    pbi.`Customer Profitability Sample`.`_Metrics`
WHERE
    `Customer[State]` in ('CA', 'WA')
GROUP BY
```

```
`Customer[Country/Region]`,  
`Industry[Industry]`
```

4. Inspect available measures and dimensions, using Spark schema.

Python

```
spark.table("pbi.`Customer Profitability  
Sample`._Metrics").printSchema()
```

5. Save the data as a delta table to your Lakehouse.

Python

```
delta_table_path = "<your delta table path>" #fill in your delta table  
path  
df.write.format("delta").mode("overwrite").save(delta_table_path)
```

Read-access limitations

The read access APIs have the following limitations:

- Queries running longer than 10s in Analysis Service are not supported (Indication inside Spark: "java.net.SocketTimeoutException: PowerBI service comm failed ")
- Power BI table access using Spark SQL is subject to [Power BI backend limitations](#).
- Predicate pushdown for `Spark_Metrics` queries is limited to a single [IN ↗](#) expression and requires at least two elements. Extra IN expressions and unsupported predicates are evaluated in Spark after data transfer.
- Predicate pushdown for Power BI tables accessed using Spark SQL doesn't support the following expressions:
 - [ISNULL ↗](#)
 - [IS_NOT_NULL ↗](#)
 - [STARTS_WITH ↗](#)
 - [ENDS_WITH ↗](#)
 - [CONTAINS ↗](#).
- The Spark session must be restarted to make new semantic models accessible in Spark SQL.

Related content

- See [sempy.functions](#) to learn about usage of semantic functions
- Tutorial: Extract and calculate Power BI measures from a Jupyter notebook

- Explore and validate relationships in semantic models
 - How to validate data with semantic link
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Detect, explore, and validate functional dependencies in your data, using semantic link

Article • 11/19/2024

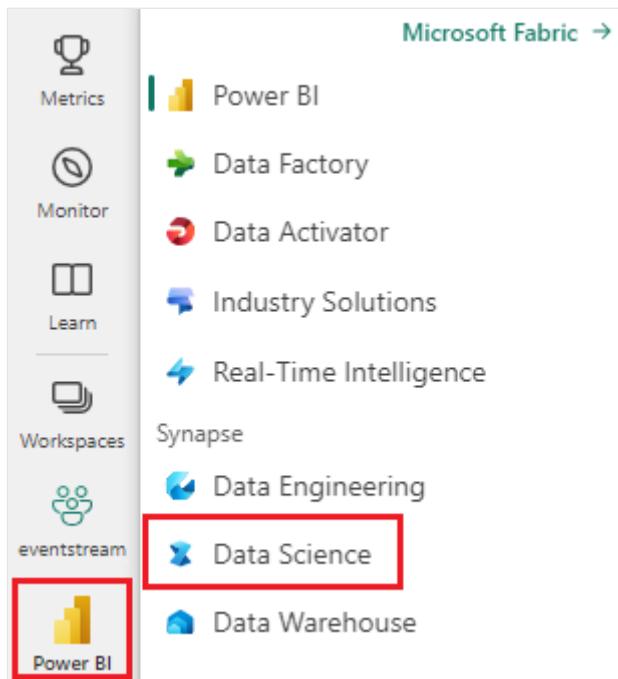
Functional dependencies are relationships between columns in a table, where the values in one column are used to determine the values in another column. Understanding these dependencies can help you uncover patterns and relationships in your data, which in turn can help with feature engineering, data cleaning, and model building tasks. Functional dependencies act as an effective invariant that allows you to find and fix data quality issues that might be hard to detect otherwise.

In this article, you use semantic link to:

- ✓ Find dependencies between columns of a FabricDataFrame
- ✓ Visualize dependencies
- ✓ Identify data quality issues
- ✓ Visualize data quality issues
- ✓ Enforce functional constraints between columns in a dataset

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



- Go to the Data Science experience found in Microsoft Fabric.
- Create a new notebook to copy/paste code into cells.
- For Spark 3.4 and above, Semantic link is available in the default runtime when using Fabric, and there's no need to install it. If you're using Spark 3.3 or below, or if you want to update to the most recent version of Semantic Link, you can run the command: `python %pip install -U semantic-link`
- Add a Lakehouse to your notebook.

For Spark 3.4 and above, Semantic link is available in the default runtime when using Fabric, and there's no need to install it. If you use Spark 3.3 or below, or if you want to update to the most recent version of Semantic Link, run this command:

Python

```
%pip install -U semantic-link
```
Find functional dependencies in data
```

The SemPy `find\_dependencies` function detects functional dependencies between the columns of a FabricDataFrame. The function uses a threshold on conditional entropy to discover approximate functional dependencies, where low conditional entropy indicates strong dependence between columns. To make the `find\_dependencies` function more selective, you can set a lower threshold on conditional entropy. The lower threshold means that only stronger dependencies will be detected.

This Python code snippet demonstrates how to use `find\_dependencies`:

```
```python
from sempy.fabric import FabricDataFrame
from sempy.dependencies import plot_dependency_metadata
```

```
import pandas as pd

df = FabricDataFrame(pd.read_csv("your_data.csv"))

deps = df.find_dependencies()
```

The `find_dependencies` function returns a FabricDataFrame with detected dependencies between columns. A list represents columns that have a 1:1 mapping. The function also removes [transitive edges](#), to try to prune the potential dependencies.

When you specify the `dropna=True` option, rows that have a NaN value in either column are eliminated from evaluation. This can result in nontransitive dependencies, as shown in this example:

[\[\]](#) Expand table

A	B	C
1	1	1
1	1	1
1	NaN	9
2	NaN	2
2	2	2

In some cases, the dependency chain can form cycles when you specify the `dropna=True` option, as shown in this example:

[\[\]](#) Expand table

A	B	C
1	1	NaN
2	1	NaN
NaN	1	1
NaN	2	1
1	NaN	1
1	NaN	2

Visualize dependencies in data

After you find functional dependencies in a dataset (using `find_dependencies`), you can visualize the dependencies with the `plot_dependency_metadata` function. This function takes the resulting FabricDataFrame from `find_dependencies` and creates a visual representation of the dependencies between columns and groups of columns.

This Python code snippet shows how to use `plot_dependencies`:

Python

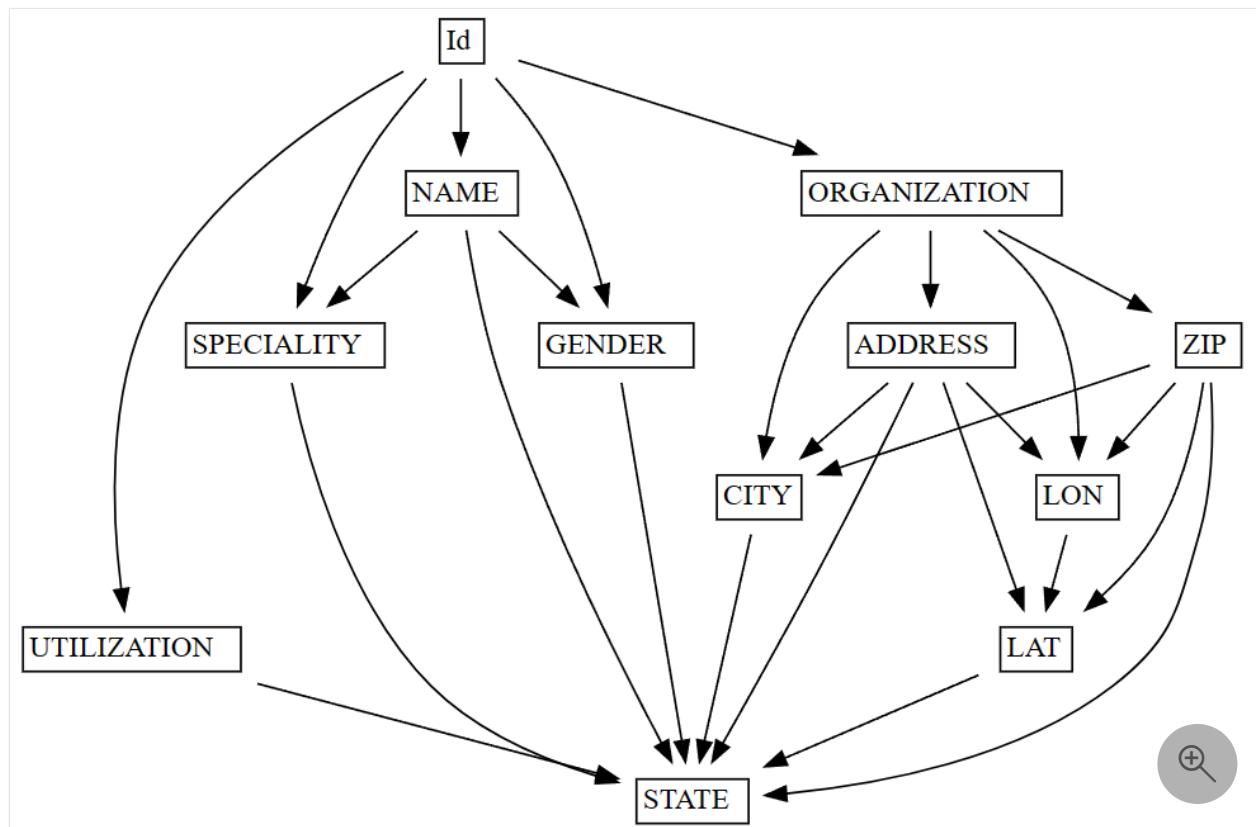
```
from sempy.fabric import FabricDataFrame
from sempy.dependencies import plot_dependency_metadata
from sempy.samples import download_synthea

download_synthea(which='small')

df = FabricDataFrame(pd.read_csv("synthea/csv/providers.csv"))

deps = df.find_dependencies()
plot_dependency_metadata(deps)
```

The `plot_dependency_metadata` function generates a visualization that shows the 1:1 groupings of columns. Columns that belong to a single group are placed in a single cell. If no suitable candidates are found, an empty FabricDataFrame is returned.



Identify data quality issues

Data quality issues can have various forms - for example, missing values, inconsistencies, or inaccuracies. Identifying and addressing these issues is important to ensure the reliability and validity of any analysis or model built on the data. One way to detect data quality issues is to examine violations of functional dependencies between columns in a dataset.

The `list_dependency_violations` function can help identify violations of functional dependencies between dataset columns. Given a determinant column and a dependent column, this function shows values that violate the functional dependency, along with the count of their respective occurrences. This can help inspect approximate dependencies and identify data quality issues.

This code snippet shows how to use the `list_dependency_violations` function:

Python

```
from sempy.fabric import FabricDataFrame
from sempy.samples import download_synthea

download_synthea(which='small')

df = FabricDataFrame(pd.read_csv("synthea/csv/providers.csv"))

violations = df.list_dependencyViolations(determinant_col="ZIP",
dependent_col="CITY")
```

In this example, the function assumes a functional dependency between the ZIP (determinant) and CITY (dependent) columns. If the dataset has data quality issues - for example, the same ZIP Code assigned to multiple cities - the function outputs the data with the problems:

[] Expand table

ZIP	CITY	count
12345	Boston	2
12345	Seattle	1

This output indicates that two different cities (Boston and Seattle) have the same ZIP Code value (12345). This suggests a data quality issue within the dataset.

The `list_dependency_violations` function provides more options that can handle missing values, show values mapped to violating values, limit the number of violations returned, and sort the results by count or determinant column.

The `list_dependency_violations` output can help identify dataset data quality issues. However, you should carefully examine the results and consider the context of your data, to determine the most appropriate course of action to address the identified issues. This approach might involve more data cleaning, validation, or exploration to ensure the reliability and validity of your analysis or model.

Visualize data quality issues

Data quality issues can damage the reliability and validity of any analysis or model built on that data. Identifying and addressing these issues is important to ensure the accuracy of your results. To detect data quality issues, you can examine violations of functional dependencies between columns in a dataset. Visualizing these violations can show the problems more clearly, and help you address them more effectively.

The `plot_dependency_violations` function can help visualize violations of functional dependencies between columns in a dataset. Given a determinant column and a dependent column, this function shows the violating values in a graphical format, to make it easier to understand the nature and extent of the data quality issues.

This code snippet shows how to use the `plot_dependency_violations` function:

Python

```
from sempy.fabric import FabricDataFrame
from sempy.dependencies import plot_dependencyViolations
from sempy.samples import download_synthea

download_synthea(which='small')

df = FabricDataFrame(pd.read_csv("synthea/csv/providers.csv"))

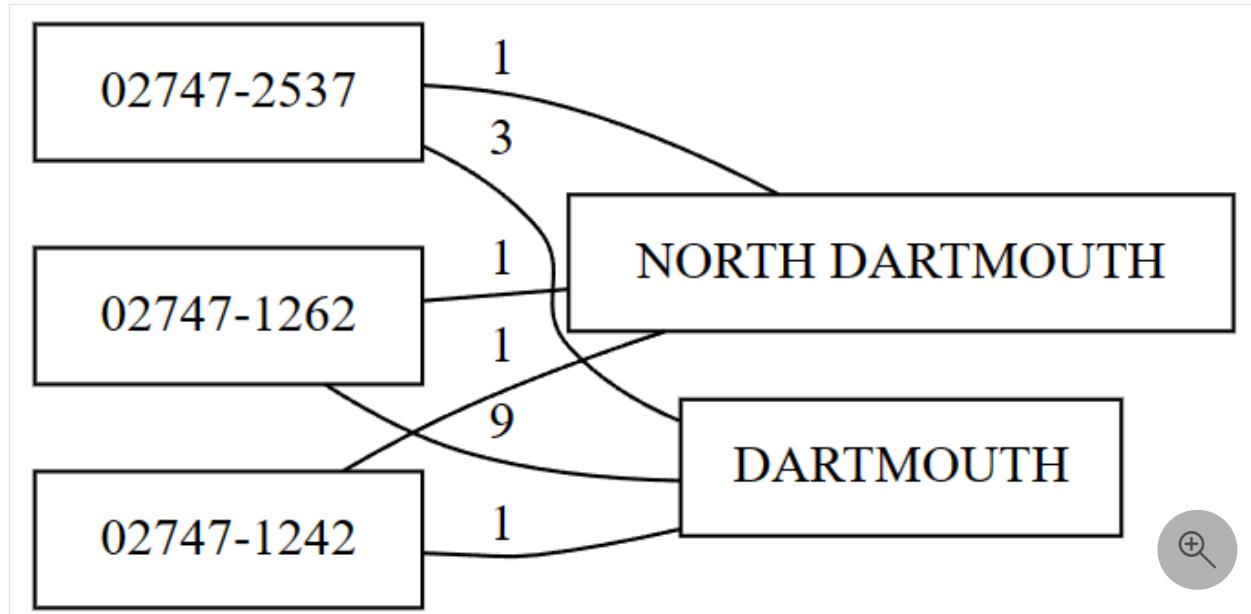
df.plot_dependencyViolations(determinant_col="ZIP", dependent_col="CITY")
```

In this example, the function assumes an existing functional dependency between the ZIP (determinant) and CITY (dependent) columns. If the dataset has data quality issues - for example, the same ZIP code assigned to multiple cities - the function generates a graph of the violating values.

The `plot_dependency_violations` function provides more options that can handle missing values, show values mapped to violating values, limit the number of violations

returned, and sort the results by count or determinant column.

The `plot_dependencyViolations` function generates a visualization that can help identify dataset data quality issues. However, you should carefully examine the results and consider the context of your data, to determine the most appropriate course of action to address the identified issues. This approach might involve more data cleaning, validation, or exploration to ensure the reliability and validity of your analysis or model.



Enforce functional constraints

Data quality is crucial for ensuring the reliability and validity of any analysis or model built on a dataset. Enforcement of functional constraints between columns in a dataset can help improve data quality. Functional constraints can help ensure that the relationships between columns have accuracy and consistency, which can lead to more accurate analysis or model results.

The `drop_dependencyViolations` function can help enforce functional constraints between columns in a dataset. It removes rows that violate a given constraint. Given a determinant column and a dependent column, this function removes rows with values that don't conform to the functional constraint between the two columns.

This code snippet shows how to use the `drop_dependencyViolations` function:

Python

