# Homework Writeup

## Using Apache Spark

### //--Covtype Data Set--

/\*

Data set contains records of the type of forest covering parcels of land in Colorado, USA. Each example contains

information describing each parcel of land. 54 total features. 581,012 examples in the data set. Not super big.

\*/

//--Preparing the Data--

val dataWithoutHeaders = spark.read.

option("inferschema", true). //predict value type

option("header", false). //no header line

csv("/proj/cse398-498/course/AAS\_CH4/covtype.data")

//dataWithoutHeaders: org.apache.spark.sql.DataFrame = [\_c0: int, \_c1: int ... 53 more fields]

//since the first line is not a header line, only have \_c0 for column names and so on

dataWithoutHeaders.head

//res6: org.apache.spark.sql.Row = [2596,51,3,258,0,510,221,232,148,6279,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,..]

val colNames = Seq( //initial column names

"Elevation", "Aspect", "Slope",

"Horizontal\_Distance\_To\_Hydrology", "Vertical\_Distance\_To\_Hydrology",

"Horizontal\_Distance\_To\_Roadways",

"Hillshade\_9am", "Hillshade\_Noon", "Hillshade\_3pm",

"Horizontal\_Distance\_To\_Fire\_Points"

) ++ ( //++ concatenates collections together

(0 until 4).map(i => s"Wilderness\_Area\_$i") //cool trick to label each column differently

) ++ (

(0 until 40).map(i => s"Soil\_Type\_$i") //same incrementing trick here

) ++ Seq("Cover\_Type") //save sequence as

val data = dataWithoutHeaders.toDF(colNames: \_\*).

withColumn("Cover\_Type", $"Cover\_Type".cast("double")) //cast each column to double (MLlib API req) and save it under Cover\_Type sequence

data.take(5).foreach(println)

/\*

colNames: Seq[String] = List(Elevation, Aspect, Slope, Horizontal\_Distance\_To\_Hydrology, Vertical\_Distance\_To\_Hydrology,...)

[2596,51,3,258,0,510,221,232,148,6279,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,5.0]

[2590,56,2,212,-6,390,220,235,151,6225,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,5.0]

[2804,139,9,268,65,3180,234,238,135,6121,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2.0]

[2785,155,18,242,118,3090,238,238,122,6211,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,2.0]

[2595,45,2,153,-1,391,220,234,150,6172,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,5.0]

\*/

### 

### //--A First Decision Tree--

val Array(trainData, testData) = data.randomSplit(Array(0.9,0.1))

trainData.cache()

testData.cache()

//Spark MLlib requires all inputs to be vectors (arrays of doubles)

import org.apache.spark.ml.feature.VectorAssembler

val inputCols = trainData.columns.filter(\_ != "Cover\_Type") //all columns except the target are input features

val assembler = new VectorAssembler(). //Transformer

setInputCols(inputCols). //input columns

setOutputCol("featureVector") //output column

val assembledTrainData = assembler.transform(trainData) //transform the data

assembledTrainData.select("featureVector").show(truncate = false) //select the output column and don't truncate

/\*

assembledTrainData: org.apache.spark.sql.DataFrame = [Elevation: int, Aspect: int ... 54 more fields]

+-----------------------------------------------------------------------------------------------------+

|featureVector |

+-----------------------------------------------------------------------------------------------------+

|(54,[0,1,2,3,4,5,6,7,8,9,13,15],[1863.0,37.0,17.0,120.0,18.0,90.0,217.0,202.0,115.0,769.0,1.0,1.0]) |

|(54,[0,1,2,5,6,7,8,9,13,18],[1874.0,18.0,14.0,90.0,208.0,209.0,135.0,793.0,1.0,1.0]) |

|(54,[0,1,2,3,4,5,6,7,8,9,13,18],[1879.0,28.0,19.0,30.0,12.0,95.0,209.0,196.0,117.0,778.0,1.0,1.0]) |

|(54,[0,1,2,3,4,5,6,7,8,9,13,15],[1888.0,33.0,22.0,150.0,46.0,108.0,209.0,185.0,103.0,735.0,1.0,1.0]) |

|(54,[0,1,2,3,4,5,6,7,8,9,13,14],[1889.0,28.0,22.0,150.0,23.0,120.0,205.0,185.0,108.0,759.0,1.0,1.0]) |

|(54,[0,1,2,3,4,5,6,7,8,9,13,18],[1889.0,353.0,30.0,95.0,39.0,67.0,153.0,172.0,146.0,600.0,1.0,1.0]) |

|(54,[0,1,2,3,4,5,6,7,8,9,13,18],[1896.0,337.0,12.0,30.0,6.0,175.0,195.0,224.0,168.0,732.0,1.0,1.0]) |

|(54,[0,1,2,3,4,5,6,7,8,9,13,15],[1898.0,34.0,23.0,175.0,56.0,134.0,210.0,184.0,99.0,765.0,1.0,1.0]) |

|(54,[0,1,2,3,4,5,6,7,8,9,13,18],[1899.0,355.0,22.0,153.0,43.0,124.0,178.0,195.0,151.0,819.0,1.0,1.0])|

|(54,[0,1,2,3,4,5,6,7,8,9,13,14],[1901.0,311.0,9.0,30.0,2.0,190.0,195.0,234.0,179.0,726.0,1.0,1.0]) |

|(54,[0,1,2,3,4,5,6,7,8,9,13,14],[1903.0,5.0,13.0,42.0,4.0,201.0,203.0,214.0,148.0,708.0,1.0,1.0]) |

|(54,[0,1,2,3,4,5,6,7,8,9,13,16],[1903.0,67.0,16.0,108.0,36.0,120.0,234.0,207.0,100.0,969.0,1.0,1.0]) |

|(54,[0,1,2,3,4,5,6,7,8,9,13,14],[1904.0,51.0,26.0,67.0,30.0,162.0,222.0,175.0,72.0,711.0,1.0,1.0]) |

|(54,[0,1,2,3,4,5,6,7,8,9,13,18],[1905.0,19.0,27.0,134.0,58.0,120.0,188.0,171.0,108.0,636.0,1.0,1.0]) |

|(54,[0,1,2,3,4,5,6,7,8,9,13,14],[1905.0,33.0,27.0,90.0,46.0,150.0,204.0,171.0,89.0,725.0,1.0,1.0]) |

|(54,[0,1,2,3,4,5,6,7,8,9,13,15],[1906.0,356.0,20.0,150.0,55.0,120.0,184.0,201.0,151.0,726.0,1.0,1.0])|

|(54,[0,1,2,3,4,5,6,7,8,9,13,18],[1908.0,323.0,32.0,150.0,52.0,120.0,125.0,190.0,196.0,765.0,1.0,1.0])|

|(54,[0,1,2,3,4,5,6,7,8,9,13,18],[1916.0,320.0,24.0,190.0,60.0,162.0,151.0,210.0,195.0,832.0,1.0,1.0])|

|(54,[0,1,2,3,4,5,6,7,8,9,13,23],[1918.0,321.0,28.0,42.0,17.0,85.0,139.0,201.0,196.0,402.0,1.0,1.0]) |

|(54,[0,1,2,3,4,5,6,7,8,9,13,18],[1919.0,44.0,26.0,162.0,68.0,150.0,216.0,173.0,77.0,706.0,1.0,1.0]) |

+-----------------------------------------------------------------------------------------------------+

only showing top 20 rows

\*/

//the output looks different than normal because this is represennted as a SparseVector(stores nonzero values and indices)

import org.apache.spark.ml.classification.DecisionTreeClassifier

import scala.util.Random

val classifier = new DecisionTreeClassifier().

setSeed(Random.nextLong()). //use random seed

setLabelCol("Cover\_Type"). //label columns by Cover\_Type

setFeaturesCol("featureVector"). //label as featureVector

setPredictionCol("prediction") //self explanatory

val model = classifier.fit(assembledTrainData) //use classifier on assembledTrainData

println(model.toDebugString) //print model

/\*

import org.apache.spark.ml.classification.DecisionTreeClassifier

import scala.util.Random

classifier: org.apache.spark.ml.classification.DecisionTreeClassifier = dtc\_f89d84fc3a06

model: org.apache.spark.ml.classification.DecisionTreeClassificationModel = DecisionTreeClassificationModel (uid=dtc\_f89d84fc3a06) of depth 5 with 63 nodes

DecisionTreeClassificationModel (uid=dtc\_f89d84fc3a06) of depth 5 with 63 nodes

If (feature 0 <= 3034.0)

If (feature 0 <= 2476.0)

If (feature 3 <= 0.0)

If (feature 13 <= 0.0)

If (feature 17 <= 0.0)

Predict: 6.0

Else (feature 17 > 0.0)

Predict: 6.0

Else (feature 13 > 0.0)

If (feature 23 <= 0.0)

Predict: 4.0

Else (feature 23 > 0.0)

Predict: 3.0

Else (feature 3 > 0.0)

If (feature 16 <= 0.0)

If (feature 9 <= 577.0)

Predict: 3.0

Else (feature 9 > 577.0)

Predict: 3.0

Else (feature 16 > 0.0)

If (feature 9 <= 1279.0)

Predict: 3.0

Else (feature 9 > 1279.0)

Predict: 4.0

Else (feature 0 > 2476.0)

If (feature 17 <= 0.0)

If (feature 15 <= 0.0)

If (feature 0 <= 2924.0)

Predict: 2.0

Else (feature 0 > 2924.0)

Predict: 2.0

Else (feature 15 > 0.0)

If (feature 9 <= 1370.0)

Predict: 3.0

Else (feature 9 > 1370.0)

Predict: 3.0

Else (feature 17 > 0.0)

If (feature 0 <= 2694.0)

If (feature 0 <= 2639.0)

Predict: 3.0

Else (feature 0 > 2639.0)

Predict: 3.0

Else (feature 0 > 2694.0)

If (feature 5 <= 1200.0)

Predict: 5.0

Else (feature 5 > 1200.0)

Predict: 2.0

Else (feature 0 > 3034.0)

If (feature 0 <= 3313.0)

If (feature 7 <= 239.0)

If (feature 0 <= 3100.0)

If (feature 3 <= 162.0)

Predict: 1.0

Else (feature 3 > 162.0)

Predict: 2.0

Else (feature 0 > 3100.0)

If (feature 5 <= 1022.0)

Predict: 1.0

Else (feature 5 > 1022.0)

Predict: 1.0

Else (feature 7 > 239.0)

If (feature 3 <= 313.0)

If (feature 0 <= 3186.0)

Predict: 1.0

Else (feature 0 > 3186.0)

Predict: 1.0

Else (feature 3 > 313.0)

If (feature 0 <= 3208.0)

Predict: 2.0

Else (feature 0 > 3208.0)

Predict: 1.0

Else (feature 0 > 3313.0)

If (feature 12 <= 0.0)

If (feature 3 <= 277.0)

If (feature 6 <= 206.0)

Predict: 1.0

Else (feature 6 > 206.0)

Predict: 7.0

Else (feature 3 > 277.0)

If (feature 10 <= 0.0)

Predict: 1.0

Else (feature 10 > 0.0)

Predict: 1.0

Else (feature 12 > 0.0)

If (feature 45 <= 0.0)

If (feature 0 <= 3370.0)

Predict: 7.0

Else (feature 0 > 3370.0)

Predict: 7.0

Else (feature 45 > 0.0)

If (feature 5 <= 914.0)

Predict: 7.0

Else (feature 5 > 914.0)

Predict: 1.0

\*/

// "Here, for historical reasons, the features are only referred to by number, not name, unfortunately" - MAD DUMB AND STUPID AND DUMB >:(

model.featureImportances.toArray.zip(inputCols). //zip up a list of input columns based on its input feature contribution

sorted.reverse.foreach(println) //print it descending

/\*

(0.8128454136618612,Elevation)

(0.03861086575747634,Soil\_Type\_3)

(0.030084100586133485,Soil\_Type\_1)

(0.029800200066844286,Horizontal\_Distance\_To\_Hydrology)

(0.025883165249676293,Hillshade\_Noon)

(0.017179393227441293,Soil\_Type\_31)

(0.01479539080531332,Horizontal\_Distance\_To\_Roadways)

(0.011681578518113746,Wilderness\_Area\_2)

(0.006427023563056407,Horizontal\_Distance\_To\_Fire\_Points)

(0.003490393305697935,Soil\_Type\_2)

(0.0030626806129993253,Wilderness\_Area\_0)

(0.002706494899578449,Wilderness\_Area\_3)

(0.0025150989332858712,Hillshade\_9am)

(9.182008125223037E-4,Soil\_Type\_9)

(0.0,Wilderness\_Area\_1)

(0.0,Vertical\_Distance\_To\_Hydrology)

(0.0,Soil\_Type\_8)

(0.0,Soil\_Type\_7)

(0.0,Soil\_Type\_6)

(0.0,Soil\_Type\_5)

(0.0,Soil\_Type\_4)

(0.0,Soil\_Type\_39)

(0.0,Soil\_Type\_38)

(0.0,Soil\_Type\_37)

(0.0,Soil\_Type\_36)

(0.0,Soil\_Type\_35)

(0.0,Soil\_Type\_34)

(0.0,Soil\_Type\_33)

(0.0,Soil\_Type\_32)

(0.0,Soil\_Type\_30)

(0.0,Soil\_Type\_29)

(0.0,Soil\_Type\_28)

(0.0,Soil\_Type\_27)

