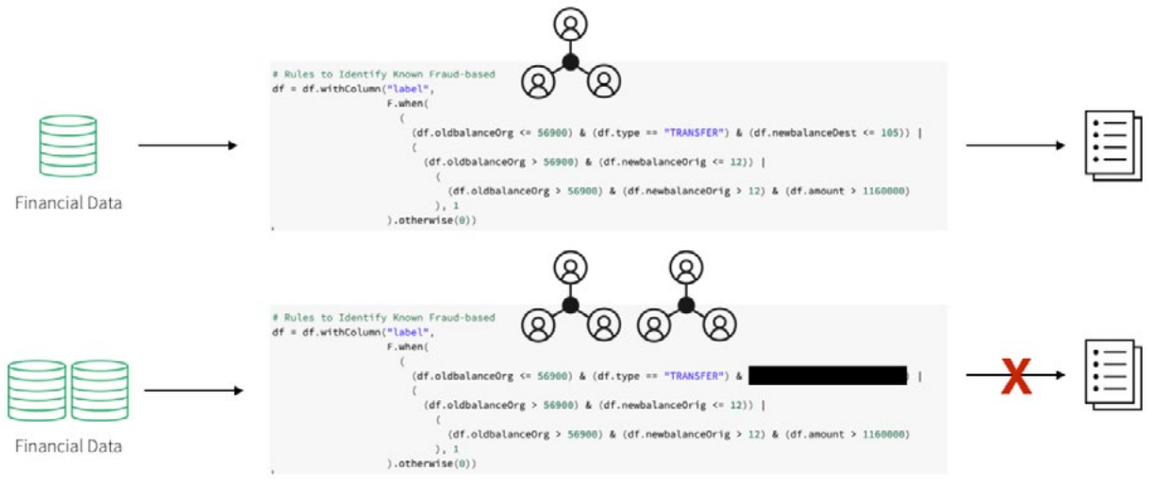
Detecting Financial Fraud at Scale with Decision Trees and MLflow on Databricks

A Databricks guide, including code samples and notebooks.

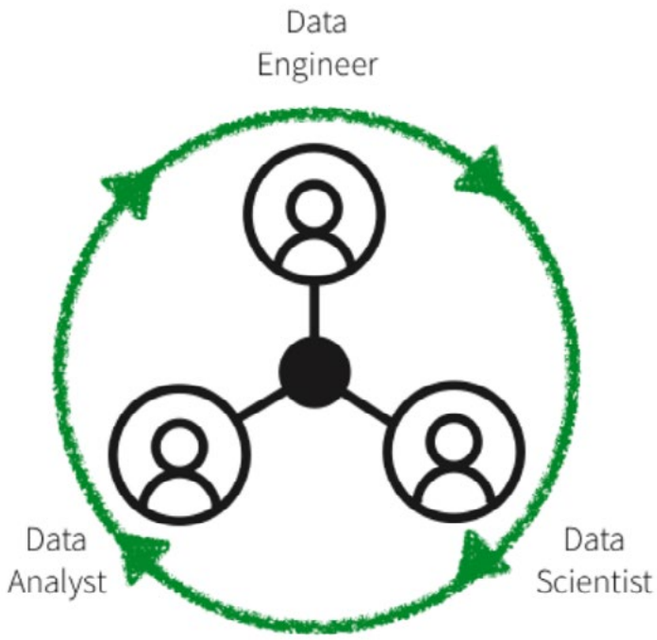
Introduction

Detecting fraudulent patterns at scale is a challenge, no matter the use case. The massive amounts of data to sift through, the complexity of the constantly evolving techniques, and the very small number of actual examples of fraudulent behavior are comparable to finding a needle in a haystack while not knowing what the needle looks like. In the world of finance, the added concerns with security and the importance of explaining how fraudulent behavior was identified further increases the complexity of the task.



To build these detection patterns, a team of domain experts often comes up with a set of rules that define fraudulent behavior. A typical workflow may include a subject matter expert in the financial fraud detection space putting together a set of requirements for a particular behavior. A data scientist may then take a subsample of the available data and build a model using these requirements and possibly some known fraud cases. To put the pattern in production, a data engineer may convert the resulting model to a set of rules with thresholds, often implemented using SQL.

This approach allows the financial institution to present a clear set of characteristics that led to the identification of fraud that is compliant with the General Data Protection Regulation (GDPR). However, this approach also poses numerous difficulties. The implementation of the detection pattern using a hardcoded set of rules is very brittle. Any changes to the pattern would take a very long time to update. This, in turn, makes it difficult to keep up with and adapt to the shift in fraudulent behaviors that are happening in the current marketplace.

