Data Engineering, Data Science and Analytics With Databricks on Google Cloud

Includes use cases with code samples and notebooks

databricks

Contents

Introduction

Enterprises struggle to innovate with data and AI because their architecture, tools and infrastructure are too complex. Different types of data and the range of use cases across data science, machine learning and analytics have led to multiple, disjointed systems that are difficult to maintain and adapt to the ever-increasing set of business requirements.

Databricks solves these problems by providing a single Lakehouse Platform that unifies all your data, analytics and AI workloads. The Lakehouse Platform combines the best elements of data lakes and data warehouses — delivering the data management and performance typically found in data warehouses with the low-cost, flexible object stores offered by data lakes. This unified platform simplifies your data architecture by eliminating the data silos that traditionally separate analytics, data science and machine learning. It's built on open source and open standards to maximize flexibility.

By partnering with Google Cloud, Databricks can deliver their platform at Google Cloud's global scale. Databricks on Google Cloud simplifies data and AI for organizations by unifying data engineering, data science and analytics with an open lakehouse architecture. By leveraging Google Kubernetes Engine (GKE), Databricks on Google Cloud enables customers to deploy Databricks in a fully containerized cloud environment — for the first time. Tight integration with Google Cloud Storage, BigQuery and the Google Cloud AI Platform enables Databricks to work seamlessly across data and AI services on Google Cloud.

What this eBook covers

This eBook covers two practical scenarios for data engineering and data science using collaborative notebooks on Databricks. Databricks notebooks are interactive workspaces that natively support Python, R, SQL and Scala so practitioners can work together with the languages and libraries of their choice to discover, visualize and share insights. We've included code samples for a hands-on experience with Databricks on Google Cloud.

Who should read this eBook

This eBook is written primarily for data engineers and data scientists interested in building data pipelines and deploying machine learning models using Databricks and the tight integrations with Google Cloud.

Data Engineering Use Case: Refine and Analyze Loan Data

For this use case, we will build an open lakehouse using Delta Lake that provides reliability, security and performance on your data lake. Our goal is to refine and analyze data in Databricks and make it available instantly for querying to derive insights. In this example, we generate aggregated loan statuses by U.S. states to identify the current distribution of loans across the country.

What you need to get started

If you are not already set up with Databricks on Google Cloud, use the following documentation to get started easily:

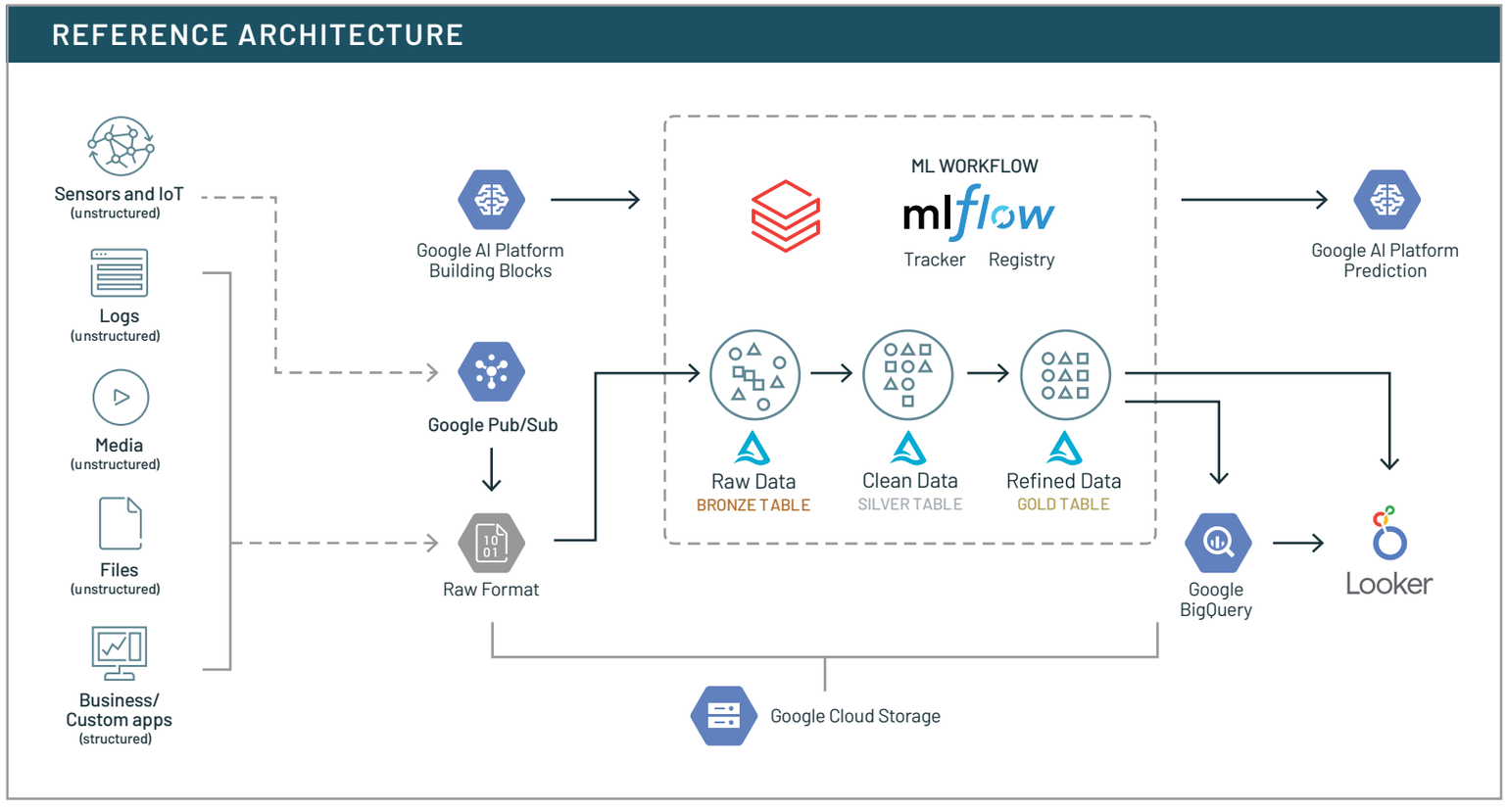
Set up your Databricks on Google Cloud account
Access Google Cloud Storage (GCS) tables in Databricks
Connect to BigQuery from Databricks
Use Looker with Databricks

