In [1]:
```
#Read dataset in Spark
df = sqlContext.read.load("dbfs:/databricks-datasets/bikeSharing/data-001/day.csv",
                          format='com.databricks.spark.csv',
                          header='true',
                          inferSchema='true')
#This is a databrick dataset of the bike-sharing demand prediction, and the test-train dateset is the given dataset
```

In [2]:
```
#2. Get summary of data and variable types
df.printSchema()
```

root -- instant: integer (nullable = true) -- dteday: timestamp (nullable = true) -- season: integer (nullable = true) -- yr: integer (nullable = true) -- mnth: integer (nullable = true) -- holiday: integer (nullable = true) -- weekday: integer (nullable = true) -- workingday: integer (nullable = true) -- weathersit: integer (nullable = true) -- temp: double (nullable = true) -- atemp: double (nullable = true) -- hum: double (nullable = true) -- windspeed: double (nullable = true) -- casual: integer (nullable = true) -- registered: integer (nullable = true) -- cnt: integer (nullable = true)

In [3]:
```
#df.show(5)
display(df.take(5))
```

instant	dteday	season	yr	mnth	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual	regis
1	2011-01-01T00:00:00.000+0000	1	0	1	0	6	0	2	0.344167	0.363625	0.805833	0.160446	331	
2	2011-01-02T00:00:00.000+0000	1	0	1	0	0	0	2	0.363478	0.353739	0.696087	0.248539	131	
3	2011-01-03T00:00:00.000+0000	1	0	1	0	1	1	1	0.196364	0.189405	0.437273	0.248309	120	
4	2011-01-04T00:00:00.000+0000	1	0	1	0	2	1	1	0.2	0.212122	0.590435	0.160296	108	
5	2011-01-05T00:00:00.000+0000	1	0	1	0	3	1	1	0.226957	0.22927	0.436957	0.1869	82	

In [4]:
```
#Given Train file from which data frame is generated
bs_df = spark.sql("select * from bike_sharing_train_csv")
display(bs_df.take(5))
```

datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count
01-01-2011 00:00	1	0	0	1	9.84	14.395	81	0.0	3	13	16
01-01-2011 01:00	1	0	0	1	9.02	13.635	80	0.0	8	32	40
01-01-2011 02:00	1	0	0	1	9.02	13.635	80	0.0	5	27	32
01-01-2011 03:00	1	0	0	1	9.84	14.395	75	0.0	3	10	13
01-01-2011 04:00	1	0	0	1	9.84	14.395	75	0.0	0	1	1

In [5]:
```
bs_df.printSchema()
```

root -- datetime: string (nullable = true) -- season: integer (nullable = true) -- holiday: integer (nullable = true) -- workingday: integer (nullable = true) -- weather: integer (nullable = true) -- temp: double (nullable = true) -- atemp: double (nullable = true) -- humidity: integer (nullable = true) -- windspeed: double (nullable = true) -- casual: integer (nullable = true) -- registered: integer (nullable = true) -- count: integer (nullable = true)

In [6]:
```
bs_df.describe().show()
```

+-------+---------------+-----------------+------------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+ summary| datetime| season| holiday| workingday| weather| temp| atemp| humidity| windspeed| casual| registered| count| +-------+---------------+-----------------+------------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+ count| 10886| 10886| 10886| 10886| 10886| 10886| 10886| 10886| 10886| 10886| 10886| 10886| mean| null|2.5066139996325556|0.02856880396839978|0.6808745177291935| 1.418427337865148|20.230859819952173|23.65508405291192| 61.88645967297446|12.799395406945093|36.02195480433584| 155.5521771082124|191.57413191254824| stddev| null|1.1161743093443237|0.16659885062470944|0.4661591687997361|0.6338385858190968| 7.791589843987573| 8.47460062648494|19.245033277394704| 8.16453732683871|49.96047657264955|151.03903308192452|181.14445383028493| min|01-01-2011 00:00| 1| 0| 0| 1| 0.82| 0.76| 0| 0.0| 0| 0| 1| max|19-12-2012 23:00| 4| 1| 1| 4| 41.0| 45.455| 100| 56.9969| 367| 886| 977| +-------+---------------+-----------------+------------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+

