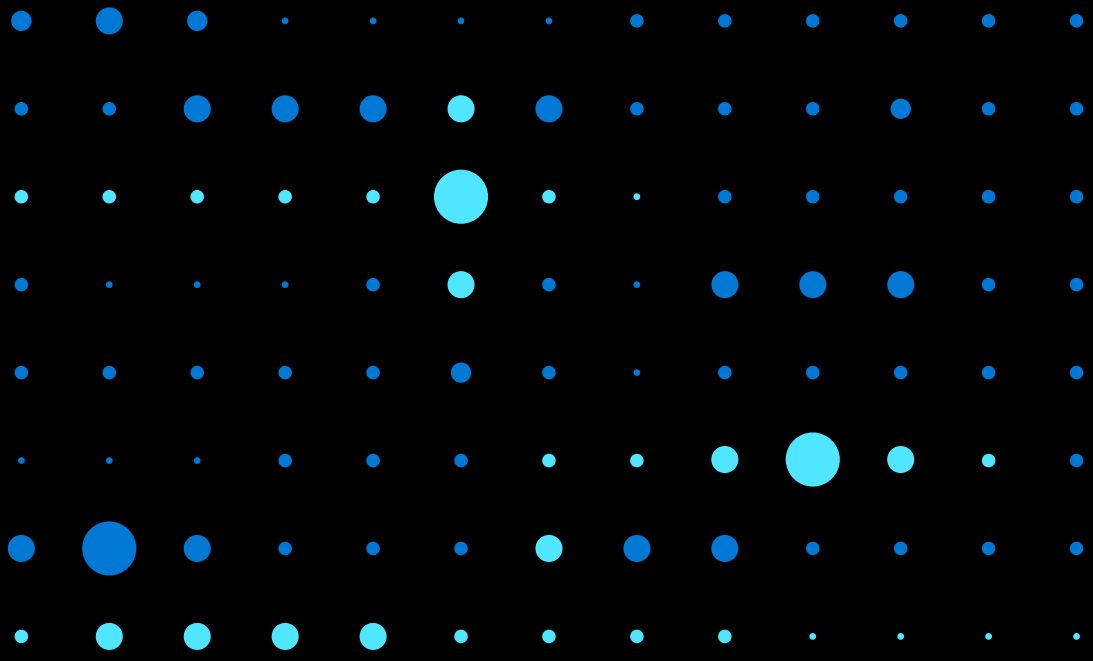
Azure Databricks

Four real-life Machine Learning use cases



Microsoft Azure

Contents

Introduction

The potential for machine learning and deep learning practitioners to make a breakthrough and drive positive outcomes is unprecedented. A myriad of data and ML tools is readily available. However, you might be wondering how to take advantage of these tools and scale model training on big data for real-life scenarios.

Azure Databricks is a cloud-service designed to provide ready-to-use clusters that can handle all analytics processes in one place, from data preparation to model building and serving, with virtually no limit so that you can scale resources as needed.

This guide walks you through four practical end-to-end machine-learning use cases on Azure Databricks:

- A loan risk analysis use case that covers importing and exploring data in Databricks, executing ETL and the ML pipeline, including model tuning with XGBoost Logistic Regression
- An advertising analytics and click prediction use case that includes collecting and exploring the advertising logs with Spark SQL and using PySpark for feature engineering and using GBTClassifier for model training and predicting the clicks
- A market basket analysis problem at scale, from ETL to data exploration using Spark SQL, and model training using FT-growth
- An example of suspicious behavior identification in videos, including a pre-processing step for creating image frames, transfer learning for featurization, and applying logistic regression to identify suspicious images in a video.

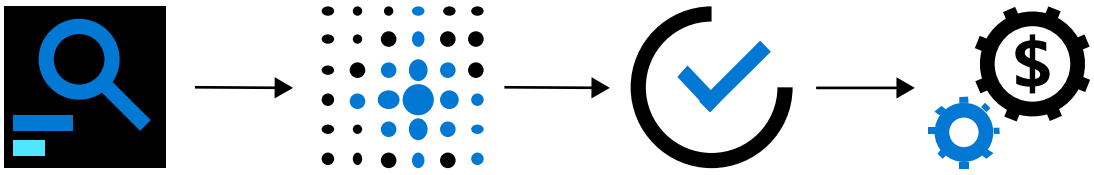
Use case 1: Loan risk analysis with XGBoost

For companies that make money from loan interest, increasing the bottom line is always the goal. Being able to assess the risk of loan applications can save a lender the cost of holding too many risky assets. Data scientists at these companies run analysis on customer data and develop business rules that will directly impact loan approval.

The data scientists who spend their time building these machine learning models are a scarce resource and far too often they are siloed into a sandbox:

- Although they work with data day in and out, they depend on data engineers to obtain up-to-date tables.
- With data growing at an exponential rate, they depend on an infrastructure team to provision compute resources.
- Once the model building process is done, they must trust software developers to correctly translate their model code to production ready code.

Azure Databricks can help bridge the gaps and smooth the workflow chain.



In addition to reducing operational friction, Azure Databricks is a central location for running the latest ML models. Users can employ the native Spark MLLib package or download any open source Python or R ML package. Azure Databricks Runtime for Machine Learning, preconfigures Databricks clusters with XGBoost, scikit-learn, and numpy, as well as popular deep learning frameworks such as TensorFlow, Keras, Horovod, and their dependencies.

Use case 1: Loan risk analysis with XGBoost

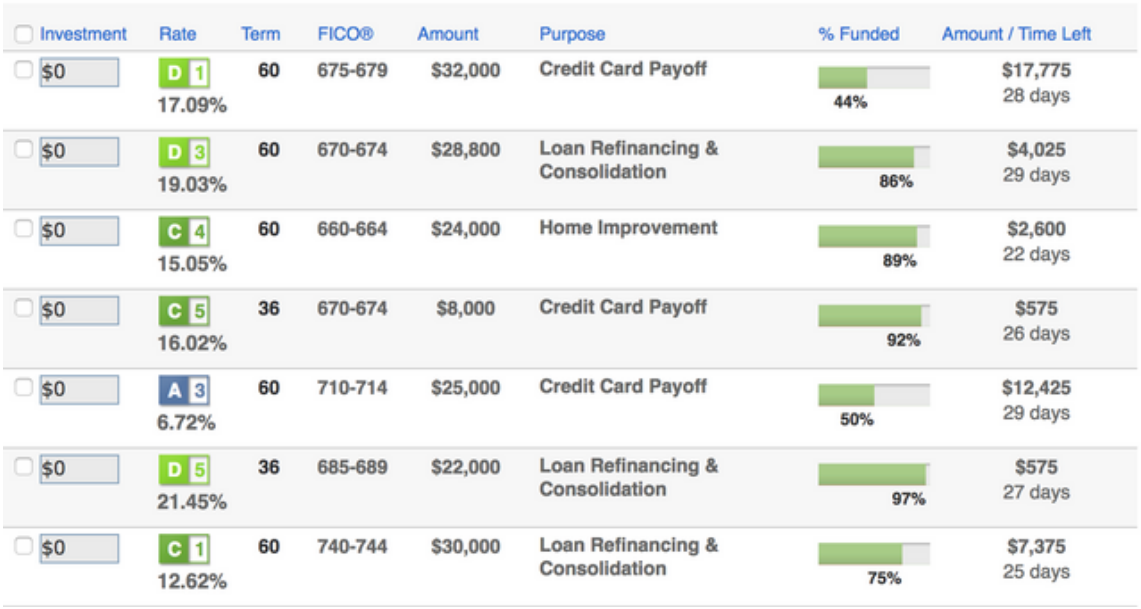
In this loan risk analysis use case, we explore how to:

- Import a sample data source to create an Azure Databricks table.
- Explore your data using Azure Databricks visualizations.
- Execute ETL code against your data.
- Execute ML Pipeline including model tuning XGBoost Logistic Regression.

Import data

This experiment uses public Lending Club loan data, which includes all funded loans from 2012 to 2017. Each loan includes applicant information provided by the applicant, the current loan status (Current, Late, Fully Paid, etc.), and the latest payment information. For more information, refer to the Lending Club Data schema.

Once you have downloaded the data locally, you can create a database and table in the Azure Databricks workspace to load this dataset. For more information, refer to Azure Databricks Documentation > Databases and Tables > Create a Table.



Investment	Rate	Term	FICO®	Amount	Purpose	% Funded	Amount / Time Left
$0	D 1 17.09%	60	675-679	$32,000	Credit Card Payoff	44%	$17,775 28 days
$0	D 3 19.03%	60	670-674	$28,800	Loan Refinancing & Consolidation	86%	$4,025 29 days
$0	C 4 15.05%	60	660-664	$24,000	Home Improvement	89%	$2,600 22 days
$0	C 5 16.02%	36	670-674	$8,000	Credit Card Payoff	92%	$575 26 days
$0	A 3 6.72%	60	710-714	$25,000	Credit Card Payoff	50%	$12,425 29 days
$0	D 5 21.45%	36	685-689	$22,000	Loan Refinancing & Consolidation	97%	$575 27 days
$0	C 1 12.62%	60	740-744	$30,000	Loan Refinancing & Consolidation	75%	$7,375 25 days

Use case 1: Loan risk analysis with XGBoost

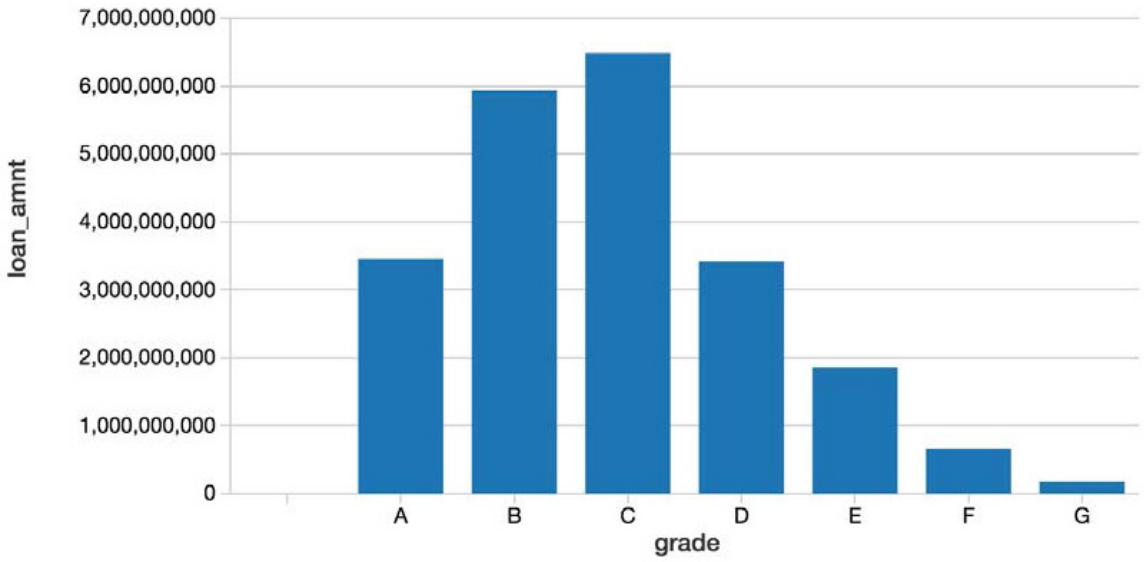
In this case, we have created the Azure Databricks Database `amy` and table `loanstats_2012_2017`. The following code snippet allows you to access this table in an Azure Databricks notebook via PySpark.

```
# Import loan statistics table
loan_stats = spark.table("amy.loanstats_2012_2017")
```

Explore your data

With the Databricks `display` command, you use Azure Databricks native visualizations. In this case, you can view the asset allocations by reviewing the loan grade and the loan amount.

```
# View bar graph of our data
display(loan_stats)
```



Munging your data with the PySpark DataFrame API

As noted in Cleaning Big Data (Forbes), 80% of a Data Scientist's work is data preparation and is often the least enjoyable aspect of the job. But with PySpark, you can write Spark SQL statements or use the PySpark DataFrame API to streamline your data preparation tasks.