(0.0,Soil\_Type\_26)

(0.0,Soil\_Type\_25)

(0.0,Soil\_Type\_24)

(0.0,Soil\_Type\_23)

(0.0,Soil\_Type\_22)

(0.0,Soil\_Type\_21)

(0.0,Soil\_Type\_20)

(0.0,Soil\_Type\_19)

(0.0,Soil\_Type\_18)

(0.0,Soil\_Type\_17)

(0.0,Soil\_Type\_16)

(0.0,Soil\_Type\_15)

(0.0,Soil\_Type\_14)

(0.0,Soil\_Type\_13)

(0.0,Soil\_Type\_12)

(0.0,Soil\_Type\_11)

(0.0,Soil\_Type\_10)

(0.0,Soil\_Type\_0)

(0.0,Slope)

(0.0,Hillshade\_3pm)

(0.0,Aspect)

\*/

//Most features (other than elevation) have no importance when predicting le cover type, which makes sense

//plant no care if soil best or not, just care if temp and water is good and matches, if not, lmao die

//The DecisionTreeClassificationModel we made is a transformer since we can transform a DF with vectors into a DF with predictions

val predictions = model.transform(assembledTrainData)

predictions.select("Cover\_Type", "prediction", "probability").

show(truncate = false)

/\*

+----------+----------+----------------------------------------------------------------------------------------------------------------+

|Cover\_Type|prediction|probability |

+----------+----------+----------------------------------------------------------------------------------------------------------------+

|6.0 |3.0 |[0.0,5.295768680824022E-5,0.05587035958269343,0.563257956892443,0.024731239739448182,0.0,0.3560874860986072,0.0]|

|6.0 |4.0 |[0.0,0.0,0.024926686217008796,0.24853372434017595,0.6356304985337243,0.0,0.09090909090909091,0.0] |

|6.0 |3.0 |[0.0,5.295768680824022E-5,0.05587035958269343,0.563257956892443,0.024731239739448182,0.0,0.3560874860986072,0.0]|

|6.0 |3.0 |[0.0,5.295768680824022E-5,0.05587035958269343,0.563257956892443,0.024731239739448182,0.0,0.3560874860986072,0.0]|

|6.0 |3.0 |[0.0,5.295768680824022E-5,0.05587035958269343,0.563257956892443,0.024731239739448182,0.0,0.3560874860986072,0.0]|

|6.0 |3.0 |[0.0,5.295768680824022E-5,0.05587035958269343,0.563257956892443,0.024731239739448182,0.0,0.3560874860986072,0.0]|

|6.0 |3.0 |[0.0,5.295768680824022E-5,0.05587035958269343,0.563257956892443,0.024731239739448182,0.0,0.3560874860986072,0.0]|

|6.0 |3.0 |[0.0,5.295768680824022E-5,0.05587035958269343,0.563257956892443,0.024731239739448182,0.0,0.3560874860986072,0.0]|

|6.0 |3.0 |[0.0,5.295768680824022E-5,0.05587035958269343,0.563257956892443,0.024731239739448182,0.0,0.3560874860986072,0.0]|

|6.0 |3.0 |[0.0,5.295768680824022E-5,0.05587035958269343,0.563257956892443,0.024731239739448182,0.0,0.3560874860986072,0.0]|

|6.0 |3.0 |[0.0,5.295768680824022E-5,0.05587035958269343,0.563257956892443,0.024731239739448182,0.0,0.3560874860986072,0.0]|

|3.0 |3.0 |[0.0,0.0,0.007609668755595345,0.6777081468218442,0.2493285586392122,0.0,0.06535362578334826,0.0] |

|6.0 |3.0 |[0.0,5.295768680824022E-5,0.05587035958269343,0.563257956892443,0.024731239739448182,0.0,0.3560874860986072,0.0]|

|6.0 |3.0 |[0.0,5.295768680824022E-5,0.05587035958269343,0.563257956892443,0.024731239739448182,0.0,0.3560874860986072,0.0]|

|6.0 |3.0 |[0.0,5.295768680824022E-5,0.05587035958269343,0.563257956892443,0.024731239739448182,0.0,0.3560874860986072,0.0]|

|3.0 |3.0 |[0.0,0.0,0.007609668755595345,0.6777081468218442,0.2493285586392122,0.0,0.06535362578334826,0.0] |

|6.0 |3.0 |[0.0,5.295768680824022E-5,0.05587035958269343,0.563257956892443,0.024731239739448182,0.0,0.3560874860986072,0.0]|

|6.0 |3.0 |[0.0,5.295768680824022E-5,0.05587035958269343,0.563257956892443,0.024731239739448182,0.0,0.3560874860986072,0.0]|

|6.0 |3.0 |[0.0,5.295768680824022E-5,0.05587035958269343,0.563257956892443,0.024731239739448182,0.0,0.3560874860986072,0.0]|

|6.0 |3.0 |[0.0,5.295768680824022E-5,0.05587035958269343,0.563257956892443,0.024731239739448182,0.0,0.3560874860986072,0.0]|

+----------+----------+----------------------------------------------------------------------------------------------------------------+

only showing top 20 rows

\*/

//probability is a vector (aka chance for each cover type) [ignore the first index, 0, since it is always 0]

import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

val evaluator = new MulticlassClassificationEvaluator().

setLabelCol("Cover\_Type").

setPredictionCol("prediction")

evaluator.setMetricName("accuracy").evaluate(predictions)

//res31: Double = 0.7025081046797959 //accuracy of classifier

evaluator.setMetricName("f1").evaluate(predictions)

//res32: Double = 0.6856366486945126 //The F1 score is the harmonic mean of the precision and recall

import org.apache.spark.mllib.evaluation.MulticlassMetrics

val predictionRDD = predictions.

select("prediction", "Cover\_Type").

as[(Double, Double)]. //convert to dataset

rdd //convert to RDD

val multiclassMetrics = new MulticlassMetrics(predictionRDD)

multiclassMetrics.confusionMatrix

/\*

122203.0 63458.0 81.0 0.0 41.0 3.0 5121.0

44925.0 206294.0 2783.0 34.0 341.0 38.0 745.0

0.0 4684.0 26996.0 354.0 38.0 102.0 0.0

0.0 10.0 1311.0 1133.0 0.0 0.0 0.0

0.0 7864.0 274.0 0.0 419.0 0.0 0.0

0.0 5058.0 10069.0 124.0 9.0 412.0 0.0

8181.0 78.0 0.0 0.0 0.0 0.0 10280.0

\*/

//the correct predictions are the counts along the diagonal and the predictions are everything else

#### //how to do confusion matrix using dataframe api

val confusionMatrix = predictions.

groupBy("Cover\_Type").

pivot("prediction", (1 to 7)).

count().

na.fill(0.0). //replace null with 0.0

orderBy("Cover\_Type")

confusionMatrix.show()

/\*

+----------+------+------+-----+----+---+---+-----+

|Cover\_Type| 1| 2| 3| 4| 5| 6| 7|

+----------+------+------+-----+----+---+---+-----+

| 1.0|122203| 63458| 81| 0| 41| 3| 5121|

| 2.0| 44925|206294| 2783| 34|341| 38| 745|

| 3.0| 0| 4684|26996| 354| 38|102| 0|

| 4.0| 0| 10| 1311|1133| 0| 0| 0|

| 5.0| 0| 7864| 274| 0|419| 0| 0|

| 6.0| 0| 5058|10069| 124| 9|412| 0|

| 7.0| 8181| 78| 0| 0| 0| 0|10280|

+----------+------+------+-----+----+---+---+-----+

\*/

//pretty asf!

#### //Create a random classifier based on cover\_type probability distribution

import org.apache.spark.sql.DataFrame

def classProbabilities(data: DataFrame): Array[Double] = { //return an array double

val total = data.count()

data.groupBy("Cover\_Type").count(). //count by the category

orderBy("Cover\_Type"). //order by category

select("count").as[Double]. //convert to data set

map(\_ / total).

collect()

}

val trainPriorProbabilities = classProbabilities(trainData)

val testPriorProbabilities = classProbabilities(testData)

trainPriorProbabilities.zip(testPriorProbabilities).map { //Sum products of pairs in training test set

case (trainProb, cvProb) => trainProb \* cvProb

//where does cvProb comes from???

}.sum

//res36: Double = 0.3772220889658342

### //--Decision Tree Hyperparameters--

/\*

Moving forward, we will no longer us AUC to measure. Now we will be using multiclass accuracy.

Hyperparameters:

1. Maximum Depth - limits the numbers of levels in decision tree (usefull to prevent overfitting)

2. Maximum Bins - Bins are really a set of values to plug in to the decision tree, more bins, more optiomal decision rule

3. Impurtiy Measure - It is the probability that a randomly chosen classification of randmonl;yu chosen example is incorrect

I\_g(p) = 1 - N(E)i=1[p^2\_i]

If only one class, the value is 0, since the subset is pure

4. Entropy - "captures how much uncertainty the collection of target values in the subset implies about

predictions for data that falls in that subset"

5. Minimum Information Gain - imposes a min infro gain, or decrease in impurity for the specified decision rules

If the ruleset doesn't improve impurity, it gets rejected

Lowers overfitting (if doesn't divide the training data, the rule not gonna do much in real world)

\*/

### //--Tuning Decision Trees--

//We are setting up a pipeline that does both steps above (vector assemble and decision tree classifier)

import org.apache.spark.ml.Pipeline

val inputCols = trainData.columns.filter(\_ != "Cover\_Type") //training data is made into input by taking out covertype columns

val assembler = new VectorAssembler(). //create assembler

setInputCols(inputCols). //using input cols

setOutputCol("featureVector") //save output columns as a featureVector

val classifier = new DecisionTreeClassifier().

setSeed(Random.nextLong()). //set as a random seed

setLabelCol("Cover\_Type").

setFeaturesCol("featureVector").

setPredictionCol("prediction")

val pipeline = new Pipeline().setStages(Array(assembler, classifier)) //pipeline is multiprocess, use Array to order stages

/\*

inputCols: Array[String] = Array(Elevation, Aspect, Slope, Horizontal\_Distance\_To\_Hydrology, Vertical\_Distance\_To\_Hydrology, ...

classifier: org.apache.spark.ml.classification.DecisionTreeClassifier = dtc\_0d2c61bd71f9

pipeline: org.apache.spark.ml.Pipeline = pipeline\_656d68e09702

import org.apache.spark.ml.tuning.ParamGridBuilder

\*/

import org.apache.spark.ml.tuning.ParamGridBuilder //to test hyperparameters

val paramGrid = new ParamGridBuilder().

addGrid(classifier.impurity, Seq("gini", "entropy")). //use both impurity measures

addGrid(classifier.maxDepth, Seq(1, 20)). //depth of 1 to 20

addGrid(classifier.maxBins, Seq(40, 300)). //40 to 300 values for rules

addGrid(classifier.minInfoGain, Seq(0.0, 0.05)). //ranges from no changes to impurity to improve impurity by 0.05

build() //this builds 16 different models

val multiclassEval = new MulticlassClassificationEvaluator().

setLabelCol("Cover\_Type").

setPredictionCol("prediction").

setMetricName("accuracy") //kinda self explanotary what happening here

/\*

paramGrid: Array[org.apache.spark.ml.param.ParamMap] =

Array({

dtc\_9f33733f0207-impurity: gini,

dtc\_9f33733f0207-maxBins: 40,

dtc\_9f33733f0207-maxDepth: 1,

dtc\_9f33733f0207-minInfoGain: 0.0

}, {

dtc\_9f33733f0207-impurity: gini,

dtc\_9f33733f0207-maxBins: 300,

dtc\_9f33733f0207-maxDepth: 1,

dtc\_9f33733f0207-minInfoGain: 0.0

}, {

dtc\_9f33733f0207-impurity: entropy,

dtc\_9f33733f0207-maxBins: 40,

dtc\_9f33733f0207-maxDepth: 1,

dtc\_9f33733f0207-minInfoGain: 0.0

}, {

dtc\_9f33733f0207-impurity: entropy,

dtc\_9f33733f0207-maxBins: 300,

dtc\_9f33733f0207-maxDepth: 1,

dtc\_9f33733f0207-minInfoGain: 0.0

}, {

dtc\_9f33733f0207-impurity: gini,

dtc\_9f33733f0207-maxBins: 40,

dtc\_9f33733f0207-maxDepth: 20,

dtc\_9f33733f0207-minInfoGain: 0.0

}, {

dtc\_9f33733f0207-impurity: gini,

dtc\_9f3373...multiclassEval: org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator = mcEval\_c1532b2b620e

\*/

import org.apache.spark.ml.tuning.TrainValidationSplit

//can use CrossValidator here, but will choose k times more expensive & doesn't add much

val validator = new TrainValidationSplit().

setSeed(Random.nextLong()).

setEstimator(pipeline).

setEvaluator(multiclassEval).

setEstimatorParamMaps(paramGrid). //HYPER PARAMETERS CAN STILL OVER FIT

setTrainRatio(0.9) //take another 10 percent and set it aside

//the left out 10% is used as a crossvalidation set (evaluate parameters that fit to training set)

//the original 10% left out to evaluate hyperparameters that fit the CV^^ (examples that arent in CV but has not been trained on [real world data])

val validatorModel = validator.fit(trainData) //returns best overall pipeline

/\*import org.apache.spark.ml.tuning.TrainValidationSplit

validator: org.apache.spark.ml.tuning.TrainValidationSplit = tvs\_2ecb5852718b

21/09/16 12:40:08 WARN RandomForest: Tree learning is using approximately 268474176 bytes per iteration, which exceeds requested limit maxMemoryUsage=268435456. This allows splitting 1647 nodes in this iteration.