In [7]:
```
bs_df.explain()
```

```
== Physical Plan == *(1) FileScan csv
default.bike_sharing_train_csv[datetime#254,season#255,holiday#256,workingday#257,weather#258,temp#259,atemp#260,humidity#261,windspeed
Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex[dbfs:/FileStore/tables/train.csv], PartitionFilters: [], PushedFilters: [],
ReadSchema: struct<datetime:string,season:int,holiday:int,workingday:int,weather:int,temp:double,atemp:double...
```

In [8]:
```
#Check for any missing value in dataset and treat it
print(bs_df.count())
df_no_null = bs_df.na.drop()
print(df_no_null.count())
```

```
10886 10886
```

In [9]:
```
#Check what are the distinct seasons present to explode them
display(bs_df.select('season').distinct())
```

season
1
3
4
2

In [10]:
```
#user defined function to help creat new columns
def valueToCategory(value, encoding_index):
   if(value == encoding_index):
      return 1
   else:
    return 0
```

In [11]:
```
#Explode season column into separate columns such as season_<val> and drop season
from pyspark.sql.functions import udf
from pyspark.sql.functions import lit
from pyspark.sql.types import *
from pyspark.sql.functions import col
udfValueToCategory = udf(valueToCategory, IntegerType())
bs_df_encoded = (bs_df.withColumn("season_1", udfValueToCategory(col('season'),lit(1)))
                      .withColumn("season_2", udfValueToCategory(col('season'),lit(2)))
                      .withColumn("season_3", udfValueToCategory(col('season'),lit(3)))
                      .withColumn("season_4", udfValueToCategory(col('season'),lit(4))))
bs_df_encoded = bs_df_encoded.drop('season')
```

In [12]:
```
display(bs_df_encoded.take(5))
```

datetime	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count	season_1	season_2	season_3	season_4
01-01-2011 00:00	0	0	1	9.84	14.395	81	0.0	3	13	16	1	0	0	0
01-01-2011 01:00	0	0	1	9.02	13.635	80	0.0	8	32	40	1	0	0	0
01-01-2011 02:00	0	0	1	9.02	13.635	80	0.0	5	27	32	1	0	0	0
01-01-2011 03:00	0	0	1	9.84	14.395	75	0.0	3	10	13	1	0	0	0
01-01-2011 04:00	0	0	1	9.84	14.395	75	0.0	0	1	1	1	0	0	0

In [13]:
```
#Execute the same for weather as weather_<val> and drop weather
display(bs_df.select('weather').distinct())
```

weather
1
3
4
2

In [14]:
```
bs_df_encoded = (bs_df_encoded.withColumn("weather_1", udfValueToCategory(col('weather'),lit(1)))
                    .withColumn("weather_2", udfValueToCategory(col('weather'),lit(2)))
                    .withColumn("weather_3", udfValueToCategory(col('weather'),lit(3)))
                    .withColumn("weather_4", udfValueToCategory(col('weather'),lit(4))))
bs_df_encoded = bs_df_encoded.drop('weather')
```

In [15]:
```
display(bs_df_encoded.take(5))
```

datetime	holiday	workingday	temp	atemp	humidity	windspeed	casual	registered	count	season_1	season_2	season_3	season_4	weather_1
01-01-2011 00:00	0	0	9.84	14.395	81	0.0	3	13	16	1	0	0	0	1
01-01-2011 01:00	0	0	9.02	13.635	80	0.0	8	32	40	1	0	0	0	1
01-01-2011 02:00	0	0	9.02	13.635	80	0.0	5	27	32	1	0	0	0	1
01-01-2011 03:00	0	0	9.84	14.395	75	0.0	3	10	13	1	0	0	0	1
01-01-2011 04:00	0	0	9.84	14.395	75	0.0	0	1	1	1	0	0	0	1

