spark_ml

Cmd 1

Cmd 2

Spark Tutorial: Machine Learning

MLlib is Spark's machine learning (ML) library. Its goal is to make practical machine learning scalable and easy.

At a high level, it provides tools such as:

- **ML Algorithms**: common learning algorithms such as classification, regression, clustering, and collaborative filtering.
- **Featurization**: feature extraction, transformation, dimensionality reduction, and selection.
- **Pipelines**: tools for constructing, evaluating, and tuning ML Pipelines.
- **Persistence**: saving and load algorithms, models, and Pipelines.
- **Utilities**: linear algebra, statistics, data handling, etc.

Spark Machine Learning Workflow

Inspired by the `scikit-learn` project, `MLlib` standardizes APIs for machine learning algorithms to make it easier to combine multiple algorithms into a single pipeline, or workflow.

- **DataFrame**: This ML API uses DataFrame from Spark SQL as an ML dataset. A DataFrame could have different columns storing text, feature vectors, true labels, and predictions.
- **Transformer**: A Transformer is an algorithm which can transform one DataFrame into another DataFrame.
  - A feature transformer might take a DataFrame, read a column (e.g., text), map it into a new column (e.g., feature vectors) by calling `transform()` , and output a new DataFrame with the mapped column appended.
  - A learning model might take a DataFrame, read the column containing feature vectors, predict the label for each feature vector, and output a new DataFrame with predicted labels appended as a column.
- **Estimator**: An Estimator is an algorithm which can be fit on a DataFrame to produce a Transformer.
  - a learning algorithm such as LogisticRegression is an Estimator, and calling `fit()` trains a LogisticRegressionModel, which is a Model and hence a Transformer.
- **Pipeline**: A Pipeline chains multiple Transformers and Estimators together to specify an ML workflow.
- **Parameter**: All Transformers and Estimators now share a common API for specifying parameters.

How a ML Pipeline Works

- A Pipeline is specified as a sequence of stages, and each stage is either a **Transformer** or an **Estimator**.
- Stages are run in order, and the input DataFrame is transformed as it passes through each stage.
- For **Transformer** stages, the `transform()` method is called on the DataFrame.
- For **Estimator** stages, the `fit()` method is called to produce a Transformer (which becomes part of the `PipelineModel` , or fitted Pipeline), and that Transformer's `transform()` method is called on the DataFrame.

Cmd 3

ML Pipeline Example: Predicting Diamonds Price

We will be using diamonds dataset as an example to illustrate how to use machine learning pipeline to predict diamonds prices.

Here is the outline:

- *Loading data to DataFrame*: Load data as DataFrame
- *EDA*: Compute statistics and create visualizations to get a better understanding of the data.
- *Train validation split*: Split the data randomly into training and test sets. We will not look at the test data until *after* learning.
- On the training dataset:
  - *Extract features*: We will index categorical (String-valued) features so that DecisionTree can handle them.

- *Learn a model*: Run DecisionTree to learn how to predict a diamond's price from a description of the diamond.
- *Tune the model*: Tune the tree depth (complexity) using the training data. (This process is also called *model selection*.)
- *Evaluate the model*: Now look at the test dataset. Compare the initial model with the tuned model to see the benefit of tuning parameters.
- *Model Ensemble*: We will modify the pipeline to ensemble two (or more) models for a better prediction.

Cmd 4

1. Loading Data as Spark DataFrame

Cmd 5

```
## Mount S3 bucket nycdsabootcamp to the Databricks File System
s3Path = "s3a://{0}:{1}@{2}".format("AKIAI2P5MSEO2JYXJVQQ",
                                    "YJboxXSbraX4rg17aqtI+HmBjWCcpu4dxv2HW+bm",
                                    "nycdsabootcamp")
mntPath = "/mnt/data/"
dbutils.fs.mount(s3Path, mntPath)
```

```
java.rmi.RemoteException: java.lang.IllegalArgumentException: requirement failed: Directory already mounted: /mnt/data; nes
ted exception is:
```

Command took 0.93 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 2:10:15 PM on Spark_ML_Practise

Cmd 6

```
# Apache Parquet is a columnar storage format available to any project in the Hadoop ecosystem,
# regardless of the choice of data processing framework, data model or programming language.
import os
diamondPath = os.path.join(mntPath, "./pyspark_3/diamonds")
diamondsDF = spark.read.parquet(diamondPath)
```

▸ (1) Spark Jobs

Command took 4.14 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 2:10:50 PM on Spark_ML_Practise

Cmd 7

```
diamondsDF.printSchema()
diamondsDF.show(5)
```

▸ (1) Spark Jobs

```
root
 |-- carat: double (nullable = true)
 |-- cut: string (nullable = true)
 |-- color: string (nullable = true)
 |-- clarity: string (nullable = true)
 |-- depth: double (nullable = true)
 |-- table: double (nullable = true)
 |-- price: double (nullable = true)
 |-- x: double (nullable = true)
 |-- y: double (nullable = true)
 |-- z: double (nullable = true)

+-----+-------+-----+-------+-----+-----+-----+----+----+----+
|carat|    cut|color|clarity|depth|table|price|   x|   y|   z|
+-----+-------+-----+-------+-----+-----+-----+----+----+----+
| 0.23|  Ideal|    E|    SI2| 61.5| 55.0|326.0|3.95|3.98|2.43|
| 0.21|Premium|    E|    SI1| 59.8| 61.0|326.0|3.89|3.84|2.31|
| 0.23|   Good|    E|    VS1| 56.9| 65.0|327.0|4.05|4.07|2.31|
| 0.29|Premium|    I|    VS2| 62.4| 58.0|334.0| 4.2|4.23|2.63|
| 0.31|   Good|    J|    SI2| 63.3| 58.0|335.0|4.34|4.35|2.75|
+-----+-------+-----+-------+-----+-----+-----+----+----+----+
only showing top 5 rows
```

Command took 2.82 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 2:11:04 PM on Spark_ML_Practise

Cmd 8

2. EDA and Feature Transformation

Let's explore the data to get a better understanding of what is in there:

1. Show statistics for the continuous varialbes.
2. Find the count and mean price for each class of the categorical variables

3. Visualize your findings using `display()` .
4. Based on the data, what machine learning algorithm might be a better choice?

