

Four Real-Life Machine Learning Use Cases

A Databricks guide, including code samples and notebooks.

Introduction

Data is the new fuel. The potential for Machine Learning and Deep Learning practitioners to make a breakthrough and drive positive outcomes is unprecedented. But how to take advantage of the myriad of data and ML tools now available at our fingertips, and scale model training on big data, for real-life scenarios?

Databricks Unified Analytics Platform is a cloud-service designed to provide you with 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.

“ Working in Databricks is like getting a seat in first class. It’s just the way flying (or more data science-ing) should be. ”

— Mary Clair Thompson, Data Scientist, Overstock.com

In this guide, we will walk you through four practical end-to-end Machine Learning use cases on 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, including collecting and exploring the advertising logs with Spark SQL, 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.
- A suspicious behavior identification in videos example, including pre-processing step to create image frames, transfer learning for featurization, and applying logistic regression to identify suspicious images in a video.

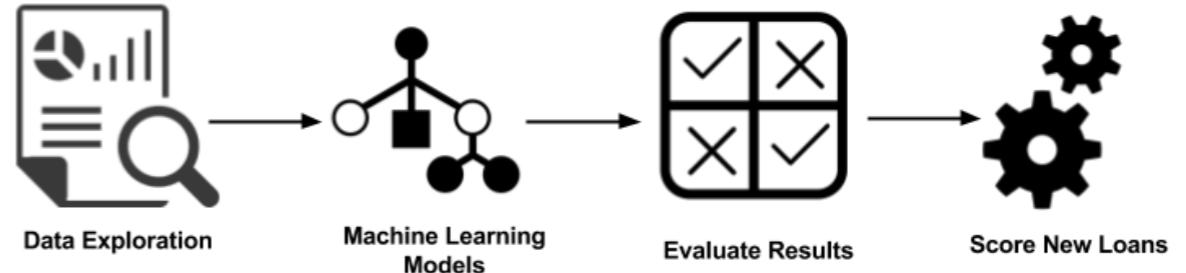
Use Case #1: Loan Risk Analysis with XGBoost

For companies that make money off of interest on loans held by their customer, it's always about increasing the bottom line. Being able to assess the risk of loan applications can save a lender the cost of holding too many risky assets. It is the data scientist's job to run analysis on your customer data and make business rules that will directly impact loan approval.

The data scientists that 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 are dependent on the data engineers to obtain up-to-date tables.
- With data growing at an exponential rate, they are dependent on the 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.

This is where the Databricks [Unified Analytics Platform](#) can help bridge those gaps between different parts of that workflow chain and reduce friction between the data scientists, data engineers, and software engineers.



In addition to reducing operational friction, Databricks is a central location to run the latest Machine Learning models. Users can leverage the native Spark MLLib package or download any open source Python or R ML package. 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.

In this blog, we will explore how to:

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

IMPORT DATA

For our experiment, we will be using the public [Lending Club Loan Data](#). It includes all funded loans from 2012 to 2017. Each loan includes applicant information provided by the applicant as well as the current loan status (Current, Late, Fully Paid, etc.) and latest payment information. [For more information, refer to the Lending Club Data schema](#).

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

Once you have downloaded the data locally, you can create a database and table within the Databricks workspace to load this dataset. For more information, refer to [Databricks Documentation > User Guide > Databases and Tables > Create a Table](#).

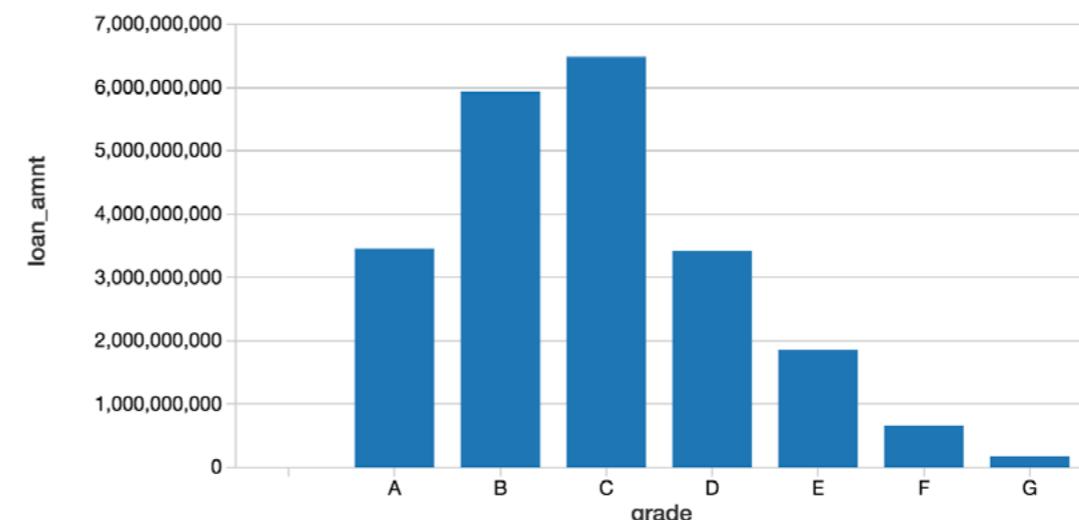
In this case, we have created the Databricks Database amy and table loanstats_2012_2017. The following code snippet allows you to access this table within a Databricks notebook via PySpark.

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

EXPLORE YOUR DATA

With the Databricks `display` command, you can make use of the Databricks native visualizations.

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



In this case, we can view the asset allocations by reviewing the loan grade and the loan amount.