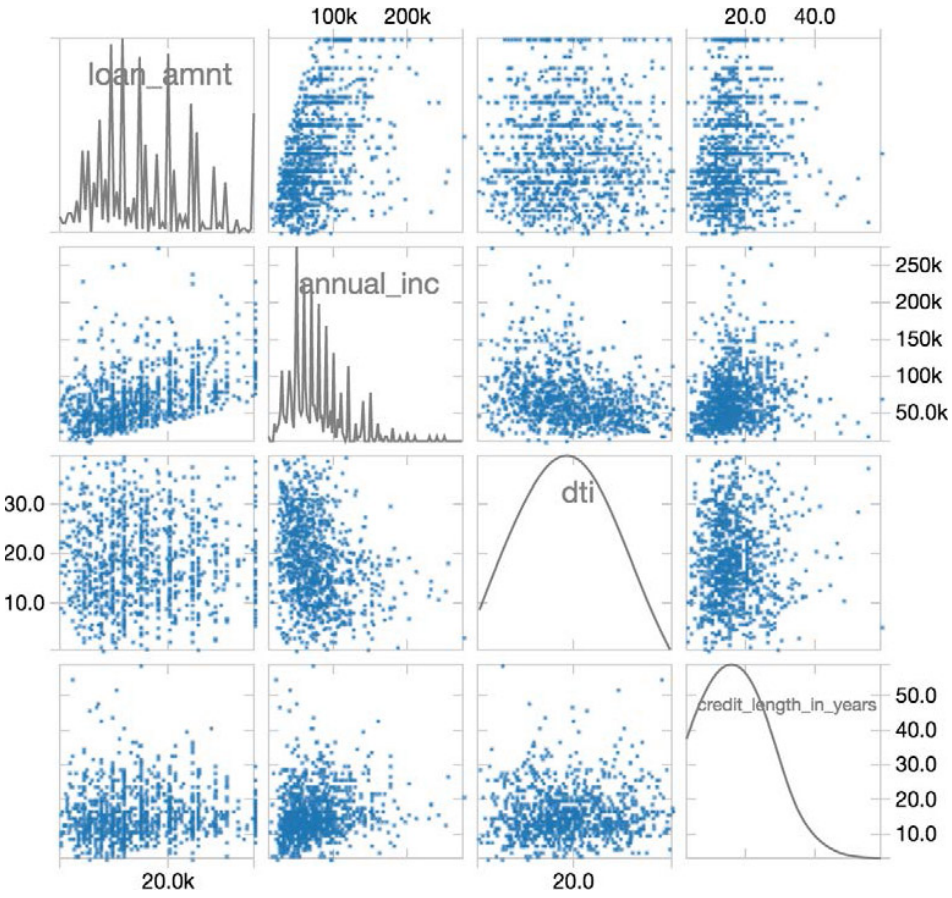
Use case 1: Loan risk analysis with XGBoost

This code snippet can simplify the filtering of your data.

```
# Import loan statistics table
loan_stats = loan_stats.filter( \
             loan_stats.loan_status.isin( \
                    [“Default”, “Charged Off”, “Fully Paid”]
                    )\
       ).withColumn(
              “bad_loan”,
              (~(loan_stats.loan_status == “Fully Paid”)
       ).cast(“string”))
```

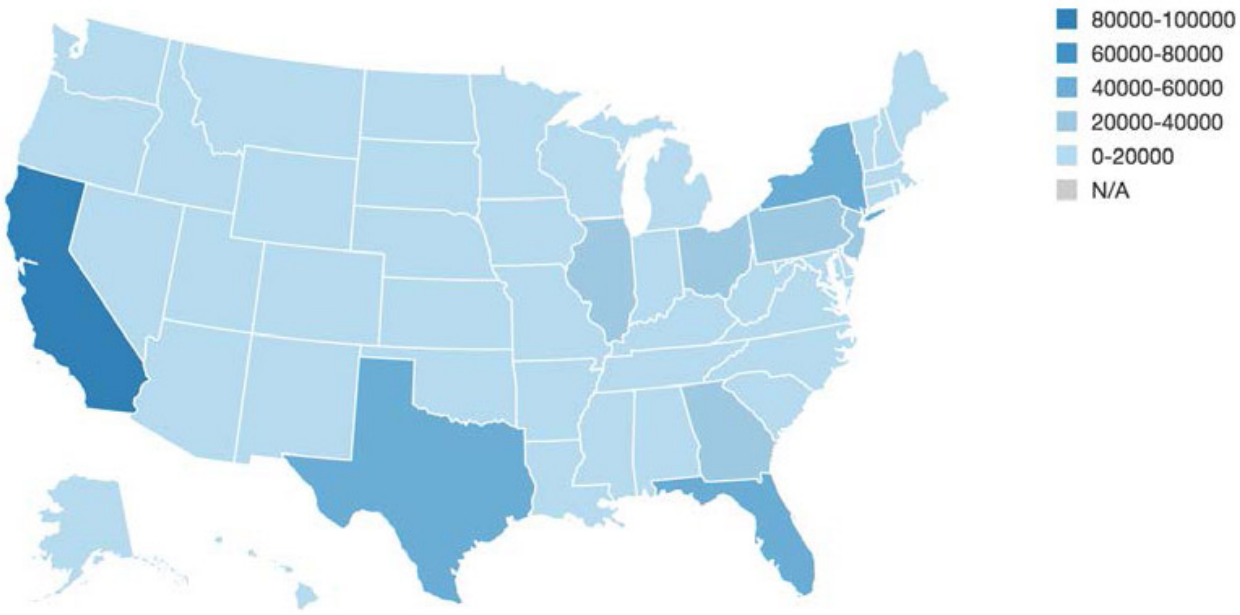
After this ETL process is completed, you can use the `display` command to review the cleansed data in a scatterplot.

```
# View bar graph of our data
display(loan_stats)
```



To view this same asset data broken out by state on a map visualization, you can use the `display` command combined with the PySpark DataFrame API, using group-by statements with agg (aggregations) such as the following code snippet.

```
# View map of our asset data
display(loan_stats.groupBy("addr_state").agg((count(col("annual_inc"))).
alias("ratio")))
```



Training the ML model using XGBoost

While you can quickly visualize the asset data, an ML model that will allow you to predict if a loan is good or bad based on the available parameters would be even better. As noted in the following code snippet, to predict `bad_loan` (defined as `label`), build the ML pipeline as follows:

- Execute an `imputer` to fill in missing values within the `numerics` attributes (output is `numerics_out`).
- Use indexers to handle the categorical values and then convert them to vectors using `OneHotEncoder` via `oneHotEncoders` (output is `categoricals_class`).
- Define the `features` for the ML pipeline by combining the `categorical_class` and `numerics_out`.
- Assemble the features together by executing the `VectorAssembler`.

Use case 1: Loan risk analysis with XGBoost

- Establish the `label` (i.e., what you are going to try to predict) as the `bad_loan` column.
- Apply the standard scaler to build the pipeline array (`pipelineAry`) before establishing which algorithm to apply.

While the previous code snippets are in Python, the following code examples are written in Scala so we can use XGBoost4J-Spark. The notebook series includes Python code that saves the data in Parquet and subsequently reads the data in Scala.

```
// Imputation estimator for completing missing values
val numerics_out = numerics.map(_ + "_out")
val imputers = new Imputer()
  .setInputCols(numerics)
  .setOutputCols(numerics_out)

// Apply StringIndexer for our categorical data
val categoricals_idx = categoricals.map(_ + "_idx")
val indexers = categoricals.map(
  x => new StringIndexer().setInputCol(x).setOutputCol(x + "_idx").
setHandleInvalid("keep")
)

// Apply OHE for our StringIndexed categorical data
val categoricals_class = categoricals.map(_ + "_class")
val oneHotEncoders = new OneHotEncoderEstimator()
  .setInputCols(categoricals_idx)
  .setOutputCols(categoricals_class)

// Set feature columns
val featureCols = categoricals_class ++ numerics_out

// Create assembler for our numeric columns (including label)
val assembler =  newVectorAssembler()
  .setInputCols(featureCols)
  .setOutputCol("features")

// Establish label
val labelIndexer = new StringIndexer()
  .setInputCol("bad_loan")
  .setOutputCol("label")
```

Use case 1: Loan risk analysis with XGBoost

```
// Apply StandardScaler
val scaler = new StandardScaler()
  .setInputCol("features")
  .setOutputCol("scaledFeatures")
  .setWithMean(true)
  .setWithStd(true)

// Build pipeline array
val pipelineAry = indexers ++ Array(oneHotEncoders, imputers, assembler,
labelIndexer, scaler)
```

Now that the ML pipeline is established, you can create the XGBoost pipeline and apply it to the training dataset.

```
// Create XGBoostEstimator
val xgBoostEstimator = new XGBoostEstimator(
      Map[String, Any](
        "num_round" -> 5,
        "objective" -> "binary:logistic",
        "nworkers" -> 16,
        "nthreads" -> 4
      )
  )  .setFeaturesCol("scaledFeatures")
  .setLabelCol("label")

// Create XGBoost Pipeline
val xgBoostPipeline = new Pipeline().setStages(pipelineAry ++
Array(xgBoostEstimator))

// Create XGBoost Model based on the training dataset
val xgBoostModel = xgBoostPipeline.fit(dataset_train)

// Test our model against the validation dataset
val predictions = xgBoostModel.transform(dataset_valid)
display(predictions.select("probabilities", "label"))
```

Note, that `"nworkers" -> 16, "nthreads" -> 4` is configured as the instances used were 16 i3.xlarges.

Once the model is created, you can test it against the validation dataset with `predictions` containing the result.

Reviewing model efficacy

You can use the `BinaryClassficationEvaluator` to determine the efficacy of your XGBoost model.

```
// Include BinaryClassificationEvaluator
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator

// Evaluate
val evaluator = new BinaryClassificationEvaluator()
       .setRawPredictionCol("probabilities")

// Calculate Validation AUC
val accuracy = evaluator.evaluate(predictions)
```

Upon calculation, the XGBoost validation data area-under-curve (AUC) is: ~0.6520.

