GarciaArnau_Homework2

January 15, 2024

Arnau García Fernandez

SPDB: Homework II

Grab 2 different Datasets (or tables). One will be for predicting a continuous variable and the other will be for the categorical variable prediction. You can chose one from class but it is mandatory to have at least one of the datasets that is not from class: it will be more interesting for me if you look for something a little bit different (but easy enough to understand!).

1. Find a dataset and load it: You gotta start somewhere…
2. Do an exploratory analysis: Check the variables, their meaning, some statistics… and obviously separate train and test! Are there missing values? Need to treat them!
3. Create the needed variables and modify the table as you need: If you discover that you need to do any changes, prepare the table so and get it ready to train!
4. Train different models: train different models with the methods you find more useful (and maybe some that are not useful but will be nice to see why they are not: Regression, trees -GDT and RF(?)-, naivebayes… you can innovate a bit if you feel like it.
5. Study the results of the model: make the proper analysis on the behaviour and predictions of the model.
6. Discussion: Here, you would then have to chose which model works best and actually implement it into a certain problem. Just argue which one (for both examples) you would use and why (and why not the others?). You have until January 21st 2024 to deliver the work.

0.0.1 Categorical variable prediction

We first start doing prediction of a categorical variable. We will be using a penguins data set from the public git hub repository of Victor Peña (teacher of the subject of Linear and generalised linear models of this master). In this data set we have the species of the penguins, the island where the penguins lives, the bill length (in mm), the bill depth (in mm), the flipper length (in mm), the body mass (in g), the sex and the year. We can find this data set in the following link (https://github.com/VicPena/VicPena.github.io/blob/master/glm/datasets/penguins_test.csv).

First, we import all the stuff that we will be using during the process:

[ ]:
```
import org.apache.spark.ml.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.ml.feature.VectorAssembler
import spark.implicits._
import org.apache.spark.ml.regression.{ GeneralizedLinearRegression,␣
 ↪GeneralizedLinearRegressionModel }
```

```
import org.apache.spark.ml.{ Pipeline, PipelineModel }
import org.apache.spark.ml.tuning.{ CrossValidator, ParamGridBuilder }
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.sql._
import org.apache.spark.sql.functions._
import org.apache.spark.mllib.evaluation.RegressionMetrics
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.feature.StringIndexer
```

[ ]:
```
import org.apache.spark.ml.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.ml.feature.VectorAssembler
import spark.implicits._
import org.apache.spark.ml.regression.{GeneralizedLinearRegression,
GeneralizedLinearRegressionModel}
import org.apache.spark.ml.{Pipeline, PipelineModel}
import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.sql._
import org.apache.spark.sql.functions._
import org.apache.spark.mllib.evaluation.RegressionMetrics
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.feature.StringIndexer
```

Now we take the csv file:

[ ]:
```
val myfile = sc.textFile("/Users/arnaugarcia/Desktop/Q1/bases de dades/zeppelin/
↪data/penguins_test.csv")
```

[ ]:
```
myfile: org.apache.spark.rdd.RDD[String] = /Users/arnaugarcia/Desktop/Q1/bases
de dades/zeppelin/data/penguins_test.csv MapPartitionsRDD[1] at textFile at
<console>:43
```

[ ]:
```
myfile.take(10)
```

[ ]:
```
res0: Array[String] = Array(species,island,bill_length_mm,bill_depth_mm,flipper_
length_mm,body_mass_g,sex,year, Adelie,Torgersen,38.9,17.8,181,3625,female,2007,
Adelie,Torgersen,36.6,17.8,185,3700,female,2007,
Adelie,Torgersen,42.5,20.7,197,4500,male,2007,
Adelie,Biscoe,37.8,18.3,174,3400,female,2007,
Adelie,Biscoe,38.2,18.1,185,3950,male,2007,
Adelie,Biscoe,40.5,17.9,187,3200,female,2007,
Adelie,Dream,37.2,18.1,178,3900,male,2007,
Adelie,Dream,39.5,17.8,188,3300,female,2007,
Adelie,Dream,36.4,17,195,3325,female,2007)
```

Using what we saw in class we can import the csv file in a properly format and we can print the observations in cells, it is, with a table format.

[ ]:
```
var penguin = spark.read
    .format("csv")
    .option("header", "true")
    .option("inferSchema", "true")
    .csv("/Users/arnaugarcia/Desktop/Q1/bases de dades/zeppelin/data/
 ↪penguins_test.csv")
```

[ ]:
```
penguin: org.apache.spark.sql.DataFrame = [species: string, island: string … 6
more fields]
```

[ ]:
```
penguin.show(5)
```

```
+-------+---------+--------------+-------------+-----------------+-----------+--
----+----+
|species|   island|bill_length_mm|bill_depth_mm|flipper_length_mm|body_mass_g|
sex|year|
+-------+---------+--------------+-------------+-----------------+-----------+--
----+----+
| Adelie|Torgersen|          38.9|         17.8|              181|
3625|female|2007|
| Adelie|Torgersen|          36.6|         17.8|              185|
3700|female|2007|
| Adelie|Torgersen|          42.5|         20.7|              197|       4500|
male|2007|
| Adelie|   Biscoe|          37.8|         18.3|              174|
3400|female|2007|
| Adelie|   Biscoe|          38.2|         18.1|              185|       3950|
male|2007|
+-------+---------+--------------+-------------+-----------------+-----------+--
----+----+
only showing top 5 rows
```

Now, we will filter and eliminate one type of the species. In our dataset we have three penguin species: “Adeelie”, “Gentoo” and “Chinstrap”. But, after review the spark documentation (see the documentation in https://spark.apache.org/docs/latest/ml-classification-regression.html#generalized-linear-regression) about the avilaible methods for classification, we only have the binomial distribution avilable. For classify three categories we need the multinomial regression. Then, to be able to work with the tools provided by spark we delete one type of the species (the Chinstrap penguin species) and we will be classifying between two type of penguins.

[ ]:
```
penguin = penguin.filter($"species"!=="Chinstrap")
```

[ ]:
```
penguin: org.apache.spark.sql.DataFrame = [species: string, island: string … 6
more fields]
```

Now, we do som exploratory analysis, computing some statistics in order to understand how our dataset behaves. In addition, we study if there are missing values. First of all, we expose a summary with different basic statistics of the variables of the penguins datase.

[ ]:
```
val summaryStats = penguin.describe()
summaryStats.show()
```

```
+-------+-------+---------+-----------------+-----------------+--------------
---+-----------------+------+-----------------+
|summary|species|   island|    bill_length_mm|     bill_depth_mm|
flipper_length_mm|      body_mass_g|   sex|             year|
+-------+-------+---------+-----------------+-----------------+--------------
---+-----------------+------+-----------------+
|  count|     80|       80|               80|               80|
80|               80|    80|               80|
|   mean|   NULL|     NULL|42.596250000000005|          16.72625|
202.2375|        4302.8125|  NULL|         2008.125|
| stddev|   NULL|
NULL|5.2968928974530565|1.8606344195026496|14.382869337505177|823.7475509462067|
NULL|0.8171422880914381|
|    min| Adelie|   Biscoe|             34.5|             13.3|
174|             2900|female|             2007|
|    max| Gentoo|Torgersen|             59.6|             20.7|
230|             6050|  male|             2009|
+-------+-------+---------+-----------------+-----------------+--------------
---+-----------------+------+-----------------+
```

[ ]:
```
summaryStats: org.apache.spark.sql.DataFrame = [summary: string, species: string
… 7 more fields]
```

Now we count distinct values for each variable. Indeed, we obtain two type of species, which is exaclty what we want for do the binary classification.

[ ]:
```
penguin.select(penguin.columns.map(species => countDistinct(col(species)).
  ↪alias(species)): _*).show()
```

```
+-------+------+--------------+-------------+-----------------+-----------+---+-
---+
|species|island|bill_length_mm|bill_depth_mm|flipper_length_mm|body_mass_g|sex|y
ear|
+-------+------+--------------+-------------+-----------------+-----------+---+-
---+
```

```
|      2|     3|             68|           46|                39|          56|  2|
3|
+-------+------+---------------+-------------+-----------------+------------+---+-
---+
```

Now we study if we have a balanced data set. What we want to assess is if we have a similar number of Adelie penguins and Gentoo penguins. Indeed, in the following chunk we see that we have a similar number, and then the data set is balanced and we can fit a model with it.

[ ]:
```
val frequencyDistribution = penguin.groupBy("species").count().
  ↪orderBy(desc("count"))
frequencyDistribution.show()
```

```
+-------+-----+
|species|count|
+-------+-----+
| Adelie|   44|
| Gentoo|   36|
+-------+-----+
```

[ ]:
```
frequencyDistribution: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] =
 [species: string, count: bigint]
```

Now we compute correlations between several continous variables. The correlation between "bill_length_mm" and "bill_depth_mm" is -0.49, meaning that when one of the variable increases the other decreases. It has a lot of sense because we are comparing the bill length and thr depth. If a penguin has a larger bill, then the depth should be smaller, otherwise the penguin would have a huge bill! On the other hand, the correlation between "body_mass_g" and "flipper_length_mm" is 0.89, which has sense again, because we are comparing the mass of the penguin with the flipper. And it is quite clear that, the heavier the penguin, the bigger the penguin and then bigger flipper.

