Matthieu Hanania

Karis Gwet

PARIS PANTHÉON-ASSAS UNIVERSITÉ

Big Data Processing

Data Frames, Machine Learning and

NLP with Spark

Table of content

I. Movielens dataset analysis

A. Building an AWS cluster

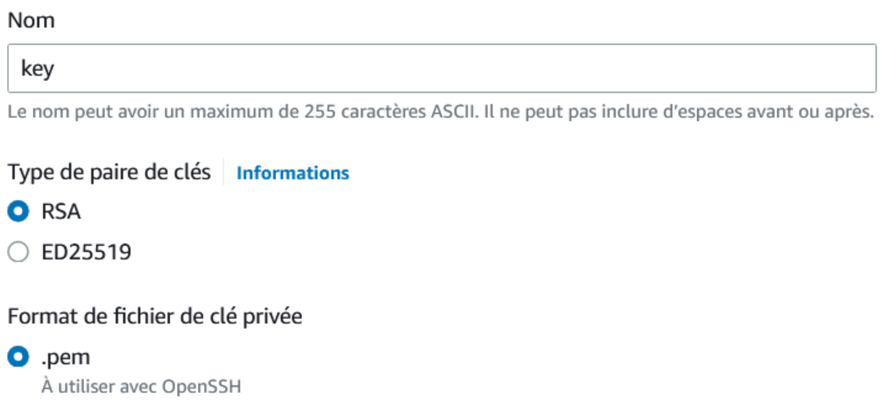
For this project, we wanted to analyze a dataset. In python, when we want to apply some threat on a dataset it takes a lot of RAM memory. That is wh, for our professional lives, we will maybe use spark instead.

Spark is a RDD system (Resilient Distributed Datasets). It means that it can execute different treatments in parallel on different clusters. It also uses the Map-Reduce algorithm. So, for Big Data projects, it is recommended to use spark.

After choosing to use Spark, we need to define clusters. We can install Spark on our computer, but we can also use a Web Service like AWS.

So, for this part, we are going to build an AWS EMR cluster.

Firstly, we created EC2 keys. They allow us to connect to our cluster. They are a file with which we can connect to the cluster using ssh protocol.


Nom

key

Le nom peut avoir un maximum de 255 caractères ASCII. Il ne peut pas inclure d'espaces avant ou après.

Type de paire de clés Informations

[✓ RSA]

ED25519

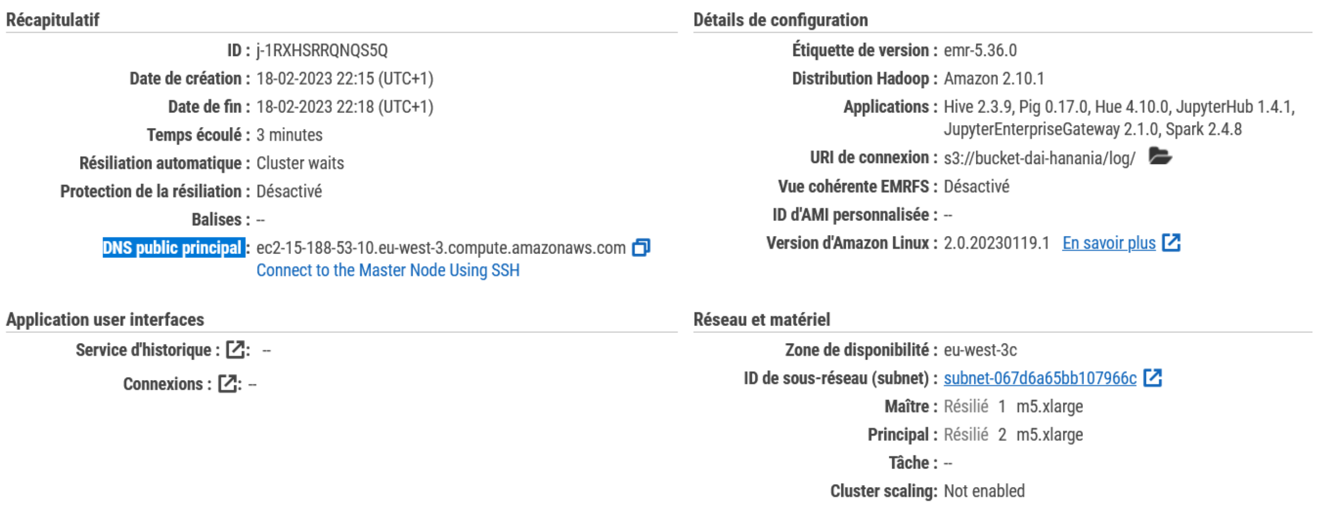
Format de fichier de clé privée

[✓ .pem]

À utiliser avec OpenSSH

Secondly, we create a S3 bucket. It is used to save the data on the AWS server. A bucket contains an URI so we can use it to connect.

Thirdly, to execute all the treatment, we need an EMR cluster. It is composed of a DNS public address we use to connect by ssh, a master node, and two slave nodes that execute all the commands.



Récapitulatif

ID :	j-1RXHSRRQNQS5Q
Date de création :	18-02-2023 22:15 (UTC+1)
Date de fin :	18-02-2023 22:18 (UTC+1)
Temps écoulé :	3 minutes
Résiliation automatique :	Cluster waits
Protection de la résiliation :	Désactivé
Balises :	--
DNS public principal :	ec2-15-188-53-10.eu-west-3.compute.amazonaws.com Connect to the Master Node Using SSH

Détails de configuration

Étiquette de version :	emr-5.36.0
Distribution Hadoop :	Amazon 2.10.1
Applications :	Hive 2.3.9, Pig 0.17.0, Hue 4.10.0, JupyterHub 1.4.1, JupyterEnterpriseGateway 2.1.0, Spark 2.4.8
URI de connexion :	s3://bucket-dai-hanania/log/
Vue cohérente EMRFS :	Désactivé
ID d'AMI personnalisée :	--
Version d'Amazon Linux :	2.0.20230119.1 En savoir plus

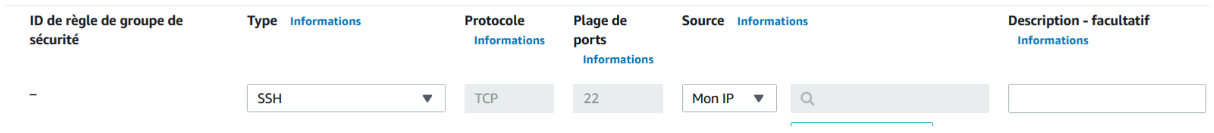
Application user interfaces

Service d'historique :	--
Connexions :	--

Réseau et matériel

Zone de disponibilité :	eu-west-3c
ID de sous-réseau (subnet) :	subnet-067d6a65bb107966c
Maître :	Résilié 1 m5.xlarge
Principal :	Résilié 2 m5.xlarge
Tâche :	--
Cluster scaling:	Not enabled

Finally, if we want to use this cluster from home, we need to modify the VPC network to add our home IP address.