Tune the model using MLlib cross validation

To tune the model using MLlib cross validation via `CrossValidator` as noted in the following code snippet, first establish the parameter grid so you can execute multiple runs with the grid of different parameter values. Using the same `BinaryClassificationEvaluator` that we had used to test the model efficacy, apply this at a larger scale with different parameters by combining the `BinaryClassificationEvaluator` and `ParamGridBuilder` and applying it to `CrossValidator()`.

```
// Build parameter grid
val paramGrid = new ParamGridBuilder()
       .addGrid(xgBoostEstimator.maxDepth, Array(4, 7))
       .addGrid(xgBoostEstimator.eta, Array(0.1, 0.6))
       .addGrid(xgBoostEstimator.round, Array(5, 10))
       .build()

// Set evaluator as a BinaryClassificationEvaluator
val evaluator = new BinaryClassificationEvaluator()
  .setRawPredictionCol("probabilities")
```

Use case 1: Loan risk analysis with XGBoost

```
// Establish CrossValidator()
val cv = new CrossValidator()
      .setEstimator(xgBoostPipeline)
      .setEvaluator(evaluator)
      .setEstimatorParamMaps(paramGrid)
      .setNumFolds(4)

// Run cross-validation, and choose the best set of parameters.
val cvModel = cv.fit(dataset_train)
```

Note, for the initial configuration of the XGBoostEstimator, we use num_round but we use round (num_round is not an attribute in the estimator)

This code snippet will run the cross-validation and choose the best set of parameters. You can then re-run your predictions and re-calculate the accuracy. Our accuracy increased slightly with a value ~0.6734.

```
// Test our model against the cvModel and validation dataset
val predictions_cv = cvModel.transform(dataset_valid)display(predictions_
cv.select("probabilities", "label"))

// Calculate cvModel Validation AUC
val accuracy = evaluator.evaluate(predictions_cv)
```

You can also review the bestModel parameters by running the following snippet.

```
// Review bestModel parameters
cvModel.bestModel.asInstanceOf[PipelineModel].stages(11).extractParamMap
```

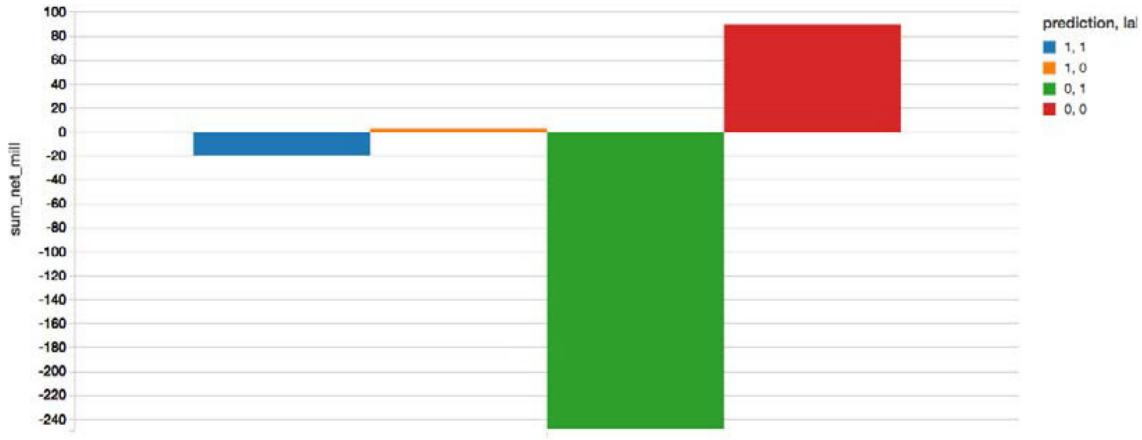
Quantify the business value

A great way to quickly understand the business value of this model is to create a confusion matrix with these definitions:

- Label=1, Prediction=1 :
  Correctly found bad loans. sum_net = loss avoided.
- Label=0, Prediction=1 :
  Incorrectly labeled bad loans. sum_net = profit forfeited.
- Label=1, Prediction=0 :
  Incorrectly labeled good loans. sum_net = loss still incurred.
- Label=0, Prediction=0 :
  Correctly found good loans. sum_net = profit retained.

The following code snippet calculates the following confusion matrix.

```
display(predictions_cv.groupBy("label", "prediction").
agg((sum(col("net"))/(1E6)).alias("sum_net_mill")))
```



To determine the value gained from implementing the model, you can calculate this as:

```
value = -(loss avoided - profit forfeited)
```

In the case of the current XGBoost model with AUC = ~0.6734, the values note the significant value gain from implementing the XGBoost model.

- value (XGBoost): 22.076

Use case 1: Loan risk analysis with XGBoost

Note, the value referenced here is in terms of millions of dollars saved from prevent lost to bad loans.

Summary

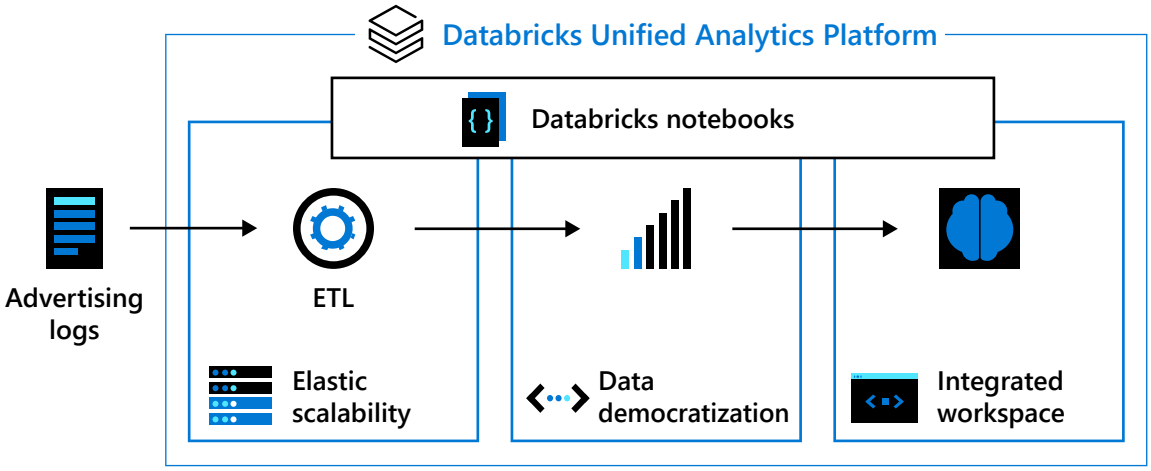
We demonstrated how you can quickly perform loan risk analysis using Azure Databricks, which include the Azure Databricks Runtime for Machine Learning. With Databricks Runtime for Machine Learning, Databricks clusters are preconfigured with XGBoost, scikit-learn, and numpy, as well as popular Deep Learning frameworks such as TensorFlow, Keras, Horovod, and their dependencies.

By removing the data engineering complexity commonly associated with such data pipelines, you can quickly import your data source into a Databricks table, explore your data using Azure Databricks visualizations, execute ETL code against your data, and build, train, and tune your ML pipeline using XGBoost logistic regression.

Use case 2:
Advertising analytics click prediction

Advertising teams want to analyze their immense data stores and varieties, which requires a scalable, extensible, and elastic platform. Advanced analytics, including classification, clustering, recognition, prediction, and recommendations, allows these organizations to gain deeper insights from their data and drive better business outcomes.

As various types of data grow in volume, Apache Spark provides an API and distributed compute engine to process data easily and in parallel, thereby decreasing time to value. Azure provides an optimized, managed cloud service around Spark and allows for self-service provisioning of computing resources and a collaborative workspace.



This use case offers a concrete example of advanced analytics for advertising, using the Click-Through Rate Prediction dataset of ad impressions and clicks from the data science website Kaggle. The goal of this workflow is to create a machine learning model that, given a new ad impression, predicts whether there will be a click. You can download this notebook to experiment.

Use case 2: Advertising analytics click prediction

Building this advanced analytics workflow involves three main steps:

- ETL
- Data exploration, for example, using SQL
- Advanced analytics and machine learning

Building the ETL process for the advertising logs

First, download the dataset to Azure Blob storage and then read it into Spark.

```
%scala
// Read CSV files of our adtech dataset
val df = spark.read
  .option("header", true)
  .option("inferSchema", true)
  .csv("/mnt/adtech/impression/csv/train.csv/")
```

This creates a Spark DataFrame—an immutable, tabular, distributed data structure on your Spark cluster. The inferred schema can be seen using `.printSchema()`.

```
%scala
df.printSchema()

# Output
id: decimal(20,0)
click: integer
hour: integer
C1: integer
banner_pos: integer
site_id: string
site_domain: string
site_category: string
app_id: string
app_domain: string
app_category: string
device_id: string
device_ip: string
device_model: string
device_type: string
device_conn_type: integer
```

```
C14: integer
C15: integer
C16: integer
C17: integer
C18: integer
C19: integer
C20: integer
C21: integer
```

To optimize the query performance from DBFS, you can convert the CSV files into Parquet format. Parquet is a columnar file format that allows for efficient querying of big data with Spark SQL or most MPP query engines. For more information on how Spark is optimized for Parquet, refer to How Apache Spark performs a fast count using the Parquet metadata.

```
%scala
// Create Parquet files from our Spark DataFrame
df.coalesce(4)
  .write
  .mode("overwrite")
  .parquet("/mnt/adtech/impression/parquet/train.csv")
```

Explore advertising logs with Spark SQL

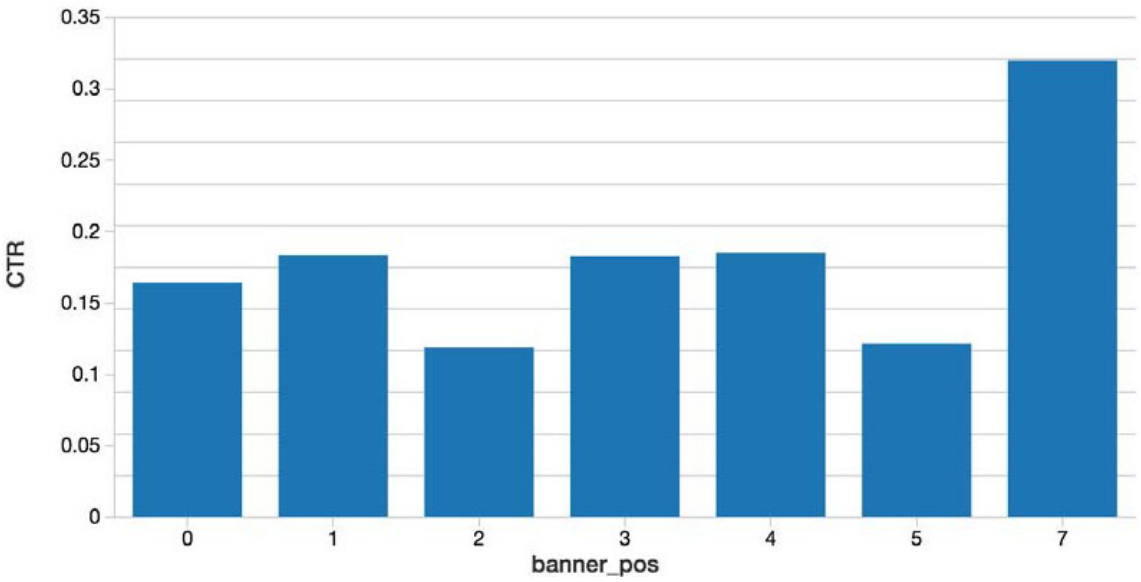
Now you can create a Spark SQL temporary view called impression on your Parquet files. To showcase the flexibility of Databricks notebooks, you can specify to use Python (instead of Scala) in another cell in your notebook.

```
%python
# Create Spark DataFrame reading the recently created Parquet files
impression = spark.read \\
.parquet("/mnt/adtech/impression/parquet/train.csv/")
# Create temporary view
impression.createOrReplaceTempView("impression")
```

