

Insurance Claims - Fraud Detection

Business case:

Insurance fraud is a huge problem in the industry. It's difficult to identify fraud claims. IHS is in a unique position to help the Auto Insurance industry with this problem.

Problem Statement:

Data is stored in different systems and its difficult to build analytics using multiple data sources. Copying data into a single platform is time consuming.

Business solution:

Use S3 as a data lake to store different sources of data in a single platform. This allows data scientists / analysis to quickly analyze the data and generate reports to predict market trends and/or make financial decisions.

Technical Solution:

Use Databricks as a single platform to pull various sources of data from API endpoints, or batch dumps into S3 for further processing. ETL the CSV datasets into efficient Parquet formats for performant processing.

In this example, we will be working with some auto insurance data to demonstrate how we can create a predictive model that predicts if an insurance claim is fraudulent or not. This will be a Binary Classification task, and we will be creating a Decision Tree model.

With the prediction data, we are able to estimate what our total predicted fraudulent claim amount is like, and zoom into various features such as a breakdown of predicted fraud count by insured hobbies - our model's best predictor.

We will cover the following steps to illustrate how we build a Machine Learning Pipeline:

- Data Import
- Data Exploration

- Data Processing
- Create Decision Tree Model
- Measuring Error Rate
- Model Tuning
- Zooming in on Prediction Data

Data Import

First download this csv file

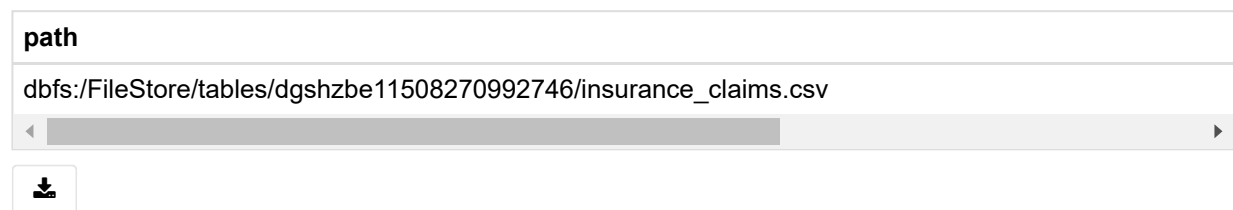
(https://raw.githubusercontent.com/joddb/sparkTestData/master/insurance_claims.csv)
locally

The data used in this example was from a CSV file that was imported using the Tables UI. Databases and Tables (<https://docs.databricks.com/user-guide/tables.html>)

After uploading the data using the UI, we can run SparkSQL queries against the table, or create a DataFrame from the table.

In this example, we will create a Spark DataFrame.

```
display(dbutils.fs.ls("/FileStore/tables/dgshzbe11508270992746/"))
```



```
# data = spark.table("insurance_claims")
```

```
fileStorePath = "/FileStore/tables/dgshzbe11508270992746/insurance_claims.csv"
```

```
data = spark.read.format("csv")\  
    .options(inferSchema="true", header="true")\  
    .load(fileStorePath)\  
    .drop("_c39")
```

```
df = data.withColumn("policy_bind_date", data.policy_bind_date.cast("string"))\  
    .withColumn("incident_date", data.incident_date.cast("string"))
```

```
# Preview data
display(df)
```

months_as_customer	age	policy_number	policy_bind_date	policy_state	policy_csl	policy_c
328	48	521585	2014-10-17 00:00:00	OH	250/500	1000
228	42	342868	2006-06-27 00:00:00	IN	250/500	2000
134	29	687698	2000-09-06 00:00:00	OH	100/300	2000
256	41	227811	1990-05-25 00:00:00	IL	250/500	2000
228	44	367455	2014-06-06	IL	500/1000	1000



Data Exploration

We have several string (categorical) columns in our dataset, along with some ints and doubles.

```
display(df.dtypes)
```

_1
months_as_customer
age
policy_number
policy_bind_date
policy_state
policy_csl
policy_deductable
policy_annual_premium
umbrella_limit