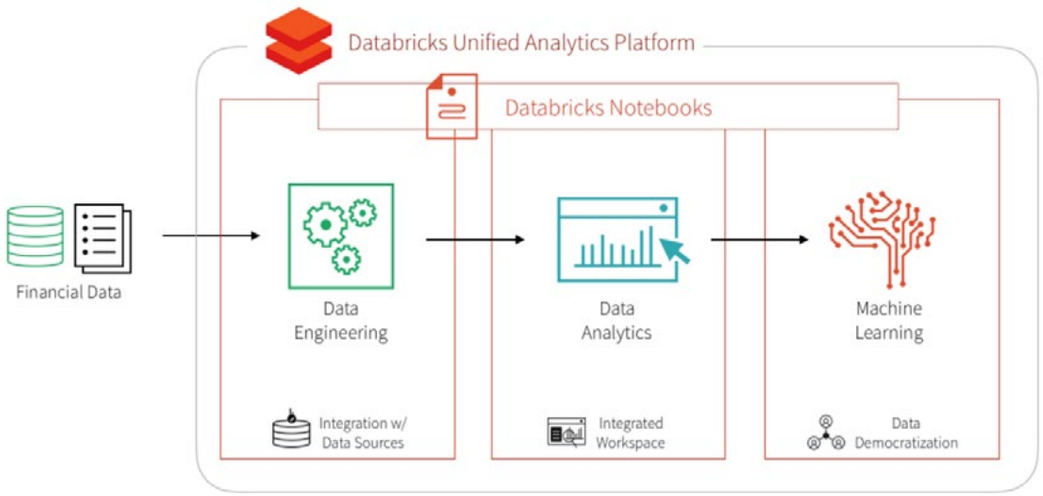


Additionally, the systems in the workflow described above are often siloed, with the domain experts, data scientists, and data engineers all compartmentalized. The data engineer is responsible for maintaining massive amounts of data and translating the work of the domain experts and data scientists into production level code. Due to a lack of common platform, the domain experts and data scientists have to rely on sampled down data that fits on a single machine for analysis. This leads to difficulty in communication and ultimately a lack of collaboration.

In this eBook, we will showcase how to convert several such rule-based detection use cases to machine learning use cases on the Databricks platform, unifying the key players in fraud detection: domain experts, data scientists, and data engineers. We will learn how to create a fraud-detection data pipeline and visualize the data leveraging a framework for building modular features from large data sets. We will also learn how to detect fraud using decision trees and Apache Spark MLlib. We will then use MLflow to iterate and refine the model to improve its accuracy.

SOLVING WITH ML

There is a certain degree of reluctance with regard to machine learning models in the financial world as they are believed to offer a "black box" solution with no way of justifying the identified fraudulent cases. GDPR requirements, as well as financial regulations, make it seemingly impossible to leverage the power of machine learning. However, several successful use cases have shown that applying machine learning to detect fraud at scale can solve a host of the issues mentioned.



Training a supervised machine learning model to detect financial fraud is very difficult due to the low number of actual confirmed examples of fraudulent behavior. However, the presence of a known set of rules that identify a particular type of fraud can help create a set of synthetic labels and an initial set of features. The output of the detection pattern that has been developed by the domain experts in the field has likely gone through the appropriate approval process to be put in production. It produces the expected fraudulent behavior flags and may, therefore, be used as a starting point to train a machine learning model. This simultaneously mitigates three concerns:

1. The lack of training labels,
2. The decision of what features to use, and
3. Having an appropriate benchmark for the model.

Training a machine learning model to recognize the rule-based fraudulent behavior flags offers a direct comparison with the expected output via a confusion matrix. Provided that the results closely match the rule-based detection pattern, this approach helps gain confidence in machine learning based fraud detection with the skeptics. The output of this model is very easy to interpret and may serve as a baseline discussion of the expected false negatives and false positives when compared to the original detection pattern.

Furthermore, the concern with machine learning models being difficult to interpret may be further assuaged if a decision tree model is used as the initial machine learning model. Because the model is being trained to a set of rules, the decision tree is likely to outperform any other machine learning model. The additional benefit is, of course, the utmost transparency of the model, which will essentially show the decision-making process for fraud, but without human intervention and the need to hard code any rules or thresholds. Of course, it must be understood that the future iterations of the model may utilize a different algorithm altogether to achieve maximum accuracy. The transparency of the model is ultimately achieved by understanding the features that went into the algorithm. Having interpretable features will yield interpretable and defensible model results.

The biggest benefit of the machine learning approach is that after the initial modeling effort, future iterations are modular and updating the set of labels, features, or model type is very easy and seamless, reducing the time to production. This is further facilitated on the Databricks Unified Analytics Platform where the domain experts, data scientists, data engineers may work off the same data set at scale and collaborate directly in the notebook environment. So let's get started!

Ingesting and Exploring the Data

We will use a synthetic dataset for this example. To load the dataset yourself, please download it to your local machine from Kaggle and then import the data via Import Data – Azure and AWS

The PaySim data simulates mobile money transactions based on a sample of real transactions extracted from one month of financial logs from a mobile money service implemented in an African country. The below table shows the information that the data set provides:

Column Name	Description
step	maps a unit of time in the real world. In this case 1 step is 1 hour of time. Total steps 744 (30 days simulation).
type	CASH-IN, CASH-OUT, DEBIT, PAYMENT and TRANSFER.
amount	amount of the transaction in local currency.
nameOrig	customer who started the transaction
oldbalanceOrg	initial balance before the transaction
newbalanceOrig	new balance after the transaction
nameDest	customer who is the recipient of the transaction
oldbalanceDest	initial balance recipient before the transaction. Note that there is not information for customers that start with M (Merchants).
newbalanceDest	new balance recipient after the transaction. Note that there is not information for customers that start with M (Merchants).