Use case 2: Advertising analytics click prediction

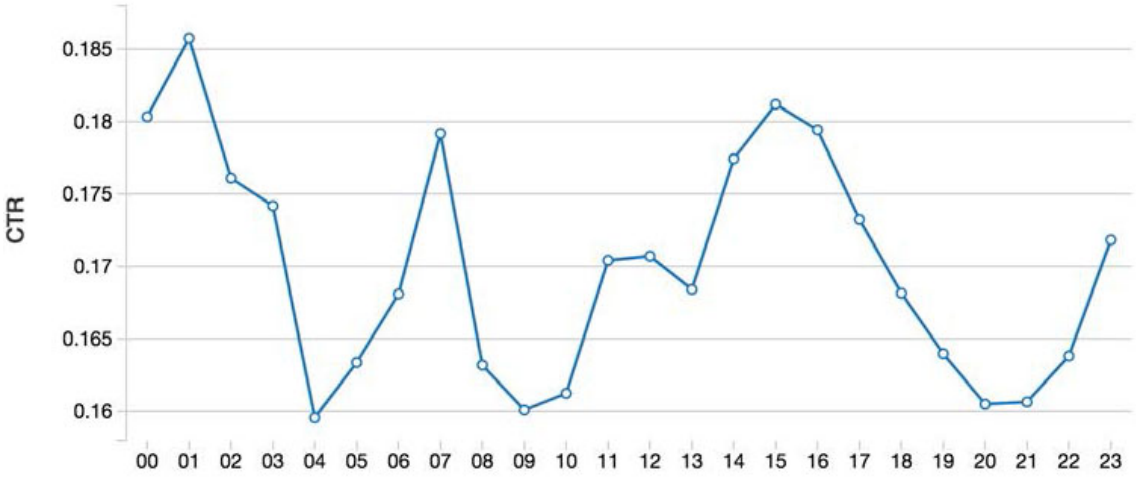
You can now explore your data with SQL. Azure Databricks and Spark support Scala, Python, R, and SQL. The following code snippets calculates the click-through rate (CTR) by banner position and hour of day.

```
%sql
-- Calculate CTR by Banner Position
select banner_pos,
sum(case when click = 1 then 1 else 0 end) / (count(1) * 1.0) as CTR
from impression
group by 1
order by 1
```



Use case 2: Advertising analytics click prediction

```
%sql
-- Calculate CTR by Hour of the day
select substr(hour, 7) as hour,
sum(case when click = 1 then 1 else 0 end) / (count(1) * 1.0) as CTR
from impression
group by 1
order by 1
```



Predict the clicks

After familiarizing yourself with the data, you can convert your data into features for input to a machine learning algorithm and produce a trained model you can use for prediction. Because Spark MLlib algorithms take a column of feature vectors of doubles as input, a typical feature engineering workflow includes:

- Identifying numeric and categorical features
- String indexing
- Assembling them all into a sparse vector

Use case 2: Advertising analytics click prediction

The following code snippet is an example of a feature engineering workflow.

```
# Include PySpark Feature Engineering methods
from pyspark.ml.feature import StringIndexer, VectorAssembler

# All of the columns (string or integer) are categorical columns
maxBins = 70
categorical = map(lambda c: c[0],
filter(lambda c: c[1] <= maxBins, strColsCount))
categorical += map(lambda c: c[0],
filter(lambda c: c[1] <= maxBins, intColsCount))

# remove ‘click’ which we are trying to predict
categorical.remove(‘click’)

# Apply string indexer to all of the categorical columns
#  And add _idx to the column name to indicate the index
#  of the categorical value
stringIndexers = map(lambda c: StringIndexer(inputCol = c, outputCol = c +
“_idx”), categorical)

# Assemble the put as the input to the VectorAssembler
#  with the output being our features
assemblerInputs = map(lambda c: c + “_idx”, categorical)vectorAssembler =
VectorAssembler(inputCols = assemblerInputs, outputCol = “features”)

# The [click] column is our label
labelStringIndexer = StringIndexer(inputCol = “click”, outputCol =
“label”)

# The stages of our ML pipeline
stages = stringIndexers + [vectorAssembler, labelStringIndexer]
```

This example uses a string indexer for GBTClassifer but does not apply OneHotEncoder (OHE).

When using StringIndexer, categorical features are kept as k-ary categorical features. A tree node will test if feature X has a value in {subset of categories}. With both StringIndexer + OHE: your categorical features are turned into a binary features. A tree node will test if feature X = category a vs. all the other categories (one vs. rest test).

When using only StringIndexer, the benefits include:

- *There are fewer features to choose.*
- *Each node’s test is more expressive than with binary 1-vs-rest features.*

Use case 2: Advertising analytics click prediction

Therefore, in the case of tree-based methods, OHE is a less expressive test and it takes up more space. But for non-tree- based algorithms such as like linear regression, you must use OHE or else the model will impose a false and misleading ordering on categories.

With the workflow created, you can create your ML pipeline.

```
from pyspark.ml import Pipeline

# Create our pipeline
pipeline = Pipeline(stages = stages)

# create transformer to add features
featurizer = pipeline.fit(impression)

# dataframe with feature and intermediate
# transformation columns appended
featurizedImpressions = featurizer.transform(impression)
```

Using `display(featurizedImpressions.select('features', 'label'))`, you can visualize the featurized dataset.

features	label
[0,12,[1,2,3,11],[2,1,1,1]]	0
[0,12,[1,2,3,11],[2,1,1,1]]	0
[0,12,[1,2,3,11],[2,1,1,1]]	0
[0,12,[1,2,3,11],[2,1,1,1]]	0
[0,12,[0,2,5,6,10,11],[1,8,1,1,1,2]]	0
[0,12,[1,2,3,5,6,11],[3,5,1,1,1,1]]	1
[0,12,[1,2,3,5,6,11],[3,5,1,1,1,1]]	1
[0,12,[1,2,3,5,6,11],[3,5,1,1,1,1]]	1
[0,12,[0,2,3,4,10,11],[3,11,26,11,1,3]]	1
[0,12,[0,2,3,4,11],[2,16,17,12,1]]	1
[0,12,[0,2,3,4,11],[2,16,17,12,1]]	1
[0,12,[0,2,3,4,11],[2,16,17,12,1]]	1
[0,12,[0,2,3,4,11],[2,16,17,12,1]]	1
[0,12,[0,2,4,11],[1,20,7,2]]	1

Showing the first 1000 rows.

Use case 2: Advertising analytics click prediction

Next, you split the featurized dataset into training and test datasets via `.randomSplit()`.

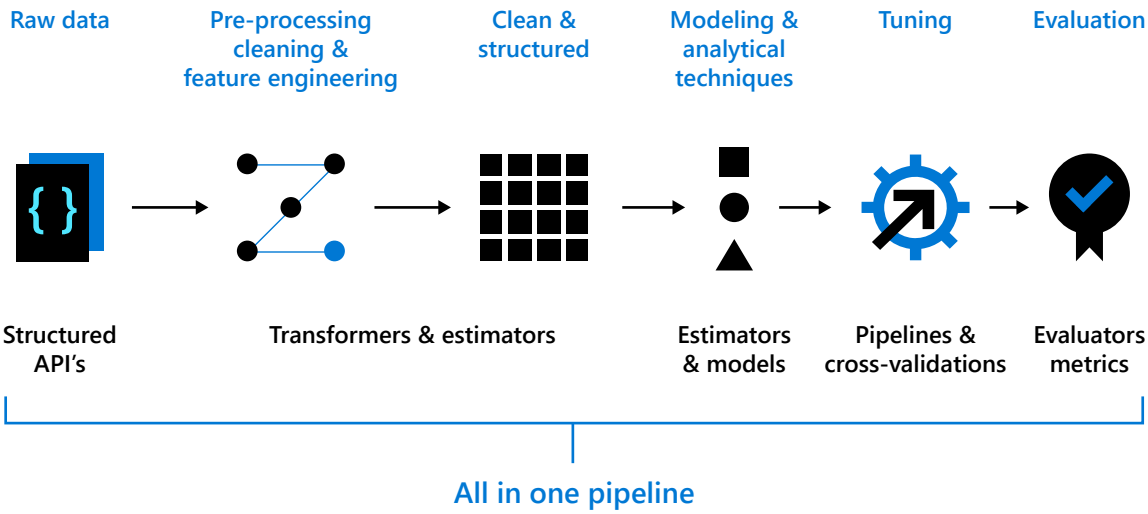
```
train, test = features \
  .select(["label", "features"]) \
  .randomSplit([0.7, 0.3], 42)
```

Now you can train, predict, and evaluate your model using the GBTClassifier. As a side note, a good primer on solving binary classification problems with Spark MLlib is Machine Learning with PySpark and MLlib — Solving a Binary Classification Problem.

```
from pyspark.ml.classification import GBTClassifier

# Train our GBTClassifier model
classifier = GBTClassifier(labelCol="label",
featuresCol="features", maxBins=maxBins, maxDepth=10, maxIter=10)
model = classifier.fit(train)

# Execute our predictions
predictions = model.transform(test)

# Evaluate our GBTClassifier model using
# BinaryClassificationEvaluator()
from pyspark.ml.evaluation import BinaryClassificationEvaluator
ev = BinaryClassificationEvaluator( \\
rawPredictionCol="rawPrediction", metricName="areaUnderROC")
print ev.evaluate(predictions)

# Output
0.7112027059
```

Summary

We demonstrated how you can simplify your advertising analytics—including click prediction—using Azure Databricks. You executed three components for click prediction: ETL, data exploration, and machine learning. You also saw how you can run the advanced analytics workflow of ETL, analysis, and machine learning pipelines all in a notebook.

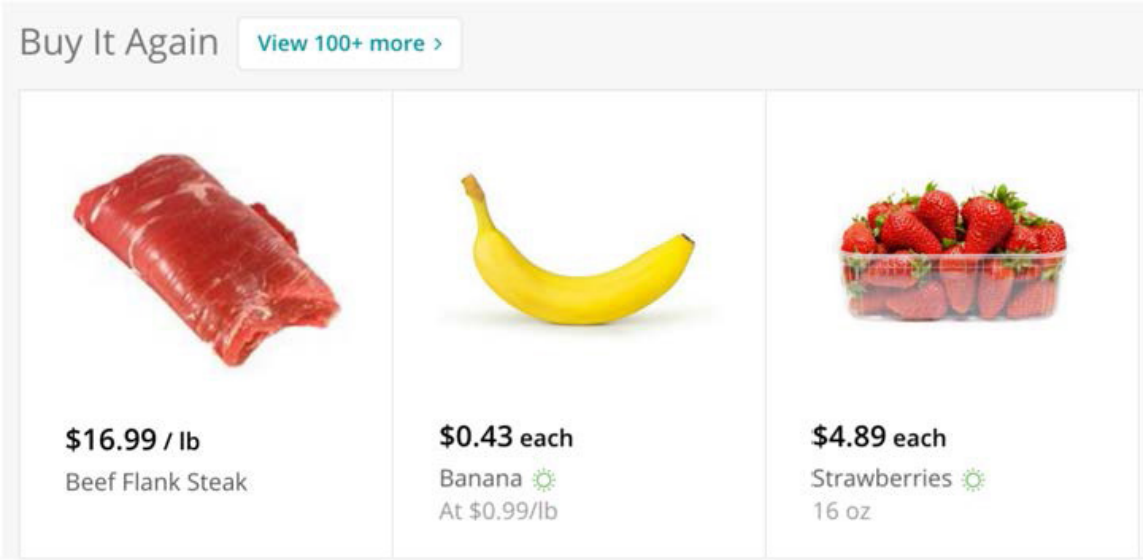


Use Case 3: Market basket analysis

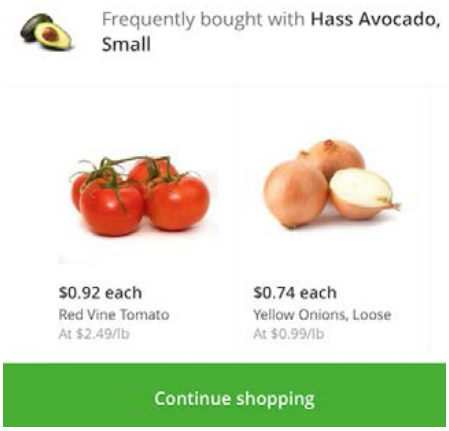
When providing recommendations to shoppers on what to purchase, you are often looking for items that are frequently purchased together (e.g., peanut butter and jelly). A key technique for uncovering associations between different items is known as market basket analysis. In your recommendation engine toolbox, the association rules generated by market basket analysis (e.g., if one purchases peanut butter, they are also likely to purchase jelly) are important and useful.