Cmd 9

```
# 1. describe(*cols) computes statistics for numeric columns.
diamondsDF.describe('carat','depth','table','x','y','z','price').show()
```

```
+-------+------------------+------------------+------------------+------------------+------------------+------------------+
-----------------+
|summary|             carat|             depth|             table|                 x|                 y|                 z|
            price|
+-------+------------------+------------------+------------------+------------------+------------------+------------------+
-----------------+
|  count|             53940|             53940|             53940|             53940|             53940|             53940|
            53940|
|   mean|0.7979397478679852| 61.74940489432624| 57.45718390804603| 5.731157211716609| 5.734525954764462|3.5387337782723316|
3932.799721913237|
| stddev|0.4740112444054196|1.4326213188336525|2.2344905628213247|1.1217607467924915|1.1421346741235616|0.7056988469499883|
3989.439738146397|
|    min|               0.2|              43.0|              43.0|               0.0|               0.0|               0.0|
            326.0|
|    max|              5.01|              79.0|              95.0|             10.74|              58.9|              31.8|
          18823.0|
+-------+------------------+------------------+------------------+------------------+------------------+------------------+
-----------------+
```

Command took 5.02 seconds -- by a user at 5/24/2017, 8:05:44 PM on unknown cluster

Cmd 10

```
# 2. groupBy(*cols) groups the DataFrame using the specified columns
#    agg(*exprs) computes aggregates and returns the result as a DataFrame.
diamondsDF.groupBy(diamondsDF.cut).agg({'*': 'count', 'price': 'mean'}).show()
diamondsDF.groupBy(diamondsDF.color).agg({'*': 'count', 'price': 'mean'}).show()
diamondsDF.groupBy(diamondsDF.clarity).agg({'*': 'count', 'price': 'mean'}).show()
```

▸ (15) Spark Jobs

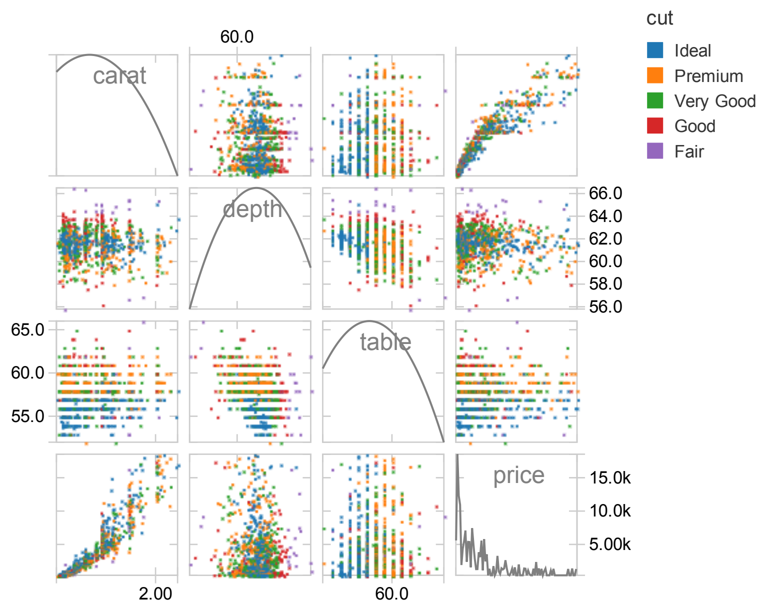
```
+---------+------------------+--------+
|      cut|        avg(price)|count(1)|
+---------+------------------+--------+
|  Premium|4584.2577042999055|   13791|
|    Ideal| 3457.541970210199|   21551|
|     Good| 3928.864451691806|    4906|
|     Fair| 4358.757763975155|    1610|
|Very Good|3981.7598907465654|   12082|
+---------+------------------+--------+

+-----+------------------+--------+
|color|        avg(price)|count(1)|
+-----+------------------+--------+
|    F| 3724.886396981765|    9542|
|    E|3076.7524752475247|    9797|
|    D|3169.9540959409596|    6775|
|    J|  5323.81801994302|    2808|
|    G| 3999.135671271697|   11292|
|    I| 5091.874953891553|    5422|
|    H| 4486.669195568401|    8304|
+-----+------------------+--------+
```

Command took 9.58 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 2:14:43 PM on Spark_ML_Practise

Cmd 11

```
# 3. sample(withReplacement, fraction) randomly sample 1000 obs of the dataset since display() currently only uses the
first 1000 rows
display(diamondsDF.sample(False, 1000.0/diamondsDF.count()))
```



Command took 6.32 seconds -- by a user at 5/24/2017, 8:09:03 PM on unknown cluster
Cmd 12

Since price is right skewed, we can apply a log transformation to make it normally distributed.

Cmd 13

```
from pyspark.sql.functions import *
diamondsDF = diamondsDF.withColumn("logPrice", log(diamondsDF.price + 1))
```

Command took 0.07 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 2:51:31 PM on Spark_ML_Practise
Cmd 14

3. Extracting, transforming and selecting features

Spark provides many algorithms for working with features, including extracting features from “raw” data, scaling, converting, or modifying features, etc.

3.1 StringIndexer

We now want to use tree based algorithms, which handles both continuous features (e.g., `"carat"`) and categorical features (e.g., `"cut"`). However, those algorithms require that categorical features be indexed as integers, i.e., [0, 1, 2, ..., numberOfCategories - 1]. `StringIndexer` is an *Estimator* that encodes a string column of labels to a column of label indices. The indices are in [0, `numLabels`), ordered by label frequencies, so the most frequent label gets index 0.

The example below shows how `StringIndexer` encodes the `cut` column to a numerical column.

