Background:

Predicting income using cenus

Predicting income using census data - BINARY CLASSIFICATION

This project is to predict whether income exceeds $50K/yr based on census data. The attributes are of both continuous and nominal types.

We elected to use a logistic regression to solve the problem; predicting if an adults household income was less than or greater or equal to $50,000 which boils down to a binary classification problem. Furthermore we endeavoured to use Big Data tools and techniques to perform this admittedly simple regression.

```
from pyspark import SparkConf, SparkContext
from pyspark.sql import SparkSession, SQLContext
from pyspark.ml import Pipeline
from pyspark.ml.feature import OneHotEncoderEstimator, StringIndexer, VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator

spark = SparkSession.builder.master("local").getOrCreate()
conf = SparkConf().setMaster("local").setAppName("MonthlyIncome")
sc = SparkContext.getOrCreate()
sqlContext = SQLContext(sc)
```

Google Cloud Platform solution:

The Google Cloud Platform (GCP) is premised on writing batch jobs to run on a server cluster reading data from Google's Cloud storage. Job control is exercised through either a web-based front end, or a

locally installed shell program; gcloud. In our instance file transfers to GCP were accomplished through the web front end but job scheduling was accomplished with the gcloud shell. The command used to schedule our PySpark job is below.

gcloud dataproc jobs submit pyspark --cluster=math5671-project gs://dataproc-7301ae5f-7cee-4524-a329-20043ede8ede-us-central1/project-script.py

The command declares that we're submitting a dataproc job that must run with the PySpark library installed. Next we describe what cluster the job will run on, GCP clusters must have globally unique names, our cluster's name is math5671-project. The final item is a URL beginning gs://, this is a globally unique path to our python script. The command output follows:

C:\Program Files (x86)\Google\Cloud SDK>gcloud dataproc jobs submit pyspark --cluster=math5671-project gs://dataproc-7301ae5f-7cee-4524-a329-20043ede8ede-us-central1/project-script.py

Job [825c530cc60e4a5789960cf363036db8] submitted.

Waiting for job output...

19/05/06 04:26:24 INFO org.spark_project.jetty.util.log: Logging initialized @2653ms

19/05/06 04:26:25 INFO org.spark_project.jetty.server.Server: jetty-9.3.z-SNAPSHOT, build timestamp: unknown, git hash: unknown

19/05/06 04:26:25 INFO org.spark_project.jetty.server.Server: Started @2740ms

19/05/06 04:26:25 INFO org.spark_project.jetty.server.AbstractConnector: Started ServerConnector@325a486{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}