With the rapid growth of e-commerce data, models like market basket analysis are increasingly executed on larger sizes of data. Having the algorithms and infrastructure necessary to generate your association rules on a distributed platform is critical. In this use case, we demonstrate how you can quickly run your market basket analysis using the Apache Spark MLlib FP-growth algorithm on Azure Databricks.

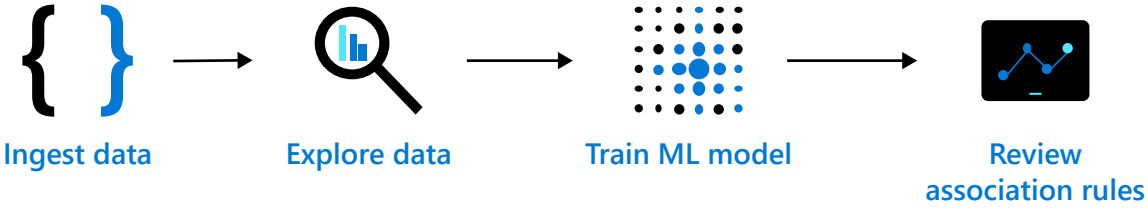
To showcase this, we will use the publicly available Instacart Online Grocery Shopping Dataset 2017. In the process, we will explore the dataset and use market basket analysis to recommend that shoppers buy a product again or suggest new items.



Use case 3: Market basket analysis



Access the notebook for the market basket analysis. The flow of this section is as follows:



- Ingest your data: Bringing in the data from your source systems. This usually involves ETL processes, but we are bypassing this step in this demo for brevity).
- Explore your data using Spark SQL: Now that you have cleansed data, explore it so you can get some business insight.
- Train your ML model using FP-growth: Execute FP-growth to execute your frequent pattern mining algorithm
- Review the association rules generated by the ML model for your recommendations model.

Ingest data

The dataset we will be working with is 3 Million Instacart Orders, Open Sourced dataset:

The "Instacart Online Grocery Shopping Dataset 2017," Accessed from https://www.instacart.com/datasets/grocery-shopping-2017 on 01/17/2018. This anonymized dataset contains a sample of over 3 million grocery orders from more than 200,000 Instacart users. For each user, we provide between 4 and 100 of their orders, along with the sequence of products purchased in each order. We also provide the week and hour of day the order was placed and a relative measure of time between orders.

Use case 3: Market basket analysis

You will need to download the file, extract the files from the gzipped TAR archive, and upload them into Azure Databricks DBFS using the Import Data utilities. You should see the following files in DBFS once the files are uploaded:

- `Orders`: 3.4M rows, 206K users
- `Products`: 50K rows
- `Aisles`: 134 rows
- `Departments`: 21 rows
- `order_products__SET`: 30M+ rows where `SET` is defined as:
  - `prior`: 3.2M previous orders
  - `train`: 131K orders for your training dataset

Refer to the Instacart Online Grocery Shopping Dataset 2017 Data Descriptions for more information, including the schema.

Create DataFrames

With your data uloaded to DBFS, you can quickly and easily create your DataFrames using `spark.read.csv`:

```
# Import Data
aisles = spark.read.csv("/mnt/bhavin/mba/instacart/csv/aisles.csv",
header=True, inferSchema=True)
departments = spark.read.csv("/mnt/bhavin/mba/instacart/csv/departments.
csv", header=True, inferSchema=True)
order_products_prior = spark.read.csv("/mnt/bhavin/mba/instacart/csv/
order_products__prior.csv", header=True, inferSchema=True)
order_products_train = spark.read.csv("/mnt/bhavin/mba/instacart/csv/
order_products__train.csv", header=True, inferSchema=True)
orders = spark.read.csv("/mnt/bhavin/mba/instacart/csv/orders.csv",
header=True, inferSchema=True)
products = spark.read.csv("/mnt/bhavin/mba/instacart/csv/products.csv",
header=True, inferSchema=True)

# Create Temporary Tables
aisles.createOrReplaceTempView("aisles")
departments.createOrReplaceTempView("departments")
order_products_prior.createOrReplaceTempView("order_products_prior")
order_products_train.createOrReplaceTempView("order_products_train")
orders.createOrReplaceTempView("orders")
products.createOrReplaceTempView("products")
```

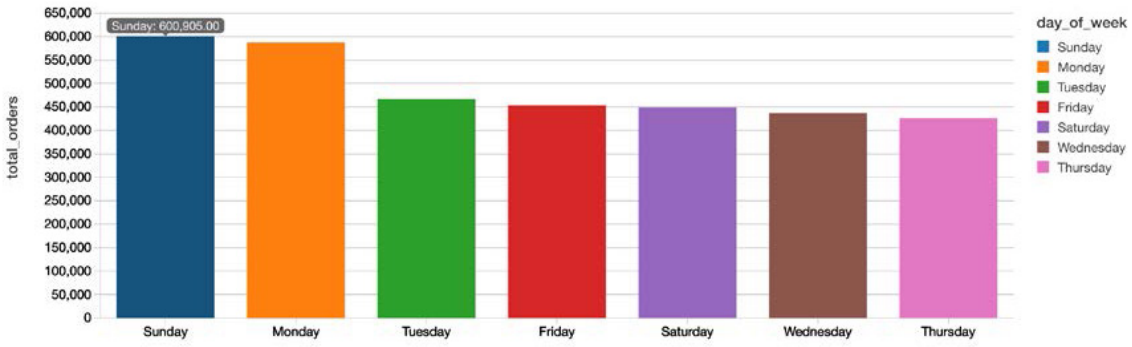
Exploratory data analysis

Now that you have created DataFrames, you can use Spark SQL for exploratory data analysis. The following queries showcase some of the quick insights you can gain from the Instacart dataset.

Orders by day of week

The following query allows you to quickly see that Sunday is the most popular day for the total number of orders while Thursday has the least number of orders.

```
%sql
select
  count(order_id) as total_orders,
  (case
       when order_dow = '0' then 'Sunday'
       when order_dow = '1' then 'Monday'
       when order_dow = '2' then 'Tuesday'
       when order_dow = '3' then 'Wednesday'
       when order_dow = '4' then 'Thursday'
       when order_dow = '5' then 'Friday'
       when order_dow = '6' then 'Saturday'
  end) as day_of_week
  from orders
group by order_dow
order by total_orders desc
```

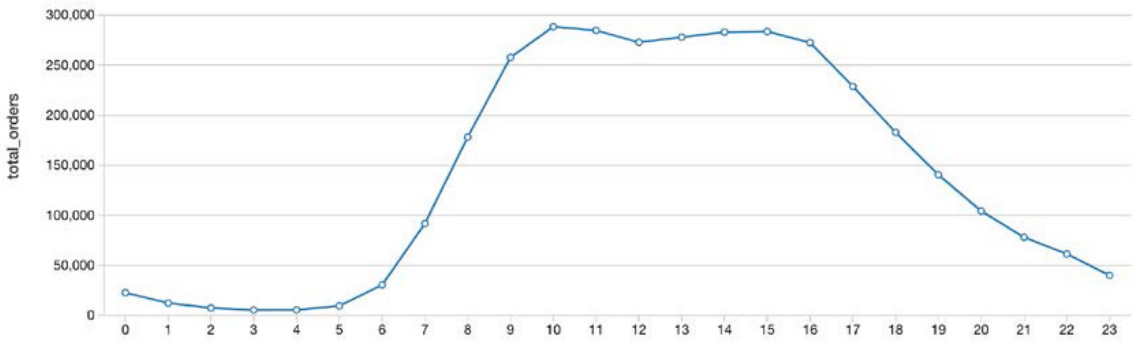


Use case 3: Market basket analysis

Orders by hour

When you break down the hours, typically people are ordering their groceries from Instacart during business working hours with highest number orders at 10:00 a.m.

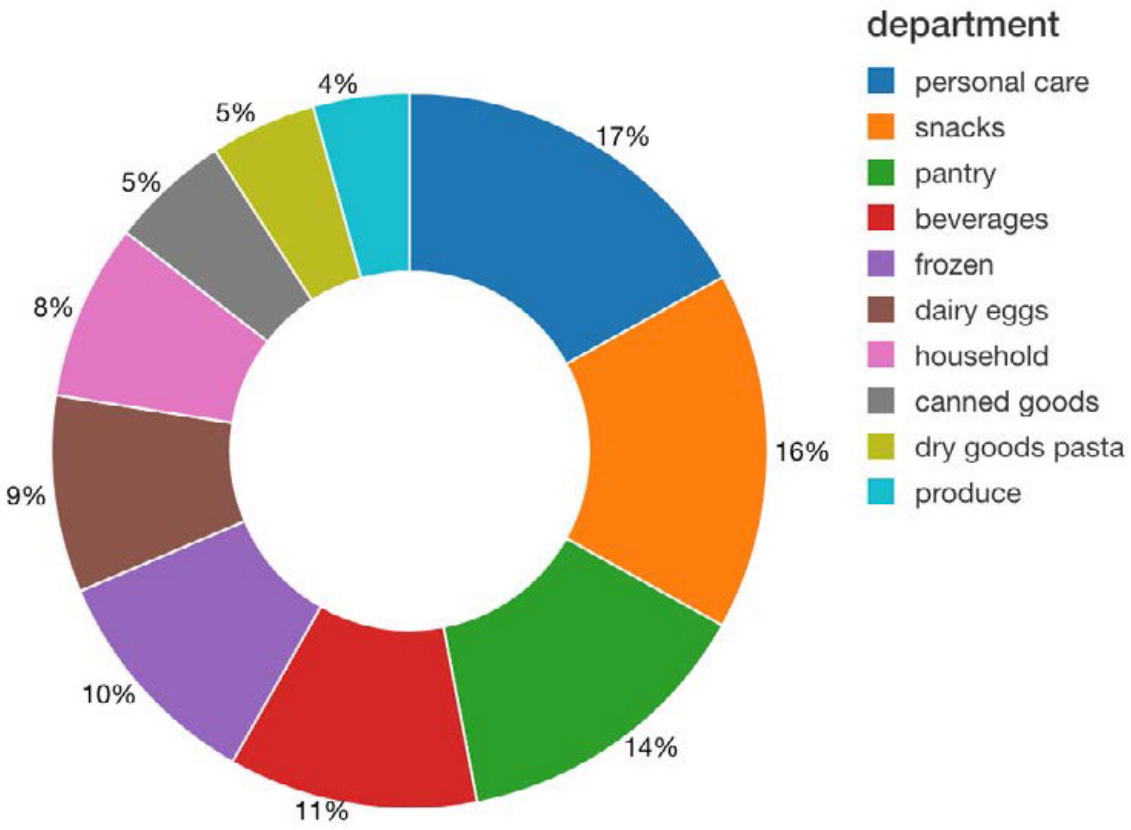
```
%sql
select
  count(order_id) as total_orders,
  order_hour_of_day as hour
  from orders
group by order_hour_of_day
order by order_hour_of_day
```