Count number of categories for every categorical column (Count Distinct).

```
# Create a List of Column Names with data type = string
stringColList = [i[0] for i in df.dtypes if i[1] == 'string']
print stringColList

['policy_bind_date', 'policy_state', 'policy_csl', 'insured_sex', 'insured_education_level', 'insured_occupation', 'insured_hobbies', 'insured_relationship', 'incident_date', 'incident_type', 'collision_type', 'incident_severity', 'authorities_contacted', 'incident_state', 'incident_city', 'incident_location', 'property_damage', 'police_report_available', 'auto_make', 'auto_model', 'fraud_reported']

from pyspark.sql.functions import *

# Create a function that performs a countDistinct(colName)
distinctList = []
def countDistinctCats(colName):
    count = df.agg(countDistinct(colName)).collect()
    distinctList.append(count)

# Apply function on every column in stringColList
map(countDistinctCats, stringColList)
print distinctList

[[Row(count(DISTINCT policy_bind_date)=951)], [Row(count(DISTINCT policy_state)=3)], [Row(count(DISTINCT policy_csl)=3)], [Row(count(DISTINCT insured_sex)=2)], [Row(count(DISTINCT insured_education_level)=7)], [Row(count(DISTINCT insured_occupation)=14)], [Row(count(DISTINCT insured_hobbies)=20)], [Row(count(DISTINCT insured_relationship)=6)], [Row(count(DISTINCT incident_date)=60)], [Row(count(DISTINCT incident_type)=4)], [Row(count(DISTINCT collision_type)=4)], [Row(count(DISTINCT incident_severity)=4)], [Row(count(DISTINCT authorities_contacted)=5)], [Row(count(DISTINCT incident_state)=7)], [Row(count(DISTINCT incident_city)=7)], [Row(count(DISTINCT incident_location)=1000)], [Row(count(DISTINCT property_damage)=3)], [Row(count(DISTINCT police_report_available)=3)], [Row(count(DISTINCT auto_make)=14)], [Row(count(DISTINCT auto_model)=39)], [Row(count(DISTINCT fraud_reported)=2)]]
```

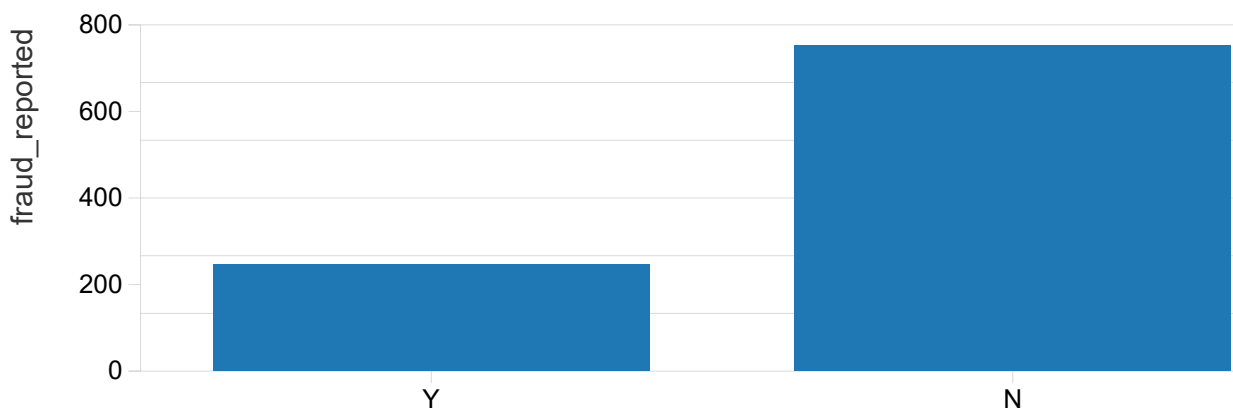
We have identified that some string columns have many distinct values (900+). We will remove these columns from our dataset in the Data Processing step to improve model accuracy.

- policy number (1000 distinct)

- policy bind date (951 distinct. Possible to narrow down to year/month to test model accuracy)
- insured zip (995 distinct)
- insured location (1000 distinct)
- incident date (60 distinct. Excluding, but possible to narrow down to year/month to test model accuracy)

Like most fraud datasets, our label distribution is skewed.

```
display(df)
```



```
# Count number of frauds vs non-frauds  
display(df.groupby("fraud_reported").count())
```