21/09/16 12:40:12 WARN RandomForest: Tree learning is using approximately 268474176 bytes per iteration, which exceeds requested limit maxMemoryUsage=268435456. This allows splitting 1647 nodes in this iteration.

21/09/16 12:40:16 WARN RandomForest: Tree learning is using approximately 268474176 bytes per iteration, which exceeds requested limit maxMemoryUsage=268435456. This allows splitting 1647 nodes in this iteration.

21/09/16 12:40:20 WARN RandomForest: Tree learning is using approximately 268474176 bytes per iteration, which exceeds requested limit maxMemoryUsage=268435456. This allows splitting 1647 nodes in this iteration.

21/09/16 12:41:05 WARN RandomForest: Tree learning is using approximately 268474176 bytes per iteration, which exceeds requested limit maxMemoryUsage=268435456. This allows splitting 1647 nodes in this iteration.

21/09/16 12:41:09 WARN RandomForest: Tree learning is using approximately 268474176 bytes per iteration, which exceeds requested limit maxMemoryUsage=268435456. This allows splitting 1647 nodes in this iteration.

21/09/16 12:41:13 WARN RandomForest: Tree learning is using approximately 268474176 bytes per iteration, which exceeds requested limit maxMemoryUsage=268435456. This allows splitting 1647 nodes in this iteration.

validatorModel: org.apache.spark.ml.tuning.TrainValidationSplitModel = tvs\_2ecb5852718b

\*/

import org.apache.spark.ml.PipelineModel

val bestModel = validatorModel.bestModel //extract best model

bestModel.asInstanceOf[PipelineModel].stages.last.extractParamMap //extract parameters

/\*

scala> bestModel.asInstanceOf[PipelineModel].stages.last.extractParamMap

res87: org.apache.spark.ml.param.ParamMap =

{

dtc\_d529a29ca28a-cacheNodeIds: false,

dtc\_d529a29ca28a-checkpointInterval: 10,

dtc\_d529a29ca28a-featuresCol: featureVector,

dtc\_d529a29ca28a-impurity: entropy,

//entropy was better than impurity

dtc\_d529a29ca28a-labelCol: Cover\_Type,

dtc\_d529a29ca28a-maxBins: 40,

//40 bins was more than enough, 300 is too much lol

dtc\_d529a29ca28a-maxDepth: 20,

//WOW WHO KNEW MORE LEAVES DO BETTER? (1 leaf vs 20)

dtc\_d529a29ca28a-maxMemoryInMB: 256,

dtc\_d529a29ca28a-minInfoGain: 0.0,

dtc\_d529a29ca28a-minInstancesPerNode: 1,

dtc\_d529a29ca28a-predictionCol: prediction,

dtc\_d529a29ca28a-probabilityCol: probability,

dtc\_d529a29ca28a-rawPredictionCol: rawPrediction,

dtc\_d529a29ca28a-seed: -6060068181667196680

}

\*/

#### //Say we want to see the results and score of each hyperparameters

val validatorModel = validator.fit(trainData) //redefine for assurance

val paramAndMetrics = validatorModel.validationMetrics. //accessing the the models metrics

zip(validatorModel.getEstimatorParamMaps).sortBy(-\_.\_1) //zip up all models params, sort by descending

paramsAndMetrics.foreach { case (metric, params) =>

println(metric)

println(params)

println()

}

/\*

0.9117241114902427

{

dtc\_d529a29ca28a-impurity: entropy,

dtc\_d529a29ca28a-maxBins: 40,

dtc\_d529a29ca28a-maxDepth: 20,

dtc\_d529a29ca28a-minInfoGain: 0.0

}

0.9095579496223594

{

dtc\_d529a29ca28a-impurity: entropy,

dtc\_d529a29ca28a-maxBins: 300,

dtc\_d529a29ca28a-maxDepth: 20,

dtc\_d529a29ca28a-minInfoGain: 0.0

}

0.9067975309588621

{

dtc\_d529a29ca28a-impurity: gini,

dtc\_d529a29ca28a-maxBins: 300,

dtc\_d529a29ca28a-maxDepth: 20,

dtc\_d529a29ca28a-minInfoGain: 0.0

}

0.9044971820726143

{

dtc\_d529a29ca28a-impurity: gini,

dtc\_d529a29ca28a-maxBins: 40,

dtc\_d529a29ca28a-maxDepth: 20,

dtc\_d529a29ca28a-minInfoGain: 0.0

}

0.738124448874746

{

dtc\_d529a29ca28a-impurity: entropy,

dtc\_d529a29ca28a-maxBins: 40,

dtc\_d529a29ca28a-maxDepth: 20,

dtc\_d529a29ca28a-minInfoGain: 0.05

}

0.7280795920714642

{

dtc\_d529a29ca28a-impurity: entropy,

dtc\_d529a29ca28a-maxBins: 300,

dtc\_d529a29ca28a-maxDepth: 20,

dtc\_d529a29ca28a-minInfoGain: 0.05

}

0.6732737798566116

{

dtc\_d529a29ca28a-impurity: gini,

dtc\_d529a29ca28a-maxBins: 300,

dtc\_d529a29ca28a-maxDepth: 20,

dtc\_d529a29ca28a-minInfoGain: 0.05

}

0.6722386228578001

{

dtc\_d529a29ca28a-impurity: gini,

dtc\_d529a29ca28a-maxBins: 40,

dtc\_d529a29ca28a-maxDepth: 20,

dtc\_d529a29ca28a-minInfoGain: 0.05

}

0.6384426638040103

{

dtc\_d529a29ca28a-impurity: gini,

dtc\_d529a29ca28a-maxBins: 300,

dtc\_d529a29ca28a-maxDepth: 1,

dtc\_d529a29ca28a-minInfoGain: 0.0

}

0.6384426638040103

{

dtc\_d529a29ca28a-impurity: gini,

dtc\_d529a29ca28a-maxBins: 300,

dtc\_d529a29ca28a-maxDepth: 1,

dtc\_d529a29ca28a-minInfoGain: 0.05

}

0.6379059157305524

{

dtc\_d529a29ca28a-impurity: gini,

dtc\_d529a29ca28a-maxBins: 40,

dtc\_d529a29ca28a-maxDepth: 1,

dtc\_d529a29ca28a-minInfoGain: 0.0

}

0.6379059157305524

{

dtc\_d529a29ca28a-impurity: gini,

dtc\_d529a29ca28a-maxBins: 40,

dtc\_d529a29ca28a-maxDepth: 1,

dtc\_d529a29ca28a-minInfoGain: 0.05

}

0.48834489897634475

{

dtc\_d529a29ca28a-impurity: entropy,

dtc\_d529a29ca28a-maxBins: 40,

dtc\_d529a29ca28a-maxDepth: 1,

dtc\_d529a29ca28a-minInfoGain: 0.0

}

0.48834489897634475

{

dtc\_d529a29ca28a-impurity: entropy,

dtc\_d529a29ca28a-maxBins: 40,

dtc\_d529a29ca28a-maxDepth: 1,

dtc\_d529a29ca28a-minInfoGain: 0.05

}

0.48834489897634475

{

dtc\_d529a29ca28a-impurity: entropy,

dtc\_d529a29ca28a-maxBins: 300,

dtc\_d529a29ca28a-maxDepth: 1,

dtc\_d529a29ca28a-minInfoGain: 0.0

}

0.48834489897634475

{

dtc\_d529a29ca28a-impurity: entropy,

dtc\_d529a29ca28a-maxBins: 300,

//LMAO WHAT 300 bins and 1 leaf... that is erm, definitely an interesting 1R

dtc\_d529a29ca28a-maxDepth: 1,

dtc\_d529a29ca28a-minInfoGain: 0.05

}

\*/

//it is kinda self explanatory what happens to the score as you read the parameters

//-How well did the data do on the cv and test-

validatorModel.validationMetrics.max

//res90: Double = 0.9117241114902427

multiclassEval.evaluate(bestModel.transform(testData))

//res91: Double = 0.9145611417005437

/\*

IN THIS CASE, NUMBERS ARE CLOSE AND MATCH, NOT ALWAYS THE CASE!!!

Tree made may be so good and fit perfectly for specific model, but overfits when compared to real stuff

A good way to double check is to test on CV for model evaluation and then test best model on an unused set(neither for training nor evaluation)

\*/

multiclassEval.evaluate(bestModel.transform(trainData))

//res94: Double = 0.9501868137964811

//The model seem to be a bit overfitted, better for data then real world, a lower maximum depth may help

### //--Categorical Features Revisited--

/\*

As of now, all features have been thought as numerical, even when severy categories are binary. This isn't a issue, since 0 and 1, values for

yes and no, are between 0 and 1, values being compared to.

This isn't that great since it forces the model to look at each underlying categorical feature individually

QUOTE FROM BOOK - PLEASE NO PUNISH - TOO GOOD OF AN EXPLANATION:

"For example, nine different soil types are actually part of the Leighcan family, and they may be related in ways that the decision tree can

exploit. If soil type were encoded as a single categorical feature with 40 soil values, then the tree could express rules like “if the soil

type is one of the nine Leighton family types” directly. However, when encoded as 40 features, the tree would have to learn a sequence of

nine decisions on soil type to do the same, this expressiveness may lead to better decisions and more efficient trees."

Whereas right nowe we have 40 features represented by 40 one valued columns, which increase memory usage and slooooows the program down

\*/

import org.apache.spark.sql.functions.\_ //LIT WE MAKING FUNCTIONS

import org.apache.spark.ml.{PipelineModel, Pipeline}

import org.apache.spark.ml.classification.{DecisionTreeClassifier,

RandomForestClassifier, RandomForestClassificationModel}

import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

import org.apache.spark.ml.feature.{VectorAssembler, VectorIndexer}

import org.apache.spark.ml.linalg.Vector

import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}

import org.apache.spark.mllib.evaluation.MulticlassMetrics

import org.apache.spark.sql.{DataFrame, SparkSession}

import scala.util.Random

import spark.implicits.\_

#### //LOAD p1.SCALA BEFORE RUNNING BELOW!@!

val data = dataWithoutHeaders.toDF(colNames: \_\*).

withColumn("Cover\_Type", $"Cover\_Type".cast("double")) //cast each column to double (MLlib API req) and save it under Cover\_Type sequence

val Array(trainData, testData) = data.randomSplit(Array(0.9,0.1))

trainData.cache()

testData.cache()

//-Lets undoing this one-hot encoding, by taking 4 columns encoding wilderness type into one, with values ranging from 0 to 3-

def unencodeOneHot(data: DataFrame): DataFrame = {

val wildernessCols = (0 until 4).map(i => s"Wilderness\_Area\_$i").toArray

val wildernessAssembler = new VectorAssembler().

setInputCols(wildernessCols).

setOutputCol("wilderness")

val unhotUDF = udf((vec: Vector) => vec.toArray.indexOf(1.0).toDouble)

val withWilderness = wildernessAssembler.transform(data).

drop(wildernessCols:\_\*).

withColumn("wilderness", unhotUDF($"wilderness"))

val soilCols = (0 until 40).map(i => s"Soil\_Type\_$i").toArray

val soilAssembler = new VectorAssembler().

setInputCols(soilCols).

setOutputCol("soil")

soilAssembler.transform(withWilderness).

drop(soilCols:\_\*).

withColumn("soil", unhotUDF($"soil"))

}

val unencTrainData = unencodeOneHot(trainData)

val unencTestData = unencodeOneHot(testData)

/\*

THIS DOESN'T MAKE THE TWO NEW COLUMNS INDICATE THAT THEY ARE SPECIAL. WE NEED TO HAVE SPARK INTERPRET IT DIFF OR ELSE FAILURE

Spark can add metadata to each column. In order to do so, we need to put data through VectorIndexer(turn input into properly label

categorical feature columns. HAVE TO ADD THIS STAGE TO THE PIPELINE.