Cmd 15

```
from pyspark.ml.feature import StringIndexer

indexer = StringIndexer(inputCol="cut", outputCol="cutIndex")
indexTransformer = indexer.fit(diamondsDF)
indexed = indexTransformer.transform(diamondsDF)
indexed.show(10)
```

▸ (2) Spark Jobs

```
+-----+---------+-----+-------+-----+-----+-----+----+----+----+--------+
|carat|      cut|color|clarity|depth|table|price|   x|   y|   z|cutIndex|
+-----+---------+-----+-------+-----+-----+-----+----+----+----+--------+
| 0.23|    Ideal|    E|    SI2| 61.5| 55.0|326.0|3.95|3.98|2.43|     0.0|
| 0.21|  Premium|    E|    SI1| 59.8| 61.0|326.0|3.89|3.84|2.31|     1.0|
| 0.23|     Good|    E|    VS1| 56.9| 65.0|327.0|4.05|4.07|2.31|     3.0|
```

```
| 0.29|  Premium|    I|    VS2| 62.4| 58.0|334.0| 4.2|4.23|2.63|     1.0|
| 0.31|     Good|    J|    SI2| 63.3| 58.0|335.0|4.34|4.35|2.75|     3.0|
| 0.24|Very Good|    J|   VVS2| 62.8| 57.0|336.0|3.94|3.96|2.48|     2.0|
| 0.24|Very Good|    I|   VVS1| 62.3| 57.0|336.0|3.95|3.98|2.47|     2.0|
| 0.26|Very Good|    H|    SI1| 61.9| 55.0|337.0|4.07|4.11|2.53|     2.0|
| 0.22|     Fair|    E|    VS2| 65.1| 61.0|337.0|3.87|3.78|2.49|     4.0|
| 0.23|Very Good|    H|    VS1| 59.4| 61.0|338.0| 4.0|4.05|2.39|     2.0|
+-----+---------+-----+-------+-----+-----+-----+----+----+----+--------+
only showing top 10 rows
```

Command took 5.64 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 2:50:41 PM on Spark_ML_Practise

Cmd 16

3.2 OneHotEncoder

Some algorithms, such as Logistic Regression, expect continuous features, and we therefore need to further encode those categorical features into binary features. `OneHotEncoder` is a *Transformer* that maps a column of label indices to a column of binary vectors, with at most a single one-value.

The example below shows how `OneHotEncoder` encodes the `cutIndex` column to a sparse matrix where each column corresponds to one possible value of that feature.

Cmd 17

```
from pyspark.ml.feature import OneHotEncoder

encoder = OneHotEncoder(inputCol = "cutIndex", outputCol = "cutClassVec")
encoded = encoder.transform(indexed)
encoded.show(10)
# The entry (4,[0],[1.0]) indicates in the sparse matrix representation there are 4 columns,
# column 0 has a value equals 1.0 and in the rest columns the values are all 0.
```

(1) Spark Jobs

```
+-----+---------+-----+-------+-----+-----+-----+----+----+----+--------+-------------+
|carat|      cut|color|clarity|depth|table|price|   x|   y|   z|cutIndex|  cutClassVec|
+-----+---------+-----+-------+-----+-----+-----+----+----+----+--------+-------------+
| 0.23|    Ideal|    E|    SI2| 61.5| 55.0|326.0|3.95|3.98|2.43|     0.0|(4,[0],[1.0])|
| 0.21|  Premium|    E|    SI1| 59.8| 61.0|326.0|3.89|3.84|2.31|     1.0|(4,[1],[1.0])|
| 0.23|     Good|    E|    VS1| 56.9| 65.0|327.0|4.05|4.07|2.31|     3.0|(4,[3],[1.0])|
| 0.29|  Premium|    I|    VS2| 62.4| 58.0|334.0| 4.2|4.23|2.63|     1.0|(4,[1],[1.0])|
| 0.31|     Good|    J|    SI2| 63.3| 58.0|335.0|4.34|4.35|2.75|     3.0|(4,[3],[1.0])|
| 0.24|Very Good|    J|   VVS2| 62.8| 57.0|336.0|3.94|3.96|2.48|     2.0|(4,[2],[1.0])|
| 0.24|Very Good|    I|   VVS1| 62.3| 57.0|336.0|3.95|3.98|2.47|     2.0|(4,[2],[1.0])|
| 0.26|Very Good|    H|    SI1| 61.9| 55.0|337.0|4.07|4.11|2.53|     2.0|(4,[2],[1.0])|
| 0.22|     Fair|    E|    VS2| 65.1| 61.0|337.0|3.87|3.78|2.49|     4.0|    (4,[],[])|
| 0.23|Very Good|    H|    VS1| 59.4| 61.0|338.0| 4.0|4.05|2.39|     2.0|(4,[2],[1.0])|
+-----+---------+-----+-------+-----+-----+-----+----+----+----+--------+-------------+
only showing top 10 rows
```

Command took 2.72 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 2:54:02 PM on Spark_ML_Practise

Cmd 18

3.3 Creating Pipeline Stages

Now we create stages that can map all the three categorical variables to indexed features and then one-hot encoded features. Later we will pass them into our machine learning pipeline.

Cmd 19

```
categoricalColumns = ["cut", "color", "clarity"]

indexStages = [] # stages in our Pipeline
for categoricalCol in categoricalColumns:
  # Category Indexing with StringIndexer
  stringIndexer = StringIndexer(inputCol=categoricalCol, outputCol=categoricalCol+"Index")
  # Add stages.  These are not run here, but will run all at once later on.
  indexStages.append(stringIndexer)
print 'StringIndexer stages:', indexStages

encodeStages = []
for categoricalCol in categoricalColumns:
  # Use OneHotEncoder to convert categorical variables into binary SparseVectors
  encoder = OneHotEncoder(inputCol=categoricalCol+"Index", outputCol=categoricalCol+"ClassVec")
  # Add stages.  These are not run here, but will run all at once later on.
  encodeStages.append(encoder)
print 'OneHotEncoder stages', encodeStages
```

```
StringIndexer stages: [StringIndexer_40a7a64b5075ae0c8c41, StringIndexer_4cd9acfae7607c2e706d, StringIndexer_4880baeda92a76
4ef2a0]
OneHotEncoder stages [OneHotEncoder_4b48a62c309ee8afb0dd, OneHotEncoder_4090b7f2fe5636a0ac14, OneHotEncoder_465589e7bb7af6a
10e58]
```

Command took 0.13 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 2:57:40 PM on Spark_ML_Practise

Cmd 20

3.4 VectorAssembler

`VectorAssembler` is a transformer that combines a given list of columns into a single vector column. It is useful for combining features into a single feature vector. The feature vector will later be used to train ML models like logistic regression and decision trees.

The example below shows how `VectorAssembler` combines three columns (`carat` , `depth` and `table`) into one column named `features` .