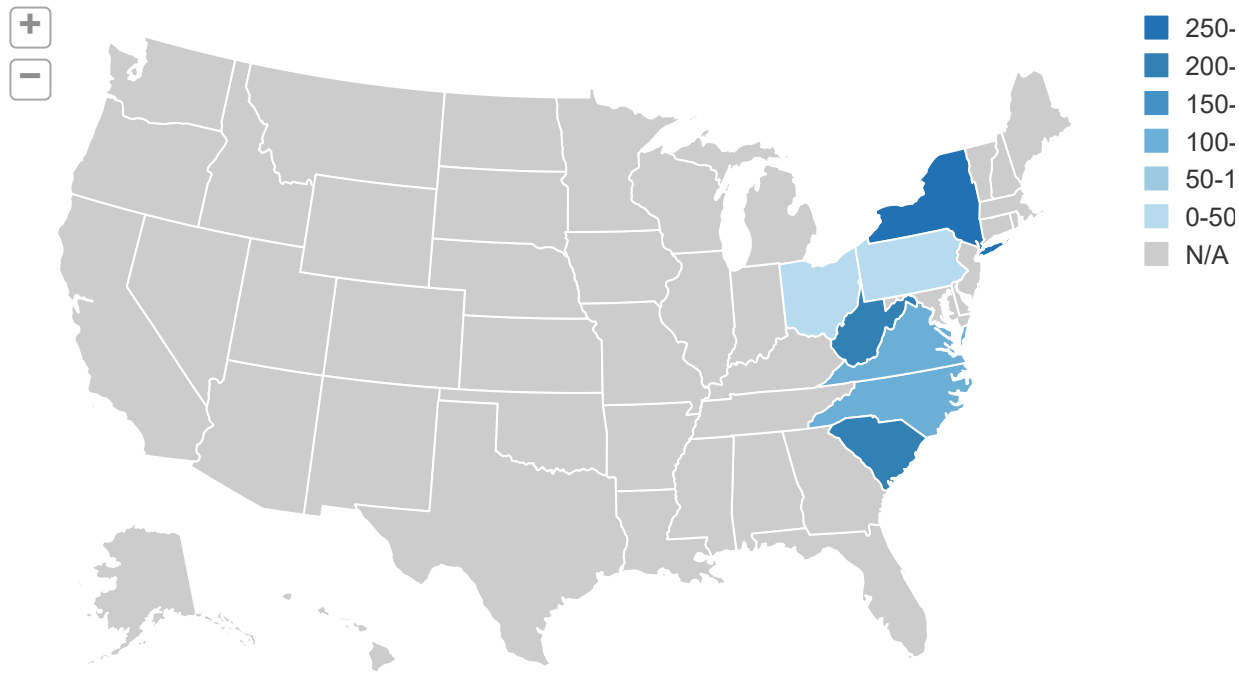
fraud_reported	
Y	
N	



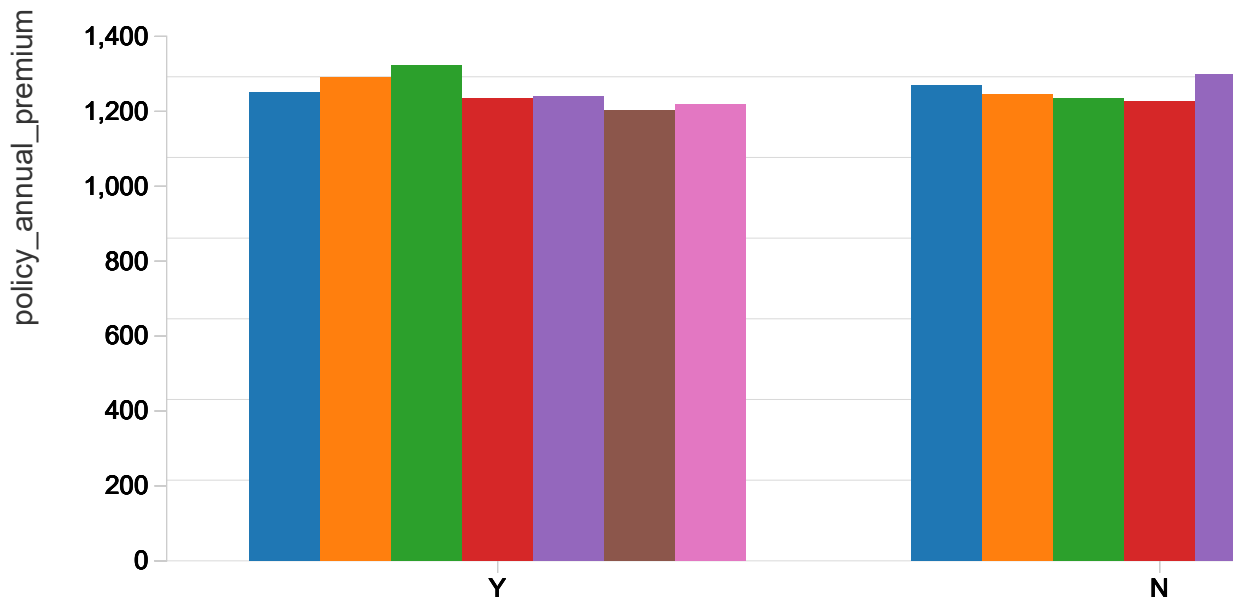
We can quickly create one-click plots using Databricks built-in visualizations to understand our data better.

Click 'Plot Options' to try out different chart types.

Fraud Count by Incident State
display(df)



Breakdown of Average Vehicle claim by insured's education level, grouped by
fraud reported
display(df)



Data Processing

Next, we will clean up the data a little and prepare it for our machine learning model.