```
from sempy.fabric import FabricDataFrame
from sempy.samples import download_synthea

download_synthea(which='small')
```

```
df = FabricDataFrame(pd.read_csv("synthea/csv/providers.csv"))

cleaned_df = df.drop_dependency_violations(determinant_col="ZIP",
                                            dependent_col="CITY")
```

Here, the function enforces a functional constraint between the ZIP (determinant) and CITY (dependent) columns. For each value of the determinant, the most common value of the dependent is picked, and all rows with other values are dropped. For example, given this dataset, the row with **CITY=Seattle** would be dropped, and the functional dependency **ZIP -> CITY** holds in the output:

[+] Expand table

ZIP	CITY
12345	Seattle
12345	Boston
12345	Boston
98765	Baltimore
00000	San Francisco

The `drop_dependency_violations` function provides the `verbose` option to control the output verbosity. By setting `verbose=1`, you can see the number of dropped rows. A `verbose=2` value shows the entire row content of the dropped rows.

The `drop_dependency_violations` function can enforce functional constraints between columns in your dataset, which can help improve data quality and lead to more accurate results in your analysis or model. However, you must carefully consider the context of your data and the functional constraints you choose to enforce, to ensure that you don't accidentally remove valuable information from your dataset.

Related content

- See the SemPy reference documentation for the `FabricDataFrame` class
- Tutorial: Clean data with functional dependencies
- Explore and validate relationships in semantic models
- Accelerate data science using semantic functions

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Explore and validate relationships in semantic models and DataFrames

Article • 06/18/2024

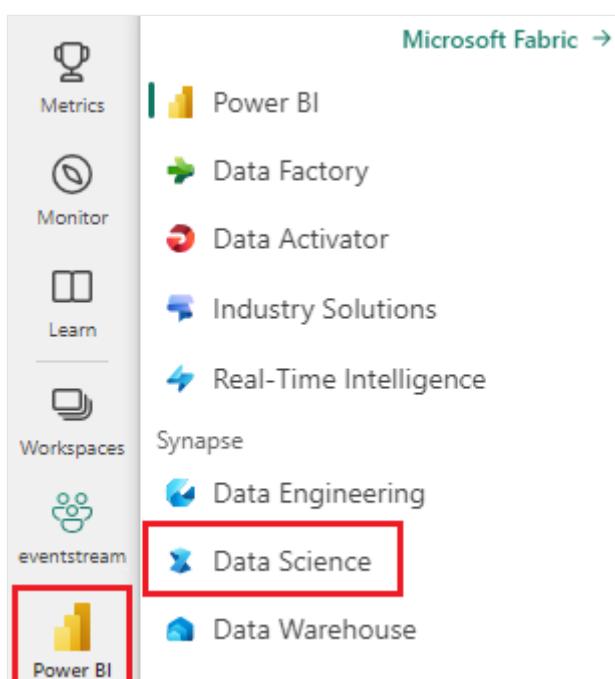
In this article, you learn to use the SemPy semantic link functions to discover and validate relationships in your Power BI semantic models and pandas DataFrames.

In data science and machine learning, it's important to understand the structure and relationships within your data. Power BI is a powerful tool that allows you to model and visualize these structures and relationships. To gain more insights or build machine learning models, you can dive deeper by using the semantic link functions in the SemPy library modules.

Data scientists and business analysts can use SemPy functions to list, visualize, and validate relationships in Power BI semantic models, or find and validate relationships in pandas DataFrames.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



- Create a [new notebook](#) to copy/paste code into cells.
- For Spark 3.4 and above, semantic link is available in the default runtime when using Fabric, and there's no need to install it. For Spark 3.3 or below, or to update to the latest version of semantic link, run the following command:

```
Python
```

```
%pip install -U semantic-link
```

- Add a [lakehouse](#) to your notebook.

List relationships in semantic models

The `list_relationships` function in the `sempy.fabric` module returns a list of all relationships found in a Power BI semantic model. The list helps you understand the structure of your data and how different tables and columns are connected.

This function works by using semantic link to provide annotated DataFrames. The DataFrames include the necessary metadata to understand the relationships within the semantic model. The annotated DataFrames make it easy to analyze the semantic model's structure and use it in machine learning models or other data analysis tasks.

To use the `list_relationships` function, you first import the `sempy.fabric` module. Then you call the function by using the name or UUID of your Power BI semantic model, as shown in the following example:

```
Python
```

```
import sempy.fabric as fabric
fabric.list_relationships("my_dataset")
```

The preceding code calls the `list_relationships` function with a Power BI semantic model called *my_dataset*. The function returns a pandas DataFrame with one row per relationship, allowing you to easily explore and analyze the relationships within the semantic model.

ⓘ Note

Your notebook, Power BI dataset semantic model, and [lakehouse](#) can be located in the same workspace or in different workspaces. By default, SemPy tries to access

your semantic model from:

- The workspace of your lakehouse, if you attached a lakehouse to your notebook.
- The workspace of your notebook, if there's no lakehouse attached.

If your semantic model isn't located in either of these workspaces, you must specify the workspace of your semantic model when you call a SemPy method.

Visualize relationships in semantic models

The `plot_relationship_metadata` function helps you visualize relationships in a semantic model so you can gain a better understanding of the model's structure. This function creates a graph that displays the connections between tables and columns. The graph makes it easier to understand the semantic model's structure and how different elements are related.

The following example shows how to use the `plot_relationship_metadata` function:

Python

```
import sempy.fabric as fabric
from sempy.relationships import plot_relationship_metadata

relationships = fabric.list_relationships("my_dataset")
plot_relationship_metadata(relationships)
```

In the preceding code, the `list_relationships` function retrieves the relationships in the `my_dataset` semantic model, and the `plot_relationship_metadata` function creates a graph to visualize the relationships.

You can customize the graph by defining which columns to include, specifying how to handle missing keys, and providing more [graphviz](#) attributes.

Validate relationships in semantic models

Now that you have a better understanding of the relationships in your semantic model, you can use the `list_relationshipViolations` function to validate these relationships and identify any potential issues or inconsistencies. The `list_relationshipViolations` function helps you validate the content of your tables to ensure that they match the relationships defined in your semantic model.

By using this function, you can identify inconsistencies with the specified relationship multiplicity and address any issues before they impact your data analysis or machine learning models.

To use the `list_relationshipViolations` function, first you import the `sempy.fabric` module and read the tables from your semantic model. Then, you call the function with a dictionary that maps table names to the DataFrames with table content.

The following example code shows how to list relationship violations:

Python

```
import sempy.fabric as fabric

tables = {
    "Sales": fabric.read_table("my_dataset", "Sales"),
    "Products": fabric.read_table("my_dataset", "Products"),
    "Customers": fabric.read_table("my_dataset", "Customers"),
}

fabric.list_relationshipViolations(tables)
```

The preceding code calls the `list_relationshipViolations` function with a dictionary that contains the *Sales*, *Products*, and *Customers* tables from the *my_dataset* semantic model. You can customize the function by setting a coverage threshold, specifying how to handle missing keys, and defining the number of missing keys to report.

The function returns a pandas DataFrame with one row per relationship violation, allowing you to easily identify and address any issues within your semantic model. By using the `list_relationshipViolations` function, you can ensure that your semantic model is consistent and accurate, allowing you to build more reliable machine learning models and gain deeper insights into your data.

Find relationships in pandas DataFrames

While the `list_relationships`, `plot_relationships_df` and `list_relationshipViolations` functions in the Fabric module are powerful tools for exploring relationships within semantic models, you might also need to discover relationships within other data sources imported as pandas DataFrames.

This is where the `find_relationships` function in the `sempy.relationship` module comes into play.

The `find_relationships` function in the `sempy.relationships` module helps data scientists and business analysts discover potential relationships within a list of pandas DataFrames. By using this function, you can identify possible connections between tables and columns, allowing you to better understand the structure of your data and how different elements are related.

The following example code shows how to find relationships in pandas DataFrames:

Python

```
from sempy.relationships import find_relationships

tables = [df_sales, df_products, df_customers]

find_relationships(tables)
```

The preceding code calls the `find_relationships` function with a list of three Pandas DataFrames: `df_sales`, `df_products`, and `df_customers`. The function returns a pandas DataFrame with one row per potential relationship, allowing you to easily explore and analyze the relationships within your data.

You can customize the function by specifying a coverage threshold, a name similarity threshold, a list of relationships to exclude, and whether to include many-to-many relationships.

Validate relationships in pandas DataFrames

After you discover potential relationships in your pandas DataFrames by using the `find_relationships` function, you can use the `list_relationshipViolations` function to validate these relationships and identify any potential issues or inconsistencies.

The `list_relationshipViolations` function validates the content of your tables to ensure that they match the discovered relationships. By using this function to identify inconsistencies with the specified relationship multiplicity, you can address any issues before they impact your data analysis or machine learning models.

The following example code shows how to find relationship violations in pandas DataFrames:

Python

```
from sempy.relationships import find_relationships,
list_relationshipViolations
```

```
tables = [df_sales, df_products, df_customers]
relationships = find_relationships(tables)

list_relationshipViolations(tables, relationships)
```

The preceding code calls the `list_relationship_violations` function with a list of three pandas DataFrames, `df_sales`, `df_products`, and `df_customers`, plus the relationships DataFrame from the `find_relationships` function. The `list_relationship_violations` function returns a pandas DataFrame with one row per relationship violation, allowing you to easily identify and address any issues within your data.

You can customize the function by setting a coverage threshold, specifying how to handle missing keys, and defining the number of missing keys to report.

By using the `list_relationship_violations` function with pandas DataFrames, you can ensure that your data is consistent and accurate, allowing you to build more reliable machine learning models and gain deeper insights into your data.

Related content

- [Learn about semantic functions](#)
- [Get started with the SemPy reference documentation](#)
- [Tutorial: Discover relationships in a semantic model by using semantic link](#)
- [Tutorial: Discover relationships in the Synthea dataset by using semantic link](#)
- [Detect, explore, and validate functional dependencies in your data](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Overview of LightGBM in SynapseML

Article • 06/25/2024

[LightGBM](#) is an open-source, distributed, high-performance gradient boosting (GBDT, GBRT, GBM, or MART) framework. This framework specializes in creating high-quality and GPU-enabled decision tree algorithms for ranking, classification, and many other machine learning tasks. LightGBM is part of Microsoft's [DMTK](#) project.

Advantages of LightGBM

- **Composability:** LightGBM models can be incorporated into existing SparkML pipelines and used for batch, streaming, and serving workloads.
- **Performance:** LightGBM on Spark is 10-30% faster than SparkML on the [Higgs dataset](#) and achieves a 15% increase in AUC. [Parallel experiments](#) have verified that LightGBM can achieve a linear speed-up by using multiple machines for training in specific settings.
- **Functionality:** LightGBM offers a wide array of [tunable parameters](#), that one can use to customize their decision tree system. LightGBM on Spark also supports new types of problems such as quantile regression.
- **Cross platform:** LightGBM on Spark is available on Spark, PySpark, and SparklyR.

LightGBM Usage

- **LightGBMClassifier:** used for building classification models. For example, to predict whether a company bankrupts or not, we could build a binary classification model with `LightGBMClassifier`.
- **LightGBMRegressor:** used for building regression models. For example, to predict housing price, we could build a regression model with `LightGBMRegressor`.
- **LightGBMRanker:** used for building ranking models. For example, to predict the relevance of website search results, we could build a ranking model with `LightGBMRanker`.

Related content

- [How to use LightGBM models with SynapseML in Microsoft Fabric](#)
- [How to use Azure AI services with SynapseML](#)
- [How to perform the same classification task with and without SynapseML](#)
- [How to use KNN model with SynapseML](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Azure AI services in SynapseML with bring your own key

Article • 11/15/2023

[Azure AI services](#) is a suite of APIs, SDKs, and services that developers can use to add cognitive features to their applications, thereby building intelligent applications. AI services empower developers even when they don't have direct AI or data science skills or knowledge. The goal of Azure AI services is to help developers create applications that can see, hear, speak, understand, and even begin to reason. The catalog of services within Azure AI services can be categorized into five main pillars: **Vision**, **Speech**, **Language**, **Web search**, and **Decision**.

ⓘ Note

Fabric seamlessly integrates with Azure AI services, allowing you to enrich your data with [Azure OpenAI Service](#), [Text Analytics](#), [Azure AI Translator](#). This is currently in public preview, for more information about the the prebuilt AI services in Fabric, see [AI services in Fabric](#).

Usage of Azure AI services with bring your own key

Vision

[Azure AI Vision](#)

- Describe: provides description of an image in human readable language ([Scala](#), [Python](#))
- Analyze (color, image type, face, adult/racy content): analyzes visual features of an image ([Scala](#), [Python](#))
- OCR: reads text from an image ([Scala](#), [Python](#))
- Recognize Text: reads text from an image ([Scala](#), [Python](#))
- Thumbnail: generates a thumbnail of user-specified size from the image ([Scala](#), [Python](#))
- Recognize domain-specific content: recognizes domain-specific content (celebrity, landmark) ([Scala](#), [Python](#))
- Tag: identifies list of words that are relevant to the input image ([Scala](#), [Python](#))

Azure AI Face

- Detect: detects human faces in an image ([Scala ↗](#), [Python ↗](#))
- Verify: verifies whether two faces belong to a same person, or a face belongs to a person ([Scala ↗](#), [Python ↗](#))
- Identify: finds the closest matches of the specific query person face from a person group ([Scala ↗](#), [Python ↗](#))
- Find similar: finds similar faces to the query face in a face list ([Scala ↗](#), [Python ↗](#))
- Group: divides a group of faces into disjoint groups based on similarity ([Scala ↗](#), [Python ↗](#))

Speech

[Azure AI Speech ↗](#)

- Speech-to-text: transcribes audio streams ([Scala ↗](#), [Python ↗](#))
- Conversation Transcription: transcribes audio streams into live transcripts with identified speakers. ([Scala ↗](#), [Python ↗](#))
- Text to Speech: Converts text to realistic audio ([Scala ↗](#), [Python ↗](#))

Language

[Text Analytics ↗](#)

- Language detection: detects language of the input text ([Scala ↗](#), [Python ↗](#))
- Key phrase extraction: identifies the key talking points in the input text ([Scala ↗](#), [Python ↗](#))
- Named entity recognition: identifies known entities and general named entities in the input text ([Scala ↗](#), [Python ↗](#))
- Sentiment analysis: returns a score between 0 and 1 indicating the sentiment in the input text ([Scala ↗](#), [Python ↗](#))
- Healthcare Entity Extraction: Extracts medical entities and relationships from text. ([Scala ↗](#), [Python ↗](#))

Translation

[Azure AI Translator ↗](#)

- Translate: Translates text. ([Scala ↗](#), [Python ↗](#))
- Transliterate: Converts text in one language from one script to another script. ([Scala ↗](#), [Python ↗](#))
- Detect: Identifies the language of a piece of text. ([Scala ↗](#), [Python ↗](#))

- BreakSentence: Identifies the positioning of sentence boundaries in a piece of text. ([Scala ↗](#), [Python ↗](#))
- Dictionary Lookup: Provides alternative translations for a word and a few idiomatic phrases. ([Scala ↗](#), [Python ↗](#))
- Dictionary Examples: Provides examples that show how terms in the dictionary are used in context. ([Scala ↗](#), [Python ↗](#))
- Document Translation: Translates documents across all supported languages and dialects while preserving document structure and data format. ([Scala ↗](#), [Python ↗](#))

Azure AI Document Intelligence

[Azure AI Document Intelligence ↗](#)

- Analyze Layout: Extract text and layout information from a given document. ([Scala ↗](#), [Python ↗](#))
- Analyze Receipts: Detects and extracts data from receipts using optical character recognition (OCR) and our receipt model. This functionality makes it easy to extract structured data from receipts such as merchant name, merchant phone number, transaction date, transaction total, and more. ([Scala ↗](#), [Python ↗](#))
- Analyze Business Cards: Detects and extracts data from business cards using optical character recognition (OCR) and our business card model. This functionality makes it easy to extract structured data from business cards such as contact names, company names, phone numbers, emails, and more. ([Scala ↗](#), [Python ↗](#))
- Analyze Invoices: Detects and extracts data from invoices using optical character recognition (OCR) and our invoice understanding deep learning models. This functionality makes it easy to extract structured data from invoices such as customer, vendor, invoice ID, invoice due date, total, invoice amount due, tax amount, ship to, bill to, line items and more. ([Scala ↗](#), [Python ↗](#))
- Analyze ID Documents: Detects and extracts data from identification documents using optical character recognition (OCR) and our ID document model, enabling you to easily extract structured data from ID documents such as first name, last name, date of birth, document number, and more. ([Scala ↗](#), [Python ↗](#))
- Analyze Custom Form: Extracts information from forms (PDFs and images) into structured data based on a model created from a set of representative training forms. ([Scala ↗](#), [Python ↗](#))
- Get Custom Model: Get detailed information about a custom model. ([Scala ↗](#), [Python ↗](#))
- List Custom Models: Get information about all custom models. ([Scala ↗](#), [Python ↗](#))

Decision

Azure AI Anomaly Detector

- Anomaly status of latest point: generates a model using preceding points and determines whether the latest point is anomalous ([Scala](#), [Python](#))
- Find anomalies: generates a model using an entire series and finds anomalies in the series ([Scala](#), [Python](#))

Search

- Bing Image search  ([Scala](#), [Python](#))
- Azure Cognitive Search ([Scala](#), [Python](#))

Next steps

- Use Azure AI services with SynapseML in Microsoft Fabric
- Use Azure AI services with SynapseML for multivariate anomaly detection
- Create a custom search engine and question-answering system

Feedback

Was this page helpful?



[Provide product feedback](#) | [Ask the community](#)

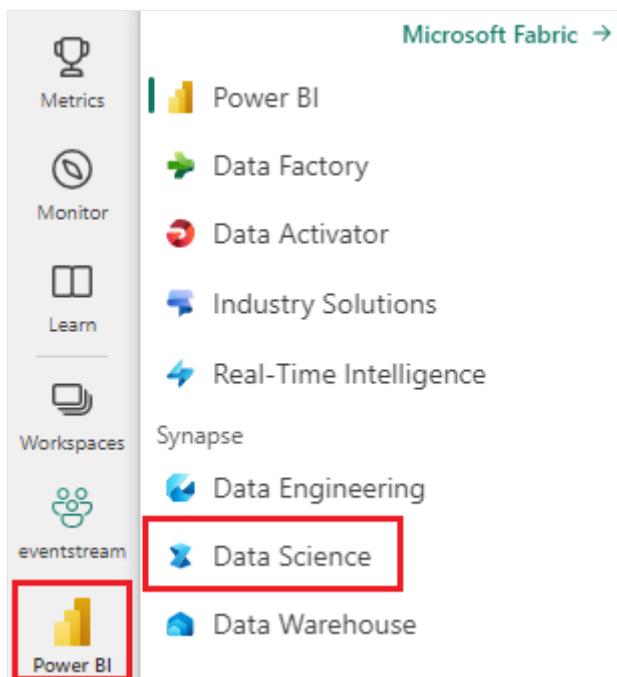
Build a model with SynapseML

Article • 06/09/2024

This article describes how to build a machine learning model by using SynapseML, and demonstrates how SynapseML can simplify complex machine learning tasks. You use SynapseML to create a small machine learning training pipeline that includes a featurization stage and a LightGBM regression stage. The pipeline predicts ratings based on review text from a dataset of book reviews. You also see how SynapseML can simplify the use of prebuilt models to solve machine learning problems.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



Prepare resources

Create the tools and resources you need to build the model and pipeline.

1. [Create a new notebook](#).
2. Attach your notebook to a lakehouse. To add an existing lakehouse or create a new one, expand **Lakehouses** under **Explorer** at left, and then select **Add**.

3. Get an Azure AI services key by following the instructions in [Quickstart: Create a multi-service resource for Azure AI services](#).
4. [Create an Azure Key Vault instance](#) and add your Azure AI services key to the key vault as a secret.
5. Make a note of your key vault name and secret name. You need this information to run the one-step transform later in this article.

Set up the environment

In your notebook, import SynapseML libraries and initialize your Spark session.

Python

