databricks Jessie Ma - Assignment 4 - Question 1

(https://databricks.com)

*** SPARK SQL ***

The Data

The first of the two datasets that we will be working with is the **Farmers Markets Directory and Geographic Data**. This dataset contains information on the longitude and latitude, state, address, name, and zip code of farmers markets in the United States. The raw data is published by the Department of Agriculture. The version on the data that is found in Databricks (and is used in this tutorial) was updated by the Department of Agriculture on Dec 01, 2015.

The second dataset we will be working with is the **SOI Tax Stats - Individual Income Tax Statistics - ZIP Code Data (SOI)**. This study provides detailed tabulations of individual income tax return data at the state and ZIP code level and is provided by the IRS. This repository only has a sample of the data: 2013 and includes "AGI". The ZIP Code data shows selected income and tax items classified by State, ZIP Code, and size of adjusted gross income. Data is based on individual income tax returns filed with the IRS and is available for Tax Years 1998, 2001, 2004 through 2013.

```
# Read The data
taxes2013 = (spark.read
  .option("header", "true")
  .csv("dbfs:/databricks-datasets/data.gov/irs_zip_code_data/data-001/2013_soi_zipcode_agi.csv"))

markets = (spark.read
  .option("header", "true")
  .csv("dbfs:/databricks-datasets/data.gov/farmers_markets_geographic_data/data-001/market_data.csv"))


# Register spark SQL tables

taxes2013.createOrReplaceTempView("taxes2013")
markets.createOrReplaceTempView("markets")


%sql
DROP TABLE IF EXISTS cleaned_taxes;

CREATE OR REPLACE TABLE cleaned_taxes AS
SELECT state, int(zipcode / 10) as zipcode,
  int(mars1) as single_returns,
  int(mars2) as joint_returns,
  int(numdep) as numdep,
  double(A02650) as total_income_amount,
  double(A00300) as taxable_interest_amount,
  double(a01000) as net_capital_gains,
  double(a00900) as biz_net_income
FROM taxes2013
```

Query returned no results

```
sqlContext.cacheTable("cleaned_taxes")

# Convert back to a dataset from a table
cleanedTaxes = spark.sql("SELECT * FROM cleaned_taxes")

summedTaxes = cleanedTaxes.groupBy("zipcode").sum() # because of AGI, where groups income groups are broken out

cleanedMarkets = (markets
  .selectExpr("*", "int(zip / 10) as zipcode")
  .groupBy("zipcode")
  .count()
  .selectExpr("double(count) as count", "zipcode as zip"))
#  selectExpr is short for Select Expression - equivalent to what we
#  might be doing in SQL SELECT expression

joined = (cleanedMarkets.join(summedTaxes, cleanedMarkets.zip == summedTaxes.zipcode, "outer"))


display(cleanedMarkets)
```

Table

	count	zip
1	5	4900
2	2	7240
3	8	4818
4	1	9852
5	2	5300
6	5	2122
7	2	9900

3,589 rows

```
display(joined)
```

Table

	count	zip	zipcode	sum(zipcode)	sum(single_returns)	sum(joint_returns)	sum(numdep)	sum(to
1	1009	null	null	null	null	null	null	null
2	1	0	0	0	66430180	52885400	96500590	927412
3	1	3	null	null	null	null	null	null
4	4	60	null	null	null	null	null	null
5	1	61	null	null	null	null	null	null
6	2	62	null	null	null	null	null	null
7	1	63	null	null	null	null	null	null

5,802 rows

deal with na values

```
prepped = joined.na.fill(0)
display(prepped)
```

Table

	count	zip	zipcode	sum(zipcode)	sum(single_returns)	sum(joint_returns)	sum(numdep)	sum(to
1	1009	0	0	0	0	0	0	0
2	1	0	0	0	66430180	52885400	96500590	927412
3	1	3	0	0	0	0	0	0
4	4	60	0	0	0	0	0	0
5	1	61	0	0	0	0	0	0
6	2	62	0	0	0	0	0	0