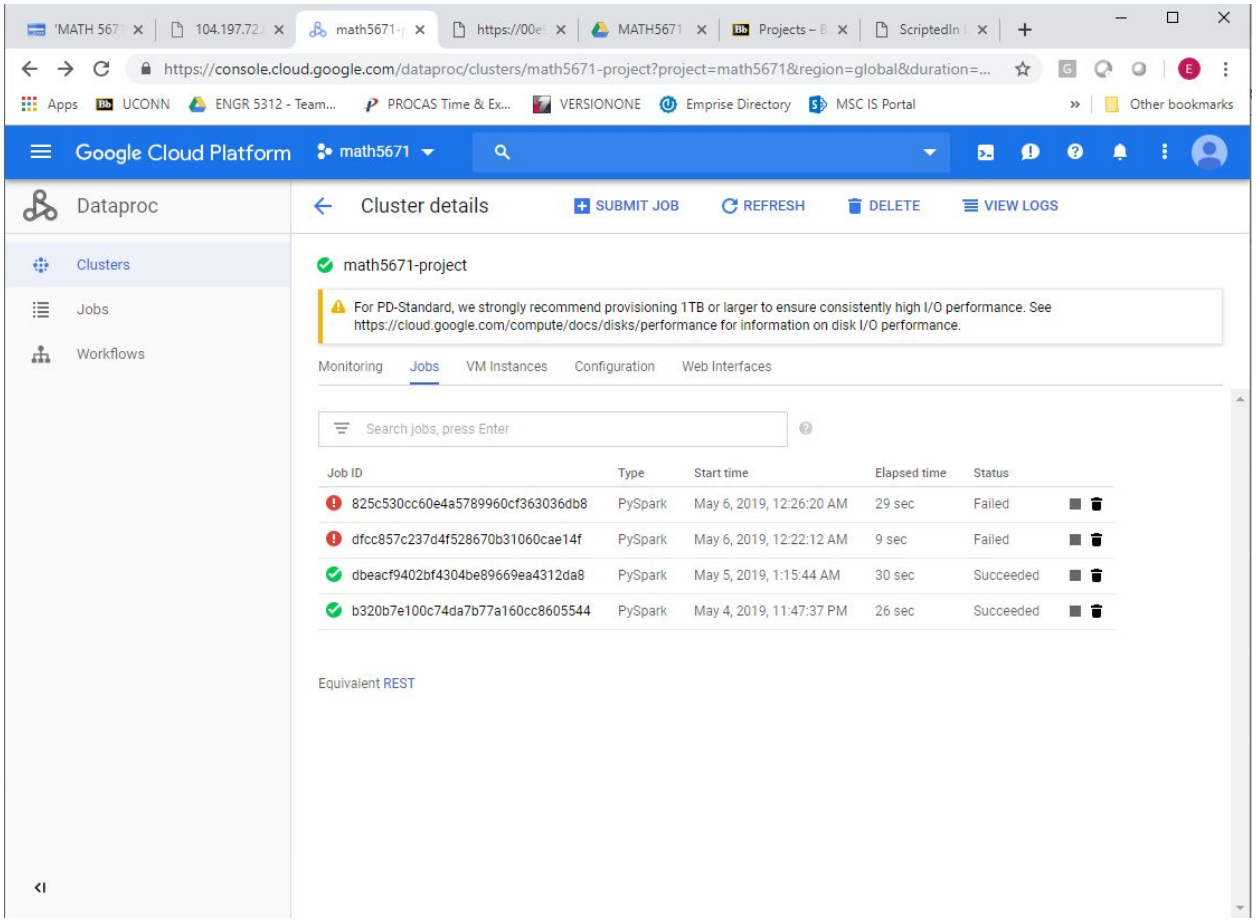
19/05/06 04:26:25 WARN org.apache.spark.scheduler.FairSchedulableBuilder: Fair Scheduler configuration file not found so jobs will be scheduled in FIFO order. To use fair scheduling, configure pools in fairscheduler.xml or set spark.scheduler.allocation.file to a file that contains the configuration.

19/05/06 04:26:46 INFO com.github.fommil.jni.JniLoader: successfully loaded /tmp/jniloader605696401646851243netlib-native_system-linux-x86_64.so

19/05/06 04:26:46 INFO breeze.optimize.OWLQN: Converged because gradient converged

Coefficients: (80,[],[])

Intercept: -1.0965730322897316



Preprocess the data and load them:

We performed some pre-processing on the provided training and test datasets locally, to lessen the number of data transformations which would need to be performed in code. Specifically, we eliminated extraneous quotation marks around text fields and replaced all '?' characters with null.

```
train_data = sqlContext.read.format("csv").option("header", "true").option("inferSchema", "true").load("training_data.csv")

test_data = sqlContext.read.format("csv").option("header", "true").option("inferSchema", "true").load("test_features.csv")


cols = train_data.columns
```

```
train_data = train_data.na.drop()
test_data = test_data.na.drop()
```

Convert categorical data into one-hot vector

Our data contained 7 categorical features stored as strings and 5 numerical features stored as doubles. The numeric features could be directly utilized by the Spark.ML LogisticRegression object, the categorical features required a transformation Pipeline. We used the StringIndexer object to create index columns for all our categorical features and the OneHotEncoderEstimator object to assign sparse vector values to the unordered categorical features. We will explain by way of example the effect of the StringIndexer and OneHotEncoder below,

Take a dataset:

Name	Direction
Glinda	North
Evanora	East
Theodora	South

The directions are vectors while names can be thought of as an enumeration, so the effect of StringIndexer and OneHotEncoder would be:

Direction	Direction Index	Glinda	Evanora	Theodora
North	0	1	0	0
East	1	0	0	1
South	2	0	1	0
West	3	0	0	0

This technique is commonly used to input categorical data into regressions.