```
from pyspark.sql import SparkSession
from synapse.ml.core.platform import *

spark = SparkSession.builder.getOrCreate()
```

Load a dataset

Load your dataset and split it into train and test sets.

Python

```
train, test = (
    spark.read.parquet(
        "wasbs://publicwasb@mmlspark.blob.core.windows.net/BookReviewsFromAmazon10K.
        parquet"
    )
    .limit(1000)
    .cache()
    .randomSplit([0.8, 0.2])
)

display(train)
```

Create the training pipeline

Create a pipeline that featurizes data using `TextFeaturizer` from the `synapse.ml.featurize.text` library and derives a rating using the `LightGBMRegressor` function.

Python

```
from pyspark.ml import Pipeline
from synapse.ml.featurize.text import TextFeaturizer
from synapse.ml.lightgbm import LightGBMRegressor

model = Pipeline(
    stages=[
        TextFeaturizer(inputCol="text", outputCol="features"),
        LightGBMRegressor(featuresCol="features", labelCol="rating",
dataTransferMode="bulk")
    ]
).fit(train)
```

Predict the output of the test data

Call the `transform` function on the model to predict and display the output of the test data as a dataframe.

Python

```
display(model.transform(test))
```

Use Azure AI services to transform data in one step

Alternatively, for these kinds of tasks that have a prebuilt solution, you can use SynapseML's integration with Azure AI services to transform your data in one step. Run the following code with these replacements:

- Replace `<secret-name>` with the name of your Azure AI Services key secret.
- Replace `<key-vault-name>` with the name of your key vault.

Python

```
from synapse.ml.services import TextSentiment
from synapse.ml.core.platform import find_secret

model = TextSentiment(
    textCol="text",
    outputCol="sentiment",
    subscriptionKey=find_secret("<secret-name>", "<key-vault-name>")
).setLocation("eastus")
```

```
display(model.transform(test))
```

Related content

- [How to use LightGBM with SynapseML](#)
 - [How to use Azure AI services with SynapseML](#)
 - [How to perform the same classification task with and without SynapseML](#)
 - [Quickstart: Create a multi-service resource for Azure AI services](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Install a different version of SynapseML on Fabric

Article • 04/06/2025

SynapseML is preinstalled on Fabric. You can install other versions with %%configure.

ⓘ Note

Fabric notebook doesn't officially support %%configure for now, and there's no guarantee of service-level agreement or future compatibility with official releases.

Install SynapseML with %%configure

The following example installs SynapseML v0.11.1 on Fabric. To use the example, paste it into a code cell in a notebook and run the cell:

Python

```
%%configure -f
{
  "name": "synapsem1",
  "conf": {
    "spark.jars.packages": "com.microsoft.azure:synapsem1_2.12:0.11.1,org.apache.spark:spark-
avro_2.12:3.3.1",
    "spark.jars.repositories": "https://mmlspark.azureedge.net/maven",
    "spark.jars.excludes": "org.scala-lang:scala-
reflect,org.apache.spark:spark-
tags_2.12,org.scalactic:scalactic_2.12,org.scalatest:scalatest_2.12,com.fast
erxml.jackson.core:jackson-databind",
    "spark.yarn.user.classpath.first": "true",
    "spark.sql.parquet.enableVectorizedReader": "false",
    "spark.sql.legacy.replaceDatabricksSparkAvro.enabled": "true"
  }
}
```

Check SynapseML version

To verify a successful installation, run the following code in a cell. The version number returned should match the version number you installed (0.11.1).

Python

```
import synapse.ml.cognitive
print(f"SynapseML cognitive version: {synapse.ml.cognitive.__version__}")
```

Python

```
import synapse.ml.lightgbm
print(f"SynapseML lightgbm version: {synapse.ml.lightgbm.__version__}")
```

Related content

- [How to use LightGBM with SynapseML](#)
- [How to use Azure AI services with SynapseML](#)
- [How to perform the same classification task with and without SynapseML](#)

Feedback

Was this page helpful?



[Provide product feedback](#) | [Ask the community](#)

Azure OpenAI for big data

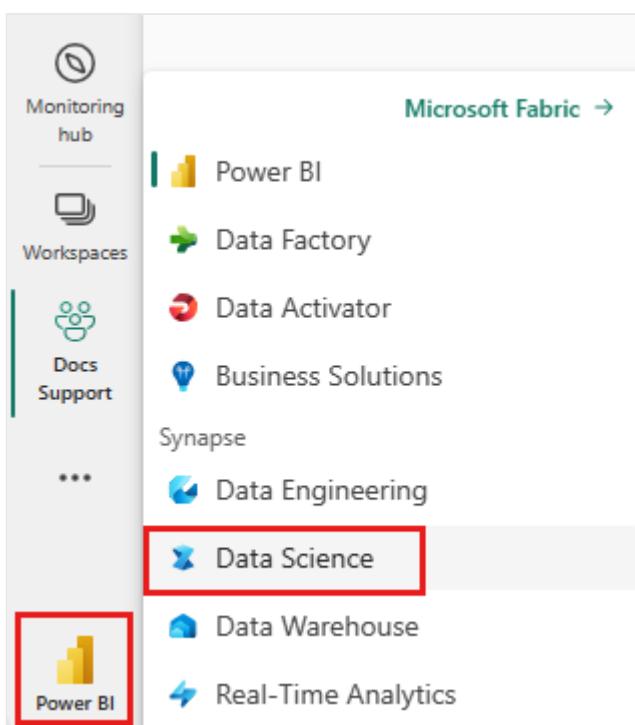
Article • 12/29/2023

The Azure OpenAI service can be used to solve a large number of natural language tasks through prompting the completion API. To make it easier to scale your prompting workflows from a few examples to large datasets of examples, we integrated the Azure OpenAI service with the distributed machine learning library [SynapseML](#). This integration makes it easy to use the [Apache Spark](#) distributed computing framework to process millions of prompts with the OpenAI service. This tutorial shows how to apply large language models at a distributed scale using Azure Open AI and Azure Synapse Analytics.

Prerequisites

The key prerequisites for this quickstart include a working Azure OpenAI resource, and an Apache Spark cluster with SynapseML installed.

- Get a [Microsoft Fabric subscription](#). Or sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Switch to the Data Science experience by using the experience switcher icon on the left side of your home page.



- Go to the Data Science experience in Microsoft Fabric.
- Create a new notebook.

- An Azure OpenAI resource: [Request Access to Azure OpenAI Service](#) before creating a resource

Import this guide as a notebook

The next step is to add this code into your Spark cluster. You can either create a notebook in your Spark platform and copy the code into this notebook to run the demo. Or download the notebook and import it into Synapse Analytics

1. [Download this demo as a notebook](#) (select **Raw**, then save the file)
2. Import the notebook [into the Synapse Workspace](#) or if using Fabric [import into the Fabric Workspace](#)
3. Install SynapseML on your cluster. See the installation instructions for Synapse at the bottom of [the SynapseML website](#). If you are using Fabric, check [Installation Guide](#). This requires pasting an extra cell at the top of the notebook you imported.
4. Connect your notebook to a cluster and follow along, editing and running the cells.

Fill in service information

Next, edit the cell in the notebook to point to your service. In particular set the `service_name`, `deployment_name`, `location`, and `key` variables to match them to your OpenAI service:

Python

```
import os
from pyspark.sql import SparkSession
from synapse.ml.core.platform import running_on_synapse, find_secret

# Bootstrap Spark Session
spark = SparkSession.builder.getOrCreate()

if running_on_synapse():
    from notebookutils.visualization import display

# Fill in the following lines with your service information
# Learn more about selecting which embedding model to choose:
https://openai.com/blog/new-and-improved-embedding-model
service_name = "synapseml-openai"
deployment_name = "gpt-35-turbo"
deployment_name_embeddings = "text-embedding-ada-002"

key = find_secret(
    "openai-api-key"
) # please replace this line with your key as a string
```

```
assert key is not None and service_name is not None
```

Create a dataset of prompts

Next, create a dataframe consisting of a series of rows, with one prompt per row.

You can also load data directly from ADLS or other databases. For more information on loading and preparing Spark dataframes, see the [Apache Spark data loading guide ↗](#).

Python

```
df = spark.createDataFrame(  
    [  
        ("Hello my name is",),  
        ("The best code is code thats",),  
        ("SynapseML is ",),  
    ]  
).toDF("prompt")
```

Create the OpenAICompletion Apache Spark Client

To apply the OpenAI Completion service to your dataframe you created, create an `OpenAICompletion` object, which serves as a distributed client. Parameters of the service can be set either with a single value, or by a column of the dataframe with the appropriate setters on the `OpenAICompletion` object. Here we're setting `maxTokens` to 200. A token is around four characters, and this limit applies to the sum of the prompt and the result. We're also setting the `promptCol` parameter with the name of the prompt column in the dataframe.

Python

```
from synapse.ml.cognitive import OpenAICompletion  
  
completion = (  
    OpenAICompletion()  
    .setSubscriptionKey(key)  
    .setDeploymentName(deployment_name)  
    .setCustomServiceName(service_name)  
    .setMaxTokens(200)  
    .setPromptCol("prompt")  
    .setErrorCol("error")
```

```
.setOutputCol("completions")
)
```

Transform the dataframe with the OpenAICompletion Client

After completing dataframe and the completion client, you can transform your input dataset and add a column called `completions` with all of the information the service adds. Select just the text for simplicity.

Python

```
from pyspark.sql.functions import col

completed_df = completion.transform(df).cache()
display(
    completed_df.select(
        col("prompt"),
        col("error"),
        col("completions.choices.text").getItem(0).alias("text"),
    )
)
```

Your output should look something like this. The completion text will be different from the sample.

[Expand table](#)

prompt	error	text
Hello my name is	null	Makaveli I'm eighteen years old and I want to be a rapper when I grow up I love writing and making music I'm from Los Angeles, CA
The best code is	code thats	understandable This is a subjective statement, and there is no definitive answer.
SynapseML is	null	A machine learning algorithm that is able to learn how to predict the future outcome of events.

More Usage Examples

Generating Text Embeddings

In addition to completing text, we can also embed text for use in downstream algorithms or vector retrieval architectures. Creating embeddings allows you to search and retrieve documents from large collections and can be used when prompt engineering isn't sufficient for the task. For more information on using `OpenAIEmbedding`, see our [embedding guide](#).

Python

```
from synapse.ml.cognitive import OpenAIEmbedding

embedding = (
    OpenAIEmbedding()
    .setSubscriptionKey(key)
    .setDeploymentName(deployment_name_embeddings)
    .setCustomServiceName(service_name)
    .setTextCol("prompt")
    .setErrorCol("error")
    .setOutputCol("embeddings")
)

display(embedding.transform(df))
```

Chat Completion

Models such as ChatGPT and GPT-4 are capable of understanding chats instead of single prompts. The `OpenAIChatCompletion` transformer exposes this functionality at scale.

Python

```
from synapse.ml.cognitive import OpenAIChatCompletion
from pyspark.sql import Row
from pyspark.sql.types import *

def make_message(role, content):
    return Row(role=role, content=content, name=role)

chat_df = spark.createDataFrame(
    [
        (
            [
                make_message(
                    "system", "You are an AI chatbot with red as your
favorite color"
                ),
                make_message("user", "Whats your favorite color"),
            ],
        )
    ]
)
```

```

),
(
[
    make_message("system", "You are very excited"),
    make_message("user", "How are you today"),
],
),
]
).toDF("messages")

chat_completion = (
    OpenAIChatCompletion()
    .setSubscriptionKey(key)
    .setDeploymentName(deployment_name)
    .setCustomServiceName(service_name)
    .setMessagesCol("messages")
    .setErrorCol("error")
    .setOutputCol("chat_completions")
)
display(
    chat_completion.transform(chat_df).select(
        "messages", "chat_completions.choices.message.content"
    )
)
)

```

Improve throughput with request batching

The example makes several requests to the service, one for each prompt. To complete multiple prompts in a single request, use batch mode. First, in the OpenAICompletion object, instead of setting the Prompt column to "Prompt", specify "batchPrompt" for the BatchPrompt column. To do so, create a dataframe with a list of prompts per row.

As of this writing there's currently a limit of 20 prompts in a single request, and a hard limit of 2048 "tokens", or approximately 1500 words.

Python

```

batch_df = spark.createDataFrame(
    [
        (["The time has come", "Pleased to", "Today stocks", "Here's to"],),
        (["The only thing", "Ask not what", "Every litter", "I am"],),
    ]
).toDF("batchPrompt")

```

Next we create the OpenAICompletion object. Rather than setting the prompt column, set the batchPrompt column if your column is of type `ArrayType[String]`.

Python

```
batch_completion = (
    OpenAICompletion()
    .setSubscriptionKey(key)
    .setDeploymentName(deployment_name)
    .setCustomServiceName(service_name)
    .setMaxTokens(200)
    .setBatchPromptCol("batchPrompt")
    .setErrorCol("error")
    .setOutputCol("completions")
)
```

In the call to transform, a request will be made per row. Since there are multiple prompts in a single row, each request is sent with all prompts in that row. The results contain a row for each row in the request.

Python

```
completed_batch_df = batch_completion.transform(batch_df).cache()
display(completed_batch_df)
```

Using an automatic minibatcher

If your data is in column format, you can transpose it to row format using SynapseML's `FixedMiniBatcherTransformer`.

Python

```
from pyspark.sql.types import StringType
from synapse.ml.stages import FixedMiniBatchTransformer
from synapse.ml.core.spark import FluentAPI

completed_automlbatch_df = (
    df.coalesce(
        1
    ) # Force a single partition so that our little 4-row dataframe makes a
      # batch of size 4, you can remove this step for large datasets
    .mlTransform(FixedMiniBatchTransformer(batchSize=4))
    .withColumnRenamed("prompt", "batchPrompt")
    .mlTransform(batch_completion)
)

display(completed_automlbatch_df)
```

Prompt engineering for translation

The Azure OpenAI service can solve many different natural language tasks through [prompt engineering](#). Here, we show an example of prompting for language translation:

```
Python
```

```
translate_df = spark.createDataFrame(  
    [  
        ("Japanese: Ookina hako \nEnglish: Big box \nJapanese: Midori  
tako\nEnglish:",),  
        (  
            "French: Quel heure et il au Montreal? \nEnglish: What time is  
it in Montreal? \nFrench: Ou est le poulet? \nEnglish:",  
            ),  
    ]  
).toDF("prompt")  
  
display(completion.transform(translate_df))
```

Prompt for question answering

Here, we prompt GPT-3 for general-knowledge question answering:

```
Python
```

```
qa_df = spark.createDataFrame(  
    [  
        (  
            "Q: Where is the Grand Canyon?\nA: The Grand Canyon is in  
Arizona.\n\nQ: What is the weight of the Burj Khalifa in kilograms?\nA:",  
            )  
    ]  
).toDF("prompt")  
  
display(completion.transform(qa_df))
```

Next steps

- [How to Build a Search Engine with SynapseML](#)
- [How to use SynapseML and Azure AI services for multivariate anomaly detection - Analyze time series](#)
- [How to use Kernel SHAP to explain a tabular classification model](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Use LightGBM models with SynapseML in Microsoft Fabric

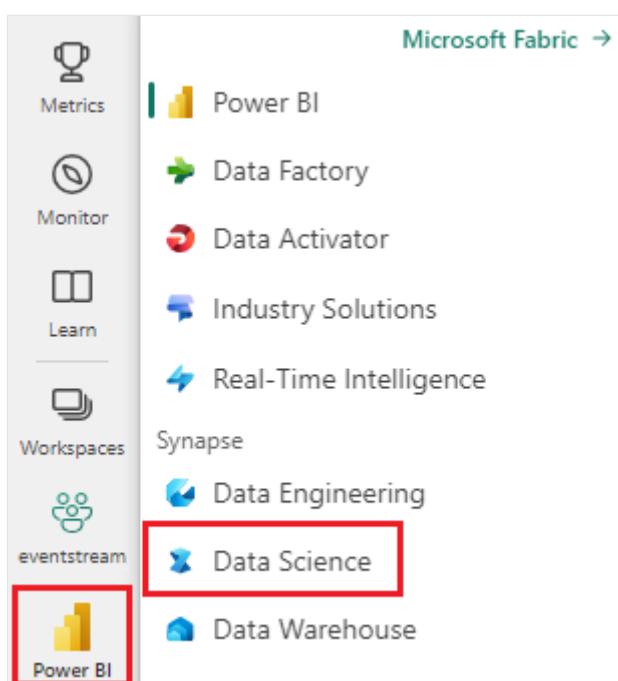
Article • 08/23/2024

The [LightGBM](#) framework specializes in creating high-quality and GPU-enabled decision tree algorithms for ranking, classification, and many other machine learning tasks. In this article, you use LightGBM to build classification, regression, and ranking models.

LightGBM is an open-source, distributed, high-performance gradient boosting (GBDT, GBRT, GBM, or MART) framework. LightGBM is part of Microsoft's [DMTK](#) project. You can use LightGBM by using LightGBMClassifier, LightGBMRegressor, and LightGBMRanker. LightGBM comes with the advantages of being incorporated into existing SparkML pipelines and used for batch, streaming, and serving workloads. It also offers a wide array of tunable parameters, that one can use to customize their decision tree system. LightGBM on Spark also supports new types of problems such as quantile regression.

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#).
- Use the experience switcher on the left side of your home page to switch to the Synapse Data Science experience.



- Go to the Data Science experience in Microsoft Fabric.
- Create a new notebook.
- Attach your notebook to a lakehouse. On the left side of your notebook, select Add to add an existing lakehouse or create a new one.

Use LightGBMClassifier to train a classification model

In this section, you use LightGBM to build a classification model for predicting bankruptcy.

1. Read the dataset.

Python

```
from pyspark.sql import SparkSession

# Bootstrap Spark Session
spark = SparkSession.builder.getOrCreate()

from synapse.ml.core.platform import *
```

Python

```
df = (
    spark.read.format("csv")
    .option("header", True)
    .option("inferSchema", True)
    .load(
        "wasbs://publicwasb@mmlspark.blob.core.windows.net/company_bankruptcy_prediction_data.csv"
    )
)
# print dataset size
print("records read: " + str(df.count()))
print("Schema: ")
df.printSchema()
```

Python

```
display(df)
```

2. Split the dataset into train and test sets.

Python

```
train, test = df.randomSplit([0.85, 0.15], seed=1)
```

3. Add a featurizer to convert features into vectors.

Python

```
from pyspark.ml.feature import VectorAssembler

feature_cols = df.columns[1:]
featurizer = VectorAssembler(inputCols=feature_cols,
outputCol="features")
train_data = featurizer.transform(train)[ "Bankrupt?", "features"]
test_data = featurizer.transform(test)[ "Bankrupt?", "features"]
```

4. Check if the data is unbalanced.

Python

```
display(train_data.groupBy("Bankrupt?").count())
```

5. Train the model using `LightGBMClassifier`.

Python

```
from synapse.ml.lightgbm import LightGBMClassifier

model = LightGBMClassifier(
    objective="binary", featuresCol="features", labelCol="Bankrupt?",
    isUnbalance=True, dataTransferMode="bulk"
)
```

Python

```
model = model.fit(train_data)
```

6. Visualize feature importance

Python

```
import pandas as pd
import matplotlib.pyplot as plt

feature_importances = model.getFeatureImportances()
fi = pd.Series(feature_importances, index=feature_cols)
fi = fi.sort_values(ascending=True)
```

```

f_index = fi.index
f_values = fi.values

# print feature importances
print("f_index:", f_index)
print("f_values:", f_values)

# plot
x_index = list(range(len(fi)))
x_index = [x / len(fi) for x in x_index]
plt.rcParams["figure.figsize"] = (20, 20)
plt.barh(
    x_index, f_values, height=0.028, align="center", color="tan",
    tick_label=f_index
)
plt.xlabel("importances")
plt.ylabel("features")
plt.show()

```

7. Generate predictions with the model

Python

```

predictions = model.transform(test_data)
predictions.limit(10).toPandas()

```

Python

```

from synapse.ml.train import ComputeModelStatistics

metrics = ComputeModelStatistics(
    evaluationMetric="classification",
    labelCol="Bankrupt?",
    scoredLabelsCol="prediction",
).transform(predictions)
display(metrics)

```

Use LightGBMRegressor to train a quantile regression model

In this section, you use LightGBM to build a regression model for drug discovery.

1. Read the dataset.

Python

```
triazines = spark.read.format("libsvm").load(  
    "wasbs://publicwasb@mmlspark.blob.core.windows.net/triazines.scale.svmlight"  
)
```

Python

```
# print some basic info  
print("records read: " + str(triazines.count()))  
print("Schema: ")  
triazines.printSchema()  
display(triazines.limit(10))
```

2. Split the dataset into train and test sets.

Python

```
train, test = triazines.randomSplit([0.85, 0.15], seed=1)
```

3. Train the model using `LightGBMRegressor`.

Python

```
from synapse.ml.lightgbm import LightGBMRegressor  
  
model = LightGBMRegressor(  
    objective="quantile", alpha=0.2, learningRate=0.3, numLeaves=31,  
    dataTransferMode="bulk"  
).fit(train)
```

Python

```
print(model.getFeatureImportances())
```

4. Generate predictions with the model.

Python

```
scoredData = model.transform(test)  
display(scoredData)
```

Python

```
from synapse.ml.train import ComputeModelStatistics

metrics = ComputeModelStatistics(
    evaluationMetric="regression", labelCol="label",
    scoresCol="prediction"
).transform(scoredData)
display(metrics)
```

Use LightGBMRanker to train a ranking model

In this section, you use LightGBM to build a ranking model.

1. Read the dataset.

Python

```
df = spark.read.format("parquet").load(
    "wasbs://publicwasb@mmlspark.blob.core.windows.net/lightGBMRanker_train
    .parquet"
)
# print some basic info
print("records read: " + str(df.count()))
print("Schema: ")
df.printSchema()
display(df.limit(10))
```

2. Train the ranking model using LightGBMRanker.

Python

```
from synapse.ml.lightgbm import LightGBMRanker

features_col = "features"
query_col = "query"
label_col = "labels"
lgbm_ranker = LightGBMRanker(
    labelCol=label_col,
    featuresCol=features_col,
    groupCol=query_col,
    predictionCol="preds",
    leafPredictionCol="leafPreds",
    featuresShapCol="importances",
    repartitionByGroupingColumn=True,
    numLeaves=32,
    numIterations=200,
    evalAt=[1, 3, 5],
    metric="ndcg",
```

```
    dataTransferMode="bulk"
)
```

Python

```
lgbm_ranker_model = lgbm_ranker.fit(df)
```

3. Generate predictions with the model.

Python

```
dt = spark.read.format("parquet").load(
    "wasbs://publicwasb@mmlspark.blob.core.windows.net/lightGBMRanker_test.
    parquet"
)
predictions = lgbm_ranker_model.transform(dt)
predictions.limit(10).toPandas()
```

Related content

- [What is Azure AI services in Azure Synapse Analytics?](#)
- [How to perform the same classification task with and without SynapseML](#)
- [How to use KNN model with SynapseML](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Use Azure AI services with SynapseML in Microsoft Fabric

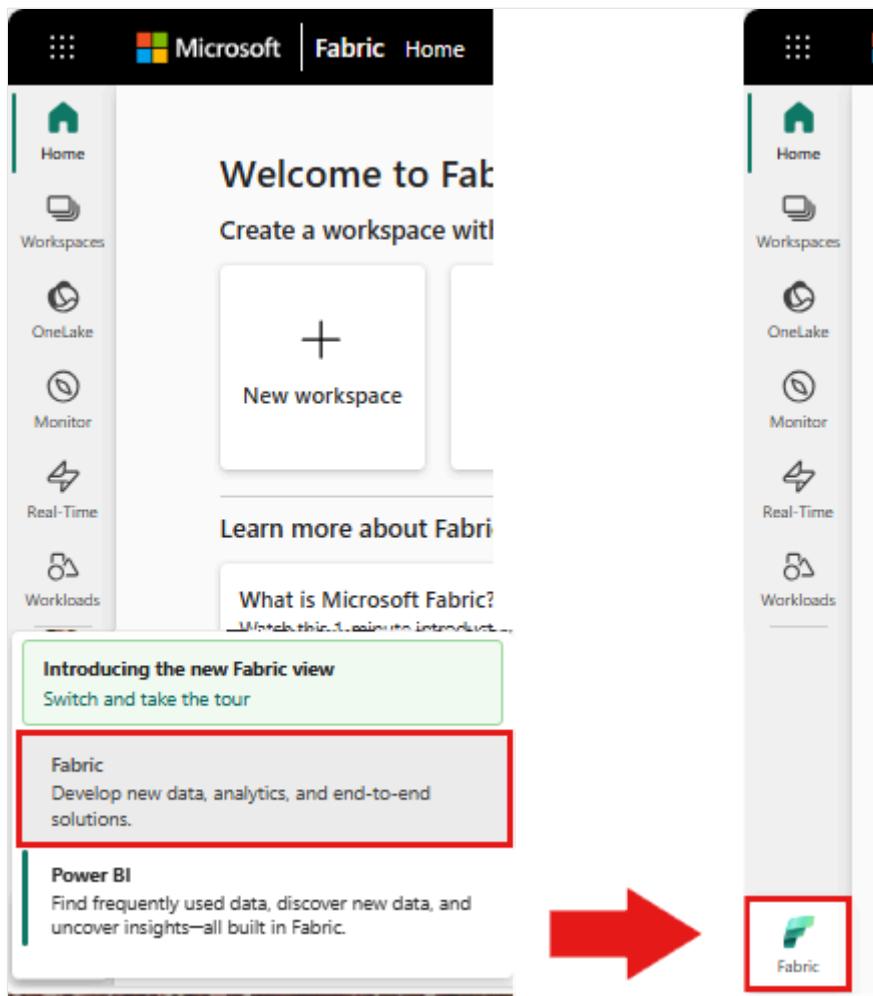
Article • 07/17/2024

Azure AI services [↗](#) help developers and organizations rapidly create intelligent, cutting-edge, market-ready, and responsible applications with out-of-the-box and pre-built and customizable APIs and models. In this article, you'll use the various services available in Azure AI services to perform tasks that include: text analytics, translation, document intelligence, vision, image search, speech to text and text to speech conversion, anomaly detection, and data extraction from web APIs.

The goal of Azure AI services is to help developers create applications that can see, hear, speak, understand, and even begin to reason. The catalog of services within Azure AI services can be categorized into five main pillars: [Vision](#) [↗](#), [Speech](#) [↗](#), [Language](#) [↗](#), [Web search](#) [↗](#), and [Decision](#) [↗](#).

Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Sign in to [Microsoft Fabric](#) [↗](#).
- Use the experience switcher on the bottom left side of your home page to switch to Fabric.



- Create a new notebook.
- Attach your notebook to a lakehouse. On the left side of your notebook, select Add to add an existing lakehouse or create a new one.
- Obtain an Azure AI services key by following [Quickstart: Create a multi-service resource for Azure AI services](#). Copy the value of the key to use in the code samples below.

Prepare your system

To begin, import required libraries and initialize your Spark session.

Python

```
from pyspark.sql.functions import udf, col
from synapse.ml.io.http import HTTPTransformer, http_udf
from requests import Request
from pyspark.sql.functions import lit
from pyspark.ml import PipelineModel
from pyspark.sql.functions import col
import os
```

Python

```
from pyspark.sql import SparkSession
from synapse.ml.core.platform import *

# Bootstrap Spark Session
spark = SparkSession.builder.getOrCreate()
```

Import Azure AI services libraries and replace the keys and locations in the following code snippet with your Azure AI services key and location.

Python

```
from synapse.ml.cognitive import *

# A general Azure AI services key for Text Analytics, Vision and Document
Intelligence (or use separate keys that belong to each service)
service_key = "<YOUR-KEY-VALUE>" # Replace <YOUR-KEY-VALUE> with your Azure
AI service key, check prerequisites for more details
service_loc = "eastus"

# A Bing Search v7 subscription key
bing_search_key = "<YOUR-KEY-VALUE>" # Replace <YOUR-KEY-VALUE> with your
Bing v7 subscription key, check prerequisites for more details

# An Anomaly Detector subscription key
anomaly_key = "<YOUR-KEY-VALUE>" # Replace <YOUR-KEY-VALUE> with your
anomaly service key, check prerequisites for more details
anomaly_loc = "westus2"

# A Translator subscription key
translator_key = "<YOUR-KEY-VALUE>" # Replace <YOUR-KEY-VALUE> with your
translator service key, check prerequisites for more details
translator_loc = "eastus"

# An Azure search key
search_key = "<YOUR-KEY-VALUE>" # Replace <YOUR-KEY-VALUE> with your search
key, check prerequisites for more details
```

Perform sentiment analysis on text

The [Text Analytics](#) service provides several algorithms for extracting intelligent insights from text. For example, you can use the service to find the sentiment of some input text. The service will return a score between 0.0 and 1.0, where low scores indicate negative sentiment and high scores indicate positive sentiment.

The following code sample returns the sentiment for three simple sentences.

Python

```

# Create a dataframe that's tied to it's column names
df = spark.createDataFrame(
    [
        ("I am so happy today, its sunny!", "en-US"),
        ("I am frustrated by this rush hour traffic", "en-US"),
        ("The cognitive services on spark aint bad", "en-US"),
    ],
    ["text", "language"],
)

# Run the Text Analytics service with options
sentiment = (
    TextSentiment()
    .setTextCol("text")
    .setLocation(service_loc)
    .setSubscriptionKey(service_key)
    .setOutputCol("sentiment")
    .setErrorCol("error")
    .setLanguageCol("language")
)

# Show the results of your text query in a table format
display(
    sentiment.transform(df).select(
        "text", col("sentiment.document.sentiment").alias("sentiment")
    )
)

```

Perform text analytics for health data

The [Text Analytics for Health Service](#) extracts and labels relevant medical information from unstructured text such as doctor's notes, discharge summaries, clinical documents, and electronic health records.

The following code sample analyzes and transforms text from doctors notes into structured data.

Python

```

df = spark.createDataFrame(
    [
        ("20mg of ibuprofen twice a day",),
        ("1tsp of Tylenol every 4 hours",),
        ("6-drops of Vitamin B-12 every evening",),
    ],
    ["text"],
)

healthcare = (

```

```
AnalyzeHealthText()
    .setSubscriptionKey(service_key)
    .setLocation(service_loc)
    .setLanguage("en")
    .setOutputCol("response")
)

display(healthcare.transform(df))
```

Translate text into a different language

Translator [↗](#) is a cloud-based machine translation service and is part of the Azure AI services family of cognitive APIs used to build intelligent apps. Translator is easy to integrate in your applications, websites, tools, and solutions. It allows you to add multi-language user experiences in 90 languages and dialects and can be used for text translation with any operating system.

The following code sample does a simple text translation by providing the sentences you want to translate and target languages you want to translate them to.

Python

```
from pyspark.sql.functions import col, flatten

# Create a dataframe including sentences you want to translate
df = spark.createDataFrame(
    [[["Hello, what is your name?", "Bye"]],),
    [
        "text",
    ],
)