Data set

We will use a publicly available loan data set from LendingClub. It contains loan information about the applicants along with their loan status, for example, fully paid, late or default. We want to refine this data set in Delta Lake, update it whenever there's a change in loan status, aggregate it by state and loan status to see which states have the highest loan records and finally, generate a dashboard that we can share with business users across the organization.

Public data set: www.kaggle.com/wordsforthewise/lending-club

Also supplied within the Databricks workspace: `dbfs:/databricks-datasets/samples/lending_club/parquet/`



Build an ETL pipeline using Delta Lake on Google Cloud Storage

In this data engineering use case, we will build an ETL pipeline using Delta Lake on Google Cloud Storage. Delta Lake is an open-format storage layer that delivers reliability, security and performance on your data lake — for both streaming and batch operations. We use a common architecture that uses tables that correspond to different quality levels in the data engineering pipeline, progressively adding structure to the data: "Bronze tables" for data ingestion, "Silver tables" for transformation/feature engineering and "Gold tables" for machine learning training or prediction or aggregated business-level data. This architecture allows data engineers to build a pipeline that begins with raw data as a "single source of truth" from which everything downstream flows.

Learn more about Delta Lake from our documentation.

Analyze data in Databricks

You can use the Silver and Gold tables to perform data analysis right inside your Databricks notebook. Delta Lake supports data manipulation language (DML) commands including UPDATE, DELETE and MERGE. The UPDATE operation is used to selectively update any rows that match a filtering condition. In this example, we use UPDATE to only update the late payments on a select number of loans.

Use SQL queries to analyze data in the Delta Lake and instantly build Redash dashboards to identify trends in your data. As the underlying source of your consolidated Gold tables is a Delta Lake table, the view you see is updated in real-time. Because you are using Spark SQL, you can run aggregate queries at scale on this data.

Read data in BigQuery and visualize it in Looker

Users may read the data in BigQuery from Delta tables for BI and visualization. Databricks has an optimized connector with Google BigQuery that allows easy access to data in BigQuery directly via its Storage API for high-performance queries. The connector has support for additional predicate pushdown, querying named tables and views, and directly running SQL on BigQuery and loading the results in an Apache Spark™ DataFrame.

Subsequently, use Looker to pull data from both Databricks and BigQuery for BI and reporting. Looker integrates with Databricks, giving users the ability to directly query the data lake, providing an entirely new visualization experience.

Notebook

- Load the loan data set into a Delta Bronze table from Google Cloud Storage
- Refine the Bronze tables, write to a Delta Silver table
- Aggregate data and write to a Delta Gold table, push into BigQuery
- Analyze the distribution of loans by state in BigQuery
- Join the Delta table and BigQuery table, visualize the data in Looker

Try this notebook with setup scripts in Databricks

STEP 1: Run the following Python setup scripts, which curate the raw CSV files from the loan status parquet data set and create temporary tables that will be needed to build the Delta Lake pipeline. Note: These scripts assume that you have a GCS instance named `gcp_launch_event`.

```
from pyspark.sql.functions import *
from pyspark.sql.types import StructType,StructField, StringType, IntegerType

# Grab the parquet dataset that comes in the sample dataset path of a Databricks
workspace
lspq_path = “dbfs:/databricks-datasets/samples/lending_club/parquet/”

# Read loanstats_2012_2017.parquet
data = spark.read.parquet(lspq_path)

# Reduce the amount of data (to run on Community Edition)
(loan_stats, loan_stats_rest) = data.randomSplit([0.10, 0.90], seed=123)

#Update with your own GCS path
loan_stats.write.format(“csv”).save(“gs://gcp_launch_event/raw/”, header=’true’)

csv_path = dbutils.fs.ls(“gs://gcp_launch_event/raw/”)[5].path
df = spark.read.option(“header”, True).option(“inferSchema”, True).csv(csv_path)
csv_schema = df.schema

df_evolution = df.filter(“addr_state == ‘VT’”).limit(5)
df_evolution.select(“addr_state”, “loan_status”, “grade”).
createOrReplaceTempView(“new_records”)

data = [(“CA”, “Late (16-30 days)”, 79),
(“CA”, “Fully Paid”, 7992)]
  ]
```

```
schema = StructType([ \
    StructField(“addr_state”,StringType(),True), \
    StructField(“loan_status”,StringType(),True), \
    StructField(“count”,IntegerType(),True)
  ])
df_merge = spark.createDataFrame(data=data,schema=schema).
createOrReplaceTempView(“merge_records”)

# Optionally insert your own database name below
sql(“””create database gcp_launch_event”””)
sql(“””use gcp_launch_event”””)
```