ID de règle de groupe de sécurité	Type Informations	Protocole Informations	Plage de ports Informations	Source Informations		Description - facultatif Informations
–	SSH	TCP	22	Mon IP		

After all this, we can connect to the cluster using ssh with the key and the address

```
ssh -i key_dai_Hanania.pem
hadoop@ec2-35-180-186-3.eu-west-3.compute.amazonaws.com
```

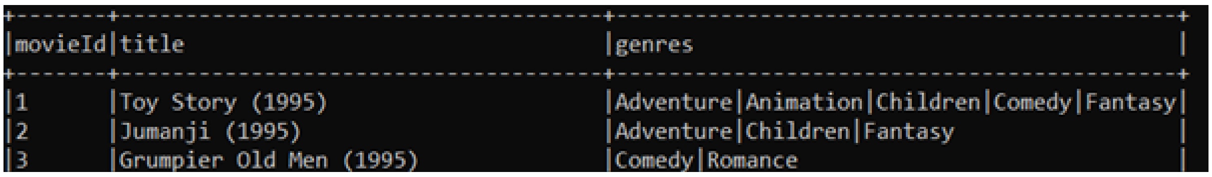
Firstly, we have to explain to the server that we want to execute pyspark commands.

```
pyspark
```

B. Creating the movielens dataset

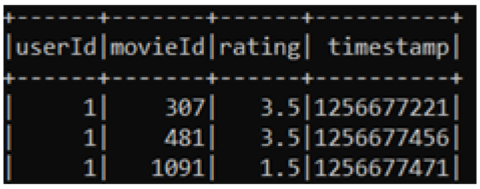
The first dataset we have to analyze is a movielens dataset. It is composed of a "**movies**" dataset, that describes all movies (title, genres, year…) and "**ratings**" that describes each rating from user (id user, id movie, rating /5). And the concatenation of theses two datasets made the **movielens**

```
df=spark.read.csv("s3://nahle-bucket-datalake/emr/input/movielens/
movies.csv", header=True, inferSchema=True)
```



```
+-------+-----------------------------------+---------------------------------------------+
|movieId|title                              |genres                                       |
+-------+-----------------------------------+---------------------------------------------+
|1      |Toy Story (1995)                   |Adventure|Animation|Children|Comedy|Fantasy|
|2      |Jumanji (1995)                     |Adventure|Children|Fantasy                  |
|3      |Grumpier Old Men (1995)            |Comedy|Romance                               |
```

```
ratings=spark.read.csv("s3://nahle-bucket-datalake/emr/input/movie
lens/ratings.csv", header=True, inferSchema=True)
```



```
+------+-------+------+----------+
|userId|movieId|rating| timestamp|
+------+-------+------+----------+
|     1|    307|   3.5|1256677221|
|     1|    481|   3.5|1256677456|
|     1|   1091|   1.5|1256677471|
```

We read these two csv files from the teacher's bucket, **header=True** means that the first line of the datasets is used to name the columns, and **inferSchema=True** attribute automatically the data types (int, string…)

After reading the two files, we can join them together. We need to import some functions that will be used later.

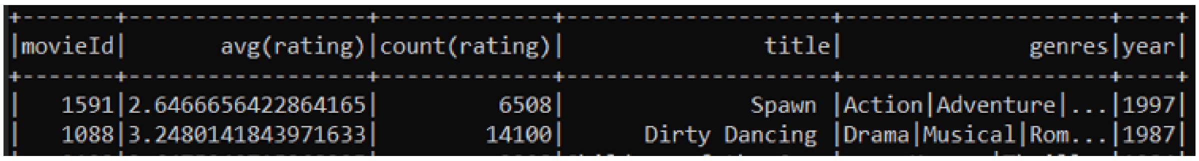
```
from pyspark.sql import functions as F
from pyspark.sql.functions import regexp_extract, regexp_replace
```

We want to obtain a final dataset that describes, for each movie, their mean rating, the number of rate, their title, their release year and their genres,

```
# grouping by movieId
final = ratings.groupBy("movieId").agg(F.mean('rating'),
F.count('rating'))

# joining the two datasets
final = final.join(df, ["movieId"],"inner").show()

# getting the year from the title ex : Titanic(1998)
final = final.withColumn("year", regexp_extract(final ["title"],
r"\((\d{4})\)", 1))
# removing the year from the title
```

```
final=final.withColumn("title", regexp_replace("title",
r"\(\d{4}\)", ""))
```