In [16]:
```
# Split datetime into meaningful columns such as hour,day,month,year,etc
from pyspark.sql.functions import split
from pyspark.sql.functions import *
from pyspark.sql.types import *
bs_df_encoded = bs_df_encoded.withColumn('hour',  split(split(bs_df_encoded['datetime'], ' ')[1], ':')[0].cast('int'))
bs_df_encoded = bs_df_encoded.withColumn('month', split(split(bs_df_encoded['datetime'], ' ')[0], '-')[0].cast('int'))
bs_df_encoded = bs_df_encoded.withColumn('day', split(split(bs_df_encoded['datetime'], ' ')[0], '-')[1].cast('int'))
bs_df_encoded = bs_df_encoded.withColumn('year', split(split(bs_df_encoded['datetime'], ' ')[0], '-')[2].cast('int'))
```

In [17]:
```
display(bs_df_encoded.take(5))
```

datetime	holiday	workingday	temp	atemp	humidity	windspeed	casual	registered	count	season_1	season_2	season_3	season_4	weather_1
01-01-2011 00:00	0	0	9.84	14.395	81	0.0	3	13	16	1	0	0	0	1
01-01-2011 01:00	0	0	9.02	13.635	80	0.0	8	32	40	1	0	0	0	1
01-01-2011 02:00	0	0	9.02	13.635	80	0.0	5	27	32	1	0	0	0	1
01-01-2011 03:00	0	0	9.84	14.395	75	0.0	3	10	13	1	0	0	0	1
01-01-2011 04:00	0	0	9.84	14.395	75	0.0	0	1	1	1	0	0	0	1

In [18]:
```
bs_df_encoded.printSchema()
bs_df_encoded = bs_df_encoded.drop('datetime')
bs_df_encoded = bs_df_encoded.withColumnRenamed("count", "label")
```

root -- datetime: string (nullable = true) -- holiday: integer (nullable = true) -- workingday: integer (nullable = true) -- temp: double (nullable = true) -- atemp: double (nullable = true) -- humidity: integer (nullable = true) -- windspeed: double (nullable = true) -- casual: integer (nullable = true) -- registered: integer (nullable = true) -- count: integer (nullable = true) -- season_1: integer (nullable = true) -- season_2: integer (nullable = true) -- season_3: integer (nullable = true) -- season_4: integer (nullable = true) -- weather_1: integer (nullable = true) -- weather_2: integer (nullable = true) -- weather_3: integer (nullable = true) -- weather_4: integer (nullable = true) -- hour: integer (nullable = true) -- month: integer (nullable = true) -- day: integer (nullable = true) -- year: integer (nullable = true)

In [19]:
```
#Split the dataset into train and train_test
from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit
train, test = bs_df_encoded.randomSplit([0.9, 0.1], seed=12345)
```

In [20]:
```
#The features are assembled to send it to model
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler

assembler = VectorAssembler(
    inputCols=["holiday","workingday","temp","atemp","humidity","windspeed","casual","registered","label","season_1","sea
    outputCol="features")

output = assembler.transform(train)
print("Assembled columns 'hour', 'day' etc  to vector column 'features'")
display(output.take(5))
print(output.count())
train_output = output.na.drop()
print(train_output.count())
```

holiday	workingday	temp	atemp	humidity	windspeed	casual	registered	label	season_1	season_2	season_3	season_4	weather_1	weather_2
0	0	3.28	2.275	79	31.0009	0	24	24	1	0	0	0	0	0

holiday	workingday	temp	atemp	humidity	windspeed	casual	registered	label	season_1	season_2	season_3	season_4	weather_1	weather_2
0	0	3.28	3.79	53	16.9979	0	26	26	1	0	0	0	1	0
0	0	3.28	4.545	53	12.998	0	1	1	1	0	0	0	1	0

holiday	workingday	temp	atemp	humidity	windspeed	casual	registered	label	season_1	season_2	season_3	season_4	weather_1	weather_2
0	0	3.28	4.545	53	12.998	0	1	1	1	0	0	0	1	0
0	0	3.28	4.545	53	12.998	1	5	6	1	0	0	0	1	0

In [21]:
```
test_output = assembler.transform(test)
print(test_output.count())
train_output = test_output.na.drop()
print(test_output.count())
print("Assembled columns 'hour', 'day' etc  to vector column 'features'")
#.select("features", "clicked")
```

1089 1089 Assembled columns 'hour', 'day' etc to vector column 'features'

In [22]:
```
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.regression import LinearRegression
lr = LinearRegression(maxIter=10)

# Fit the model
lrModel = lr.fit(train_output)
```