STEP 2: Ingest the loan status data and write them into the Delta Bronze table:

```
df_csv = spark.read.schema(csv_schema).option(“header”, True).csv(“gs://gcp_
launch_event/raw/”)
df_csv.write.format(“delta”).saveAsTable(“loanstats_delta_bronze”)
display(table(“loanstats_delta_bronze”))
```

Your Delta Bronze table snippet should look like this, and you should see the loan applicant information:

	id	member_id	loan_amnt	funded_amnt	funded_amnt_inv	term	int_rate	installment	grade	sub_grade
1	null	null	1000	1000	1000	36 months	5.32%	30.12	A	A1
2	null	null	1000	1000	1000	36 months	5.32%	30.12	A	A1
3	null	null	1000	1000	1000	36 months	5.32%	30.12	A	A1
4	null	null	1000	1000	1000	36 months	6.00%	30.42	D	D1
5	null	null	1000	1000	1000	36 months	6.49%	30.65	A	A2
6	null	null	1000	1000	1000	36 months	6.49%	30.65	A	A2

STEP 3: However there are null values and a lot of columns that we don't need. Let's remove all but two columns — "addr_state" and "loan_status" — and create a new Delta Silver table using SQL. You will need to add a %sql magic command in the cell to convert your Python cell into a SQL cell. (Note: You can always run the following in PySpark.)

%sql

```
CREATE TABLE loanstats_delta_silver
USING delta
AS
  SELECT addr_state, loan_status
  FROM loanstats_delta_bronze
  WHERE addr_state is not null;

SELECT * FROM loanstats_delta_silver;
```

Your simplified Silver table should look similar to the following:

	addr_state	loan_status
1	MN	Late (31-120 days)
2	CA	Current
3	NJ	Current
4	TX	Current
5	IL	Charged Off
6	MI	Current

STEP 4: Let's let the underwriter assign a "grade" to this data to determine the risk of the loan applicant. We will append an extra column "grade" to the table and evolve our Silver table schema with Python:

```
(table("new_records")
 .write
 .format("delta")
 .option("mergeSchema", True) #Add merge schema option
 .mode("append")
 .saveAsTable("loanstats_delta_silver"))

display(table("loanstats_delta_silver").filter("addr_state == 'VT'").
orderBy(col("grade").desc()).limit(20))
```

Your final Silver table should look like the following:

	addr_state	loan_status	grade
1	VT	Fully Paid	D
2	VT	Late (31-120 days)	C
3	VT	Current	C
4	VT	Fully Paid	C
5	VT	Fully Paid	B
6	VT	Charged Off	null

You will see that records that did not contain a grade column now have null values in that column.

Note: Delta Lake has schema enforcement by default. Without adding the mergeSchema option, your append operation will fail due to a schema mismatch.

STEP 5: Let's see the distribution of loan status type by state. We will create our final Gold tables and aggregate by grouping the records by state and loan status and then running a count using SQL:

```
CREATE TABLE loanstats_delta_gold
USING delta AS
SELECT addr_state, loan_status, count(*) AS count FROM loanstats_delta_silver
GROUP BY addr_state, loan_status;

SELECT * FROM loanstats_delta_gold WHERE addr_state = 'CA' ORDER BY loan_status;
```

Let's only look at records in California. Your Gold table, filtered by addr_state = CA, will look like the following:

	addr_state	loan_status	count
1	CA	Charged Off	1896
2	CA	Current	10046
3	CA	Default	1
4	CA	Fully Paid	7990
5	CA	In Grace Period	165
6	CA	Late (16-30 days)	81

We just received a notice that more loans were fully paid, including two loans with payments that were late by 16–30 days, and we need to UPDATE our Delta table. Run the following SQL command to update the count:

```
MERGE INTO loanstats_delta_gold g
USING merge_records m
ON g.addr_state = m.addr_state AND g.loan_status = m.loan_status
WHEN MATCHED THEN UPDATE SET *;

SELECT * FROM loanstats_delta_gold WHERE addr_state = 'CA';
```