We will first remove the columns that we have identified earlier that have too many distinct categories and cannot be converted to numeric.

```
colsToDelete = ["policy_number", "policy_bind_date", "insured_zip",  
"incident_location", "incident_date"]  
filteredStringColList = [i for i in stringColList if i not in colsToDelete]
```

We will convert categorical columns to numeric to pass them into various algorithms. This can be done using the StringIndexer.

Here, we are generating a StringIndexer for each categorical column and appending it as a stage of our ML Pipeline.

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

```
transformedCols = [categoricalCol + "Index" for categoricalCol in
filteredStringColList]
stages = [StringIndexer(inputCol = categoricalCol, outputCol = categoricalCol +
"Index") for categoricalCol in filteredStringColList]
stages
```

```
Out[13]:
```

```
[StringIndexer_408181119f0227ff147c,
StringIndexer_41148ddb6cb856a8d31c,
StringIndexer_4a3bb118c52da71fe6cc,
StringIndexer_489ba34db8df41a53a81,
StringIndexer_4b3da0485629810fd9d4,
StringIndexer_4fd78b353a55d8e936dc,
StringIndexer_4348aef57b30273bf8f9,
StringIndexer_4fe88cf62d37386f5fb5,
StringIndexer_4e4786411c4e2e8a6969,
StringIndexer_4a9aa5d61aed15ac6f3d,
StringIndexer_4131b2d203b0c3cf0849,
StringIndexer_426d810fe593d4ee3767,
StringIndexer_415dbc8bc66636aa8a38,
StringIndexer_4ae19c90e69bc752ae31,
StringIndexer_4c33a3409dc7df076541,
StringIndexer_4b0f884df5b3afe78fa4,
StringIndexer_4d458b893ab4bba5a62a,
StringIndexer_4e4cb09fd5b3bfe8f908]
```


As an example, this is what the transformed dataset will look like after applying the StringIndexer on all categorical columns.

```
from pyspark.ml import Pipeline
```

```
indexer = Pipeline(stages=stages)
indexed = indexer.fit(df).transform(df)
display(indexed)
```

months_as_customer	age	policy_number	policy_bind_date	policy_state	policy_csl	policy_c
328	48	521585	2014-10-17 00:00:00	OH	250/500	1000
228	42	342868	2006-06-27 00:00:00	IN	250/500	2000

134	29	687698	2000-09-06 00:00:00	OH	100/300	2000
-----	----	--------	------------------------	----	---------	------



Use the VectorAssembler to combine all the feature columns into a single vector column. This will include both the numeric columns and the indexed categorical columns.

```
from pyspark.ml.feature import VectorAssembler

# In this dataset, numericColList will contain columns of type Int and Double
numericColList = [i[0] for i in df.dtypes if i[1] != 'string']
assemblerInputs = map(lambda c: c + "Index", filteredStringColList) +
numericColList

# Remove label from list of features
label = "fraud_reportedIndex"
assemblerInputs.remove(label)
assemblerInputs
assembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features")

# Append assembler to stages, which currently contains the StringIndexer
transformers
stages += [assembler]
```

Generate transformed dataset. This will be the dataset that we will use to create our machine learning models.

```

pipeline = Pipeline(stages=stages)
pipelineModel = pipeline.fit(df)
transformed_df = pipelineModel.transform(df)

# Rename label column
transformed_df = transformed_df.withColumnRenamed('fraud_reportedIndex',
'label')

# Keep relevant columns (original columns, features, labels)
originalCols = df.columns
selectedcols = ["label", "fraud_reported", "features"] + originalCols
dataset = transformed_df.select(selectedcols)
display(dataset)

```

label	fraud_reported	features
1	Y	▶ [1,35,[],[0,0,1,3,5,17,3,1,1,2,0,1,2,2,2,13,328,48,521585,1000,1406.91,0,466
1	Y	▶ [1,35,[],[2,0,1,3,0,0,1,2,3,0,0,3,5,0,0,12,16,228,42,342868,2000,1197.22,50000
0	N	▶ [1,35,[],[0,1,0,5,4,13,0,0,0,0,0,0,2,1,1,0,0,134,29,687698,2000,1413.14,500000
1	Y	▶ [1,35,[], [1,0,0,5,9,13,5,1,2,2,0,6,1,0,1,4,21,256,41,227811,2000,1415.74,6000000,6081
0	N	▶ [1,35,[],



By selecting "label" and "fraud reported", we can infer that 0 corresponds to **No Fraud Reported** and 1 corresponds to **Fraud Reported**.

