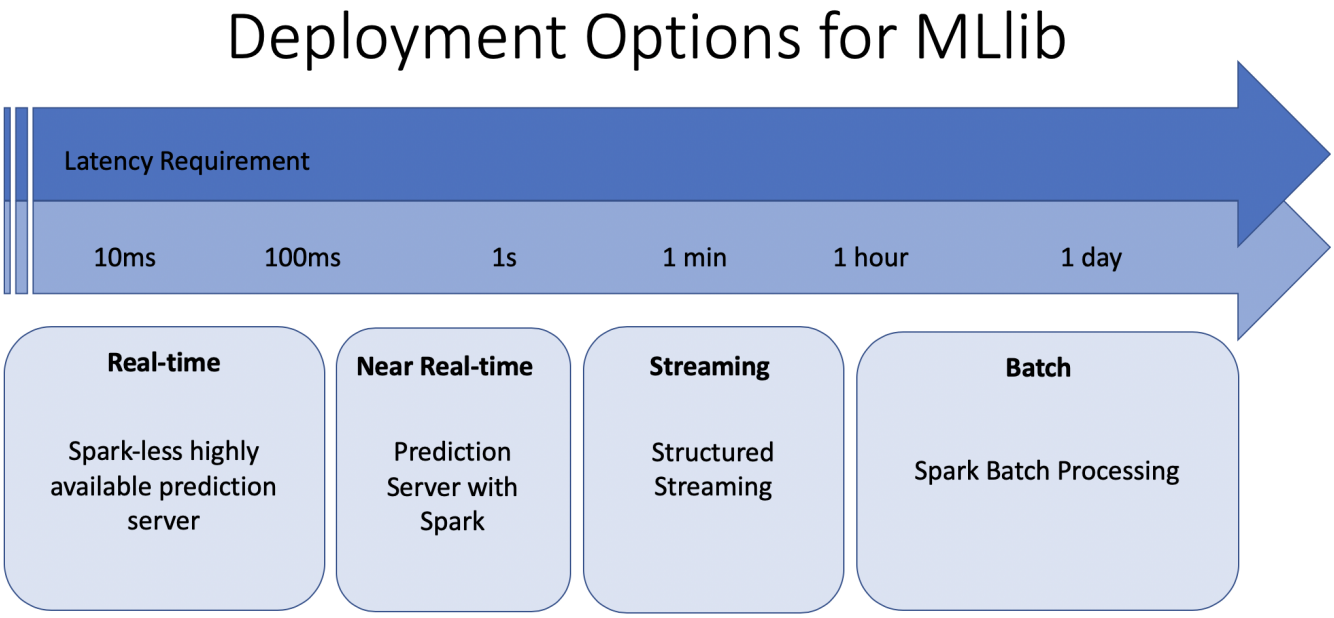
-sandbox

--i18n-59431a59-5305-45dc-81c5-bc13132e61ce



There are four main deployment options:

- Batch pre-compute
- Structured streaming
- Low-latency model serving
- Mobile/embedded (outside scope of class)

We have already seen how to do batch predictions using Spark. Now let's look at how to make predictions on streaming data.

In this lesson you:

- Apply a SparkML model on a simulated stream of data

In [0]:

```
%run "../Includes/Classroom-Setup"
```

--i18n-46846e08-4b50-4297-a871-98beaf65c3f7

Load in Model & Data

We are loading in a repartitioned version of our dataset (100 partitions instead of 4) to see more incremental progress of the streaming predictions.

In [0]:

```
from pyspark.ml.pipeline import PipelineModel

pipeline_path = f"{DA.paths.datasets}/airbnb/sf-listings/models/sf-listings-2019-03
pipeline_model = PipelineModel.load(pipeline_path)

repartitioned_path =  f"{DA.paths.datasets}/airbnb/sf-listings/sf-listings-2019-03-
schema = spark.read.parquet(repartitioned_path).schema
```

--i18n-6d5976b8-54b3-4379-9240-2fb9b7941f4c

Simulate streaming data

NOTE: You must specify a schema when creating a streaming source DataFrame.

In [0]:

```
streaming_data = (spark
                  .readStream
                  .schema(schema) # Can set the schema this way
                  .option("maxFilesPerTrigger", 1)
                  .parquet(repartitioned_path))
```

--i18n-29c9d057-1b46-41ff-a7a0-2d80a113e7a3

Make Predictions

In [0]:

```
stream_pred = pipeline_model.transform(streaming_data)
```

--i18n-d0c54563-04fc-48f3-b739-9acc85723d51

Let's save our results.

In [0]:

```
import re

checkpoint_dir = f"{DA.paths.working_dir}/stream_checkpoint"
# Clear out the checkpointing directory
dbutils.fs.rm(checkpoint_dir, True)

query = (stream_pred.writeStream
                    .format("memory")
                    .option("checkpointLocation", checkpoint_dir)
                    .outputMode("append")
                    .queryName("pred_stream")
                    .start())
```

In [0]:

```
DA.block_until_stream_is_ready(query)
```

--i18n-3654909a-da6d-4e8e-919a-9802e8292e77

While this is running, take a look at the new Structured Streaming tab in the Spark UI.

In [0]:

```
%sql
select * from pred_stream
```

In [0]:

```
%sql
select count(*) from pred_stream
```

--i18n-fb17c70a-c926-446c-a94a-900afc08efff

Now that we are done, make sure to stop the stream

In [0]:

```
for stream in spark.streams.active:
    print(f"Stopping {stream.name}")
    stream.stop()             # Stop the active stream
    stream.awaitTermination() # Wait for it to actually stop
```

--i18n-622245a0-07c0-43ee-967c-41cb4a601152

What about Model Export?

- ONNX (https://onnx.ai/)
  - ONNX is very popular in the deep learning community allowing developers to switch between libraries and languages, but only has experimental support for MLlib.
- DIY (Reimplement it yourself)
  - Error-prone, fragile

- 3rd party libraries
  - See XGBoost notebook
  - H2O (https://www.h2o.ai/products/h2o-sparkling-water/)

--i18n-39b0e95b-29e0-462f-a7ec-17bb6c5469ef

Low-Latency Serving Solutions

Low-latency serving can operate as quickly as tens to hundreds of milliseconds. Custom solutions are normally backed by Docker and/or Flask (though Flask generally isn't recommended in production unless significant precations are taken). Managed solutions also include:

- MLflow Model Serving (https://docs.databricks.com/applications/mlflow/model-serving.html)
- Azure Machine Learning (https://azure.microsoft.com/en-us/services/machine-learning/)
- SageMaker (https://aws.amazon.com/sagemaker/)

-sandbox

--i18n-62811f6d-e550-4c60-8903-f38d7ed56ca7

Regression: Predicting Rental Price

In this notebook, we will use the dataset we cleansed in the previous lab to predict Airbnb rental prices in San Francisco.

In this lesson you:

- Use the SparkML API to build a linear regression model
- Identify the differences between estimators and transformers

In [0]:

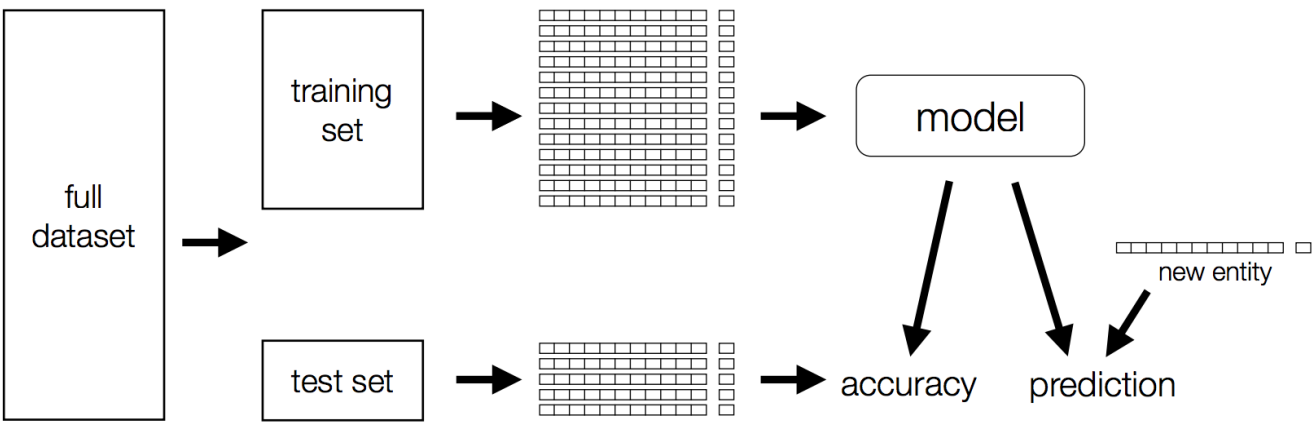
```
%run "./Includes/Classroom-Setup"
```

In [0]:

```
file_path = f"{DA.paths.datasets}/airbnb/sf-listings/sf-listings-2019-03-06-clean.d
airbnb_df = spark.read.format("delta").load(file_path)
```

--i18n-ee10d185-fc70-48b8-8efe-ea2feee28e01

Train/Test Split



Question: Why is it necessary to set a seed? What happens if I change my cluster configuration?

In [0]:

```
train_df, test_df = airbnb_df.randomSplit([.8, .2], seed=42)
print(train_df.cache().count())
```

--i18n-b70f996a-31a2-4b62-a699-dc6026105465

Let's change the # of partitions (to simulate a different cluster configuration), and see if we get the same number of data points in our training set.

In [0]:

```
train_repartition_df, test_repartition_df = (airbnb_df
                                             .repartition(24)
                                             .randomSplit([.8, .2], seed=42))

print(train_repartition_df.count())
```

--i18n-5b96c695-717e-4269-84c7-8292ceff9d83

Linear Regression

We are going to build a very simple model predicting **`price`** just given the number of **`bedrooms`** .

Question: What are some assumptions of the linear regression model?

In [0]:

```
display(train_df.select("price", "bedrooms"))
```

In [0]:

```
display(train_df.select("price", "bedrooms").summary())
```

In [0]:

```
display(train_df)
```

--i18n-4171a9ae-e928-41e3-9689-c6fcc2b3d57c

There does appear to be some outliers in our dataset for the price ($10,000 a night??). Just keep this in mind when we are building our models.

We will use LinearRegression (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.regression.LinearRegression.html?highlight=linearregression#pyspark.ml.regression.LinearRegression) to build our first model.

The cell below will fail because the Linear Regression estimator expects a vector of values as input. We will fix that with VectorAssembler below.

In [0]:

```
from pyspark.ml.regression import LinearRegression

lr = LinearRegression(featuresCol="bedrooms", labelCol="price")

# Uncomment when running
# lr_model = lr.fit(train_df)
```

--i18n-f1353d2b-d9b8-4c8c-af18-2abb8f0d0b84

Vector Assembler

What went wrong? Turns out that the Linear Regression **estimator** (**`.fit()`**) expected a column of Vector type as input.

We can easily get the values from the **`bedrooms`** column into a single vector using VectorAssembler (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.VectorAssembler.html?highlight=vectorassembler#pyspark.ml.feature.VectorAssembler). VectorAssembler is an example of a **transformer**. Transformers take in a DataFrame, and return a new DataFrame with one or more columns appended to it. They do not learn from your data, but apply rule based transformations.

You can see an example of how to use VectorAssembler on the ML Programming Guide (https://spark.apache.org/docs/latest/ml-features.html#vectorassembler).

In [0]:

```
from pyspark.ml.feature import VectorAssembler

vec_assembler = VectorAssembler(inputCols=["bedrooms"], outputCol="features")

vec_train_df = vec_assembler.transform(train_df)
```

In [0]:

```
lr = LinearRegression(featuresCol="features", labelCol="price")
lr_model = lr.fit(vec_train_df)
```

--i18n-ab8f4965-71db-487d-bbb3-329216580be5

Inspect the model

In [0]:

```
m = lr_model.coefficients[0]
b = lr_model.intercept

print(f"The formula for the linear regression line is y = {m:.2f}x + {b:.2f}")
```

--i18n-ae6dfaf9-9164-4dcc-a699-31184c4a962e

Apply model to test set

In [0]:

```
vec_test_df = vec_assembler.transform(test_df)

pred_df = lr_model.transform(vec_test_df)

pred_df.select("bedrooms", "features", "price", "prediction").show()
```

--i18n-8d73c3ee-34bc-4f8b-b2ba-03597548680c

Evaluate Model

Let's see how our linear regression model with just one variable does. Does it beat our baseline model?

In [0]:

```
from pyspark.ml.evaluation import RegressionEvaluator

regression_evaluator = RegressionEvaluator(predictionCol="prediction", labelCol="pr

rmse = regression_evaluator.evaluate(pred_df)
print(f"RMSE is {rmse}")
```

--i18n-703fbf0b-a2e1-4086-b002-8f63e06afdd8

Wahoo! Our RMSE is better than our baseline model. However, it's still not that great. Let's see how we can further decrease it in future notebooks.

-sandbox

--i18n-d6718279-32b1-490e-8a38-f1d6e3578184

Training with Pandas Function API

This notebook demonstrates how to use Pandas Function API to manage and scale machine learning models for IoT devices.

In this lesson you:

- Use **`.groupBy().applyInPandas()`** (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.GroupedData.applyInPandas.html?highlight=applyinpandas#pyspark.sql.GroupedData.applyInPandas) to build many models in parallel for each IoT Device

In [0]:

```
%run ./Includes/Classroom-Setup
```

--i18n-35af29dc-0fc5-4e37-963d-3fbe86f4ba59

Create dummy data with:

- **`device_id`** : 10 different devices
- **`record_id`** : 10k unique records
- **`feature_1`** : a feature for model training
- **`feature_2`** : a feature for model training
- **`feature_3`** : a feature for model training
- **`label`** : the variable we're trying to predict

In [0]:

```
import pyspark.sql.functions as f

df = (spark
      .range(1000*100)
      .select(f.col("id").alias("record_id"), (f.col("id")%10).alias("device_id"))
      .withColumn("feature_1", f.rand() * 1)
      .withColumn("feature_2", f.rand() * 2)
      .withColumn("feature_3", f.rand() * 3)
      .withColumn("label", (f.col("feature_1") + f.col("feature_2") + f.col("featur
     )

display(df)
```

--i18n-b5f90a62-80fd-4173-adf0-6e73d0e31309

Define the return schema

In [0]:

```
train_return_schema = "device_id integer, n_used integer, model_path string, mse fl
```

--i18n-e2ac315f-e950-48c6-9bb8-9ceede8f93dd

Define a pandas function that takes all the data for a given device, train a model, saves it as a nested run, and returns a spark object with the above schema

In [0]:

```
import mlflow
import mlflow.sklearn
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

def train_model(df_pandas: pd.DataFrame) -> pd.DataFrame:
    """
    Trains an sklearn model on grouped instances
    """
    # Pull metadata
    device_id = df_pandas["device_id"].iloc[0]
    n_used = df_pandas.shape[0]
    run_id = df_pandas["run_id"].iloc[0] # Pulls run ID to do a nested run

    # Train the model
    X = df_pandas[["feature_1", "feature_2", "feature_3"]]
    y = df_pandas["label"]
    rf = RandomForestRegressor()
    rf.fit(X, y)

    # Evaluate the model
    predictions = rf.predict(X)
    mse = mean_squared_error(y, predictions) # Note we could add a train/test split

    # Resume the top-level training
    with mlflow.start_run(run_id=run_id) as outer_run:
        # Small hack for running as a job
        experiment_id = outer_run.info.experiment_id
        print(f"Current experiment_id = {experiment_id}")

        # Create a nested run for the specific device
        with mlflow.start_run(run_name=str(device_id), nested=True, experiment_id=e
            mlflow.sklearn.log_model(rf, str(device_id))
            mlflow.log_metric("mse", mse)
            mlflow.set_tag("device", str(device_id))

            artifact_uri = f"runs:/{run.info.run_id}/{device_id}"
            # Create a return pandas DataFrame that matches the schema above
            return_df = pd.DataFrame([[device_id, n_used, artifact_uri, mse]],
                                     columns=["device_id", "n_used", "model_path", "

    return return_df
```

--i18n-2b6bf899-de7c-4ab9-b343-a11a832ddd77

Apply the pandas function to grouped data.

Note that the way you would apply this in practice depends largely on where the data for inference is located. In this example, we'll reuse the training data which contains our device and run id's.

In [0]:

```
with mlflow.start_run(run_name="Training session for all devices") as run:
    run_id = run.info.run_id

    model_directories_df = (df
        .withColumn("run_id", f.lit(run_id)) # Add run_id
        .groupby("device_id")
        .applyInPandas(train_model, schema=train_return_schema)
        .cache()
    )

combined_df = df.join(model_directories_df, on="device_id", how="left")
display(combined_df)
```

--i18n-3f660cc6-4979-48dd-beea-9dab9b536230

Define a pandas function and return schema to apply the model. *This needs only one read from DBFS per device.*

In [0]:

```
apply_return_schema = "record_id integer, prediction float"

def apply_model(df_pandas: pd.DataFrame) -> pd.DataFrame:
    """
    Applies model to data for a particular device, represented as a pandas DataFram
    """
    model_path = df_pandas["model_path"].iloc[0]

    input_columns = ["feature_1", "feature_2", "feature_3"]
    X = df_pandas[input_columns]

    model = mlflow.sklearn.load_model(model_path)
    prediction = model.predict(X)

    return_df = pd.DataFrame({
        "record_id": df_pandas["record_id"],
        "prediction": prediction
    })
    return return_df

prediction_df = combined_df.groupby("device_id").applyInPandas(apply_model, schema=
display(prediction_df)
```

--i18n-d760694c-8be7-4cbb-8825-8b8aa0d740db

Serving Multiple Models from a Registered Model

MLflow allows models to deploy as real-time REST APIs. At the moment, a single MLflow model serves from one instance (typically one VM). However, sometimes multiple models need to be served from a single endpoint. Imagine 1000 similar models that need to be served with different inputs. Running 1000 endpoints could waste resources, especially if certain models are underutilized.

One way around this is to package many models into a single custom model, which internally routes to one of the models based on the input and deploys that 'bundle' of models as a single 'model.'

Below we demonstrate creating such a custom model that bundles all of the models we trained for each device. For every row of data fed to this model, the model will determine the device id and then use the appropriate model trained on that device id to make predictions for a given row.

First, we need to access the models for each device id.

In [0]:

```
experiment_id = run.info.experiment_id

model_df = (spark.read.format("mlflow-experiment")
            .load(experiment_id)
            .filter("tags.device IS NOT NULL")
            .orderBy("end_time", ascending=False)
            .select("tags.device", "run_id")
            .limit(10))

display(model_df)
```

--i18n-b9b38048-397b-4eb3-a7c7-541aef502d4a

We create a dictionary mapping device ids to the model trained on that device id.

In [0]:

```
device_to_model = {row["device"]: mlflow.sklearn.load_model(f"runs:/{row['run_id']}

device_to_model
```

--i18n-f1081d85-677f-4a55-a3f5-a7e3a6710d3a

We create a custom model that takes the device id to model mappings as an attribute and delegates input to the appropriate model based on the device id.

In [0]:

```
from mlflow.pyfunc import PythonModel

class OriginDelegatingModel(PythonModel):

    def __init__(self, device_to_model_map):
        self.device_to_model_map = device_to_model_map

    def predict_for_device(self, row):
        '''
        This method applies to a single row of data by
        fetching the appropriate model and generating predictions
        '''
        model = self.device_to_model_map.get(str(row["device_id"]))
        data = row[["feature_1", "feature_2", "feature_3"]].to_frame().T
        return model.predict(data)[0]

    def predict(self, model_input):
        return model_input.apply(self.predict_for_device, axis=1)
```

--i18n-da424f95-113f-4feb-a20c-6d0178d03bdb

Here we demonstrate the use of this model.

In [0]:

```
example_model = OriginDelegatingModel(device_to_model)
example_model.predict(combined_df.toPandas().head(20))
```

--i18n-624309e5-7ba8-4968-92d4-3fe71e36375b

From here we can log and then register the model to be used for serving models for all the device ids from one instance.

In [0]:

```
with mlflow.start_run():
    model = OriginDelegatingModel(device_to_model)
    mlflow.pyfunc.log_model("model", python_model=model)
```

-sandbox

--i18n-2630af5a-38e6-482e-87f1-1a1633438bb6

AutoML

Databricks AutoML (https://docs.databricks.com/applications/machine-learning/automl.html) helps you automatically build machine learning models both through a UI and programmatically. It prepares the dataset for model training and then performs and records a set of trials (using HyperOpt), creating, tuning, and evaluating multiple models.

In this lesson you will:

- Use AutoML to automatically train and tune your models
- Run AutoML in Python and through the UI
- Interpret the results of an AutoML run

In [0]:

```
%run "./Includes/Classroom-Setup"
```

--i18n-7aa84cf3-1b6c-4ba4-9249-00359ee8d70a

Currently, AutoML uses a combination of XGBoost and sklearn (only single node models) but optimizes the hyperparameters within each.

In [0]:

```
file_path = f"{DA.paths.datasets}/airbnb/sf-listings/sf-listings-2019-03-06-clean.d
airbnb_df = spark.read.format("delta").load(file_path)
train_df, test_df = airbnb_df.randomSplit([.8, .2], seed=42)
```

--i18n-1b5c8a94-3ac2-4977-bfe4-51a97d83ebd9

We can now use AutoML to search for the optimal regression (https://docs.databricks.com/applications/machine-learning/automl.html#regression) model.

Required parameters:

- **`dataset`** - Input Spark or pandas DataFrame that contains training features and targets. If using a Spark DataFrame, it will convert it to a Pandas DataFrame under the hood by calling .toPandas() - just be careful

you don't OOM!
- **`target_col`** - Column name of the target labels

We will also specify these optional parameters:

- **`primary_metric`** - Primary metric to select the best model. Each trial will compute several metrics, but this one determines which model is selected from all the trials. One of **`r2`** (default, R squared), **`mse`** (mean squared error), **`rmse`** (root mean squared error), **`mae`** (mean absolute error) for regression problems.
- **`timeout_minutes`** - The maximum time to wait for the AutoML trials to complete. **`timeout_minutes=None`** will run the trials without any timeout restrictions
- **`max_trials`** - The maximum number of trials to run. When **`max_trials=None`**, maximum number of trials will run to completion.

In [0]:

```
from databricks import automl

summary = automl.regress(train_df, target_col="price", primary_metric="rmse", timeo
```

--i18n-57d884c6-2099-4f34-b840-a4e873308ffe

After running the previous cell, you will notice two notebooks and an MLflow experiment:

- **`Data exploration notebook`** - we can see a Profiling Report which organizes the input columns and discusses values, frequency and other information
- **`Best trial notebook`** - shows the source code for reproducing the best trial conducted by AutoML
- **`MLflow experiment`** - contains high level information, such as the root artifact location, experiment ID, and experiment tags. The list of trials contains detailed summaries of each trial, such as the notebook and model location, training parameters, and overall metrics.

Dig into these notebooks and the MLflow experiment - what do you find?

Additionally, AutoML shows a short list of metrics from the best run of the model.

In [0]:

```
print(summary.best_trial)
```

--i18n-3c0cd1ec-8965-4af3-896d-c30938033abf

Now we can test the model that we got from AutoML against our test data. We'll be using [mlflow.pyfunc.spark_udf (https://mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#mlflow.pyfunc.spark_udf)](https://mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#mlflow.pyfunc.spark_udf) to register our model as a UDF and apply it in parallel to our test data.

In [0]:

```
# Load the best trial as an MLflow Model
import mlflow

model_uri = f"runs:/{summary.best_trial.mlflow_run_id}/model"

predict = mlflow.pyfunc.spark_udf(spark, model_uri)
pred_df = test_df.withColumn("prediction", predict(*test_df.drop("price").columns))
display(pred_df)
```

In [0]:

```
from pyspark.ml.evaluation import RegressionEvaluator

regression_evaluator = RegressionEvaluator(predictionCol="prediction", labelCol="pr
rmse = regression_evaluator.evaluate(pred_df)
print(f"RMSE on test dataset: {rmse:.3f}")
```

-sandbox

--i18n-b27f81af-5fb6-4526-b531-e438c0fda55e

MLflow

MLflow (https://mlflow.org/docs/latest/concepts.html) seeks to address these three core issues:

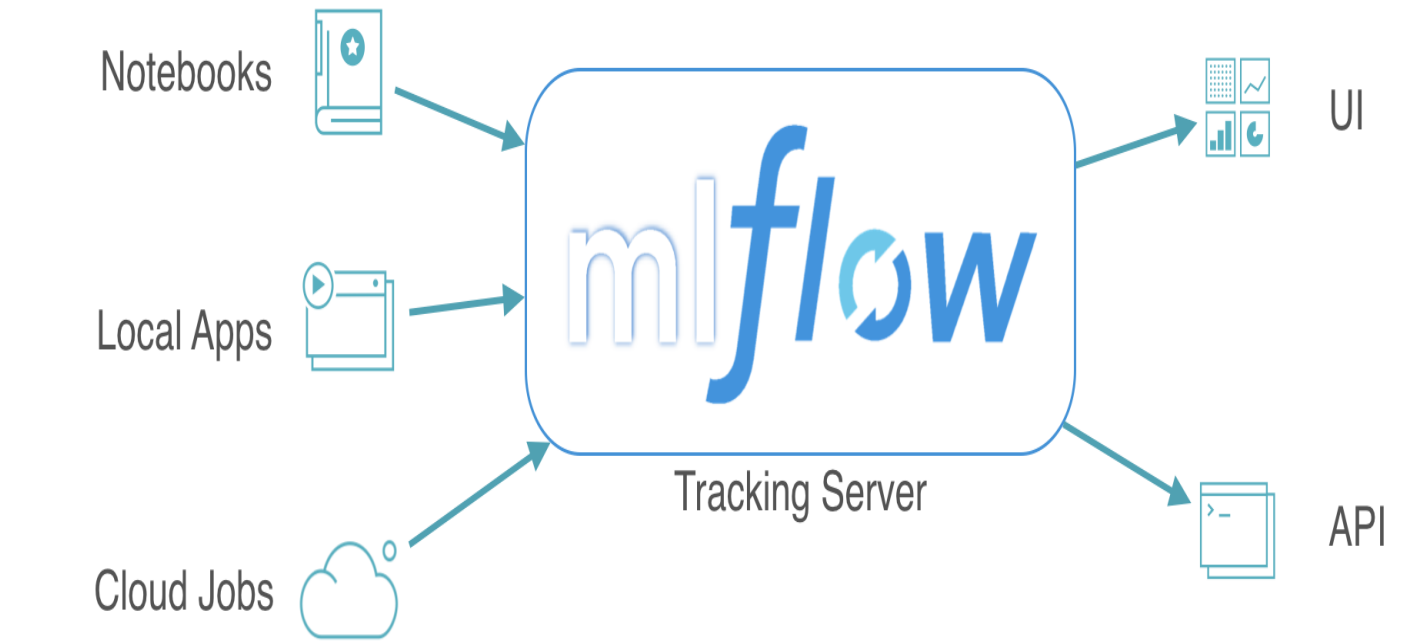
- It's difficult to keep track of experiments
- It's difficult to reproduce code
- There's no standard way to package and deploy models

In the past, when examining a problem, you would have to manually keep track of the many models you created, as well as their associated parameters and metrics. This can quickly become tedious and take up valuable time, which is where MLflow comes in.

MLflow is pre-installed on the Databricks Runtime for ML.

In this lesson you:

- Use MLflow to track experiments, log metrics, and compare runs



In [0]:

```
%run "./Includes/Classroom-Setup"
```

--i18n-c1a29688-f50a-48cf-9163-ebcc381dfe38

Let's start by loading in our SF Airbnb Dataset.