In addition to the categorical features the training dataset also required an index to be created describing a label which indicated if the individual's household income was greater than or equal to $50,000.

In[5]:

```
categoricalColumns = ["workclass", "marital-status", "occupation", "relationship", "race", "sex",
"native-country"]

stages = []  stages in our Pipeline

for categoricalCol in categoricalColumns:

     Category Indexing with StringIndexer

    stringIndexer = StringIndexer(inputCol=categoricalCol, outputCol=categoricalCol + "Index",
handleInvalid="skip")

     Use OneHotEncoder to convert categorical variables into binary SparseVectors

    encoder = OneHotEncoderEstimator(inputCol=categoricalCol + "Index", outputCol=categoricalCol +
"classVec")

    encoder = OneHotEncoderEstimator(inputCols=[stringIndexer.getOutputCol()],
outputCols=[categoricalCol + "classVec"])

     Add stages.  These are not run here, but will run all at once later on.

    stages += [stringIndexer, encoder]
```

Transform features into a vector

Lastly we used the VectorAssembler object to combine the many numerical input features and the categorical features indexes into a single "features" vector.

```
 Transform all features into a vector using VectorAssembler

numericCols = ["age", "fnlwgt", "education-num", "hours-per-week"]

assemblerInputs = [c + "classVec" for c in categoricalColumns] + numericCols

assembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features")

stages += [assembler]
```

Create dataframe to use on the model

Create test_data DF before defining labels, which the test data dosent have

```
partialTestPipeline = Pipeline().setStages(stages)
pipelineModel = partialTestPipeline.fit(test_data)
preppedTestDataDF = pipelineModel.transform(test_data)
```

Convert label into label indices using the StringIndexer

```
label_stringIdx = StringIndexer(inputCol="income", outputCol="label")
stages += [label_stringIdx]
```

Pipeline stages are now defined, next create pipeline with these stages

```
partialTrainingPipeline = Pipeline().setStages(stages)
pipelineModel = partialTrainingPipeline.fit(train_data)
preppedTrainingDataDF = pipelineModel.transform(train_data)
```

LOGISTIC REGRESSION MODEL

We create a logistic model and estimate the parameters using the MLLib. Our script outputs the coefficients and intercept of the model but due to difficulties with the Spark and GCP technologies we could not print out the predicted labels of the test dataset.

In Spark the LogisticRegression object expects two, and only two, columns in its input dataset: a "label" column which is the correct answer used to train the regression and a "features" column which contains a vector of input features. With these two columns we created a model with Spark's Logistic Regression object. The we used that model to predict labels for our test dataset.

Our script outputs the coefficients and intercept of the model but due to difficulties with the Spark and GCP technologies we could not print out the predicted labels of the test dataset. A stack trace of the show() method error indicates it arose during intercommunication between the RDD data model and Spark ML library.

```
lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8, labelCol="label",
featuresCol="features")

lrModel = lr.fit(preppedTrainingDataDF.select(["label", "features"]))


prediction = lrModel.transform(preppedTestDataDF)


prediction = lrModel.transform(preppedTestDataDF)


 Print the coefficients and intercept for logistic regression

print("Coefficients: " + str(lrModel.coefficients))

print("Intercept: " + str(lrModel.intercept))
```

In total we spent about 26 dollar in computer time on the GCP. This included a minimal amount of storage but several operating days for our remote cluster. For this 26 dollar, we got a server cluster that could calculate our logistic regression in about 30 seconds. Interestingly, this was also the time the cluster took to complete even a trivial job, which indicates that the calculations we were asking of the cluster are themselves trivial.

Conclusion:

Our project serves as a proof-of-concept that the Spark machine learning library is accessible and scalable on Google's server clusters. Although our inability to get the remote cluster to run the show() method on our prediction dataframe shows the non-trivial complications that arise when operating on this exotic platform. Opaque errors were our nemesis during this project. While many Python errors have helpful messages the combination of using a local shell to execute scripts on remote machines running a Python wrapper to a Scala extension of Java APIs results in opaque error messages with unhelpful stack traces.

sus data - BINARY CLASSIFICATION

This project is to predict whether income exceeds $50K/yr based on census data. The attributes are of both continuous and nominal types.