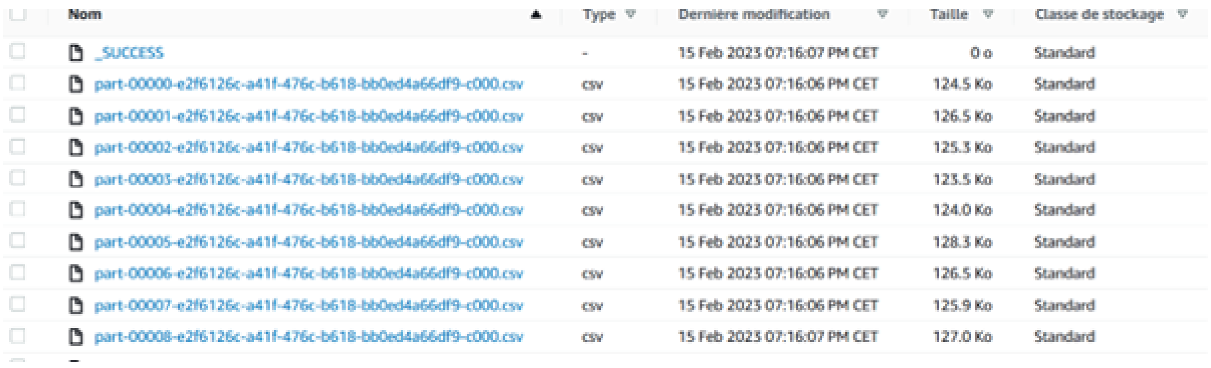
And finally we got :



```
+-------+------------------+-------------+--------------------+--------------------+----+
|movieId|       avg(rating)|count(rating)|               title|              genres|year|
+-------+------------------+-------------+--------------------+--------------------+----+
|   1591|2.6466656422864165|         6508|              Spawn |Action|Adventure|...|1997|
|   1088|3.2480141843971633|        14100|       Dirty Dancing |Drama|Musical|Rom...|1987|
```

And we save the model in our S3 bucket. As it is a map reduce cluster, the dataset is divided into multiple subclusters.

```
final.write.csv("s3://bucket-dai-hanania/output/movielens", header
= True)
```



Nom	Type	Dernière modification	Taille	Classe de stockage
_SUCCESS	-	15 Feb 2023 07:16:07 PM CET	0 o	Standard
part-00000-e2f6126c-a41f-476c-b618-bb0ed4a66df9-c000.csv	csv	15 Feb 2023 07:16:06 PM CET	124.5 Ko	Standard
part-00001-e2f6126c-a41f-476c-b618-bb0ed4a66df9-c000.csv	csv	15 Feb 2023 07:16:06 PM CET	126.5 Ko	Standard
part-00002-e2f6126c-a41f-476c-b618-bb0ed4a66df9-c000.csv	csv	15 Feb 2023 07:16:06 PM CET	125.3 Ko	Standard
part-00003-e2f6126c-a41f-476c-b618-bb0ed4a66df9-c000.csv	csv	15 Feb 2023 07:16:06 PM CET	123.5 Ko	Standard
part-00004-e2f6126c-a41f-476c-b618-bb0ed4a66df9-c000.csv	csv	15 Feb 2023 07:16:06 PM CET	124.0 Ko	Standard
part-00005-e2f6126c-a41f-476c-b618-bb0ed4a66df9-c000.csv	csv	15 Feb 2023 07:16:06 PM CET	128.3 Ko	Standard
part-00006-e2f6126c-a41f-476c-b618-bb0ed4a66df9-c000.csv	csv	15 Feb 2023 07:16:06 PM CET	126.5 Ko	Standard
part-00007-e2f6126c-a41f-476c-b618-bb0ed4a66df9-c000.csv	csv	15 Feb 2023 07:16:06 PM CET	125.9 Ko	Standard
part-00008-e2f6126c-a41f-476c-b618-bb0ed4a66df9-c000.csv	csv	15 Feb 2023 07:16:07 PM CET	127.0 Ko	Standard

C. Apply the SQL requests

After creating the requested dataset, we can read it again, create an SQL view and do some requests.

```
from pyspark.sql.functions import split
# read CSV
final = spark.read.option("header", "true").option("inferSchema",
"true").csv("s3://bucket-dai-hanania/output/movielens/")

#rename a column
final = final.withColumnRenamed("avg(rating)","score")
```

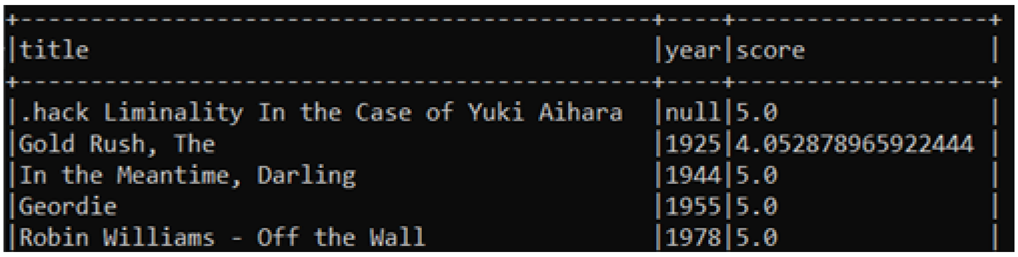
```
#tranform the genres column into a list
final = final.withColumn('genres',split(final.genres, '\|'))

#create the sql view
final.createOrReplaceTempView("movielens")
```

<u>Best movie per year</u>

```
spark.sql(

“SELECT title, year, score FROM ( SELECT title, year, score,
ROW_NUMBER() OVER (PARTITION BY year ORDER BY score DESC) AS rank
FROM movielens) ranked_movies WHERE rank = 1”

).show(truncate=False)
```

This request contains multiple sub-requests. Firstly, for all year, we display the movies by their score descending, and using **row_number()**, we attribute an incremental value for all movies. So the movies that have the best score (mean of the marks given by the users) for each year, will have 1 for its ranking column.
We could also use the number of votes to calculate the rank, but the mean of marks use it already.

After creating this column rank, we can select all the movies that have rank=1 and get this result :



```
+-----------------------------------------+----+-----------------+
|title                                    |year|score            |
+-----------------------------------------+----+-----------------+
|.hack Liminality In the Case of Yuki Aihara |null|5.0              |
|Gold Rush, The                           |1925|4.052878965922444 |
|In the Meantime, Darling                 |1944|5.0              |
|Geordie                                  |1955|5.0              |
|Robin Williams - Off the Wall            |1978|5.0              |
```

Globally, the best movies always have 5 as a mean score. But for some years, like 1925, the best movies had just 4.05 as a mean score.