In [0]:

```
file_path = f"{DA.paths.datasets}/airbnb/sf-listings/sf-listings-2019-03-06-clean.d
airbnb_df = spark.read.format("delta").load(file_path)

train_df, test_df = airbnb_df.randomSplit([.8, .2], seed=42)
print(train_df.cache().count())
```

--i18n-9ab8c080-9012-4f38-8b01-3846c1531a80

MLflow Tracking

MLflow Tracking is a logging API specific for machine learning and agnostic to libraries and environments that do the training. It is organized around the concept of **runs**, which are executions of data science code. Runs are aggregated into **experiments** where many runs can be a part of a given experiment and an MLflow server can host many experiments.

You can use [mlflow.set_experiment()](https://mlflow.org/docs/latest/python_api/mlflow.html#mlflow.set_experiment) to set an experiment, but if you do not specify an experiment, it will automatically be scoped to this notebook.

--i18n-82786653-4926-4790-b867-c8ccb208b451

Track Runs

Each run can record the following information:

- **Parameters:** Key-value pairs of input parameters such as the number of trees in a random forest model
- **Metrics:** Evaluation metrics such as RMSE or Area Under the ROC Curve
- **Artifacts:** Arbitrary output files in any format. This can include images, pickled models, and data files
- **Source:** The code that originally ran the experiment

NOTE: For Spark models, MLflow can only log PipelineModels.

In [0]:

```
import mlflow
import mlflow.spark
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorAssembler
from pyspark.ml import Pipeline
from pyspark.ml.evaluation import RegressionEvaluator

with mlflow.start_run(run_name="LR-Single-Feature") as run:
    # Define pipeline
    vec_assembler = VectorAssembler(inputCols=["bedrooms"], outputCol="features")
    lr = LinearRegression(featuresCol="features", labelCol="price")
    pipeline = Pipeline(stages=[vec_assembler, lr])
    pipeline_model = pipeline.fit(train_df)

    # Log parameters
    mlflow.log_param("label", "price")
    mlflow.log_param("features", "bedrooms")

    # Log model
    mlflow.spark.log_model(pipeline_model, "model", input_example=train_df.limit(5)

    # Evaluate predictions
    pred_df = pipeline_model.transform(test_df)
    regression_evaluator = RegressionEvaluator(predictionCol="prediction", labelCol
    rmse = regression_evaluator.evaluate(pred_df)

    # Log metrics
    mlflow.log_metric("rmse", rmse)
```

--i18n-44bc7cac-de4a-47e7-bfff-6d2eb58172cd

There, all done! Let's go through the other two linear regression models and then compare our runs.

Question: Does anyone remember the RMSE of the other runs?

Next let's build our linear regression model but use all of our features.

In [0]:

```
from pyspark.ml.feature import RFormula

with mlflow.start_run(run_name="LR-All-Features") as run:
    # Create pipeline
    r_formula = RFormula(formula="price ~ .", featuresCol="features", labelCol="pri
    lr = LinearRegression(labelCol="price", featuresCol="features")
    pipeline = Pipeline(stages=[r_formula, lr])
    pipeline_model = pipeline.fit(train_df)

    # Log pipeline
    mlflow.spark.log_model(pipeline_model, "model", input_example=train_df.limit(5)

    # Log parameter
    mlflow.log_param("label", "price")
    mlflow.log_param("features", "all_features")

    # Create predictions and metrics
    pred_df = pipeline_model.transform(test_df)
    regression_evaluator = RegressionEvaluator(labelCol="price", predictionCol="pre
    rmse = regression_evaluator.setMetricName("rmse").evaluate(pred_df)
    r2 = regression_evaluator.setMetricName("r2").evaluate(pred_df)

    # Log both metrics
    mlflow.log_metric("rmse", rmse)
    mlflow.log_metric("r2", r2)
```

--i18n-70188282-8d26-427d-b374-954e9a058000

Finally, we will use Linear Regression to predict the log of the price, due to its log normal distribution.

We'll also practice logging artifacts to keep a visual of our log normal histogram.

In [0]:

```
from pyspark.sql.functions import col, log, exp
import matplotlib.pyplot as plt

with mlflow.start_run(run_name="LR-Log-Price") as run:
    # Take log of price
    log_train_df = train_df.withColumn("log_price", log(col("price")))
    log_test_df = test_df.withColumn("log_price", log(col("price")))

    # Log parameter
    mlflow.log_param("label", "log_price")
    mlflow.log_param("features", "all_features")

    # Create pipeline
    r_formula = RFormula(formula="log_price ~ . - price", featuresCol="features", l
    lr = LinearRegression(labelCol="log_price", predictionCol="log_prediction")
    pipeline = Pipeline(stages=[r_formula, lr])
    pipeline_model = pipeline.fit(log_train_df)

    # Log model
    mlflow.spark.log_model(pipeline_model, "log-model", input_example=log_train_df.

    # Make predictions
    pred_df = pipeline_model.transform(log_test_df)
    exp_df = pred_df.withColumn("prediction", exp(col("log_prediction")))

    # Evaluate predictions
    rmse = regression_evaluator.setMetricName("rmse").evaluate(exp_df)
    r2 = regression_evaluator.setMetricName("r2").evaluate(exp_df)

    # Log metrics
    mlflow.log_metric("rmse", rmse)
    mlflow.log_metric("r2", r2)

    # Log artifact
    plt.clf()

    log_train_df.toPandas().hist(column="log_price", bins=100)
    fig = plt.gcf()
    mlflow.log_figure(fig, f"{DA.username}_log_normal.png")
    plt.show()
```

--i18n-66785d5e-e1a7-4896-a8a9-5bfcd18acc5c

That's it! Now, let's use MLflow to easily look over our work and compare model performance. You can either query past runs programmatically or use the MLflow UI.

--i18n-0b1a68e1-bd5d-4f78-a452-90c7ebcdef39

Querying Past Runs

You can query past runs programmatically in order to use this data back in Python. The pathway to doing this is an **`MlflowClient`** object.

In [0]:

```
from mlflow.tracking import MlflowClient

client = MlflowClient()
```

In [0]:

```
client.list_experiments()
```

--i18n-dcd771b2-d4ed-4e9c-81e5-5a3f8380981f

You can also use search_runs (https://mlflow.org/docs/latest/search-syntax.html) to find all runs for a given experiment.

In [0]:

```
experiment_id = run.info.experiment_id
runs_df = mlflow.search_runs(experiment_id)

display(runs_df)
```

--i18n-68990866-b084-40c1-beee-5c747a36b918

Pull the last run and look at metrics.

In [0]:

```
runs = client.search_runs(experiment_id, order_by=["attributes.start_time desc"], m
runs[0].data.metrics
```

In [0]:

```
runs[0].info.run_id
```

--i18n-cfbbd060-6380-444f-ba88-248e10a56559

Examine the results in the UI. Look for the following:

1. The **`Experiment ID`**
2. The artifact location. This is where the artifacts are stored in DBFS.
3. The time the run was executed. **Click this to see more information on the run.**
4. The code that executed the run.

After clicking on the time of the run, take a look at the following:

1. The Run ID will match what we printed above
2. The model that we saved, included a pickled version of the model as well as the Conda environment and the **`MLmodel`** file.

Note that you can add notes under the "Notes" tab to help keep track of important information about your models.

Also, click on the run for the log normal distribution and see that the histogram is saved in "Artifacts".

--i18n-63ca7584-2a86-421b-a57e-13d48db8a75d

Load Saved Model

Let's practice loading (https://www.mlflow.org/docs/latest/python_api/mlflow.spark.html) our logged log-normal model.

In [0]:

```
model_path = f"runs:/{run.info.run_id}/log-model"
loaded_model = mlflow.spark.load_model(model_path)

display(loaded_model.transform(test_df))
```

-sandbox

--i18n-04aa5a94-e0d3-4bec-a9b5-a0590c33a257

Model Registry

MLflow Model Registry is a collaborative hub where teams can share ML models, work together from experimentation to online testing and production, integrate with approval and governance workflows, and monitor ML deployments and their performance. This lesson explores how to manage models using the MLflow model registry.

This demo notebook will use scikit-learn on the Airbnb dataset, but in the lab you will use MLlib.

In this lesson you:

- Register a model using MLflow
- Manage the model lifecycle
- Archive and delete models

If you would like to set up a model serving endpoint, you will need cluster creation (https://docs.databricks.com/applications/mlflow/model-serving.html#requirements) permissions.

In [0]:

```
%run "./Includes/Classroom-Setup"
```

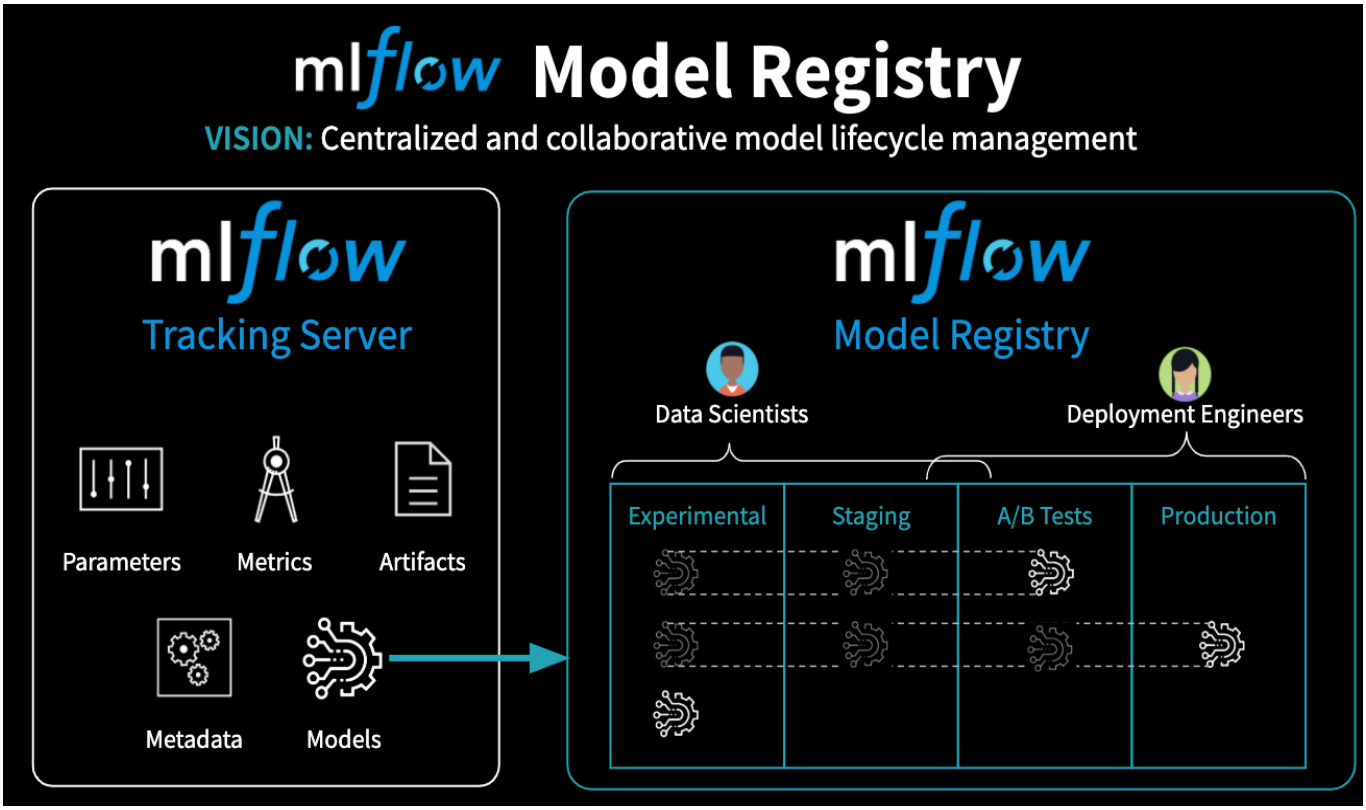
-sandbox --i18n-5802ff47-58b5-4789-973d-2fb855bf347a

Model Registry

The MLflow Model Registry component is a centralized model store, set of APIs, and UI, to collaboratively manage the full lifecycle of an MLflow Model. It provides model lineage (which MLflow Experiment and Run produced the model), model versioning, stage transitions (e.g. from staging to production), annotations (e.g. with comments, tags), and deployment management (e.g. which production jobs have requested a specific model version).

Model registry has the following features:

- **Central Repository:** Register MLflow models with the MLflow Model Registry. A registered model has a unique name, version, stage, and other metadata.
- **Model Versioning:** Automatically keep track of versions for registered models when updated.
- **Model Stage:** Assigned preset or custom stages to each model version, like “Staging” and “Production” to represent the lifecycle of a model.
- **Model Stage Transitions:** Record new registration events or changes as activities that automatically log users, changes, and additional metadata such as comments.
- **CI/CD Workflow Integration:** Record stage transitions, request, review and approve changes as part of CI/CD pipelines for better control and governance.



See [the MLflow docs (https://mlflow.org/docs/latest/registry.html)](https://mlflow.org/docs/latest/registry.html) for more details on the model registry.

--i18n-7f34f7da-b5d2-42af-b24d-54e1730db95f

Registering a Model

The following workflow will work with either the UI or in pure Python. This notebook will use pure Python.

Explore the UI throughout this lesson by clicking the "Models" tab on the left-hand side of the screen.

--i18n-cbc59424-e45b-4179-a586-8c14a66a61a1

Train a model and log it to MLflow using [autologging (https://docs.databricks.com/applications/mlflow/databricks-autologging.html)](https://docs.databricks.com/applications/mlflow/databricks-autologging.html). Autologging allows you to log metrics, parameters, and models without the need for explicit log statements.

There are a few ways to use autologging:

1. Call **`mlflow.autolog()`** before your training code. This will enable autologging for each supported library you have installed as soon as you import it.
2. Enable autologging at the workspace level from the admin console
3. Use library-specific autolog calls for each library you use in your code. (e.g. **`mlflow.spark.autolog()`**)

Here we are only using numeric features for simplicity of building the random forest.

In [0]:

```
import mlflow
import mlflow.sklearn
from mlflow.models.signature import infer_signature

import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

df = pd.read_csv(f"{DA.paths.datasets}/airbnb/sf-listings/airbnb-cleaned-mlflow.csv
X_train, X_test, y_train, y_test = train_test_split(df.drop(["price"], axis=1), df[

with mlflow.start_run(run_name="LR Model") as run:
    mlflow.sklearn.autolog(log_input_examples=True, log_model_signatures=True, log_
    lr = LinearRegression()
    lr.fit(X_train, y_train)
    signature = infer_signature(X_train, lr.predict(X_train))
```

--i18n-1322cac5-9638-4cc9-b050-3545958f3936

Create a unique model name so you don't clash with other workspace users.

Note that a registered model name must be a non-empty UTF-8 string and cannot contain forward slashes(/), periods(.), or colons(:).

In [0]:

```
model_name = f"{DA.cleaned_username}_sklearn_lr"
model_name
```

--i18n-0777e3f5-ba7c-41c4-a477-9f0a5a809664

Register the model.

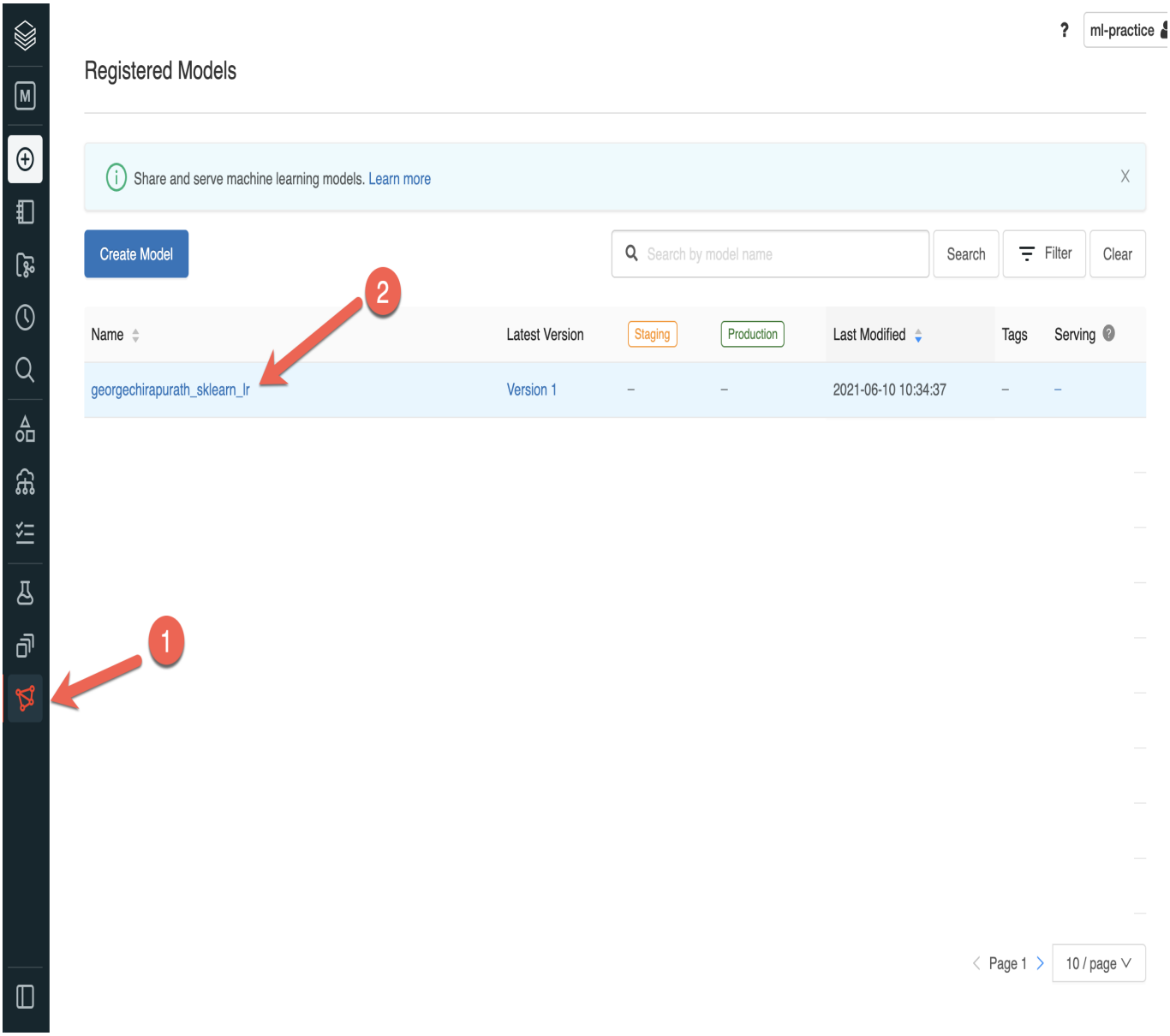
In [0]:

```
run_id = run.info.run_id
model_uri = f"runs:/{run_id}/model"

model_details = mlflow.register_model(model_uri=model_uri, name=model_name)
```

-sandbox --i18n-22756858-ff7f-4392-826f-f401a81230c4

Open the *Models* tab on the left of the screen to explore the registered model. Note the following:

- It logged who trained the model and what code was used
- It logged a history of actions taken on this model
- It logged this model as a first version



--i18n-481cba23-661f-4de7-a1d8-06b6be8c57d3

Check the status.

In [0]:

```
from mlflow.tracking.client import MlflowClient

client = MlflowClient()
model_version_details = client.get_model_version(name=model_name, version=1)

model_version_details.status
```

--i18n-10556266-2903-4afc-8af9-3213d244aa21

Now add a model description

In [0]:

```
client.update_registered_model(
    name=model_details.name,
    description="This model forecasts Airbnb housing list prices based on various l
)
```

--i18n-5abeafb2-fd60-4b0d-bf52-79320c10d402

Add a version-specific description.

In [0]:

```
client.update_model_version(
    name=model_details.name,
    version=model_details.version,
    description="This model version was built using OLS linear regression with skle
)
```

--i18n-aaac467f-3a52-4428-a119-8286cb0ac158

Deploying a Model

The MLflow Model Registry defines several model stages: **`None`**, **`Staging`**, **`Production`**, and **`Archived`**. Each stage has a unique meaning. For example, **`Staging`** is meant for model testing, while **`Production`** is for models that have completed the testing or review processes and have been deployed to applications.

Users with appropriate permissions can transition models between stages.

--i18n-dff93671-f891-4779-9e41-a0960739516f

Now that you've learned about stage transitions, transition the model to the **`Production`** stage.

In [0]:

```
import time

time.sleep(10) # In case the registration is still pending
```

In [0]:

```
client.transition_model_version_stage(
    name=model_details.name,
    version=model_details.version,
    stage="Production"
)
```

--i18n-4dc7e8b7-da38-4ce1-a238-39cad74d97c5

Fetch the model's current status.

In [0]:

```
model_version_details = client.get_model_version(
    name=model_details.name,
    version=model_details.version
)
print(f"The current model stage is: '{model_version_details.current_stage}'")
```

--i18n-ba563293-bb74-4318-9618-a1dcf86ec7a3

Fetch the latest model using a **`pyfunc`** . Loading the model in this way allows us to use the model regardless of the package that was used to train it.

You can load a specific version of the model too.

In [0]:

```
import mlflow.pyfunc

model_version_uri = f"models:/{model_name}/1"

print(f"Loading registered model version from URI: '{model_version_uri}'")
model_version_1 = mlflow.pyfunc.load_model(model_version_uri)
```

--i18n-e1bb8ae5-6cf3-42c2-aebd-bde925a9ef30

Apply the model.

In [0]:

```
model_version_1.predict(X_test)
```

--i18n-75a9c277-0115-4cef-b4aa-dd69a0a5d8a0

Deploying a New Model Version

The MLflow Model Registry enables you to create multiple model versions corresponding to a single registered model. By performing stage transitions, you can seamlessly integrate new model versions into your staging or production environments.

--i18n-2ef7acd0-422a-4449-ad27-3a26f217ab15

Create a new model version and register that model when it's logged.

In [0]:

```
from sklearn.linear_model import Ridge

with mlflow.start_run(run_name="LR Ridge Model") as run:
    alpha = .9
    ridge_regression = Ridge(alpha=alpha)
    ridge_regression.fit(X_train, y_train)

    # Specify the `registered_model_name` parameter of the `mlflow.sklearn.log_mode
    # function to register the model with the MLflow Model Registry. This automatic
    # creates a new model version

    mlflow.sklearn.log_model(
        sk_model=ridge_regression,
        artifact_path="sklearn-ridge-model",
        registered_model_name=model_name,
    )

    mlflow.log_params(ridge_regression.get_params())
    mlflow.log_metric("mse", mean_squared_error(y_test, ridge_regression.predict(X_
```

--i18n-dc1dd6b4-9e9e-45be-93c4-5500a10191ed

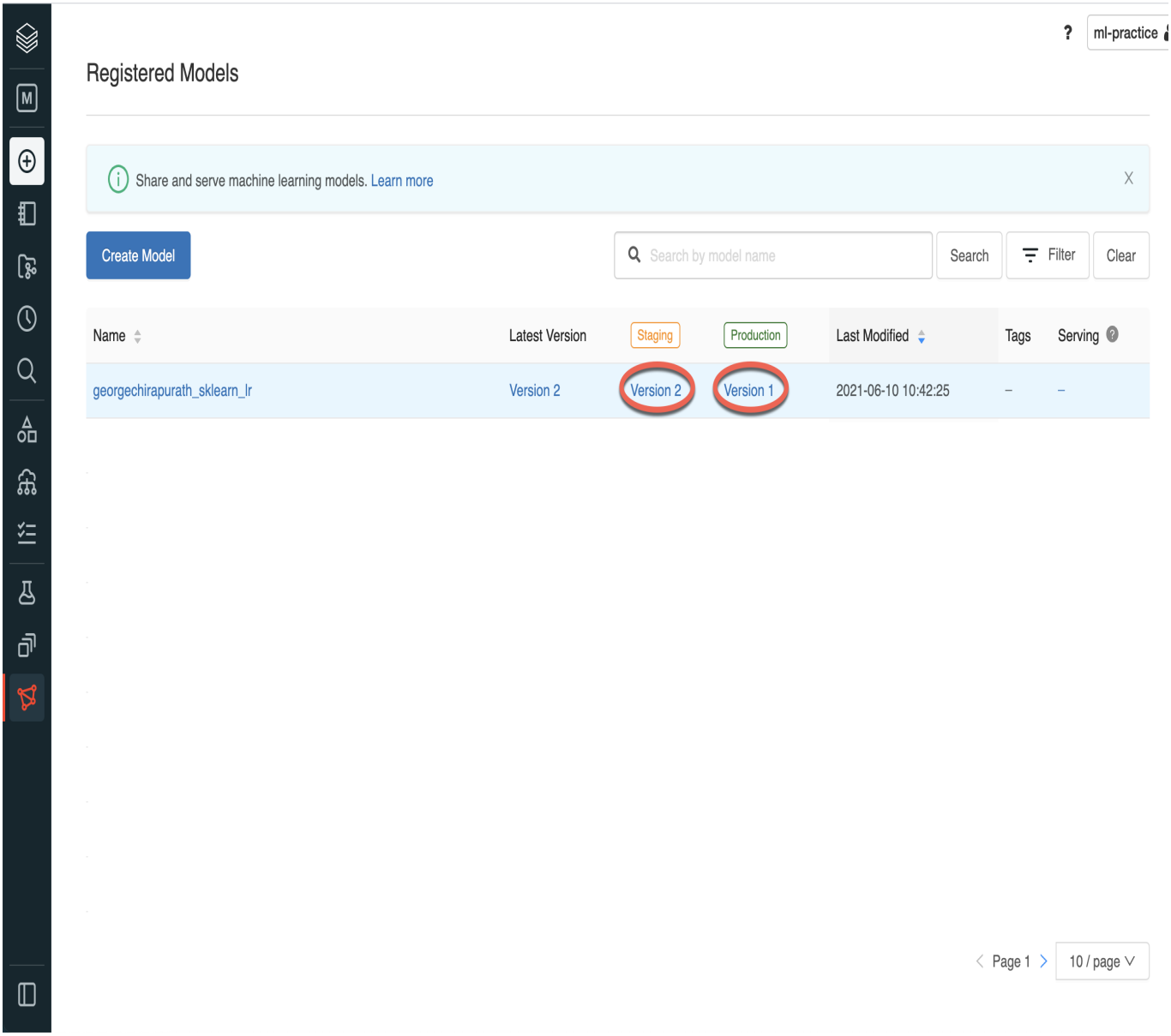
Put the new model into staging.

In [0]:

```
import time

time.sleep(10)

client.transition_model_version_stage(
    name=model_details.name,
    version=2,
    stage="Staging"
)
```

-sandbox --i18n-fe857eeb-6119-4927-ad79-77eaa7bffe3a

Check the UI to see the new model version.



--i18n-6f568dd2-0413-4b78-baf6-23debb8a5118

Use the search functionality to grab the latest model version.

In [0]:

```
model_version_infos = client.search_model_versions(f"name = '{model_name}'")
new_model_version = max([model_version_info.version for model_version_info in model
```

--i18n-4fb5d7c9-b0c0-49d5-a313-ac95da7e0f91

Add a description to this new version.

In [0]:

```
client.update_model_version(
    name=model_name,
    version=new_model_version,
    description=f"This model version is a ridge regression model with an alpha valu
)
```

--i18n-10adff21-8116-4a01-a309-ce5a7d233fcf

Since this model is now in staging, you can execute an automated CI/CD pipeline against it to test it before going into production. Once that is completed, you can push that model into production.

In [0]:

```
client.transition_model_version_stage(
    name=model_name,
    version=new_model_version,
    stage="Production",
    archive_existing_versions=True # Archieve existing model in production
)
```

--i18n-e3caaf08-a721-425b-8765-050c757d1d2e

Delete version 1.

You cannot delete a model that is not first archived.

In [0]:

```
client.delete_model_version(
    name=model_name,
    version=1
)
```

--i18n-a896f3e5-d83c-4328-821f-a67d60699f0e

Archive version 2 of the model too.