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 eBook, 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

EXPLORING THE DATA

Creating the DataFrames – Now that we have uploaded the data to Databricks File System (DBFS), we can quickly and easily create DataFrames using Spark SQL

```
# Create df DataFrame which contains our simulated financial
fraud detection dataset
df = spark.sql("select step, type, amount, nameOrig,
oldbalanceOrg, newbalanceOrig, nameDest, oldbalanceDest,
newbalanceDest from sim_fin_fraud_detection")
```

Now that we have created the DataFrame, let's take a look at the schema and the first thousand rows to review the data.

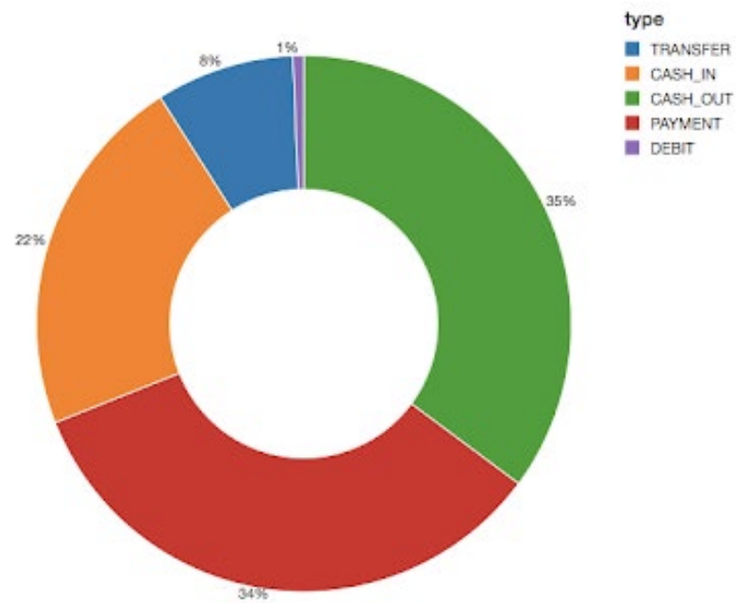
```
# Review the schema of your data
df.printSchema()
root
|-- step: integer (nullable = true)
|-- type: string (nullable = true)
|-- amount: double (nullable = true)
|-- nameOrig: string (nullable = true)
|-- oldbalanceOrg: double (nullable = true)
|-- newbalanceOrig: double (nullable = true)
|-- nameDest: string (nullable = true)
|-- oldbalanceDest: double (nullable = true)
|-- newbalanceDest: double (nullable = true)
```

step	type	amount	nameOrig	oldbalanceOrg	newbalanceOrig
1	PAYMENT	9839.64	C1231006815	170136	160296.36
1	PAYMENT	1864.28	C1666544295	21249	19384.72
1	TRANSFER	181	C1305486145	181	0
1	CASH_OUT	181	C840083671	181	0
1	PAYMENT	11668.14	C2048537720	41554	29885.86
1	PAYMENT	7817.71	C90045638	53860	46042.29
1	PAYMENT	7107.77	C154988899	183195	176087.23
1	PAYMENT	7861.64	C1912850431	176087.23	168225.59

TYPES OF TRANSACTIONS

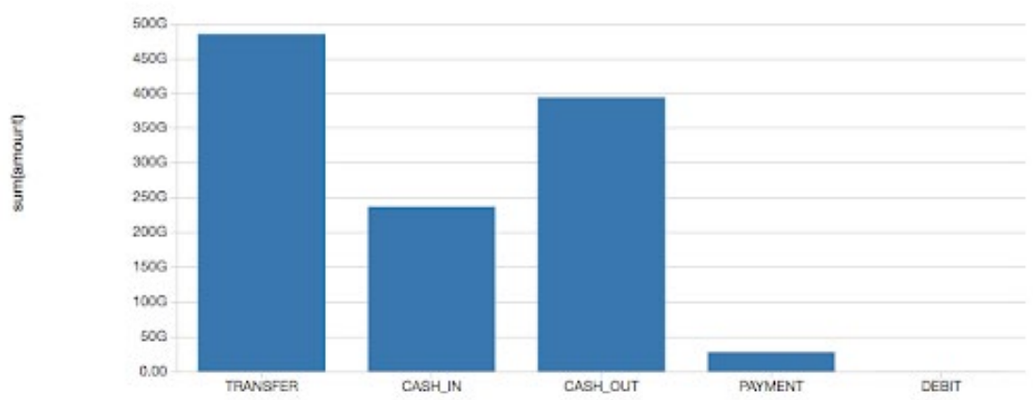
Let's visualize the data to understand the types of transactions the data captures and their contribution to the overall transaction volume.

```
%sql
-- Organize by Type
select type, count(1) from financials group by type
```



To get an idea of how much money we are talking about, let's also visualize the data based on the types of transactions and on their contribution to the amount of cash transferred (i.e. sum(amount)).

```
%sql
select type, sum(amount) from financials group by type
```