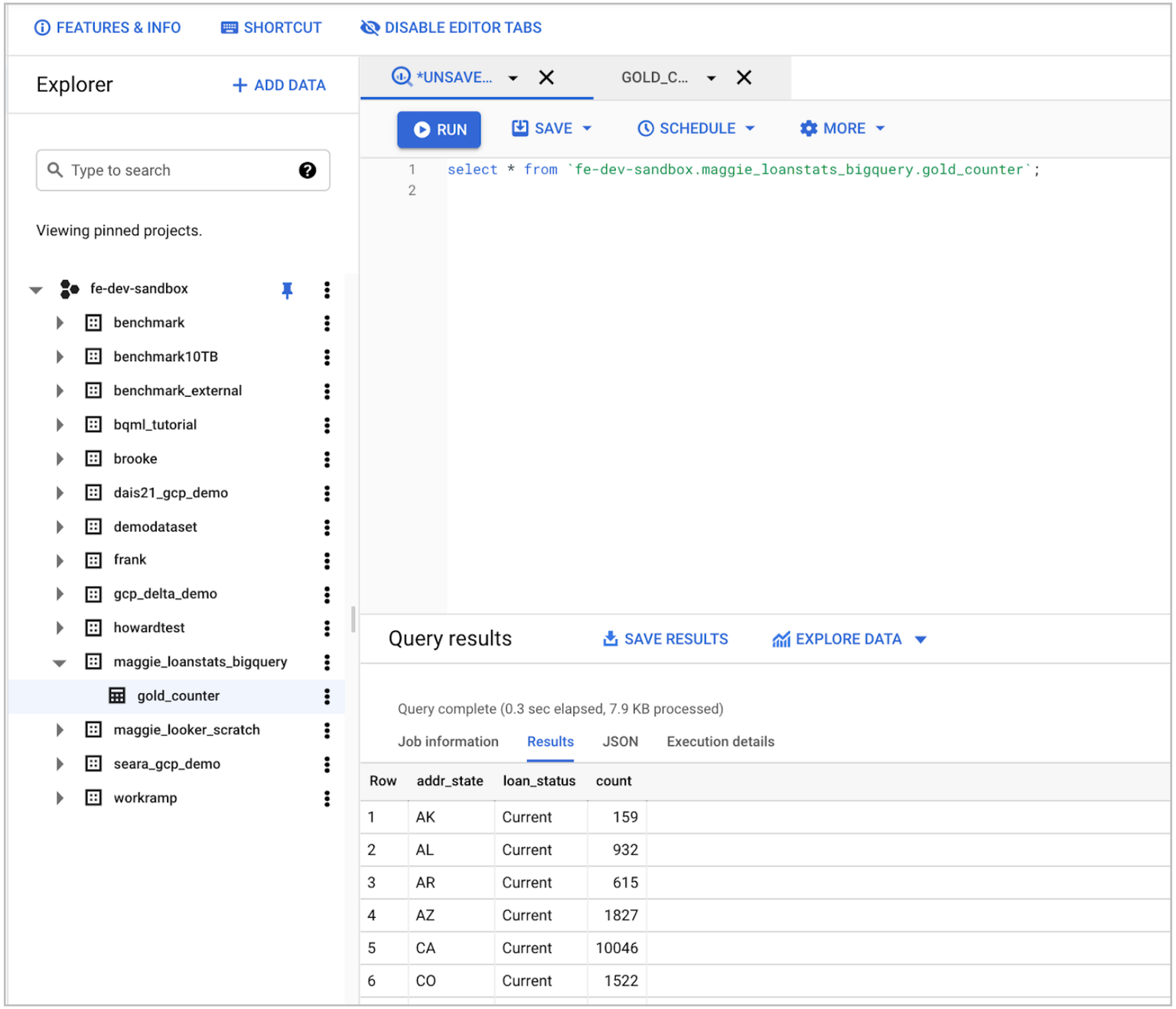
Your Gold table should now be updated with the new count!

	addr_state	loan_status	count
1	CA	Charged Off	1896
2	CA	Current	10046
3	CA	Default	1
4	CA	Fully Paid	7992
5	CA	In Grace Period	165
6	CA	Late (16-30 days)	79

STEP 6: You may choose to share the Gold tables with analysts who use BigQuery as their data source. Let's use the built-in BigQuery connector to write our Gold table into a BigQuery table called fe-dev-sandbox.loanstats_bigquery.gold_counter in Python:

```
(table("loanstats_delta_gold").write
  .format("bigquery")
  .mode("overwrite")
  .option("temporaryGcsBucket", "gcp_launch_event")
#Create a temporary GCS bucket
  .option("table", "fe-dev-sandbox.loanstats_bigquery.gold_counter")
#Write into a BigQuery table
  .save())
```

Our BigQuery table fe-dev-sandbox.loanstats_bigquery.gold_counter will look like the following:



STEP 7: Your analysts may also build business-level reports and will need to share their insights. Pull this BigQuery table into a Looker dashboard and visualize the data. Bonus: Connect Looker to a Delta table directly and visualize the Silver or Bronze layers!