7	1	63	0	0	0	0	0	0

5,802 rows

```
display(prepped)
```

Table

	count	zip	zipcode	sum(zipcode)	sum(single_returns)	sum(joint_returns)	sum(numdep)	sum(to
1	1009	0	0	0	0	0	0	0
2	1	0	0	0	66430180	52885400	96500590	927412
3	1	3	0	0	0	0	0	0
4	4	60	0	0	0	0	0	0
5	1	61	0	0	0	0	0	0
6	2	62	0	0	0	0	0	0
7	1	63	0	0	0	0	0	0

5,802 rows

Now that all of our data is prepped. We're going to have to put all of it into one column of a vector type for Spark MLLib. This makes it easy to embed a prediction right in a DataFrame and also makes it very clear as to what is getting passed into the model and what isn't without having to convert it to a numpy array or specify an R formula. This also makes it easy to incrementally add new features, simply by adding to the vector. In the below case rather than specifically adding them in.

```
nonFeatureCols = ["zip", "zipcode", "count"]
featureCols = [item for item in prepped.columns if item not in nonFeatureCols]


# VectorAssembler Assembles all of these columns into one single vector. To do this, set the input columns and output column. Then tha
data to the final dataset.
from pyspark.ml.feature import VectorAssembler

assembler = (VectorAssembler()
  .setInputCols(featureCols)
  .setOutputCol("features"))

finalPrep = assembler.transform(prepped)
```

Now split the dataset 70-30 for training and testing purposes.A validation set can be created as well, we are omitting it here. It's worth noting that MLLib also supports performing hyperparameter tuning with cross validation and pipelines. All this can be found in the Databrick's Guide (https://docs.databricks.com).

```
training, test = finalPrep.randomSplit([0.7, 0.3])

#  Going to cache the data to make sure things stay snappy!
training.cache()
test.cache()

print(training.count()) # Why execute count here??
print(test.count())
```

```
4166
1636
```

Apache Spark MLLib

At a high level, we're going to create an instance of a `regressor` or `classifier`, that in turn will then be trained and return a `Model` type. Whenever you access Spark MLLib you should be sure to import/train on the name of the algorithm you want as opposed to the `Model` type. For example:

You should import:

`org.apache.spark.ml.regression.LinearRegression`

as opposed to:

`org.apache.spark.ml.regression.LinearRegressionModel`

In the below example, we're going to use linear regression.

The linear regression that is available in Spark MLLib supports an elastic net parameter allowing you to set a threshold of how much you would like to mix l1 and l2 regularization, for more information on Elastic net regularization see Wikipedia (https://en.wikipedia.org/wiki/Elastic_net_regularization).

```
from pyspark.ml.regression import LinearRegression

lrModel = (LinearRegression()
  .setLabelCol("count")
  .setFeaturesCol("features")
  .setElasticNetParam(0.5))

print("Printing out the model Parameters:")
print("-"*20)
print(lrModel.explainParams())
print("-"*20)
```

```
Printing out the model Parameters:
--------------------
aggregationDepth: suggested depth for treeAggregate (>= 2). (default: 2)
elasticNetParam: the ElasticNet mixing parameter, in range [0, 1]. For alpha = 0, the penalty is an L2 penalty. For alpha = 1, i
t is an L1 penalty. (default: 0.0, current: 0.5)
epsilon: The shape parameter to control the amount of robustness. Must be > 1.0. Only valid when loss is huber (default: 1.35)
featuresCol: features column name. (default: features, current: features)
fitIntercept: whether to fit an intercept term. (default: True)
labelCol: label column name. (default: label, current: count)
loss: The loss function to be optimized. Supported options: squaredError, huber. (default: squaredError)
maxBlockSizeInMB: maximum memory in MB for stacking input data into blocks. Data is stacked within partitions. If more than rema
ining data size in a partition then it is adjusted to the data size. Default 0.0 represents choosing optimal value, depends on s
pecific algorithm. Must be >= 0. (default: 0.0)
maxIter: max number of iterations (>= 0). (default: 100)
predictionCol: prediction column name. (default: prediction)
regParam: regularization parameter (>= 0). (default: 0.0)
solver: The solver algorithm for optimization. Supported options: auto, normal, l-bfgs. (default: auto)
standardization: whether to standardize the training features before fitting the model. (default: True)
tol: the convergence tolerance for iterative algorithms (>= 0). (default: 1e-06)
weightCol: weight column name. If this is not set or empty, we treat all instance weights as 1.0. (undefined)
--------------------
```