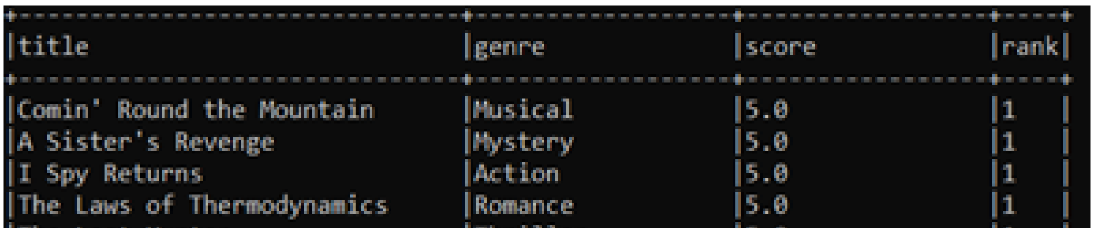
Furthermore, we can see that some movies don’t have years, so their year column is **null**

<u>Best movie par genre</u>

```
spark.sql(

"SELECT * FROM ( SELECT title, genre, score, ROW_NUMBER() OVER
(PARTITION BY genre ORDER BY score DESC) as rank FROM movielens
LATERAL VIEW explode(genres) exploded_genre AS genre)
ranked_movies WHERE rank = 1"

).show(truncate=False)
```

This request is similar to the previous one. But, we also use the **explode** function on the **genre** column. Associated with the **lateral view**, This function will create multiple lines for every genre of a movie : title :[genreA, genreB] will create two lines title :A and title:B.

So now we can get the ranking with **row number()**, and select partition the data by genre.



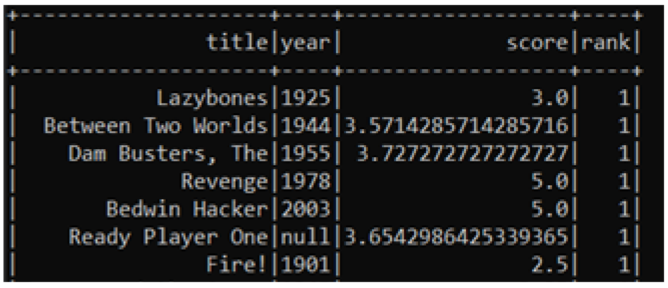
```
+-------------------------------+----------------+----------------+----+
|title                          |genre           |score           |rank|
+-------------------------------+----------------+----------------+----+
|Comin' Round the Mountain      |Musical         |5.0             |1   |
|A Sister's Revenge             |Mystery         |5.0             |1   |
|I Spy Returns                  |Action          |5.0             |1   |
|The Laws of Thermodynamics     |Romance         |5.0             |1   |
```

We can see that all movies have at least one genre.

<u>For 'action' movie per year</u>

```
spark.sql(

"SELECT * FROM (SELECT *, ROW_NUMBER() OVER (PARTITION BY year ORDER BY
score DESC) AS rank FROM ( SELECT title, year, score FROM movielens
LATERAL VIEW explode(genres) AS genre WHERE genre = 'Action' )
action_movies ) ranked WHERE rank = 1"

).show(truncate=False)
```

This request is a bit different from the previous one.
Here we have to preselect all the movies that have **Action** in their genre list, and then we can apply the same treatment as before, **row_number()** to select the best score partitioned by year

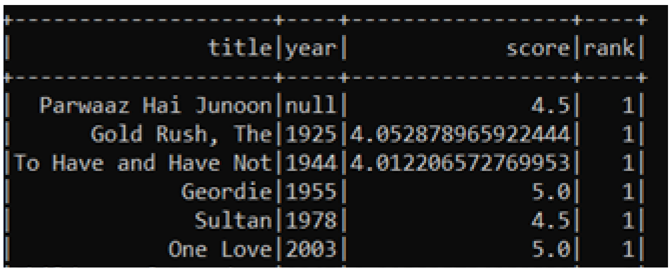


```
+--------------------+----+------------------+----+
|               title|year|             score|rank|
+--------------------+----+------------------+----+
|           Lazybones|1925|               3.0|   1|
|   Between Two Worlds|1944|3.5714285714285716|   1|
|     Dam Busters, The|1955| 3.727272727272727|   1|
|              Revenge|1978|               5.0|   1|
|         Bedwin Hacker|2003|               5.0|   1|
|     Ready Player One|null|3.6542986425339365|   1|
|                Fire!|1901|               2.5|   1|
```

Best romance per year

```
park.sql("

SELECT * FROM (SELECT *, ROW_NUMBER() OVER (PARTITION BY year ORDER BY
score DESC) AS rank FROM ( SELECT title, year, score FROM movielens
LATERAL VIEW explode(genres) AS genre WHERE genre = 'Romance' ) romance)
ranked WHERE rank = 1

")
```

This request is exactly the same as the previous one. But we replace **action** by **romance**

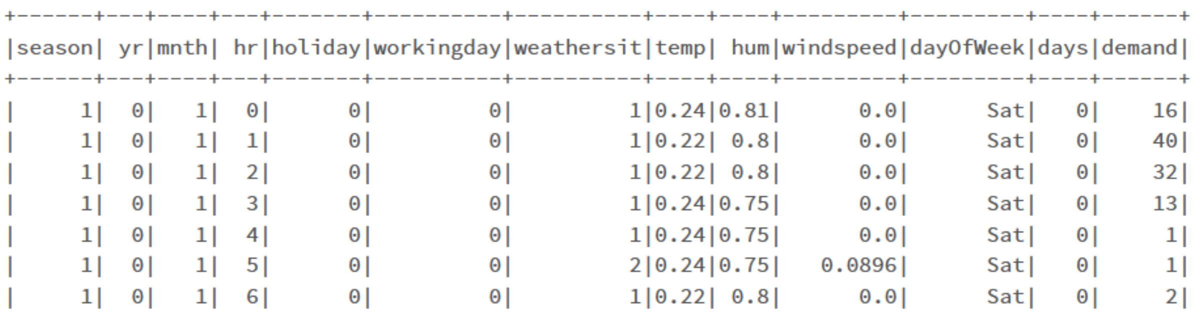


```
+-------------------+----+-----------------+----+
|              title|year|            score|rank|
+-------------------+----+-----------------+----+
|  Parwaaz Hai Junoon|null|              4.5|   1|
|      Gold Rush, The|1925|4.052878965922444|   1|
|To Have and Have Not|1944|4.012206572769953|   1|
|             Geordie|1955|              5.0|   1|
|              Sultan|1978|              4.5|   1|
|            One Love|2003|              5.0|   1|
```

We can see that the first movie doesn't have any year.