Next, split data into training and test sets.

```

### Randomly split data into training and test sets. set seed for
reproducibility
(trainingData, testData) = dataset.randomSplit([0.7, 0.3], seed = 100)
print trainingData.count()
print testData.count()

```

685

315

Databricks makes it easy to use multiple languages in the same notebook for your analyses. Just register your dataset as a temporary table and you can access it using a different language!

```
# Register data as temp table to jump to Scala
trainingData.createOrReplaceTempView("trainingData")
testData.createOrReplaceTempView("testData")
```

Create Decision Tree Model

We will create a decision tree model in Scala using the trainingData. This will be our initial model.

```
%scala
```

```
import org.apache.spark.ml.classification.{DecisionTreeClassifier,
DecisionTreeClassificationModel}
```

```
// Create DataFrames using our earlier registered temporary tables
val trainingData = spark.table("trainingData")
val testData = spark.table("testData")
```

```
// Create initial Decision Tree Model
val dt = new DecisionTreeClassifier()
    .setLabelCol("label")
    .setFeaturesCol("features")
    .setMaxDepth(5)
    .setMaxBins(40)
```

```
// Train model with Training Data
val dtModel = dt.fit(trainingData)
```

```
import org.apache.spark.ml.classification.{DecisionTreeClassifier, DecisionTreeClassificationModel}
trainingData: org.apache.spark.sql.DataFrame = [label: double, fraud_reported: string ... 40 more fields]
testData: org.apache.spark.sql.DataFrame = [label: double, fraud_reported: string ... 40 more fields]
dt: org.apache.spark.ml.classification.DecisionTreeClassifier = dtc_ef22c254d204
dtModel: org.apache.spark.ml.classification.DecisionTreeClassificationModel =
```

DecisionTreeClassificationModel (uid=dtc_ef22c254d204) of depth 5 with 49 nodes

```
%scala

// Make predictions on test data using the transform() method.
// .transform() will only use the 'features' column as input.
val dtPredictions = dtModel.transform(testData)

dtPredictions: org.apache.spark.sql.DataFrame = [label: double, fraud_reported: string ... 43 more fields]

%scala

// View model's predictions and probabilities of each prediction class
val selected = dtPredictions.select("label", "prediction", "probability")
display(selected)
```

label	prediction	probability
0	1	▶ [1,2,[],[0,1]]
0	0	▶ [1,2,[],[0.9615384615384616,0.038461538461538464]]
0	0	▶ [1,2,[],[0.9615384615384616,0.038461538461538464]]
0	0	▶ [1,2,[],[0.9615384615384616,0.038461538461538464]]
0	0	▶ [1,2,[],[1,0]]
0	0	▶ [1,2,[],[1,0]]
0	0	▶ [1,2,[],[0.9615384615384616,0.038461538461538464]]
0	0	▶ [1,2,[],[1,0]]
0	1	▶ [1,2,[],[0,1]]



Measuring Error Rate

Evaluate our initial model using the BinaryClassificationEvaluator.

```
%scala

import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator

// Evaluate model

val evaluator = new BinaryClassificationEvaluator()
evaluator.evaluate(dtPredictions)

import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
evaluator: org.apache.spark.ml.evaluation.BinaryClassificationEvaluator = binE
val_f84a7841ddaf
res4: Double = 0.7323090725580352
```

Model Tuning

We can tune our models using built-in libraries like `ParamGridBuilder` for Grid Search, and `CrossValidator` for Cross Validation. In this example, we will test out a combination of Grid Search with 5-fold Cross Validation.

Here, we will see if we can improve accuracy rates from our initial model.

```
%scala

// View tunable parameters for Decision Trees
dt.explainParams

res5: String =
cacheNodeIds: If false, the algorithm will pass trees to executors to match
instances with nodes. If true, the algorithm will cache node IDs for each in
stance. Caching can speed up training of deeper trees. (default: false)
checkpointInterval: set checkpoint interval (>= 1) or disable checkpoint (-
1). E.g. 10 means that the cache will get checkpointed every 10 iterations
(default: 10)
featuresCol: features column name (default: features, current: features)
impurity: Criterion used for information gain calculation (case-insensitiv
e). Supported options: entropy, gini (default: gini)
labelCol: label column name (default: label, current: label)
maxBins: Max number of bins for discretizing continuous features. Must be >
=2 and >= number of categories for any categorical feature. (default: 32, cu
rrent: 40)
maxDepth: Maximum depth of the tree. (>= 0) E.g., depth 0 means 1 leaf node;
depth 1 means 1 internal node + 2 leaf nodes. (default: 5, current: 5)
maxMemoryInMB: Maximum memory in MB allocated to histogram aggregation. (def
```

```
ault: 256)
minInfoGain: Minimum information gain for a split to be considered at a tree
node. (default: 0.0)
```