# Run the Translator service with options
translate = (
    Translate()
    .setSubscriptionKey(translator_key)
    .setLocation(translator_loc)
    .setTextCol("text")
    .setToLanguage(["zh-Hans"])
    .setOutputCol("translation")
)

# Show the results of the translation.
display(
    translate.transform(df)
    .withColumn("translation", flatten(col("translation.translations")))
    .withColumn("translation", col("translation.text"))
    .select("translation")
)
```

Extract information from a document into structured data

Azure AI Document Intelligence [↗](#) is a part of Azure AI services that lets you build automated data processing software using machine learning technology. With Azure AI Document Intelligence, you can identify and extract text, key/value pairs, selection marks, tables, and structure from your documents. The service outputs structured data that includes the relationships in the original file, bounding boxes, confidence and more.

The following code sample analyzes a business card image and extracts its information into structured data.

Python

```
from pyspark.sql.functions import col, explode

# Create a dataframe containing the source files
imageDf = spark.createDataFrame(
    [
        (
            "https://mmlspark.blob.core.windows.net/datasets/FormRecognizer/business_card.jpg",
        )
    ],
    [
        "source",
    ],
)

# Run the Form Recognizer service
analyzeBusinessCards = (
    AnalyzeBusinessCards()
        .setSubscriptionKey(service_key)
        .setLocation(service_loc)
        .setImageUrlCol("source")
        .setOutputCol("businessCards")
)

# Show the results of recognition.
display(
    analyzeBusinessCards.transform(imageDf)
        .withColumn(
            "documents",
            explode(col("businessCards.analyzeResult.documentResults.fields")))
        )
        .select("source", "documents")
)
```

Analyze and tag images

Computer Vision [↗](#) analyzes images to identify structure such as faces, objects, and natural-language descriptions.

The following code sample analyzes images and labels them with *tags*. Tags are one-word descriptions of things in the image, such as recognizable objects, people, scenery, and actions.

Python

```
# Create a dataframe with the image URLs
base_url = "https://raw.githubusercontent.com/Azure-Samples/cognitive-
services-sample-data-files/master/ComputerVision/Images/"
df = spark.createDataFrame(
    [
        (base_url + "objects.jpg",),
        (base_url + "dog.jpg",),
        (base_url + "house.jpg",),
    ],
    [
        "image",
    ],
)
# Run the Computer Vision service. Analyze Image extracts information
# from/about the images.
analysis = (
    AnalyzeImage()
    .setLocation(service_loc)
    .setSubscriptionKey(service_key)
    .setVisualFeatures(
        ["Categories", "Color", "Description", "Faces", "Objects", "Tags"]
    )
    .setOutputCol("analysis_results")
    .setImageUrlCol("image")
    .setErrorCol("error")
)
# Show the results of what you wanted to pull out of the images.
display(analysis.transform(df).select("image",
    "analysis_results.description.tags"))
```

Search for images that are related to a natural language query

Bing Image Search [↗](#) searches the web to retrieve images related to a user's natural language query.

The following code sample uses a text query that looks for images with quotes. The output of the code is a list of image URLs that contain photos related to the query.

Python

```
# Number of images Bing will return per query
imgsPerBatch = 10
# A list of offsets, used to page into the search results
offsets = [(i * imgsPerBatch,) for i in range(100)]
# Since web content is our data, we create a dataframe with options on that
# data: offsets
bingParameters = spark.createDataFrame(offsets, ["offset"])

# Run the Bing Image Search service with our text query
bingSearch = (
    BingImageSearch()
    .setSubscriptionKey(bing_search_key)
    .setOffsetCol("offset")
    .setQuery("Martin Luther King Jr. quotes")
    .setCount(imgsPerBatch)
    .setOutputCol("images")
)

# Transformer that extracts and flattens the richly structured output of
# Bing Image Search into a simple URL column
getUrls = BingImageSearch.getUrlTransformer("images", "url")

# This displays the full results returned, uncomment to use
# display(bingSearch.transform(bingParameters))

# Since we have two services, they are put into a pipeline
pipeline = PipelineModel(stages=[bingSearch, getUrls])

# Show the results of your search: image URLs
display(pipeline.transform(bingParameters))
```

Transform speech to text

The [Speech-to-text](#) service converts streams or files of spoken audio to text. The following code sample transcribes one audio file to text.

Python

```
# Create a dataframe with our audio URLs, tied to the column called "url"
df = spark.createDataFrame(
    [("https://mmlspark.blob.core.windows.net/datasets/Speech/audio2.wav",)],
    ["url"]
)
```

```

# Run the Speech-to-text service to translate the audio into text
speech_to_text = (
    SpeechToTextSDK()
    .setSubscriptionKey(service_key)
    .setLocation(service_loc)
    .setOutputCol("text")
    .setAudioDataCol("url")
    .setLanguage("en-US")
    .setProfanity("Masked")
)

# Show the results of the translation
display(speech_to_text.transform(df).select("url", "text.DisplayText"))

```

Transform text to speech

[Text to speech](#) is a service that allows you to build apps and services that speak naturally, choosing from more than 270 neural voices across 119 languages and variants.

The following code sample transforms text into an audio file that contains the content of the text.

Python

```

from synapse.ml.cognitive import TextToSpeech

fs = ""
if running_on_databricks():
    fs = "dbfs:"
elif running_on_synapse_internal():
    fs = "Files"

# Create a dataframe with text and an output file location
df = spark.createDataFrame(
    [
        (
            "Reading out loud is fun! Check out aka.ms/spark for more
information",
            fs + "/output.mp3",
        )
    ],
    ["text", "output_file"],
)

tts = (
    TextToSpeech()
    .setSubscriptionKey(service_key)
    .setTextCol("text")
    .setLocation(service_loc)
)

```

```
.setVoiceName("en-US-JennyNeural")
.setOutputFileCol("output_file")
)

# Check to make sure there were no errors during audio creation
display(tts.transform(df))
```

Detect anomalies in time series data

Anomaly Detector [↗](#) is great for detecting irregularities in your time series data. The following code sample uses the Anomaly Detector service to find anomalies in entire time series data.

Python

```
# Create a dataframe with the point data that Anomaly Detector requires
df = spark.createDataFrame(
    [
        ("1972-01-01T00:00:00Z", 826.0),
        ("1972-02-01T00:00:00Z", 799.0),
        ("1972-03-01T00:00:00Z", 890.0),
        ("1972-04-01T00:00:00Z", 900.0),
        ("1972-05-01T00:00:00Z", 766.0),
        ("1972-06-01T00:00:00Z", 805.0),
        ("1972-07-01T00:00:00Z", 821.0),
        ("1972-08-01T00:00:00Z", 20000.0),
        ("1972-09-01T00:00:00Z", 883.0),
        ("1972-10-01T00:00:00Z", 898.0),
        ("1972-11-01T00:00:00Z", 957.0),
        ("1972-12-01T00:00:00Z", 924.0),
        ("1973-01-01T00:00:00Z", 881.0),
        ("1973-02-01T00:00:00Z", 837.0),
        ("1973-03-01T00:00:00Z", 9000.0),
    ],
    ["timestamp", "value"],
).withColumn("group", lit("series1"))

# Run the Anomaly Detector service to look for irregular data
anomaly_detector = (
    SimpleDetectAnomalies()
    .setSubscriptionKey(anomaly_key)
    .setLocation(anomaly_loc)
    .setTimestampCol("timestamp")
    .setValueCol("value")
    .setOutputCol("anomalies")
    .setGroupbyCol("group")
    .setGranularity("monthly")
)

# Show the full results of the analysis with the anomalies marked as "True"
display(
```

```
    anamoly_detector.transform(df).select("timestamp", "value",
    "anomalies.isAnomaly")
)
```

Get information from arbitrary web APIs

With HTTP on Spark, you can use any web service in your big data pipeline. The following code sample uses the [World Bank API](#) to get information about various countries around the world.

Python

```
# Use any requests from the python requests library

def world_bank_request(country):
    return Request(
        "GET", "http://api.worldbank.org/v2/country/{}?
format=json".format(country)
    )

# Create a dataframe with specifies which countries we want data on
df = spark.createDataFrame([("br",), ("usa",)], ["country"]).withColumn(
    "request", http_udf(world_bank_request)(col("country")))
)

# Much faster for big data because of the concurrency :)
client = (
    HTTPTransformer().setConcurrency(3).setInputCol("request").setOutputCol("res
ponse")
)

# Get the body of the response

def get_response_body(resp):
    return resp.entity.content.decode()

# Show the details of the country data returned
display(
    client.transform(df).select(
        "country", udf(get_response_body)(col("response")).alias("response")
    )
)
```

Related content

- How to perform the same classification task with and without SynapseML
 - How to use KNN model with SynapseML
 - How to use ONNX with SynapseML - Deep Learning
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

Classification tasks using SynapseML

Article • 04/08/2025

This article shows how to perform the same classification task with two methods. One uses plain `pyspark`, and one uses the `synapseml` library. Although the methods yield the same performance, they highlight the simplicity of `synapseml` compared to `pyspark`.

The task predicts whether a specific customer review of book sold on Amazon is good (rating > 3) or bad, based on the review text. To build the task, you train LogisticRegression learners with different hyperparameters, and then choose the best model.

Prerequisites

Attach your notebook to a lakehouse. On the left side, select **Add** to add an existing lakehouse or create a lakehouse.

Setup

Import the necessary Python libraries and get a spark session.

Python

```
from pyspark.sql import SparkSession

# Bootstrap Spark Session
spark = SparkSession.builder.getOrCreate()
```

Read the data

Download and read in the data.

Python

```
rawData = spark.read.parquet(
    "wasbs://publicwasb@mmlspark.blob.core.windows.net/BookReviewsFromAmazon10K.parquet"
)
rawData.show(5)
```

Extract features and process data

Real data has more complexity compared to the dataset we downloaded earlier. A dataset often has features of multiple types - for example, text, numeric, and categorical. To show the difficulties of working with these datasets, add two numerical features to the dataset: the **word count** of the review and the **mean word length**:

Python

```
from pyspark.sql.functions import udf
from pyspark.sql.types import *

def wordCount(s):
    return len(s.split())

def wordLength(s):
    import numpy as np

    ss = [len(w) for w in s.split()]
    return round(float(np.mean(ss)), 2)

wordLengthUDF = udf(wordLength, DoubleType())
wordCountUDF = udf(wordCount, IntegerType())
```

Python

```
from synapse.ml.stages import UDFTransformer

wordLength = "wordLength"
wordCount = "wordCount"
wordLengthTransformer = UDFTransformer(
    inputCol="text", outputCol=wordLength, udf=wordLengthUDF
)
wordCountTransformer = UDFTransformer(
    inputCol="text", outputCol=wordCount, udf=wordCountUDF
)
```

Python

```
from pyspark.ml import Pipeline

data = (
    Pipeline(stages=[wordLengthTransformer, wordCountTransformer])
    .fit(rawData)
    .transform(rawData)
    .withColumn("label", rawData["rating"] > 3)
    .drop("rating")
)
```

```
Python
```

```
data.show(5)
```

Classify using pyspark

To choose the best LogisticRegression classifier using the `pyspark` library, you must *explicitly* perform these steps:

1. Process the features
 - Tokenize the text column
 - Hash the tokenized column into a vector using hashing
 - Merge the numeric features with the vector
2. To process the label column, cast that column into the proper type
3. Train multiple LogisticRegression algorithms on the `train` dataset with different hyperparameters
4. Compute the area under the ROC curve for each of the trained models, and select the model with the highest metric as computed on the `test` dataset
5. Evaluate the best model on the `validation` set

```
Python
```

```
from pyspark.ml.feature import Tokenizer, HashingTF
from pyspark.ml.feature import VectorAssembler

# Featurize text column
tokenizer = Tokenizer(inputCol="text", outputCol="tokenizedText")
numFeatures = 10000
hashingScheme = HashingTF(
    inputCol="tokenizedText", outputCol="TextFeatures", numFeatures=numFeatures
)
tokenizedData = tokenizer.transform(data)
featurizedData = hashingScheme.transform(tokenizedData)

# Merge text and numeric features in one feature column
featureColumnsArray = ["TextFeatures", "wordCount", "wordLength"]
assembler = VectorAssembler(inputCols=featureColumnsArray, outputCol="features")
assembledData = assembler.transform(featurizedData)

# Select only columns of interest
# Convert rating column from boolean to int
processedData = assembledData.select("label", "features").withColumn(
    "label", assembledData.label.cast(IntegerType())
)
```

Python

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.classification import LogisticRegression

# Prepare data for learning
train, test, validation = processedData.randomSplit([0.60, 0.20, 0.20], seed=123)

# Train the models on the 'train' data
lrHyperParams = [0.05, 0.1, 0.2, 0.4]
logisticRegressions = [
    LogisticRegression(regParam=hyperParam) for hyperParam in lrHyperParams
]
evaluator = BinaryClassificationEvaluator(
    rawPredictionCol="rawPrediction", metricName="areaUnderROC"
)
metrics = []
models = []

# Select the best model
for learner in logisticRegressions:
    model = learner.fit(train)
    models.append(model)
    scoredData = model.transform(test)
    metrics.append(evaluator.evaluate(scoredData))
bestMetric = max(metrics)
bestModel = models[metrics.index(bestMetric)]

# Get AUC on the validation dataset
scoredVal = bestModel.transform(validation)
print(evaluator.evaluate(scoredVal))
```

Classify using SynapseML

The `synapseml` option involves simpler steps:

1. The `TrainClassifier` Estimator featurizes the data internally, as long as the columns selected in the `train`, `test`, `validation` dataset represent the features
2. The `FindBestModel` Estimator finds the best model from a pool of trained models; to do this, it finds the model that performs best on the `test` dataset given the specified metric
3. The `ComputeModelStatistics` Transformer computes the different metrics on a scored dataset (in our case, the `validation` dataset) at the same time

Python

```
from synapse.ml.train import TrainClassifier, ComputeModelStatistics
from synapse.ml.automl import FindBestModel
```

```

# Prepare data for learning
train, test, validation = data.randomSplit([0.60, 0.20, 0.20], seed=123)

# Train the models on the 'train' data
lrHyperParams = [0.05, 0.1, 0.2, 0.4]
logisticRegressions = [
    LogisticRegression(regParam=hyperParam) for hyperParam in lrHyperParams
]
lrmodels = [
    TrainClassifier(model=lrm, labelCol="label", numFeatures=10000).fit(train)
    for lrm in logisticRegressions
]

# Select the best model
bestModel = FindBestModel(evaluationMetric="AUC", models=lrmodels).fit(test)

# Get AUC on the validation dataset
predictions = bestModel.transform(validation)
metrics = ComputeModelStatistics().transform(predictions)
print(
    "Best model's AUC on validation set = "
    + "{0:.2f}%".format(metrics.first()["AUC"] * 100)
)

```

Related content

- [How to use the k-NN \(K-Nearest-Neighbors\) model with SynapseML](#)
- [How to use ONNX with SynapseML - Deep Learning](#)
- [How to use Kernel SHAP to explain a tabular classification model](#)

Explore art across culture and mediums with the fast, conditional, k-nearest neighbors algorithm

Article • 04/04/2025

This article describes match-finding via the [k-nearest-neighbors algorithm](#). You build code resources that allow queries involving cultures and mediums of art amassed from the Metropolitan Museum of Art in NYC and the Amsterdam Rijksmuseum.

Prerequisites

- A notebook attached to a lakehouse. Visit [Explore the data in your lakehouse with a notebook](#) for more information.

Overview of the BallTree

The k-NN model relies on the [BallTree](#) data structure. The BallTree is a recursive binary tree, where each node (or "ball") contains a partition, or subset, of the data points you want to query. To build a BallTree, determine the "ball" center (based on a certain specified feature) closest to each data point. Then, assign each data point to that corresponding closest "ball." Those assignments create a structure that allows for binary tree-like traversals, and lends itself to finding k-nearest neighbors at a BallTree leaf.