In [23]:
```
# Print the coefficients and intercept for logistic regression
print("Coefficients: " + str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))
```

Coefficients:
[0.241007000329,0.0240300559307,-0.00329772606512,-0.0038201221511,-0.00311898556861,0.00062578456448,0.563257347413,0.563189130
Intercept: 173.7435412550812

In [24]:
```
import pyspark.sql.functions
predictions = lrModel.transform(test_output)\
    .select("features", "label", "prediction")\
    .take(10)
display(predictions)

from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.mllib.evaluation import BinaryClassificationMetrics
# testRDD = test.rdd
# predictionAndLabels = testRDD.map(lambda lp: (float(model.predict(lp.features)), lp.label))
# # Evaluate model
# metrics = BinaryClassificationMetrics(predictionAndLabels)
# f1Score = metrics.fMeasure()
# print(f1Score)
from pyspark.ml.evaluation import RegressionEvaluator
lr_evaluator = RegressionEvaluator(predictionCol="prediction", labelCol="label",metricName="r2")
# print("R Squared (R2) on test data = %g" % lr_evaluator.evaluate(predictions))
```

features	label	prediction
List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(3.28, 4.545, 53.0, 12.998, 18.0, 18.0, 1.0, 1.0, 7.0, 12.0, 2.0, 2012.0))	18	17.977026052947167
List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(4.1, 3.03, 39.0, 30.0026, 22.0, 22.0, 1.0, 1.0, 23.0, 8.0, 1.0, 2011.0))	22	22.015787070326525
List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(5.74, 7.575, 43.0, 11.0014, 28.0, 28.0, 1.0, 1.0, 22.0, 12.0, 2.0, 2012.0))	28	28.058417248633106
List(1, 21, List(), List(0.0, 0.0, 6.56, 6.06, 40.0, 31.0009, 4.0, 92.0, 96.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 18.0, 12.0, 2.0, 2012.0))	96	96.06176876485841
List(1, 21, List(), List(0.0, 0.0, 6.56, 6.82, 40.0, 22.0028, 4.0, 44.0, 48.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 18.0, 9.0, 1.0, 2011.0))	48	47.96614804695298
List(1, 21, List(), List(0.0, 0.0, 6.56, 6.82, 47.0, 19.0012, 5.0, 38.0, 43.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 2.0, 15.0, 1.0, 2012.0))	43	42.88936084849334
List(1, 21, List(), List(0.0, 0.0, 6.56, 6.82, 48.0, 26.0027, 1.0, 24.0, 25.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 3.0, 15.0, 1.0, 2012.0))	25	24.89586081959351
List(1, 21, List(), List(0.0, 0.0, 6.56, 9.85, 59.0, 6.0032, 2.0, 18.0, 20.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 15.0, 1.0, 2011.0))	20	19.91497309035597
List(0, 21, List(2, 3, 4, 5, 6, 7, 8, 9, 13, 18, 19, 20), List(6.56, 9.85, 69.0, 6.0032, 3.0, 27.0, 30.0, 1.0, 1.0, 12.0, 2.0, 2011.0))	30	29.944761523582343
List(0, 21, List(2, 3, 4, 7, 8, 9, 13, 17, 18, 19, 20), List(6.56, 11.365, 59.0, 1.0, 1.0, 1.0, 1.0, 5.0, 15.0, 1.0, 2011.0))	1	0.8497462836939462

In [25]:
```
# Parameter grid search for best parameters to give good predictions
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.regression import LinearRegression
from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit
# We use a ParamGridBuilder to construct a grid of parameters to search over.
# TrainValidationSplit will try all combinations of values and determine best model using
# the evaluator.
paramGrid = ParamGridBuilder()\
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .addGrid(lr.fitIntercept, [False, True])\
    .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])\
    .build()

# In this case the estimator is simply the linear regression.
# A TrainValidationSplit requires an Estimator, a set of Estimator ParamMaps, and an Evaluator.
tvs = TrainValidationSplit(estimator=lr,
                           estimatorParamMaps=paramGrid,
                           evaluator=RegressionEvaluator(),
                           # 80% of the data will be used for training, 20% for validation.
                           trainRatio=0.8)

# Run TrainValidationSplit, and choose the best set of parameters.
model = tvs.fit(train_output)

# Make predictions on test data. model is the model with combination of parameters
# that performed best.
display(model.transform(test_output)\
    .select("features", "label", "prediction")\
    .take(5))
```