In [0]:

```
client.transition_model_version_stage(
    name=model_name,
    version=2,
    stage="Archived"
)
```

--i18n-0eb4929d-648b-4ae6-bca3-aff8af50f15f

Now delete the entire registered model.

In [0]:

```
client.delete_registered_model(model_name)
```

--i18n-6fe495ec-f481-4181-a006-bea55a6cef09

Review

Question: How does MLflow tracking differ from the model registry?
Answer: Tracking is meant for experimentation and development. The model registry is designed to take a model from tracking and put it through staging and into production. This is often the point that a data engineer or a machine learning engineer takes responsibility for the deployment process.

Question: Why do I need a model registry?
Answer: Just as MLflow tracking provides end-to-end reproducibility for the machine learning training process, a model registry provides reproducibility and governance for the deployment process. Since production systems are mission critical, components can be isolated with ACL's so only specific individuals can alter production models. Version control and CI/CD workflow integration is also a critical dimension of deploying models into production.

Question: What can I do programmatically versus using the UI?
Answer: Most operations can be done using the UI or in pure Python. A model must be tracked using Python, but from that point on everything can be done either way. For instance, a model logged using the MLflow tracking API can then be registered using the UI and can then be pushed into production.

--i18n-ecf5132e-f80d-4374-a325-28b4e96d5b61

Additional Topics & Resources

Q: Where can I find out more information on MLflow Model Registry?
A: Check out the MLflow documentation (https://mlflow.org/docs/latest/registry.html)

-sandbox

--i18n-54c3040f-38b6-4562-8dd3-61a8bb6aeba1

Time Series Forecasting

Working with time series data is an often under-represented skill in data science. In this notebook, you explore three main approaches to time series: Prophet, ARIMA, and exponential smoothing.

In this lesson you will:

- Introduce the main concepts in Time Series
- Forecast COVID data using Prophet
- Forecast using ARIMA
- Forecast using Exponential Smoothing

In this notebook we will be using the Coronavirus dataset (https://www.kaggle.com/kimjihoo/coronavirusdataset) containing data about Coronavirus patients in South Korea.

In [0]:

```
%pip install --upgrade pystan==2.19.1.1 fbprophet
```

In [0]:

```
%run ../Includes/Classroom-Setup
```

--i18n-5537f13d-b402-464f-8814-bb981709ffb2

<a href="https://en.wikipedia.org/wiki/Time_series" (https://en.wikipedia.org/wiki/Time_series"), target="_blank">Time Series

A time series is a series of data points indexed (or listed or graphed) in time order. Most commonly, a time series is a sequence taken at successive equally spaced points in time. Thus it is a sequence of discrete-time data. Examples of time series include:

- Heights of ocean tides
- Counts of sunspots
- Daily closing value of the Dow Jones Industrial Average

In this notebook, we will be focusing on time series forecasting, or, the use of a model to predict future values based on previously observed values.

In [0]:

```
file_path = f"{DA.paths.datasets}/COVID/coronavirusdataset/Time.csv"

spark_df = (spark
            .read
            .option("inferSchema", True)
            .option("header", True)
            .csv(file_path)
           )

display(spark_df)
```

--i18n-91681688-70e2-4eee-b18a-4afa353bce3f

Convert the Spark DataFrame to a Pandas DataFrame.

In [0]:

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

--i18n-920f1e35-2a54-4588-b4bf-72c6bed85e07

Looking at the data, the time column (what time of day the data was reported) is not especially relevant to our forecast, so we can go ahead and drop it.

In [0]:

```
df = df.drop(columns="time")
df.head()
```

--i18n-f5c365d6-4d8b-49a2-a8be-36aa3232d6c1

Prophet

Facebook's Prophet (https://facebook.github.io/prophet/) is widely considered the easiest way to forecast because it generally does all the heavy lifting for the user. Let's take a look at how Prophet works with our dataset.

In [0]:

```
import pandas as pd
from fbprophet import Prophet
import logging

# Suppresses `java_gateway` messages from Prophet as it runs.
logging.getLogger("py4j").setLevel(logging.ERROR)
```

--i18n-a0a43507-9db1-41da-b7bf-8d5c2f4b2a67

Prophet expects certain column names for its input DataFrame. The date column must be renamed ds, and the column to be forecast should be renamed y. Let's go ahead and forecast the number of confirmed patients in South Korea.

In [0]:

```
prophet_df = pd.DataFrame()
prophet_df["ds"] = pd.to_datetime(df["date"])
prophet_df["y"] = df["confirmed"]
prophet_df.head()
```

--i18n-daf04369-b1c3-4c84-80c1-0f7da47fc3e6

Next, let's specify how many days we want to forecast for. We can do this using the **`Prophet.make_future_dataframe`** method. With the size of our data, let's take a look at the numbers a month from now.

We can see dates up to one month in the future.

In [0]:

```
prophet_obj = Prophet()
prophet_obj.fit(prophet_df)
prophet_future = prophet_obj.make_future_dataframe(periods=30)
prophet_future.tail()
```

--i18n-b79ef0fd-8017-4d04-a1f4-f4e8f04dfc87

Finally, we can run the **`predict`** method to forecast our data points. The **`yhat`** column contains the forecasted values. You can also look at the entire DataFrame to see what other values Prophet generates.

In [0]:

```
prophet_forecast = prophet_obj.predict(prophet_future)
prophet_forecast[['ds', 'yhat']].tail()
```

--i18n-02352d36-96cb-4c11-aca6-d47a194f9942

Let's take a look at a graph representation of our forecast using **`plot`**

In [0]:

```
prophet_plot = prophet_obj.plot(prophet_forecast)
```

--i18n-d260f48c-7aaa-4cf2-8ab1-50c3a9c7318d

We can also use **`plot_components`** to get a more detailed look at our forecast.

In [0]:

```
prophet_plot2 = prophet_obj.plot_components(prophet_forecast)
```

--i18n-44baaf89-7f68-48a7-9cfd-8ed431613bfa

We can also use Prophet to identify changepoints (https://facebook.github.io/prophet/docs/trend_changepoints.html), points where the dataset had an abrupt change. This is especially useful for our dataset because it could identify time periods where Coronavirus cases spiked.

In [0]:

```
from fbprophet.plot import add_changepoints_to_plot

prophet_plot = prophet_obj.plot(prophet_forecast)
changepts = add_changepoints_to_plot(prophet_plot.gca(), prophet_obj, prophet_forec
```

In [0]:

```
print(prophet_obj.changepoints)
```

--i18n-93d9af60-11de-472f-8a26-f9a679ff29f2

Next, let's find out if there's any correlation between holidays in South Korea and increases in confirmed cases. We can use the built-in **`add_country_holidays`** method (https://facebook.github.io/prophet/docs/seasonality,_holiday_effects,_and_regressors.html#built-in-country-holidays) to find out about any trends.

You can find a complete list of country codes here (https://github.com/dr-prodigy/python-holidays/blob/master/holidays/countries/).

In [0]:

```
holidays = pd.DataFrame({"ds": [], "holiday": []})
prophet_holiday = Prophet(holidays=holidays)

prophet_holiday.add_country_holidays(country_name='KR')
prophet_holiday.fit(prophet_df)
```

--i18n-0a5f7168-3877-4457-81d4-4ceebce8ec02

You can check what holidays are included by running the following cell.

In [0]:

```
prophet_holiday.train_holiday_names
```

In [0]:

```
prophet_future = prophet_holiday.make_future_dataframe(periods=30)
prophet_forecast = prophet_holiday.predict(prophet_future)
prophet_plot_holiday = prophet_holiday.plot_components(prophet_forecast)
```

--i18n-81681565-2eb4-467c-8c5e-c6546c7230aa

ARIMA

ARIMA stands for Auto-Regressive (AR) Integrated (I) Moving Average (MA). An ARIMA model is a form of regression analysis that gauges the strength of one dependent variable relative to other changing variables.

Much like Prophet, ARIMA predicts future values based on the past values of your dataset. Unlike Prophet, ARIMA has a lot more set-up work but can be applied to a wide variety of time series.

To create our ARIMA model, we need to find the following parameters:

- **p** : The number of lag observations included in the model, also called the lag order.
- **d** : The number of times that the raw observations are differenced, also called the degree of differencing.
- **q** : The size of the moving average window, also called the order of moving average.

--i18n-99b5826a-6cf1-4bb0-a859-8cce45c50f74

Start with making our new ARIMA DataFrame. Since we already forecast the confirmed cases using Prophet, let's take a look at predictions for released patients.

In [0]:

```
arima_df = pd.DataFrame()
arima_df["date"] = pd.to_datetime(df["date"])
arima_df["released"] = df["released"]
arima_df.head()
```

--i18n-286ec724-d52c-46ff-bd35-84201bede0a5

The first step of creating an ARIMA model is to find the d-parameter by making sure your dataset is stationary. This is easy to check using an Augmented Dickey Fuller Test (https://en.wikipedia.org/wiki/Augmented_Dickey%E2%80%93Fuller_test) from the **statsmodels** library.

Since the P-value is larger than the ADF statistic, we will have to difference the dataset. Differencing helps stabilize the mean of the dataset, therefore removing the influence of past trends and seasonality on your data.

In [0]:

```
from statsmodels.tsa.stattools import adfuller
from numpy import log

result = adfuller(df.released.dropna())
print(f'ADF Statistic: {result[0]}')
print(f'p-value: {result[1]}')
```

--i18n-8766e03d-4623-40c2-a5fa-25684db75670

To difference the dataset, call **diff** on the value column. We are looking for a near-stationary series which roams around a defined mean and an ACF plot that reaches zero fairly quickly. After looking at our graphs, we can determine that our d-parameter should either be 1 or 2.

In [0]:

```
import matplotlib.pyplot as plt
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

plt.rcParams.update({"figure.figsize":(9,7), "figure.dpi":120})

# Original Series
fig, axes = plt.subplots(3, 2, sharex=True)
axes[0, 0].plot(arima_df.released); axes[0, 0].set_title('Original Series')
plot_acf(arima_df.released, ax=axes[0, 1])

# 1st Differencing
axes[1, 0].plot(arima_df.released.diff()); axes[1, 0].set_title('1st Order Differen
plot_acf(arima_df.released.diff().dropna(), ax=axes[1, 1])

# 2nd Differencing
axes[2, 0].plot(arima_df.released.diff().diff()); axes[2, 0].set_title('2nd Order D
plot_acf(arima_df.released.diff().diff().dropna(), ax=axes[2, 1])

plt.show()
```

--i18n-7e2a2b58-4516-4036-9f93-e6514c529de5

In the next section we'll find the required number of AR terms using the Partial Autocorrection Plot. This is the p-parameter.

Partial Autocorrection is the correlation between a series and its lag. From the graphs, our p-parameter should be 1.

In [0]:

```
plt.rcParams.update({"figure.figsize":(9,3), "figure.dpi":120})

fig, axes = plt.subplots(1, 2, sharex=True)
axes[0].plot(arima_df.released.diff()); axes[0].set_title('1st Differencing')
axes[1].set(ylim=(0,5))
plot_pacf(arima_df.released.diff().dropna(), ax=axes[1])

plt.show()
```

--i18n-42204bfb-229b-4695-82f1-56b34ad04ba2

Finally, we'll find the q-parameter by looking at the ACF plot to find the number of Moving Average terms. A Moving Average incorporates the dependency between an observation and a residual error applied to lagged observations. From the graphs, our q-parameter should be 1.

In [0]:

```
fig, axes = plt.subplots(1, 2, sharex=True)
axes[0].plot(arima_df.released.diff()); axes[0].set_title('1st Differencing')
axes[1].set(ylim=(0,1.2))
plot_acf(arima_df.released.diff().dropna(), ax=axes[1])

plt.show()
```

--i18n-7855a2df-0091-4a6d-b05f-538c20784b66

Once we have found our p, d, and q parameter values, we can fit our ARIMA model by passing the parameters in. The following cell shows a summary of the model including dataset information and model coefficients.

In [0]:

```
from statsmodels.tsa.arima_model import ARIMA

# p, d, q
# 1, 2, 1 ARIMA Model
model = ARIMA(arima_df.released, order=(1,2,1))
arima_fit = model.fit(disp=0)
print(arima_fit.summary())
```

--i18n-b787949f-d55a-4178-baf2-66b023567904

Finally, let's split our data into train and test data to test the accuracy of our model. Note that since we have to split the data sequentially for time series, functions like sklearn's **`train_test_split`** cannot be used here.

In [0]:

```
split_ind = int(len(arima_df)*.7)
train_df = arima_df[ :split_ind]
test_df = arima_df[split_ind: ]
#train_df.tail()
#test_df.head()
```

--i18n-5e50d9a6-321f-4577-afa3-e146ad7a38ab

To forecast we use Out of Sample Cross Validation. We can see from the graph that our forecast is slightly more linear than the actual values but overall the values are pretty close to expected.

In [0]:

```
train_model = ARIMA(train_df.released, order=(1,2,1))
train_fit = train_model.fit()

fc, se, conf = train_fit.forecast(int(len(arima_df)-split_ind))

fc_series = pd.Series(fc, index=test_df.index)

plt.plot(train_df.released, label='train', color="dodgerblue")
plt.plot(test_df.released, label='actual', color="orange")
plt.plot(fc_series, label='forecast', color="green")
plt.title('Forecast vs Actuals')
plt.ylabel("Number of Released Patients")
plt.xlabel("Day Number")
plt.legend(loc='upper left', fontsize=8)
plt.show()
```

--i18n-ce127e7b-c717-4d46-b354-77bf6a0f8dc0

Exponential Smoothing

Exponential smoothing ((https://en.wikipedia.org/wiki/Exponential_smoothing) is a rule of thumb technique for smoothing time series data using the exponential window function. Whereas in the simple moving average the past observations are weighted equally, exponential functions are used to assign exponentially decreasing weights over time. It is an easily learned and easily applied procedure for making some determination based on prior assumptions by the user, such as seasonality. Exponential smoothing is often used for analysis of time-series data.

There are three types of Exponential Smoothing:

- Single Exponential Smoothing (SES)
  - Used for datasets without trends or seasonality.
- Double Exponential Smoothing (also known as Holt's Linear Smoothing)
  - Used for datasets with trends but without seasonality.
- Triple Exponential Smoothing (also known as Holt-Winters Exponential Smoothing)
  - Used for datasets with both trends and seasonality.

In our case, the Coronavirus dataset has a clear trend, but seasonality is not especially important, therefore we will be using double exponential smoothing.

--i18n-294eaf9b-8ba0-4137-bb52-d27b39f3d34f

Since we have already forecast the other two columns, let's take a look at a forecast for the number of coronavirus related deaths.

In [0]:

```
exp_df = pd.DataFrame()
exp_df["date"] = pd.to_datetime(df["date"])
exp_df["deceased"] = df["deceased"]
exp_df.head()
```

--i18n-7ddb8a5e-453b-491d-ac0d-3087a2c7f955

Holt's Linear Smoothing only works on data points that are greater than 0, therefore we have to drop the corresponding rows. Additionally, we need to set the index of our DataFrame to the date column.

In [0]:

```
exp_df = exp_df[exp_df["deceased"] != 0]
exp_df = exp_df.set_index("date")
exp_df.head()
```

--i18n-b4fb0e59-5554-44b0-81b3-efbfeca88e33

Luckily, statsmodel does most of the work for us. However, we still have to tweak the parameters to get an accurate forecast. The available parameters here are α or **`smoothing_level`** and β or **`smoothing_slope`** . α defines the smoothing factor of the level and β defines the smoothing factor of the trend.

In the cell below, we are trying three different kinds of predictions. The first, Holt's Linear Trend, forecasts with a linear trend. The second, Exponential Trend, forecasts with an exponential trend. The third, Additive Damped Trend, damps the forecast trend linearly.

In [0]:

```
from statsmodels.tsa.holtwinters import Holt

exp_fit1 = Holt(exp_df.deceased).fit(smoothing_level=0.8, smoothing_slope=0.2, opti
exp_forecast1 = exp_fit1.forecast(30).rename("Holt's linear trend")

exp_fit2 = Holt(exp_df.deceased, exponential=True).fit(smoothing_level=0.8, smoothi
exp_forecast2 = exp_fit2.forecast(30).rename("Exponential trend")

exp_fit3 = Holt(exp_df.deceased, damped=True).fit(smoothing_level=0.8, smoothing_sl
exp_forecast3 = exp_fit3.forecast(30).rename("Additive damped trend")
```

--i18n-4e3c58f7-afdc-42c6-805f-6f34770ea4d8

After plotting the three models, we can see that the standard Holt's Linear, and the Exponential trend lines give very similar forecasts while the Additive Damped trend gives a slightly lower number of deceased patients.

In [0]:

```
exp_fit1.fittedvalues.plot(color="orange", label="Holt's linear trend")
exp_fit2.fittedvalues.plot(color="red", label="Exponential trend")
exp_fit3.fittedvalues.plot(color="green", label="Additive damped trend")

plt.legend()
plt.ylabel("Number of Deceased Patients")
plt.xlabel("Day Number")
plt.show()
```

--i18n-f3bbd647-3586-482f-880a-369258cfc7d0

We can zoom in on the forecast part of our graph to see the graph in more detail.

We can see that the exponential trendline starts in-line with the linear trendline but slowly starts resembling an exponential trend towards the end of the graph. The damped trendline starts and ends below the other trendlines.

In [0]:

```
exp_forecast1.plot(legend=True, color="orange")
exp_forecast2.plot(legend=True, color="red")
exp_forecast3.plot(legend=True, color="green")

plt.ylabel("Number of Deceased Patients")
plt.xlabel("Day Number")
plt.show()
```

-sandbox

--i18n-b9944704-a562-44e0-8ef6-8639f11312ca

XGBoost

Up until this point, we have only used SparkML. Let's look a third party library for Gradient Boosted Trees.

Ensure that you are using the Databricks Runtime for ML (https://docs.microsoft.com/en-us/azure/databricks/runtime/mlruntime) because that has Distributed XGBoost already installed.

Question: How do gradient boosted trees differ from random forests? Which parts can be parallelized?

In this lesson you:

- Use 3rd party libraries (XGBoost) to further improve your model

In [0]:

```
%run "./Includes/Classroom-Setup"
```

--i18n-3e08ca45-9a00-4c6a-ac38-169c7e87d9e4

Data Preparation

Let's go ahead and index all of our categorical features, and set our label to be **`log(price)`** .

In [0]:

```
from pyspark.sql.functions import log, col
from pyspark.ml.feature import StringIndexer, VectorAssembler

file_path = f"{DA.paths.datasets}/airbnb/sf-listings/sf-listings-2019-03-06-clean.d
airbnb_df = spark.read.format("delta").load(file_path)
train_df, test_df = airbnb_df.withColumn("label", log(col("price"))).randomSplit([.

categorical_cols = [field for (field, dataType) in train_df.dtypes if dataType == "
index_output_cols = [x + "Index" for x in categorical_cols]

string_indexer = StringIndexer(inputCols=categorical_cols, outputCols=index_output_

numeric_cols = [field for (field, dataType) in train_df.dtypes if ((dataType == "do
assembler_inputs = index_output_cols + numeric_cols
vec_assembler = VectorAssembler(inputCols=assembler_inputs, outputCol="features")
```

--i18n-733cd880-143d-42c2-9f29-602e48f60efe

Pyspark Distributed XGBoost

Let's create our distributed XGBoost model. While technically not part of MLlib, you can integrate XGBoost (https://databricks.github.io/spark-deep-learning/_modules/sparkdl/xgboost/xgboost.html) into your ML Pipelines.

To use the distributed version of Pyspark XGBoost you can specify two additional parameters:

- **`num_workers`** : The number of workers to distribute over. Requires MLR 9.0+.
- **`use_gpu`** : Enable to utilize GPU based training for faster performance (optional).

NOTE: **`use_gpu`** requires an ML GPU runtime. Currently, at most one GPU per worker will be used when doing distributed training.

In [0]:

```
from sparkdl.xgboost import XgboostRegressor
from pyspark.ml import Pipeline

params = {"n_estimators": 100, "learning_rate": 0.1, "max_depth": 4, "random_state"

xgboost = XgboostRegressor(**params)

pipeline = Pipeline(stages=[string_indexer, vec_assembler, xgboost])
pipeline_model = pipeline.fit(train_df)
```

--i18n-8d5f8c24-ee0b-476e-a250-95ce2d73dd28

Evaluate

Now we can evaluate how well our XGBoost model performed. Don't forget to exponentiate!

In [0]:

```
from pyspark.sql.functions import exp, col

log_pred_df = pipeline_model.transform(test_df)

exp_xgboost_df = log_pred_df.withColumn("prediction", exp(col("prediction")))

display(exp_xgboost_df.select("price", "prediction"))
```

--i18n-364402e1-8073-4b24-8e03-c7e2566f94d2

Compute some metrics.

In [0]:

```
from pyspark.ml.evaluation import RegressionEvaluator

regression_evaluator = RegressionEvaluator(predictionCol="prediction", labelCol="pr

rmse = regression_evaluator.evaluate(exp_xgboost_df)
r2 = regression_evaluator.setMetricName("r2").evaluate(exp_xgboost_df)
print(f"RMSE is {rmse}")
print(f"R2 is {r2}")
```

--i18n-21cf0d1b-c7a8-43c0-8eea-7677bb0d7847

Alternative Gradient Boosted Approaches

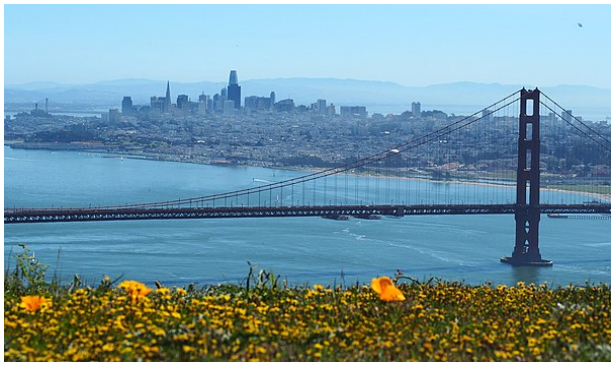
There are lots of other gradient boosted approaches, such as CatBoost (https://catboost.ai/), LightGBM (https://github.com/microsoft/LightGBM), vanilla gradient boosted trees in SparkML (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.classification.GBTClassifier.html?highlight=gbt#pyspark.ml.classification.GBTClassifier)/scikit-learn (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html), etc. Each of these has their respective pros and cons (https://towardsdatascience.com/catboost-vs-light-gbm-vs-xgboost-5f93620723db) that you can read more about.

-sandbox

--i18n-8c6d3ef3-e44b-4292-a0d3-1aaba0198525

Data Cleansing

We will be using Spark to do some exploratory data analysis & cleansing of the SF Airbnb rental dataset from [Inside Airbnb (http://insideairbnb.com/get-the-data.html)](http://insideairbnb.com/get-the-data.html).



In this lesson you:

- Impute missing values
- Identify & remove outliers

In [0]:

```
%run "./Includes/Classroom-Setup"
```

--i18n-969507ea-bffc-4255-9a99-2306a594625f

Let's load the Airbnb dataset in.

In [0]:

```
file_path = f"{DA.paths.datasets}/airbnb/sf-listings/sf-listings-2019-03-06.csv"

raw_df = spark.read.csv(file_path, header="true", inferSchema="true", multiLine="tr

display(raw_df)
```

In [0]:

```
raw_df.columns
```

--i18n-94856418-c319-4915-a73e-5728fcd44101

For the sake of simplicity, only keep certain columns from this dataset. We will talk about feature selection later.

In [0]:

```
columns_to_keep = [
    "host_is_superhost",
    "cancellation_policy",
    "instant_bookable",
    "host_total_listings_count",
    "neighbourhood_cleansed",
    "latitude",
    "longitude",
    "property_type",
    "room_type",
    "accommodates",
    "bathrooms",
    "bedrooms",
    "beds",
    "bed_type",
    "minimum_nights",
    "number_of_reviews",
    "review_scores_rating",
    "review_scores_accuracy",
    "review_scores_cleanliness",
    "review_scores_checkin",
    "review_scores_communication",
    "review_scores_location",
    "review_scores_value",
    "price"
]

base_df = raw_df.select(columns_to_keep)
base_df.cache().count()
display(base_df)
```

--i18n-a12c5a59-ad1c-4542-8695-d822ec10c4ca

Fixing Data Types

Take a look at the schema above. You'll notice that the **`price`** field got picked up as string. For our task, we need it to be a numeric (double type) field.

Let's fix that.

In [0]:

```
from pyspark.sql.functions import col, translate

fixed_price_df = base_df.withColumn("price", translate(col("price"), "$,", "").cast

display(fixed_price_df)
```

--i18n-4ad08138-4563-4a93-b038-801832c9bc73

Summary statistics

Two options:

- **`describe`** : count, mean, stddev, min, max
- **`summary`** : describe + interquartile range (IQR)

Question: When to use IQR/median over mean? Vice versa?

In [0]:

```
display(fixed_price_df.describe())
```

In [0]:

```
display(fixed_price_df.summary())
```

--i18n-bd55efda-86d0-4584-a6fc-ef4f221b2872

Dbutils Data Summary

We can also use **`dbutils.data.summarize`** to see more detailed summary statistics and data plots.

In [0]:

```
dbutils.data.summarize(fixed_price_df)
```

--i18n-e9860f92-2fbe-4d23-b728-678a7bb4734e

Getting rid of extreme values

Let's take a look at the *min* and *max* values of the **`price`** column.

In [0]:

```
display(fixed_price_df.select("price").describe())
```

--i18n-4a8fe21b-1dac-4edf-a0a3-204f170b05c9

There are some super-expensive listings, but it's up to the SME (Subject Matter Experts) to decide what to do with them. We can certainly filter the "free" Airbnbs though.

Let's see first how many listings we can find where the *price* is zero.

In [0]:

```
fixed_price_df.filter(col("price") == 0).count()
```

--i18n-bf195d9b-ea4d-4a3e-8b61-372be8eec327

Now only keep rows with a strictly positive *price*.

In [0]:

```
pos_prices_df = fixed_price_df.filter(col("price") > 0)
```

--i18n-dc8600db-ebd1-4110-bfb1-ce555bc95245

Let's take a look at the *min* and *max* values of the *minimum_nights* column:

In [0]:

```
display(pos_prices_df.select("minimum_nights").describe())
```

In [0]:

```
display(pos_prices_df
        .groupBy("minimum_nights").count()
        .orderBy(col("count").desc(), col("minimum_nights"))
       )
```

--i18n-5aa4dfa8-d9a1-42e2-9060-a5dcc3513a0d

A minimum stay of one year seems to be a reasonable limit here. Let's filter out those records where the *minimum_nights* is greater then 365.

In [0]:

```
min_nights_df = pos_prices_df.filter(col("minimum_nights") <= 365)

display(min_nights_df)
```

--i18n-25a35390-d716-43ad-8f51-7e7690e1c913

Handling Null Values

There are a lot of different ways to handle null values. Sometimes, null can actually be a key indicator of the thing you are trying to predict (e.g. if you don't fill in certain portions of a form, probability of it getting approved decreases).

Some ways to handle nulls:

- Drop any records that contain nulls
- Numeric:
  - Replace them with mean/median/zero/etc.
- Categorical:

    - Replace them with the mode
    - Create a special category for null
- Use techniques like ALS (Alternating Least Squares) which are designed to impute missing values

If you do ANY imputation techniques for categorical/numerical features, you MUST include an additional field specifying that field was imputed.

SparkML's Imputer (covered below) does not support imputation for categorical features.

--i18n-83e56fca-ce6d-4e3c-8042-0c1c7b9eaa5a

Impute: Cast to Double

SparkML's Imputer (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.Imputer.html?highlight=imputer#pyspark.ml.feature.Imputer) requires all fields be of type double. Let's cast all integer fields to double.