RULES-BASED MODEL

We are not likely to start with a large data set of known fraud cases to train our model. In most practical applications, fraudulent detection patterns are identified by a set of rules established by the domain experts. Here, we create a column called `label` based on these rules.

```
# Rules to Identify Known Fraud-based
df = df.withColumn("label",
                   F.when(
                     (
                       (df.oldbalanceOrg <= 56900) & (df.type ==
"TRANSFER") & (df.newbalanceDest <= 105)) | ( (df.oldbalanceOrg
> 56900) & (df.newbalanceOrig <= 12)) | ( (df.oldbalanceOrg >
56900) & (df.newbalanceOrig > 12) & (df.amount > 1160000)
                           ), 1
                   ).otherwise(0))
```

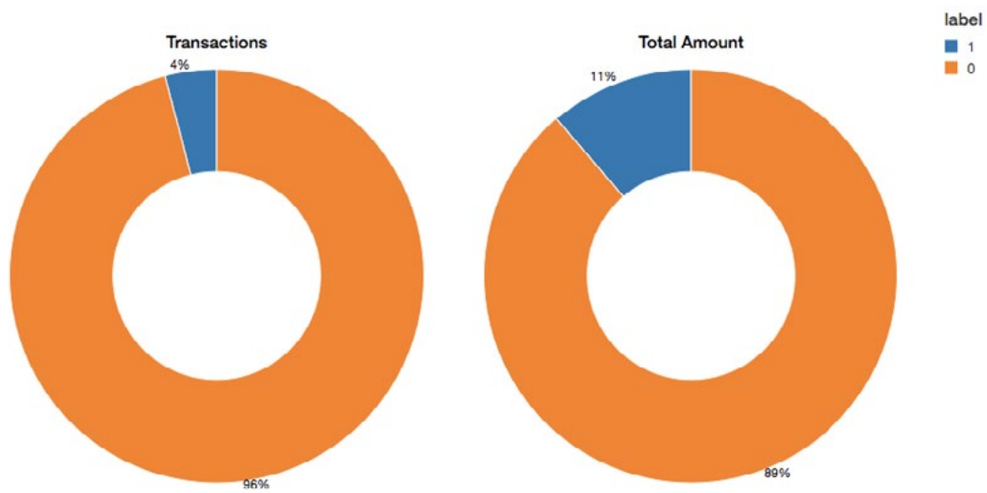
After this ETL process is completed, you can use the `display` command again to review the cleansed data in a scatterplot.

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

VISUALIZING DATA FLAGGED BY RULES

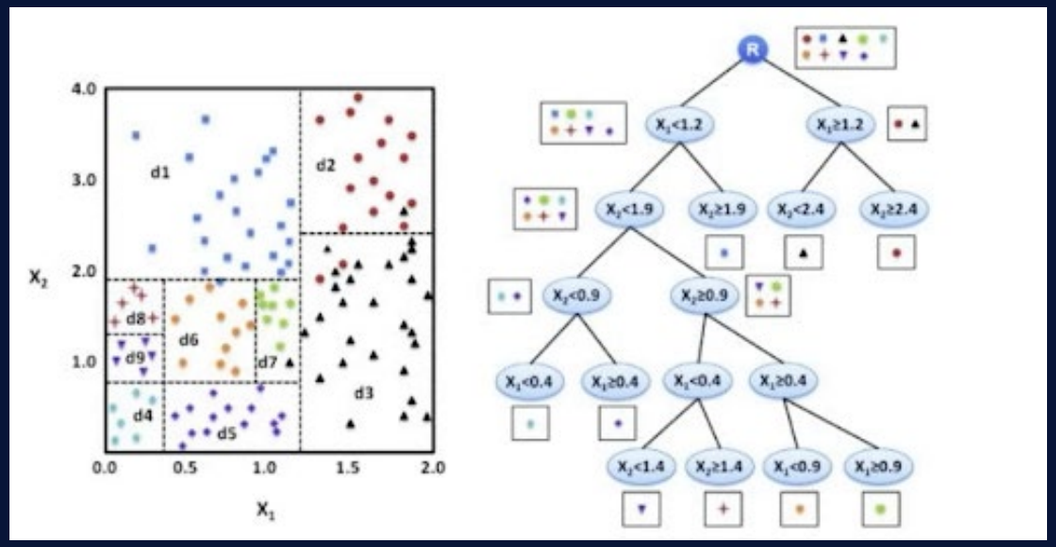
These rules often flag quite a large number of fraudulent cases. Let's visualize the number of flagged transactions. We can see that the rules flag about 4% of the cases and 11% of the total dollar amount as fraudulent.

```
%sql
select label, count(1) as `Transactions`, sum(amount) as `Total Amount` from financials_labeled group by label
```



Selecting the Appropriate Machine Learning Models

In many cases, a black box approach to fraud detection cannot be used. First, the domain experts need to be able to understand why a transaction was identified as fraudulent. Then, if action is to be taken, the evidence has to be presented in court. The decision tree is an easily interpretable model and is a great starting point for this use case. Read this blog "The wise old tree" on decision trees to learn more.