[ ]:
```
val correlation1 = penguin.stat.corr("bill_length_mm", "bill_depth_mm")
val correlation2 = penguin.stat.corr("body_mass_g", "flipper_length_mm")

println(s"Correlation: $correlation1")
println(s"Correlation: $correlation2")
```

```
Correlation: -0.4873960332591598
Correlation: 0.8865533365292193
```

[ ]:
```
correlation1: Double = -0.4873960332591598
correlation2: Double = 0.8865533365292193
```

Finally, we check for missing values in all the variables of the data set. We can observe that we

have no missing values. So we can keep with our study on the data set without concern on this.

```
[ ]: penguin.select(penguin.columns.map(colName => sum(when(col(colName).isNull, 1).
       ↪otherwise(0)).alias(colName)): _*).show()
```

```
+-------+------+--------------+-------------+-----------------+-----------+---+-
---+
|species|island|bill_length_mm|bill_depth_mm|flipper_length_mm|body_mass_g|sex|y
ear|
+-------+------+--------------+-------------+-----------------+-----------+---+-
---+
|      0|     0|             0|            0|                0|          0|  0|
0|
+-------+------+--------------+-------------+-----------------+-----------+---+-
---+
```

Now, we create an instance for splitting our data into test and training data subsets. We choose to use the 70% of the data for training and the 30% of the data for test.

```
[ ]: val splits = penguin.randomSplit(Array(0.7, 0.3), seed = 1)
```

```
[ ]: splits: Array[org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]] =
     Array([species: string, island: string … 6 more fields], [species: string,
     island: string … 6 more fields])
```

```
[ ]: val penguin_train = splits(0).toDF
     penguin_train.show(5)
```

```
+-------+------+--------------+-------------+-----------------+-----------+-----
-+----+
|species|island|bill_length_mm|bill_depth_mm|flipper_length_mm|body_mass_g|
sex|year|
+-------+------+--------------+-------------+-----------------+-----------+-----
-+----+
| Adelie|Biscoe|          34.5|         18.1|              187|
2900|female|2008|
| Adelie|Biscoe|          35.7|         16.9|              185|
3150|female|2008|
| Adelie|Biscoe|          37.6|         19.1|              194|       3750|
male|2008|
| Adelie|Biscoe|          37.7|         16.0|              183|
3075|female|2009|
| Adelie|Biscoe|          38.2|         18.1|              185|       3950|
male|2007|
+-------+------+--------------+-------------+-----------------+-----------+-----
-+----+
```

```
only showing top 5 rows
```

[ ]:
```
penguin_train: org.apache.spark.sql.DataFrame = [species: string, island: string
… 6 more fields]
```

[ ]:
```
val penguin_test = splits(1).toDF
penguin_test.show(5)
```

```
+-------+------+--------------+-------------+-----------------+-----------+-----
-+----+
|species|island|bill_length_mm|bill_depth_mm|flipper_length_mm|body_mass_g|
sex|year|
+-------+------+--------------+-------------+-----------------+-----------+-----
-+----+
| Adelie|Biscoe|          37.8|         18.3|              174|
3400|female|2007|
| Adelie|Biscoe|          38.1|         17.0|              181|
3175|female|2009|
| Adelie|Biscoe|          42.2|         19.5|              197|       4275|
male|2009|
| Adelie|Biscoe|          42.7|         18.3|              196|       4075|
male|2009|
| Adelie| Dream|          36.0|         17.1|              187|
3700|female|2009|
+-------+------+--------------+-------------+-----------------+-----------+-----
-+----+
only showing top 5 rows
```

[ ]:
```
penguin_test: org.apache.spark.sql.DataFrame = [species: string, island: string
… 6 more fields]
```

If we take a look into the documentation, we see that for doing binary classifcation we must transofrm the variable to classify to $0, 1$. This can be done using the *stringIndexer* method. What we will do in the following chunks is prepare the training dataset for be trained. It is, we will create a new variable called *label* such that if the species of the penguin is "Adelie" then label is 0. And, if the species is "Gentoo", then the label is 1. Also, we will use the features engineering. And, to use these methods optimally, we will be using the useful *pipeline*, which allows us to use different methods in the same code line.

[ ]:
```
import org.apache.spark.ml.Pipeline
val labelindexer = new StringIndexer()
      .setInputCol("species")
      .setOutputCol("label")
val assembler = new VectorAssembler()
```

```
      .setInputCols(Array("bill_length_mm", "bill_depth_mm",␣
 ↪"flipper_length_mm", "body_mass_g", "year"))
      .setOutputCol("features")
val pipeline = new Pipeline()
      .setStages(Array(labelindexer, assembler))
```

[ ]:
```
import org.apache.spark.ml.Pipeline
labelindexer: org.apache.spark.ml.feature.StringIndexer = strIdx_fb1d9a03c303
assembler: org.apache.spark.ml.feature.VectorAssembler = VectorAssembler:
uid=vecAssembler_1a455dfe192c, handleInvalid=error, numInputCols=5
pipeline: org.apache.spark.ml.Pipeline = pipeline_5b90aeeb2226
```

Then, the training data set prepared for apply the generalised linear rergression is:

[ ]:
```
var traindF = pipeline.fit(penguin_train).transform(penguin_train)
traindF.show(5)
```

```
+-------+------+--------------+-------------+-----------------+-----------+-----
-+----+-----+--------------------+
|species|island|bill_length_mm|bill_depth_mm|flipper_length_mm|body_mass_g|
sex|year|label|            features|
+-------+------+--------------+-------------+-----------------+-----------+-----
-+----+-----+--------------------+
| Adelie|Biscoe|          34.5|         18.1|              187|
2900|female|2008|  0.0|[34.5,18.1,187.0,…|
| Adelie|Biscoe|          35.7|         16.9|              185|
3150|female|2008|  0.0|[35.7,16.9,185.0,…|
| Adelie|Biscoe|          37.6|         19.1|              194|       3750|
male|2008|  0.0|[37.6,19.1,194.0,…|
| Adelie|Biscoe|          37.7|         16.0|              183|
3075|female|2009|  0.0|[37.7,16.0,183.0,…|
| Adelie|Biscoe|          38.2|         18.1|              185|       3950|
male|2007|  0.0|[38.2,18.1,185.0,…|
+-------+------+--------------+-------------+-----------------+-----------+-----
-+----+-----+--------------------+
only showing top 5 rows
```

[ ]:
```
traindF: org.apache.spark.sql.DataFrame = [species: string, island: string … 8
more fields]
```

We can prepare in the same way the tets data:

[ ]:
```
var testdf = pipeline.fit(penguin_test).transform(penguin_test)
testdf.show(5)
```

```
+-------+------+--------------+-------------+-----------------+-----------+-----
-+----+-----+--------------------+
|species|island|bill_length_mm|bill_depth_mm|flipper_length_mm|body_mass_g|
sex|year|label|            features|
+-------+------+--------------+-------------+-----------------+-----------+-----
-+----+-----+--------------------+
| Adelie|Biscoe|          37.8|         18.3|              174|
3400|female|2007|  0.0|[37.8,18.3,174.0,…|
| Adelie|Biscoe|          38.1|         17.0|              181|
3175|female|2009|  0.0|[38.1,17.0,181.0,…|
| Adelie|Biscoe|          42.2|         19.5|              197|       4275|
male|2009|  0.0|[42.2,19.5,197.0,…|
| Adelie|Biscoe|          42.7|         18.3|              196|       4075|
male|2009|  0.0|[42.7,18.3,196.0,…|
| Adelie| Dream|          36.0|         17.1|              187|
3700|female|2009|  0.0|[36.0,17.1,187.0,…|
+-------+------+--------------+-------------+-----------------+-----------+-----
-+----+-----+--------------------+
only showing top 5 rows
```

[ ]:
```
testdf: org.apache.spark.sql.DataFrame = [species: string, island: string … 8
more fields]
```

Now, we create an instance of the generalized linear regression method, using the binomial regression and the logit funciton.

[ ]:
```
val glr = new GeneralizedLinearRegression()
            .setFamily("binomial")
            .setLink("logit")
            .setFeaturesCol("features")
            .setLabelCol("label")
```

[ ]:
```
glr: org.apache.spark.ml.regression.GeneralizedLinearRegression =
glm_5215fff0aa03
```

We train the data with the previous "estimator" created:

[ ]:
```
val glrModel = glr.fit(traindF)
```

[ ]:
```
glrModel: org.apache.spark.ml.regression.GeneralizedLinearRegressionModel =
GeneralizedLinearRegressionModel: uid=glm_5215fff0aa03, family=binomial,
link=logit, numFeatures=5
```

Now, we have our model trained. But now, what we will do is to use cross validation for assessing how good is our model classifying the penguins species.