In [0]:

```
from pyspark.sql.functions import col
from pyspark.sql.types import IntegerType

integer_columns = [x.name for x in min_nights_df.schema.fields if x.dataType == Int
doubles_df = min_nights_df

for c in integer_columns:
    doubles_df = doubles_df.withColumn(c, col(c).cast("double"))

columns = "\n - ".join(integer_columns)
print(f"Columns converted from Integer to Double:\n - {columns}")
```

--i18n-69b58107-82ad-4cec-8984-028a5df1b69e

Add a dummy column to denote presence of null values before imputing.

In [0]:

```
from pyspark.sql.functions import when

impute_cols = [
    "bedrooms",
    "bathrooms",
    "beds",
    "review_scores_rating",
    "review_scores_accuracy",
    "review_scores_cleanliness",
    "review_scores_checkin",
    "review_scores_communication",
    "review_scores_location",
    "review_scores_value"
]

for c in impute_cols:
    doubles_df = doubles_df.withColumn(c + "_na", when(col(c).isNull(), 1.0).otherw
```

In [0]:

```
display(doubles_df.describe())
```

--i18n-c88f432d-1252-4acc-8c91-4834c00da789

Transformers and Estimators

Spark ML standardizes APIs for machine learning algorithms to make it easier to combine multiple algorithms into a single pipeline, or workflow. Let's cover two key concepts introduced by the Spark ML API: **`transformers`** and **`estimators`**.

Transformer: Transforms one DataFrame into another DataFrame. It accepts a DataFrame as input, and returns a new DataFrame with one or more columns appended to it. Transformers do not learn any parameters from your data and simply apply rule-based transformations. It has a **`.transform()`** method.

Estimator: An algorithm which can be fit on a DataFrame to produce a Transformer. E.g., a learning algorithm is an Estimator which trains on a DataFrame and produces a model. It has a **`.fit()`** method because it learns (or "fits") parameters from your DataFrame.

In [0]:

```
from pyspark.ml.feature import Imputer

imputer = Imputer(strategy="median", inputCols=impute_cols, outputCols=impute_cols)

imputer_model = imputer.fit(doubles_df)
imputed_df = imputer_model.transform(doubles_df)
```

--i18n-4df06e83-27e6-4cc6-b66d-883317b2a7eb

OK, our data is cleansed now. Let's save this DataFrame to Delta so that we can start building models with it.

In [0]:

```
imputed_df.write.format("delta").mode("overwrite").save(f"{DA.paths.working_dir}/im
```

-sandbox

--i18n-263caa08-bb08-4022-8d8f-bd2f51d77752

Classification: Logistic Regression

Up until this point, we have only examined regression use cases. Now let's take a look at how to handle classification.

For this lab, we will use the same Airbnb dataset, but instead of predicting price, we will predict if host is a superhost (https://www.airbnb.com/superhost) or not in San Francisco.

In this lesson you:

- Build a Logistic Regression model
- Use various metrics to evaluate model performance

In [0]:

```
%run "../Includes/Classroom-Setup"
```

In [0]:

```
file_path = f"{DA.paths.datasets}/airbnb/sf-listings/sf-listings-2019-03-06-clean.d
airbnb_df = spark.read.format("delta").load(file_path)
```

--i18n-3f07e772-c15d-46e4-8acd-866b661fbb9b

Baseline Model

Before we build any Machine Learning models, we want to build a baseline model to compare to. We are going to start by predicting if a host is a superhost (https://www.airbnb.com/superhost).

For our baseline model, we are going to predict no on is a superhost and evaluate our accuracy. We will examine other metrics later as we build more complex models.

0. Convert our **`host_is_superhost`** column (t/f) into 1/0 and call the resulting column **`label`** . DROP the **`host_is_superhost`** afterwards.
1. Add a column to the resulting DataFrame called **`prediction`** which contains the literal value **`0.0`** . We will make a constant prediction that no one is a superhost.

After we finish these two steps, then we can evaluate the "model" accuracy.

Some helpful functions:

- when() (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.functions.when.html#pyspark.sql.fu
- withColumn() (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.DataFrame.withColumn.html?highlight=withcolumn#pyspark.sql.DataFrame.withColumn)
- lit() (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.functions.lit.html?highlight=lit#pyspark.sql.functions.lit)

In [0]:

```
# ANSWER

from pyspark.sql.functions import when, col, lit

label_df = airbnb_df.select(when(col("host_is_superhost") == "t", 1.0).otherwise(0.

pred_df = label_df.withColumn("prediction", lit(0.0))
```

--i18n-d04eb817-2010-4021-a898-42ca8abaa00d

Evaluate model

For right now, let's use accuracy as our metric. This is available from MulticlassClassificationEvaluator (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.evaluation.MulticlassClassificationEvalu highlight=multiclassclassificationevaluator#pyspark.ml.evaluation.MulticlassClassificationEvaluator).

In [0]:

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

mc_evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
print(f"The accuracy is {100*mc_evaluator.evaluate(pred_df):.2f}%")
```

--i18n-5fe00f31-d186-4ab8-b6bb-437f7ddc4a00

Train-Test Split

Alright! Now we have built a baseline model. The next step is to split our data into a train-test split.

In [0]:

```
train_df, test_df = label_df.randomSplit([.8, .2], seed=42)
print(train_df.cache().count())
```

--i18n-a7998d44-af91-4dfa-b80c-8b96ebfe5311

Visualize

Let's look at the relationship between **`review_scores_rating`** and **`label`** in our training dataset.

In [0]:

```
display(train_df.select("review_scores_rating", "label"))
```

--i18n-1ce4ba05-f558-484d-a8e8-53bde1e119fc

Logistic Regression

Now build a [logistic regression model (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.classification.LogisticRegression.html?highlight=logisticregression#pyspark.ml.classification.LogisticRegression)](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.classification.LogisticRegression.html?highlight=logisticregression#pyspark.ml.classification.LogisticRegression) using all of the features (HINT: use RFormula). Put the pre-processing step and the Logistic Regression Model into a Pipeline.

In [0]:

```
# ANSWER
from pyspark.ml import Pipeline
from pyspark.ml.feature import RFormula
from pyspark.ml.classification import LogisticRegression

r_formula = RFormula(formula="label ~ .",
                     featuresCol="features",
                     labelCol="label",
                     handleInvalid="skip") # Look at handleInvalid

lr = LogisticRegression(labelCol="label", featuresCol="features")
pipeline = Pipeline(stages=[r_formula, lr])
pipeline_model = pipeline.fit(train_df)
pred_df = pipeline_model.transform(test_df)
```

--i18n-3a06d71c-8551-44c8-b33e-8ae40a443713

Evaluate

What is AUROC useful for? Try adding additional evaluation metrics, like Area Under PR Curve.

In [0]:

```
# ANSWER
from pyspark.ml.evaluation import BinaryClassificationEvaluator, MulticlassClassifi

mc_evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
print(f"The accuracy is {100*mc_evaluator.evaluate(pred_df):.2f}%")

bc_evaluator = BinaryClassificationEvaluator(metricName="areaUnderROC")
print(f"The area under the ROC curve: {bc_evaluator.evaluate(pred_df):.2f}")

bc_evaluator.setMetricName("areaUnderPR")
print(f"The area under the PR curve: {bc_evaluator.evaluate(pred_df):.2f}")
```

--i18n-0ef0e2b9-6ce9-4377-8587-83b5260fd05a

Add Hyperparameter Tuning

Try changing the hyperparameters of the logistic regression model using the cross-validator. By how much can you improve your metrics?

In [0]:

```
# ANSWER
from pyspark.ml.tuning import ParamGridBuilder
from pyspark.ml.tuning import CrossValidator

param_grid = (ParamGridBuilder()
            .addGrid(lr.regParam, [0.1, 0.2])
            .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])
            .build())

cv = CrossValidator(estimator=lr, evaluator=mc_evaluator, estimatorParamMaps=param_
                    numFolds=3, parallelism=4, seed=42)

pipeline = Pipeline(stages=[r_formula, cv])

pipeline_model = pipeline.fit(train_df)

pred_df = pipeline_model.transform(test_df)
```

--i18n-111f2dc7-5535-45b7-82f6-ad2e5f2cbf16

Evaluate again

In [0]:

```
mc_evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
print(f"The accuracy is {100*mc_evaluator.evaluate(pred_df):.2f}%")

bc_evaluator = BinaryClassificationEvaluator(metricName="areaUnderROC")
print(f"The area under the ROC curve: {bc_evaluator.evaluate(pred_df):.2f}")
```

--i18n-7e88e044-0a34-4815-8eab-1dc37532a082

Super Bonus

Try using MLflow to track your experiments!

-sandbox

--i18n-1fa7a9c8-3dad-454e-b7ac-555020a4bda8

Hyperopt

Hyperopt is a Python library for "serial and parallel optimization over awkward search spaces, which may include real-valued, discrete, and conditional dimensions".

In the machine learning workflow, hyperopt can be used to distribute/parallelize the hyperparameter optimization process with more advanced optimization strategies than are available in other libraries.

There are two ways to scale hyperopt with Apache Spark:

- Use single-machine hyperopt with a distributed training algorithm (e.g. MLlib)
- Use distributed hyperopt with single-machine training algorithms (e.g. scikit-learn) with the SparkTrials class.

In this lesson, we will use single-machine hyperopt with MLlib, but in the lab, you will see how to use hyperopt to distribute the hyperparameter tuning of single node models.

Unfortunately you can't use hyperopt to distribute the hyperparameter optimization for distributed training algorithms at this time. However, you do still get the benefit of using more advanced hyperparameter search algorthims (random search, TPE, etc.) with Spark ML.

Resources:

0. Documentation (http://hyperopt.github.io/hyperopt/scaleout/spark/)
1. Hyperopt on Databricks (https://docs.databricks.com/applications/machine-learning/automl/hyperopt/index.html)
2. Hyperparameter Tuning with MLflow, Apache Spark MLlib and Hyperopt (https://databricks.com/blog/2019/06/07/hyperparameter-tuning-with-mlflow-apache-spark-mllib-and-hyperopt.html)
3. How (Not) to Tune Your Model With Hyperopt (https://databricks.com/blog/2021/04/15/how-not-to-tune-your-model-with-hyperopt.html)

In this lesson you:

- Use hyperopt to find the optimal parameters for an MLlib model using TPE

In [0]:

```
%run "./Includes/Classroom-Setup"
```

--i18n-2340cdf4-9753-41b4-a613-043b90f0f472

Let's start by loading in our SF Airbnb Dataset.

In [0]:

```
file_path = f"{DA.paths.datasets}/airbnb/sf-listings/sf-listings-2019-03-06-clean.d
airbnb_df = spark.read.format("delta").load(file_path)
train_df, val_df, test_df = airbnb_df.randomSplit([.6, .2, .2], seed=42)
```

--i18n-37bbd5bd-f330-4d02-8af6-1b185612cdf8

We will then create our random forest pipeline and regression evaluator.

In [0]:

```
from pyspark.ml.feature import StringIndexer, VectorAssembler
from pyspark.ml import Pipeline
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.evaluation import RegressionEvaluator

categorical_cols = [field for (field, dataType) in train_df.dtypes if dataType == "
index_output_cols = [x + "Index" for x in categorical_cols]

string_indexer = StringIndexer(inputCols=categorical_cols, outputCols=index_output_

numeric_cols = [field for (field, dataType) in train_df.dtypes if ((dataType == "do
assembler_inputs = index_output_cols + numeric_cols
vec_assembler = VectorAssembler(inputCols=assembler_inputs, outputCol="features")

rf = RandomForestRegressor(labelCol="price", maxBins=40, seed=42)
pipeline = Pipeline(stages=[string_indexer, vec_assembler, rf])
regression_evaluator = RegressionEvaluator(predictionCol="prediction", labelCol="pr
```

--i18n-e4627900-f2a5-4f65-881e-1374187dd4f9

Next, we get to the hyperopt-specific part of the workflow.

First, we define our **objective function**. The objective function has two primary requirements:

1. An **input `params`** including hyperparameter values to use when training the model
2. An **output** containing a loss metric on which to optimize

In this case, we are specifying values of **`max_depth`** and **`num_trees`** and returning the RMSE as our loss metric.

We are reconstructing our pipeline for the **`RandomForestRegressor`** to use the specified hyperparameter values.

In [0]:

```
def objective_function(params):
    # set the hyperparameters that we want to tune
    max_depth = params["max_depth"]
    num_trees = params["num_trees"]

    with mlflow.start_run():
        estimator = pipeline.copy({rf.maxDepth: max_depth, rf.numTrees: num_trees})
        model = estimator.fit(train_df)

        preds = model.transform(val_df)
        rmse = regression_evaluator.evaluate(preds)
        mlflow.log_metric("rmse", rmse)

    return rmse
```

--i18n-d4f9dd2b-060b-4eef-8164-442b2be242f4

Next, we define our search space.

This is similar to the parameter grid in a grid search process. However, we are only specifying the range of values rather than the individual, specific values to be tested. It's up to hyperopt's optimization algorithm to choose the actual values.

See the documentation (https://github.com/hyperopt/hyperopt/wiki/FMin) for helpful tips on defining your search space.

In [0]:

```
from hyperopt import hp

search_space = {
    "max_depth": hp.quniform("max_depth", 2, 5, 1),
    "num_trees": hp.quniform("num_trees", 10, 100, 1)
}
```

--i18n-27891521-e481-4734-b21c-b2c5fe1f01fe

`fmin()` generates new hyperparameter configurations to use for your **`objective_function`**. It will evaluate 4 models in total, using the information from the previous models to make a more informative decision for the the next hyperparameter to try.

Hyperopt allows for parallel hyperparameter tuning using either random search or Tree of Parzen Estimators (TPE). Note that in the cell below, we are importing **`tpe`**. According to the documentation (http://hyperopt.github.io/hyperopt/scaleout/spark/), TPE is an adaptive algorithm that

> iteratively explores the hyperparameter space. Each new hyperparameter setting tested will be chosen based on previous results.

Hence, **`tpe.suggest`** is a Bayesian method.

MLflow also integrates with Hyperopt, so you can track the results of all the models you've trained and their results as part of your hyperparameter tuning. Notice you can track the MLflow experiment in this notebook, but you can also specify an external experiment.

In [0]:

```
from hyperopt import fmin, tpe, Trials
import numpy as np
import mlflow
import mlflow.spark
mlflow.pyspark.ml.autolog(log_models=False)

num_evals = 4
trials = Trials()
best_hyperparam = fmin(fn=objective_function,
                       space=search_space,
                       algo=tpe.suggest,
                       max_evals=num_evals,
                       trials=trials,
                       rstate=np.random.default_rng(42))

# Retrain model on train & validation dataset and evaluate on test dataset
with mlflow.start_run():
    best_max_depth = best_hyperparam["max_depth"]
    best_num_trees = best_hyperparam["num_trees"]
    estimator = pipeline.copy({rf.maxDepth: best_max_depth, rf.numTrees: best_num_t
    combined_df = train_df.union(val_df) # Combine train & validation together

    pipeline_model = estimator.fit(combined_df)
    pred_df = pipeline_model.transform(test_df)
    rmse = regression_evaluator.evaluate(pred_df)

    # Log param and metrics for the final model
    mlflow.log_param("maxDepth", best_max_depth)
    mlflow.log_param("numTrees", best_num_trees)
    mlflow.log_metric("rmse", rmse)
    mlflow.spark.log_model(pipeline_model, "model")
```

-sandbox

--i18n-3bdc2b9e-9f58-4cb7-8c55-22bade9f79df

Decision Trees

In the previous notebook, you were working with the parametric model, Linear Regression. We could do some more hyperparameter tuning with the linear regression model, but we're going to try tree based methods and see if our performance improves.

In this lesson you:

- Identify the differences between single node and distributed decision tree implementations
- Get the feature importance
- Examine common pitfalls of decision trees

In [0]:

```
%run "./Includes/Classroom-Setup"
```

In [0]:

```
file_path = f"{DA.paths.datasets}/airbnb/sf-listings/sf-listings-2019-03-06-clean.d
airbnb_df = spark.read.format("delta").load(file_path)
train_df, test_df = airbnb_df.randomSplit([.8, .2], seed=42)
```

--i18n-9af16c65-168c-4078-985d-c5f8991f171f

How to Handle Categorical Features?

We saw in the previous notebook that we can use StringIndexer/OneHotEncoder/VectorAssembler or RFormula.

However, for decision trees, and in particular, random forests, we should not OHE our variables.

There is an excellent blog (https://towardsdatascience.com/one-hot-encoding-is-making-your-tree-based-ensembles-worse-heres-why-d64b282b5769#:~:text=One%2Dhot%20encoding%20categorical%20variables,importance%20resulting%20in%2 on this, and the essence is:

> "One-hot encoding categorical variables with high cardinality can cause inefficiency in tree-based methods. Continuous variables will be given more importance than the dummy variables by the algorithm, which will obscure the order of feature importance and can result in poorer performance."

In [0]:

```
from pyspark.ml.feature import StringIndexer

categorical_cols = [field for (field, dataType) in train_df.dtypes if dataType == "
index_output_cols = [x + "Index" for x in categorical_cols]

string_indexer = StringIndexer(inputCols=categorical_cols, outputCols=index_output_
```

--i18n-35e2f231-2ebb-4889-bc55-089200dd1605

VectorAssembler

Let's use the VectorAssembler (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.VectorAssembler.html?highlight=vectorassembler#pyspark.ml.feature.VectorAssembler) to combine all of our categorical and numeric inputs.

In [0]:

```
from pyspark.ml.feature import VectorAssembler

# Filter for just numeric columns (and exclude price, our label)
numeric_cols = [field for (field, dataType) in train_df.dtypes if ((dataType == "do
# Combine output of StringIndexer defined above and numeric columns
assembler_inputs = index_output_cols + numeric_cols
vec_assembler = VectorAssembler(inputCols=assembler_inputs, outputCol="features")
```

--i18n-2096f7aa-7fab-4807-b45f-fcbd0424a3e8

Decision Tree

Now let's build a DecisionTreeRegressor (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.regression.DecisionTreeRegressor.html?highlight=decisiontreeregressor#pyspark.ml.regression.DecisionTreeRegressor) with the default hyperparameters.

In [0]:

```
from pyspark.ml.regression import DecisionTreeRegressor

dt = DecisionTreeRegressor(labelCol="price")
```

--i18n-506ab7fa-0952-4c55-ad9b-afefb6469380

Fit Pipeline

The following cell is expected to error, but we subsequently fix this.

In [0]:

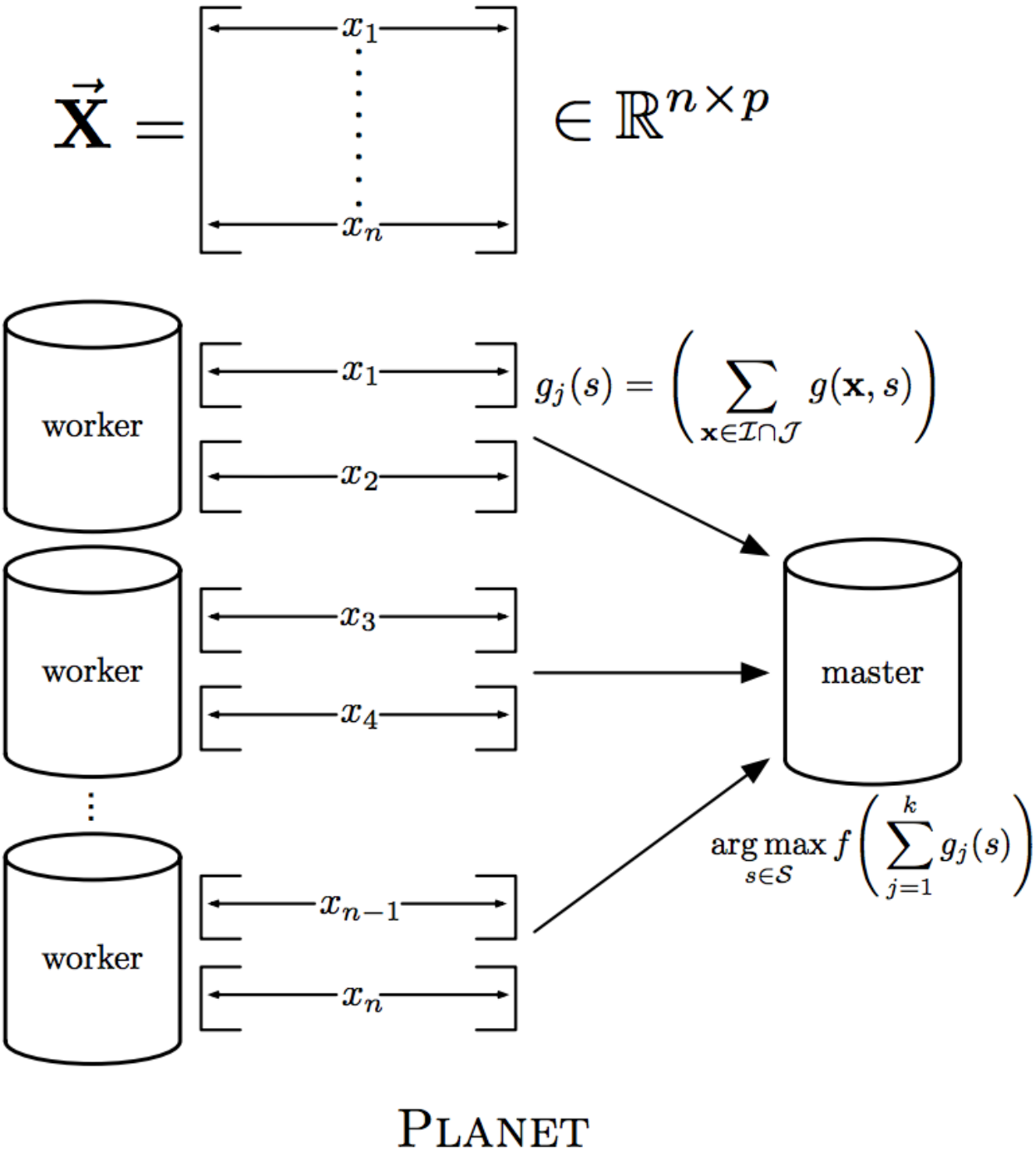
```
from pyspark.ml import Pipeline

# Combine stages into pipeline
stages = [string_indexer, vec_assembler, dt]
pipeline = Pipeline(stages=stages)

# Uncomment to perform fit
# pipeline_model = pipeline.fit(train_df)
```

--i18n-d0791ff8-8d79-4d32-937d-9fcfbac4e9bd

maxBins

What is this parameter maxBins (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.regression.DecisionTreeRegressor.html highlight=decisiontreeregressor#pyspark.ml.regression.DecisionTreeRegressor.maxBins)? Let's take a look at the PLANET implementation of distributed decision trees to help explain the **`maxBins`** parameter.

--i18n-1f9c229e-6f8c-4174-9927-c284e64e5753



--i18n-3b7e60c3-22de-4794-9cd4-6713255b79a4

In Spark, data is partitioned by row. So when it needs to make a split, each worker has to compute summary statistics for every feature for each split point. Then these summary statistics have to be aggregated (via tree reduce) for a split to be made.

Think about it: What if worker 1 had the value **32** but none of the others had it. How could you communicate how good of a split that would be? So, Spark has a maxBins parameter for discretizing continuous variables into buckets, but the number of buckets has to be as large as the categorical variable with the highest cardinality.

--i18n-0552ed6a-120f-4e49-ae3a-5f92bd9f863d

Let's go ahead and increase maxBins to **40** .

In [0]:

```
dt.setMaxBins(40)
```

--i18n-92252524-e388-439b-a92b-958cc332a861

Take two.

In [0]:

```
pipeline_model = pipeline.fit(train_df)
```

--i18n-2426e78b-9bd2-4b7d-a65b-52054906e438

Feature Importance

Let's go ahead and get the fitted decision tree model, and look at the feature importance scores.

In [0]:

```
dt_model = pipeline_model.stages[-1]
display(dt_model)
```

In [0]:

```
dt_model.featureImportances
```

--i18n-823c20ff-f20b-4853-beb0-4b324debb2e6

Interpreting Feature Importance

Hmmm... it's a little hard to know what feature 4 vs 11 is. Given that the feature importance scores are "small data", let's use Pandas to help us recover the original column names.

In [0]:

```
import pandas as pd

features_df = pd.DataFrame(list(zip(vec_assembler.getInputCols(), dt_model.featureI
features_df
```

--i18n-1fe0f603-add5-4904-964b-7288ae98b2e8

Why so few features are non-zero?

With SparkML, the default **`maxDepth`** is 5, so there are only a few features we could consider (we can also split on the same feature many times at different split points).

Let's use a Databricks widget to get the top-K features.

In [0]:

```
dbutils.widgets.text("top_k", "5")
top_k = int(dbutils.widgets.get("top_k"))

top_features = features_df.sort_values(["importance"], ascending=False)[:top_k]["fe
print(top_features)
```

--i18n-d9525bf7-b871-45c8-b0f9-dca5fd7ae825

Scale Invariant

With decision trees, the scale of the features does not matter. For example, it will split 1/3 of the data if that split point is 100 or if it is normalized to be .33. The only thing that matters is how many data points fall left and right of that split point - not the absolute value of the split point.

This is not true for linear regression, and the default in Spark is to standardize first. Think about it: If you measure shoe sizes in American vs European sizing, the corresponding weight of those features will be very different even those those measures represent the same thing: the size of a person's foot!

--i18n-bad0dd6d-05ba-484b-90d6-cfe16a1bc11e

Apply model to test set

In [0]:

```
pred_df = pipeline_model.transform(test_df)

display(pred_df.select("features", "price", "prediction").orderBy("price", ascendin
```

--i18n-094553a3-10c0-4e08-9a58-f94430b4a512

Pitfall

What if we get a massive Airbnb rental? It was 20 bedrooms and 20 bathrooms. What will a decision tree predict?

It turns out decision trees cannot predict any values larger than they were trained on. The max value in our training set was $10,000, so we can't predict any values larger than that.

In [0]:

```
from pyspark.ml.evaluation import RegressionEvaluator

regression_evaluator = RegressionEvaluator(predictionCol="prediction", labelCol="pr

rmse = regression_evaluator.evaluate(pred_df)
r2 = regression_evaluator.setMetricName("r2").evaluate(pred_df)
print(f"RMSE is {rmse}")
print(f"R2 is {r2}")
```

--i18n-033a9c19-0f9d-4c33-aa5e-f58665637448

Uh oh!

This model is way worse than the linear regression model, and it's even worse than just predicting the average value.

In the next few notebooks, let's look at hyperparameter tuning and ensemble models to improve upon the performance of our single decision tree.

-sandbox

--i18n-94727771-3f7d-41a7-bcbd-774b1fc5837c

Distributed K-Means

In this notebook, we are going to use K-Means to cluster our data. We will be using the Iris dataset, which has labels (the type of iris), but we will only use the labels to evaluate the model, not to train it.

At the end, we will look at how it is implemented in the distributed setting.

In this lesson you:

- Build a K-Means model
- Analyze the computation and communication of K-Means in a distributed setting

In [0]:

```
from sklearn.datasets import load_iris
import pandas as pd

# Load in a Dataset from sklearn and convert to a Spark DataFrame
iris = load_iris()
iris_pd = pd.concat([pd.DataFrame(iris.data, columns=iris.feature_names), pd.DataFr
iris_df = spark.createDataFrame(iris_pd)
display(iris_df)
```

--i18n-efd06e75-816c-4ab5-84b5-dd1da377fa01

Notice that we have four values as "features". We'll reduce those down to two values (for visualization purposes) and convert them to a **`DenseVector`** . To do that we'll use the **`VectorAssembler`** .

In [0]:

```
from pyspark.ml.feature import VectorAssembler

vec_assembler = VectorAssembler(inputCols=["sepal length (cm)", "sepal width (cm)"]
iris_two_features_df = vec_assembler.transform(iris_df)
display(iris_two_features_df)
```

In [0]:

```
from pyspark.ml.clustering import KMeans

kmeans = KMeans(k=3, seed=221, maxIter=20)