Setup

Import the necessary Python libraries, and prepare the dataset:

Python

```
from synapse.ml.core.platform import *

if running_on_binder():
    from IPython import get_ipython
```

Python

```
from pyspark.sql.types import BooleanType
from pyspark.sql.types import *
from pyspark.ml.feature import Normalizer
```

```

from pyspark.sql.functions import lit, array, array_contains, udf, col,
struct
from synapse.ml.nn import ConditionalKNN, ConditionalKNNModel
from PIL import Image
from io import BytesIO

import requests
import numpy as np
import matplotlib.pyplot as plt
from pyspark.sql import SparkSession

# Bootstrap Spark Session
spark = SparkSession.builder.getOrCreate()

```

The dataset comes from a table that contains artwork information from both the Met Museum and the Rijksmuseum. The table has this schema:

- **ID**: A unique identifier for each specific piece of art
 - Sample Met ID: 388395
 - Sample Rijks ID: SK-A-2344
- **Title**: Art piece title, as written in the museum's database
- **Artist**: Art piece artist, as written in the museum's database
- **Thumbnail_Url**: Location of a JPEG thumbnail of the art piece
- **Image_Url** Website URL location of the art piece image, hosted on the Met/Rijks website
- **Culture**: Culture category of the art piece
 - Sample culture categories: *latin American, Egyptian*, etc.
- **Classification**: Medium category of the art piece
 - Sample medium categories: *woodwork, paintings*, etc.
- **Museum_Page**: URL link to the art piece, hosted on the Met/Rijks website
- **Norm_Features**: Embedding of the art piece image
- **Museum**: The museum hosting the actual art piece

Python

```

# loads the dataset and the two trained conditional k-NN models for querying
# by medium and culture
df = spark.read.parquet(
    "wasbs://publicwasb@mmlspark.blob.core.windows.net/met_and rijks.parquet"
)
display(df.drop("Norm_Features"))

```

To build the query, define the categories

Use two k-NN models: one for culture, and one for medium:

Python

```
# mediums = ['prints', 'drawings', 'ceramics', 'textiles', 'paintings',
"musical instruments", "glass", 'accessories', 'photographs', "metalwork",
#           "sculptures", "weapons", "stone", "precious", "paper",
"woodwork", "leatherwork", "uncategorized"]

mediums = ["paintings", "glass", "ceramics"]

# cultures = ['african (general)', 'american', 'ancient american', 'ancient
asian', 'ancient european', 'ancient middle-eastern', 'asian (general)',
#             'austrian', 'belgian', 'british', 'chinese', 'czech', 'dutch',
'egyptian']#, 'european (general)', 'french', 'german', 'greek',
#             'iranian', 'italian', 'japanese', 'latin american', 'middle
eastern', 'roman', 'russian', 'south asian', 'southeast asian',
#             'spanish', 'swiss', 'various']

cultures = ["japanese", "american", "african (general)"]

# Uncomment the above for more robust and large scale searches!

classes = cultures + mediums

medium_set = set(mediums)
culture_set = set(cultures)
selected_ids = {"AK-RBK-17525-2", "AK-MAK-1204", "AK-RAK-2015-2-9"}

small_df = df.where(
    udf(
        lambda medium, culture, id_val: (medium in medium_set)
        or (culture in culture_set)
        or (id_val in selected_ids),
        BooleanType(),
    )("Classification", "Culture", "id")
)

small_df.count()
```

Define and fit conditional k-NN models

Create conditional k-NN models for both the medium and culture columns. Each model takes

- an output column
- a features column (feature vector)
- a values column (cell values under the output column)
- a label column (the quality that the respective k-NN is conditioned on)

Python

```
medium_cknn = (
    ConditionalKNN()
    .setOutputCol("Matches")
    .setFeaturesCol("Norm_Features")
    .setValuesCol("Thumbnail_Url")
    .setLabelCol("Classification")
    .fit(small_df)
)
```

Python

```
culture_cknn = (
    ConditionalKNN()
    .setOutputCol("Matches")
    .setFeaturesCol("Norm_Features")
    .setValuesCol("Thumbnail_Url")
    .setLabelCol("Culture")
    .fit(small_df)
)
```

Define matching and visualizing methods

After the initial dataset and category setup, prepare the methods to query and visualize the results of the conditional k-NN:

`addMatches()` creates a Dataframe with a handful of matches per category:

Python

```
def add_matches(classes, cknn, df):
    results = df
    for label in classes:
        results = cknn.transform(
            results.withColumn("conditioner", array(lit(label)))
            .withColumnRenamed("Matches", "Matches_{}".format(label)))
    return results
```

`plot_urls()` calls `plot_img` to visualize top matches for each category into a grid:

Python

```
def plot_img(axis, url, title):
    try:
        response = requests.get(url)
        img = Image.open(BytesIO(response.content)).convert("RGB")
```

```

        axis.imshow(img, aspect="equal")
    except:
        pass
    if title is not None:
        axis.set_title(title, fontsize=4)
    axis.axis("off")

def plot_urls(url_arr, titles, filename):
    nx, ny = url_arr.shape

    plt.figure(figsize=(nx * 5, ny * 5), dpi=1600)
    fig, axes = plt.subplots(ny, nx)

    # reshape required in the case of 1 image query
    if len(axes.shape) == 1:
        axes = axes.reshape(1, -1)

    for i in range(nx):
        for j in range(ny):
            if j == 0:
                plot_img(axes[j, i], url_arr[i, j], titles[i])
            else:
                plot_img(axes[j, i], url_arr[i, j], None)

    plt.savefig(filename, dpi=1600) # saves the results as a PNG

    display(plt.show())

```

Put everything together

To take in

- the data
- the conditional k-NN models
- the art ID values to query on
- the file path where the output visualization is saved

define a function called `test_all()`

The medium and culture models were previously trained and loaded.

Python

```

# main method to test a particular dataset with two conditional k-NN models
# and a set of art IDs, saving the result to filename.png

def test_all(data, cknn_medium, cknn_culture, test_ids, root):
    is_nice_obj = udf(lambda obj: obj in test_ids, BooleanType())
    test_df = data.where(is_nice_obj("id"))

```

```

results_df_medium = add_matches(mediums, cknn_medium, test_df)
results_df_culture = add_matches(cultures, cknn_culture,
results_df_medium)

results = results_df_culture.collect()

original_urls = [row["Thumbnail_Url"] for row in results]

culture_urls = [
    [row["Matches_{}".format(label)][0]["value"] for row in results]
    for label in cultures
]
culture_url_arr = np.array([original_urls] + culture_urls)[:, :]
plot_urls(culture_url_arr, ["Original"] + cultures, root +
"matches_by_culture.png")

medium_urls = [
    [row["Matches_{}".format(label)][0]["value"] for row in results]
    for label in mediums
]
medium_url_arr = np.array([original_urls] + medium_urls)[:, :]
plot_urls(medium_url_arr, ["Original"] + mediums, root +
"matches_by_medium.png")

return results_df_culture

```

Demo

The following cell performs batched queries, given the desired image IDs and a filename to save the visualization.

Python

```
# sample query
result_df = test_all(small_df, medium_cknn, culture_cknn, selected_ids,
root=".")
```

Related content

- [How to use ONNX with SynapseML - Deep Learning](#)
- [How to use Kernel SHAP to explain a tabular classification model](#)
- [How to use SynapseML for multivariate anomaly detection](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

ONNX inference on Spark

Article • 04/08/2025

In this example, you train a LightGBM model, and convert that model to the [ONNX](#) format. Once converted, you use the model to infer some test data on Spark.

This example uses these Python packages and versions:

- `onnxmлltools==1.7.0`
- `lightgbm==3.2.1`

Prerequisites

- Attach your notebook to a lakehouse. On the left side, select **Add** to add an existing lakehouse or create a lakehouse.
- You might need to install `onnxmлltools`. To do this, add `!pip install onnxmлltools==1.7.0` in a notebook code cell, and then run that cell.
- You might need to install `lightgbm`. To do this, add `!pip install lightgbm==3.2.1` in a notebook code cell, and then run that cell.

Load the example data

To load the example data, add these code examples to cells in your notebook, and then run those cells:

Python

```
from pyspark.sql import SparkSession

# Bootstrap Spark Session
spark = SparkSession.builder.getOrCreate()

from synapse.ml.core.platform import *
```

Python

```
df = (
    spark.read.format("csv")
    .option("header", True)
    .option("inferSchema", True)
    .load(
        "wasbs://publicwasb@mmlspark.blob.core.windows.net/company_bankruptcy_prediction_data.csv"
    )
```

```
)  
)  
  
display(df)
```

The output should look similar to the following table. The specific columns shown, the number of rows, and the actual values in the table might differ:

[+] Expand table

Interest Coverage Ratio	Net Income Flag	Equity to Liability
0.5641	1.0	0.0165
0.5702	1.0	0.0208
0.5673	1.0	0.0165

Use LightGBM to train a model

Python

```
from pyspark.ml.feature import VectorAssembler  
from synapse.ml.lightgbm import LightGBMClassifier  
  
feature_cols = df.columns[1:]  
featurizer = VectorAssembler(inputCols=feature_cols, outputCol="features")  
  
train_data = featurizer.transform(df)[ "Bankrupt?", "features"]  
  
model = (  
    LightGBMClassifier(featuresCol="features", labelCol="Bankrupt?",  
    dataTransferMode="bulk")  
    .setEarlyStoppingRound(300)  
    .setLambdaL1(0.5)  
    .setNumIterations(1000)  
    .setNumThreads(-1)  
    .setMaxDeltaStep(0.5)  
    .setNumLeaves(31)  
    .setMaxDepth(-1)  
    .setBaggingFraction(0.7)  
    .setFeatureFraction(0.7)  
    .setBaggingFreq(2)  
    .setObjective("binary")  
    .setIsUnbalance(True)  
    .setMinSumHessianInLeaf(20)  
    .setMinGainToSplit(0.01)  
)
```

```
model = model.fit(train_data)
```

Convert the model to ONNX format

The following code exports the trained model to a LightGBM booster, and then converts the model to the ONNX format:

Python

```
import lightgbm as lgb
from lightgbm import Booster, LGBMClassifier

def convertModel(lgbm_model: LGBMClassifier or Booster, input_size: int) -> bytes:
    from onnxxmltools.convert import convert_lightgbm
    from onnxconverter_common.data_types import FloatTensorType

    initial_types = [("input", FloatTensorType([-1, input_size]))]
    onnx_model = convert_lightgbm(
        lgbm_model, initial_types=initial_types, target_opset=9
    )
    return onnx_model.SerializeToString()

booster_model_str = model.getLightGBMBooster().modelStr().get()
booster = lgb.Booster(model_str=booster_model_str)
model_payload_ml = convertModel(booster, len(feature_cols))
```

After conversion, load the ONNX payload into an `ONNXModel`, and inspect the model inputs and outputs:

Python

```
from synapse.ml.onnx import ONNXModel

onnx_ml = ONNXModel().setModelPayload(model_payload_ml)

print("Model inputs:" + str(onnx_ml.getModelInputs()))
print("Model outputs:" + str(onnx_ml.getModelOutputs()))
```

Map the model input to the column name (FeedDict) of the input dataframe, and map the column names of the output dataframe to the model outputs (FetchDict):

Python

```

onnx_ml = (
    onnx_ml.setDeviceType("CPU")
    .setFeedDict({"input": "features"})
    .setFetchDict({"probability": "probabilities", "prediction": "label"})
    .setMiniBatchSize(5000)
)

```

Use the model for inference

To perform inference with the model, the following code creates test data, and transforms the data through the ONNX model:

Python

```

from pyspark.ml.feature import VectorAssembler
import pandas as pd
import numpy as np

n = 1000 * 1000
m = 95
test = np.random.rand(n, m)
testPd = pd.DataFrame(test)
cols = list(map(str, testPd.columns))
testDf = spark.createDataFrame(testPd)
testDf = testDf.union(testDf).repartition(200)
testDf = (
    VectorAssembler()
    .setInputCols(cols)
    .setOutputCol("features")
    .transform(testDf)
    .drop(*cols)
    .cache()
)

display(onnx_ml.transform(testDf))

```

The output should look similar to the following table, though the values and number of rows might differ:

[] Expand table

Index	Features	Prediction	Probability
1	{"type":1,"values":[0.105...]	0	{"0":0.835...}
2	{"type":1,"values":[0.814...]	0	{"0":0.658...}

Related content

- [How to use Kernel SHAP to explain a tabular classification model](#)
- [How to use SynapseML for multivariate anomaly detection](#)
- [How to Build a Search Engine with SynapseML](#)

Interpretability - Tabular SHAP explainer

Article • 04/04/2025

This example uses Kernel SHAP to explain a tabular classification model built from the Adults Census dataset.

Import the required packages, and define the UDFs we need later:

Python

```
import pyspark
from synapse.ml.explainers import *
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
from pyspark.sql.types import *
from pyspark.sql.functions import *
import pandas as pd
from pyspark.sql import SparkSession

# Bootstrap Spark Session
spark = SparkSession.builder.getOrCreate()

from synapse.ml.core.platform import *

vec_access = udf(lambda v, i: float(v[i]), FloatType())
vec2array = udf(lambda vec: vec.toArray().tolist(), ArrayType(FloatType()))
```

Read the data, and train a binary classification model:

Python

```
df = spark.read.parquet(
    "wasbs://publicwasb@mmlspark.blob.core.windows.net/AdultCensusIncome.parquet"
)

labelIndexer = StringIndexer(
    inputCol="income", outputCol="label", stringOrderType="alphabetAsc"
).fit(df)
print("Label index assigment: " + str(set(zip(labelIndexer.labels, [0, 1]))))

training = labelIndexer.transform(df).cache()
display(training)
categorical_features = [
    "workclass",
    "education",
    "marital-status",
```

```

    "occupation",
    "relationship",
    "race",
    "sex",
    "native-country",
]
categorical_features_idx = [col + "_idx" for col in categorical_features]
categorical_features_enc = [col + "_enc" for col in categorical_features]
numeric_features = [
    "age",
    "education-num",
    "capital-gain",
    "capital-loss",
    "hours-per-week",
]
strIndexer = StringIndexer(
    inputCols=categorical_features, outputCols=categorical_features_idx
)
onehotEnc = OneHotEncoder(
    inputCols=categorical_features_idx, outputCols=categorical_features_enc
)
vectAssem = VectorAssembler(
    inputCols=categorical_features_enc + numeric_features,
    outputCol="features"
)
lr = LogisticRegression(featuresCol="features", labelCol="label",
weightCol="fnlwgt")
pipeline = Pipeline(stages=[strIndexer, onehotEnc, vectAssem, lr])
model = pipeline.fit(training)

```

After the model is trained, randomly select some observations to explain:

Python

```

explain_instances = (
    model.transform(training).orderBy(rand()).limit(5).repartition(200).cache()
)
display(explain_instances)

```

Create a TabularSHAP explainer, set the input columns to all the features the model takes. Next, specify the model and the target output column we want to explain. Here, we want to explain the "probability" output, which is a vector of length 2, and we only look at class 1 probability. Specify targetClasses of `[0, 1]` to explain class 0 and 1 probability at the same time. Finally, sample 100 rows from the training data for background data, which is used for integrating out features in Kernel SHAP:

Python

```

shap = TabularSHAP(
    inputCols=categorical_features + numeric_features,
    outputCol="shapValues",
    numSamples=5000,
    model=model,
    targetCol="probability",
    targetClasses=[1],
    backgroundData=broadcast(training.orderBy(rand()).limit(100).cache()),
)
shap_df = shap.transform(explain_instances)

```

With the resulting dataframe, extract

- the class 1 probability of the model output
- the SHAP values for the target class
- the original features
- the true label

Then, convert the dataframe to a pandas dataframe for visualization.

For each observation, the first element in the SHAP values vector is the base value (the mean output of the background dataset). Each of the following element is the SHAP values for each feature:

Python

```

shaps = (
    shap_df.withColumn("probability", vec_access(col("probability"),
lit(1)))
    .withColumn("shapValues", vec2array(col("shapValues").getItem(0)))
    .select(
        ["shapValues", "probability", "label"] + categorical_features +
numeric_features
    )
)

shaps_local = shaps.toPandas()
shaps_local.sort_values("probability", ascending=False, inplace=True,
ignore_index=True)
pd.set_option("display.max_colwidth", None)
shaps_local

```

Use plotly subplot to visualize the SHAP values:

Python

```

from plotly.subplots import make_subplots
import plotly.graph_objects as go
import pandas as pd

features = categorical_features + numeric_features
features_with_base = ["Base"] + features

rows = shaps_local.shape[0]

fig = make_subplots(
    rows=rows,
    cols=1,
    subplot_titles="Probability: "
    + shaps_local["probability"].apply("{:.2%}".format)
    + "; Label: "
    + shaps_local["label"].astype(str),
)

for index, row in shaps_local.iterrows():
    feature_values = [0] + [row[feature] for feature in features]
    shap_values = row["shapValues"]
    list_of_tuples = list(zip(features_with_base, feature_values,
    shap_values))
    shap_pdf = pd.DataFrame(list_of_tuples, columns=["name", "value",
    "shap"])
    fig.add_trace(
        go.Bar(
            x=shap_pdf["name"],
            y=shap_pdf["shap"],
            hovertext="value: " + shap_pdf["value"].astype(str),
        ),
        row=index + 1,
        col=1,
    )

fig.update_yaxes(range=[-1, 1], fixedrange=True, zerolinecolor="black")
fig.update_xaxes(type="category", tickangle=45, fixedrange=True)
fig.update_layout(height=400 * rows, title_text="SHAP explanations")
fig.show()

```

Related content

- [How to use SynapseML for multivariate anomaly detection](#)
- [How to Build a Search Engine with SynapseML](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Multivariate Anomaly Detection with Isolation Forest

Article • 01/19/2024

This article shows how you can use SynapseML on Apache Spark for multivariate anomaly detection. Multivariate anomaly detection allows for the detection of anomalies among many variables or timeseries, taking into account all the inter-correlations and dependencies between the different variables. In this scenario, we use SynapseML to train an Isolation Forest model for multivariate anomaly detection, and we then use to the trained model to infer multivariate anomalies within a dataset containing synthetic measurements from three IoT sensors.