[ ]:
```
val numFolds = 10
val MaxIter: Seq[Int] = Seq(100)
val RegParam: Seq[Double] = Seq(0.01)
val Tol: Seq[Double] = Seq(1e-4)
```

[ ]:
```
numFolds: Int = 10
MaxIter: Seq[Int] = List(100)
RegParam: Seq[Double] = List(0.01)
Tol: Seq[Double] = List(1.0E-4)
```

We define the parameter grid:

[ ]:
```
val paramGrid = new ParamGridBuilder()
      .addGrid(glrModel.maxIter, MaxIter)
      .addGrid(glrModel.regParam, RegParam)
      .addGrid(glrModel.tol, Tol)
      .build()
```

[ ]:
```
paramGrid: Array[org.apache.spark.ml.param.ParamMap] =
Array({
        glm_5215fff0aa03-maxIter: 100,
        glm_5215fff0aa03-regParam: 0.01,
        glm_5215fff0aa03-tol: 1.0E-4
})
```

Now, we use again the pipeline, which is very useful for do our code in a single line. In this case we define a pipeline with the objects `labelindexer`, `assembler` and `glr`. And then, we will pass the initial train data set, and the 0,1 labels will be created, also the features engineering, and in adittion the model will be trained with the estimator created before.

[ ]:
```
val pipeline2 = new Pipeline()
      .setStages(Array(labelindexer, assembler, glr))
```

[ ]:
```
pipeline2: org.apache.spark.ml.Pipeline = pipeline_d184cb1e0a4a
```

Here we import the necessary objects.

[ ]:
```
import org.apache.spark.ml.classification.{ BinaryLogisticRegressionSummary,␣
 ↪LogisticRegression, LogisticRegressionModel }
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.tuning.{ ParamGridBuilder, CrossValidator }
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
import org.apache.spark.ml.{Pipeline, PipelineModel}
```

[ ]:
```
import org.apache.spark.ml.classification.{BinaryLogisticRegressionSummary,
LogisticRegression, LogisticRegressionModel}
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.tuning.{ParamGridBuilder, CrossValidator}
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
import org.apache.spark.ml.{Pipeline, PipelineModel}
```

We create the binary classificator (remember that we are doing a binary classification, species 0 of penguin or species 1 of penguin).

[ ]:
```
val evaluator = new BinaryClassificationEvaluator()
      .setLabelCol("label")
      .setRawPredictionCol("prediction")
```

[ ]:
```
evaluator: org.apache.spark.ml.evaluation.BinaryClassificationEvaluator =
BinaryClassificationEvaluator: uid=binEval_2596200c0619,
metricName=areaUnderROC, numBins=1000
```

We declare the cross validator object with the different objects defined before.

[ ]:
```
val crossval = new CrossValidator()
      .setEstimator(pipeline2)
      .setEvaluator(evaluator)
      .setEstimatorParamMaps(paramGrid)
      .setNumFolds(numFolds)
```

[ ]:
```
crossval: org.apache.spark.ml.tuning.CrossValidator = cv_332298d102db
```

Now, it is time to fit the model using the initial train data set and the cross validator object.

[ ]:
```
val cvModel = crossval.fit(penguin_train)
```

[ ]:
```
cvModel: org.apache.spark.ml.tuning.CrossValidatorModel = CrossValidatorModel:
uid=cv_332298d102db, bestModel=pipeline_d184cb1e0a4a, numFolds=10
```

We do predictions with respect the test data set.

[ ]:
```
var predDF = cvModel.transform(penguin_test)
predDF.show(5)
```

```
+-------+------+--------------+-------------+-----------------+-----------+-----
-+----+-----+--------------------+----------+
|species|island|bill_length_mm|bill_depth_mm|flipper_length_mm|body_mass_g|
sex|year|label|            features|prediction|
```

```
+-------+------+-------------+------------+-----------------+-----------+-----
-+----+-----+--------------------+----------+
| Adelie|Biscoe|         37.8|        18.3|              174|
3400|female|2007|  0.0|[37.8,18.3,174.0,…|   1.0E-16|
| Adelie|Biscoe|         38.1|        17.0|              181|
3175|female|2009|  0.0|[38.1,17.0,181.0,…|   1.0E-16|
| Adelie|Biscoe|         42.2|        19.5|              197|       4275|
male|2009|  0.0|[42.2,19.5,197.0,…|   1.0E-16|
| Adelie|Biscoe|         42.7|        18.3|              196|       4075|
male|2009|  0.0|[42.7,18.3,196.0,…|   1.0E-16|
| Adelie| Dream|         36.0|        17.1|              187|
3700|female|2009|  0.0|[36.0,17.1,187.0,…|   1.0E-16|
+-------+------+-------------+------------+-----------------+-----------+-----
-+----+-----+--------------------+----------+
only showing top 5 rows
```

[ ]:
```
predDF: org.apache.spark.sql.DataFrame = [species: string, island: string … 9
more fields]
```

We can observe in the predictions that the predicted values are not exaclty 0 or 1. This is because these are the probabilities returned by a logistic regression, thus we have to choose a threshold.

In order to see more clear how good are the predictions, we select only the variables "species", "label" and "predicted_label". We print several observations of this data set filtered.

[ ]:
```
val result = predDF.select("species", "label", "prediction")
val resutDF = result.withColumnRenamed("prediction", "Predicted_label")
resutDF.show(100)
```

```
+-------+-----+--------------------+
|species|label|     Predicted_label|
+-------+-----+--------------------+
| Adelie|  0.0|              1.0E-16|
| Adelie|  0.0|              1.0E-16|
| Adelie|  0.0|              1.0E-16|
| Adelie|  0.0|              1.0E-16|
| Adelie|  0.0|              1.0E-16|
| Adelie|  0.0|              1.0E-16|
| Adelie|  0.0|              1.0E-16|
| Adelie|  0.0|              1.0E-16|
| Adelie|  0.0|              1.0E-16|
| Adelie|  0.0|              1.0E-16|
| Adelie|  0.0|              1.0E-16|
| Adelie|  0.0|              1.0E-16|
| Adelie|  0.0|              1.0E-16|
| Adelie|  0.0|              1.0E-16|
| Adelie|  0.0|9.109044006745272E-9|
| Gentoo|  1.0|  0.9999999999999999|
```

```
| Gentoo|  1.0|  0.9999999999999999|
| Gentoo|  1.0|  0.9999999999999999|
| Gentoo|  1.0|  0.9999999999999999|
| Gentoo|  1.0|  0.9999999999999999|
| Gentoo|  1.0|  0.9999999999999999|
| Gentoo|  1.0|  0.9999999999999999|
| Gentoo|  1.0|  0.9999999999999999|
+-------+-----+--------------------+
```

[ ]:
```
result: org.apache.spark.sql.DataFrame = [species: string, label: double … 1
more field]
resutDF: org.apache.spark.sql.DataFrame = [species: string, label: double … 1
more field]
```

We can observe that the predictions obtained are good. Notwithstanding we can see that there are some observations not well predicted. Now we want to compute different metrics for assess the quality of the model. First, we should transform the predicted values into 0 or 1. For this, we take as a threshold 0.5, because is the more natural one.

[ ]:
```
// Define the condition and transformation
val condition = col("prediction") < 0.5
val transformation = when(condition, 0.0).otherwise(1.0)

// Apply the transformation using withColumn
predDF = predDF.withColumn("prediction", transformation)

val result = predDF.select("species", "label", "prediction")
val resutDF = result.withColumnRenamed("prediction", "Predicted_label")
resutDF.show(100)
```

```
+-------+-----+---------------+
|species|label|Predicted_label|
+-------+-----+---------------+
| Adelie|  0.0|            0.0|
| Adelie|  0.0|            0.0|
| Adelie|  0.0|            0.0|
| Adelie|  0.0|            0.0|
| Adelie|  0.0|            0.0|
| Adelie|  0.0|            0.0|
| Adelie|  0.0|            0.0|
| Adelie|  0.0|            0.0|
| Adelie|  0.0|            0.0|
| Adelie|  0.0|            0.0|
| Adelie|  0.0|            0.0|
| Adelie|  0.0|            0.0|
| Adelie|  0.0|            0.0|
```

```
| Adelie|  0.0|            0.0|
| Gentoo|  1.0|            1.0|
| Gentoo|  1.0|            1.0|
| Gentoo|  1.0|            1.0|
| Gentoo|  1.0|            1.0|
| Gentoo|  1.0|            1.0|
| Gentoo|  1.0|            1.0|
| Gentoo|  1.0|            1.0|
| Gentoo|  1.0|            1.0|
+-------+-----+---------------+
```