\*/

val inputCols = unencTrainData.columns.filter(\_ != "Cover\_Type")

val assembler = new VectorAssembler().

setInputCols(inputCols).

setOutputCol("featureVector")

val indexer = new VectorIndexer().

setMaxCategories(40). //set to 40 because soil has 40 values

setInputCol("featureVector").

setOutputCol("indexedVector")

//this works because all values from 0-40 is present, but in scarce data set, may need to use VectorIndexerModel to manually map

val classifier = new DecisionTreeClassifier().

setSeed(Random.nextLong()).

setLabelCol("Cover\_Type").

setFeaturesCol("indexedVector").

setPredictionCol("prediction")

val pipeline = new Pipeline().setStages(Array(assembler, indexer, classifier))

#### //-BELOW THIS IS TO TEST THE PIPELINE-

val paramGrid = new ParamGridBuilder().

addGrid(classifier.impurity, Seq("gini", "entropy")). //use both impurity measures

addGrid(classifier.maxDepth, Seq(1, 20)). //depth of 1 to 20

addGrid(classifier.maxBins, Seq(40, 300)). //40 to 300 values for rules

addGrid(classifier.minInfoGain, Seq(0.0, 0.05)). //ranges from no changes to impurity to improve impurity by 0.05

build() //this builds 16 different models

val multiclassEval = new MulticlassClassificationEvaluator().

setLabelCol("Cover\_Type").

setPredictionCol("prediction").

setMetricName("accuracy") //kinda self explanotary what happening here

import org.apache.spark.ml.tuning.TrainValidationSplit

//can use CrossValidator here, but will choose k times more expensive & doesn't add much

val validator = new TrainValidationSplit().

setSeed(Random.nextLong()).

setEstimator(pipeline).

setEvaluator(multiclassEval).

setEstimatorParamMaps(paramGrid). //HYPER PARAMETERS CAN STILL OVER FIT

setTrainRatio(0.9) //take another 10 percent and set it aside

//the left out 10% is used as a crossvalidation set (evaluate parameters that fit to training set)

//the original 10% left out to evaluate hyperparameters that fit the CV^^ (examples that aren't in CV but has not been trained on [real world data])

val validatorModel = validator.fit(unencTrainData) //returns best overall pipeline

val bestModel = validatorModel.bestModel

println(bestModel.asInstanceOf[PipelineModel].stages.last.extractParamMap)

val testAccuracy = multiclassEval.evaluate(bestModel.transform(unencTestData))

/\*

validatorModel: org.apache.spark.ml.tuning.TrainValidationSplitModel = tvs\_57824b5a24d2

bestModel: org.apache.spark.ml.Model[\_] = pipeline\_4f8064e18313

{

dtc\_6bfd33904b24-cacheNodeIds: false,

dtc\_6bfd33904b24-checkpointInterval: 10,

dtc\_6bfd33904b24-featuresCol: indexedVector,

dtc\_6bfd33904b24-impurity: gini,

dtc\_6bfd33904b24-labelCol: Cover\_Type,

dtc\_6bfd33904b24-maxBins: 300,

dtc\_6bfd33904b24-maxDepth: 20,

dtc\_6bfd33904b24-maxMemoryInMB: 256,

dtc\_6bfd33904b24-minInfoGain: 0.0,

dtc\_6bfd33904b24-minInstancesPerNode: 1,

dtc\_6bfd33904b24-predictionCol: prediction,

dtc\_6bfd33904b24-probabilityCol: probability,

dtc\_6bfd33904b24-rawPredictionCol: rawPrediction,

dtc\_6bfd33904b24-seed: -4035647379714359620

}

testAccuracy: Double = 0.9245599764628512

\*/

### //--Random Decision Tree Forests--

/\*

The reason our outputs don't match with the book is due to the randomness. Can't possibly explore every decision rule at every level.

DTrees use several heuristics to decide; only a few features are randomly analyzed at time (new tree er time); trade a bit of accuracy for speed

THE RANDOMNESS IN PROCESS creates independence between each tree (how many taxis are there and the group effect)

Trees that are auto generated to fit noise (OVERFITTING) will be on average voted out.

Making forest is easy, just change classifier

\*/

#### /\*HERE IS THE NEWWWWWWWWWW CODE\*/

import org.apache.spark.ml.classification.RandomForestClassifier

val classifier = new RandomForestClassifier().

setSeed(Random.nextLong()).

setLabelCol("Cover\_Type").

setFeaturesCol("indexedVector").

setPredictionCol("prediction")

import org.apache.spark.ml.Pipeline

#### //-The code below is to test the resulting decision tree forests-

import org.apache.spark.ml.feature.VectorIndexer

val inputCols = unencTrainData.columns.filter(\_ != "Cover\_Type") //all columns except the target are input features

val assembler = new VectorAssembler().

setInputCols(inputCols).

setOutputCol("featureVector")

val indexer = new VectorIndexer().

setMaxCategories(40). //set to 40 because soil has 40 values

setInputCol("featureVector").

setOutputCol("indexedVector")

//this works because all values form 0-40 is present, but in scarce data set, may need to use VectorIndexerModel to manually map

val pipeline = new Pipeline().setStages(Array(assembler, indexer, classifier))

#### //-BELOW THIS IS TO TEST THE PIPELINE-

import org.apache.spark.ml.tuning.ParamGridBuilder //to test hyperparameters

val paramGrid = new ParamGridBuilder().

addGrid(classifier.impurity, Seq("entropy")).

addGrid(classifier.maxDepth, Seq(20)).

addGrid(classifier.maxBins, Seq(40)).

addGrid(classifier.minInfoGain, Seq(0.0)).

build()

val multiclassEval = new MulticlassClassificationEvaluator().

setLabelCol("Cover\_Type").

setPredictionCol("prediction").

setMetricName("accuracy") //kinda self explanotary what happening here

import org.apache.spark.ml.tuning.TrainValidationSplit

//can use CrossValidator here, but will cost k times more expensive & doesn't add much

val validator = new TrainValidationSplit().

setSeed(Random.nextLong()).

setEstimator(pipeline).

setEvaluator(multiclassEval).

setEstimatorParamMaps(paramGrid). //HYPER PARAMETERS CAN STILL OVER FIT

setTrainRatio(0.9) //take another 10 percent and set it aside

//the left out 10% is used as a cross validation set (evaluate parameters that fit to training set)

//the original 10% left out to evaluate hyperparameters that fit the CV^^ (examples that aren't in CV but has not been trained on [real world data])

val validatorModel = validator.fit(unencTrainData) //returns best overall pipeline

val bestModel = validatorModel.bestModel

println(bestModel.asInstanceOf[PipelineModel].stages.last.extractParamMap)

val testAccuracy = multiclassEval.evaluate(bestModel.transform(unencTestData))

/\*

bestModel: org.apache.spark.ml.Model[\_] = pipeline\_e8dfaf103521

{

rfc\_afd6081cae0e-cacheNodeIds: false,

rfc\_afd6081cae0e-checkpointInterval: 10,

rfc\_afd6081cae0e-featureSubsetStrategy: auto,

rfc\_afd6081cae0e-featuresCol: indexedVector,

rfc\_afd6081cae0e-impurity: entropy,

rfc\_afd6081cae0e-labelCol: Cover\_Type,

rfc\_afd6081cae0e-maxBins: 40,

rfc\_afd6081cae0e-maxDepth: 20,

rfc\_afd6081cae0e-maxMemoryInMB: 256,

rfc\_afd6081cae0e-minInfoGain: 0.0,

rfc\_afd6081cae0e-minInstancesPerNode: 1,

rfc\_afd6081cae0e-numTrees: 20,

rfc\_afd6081cae0e-predictionCol: prediction,

rfc\_afd6081cae0e-probabilityCol: probability,

rfc\_afd6081cae0e-rawPredictionCol: rawPrediction,

rfc\_afd6081cae0e-seed: 8386058293792309184,

rfc\_afd6081cae0e-subsamplingRate: 1.0

}

testAccuracy: Double = 0.9522107081174439 //THATS A REAAALY GOOD ACCURACY!!!

\*/

/\*

I ran the code with training data and got:

val trainAccuracy = multiclassEval.evaluate(bestModel.transform(unencTrainData))

trainAccuracy: Double = 0.983749177996299

The model is overfitting still. As mentioned before, might be useful to decrease the length of the depth

\*/

import org.apache.spark.ml.classification.RandomForestClassificationModel

val forestModel = bestModel.asInstanceOf[PipelineModel].

stages.last.asInstanceOf[RandomForestClassificationModel]

forestModel.featureImportances.toArray.zip(inputCols).

sorted.reverse.foreach(println)

/\*

(0.2926106864847985,Elevation)

(0.18835406389537518,soil)

(0.10707829602517423,Horizontal\_Distance\_To\_Roadways)

(0.10122135445840696,Horizontal\_Distance\_To\_Fire\_Points)

(0.06818651488923709,wilderness)

(0.05168345030406425,Horizontal\_Distance\_To\_Hydrology)

(0.04499563182149587,Vertical\_Distance\_To\_Hydrology)

(0.03328427707076898,Hillshade\_Noon)

(0.03107486265165606,Aspect)

(0.030510493101422835,Hillshade\_9am)

(0.02618167199051451,Hillshade\_3pm)

(0.024818697307085434,Slope)

\*/

## //--Making Predictions--

bestModel.transform(unencTestData.drop("Cover\_Type")).select("prediction").show()) //PREDICTED VALUES

/\*

+----------+

|prediction|

+----------+

| 6.0|

| 6.0|

| 6.0|

| 6.0|

| 6.0|

| 3.0|

| 6.0|

| 6.0|

| 3.0|

| 3.0| //wrong

| 6.0|

| 4.0|

| 6.0|

| 3.0| //wrong

| 4.0|

| 3.0|

| 3.0|

| 4.0|

| 6.0|

| 6.0| //wrong

+----------+

\*/

//out of the 20 shown, 17 was correct (0.85)

testData.select("Cover\_Type").show() //ACTUAL VALUES

/\*

+----------+

|Cover\_Type|

+----------+

| 6.0|

| 6.0|

| 6.0|

| 6.0|

| 6.0|

| 3.0|

| 6.0|

| 6.0|

| 3.0|

| 6.0|

| 6.0|

| 4.0|

| 6.0|

| 4.0|

| 4.0|

| 3.0|

| 3.0|

| 4.0|

| 6.0|

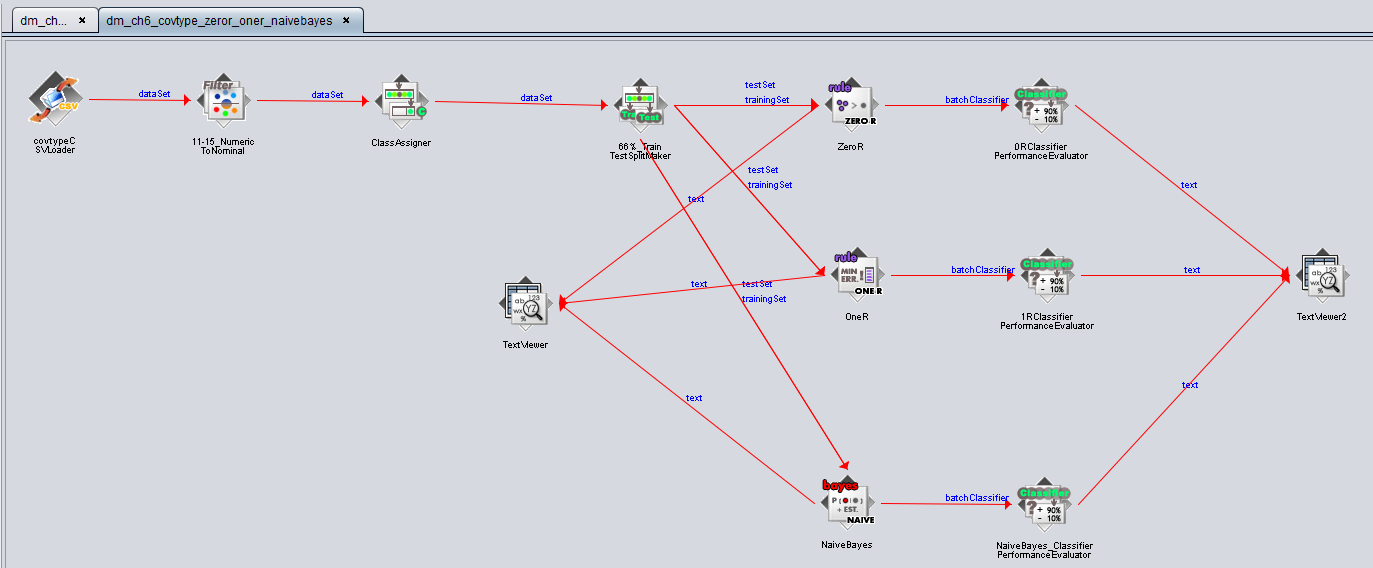
| 4.0|

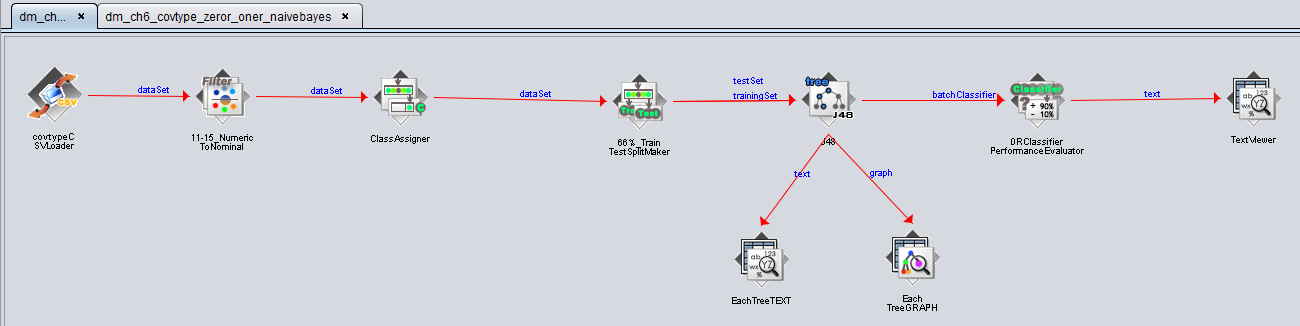
+----------+

\*/

## Using Weka Explorer

### Layout





### ZeroR

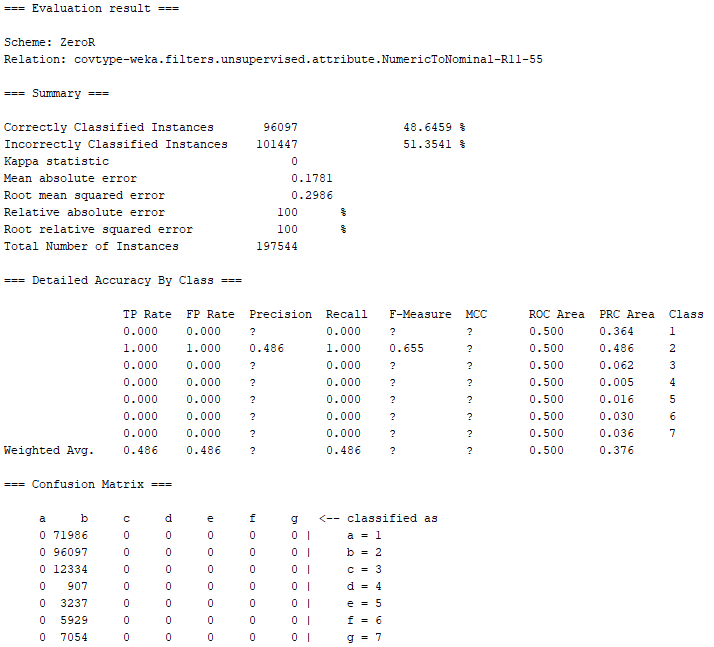
#### Classifier model

Scheme: ZeroR

Relation: covtype-weka.filters.unsupervised.attribute.NumericToNominal-R11-55

ZeroR predicts class value: 2

#### Evaluation Results



Given that it is the 0r, it isn’t a surprise when it chose class 2, the mode, as the rule. The funny thing is that unlike before (aka with block\_1.csv), our zero rule still doesn't score above 50, since our dataset mode class doesn’t take a majority. Class 1 however had the largest PRC area afterwards, since class 1 is the 2nd largest class.