Now finally we can go about fitting our model! You'll see that we're going to do this in a series of steps. First we'll fit it, then we'll use it to make predictions via the `transform` method. This is the same way you would make predictions with your model in the future however in this case we're using it to evaluate how our model is doing. We'll be using regression metrics to get some idea of how our model is performing, we'll then print out those values to be able to evaluate how it performs.

```
from pyspark.mllib.evaluation import RegressionMetrics
lrFitted = lrModel.fit(training)
```

```
%fs ls /databricks-datasets/songs/data-001/
```

Table

	path	name	size	modificationTime
1	dbfs:/databricks-datasets/songs/data-001/header.txt	header.txt	377	1454633901000
2	dbfs:/databricks-datasets/songs/data-001/part-00000	part-00000	52837	1454547464000
3	dbfs:/databricks-datasets/songs/data-001/part-00001	part-00001	52469	1454547465000
4	dbfs:/databricks-datasets/songs/data-001/part-00002	part-00002	51778	1454547465000
5	dbfs:/databricks-datasets/songs/data-001/part-00003	part-00003	50551	1454547465000
6	dbfs:/databricks-datasets/songs/data-001/part-00004	part-00004	53449	1454547465000
7	dbfs:/databricks-datasets/songs/data-001/part-00005	part-00005	53301	1454547465000

113 rows

Now you'll see that since we're working with exact numbers (you can't have 1/2 a farmer's market for example), I'm going to check equality by first rounding the value to the nearest digital value.

```
holdout = (lrFitted
  .transform(test)
  .selectExpr("prediction as raw_prediction",
    "double(round(prediction)) as prediction",
    "count",
    """CASE double(round(prediction)) = count
  WHEN true then 1
  ELSE 0
END as equal"""))

display(holdout)
```

Table

	raw_prediction	prediction	count	equal
1	1.3764440047805635	1	0	0
2	1.6416592612751493	2	0	0
3	1.5166831169264767	2	0	0
4	1.4075361079884272	1	0	0
5	1.4439636672805132	1	1	1
6	1.376703484500583	1	1	1
7	1.4446646702744959	1	1	1

1,636 rows

Now let's see what proportion was exactly correct.

```
display(holdout.selectExpr("sum(equal)/sum(1)"))
```

Table

	(sum(equal) / sum(1))
1	0.24144254278728605

1 row

```
  # have to do a type conversion for RegressionMetrics
  rm = RegressionMetrics(holdout.select("prediction", "count").rdd.map(lambda x:  (x[0], x[1])))

  print("MSE: ", rm.meanSquaredError)
  print("MAE: ", rm.meanAbsoluteError)
  print("RMSE Squared: ", rm.rootMeanSquaredError)
  print("R Squared: ", rm.r2)
  print("Explained Variance: ", rm.explainedVariance, "\n")
```

```
MSE:  2.5916870415647923
MAE:  1.2041564792176038
RMSE Squared:  1.609871746930417
R Squared:  0.007419484406576582
Explained Variance:  0.5413432338400648
```

Jessie Ma - Assignment 4 - Question 1 Starts Here

These results appear to be sub-optimal, so let's try exploring another way to train the model. Rather than training on a single model with hard-coded parameters, let's train using a pipeline (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.ml.Pipeline).

A pipeline is going to give us some nice benefits in that it will allow us to use a couple of transformations we need in order to transform our raw data into the prepared data for the model but also it provides a simple, straightforward way to try out a lot of different combinations of parameters. This is a process called hyperparameter tuning (https://en.wikipedia.org/wiki/Hyperparameter_optimization) or grid search. To review, grid search is where you set up the exact parameters that you would like to test and MLLib will automatically create all the necessary combinations of these to test.

For example, below we'll set `numTrees` to 20 and 60 and `maxDepth` to 5 and 10. The parameter grid builder will automatically construct all the combinations of these two variable (along with the other ones that we might specify too). Additionally we're also going to use cross validation (https://en.wikipedia.org/wiki/Cross-validation_(statistics)) to tune our hyperparameters, this will allow us to attempt to try to control overfitting (https://en.wikipedia.org/wiki/Overfitting) of our model.

