Spark_Linear_Regression

February 10, 2017

0.1 Generate Dummy Data

Generate data according to $y = x_1 + 2x_2 + 3x_3 + \xi$ (where ξ represents random noise such that $\xi \sim U[-10, 10]$)

```
In [1]: import scala.math._
        val sqlContext = org.apache.spark.sql.SQLContext.getOrCreate(sc)
        import sqlContext.implicits._


        // gen data according to y = x1 + 2*x2 + 3*x3 + random noise
        def genRanRow: (Double, Double, Double, Double) = {
            val (x1, x2, x3) = (100*random, 100*random, 100*random)
            (x1, x2, x3, x1 + 2*x2 + 3*x3 + (20*random - 10))
        }

        val myDF = sc.parallelize((0 to 10000).map(x => genRanRow)).toDF("x1", "x2'

        myDF.show
```

```
+------------------+------------------+------------------+------------------+
|                x1|                x2|                x3|                 y|
+------------------+------------------+------------------+------------------+
|45.313454186562815| 40.64844204181767| 70.38053786933851|  338.032829891297|
|29.271923155629764|26.458317702278666| 9.461358559389232|102.93713082550283|
| 87.46401504983163|24.015888193007086| 73.12760975971955| 355.6219246002725|
| 59.03232817833298|18.674466357522455| 59.57460662128722|266.81308640825546|
| 62.80822316119333| 62.24705357840472| 51.39587770088898| 333.3419489843179|
| 79.52719220537526| 95.69022289408964|  36.8885420535219|389.15854511423174|
| 44.29286905609108| 84.46967639231525|28.022439619776296| 303.7327855826478|
|19.995897905781813| 98.56854913056992| 51.29677669145447|378.40457203878213|
| 9.210201049943901| 34.82214209547628| 38.58696932719751|200.59918857976706|
| 2.030680789828432|23.110500887094755| 99.08819391641833|349.55477277934995|
| 50.38270634909791| 79.91283177858809|  10.5912000687845|245.31432314832657|
| 5.525215611660872| 72.82544865884553|28.579682116994565|  235.214819187385|
| 54.78564353079366|37.867615163214765| 75.01755718060478| 358.9619489170667|
| 46.00625606168802|47.747238351715815| 35.19831605580481|244.60897529827457|
| 93.74795305106784| 63.73110116196719| 60.65754834759981|403.36405360290445|
```

```
|19.988947888409015|16.361932755450603| 48.90363878463977| 191.1306012514847|
|  93.8721430097144|50.987269723220074|46.600576959911585| 329.0409022141387|
| 99.99646427406998|42.912750960257284| 1.840644969206362|183.97345558233462|
| 82.81379992802306|33.965112038376446| 73.27376843877421|  363.150298774122|
| 85.88456050701078| 67.34781435471429| 89.11410858055689|494.39065225313163|
+------------------+------------------+------------------+------------------+
only showing top 20 rows
```

0.2 Run Linear Regression

Here are the steps to running linear regression:

1. Split the data into training and testing sets

- Make a `VectorAssembler` which combines our feature columns into a single `features` column
- Make a `LinearRegression` object with label column `y`
- Put the `VectorAssembler` and `LinearRegression` objects into a single `Pipeline` object
- Fit the `Pipeline` to the training data. This runs the `VectorAssembler` and the `LinearRegression`
- Tranform the testing data with our `Pipeline` to make predictions. This runs the `VectorAssembler` and applies the learned `LinearRegression` method
- Extract the `LinearRegressionModel` from our `Pipeline` to check the Root Mean Square Error = $\sqrt{\frac{\sum_i^n (x_i - \hat{x_i})^2}{n}}$ and the coffificients x_1, x_2, and x_3

In [2]:
```
/* Import needed classes*/
import org.apache.spark.ml.feature.{VectorAssembler, StandardScaler}
import org.apache.spark.ml.regression.{LinearRegression, LinearRegressionMo
import org.apache.spark.ml.Pipeline

/* (1) Split the Data */
val Array(trainingData, testData) = myDF.randomSplit(Array(0.7, 0.3))

/* (2) Make your feature vector assembler */
val assembler = new VectorAssembler().
    setInputCols(Array("x1", "x2", "x3")).
    setOutputCol("features")

/* (3) Make your Linear Regression model */
val lr = new LinearRegression().
    setLabelCol("y"). // Output column name
    setFeaturesCol("features"). // Features column name
    setStandardization(true) // Standardize training data

/* (4) Put the assembler and regression model into a pipeline */
val pipeline = new Pipeline().setStages(Array(assembler, lr))
```

```
/* (5) Run the pipeline on your training data */
val model = pipeline.fit(trainingData)

/* (6) Make the predictions */
val predictions = model.transform(testData).persist

/* (7) Pull out the linear regression model from the pipeline, generate sur
val lrModel = model.stages(1).asInstanceOf[LinearRegressionModel]
val trainingSummary = lrModel.summary

/* (7.1) Print the Root Mean Square Error */
println(s"RMSE: ${trainingSummary.rootMeanSquaredError}")

/* (7.2) Print the Coefficients */
println(lrModel.coefficients)
```

```
RMSE: 5.771114729395559
[0.9984630123395857,1.9977193553047907,2.9995259790964734]
```

0.3 Conclusion

As we can see the model found the correct coefficients (within a small ε) with a corresponding error of 5.771 (this is irreduciable error due to ξ).

In fact, we can compute the exact irreducible error introduced by ξ by looking at the root of the mean. Thus the irreducible error, which we will call IE, can be calculated as $IE = \sqrt{\mathbb{E}\left(\xi^2\right)}$.

We know that $\xi \sim U[-10, 10]$ so we can compute $\mathbb{E}\left(\xi^2\right)$ using the general form for the expected value of powers of uniformly distributed random variables.

Expected Value of Powers of Uniformly Distributed Variables Given $X \sim U[a, b]$, the expected value of X^n can be calculated as $\mathbb{E}\left(X^n\right) = \dfrac{1}{n+1}\left(\dfrac{b^{n+1} - a^{n+1}}{b-a}\right)$

In our case $\xi \sim U[-10, 10]$ so $a = -10$, $b = 10$, and $n = 2$:

```
In [3]: val a = -10d
        val b = 10d
        val n = 2d

        val E = (1/(n+1))*(math.pow(b,n+1) - math.pow(a, n+1))/(b - a)
        println(E)
```

```
33.3333333333333
```

We now know the value of $\mathbb{E}\left(\xi^2\right)$, to finish our computation we just need to take the root:

```
In [4]: val IE = math.sqrt(E)
        println(IE)
```

```
5.773502691896257
```

Thus we can see that the RMSE of our model, 5.771, is pretty close the minimal irreducible error of 5.774.