We elected to use a logistic regression to solve the problem; predicting if an adults household income was less than or greater or equal to $50,000 which boils down to a binary classification problem. Furthermore we endeavoured to use Big Data tools and techniques to perform this admittedly simple regression.

```
from pyspark import SparkConf, SparkContext
from pyspark.sql import SparkSession, SQLContext
from pyspark.ml import Pipeline
from pyspark.ml.feature import OneHotEncoderEstimator, StringIndexer, VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator

spark = SparkSession.builder.master("local").getOrCreate()
conf = SparkConf().setMaster("local").setAppName("MonthlyIncome")
sc = SparkContext.getOrCreate()
sqlContext = SQLContext(sc)
```

Google Cloud Platform solution:

The Google Cloud Platform (GCP) is premised on writing batch jobs to run on a server cluster reading data from Google's Cloud storage. Job control is exercised through either a web-based front end, or a locally installed shell program; gcloud. In our instance file transfers to GCP were accomplished through the web front end but job scheduling was accomplished with the gcloud shell. The command used to schedule our PySpark job is below.

```
gcloud dataproc jobs submit pyspark --cluster=math5671-project
gs://dataproc-7301ae5f-7cee-4524-a329-20043ede8ede-us-central1/project-script.py
```

The command declares that we're submitting a dataproc job that must run with the PySpark library installed. Next we describe what cluster the job will run on, GCP clusters must have globally unique

names, our cluster's name is math5671-project. The final item is a URL beginning gs://, this is a globally unique path to our python script. The command output follows:

C:\Program Files (x86)\Google\Cloud SDK>gcloud dataproc jobs submit pyspark --cluster=math5671-project gs://dataproc-7301ae5f-7cee-4524-a329-20043ede8ede-us-central1/project-script.py

Job [825c530cc60e4a5789960cf363036db8] submitted.

Waiting for job output...

19/05/06 04:26:24 INFO org.spark_project.jetty.util.log: Logging initialized @2653ms

19/05/06 04:26:25 INFO org.spark_project.jetty.server.Server: jetty-9.3.z-SNAPSHOT, build timestamp: unknown, git hash: unknown

19/05/06 04:26:25 INFO org.spark_project.jetty.server.Server: Started @2740ms

19/05/06 04:26:25 INFO org.spark_project.jetty.server.AbstractConnector: Started ServerConnector@325a486{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}