II. Bike rental model

This second exercise is about Bike rental predictions. We have a dataset that displays various information such as the weather, the hour, etc, and the number of demand per hour. Our objective is to predict the demand.



```
+------+---+----+---+-------+----------+----------+----+----+---------+---------+----+------+
|season| yr|mnth| hr|holiday|workingday|weathersit|temp| hum|windspeed|dayOfWeek|days|demand|
+------+---+----+---+-------+----------+----------+----+----+---------+---------+----+------+
|     1|  0|   1|  0|      0|         0|         1|0.24|0.81|      0.0|      Sat|   0|    16|
|     1|  0|   1|  1|      0|         0|         1|0.22| 0.8|      0.0|      Sat|   0|    40|
|     1|  0|   1|  2|      0|         0|         1|0.22| 0.8|      0.0|      Sat|   0|    32|
|     1|  0|   1|  3|      0|         0|         1|0.24|0.75|      0.0|      Sat|   0|    13|
|     1|  0|   1|  4|      0|         0|         1|0.24|0.75|      0.0|      Sat|   0|     1|
|     1|  0|   1|  5|      0|         0|         2|0.24|0.75|   0.0896|      Sat|   0|     1|
|     1|  0|   1|  6|      0|         0|         1|0.22| 0.8|      0.0|      Sat|   0|     2|
```

A. Model done in class

We have started this TP in class, so we start by analyzing its performances

Firstly, we transform the day into a category

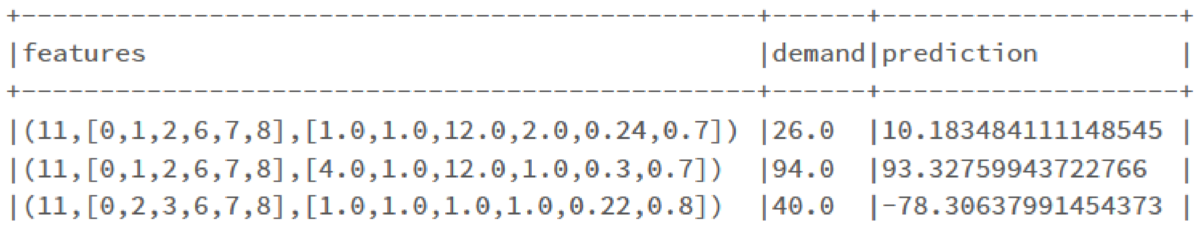
```
StringIndexer(inputCol='dayOfWeek',
outputCol='day_cat').fit(rowData).transform(rowData)
```

Secondly, we generate a vector of all the column

```
VectorAssembler(inputCols=
['season','yr','mnth','hr','holiday','workingday','weathersit','te
mp','hum','windspeed','day_cat'],outputCol = 'features')
```

To predict the demand, we use the features column, and we split the dataframe into a train and a test sub dataset

```
modelData = data.select('features', 'demand')
trainData, testData = modelData.randomSplit([0.7, 0.3])
```



```
+--------------------------------------------+------+-------------------+
|features                                    |demand|prediction         |
+--------------------------------------------+------+-------------------+
|(11,[0,1,2,6,7,8],[1.0,1.0,12.0,2.0,0.24,0.7])|26.0  |10.183484111148545 |
|(11,[0,1,2,6,7,8],[4.0,1.0,12.0,1.0,0.3,0.7]) |94.0  |93.32759943722766  |
|(11,[0,2,3,6,7,8],[1.0,1.0,1.0,1.0,0.22,0.8]) |40.0  |-78.30637991454373 |
```

Then we create a linear regression

```
lr = LinearRegression(labelCol='demand')
lrModel = lr.fit(trainData)
```

And then we evaluate the model on the train dataset

```
#test the model on the test Data
testResults = lrModel.evaluate(testData)
```

And the r2 score of the model on the test dataset is :

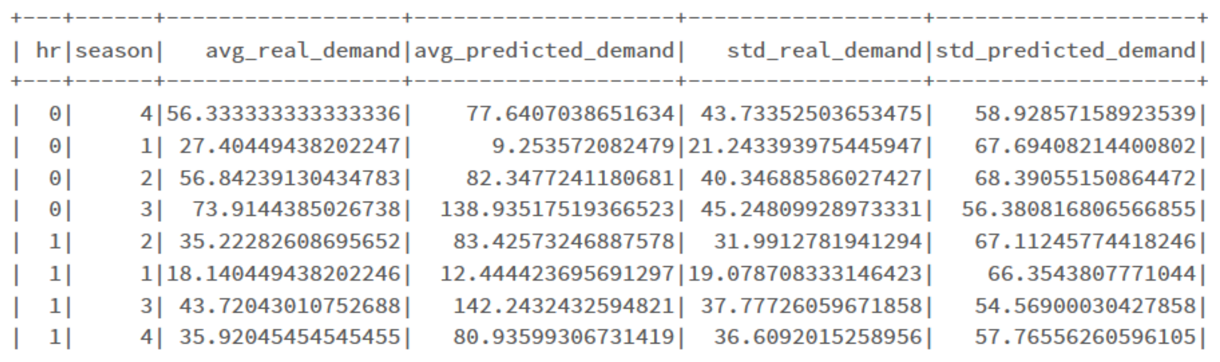
```
r2=0.382574
rootMeanSquaredError=142.383
```

So , the way we trained the model in this first part is very far from being satisfactory

We can get some insight of the prediction

<u>Standard deviation and Average of the demand and the prediction by hour and seasons</u>

```
#Hour and demand.
import pyspark.sql.functions as F
pred.groupBy("hr", "season").agg(

  F.avg("demand").alias("avg_real_demand"),
  F.avg("prediction").alias("avg_predicted_demand"),
  F.stddev("demand").alias("std_real_demand"),
  F.stddev("prediction").alias("std_pred_demand")

).sort('avg_real_demand').sort('hr').show()
```