### 

### OneR

#### Classifier model

Scheme: OneR

Relation: covtype-weka.filters.unsupervised.attribute.NumericToNominal-R11-55

Elevation:

< 1914.5 -> 6

< 1926.5 -> 3

< 1928.5 -> 6

< 1932.5 -> 3

< 1936.5 -> 6

< 1947.5 -> 3

< 1949.5 -> 6

< 1950.5 -> 3

< 1951.5 -> 6

< 1971.5 -> 3

< 1973.5 -> 6

< 1982.5 -> 3

< 1983.5 -> 6

< 1991.5 -> 3

< 1992.5 -> 6

< 1993.5 -> 3

< 1994.5 -> 6

< 2423.5 -> 3

< 2424.5 -> 6

< 2445.5 -> 3

< 2447.5 -> 6

< 2451.5 -> 3

< 2452.5 -> 6

< 2453.5 -> 3

< 2454.5 -> 6

< 2486.5 -> 3

< 2487.5 -> 2

< 2490.5 -> 3

< 2491.5 -> 2

< 2494.5 -> 3

< 2496.5 -> 2

< 2498.5 -> 3

< 2500.5 -> 2

< 2501.5 -> 3

< 2508.5 -> 2

< 2510.5 -> 3

< 3047.5 -> 2

< 3048.5 -> 1

< 3051.5 -> 2

< 3056.5 -> 1

< 3057.5 -> 2

< 3059.5 -> 1

< 3060.5 -> 2

< 3066.5 -> 1

< 3067.5 -> 2

< 3069.5 -> 1

< 3070.5 -> 2

< 3378.5 -> 1

< 3379.5 -> 7

< 3382.5 -> 1

< 3383.5 -> 7

< 3386.5 -> 1

< 3388.5 -> 7

< 3391.5 -> 1

< 3393.5 -> 7

< 3394.5 -> 1

< 3397.5 -> 7

< 3399.5 -> 1

< 3422.5 -> 7

< 3423.5 -> 1

< 3425.5 -> 7

< 3426.5 -> 1

< 3429.5 -> 7

< 3430.5 -> 1

< 3431.5 -> 7

< 3433.5 -> 1

< 3434.5 -> 7

< 3444.5 -> 1

< 3445.5 -> 7

< 3446.5 -> 1

< 3448.5 -> 7

< 3449.5 -> 1

< 3453.5 -> 7

< 3455.5 -> 1

< 3458.5 -> 7

< 3460.5 -> 1

< 3462.5 -> 7

< 3463.5 -> 1

< 3465.5 -> 7

< 3467.5 -> 1

< 3468.5 -> 7

< 3469.5 -> 1

< 3472.5 -> 7

< 3473.5 -> 1

< 3476.5 -> 7

< 3479.5 -> 1

< 3480.5 -> 7

< 3482.5 -> 1

< 3483.5 -> 7

< 3488.5 -> 1

< 3489.5 -> 7

< 3491.5 -> 1

< 3492.5 -> 7

< 3495.5 -> 1

< 3496.5 -> 7

< 3497.5 -> 1

< 3498.5 -> 7

< 3499.5 -> 1

< 3500.5 -> 7

< 3508.5 -> 1

< 3509.5 -> 7

< 3510.5 -> 1

< 3511.5 -> 7

< 3514.5 -> 1

< 3515.5 -> 7

< 3551.5 -> 1

< 3552.5 -> 7

< 3566.5 -> 1

< 3567.5 -> 7

< 3570.5 -> 1

< 3571.5 -> 7

< 3573.5 -> 1

< 3575.5 -> 7

< 3577.5 -> 1

< 3578.5 -> 7

< 3580.5 -> 1

< 3582.5 -> 7

< 3596.5 -> 1

< 3598.5 -> 7

< 3609.5 -> 1

< 3611.5 -> 7

< 3639.5 -> 1

< 3641.5 -> 7

< 3670.5 -> 1

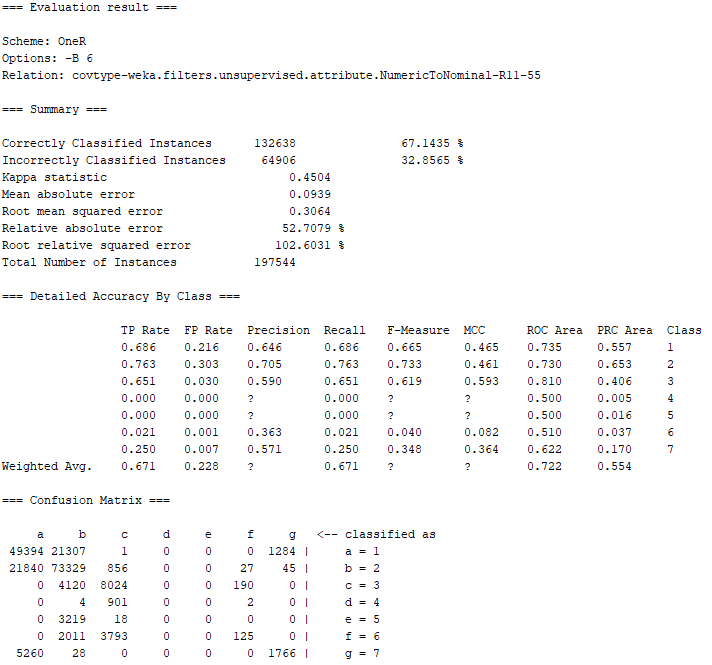
< 3673.5 -> 7

< 3680.5 -> 1

>= 3680.5 -> 7

(258846/383468 instances correct)

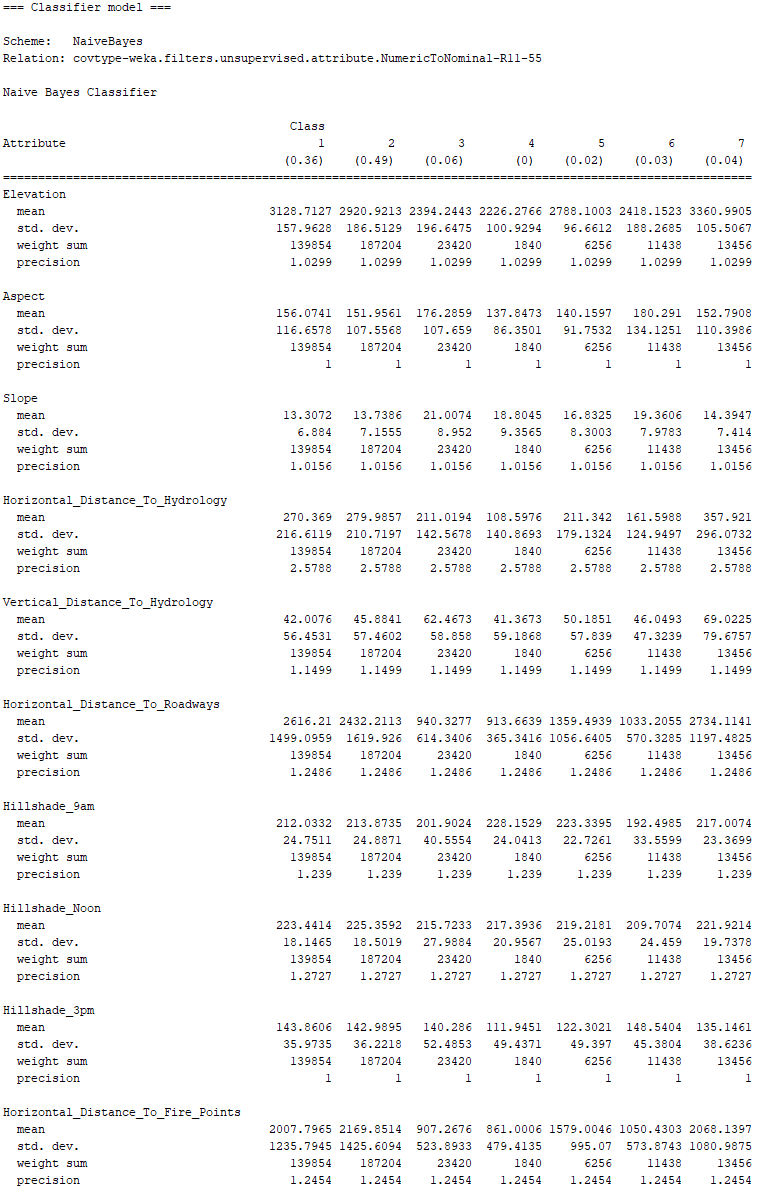
#### Evaluation Results



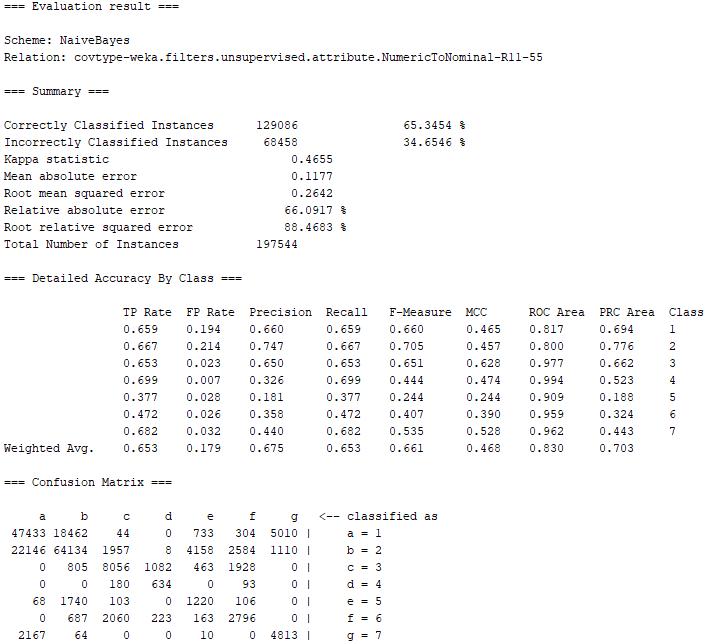
This here makes total sense. Plants care about elevation the most, and so does the random decision tree forest when based on the most important feature. What's interesting is how long this rule is and how each of those specific values actually correlates with multiple instances. Classification got to about 67% here and kappa statistics almost hit 0.5 (0 is random guessing, 1 is classifying). Looking at the confusion matrix, we can see that class 4 and 5 are never predicted, which may mean that elevation isn’t the most important when dealing with those specific plants in class 4 and 5.

### NaiveBayes

#### Classifier model



#### Evaluation Results



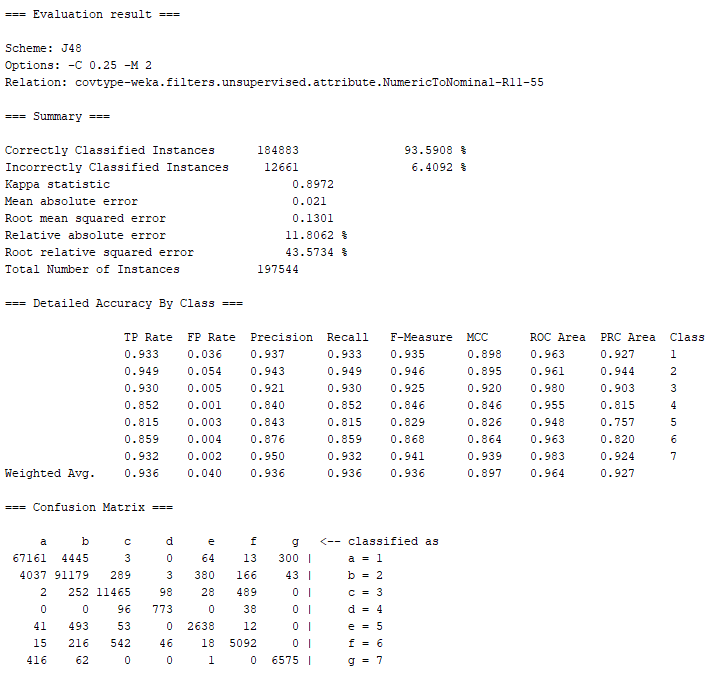
Naive Bayes here did an awful job. It dropped the classification rate down to 65.34, however, kappa statistic went up by 0.0155 after one rule (0.45). This means, even though the algorithm may not be ass good as classifying like the one rule, it is considered more classifying than guessing for one rule, which makes sense due to it never once classifying anything as 4 or 5.