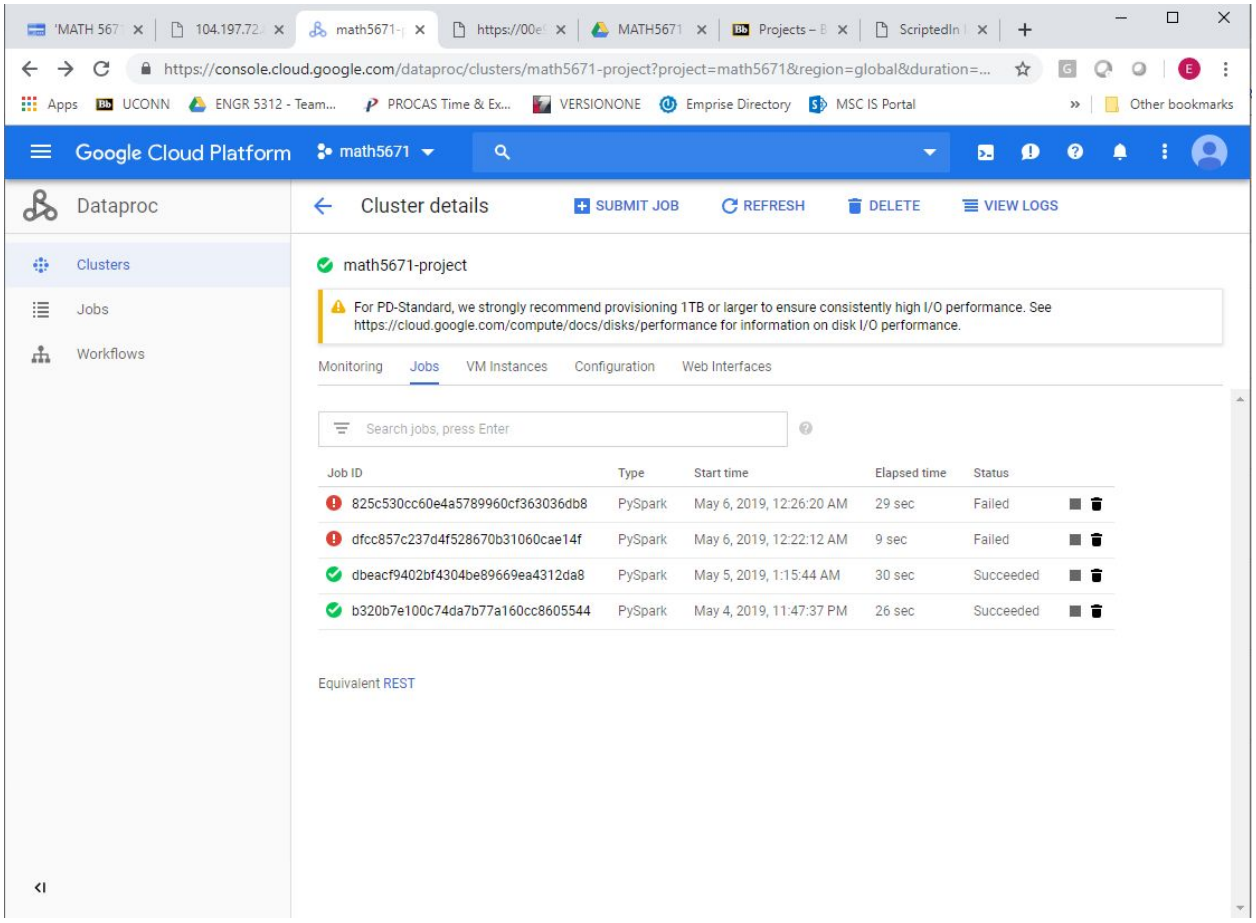
19/05/06 04:26:25 WARN org.apache.spark.scheduler.FairSchedulableBuilder: Fair Scheduler configuration file not found so jobs will be scheduled in FIFO order. To use fair scheduling, configure pools in fairscheduler.xml or set spark.scheduler.allocation.file to a file that contains the configuration.

19/05/06 04:26:46 INFO com.github.fommil.jni.JniLoader: successfully loaded /tmp/jniloader605696401646851243netlib-native_system-linux-x86_64.so

19/05/06 04:26:46 INFO breeze.optimize.OWLQN: Converged because gradient converged

Coefficients: (80,[],[])

Intercept: -1.0965730322897316



Preprocess the data and load them:

We performed some pre-processing on the provided training and test datasets locally, to lessen the number of data transformations which would need to be performed in code. Specifically, we eliminated extraneous quotation marks around text fields and replaced all '?' characters with null.

```
train_data = sqlContext.read.format("csv").option("header", "true").option("inferSchema", "true").load("training_data.csv")

test_data = sqlContext.read.format("csv").option("header", "true").option("inferSchema", "true").load("test_features.csv")


cols = train_data.columns
```

```
train_data = train_data.na.drop()
test_data = test_data.na.drop()
```

Convert categorical data into one-hot vector

Our data contained 7 categorical features stored as strings and 5 numerical features stored as doubles. The numeric features could be directly utilized by the Spark.ML LogisticRegression object, the categorical features required a transformation Pipeline. We used the StringIndexer object to create index columns for all our categorical features and the OneHotEncoderEstimator object to assign sparse vector values to the unordered categorical features. We will explain by way of example the effect of the StringIndexer and OneHotEncoder below,

Take a dataset:

Name	Direction
Glinda	North
Evanora	East
Theodora	South

The directions are vectors while names can be thought of as an enumeration, so the effect of StringIndexer and OneHotEncoder would be:

Direction	Direction Index	Glinda	Evanora	Theodora
North	0	1	0	0
East	1	0	0	1
South	2	0	1	0
West	3	0	0	0

This technique is commonly used to input categorical data into regressions.