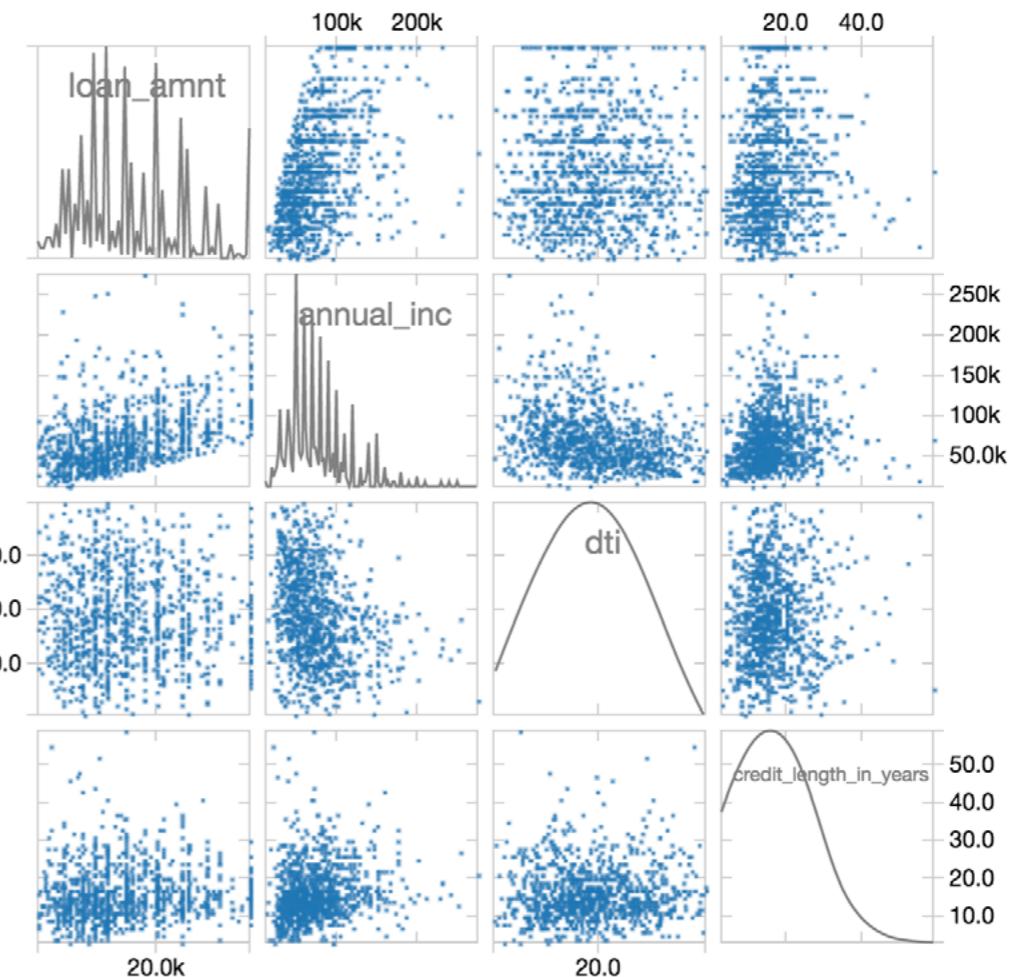
MUNGING YOUR DATA WITH THE PYSPARK DATAFRAME API

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. Below is a code snippet to 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")
```

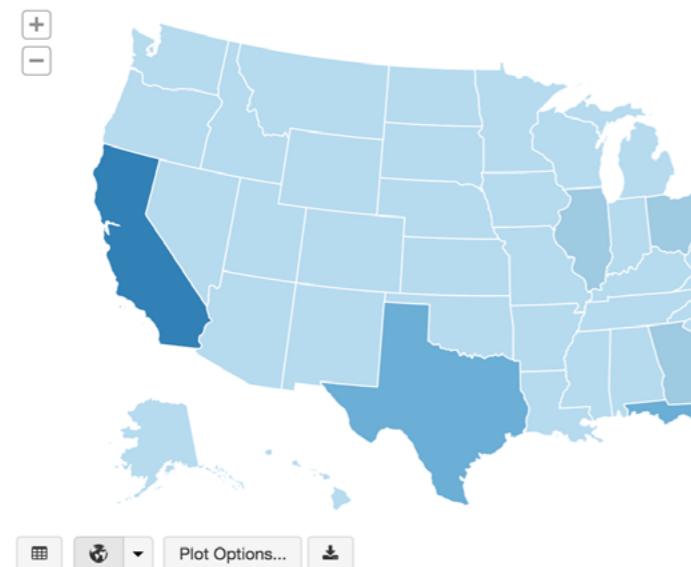
After this ETL process is completed, you can use the `display` command again to review the cleansed data in a scatterplot.

```
# 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 OUR ML MODEL USING XGBOOST

While we can quickly visualize our asset data, we would like to see if we can create a machine learning model that will allow us to predict if a loan is good or bad based on the available parameters. As noted in the following code snippet, we will predict `bad_loan` (defined as `label`) by building our ML pipeline as follows:

- Executes an `imputer` to fill in missing values within the `numerics` attributes (output is `numerics_out`)
- Using `indexers` to handle the categorical values and then converting them to vectors using `OneHotEncoder` via `oneHotEncoders` (output is `categoricals_class`).
- The features for our ML pipeline are defined by combining the `categoricals_class` and `numerics_out`.
- Next, we will assemble the features together by executing the `VectorAssembler`.
- As noted previously, we will establish our `label` (i.e. what we are going to try to predict) as the `bad_loan` column.
- Prior to establishing which algorithm to apply, apply the standard scaler to build our pipeline array (`pipelineAry`).

While the previous code snippets are in Python, the following code examples are written in Scala to allow us to utilize 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 = new VectorAssembler()
  .setInputCols(featureCols)
  .setOutputCol("features")

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

// 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 we have established our pipeline, let's create our XGBoost pipeline and apply it to our 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.

Now that we have our model, we can test our model against the validation dataset with predictions containing the result.

REVIEWING MODEL EFFICACY

Now that we have built and trained our XGBoost model, let's determine its efficacy by using the `BinaryClassificationEvaluator`.

```
// 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-the-curve (AUC) is: ~0.6520.

TUNE MODEL USING MLLIB CROSS VALIDATION

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

// 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 our cross-validation and choose the best set of parameters. We can then re-run our predictions and re-calculate the accuracy.

```
// 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)
```

Our accuracy increased slightly with a value ~0.6734.

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. The definition of our matrix is as follows:

- 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 :

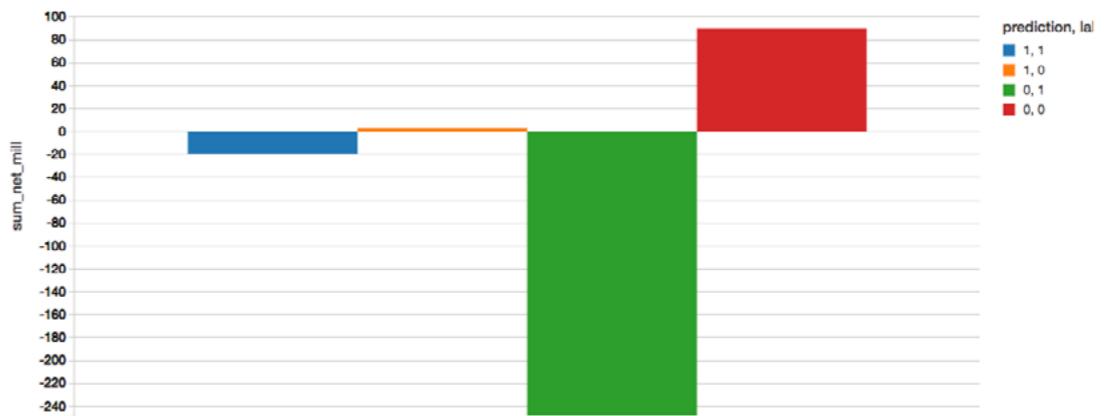
Incorrectly labeled good loans. sum_net = loss still incurred.

- 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, we can calculate this as

$$\text{value} = -(\text{loss avoided} - \text{profit forfeited})$$

Our current XGBoost model with AUC = ~0.6734, the values note the significant value gain from implementing our XGBoost model.

- value (XGBoost): 22.076

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 the [Databricks Unified Analytics Platform \(UAP\)](#) which includes the 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.

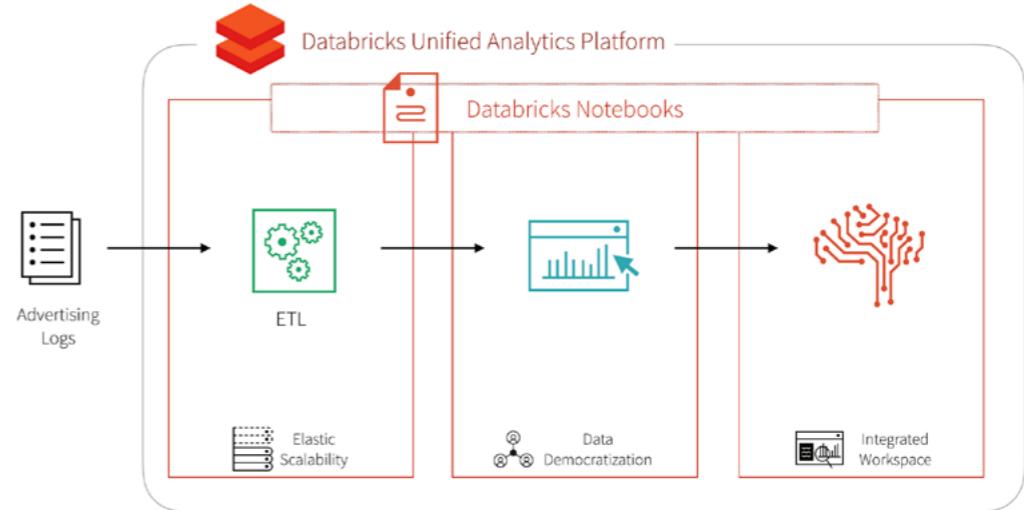
By removing the data engineering complexities commonly associated with such data pipelines, we could quickly import our data source into a Databricks table, explore your data using Databricks Visualizations, execute ETL code against your data, and build, train, and tune your ML pipeline using XGBoost logistic regression.

Start experimenting with this free Databricks [notebook](#).



Use Case #2: Advertising Analytics Click Prediction

Advertising teams want to analyze their immense stores and varieties of data requiring a scalable, extensible, and elastic platform. Advanced analytics, including but not limited to classification, clustering, recognition, prediction, and recommendations allow these organizations to gain deeper insights from their data and drive business outcomes. As data of various types grow in volume, Apache Spark provides an API and distributed compute engine to process data easily and in parallel, thereby decreasing time to value. The [Databricks Unified Analytics Platform](#) provides an optimized, managed cloud service around Spark, and allows for self-service provisioning of computing resources and a collaborative workspace.



Let's look at a concrete example with 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 or not there will be a click.

To build our advanced analytics workflow, let's focus on the three main steps:

- ETL
- Data Exploration, for example, using SQL
- Advanced Analytics / Machine Learning

BUILDING THE ETL PROCESS FOR THE ADVERTISING LOGS

First, we download the dataset to our blob storage, either [AWS S3](#) or [Microsoft Azure Blob storage](#). Once we have the data in blob storage, we can 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 our 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](#), we 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")
```