[ ]:
```
condition: org.apache.spark.sql.Column = (prediction < 0.5)
transformation: org.apache.spark.sql.Column = CASE WHEN (prediction < 0.5) THEN
0.0 ELSE 1.0 END
predDF: org.apache.spark.sql.DataFrame = [species: string, island: string … 9
more fields]
result: org.apache.spark.sql.DataFrame = [species: string, label: double … 1
more field]
resutDF: org.apache.spark.sql.DataFrame = [species: string, label: double … 1
more field]
```

[ ]:
```
val tVSpDF = predDF.select("label", "prediction") // True vs predicted labels
val TC = predDF.count() //Total count

val tp = tVSpDF.filter($"prediction" === 0.0).filter($"label" ===␣
 ↪$"prediction").count() / TC.toDouble
val tn = tVSpDF.filter($"prediction" === 1.0).filter($"label" ===␣
 ↪$"prediction").count() / TC.toDouble
val fp = tVSpDF.filter($"prediction" === 1.0).filter(not($"label" ===␣
 ↪$"prediction")).count() / TC.toDouble
val fn = tVSpDF.filter($"prediction" === 0.0).filter(not($"label" ===␣
 ↪$"prediction")).count() / TC.toDouble

val MCC = (tp * tn - fp * fn) / math.sqrt((tp + fp) * (tp + fn) * (fp + tn) *␣
 ↪(tn + fn)) // Calculating Matthews correlation coefficient
```

[ ]:
```
tVSpDF: org.apache.spark.sql.DataFrame = [label: double, prediction: double]
TC: Long = 22
tp: Double = 0.6363636363636364
tn: Double = 0.36363636363636365
fp: Double = 0.0
fn: Double = 0.0
MCC: Double = 1.0
```

[ ]:
```
println("True positive rate: " + tp *100 + "%")
println("False positive rate: " + fp * 100 + "%")
println("True negative rate: " + tn * 100 + "%")
println("False negative rate: " + fn * 100 + "%")
println("Matthews correlation coefficient: " + MCC)
```

```
True positive rate: 63.63636363636363%
False positive rate: 0.0%
True negative rate: 36.36363636363637%
False negative rate: 0.0%
Matthews correlation coefficient: 1.0
```

It is time to interpret the values obtained in the computed metrics. Regarding the Matthews Correlation Coefficient (MCC), we have searched information on the web (see https://en.wikipedia.org/wiki/Phi_coefficient). We have found that "The MCC is in essence a correlation coefficient between the observed and predicted binary classifications; it returns a value between -1 and $+1$. A coefficient of $+1$ represents a perfect prediction, 0 no better than random prediction and -1 indicates total disagreement between prediction and observation." We obtained a MCC of 1, thus, MCC is indicating that our model is doing the classifications perfect!

If we study the other metric obtained: we have obtained a false negative rate of 0.0%, which means that no penguins were classified as species class 0, it is as Adelie, when actually the species was 1, it is Gentoo. In addition, we have obtained a false positive rate of 0%, meaning that no penguins were classified as Gentoo when they are actually Adelie. These two metrics indicates how many penguins are not well classified, and we see that all penguins are well classified for our model.

Now, we compute the accuracy:

[ ]:
```
val accuracy = evaluator.evaluate(predDF)
println("Classification accuracy: " + accuracy)
```

```
Classification accuracy: 1.0
```

[ ]:
```
accuracy: Double = 1.0
```

We can observe that we have obtained a nice accuracy, our model is classifying perfect the penguins!

We compute ohter metrics, we compute the area under the precision-recall curve and the area under the ROC curve. As our model is doing perfect the classification we expect a AUC of one.

[ ]:
```
val predictionAndLabels = predDF
      .select("prediction", "label")
      .rdd.map(x => (x(0).asInstanceOf[Double], x(1)
        .asInstanceOf[Double]))
```

[ ]:
```
predictionAndLabels: org.apache.spark.rdd.RDD[(Double, Double)] =
MapPartitionsRDD[2996] at map at <console>:54
```

[ ]:
```
val metrics = new BinaryClassificationMetrics(predictionAndLabels)
val areaUnderPR = metrics.areaUnderPR
println("Area under the precision-recall curve: " + areaUnderPR)
val areaUnderROC = metrics.areaUnderROC
println("Area under the receiver operating characteristic (ROC) curve: " +␣
 ↪areaUnderROC)
```

```
Area under the precision-recall curve: 1.0
Area under the receiver operating characteristic (ROC) curve: 1.0
```

[ ]:
```
metrics: org.apache.spark.mllib.evaluation.BinaryClassificationMetrics =
org.apache.spark.mllib.evaluation.BinaryClassificationMetrics@74edef6c
areaUnderPR: Double = 1.0
areaUnderROC: Double = 1.0
```

As we were expecting, as our model is doing perfect the classification we obtain area under PR of 1 and the same for the area under the ROC.

We finally fit another model, this time a Naive Bayes model, because is a model that I have never used before, and I think that is inetresting to fit this class of model and see how it behaves. In addition, I am interested in compare the results with the ones obtained with the binomial regression.

We will be using the same scheme than the used previously for fitting the Naive Bayes model.

[ ]:
```
import org.apache.spark.ml.classification.{ BinaryLogisticRegressionSummary,␣
 ↪NaiveBayes, NaiveBayesModel }

val nb = new NaiveBayes()
      .setLabelCol("label")
      .setFeaturesCol("features")

val pipeline = new Pipeline().setStages(Array(labelindexer, assembler, nb))

val paramGridnv = new ParamGridBuilder()
      .addGrid(nb.smoothing, Array(1e-2, 1e-4, 1e-6, 1e-8))
      .build()

val evaluator = new BinaryClassificationEvaluator()
      .setLabelCol("label")
      .setRawPredictionCol("prediction")

val crossval = new CrossValidator()
      .setEstimator(pipeline)
      .setEvaluator(evaluator)
      .setEstimatorParamMaps(paramGridnv)
      .setNumFolds(numFolds)
```

```
val cvModel = crossval.fit(penguin_train)
```

[ ]:
```
import org.apache.spark.ml.classification.{BinaryLogisticRegressionSummary,
NaiveBayes, NaiveBayesModel}
nb: org.apache.spark.ml.classification.NaiveBayes = nb_30ee377e5545
pipeline: org.apache.spark.ml.Pipeline = pipeline_11822bc50bbc
paramGridnv: Array[org.apache.spark.ml.param.ParamMap] =
Array({
        nb_30ee377e5545-smoothing: 0.01
}, {
        nb_30ee377e5545-smoothing: 1.0E-4
}, {
        nb_30ee377e5545-smoothing: 1.0E-6
}, {
        nb_30ee377e5545-smoothing: 1.0E-8
})
evaluator: org.apache.spark.ml.evaluation.BinaryClassificationEvaluator =
BinaryClassificationEvaluator: uid=binEval_8be04f12c147,
metricName=areaUnderROC, numBins=1000
crossval: org.apache.spark.ml.tuning.CrossValidator = cv_af3d8ee368e6
cvModel: org.apache.spark.ml.tuning.CrossValidatorModel = CrossValidatorModel:
uid=cv_af3d8ee368e6, b…
```

[ ]:
```
val predDF = cvModel.transform(penguin_test)
val result = predDF.select("label", "prediction", "probability")
val resutDF = result.withColumnRenamed("prediction", "Predicted_label")
resutDF.show(5)
```

```
+-----+---------------+--------------------+
|label|Predicted_label|         probability|
+-----+---------------+--------------------+
|  0.0|            0.0|[1.0,5.2530344100…|
|  0.0|            0.0|[1.0,1.7794999354…|
|  0.0|            0.0|[0.80895914643810…|
|  0.0|            0.0|[0.99999999968301…|
|  0.0|            0.0|[1.0,8.9705880274…|
+-----+---------------+--------------------+
only showing top 5 rows
```

[ ]:
```
predDF: org.apache.spark.sql.DataFrame = [species: string, island: string … 11
more fields]
result: org.apache.spark.sql.DataFrame = [label: double, prediction: double …
1 more field]
resutDF: org.apache.spark.sql.DataFrame = [label: double, Predicted_label:
double … 1 more field]
```

We observe that in this case we have obtained the predicted values direclty as integers, and then we don't need to choose a threshold. In addition, in this case we obtain also the probabilities, while in the generalized linear model fitted previously we did not have the probabilities.

We compte the same metrics than before, and we observe that we obtain the same results. Thus, the Naive Bayes model is working well, and at the same level than the generalized linear model.