### J48

#### Classifier model

#### 

#### Evaluation Results



Since we didn’t write an algo for the other classifiers we used in weka, I can now finally compare the decision trees we made in apache vs the one we made in spark. In spark we created a tree with 300 max bins, 20 max depth, and got a test score of 0.924 whereas here, we scored 93.6%, which is better. This must be due to the randomness used in our model. The weka model did much better, not to mention, and gave us easier access to our data. Our ROC area here on average was 0.964, which is insanely good, with our kappa stat being almost 0.9 (0.8972). Something interesting I noticed was that from both the Naive Bayes model and J48 model from weka, the algo never once predicted a certain class when predicting the actual class, for example, almost all class 7 was classified correctly, with the only two false classifications for instances of class 7 were classes 1 and 2.

# 

# Extension Writeup

## V1: Using Different Hyperparameters

### Code

import org.apache.spark.sql.functions.\_ //LIT WE MAKING FUNCTIONS

import org.apache.spark.ml.{PipelineModel, Pipeline}

import org.apache.spark.ml.classification.{DecisionTreeClassifier,

RandomForestClassifier, RandomForestClassificationModel}

import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

import org.apache.spark.ml.feature.{VectorAssembler, VectorIndexer}

import org.apache.spark.ml.linalg.Vector

import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}

import org.apache.spark.mllib.evaluation.MulticlassMetrics

import org.apache.spark.sql.{DataFrame, SparkSession}

import scala.util.Random

import spark.implicits.\_

//--Preparing the Data--

val dataWithoutHeaders = spark.read.

option("inferschema", true). //predict value type

option("header", false). //no header line

csv("/proj/cse398-498/course/AAS\_CH4/covtype.data")

val colNames = Seq( //initial column names

"Elevation", "Aspect", "Slope",

"Horizontal\_Distance\_To\_Hydrology", "Vertical\_Distance\_To\_Hydrology",

"Horizontal\_Distance\_To\_Roadways",

"Hillshade\_9am", "Hillshade\_Noon", "Hillshade\_3pm",

"Horizontal\_Distance\_To\_Fire\_Points"

) ++ ( //++ concatenates collections together

(0 until 4).map(i => s"Wilderness\_Area\_$i") //cool trick to label each column differently

) ++ (

(0 until 40).map(i => s"Soil\_Type\_$i") //same incrementing trick here

) ++ Seq("Cover\_Type") //save sequence as

val data = dataWithoutHeaders.toDF(colNames: \_\*).

withColumn("Cover\_Type", $"Cover\_Type".cast("double")) //cast each column to double (MLlib API req) and save it under Cover\_Type sequence

//-Lets undoing this one-hot encoding, by taking 4 columns encoding wilderness type into one, with values ranging from 0 to 3-

def unencodeOneHot(data: DataFrame): DataFrame = {

val wildernessCols = (0 until 4).map(i => s"Wilderness\_Area\_$i").toArray

val wildernessAssembler = new VectorAssembler().

setInputCols(wildernessCols).

setOutputCol("wilderness")

val unhotUDF = udf((vec: Vector) => vec.toArray.indexOf(1.0).toDouble)

val withWilderness = wildernessAssembler.transform(data).

drop(wildernessCols:\_\*).

withColumn("wilderness", unhotUDF($"wilderness"))

val soilCols = (0 until 40).map(i => s"Soil\_Type\_$i").toArray

val soilAssembler = new VectorAssembler().

setInputCols(soilCols).

setOutputCol("soil")

soilAssembler.transform(withWilderness).

drop(soilCols:\_\*).

withColumn("soil", unhotUDF($"soil"))

}

//--A First Decision Tree--

val Array(trainData, testData) = data.randomSplit(Array(0.9,0.1))

trainData.cache()

testData.cache()

val unencTrainData = unencodeOneHot(trainData)

val unencTestData = unencodeOneHot(testData)

//Spark MLlib requires all inputs to be vectors (arrays of doubles)

val inputCols = unencTrainData.columns.filter(\_ != "Cover\_Type") //training data is made into input by taking out covertype columns

val assembler = new VectorAssembler(). //create assembler

setInputCols(inputCols). //using input cols

setOutputCol("featureVector") //save output columns as a featureVector

val classifier = new DecisionTreeClassifier().

setSeed(Random.nextLong()). //set as a random seed

setLabelCol("Cover\_Type").

setFeaturesCol("featureVector").

setPredictionCol("prediction")

val pipeline = new Pipeline().setStages(Array(assembler, classifier)) //pipeline is multiprocess, use Array to order stages

val paramGrid = new ParamGridBuilder().

addGrid(classifier.maxMemoryInMB, Seq(1024)).

addGrid(classifier.impurity, Seq("entropy")). //use both impurity measures

addGrid(classifier.maxDepth, Seq(10, 20, 30)). //depth of 1 to 20

addGrid(classifier.maxBins, Seq(20, 40, 60, 80, 100)). //40 to 300 values for rules

addGrid(classifier.minInfoGain, Seq(0.0, 0.01, 0.02)). //ranges from no changes to impurity to improve impurity by 0.05

//addGrid(classifier.minInstancesPerNode, Seq(500, 1000, 2000))

//the first time I ran this, I tried to use 500, 1000, 2000 for minInstance, but ended up drastically dropping the success of the function

build()

val multiclassEval = new MulticlassClassificationEvaluator().

setLabelCol("Cover\_Type").

setPredictionCol("prediction").

setMetricName("accuracy") //kinda self explanotary what happening here

val validator = new TrainValidationSplit().

setSeed(Random.nextLong()).

setEstimator(pipeline).

setEvaluator(multiclassEval).

setEstimatorParamMaps(paramGrid). //HYPER PARAMETERS CAN STILL OVER FIT

setTrainRatio(0.9) //take another 10 percent and set it aside

//the left out 10% is used as a crossvalidation set (evaluate parameters that fit to training set)

//the original 10% left out to evaluate hyperparameters that fit the CV^^ (examples that arent in CV but has not been trained on [real world data])

val validatorModel = validator.fit(unencTrainData) //returns best overall pipeline

val bestModel = validatorModel.bestModel //extract best model

bestModel.asInstanceOf[PipelineModel].stages.last.extractParamMap //extract parameters

val paramsAndMetrics = validatorModel.validationMetrics. //accessing the the models metrics

zip(validatorModel.getEstimatorParamMaps).sortBy(-\_.\_1) //zip up all models params, sort by descending

paramsAndMetrics.foreach { case (metric, params) =>

println(metric)

println(params)

println()

}

/\*

0.9402965114057544

{

dtc\_e253ce313ef8-impurity: entropy,

dtc\_e253ce313ef8-maxBins: 60,

dtc\_e253ce313ef8-maxDepth: 30,

dtc\_e253ce313ef8-maxMemoryInMB: 1024,

dtc\_e253ce313ef8-minInfoGain: 0.0

}

0.9396278323334989

{

dtc\_e253ce313ef8-impurity: entropy,

dtc\_e253ce313ef8-maxBins: 100,

dtc\_e253ce313ef8-maxDepth: 30,

dtc\_e253ce313ef8-maxMemoryInMB: 1024,

dtc\_e253ce313ef8-minInfoGain: 0.0

}

0.9387489969813916

{

dtc\_e253ce313ef8-impurity: entropy,

dtc\_e253ce313ef8-maxBins: 100,

dtc\_e253ce313ef8-maxDepth: 30,

dtc\_e253ce313ef8-maxMemoryInMB: 1024,

dtc\_e253ce313ef8-minInfoGain: 0.01

}

0.9370104313935271

{

dtc\_e253ce313ef8-impurity: entropy,

dtc\_e253ce313ef8-maxBins: 40,

dtc\_e253ce313ef8-maxDepth: 30,

dtc\_e253ce313ef8-maxMemoryInMB: 1024,

dtc\_e253ce313ef8-minInfoGain: 0.0

}

0.936876695579076

{

dtc\_e253ce313ef8-impurity: entropy,

dtc\_e253ce313ef8-maxBins: 60,

dtc\_e253ce313ef8-maxDepth: 30,

dtc\_e253ce313ef8-maxMemoryInMB: 1024,

dtc\_e253ce313ef8-minInfoGain: 0.01

}

\*/

//-How well did the data do on the cv and test-

//print("CV Results")

validatorModel.validationMetrics.max

//0.9408314546635589

//print("UnencTestData")

multiclassEval.evaluate(bestModel.transform(unencTestData))

//0.9422549869904596

//print("UnencTrainingData")

multiclassEval.evaluate(bestModel.transform(unencTrainData))

//0.9994898368624394

//Does this even fall into the category of overfitting?

### Analysis

Before I even got started, I wanted to take a look at all the classifier parameters we can modify and add to our param grid. The page listed a couple new ones, but a majority of the ones provided were not applicable to DecisionTreeClassifier. When I tried to use minInstancePerNode at first, it significantly dropped the rate down to 70% ish when requiring a minInstance of 500. Since we knew 40 was the best max number of bins, I decided to increment only by 20 to see how much it affected the data. The result showed that around 40, 60, and 100 bins did the best on average. Mininfo gain didn’t matter whatsoever, with 0.0 being always the best. And for maxDepth, I tried using increments of 10, down from 30, since 30 was the max depth allowed (which is dumb). Our result for cv was 94% and our unencTestData was 94.2, which was a 2% gain from the other decision tree.

## Speeding up the program with maxMemoryInMB

### Code

import java.util.Calendar;

import org.apache.spark.sql.functions.\_

import org.apache.spark.ml.{PipelineModel, Pipeline}

import org.apache.spark.ml.classification.{DecisionTreeClassifier,

RandomForestClassifier, RandomForestClassificationModel}

import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

import org.apache.spark.ml.feature.{VectorAssembler, VectorIndexer}

import org.apache.spark.ml.linalg.Vector

import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}

import org.apache.spark.mllib.evaluation.MulticlassMetrics

import org.apache.spark.sql.{DataFrame, SparkSession}

import scala.util.Random

import spark.implicits.\_

//--Preparing the Data--

val dataWithoutHeaders = spark.read.

option("inferschema", true). //predict value type

option("header", false). //no header line

csv("/proj/cse398-498/course/AAS\_CH4/covtype.data")

val colNames = Seq( //initial column names

"Elevation", "Aspect", "Slope",

"Horizontal\_Distance\_To\_Hydrology", "Vertical\_Distance\_To\_Hydrology",

"Horizontal\_Distance\_To\_Roadways",

"Hillshade\_9am", "Hillshade\_Noon", "Hillshade\_3pm",

"Horizontal\_Distance\_To\_Fire\_Points"

) ++ ( //++ concatenates collections together

(0 until 4).map(i => s"Wilderness\_Area\_$i") //cool trick to label each column differently

) ++ (

(0 until 40).map(i => s"Soil\_Type\_$i") //same incrementing trick here

) ++ Seq("Cover\_Type") //save sequence as

val data = dataWithoutHeaders.toDF(colNames: \_\*).

withColumn("Cover\_Type", $"Cover\_Type".cast("double")) //cast each column to double (MLlib API req) and save it under Cover\_Type sequence

//-Lets undoing this one-hot encoding, by taking 4 columns encoding wilderness type into one, with values ranging from 0 to 3-

def unencodeOneHot(data: DataFrame): DataFrame = {

val wildernessCols = (0 until 4).map(i => s"Wilderness\_Area\_$i").toArray

val wildernessAssembler = new VectorAssembler().

setInputCols(wildernessCols).

setOutputCol("wilderness")

val unhotUDF = udf((vec: Vector) => vec.toArray.indexOf(1.0).toDouble)

val withWilderness = wildernessAssembler.transform(data).

drop(wildernessCols:\_\*).

withColumn("wilderness", unhotUDF($"wilderness"))

val soilCols = (0 until 40).map(i => s"Soil\_Type\_$i").toArray

val soilAssembler = new VectorAssembler().

setInputCols(soilCols).

setOutputCol("soil")

soilAssembler.transform(withWilderness).

drop(soilCols:\_\*).

withColumn("soil", unhotUDF($"soil"))

}

//--A First Decision Tree--

val Array(trainData, testData) = data.randomSplit(Array(0.9,0.1))

trainData.cache()

testData.cache()

val unencTrainData = unencodeOneHot(trainData)

val unencTestData = unencodeOneHot(testData)

//Spark MLlib requires all inputs to be vectors (arrays of doubles)

val inputCols = unencTrainData.columns.filter(\_ != "Cover\_Type") //training data is made into input by taking out covertype columns

val assembler = new VectorAssembler(). //create assembler

setInputCols(inputCols). //using input cols

setOutputCol("featureVector") //save output columns as a featureVector

val classifier = new DecisionTreeClassifier().