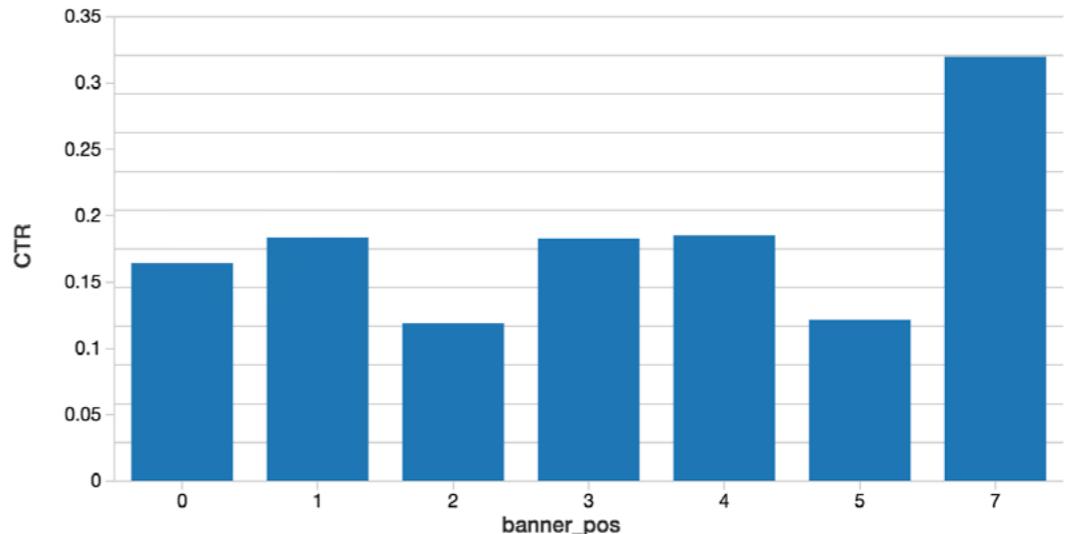
EXPLORE ADVERTISING LOGS WITH SPARK SQL

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

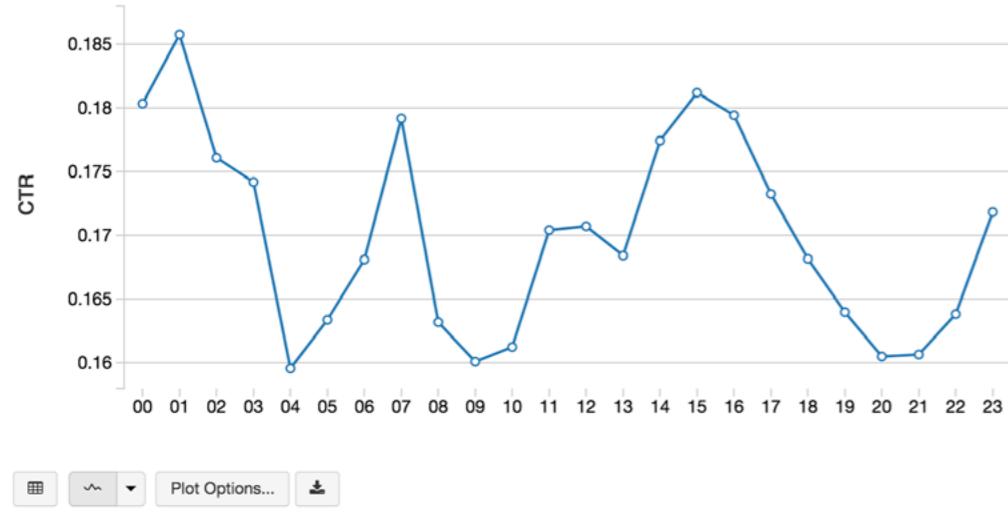
```
%python  
# Create Spark DataFrame reading the recently created Parquet files  
impression = spark.read \  
.parquet("/mnt/adtech/impression/parquet/train.csv")  
  
# Create temporary view  
impression.createOrReplaceTempView("impression")
```

We can now explore our data with the familiar and ubiquitous SQL language. 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
```



```
%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

Once we have familiarized ourselves with our data, we can proceed to the machine learning phase, where we convert our data into features for input to a machine learning algorithm and produce a trained model with which we can predict. 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

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]
```

In our use of GBTClassifier, you may have noticed that while we use string indexer but we are not applying One Hot Encoder (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 bunch of 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

Therefore, for because for tree based methods, it is preferable to not use OHE as it 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.

Thanks to Brooke Wenig and Joseph Bradley for contributing to this post!

With our workflow created, we can create our ML pipeline.

```
from pyspark.ml import Pipeline

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

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

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

Using `display(featurizedImpressions.select('features', 'label'))`, we can visualize our 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,[0,2,4,11],[1,20,7,2]]	1

Next, we will split our featurized dataset into training and test datasets via `.randomSplit()`.