CREATING THE TRAINING SET

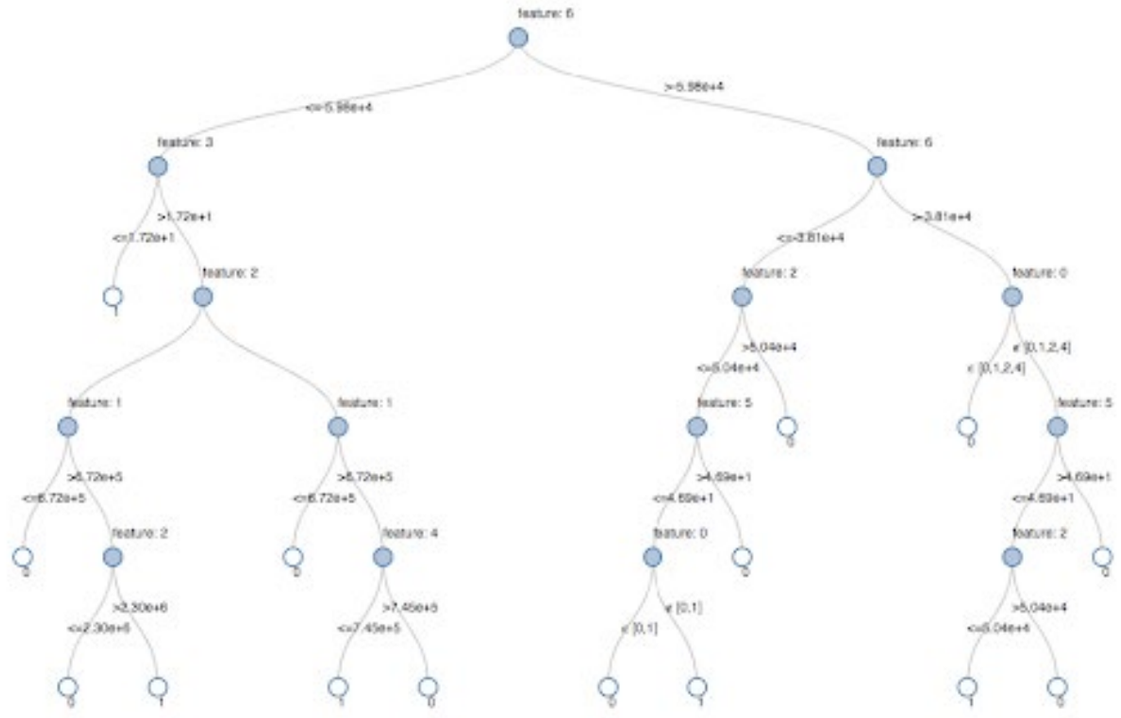
To build and validate our ML model, we will do an 80/20 split using `.randomSplit.` This will set aside a randomly chosen 80% of the data for training and the remaining 20% to validate the results.

```
# Split our dataset between training and test datasets
(train, test) = df.randomSplit([0.8, 0.2], seed=12345)
```

CREATING THE TRAINING SET

To prepare the data for the model, we must first convert categorical variables to numeric using `.StringIndexer.` We then must assemble all of the features we would like for the model to use. We create a pipeline to contain these feature preparation steps in addition to the decision tree model so that we may repeat these steps on different data sets. Note that we fit the pipeline to our training data first and will then use it to transform our test data in a later step.

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import DecisionTreeClassifier

# Encodes a string column of labels to a column of label indices
indexer = StringIndexer(inputCol = "type", outputCol =
"typeIndexed")

# VectorAssembler is a transformer that combines a given list of
columns into a single vector column
va = VectorAssembler(inputCols = ["typeIndexed", "amount",
"oldbalanceOrg", "newbalanceOrig", "oldbalanceDest",
"newbalanceDest", "orgDiff", "destDiff"], outputCol =
"features")

# Using the DecisionTree classifier model
dt = DecisionTreeClassifier(labelCol = "label", featuresCol =
"features", seed = 54321, maxDepth = 5)