Cmd 21

```
from pyspark.ml.feature import VectorAssembler

assembler = VectorAssembler(inputCols=["carat", "depth", "table"],
                            outputCol="features")

output = assembler.transform(diamondsDF)
output.show(10)
```

▸ (1) Spark Jobs

```
+-----+---------+-----+-------+-----+-----+-----+----+----+----+------------------+----------------+
|carat|      cut|color|clarity|depth|table|price|   x|   y|   z|          logPrice|        features|
+-----+---------+-----+-------+-----+-----+-----+----+----+----+------------------+----------------+
| 0.23|    Ideal|    E|    SI2| 61.5| 55.0|326.0|3.95|3.98|2.43|5.7899601708972535|[0.23,61.5,55.0]|
| 0.21|  Premium|    E|    SI1| 59.8| 61.0|326.0|3.89|3.84|2.31|5.7899601708972535|[0.21,59.8,61.0]|
| 0.23|     Good|    E|    VS1| 56.9| 65.0|327.0|4.05|4.07|2.31| 5.793013608384144|[0.23,56.9,65.0]|
| 0.29|  Premium|    I|    VS2| 62.4| 58.0|334.0| 4.2|4.23|2.63| 5.814130531825066|[0.29,62.4,58.0]|
| 0.31|     Good|    J|    SI2| 63.3| 58.0|335.0|4.34|4.35|2.75| 5.817111159963204|[0.31,63.3,58.0]|
| 0.24|Very Good|    J|   VVS2| 62.8| 57.0|336.0|3.94|3.96|2.48| 5.820082930352362|[0.24,62.8,57.0]|
| 0.24|Very Good|    I|   VVS1| 62.3| 57.0|336.0|3.95|3.98|2.47| 5.820082930352362|[0.24,62.3,57.0]|
| 0.26|Very Good|    H|    SI1| 61.9| 55.0|337.0|4.07|4.11|2.53| 5.823045895483019|[0.26,61.9,55.0]|
| 0.22|     Fair|    E|    VS2| 65.1| 61.0|337.0|3.87|3.78|2.49| 5.823045895483019|[0.22,65.1,61.0]|
| 0.23|Very Good|    H|    VS1| 59.4| 61.0|338.0| 4.0|4.05|2.39|  5.82600010738045|[0.23,59.4,61.0]|
+-----+---------+-----+-------+-----+-----+-----+----+----+----+------------------+----------------+
only showing top 10 rows
```

Command took 2.42 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 2:59:39 PM on Spark_ML_Practise

Cmd 22

Depending on the machine learning algorithms, we need different treatments for categorical features.

- For algorithms (e.g. Linear Regression algorithm) that support only numerical features, we need to use one-hot encoded features.
- For algorithms (e.g. Decision Tree algorithm) that support both numerical and categorical features, we may use either indexed or one-hot encoded features.

Now we create an `assembler` that combines all numerical features and indexed categorical features into one called `features` which will be used as the inputs of the Decision Tree regressor.

Cmd 23

```
# Use Vector Assembler to create Feature Vector
# We wil pass the assembler into our Pipeline later
idxAssembler = VectorAssembler(inputCols=["carat",
                                          "cutIndex",
                                          "colorIndex",
                                          "clarityIndex",
                                          "depth", "table",
                                          "x", "y", "z"],
                               outputCol="features")
```

Command took 0.07 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 3:01:51 PM on Spark_ML_Practise

Cmd 24

4. Train Validation Splitting

We randomly split the dataset into 2 parts: training dataset (70%) and test dataset (30%). We will do all of our learning and tuning on the training set and validate our model on the test set to avoid overfitting issue.

Cmd 25

```
# Split data approximately into training (70%) and test (30%)
training, test = diamondsDF.randomSplit([0.7, 0.3])

# Cache the training and test datasets.
training.cache()
test.cache()
```

```
Out[11]: DataFrame[carat: double, cut: string, color: string, clarity: string, depth: double, table: double, price: double,
 x: double, y: double, z: double, logPrice: double]
```

Command took 0.22 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 3:01:57 PM on Spark_ML_Practise

Cmd 26

```
print "Training set size: ", training.count()
print "Validation set size: ", test.count()
```

▸ (2) Spark Jobs

```
Training set size:  37923
Validation set size:  16017
```

Command took 5.43 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 3:02:05 PM on Spark_ML_Practise

Cmd 27

5. Building the ML Pipeline with DecisionTreeRegressor

Now we have our training data and we will train a *Regression Tree* model by chaining all the `Estimators` and `Transformers` with a `Pipeline`. This will create a pipeline estimator.

Cmd 28

```
from pyspark.ml.regression import DecisionTreeRegressor
from pyspark.ml import Pipeline

# create a DecisionTree regressor estimator.
dt = DecisionTreeRegressor(featuresCol="features",
                           labelCol="logPrice")

# chain all the estimators and transformers stages into a Pipeline estimator
dtPipeline = Pipeline(stages = indexStages + [idxAssembler, dt])

# fit the estimator with training dataset to get a compiled pipeline with transformers and fitted models.
dtModel = dtPipeline.fit(training)
```

▸ (11) Spark Jobs

Command took 2.62 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 3:05:27 PM on Spark_ML_Practise

Cmd 29

6. Model Evaluation

A Pipeline is an Estimator. Thus, after a Pipeline's `fit()` method runs, it produces a `PipelineModel`, which is a `Transformer`.

Cmd 30

```
# Make predictions with test dataset.
predictions = dtModel.transform(test)
predictions.printSchema()
predictions.show()
```

▸ (1) Spark Jobs

```
root
 |-- carat: double (nullable = true)
 |-- cut: string (nullable = true)
 |-- color: string (nullable = true)
 |-- clarity: string (nullable = true)
 |-- depth: double (nullable = true)
 |-- table: double (nullable = true)
 |-- price: double (nullable = true)
 |-- x: double (nullable = true)
 |-- y: double (nullable = true)
 |-- z: double (nullable = true)
 |-- logPrice: double (nullable = true)
 |-- cutIndex: double (nullable = true)
 |-- colorIndex: double (nullable = true)
 |-- clarityIndex: double (nullable = true)
 |-- features: vector (nullable = true)
 |-- prediction: double (nullable = true)

+-----+-------+-----+-------+-----+-----+-----+----+----+----+------------------+--------+----------+------------+------
--------------+-----------------+
|carat|    cut|color|clarity|depth|table|price|   x|   y|   z|          logPrice|cutIndex|colorIndex|clarityIndex|
```

Command took 0.33 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 3:05:48 PM on Spark_ML_Practise

Cmd 31

We now can compute the error using the `RegressionEvaluator`. We will choose `rmse` (root mean squared error) as the error metric.