Understand shelf space by department

As you dive deeper into your market basket analysis, you can uncover insights on the number of products by department to understand how much shelf space is being used.

```
%sql
select d.department, count(distinct p.product_id) as products
  from products p
    inner join departments d
      on d.department_id = p.department_id
group by d.department
order by products desc
limit 10
```



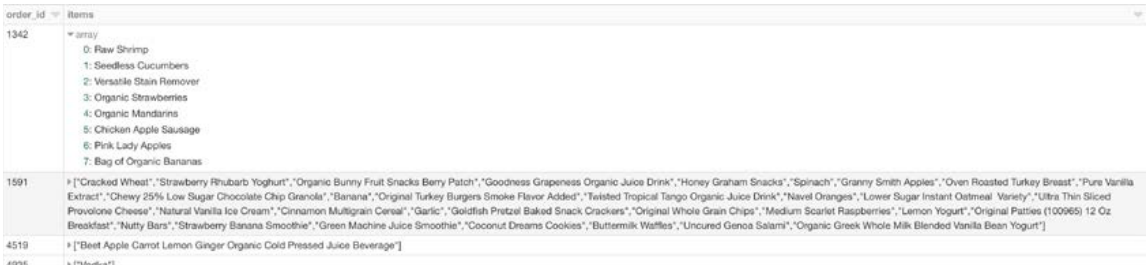
Use case 3: Market basket analysis

Organize shopping basket

To prepare your data for downstream processing, organize your data by shopping basket. That is, each row of your `DataFrame` represents an `order_id` with each `items` column containing an array of items.

```
# Organize the data by shopping basket
from pyspark.sql.functions import collect_set, col, count
rawData = spark.sql("select p.product_name, o.order_id from products p
inner join order_products_train o where o.product_id = p.product_id")
baskets = rawData.groupBy('order_id').agg(collect_set('product_name').
alias('items'))
baskets.createOrReplaceTempView('baskets')
```

Just like the preceding graphs, you can visualize the nested items using the `display` command in the Azure Databricks notebook.



Train ML model

To understand how the frequency of items purchased are associated with one another (e.g., how many times are peanut butter and jelly purchased together), we use association rule mining for market basket analysis. Spark MLlib implements two algorithms related to frequency pattern mining (FPM): `FP-growth` and `PrefixSpan`. The distinction is that `FP-growth` does not use order information in the itemsets, if any, while `PrefixSpan` is designed for sequential pattern mining where the itemsets are ordered.

You can just use `FP-growth` in this use case because the order information is not important.

Note: We are using the `Scala API` so we can configure `setMinConfidence`.

Use case 3: Market basket analysis

```
%scala
import org.apache.spark.ml.fpm.FPGrowth

// Extract out the items
val baskets_ds = spark.sql("select items from baskets").
as[Array[String]].toDF("items")

// Use FPGrowth
val fpgrowth = new FPGrowth().setItemsCol("items").
setMinSupport(0.001).setMinConfidence(0)
val model = fpgrowth.fit(baskets_ds)

// Calculate frequent itemsets
val mostPopularItemInABasket = model.freqItemsets
mostPopularItemInABasket.
createOrReplaceTempView("mostPopularItemInABasket")
```

With Databricks notebooks, you can use the `%scala` to execute Scala code within a new cell in the same Python notebook.

With the `mostPopularItemInABasket DataFrame` created, we can use Spark SQL to query for the most popular items in a basket where there are more than 2 items as follows.

```
%sql
select items, freq from mostPopularItemInABasket where
size(items) > 2 order by freq desc limit 20
```

items	freq
array 0: Organic Hass Avocado 1: Organic Strawberries 2: Bag of Organic Bananas	506
["Organic Raspberries","Organic Strawberries","Bag of Organic Bananas"]	452
["Organic Baby Spinach","Organic Strawberries","Bag of Organic Bananas"]	413
["Organic Raspberries","Organic Hass Avocado","Bag of Organic Bananas"]	355
["Organic Hass Avocado","Organic Baby Spinach","Bag of Organic Bananas"]	353
["Organic Avocado","Organic Baby Spinach","Banana"]	338
["Organic Avocado","Large Lemon","Banana"]	311

As you can see, the most frequent purchases of more than two items are organic avocados, organic strawberries, and organic bananas. Interestingly, the top five items frequently purchased together are various permutations of organic avocados, organic strawberries, organic bananas, organic raspberries, and organic baby spinach. From the perspective of recommendations, the `freqItemsets` can be the basis for the

buy-it-again recommendation because if a shopper has purchased the items previously, it makes sense to recommend that they purchase it again.

Review association rules

In addition to `freqItemSets`, the `FP-growth` model also generates `associationRules`. For example, if a shopper purchases peanut butter, what is the probability (or confidence) that they will also purchase jelly? For more information, a good reference is A Gentle Introduction on Market Basket Analysis — Association Rules.

```
%scala
// Display generated association rules.
val ifThen = model.associationRules
ifThen.createOrReplaceTempView("ifThen")
```

A good way to think about association rules is that the model determines that if you purchased something (i.e., the antecedent), then you will purchase this other thing (i.e., the consequent) with the following confidence.

```
%sql
select antecedent as `antecedent (if)`, consequent as
`consequent (then)`, confidence from ifThen order by confidence
desc limit 20
```

antecedent (if)	consequent (then)	confidence
array 0: Organic Raspberries 1: Organic Hass Avocado 2: Organic Strawberries	["Bag of Organic Bananas"]	0.6038461538461538
["Organic Kiwi","Organic Hass Avocado"]	["Bag of Organic Bananas"]	0.5665236051502146
["Organic Hass Avocado","Organic Baby Spinach","Organic Strawberries"]	["Bag of Organic Bananas"]	0.5470085470085471
["Organic Whole String Cheese","Organic Hass Avocado"]	["Bag of Organic Bananas"]	0.5422885572139303
["Yellow Onions","Strawberries"]	["Banana"]	0.5306122448979592
["Organic Navel Orange","Organic Raspberries"]	["Bag of Organic Bananas"]	0.5284974093264249
["Organic Navel Orange","Organic Hass Avocado"]	["Bag of Organic Bananas"]	0.5283018867924528
["Organic Cucumber","Organic Hass Avocado","Organic Strawberries"]	["Bag of Organic Bananas"]	0.5274725274725275
["Organic Fuji Apple","Seedless Red Grapes"]	["Banana"]	0.5138121546961326
["Organic Raspberries","Organic Hass Avocado"]	["Bag of Organic Bananas"]	0.5115273775216138
["Organic Unsweetened Almond Milk","Organic Hass Avocado"]	["Bag of Organic Bananas"]	0.509009009009009
["Organic Fuji Apple","Strawberries"]	["Banana"]	0.5073170731707317

As you can see in the graph, there is relatively strong confidence that if a shopper has organic raspberries, organic avocados, and organic strawberries in their basket, it probably makes sense to recommend organic bananas as well. Interestingly, the top 10 (based on descending confidence) association rules—i.e., purchase recommendations—are associated with organic bananas or bananas.

Discussion

In summary, we demonstrated how to explore shopping cart data and execute market basket analysis to identify items frequently purchased together and generate association rules. By using Azure Databricks, in the same notebook, you can:

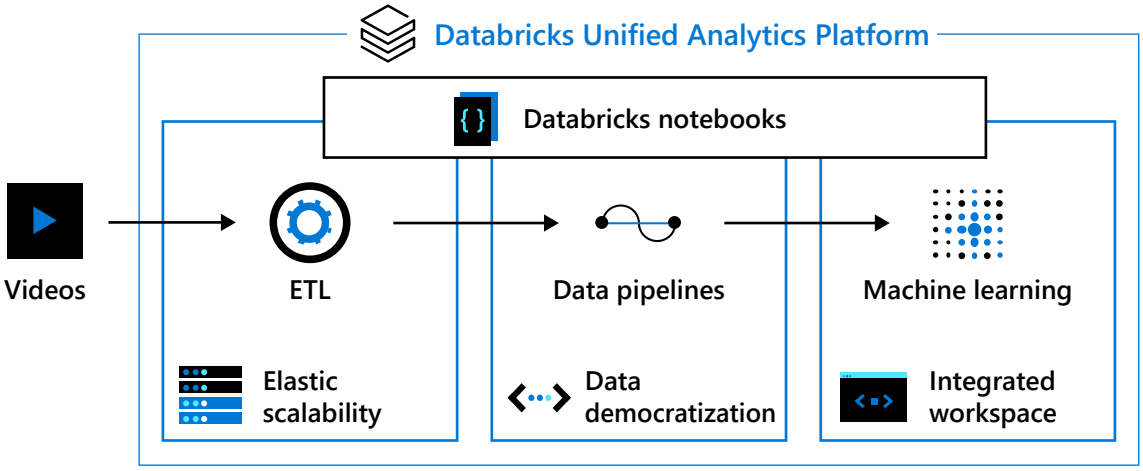
- Visualize data.
- Execute Python, Scala, and SQL
- Run our FP-growth algorithm on an auto-scaling distributed Spark cluster

And all of it is managed by Azure Databricks.

Putting these components together simplifies the data flow and management of your infrastructure for you and your data practitioners.

Use Case 4: Suspicious behavior identification in videos

Operationalizing and automating the process of video identification and categorization is increasing important to numerous industries and activities. Applications range from identifying the correct cat video to visually categorizing objects. With millions of users around the world generating and consuming billions of minutes of video daily, you will need the infrastructure to handle this massive scale.



With rapidly scalable infrastructure, multiple machine learning and deep learning packages, and high-performance mathematical computing, video processing can be complex and confusing. Data scientists and engineers tasked with this endeavor will continuously encounter a number of architectural questions:

1. How can we scale video processing and how scalable will the infrastructure be when built?
2. With a heavy data science component, how can I integrate, maintain, and optimize the various machine learning and deep learning packages in addition to my Apache Spark infrastructure?
3. How will the data engineers, data analysts, data scientists, and business stakeholders work together?

Use case 4: Suspicious behavior identification in videos

A solution to this problem is Azure Databricks, which includes notebooks, collaboration, and workspace features that allow different personas in your organization to come together and collaborate in one place. Azure Databricks includes the Azure Databricks Runtime for Machine Learning which is preconfigured and optimized with Machine Learning frameworks, including but not limited to XGBoost, scikit-learn, TensorFlow, Keras, and Horovod. Databricks provides optimized auto-scale clusters for reduced costs as well as GPU support in Azure.

This use case demonstrates how to combine distributed computing with Apache Spark and deep learning pipelines (Keras, TensorFlow, and Spark Deep Learning pipelines) with the Azure Databricks Runtime for Machine Learning to classify and identify suspicious videos.

Classifying suspicious videos

In our scenario, we have a set of videos from the EC Funded CAVIAR project/IST 2001 37540 datasets. We are using the Clips from INRIA (1st Set) with six basic scenarios acted out by the CAVIAR team members including:

- Walking
- Browsing
- Resting, slumping, or fainting
- Leaving bags behind
- People and groups meeting, walking together and splitting up
- Two people fighting

In this guide and the associated Identifying Suspicious Behavior in Video notebooks, you will pre-process, extract image features, and apply machine learning to these videos.