# Create our pipeline stages
pipeline = Pipeline(stages=[indexer, va, dt])

# View the Decision Tree model (prior to CrossValidator)
dt_model = pipeline.fit(train)
```



Visual representation of the Decision Tree model

VISUALIZING THE MODEL

Calling `display()` on the last stage of the pipeline, which is the decision tree model, allows us to view the initial fitted model with the chosen decisions at each node. This helps to understand how the algorithm arrived at the resulting predictions.

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

MODEL TUNING

To ensure we have the best fitting tree model, we will cross-validate the model with several parameter variations. Given that our data consists of 96% negative and 4% positive cases, we will use the Precision-Recall (PR) evaluation metric to account for the unbalanced distribution.

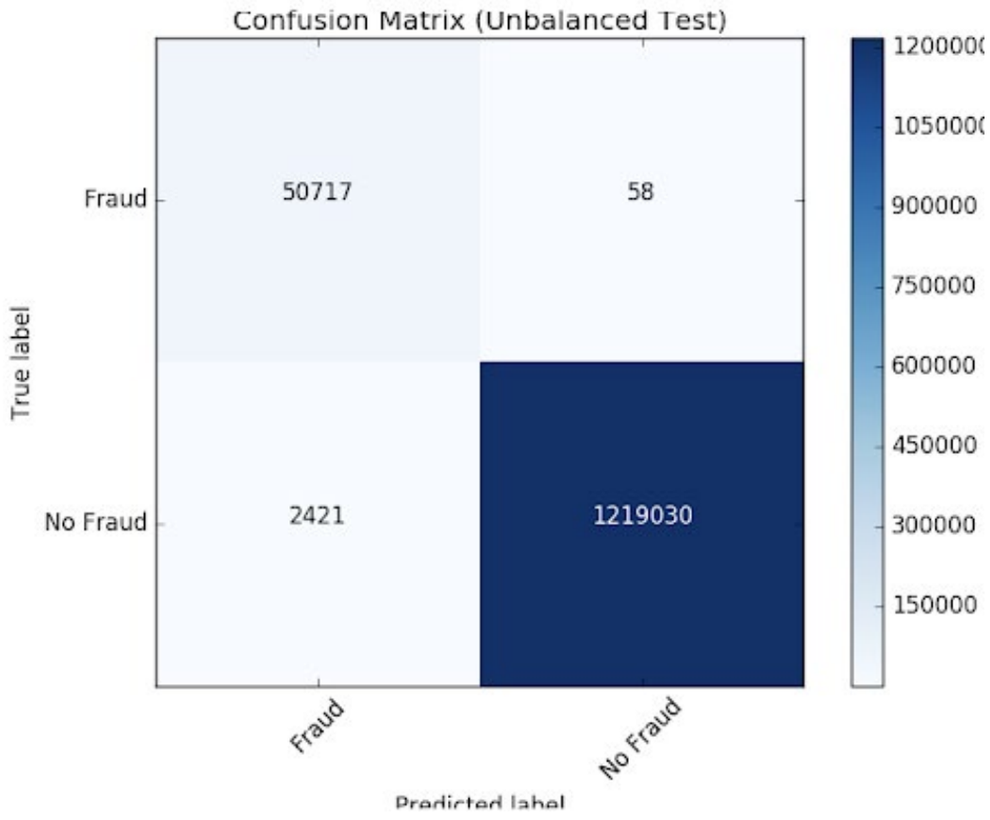
```
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

# Build the grid of different parameters
paramGrid = ParamGridBuilder() \
.addGrid(dt.maxDepth, [5, 10, 15]) \
.addGrid(dt.maxBins, [10, 20, 30]) \
.build()

# Build out the cross validation
crossval = CrossValidator(estimator = dt,
                          estimatorParamMaps = paramGrid,
                          evaluator = evaluatorPR,
                          numFolds = 3)
# Build the CV pipeline
pipelineCV = Pipeline(stages=[indexer, va, crossval])

# Train the model using the pipeline, parameter grid, and
preceding BinaryClassificationEvaluator
cvModel_u = pipelineCV.fit(train)
```

MODEL PERFORMANCE

We evaluate the model by comparing the Precision-Recall (PR) and Area under the ROC curve (AUC) metrics for the training and test sets. Both PR and AUC appear to be very high.

```
# Build the best model (training and test datasets)
train_pred = cvModel_u.transform(train)
test_pred = cvModel_u.transform(test)

# Evaluate the model on training datasets
pr_train = evaluatorPR.evaluate(train_pred)
auc_train = evaluatorAUC.evaluate(train_pred)

# Evaluate the model on test datasets
pr_test = evaluatorPR.evaluate(test_pred)
auc_test = evaluatorAUC.evaluate(test_pred)

# Print out the PR and AUC values
print(“PR train:”, pr_train)
print(“AUC train:”, auc_train)
print(“PR test:”, pr_test)
print(“AUC test:”, auc_test)

---
# Output:
# PR train: 0.9537894984523128
# AUC train: 0.998647996459481
# PR test: 0.9539170535377599
# AUC test: 0.9984378183482442
```

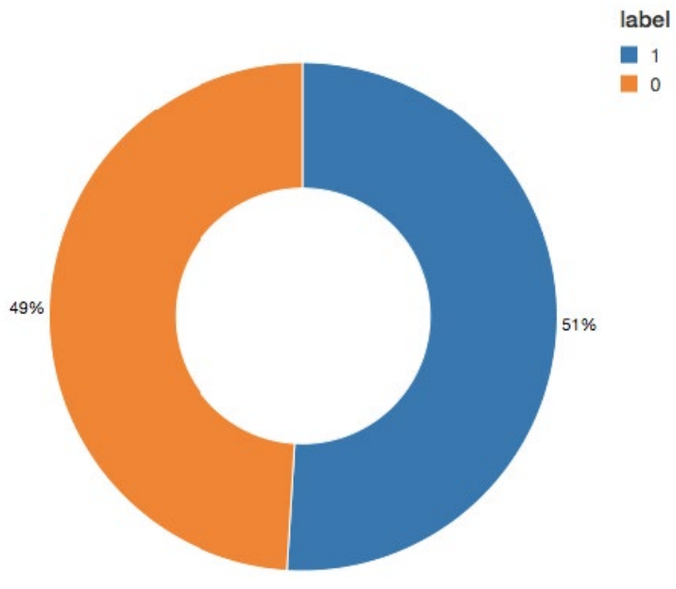
To see how the model misclassified the results, let’s use matplotlib and pandas to visualize our confusion matrix.