setSeed(Random.nextLong()). //set as a random seed

setLabelCol("Cover\_Type").

setFeaturesCol("featureVector").

setPredictionCol("prediction")

val pipeline = new Pipeline().setStages(Array(assembler, classifier)) //pipeline is multiprocess, use Array to order stages

//ITERATION 0

print("ITERATION 0\n")

val paramGrid = new ParamGridBuilder().

addGrid(classifier.maxMemoryInMB, Seq(128)).

addGrid(classifier.impurity, Seq("entropy")). //use both impurity measures

addGrid(classifier.maxDepth, Seq(30)). //depth of 1 to 20

addGrid(classifier.maxBins, Seq(20, 40, 60, 80, 100)). //40 to 300 values for rules

addGrid(classifier.minInfoGain, Seq(0.0)). //ranges from no changes to impurity to improve impurity by 0.05

//addGrid(classifier.minInstancesPerNode, Seq(500, 1000, 2000))

//the first time I ran this, I tried to use 500, 1000, 2000 for minInstance, but ended up drastically dropping the success of the function

build()

val multiclassEval = new MulticlassClassificationEvaluator().

setLabelCol("Cover\_Type").

setPredictionCol("prediction").

setMetricName("accuracy") //kinda self explanotary what happening here

val validator = new TrainValidationSplit().

setSeed(Random.nextLong()).

setEstimator(pipeline).

setEvaluator(multiclassEval).

setEstimatorParamMaps(paramGrid). //HYPER PARAMETERS CAN STILL OVER FIT

setTrainRatio(0.9) //take another 10 percent and set it aside

//the left out 10% is used as a crossvalidation set (evaluate parameters that fit to training set)

//the original 10% left out to evaluate hyperparameters that fit the CV^^ (examples that arent in CV but has not been trained on [real world data])

val start = Calendar.getInstance()

val sh = start.get(Calendar.HOUR)

val sm = start.get(Calendar.MINUTE)

val ss = start.get(Calendar.SECOND)

printf("Start Time: %02d:%02d:%02d\n", sh, sm, ss);

val validatorModel = validator.fit(unencTrainData) //returns best overall pipeline

val end = Calendar.getInstance()

val eh = end.get(Calendar.HOUR)

val em = end.get(Calendar.MINUTE)

val es = end.get(Calendar.SECOND)

printf("End Time: %02d:%02d:%02d\n", eh, em, es);

val dh = eh-sh

val dm = em-sm

val ds = es-ss

printf("Total Time: %02d:%02d:%02d\n", dh, dm, ds);

//ITERATION 1

print("ITERATION 1\n")

val paramGrid = new ParamGridBuilder().

addGrid(classifier.maxMemoryInMB, Seq(256)).

addGrid(classifier.impurity, Seq("entropy")). //use both impurity measures

addGrid(classifier.maxDepth, Seq(30)). //depth of 1 to 20

addGrid(classifier.maxBins, Seq(20, 40, 60, 80, 100)). //40 to 300 values for rules

addGrid(classifier.minInfoGain, Seq(0.0)). //ranges from no changes to impurity to improve impurity by 0.05

//addGrid(classifier.minInstancesPerNode, Seq(500, 1000, 2000))

//the first time I ran this, I tried to use 500, 1000, 2000 for minInstance, but ended up drastically dropping the success of the function

build()

val multiclassEval = new MulticlassClassificationEvaluator().

setLabelCol("Cover\_Type").

setPredictionCol("prediction").

setMetricName("accuracy") //kinda self explanotary what happening here

val validator = new TrainValidationSplit().

setSeed(Random.nextLong()).

setEstimator(pipeline).

setEvaluator(multiclassEval).

setEstimatorParamMaps(paramGrid). //HYPER PARAMETERS CAN STILL OVER FIT

setTrainRatio(0.9) //take another 10 percent and set it aside

//the left out 10% is used as a crossvalidation set (evaluate parameters that fit to training set)

//the original 10% left out to evaluate hyperparameters that fit the CV^^ (examples that arent in CV but has not been trained on [real world data])

val start = Calendar.getInstance()

val sh = start.get(Calendar.HOUR)

val sm = start.get(Calendar.MINUTE)

val ss = start.get(Calendar.SECOND)

printf("Start Time: %02d:%02d:%02d\n", sh, sm, ss);

val validatorModel = validator.fit(unencTrainData) //returns best overall pipeline

val end = Calendar.getInstance()

val eh = end.get(Calendar.HOUR)

val em = end.get(Calendar.MINUTE)

val es = end.get(Calendar.SECOND)

printf("End Time: %02d:%02d:%02d\n", eh, em, es);

val dh = eh-sh

val dm = em-sm

val ds = es-ss

printf("Total Time: %02d:%02d:%02d\n", dh, dm, ds);

//ITERATION 2

print("ITERATION 2\n")

val paramGrid = new ParamGridBuilder().

addGrid(classifier.maxMemoryInMB, Seq(512)).

addGrid(classifier.impurity, Seq("entropy")). //use both impurity measures

addGrid(classifier.maxDepth, Seq(30)). //depth of 1 to 20

addGrid(classifier.maxBins, Seq(20, 40, 60, 80, 100)). //40 to 300 values for rules

addGrid(classifier.minInfoGain, Seq(0.0)). //ranges from no changes to impurity to improve impurity by 0.05

//addGrid(classifier.minInstancesPerNode, Seq(500, 1000, 2000))

//the first time I ran this, I tried to use 500, 1000, 2000 for minInstance, but ended up drastically dropping the success of the function

build()

val multiclassEval = new MulticlassClassificationEvaluator().

setLabelCol("Cover\_Type").

setPredictionCol("prediction").

setMetricName("accuracy") //kinda self explanotary what happening here

val validator = new TrainValidationSplit().

setSeed(Random.nextLong()).

setEstimator(pipeline).

setEvaluator(multiclassEval).

setEstimatorParamMaps(paramGrid). //HYPER PARAMETERS CAN STILL OVER FIT

setTrainRatio(0.9) //take another 10 percent and set it aside

//the left out 10% is used as a crossvalidation set (evaluate parameters that fit to training set)

//the original 10% left out to evaluate hyperparameters that fit the CV^^ (examples that arent in CV but has not been trained on [real world data])

val start = Calendar.getInstance()

val sh = start.get(Calendar.HOUR)

val sm = start.get(Calendar.MINUTE)

val ss = start.get(Calendar.SECOND)

printf("Start Time: %02d:%02d:%02d\n", sh, sm, ss);

val validatorModel = validator.fit(unencTrainData) //returns best overall pipeline

val end = Calendar.getInstance()

val eh = end.get(Calendar.HOUR)

val em = end.get(Calendar.MINUTE)

val es = end.get(Calendar.SECOND)

printf("End Time: %02d:%02d:%02d\n", eh, em, es);

val dh = eh-sh

val dm = em-sm

val ds = es-ss

printf("Total Time: %02d:%02d:%02d\n", dh, dm, ds);

//ITERATION 3

print("ITERATION 3\n")

val paramGrid = new ParamGridBuilder().

addGrid(classifier.maxMemoryInMB, Seq(1024)).

addGrid(classifier.impurity, Seq("entropy")). //use both impurity measures

addGrid(classifier.maxDepth, Seq(30)). //depth of 1 to 20

addGrid(classifier.maxBins, Seq(20, 40, 60, 80, 100)). //40 to 300 values for rules

addGrid(classifier.minInfoGain, Seq(0.0)). //ranges from no changes to impurity to improve impurity by 0.05

//addGrid(classifier.minInstancesPerNode, Seq(500, 1000, 2000))

//the first time I ran this, I tried to use 500, 1000, 2000 for minInstance, but ended up drastically dropping the success of the function

build()

val multiclassEval = new MulticlassClassificationEvaluator().

setLabelCol("Cover\_Type").

setPredictionCol("prediction").

setMetricName("accuracy") //kinda self explanotary what happening here

val validator = new TrainValidationSplit().

setSeed(Random.nextLong()).

setEstimator(pipeline).

setEvaluator(multiclassEval).

setEstimatorParamMaps(paramGrid). //HYPER PARAMETERS CAN STILL OVER FIT

setTrainRatio(0.9) //take another 10 percent and set it aside

//the left out 10% is used as a crossvalidation set (evaluate parameters that fit to training set)

//the original 10% left out to evaluate hyperparameters that fit the CV^^ (examples that arent in CV but has not been trained on [real world data])

val start = Calendar.getInstance()

val sh = start.get(Calendar.HOUR)

val sm = start.get(Calendar.MINUTE)

val ss = start.get(Calendar.SECOND)

printf("Start Time: %02d:%02d:%02d\n", sh, sm, ss);

val validatorModel = validator.fit(unencTrainData) //returns best overall pipeline

val end = Calendar.getInstance()

val eh = end.get(Calendar.HOUR)

val em = end.get(Calendar.MINUTE)

val es = end.get(Calendar.SECOND)

printf("End Time: %02d:%02d:%02d\n", eh, em, es);

val dh = eh-sh

val dm = em-sm

val ds = es-ss

printf("Total Time: %02d:%02d:%02d\n", dh, dm, ds);

//ITERATION 4

print("ITERATION 4\n")

val paramGrid = new ParamGridBuilder().

addGrid(classifier.maxMemoryInMB, Seq(2048)).

addGrid(classifier.impurity, Seq("entropy")). //use both impurity measures

addGrid(classifier.maxDepth, Seq(30)). //depth of 1 to 20

addGrid(classifier.maxBins, Seq(20, 40, 60, 80, 100)). //40 to 300 values for rules

addGrid(classifier.minInfoGain, Seq(0.0)). //ranges from no changes to impurity to improve impurity by 0.05

//addGrid(classifier.minInstancesPerNode, Seq(500, 1000, 2000))

//the first time I ran this, I tried to use 500, 1000, 2000 for minInstance, but ended up drastically dropping the success of the function

build()

val multiclassEval = new MulticlassClassificationEvaluator().

setLabelCol("Cover\_Type").

setPredictionCol("prediction").

setMetricName("accuracy") //kinda self explanotary what happening here

val validator = new TrainValidationSplit().

setSeed(Random.nextLong()).

setEstimator(pipeline).

setEvaluator(multiclassEval).

setEstimatorParamMaps(paramGrid). //HYPER PARAMETERS CAN STILL OVER FIT

setTrainRatio(0.9) //take another 10 percent and set it aside

//the left out 10% is used as a crossvalidation set (evaluate parameters that fit to training set)

//the original 10% left out to evaluate hyperparameters that fit the CV^^ (examples that arent in CV but has not been trained on [real world data])

val start = Calendar.getInstance()

val sh = start.get(Calendar.HOUR)

val sm = start.get(Calendar.MINUTE)

val ss = start.get(Calendar.SECOND)

printf("Start Time: %02d:%02d:%02d\n", sh, sm, ss);

val validatorModel = validator.fit(unencTrainData) //returns best overall pipeline

val end = Calendar.getInstance()

val eh = end.get(Calendar.HOUR)

val em = end.get(Calendar.MINUTE)

val es = end.get(Calendar.SECOND)

printf("End Time: %02d:%02d:%02d\n", eh, em, es);

val dh = eh-sh

val dm = em-sm

val ds = es-ss

printf("Total Time: %02d:%02d:%02d\n", dh, dm, ds);

//ITERATION 5

print("ITERATION 4\n")

val paramGrid = new ParamGridBuilder().

addGrid(classifier.maxMemoryInMB, Seq(4096)).

addGrid(classifier.impurity, Seq("entropy")). //use both impurity measures

addGrid(classifier.maxDepth, Seq(30)). //depth of 1 to 20

addGrid(classifier.maxBins, Seq(20, 40, 60, 80, 100)). //40 to 300 values for rules

addGrid(classifier.minInfoGain, Seq(0.0)). //ranges from no changes to impurity to improve impurity by 0.05

//addGrid(classifier.minInstancesPerNode, Seq(500, 1000, 2000))

//the first time I ran this, I tried to use 500, 1000, 2000 for minInstance, but ended up drastically dropping the success of the function

build()

val multiclassEval = new MulticlassClassificationEvaluator().

setLabelCol("Cover\_Type").

setPredictionCol("prediction").

setMetricName("accuracy") //kinda self explanotary what happening here

val validator = new TrainValidationSplit().

setSeed(Random.nextLong()).

setEstimator(pipeline).

setEvaluator(multiclassEval).

setEstimatorParamMaps(paramGrid). //HYPER PARAMETERS CAN STILL OVER FIT

setTrainRatio(0.9) //take another 10 percent and set it aside

//the left out 10% is used as a crossvalidation set (evaluate parameters that fit to training set)

//the original 10% left out to evaluate hyperparameters that fit the CV^^ (examples that arent in CV but has not been trained on [real world data])

val start = Calendar.getInstance()

val sh = start.get(Calendar.HOUR)

val sm = start.get(Calendar.MINUTE)

val ss = start.get(Calendar.SECOND)

printf("Start Time: %02d:%02d:%02d\n", sh, sm, ss);

val validatorModel = validator.fit(unencTrainData) //returns best overall pipeline

val end = Calendar.getInstance()

val eh = end.get(Calendar.HOUR)

val em = end.get(Calendar.MINUTE)

val es = end.get(Calendar.SECOND)

printf("End Time: %02d:%02d:%02d\n", eh, em, es);

val dh = eh-sh

val dm = em-sm

val ds = es-ss

printf("Total Time: %02d:%02d:%02d\n", dh, dm, ds);