features	label	prediction
List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(3.28, 4.545, 53.0, 12.998, 18.0, 18.0, 1.0, 1.0, 7.0, 12.0, 2.0, 2012.0))	18	17.99775672379881
List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(4.1, 3.03, 39.0, 30.0026, 22.0, 22.0, 1.0, 1.0, 23.0, 8.0, 1.0, 2011.0))	22	22.002205520528005
List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(5.74, 7.575, 43.0, 11.0014, 28.0, 28.0, 1.0, 1.0, 22.0, 12.0, 2.0, 2012.0))	28	28.002288892258296
List(1, 21, List(), List(0.0, 0.0, 6.56, 6.06, 40.0, 31.0009, 4.0, 92.0, 96.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 18.0, 12.0, 2.0, 2012.0))	96	95.99923333047171
List(1, 21, List(), List(0.0, 0.0, 6.56, 6.82, 40.0, 22.0028, 4.0, 44.0, 48.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 18.0, 9.0, 1.0, 2011.0))	48	48.00084264442458

In [26]:
```
# Random Forest Classifier model
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator
rf = RandomForestRegressor(labelCol="label", featuresCol="features", numTrees=100)
# Train model.  This also runs the indexers.
rf_model = rf.fit(train_output)
# rf_model.persist()
# Make predictions.
predictions = rf_model.transform(test_output)

# Select example rows to display.
display(predictions.select("prediction", "label", "features").take(5))

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

prediction	label	features
26.980856309621263	18	List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(3.28, 4.545, 53.0, 12.998, 18.0, 18.0, 1.0, 1.0, 7.0, 12.0, 2.0, 2012.0))
33.05357800429468	22	List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(4.1, 3.03, 39.0, 30.0026, 22.0, 22.0, 1.0, 1.0, 23.0, 8.0, 1.0, 2011.0))
36.28714272390532	28	List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(5.74, 7.575, 43.0, 11.0014, 28.0, 28.0, 1.0, 1.0, 22.0, 12.0, 2.0, 2012.0))
90.99638717240707	96	List(1, 21, List(), List(0.0, 0.0, 6.56, 6.06, 40.0, 31.0009, 4.0, 92.0, 96.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 18.0, 12.0, 2.0, 2012.0))
53.02938717787259	48	List(1, 21, List(), List(0.0, 0.0, 6.56, 6.82, 40.0, 22.0028, 4.0, 44.0, 48.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 18.0, 9.0, 1.0, 2011.0))

In [27]:
```
# GBT Regressor model
from pyspark.ml.regression import GBTRegressor
gbt = GBTRegressor(featuresCol="features", maxIter=10)

gbt_model = gbt.fit(train_output)
# Make predictions.
predictions = gbt_model.transform(test_output)


gbt_model.write().overwrite().save("bike_sharing_gbt.model")
# Select example rows to display.
display(predictions.select("prediction", "label", "features").take(5))

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

prediction	label	features
16.889233655915504	18	List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(3.28, 4.545, 53.0, 12.998, 18.0, 18.0, 1.0, 1.0, 7.0, 12.0, 2.0, 2012.0))
16.92511614166162	22	List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(4.1, 3.03, 39.0, 30.0026, 22.0, 22.0, 1.0, 1.0, 23.0, 8.0, 1.0, 2011.0))
30.32523546505164	28	List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(5.74, 7.575, 43.0, 11.0014, 28.0, 28.0, 1.0, 1.0, 22.0, 12.0, 2.0, 2012.0))
95.83936857912708	96	List(1, 21, List(), List(0.0, 0.0, 6.56, 6.06, 40.0, 31.0009, 4.0, 92.0, 96.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 18.0, 12.0, 2.0, 2012.0))
46.025330660611566	48	List(1, 21, List(), List(0.0, 0.0, 6.56, 6.82, 40.0, 22.0028, 4.0, 44.0, 48.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 18.0, 9.0, 1.0, 2011.0))