[ ]:
```
val tVSpDF = predDF.select("label", "prediction") // True vs predicted labels
val TC = predDF.count()

val tp = tVSpDF.filter($"prediction" === 0.0).filter($"label" ===␣
 ↪$"prediction").count() / TC.toDouble
val tn = tVSpDF.filter($"prediction" === 1.0).filter($"label" ===␣
 ↪$"prediction").count() / TC.toDouble
val fp = tVSpDF.filter($"prediction" === 1.0).filter(not($"label" ===␣
 ↪$"prediction")).count() / TC.toDouble
val fn = tVSpDF.filter($"prediction" === 0.0).filter(not($"label" ===␣
 ↪$"prediction")).count() / TC.toDouble
val MCC = (tp * tn - fp * fn) / math.sqrt((tp + fp) * (tp + fn) * (fp + tn) *␣
 ↪(tn + fn))
val accuracy = evaluator.evaluate(tVSpDF)


println("True positive rate: " + tp *100 + "%")
println("False positive rate: " + fp * 100 + "%")
println("True negative rate: " + tn * 100 + "%")
println("False negative rate: " + fn * 100 + "%")
println("Matthews correlation coefficient: " + MCC)
println("Classification accuracy: " + accuracy)
```

```
True positive rate: 59.09090909090909%
False positive rate: 4.545454545454546%
True negative rate: 36.36363636363637%
False negative rate: 0.0%
Matthews correlation coefficient: 0.9085135251589957
Classification accuracy: 0.9642857142857143
```

[ ]:
```
tVSpDF: org.apache.spark.sql.DataFrame = [label: double, prediction: double]
TC: Long = 22
tp: Double = 0.5909090909090909
tn: Double = 0.36363636363636365
fp: Double = 0.045454545454545456
fn: Double = 0.0
MCC: Double = 0.9085135251589957
accuracy: Double = 0.9642857142857143
```

In this case, the model is not doing the classification perfect, but we are getting nice results also.

0.0.2 Continuous variable prediction

One of my great hobbies is the mountains. I am from Cardedeu, a village in Vallès Oriental located at the foothills of Montseny. I know perfectly all the paths and trails that surround my town, and I frequent a lot the mountains that form the natural park of Montseny. Due to my interest in mountains and forests I thought it would be interesting to use as a data set one that is related to this. In this part of the work we will use a data set downloaded from the National Forest Inventory of Catalonia. The NFI has a very interesting web where they upload a lot of data related to the natural environment in Catalonia, see the website on https://laboratoriforestal.creaf.cat/nfi_app/. This web allows us a very attractive visualization of the data in map format and also in table format. In addition, it allows us to download in csv format the data we want. What I have done has been to select data on forests in the Vallés Occidental, Vallés Oriental, Osona and Moianès. The objective of this second part of the work is to predict the continuous variable of tree density. In the data set that we have downloaded we have a total of 696 observations. It should be noted that the observations in the data set are from plots (not individual trees). The variables we have are plot ID (unique identifier for each plot), basal area in m^2/ha, total aerial biomass in t/ha, accumulated aerial CO2 in t/ha, dbh (diameter at breast height) in cm, tree density in trees/ha, dominant species in density, county, municipality, municipality id.

Our goal for predicting the density of trees, will be fit a linear model firstly, which is one of the simplest but most useful models that we have. And then we will try to fit a generalized linear model using the Gamma distribution. I am interested in study cases where the Gamma distribution works better for modeling, and I think that this part of the work is a good opportunity for compare the normal linear model and the generalized linear model and see which works better.

The first step, as always, is import the data from the csv file.

[ ]:
```
var trees = spark.read
    .format("csv")
    .option("header", "true")
    .option("inferSchema", "true")
    .csv("/Users/arnaugarcia/Desktop/Q1/bases de dades/zeppelin/data/arbres_cat.
 ↪csv")
```

[ ]:
```
trees: org.apache.spark.sql.DataFrame = [plot_id: string, basal_area: double …
8 more fields]
```

We print several observations of the data set:

[ ]:
```
trees.show(6)
```

```
+-------+----------+--------------------+----------------------+-----+-------+--
----------------------+------------+------------------+--------------------+
|plot_id|basal_area|aerial_biomass_total|accumulated_aerial_co2|  dbh|density|de
nsity_species_dominant|admin_region|admin_municipality|admin_municipality_id|
+-------+----------+--------------------+----------------------+-----+-------+--
----------------------+------------+------------------+--------------------+
```

```
|P_01685|     42.73|               145.4|                272.71|21.86|1138.84|
Pinus nigra|     Moianès|              Moià|                81385|
|P_00480|     22.31|               88.69|                165.49|20.35| 686.13|
Pinus sylvestris|       Osona|            Alpens|                80044|
|P_00499|     14.64|               57.67|                102.97|17.98| 576.49|
Quercus humilis|       Osona|           Alpens|                80044|
|P_00520|     21.33|               97.69|                182.39|21.57| 583.57|
Pinus sylvestris|       Osona|            Alpens|                80044|
|P_00521|     17.43|               64.88|                120.17|17.99| 686.13|
Pinus sylvestris|       Osona|            Alpens|                80044|
|P_00535|     10.87|               36.08|                 67.31|15.89|  548.2|
Pinus sylvestris|       Osona|             Lluçà|                81094|
+-------+----------+--------------------+----------------------+-----+-------+--
----------------------+------------+------------------+---------------------+
only showing top 6 rows
```

Now, it is time to develop an exploratory analysis, compute some statistics and see if there are missing values in our data set. We will be using similar code than in the first part. First, we

[ ]:
```
val summaryStats = trees.describe()
summaryStats.show()
```

```
+-------+-------+------------------+--------------------+----------------------
+------------------+-----------------+-----------------------+--------------+-
-------------------+---------------------+
|summary|plot_id|        basal_area|aerial_biomass_total|accumulated_aerial_co2|
dbh|           density|density_species_dominant|   admin_region|
admin_municipality|admin_municipality_id|
+-------+-------+------------------+--------------------+----------------------
+------------------+-----------------+-----------------------+--------------+-
-------------------+---------------------+
|  count|    696|               696|                 696|                   696|
696|               696|                    696|            696|
696|                  696|
|   mean|   NULL|18.688433908045976|   87.34110632183904|
157.70793103448278|18.639727011494255|832.2389655172417|
NULL|           NULL|                NULL|     87384.6393678161|
| stddev|   NULL| 9.877297419153532|  59.170255531285314|    106.96388027499285|
6.162784553312723|591.2291099467486|                   NULL|           NULL|
NULL|    21473.42759799767|
|    min|P_00480|              0.45|                1.06|                  1.97|
7.63|             5.09|          Acer campestre|        Moianès|
Aiguafreda|                80044|
|    max|P_08456|             56.93|               459.2|                859.83|
60.25|          3388.23|             Ulmus minor|Vallès Oriental|les Masies de
Vol…|               172207|
+-------+-------+------------------+--------------------+----------------------
```

```
+-----------------+----------------+----------------------+--------------+-
------------------+--------------------+
```

```
[ ]: summaryStats: org.apache.spark.sql.DataFrame = [summary: string, plot_id: string
     … 9 more fields]
```

For the density variable, which is the one of most interest because is the one that we will try to predict, we see that the minimum and the maximum seems to be very extreme values, in the sense that are values very far from the mean. This gives us an idea that the predictions we are going to make are not simple, in the sense that there is a very large range of values of the density variable and therefore many different values to predict.

Now we count the distinct values for each variable.

```
[ ]: trees.select(trees.columns.map(plot_id => countDistinct(col(plot_id)).
       ↪alias(plot_id)): _*).show()
```

```
+-------+----------+-------------------+---------------------+---+-------+----
--------------------+------------+------------------+---------------------+
|plot_id|basal_area|aerial_biomass_total|accumulated_aerial_co2|dbh|density|dens
ity_species_dominant|admin_region|admin_municipality|admin_municipality_id|
+-------+----------+-------------------+---------------------+---+-------+----
--------------------+------------+------------------+---------------------+
|    696|       635|                687|                   688|573|    548|
34|           4|               102|                  102|
+-------+----------+-------------------+---------------------+---+-------+----
--------------------+------------+------------------+---------------------+
```

As we were expecting, we obtain 696 different values for the variable plot_id, which is exacltly the number of observations (remember that the plot_id is a unique id for each plot). The different values for the admin_region is 4, which is also what we were expecting because we have selected only four catalan "comarques".

In the following chunk code we compute correlations between several variables. In this case we are hugely interested in how behaves the density variable with respect other continuous variables.

```
[ ]: val correlation1 = trees.stat.corr("density", "basal_area")
     val correlation2 = trees.stat.corr("density", "aerial_biomass_total")
     val correlation3 = trees.stat.corr("density", "dbh")

     println(s"Correlation between density and basal area: $correlation1")
     println(s"Correlation between density and aerial biomass: $correlation2")
     println(s"Correlation between density and dbh: $correlation3")
```

```
Correlation between density and basal area: 0.6361732127094407
Correlation between density and aerial biomass: 0.4122166574338511
Correlation between density and dbh: -0.5200283339124606
```

[ ]:
```
correlation1: Double = 0.6361732127094407
correlation2: Double = 0.4122166574338511
correlation3: Double = -0.5200283339124606
```

Let us interpret the correlations obtained: - The correlation between density and basal area is 0.64, which indicates that the greater basal area the greater the density of trees. This makes sense, the basal area is computed doing the sum of cross sectional area of tree trunks, divided by the surface area of the plot. Then, the more basal area, the more the sum of the cross sectional area of tree trunks (obiously the surface area does not change), which may indicate more trees, and then more density, or thicker trees. - The correlation between aerial biomass and density is 0.41, which indicates that the greater the aerial biomass the greater the density.The arial biomass is the sum of total amount of organic matter from leaves, branches, trunk and bark. Thus, makes sense that the more organic matter, the more trees and then more density. - The correlation between density and dbh is -0.52, which indicates that the lower the dbh, the lower the density. It makes a lot of sense, because the dbh is the diameter at the breast height of the tree, then if the dbh is big, there is less space for other trees and then there is less density of trees.