```
# Reset the DataFrames for no fraud (`dfn`) and fraud (`dfy`)
dfn = train.filter(train.label == 0)
dfy = train.filter(train.label == 1)

# Calculate summary metrics
N = train.count()
y = dfy.count()
p = y/N

# Create a more balanced training dataset
train_b = dfn.sample(False, p, seed = 92285).union(dfy)

# Print out metrics
print("Total count: %s, Fraud cases count: %s, Proportion of
fraud cases: %s" % (N, y, p))
print("Balanced training dataset count: %s" % train_b.count())

---
# Output:
# Total count: 5090394, Fraud cases count: 204865, Proportion of
fraud cases: 0.040245411258932016
# Balanced training dataset count: 401898
---

# Display our more balanced training dataset
display(train_b.groupBy("label").count())
```

BALANCING THE CLASSES

We see that the model is identifying 2421 more cases than the original rules identified. This is not as alarming as detecting more potential fraudulent cases could be a good thing. However, there are 58 cases that were not detected by the algorithm but were originally identified. We are going to attempt to improve our prediction further by balancing our classes using undersampling. That is, we will keep all the fraud cases and then downsample the non-fraud cases to match that number to get a balanced data set. When we visualized our new data set, we see that the yes and no cases are 50/50.



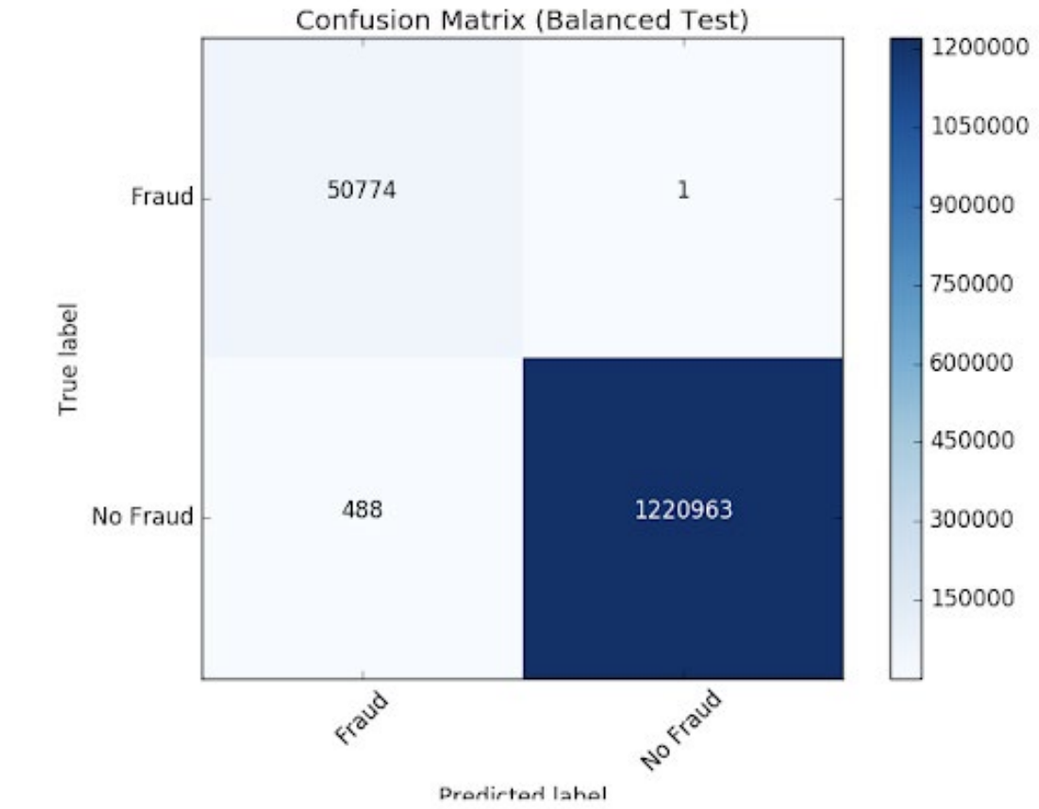
UPDATING THE PIPELINE

Now let's update the ML pipeline and create a new cross validator. Because we are using ML pipelines, we only need to update it with the new dataset and we can quickly repeat the same pipeline steps.

```
# Re-run the same ML pipeline (including parameters grid)
crossval_b = CrossValidator(estimator = dt,
estimatorParamMaps = paramGrid,
evaluator = evaluatorAUC,
numFolds = 3)
pipelineCV_b = Pipeline(stages=[indexer, va, crossval_b])

# Train the model using the pipeline, parameter grid, and
BinaryClassificationEvaluator using the `train_b` dataset
cvModel_b = pipelineCV_b.fit(train_b)

# Build the best model (balanced training and full test datasets)
train_pred_b = cvModel_b.transform(train_b)
test_pred_b = cvModel_b.transform(test)

# Evaluate the model on the balanced training datasets
pr_train_b = evaluatorPR.evaluate(train_pred_b)
auc_train_b = evaluatorAUC.evaluate(train_pred_b)