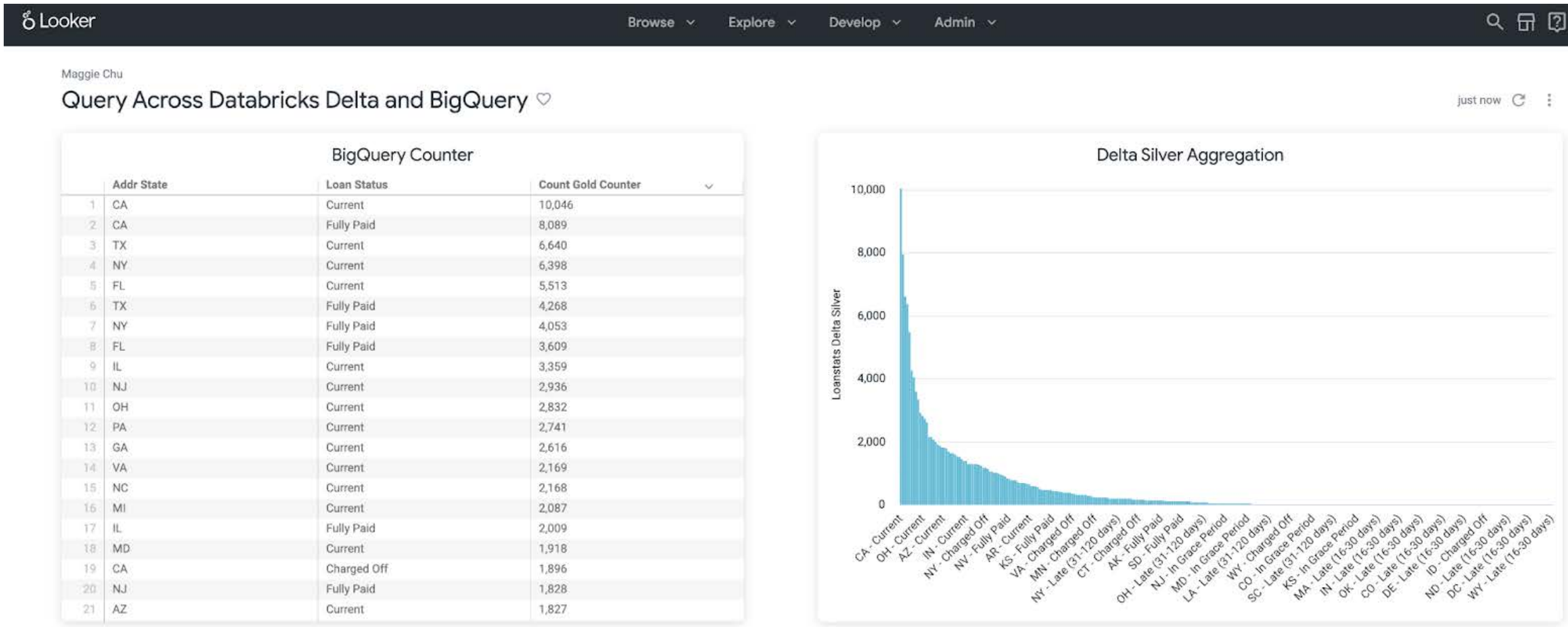


Figure 1: Looker dashboard

Data Science Use Case: Predicting Bike Trip Time

Databricks Machine Learning is an integrated end-to-end machine learning environment incorporating managed services for experiment tracking, model training, feature development and management, as well as feature and model serving. In this use case, we will create feature tables and access them for model training and inference, share, manage and serve the model using Model Registry and, finally, deploy the model for generating predictions.

In this example, we will use machine learning to predict how long a bike trip will take between two stations, given the subscription type, time of day and other features.

What you need to get started

If you are not already set up with Databricks on Google Cloud, use the following documentation to get started easily:

Set up your Databricks on Google Cloud account
Use Databricks Runtime for Machine Learning with MLflow
Connect to BigQuery from Databricks
Add an MLflow Model to the Model Registry

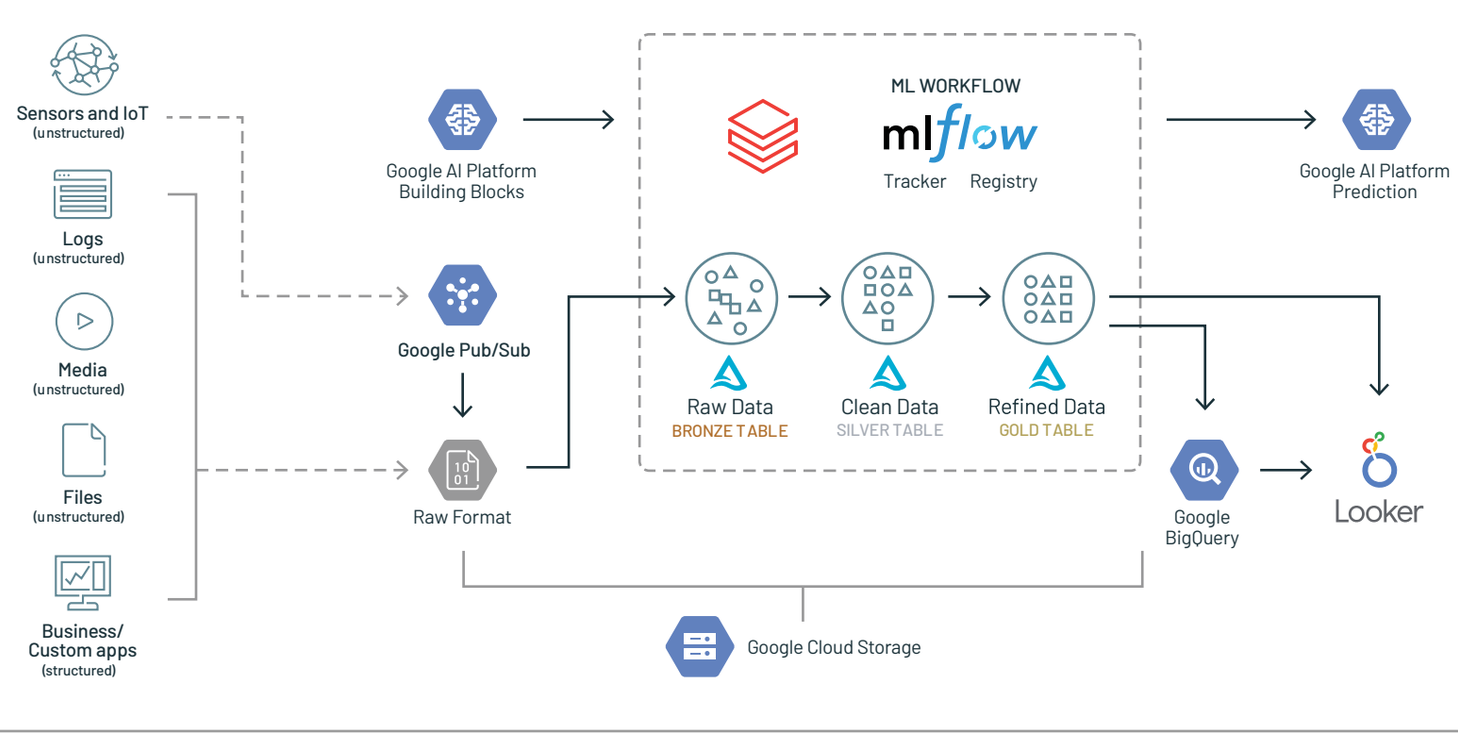
Data set

We will use the publicly available data set for bike sharing in Austin, Texas, to predict bike utilization in that city. The data set includes information such as subscription type, trip ID, start and end station locations, and start and end ride times, etc. However, some of the stations are now closed. We need to join this with the bikeshare_stations data set to get the latest information on active stations. You will find both these data sets available at the link below.