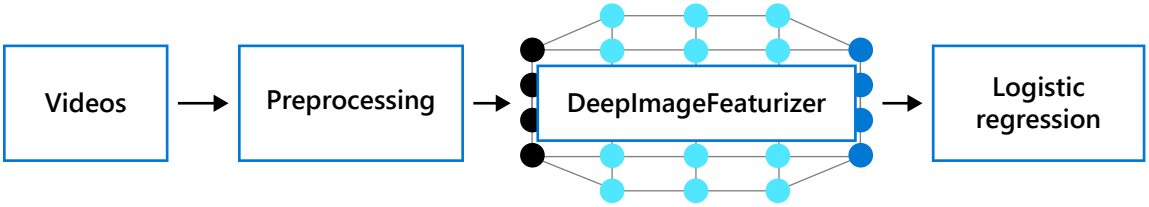
Use case 4: Suspicious behavior identification in videos

Source: Reenactment of a fight scene by CAVIAR members — EC Funded CAVIAR project/IST 2001 37540 http://groups.inf.ed.ac.uk/vision/CAVIAR/CAVIARDATA1/

For example, suspicious images (such as the example here) extracted from our test dataset (such as this video) will be identified by applying a machine learning model trained against a different set of images extracted from the training video dataset.

High-level data flow

This graphic describes the high-level data flow for processing the source videos to the training and testing of a logistic regression model.



The high-level data flow is:

- Videos: Use the EC Funded CAVIAR project/IST 2001 37540 Clips from INRIA (1st videos) as the set of training and test datasets (i.e., training and test set of videos).
- Preprocessing: Extract images from those videos to create a set of training and test set of images.
- DeepImageFeaturizer: Using Spark Deep Learning Pipeline's DeepImageFeaturizer, create a training and test set of image features.
- Logistic regression: Train and fit a logistic regression model to classify suspicious vs. not suspicious image features (and ultimately video segments).

The libraries needed for this installation are:

- h5py
- TensorFlow
- Keras
- Spark Deep Learning Pipelines
- TensorFrames
- OpenCV

Use case 4: Suspicious behavior identification in videos

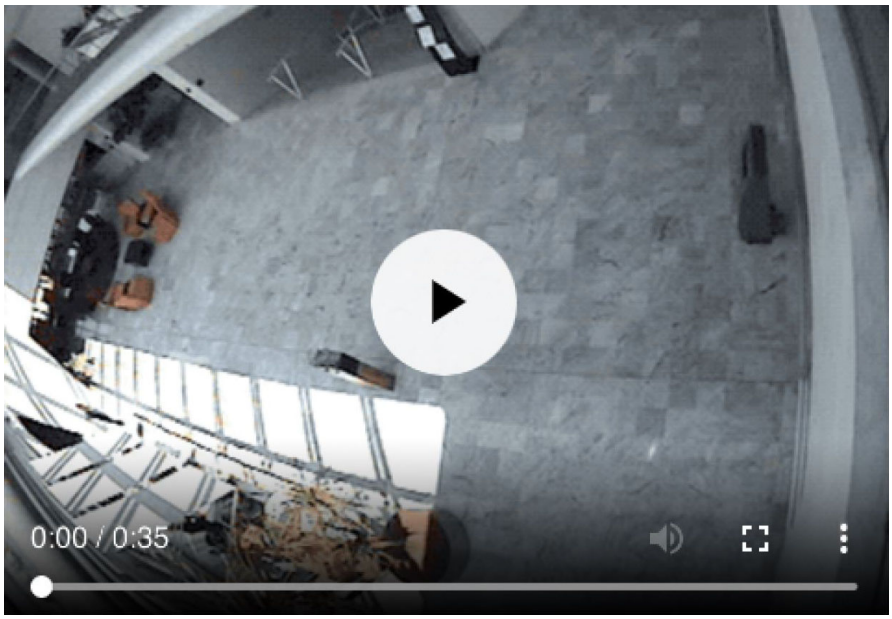
With Azure Databricks Runtime for Machine Learning, all but the OpenCV are already installed and configured to run your Deep Learning pipelines with Keras, TensorFlow, and Spark Deep Learning pipelines. With Azure Databricks, you also have the benefits of:

- Clusters that autoscale
- Being able to choose multiple cluster types
- The Azure Databricks workspace environment including collaboration and multi-language support
- Azure Databricks for analytics

Source videos

This graphic describes the high-level data flow for processing the source videos to the training and testing of a logistic regression model.

To help jump-start your video processing, we have copied the CAVIAR Clips from INRIA (1st Set) videos [EC Funded CAVIAR project/IST 2001 37540] to `/databricks-datasets.`



- Training Videos (`srcVideoPath`): `/databricks-datasets/cctvVideos/train/`
- Test Videos (`srcTestVideoPath`): `/databricks-datasets/cctvVideos/test/`
- Labeled Data (`labeledDataPath`): `/databricks-datasets/cctvVideos/labels/cctvFrames_train_ labels.csv`

Pre-processing

You will ultimately execute your machine learning models (logistic regression) against the features of individual images from the videos. The first (pre-processing) step will be to extract individual images from the video. One approach (included in the Azure Databricks notebook) is to use OpenCV to extract the images per second as noted in the following code snippet.

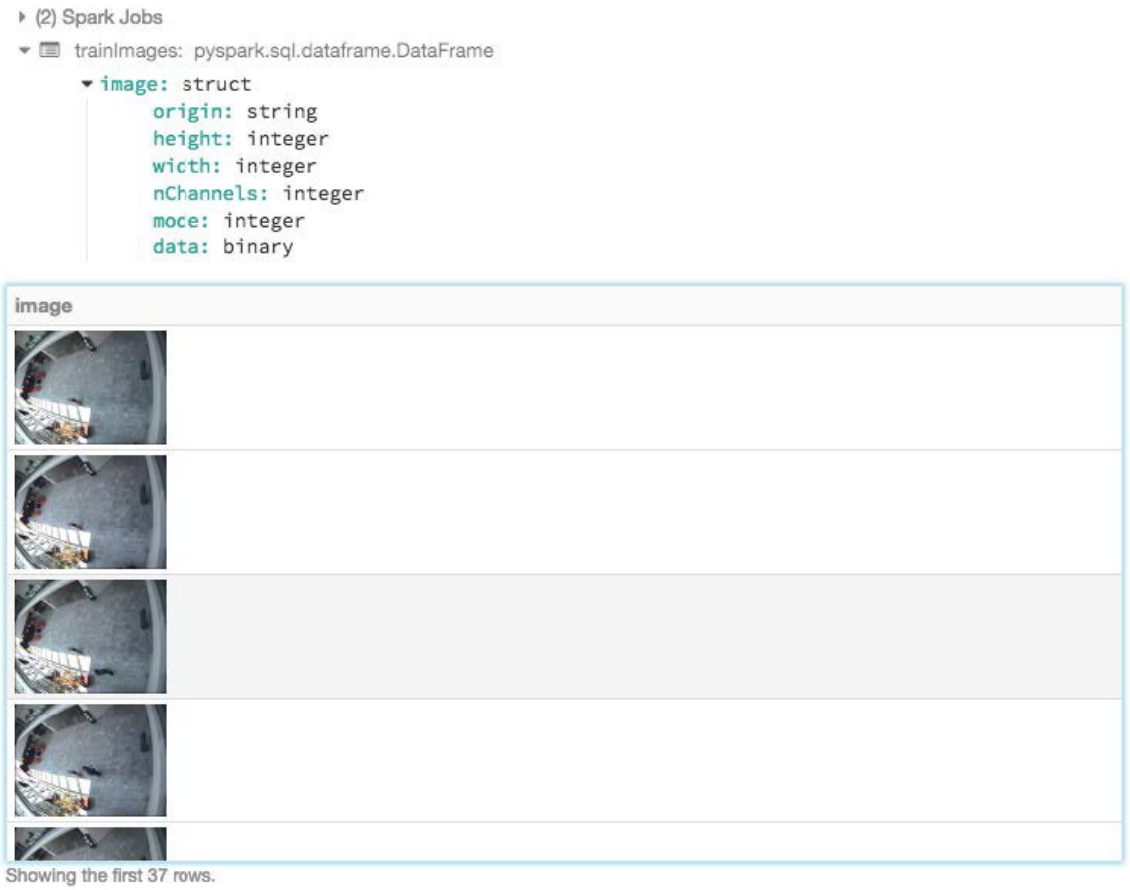
```
## Extract one video frame per second and save frame as JPG
def extractImages(pathIn):
  count = 0
  srcVideos = "/dbfs" + src + "(.*).mpg"
  p = re.compile(srcVideos)
  vidName = str(p.search(pathIn).group(1))
  vidcap = cv2.VideoCapture(pathIn)
  success,image = vidcap.read()
  success = True
  while success:
   vidcap.set(cv2.CAP_PROP_POS_MSEC,(count*1000))
   success,image = vidcap.read()
   print ('Read a new frame: ', success)
   cv2.imwrite("/dbfs" + tgt + vidName + "frame%04d.jpg" %
   count, image)    # save frame as JPEG file
   count = count + 1
   print ('Wrote a new frame')
```

In this case, we're extracting the videos from our DBFS location and using OpenCV's VideoCapture method to create image frames (taken every 1000ms) and saving those images to DBFS. The full code example can be found in the Identify Suspicious Behavior in Video Databricks notebooks.

Use case 4: Suspicious behavior identification in videos

Once you have extracted the images, you can read and view the extracted images using the following code snippet with the output similar to the following screenshot:

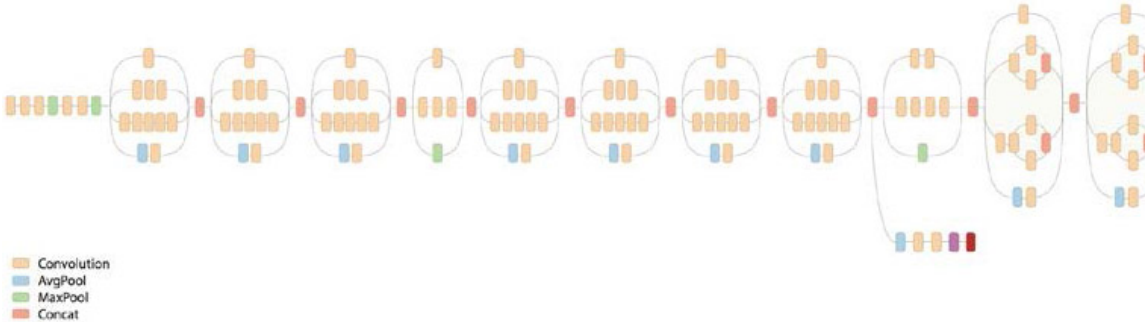
```
from pyspark.ml.image import ImageSchema

trainImages = ImageSchema.readImages(targetImgPath)
display(trainImages)
```



Note: We will perform this task on both the training and test set of videos.

DeepImageFeaturizer

As noted in A Gentle Introduction to Transfer Learning for Deep Learning, transfer learning is a technique where a model trained on one task (e.g., identifying images of cars) is re-purposed on another related task (e.g. identifying images of trucks). In our scenario, we are using Spark Deep Learning Pipelines to perform transfer learning on our images.



Source: Inception in TensorFlow

As noted in the following code snippet, we are using the Inception V3 model (Inception in TensorFlow) in the DeepImageFeaturizer to automatically extract the last layer of a pre-trained neural network to transform these images to numeric features.

```
# Build featurizer using DeepImageFeaturizer and InceptionV3
featurizer = DeepImageFeaturizer( \
       inputCol="image", \
       outputCol="features", \
       modelName="InceptionV3" \
)

# Transform images to pull out
# - image (origin, height, width, nChannels, mode, data)
# - and features (udt)
features = featurizer.transform(images)

# Push feature information into Parquet file format
features.select( \
  "Image.origin", "features" \
).coalesce(2).write.mode("overwrite").parquet(filePath)
```

Both the training and test set of images (sourced from their respective videos) will be processed by the DeepImageFeaturizer and ultimately saved as features stored in Parquet files.