The next step, is check if there are missing values in our data set.

[ ]:
```
trees.select(trees.columns.map(colName => sum(when(col(colName).isNull, 1).
  ↪otherwise(0)).alias(colName)): _*).show()
```

```
+-------+----------+--------------------+----------------------+---+-------+----
--------------------+------------+------------------+--------------------+
|plot_id|basal_area|aerial_biomass_total|accumulated_aerial_co2|dbh|density|dens
ity_species_dominant|admin_region|admin_municipality|admin_municipality_id|
+-------+----------+--------------------+----------------------+---+-------+----
--------------------+------------+------------------+--------------------+
|      0|         0|                   0|                     0|  0|      0|
0|           0|                 0|                    0|
+-------+----------+--------------------+----------------------+---+-------+----
--------------------+------------+------------------+--------------------+
```

We can see that there are no missing values, we are fortunate that the NFI databases are of good quality. Then, we can keep with our analysis.

In the following chunk we use the feature engineering, as we did in the previous section. This procedure is always mandatory if one wants to fit models using spark. We use the method `VectorAssembler` as we did in the previoius section, and we include in the correspondent array the independent variables of our model (it is, the variables with respect we will be modeling the density of trees).

When we try to include in our vector assembler variables that are not numerical, ie variable of string type, we obtain an error. Then, what we should do is convert these variables into numerical. For this purpose, the `StringIndexer` method used in the previous section will be useful. Then, in the following chunk we convert to numerical the categorical variables putting indexes.

We will use the useful pipeline in order to use at the same time the `StringIndexer` method and

the `VectorAssembler` method. We define use twice the `StringIndexer` method, for convert the categorical variables density_species_dominant and admin_region into categorical but numerical variables. Notice that we have the admin_municipality_id, which is a numerical id of the different municipalities, and then we don't need to transform this categorical variable. Then, with a pipeline we gather the different operations to our dataset.

[ ]:
```
import org.apache.spark.ml.Pipeline
val labelindexer1 = new StringIndexer()
      .setInputCol("density_species_dominant")
      .setOutputCol("species_index")
val labelindexer2 = new StringIndexer()
      .setInputCol("admin_region")
      .setOutputCol("comarca_index")
val assembler = new VectorAssembler()
      .setInputCols(Array("basal_area", "aerial_biomass_total",␣
 ↪"accumulated_aerial_co2", "dbh", "species_index", "comarca_index",␣
 ↪"admin_municipality_id"))
      .setOutputCol("features")
val pipeline_trees = new Pipeline()
      .setStages(Array(labelindexer1, labelindexer2, assembler))
```

[ ]:
```
import org.apache.spark.ml.Pipeline
labelindexer1: org.apache.spark.ml.feature.StringIndexer = strIdx_3d7d902674ff
labelindexer2: org.apache.spark.ml.feature.StringIndexer = strIdx_8d7e11b183a4
assembler: org.apache.spark.ml.feature.VectorAssembler = VectorAssembler:
uid=vecAssembler_91806c22242d, handleInvalid=error, numInputCols=7
pipeline_trees: org.apache.spark.ml.Pipeline = pipeline_a0fe77d6e63d
```

[ ]:
```
var trees_format = pipeline_trees.fit(trees).transform(trees)
trees_format.show(5)
```

```
+-------+---------+------------------+--------------------+-----+-------+--
----------------------+-----------+-----------------+--------------------+---
---------+-------------+--------------------+
|plot_id|basal_area|aerial_biomass_total|accumulated_aerial_co2|  dbh|density|de
nsity_species_dominant|admin_region|admin_municipality|admin_municipality_id|spe
cies_index|comarca_index|            features|
+-------+---------+------------------+--------------------+-----+-------+--
----------------------+-----------+-----------------+--------------------+---
---------+-------------+--------------------+
|P_01685|     42.73|             145.4|               272.71|21.86|1138.84|
Pinus nigra|     Moianès|              Moià|                81385|          8.0|
2.0|[42.73,145.4,272…|
|P_00480|     22.31|             88.69|               165.49|20.35| 686.13|
Pinus sylvestris|       Osona|            Alpens|                80044|
2.0|          0.0|[22.31,88.69,165…|
|P_00499|     14.64|             57.67|               102.97|17.98| 576.49|
```

```
Quercus humilis|       Osona|            Alpens|                80044|
1.0|          0.0|[14.64,57.67,102…|
|P_00520|     21.33|               97.69|               182.39|21.57| 583.57|
Pinus sylvestris|       Osona|            Alpens|                80044|
2.0|          0.0|[21.33,97.69,182…|
|P_00521|     17.43|               64.88|               120.17|17.99| 686.13|
Pinus sylvestris|       Osona|            Alpens|                80044|
2.0|          0.0|[17.43,64.88,120…|
+-------+----------+--------------------+----------------------+-----+-------+--
----------------------+------------+------------------+----------------------+---
---------+------------+--------------------+
only showing top 5 rows
```

```
[ ]: trees_format: org.apache.spark.sql.DataFrame = [plot_id: string, basal_area:
     double … 11 more fields]
```

In order to work with the spark methods we need only two columns, one named label, with the response variable (density in our case), and other column with the features created.

```
[ ]: trees_format = trees_format.select($"density".alias("label"), $"features")
     trees_format.show(3)
```

```
+-------+--------------------+
|  label|            features|
+-------+--------------------+
|1138.84|[42.73,145.4,272…|
| 686.13|[22.31,88.69,165…|
| 576.49|[14.64,57.67,102…|
+-------+--------------------+
only showing top 3 rows
```

```
[ ]: trees_format: org.apache.spark.sql.DataFrame = [label: double, features: vector]
```

The next step is split the data into a test and train dsta sets. First we create an instance for random split de data, and then we divide the data set using the instance.

```
[ ]: val splits = trees_format.randomSplit(Array(0.7, 0.3), seed = 2)
```

```
[ ]: splits: Array[org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]] =
     Array([label: double, features: vector], [label: double, features: vector])
```

```
[ ]: val trees_train = splits(0).toDF
     trees_train.show(3)
```

```
+-----+--------------------+
|label|            features|
+-----+--------------------+
| 5.09|[1.45,10.16,17.92…|
|10.19|[2.44,15.16,27.32…|
|14.15|[0.9,3.51,6.23,28…|
+-----+--------------------+
only showing top 3 rows
```

[ ]:
```
trees_train: org.apache.spark.sql.DataFrame = [label: double, features: vector]
```

[ ]:
```
val trees_test = splits(1).toDF
trees_test.show(3)
```

```
+-----+--------------------+
|label|            features|
+-----+--------------------+
|14.15|[1.02,3.27,6.15,3…|
|28.29|[2.34,9.62,17.06,…|
|31.83|[0.94,3.24,5.68,1…|
+-----+--------------------+
only showing top 3 rows
```

[ ]:
```
trees_test: org.apache.spark.sql.DataFrame = [label: double, features: vector]
```

Now, it is time to create a normal linear model and to fit this model to our train data set. We want to use the normal linear model: $density = \beta_0 + \beta_1 basal - area + \beta_2 aerial - biomass - total + \beta_3 accumulated - aerial - co2 + \beta_4 dbh + \beta_5 species - index + \beta_6 comarca - index + \beta_7 admin - municipality - id$.