# Evaluate the model on full test datasets
pr_test_b = evaluatorPR.evaluate(test_pred_b)
auc_test_b = evaluatorAUC.evaluate(test_pred_b)

# Print out the PR and AUC values
print("PR train:", pr_train_b)
print("AUC train:", auc_train_b)
print("PR test:", pr_test_b)
print("AUC test:", auc_test_b)

---
# Output:
# PR train: 0.999629161563572
# AUC train: 0.9998071389056655
# PR test: 0.9904709171789063
# AUC test: 0.9997903902204509
```

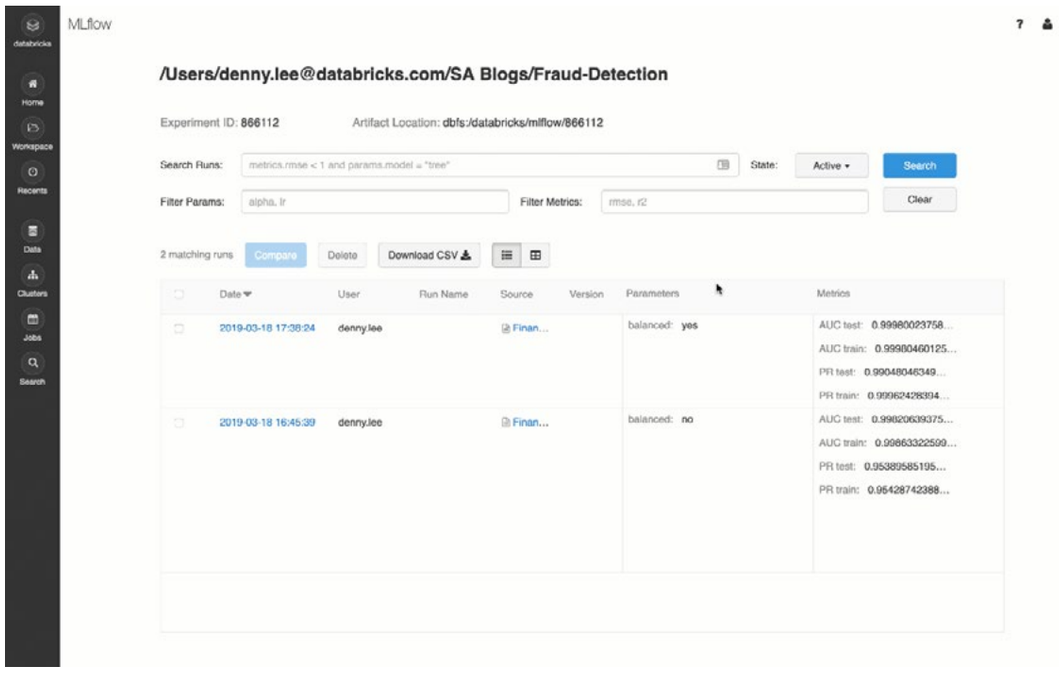
REVIEW THE RESULTS

Now let's look at the results of our new confusion matrix. The model misidentified only one fraudulent case. Balancing the classes seems to have improved the model.



MODEL FEEDBACK AND USING MLFLOW

Once a model is chosen for production, we want to continuously collect feedback to ensure that the model is still identifying the behavior of interest. Since we are starting with a rule-based label, we want to supply future models with verified true labels based on human feedback. This stage is crucial for maintaining confidence and trust in the machine learning process. Since analysts are not able to review every single case, we want to ensure we are presenting them with carefully chosen cases to validate the model output. For example, predictions, where the model has low certainty, are good candidates for analysts to review. The addition of this type of feedback will ensure the models will continue to improve and evolve with the changing landscape.



MLflow helps us throughout this cycle as we train different model versions. We can keep track of our experiments, comparing the results of different model configurations and parameters. For example here, we can compare the PR and AUC of the models trained on balanced and unbalanced data sets using the MLflow UI. Data scientists can use MLflow to keep track of the various model metrics and any additional visualizations and artifacts to help make the decision of which model should be deployed in production. The data engineers will then be able to easily retrieve the chosen model along with the library versions used for training as a .jar file to be deployed on new data in production. Thus, the collaboration between the domain experts who review the model results, the data scientists who update the models, and the data engineers who deploy the models in production, will be strengthened throughout this iterative process.

CONCLUSION

We have reviewed an example of how to use a rule-based fraud detection label and convert it to a machine learning model using Databricks with MLflow. This approach allows us to build a scalable, modular solution that will help us keep up with ever-changing fraudulent behavior patterns. Building a machine learning model to identify fraud allows us to create a feedback loop that allows the model to evolve and identify new potential fraudulent patterns. We have seen how a decision tree model, in particular, is a great starting point to introduce machine learning to a fraud detection program due to its interpretability and excellent accuracy.

A major benefit of using the Databricks platform for this effort is that it allows for data scientists, engineers, and business users to seamlessly work together throughout the process. Preparing the data, building models, sharing the results, and putting the models into production can now happen on the same platform, allowing for unprecedented collaboration. This approach builds trust across the previously siloed teams, leading to an effective and dynamic fraud detection program.

Try this notebook by signing up for a free trial in just a few minutes and get started creating your own models.