Create a ParamGrid to perform Grid Search. We will be adding various values of maxDepth and maxBins.

```
%scala
```

```
import org.apache.spark.ml.tuning.{ParamGridBuilder, CrossValidator}
```

```
val paramGrid = new ParamGridBuilder()
    .addGrid(dt.maxDepth, Array(3, 10, 15))
    .addGrid(dt.maxBins, Array(40, 50))
    .build()
```

```
import org.apache.spark.ml.tuning.{ParamGridBuilder, CrossValidator}
```

```
paramGrid: Array[org.apache.spark.ml.param.ParamMap] =
```

```
Array({
    dtc_ef22c254d204-maxBins: 40,
    dtc_ef22c254d204-maxDepth: 3
}, {
    dtc_ef22c254d204-maxBins: 40,
    dtc_ef22c254d204-maxDepth: 10
}, {
    dtc_ef22c254d204-maxBins: 40,
    dtc_ef22c254d204-maxDepth: 15
}, {
    dtc_ef22c254d204-maxBins: 50,
    dtc_ef22c254d204-maxDepth: 3
}, {
    dtc_ef22c254d204-maxBins: 50,
    dtc_ef22c254d204-maxDepth: 10
}, {
    dtc_ef22c254d204-maxBins: 50,
    dtc_ef22c254d204-maxDepth: 15
})
```

Perform 5-fold Cross Validation.

```
%scala

// Create 5-fold CrossValidator
val cv = new CrossValidator()
    .setEstimator(dt)
    .setEvaluator(evaluator)
    .setEstimatorParamMaps(paramGrid)
    .setNumFolds(5)

// Run cross validations
val cvModel = cv.fit(trainingData)

cv: org.apache.spark.ml.tuning.CrossValidator = cv_371536833f2a
cvModel: org.apache.spark.ml.tuning.CrossValidatorModel = cv_371536833f2a
```

We can print out what our Tree Model looks like using `toDebugString`.

```
%scala

val bestTreeModel =
cvModel.bestModel.asInstanceOf[DecisionTreeClassificationModel]
println("Learned classification tree model:\n" + bestTreeModel.toDebugString)
```

```
Learned classification tree model:
DecisionTreeClassificationModel (uid=dtc_ef22c254d204) of depth 3 with 15 nodes
  If (feature 9 in {0.0,1.0,3.0})
    If (feature 5 in {0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0,14.0,16.0,17.0,19.0})
      If (feature 16 in {0.0,1.0,3.0,4.0,8.0,10.0,12.0,15.0,17.0,19.0,20.0,21.0,23.0,25.0,26.0,27.0,28.0,29.0,30.0,31.0,32.0,33.0,34.0,36.0,37.0,38.0})
        Predict: 0.0
      Else (feature 16 not in {0.0,1.0,3.0,4.0,8.0,10.0,12.0,15.0,17.0,19.0,20.0,21.0,23.0,25.0,26.0,27.0,28.0,29.0,30.0,31.0,32.0,33.0,34.0,36.0,37.0,38.0})
        Predict: 0.0
      Else (feature 5 not in {0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0,14.0,16.0,17.0,19.0})
        If (feature 16 in {24.0,31.0})
          Predict: 0.0
        Else (feature 16 not in {24.0,31.0})
          Predict: 1.0
    Else (feature 9 not in {0.0,1.0,3.0})
      If (feature 16 in {0.0,1.0,2.0,4.0,5.0,6.0,8.0,9.0,10.0,12.0,13.0,14.0,16.0,17.0,19.0})
        Predict: 0.0
      Else (feature 16 not in {0.0,1.0,2.0,4.0,5.0,6.0,8.0,9.0,10.0,12.0,13.0,14.0,16.0,17.0,19.0})
        Predict: 1.0
```

```
%scala
```

```
// Use test set here so we can measure the accuracy of our model on new data
val cvPredictions = cvModel.transform(testData)
```

```
cvPredictions: org.apache.spark.sql.DataFrame = [label: double, fraud_reported: string ... 43 more fields]
```

```
%scala
```

```
// cvModel uses the best model found from the Cross Validation
```

```
// Evaluate best model
```