We import the necessary items:

[ ]:
```
import org.apache.spark.ml.regression.LinearRegressionModel
import org.apache.spark.ml.regression.LinearRegression
```

[ ]:
```
import org.apache.spark.ml.regression.LinearRegressionModel
import org.apache.spark.ml.regression.LinearRegression
```

Now we create a normal linear model estimator:

[ ]:
```
val lr = new LinearRegression()
            .setMaxIter(10000)
```

```
[ ]: lr: org.apache.spark.ml.regression.LinearRegression = linReg_593277aa7317
```

And now we fit the model with the training data set:

```
[ ]: val linear_model = lr.fit(trees_train)
```

```
[ ]: linear_model: org.apache.spark.ml.regression.LinearRegressionModel =
     LinearRegressionModel: uid=linReg_593277aa7317, numFeatures=7
```

The coefficients and the intercept of our normal linear model are:

```
[ ]: println(s"Intercept: ${linear_model.intercept}")
     println(s"Coefficients: ${linear_model.coefficients}")
```

```
Intercept: 436.1380936496392
Coefficients: [79.67746000024212,68.90600838677362,-42.21008818547648,-
30.2181412264526,-1.2128290989741863,16.61569298529879,0.001271400606837391]
```

We now compute two important metrics for assess thr quality of our model, the MAE (Mean Absolute Error) and RMSE(Root Mean Squared Error) over the training set.

```
[ ]: println(s"MAE: ${linear_model.summary.meanAbsoluteError}")
     println(s"RMSE: ${linear_model.summary.rootMeanSquaredError}")
```

```
MAE: 184.30215348400958
RMSE: 264.62229044835624
```

Now, in order to see if our model is woking good we apply the fitted model to the test data set, and we compare the predicter values obtained with the true values.

```
[ ]: import org.apache.spark.mllib.evaluation.RegressionMetrics
     val test = linear_model.transform(trees_test)
     test.show(40)
```

```
+------+--------------------+-------------------+
| label|            features|         prediction|
+------+--------------------+-------------------+
| 14.15|[1.02,3.27,6.15,3…| -287.8088538577296|
| 28.29|[2.34,9.62,17.06,…| -278.5035563341956|
| 31.83|[0.94,3.24,5.68,1…|  45.00945689438515|
| 51.07|[2.15,10.47,19.26…|  -60.0618515837682|
| 61.68|[4.12,13.53,25.13…| -93.56367616458823|
| 84.88|[7.13,48.62,84.81…|-58.103598085878616|
|105.68|[5.57,27.2,49.76,…|-24.198375826501092|
|109.64|[2.26,7.39,13.75,…|  157.0179698908825|
|109.64|[2.81,14.11,24.68…| 165.15342718236633|
|119.83|[5.13,28.34,51.98…|-11.262788988199077|
|127.32|[0.62,1.3,2.28,7…|  367.3287852698767|
```

```
|127.32|[0.68,1.15,2.15,8…|  380.2689991386181|
|135.95|[10.49,49.61,91.4…| 15.027471146861842|
|152.08|[5.55,15.02,28.04…| 191.86251262210743|
|159.15|[1.26,3.95,7.12,1…|  360.2031483966067|
| 173.3|[1.75,6.64,11.63,…| 319.19352606987536|
|179.95|[6.45,24.96,46.53…| 188.24452211132697|
|189.01|[8.68,46.01,83.49…| 195.64882942157442|
|195.51|[22.24,190.92,337…|  75.68021218158441|
|198.06|[6.01,23.01,42.15…|  259.3325837449737|
|204.71|[10.4,56.76,102.5…| 234.29389775132015|
|204.71|[13.5,88.99,157.4…| 196.12996525149686|
|211.78|[9.24,43.43,76.83…| 271.61301627824173|
|229.89|[9.55,44.92,79.4,…|  349.2803375131356|
|236.96|[4.01,13.55,24.89…|  329.7561395382592|
| 238.1|[13.81,57.88,106…| 344.33867218767216|
|247.57|[7.73,25.55,47.46…|  338.5002470272436|
|250.69|[10.9,36.62,68.57…|  376.9799898973318|
|270.35|[7.75,42.11,74.9,…|  335.1948399760164|
| 272.9|[26.84,216.83,383…|  392.1376256844703|
|282.94|[7.92,34.02,60.24…| 400.53842628905034|
|298.65|[10.28,40.94,74.1…|   417.939596806598|
|300.48|[20.66,122.09,222…|  345.6584517081412|
|318.31|[5.6,21.38,37.52,…|  455.6156823800809|
|319.44|[15.16,71.46,129…|  447.5008890561664|
|324.96|[5.93,29.65,53.78…|  373.3220068294725|
|328.92|[6.59,26.64,46.98…| 451.85002088119523|
|335.99|[11.39,38.0,70.45…| 494.89649855330765|
|347.03|[45.83,365.18,670…| -81.53105112642169|
|357.21|[16.12,68.17,126…| 486.38896535496167|
+------+------------------+------------------+
only showing top 40 rows
```

[ ]:
```
import org.apache.spark.mllib.evaluation.RegressionMetrics
test: org.apache.spark.sql.DataFrame = [label: double, features: vector … 1
more field]
```

It seems thar there are bad predictions, we obtain different negative predictions! Which have no sense. On the other hand, it seems that there are a lot of values quite well predicted.

[ ]:
```
val predictions = test.select("prediction").rdd.map(_.getDouble(0))
predictions.collect()(0)
predictions.collect()(1)
predictions.collect()(2)
```

[ ]:
```
predictions: org.apache.spark.rdd.RDD[Double] = MapPartitionsRDD[254] at map at
<console>:49
```

```
res35: Double = 45.00945689438515
```

[ ]:
```
val labels = test.select("label").rdd.map(_.getDouble(0))
labels.collect()(0)
labels.collect()(1)
labels.collect()(2)
```

[ ]:
```
labels: org.apache.spark.rdd.RDD[Double] = MapPartitionsRDD[275] at map at
<console>:49
res38: Double = 31.83
```

[ ]:
```
predictions.zip(labels).toDF.show(7)
```

```
+------------------+------+
|                _1|    _2|
+------------------+------+
| -287.8088538577296| 14.15|
| -278.5035563341956| 28.29|
|  45.00945689438515| 31.83|
|  -60.0618515837682| 51.07|
| -93.56367616458823| 61.68|
|-58.103598085878616| 84.88|
|-24.198375826501092|105.68|
+------------------+------+
only showing top 7 rows
```

We see now the MAE and the RMSE for the test data set:

[ ]:
```
println(s"MAE: ${new RegressionMetrics(predictions.zip(labels)).
  ↪meanAbsoluteError}")
println(s"RMSE: ${new RegressionMetrics(predictions.zip(labels)).
  ↪rootMeanSquaredError}")
```

```
MAE: 198.62821943451837
RMSE: 265.5134459147396
```

Thus, we have: - Train data set: $MAE = 184.30215348400958$ and $RMSE = 264.62229044835624$. - Test data set: $MAE = 198.62821943451837$ and $RMSE = 265.5134459147396$.

We can observe that we are obtaining very similar quantity, which is a good indication.

Now, we fit a generalized linear model using the Gamma distribution and the logarithm as a link function. We start creating an estimator of this glm:

[ ]:
```
val gamma = new GeneralizedLinearRegression()
            .setFamily("gamma")
            .setLink("log")
```

```
            .setFeaturesCol("features")
            .setLabelCol("label")
```

[ ]:
```
gamma: org.apache.spark.ml.regression.GeneralizedLinearRegression =
glm_eb557c9c3874
```

We fit the model with the training data set:

[ ]:
```
val gamma_model = gamma.fit(trees_train)
```

[ ]:
```
gamma_model: org.apache.spark.ml.regression.GeneralizedLinearRegressionModel =
GeneralizedLinearRegressionModel: uid=glm_eb557c9c3874, family=gamma, link=log,
numFeatures=7
```

Looking at the spark documentation (see https://spark.apache.org/docs/latest/ml-classification-regression.html#generalized-linear-regression) we have found that we can compute the following quantities related with the fitted model that are key.

[ ]:
```
val summary = gamma_model.summary
println(s"Coefficient Standard Errors: ${summary.coefficientStandardErrors.
  ↪mkString(",")}")
println(s"T Values: ${summary.tValues.mkString(",")}")
println(s"P Values: ${summary.pValues.mkString(",")}")
println(s"Dispersion: ${summary.dispersion}")
println(s"Null Deviance: ${summary.nullDeviance}")
println(s"Residual Degree Of Freedom Null: ${summary.
  ↪residualDegreeOfFreedomNull}")
println(s"Deviance: ${summary.deviance}")
println(s"Residual Degree Of Freedom: ${summary.residualDegreeOfFreedom}")
println(s"AIC: ${summary.aic}")
println("Deviance Residuals: ")
summary.residuals().show()
```

```
Coefficient Standard Errors: 0.0035617171613868213,0.005835937380874954,0.003337
727323074195,0.0023225515860912185,0.0022646874176271595,0.01116777140704118,5.7
68325951989856E-7,0.07487153599031958
T Values: 21.340519100129626,1.6091380258261347,-1.9366082777667895,-
37.68955192642184,-2.5703181894336073,-
0.7661312030136829,0.4168110627069901,91.9416285111697
P Values: 0.0,0.10824089180329333,0.053773725874546 75,0.0,0.0104597281203902,0.
44397326489535405,0.6770021704955014,0.0
Dispersion: 0.06650579221999973
Null Deviance: 317.2640840876833
Residual Degree Of Freedom Null: 489
Deviance: 45.62673842741864
Residual Degree Of Freedom: 482
```

```
AIC: 6529.7029807387335
Deviance Residuals:
+--------------------+
|   devianceResiduals|
+--------------------+
|-0.10594256040829246|
| 0.27291385727787965|
| -1.3880137980016374|
| -1.2988922107182197|
| -0.5049597804949391|
| -1.1070951774037816|
|  -1.008568076141114|
| -0.3574987988977449|
| -1.6557142388872728|
|  -1.545992593853271|
| -1.4993919355725527|
| -1.4664576860933733|
|  -0.538423818319976|
| -0.4571537823254411|
| -1.0126835531177691|
| -0.7159472902353702|
|  -1.145964024156902|
|  0.5209147191801561|
|-0.01349867274058…|
| -0.8368457734836718|
+--------------------+
only showing top 20 rows
```