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

Next, we will train, predict, and evaluate our model using the GBTClassifier. As a side note, a good primer on solving binary classification problems with Spark MLlib is Susan Li's [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)

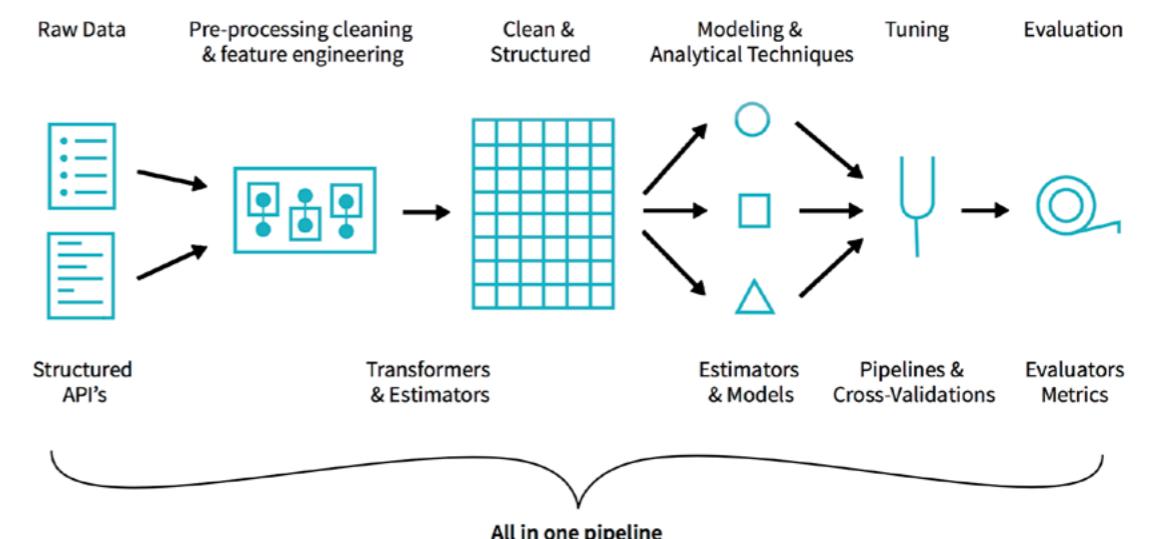
# Evaluate our GBTClassifier model using
#   BinaryClassificationEvaluator()
from pyspark.ml.evaluation import BinaryClassificationEvaluator
ev = BinaryClassificationEvaluator( \\
    rawPredictionCol="rawPrediction", metricName="areaUnderROC")
print ev.evaluate(predictions)

# Output
0.7112027059
```

With our predictions, we can evaluate the model according to some evaluation metric, for example, area under the ROC curve, and view features by importance. We can also see the AUC value which in this case is 0.7112027059.

SUMMARY

We demonstrated how you can simplify your advertising analytics – including click prediction – using the Databricks [Unified Analytics Platform](#) (UAP). With Databricks UAP, we were quickly able to execute our three components for click prediction: ETL, data exploration, and machine learning. We've illustrated how you can run our advanced analytics workflow of ETL, analysis, and machine learning pipelines all within a few Databricks notebook.



By removing the data engineering complexities commonly associated with such data pipelines with the Databricks Unified Analytics Platform, this allows different sets of users i.e. data engineers, data analysts, and data scientists to easily work together.

Start experimenting with this free Databricks [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 to uncover 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, then they are likely to purchase jelly) is an important and useful technique. With the rapid growth e-commerce data, it is necessary to execute models like market basket analysis on increasing larger sizes of data. That is, it will be important to have the algorithms and infrastructure necessary to generate your association rules on a distributed platform. In this blog post, we will discuss how you can quickly run your market basket analysis using Apache Spark MLlib FP-growth algorithm on Databricks.

To showcase this, we will use the publicly available [Instacart Online Grocery Shopping Dataset 2017](#). In the process, we will explore the dataset as well as perform our market basket analysis to recommend shoppers to buy it again or recommend to buy new items.

The screenshot shows a user interface for Instacart. At the top, there's a header "Buy It Again" with a link "View 100+ more >". Below this, there are three items recommended based on previous purchases:

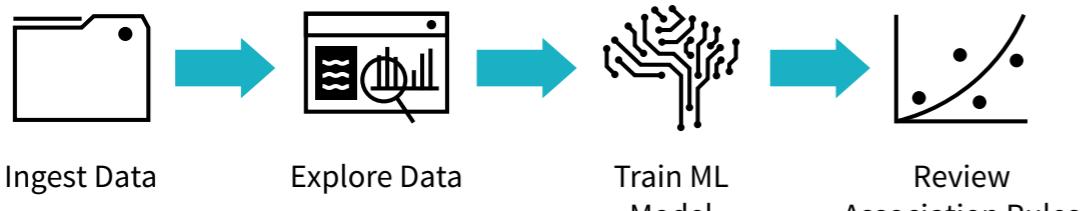
Item Image	Price	Description
	\$16.99 / lb	Beef Flank Steak
	\$0.43 each	Banana At \$0.99/lb
	\$4.89 each	Strawberries 16 oz

Below these recommendations, there's a section titled "Frequently bought with Hass Avocado, Small" which lists two items:

Item Image	Price	Description
	\$0.92 each	Red Vine Tomato At \$2.49/lb
	\$0.74 each	Yellow Onions, Loose At \$0.99/lb

At the bottom right of the interface is a green button labeled "Continue shopping".

The flow of this post, as well as the associated notebook, is as follows:



- **Ingest your data:** Bringing in the data from your source systems; often involving ETL processes (though we will bypass 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 recommendationsmodel to classify suspicious vs. not suspicious image features (and ultimately video segments).

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, 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.

You will need to download the file, extract the files from the gzipped TAR archive, and upload them into Databricks DBFS using the [Import Data](#) utilities. You should see the following files within 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

Now that you have uploaded your data 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")
```