Logistic regression

At this point, we now have a set of numeric features to fit and test our ML model against. Because we have a training and test dataset and we are trying to classify whether an image (and its associated video) are suspicious, we have a classic supervised classification problem where we can give logistic regression a try.

This use case is supervised because included with the source dataset is the `labeledDataPath`, which contains a labeled data CSV file (a mapping of image frame name and suspicious flag). The following code snippet reads in this hand-labeled data (`labels_df`) and joins it to the training features Parquet files (`featureDF`) to create our train dataset.

```
# Read in hand-labeled data
from pyspark.sql.functions import expr
labels = spark.read.csv( \
    labeledDataPath, header=True, inferSchema=True \
)
labels_df = labels.withColumn(
    "filePath", expr("concat('" + prefix + "', ImageName)") \
).drop('ImageName')

# Read in features data (saved in Parquet format)
featureDF = spark.read.parquet(imgFeaturesPath)

# Create train-ing dataset by joining labels and features
train = featureDF.join( \
    labels_df, featureDF.origin == labels_df.filePath \
).select("features", "label", featureDF.origin)
```

You can now fit a logistic regression model (lrModel) against this dataset as noted in the following code snippet.

```
from pyspark.ml.classification import LogisticRegression

# Fit LogisticRegression Model
lr = LogisticRegression( \
maxIter=20, regParam=0.05, elasticNetParam=0.3,
labelCol="label")
lrModel = lr.fit(train)
```

Use case 4: Suspicious behavior identification in videos

After training the model, you can now generate predictions on test dataset, i.e,. let your LR model predict which test videos are categorized as suspicious. As noted in the following code snippet, you load your test data (`featuresTestDF`) from Parquet and then generate the predictions on your test data (`result`) using the previously trained model (`lrModel`).

```
from pyspark.ml.classification import LogisticRegression,
LogisticRegressionModel

# Load Test Data
featuresTestDF = spark.read.parquet(imgFeaturesTestPath)

# Generate predictions on test data
result = lrModel.transform(featuresTestDF)
result.createOrReplaceTempView("result")
```

Now that you have the results from our test run, you can also extract out the second element (`prob2`) of the `probability` vector so we can sort by it.

```
from pyspark.sql.functions import udf
from pyspark.sql.types import FloatType

# Extract first and second elements of the StructType
firstelement=udf(lambda v:float(v[0]),FloatType())
secondelement=udf(lambda v:float(v[1]),FloatType())

# Second element is what we need for probability
predictions = result.withColumn("prob2",
secondelement('probability'))
predictions.createOrReplaceTempView("predictions")
```

In our example, the first row of the `predictions` DataFrame classifies the image as non-suspicious with `prediction = 0`. As we're using binary logistic regression, the probability `StructType` of (`firstelement, secondelement`) means (`probability of prediction = 0, probability of prediction = 1`). Our focus is to review suspicious images hence why order by the second element (`prob2`).

Use case 4: Suspicious behavior identification in videos

We can execute the following Spark SQL query to review any suspicious images (`where prediction = 1`) ordered by `prob2`.

```
%sql
select origin, probability, prob2, prediction from predictions
where prediction = 1 order by prob2 desc
```

origin	probability	prob2	prediction
dbfs:/mnt/tardis/videos/cctvFrames/test/Fight_OneManDownframe0024.jpg	▸ ["1","2",[],[0.019603140170946682,0.9803968598290532]]	0.98039687	1
dbfs:/mnt/tardis/videos/cctvFrames/test/Fight_OneManDownframe0014.jpg	▸ ["1","2",[],[0.03504309818023664,0.9649569018197635]]	0.9649569	1
dbfs:/mnt/tardis/videos/cctvFrames/test/Fight_OneManDownframe0017.jpg	▸ ["1","2",[],[0.04389332024139191,0.9561066797586081]]	0.95610666	1
dbfs:/mnt/tardis/videos/cctvFrames/test/Fight_OneManDownframe0016.jpg	▸ ["1","2",[],[0.07955018324337278,0.9204498167566273]]	0.9204498	1
dbfs:/mnt/tardis/videos/cctvFrames/test/Fight_OneManDownframe0019.jpg	▸ ["1","2",[],[0.12001899045090385,0.8799810095490961]]	0.879981	1
dbfs:/mnt/tardis/videos/cctvFrames/test/LeftBoxframe0027.jpg	▸ ["1","2",[],[0.12756099937933807,0.872439000620662]]	0.872439	1
dbfs:/mnt/tardis/videos/cctvFrames/test/LeftBoxframe0033.jpg	▸ ["1","2",[],[0.17766534327189748,0.8223346567281026]]	0.82233465	1
dbfs:/mnt/tardis/videos/cctvFrames/test/LeftBag_AtChairframe0022.jpg	▸ ["1","2",[],[0.19767648629846,0.8023235137015401]]	0.8023235	1
dbfs:/mnt/tardis/videos/cctvFrames/test/LeftBag_AtChairframe0023.jpg	▸ ["1","2",[],[0.21488110614298567,0.7851188938570143]]	0.7851189	1
dbfs:/mnt/tardis/videos/cctvFrames/test/LeftBoxframe0034.jpg	▸ ["1","2",[],[0.2186721830719919,0.781327816928008]]	0.78132784	1

Based on the above results, we can now view the top three frames that are classified as suspicious.

```
displayImg("dbfs:/mnt/tardis/videos/cctvFrames/test/Fight_
OneManDownframe0024.jpg")
```

Use case 4: Suspicious behavior identification in videos

```
displayImg("dbfs:/mnt/tardis/videos/cctvFrames/test/Fight_
OneManDownframe0014.jpg")
```

```
displayImg("dbfs:/mnt/tardis/videos/cctvFrames/test/Fight_
OneManDownframe0017.jpg")
```

Use case 4: Suspicious behavior identification in videos

Based on the results, you can quickly identify the video as noted below.

```
displayDbfsVid("databricks-datasets/cctvVideos/mp4/test/Fight_OneManDown.
mp4")
```

Source: Reenactment of a fight scene by CAVIAR members – EC Funded CAVIAR project/IST 2001 37540 http://groups.inf.ed.ac.uk/vision/CAVIAR/CAVIARDATA1/

Summary

In closing, we demonstrated:

- How to classify and identify suspicious video using the Azure Databricks
- How the Azure Databricks workspace allows for collaboration and visualization of ML models, videos, and extracted videos
- How to leverage Azure Databricks Runtime for Machine Learning, which comes preconfigured with Keras, TensorFlow, TensorFrames, and other machine learning and deep learning libraries, to simplify maintenance of these various libraries
- How optimized autoscaling of clusters with GPU support scale up and scale out high performance numerical computing

Putting these components together simplifies the data flow and management of video classification (and other machine learning and deep learning problems) for you and your data practitioners.

Customer case study: renewables.AI

Solar power producers need to predict when they will generate electricity and coordinate distribution with multiple energy markets. renewables.AI uses Azure Databricks and other Azure services to streamline product development, drive solar industry revenue, and promote a stable energy future for everyone.

Power, capacity, and friction-free collaboration

renewables.AI uses Microsoft Azure cloud services and Apache Spark to support a cloud-based data service, called Advanced Data Analytics for Management of Assets (ADAMA), that energy generators and utilities can use to predict solar energy production and infer trading volumes and prices.

To connect the Apache Spark architecture in ADAMA to Azure services, renewables.AI uses Azure Databricks. Company data scientists, developers, and engineers use interactive Azure Databricks notebooks to get started quickly, share code, and work together simultaneously.

"Instead of one data scientist writing AI code and being the only person who understands it, everybody uses Azure Databricks to share code and develop together," says Andy Cross, Codirector at renewables.AI in London.

Increased revenue and savings

According to renewables.AI, approximately 85% of the world's current solar power is being driven by subsidies. In three to four years, market-driven solar will add at least 200,000 megawatts which is likely to generate more than €9 billion of annual revenues to the existing global portfolio. Says Cross. "Renewables.AI will give the sector the computing firepower it needs to achieve this, and our initial milestones will be to help our clients increase revenues by €100 million and savings of double that amount."

Customer case study: LINX Cargo Care Group

A national supply chain and logistics provider headquartered in Sydney, Australia, LINX Cargo Care Group adopted Microsoft Azure and Azure Databricks to benefit from a scalable, flexible, and agile cloud and analytics platform, which it uses to quickly develop IT solutions that propel its business forward.

In search of an agile platform

LINX provides transport services and end-to-end supply chain and logistics solutions for a broad range of industries, including agriculture, aluminum, steel, automotive, forestry, food and beverage, mining, marine, oil and gas, major retail, and resources. When an investment firm purchased 70 percent of LINX Cargo Care Group, the company decided to replace their legacy technology with a platform that could prepare them for the future. They wanted an agile solution that they could use to come up with new ideas and execute them quickly,

They decided on Microsoft Azure and Azure Databricks, a fully managed environment that could provide a platform for the present—and the future.

A new data warehouse

A data warehouse to provide visibility into its financial, payroll, and operational data was critical. LINX deployed Azure Data Factory to integrate structured, unstructured, and streaming data from disparate systems and Azure Data Lake Storage to create a hyper-scale, Hadoop-compatible repository. Azure Databricks runs on top of the data lake to perform analytics at scale while Microsoft Power BI is used for reporting.

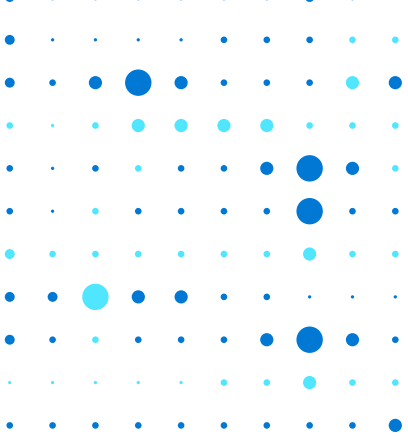
Customer case study: LINX Cargo Care Group

Azure Databricks delivers the requisite scalability, flexibility, and agility. Thomas Gianniodis, General Manager IT, explains: "Some of the datasets we get from the business aren't standard. We couldn't have loaded that data without a platform as flexible as Azure Databricks."

As part of the data warehouse project, LINX developed an application that staff can use to see whether customers have paid their invoices. "Using Azure Databricks, we created a dashboard in three days that shows what was paid, owed, the gap, and payment history—and reveals patterns or trends," says Gianniodis. "That would have taken three weeks to build manually."

Outstanding fund recovery

"By using Azure Databricks analytics to understand payment patterns, our customer service reps will be able to make targeted, proactive collection phone calls rather than reactive ones," says Gianniodis. "We expect to recover 90 percent of outstanding funds by their due dates and thereby return capital to our investors."



Learn more

Azure Databricks enables data science teams to collaborate with data engineering and lines of business to build data and machine learning products. Users achieve faster time-to-value with Azure Databricks by creating analytic workflows that go from ETL interactive exploration. A fully managed, scalable, and secure cloud infrastructure reduces operational complexity and total cost of ownership.

To learn how you can build scalable, real-time data and machine learning pipelines, visit https://azure.microsoft.com/en-us/services/databricks/

Resources

Documentation

- Azure Databricks best practices
- Introduction to Delta Lake
- Introduction to MLFlow

Webinars

- Get started with Apache Spark
- Azure Databricks best practices
- Machine Learning life cycle management with Azure Databricks and Azure Machine Learning

More info

- Azure Databricks pricing page
- Azure Databricks unit pre-purchase plan

Contact us

- Talk to a sales expert

Invent with purpose

Microsoft Azure