Public data set: www.kaggle.com/jboysen/austin-bike

Also supplied within this Databricks notebook

REFERENCE ARCHITECTURE



Automate experiment tracking with Managed MLflow

We use Databricks' collaborative features for data science and machine learning to manage the full ML lifecycle from experimentation to production. In particular, we leverage MLflow — an open source tool created by Databricks to manage the machine learning lifecycle. Databricks provides a managed MLflow offering, which comes with built-in integrations with the Databricks workspace to securely share, version, manage and reproduce machine learning experiments.

MLflow Tracking
Record and query experiments: code, data, configuration and results

MLflow Projects
Package data science code in a format to reproduce runs on any platform

MLflow Models
Deploy machine learning models in diverse serving environments

MLflow Registry
Store, annotate, discover and manage models in a central repository

Deploying an MLflow model

Databricks simplifies the workflow of deploying ML models in a batch or streaming fashion using MLflow's spark_udf. For real-time serving, use Databricks MLflow Model Serving, which simplifies deploying a model and keeping it updated. Databricks MLflow Model Serving ties directly into the MLflow Model Registry to automatically deploy new versions of a model and route requests to them. The model registry can store models from all machine learning libraries (TensorFlow, scikit-learn, etc.), and lets you store multiple versions of a model, review them and promote them to different lifecycle stages, such as Staging and Production. Once we have built and versioned your model with the MLflow Model Registry, you may choose to write the inference to a BigQuery table.

Notebook

- Load data from BigQuery
  —Use Apache Spark for feature preprocessing
- Build an sklearn model and automatically track parameters, metrics, artifacts, etc. with MLflow
  —Version the model with MLflow Model Registry
- Deploy the model
  —Write inference to a BigQuery table

Try this notebook in Databricks

STEP 1: Use Apache Spark for feature preprocessing.

We start with Apache Spark for distributed feature preprocessing, including joining our two bike sharing DataFrames, selecting columns, filtering out records and then dropping null records.

In this example, built-in Databricks visualization helps you view trends, such as the start times of the bike rides in Austin.

As you'll see in the code snippet below, we are joining the two DataFrames and then filtering. However, you might be wondering: Would it be faster to filter and then join? That is precisely what the Spark Catalyst Optimizer does under the hood! If you dig into the Spark UI, you'll see that it pushes the filter before the join. While it is still a best practice to rearrange your code to filter and then join, the Catalyst Optimizer can rearrange the execution order of your query with the guarantee that it will not change the end result.

Join data sets and visualize

```
stations_df = spark.read.format("bigquery").option("table",
"bigquery-public-data.austin_bikeshare.bikeshare_stations").load()
spark_df = (trips_df
 .join(stations_df, on=trips_df.start_station_name==stations_df.name)
 .selectExpr("subscriber_type", "dayofweek(start_time) as day_of_week",
"hour(start_time) as start_hour", "start_station_id",
            "end_station_id", "duration_minutes")
 .filter("duration_minutes > 0 AND duration_minutes < 180")
 .filter("status = 'active'")
 .dropna())
display(spark_df)
```

