Loan Risk Analysis with Databricks and XGBoost

A Databricks guide, including code samples and notebooks.

databricks

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? How to streamline processes, speed up discovery, and scale implementations 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 eBook, we will walk you through a practical end-to-end Machine Learning use case 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.

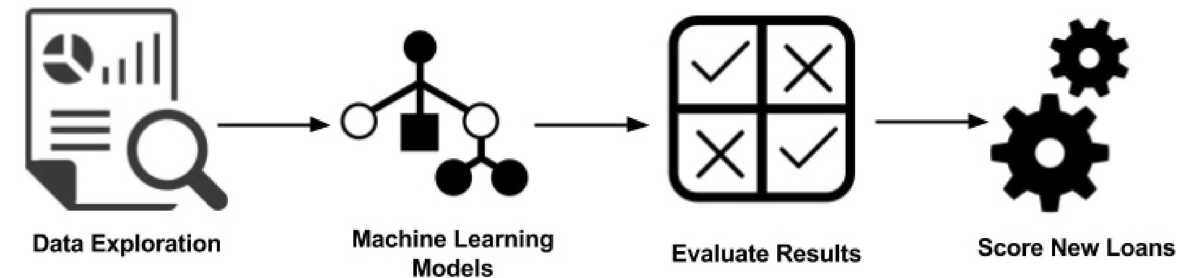
Loan Risk Analysis

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 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

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.

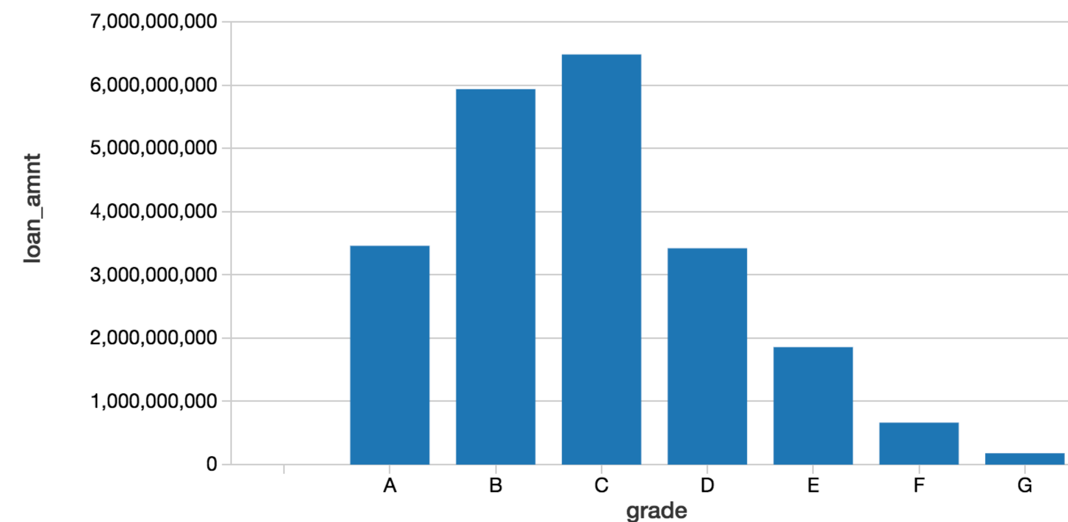
In this case, we have created the Databricks Database amy and table `loanstats_2012_2017`. The following code snippet allows you to access this table within a Databricks notebook via PySpark.

```
# 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.

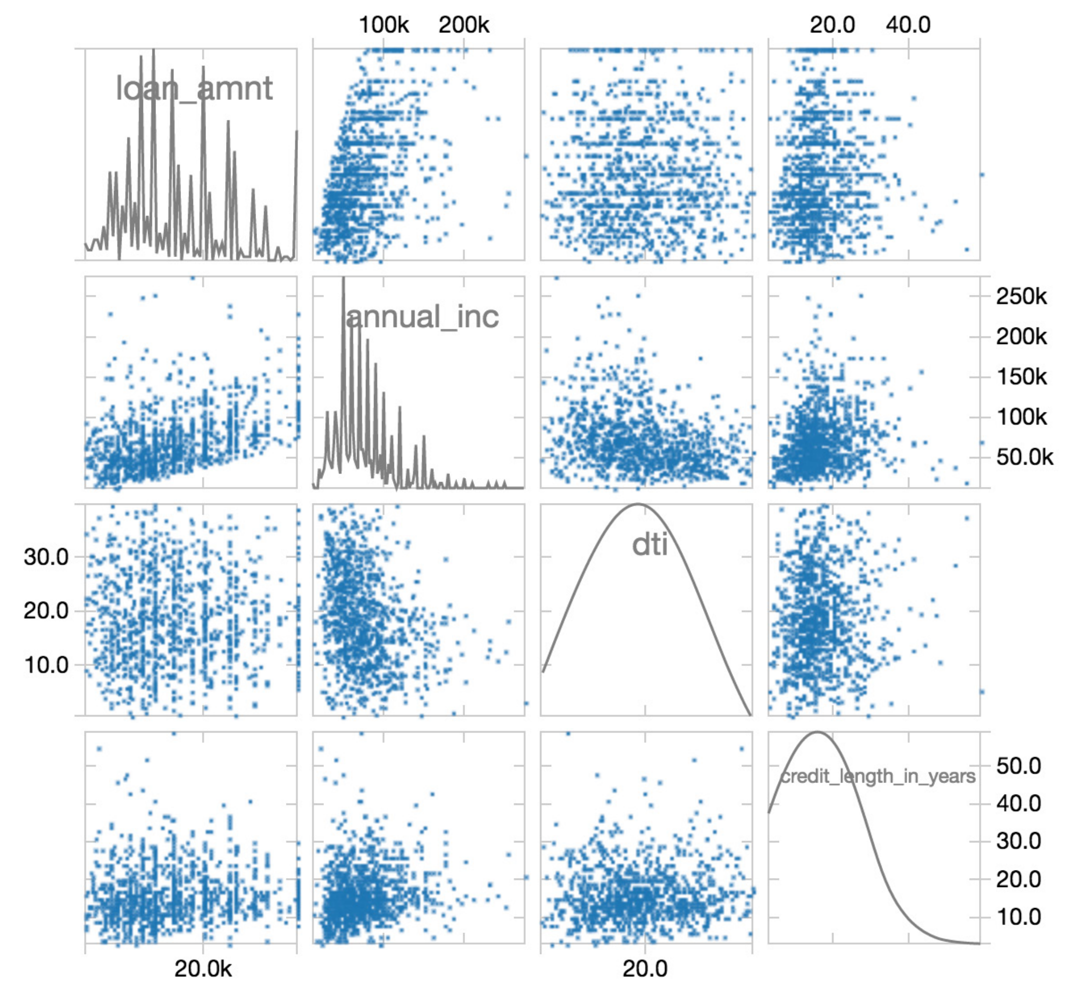
MUNGING YOUR DATA WITH THE PYSPARK DATAFRAME API

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"))
```