Cmd 32

```
from pyspark.ml.evaluation import RegressionEvaluator

# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(labelCol="logPrice",
                                predictionCol="prediction",
                                metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (rmse) on test dataset = %g" % rmse)
```

▸ (1) Spark Jobs

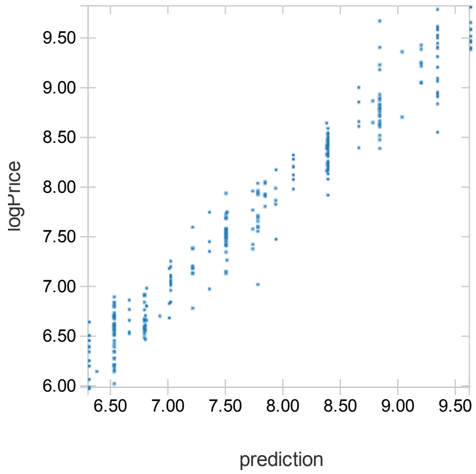
```
Root Mean Squared Error (rmse) on test dataset = 0.207136
```

Command took 1.51 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 3:06:58 PM on Spark_ML_Practise

Cmd 33

```
display(predictions.sample(False, 1000.0/diamondsDF.count()))
```

▸ (2) Spark Jobs



Command took 3.08 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 3:09:04 PM on Spark_ML_Practise

Cmd 34

7. Model Tuning

The `DecisionTreeRegressor` takes several parameters. We can try to tune those parameters to improve the model performance or prevent overfitting on the training data.

7.1 Tuning Parameters

`explainParams()` returns the documentation of all params with their default values and user-supplied values. The parameters that has been used in our model can therefore been found via:

Cmd 35

```
print "Decision Tree Model was fit using parameters: ", dt.explainParams()
```

```
Decision Tree Model was fit using parameters:  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 instance. Caching can speed up training of deepe
r trees. Users can set how often should the cache be checkpointed or disable it by setting checkpointInterval. (default: F
alse)
checkpointInterval: set checkpoint interval (>= 1) or disable checkpoint (-1). E.g. 10 means that the cache will get check
pointed every 10 iterations. (default: 10)
featuresCol: features column name. (default: features, current: features)
impurity: Criterion used for information gain calculation (case-insensitive). Supported options: variance (default: varian
ce)
labelCol: label column name. (default: label, current: logPrice)
maxBins: Max number of bins for discretizing continuous features.  Must be >=2 and >= number of categories for any categor
ical feature. (default: 32)
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)
maxMemoryInMB: Maximum memory in MB allocated to histogram aggregation. If too small, then 1 node will be split per iterat
ion, and its aggregates may exceed this size. (default: 256)
minInfoGain: Minimum information gain for a split to be considered at a tree node. (default: 0.0)
minInstancesPerNode: Minimum number of instances each child must have after split. If a split causes the left or right chi
ld to have fewer than minInstancesPerNode, the split will be discarded as invalid. Should be >= 1. (default: 1)
predictionCol: prediction column name. (default: prediction)
seed: random seed. (default: -2808853809871465425)
varianceCol: column name for the biased sample variance of prediction. (undefined)
```

Command took 0.04 seconds -- by a user at 3/21/2017, 11:15:48 AM on unknown cluster

Cmd 36

Particularly, our current tree has the `maxDepth` and `maxBins`:

Cmd 37

```
print "Max Depth of the Decision Tree regressor: ", dt.getMaxDepth()
print "Max Bins of the Decision Tree regressor: ", dt.getMaxBins()
```

```
Max Depth of the Decision Tree regressor:  5
Max Bins of the Decision Tree regressor:  32
```

Command took 0.04 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 3:14:24 PM on Spark_ML_Practise

Cmd 38

7.2 Model Selection via Cross-Validation

We can perform *cross-validation* by using `CrossValidator` to select the best paramter set from a grid of parameters.

Cmd 39

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

dtParamGrid = ParamGridBuilder()\
.addGrid(dt.maxBins, range(26, 40, 2))\
.addGrid(dt.maxDepth, range(5, 15, 2)).build()

crossval = CrossValidator(estimator=dtPipeline,
                          estimatorParamMaps=dtParamGrid,
                          evaluator=evaluator,
                          numFolds=5)

# Run cross-validation, and choose the best set of parameters.
# for Databricks community edition the following line takes ~6 mins to run
cvModelDT = crossval.fit(training)
```

▸ (54) Spark Jobs

Command took 4.35 minutes -- by meghana.rwgsql@gmail.com at 5/24/2017, 9:16:40 PM on Spark_ML

Cmd 40

We can get access to the `DecisionTreeRegressionModel` that used in the best `pipelineModel` from its last stage.

Cmd 41

```python
best_dt = cvModelDT.bestModel.stages[-1]
print 'The depth of the decision tree is', best_dt.depth
print 'The number of nodes of the decision tree is', best_dt.numNodes
```

```
The depth of the decision tree is 11
The number of nodes of the decision tree is 3291
```

Command took 0.04 seconds -- by meghana.rwgsql@gmail.com at 5/24/2017, 9:21:14 PM on Spark_ML

Cmd 42

To make prediciton on the test dataset, cvModel uses the best model

Cmd 43

```python
# Make predictions on test dataset. cvModelDT uses the best model.
cvPredict = cvModelDT.transform(test)
cvPredict.select("features", "logPrice","prediction").show(10)
```

▸ (1) Spark Jobs

```
+--------------------+-----------------+-----------------+
|            features|         logPrice|       prediction|
+--------------------+-----------------+-----------------+
|[0.2,0.0,4.0,1.0,...|5.908082938168931|6.217727189591819|
|[0.2,1.0,4.0,1.0,...|5.908082938168931|5.924863523122549|
|[0.2,1.0,1.0,2.0,...|5.846438775057725| 6.22241095771072|
|[0.2,2.0,1.0,1.0,...|5.908082938168931|6.072691943939274|
|[0.21,1.0,4.0,1.0...|5.958424693029782|5.924863523122549|
|[0.21,1.0,4.0,1.0...|5.958424693029782|5.924863523122549|
|[0.22,1.0,4.0,1.0...|6.003887067106539|6.120689103257942|
|[0.23,3.0,1.0,5.0...|6.129050210060545|6.241166975460359|
|[0.23,3.0,1.0,4.0...|6.226536669287466|6.241166975460359|
|[0.23,3.0,2.0,3.0...|5.932245187448011|6.072691943939274|
+--------------------+-----------------+-----------------+
only showing top 10 rows
```

Command took 0.32 seconds -- by meghana.rwgsql@gmail.com at 5/24/2017, 9:22:30 PM on Spark_ML

Cmd 44

Now the `rmse` on test dataset is reduced to:

Cmd 45