[ ]:
```
summary:
org.apache.spark.ml.regression.GeneralizedLinearRegressionTrainingSummary =
Coefficients:
             Feature Estimate Std Error  T Value P Value
         (Intercept)   6.8838    0.0749  91.9416  0.0000
          basal_area   0.0760    0.0036  21.3405  0.0000
aerial_biomass_total   0.0094    0.0058   1.6091  0.1082
accumulated_aeria…  -0.0065    0.0033  -1.9366  0.0534
                 dbh  -0.0875    0.0023 -37.6896  0.0000
       species_index  -0.0058    0.0023  -2.5703  0.0105
       comarca_index  -0.0086    0.0112  -0.7661  0.4440
admin_municipalit…   0.0000    0.0000   0.4168  0.6770

(Dispersion parameter for gamma family taken to be 0.0665)
   Null deviance: 317.2641 on 482 degrees of freedom
Residual deviance: 45.6267 on 482 degrees of freedom
AIC: 6529.7030
```

We can observe in the previous output thay almost all the variables are statistically significant (pvalue less than $0,05$). We can observe other key quantities as the AIC, the model parameter estimation, etc.

Now, we apply the model to the test data and we see how are the predictions of our model.

[ ]:
```
val test_gamma = gamma_model.transform(trees_test)
test_gamma.show(40)
```

```
+------+------------------+-----------------+
| label|          features|       prediction|
+------+------------------+-----------------+
| 14.15|[1.02,3.27,6.15,3…| 70.32148138626481|
| 28.29|[2.34,9.62,17.06,…| 66.50290470490945|
| 31.83|[0.94,3.24,5.68,1…|191.18264932193597|
| 51.07|[2.15,10.47,19.26…| 140.0210126492307|
| 61.68|[4.12,13.53,25.13…|  98.0861011068805|
| 84.88|[7.13,48.62,84.81…| 87.07983175943683|
|105.68|[5.57,27.2,49.76,…| 147.3422740498355|
|109.64|[2.26,7.39,13.75,…| 277.7888641418412|
|109.64|[2.81,14.11,24.68…| 245.0227338565646|
|119.83|[5.13,28.34,51.98…|153.97179476382135|
|127.32|[0.62,1.3,2.28,7…| 488.4773303714757|
|127.32|[0.68,1.15,2.15,8…|486.32064862724854|
|135.95|[10.49,49.61,91.4…|121.08474346719352|
|152.08|[5.55,15.02,28.04…| 211.5428442679996|
|159.15|[1.26,3.95,7.12,1…| 440.0582092626804|
| 173.3|[1.75,6.64,11.63,…| 412.8557085151365|
|179.95|[6.45,24.96,46.53…|221.12802561786668|
|189.01|[8.68,46.01,83.49…|201.84001739369108|
|195.51|[22.24,190.92,337…| 118.2755082414303|
|198.06|[6.01,23.01,42.15…|256.95923680484566|
|204.71|[10.4,56.76,102.5…|202.73118624502962|
|204.71|[13.5,88.99,157.4…|159.32125399343275|
|211.78|[9.24,43.43,76.83…| 192.7357631223556|
|229.89|[9.55,44.92,79.4,…|249.37259223201843|
|236.96|[4.01,13.55,24.89…| 352.6921710155192|
| 238.1|[13.81,57.88,106…|217.85054064083766|
|247.57|[7.73,25.55,47.46…| 282.5334331255303|
|250.69|[10.9,36.62,68.57…|256.55816523595274|
|270.35|[7.75,42.11,74.9,…| 305.8156537180321|
| 272.9|[26.84,216.83,383…|214.47292135559684|
|282.94|[7.92,34.02,60.24…| 322.7872490605809|
|298.65|[10.28,40.94,74.1…| 314.9108149924939|
|300.48|[20.66,122.09,222…| 266.8434849495317|
|318.31|[5.6,21.38,37.52,…|387.47533435210863|
|319.44|[15.16,71.46,129…| 276.5101783932404|
|324.96|[5.93,29.65,53.78…|374.27770227449736|
```

```
|328.92|[6.59,26.64,46.98…| 381.2856420318415|
|335.99|[11.39,38.0,70.45…|339.83582796531334|
|347.03|[45.83,365.18,670…| 345.4722622862983|
|357.21|[16.12,68.17,126…| 338.2946230689163|
+------+--------------------+------------------+
only showing top 40 rows
```

[ ]:
```
test_gamma: org.apache.spark.sql.DataFrame = [label: double, features: vector
… 1 more field]
```

We can see that in this case we don't obtain negative predictions. Thus, this is an argument in favour of the gamma model versus the normal linear model. In addition, we can observe that the predictions are quite good, and seems to be better than the predictions of the normal linear model in general.

In order to compare the models we compute the RMSE and the MAE of the test data set as we did before, and compare the values obtained for the gamma model and for the normal linear model.

[ ]:
```
val predictions = test_gamma.select("prediction").rdd.map(_.getDouble(0))
predictions.collect()(0)
predictions.collect()(1)
predictions.collect()(2)
```

[ ]:
```
predictions: org.apache.spark.rdd.RDD[Double] = MapPartitionsRDD[369] at map at
<console>:52
res47: Double = 191.18264932193597
```

[ ]:
```
val labels = test_gamma.select("label").rdd.map(_.getDouble(0))
labels.collect()(0)
labels.collect()(1)
labels.collect()(2)
```

[ ]:
```
labels: org.apache.spark.rdd.RDD[Double] = MapPartitionsRDD[376] at map at
<console>:52
res48: Double = 31.83
```

[ ]:
```
predictions.zip(labels).toDF.show(20)
```

```
+------------------+------+
|                _1|    _2|
+------------------+------+
| 70.32148138626481| 14.15|
| 66.50290470490945| 28.29|
|191.18264932193597| 31.83|
| 140.0210126492307| 51.07|
```

```
|  98.0861011068805| 61.68|
| 87.07983175943683| 84.88|
| 147.3422740498355|105.68|
| 277.7888641418412|109.64|
| 245.0227338565646|109.64|
|153.97179476382135|119.83|
| 488.4773303714757|127.32|
|486.32064862724854|127.32|
|121.08474346719352|135.95|
| 211.5428442679996|152.08|
| 440.0582092626804|159.15|
| 412.8557085151365| 173.3|
|221.12802561786668|179.95|
|201.84001739369108|189.01|
| 118.2755082414303|195.51|
|256.95923680484566|198.06|
+------------------+------+
only showing top 20 rows
```

[ ]:
```
println(s"MAE: ${new RegressionMetrics(predictions.zip(labels)).
  ↪meanAbsoluteError}")
println(s"RMSE: ${new RegressionMetrics(predictions.zip(labels)).
  ↪rootMeanSquaredError}")
```

```
MAE: 165.03570518042602
RMSE: 324.3085890531792
```

If we compare the quantities obtained with the ones obtained in the case of the normal linear model, we can see that we have obtained less MAE, indicating that the gamma model is better. But, on the other hand, we have obtained a higher RMSE, indicating that the normal linear model is better.

Thus, it is not easy to decide which of the two models is better. Notwithstanding, looking at the results obtained and above all, taking into account that the gamma model does not return negative values, I think I would choose the gamma model with link function the logarithm before the normal linear model.

Discussion: Here, you would then have to chose which model works best and actually implement it into a certain problem. Just argue which one (for both examples) you would use and why (and why not the others?).

In the case of the prediction of the categorical variable, the two models used (Naive Bayes and binomial glm) classify well. However, I think that in this case, i.e., to solve the problem of classifying penguin species given the dataset we have, I would stick with Binomial GLM with threshold 0.5, becaus the model is classifying the penguins with no fail. On the other hand Naive Bayes is interesting because this model gives us not only the predictions but also the probabilities, which gives us extra information that the glm binomial model does not give us.

For the models that we have used to predict a continuous variable, I think I would choose the

gamma model because of what I said before, the fact that the normal linear model gives negative predictions makes me choose the gamma model. Well, despite the fact that the models behave similarly in terms of MAE and RMSE metrics, predicting tree density values in a forest with negative values is quite serious. On the other hand, I think that is interesing to study the normal linear model but with transformations in the variables. It is usual to model with normal linear models with transformations, and doing the right transformations can give us good results. Thus, compare the normal linear model with the Gamma regression model adding transformations would be an interesting topic for future research.