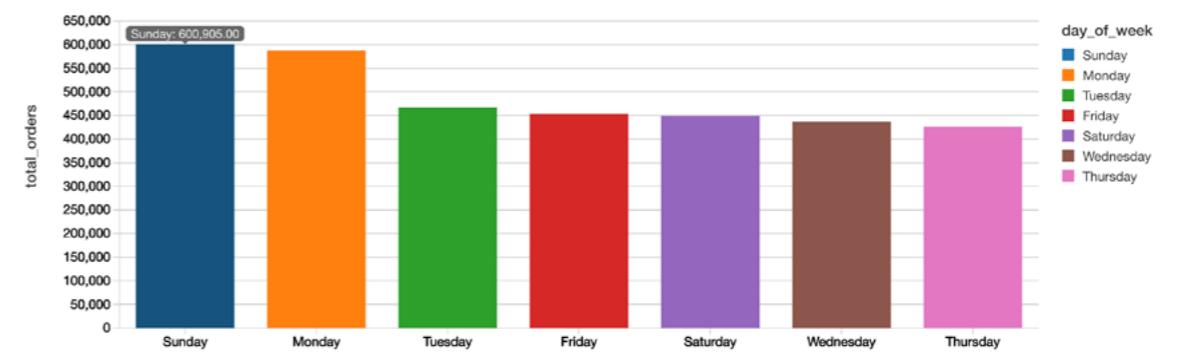
EXPLORATORY DATA ANALYSIS

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

ORDERS BY DAY OF WEEK

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

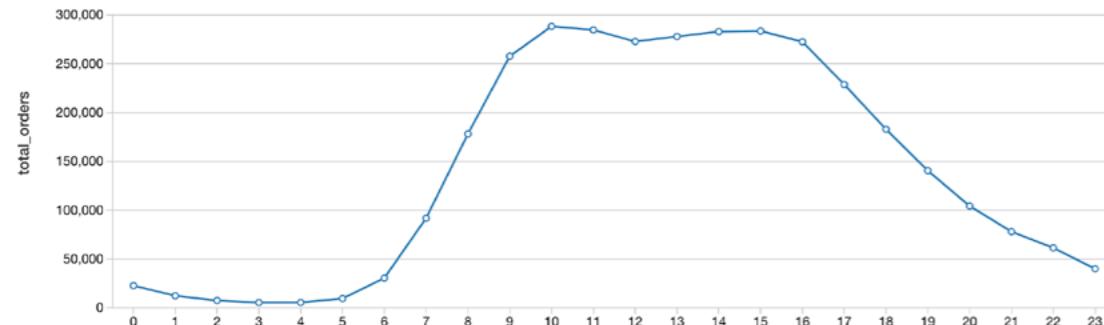
```
%sql
select
    count(order_id) as total_orders,
    (case
        when order_dow = '0' then 'Sunday'
        when order_dow = '1' then 'Monday'
        when order_dow = '2' then 'Tuesday'
        when order_dow = '3' then 'Wednesday'
        when order_dow = '4' then 'Thursday'
        when order_dow = '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
```



ORDERS BY HOUR

When breaking down the hours typically people are ordering their groceries from Instacart during business working hours with highest number orders at 10:00am.

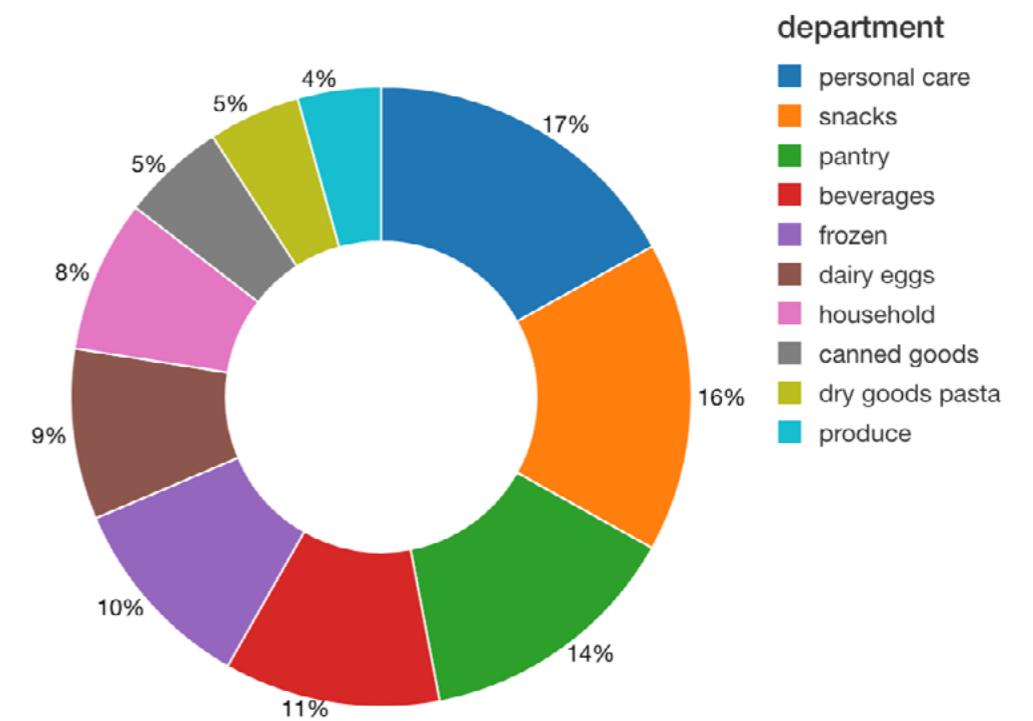
```
%sql  
select  
    count(order_id) as total_orders,  
    order_hour_of_day as hour  
    from orders  
group by order_hour_of_day  
order by order_hour_of_day
```



UNDERSTAND SHELF SPACE BY DEPARTMENT

As we dive deeper into our market basket analysis, we can gain insight 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
```



As can see from the preceding image, typically the number of unique items (i.e. products) involve personal care and snacks.

ORGANIZE SHOPPING BASKET

To prepare our data for downstream processing, we will organize our data by shopping basket. That is, each row of our 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, we can visualize the nested items using the `display` command in our Databricks notebooks.

order_id	items
1342	array: 0: Raw Shrimp 1: Seedless Cucumbers 2: Versatile Stain Remover 3: Organic Strawberries 4: Organic Mandarins 5: Chicken Apple Sausage 6: Pink Lady Apples 7: Bag of Organic Bananas
1591	["Cracked Wheat", "Strawberry Rhubarb Yoghurt", "Organic Bunny Fruit Snacks Berry Patch", "Goodness Grapeness Organic Juice Drink", "Honey Graham Snacks", "Spinach", "Granny Smith Apples", "Oven Roasted Turkey Breast", "Pure Vanilla Extract", "Chewy 25% Low Sugar Chocolate Chip Granola", "Banana", "Original Turkey Burgers Smoke Flavor Added", "Twisted Tropical Tango Organic Juice Drink", "Navel Oranges", "Lower Sugar Instant Oatmeal Variety", "Ultra Thin Sliced Provolone Cheese", "Natural Vanilla Ice Cream", "Cinnamon Multigrain Cereal", "Garlic", "Goldfish Pretzel Baked Snack Crackers", "Original Whole Grain Chips", "Medium Scarlet Raspberries", "Lemon Yogurt", "Original Patties (100g) 12 Oz Breakfast", "Nutty Bars", "Strawberry Banana Smoothie", "Green Machine Juice Smoothie", "Coconut Dreams Cookies", "Buttermilk Waffles", "Uncured Genoa Salami", "Organic Greek Whole Milk Blended Vanilla Bean Yogurt"]
4519	["Beet Apple Carrot Lemon Ginger Organic Cold Pressed Juice Beverage"]
4096	["Mandarin"]

TRAIN ML MODEL

To understand the frequency of items are associated with each other (e.g. how many times does peanut butter and jelly get purchased together), we will 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. We will use FP-growth as the order information is not important for this use case.

Note, we will be using the Scala API so we can configure `setMinConfidence`

```
%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 with the following query.

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

items	freq
array	506
0: Organic Hass Avocado	452
1: Organic Strawberries	413
2: Bag of Organic Bananas	355
["Organic Raspberries", "Organic Strawberries", "Bag of Organic Bananas"]	353
["Organic Baby Spinach", "Organic Strawberries", "Bag of Organic Bananas"]	338
["Organic Raspberries", "Organic Hass Avocado", "Bag of Organic Bananas"]	333
["Organic Hass Avocado", "Organic Baby Spinach", "Bag of Organic Bananas"]	311
["Organic Avocado", "Organic Baby Spinach", "Banana"]	...
["Organic Avocado", "Large Lemon", "Banana"]	...