```
# Make predictions on test documents. cvModel uses the best model
rmse = evaluator.evaluate(cvPredict)
print('Root Mean Squared Error (RMSE) on test data = %g' % rmse)
```

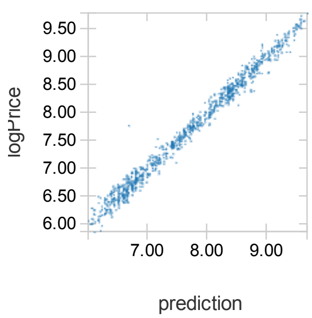
▸ (1) Spark Jobs

```
Root Mean Squared Error (RMSE) on test data = 0.118007
```

Command took 1.38 seconds -- by meghana.rwgsql@gmail.com at 5/24/2017, 9:22:44 PM on Spark_ML

Cmd 46

```
display(cvPredict.sample(False, 1000.0/cvPredict.count()))
```



Showing sample based on the first 1000 rows.

Command took 0.27 seconds -- by a user at 3/21/2017, 11:31:10 AM on unknown cluster

Cmd 47

8. Saving and Loading Pipelines

Often times it is worth it to save a model or a pipeline to disk for later use. We can:

- use `.save(path)` to save this ML instance to the given path, and
- use `.load(path)` to reads an ML instance from the input path.

Cmd 48

```
# To save a PipelineModel
cvModelDT.bestModel.write().overwrite().save('best_pipe_dt')
```

▸ (10) Spark Jobs

Command took 8.74 seconds -- by meghana.rwgsql@gmail.com at 5/24/2017, 9:23:21 PM on Spark_ML

Cmd 49

```
%fs ls /best_pipe_dt/
```

path
dbfs:/best_pipe_dt/metadata/
dbfs:/best_pipe_dt/stages/

Command took 2.03 seconds -- by meghana.rwgsql@gmail.com at 5/24/2017, 9:23:59 PM on Spark_ML

Cmd 50

```
from pyspark.ml import PipelineModel

# To load a PipeLineModel
best_pipe_dt = PipelineModel.load("best_pipe_dt")
```

Command took 5.04 seconds -- by a user at 3/21/2017, 11:34:26 AM on unknown cluster

Cmd 51

Exercise

1. Follow the same steps to train a `RandomForestRegressor` (or a `LinearRegression`, which requires `encodeStages` in the pipeline) with the default settings. What's your `rmse` on the test dataset?
2. Use Cross-Validation to tune your model.

Cmd 52

Answer 1: LinearRegression

Cmd 53

```
from pyspark.ml.regression import LinearRegression

lr = LinearRegression(featuresCol="lmFeatures", labelCol="logPrice")
# create assember to include encoded features
lmAssembler = VectorAssembler(inputCols=["carat",
                                         "cutClassVec",
                                         "colorClassVec",
                                         "clarityClassVec",
                                         "depth", "table",
                                         "x", "y", "z"],
                              outputCol="lmFeatures")

# Chain indexer, encoder and lr in a Pipeline
lrPipeline = Pipeline(stages = indexStages + encodeStages + [lmAssembler, lr])

# Train model. This also runs the indexer and encoder.
lrModel = lrPipeline.fit(training)
```

▸ (8) Spark Jobs

Command took 2.72 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 5:46:16 PM on Spark_ML_Practise

Cmd 54

```
# Make predictions on test data
lrPredict = lrModel.transform(test)
rmse = evaluator.evaluate(lrPredict)
print('Root Mean Squared Error (RMSE) on test data = %g' % rmse)
```

▸ (1) Spark Jobs

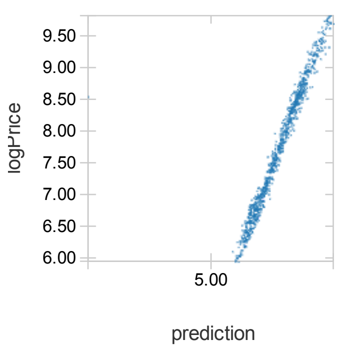
```
Root Mean Squared Error (RMSE) on test data = 0.224987
```

Command took 0.42 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 5:50:57 PM on Spark_ML_Practise

Cmd 55

```
display(lrPredict.sample(False, 1000.0/lrPredict.count()))
```

▸ (2) Spark Jobs



Showing sample based on the first 1000 rows.

Command took 0.56 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 5:51:06 PM on Spark_ML_Practise

Cmd 56

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

lrParamGrid = ParamGridBuilder()\
.addGrid(lr.regParam, [0, .1, 1, 10])\
.addGrid(lr.elasticNetParam, [0, .25, .5, .75, 1]).build()

lrCrossval = CrossValidator(estimator=lrPipeline,
                            estimatorParamMaps=lrParamGrid,
                            evaluator=evaluator,
                            numFolds=5)

cvModelLR = lrCrossval.fit(training)
```

▸ (70) Spark Jobs

Command took 3.34 minutes -- by meghana.rwgsql@gmail.com at 5/29/2017, 5:53:27 PM on Spark_ML_Practise
Cmd 57

```python
# Make predictions on test data
lrPredict = cvModelLR.transform(test)
rmse = evaluator.evaluate(lrPredict)
print('Root Mean Squared Error (RMSE) on test data = %g' % rmse)
```

▸ (1) Spark Jobs

Root Mean Squared Error (RMSE) on test data = 0.224987

Command took 0.32 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 5:57:05 PM on Spark_ML_Practise
Cmd 58

```python
display(lrPredict.sample(False, 1000.0/lrPredict.count()))
```