To learn more about the Isolation Forest model, refer to the original paper by [Liu et al.](#).

Prerequisites

- Attach your notebook to a lakehouse. On the left side, select **Add** to add an existing lakehouse or create a lakehouse.

Library imports

Python

```
from IPython import get_ipython
from IPython.terminal.interactiveshell import TerminalInteractiveShell
import uuid
import mlflow

from pyspark.sql import functions as F
from pyspark.ml.feature import VectorAssembler
from pyspark.sql.types import *
from pyspark.ml import Pipeline

from synapse.ml.isolationforest import *
from synapse.ml.explainers import *
```

Python

```
%matplotlib inline
```

Python

```
from pyspark.sql import SparkSession

# Bootstrap Spark Session
spark = SparkSession.builder.getOrCreate()

from synapse.ml.core.platform import *

if running_on_synapse():
    shell = TerminalInteractiveShell.instance()
    shell.define_macro("foo", """a,b=10,20""")
```

Input data

Python

```
# Table inputs
timestampColumn = "timestamp" # str: the name of the timestamp column in
the table
inputCols = [
    "sensor_1",
    "sensor_2",
    "sensor_3",
] # list(str): the names of the input variables

# Training Start time, and number of days to use for training:
trainingStartTime = (
    "2022-02-24T06:00:00Z" # datetime: datetime for when to start the
training
)
trainingEndTime = (
    "2022-03-08T23:55:00Z" # datetime: datetime for when to end the
training
)
inferenceStartTime = (
    "2022-03-09T09:30:00Z" # datetime: datetime for when to start the
training
)
inferenceEndTime = (
    "2022-03-20T23:55:00Z" # datetime: datetime for when to end the
training
)

# Isolation Forest parameters
contamination = 0.021
num_estimators = 100
max_samples = 256
```

```
max_features = 1.0
```

Read data

Python

```
df = (
    spark.read.format("csv")
    .option("header", "true")
    .load(
        "wasbs://publicwasb@mmlspark.blob.core.windows.net/generated_sample_mvad_dat
        a.csv"
    )
)
```

cast columns to appropriate data types

Python

```
df = (
    df.orderBy(timestampColumn)
    .withColumn("timestamp", F.date_format(timestampColumn, "yyyy-MM-
    dd'T'HH:mm:ss'Z'"))
    .withColumn("sensor_1", F.col("sensor_1").cast(DoubleType()))
    .withColumn("sensor_2", F.col("sensor_2").cast(DoubleType()))
    .withColumn("sensor_3", F.col("sensor_3").cast(DoubleType()))
    .drop("_c5")
)

display(df)
```

Training data preparation

Python

```
# filter to data with timestamps within the training window
df_train = df.filter(
    (F.col(timestampColumn) >= trainingStartTime)
    & (F.col(timestampColumn) <= trainingEndTime)
)
display(df_train)
```

Test data preparation

Python

```
# filter to data with timestamps within the inference window
df_test = df.filter(
    (F.col(timestampColumn) >= inferenceStartTime)
    & (F.col(timestampColumn) <= inferenceEndTime)
)
display(df_test)
```

Train Isolation Forest model

Python

```
isolationForest = (
    IsolationForest()
    .setNumEstimators(num_estimators)
    .setBootstrap(False)
    .setMaxSamples(max_samples)
    .setMaxFeatures(max_features)
    .setFeaturesCol("features")
    .setPredictionCol("predictedLabel")
    .setScoreCol("outlierScore")
    .setContamination(contamination)
    .setContaminationError(0.01 * contamination)
    .setRandomSeed(1)
)
```

Next, we create an ML pipeline to train the Isolation Forest model. We also demonstrate how to create an MLflow experiment and register the trained model.

MLflow model registration is strictly only required if accessing the trained model at a later time. For training the model, and performing inferencing in the same notebook, the model object model is sufficient.

Python

```
va = VectorAssembler(inputCols=inputCols, outputCol="features")
pipeline = Pipeline(stages=[va, isolationForest])
model = pipeline.fit(df_train)
```

Perform inferencing

Load the trained Isolation Forest Model

Perform inferencing

Python

```
df_test_pred = model.transform(df_test)
display(df_test_pred)
```

Premade Anomaly Detector

[Azure AI Anomaly Detector ↗](#)

- Anomaly status of latest point: generates a model using preceding points and determines whether the latest point is anomalous ([Scala ↗](#), [Python ↗](#))
- Find anomalies: generates a model using an entire series and finds anomalies in the series ([Scala ↗](#), [Python ↗](#))

Next steps

- [How to Build a Search Engine with SynapseML](#)
- [How to use SynapseML and Azure AI services for multivariate anomaly detection - Analyze time series](#)
- [How to use SynapseML to tune hyperparameters](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Tutorial: Create a custom search engine and question-answering system

Article • 08/31/2023

In this tutorial, learn how to index and query large data loaded from a Spark cluster. You set up a Jupyter Notebook that performs the following actions:

- Load various forms (invoices) into a data frame in an Apache Spark session
- Analyze them to determine their features
- Assemble the resulting output into a tabular data structure
- Write the output to a search index hosted in Azure Cognitive Search
- Explore and query over the content you created

1 - Set up dependencies

We start by importing packages and connecting to the Azure resources used in this workflow.

Python

```
import os
from pyspark.sql import SparkSession
from synapse.ml.core.platform import running_on_synapse, find_secret

# Bootstrap Spark Session
spark = SparkSession.builder.getOrCreate()

cognitive_key = find_secret("cognitive-api-key") # replace with your
cognitive api key
cognitive_location = "eastus"

translator_key = find_secret("translator-key") # replace with your cognitive
api key
translator_location = "eastus"

search_key = find_secret("azure-search-key") # replace with your cognitive
api key
search_service = "mmlspark-azure-search"
search_index = "form-demo-index-5"

openai_key = find_secret("openai-api-key") # replace with your open ai api
key
openai_service_name = "synapseml-openai"
openai_deployment_name = "gpt-35-turbo"
openai_url = f"https://{{openai_service_name}}.openai.azure.com/"
```

2 - Load data into Spark

This code loads a few external files from an Azure storage account that's used for demo purposes. The files are various invoices, and they're read into a data frame.

Python

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

def blob_to_url(blob):
    [prefix, postfix] = blob.split("@")
    container = prefix.split("/")[-1]
    split_postfix = postfix.split("/")
    account = split_postfix[0]
    filepath = "/".join(split_postfix[1:])
    return "https://{}.{}/{}".format(account, container, filepath)

df2 = (
    spark.read.format("binaryFile")

    .load("wasbs://ignite2021@mmlsparkdemo.blob.core.windows.net/form_subset/*")
    .select("path")
    .limit(10)
    .select(udf(blob_to_url, StringType())("path").alias("url"))
    .cache()
)
display(df2)
```

3 - Apply form recognition

This code loads the [AnalyzeInvoices transformer](#) and passes a reference to the data frame containing the invoices. It calls the pre-built invoice model of Azure Forms Analyzer.

Python

```
from synapse.ml.cognitive import AnalyzeInvoices

analyzed_df = (
    AnalyzeInvoices()
    .setSubscriptionKey(cognitive_key)
    .setLocation(cognitive_location)
    .setImageUrlCol("url")
    .setOutputCol("invoices")
    .setErrorCol("errors")
```

```
.setConcurrency(5)
.transform(df2)
.cache()
)

display(analyzed_df)
```

4 - Simplify form recognition output

This code uses the [FormOntologyLearner](#), a transformer that analyzes the output of Form Recognizer transformers and infers a tabular data structure. The output of AnalyzeInvoices is dynamic and varies based on the features detected in your content.

FormOntologyLearner extends the utility of the AnalyzeInvoices transformer by looking for patterns that can be used to create a tabular data structure. Organizing the output into multiple columns and rows makes for simpler downstream analysis.

Python

```
from synapse.ml.cognitive import FormOntologyLearner

organized_df = (
    FormOntologyLearner()
    .setInputCol("invoices")
    .setOutputCol("extracted")
    .fit(analyzed_df)
    .transform(analyzed_df)
    .select("url", "extracted.*")
    .cache()
)

display(organized_df)
```

With our nice tabular dataframe, we can flatten the nested tables found in the forms with some SparkSQL

Python

```
from pyspark.sql.functions import explode, col

itemized_df = (
    organized_df.select("*", explode(col("Items")).alias("Item"))
    .drop("Items")
    .select("Item.*", "*")
    .drop("Item")
)
```

```
display(itemized_df)
```

5 - Add translations

This code loads [Translate](#), a transformer that calls the Azure AI Translator service in Azure AI services. The original text, which is in English in the "Description" column, is machine-translated into various languages. All of the output is consolidated into "output.translations" array.

Python

```
from synapse.ml.cognitive import Translate

translated_df = (
    Translate()
    .setSubscriptionKey(translator_key)
    .setLocation(translator_location)
    .setTextCol("Description")
    .setErrorCol("TranslationError")
    .setOutputCol("output")
    .setToLanguage(["zh-Hans", "fr", "ru", "cy"])
    .setConcurrency(5)
    .transform(itemized_df)
    .withColumn("Translations", col("output.translations")[0])
    .drop("output", "TranslationError")
    .cache()
)

display(translated_df)
```

6 - Translate products to emojis with OpenAI □

Python

```
from synapse.ml.cognitive.openai import OpenAIPrompt
from pyspark.sql.functions import trim, split

emoji_template = """
    Your job is to translate item names into emoji. Do not add anything but
    the emoji and end the translation with a comma

    Two Ducks: 🦆🦆,
    Light Bulb: 💡,
    Three Peaches: 🍑🍑🍑,
    Two kitchen stoves: 🍲🍲,
    A red car: 🚗,
```

```

A person and a cat: 🐱,
A {Description}: """

prompter = (
    OpenAIPrompt()
    .setSubscriptionKey(openai_key)
    .setDeploymentName(openai_deployment_name)
    .setUrl(openai_url)
    .setMaxTokens(5)
    .setPromptTemplate(emoji_template)
    .setErrorCol("error")
    .setOutputCol("Emoji")
)
emoji_df = (
    prompter.transform(translated_df)
    .withColumn("Emoji", trim(split(col("Emoji"), ",").getItem(0)))
    .drop("error", "prompt")
    .cache()
)

```

Python

```
display(emoji_df.select("Description", "Emoji"))
```

7 - Infer vendor address continent with OpenAI

Python

```

continent_template = """
Which continent does the following address belong to?

Pick one value from Europe, Australia, North America, South America, Asia,
Africa, Antarctica.

Dont respond with anything but one of the above. If you don't know the
answer or cannot figure it out from the text, return None. End your answer
with a comma.

```

```

Address: "6693 Ryan Rd, North Whales",
Continent: Europe,
Address: "6693 Ryan Rd",
Continent: None,
Address: "{VendorAddress}",
Continent:""""

```

```

continent_df = (
    prompter.setOutputCol("Continent")
    .setPromptTemplate(continent_template)
    .transform(emoji_df)
)
```

```
.withColumn("Continent", trim(split(col("Continent"), ",").getItem(0)))
.drop("error", "prompt")
.cache()
)
```

Python

```
display(continent_df.select("VendorAddress", "Continent"))
```

8 - Create an Azure Search Index for the Forms

Python

```
from synapse.ml.cognitive import *
from pyspark.sql.functions import monotonically_increasing_id, lit

(
    continent_df.withColumn("DocID",
monotonically_increasing_id().cast("string"))
    .withColumn("SearchAction", lit("upload"))
    .writeToAzureSearch(
        subscriptionKey=search_key,
        actionCol="SearchAction",
        serviceName=search_service,
        indexName=search_index,
        keyCol="DocID",
    )
)
```

9 - Try out a search query

Python

```
import requests

search_url = "https://{}.search.windows.net/indexes/{}/docs/search?api-
version=2019-05-06".format(
    search_service, search_index
)
requests.post(
    search_url, json={"search": "door"}, headers={"api-key": search_key}
).json()
```

10 - Build a chatbot that can use Azure Search as a tool

Python

```
import json
import openai

openai.api_type = "azure"
openai.api_base = openai_url
openai.api_key = openai_key
openai.api_version = "2023-03-15-preview"

chat_context_prompt = f"""
You are a chatbot designed to answer questions with the help of a search
engine that has the following information:

{continent_df.columns}

If you dont know the answer to a question say "I dont know". Do not lie or
hallucinate information. Be brief. If you need to use the search engine to
solve the please output a json in the form of {"query": "example_query"}
"""

def search_query_prompt(question):
    return f"""
Given the search engine above, what would you search for to answer the
following question?

Question: "{question}"

Please output a json in the form of {"query": "example_query"}
"""

def search_result_prompt(query):
    search_results = requests.post(
        search_url, json={"search": query}, headers={"api-key": search_key}
    ).json()
    return f"""

You previously ran a search for "{query}" which returned the following
results:

{search_results}

You should use the results to help you answer questions. If you dont know
the answer to a question say "I dont know". Do not lie or hallucinate
information. Be Brief and mention which query you used to solve the problem.
"""
```

```

def prompt_gpt(messages):
    response = openai.ChatCompletion.create(
        engine=openai_deployment_name, messages=messages, max_tokens=None,
        top_p=0.95
    )
    return response["choices"][0]["message"]["content"]

def custom_chatbot(question):
    while True:
        try:
            query = json.loads(
                prompt_gpt(
                    [
                        {"role": "system", "content": chat_context_prompt},
                        {"role": "user", "content": search_query_prompt(question)},
                    ]
                )
            )["query"]

            return prompt_gpt(
                [
                    {"role": "system", "content": chat_context_prompt},
                    {"role": "system", "content": search_result_prompt(query)},
                    {"role": "user", "content": question},
                ]
            )
        except Exception as e:
            raise e

```

11 - Asking our chatbot a question

Python

```
custom_chatbot("What did Luke Diaz buy?")
```

12 - A quick double check

Python

```

display(
    continent_df.where(col("CustomerName") == "Luke Diaz")
    .select("Description")
    .distinct()
)

```

Next steps

- How to use LightGBM with SynapseML
- How to use SynapseML and Azure AI services for multivariate anomaly detection - Analyze time series
- How to use SynapseML to tune hyperparameters

Recipe: Azure AI services - Multivariate Anomaly Detection

Article • 04/10/2025

This recipe shows how to use SynapseML and Azure AI services, on Apache Spark, for multivariate anomaly detection. Multivariate anomaly detection involves detection of anomalies among many variables or time series, while accounting for all the inter-correlations and dependencies between the different variables. This scenario uses SynapseML and the Azure AI services to train a model for multivariate anomaly detection. We then use the model to infer multivariate anomalies within a dataset that contains synthetic measurements from three IoT sensors.

ⓘ Important

Starting September, 20, 2023, you can't create new Anomaly Detector resources. The Anomaly Detector service will be retired on October 1, 2026.

For more information about the Azure AI Anomaly Detector, visit the [Anomaly Detector](#) information resource.

Prerequisites

- An Azure subscription - [Create one for free ↗](#)
- Attach your notebook to a lakehouse. On the left side, select **Add** to add an existing lakehouse or create a lakehouse.

Setup

Starting with an existing [Anomaly Detector](#) resource, you can explore ways to handle data of various forms. The catalog of services within Azure AI provides several options:

- [Decision ↗](#)
- [Document Intelligence ↗](#)
- [Language ↗](#)
- [Speech ↗](#)
- [Translation ↗](#)
- [Vision ↗](#)
- [Web search ↗](#)

Create an Anomaly Detector resource

- In the Azure portal, select **Create** in your resource group, and then type **Anomaly Detector**. Select the Anomaly Detector resource.
- Name the resource, and ideally use the same region as the rest of your resource group. Use the default options for the rest, and then select **Review + Create** and then **Create**.
- After you create the Anomaly Detector resource, open it, and select the **Keys and Endpoints** panel in the left nav. Copy the key for the Anomaly Detector resource into the **ANOMALY_API_KEY** environment variable, or store it in the **anomalyKey** variable.

Create a Storage Account resource

To save intermediate data, you must create an Azure Blob Storage Account. Within that storage account, create a container for storing the intermediate data. Make note of the container name, and copy the connection string to that container. You need it to later populate the **containerName** variable and the **BLOB_CONNECTION_STRING** environment variable.

Enter your service keys

First, set up the environment variables for our service keys. The next cell sets the **ANOMALY_API_KEY** and the **BLOB_CONNECTION_STRING** environment variables, based on the values stored in our Azure Key Vault. If you run this tutorial in your own environment, be sure to set these environment variables before you proceed:

Python

```
import os
from pyspark.sql import SparkSession
from synapse.ml.core.platform import find_secret

# Bootstrap Spark Session
spark = SparkSession.builder.getOrCreate()
```

Read the **ANOMALY_API_KEY** and **BLOB_CONNECTION_STRING** environment variables, and set the **containerName** and **location** variables:

Python