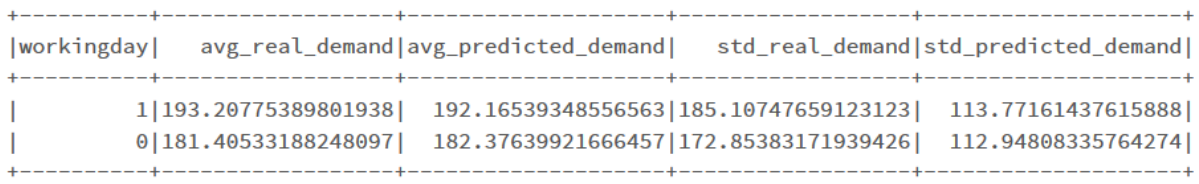


```
+---+------+------------------+--------------------+------------------+--------------------+
| hr|season|   avg_real_demand|avg_predicted_demand|   std_real_demand|std_predicted_demand|
+---+------+------------------+--------------------+------------------+--------------------+
|  0|     4|56.333333333333336|    77.6407038651634| 43.73352503653475|   58.92857158923539|
|  0|     1| 27.40449438202247|      9.253572082479|21.243393975445947|   67.69408214400802|
|  0|     2| 56.84239130434783|    82.3477241180681| 40.34688586027427|   68.39055150864472|
|  0|     3|  73.9144385026738|  138.93517519366523| 45.24809928973331|  56.380816806566855|
|  1|     2| 35.22282608695652|   83.42573246887578|  31.9912781941294|   67.11245774418246|
|  1|     1|18.140449438202246|  12.444423695691297|19.078708333146423|    66.3543807771044|
|  1|     3| 43.72043010752688|   142.243243259482l| 37.77726059671858|   54.56900030427858|
|  1|     4| 35.92045454545455|   80.93599306731419|  36.6092015258956|   57.76556260596105|
```

We can see that depending of the hour, it is not always the same season that gets the highest demand

Standard deviation and Average of the demand and the prediction by working day

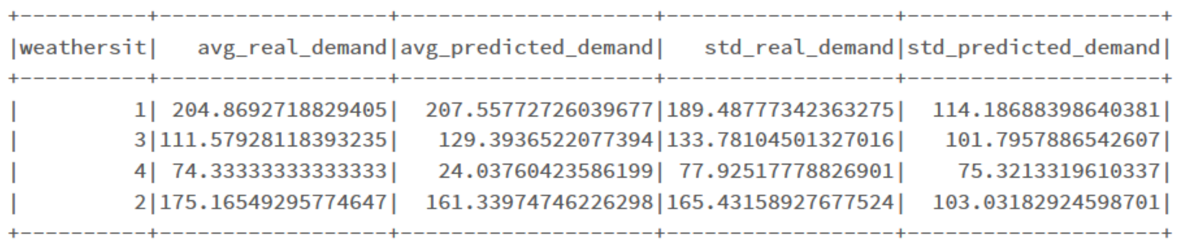
```
pred.groupBy("workingday").agg(
      F.avg("demand").alias("avg_real_demand"),
      F.avg("prediction").alias("avg_predicted_demand"),
      F.stddev("demand").alias("std_real_demand"),
      F.stddev("prediction").alias("std_predicted_demand")

).show()
```



```
+----------+------------------+--------------------+------------------+--------------------+
|workingday|   avg_real_demand|avg_predicted_demand|   std_real_demand|std_predicted_demand|
+----------+------------------+--------------------+------------------+--------------------+
|         1|193.20775389801938|  192.16539348556563|185.10747659123123|  113.77161437615888|
|         0|181.40533188248097|  182.37639921666457|172.85383171939426|  112.94808335764274|
+----------+------------------+--------------------+------------------+--------------------+
```

We can see that working day does not have an important impact on the demand

```
import pyspark.sql.functions as F
pred.groupBy("weathersit").agg(
F.avg("demand").alias("avg_real_demand"),
F.avg("prediction").alias("avg_predicted_demand"),
F.stddev("demand").alias("std_real_demand"),
F.stddev("prediction").alias("std_predicted_demand")
).show()
```

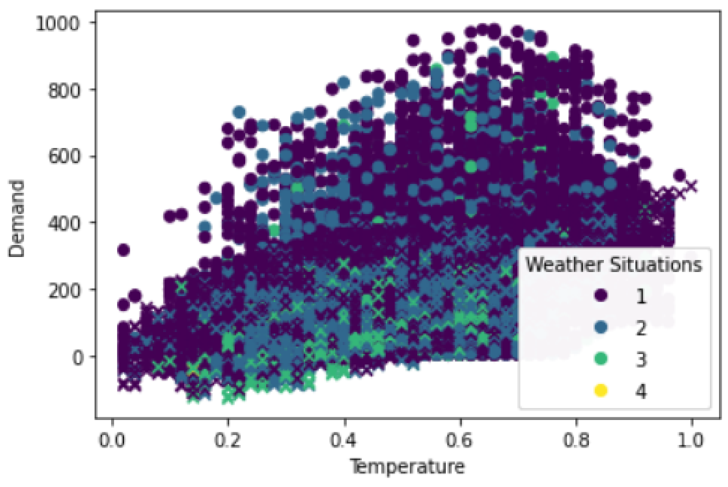


```
+----------+------------------+--------------------+------------------+--------------------+
|weathersit|   avg_real_demand|avg_predicted_demand|   std_real_demand|std_predicted_demand|
+----------+------------------+--------------------+------------------+--------------------+
|         1| 204.8692718829405|  207.55772726039677|189.48777342363275|  114.18688398640381|
|         3|111.57928118393235|   129.3936522077394|133.78104501327016|   101.7957886542607|
|         4| 74.33333333333333|   24.03760423586199| 77.92517778826901|    75.3213319610337|
|         2|175.16549295774647|  161.33974746226298|165.43158927677524|  103.03182924598701|
+----------+------------------+--------------------+------------------+--------------------+
```

But the weather has a important impact

Plotting the predictions and the real demand by temperature and weathersit

```
pred_ = pred.toPandas()
fig, ax = plt.subplots()
scatter = ax.scatter(pred_['temp'], pred_["demand"],
c=pred_["weathersit"])
ax.scatter(pred_['temp'], pred_["prediction"],
c=pred_["weathersit"], marker="x")
```



We can see that the temperature has an important impact on the demand. Indeed, there is 5 times more demand when it is 0.7 than is is 0.