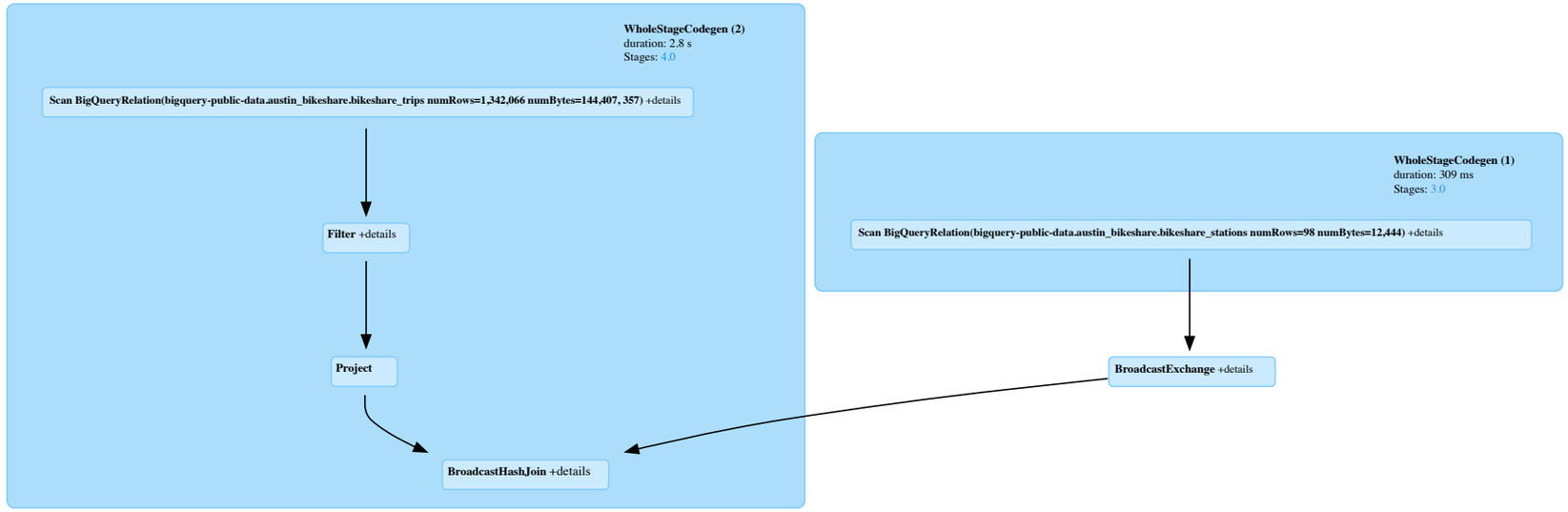


Figure 2: Spark UI query plan details

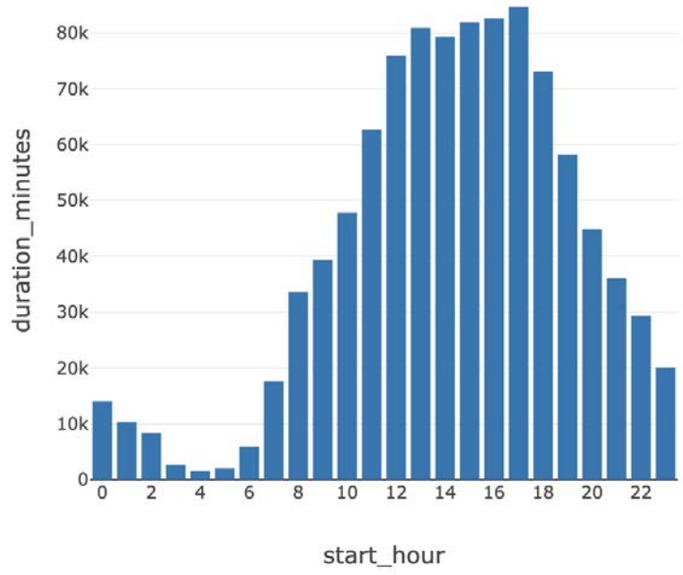


Figure 3: Distribution of bike ride start times

STEP 2: Convert the Spark DataFrame to a pandas DataFrame and build an sklearn model on the processed data set.

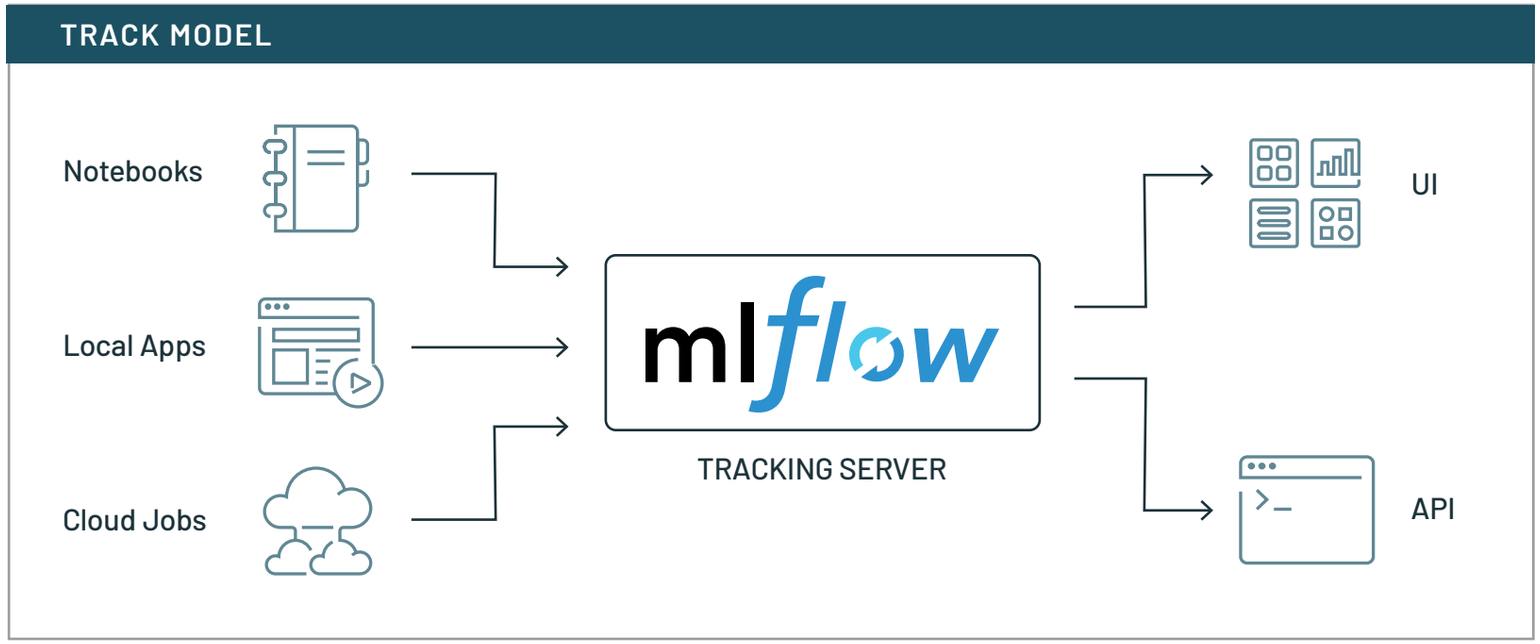
For this example, we will build a linear regression model predicting the duration of a trip in minutes given our features. For the categorical features, we'll also one hot encode them. But before we fit our model to our data set, we're going to make one tweak. We're going to import MLflow and enable the sklearn autologging capability to automatically track our experiment to the MLflow tracking server. We can then query the results of this run from the MLflow tracking server via the UI or API.