Lastly we'll need to set up a Regression Evaluator (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.ml.evaluation.RegressionEvaluator) that will evaluate the models that we choose based on some metric (the default is RMSE). The key take away is that the pipeline will automatically optimize for our given metric choice by exploring the parameter grid that we set up rather than us having to do it manually like we would have had to do above.

Now we can go about training our random forest!

note: this might take a little while because of the number of combinations that we're trying and limitations in workers available.

Jessie: I tuned parameters directly in the original code block provided below. I lowered values for maxDepth and numTrees because the original block ran for more than 1 hour, timing out the cluster. You'll notice reducing the parameter values did reduce the computational load of the training phase but also reduced the proportion of "exactly" correct (from 0.38 to 0.29).

```
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
from pyspark.ml.evaluation import RegressionEvaluator

from pyspark.ml import Pipeline

rfModel = (RandomForestRegressor()
  .setLabelCol("count")
  .setFeaturesCol("features"))

paramGrid = (ParamGridBuilder()
  .addGrid(rfModel.maxDepth, [2, 6])
  .addGrid(rfModel.numTrees, [5, 10])
  #.addGrid(rfModel.maxBins, [32, 64])
  #.addGrid(rfModel.minInstancesPerNode, [1, 5])
  .build())
# Note, that this parameter grid will take a long time
# to run in the community edition due to limited number
# of workers available! Be patient for it to run!
# If you want it to run faster, remove some of
# the above parameters and it'll speed right up!

stages = [rfModel]

pipeline = Pipeline().setStages(stages)

cv = (CrossValidator() # you can feel free to change the number of folds used in cross validation as well
  .setEstimator(pipeline) # the estimator can also just be an individual model rather than a pipeline
  .setEstimatorParamMaps(paramGrid)
  .setEvaluator(RegressionEvaluator().setLabelCol("count")))

pipelineFitted = cv.fit(training)
```

Now we've trained our model! Let's take a look at which version performed best!

```
print("The Best Parameters:\n--------------------")
print(pipelineFitted.bestModel.stages[0])
pipelineFitted.bestModel.stages[0].extractParamMap()
```

```
The Best Parameters:
--------------------
RandomForestRegressionModel: uid=RandomForestRegressor_5c3328071865, numTrees=5, numFeatures=8
```

```
{Param(parent='RandomForestRegressor_5c3328071865', name='bootstrap', doc='Whether bootstrap samples are used when building trees.'): True,
 Param(parent='RandomForestRegressor_5c3328071865', name='cacheNodeIds', doc='If false, the algorithm will pass trees to executors to match instances with nodes. If true, the algorithm will cache node IDs for each instance. Caching can speed up training of deeper trees. Users can set how often should the cache be checkpointed or disable it by setting checkpointInterval.'): False,
 Param(parent='RandomForestRegressor_5c3328071865', name='checkpointInterval', doc='set checkpoint interval (>= 1) or disable checkpoint (-1). E.g. 10 means that the cache will get checkpointed every 10 iterations. Note: this setting will be ignored if the checkpoint directory is not set in the SparkContext.'): 10,
 Param(parent='RandomForestRegressor_5c3328071865', name='featureSubsetStrategy', doc="The number of features to consider for splits at each tree node. Supported options: 'auto' (choose automatically for task: If numTrees == 1, set to 'all'. If numTrees > 1 (forest), set to 'sqrt' for classification and to 'onethird' for regression), 'all' (use all features), 'onethird' (use 1/3 of the features), 'sqrt' (use sqrt(number of features)), 'log2' (use log2(number of features)), 'n' (when n is in the range (0, 1.0], use n * number of features. When n is in the range (1, number of features), use n features). default = 'auto'"): 'auto',
 Param(parent='RandomForestRegressor_5c3328071865', name='featuresCol', doc='features column name.'): 'features',
 Param(parent='RandomForestRegressor_5c3328071865', name='impurity', doc='Criterion used for information gain calculation (case-insensitive). Supported options: variance'): 'variance',
 Param(parent='RandomForestRegressor_5c3328071865', name='labelCol', doc='label column name.'): 'count',
 Param(parent='RandomForestRegressor_5c3328071865', name='leafCol', doc='Leaf indices column name. Predicted leaf index of each instance in each tree by preorder.'): '',
 Param(parent='RandomForestRegressor_5c3328071865', name='maxBins', doc='Max number of bins for discretizing continuous feature
```