Furthermore, the weather situation is also important as the 1 and 2 are the most common.

B. Model improvement

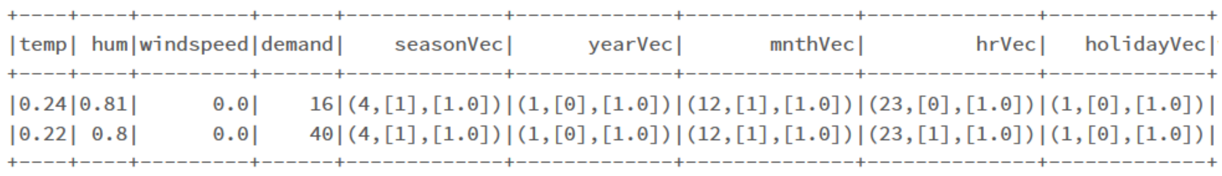
To improve the model, we , we started by transform the days into a category, and we delete the **days** and **dayOfWeek** column as they are not useful.

```
# I transform all the data into numericals, and I drop the non
numericals
indexer = StringIndexer(inputCol='dayOfWeek', outputCol='day_cat')
indexed_data =indexer.fit(rowData).transform(rowData)

#removing the old dayOfWeek, and days as they are not usefull for
the
indexed_data = indexed_data.drop('dayOfWeek','days')
```

Then we use one hot encoding to add dummies variables

```
encoder = OneHotEncoder(
inputCols=['season', 'yr','mnth','hr','holiday','workingday',
'weathersit','day_cat'],
outputCols=['seasonVec','yearVec','mnthVec','hrVec','holidayVec'
,'workingdayVec' ,'weathersitVec', 'day_catVec']
)

#apply on the dataset
model = encoder.fit(indexed_data)
encoded_data = model.transform(indexed_data)
#dropping the old column
encoded_data = encoded_data.drop('season',
'yr','mnth','hr','holiday','workingday','weathersit','day_cat')
```

So, all columns as like this



```
+----+----+---------+------+-------------+-------------+--------------+--------------+-------------+
|temp| hum|windspeed|demand|    seasonVec|      yearVec|       mnthVec|         hrVec|   holidayVec|
+----+----+---------+------+-------------+-------------+--------------+--------------+-------------+
|0.24|0.81|      0.0|    16|(4,[1],[1.0])|(1,[0],[1.0])|(12,[1],[1.0])|(23,[0],[1.0])|(1,[0],[1.0])|
|0.22| 0.8|      0.0|    40|(4,[1],[1.0])|(1,[0],[1.0])|(12,[1],[1.0])|(23,[1],[1.0])|(1,[0],[1.0])|
+----+----+---------+------+-------------+-------------+--------------+--------------+-------------+
```

Then, we group all the columns together exect the **demand**

```
col = ['seasonVec', 'yearVec','mnthVec','hrVec','holidayVec',
'workingdayVec','weathersitVec','day_catVec','temp','hum','windspe
ed']
vector= VectorAssembler(inputCols=col, outputCol = 'features')
```

And we normalize the data,

```
#standard scaler
standarScaler = StandardScaler(inputCol="features",
 outputCol="STfeatures")
STdata = standarScaler.fit(data).transform(data)

#minmaxscaler
mmScaler = MinMaxScaler(inputCol="STfeatures",
 outputCol="NRfeatures")
NRdata = mmScaler.fit(STdata).transform(STdata)
```

Now we have the data in a good format, we can create our model.

To create the model, we start by selecting the column we want to predict : **demand** and the column we want to use **NRfeatures** that represents the features normalized. We also split the dataset into a train and test sub datasets

```
modelData = NRdata.select('NRfeatures', 'demand')
trainData, testData = modelData.randomSplit([0.8, 0.2])
```

```
#create the linear regression
lr = LinearRegression(featuresCol="NRfeatures",labelCol='demand')

lrModel = lr.fit(trainData)
predictions = lrModel.transform(testData)

# we can see that the model is good !
print(lrModel.summary.r2)
lrModel.summary.meanAbsoluteError

r2 : 0.6885523286532669
MAE : 75.00301289062446
```

The results are pretty good on the training dataset!

We also use an evaluator to evaluate the model on the test dataset

```
lr_evaluator = RegressionEvaluator(predictionCol="prediction",
labelCol="demand",metricName="r2")

r2 : 0.6753477359544817
```

C. The second improvement

For the second improvement, we will use cross validation to test different models. Firstly we create a pipeline that will apply the linear regression

```
final_pipeline = Pipeline(stages=[lr])
```

And we also create a cross validator that will test different values for the model and find the best model

```
param_grid = ParamGridBuilder() \
      .addGrid(lr.regParam, [0.1, 0.01]) \
      .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \
      .build()

cv = CrossValidator(estimator=final_pipeline,
estimatorParamMaps=param_grid, evaluator=lr_evaluator, numFolds=5)
```

We fit it on the train dataset, and test it on the test dataset

```
# fit the cross-validator to the data
cv_model = cv.fit(trainData)

#transform on test data
tuned_lr_accuracy = lr_evaluator.evaluate(cv_model.
  transform(testData))

tuned_lr_mae = lr_evaluator.evaluate(cv_model.transform(testData),
  {lr_evaluator.metricName: "mae"})

tuned logistic regression accuracy :0.676911332147188
tuned logistic regression mae:74.5997047411219
```