#  Call fit on the estimator and pass in iris_two_features_df
model = kmeans.fit(iris_two_features_df)

# Obtain the clusterCenters from the KMeansModel
centers = model.clusterCenters()

# Use the model to transform the DataFrame by adding cluster predictions
transformed_df = model.transform(iris_two_features_df)

print(centers)
```

In [0]:

```
model_centers = []
iterations = [0, 2, 4, 7, 10, 20]
for i in iterations:
    kmeans = KMeans(k=3, seed=221, maxIter=i)
    model = kmeans.fit(iris_two_features_df)
    model_centers.append(model.clusterCenters())
```

In [0]:

```
print("model_centers:")
for centroids in model_centers:
    print(centroids)
```

--i18n-840acc4b-58f7-439d-afe7-5a70d5718dc1

Let's visualize how our clustering performed against the true labels of our data.

Remember: K-means doesn't use the true labels when training, but we can use them to evaluate.

Here, the star marks the cluster center.

In [0]:

```
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np

def prepare_subplot(xticks, yticks, figsize=(10.5, 6), hideLabels=False, gridColor=
    """Template for generating the plot layout."""
    fig, ax_list = plt.subplots(subplots[0], subplots[1], figsize=figsize, facecolo
                                edgecolor="white")
    if not isinstance(ax_list, np.ndarray):
        ax_list = np.array([ax_list])

    for ax in ax_list.flatten():
        ax.axes.tick_params(labelcolor="#999999", labelsize="10")
        for axis, ticks in [(ax.get_xaxis(), xticks), (ax.get_yaxis(), yticks)]:
            axis.set_ticks_position("none")
            axis.set_ticks(ticks)
            axis.label.set_color("#999999")
            if hideLabels: axis.set_ticklabels([])
        ax.grid(color=gridColor, linewidth=gridWidth, linestyle="-")
        map(lambda position: ax.spines[position].set_visible(False), ["bottom", "to

    if ax_list.size == 1:
        ax_list = ax_list[0]  # Just return a single axes object for a regular plot
    return fig, ax_list
```

In [0]:

```
data = iris_two_features_df.select("features", "label").collect()
features, labels = zip(*data)

x, y = zip(*features)
centers = model_centers[5]
centroid_x, centroid_y = zip(*centers)
color_map = "Set1"

fig, ax = prepare_subplot(np.arange(-1, 1.1, .4), np.arange(-1, 1.1, .4), figsize=(
plt.scatter(x, y, s=14**2, c=labels, edgecolors="#8cbfd0", alpha=0.80, cmap=color_m
plt.scatter(centroid_x, centroid_y, s=22**2, marker="*", c="yellow")
cmap = cm.get_cmap(color_map)

color_index = [.5, .99, .0]
for i, (x,y) in enumerate(centers):
    print(cmap(color_index[i]))
    for size in [.10, .20, .30, .40, .50]:
        circle1=plt.Circle((x,y), size, color=cmap(color_index[i]), alpha=.10, line
        ax.add_artist(circle1)

ax.set_xlabel("Sepal Length"), ax.set_ylabel("Sepal Width")
fig
```

--i18n-b5b5d89a-1595-4e0c-99a1-54209435cf81

In addition to seeing the overlay of the clusters at each iteration, we can see how the cluster centers moved with each iteration (and what our results would have looked like if we used fewer iterations).

In [0]:

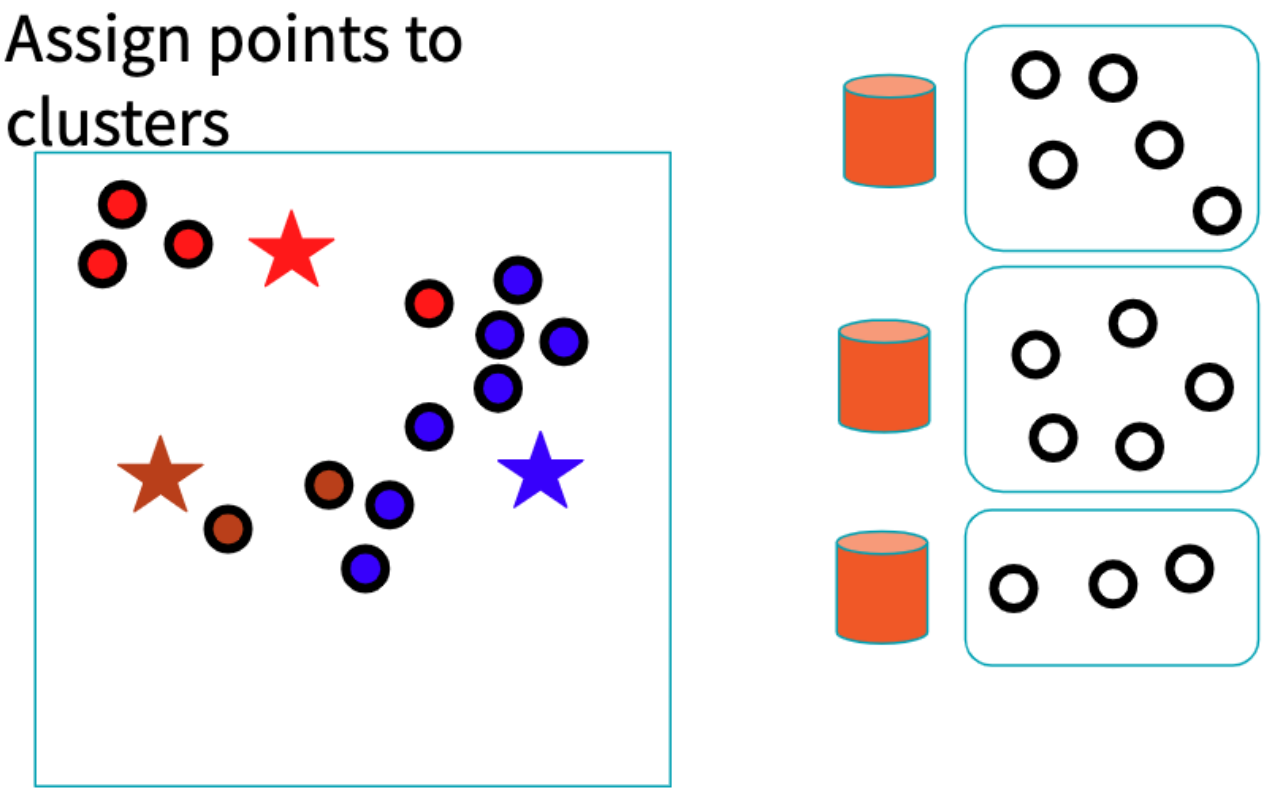
```
x, y = zip(*features)

old_centroid_x, old_centroid_y = None, None

fig, ax_list = prepare_subplot(np.arange(-1, 1.1, .4), np.arange(-1, 1.1, .4), figs
                               subplots=(3, 2))
ax_list = ax_list.flatten()

for i,ax in enumerate(ax_list[:]):
    ax.set_title("K-means for {0} iterations".format(iterations[i]), color="#999999
    centroids = model_centers[i]
    centroid_x, centroid_y = zip(*centroids)

    ax.scatter(x, y, s=10**2, c=labels, edgecolors="#8cbfd0", alpha=0.80, cmap=colo
    ax.scatter(centroid_x, centroid_y, s=16**2, marker="*", c="yellow", zorder=2)
    if old_centroid_x and old_centroid_y:
      ax.scatter(old_centroid_x, old_centroid_y, s=16**2, marker="*", c="grey", zor
    cmap = cm.get_cmap(color_map)

    color_index = [.5, .99, 0.]
    for i, (x1,y1) in enumerate(centroids):
      print(cmap(color_index[i]))
      circle1=plt.Circle((x1,y1),.35,color=cmap(color_index[i]), alpha=.40)
      ax.add_artist(circle1)

    ax.set_xlabel("Sepal Length"), ax.set_ylabel("Sepal Width")
    old_centroid_x, old_centroid_y = centroid_x, centroid_y

plt.tight_layout()

fig
```

--i18n-06e7d08a-e824-435d-9835-adc29bd5c12e

So let's take a look at what's happening here in the distributed setting.

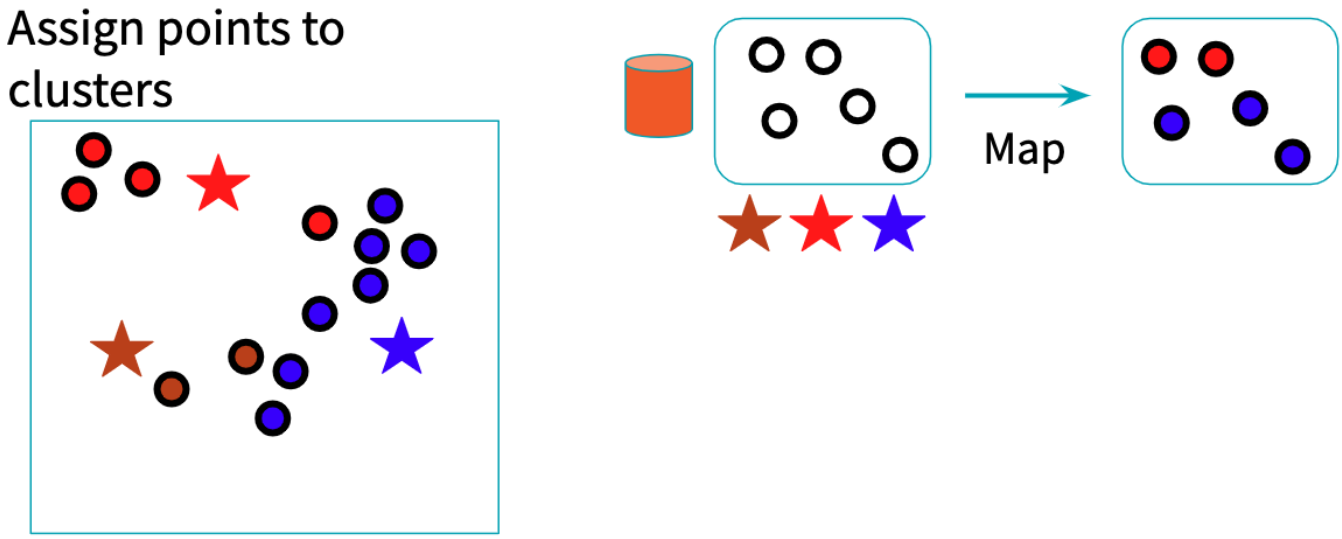
--i18n-edc1d38d-5cc3-4bf5-bfc5-bb85a145bb16

Map stage



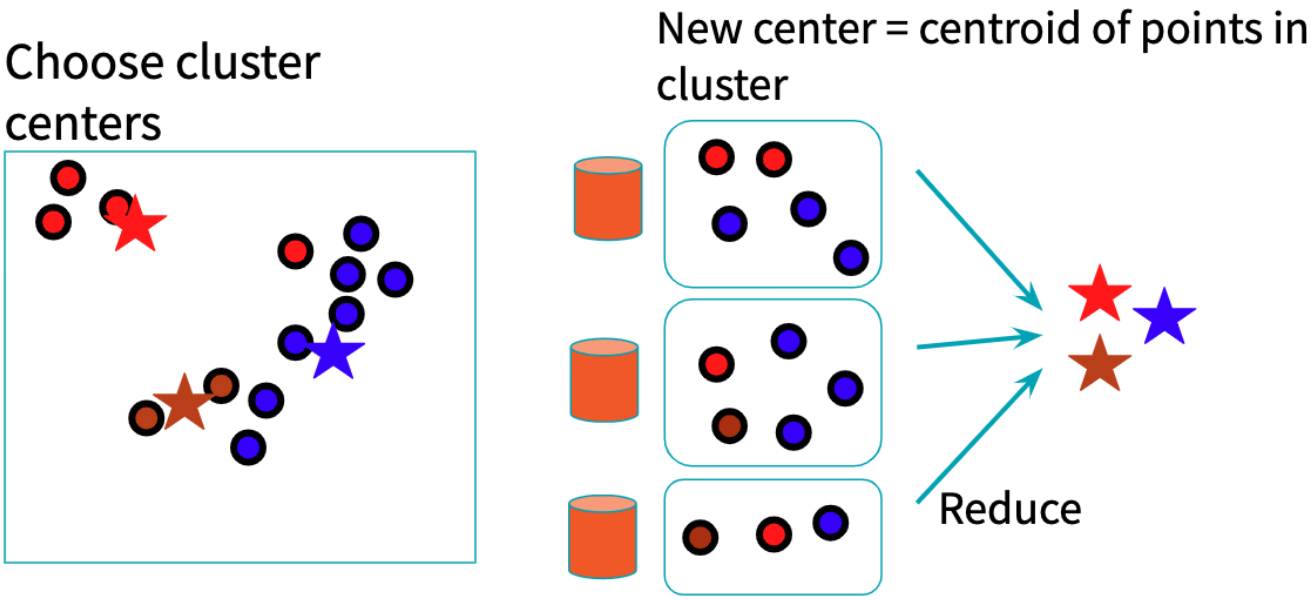
--i18n-aa078ae4-fbfd-4dc2-b0cb-92bc10714981

Map stage



--i18n-9cf17004-1750-49fe-bb92-ce38c54c1ced

Reduce stage



--i18n-80c66031-e786-404e-8c77-c90a91fa3f4a

Communication

Map: Assign points to clusters
→ Send k cluster centers to P workers
Reduce: Choose cluster centers
→ Aggregate k cluster centers from n points
→ But can aggregate within each worker first!

Centers & points are d-dimensional vectors

Total communication: O(kdP)

Communication does not depend on # points n! ☺

Communication depends on:
- # workers P ☹
  → Parallelized via tree aggregation ☺
- # dimensions d
  → OK since K-Means can fail in high-d
- # clusters k
  → Hard to avoid

--i18n-e0f585c1-1d13-4f8c-b9ae-dfd184547653

Take Aways

When designing/choosing distributed ML algorithms

- Communication is key!

- Consider your data/model dimensions & how much data you need.
- Data partitioning/organization is important.

-sandbox

--i18n-64f90be2-bcc0-40c9-bbe7-3b501323e71c

Databricks Best Practices

In this notebook, we will explore a wide array of best practices for working with Databricks.

In this lesson you:

- Explore a general framework for debugging slow running jobs
- Identify the security implications of various data access paradigms
- Determine various cluster configuration issues including machine types, libraries, and jobs
- Integrate Databricks notebooks and jobs with version control and the CLI

--i18n-0c063b0a-ccbb-486f-8568-1fe52cfa8971

Slow Running Jobs

The most common issues with slow running jobs are:

- **`Spill`** : Data is exhausting the cluster's memory and is spilling onto disk. Resolution: a cluster with more memory resources
- **`Shuffle`** : Large amounts of data are being transferred across the cluster. Resolution: optimize joins or refactor code to avoid shuffles
- **`Skew/Stragglers`** : Partitioned data (in files or in memory) is skewed causing the "curse of the last reducer" where some partitions take longer to run. Resolution: repartition to a multiple of the available cores or use skew hints
- **`Small/Large Files`** : Too many small files are exhausting cluster resources since each file read needs its own thread or few large files are causing unused threads. Resolution: rewrite data in a more optimized way or perform Delta file compaction

Your debugging toolkit:

- Ganglia for CPU, network, and memory resources at a cluster or node level
- Spark UI for most everything else (especially the storage and executor tabs)
- Driver or worker logs for errors (especially with background processes)
- Notebook tab of the clusters section to see if the intern is hogging your cluster again

--i18n-35f52f8a-6a95-4273-8e04-ead835c2c184

Data Access and Security

A few notes on data access:

- Mount data for easy access (https://docs.databricks.com/data/databricks-file-system.html#mount-storage)
- Use secrets to secure credentials (https://docs.databricks.com/dev-tools/cli/secrets-cli.html#secrets-cli) (this keeps credentials out of the code)
- Credential passthrough works in AWS (https://docs.databricks.com/dev-tools/cli/secrets-cli.html#secrets-cli) and Azure (https://docs.microsoft.com/en-us/azure/databricks/security/credential-passthrough/adls-passthrough)

--i18n-2c6e2b76-709f-43e9-9fd2-731713fe30a7

Cluster Configuration, Libraries, and Jobs

Cluster types are:

- Memory optimized (with or without Delta Cache Acceleration (https://docs.databricks.com/delta/optimizations/delta-cache.html)
- Compute optimized
- Storage optimized
- GPU accelerated
- General Purpose

General rules of thumb:

- Smaller clusters of larger machine types for machine learning
- One cluster per production workload
- Don't share clusters for ML training (even in development)
- See the docs for more specifics (https://docs.databricks.com/clusters/configure.html)

--i18n-6368d08e-4f54-4504-8a83-5e099c7aeb34

Library installation best practices:

- Notebook-scoped Python libraries (https://docs.databricks.com/libraries/notebooks-python-libraries.html) ensure users on same cluster can have different libraries. Also good for saving notebooks with their library dependencies
- Init scripts (https://docs.databricks.com/clusters/init-scripts.html) ensure that code is ran before the JVM starts (good for certain libraries or environment configuration)
- Some configuration variables need to be set on cluster start

--i18n-dd0026c2-92e2-4761-9308-75ad353649d4

Jobs best practices:

- Use notebook workflows (https://docs.databricks.com/notebooks/notebook-workflows.html)
- Widgets (https://docs.databricks.com/notebooks/widgets.html) work for parameter passing
- You can also run jars and wheels
- Use the CLI for orchestration tools (e.g. Airflow)
- See the docs for more specifics (https://docs.databricks.com/jobs.html)
- Always specify a timeout interval to prevent infinitely running jobs

--i18n-ea44ac8c-88c8-443a-a370-b4671af6f1e9

CLI and Version Control

The Databricks CLI (https://github.com/databricks/databricks-cli):

- Programmatically export out all your notebooks to check into github
- Can also import/export data, execute jobs, create clusters, and perform most other Workspace tasks

Git integration can be accomplished in a few ways:

- Use the CLI to import/export notebooks and check into git manually
- Use the built-in git integration (https://docs.databricks.com/notebooks/github-version-control.html)
- Use the next generation workspace for alternative project integration (https://www.youtube.com/watch?v=HsfMmBfQtvI)

--i18n-4bbc8017-a03b-4b3e-810f-9375e5afd7e2

Time permitting: exploring the admin console! (https://docs.databricks.com/administration-guide/index.html)

-sandbox

--i18n-60a5d18a-6438-4ee3-9097-5145dc31d938

Linear Regression: Improving our model

In this notebook we will be adding additional features to our model, as well as discuss how to handle categorical features.

In this lesson you:

- One Hot Encode categorical variables
- Use the Pipeline API
- Save and load models

In [0]:

```
%run "./Includes/Classroom-Setup"
```

In [0]:

```
file_path = f"{DA.paths.datasets}/airbnb/sf-listings/sf-listings-2019-03-06-clean.d
airbnb_df = spark.read.format("delta").load(file_path)
```

--i18n-f8b3c675-f8ce-4339-865e-9c64f05291a6

Train/Test Split

Let's use the same 80/20 split with the same seed as the previous notebook so we can compare our results apples to apples (unless you changed the cluster config!)

In [0]:

```
train_df, test_df = airbnb_df.randomSplit([.8, .2], seed=42)
```

--i18n-09003d63-70c1-4fb7-a4b7-306101a88ae3

Categorical Variables

There are a few ways to handle categorical features:

- Assign them a numeric value
- Create "dummy" variables (also known as One Hot Encoding)
- Generate embeddings (mainly used for textual data)

One Hot Encoder

Here, we are going to One Hot Encode (OHE) our categorical variables. Spark doesn't have a **`dummies`** function, and OHE is a two step process. First, we need to use StringIndexer (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.StringIndexer.html?highlight=stringindexer#pyspark.ml.feature.StringIndexer) to map a string column of labels to an ML column of label indices.

Then, we can apply the OneHotEncoder (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.OneHotEncoder.html?highlight=onehotencoder#pyspark.ml.feature.OneHotEncoder) to the output of the StringIndexer.

In [0]:

```
from pyspark.ml.feature import OneHotEncoder, StringIndexer

categorical_cols = [field for (field, dataType) in train_df.dtypes if dataType == "
index_output_cols = [x + "Index" for x in categorical_cols]
ohe_output_cols = [x + "OHE" for x in categorical_cols]

string_indexer = StringIndexer(inputCols=categorical_cols, outputCols=index_output_
ohe_encoder = OneHotEncoder(inputCols=index_output_cols, outputCols=ohe_output_cols
```

--i18n-dedd7980-1c27-4f35-9d94-b0f1a1f92839

Vector Assembler

Now we can combine our OHE categorical features with our numeric features.

In [0]:

```
from pyspark.ml.feature import VectorAssembler

numeric_cols = [field for (field, dataType) in train_df.dtypes if ((dataType == "do
assembler_inputs = ohe_output_cols + numeric_cols
vec_assembler = VectorAssembler(inputCols=assembler_inputs, outputCol="features")
```

--i18n-fb06fb9b-5dac-46df-aff3-ddee6dc88125

Linear Regression

Now that we have all of our features, let's build a linear regression model.

In [0]:

```
from pyspark.ml.regression import LinearRegression

lr = LinearRegression(labelCol="price", featuresCol="features")
```

--i18n-a7aabdd1-b384-45fc-bff2-f385cc7fe4ac

Pipeline

Let's put all these stages in a Pipeline. A Pipeline (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.Pipeline.html?highlight=pipeline#pyspark.ml.Pipeline) is a way of organizing all of our transformers and estimators.

This way, we don't have to worry about remembering the same ordering of transformations to apply to our test dataset.

In [0]:

```
from pyspark.ml import Pipeline

stages = [string_indexer, ohe_encoder, vec_assembler, lr]
pipeline = Pipeline(stages=stages)

pipeline_model = pipeline.fit(train_df)
```

--i18n-c7420125-24be-464f-b609-1bb4e765d4ff

Saving Models

We can save our models to persistent storage (e.g. DBFS) in case our cluster goes down so we don't have to recompute our results.

In [0]:

```
pipeline_model.write().overwrite().save(DA.paths.working_dir)
```

--i18n-15f4623d-d99a-42d6-bee8-d7c4f79fdecb

Loading models

When you load in models, you need to know the type of model you are loading back in (was it a linear regression or logistic regression model?).

For this reason, we recommend you always put your transformers/estimators into a Pipeline, so you can always load the generic PipelineModel back in.

In [0]:

```
from pyspark.ml import PipelineModel

saved_pipeline_model = PipelineModel.load(DA.paths.working_dir)
```

--i18n-1303ef7d-1a57-4573-8afe-561f7730eb33

Apply model to test set

In [0]:

```
pred_df = saved_pipeline_model.transform(test_df)

display(pred_df.select("features", "price", "prediction"))
```

--i18n-9497f680-1c61-4bf1-8ab4-e36af502268d

Evaluate model

How is our R2 doing?

In [0]:

```
from pyspark.ml.evaluation import RegressionEvaluator

regression_evaluator = RegressionEvaluator(predictionCol="prediction", labelCol="pr

rmse = regression_evaluator.evaluate(pred_df)
r2 = regression_evaluator.setMetricName("r2").evaluate(pred_df)
print(f"RMSE is {rmse}")
print(f"R2 is {r2}")
```

--i18n-cc0618e0-59d9-4a6d-bb90-a7945da1457e

As you can see, our RMSE decreased when compared to the model without one-hot encoding, and the R2 increased as well!

-sandbox

--i18n-2ab084da-06ed-457d-834a-1d19353e5c59

Random Forests and Hyperparameter Tuning

Now let's take a look at how to tune random forests using grid search and cross validation in order to find the optimal hyperparameters. Using the Databricks Runtime for ML, MLflow automatically logs your experiments with the SparkML cross-validator!

In this lesson you:

- Tune hyperparameters using Grid Search
- Optimize a SparkML pipeline

In [0]:

```
%run "./Includes/Classroom-Setup"
```

In [0]:

```
from pyspark.ml.feature import StringIndexer, VectorAssembler
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml import Pipeline

file_path = f"{DA.paths.datasets}/airbnb/sf-listings/sf-listings-2019-03-06-clean.d
airbnb_df = spark.read.format("delta").load(file_path)
train_df, test_df = airbnb_df.randomSplit([.8, .2], seed=42)

categorical_cols = [field for (field, dataType) in train_df.dtypes if dataType == "
index_output_cols = [x + "Index" for x in categorical_cols]

string_indexer = StringIndexer(inputCols=categorical_cols, outputCols=index_output_

numeric_cols = [field for (field, dataType) in train_df.dtypes if ((dataType == "do
assembler_inputs = index_output_cols + numeric_cols
vec_assembler = VectorAssembler(inputCols=assembler_inputs, outputCol="features")

rf = RandomForestRegressor(labelCol="price", maxBins=40)
stages = [string_indexer, vec_assembler, rf]
pipeline = Pipeline(stages=stages)
```

--i18n-4561938e-90b5-413c-9e25-ef15ba40e99c

ParamGrid

First let's take a look at the various hyperparameters we could tune for random forest.

Pop quiz: what's the difference between a parameter and a hyperparameter?

In [0]:

```
print(rf.explainParams())
```

--i18n-819de6f9-75d2-45df-beb1-6b59ecd2cfd2

There are a lot of hyperparameters we could tune, and it would take a long time to manually configure.

Instead of a manual (ad-hoc) approach, let's use Spark's ParamGridBuilder (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.tuning.ParamGridBuilder.html?highlight=paramgridbuilder#pyspark.ml.tuning.ParamGridBuilder) to find the optimal hyperparameters in a more systematic approach.

Let's define a grid of hyperparameters to test:

- **`maxDepth`** : max depth of each decision tree (Use the values **`2, 5`**)
- **`numTrees`** : number of decision trees to train (Use the values **`5, 10`**)

`addGrid()` accepts the name of the parameter (e.g. **`rf.maxDepth`**), and a list of the possible values (e.g. **`[2, 5]`**).

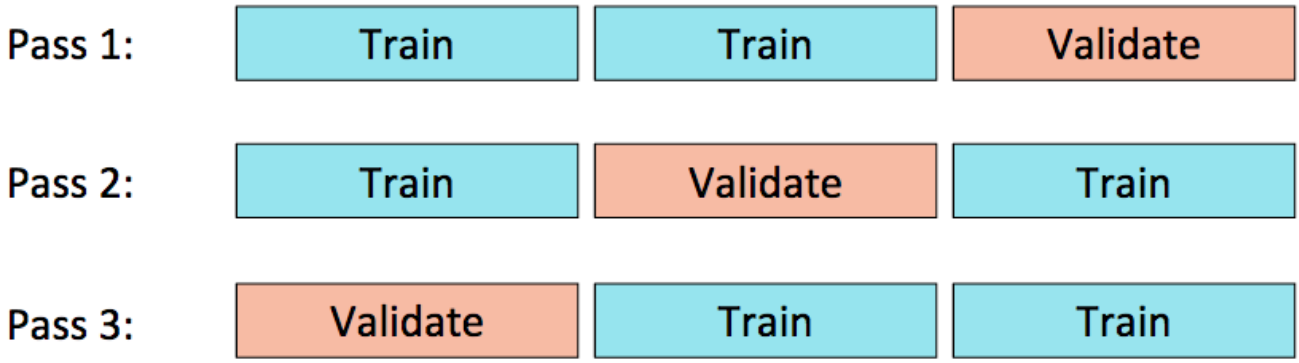
In [0]:

```
from pyspark.ml.tuning import ParamGridBuilder

param_grid = (ParamGridBuilder()
              .addGrid(rf.maxDepth, [2, 5])
              .addGrid(rf.numTrees, [5, 10])
              .build())
```

--i18n-9f043287-11b8-482d-8501-2f7d8b1458ea

Cross Validation

We are also going to use 3-fold cross validation to identify the optimal hyperparameters.



With 3-fold cross-validation, we train on 2/3 of the data, and evaluate with the remaining (held-out) 1/3. We repeat this process 3 times, so each fold gets the chance to act as the validation set. We then average the results of the three rounds.

--i18n-ec0440ab-071d-4201-be86-5eeedaf80a4f

We pass in the **`estimator`** (pipeline), **`evaluator`** , and **`estimatorParamMaps`** to CrossValidator (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.tuning.CrossValidator.html?highlight=crossvalidator#pyspark.ml.tuning.CrossValidator) so that it knows:

- Which model to use
- How to evaluate the model
- What hyperparameters to set for the model

We can also set the number of folds we want to split our data into (3), as well as setting a seed so we all have the same split in the data.

In [0]:

```
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.tuning import CrossValidator

evaluator = RegressionEvaluator(labelCol="price", predictionCol="prediction")

cv = CrossValidator(estimator=pipeline, evaluator=evaluator, estimatorParamMaps=par
                    numFolds=3, seed=42)
```

--i18n-673c9261-a861-4ace-b008-c04565230a8e

Question: How many models are we training right now?

In [0]:

```
cv_model = cv.fit(train_df)
```

--i18n-c9bc1596-7b0f-4595-942c-109cfca51698

Parallelism Parameter

Hmmm... that took a long time to run. That's because the models were being trained sequentially rather than in parallel!

In Spark 2.3, a parallelism (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.tuning.CrossValidator.html?highlight=crossvalidator#pyspark.ml.tuning.CrossValidator.parallelism) parameter was introduced. From the docs: **the number of threads to use when running parallel algorithms (>= 1)** .

Let's set this value to 4 and see if we can train any faster. The Spark docs (https://spark.apache.org/docs/latest/ml-tuning.html) recommend a value between 2-10.

In [0]:

```
cv_model = cv.setParallelism(4).fit(train_df)
```

--i18n-2d00b40f-c5e7-4089-890b-a50ccced34c6

Question: Hmmm... that still took a long time to run. Should we put the pipeline in the cross validator, or the cross validator in the pipeline?

It depends if there are estimators or transformers in the pipeline. If you have things like StringIndexer (an estimator) in the pipeline, then you have to refit it every time if you put the entire pipeline in the cross validator.

However, if there is any concern about data leakage from the earlier steps, the safest thing is to put the pipeline inside the CV, not the other way. CV first splits the data and then .fit() the pipeline. If it is placed at the end of the pipeline, we potentially can leak the info from hold-out set to train set.

In [0]:

```
cv = CrossValidator(estimator=rf, evaluator=evaluator, estimatorParamMaps=param_gri
                    numFolds=3, parallelism=4, seed=42)

stages_with_cv = [string_indexer, vec_assembler, cv]
pipeline = Pipeline(stages=stages_with_cv)

pipeline_model = pipeline.fit(train_df)
```

--i18n-dede990c-2551-4c07-8aad-d697ae827e71

Let's take a look at the model with the best hyperparameter configuration

In [0]:

```
list(zip(cv_model.getEstimatorParamMaps(), cv_model.avgMetrics))
```

In [0]:

```
pred_df = pipeline_model.transform(test_df)

rmse = evaluator.evaluate(pred_df)
r2 = evaluator.setMetricName("r2").evaluate(pred_df)
print(f"RMSE is {rmse}")
print(f"R2 is {r2}")
```

--i18n-8f80daf2-8f0b-4cab-a8e6-4060c78d94b0

Progress! Looks like we're out-performing decision trees.

-sandbox

--i18n-2b5dc285-0d50-4ea7-a71b-8a7aa355ad7c

Inference with Pandas UDFs

In this lesson you:

- Build a scikit-learn model, track it with MLflow, and apply it at scale using the Pandas Scalar Iterator UDFs and **`mapInPandas()`**

To learn more about Pandas UDFs, you can refer to this blog post (https://databricks.com/blog/2020/05/20/new-pandas-udfs-and-python-type-hints-in-the-upcoming-release-of-apache-spark-3-0.html) to see what's new in Spark 3.0.

In [0]:

```
%run ./Includes/Classroom-Setup
```

--i18n-8b52bca0-45f0-4ada-be31-c2c473fb8e77

Train sklearn model and log it with MLflow

In [0]:

```
import mlflow.sklearn
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split

with mlflow.start_run(run_name="sklearn-random-forest") as run:
    # Enable autologging
    mlflow.sklearn.autolog(log_input_examples=True, log_model_signatures=True, log_
    # Import the data
    df = pd.read_csv(f"{DA.paths.datasets}/airbnb/sf-listings/airbnb-cleaned-mlflow
    X_train, X_test, y_train, y_test = train_test_split(df.drop(["price"], axis=1),

    # Create model
    rf = RandomForestRegressor(n_estimators=100, max_depth=10, random_state=42)
    rf.fit(X_train, y_train)
```

--i18n-7ebcaaf9-c6f5-4c92-865a-c7f2c7afb555

Create Spark DataFrame

In [0]:

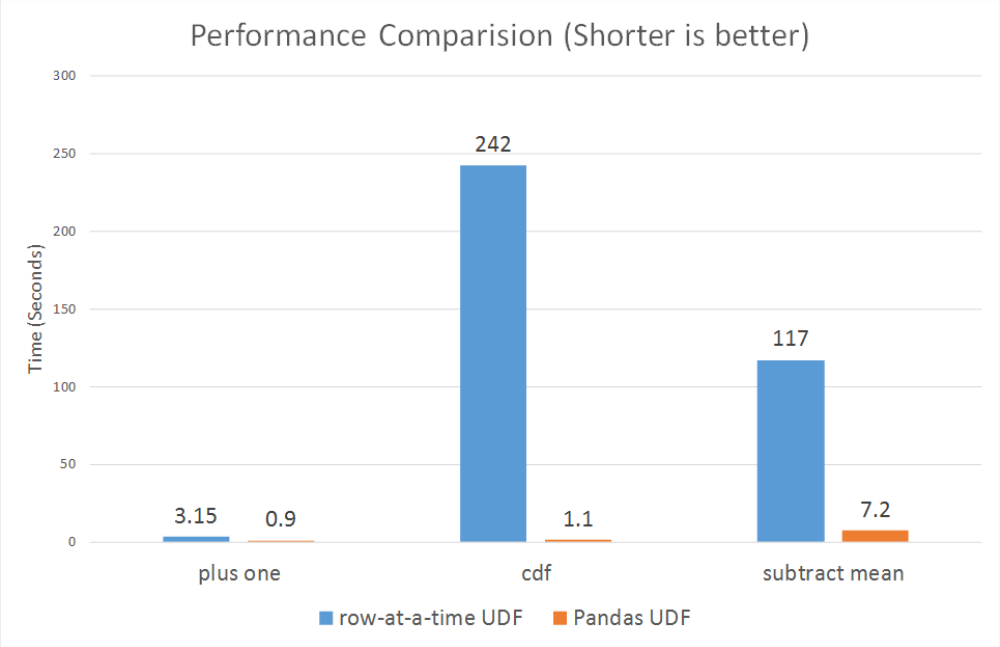
```
spark_df = spark.createDataFrame(X_test)
```

--i18n-1cdc4475-f55f-4126-9d38-dedb19577f4e

Pandas/Vectorized UDFs

As of Spark 2.3, there are Pandas UDFs available in Python to improve the efficiency of UDFs. Pandas UDFs utilize Apache Arrow to speed up computation. Let's see how that helps improve our processing time.

- Blog post (https://databricks.com/blog/2017/10/30/introducing-vectorized-udfs-for-pyspark.html)
- Documentation (https://spark.apache.org/docs/latest/sql-programming-guide.html#pyspark-usage-guide-for-pandas-with-apache-arrow)



The user-defined functions are executed by:

- Apache Arrow (https://arrow.apache.org/), is an in-memory columnar data format that is used in Spark to efficiently transfer data between JVM and Python processes with near-zero (de)serialization cost. See more here (https://spark.apache.org/docs/latest/sql-pyspark-pandas-with-arrow.html).
- pandas inside the function, to work with pandas instances and APIs.

NOTE: In Spark 3.0, you should define your Pandas UDF using Python type hints.

In [0]:

```
from pyspark.sql.functions import pandas_udf

@pandas_udf("double")
def predict(*args: pd.Series) -> pd.Series:
    model_path = f"runs:/{run.info.run_id}/model"
    model = mlflow.sklearn.load_model(model_path) # Load model
    pdf = pd.concat(args, axis=1)
    return pd.Series(model.predict(pdf))

prediction_df = spark_df.withColumn("prediction", predict(*spark_df.columns))
display(prediction_df)
```

--i18n-e97526c6-ef40-4d55-9763-ee3ebe846096

Pandas Scalar Iterator UDF

If your model is very large, then there is high overhead for the Pandas UDF to repeatedly load the same model for every batch in the same Python worker process. In Spark 3.0, Pandas UDFs can accept an iterator of pandas.Series or pandas.DataFrame so that you can load the model only once instead of loading it for every series in the iterator.

This way the cost of any set-up needed will be incurred fewer times. When the number of records you're working with is greater than **`spark.conf.get('spark.sql.execution.arrow.maxRecordsPerBatch')`** , which is 10,000 by default, you should see speed ups over a pandas scalar UDF because it iterates through batches of pd.Series.

It has the general syntax of:

```
@pandas_udf(...)
def predict(iterator):
    model = ... # load model
    for features in iterator:
        yield model.predict(features)
```

In [0]:

```
from typing import Iterator, Tuple

@pandas_udf("double")
def predict(iterator: Iterator[pd.DataFrame]) -> Iterator[pd.Series]:
    model_path = f"runs:/{run.info.run_id}/model"
    model = mlflow.sklearn.load_model(model_path) # Load model
    for features in iterator:
        pdf = pd.concat(features, axis=1)
        yield pd.Series(model.predict(pdf))

prediction_df = spark_df.withColumn("prediction", predict(*spark_df.columns))
display(prediction_df)
```

--i18n-23b8296e-e0bc-481e-bd35-4048d532c71d

Pandas Function API

Instead of using a Pandas UDF, we can use a Pandas Function API. This new category in Apache Spark 3.0 enables you to directly apply a Python native function, which takes and outputs Pandas instances against a PySpark DataFrame. Pandas Functions APIs supported in Apache Spark 3.0 are: grouped map, map, and co-grouped map.

`mapInPandas()` takes an iterator of pandas.DataFrame as input, and outputs another iterator of pandas.DataFrame. It's flexible and easy to use if your model requires all of your columns as input, but it requires serialization/deserialization of the whole DataFrame (as it is passed to its input). You can control the size of each pandas.DataFrame with the **`spark.sql.execution.arrow.maxRecordsPerBatch`** config.

In [0]:

```
def predict(iterator: Iterator[pd.DataFrame]) -> Iterator[pd.DataFrame]:
    model_path = f"runs:/{run.info.run_id}/model"
    model = mlflow.sklearn.load_model(model_path) # Load model
    for features in iterator:
        yield pd.concat([features, pd.Series(model.predict(features), name="predict

display(spark_df.mapInPandas(predict, """`host_total_listings_count` DOUBLE,`neighb
```

--i18n-d13b87a7-0625-4acc-88dc-438cf06e18bd

Or you can define the schema like this below.

In [0]:

```
from pyspark.sql.functions import lit
from pyspark.sql.types import DoubleType

schema = spark_df.withColumn("prediction", lit(None).cast(DoubleType())).schema
display(spark_df.mapInPandas(predict, schema))
```

-sandbox

--i18n-b69335d5-86c7-40c5-b430-509a7444dae7

Feature Store

The Databricks Feature Store (https://docs.databricks.com/applications/machine-learning/feature-store.html) is a centralized repository of features. It enables feature sharing and discovery across your organization and also ensures that the same feature computation code is used for model training and inference.

Check out Feature Store Python API documentation here (https://docs.databricks.com/dev-tools/api/python/latest/index.html#feature-store-python-api-reference).

In this lesson you will:

- Build a feature store with the Databricks Feature Store
- Update feature tables
- Perform batch scoring

In [0]:

```
%run "./Includes/Classroom-Setup"
```

In [0]:

```
from pyspark.sql.functions import monotonically_increasing_id, lit, expr, rand
import uuid
from databricks import feature_store
from pyspark.sql.types import StringType, DoubleType
from databricks.feature_store import feature_table, FeatureLookup
import mlflow
import mlflow.sklearn
from mlflow.models.signature import infer_signature
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
```

--i18n-5dcd3e8e-2553-429f-bbe1-aef0bc1ef0ab

Let's load in our data and generate a unique ID for each listing. The **`index`** column will serve as the "key" of the feature table and used to lookup features.

In [0]:

```
file_path = f"{DA.paths.datasets}/airbnb/sf-listings/sf-listings-2019-03-06-clean.d
airbnb_df = spark.read.format("delta").load(file_path).coalesce(1).withColumn("inde
display(airbnb_df)
```

--i18n-a04b29f6-e7a6-4e6a-875f-945edf938e9e

Create a new database and unique table name (in case you re-run the notebook multiple times)

In [0]:

```
spark.sql(f"CREATE DATABASE IF NOT EXISTS {DA.cleaned_username}")
table_name = f"{DA.cleaned_username}.airbnb_" + str(uuid.uuid4())[:6]
print(table_name)
```

--i18n-a0712a39-b413-490f-a59e-dbd7f533e9a9

Let's start creating a [Feature Store Client (https://docs.databricks.com/applications/machine-learning/feature-store.html#create-a-feature-table-in-databricks-feature-store)](https://docs.databricks.com/applications/machine-learning/feature-store.html#create-a-feature-table-in-databricks-feature-store) so we can populate our feature store.

In [0]:

```
fs = feature_store.FeatureStoreClient()
# help(fs.create_table)
```

--i18n-90998fdb-87ed-4cdd-8844-fbd59ac5631f

Create Feature Table

Next, we can create the Feature Table using the **`create_table`** method.

This method takes a few parameters as inputs:

- **`name`** - A feature table name of the form **`<database_name>.<table_name>`**
- **`primary_keys`** - The primary key(s). If multiple columns are required, specify a list of column names.
- **`df`** - Data to insert into this feature table. The schema of **`features_df`** will be used as the feature table schema.
- **`schema`** - Feature table schema. Note that either **`schema`** or **`features_df`** must be provided.
- **`description`** - Description of the feature table
- **`partition_columns`** - Column(s) used to partition the feature table.

In [0]:

```
## select numeric features and exclude target column "price"
numeric_cols = [x.name for x in airbnb_df.schema.fields if (x.dataType == DoubleTyp
numeric_features_df = airbnb_df.select(["index"] + numeric_cols)
display(numeric_features_df)
```

In [0]:

```
fs.create_table(
    name=table_name,
    primary_keys=["index"],
    df=numeric_features_df,
    schema=numeric_features_df.schema,
    description="Numeric features of airbnb data"
)
```

--i18n-4a7cbb2e-87a2-4ea8-85e6-207ec5e42147

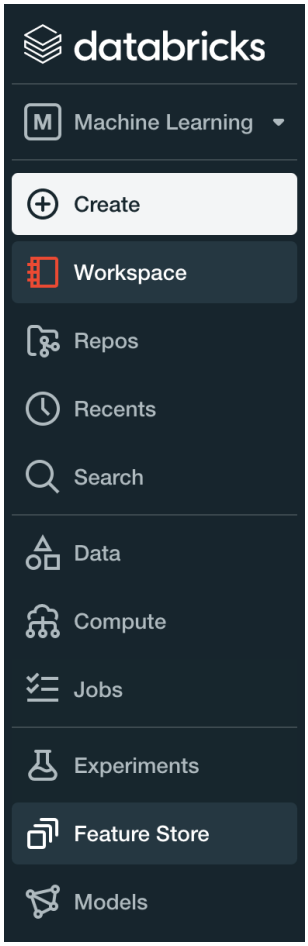
Alternatively, you can **`create_table`** with schema only (without **`df`**), and populate data to the feature table with **`fs.write_table`**. **`fs.write_table`** supports both **`overwrite`** and **`merge`** modes.

Example:

```
fs.create_table(
    name=table_name,
    primary_keys=["index"],
    schema=numeric_features_df.schema,
    description="Original Airbnb data"
)

fs.write_table(
    name=table_name,
    df=numeric_features_df,
    mode="overwrite"
)
```

--i18n-44586907-302a-4916-93f6-e92210619c6f

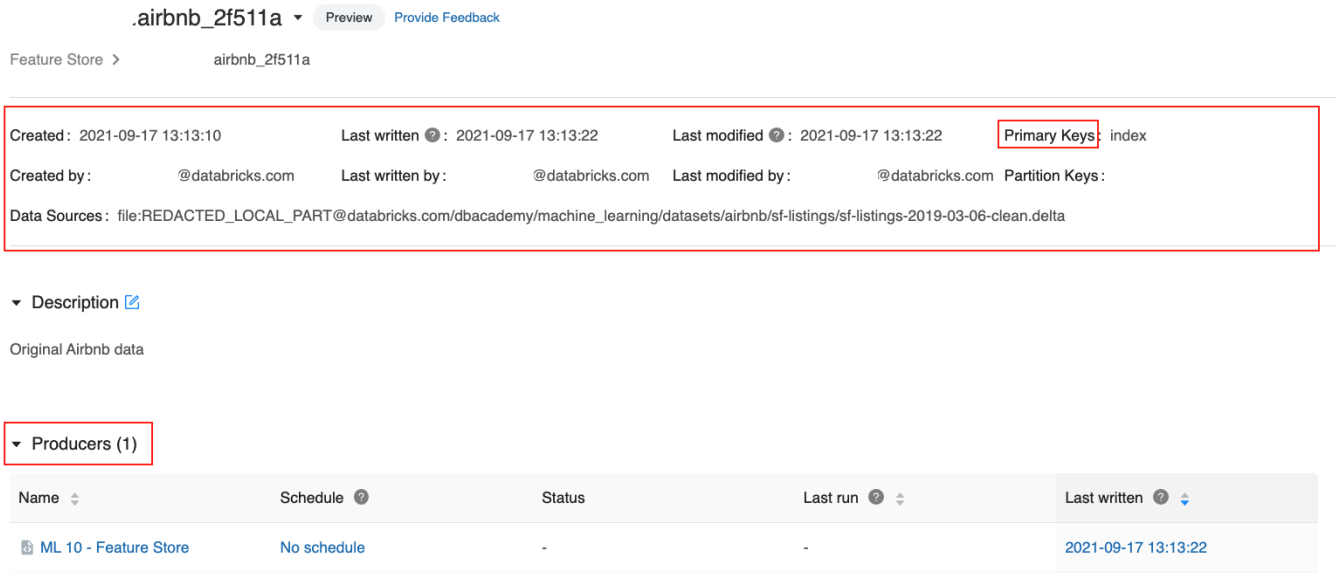
Now let's explore the UI and see how it tracks the tables that we created. Navigate to the UI by first ensuring that you are in the Machine Learning workspace, and then clicking on the Feature Store icon on the bottom-left of the navigation bar.



--i18n-cf0ad0d0-8456-471b-935c-8a34a836fca7

In this screenshot, we can see the feature table that we created.

Note the section of **`Producers`** . This section indicates which notebook produces the feature table.



--i18n-b07da702-485e-44b8-bd00-f0330c8b7657

We can also look at the metadata of the feature store via the FeatureStore client by using **`get_table()`** .

In [0]:

```
fs.get_table(table_name).path_data_sources
```

In [0]:

```
fs.get_table(table_name).description
```

--i18n-1df7795c-1a07-47ae-92a8-1c5f7aec75ae

Train a model with feature store

--i18n-bcbf72b7-a013-40fd-bf55-a2b179a7728e

The prediction target **`price`** should NOT BE included as a feature in the registered feature table.

Further, there may be other information that *can* be supplied at inference time, but does not make sense to consider a feature to *look up*.

In this (fictional) example, we made up a feature **`score_diff_from_last_month`** . It is a feature generated at inference time and used in training as well.

In [0]:

```
## inference data -- index (key), price (target) and a online feature (make up a fi
inference_data_df = airbnb_df.select("index", "price", (rand() * 0.5-0.25).alias("s
display(inference_data_df)
```

--i18n-b8301fa9-27bd-4d3b-bf13-9ab784205d81

Build a training dataset that will use the indicated "key" to lookup features from the feature table and also the online feature **`score_diff_from_last_month`** . We will use FeatureLookup (https://docs.databricks.com/dev-tools/api/python/latest/index.html) and if you specify no features, it will return all of them except the primary key.

In [0]:

```
def load_data(table_name, lookup_key):
    model_feature_lookups = [FeatureLookup(table_name=table_name, lookup_key=lookup

    # fs.create_training_set will look up features in model_feature_lookups with ma
    training_set = fs.create_training_set(inference_data_df, model_feature_lookups,
    training_pd = training_set.load_df().toPandas()

    # Create train and test datasets
    X = training_pd.drop("price", axis=1)
    y = training_pd["price"]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random
    return X_train, X_test, y_train, y_test, training_set

X_train, X_test, y_train, y_test, training_set = load_data(table_name, "index")
X_train.head()
```

--i18n-eae1aa4a-f770-4173-9502-cb946e6949d2

Train a **RandomForestRegressor** model and log the model with the Feature Store. An MLflow run is started to track the autologged components as well as the Feature Store logged model. However, we will disable the MLflow model autologging as the model will be explicitly logged via the Feature Store.

NOTE: This is an overly simplistic example, used solely for demo purposes.

In [0]:

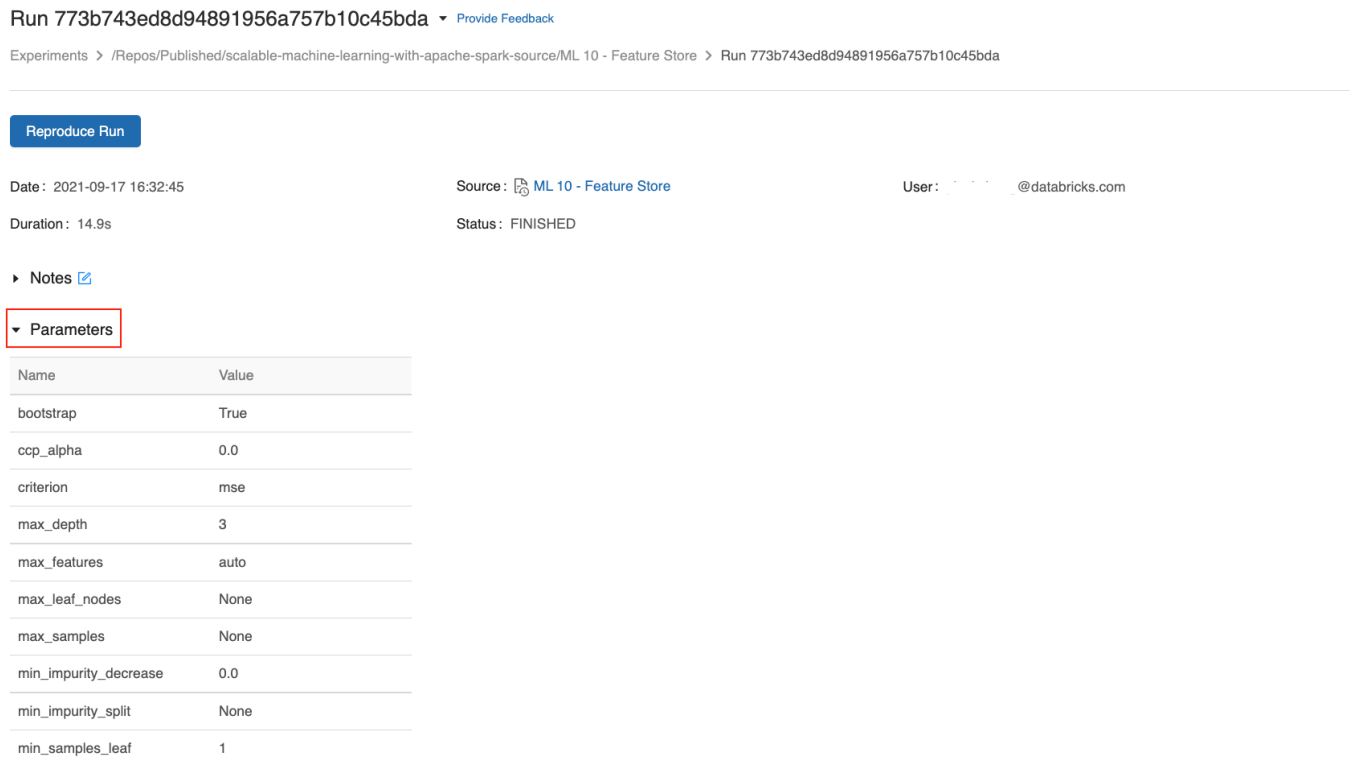
```
from mlflow.tracking.client import MlflowClient

client = MlflowClient()

try:
    client.delete_registered_model(f"feature_store_airbnb_{DA.cleaned_username}") #
except:
    None
```

In [0]:

```
# Disable model autologging and instead log explicitly via the FeatureStore
mlflow.sklearn.autolog(log_models=False)

def train_model(X_train, X_test, y_train, y_test, training_set, fs):
    ## fit and log model
    with mlflow.start_run() as run:

        rf = RandomForestRegressor(max_depth=3, n_estimators=20, random_state=42)
        rf.fit(X_train, y_train)
        y_pred = rf.predict(X_test)

        mlflow.log_metric("test_mse", mean_squared_error(y_test, y_pred))
        mlflow.log_metric("test_r2_score", r2_score(y_test, y_pred))

        fs.log_model(
            model=rf,
            artifact_path="feature-store-model",
            flavor=mlflow.sklearn,
            training_set=training_set,
            registered_model_name=f"feature_store_airbnb_{DA.cleaned_username}",
            input_example=X_train[:5],
            signature=infer_signature(X_train, y_train)
        )

train_model(X_train, X_test, y_train, y_test, training_set, fs)
```

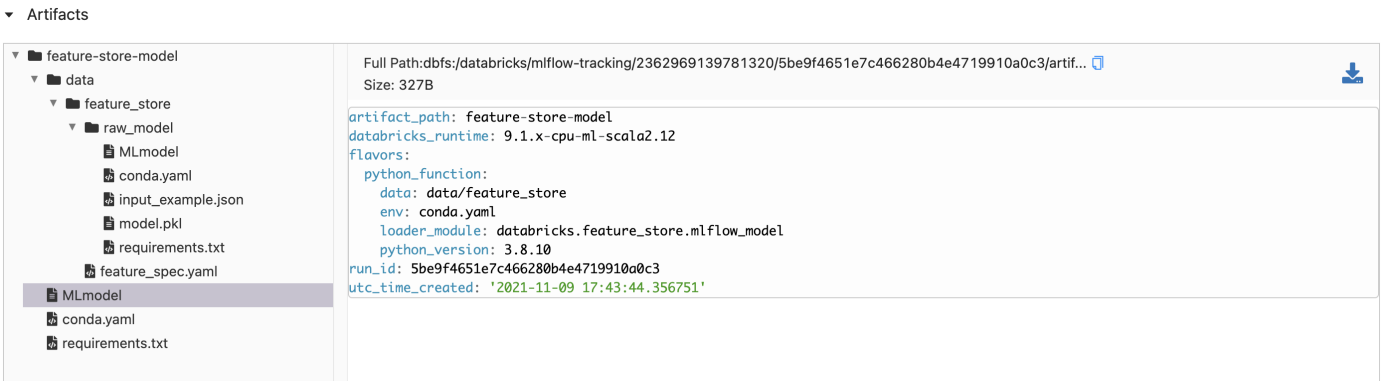
--i18n-40b7718f-101c-4ac4-8639-545b8ef6d932

Now, view the run from MLflow UI. You can find the model parameters logged with MLflow autolog.



--i18n-f03314dc-1ade-4bd8-958f-ddf04ac1bb13

Notice the saved model artifact **feature_store_model** : packaged feature store model that can be used directly for batch scoring - logged from **`fs.log_model`**



--i18n-acd4d5a4-c4ed-4695-a911-5fd88dcfa513

The **`feature_store_model`** is registered in the MLflow model registry as well. You can find it in **`Models`** page. It is also logged at the feature store page, indicating which features in the feature table are used for the model. We will exam feature/model lineage through the UI together later.

--i18n-921dc6c9-b9ed-43c7-86ff-608791a11367

Feature Store Batch Scoring

Apply a feature store registered MLflow model to features with **`score_batch`** . Input data only need the key column **`index`** and online feature **`score_diff_from_last_month`** . Everything else is looked up.

In [0]:

```
## For sake of simplicity, we will just predict on the same inference_data_df
batch_input_df = inference_data_df.drop("price") # Exclude true label
predictions_df = fs.score_batch(f"models:/feature_store_airbnb_{DA.cleaned_username
                                batch_input_df, result_type="double")
display(predictions_df)
```

--i18n-fa42d4d3-a6a6-4205-b799-032154d1d8a3

Overwrite feature table

Lastly, we'll condense some of the review columns and update the feature table: we'll do this by calculating the average review score for each listing.

In [0]:

```
## select numeric features and aggregate the review scores
review_columns = ["review_scores_accuracy", "review_scores_cleanliness", "review_sc
                  "review_scores_communication", "review_scores_location", "review_s

condensed_review_df = (airbnb_df
                       .select(["index"] + numeric_cols)
                       .withColumn("average_review_score", expr("+".join(review_col
                       .drop(*review_columns)
                      )

display(condensed_review_df)
```

--i18n-da3ee1df-391c-4f26-99d0-82937e91a40a

Let's now drop those features using **`overwrite`** .

In [0]:

```
fs.write_table(
    name=table_name,
    df=condensed_review_df,
    mode="overwrite"
)
```

--i18n-ae45b580-e79e-4f54-85a0-1274cb5f5c5f

Explore the feature permission, lineage and freshness from Feature Store UI

--i18n-5d4d8425-b9b7-4e47-8856-91e1142e9c47

On the UI, we can see that:

- A new column has been added to the feature list
- Columns that we deleted are also still present. However, the deleted features will have **`null`** as their values when we read in the table

- The "Models" column are populated, listing models use the features from the table
- The last column **`Notebooks`** are populated. This column indicates which notebooks consume the features in the feature table

Feature	Data Type	Consumers			
		Models	Endpoints	Jobs	Notebooks
accommodates	DOUBLE	feature_store_airbnb_feifei_wang/1	-	-	ML 10 - Feature Store_20220613
average_review_score	DOUBLE	-	-	-	-
bathrooms	DOUBLE	feature_store_airbnb_feifei_wang/1	-	-	ML 10 - Feature Store_20220613
bathrooms_na	DOUBLE	feature_store_airbnb_feifei_wang/1	-	-	ML 10 - Feature Store_20220613
bedrooms	DOUBLE	feature_store_airbnb_feifei_wang/1	-	-	ML 10 - Feature Store_20220613
bedrooms_na	DOUBLE	feature_store_airbnb_feifei_wang/1	-	-	ML 10 - Feature Store_20220613
beds	DOUBLE	feature_store_airbnb_feifei_wang/1	-	-	ML 10 - Feature Store_20220613
beds_na	DOUBLE	feature_store_airbnb_feifei_wang/1	-	-	ML 10 - Feature Store_20220613
host_total_listings_count	DOUBLE	feature_store_airbnb_feifei_wang/1	-	-	ML 10 - Feature Store_20220613
index	LONG	-	-	-	-

< 1 2 3 >

--i18n-884ff3ff-f965-4c37-8cff-f6a1600ee0b6

Now, let's read in the feature data from the Feature Store. By default, **`fs.read_table()`** reads in the latest version of the feature table. To read in the specific version of feature table, you can optionally specify the argument **`as_of_delta_timestamp`** by passing a date in a timestamp format or string.

Note that the values of the deleted columns have been replaced by **`null`** .

In [0]:

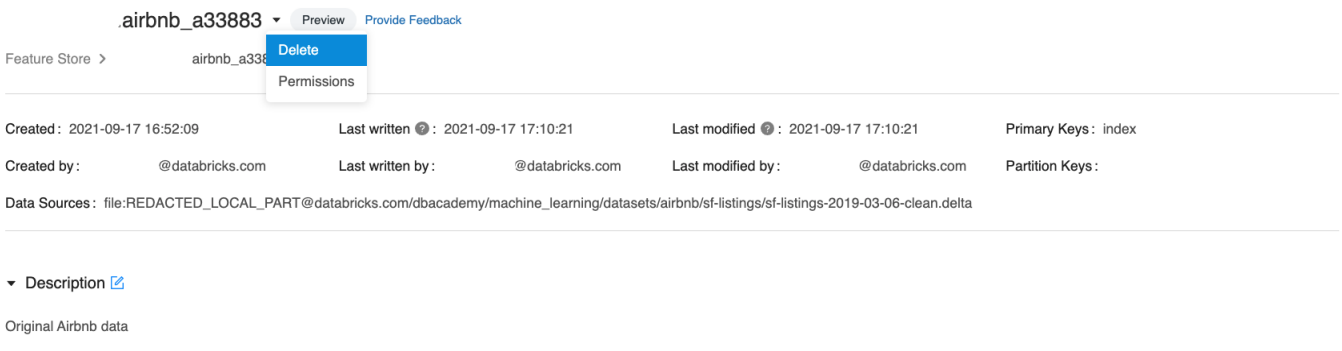
```
# Displays most recent table
display(fs.read_table(name=table_name))
```

--i18n-4148328d-4046-4251-b4db-f9e427b2e0f9

If you need to use the features for real-time serving, you can publish your features to an online store (https://docs.databricks.com/applications/machine-learning/feature-store.html#publish-features-to-an-online-feature-store).

We can perform control who has permissions to the feature table on the UI.

To delete the table, use the **`delete`** button on the UI. **You need to delete the delta table from database as well.**



--i18n-81e53dea-dc51-418c-b366-eed3a9c4ce2f

Retrain a new model with the new average_review_score feature

In [0]:

```
def load_data(table_name, lookup_key):
    model_feature_lookups = [FeatureLookup(table_name=table_name, lookup_key=lookup

    # fs.create_training_set will look up features in model_feature_lookups with ma
    training_set = fs.create_training_set(inference_data_df, model_feature_lookups,
    training_pd = training_set.load_df().drop(*review_columns).toPandas()  #remove

    # Create train and test datasets
    X = training_pd.drop("price", axis=1)
    y = training_pd["price"]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random
    return X_train, X_test, y_train, y_test, training_set

X_train, X_test, y_train, y_test, training_set = load_data(table_name, "index")
X_train.head()
```

--i18n-94873d7f-3bb9-4d5f-a414-c24480a84f3b

Build a training dataset that will use the indicated `key` to lookup features.

In [0]:

```
def train_model(X_train, X_test, y_train, y_test, training_set, fs):
    ## fit and log model
    with mlflow.start_run() as run:

        rf = RandomForestRegressor(max_depth=3, n_estimators=20, random_state=42)
        rf.fit(X_train, y_train)
        y_pred = rf.predict(X_test)

        mlflow.log_metric("test_mse", mean_squared_error(y_test, y_pred))
        mlflow.log_metric("test_r2_score", r2_score(y_test, y_pred))

        fs.log_model(
            model=rf,
            artifact_path="feature-store-model",
            flavor=mlflow.sklearn,
            training_set=training_set,
            registered_model_name=f"feature_store_airbnb_{DA.cleaned_username}",
            input_example=X_train[:5],
            signature=infer_signature(X_train, y_train)
        )

train_model(X_train, X_test, y_train, y_test, training_set, fs)
```

--i18n-b0ffd91d-c73f-4f86-a02a-43ffdc73460c

Feature Store Batch Scoring

Apply the feature store registered MLflow model version2 to features with **`score_batch`** .

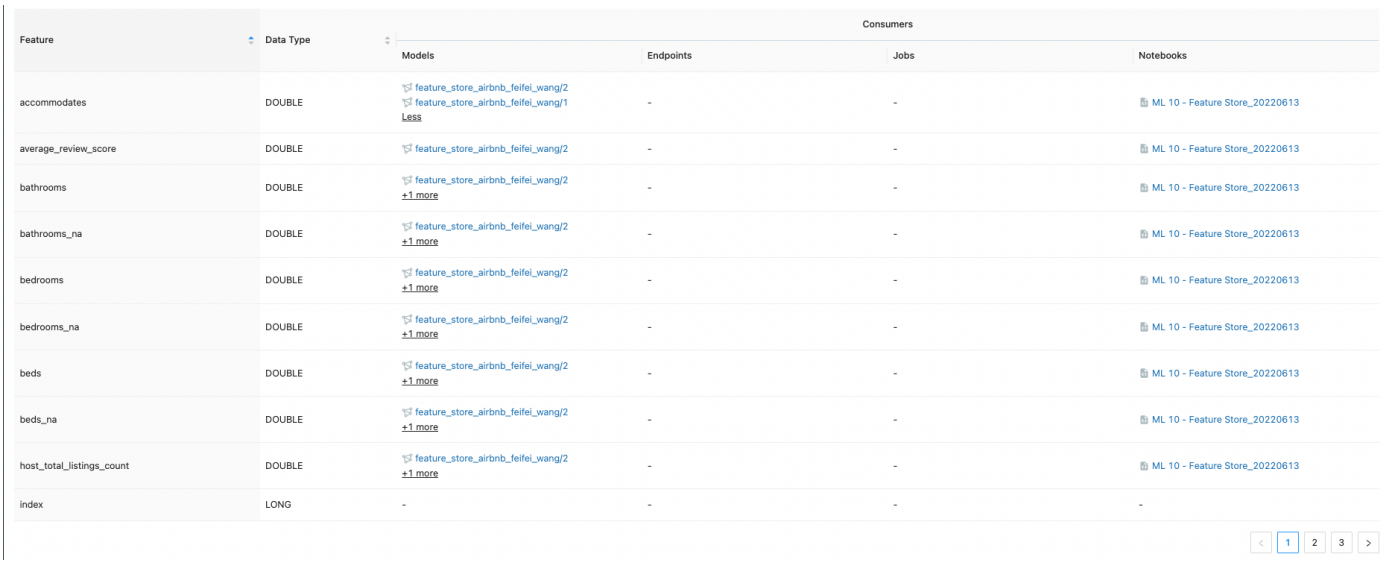
In [0]:

```
## For sake of simplicity, we will just predict on the same inference_data_df
batch_input_df = inference_data_df.drop("price") # Exclude true label
predictions_df = fs.score_batch(f"models:/feature_store_airbnb_{DA.cleaned_username
                                batch_input_df, result_type="double")
display(predictions_df)
```

--i18n-67471f1c-0dc0-445f-ae6a-beafb3508a16

On the UI, we can see that:

- The model version 2 is using the newly created feature of average_review_score
- Columns that we deleted are also still present. However, the deleted features will have **`null`** as their values when we read in the table
- The "Models" column are populated, listing versions of models use the versions of features from the table
- The last column **`Notebooks`** are populated. This column indicates which notebooks consume the features in the feature table



Feature	Data Type	Consumers			
		Models	Endpoints	Jobs	Notebooks
accommodates	DOUBLE	feature_store_airbnb_feifei_wang/2 feature_store_airbnb_feifei_wang/1 Less	-	-	ML 10 - Feature Store_20220613
average_review_score	DOUBLE	feature_store_airbnb_feifei_wang/2	-	-	ML 10 - Feature Store_20220613
bathrooms	DOUBLE	feature_store_airbnb_feifei_wang/2 +1 more	-	-	ML 10 - Feature Store_20220613
bathrooms_na	DOUBLE	feature_store_airbnb_feifei_wang/2 +1 more	-	-	ML 10 - Feature Store_20220613
bedrooms	DOUBLE	feature_store_airbnb_feifei_wang/2 +1 more	-	-	ML 10 - Feature Store_20220613
bedrooms_na	DOUBLE	feature_store_airbnb_feifei_wang/2 +1 more	-	-	ML 10 - Feature Store_20220613
beds	DOUBLE	feature_store_airbnb_feifei_wang/2 +1 more	-	-	ML 10 - Feature Store_20220613
beds_na	DOUBLE	feature_store_airbnb_feifei_wang/2 +1 more	-	-	ML 10 - Feature Store_20220613
host_total_listings_count	DOUBLE	feature_store_airbnb_feifei_wang/2 +1 more	-	-	ML 10 - Feature Store_20220613
index	LONG	-	-	-	-

< 1 2 3 >

-sandbox

--i18n-c311be95-77f9-477b-93a5-c9289b3dedb6

Pandas API on Spark

The pandas API on Spark project makes data scientists more productive when interacting with big data, by implementing the pandas DataFrame API on top of Apache Spark. By unifying the two ecosystems with a familiar API, pandas API on Spark offers a seamless transition between small and large data.

Some of you might be familiar with the Koalas (https://github.com/databricks/koalas) project, which has been merged into PySpark in 3.2. For Apache Spark 3.2 and above, please use PySpark directly as the standalone Koalas project is now in maintenance mode. See this blog post (https://databricks.com/blog/2021/10/04/pandas-api-on-upcoming-apache-spark-3-2.html).

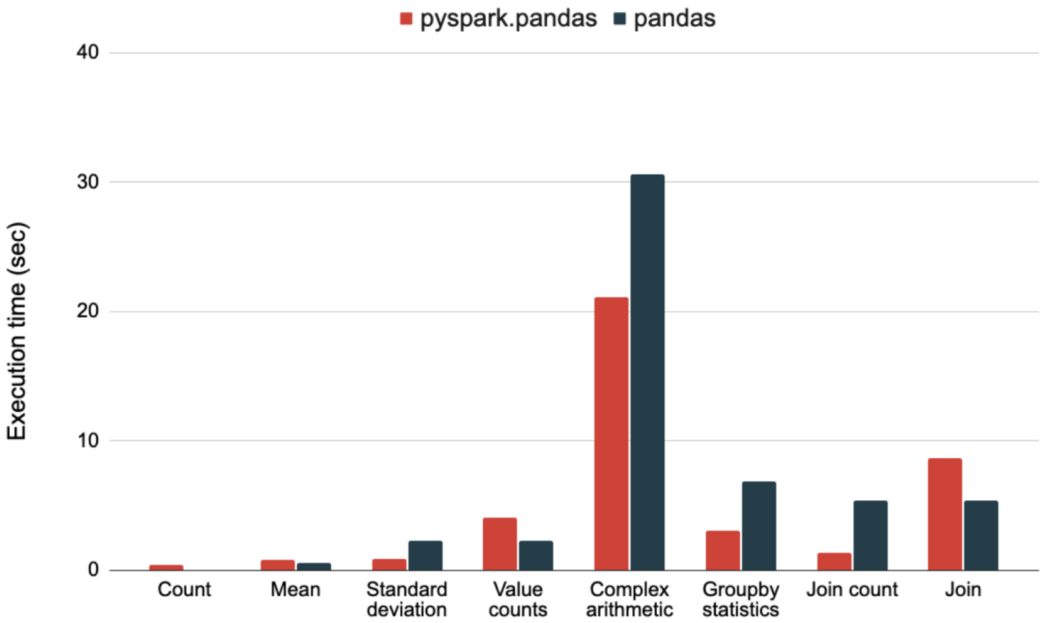
In this lesson you:

- Demonstrate the similarities of the pandas API on Spark API with the pandas API
- Understand the differences in syntax for the same DataFrame operations in pandas API on Spark vs PySpark

--i18n-d711990a-af32-4357-b710-d2db434e4f15

Performance Comparison - 31 GB

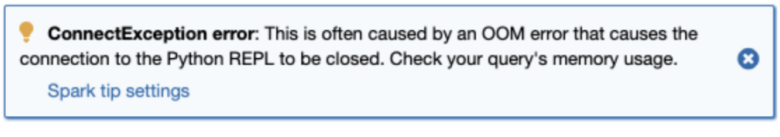
Lower is better



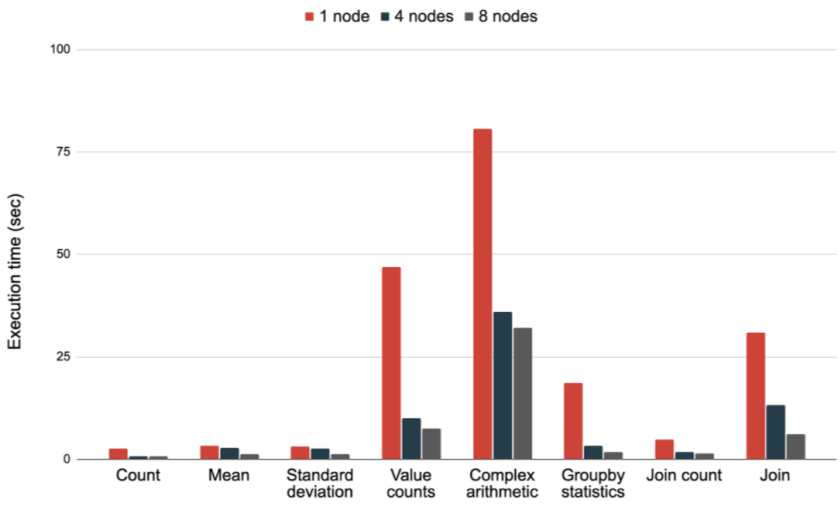
Performance Comparison - 95 GB

Lower is better

pandas



pyspark.pandas



Pandas DataFrames are mutable, eagerily evaluated, and maintain row order. They are restricted to a single machine, and are very performant when the data sets are small, as shown in a).

Spark DataFrames are distributed, lazily evaluated, immutable, and do not maintain row order. They are very performant when working at scale, as shown in b) and c).

pandas API on Spark provides the best of both worlds: pandas API with the performance benefits of Spark. However, it is not as fast as implementing your solution natively in Spark, and let's see why below.

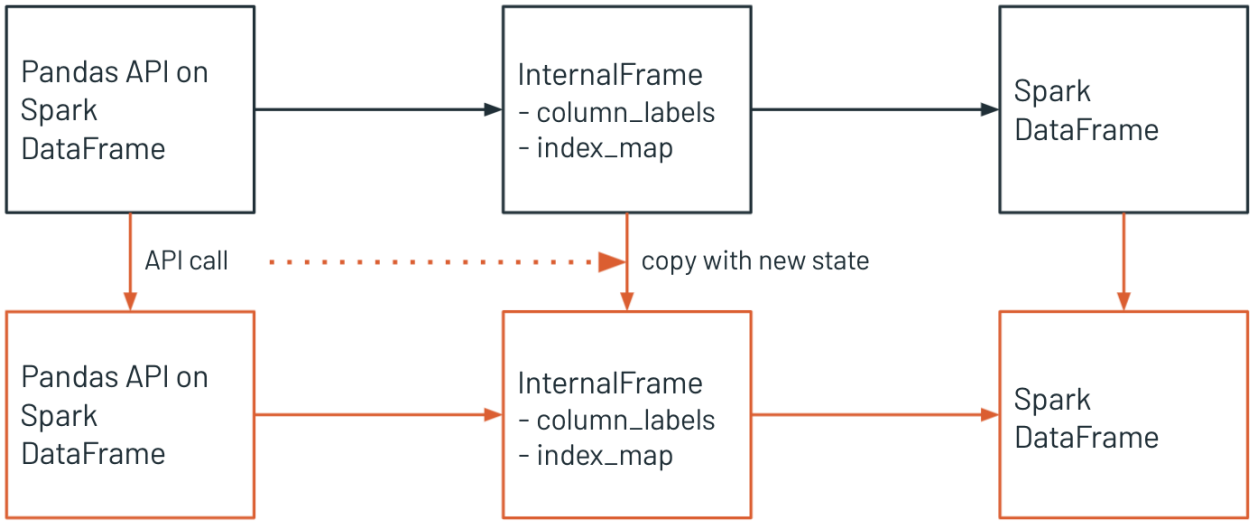
--i18n-c3080510-c8d9-4020-9910-37199f0ad5de

InternalFrame

The InternalFrame holds the current Spark DataFrame and internal immutable metadata.

It manages mappings from pandas API on Spark column names to Spark column names, as well as from pandas API on Spark index names to Spark column names.

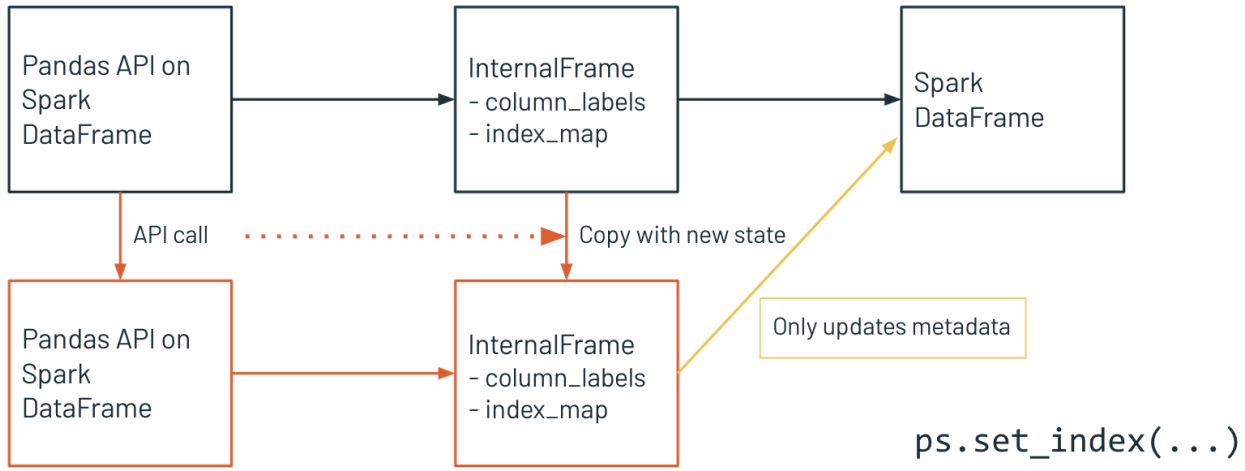
If a user calls some API, the pandas API on Spark DataFrame updates the Spark DataFrame and metadata in InternalFrame. It creates or copies the current InternalFrame with the new states, and returns a new pandas API on Spark DataFrame.



--i18n-785ed714-6726-40d5-b7fb-c63c094e568e

InternalFrame Metadata Updates Only

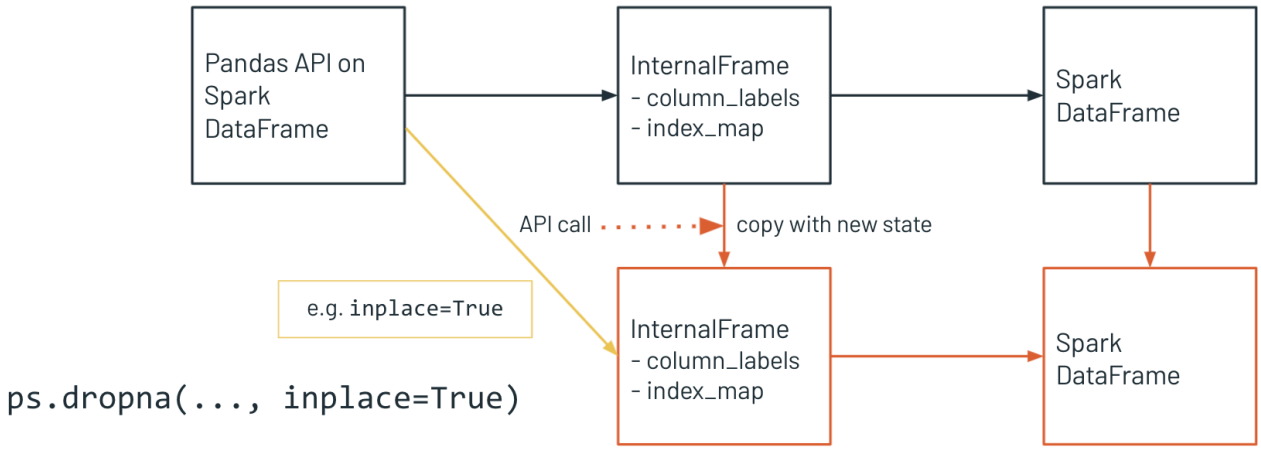
Sometimes the update of Spark DataFrame is not needed but of metadata only, then new structure will be like this.



--i18n-e6d7a47f-a4c8-4178-bc70-62c2ac6764d5

InternalFrame Inplace Updates

On the other hand, sometimes pandas API on Spark DataFrame updates internal state instead of returning a new DataFrame, for example, the argument inplace=True is provided, then new structure will be like this.



--i18n-23a2fc6d-1360-4e41-beab-b1fe8e23aac3

Read in the dataset

- PySpark
- pandas
- pandas API on Spark

In [0]:

```
%run "./Includes/Classroom-Setup"
```

--i18n-1be64dea-9d63-476d-a7d6-9f6fa4ccd784

Read in Parquet with PySpark

In [0]:

```
spark_df = spark.read.parquet(f"{DA.paths.datasets}/airbnb/sf-listings/sf-listings-
display(spark_df)
```

--i18n-00b99bdc-e4d1-44d2-b117-ae2cd97d0490

Read in Parquet with pandas

In [0]:

```
import pandas as pd

pandas_df = pd.read_parquet(f"{DA.paths.datasets.replace('dbfs:/', '/dbfs/')}/airbn
pandas_df.head()
```

--i18n-e75a3ba6-98f6-4b39-aecb-345109cb2ce9

Read in Parquet with pandas API on Spark. You'll notice pandas API on Spark generates an index column for you, like in pandas.

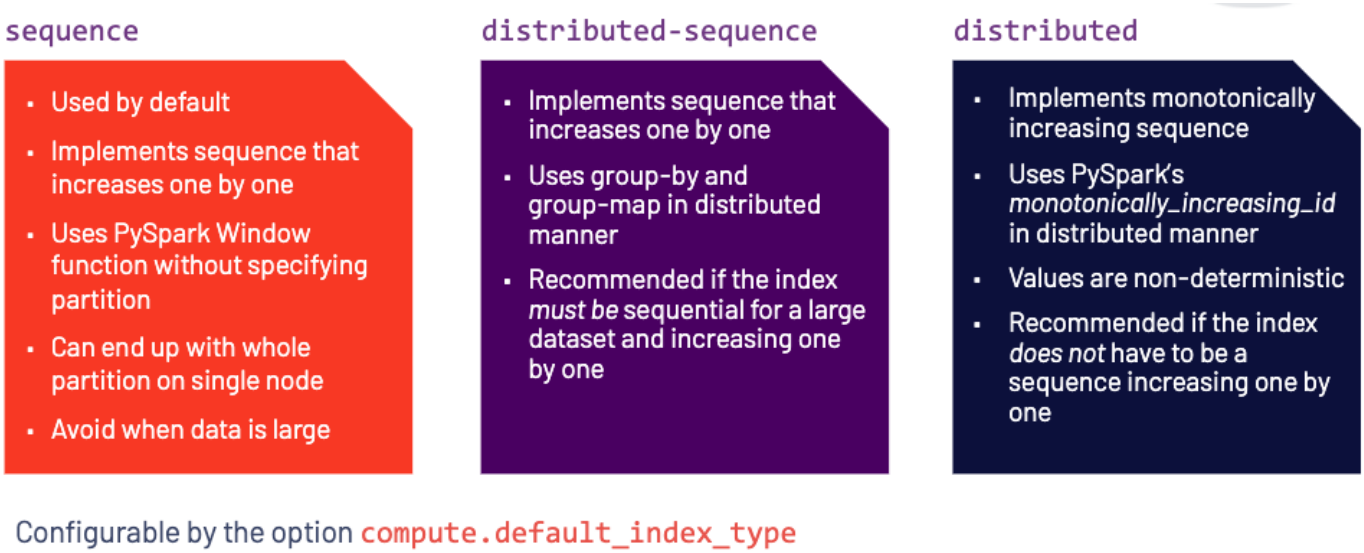
Pandas API on Spark also supports reading from Delta (**`read_delta`**), but pandas does not support that yet.

In [0]:

```
import pyspark.pandas as ps

df = ps.read_parquet(f"{DA.paths.datasets}/airbnb/sf-listings/sf-listings-2019-03-0
df.head()
```

--i18n-f099c73b-0bd8-4ff1-a12e-578ffb0cb152

Index Types (https://koalas.readthedocs.io/en/latest/user_guide/options.html#default-index-type)



In [0]:

```
ps.set_option("compute.default_index_type", "distributed-sequence")
df_dist_sequence = ps.read_parquet(f"{DA.paths.datasets}/airbnb/sf-listings/sf-list
df_dist_sequence.head()
```

--i18n-07b3f029-f81b-442f-8cdd-cb2d29033a35

Converting to pandas API on Spark DataFrame to/from Spark DataFrame

--i18n-ed25204e-2822-4694-b3b3-968ea8ef7343

Creating a pandas API on Spark DataFrame from PySpark DataFrame

In [0]:

```
df = ps.DataFrame(spark_df)
display(df)
```

--i18n-a41480c7-1787-4bd6-a4c3-c85552a5f762

Alternative way of creating a pandas API on Spark DataFrame from PySpark DataFrame

In [0]:

```
df = spark_df.to_pandas_on_spark()
display(df)
```

--i18n-5abf965b-2f69-469e-a0cf-ba8ffd714764

Go from a pandas API on Spark DataFrame to a Spark DataFrame

In [0]:

```
display(df.to_spark())
```

--i18n-480e9e60-9286-4f4c-9db3-b650b32cb7ce

Value Counts

--i18n-99f93d32-d09d-4fea-9ac9-57099eb2c819

Get value counts of the different property types with PySpark

In [0]:

```
display(spark_df.groupby("property_type").count().orderBy("count", ascending=False))
```

--i18n-150b6a18-123d-431a-84b1-ad2d2b7beae2

Get value counts of the different property types with pandas API on Spark

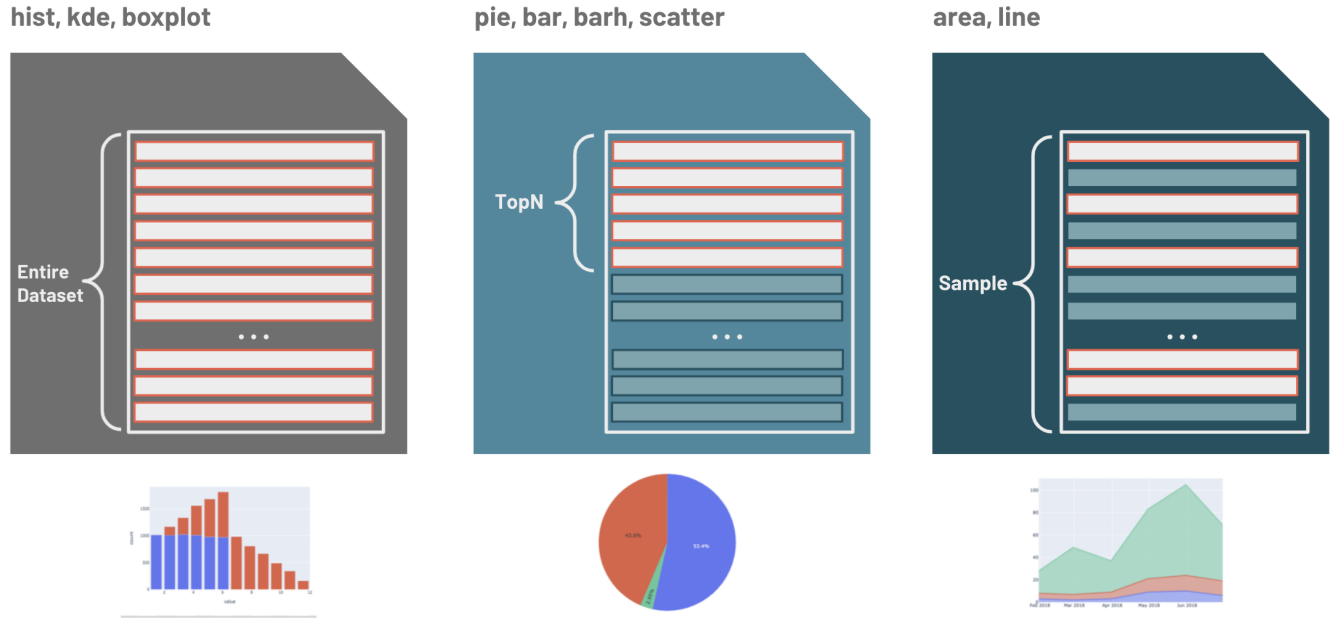
In [0]:

```
df["property_type"].value_counts()
```

--i18n-767f19b5-137f-4b33-9ef4-e5bb48603299

Visualizations

Based on the type of visualization, the pandas API on Spark has optimized ways to execute the plotting.



In [0]:

```
df.plot(kind="hist", x="bedrooms", y="price", bins=200)
```

--i18n-6b70f1df-dfe1-43de-aeec-5541b036927c

SQL on pandas API on Spark DataFrames

In [0]:

```
ps.sql("SELECT distinct(property_type) FROM {df}")
```

--i18n-7345361b-e6c4-4ce3-9ba4-8f132c8c8df2

Interesting Facts

- With pandas API on Spark you can read from Delta Tables and read in a directory of files
- If you use apply on a pandas API on Spark DF and that DF is <1000 (by default), pandas API on Spark will use pandas as a shortcut - this can be adjusted using **`compute.shortcut_limit`**
- When you create bar plots, the top n rows are only used - this can be adjusted using **`plotting.max_rows`**
- How to utilize **`.apply`** docs (https://koalas.readthedocs.io/en/latest/reference/api/databricks.koalas.DataFrame.apply.html#databricks.koa with its use of return type hints similar to pandas UDFs
- How to check the execution plan, as well as caching a pandas API on Spark DF (which aren't immediately intuitive)
- Koalas are marsupials whose max speed is 30 kph (20 mph)

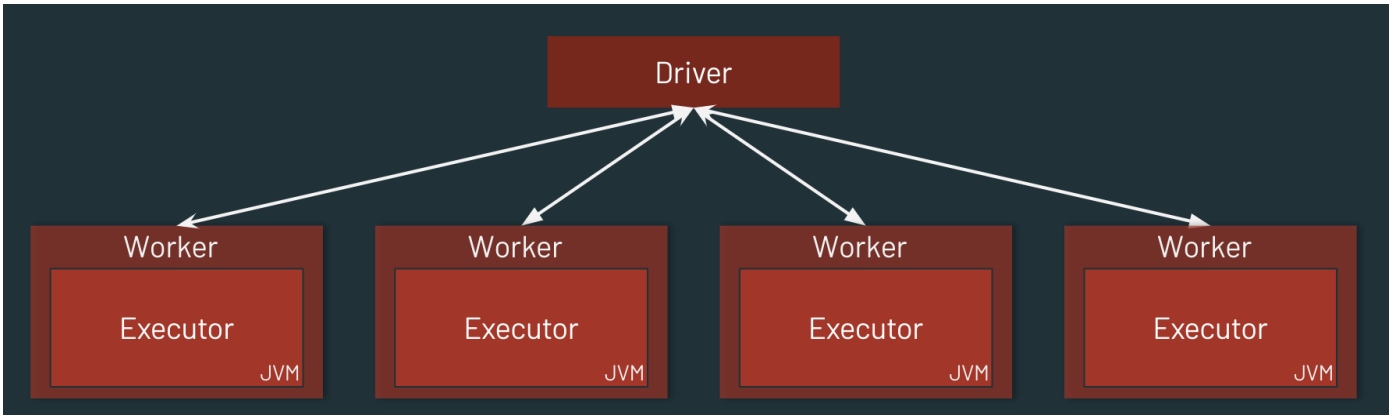
-sandbox

--i18n-1108b110-983d-4034-9156-6b95c04dc62c

Spark Review

In this lesson you:

- Create a Spark DataFrame
- Analyze the Spark UI
- Cache data
- Go between Pandas and Spark DataFrames



In [0]:

```
%run "./Includes/Classroom-Setup"
```

--i18n-df081f79-6894-4174-a554-fa0943599408

Spark DataFrame

In [0]:

```
from pyspark.sql.functions import col, rand

df = (spark.range(1, 1000000)
      .withColumn("id", (col("id") / 1000).cast("integer"))
      .withColumn("v", rand(seed=1)))
```

--i18n-a0c6912d-a8d6-449b-a3ab-5ca91c7f9805

Why were no Spark jobs kicked off above? Well, we didn't have to actually "touch" our data, so Spark didn't need to execute anything across the cluster.

In [0]:

```
display(df.sample(.001))
```

--i18n-6eadef21-d75c-45ba-8d77-419d1ce0c06c

Views

How can I access this in SQL?

In [0]:

```
df.createOrReplaceTempView("df_temp")
```

In [0]:

```
%sql
SELECT * FROM df_temp LIMIT 10
```

--i18n-2593e6b0-d34b-4086-9fed-c4956575a623

Count

Let's see how many records we have.

In [0]:

```
df.count()
```

--i18n-5d00511e-15da-48e7-bd26-e89fbe56632c

Spark UI

Open up the Spark UI - what are the shuffle read and shuffle write fields? The command below should give you a clue.

In [0]:

```
df.rdd.getNumPartitions()
```

--i18n-50330454-0168-4f50-8355-0204632b20ec

Cache

For repeated access, it will be much faster if we cache our data.

In [0]:

```
df.cache().count()
```

--i18n-7dd81880-1575-410c-a168-8ac081a97e9d

Re-run Count

Wow! Look at how much faster it is now!

In [0]:

```
df.count()
```

--i18n-ce238b9e-fee4-4644-9469-b7d9910f6243

Collect Data

When you pull data back to the driver (e.g. call **`.collect()`**, **`.toPandas()`**, etc), you'll need to be careful of how much data you're bringing back. Otherwise, you might get OOM exceptions!

A best practice is explicitly limit the number of records, unless you know your data set is small, before calling **`.collect()`** or **`.toPandas()`**.

In [0]:

```
df.limit(10).toPandas()
```

--i18n-279e3325-b121-402b-a2d0-486e1cc26fc0

What's new in Spark 3.0 (https://www.youtube.com/watch?v=l6SuXvhorDY&feature=emb_logo)

- Adaptive Query Execution (https://www.youtube.com/watch?v=jzrEc4r90N8&feature=emb_logo)
  - Dynamic query optimization that happens in the middle of your query based on runtime statistics
    - Dynamically coalesce shuffle partitions
    - Dynamically switch join strategies
    - Dynamically optimize skew joins
  - Enable it with: **`spark.sql.adaptive.enabled=true`**
- Dynamic Partition Pruning (DPP)

    - Avoid partition scanning based on the query results of the other query fragments
- Join Hints
- Improved Pandas UDFs (https://www.youtube.com/watch?v=UZl0pHG-2HA&feature=emb_logo)
    - Type Hints
    - Iterators
    - Pandas Function API (mapInPandas, applyInPandas, etc)
- And many more! See the migration guide (https://spark.apache.org/docs/latest/api/python/migration_guide/pyspark_2.4_to_3.0.html) and resources linked above.

-sandbox

--i18n-fd2d84ac-6a17-44c2-bb92-18b0c7fef797

Delta Review

There are a few key operations necessary to understand and make use of Delta Lake (https://docs.delta.io/latest/quick-start.html#create-a-table).

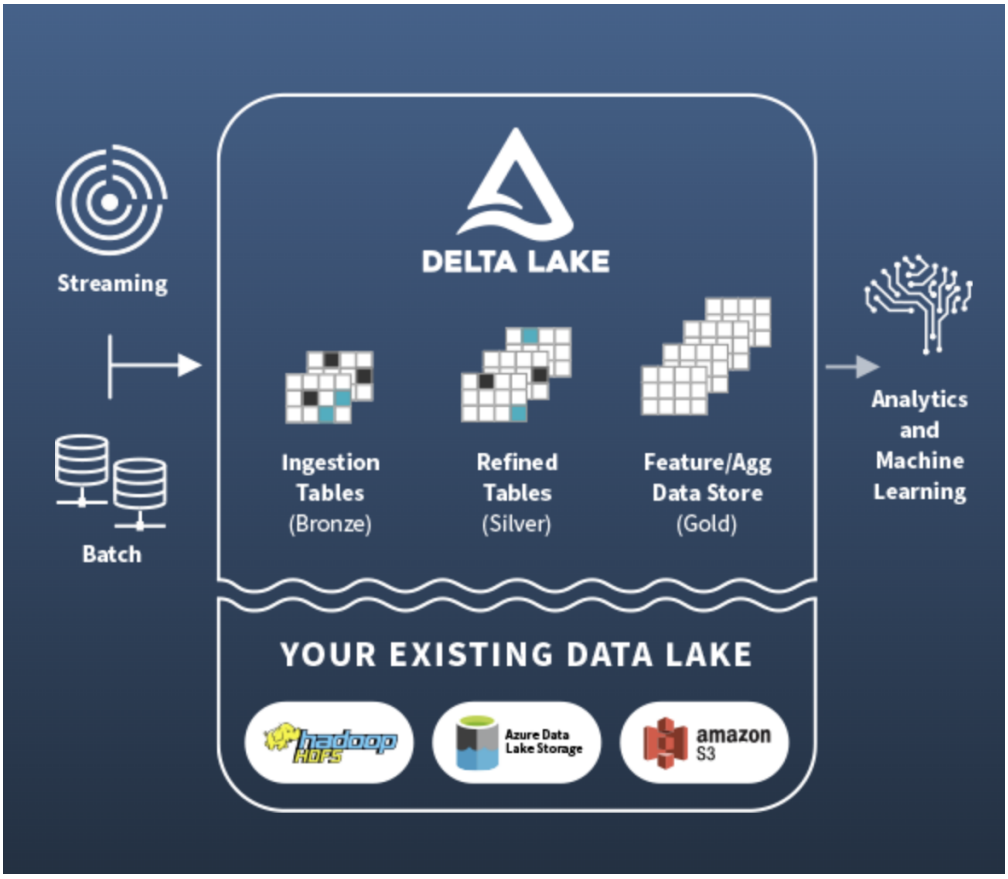
In this lesson you will:

- Create a Delta Table
- Read data from your Delta Table
- Update data in your Delta Table
- Access previous versions of your Delta Table using time travel (https://databricks.com/blog/2019/02/04/introducing-delta-time-travel-for-large-scale-data-lakes.html)
- Understand the Transaction Log (https://databricks.com/blog/2019/08/21/diving-into-delta-lake-unpacking-the-transaction-log.html)

In this notebook we will be using the SF Airbnb rental dataset from Inside Airbnb (http://insideairbnb.com/get-the-data.html).

--i18n-68fcecd4-2280-411c-94c1-3e111683c6a3

###Why Delta Lake?



At a glance, Delta Lake is an open source storage layer that brings both **reliability and performance** to data lakes. Delta Lake provides ACID transactions, scalable metadata handling, and unifies streaming and batch data processing.

Delta Lake runs on top of your existing data lake and is fully compatible with Apache Spark APIs. [For more information (https://docs.databricks.com/delta/delta-intro.html)](https://docs.databricks.com/delta/delta-intro.html)

In [0]:

```
%run "./Includes/Classroom-Setup"
```

Python interpreter will be restarted. Python interpreter will be restarted.

In [0]:

```
!pip install mlflow
```

```
Collecting mlflow Downloading mlflow-1.28.0-py3-none-any.whl (17.0 MB) | 10 kB 3.7 MB/s
eta 0:00:05 | | 20 kB 1.3 MB/s eta 0:00:14 | | 30 kB 1.9 MB/s eta 0:00:09 | | 40 kB 767 kB/s
eta 0:00:23 | | 51 kB 828 kB/s eta 0:00:21 ▌ | 61 kB 991 kB/s eta 0:00:18 ▌ | 71 kB 1.1
MB/s eta 0:00:16 ▌ | 81 kB 1.1 MB/s eta 0:00:16 ▌ | 92 kB 1.3 MB/s eta 0:00:14 ▌ | 102
kB 981 kB/s eta 0:00:18 ▌ | 112 kB 981 kB/s eta 0:00:18 ▌ | 122 kB 981 kB/s eta 0:00:18 |
▌ | 133 kB 981 kB/s eta 0:00:18 ▌ | 143 kB 981 kB/s eta 0:00:18 ▌ | 153 kB 981 kB/s eta
0:00:18 ▌ | 163 kB 981 kB/s eta 0:00:18 ▌ | 174 kB 981 kB/s eta 0:00:18 ▌ | 184 kB 981
kB/s eta 0:00:18 ▌ | 194 kB 981 kB/s eta 0:00:18 ▌ | 204 kB 981 kB/s eta 0:00:18 ▌ | 215
kB 981 kB/s eta 0:00:18 ▌ | 225 kB 981 kB/s eta 0:00:18 ▌ | 235 kB 981 kB/s eta 0:00:18 |
▌ | 245 kB 981 kB/s eta 0:00:18 ▌ | 256 kB 981 kB/s eta 0:00:18 ▌ | 266 kB 981 kB/s eta
0:00:18 ▌ | 276 kB 981 kB/s eta 0:00:18 ▌ | 286 kB 981 kB/s eta 0:00:18 ▌ | 296 kB 981
kB/s eta 0:00:17 ▌ | 307 kB 981 kB/s eta 0:00:17 ▌ | 317 kB 981 kB/s eta 0:00:17 ▌ | 327
kB 981 kB/s eta 0:00:17 ▌ | 337 kB 981 kB/s eta 0:00:17 ▌ | 348 kB 981 kB/s eta 0:00:17 |
▌ | 358 kB 981 kB/s eta 0:00:17 ▌ | 368 kB 981 kB/s eta 0:00:17 ▌ | 378 kB 981 kB/s eta
0:00:17 ▌ | 389 kB 981 kB/s eta 0:00:17 ▌ | 399 kB 981 kB/s eta 0:00:17 ▌ | 409 kB 981
kB/s eta 0:00:17 ▌ | 419 kB 981 kB/s eta 0:00:17 ▌ | 430 kB 981 kB/s eta 0:00:17 ▌ | 440
kB 981 kB/s eta 0:00:17 ▌ | 450 kB 981 kB/s eta 0:00:17 ▌ | 460 kB 981 kB/s eta 0:00:17 |
```

--i18n-8ce92b68-6e6c-4fd0-8d3c-a57f27e5bdd9

###Creating a Delta Table First we need to read the Airbnb dataset as a Spark DataFrame

In [0]:

```
file_path = f"{DA.paths.datasets}/airbnb/sf-listings/sf-listings-2019-03-06-clean.p
airbnb_df = spark.read.format("parquet").load(file_path)

display(airbnb_df)
```

host_is_superhost	cancellation_policy	instant_bookable	host_total_listings_count	neighbourhood_clean
t	moderate	t	1.0	Western Addi
f	strict_14_with_grace_period	f	2.0	Bernal Heig
f	strict_14_with_grace_period	f	10.0	Haight Ashb
f	strict_14_with_grace_period	f	10.0	Haight Ashb
f	strict_14_with_grace_period	f	2.0	Western Addi
f	moderate	f	1.0	Western Addi
t	strict_14_with_grace_period	t	2.0	Miss

--i18n-c100b529-ac6b-4540-a3ff-4afa63577eee

The cell below converts the data to a Delta table using the schema provided by the Spark DataFrame.

In [0]:

```
# Converting Spark DataFrame to Delta Table
dbutils.fs.rm(DA.paths.working_dir, True)
airbnb_df.write.format("delta").mode("overwrite").save(DA.paths.working_dir)
```

--i18n-090a31f6-1082-44cf-8e2a-6c659ea796ea

A Delta directory can also be registered as a table in the metastore.

In [0]:

```
spark.sql(f"CREATE DATABASE IF NOT EXISTS {DA.cleaned_username}")
spark.sql(f"USE {DA.cleaned_username}")

airbnb_df.write.format("delta").mode("overwrite").saveAsTable("delta_review")
```

--i18n-732577c2-095d-4278-8466-74e494a9c1bd

Delta supports partitioning. Partitioning puts data with the same value for the partitioned column into its own directory. Operations with a filter on the partitioned column will only read directories that match the filter. This optimization is called partition pruning. Choose partition columns based in the patterns in your data, this dataset for example might benefit if partitioned by neighborhood.

In [0]:

```
airbnb_df.write.format("delta").mode("overwrite").partitionBy("neighbourhood_cleans
```

--i18n-e9ce863b-5761-4676-ae0b-95f3f5f027f6

###Understanding the Transaction Log (https://databricks.com/blog/2019/08/21/diving-into-delta-lake-unpacking-the-transaction-log.html) Let's take a look at the Delta Transaction Log. We can see how Delta stores the different neighborhood partitions in separate files. Additionally, we can also see a directory called _delta_log.

In [0]:

```
display(dbutils.fs.ls(DA.paths.working_dir))
```

path	name	size	modification
dbfs:/mnt/dbacademy-users/devesh.vijay@celebaltech.com/scalable-machine-learning-with-apache-spark/_delta_log/	_delta_log/	0	166168406
dbfs:/mnt/dbacademy-users/devesh.vijay@celebaltech.com/scalable-machine-learning-with-apache-spark/neighbourhood_cleansed=Bayview/	neighbourhood_cleansed=Bayview/	0	166168405
dbfs:/mnt/dbacademy-users/devesh.vijay@celebaltech.com/scalable-machine-learning-with-apache-spark/neighbourhood_cleansed=Bernal Heights/	neighbourhood_cleansed=Bernal Heights/	0	166168405
dbfs:/mnt/dbacademy-users/devesh.vijay@celebaltech.com/scalable-machine-learning-with-apache-spark/neighbourhood_cleansed=Castro%2FUpper Market/	neighbourhood_cleansed=Castro%2FUpper Market/	0	166168405
dbfs:/mnt/dbacademy-			

--i18n-ac970bba-1cf6-4aa3-91bb-74a797496eef

When a user creates a Delta Lake table, that table's transaction log is automatically created in the _delta_log subdirectory. As he or she makes changes to that table, those changes are recorded as ordered, atomic commits in the transaction log. Each commit is written out as a JSON file, starting with 000000.json. Additional changes to the table generate more JSON files.

In [0]:

```
display(dbutils.fs.ls(f"{DA.paths.working_dir}/_delta_log/"))
```

--i18n-2905b874-373b-493d-9084-8ff4f7583ccc

Next, let's take a look at a Transaction Log File.

The four columns (https://docs.databricks.com/delta/delta-utility.html) each represent a different part of the very first commit to the Delta Table where the table was created.

- The add column has statistics about the DataFrame as a whole and individual columns.
- The commitInfo column has useful information about what the operation was (WRITE or READ) and who executed the operation.
- The metaData column contains information about the column schema.
- The protocol version contains information about the minimum Delta version necessary to either write or read to this Delta Table.

In [0]:

```
display(spark.read.json(f"{DA.paths.working_dir}/_delta_log/00000000000000000000.js
```

---------- NameError Traceback (most recent call last) <command-3596561756563669> in <module> ----> 1 display(spark.read.json(f"{DA.paths.working_dir}/_delta_log/00000000000000000000.json")) NameError: name 'DA' is not defined

--i18n-8f79d1df-d777-4364-9783-b52bc0eed81a

The second transaction log has 39 rows. This includes metadata for each partition.

In [0]:

```
display(spark.read.json(f"{DA.paths.working_dir}/_delta_log/00000000000000000001.js
```

--i18n-18500df8-b905-4f24-957c-58040920d554

Finally, let's take a look at the files inside one of the Neighborhood partitions. The file inside corresponds to the partition commit (file 01) in the _delta_log directory.

In [0]:

```
display(dbutils.fs.ls(f"{DA.paths.working_dir}/neighbourhood_cleansed=Bayview/"))
```

--i18n-9f817cd0-87ec-457b-8776-3fc275521868

Reading data from your Delta table

In [0]:

```
df = spark.read.format("delta").load(DA.paths.working_dir)
display(df)
```

--i18n-faba817b-7cbf-49d4-a32c-36a40f582021

#Updating your Delta Table

Let's filter for rows where the host is a superhost.

In [0]:

```
df_update = airbnb_df.filter(airbnb_df["host_is_superhost"] == "t")
display(df_update)
```

In [0]:

```
df_update.write.format("delta").mode("overwrite").save(DA.paths.working_dir)
```

In [0]:

```
df = spark.read.format("delta").load(DA.paths.working_dir)
display(df)
```

--i18n-e4cafdf4-a346-4729-81a6-fdea70f4929a

Let's look at the files in the Bayview partition post-update. Remember, the different files in this directory are snapshots of your DataFrame corresponding to different commits.

In [0]:

```
display(dbutils.fs.ls(f"{DA.paths.working_dir}/neighbourhood_cleansed=Bayview/"))
```

--i18n-25ca7489-8077-4b23-96af-8d801982367c

#Delta Time Travel

--i18n-c6f2e771-502d-46ed-b8d4-b02e3e4f4134

Oops, actually we need the entire dataset! You can access a previous version of your Delta Table using Delta Time Travel (https://databricks.com/blog/2019/02/04/introducing-delta-time-travel-for-large-scale-data-lakes.html). Use the following two cells to access your version history. Delta Lake will keep a 30 day version history by default, though it can maintain that history for longer if needed.

In [0]:

```
%sql
DROP TABLE IF EXISTS train_delta;
CREATE TABLE train_delta USING DELTA LOCATION '${DA.paths.working_dir}'
```

In [0]:

```
%sql
DESCRIBE HISTORY train_delta
```

--i18n-61faa23f-d940-479c-95fe-5aba72c29ddf

Using the **`versionAsOf`** option allows you to easily access previous versions of our Delta Table.

In [0]:

```
df = spark.read.format("delta").option("versionAsOf", 0).load(DA.paths.working_dir)
display(df)
```

--i18n-5664be65-8fd2-4746-8065-35ee8b563797

You can also access older versions using a timestamp.

Replace the timestamp string with the information from your version history. Note that you can use a date without the time information if necessary.

In [0]:

```
# Use your own timestamp
# time_stamp_string = "FILL_IN"

# OR programatically get the first verion's timestamp value
time_stamp_string = str(spark.sql("DESCRIBE HISTORY train_delta").collect()[-1]["ti

df = spark.read.format("delta").option("timestampAsOf", time_stamp_string).load(DA.
display(df)
```

--i18n-6cbe5204-fe27-438a-af54-87492c2563b5

Now that we're happy with our Delta Table, we can clean up our directory using **`VACUUM`** . Vacuum accepts a retention period in hours as an input.

--i18n-4da7827c-b312-4b66-8466-f0245f3787f4

Uh-oh, our code doesn't run! By default, to prevent accidentally vacuuming recent commits, Delta Lake will not let users vacuum a period under 7 days or 168 hours. Once vacuumed, you cannot return to a prior commit through time travel, only your most recent Delta Table will be saved.

Try changing the vacuum parameter to different values.

In [0]:

```
# from delta.tables import DeltaTable

# delta_table = DeltaTable.forPath(spark, DA.paths.working_dir)
# delta_table.vacuum(0)
```

--i18n-1150e320-5ed2-4a38-b39f-b63157bca94f

We can workaround this by setting a spark configuration that will bypass the default retention period check.

In [0]:

```
from delta.tables import DeltaTable

spark.conf.set("spark.databricks.delta.retentionDurationCheck.enabled", "false")
delta_table = DeltaTable.forPath(spark, DA.paths.working_dir)
delta_table.vacuum(0)
```

--i18n-b845b2ea-2c11-4d6e-b083-d5908b65d313

Let's take a look at our Delta Table files now. After vacuuming, the directory only holds the partition of our most recent Delta Table commit.

In [0]:

```
display(dbutils.fs.ls(f"{DA.paths.working_dir}/neighbourhood_cleansed=Bayview/"))
```

--i18n-a7bcdad3-affb-4b00-b791-07c14f5e59d5

Since vacuuming deletes files referenced by the Delta Table, we can no longer access past versions. The code below should throw an error.

In [0]:

```
# df = spark.read.format("delta").option("versionAsOf", 0).load(DA.paths.working_di
# display(df)
```