As well as our regression metrics on the test set.

```
%fs
```

dbutils.fs provides utilities for working with FileSystems. Most methods in this package can take either a DBFS path (e.g., "/foo" or "dbfs:/foo"), or another FileSystem URI. For more info about a method, use **dbutils.fs.help("methodName")**. In notebooks, you can also use the %fs shorthand to access DBFS. The %fs shorthand maps straightforwardly onto dbutils calls. For example, "%fs head --maxBytes=10000 /file/path" translates into "dbutils.fs.head("/file/path", maxBytes = 10000)".

mount

mount(source: String, mountPoint: String, encryptionType: String = "", owner: String = null, extraConfigs: Map = Map.empty[String, String]): boolean -> Mounts the given source directory into DBFS at the given mount point
mounts: Seq -> Displays information about what is mounted within DBFS
refreshMounts: boolean -> Forces all machines in this cluster to refresh their mount cache, ensuring they receive the most recent information
unmount(mountPoint: String): boolean -> Deletes a DBFS mount point
updateMount(source: String, mountPoint: String, encryptionType: String = "", owner: String = null, extraConfigs: Map = Map.empty[String, String]): boolean -> Similar to mount(), but updates an existing mount point (if present) instead of creating a new one

fsutils

cp(from: String, to: String, recurse: boolean = false): boolean -> Copies a file or directory, possibly across FileSystems
head(file: String, maxBytes: int = 65536): String -> Returns up to the first 'maxBytes' bytes of the given file as a String encoded in UTF-8
ls(dir: String): Seq -> Lists the contents of a directory
mkdirs(dir: String): boolean -> Creates the given directory if it does not exist, also creating any necessary parent directories
mv(from: String, to: String, recurse: boolean = false): boolean -> Moves a file or directory, possibly across FileSystems
put(file: String, contents: String, overwrite: boolean = false): boolean -> Writes the given String out to a file, encoded in UTF-8
rm(dir: String, recurse: boolean = false): boolean -> Removes a file or directory

```
pipelineFitted.bestModel
```

```
PipelineModel_5a949c27af55
```

```
holdout2 = (pipelineFitted.bestModel
  .transform(test)
  .selectExpr("prediction as raw_prediction",
    "double(round(prediction)) as prediction",
    "count",
    """CASE double(round(prediction)) = count
  WHEN true then 1
  ELSE 0
END as equal"""))

display(holdout2)
```

Table

	raw_prediction	prediction	count	equal
1	2.387931550367706	2	0	0
2	0.867301012897878	1	0	0
3	0.867301012897878	1	0	0
4	0.867301012897878	1	0	0
5	0.867301012897878	1	1	1
6	0.867301012897878	1	1	1
7	0.867301012897878	1	1	1

1,636 rows

```
rm2 = RegressionMetrics(holdout2.select("prediction", "count").rdd.map(lambda x:  (x[0], x[1])))

print("MSE: ", rm2.meanSquaredError)
print("MAE: ", rm2.meanAbsoluteError)
print("RMSE Squared: ", rm2.rootMeanSquaredError)
print("R Squared: ", rm2.r2)
print("Explained Variance: ", rm2.explainedVariance, "\n")
```

```
MSE:  12.636919315403421
MAE:  1.4315403422982884
RMSE Squared:  3.554844485403464
R Squared:  -3.8397664102307623
Explained Variance:  10.241766473777654
```

Finally we'll see an improvement in our "exactly right" proportion as well!

```
display(holdout2.selectExpr("sum(equal)/sum(1)"))
```

Table

	(sum(equal) / sum(1))
1	0.29034229828850855

1 row

Jessie: I was playing around and tuning the parameters further, even adding a new parameter 'minInstancesperNode' but unfortunately it significantly increased training time and I had to cancel the command.

```
Cancelled
```