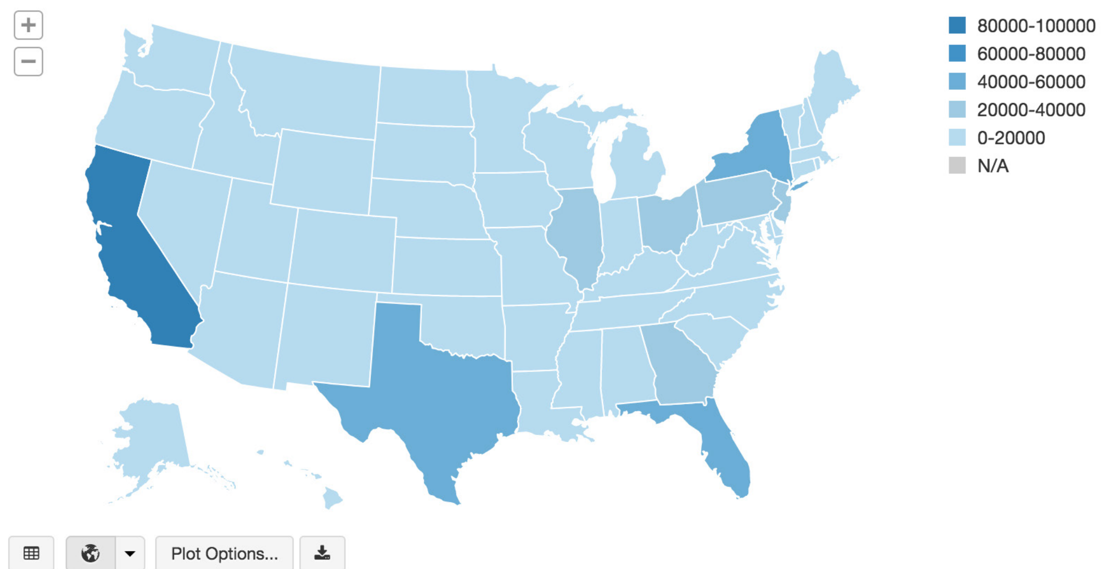
After this ETL process is completed, you can use the `display` command again to review the cleansed data in a scatter plot.

```
# 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 `categorical_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 out 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-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
```

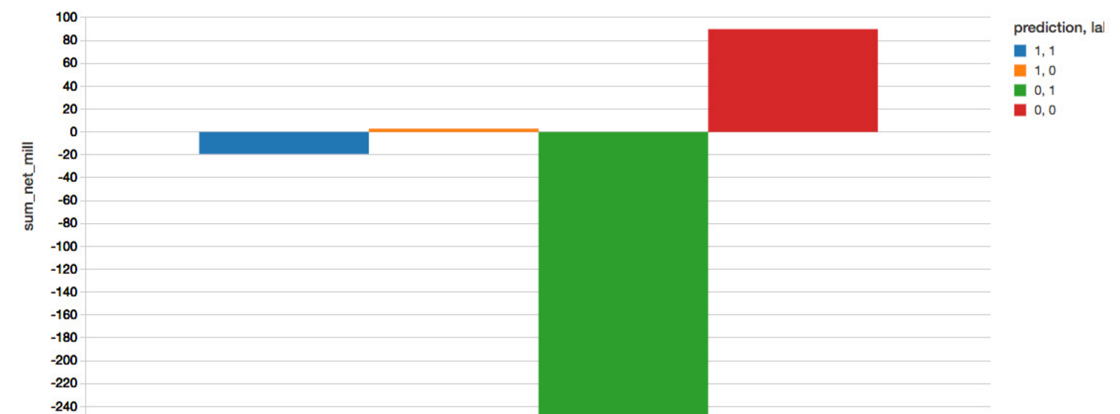
QUANTIFY THE BUSINESS VALUE

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

```
value = -(loss avoided - 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.

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

© Databricks 2018. All rights reserved. Apache, Apache Spark, Spark and the Spark logo are trademarks of the Apache Software Foundation. Privacy Policy | Terms of Use