```
# An Anomaly Dector subscription key
anomalyKey = find_secret("anomaly-api-key") # use your own anomaly api key
# Your storage account name
storageName = "anomalydetectiontest" # use your own storage account name
# A connection string to your blob storage account
storageKey = find_secret("madtest-storage-key") # use your own storage key
```

```
# A place to save intermediate MVAD results
intermediateSaveDir = (
    "wasbs://madtest@anomalydetectiontest.blob.core.windows.net/intermediateData"
)
# The location of the anomaly detector resource that you created
location = "westus2"
```

Connect to our storage account, so that the anomaly detector can save intermediate results in that storage account:

Python

```
spark.sparkContext._jsc.hadoopConfiguration().set(
    f"fs.azure.account.key.{storageName}.blob.core.windows.net", storageKey
)
```

Import all the necessary modules:

Python

```
import numpy as np
import pandas as pd

import pyspark
from pyspark.sql.functions import col
from pyspark.sql.functions import lit
from pyspark.sql.types import DoubleType
import matplotlib.pyplot as plt

import synapse.ml
from synapse.ml.cognitive import *
```

Read the sample data into a Spark DataFrame:

Python

```
df = (
    spark.read.format("csv")
    .option("header", "true")
    .load("wasbs://publicwasb@mmlspark.blob.core.windows.net/MVAD/sample.csv")
)

df = (
    df.withColumn("sensor_1", col("sensor_1").cast(DoubleType()))
    .withColumn("sensor_2", col("sensor_2").cast(DoubleType()))
    .withColumn("sensor_3", col("sensor_3").cast(DoubleType()))
)
```

```
# Let's inspect the dataframe:  
df.show(5)
```

We can now create an `estimator` object, which we use to train our model. We specify the start and end times for the training data. We also specify the input columns to use, and the name of the column that contains the timestamps. Finally, we specify the number of data points to use in the anomaly detection sliding window, and we set the connection string to the Azure Blob Storage Account:

Python

```
trainingStartTime = "2020-06-01T12:00:00Z"  
trainingEndTime = "2020-07-02T17:55:00Z"  
timestampColumn = "timestamp"  
inputColumns = ["sensor_1", "sensor_2", "sensor_3"]  
  
estimator = (  
    FitMultivariateAnomaly()  
    .setSubscriptionKey(anomalyKey)  
    .setLocation(location)  
    .setStartTime(trainingStartTime)  
    .setEndTime(trainingEndTime)  
    .setIntermediateSaveDir(intermediateSaveDir)  
    .setTimestampCol(timestampColumn)  
    .setInputCols(inputColumns)  
    .setSlidingWindow(200)  
)
```

Let's fit the `estimator` to the data:

Python

```
model = estimator.fit(df)
```

Once the training is done, we can use the model for inference. The code in the next cell specifies the start and end times for the data in which we'd like to detect the anomalies:

Python

```
inferenceStartTime = "2020-07-02T18:00:00Z"  
inferenceEndTime = "2020-07-06T05:15:00Z"  
  
result = (  
    model.setStartTime(inferenceStartTime)  
    .setEndTime(inferenceEndTime)  
    .setOutputCol("results")  
    .setErrorCol("errors")  
    .setInputCols(inputColumns)
```

```
.setTimestampCol(timestampColumn)
.transform(df)
)

result.show(5)
```

In the previous cell, `.show(5)` showed us the first five dataframe rows. The results were all `null` because they landed outside the inference window.

To show the results only for the inferred data, select the needed columns. We can then order the rows in the dataframe by ascending order, and filter the result to show only the rows in the inference window range. Here, `inferenceEndTime` matches the last row in the dataframe, so can ignore it.

Finally, to better plot the results, convert the Spark dataframe to a Pandas dataframe:

Python

```
rdf = (
    result.select(
        "timestamp",
        *inputColumns,
        "results.contributors",
        "results.isAnomaly",
        "results.severity"
    )
    .orderBy("timestamp", ascending=True)
    .filter(col("timestamp") >= lit(inferenceStartTime))
    .toPandas()
)

rdf
```

Format the `contributors` column that stores the contribution score from each sensor to the detected anomalies. The next cell handles this, and splits the contribution score of each sensor into its own column:

Python

```
def parse(x):
    if type(x) is list:
        return dict([item[::-1] for item in x])
    else:
        return {"series_0": 0, "series_1": 0, "series_2": 0}

rdf["contributors"] = rdf["contributors"].apply(parse)
rdf = pd.concat([
    rdf.drop(["contributors"], axis=1), pd.json_normalize(rdf["contributors"])],
axis=1)
```

```
)  
rdf
```

We now have the contribution scores of sensors 1, 2, and 3 in the `series_0`, `series_1`, and `series_2` columns respectively.

To plot the results, run the next cell. The `minSeverity` parameter specifies the minimum severity of the anomalies to plot:

Python

```
minSeverity = 0.1

##### Main Figure #####
plt.figure(figsize=(23, 8))
plt.plot(
    rdf[ "timestamp"],
    rdf[ "sensor_1"],
    color="tab:orange",
    linestyle="solid",
    linewidth=2,
    label="sensor_1",
)
plt.plot(
    rdf[ "timestamp"],
    rdf[ "sensor_2"],
    color="tab:green",
    linestyle="solid",
    linewidth=2,
    label="sensor_2",
)
plt.plot(
    rdf[ "timestamp"],
    rdf[ "sensor_3"],
    color="tab:blue",
    linestyle="solid",
    linewidth=2,
    label="sensor_3",
)
plt.grid(axis="y")
plt.tick_params(axis="x", which="both", bottom=False, labelbottom=False)
plt.legend()

anoms = list(rdf[ "severity"] >= minSeverity)
_, _, ymin, ymax = plt.axis()
plt.vlines(np.where(anoms), ymin=ymin, ymax=ymax, color="r", alpha=0.8)

plt.legend()
plt.title(
    "A plot of the values from the three sensors with the detected anomalies
highlighted in red."
)
```

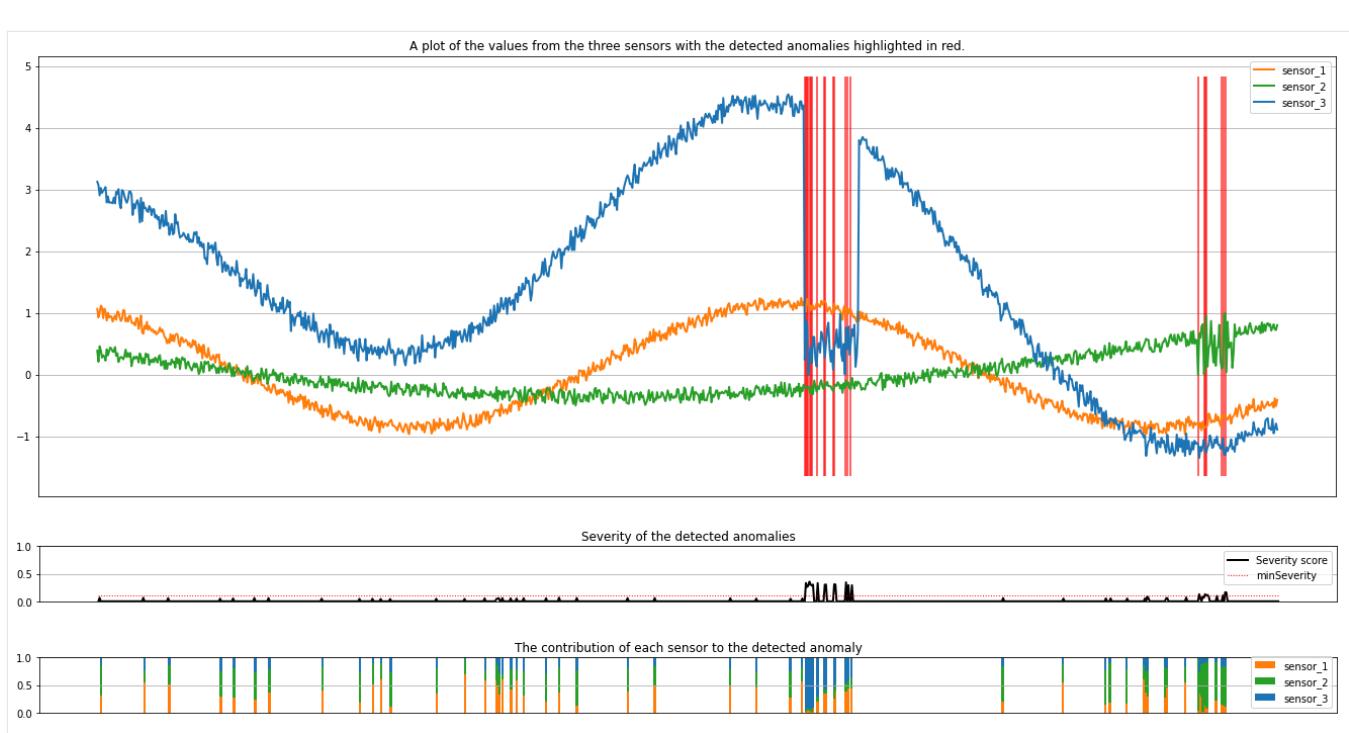
```

plt.show()

##### Severity Figure #####
plt.figure(figsize=(23, 1))
plt.tick_params(axis="x", which="both", bottom=False, labelbottom=False)
plt.plot(
    rdf[ "timestamp"],
    rdf[ "severity"],
    color="black",
    linestyle="solid",
    linewidth=2,
    label="Severity score",
)
plt.plot(
    rdf[ "timestamp"],
    [minSeverity] * len(rdf[ "severity"]),
    color="red",
    linestyle="dotted",
    linewidth=1,
    label="minSeverity",
)
plt.grid(axis="y")
plt.legend()
plt.ylim([0, 1])
plt.title("Severity of the detected anomalies")
plt.show()

##### Contributors Figure #####
plt.figure(figsize=(23, 1))
plt.tick_params(axis="x", which="both", bottom=False, labelbottom=False)
plt.bar(
    rdf[ "timestamp"], rdf[ "series_0"], width=2, color="tab:orange",
label="sensor_1"
)
plt.bar(
    rdf[ "timestamp"],
    rdf[ "series_1"],
    width=2,
    color="tab:green",
    label="sensor_2",
    bottom=rdf[ "series_0"],
)
plt.bar(
    rdf[ "timestamp"],
    rdf[ "series_2"],
    width=2,
    color="tab:blue",
    label="sensor_3",
    bottom=rdf[ "series_0"] + rdf[ "series_1"],
)
plt.grid(axis="y")
plt.legend()
plt.ylim([0, 1])
plt.title("The contribution of each sensor to the detected anomaly")
plt.show()

```



The plots show the raw data from the sensors (inside the inference window) in orange, green, and blue. The red vertical lines in the first figure show the detected anomalies that have a severity greater than or equal to `minSeverity`.

The second plot shows the severity score of all the detected anomalies, with the `minSeverity` threshold shown in the dotted red line.

Finally, the last plot shows the contribution of the data from each sensor to the detected anomalies. It helps us diagnose and understand the most likely cause of each anomaly.

Related content

- [How to use LightGBM with SynapseML](#)
- [How to use AI services with SynapseML](#)
- [How to use SynapseML to tune hyperparameters](#)

HyperParameterTuning - fighting breast cancer

Article • 04/04/2025

This tutorial shows how to use SynapseML to identify the best combination of hyperparameters for chosen classifiers, to build more accurate and reliable models. The tutorial shows how to perform distributed randomized grid search hyperparameter tuning to build a model that identifies breast cancer.

Set up the dependencies

Import pandas and set up a Spark session:

Python

```
import pandas as pd
from pyspark.sql import SparkSession

# Bootstrap Spark Session
spark = SparkSession.builder.getOrCreate()
```

Read the data, and split it into tuning and test sets:

Python

```
data = spark.read.parquet(
    "wasbs://publicwasb@mmlspark.blob.core.windows.net/BreastCancer.parquet"
).cache()
tune, test = data.randomSplit([0.80, 0.20])
tune.limit(10).toPandas()
```

Define the models to use:

Python

```
from synapse.ml.automl import TuneHyperparameters
from synapse.ml.train import TrainClassifier
from pyspark.ml.classification import (
    LogisticRegression,
    RandomForestClassifier,
    GBTClassifier,
)
logReg = LogisticRegression()
```

```
randForest = RandomForestClassifier()
gbt = GBTClassifier()
smlmodels = [logReg, randForest, gbt]
mmlmodels = [TrainClassifier(model=model, labelCol="Label") for model in
smlmodels]
```

Use AutoML to find the best model

Import the SynapseML AutoML classes from `synapse.ml.automl`. Specify the hyperparameters with `HyperparamBuilder`. Add either `DiscreteHyperParam` or `RangeHyperParam` hyperparameters. `TuneHyperparameters` randomly chooses values from a uniform distribution:

Python

```
from synapse.ml.automl import *

paramBuilder = (
    HyperparamBuilder()
    .addHyperparam(logReg, logReg.regParam, RangeHyperParam(0.1, 0.3))
    .addHyperparam(randForest, randForest.numTrees, DiscreteHyperParam([5,
10]))
    .addHyperparam(randForest, randForest.maxDepth, DiscreteHyperParam([3,
5]))
    .addHyperparam(gbt, gbt.maxBins, RangeHyperParam(8, 16))
    .addHyperparam(gbt, gbt.maxDepth, DiscreteHyperParam([3, 5]))
)
searchSpace = paramBuilder.build()
# The search space is a list of params to tuples of estimator and hyperparam
print(searchSpace)
randomSpace = RandomSpace(searchSpace)
```

Run `TuneHyperparameters` to get the best model:

Python

```
bestModel = TuneHyperparameters(
    evaluationMetric="accuracy",
    models=mmlmodels,
    numFolds=2,
    numRuns=len(mmlmodels) * 2,
    parallelism=1,
    paramSpace=randomSpace.space(),
    seed=0,
).fit(tune)
```

Evaluate the model

View the parameters of the best model, and retrieve the underlying best model pipeline:

Python

```
print(bestModel.getBestModelInfo())
print(bestModel.getBestModel())
```

Score against the test set, and view the metrics:

Python

```
from synapse.ml.train import ComputeModelStatistics

prediction = bestModel.transform(test)
metrics = ComputeModelStatistics().transform(prediction)
metrics.limit(10).toPandas()
```

Related content

- [How to use LightGBM with SynapseML](#)
- [How to use Azure AI services with SynapseML](#)
- [How to perform the same classification task with and without SynapseML](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

Getting started

Article • 09/26/2023

Semantic Link is a feature that allows you to establish a connection between [Power BI datasets](#) and [Synapse Data Science in Microsoft Fabric](#).

The primary goals of Semantic Link are to facilitate data connectivity, enable the propagation of semantic information, and seamlessly integrate with established tools used by data scientists, such as notebooks.

Semantic Link helps you to preserve domain knowledge about data semantics in a standardized way that can speed up data analysis and reduce errors.

[Package \(PyPi\)](#) ↗ | [API reference documentation](#) | [Product documentation](#) | [Samples](#) ↗

By downloading, installing, using or accessing this distribution package for Semantic Link, you agree to the [Terms of Service](#) ↗.

This package has been tested with Microsoft Fabric.

Prerequisites

- A [Microsoft Fabric subscription](#). Or sign up for a free [Microsoft Fabric \(Preview\)](#) trial.
- Sign in to [Microsoft Fabric](#) ↗.
- Create [a new notebook](#) or [a new spark job](#) to use this package.

! Note

Semantic Link is supported only within Microsoft Fabric.

About the Semantic Link packages

The functionalities for Semantic Link are split into multiple packages to allow for a modular installation. If you want to install only a subset of the Semantic Link functionalities, you can install the individual packages instead of the `semantic-link` meta-package. This can help solve dependency issues. The following are some of the available packages:

- [semantic-link](#) ↗ - The meta-package that depends on all the individual Semantic Link packages and serves as a convenient way to install all the Semantic Link packages at once.
- [semantic-link-sempy](#) ↗ - The package that contains the core Semantic Link functionality.

- [semantic-link-functions-holidays](#) - A package that contains semantic functions for holidays and dependence on [holidays](#).
- [semantic-link-functions-geopandas](#) - A package that contains semantic functions for geospatial data and dependence on [geopandas](#).
- ...

Install the `semantic-link` meta package

To install the `semantic-link` package in Microsoft Fabric, you have two options:

- Install the `SemPy` Python library in your notebook kernel by executing this code in a notebook cell:

```
Python
```

```
%pip install semantic-link
```

- Alternatively, you can add Semantic Link to your workspace libraries directly. For more information, see [Install workspace libraries](#).

Key concepts

SemPy offers the following capabilities:

- Connectivity to Power BI
- Connectivity through Power BI Spark native connector
- Data augmentation with Power BI measures
- Semantic propagation for pandas users
- Built-in and custom semantic functions

Next steps

View our [Samples](#)