In addition to the categorical features the training dataset also required an index to be created describing a label which indicated if the individual's household income was greater than or equal to $50,000.

In[5]:

```
categoricalColumns = ["workclass", "marital-status", "occupation", "relationship", "race", "sex",
"native-country"]

stages = []  stages in our Pipeline

for categoricalCol in categoricalColumns:

     Category Indexing with StringIndexer

    stringIndexer = StringIndexer(inputCol=categoricalCol, outputCol=categoricalCol + "Index",
handleInvalid="skip")

     Use OneHotEncoder to convert categorical variables into binary SparseVectors

    encoder = OneHotEncoderEstimator(inputCol=categoricalCol + "Index", outputCol=categoricalCol +
"classVec")

    encoder = OneHotEncoderEstimator(inputCols=[stringIndexer.getOutputCol()],
outputCols=[categoricalCol + "classVec"])

     Add stages.  These are not run here, but will run all at once later on.

    stages += [stringIndexer, encoder]
```

Transform features into a vector

Lastly we used the VectorAssembler object to combine the many numerical input features and the categorical features indexes into a single "features" vector.

```
 Transform all features into a vector using VectorAssembler

numericCols = ["age", "fnlwgt", "education-num", "hours-per-week"]

assemblerInputs = [c + "classVec" for c in categoricalColumns] + numericCols

assembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features")

stages += [assembler]
```

Create dataframe to use on the model

Create test_data DF before defining labels, which the test data dosent have

```
partialTestPipeline = Pipeline().setStages(stages)
pipelineModel = partialTestPipeline.fit(test_data)
preppedTestDataDF = pipelineModel.transform(test_data)
```

Convert label into label indices using the StringIndexer

```
label_stringIdx = StringIndexer(inputCol="income", outputCol="label")
stages += [label_stringIdx]
```

Pipeline stages are now defined, next create pipeline with these stages

```
partialTrainingPipeline = Pipeline().setStages(stages)
pipelineModel = partialTrainingPipeline.fit(train_data)
preppedTrainingDataDF = pipelineModel.transform(train_data)
```

LOGISTIC REGRESSION MODEL

We create a logistic model and estimate the parameters using the MLLib. Our script outputs the coefficients and intercept of the model but due to difficulties with the Spark and GCP technologies we could not print out the predicted labels of the test dataset.

In Spark the LogisticRegression object expects two, and only two, columns in its input dataset: a "label" column which is the correct answer used to train the regression and a "features" column which contains a vector of input features. With these two columns we created a model with Spark's Logistic Regression object. The we used that model to predict labels for our test dataset.

Our script outputs the coefficients and intercept of the model but due to difficulties with the Spark and GCP technologies we could not print out the predicted labels of the test dataset. A stack trace of the show() method error indicates it arose during intercommunication between the RDD data model and Spark ML library.

```
lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8, labelCol="label",
featuresCol="features")

lrModel = lr.fit(preppedTrainingDataDF.select(["label", "features"]))


prediction = lrModel.transform(preppedTestDataDF)


prediction = lrModel.transform(preppedTestDataDF)


 Print the coefficients and intercept for logistic regression

print("Coefficients: " + str(lrModel.coefficients))

print("Intercept: " + str(lrModel.intercept))
```

In total we spent about 26 dollar in computer time on the GCP. This included a minimal amount of storage but several operating days for our remote cluster. For this 26 dollar, we got a server cluster that could calculate our logistic regression in about 30 seconds. Interestingly, this was also the time the cluster took to complete even a trivial job, which indicates that the calculations we were asking of the cluster are themselves trivial.

Conclusion:

Our project serves as a proof-of-concept that the Spark machine learning library is accessible and scalable on Google's server clusters. Although our inability to get the remote cluster to run the show() method on our prediction dataframe shows the non-trivial complications that arise when operating on this exotic platform. Opaque errors were our nemesis during this project. While many Python errors have helpful messages the combination of using a local shell to execute scripts on remote machines running a Python wrapper to a Scala extension of Java APIs results in opaque error messages with unhelpful stack traces.