▸ (2) Spark Jobs

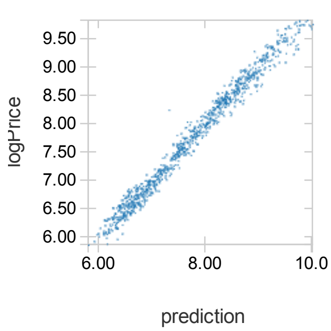
carat	cut	color	clarity	depth	table	price	x	y	z	logPrice	cutIndex	colorIndex	clarityIndex	cutClassVec	col
0.23	Good	E	VVS2	62.2	60	505	3.9	3.94	2.44	6.226536669287466	3	1	4	▸[0,4,[3],[1]]	▸[0
0.23	Ideal	H	VVS1	61.1	55	484	3.98	4.01	2.44	6.184148890937483	0	3	5	▸[0,4,[0],[1]]	▸[0
0.23	Very Good	D	VS1	62.4	56	468	3.93	3.98	2.46	6.150602768446279	2	4	3	▸[0,4,[2],[1]]	▸[0
0.23	Very Good	D	VS2	60.2	57	577	4.02	4.07	2.43	6.359573868672378	2	4	1	▸[0,4,[2],[1]]	▸[0
0.23	Very Good	D	VS2	61.6	58	402	3.96	3.99	2.45	5.998936561946683	2	4	1	▸[0,4,[2],[1]]	▸[0
0.23	Very Good	D	VVS1	60.5	55	472	4.01	4.02	2.43	6.159095388491933	2	4	5	▸[0,4,[2],[1]]	▸[0
0.23	Very Good	E	VVS1	61.8	59	472	3.89	3.91	2.41	6.159095388491933	2	1	5	▸[0,4,[2],[1]]	▸[0
0.23	Very Good	E	VVS2	59	63	485	3.98	4.05	2.37	6.186208623900494	2	1	4	▸[0,4,[2],[1]]	▸[0
0.23	Very Good	E	VVS2	61.2	58	530	3.93	3.98	2.42	6.274762021241939	2	1	4	▸[0,4,[2],[1]]	▸[0

Showing the first 1000 rows.

Command took 0.48 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 6:01:21 PM on Spark_ML_Practise
Cmd 59

```python
display(lrPredict.sample(False, 1000.0/lrPredict.count()))
```

▸ (2) Spark Jobs



Command took 0.46 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 6:05:36 PM on Spark_ML_Practise
Cmd 60

Answer 2: RandomForestRegressor

Cmd 61

```
from pyspark.ml.regression import RandomForestRegressor

# Create a RandomForest Regressor.
rf = RandomForestRegressor(featuresCol="features",
                           labelCol="logPrice")

# Chain indexer and rf in a Pipeline
rfPipeline = Pipeline(stages = indexStages + [idxAssembler, rf])

# Train model. This also runs the indexer.
rfModel = rfPipeline.fit(training)
```

▸ (12) Spark Jobs

Command took 2.82 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 5:58:45 PM on Spark_ML_Practise

Cmd 62

```
# Make predictions.
rfPredictions = rfModel.transform(test)
rmseRF = evaluator.evaluate(rfPredictions)
print("Root Mean Squared Error (rmse) on test data = %g" % rmseRF)
```

▸ (1) Spark Jobs

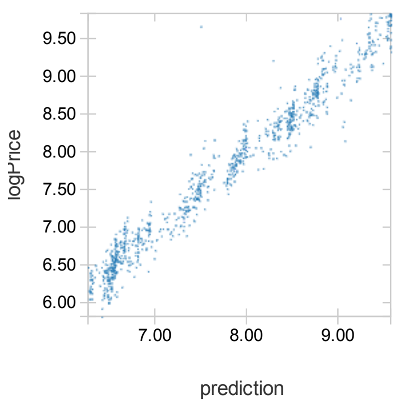
```
Root Mean Squared Error (rmse) on test data = 0.195575
```

Command took 1.21 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 5:59:03 PM on Spark_ML_Practise

Cmd 63

```
display(rfPredictions.sample(False, 1000.0/rfPredictions.count()))
```

▸ (2) Spark Jobs



Command took 0.32 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 6:03:30 PM on Spark_ML_Practise

Cmd 64

```
#display(rfPredictions.sample(False, 1000.0/cvPredict.count()))
```

```
NameError: name 'cvPredict' is not defined
```

Command took 0.19 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 5:59:12 PM on Spark_ML_Practise

Cmd 65

```
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
rfParamGrid = ParamGridBuilder()\
.addGrid(rf.numTrees, [20, 40, 80])\
.addGrid(rf.maxDepth, [3,5,7])\
.addGrid(rf.featureSubsetStrategy, ['all', 'sqrt'])\
.addGrid(rf.subsamplingRate, [.7, 1.0])\
.build()

cvRF = CrossValidator(estimator=rfPipeline,
                      estimatorParamMaps=rfParamGrid,
                      evaluator=evaluator,
                      numFolds=5)

# Run cross-validation, and choose the best set of parameters.
# for Databricks community edition the following line takes ~13 mins to run
cvModelRF = cvRF.fit(training)
```

(58) Spark Jobs

Command took 14.66 minutes -- by meghana.rwgsql@gmail.com at 5/29/2017, 6:11:12 PM on Spark_ML_Practise

Cmd 66

```
cvPredictRF = cvModelRF.transform(test)
rmseRF = evaluator.evaluate(cvPredictRF)
print("Root Mean Squared Error (rmse) on test data = %g" % rmseRF)
```

(1) Spark Jobs

```
Root Mean Squared Error (rmse) on test data = 0.143528
```

Command took 0.98 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 6:35:32 PM on Spark_ML_Practise

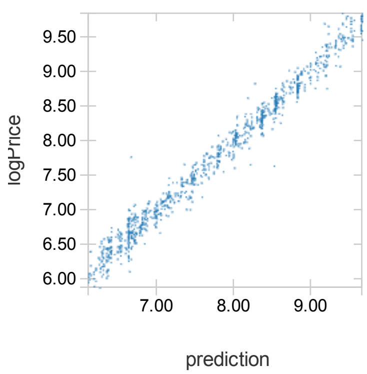
Cmd 67

```
best_rf = cvModelRF.bestModel.stages[-1]
```

Command took 0.07 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 6:35:42 PM on Spark_ML_Practise

Cmd 68

```
display(cvPredictRF.sample(False, 1000.0/cvPredictRF.count()))
```



Command took 0.26 seconds -- by a user at 3/21/2017, 1:57:38 PM on unknown cluster

Cmd 69

Answer 3: GBTRegressor

Cmd 70

```
from pyspark.ml.regression import GBTRegressor

# Create a RandomForest Regressor.
gbt = GBTRegressor(featuresCol="features",
                   labelCol="logPrice")

# Chain indexer and rf in a Pipeline
gbtPipeline = Pipeline(stages = indexStages + [assembler, gbt])

# Train model. This also runs the indexer.
gbtModel = gbtPipeline.fit(training)
```