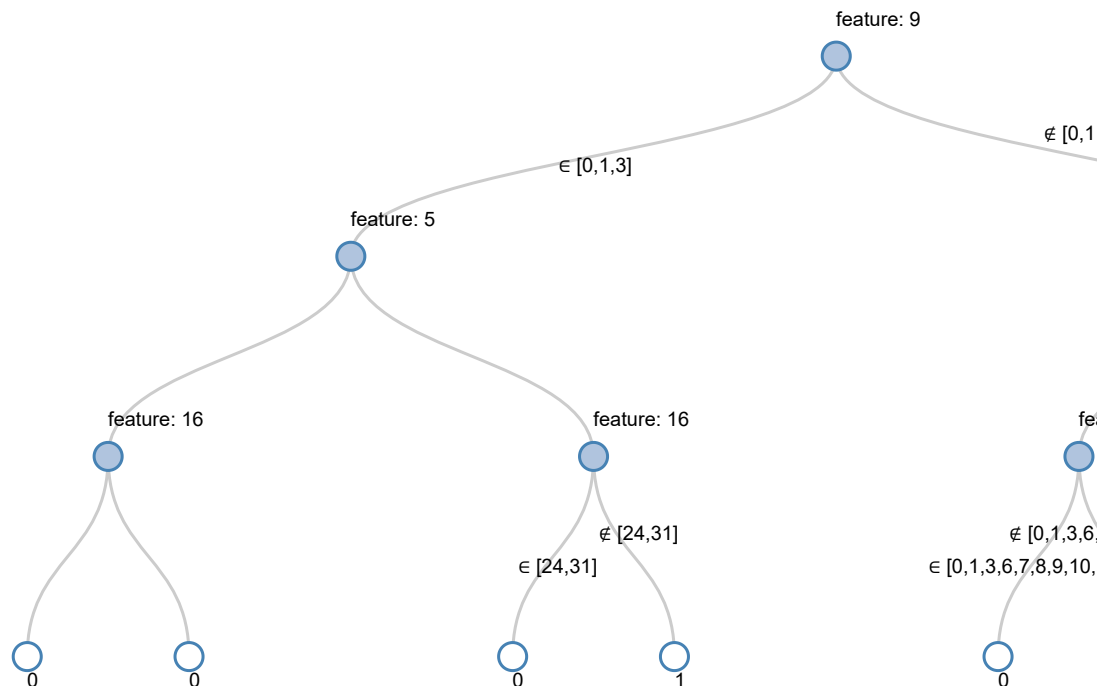
```
evaluator.evaluate(cvPredictions)
```

```
res8: Double = 0.8407816530223169
```

Using the same evaluator as before, we can see that Cross Validation improved our model's accuracy from 0.732 to 0.841!

```
%scala
```

```
display(bestTreeModel)
```




```
%scala
```

```
println(bestTreeModel.featureImportances)
```

```
(35,[4,5,9,16],[0.07686293230668599,0.36268113299007354,0.459384914182288,0.1010710205209526])
```

We know that feature 9 is our Decision Tree model's root node and that 5 is a close 2nd in importance. Let's see what they correspond to.

```
print assemblerInputs[9]  
print assemblerInputs[5]
```

```
incident_severityIndex  
insured_hobbiesIndex
```

Turns out that incident severity and insured hobbies are the best predictors for whether an insurance claim is fraudulent or not!

Zooming in on Prediction Data

We can further analyze the resulting prediction data. As an example, we can view an estimate of what our total predicted fraudulent claim amount is like, and zoom into a breakdown of predicted fraud count by incident severity and insured hobbies since those are our model's best predictors.

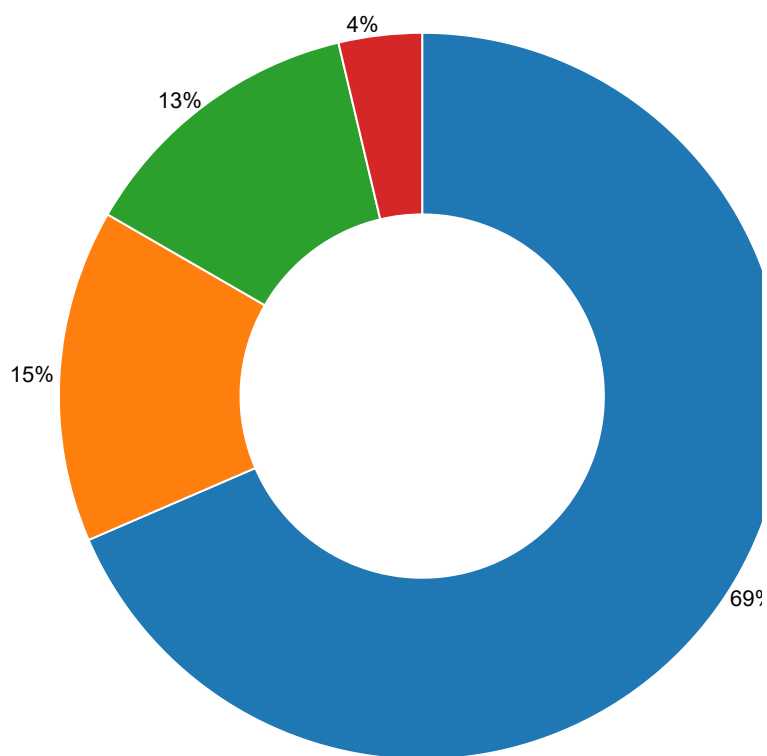
Let's hop back to Python for this.

```
%scala
```