As can be seen in the preceding table, the most frequent purchases of more than two items involve organic avocados, organic strawberries, and organic bananas. Interesting, the top five frequently purchased together items involve 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 in that 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 Susan Li's [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 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	["Bag of Organic Bananas"]	0.6038461538461538
0: Organic Raspberries		
1: Organic Hass Avocado		
2: Organic Strawberries		
["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 can be seen in the preceding graph, there is relatively strong confidence that if a shopper has organic raspberries, organic avocados, and organic strawberries in their basket, then it may make 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

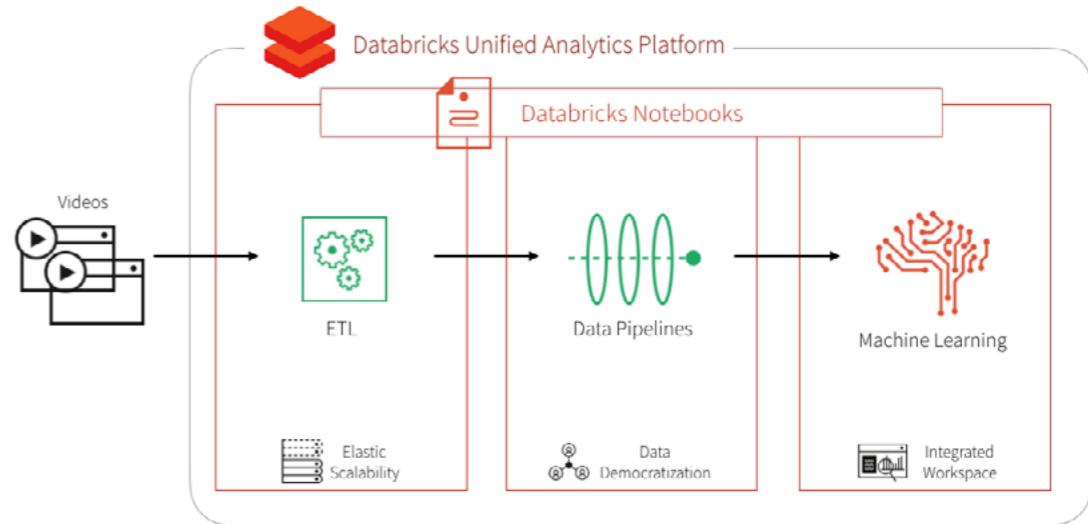
In summary, we demonstrated how to explore our shopping cart data and execute market basket analysis to identify items frequently purchased together as well as generating association rules. By using Databricks, in the same notebook we can visualize our data; execute Python, Scala, and SQL; and run our FP-growth algorithm on an auto-scaling distributed Spark cluster – all managed by Databricks. Putting these components together simplifies the data flow and management of your infrastructure for you and your data practitioners.

Start experimenting with this free Databricks [notebook](#).



Use Case #4: Suspicious Behavior Identification in Videos

With the exponential growth of cameras and visual recordings, it is becoming increasingly important to operationalize and automate the process of video identification and categorization. Applications ranging from identifying the correct cat video to visually categorizing objects are becoming more prevalent. 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 the complexities of rapidly scalable infrastructure, managing 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 to scale and how scalable will the infrastructure be when built?
2. With a heavy data sciences 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?

Our solution to this problem is the [Databricks Unified Analytics Platform](#) which includes the Databricks notebooks, collaboration, and workspace features that allows different personas of your organization to come together and collaborate in a single workspace. Databricks includes the [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 both [AWS](#) and [Azure](#).

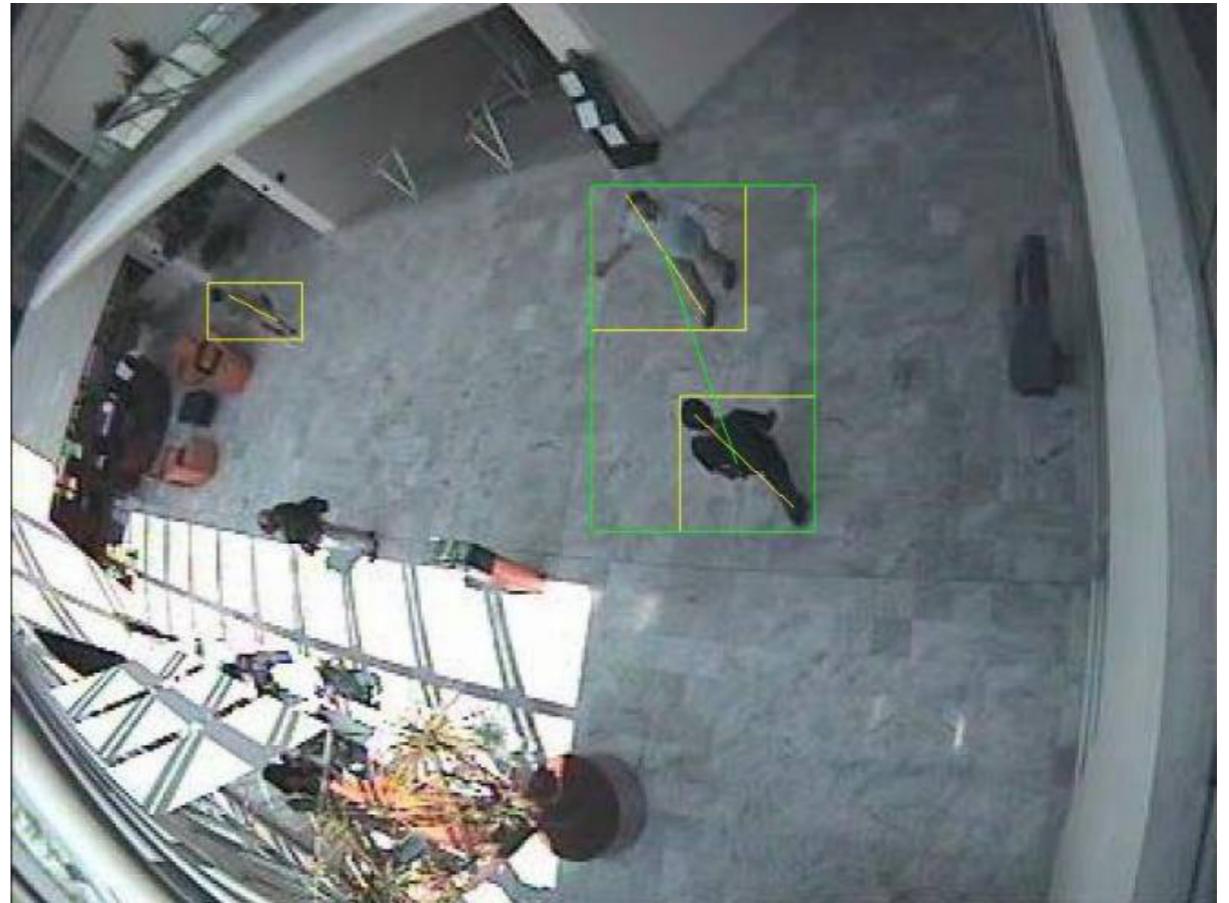
In this blog, we will show how you can combine distributed computing with Apache Spark and deep learning pipelines (Keras, TensorFlow, and Spark Deep Learning pipelines) with the [Databricks Runtime for Machine Learning](#) to classify and identify suspicious videos.

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/groups meeting, walking together and splitting up
- Two people fighting

In this blog post and the associated [Identifying Suspicious Behavior in Video Databricks notebooks](#), we will pre-process, extract image features, and apply our machine learning against these videos.