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
import mlflow
df = spark_df.toPandas()
X = df.drop(columns=[“duration_minutes”])
y = df.duration_minutes

cat_features = [“subscriber_type”, “day_of_week”, “start_station_id”,
“end_station_id”]

ohe = OneHotEncoder(handle_unknown=“ignore”)
preprocessor = ColumnTransformer(transformers=[(“ohe”, ohe, cat_features)])
lr = LinearRegression()

pipeline = Pipeline([
   (“preprocessor”, preprocessor),
   (“lr”, lr)
])
mlflow.sklearn.autolog()
pipeline.fit(X,y)
```



STEP 3: Register the MLflow model in the central model registry.

Use the MLflow tracking API to automatically track parameters, metrics, model, etc. for the sklearn model developed in this notebook. Then, register this model with the Model Registry to version it and make it more discoverable. The MLflow Model Registry allows you to keep track of all the pending requests and the activities performed on the model, giving you a view of the full lineage of the model. No need to specify whether it is an sklearn model or a TensorFlow model, MLflow automatically keeps track of that.

```
logged_model = “models:/Austin_Bike_Share/Staging”
loaded_model = mlflow.pyfunc.spark_udf(spark, model_uri=logged_model)
```

STEP 4: Deploy the model for batch inference and write predictions to BigQuery.

Once the model is moved into staging, we can apply it to a Spark DataFrame.

```
features = spark_df.drop("duration_minutes").columns
pred_df = spark_df.withColumn("prediction", loaded_model(*features))
display(pred_df)
```

Your predictions may look something like this.

	subscriber_type	day_of_week	start_hour	start_station_id	end_station_id	duration_minutes	prediction
1	Founding Member (Austin B-cycle)	6	11	2823	2823	16	22.286556168692155
2	Annual Membership (Austin B-cycle)	6	13	2823	2823	23	22.530732982991914
3	Annual Membership (Austin B-cycle)	6	15	2823	2823	4	22.530732982991914
4	Annual Membership (Austin B-cycle)	6	16	2823	2536	10	10.751374364991559
5	Weekender	5	19	2568	2536	3	14.562198063999087
6	Weekender	5	8	2568	2536	11	14.562198063999087

Once you have your results, you may choose to write the predictions to BigQuery. As mentioned in the data engineering use case, Databricks has an optimized connector with Google BigQuery that allows easy access to data in BigQuery directly via its Storage API.

```
(pred_df
 .write
 .format("bigquery")
 .mode("overwrite")
 .option("temporaryGcsBucket", "brooke-gcp-demo")
 .option("table", "fe-dev-sandbox.brooke.austin_bike_pred_df")
 .save())
```

OPTIONAL: Deploy to the Google Cloud AI Platform for real-time inferencing

In addition to batch or streaming inference leveraging Spark, we can also deploy to the Google Cloud AI Platform for real-time inference. Google Cloud's AI Platform is a fully managed, end-to-end platform for data science and machine learning, which provides tools for point-and-click data science using AutoML as well as advanced model optimization. The AI Platform helps data scientists and ML engineers be productive through ideation, experimentation, deployment and management of models at scale. It provides a low latency, auto-scaling, Kubernetes-based serving infrastructure for model deployment and model monitoring.

Once you have your model deployed in the AI Platform, you can pass sample records to generate predictions. We have our subscription type, day of week, time of day, start station ID and end station ID. Let's quickly generate some predictions.

```
{
  "instances": [
    ["Weekender", 7, 13, 2568, 2536],
    ["24 Hour Walk Up", 4, 9, 3293, 2576],
    ["Annual", 5, 11, 2823, 2823]
  ]
}
```

Result:

```
{
  "predictions": [
    23.360899363705965,
    20.397530185653874,
    21.56391079297561
  ]
}
```

What if we could convince a weekend rider to get an annual subscription? They'd likely ride more frequently, and for the same trip, they'd reduce their expected travel time by 5 minutes! Test out your model and generate quick predictions with the AI Platform.

```
{
  "instances": [
    ["Annual", 7, 13, 2568, 2536],
    ["24 Hour Walk Up", 4, 9, 3293, 2576],
    ["Annual", 5, 11, 2823, 2823]
  ]
}
```

Result:


```
{
  "predictions": [
    18.1635271025103,
    20.397530185653874,
    21.56391079297561
  ]
}
```

About Databricks

Databricks is the data and AI company. More than 5,000 organizations worldwide — including Comcast, Condé Nast, H&M and over 40% of the Fortune 500 — rely on the Databricks Lakehouse Platform to unify their data, analytics and AI. Databricks is headquartered in San Francisco, with offices around the globe. Founded by the original creators of Apache Spark™, Delta Lake and MLflow, Databricks is on a mission to help data teams solve the world's toughest problems. To learn more, follow Databricks on Twitter, LinkedIn and Facebook.

Sign up for a free trial >