▸ (57) Spark Jobs

Command took 11.16 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 6:40:55 PM on Spark_ML_Practise

Cmd 71

```
gbtPredictions = gbtModel.transform(test)
rmseGBT = evaluator.evaluate(gbtPredictions)
print("Root Mean Squared Error (rmse) on test data = %g" % rmseGBT)
```

▸ (1) Spark Jobs

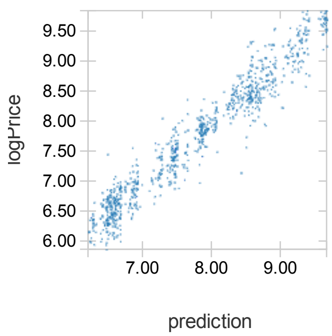
```
Root Mean Squared Error (rmse) on test data = 0.244245
```

Command took 0.83 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 6:49:59 PM on Spark_ML_Practise

Cmd 72

```
display(gbtPredictions.sample(False, 1000.0/gbtPredictions.count()))
```

▸ (2) Spark Jobs



Command took 0.28 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 6:50:12 PM on Spark_ML_Practise

Cmd 73

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

paramGrid = ParamGridBuilder()\
  .addGrid(gbt.maxIter, [20, 40, 60])\
  .addGrid(gbt.maxDepth, [3, 5, 7])\
  .addGrid(gbt.stepSize, [.05, .1, .2])\
  .build()

cvGBT = CrossValidator(estimator=gbtPipeline,
                       estimatorParamMaps=paramGrid,
                       evaluator=evaluator,
                       numFolds=5)

# Run cross-validation, and choose the best set of parameters.
# for Databricks community edition the following line takes ~1 hour to run
cvModelGBT = cvGBT.fit(training)
```

▸ (57) Spark Jobs

Command took 56.95 minutes -- by meghana.rwgsql@gmail.com at 5/29/2017, 6:58:35 PM on Spark_ML_Practise

Cmd 74

```
cvPredictGBT = cvModelGBT.transform(test)
rmseGBT = evaluator.evaluate(cvPredictGBT)
print("Root Mean Squared Error (rmse) on test data = %g" % rmseGBT)
```

▸ (1) Spark Jobs

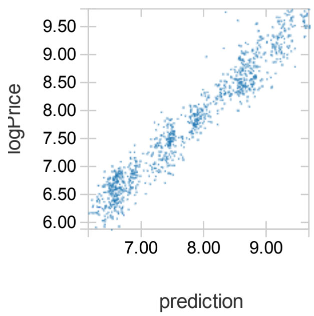
```
Root Mean Squared Error (rmse) on test data = 0.241804
```

Command took 1.21 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 9:31:58 PM on Spark_ML_Practise

Cmd 75

```
display(cvPredictGBT.sample(False, 1000.0/cvPredictGBT.count()))
```

▸ (2) Spark Jobs



Showing sample based on the first 1000 rows.

Command took 0.35 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 10:11:00 PM on Spark_ML_Practise

Cmd 76

9 Model Ensemble

Cmd 77

```
dt = DecisionTreeRegressor(featuresCol="features",
                           labelCol="logPrice",
                           predictionCol="dtPrediction")

lr = LinearRegression(featuresCol="lmFeatures",
                      labelCol="logPrice",
                      predictionCol="lrPrediction")

ensAssembler = VectorAssembler(inputCols=["lrPrediction",
                                          "dtPrediction"],
                               outputCol="ensFeatures")

rf = RandomForestRegressor(featuresCol="ensFeatures",
                           labelCol="logPrice")

ensPipeline = Pipeline(stages = indexStages + encodeStages + [idxAssembler, dt] + [lmAssembler, lr] + [ensAssembler,
rf])

# Train model. This also runs the indexer.
ensModel = ensPipeline.fit(training)
```

▸ (25) Spark Jobs

Command took 5.93 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 10:12:37 PM on Spark_ML_Practise

Cmd 78

```
ensPredict = ensModel.transform(test)
rmseEns = evaluator.evaluate(ensPredict)
print("Root Mean Squared Error (rmse) on test data = %g" % rmseEns)
```

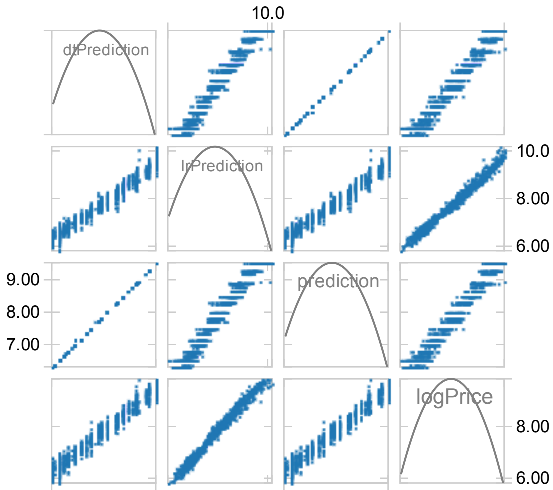
▸ (1) Spark Jobs

```
Root Mean Squared Error (rmse) on test data = 0.141166
```

Command took 1.41 seconds -- by meghana.rwgsql@gmail.com at 5/29/2017, 10:12:53 PM on Spark_ML_Practise

Cmd 79

```
display(ensPredict.sample(False, 1000.0/ensPredict.count()))
```



Command took 0.41 seconds -- by a user at 3/21/2017, 2:21:17 PM on unknown cluster

Cmd 80

```
ensParamGrid = ParamGridBuilder()\
.addGrid(dt.maxDepth, range(11, 13))\
.addGrid(rf.numTrees, [40, 80])\
.addGrid(rf.maxDepth, [3,5,7])\
.build()

ensCV = CrossValidator(estimator=ensPipeline,
                       estimatorParamMaps=ensParamGrid,
                       evaluator=evaluator,
                       numFolds=5)

cvModelEns = ensCV.fit(training)
```

Command took 6.81 minutes -- by a user at 3/21/2017, 2:23:53 PM on unknown cluster

Cmd 81

```
ensPredict = cvModelEns.transform(test)
rmseEns = evaluator.evaluate(ensPredict)
print("Root Mean Squared Error (rmse) on test data = %g" % rmseEns)
```

Cmd 82