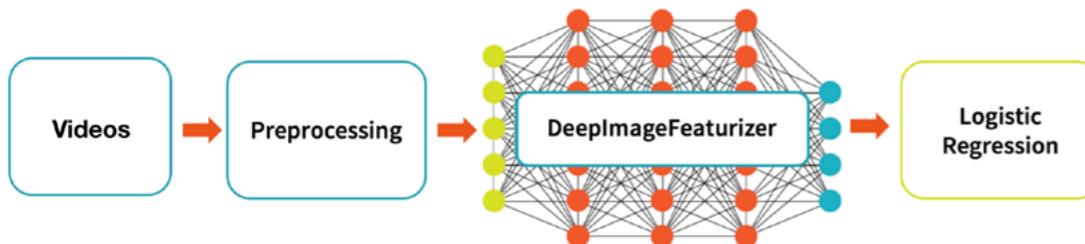
Source: Reenactment of a fight scene by CAVIAR members — EC Funded CAVIAR project/IST 2001 37540
<http://groups.inf.ed.ac.uk/vision/CAVIAR/CAVIARDATA1/>

For example, we will identify suspicious images (such as the one below) extracted from our test dataset (such as [above video](#)) by applying a machine learning model trained against a different set of images extracted from our training video dataset.



HIGH-LEVEL DATA FLOW

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



The high-level data flow we will be performing is:

- **Videos:** Utilize the [EC Funded CAVIAR project/IST 2001 37540](#) Clips from INRIA (1st videos) as our 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:** We will then train and fit a logistic regression model to classify suspicious vs. not suspicious image features (and ultimately video segments).

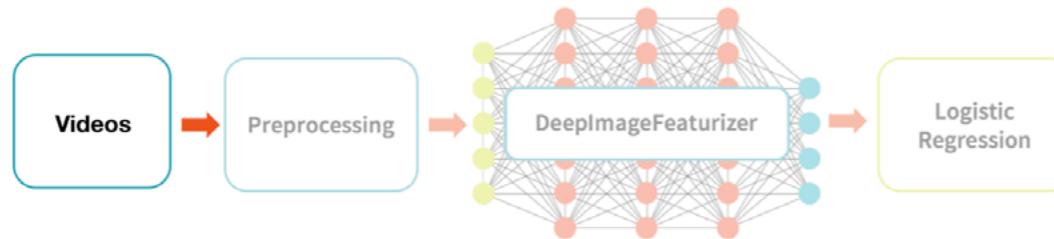
The libraries needed to perform this installation:

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

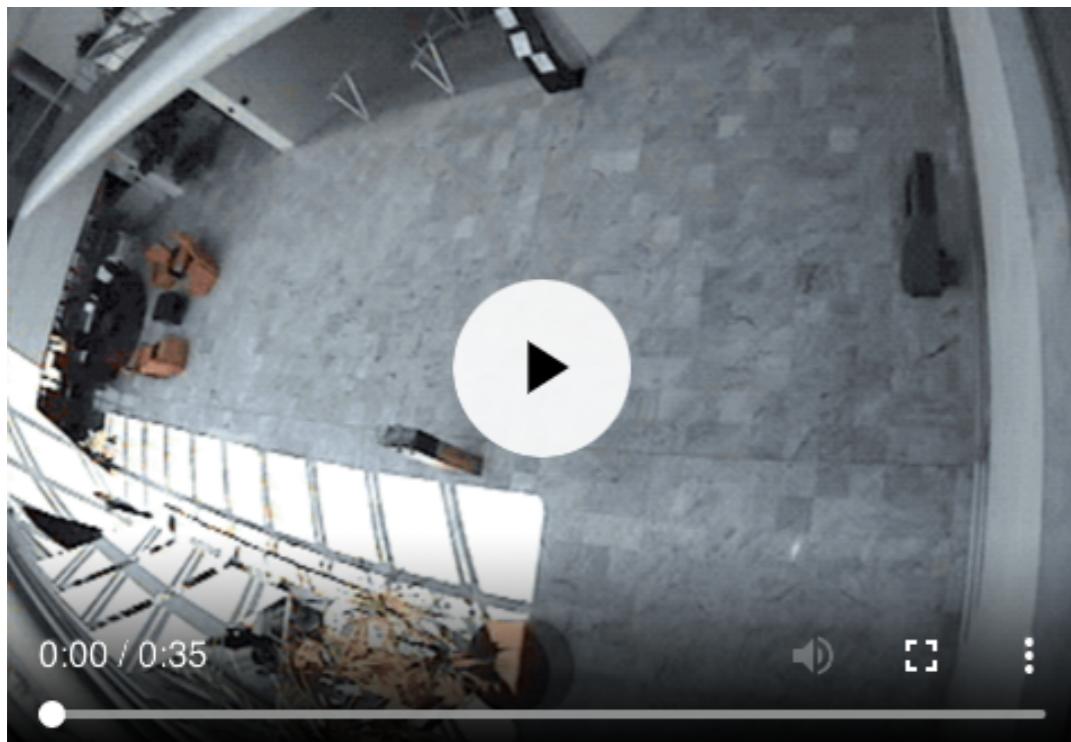
With Databricks Runtime for ML, all but the OpenCV is already pre-installed and configured to run your Deep Learning pipelines with Keras, TensorFlow, and Spark Deep Learning pipelines. With Databricks you also have the benefits of clusters that autoscale, being able to choose multiple cluster types, Databricks workspace environment including collaboration and multi-language support, and the Databricks Unified Analytics Platform to address all your analytics needs end-to-end.

SOURCE VIDEOS

The graphic below describes our high-level data flow for processing our 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/

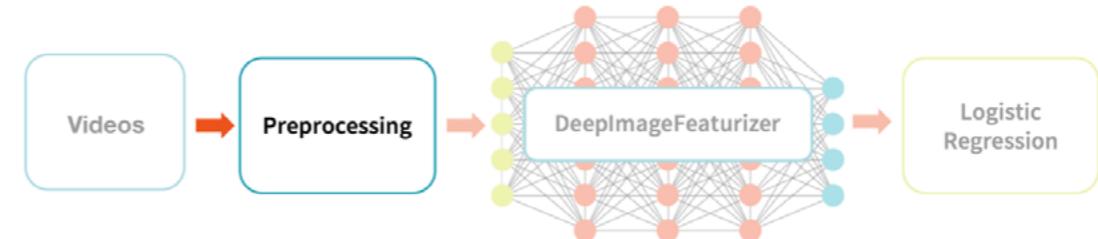
• Test Videos (srcTestVideoPath):

/databricks-datasets/cctvVideos/test/

• Labeled Data (labeledDataPath):

/databricks-datasets/cctvVideos/labels/cctvFrames_train_labels.csv

PREPROCESSING



We will ultimately execute our machine learning models (logistic regression) against the features of individual images from the videos. The first (preprocessing) step will be to extract individual images from the video. One approach (included in the 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.

Once you have extracted the images, you can read and view the extracted images using the following code snippet:

```

from pyspark.ml.image import ImageSchema

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

```

with the output similar to the following screenshot.