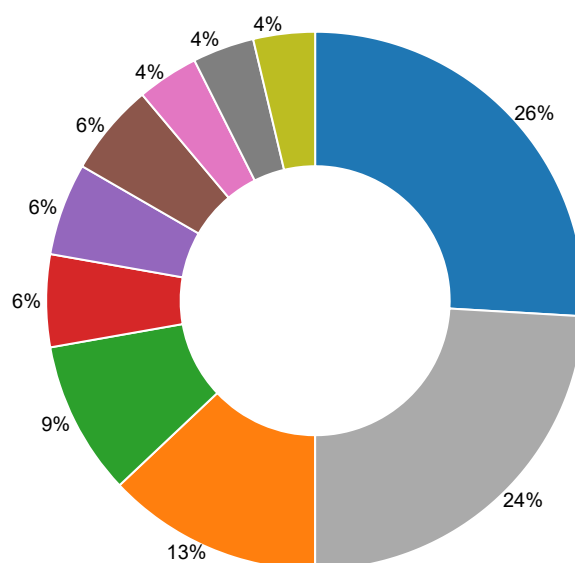
```
cvPredictions.createOrReplaceTempView("cvPredictions")
```

```
# Select columns to zoom into (In this example: Total Claim Amount and Auto
Make)
# Filter for data points that were predicted to be Fraud cases
cvPredictions = sqlContext.sql("SELECT * FROM cvPredictions")
incidentSeverityDF = cvPredictions.select("prediction", "total_claim_amount",
"incident_severity").filter("prediction = 1")
insuredHobbiesDF = cvPredictions.select("prediction", "total_claim_amount",
"insured_hobbies").filter("prediction = 1")

# View Count of Predicted Fraudulent Claims by Incident Severity
display(incidentSeverityDF)
```



```
# View Count of Predicted Fraudulent Claims by Insured Hobbies
display(insuredHobbiesDF)
```



Looks like people who are in major accidents and play chess or are into cross-fit are more prone to committing fraud.

APPENDIX

Gradient Boosting Trees and Random Forrest

Modeling and Evaluation

```
%scala
```

```
import org.apache.spark.ml.classification.{GBTCClassifier,  
GBTCClassificationModel}
```

```
// Create initial Decision Tree Model
```

```
val gbt = new GBTCClassifier()  
  .setLabelCol("label")  
  .setFeaturesCol("features")  
  .setMaxDepth(5)  
  .setMaxBins(40)
```

```
// Train model with Training Data
```

```
val gbtModel = gbt.fit(trainingData)
```

```
// Make predictions on test data using the transform() method.
```

```
// .transform() will only use the 'features' column as input.
```

```
val gbtPredictions = gbtModel.transform(testData)
```

```
import org.apache.spark.ml.classification.{GBTCClassifier, GBTCClassificationMod  
el}
```

```
gbt: org.apache.spark.ml.classification.GBTCClassifier = gbtc_49be1837104c
```

```
gbtModel: org.apache.spark.ml.classification.GBTCClassificationModel = GBTCClass  
ificationModel (uid=gbtc_49be1837104c) with 20 trees
```

```
gbtPredictions: org.apache.spark.sql.DataFrame = [label: double, fraud_reported:  
d: string ... 43 more fields]
```

```
%scala
```

```
import org.apache.spark.ml.classification.{RandomForestClassifier,  
RandomForestClassificationModel}
```

```
// Create initial Decision Tree Model
```

```
val rf = new RandomForestClassifier()  
  .setLabelCol("label")  
  .setFeaturesCol("features")  
  .setMaxDepth(5)  
  .setMaxBins(40)
```

```
// Train model with Training Data
```

```
val rfModel = rf.fit(trainingData)
```

```
// Make predictions on test data using the transform() method.
```

```
// .transform() will only use the 'features' column as input.
```

```
val rfPredictions = rfModel.transform(testData)
```

```
import org.apache.spark.ml.classification.{RandomForestClassifier, RandomForestClassificationModel}
rf: org.apache.spark.ml.classification.RandomForestClassifier = rfc_b2be08b75af7
rfModel: org.apache.spark.ml.classification.RandomForestClassificationModel =
RandomForestClassificationModel (uid=rfc_b2be08b75af7) with 20 trees
rfPredictions: org.apache.spark.sql.DataFrame = [label: double, fraud_reported: string ... 43 more fields]

GBT Eval - 0.8001009308063254
RF Eval - 0.8646405741841418
```