### 

### Result

ITERATION 0

Start Time: 08:34:15

End Time: 08:36:24

Total Time: 00:02:09

ITERATION 1

Start Time: 08:03:28

End Time: 08:05:43

Total Time: 00:02:15

ITERATION 2

Start Time: 08:05:44

End Time: 08:07:56

Total Time: 00:02:12

ITERATION 3

Start Time: 08:07:57

End Time: 08:10:04

Total Time: 00:02:07

ITERATION 4

Start Time: 08:10:05

End Time: 08:12:16

Total Time: 00:02:11

ITERATION 5

Start Time: 08:12:17

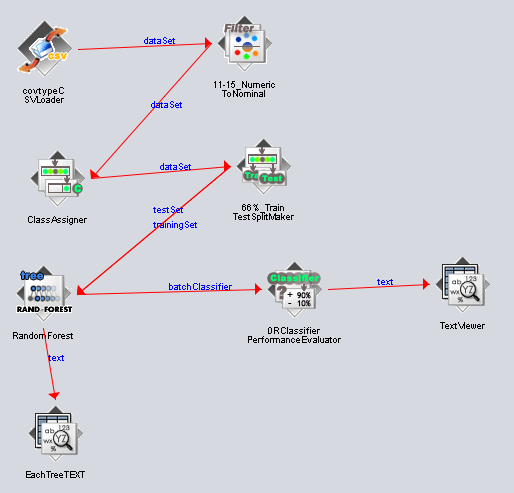
End Time: 08:14:28

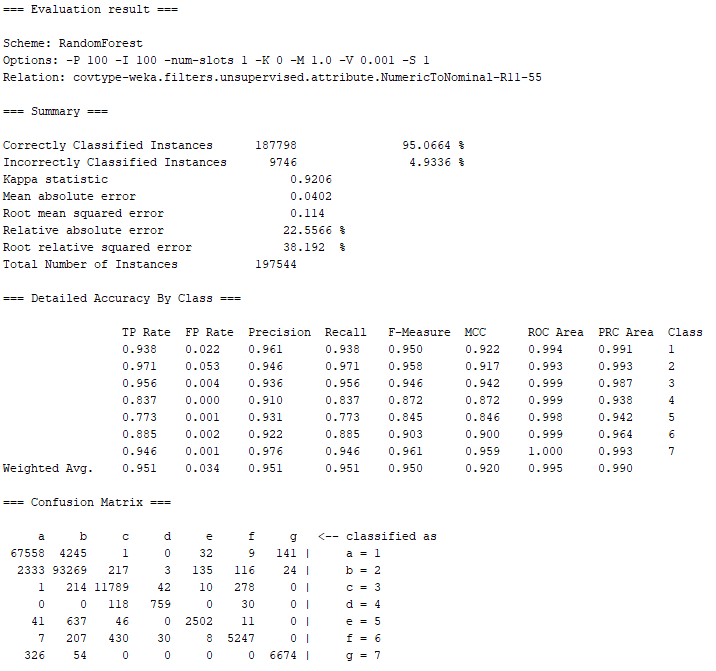
Total Time: 00:02:11

### Analysis

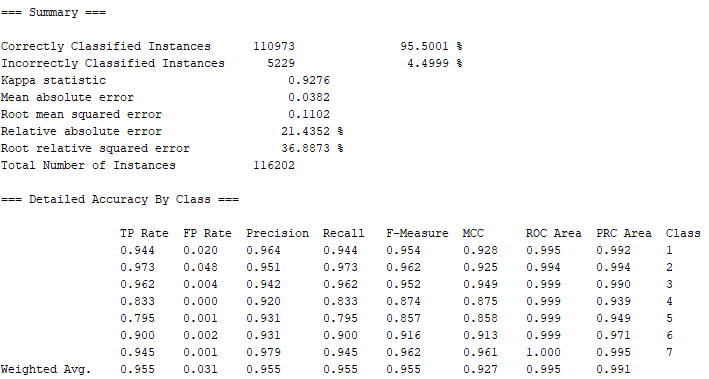
As of right now, it seems like increasing the mb doesn’t do anything. Even decreasing it does nothing. I am wondering if this is due to my system being on the SSH or is this just something that occurs?

## Weka Random Decision Tree Forest





Since weka seemed to do a bit better than apache, I tried using the random decision forest to compare against the one we made in spark. Surprisingly, spark did better by 0.002%, which makes me believe that if it did a bit better on the randomness of the algo, it would do much better! I also want to see what happens if I give it more of the dataset than 66%.



Increasing the percentage gave us an increase of 0.5, which is really good! (80% of the data was used, which is exactly how much apache spark uses, since we use 10% for test, 10% for cv testing).

## 

## V2: Using Different Hyperparameters

### Code

import org.apache.spark.sql.functions.\_ //LIT WE MAKING FUNCTIONS

import org.apache.spark.ml.{PipelineModel, Pipeline}

import org.apache.spark.ml.classification.{DecisionTreeClassifier,

RandomForestClassifier, RandomForestClassificationModel}

import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

import org.apache.spark.ml.feature.{VectorAssembler, VectorIndexer}

import org.apache.spark.ml.linalg.Vector

import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}

import org.apache.spark.mllib.evaluation.MulticlassMetrics

import org.apache.spark.sql.{DataFrame, SparkSession}

import scala.util.Random

import spark.implicits.\_

//--Preparing the Data--

val dataWithoutHeaders = spark.read.

option("inferschema", true). //predict value type

option("header", false). //no header line

csv("/proj/cse398-498/course/AAS\_CH4/covtype.data")

val colNames = Seq( //initial column names

"Elevation", "Aspect", "Slope",

"Horizontal\_Distance\_To\_Hydrology", "Vertical\_Distance\_To\_Hydrology",

"Horizontal\_Distance\_To\_Roadways",

"Hillshade\_9am", "Hillshade\_Noon", "Hillshade\_3pm",

"Horizontal\_Distance\_To\_Fire\_Points"

) ++ ( //++ concatenates collections together

(0 until 4).map(i => s"Wilderness\_Area\_$i") //cool trick to label each column differently

) ++ (

(0 until 40).map(i => s"Soil\_Type\_$i") //same incrementing trick here

) ++ Seq("Cover\_Type") //save sequence as

val data = dataWithoutHeaders.toDF(colNames: \_\*).

withColumn("Cover\_Type", $"Cover\_Type".cast("double")) //cast each column to double (MLlib API req) and save it under Cover\_Type sequence

//-Lets undoing this one-hot encoding, by taking 4 columns encoding wilderness type into one, with values ranging from 0 to 3-

def unencodeOneHot(data: DataFrame): DataFrame = {

val wildernessCols = (0 until 4).map(i => s"Wilderness\_Area\_$i").toArray

val wildernessAssembler = new VectorAssembler().

setInputCols(wildernessCols).

setOutputCol("wilderness")

val unhotUDF = udf((vec: Vector) => vec.toArray.indexOf(1.0).toDouble)

val withWilderness = wildernessAssembler.transform(data).

drop(wildernessCols:\_\*).

withColumn("wilderness", unhotUDF($"wilderness"))

val soilCols = (0 until 40).map(i => s"Soil\_Type\_$i").toArray

val soilAssembler = new VectorAssembler().

setInputCols(soilCols).

setOutputCol("soil")

soilAssembler.transform(withWilderness).

drop(soilCols:\_\*).

withColumn("soil", unhotUDF($"soil"))

}

//--A First Decision Tree--

val Array(trainData, testData) = data.randomSplit(Array(0.9,0.1))

trainData.cache()

testData.cache()

val unencTrainData = unencodeOneHot(trainData)

val unencTestData = unencodeOneHot(testData)

//Spark MLlib requires all inputs to be vectors (arrays of doubles)

val inputCols = unencTrainData.columns.filter(\_ != "Cover\_Type") //training data is made into input by taking out covertype columns

val assembler = new VectorAssembler(). //create assembler

setInputCols(inputCols). //using input cols

setOutputCol("featureVector") //save output columns as a featureVector

val classifier = new DecisionTreeClassifier().

setSeed(Random.nextLong()). //set as a random seed

setLabelCol("Cover\_Type").

setFeaturesCol("featureVector").

setPredictionCol("prediction")

val pipeline = new Pipeline().setStages(Array(assembler, classifier)) //pipeline is multiprocess, use Array to order stages

val paramGrid = new ParamGridBuilder().

addGrid(classifier.impurity, Seq("entropy")). //use both impurity measures

addGrid(classifier.maxDepth, Seq(30)). //use max max depth

addGrid(classifier.maxBins, Seq(40, 60, 80, 100)). //40 to 300 values for rules

addGrid(classifier.minInfoGain, Seq(0.0)). //0.0 since it doesn't seem to affect the data

addGrid(classifier.minInstancesPerNode, Seq(5, 10, 25, 100)).

build()

val multiclassEval = new MulticlassClassificationEvaluator().

setLabelCol("Cover\_Type").

setPredictionCol("prediction").

setMetricName("accuracy") //kinda self explanotary what happening here

val validator = new TrainValidationSplit().

setSeed(Random.nextLong()).

setEstimator(pipeline).

setEvaluator(multiclassEval).

setEstimatorParamMaps(paramGrid). //HYPER PARAMETERS CAN STILL OVER FIT

setTrainRatio(0.9) //take another 10 percent and set it aside

//the left out 10% is used as a crossvalidation set (evaluate parameters that fit to training set)

//the original 10% left out to evaluate hyperparameters that fit the CV^^ (examples that arent in CV but has not been trained on [real world data])

val validatorModel = validator.fit(unencTrainData) //returns best overall pipeline

val bestModel = validatorModel.bestModel //extract best model

bestModel.asInstanceOf[PipelineModel].stages.last.extractParamMap //extract parameters

val paramsAndMetrics = validatorModel.validationMetrics. //accessing the the models metrics

zip(validatorModel.getEstimatorParamMaps).sortBy(-\_.\_1) //zip up all models params, sort by descending

paramsAndMetrics.foreach { case (metric, params) =>

println(metric)

println(params)

println()

}

//-How well did the data do on the cv and test-

print("CV Results")

validatorModel.validationMetrics.max

print("UnencTestData")

multiclassEval.evaluate(bestModel.transform(unencTestData))

print("UnencTrainingData")

multiclassEval.evaluate(bestModel.transform(unencTrainData))

### Results

0.9306456536670992

{

dtc\_fd915673ac55-impurity: entropy,

dtc\_fd915673ac55-maxBins: 100,

dtc\_fd915673ac55-maxDepth: 30,

dtc\_fd915673ac55-minInfoGain: 0.0,

dtc\_fd915673ac55-minInstancesPerNode: 5

}

0.9301114248645348

{

dtc\_fd915673ac55-impurity: entropy,

dtc\_fd915673ac55-maxBins: 80,

dtc\_fd915673ac55-maxDepth: 30,

dtc\_fd915673ac55-minInfoGain: 0.0,

dtc\_fd915673ac55-minInstancesPerNode: 5

}

0.9274593604518049

{

dtc\_fd915673ac55-impurity: entropy,

dtc\_fd915673ac55-maxBins: 40,

dtc\_fd915673ac55-maxDepth: 30,

dtc\_fd915673ac55-minInfoGain: 0.0,

dtc\_fd915673ac55-minInstancesPerNode: 5

} ...

CV Results

res103: Double = 0.9308936884682897

UnencTest

res105: Double = 0.9325177721029747

UnencTrain

res107: Double = 0.9705002905998593

### Analysis

I think this data can do better, since clearly 5 minInstances were doing significantly better than anything else. It also did help with the overfitting a bit. So I did it again, with only 1, 2, and 3, and got that 1 was the best minInstance and got the following results: CV Results 0.9392545750694644, UnencTest 0.9417283779330917, UnencTrain 0.9995815427886829. It just seems like minInstance is really only good for preventing overfitting, which also decreases the overall classification rate.

## 

NOT REALLY EXTENSION - Just Curious

## RandomForest Hyperparameter Tuning

### Iteration 1

val paramGrid = new ParamGridBuilder().

addGrid(classifier.impurity, Seq("entropy")). //use both impurity measures

addGrid(classifier.maxDepth, Seq(30)). //use max max depth

addGrid(classifier.maxBins, Seq(40)). //40 values for rules

addGrid(classifier.minInfoGain, Seq(0.0)). //0.0 since it doesn't seem to affect the data

addGrid(classifier.minInstancesPerNode, Seq(1)).

build()

0.9595951851567891

{

rfc\_1e08a78abd5b-impurity: entropy,

rfc\_1e08a78abd5b-maxBins: 40,

rfc\_1e08a78abd5b-maxDepth: 30,

rfc\_1e08a78abd5b-minInfoGain: 0.0,

rfc\_1e08a78abd5b-minInstancesPerNode: 1

}

CV Results: Double = 0.9595951851567891

res161: Double = 0.9613621592036398

res163: Double = 0.9991177810183218

### Iteration 2

{

rfc\_bab14776d312-impurity: entropy,

rfc\_bab14776d312-maxBins: 100,

rfc\_bab14776d312-maxDepth: 30,

rfc\_bab14776d312-minInfoGain: 0.0,

rfc\_bab14776d312-minInstancesPerNode: 1

}

CV Resultsres9: Double = 0.9628713820203182

res11: Double = 0.9648687317922463

res13: Double = 0.9995697904975325