The screenshot shows a Databricks notebook interface. At the top, there is a schema browser for the `trainImages` DataFrame. The schema is defined as follows:

```

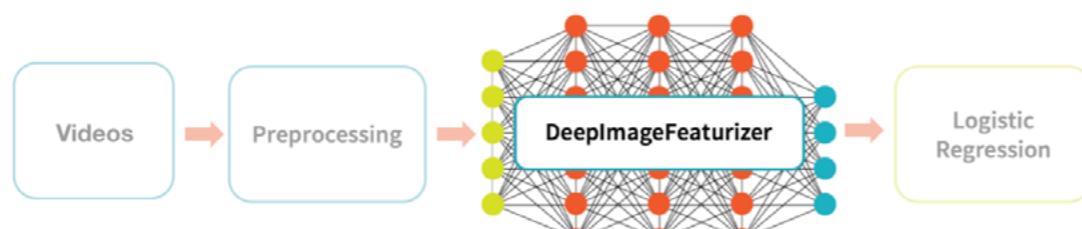
(2) Spark Jobs
  - trainImages: pyspark.sql.dataframe.DataFrame
    - image: struct
      - origin: string
      - height: integer
      - width: integer
      - nChannels: integer
      - mode: integer
      - data: binary

```

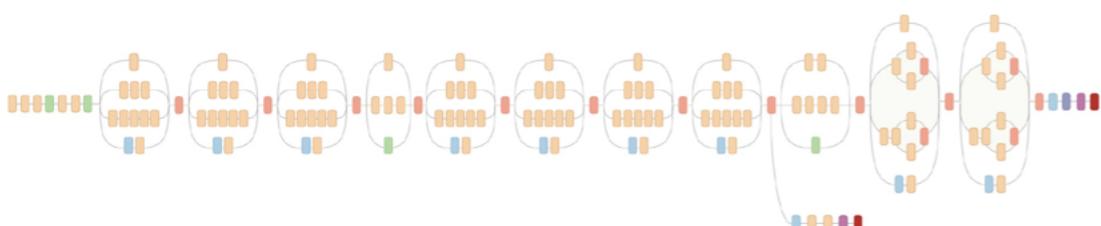
Below the schema, there is a preview section titled "image" which displays four small thumbnail images of a room interior. Below the thumbnails, it says "Showing the first 37 rows." At the bottom of the preview area, there are several icons for filtering and sorting, and a checked checkbox labeled "Show image preview".

Note, we will perform this task on both the training and test set of videos.

DEEPIIMAGEFEATURIZER



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 will be using Spark Deep Learning Pipelines to perform [transfer learning on our images](#).



Legend:

- Convolution
- AvgPool
- MaxPool
- Concat
- Dropout
- Fully connected
- Softmax

Source: [Inception in TensorFlow](#)

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

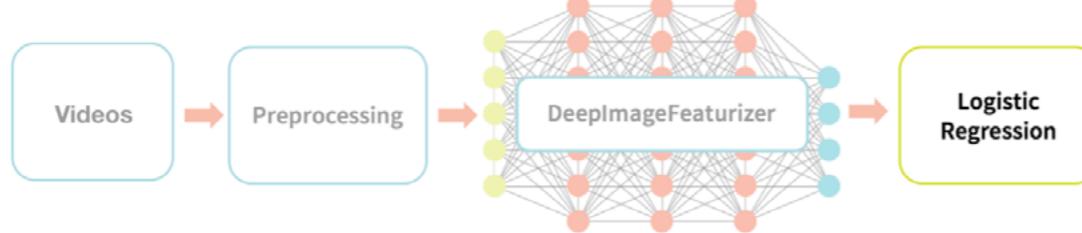
```
# Build featurizer using DeepImageFeaturizer and InceptionV3
featurizer = DeepImageFeaturizer( \
    inputCol="image", \
    outputCol="features", \
    modelName="InceptionV3" \
)

# 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



In the previous steps, we had gone through the process of converting our source training and test videos into images and then extracted and saved the features in Parquet format using OpenCV and Spark Deep Learning Pipelines DeepImageFeaturizer (with Inception V3). 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 this 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)
```

We 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)
```

After training our model, we can now generate predictions on our test dataset, i.e. let our LR model predict which test videos are categorized as suspicious. As noted in the following code snippet, we load our test data (`featuresTestDF`) from Parquet and then generate the predictions on our 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 we have the results from our test run, we 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`).

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"}, [0.01960314017094682, 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, 0.9204498167586273]]	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, 0.8223346567281025]]	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_  
OneManDownframe0014.jpg")
```



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



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



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 Databricks [Unified Analytics Platform](#): Databricks workspace to allow for collaboration and visualization of ML models, videos, and extracted images, 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, and optimized autoscaling of clusters with GPU support to scale up and scale out your 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.

Start experimenting with this free Databricks [notebook](#) and the [Databricks Runtime for Machine Learning](#) today.

Customer Case Study: HP



HP Inc. was created in 2015 when Hewlett-Packard was split into two separate companies, HP Enterprise, and HP Inc. HP Inc. creates technology that makes life better for everyone, everywhere. Through a portfolio of printers, PCs, mobile devices, solutions, and services, their stated mission is to engineer experiences that amaze.

THE CHALLENGES

- **Siloed data:** Data stored in multiple systems, in multiple places, that was hard to access and analyze
- **Existing tools difficult to manage:** Reliance on old data warehousing technologies and tooling resulting in heavy DevOps work, which typically fell on the data science team

THE SOLUTION

Databricks provides HP Inc. with a scalable, affordable, unified analytics platform that allows them to predict product issues and uncover new customer services:

- **Simplified management of Apache Spark™:** Turnkey unified platform removed infrastructure complexity and empowered the data scientists to focus on the data and not DevOps
- **Analysis of high volume data in flight:** Analyze data from more than 20 million devices with a seamless workflow
- **Improved collaboration and productivity:** A unified platform for business analysts, data scientists, and subject matter experts

[Watch the Webinar: From Data Prep to Deep Learning:
How HP Unifies Analytics with Databricks](#)

Customer Case Study: Rue Gilt Groupe

RUE GILT G R O U P E

Rue La La strives to be the most engaging off-price, online style destination, connecting world-class brands with the next-generation shopper.

THE CHALLENGES

- In order to give each of their 19M shoppers a unique shopping experience, they must ingest and analyze over 400GB of data per day.
- Their data pipeline consists of 20+ different queries to transform the data for the data science team which was a highly computationally expensive process.
- Scale was also a challenge as they lacked a distributed system to meet the performance goals they required.

THE SOLUTION

Databricks provides Rue La La with a fully managed, distributed platform that allows them to ETL their data much faster and leverage machine learning to power their recommendation engine.

- Estimated query speedups of 60%.
- Parallel model building and prediction resulted in a 25x time savings over competing platforms.
- All of these improvements and benefits have made a tangible impact to their business, contributing to a 10x increase in purchase engagement, repeat visitation rates of over 60%, and incremental revenue gains of over 5% in a beta release program.

[Read the full Case Study:](#)

[Building a Hybridized Collaborative Filtering Recommendation Engine](#)

“ Databricks has enabled our data science team to work more collaboratively, empowering them to easily iterate and modify the models in a distributed fashion much faster. **”**

— Benjamin Wilson, Lead Data Scientist at Rue La La

Learn More



Databricks, founded by the original creators of Apache Spark™, is on a mission to accelerate innovation for our customers by unifying data science, engineering and business teams. The Databricks Unified Analytics Platform powered by Apache Spark 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 Databricks by creating analytic workflows that go from ETL through interactive exploration. We also make it easier for users to focus on their data by providing a fully managed, scalable, and secure cloud infrastructure that reduces operational complexity and total cost of ownership.

To learn how you can build scalable, real-time data and machine learning pipelines:

[SCHEDULE A PERSONALIZED DEMO](#)

[SIGN-UP FOR A FREE TRIAL](#)