D. Comparison with other machine learning model

After obtaining this first result, we decide ton compare its score with other machine learning models

```
# create the models Random forest and decision tree
dt = DecisionTreeRegressor(featuresCol="NRfeatures",
labelCol="demand")

rf = RandomForestRegressor(featuresCol="NRfeatures",
labelCol="demand")

#The pipelines
dt_pipeline = Pipeline(stages=[dt])

rf_pipeline = Pipeline(stages=[rf])

# create the parameter grids for each model
dt_param_grid = ParamGridBuilder() \
      .addGrid(dt.maxDepth, [5, 10,15,20]) \
      .build()

rf_param_grid = ParamGridBuilder() \
      .addGrid(rf.maxDepth, [5, 10,15,20]) \
      .addGrid(rf.numTrees, [10, 20]) \
      .build()

# set up the cross-validators for each model
dt_cv = CrossValidator(estimator=dt_pipeline,
estimatorParamMaps=dt_param_grid, evaluator=lr_evaluator,
numFolds=5)

rf_cv = CrossValidator(estimator=rf_pipeline,
estimatorParamMaps=rf_param_grid, evaluator=lr_evaluator,
numFolds=5)

# fit the cross-validators to the data and evaluate the best
models
dt_cv_model = dt_cv.fit(trainData)
rf_cv_model = rf_cv.fit(trainData)
```

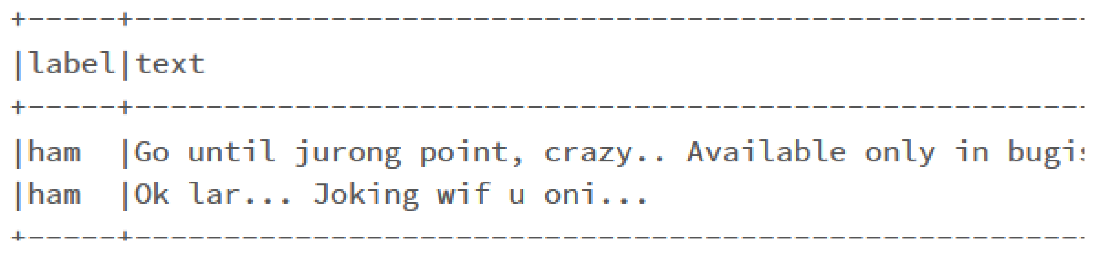
```
#get best models
best_dt_model = dt_cv_model.bestModel
best_rf_model = rf_cv_model.bestModel

#predictions
dt_predictions = best_dt_model.transform(testData)
rf_predictions = best_rf_model.transform(testData)

#evaluation
dt_mae lr_evaluator.evaluate(dt_predictions,
{lr_evaluator.metricName: "mae"})

dt_r2 = lr_evaluator.evaluate(dt_predictions)
```

Finally, we can compare the different models. It is the random forest model that has the best accuracy score.

III. SMS Spam Collection

A. Data Preparation

For this exercise, we have a complete dataset. It represents some SMS and explains if they are spam or safe (ham). The objective is to predict if a SMS is safe or not.



```
+-----+-------------------------------------------------------
|label|text
+-----+-------------------------------------------------------
|ham  |Go until jurong point, crazy.. Available only in bugi
|ham  |Ok lar... Joking wif u oni...
+-----+-------------------------------------------------------
```

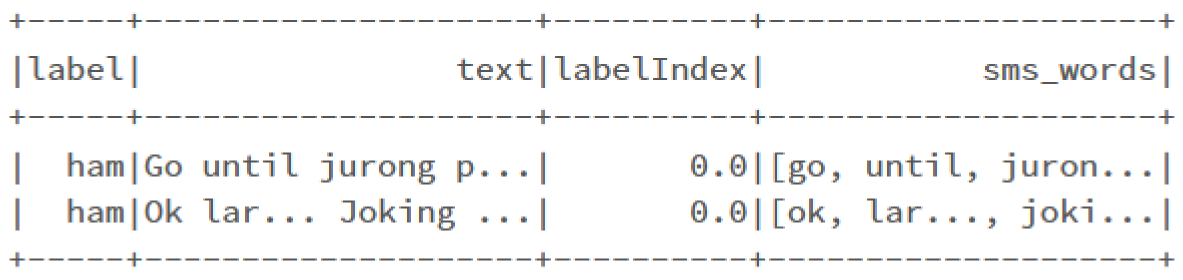
The dataset is not in a good format to make predictions on it, we need to make transformations.

The first is to transform the label into categories. We use **StringIndexer** to transform “ham” on 0, and “spam” on 1

```
indexer = StringIndexer(inputCol="label", outputCol="labelIndex")
```

We also transform the text into vectors

```
tokenizer = Tokenizer(inputCol="text", outputCol="sms_words")
```



```
+-----+--------------------+----------+--------------------+
|label|                text|labelIndex|           sms_words|
+-----+--------------------+----------+--------------------+
|  ham|Go until jurong p...|       0.0|[go, until, juron...|
|  ham|Ok lar... Joking ...|       0.0|[ok, lar..., joki...|
+-----+--------------------+----------+--------------------+
```

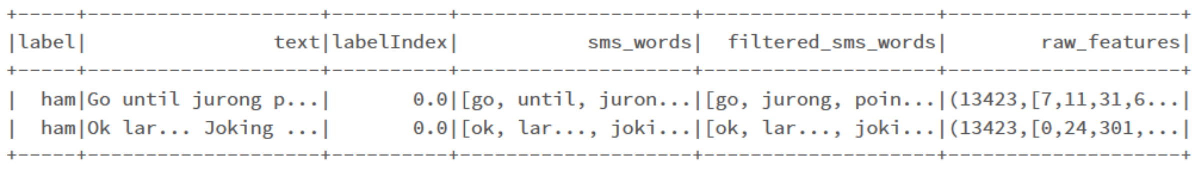
Then, we remove stop words because they are not useful for the prediction

```
remover = StopWordsRemover(inputCol="sms_words",
outputCol="filtered_sms_words")
```

And we transform this list of words into list of number

CountVectorizer is used to extracts a vocabulary from document collections.

```
count_vectorizer = CountVectorizer(inputCol="filtered_sms_words",
outputCol="raw_features")
```



```
+-----+--------------------+----------+--------------------+--------------------+--------------------+
|label|                text|labelIndex|           sms_words|  filtered_sms_words|        raw_features|
+-----+--------------------+----------+--------------------+--------------------+--------------------+
|  ham|Go until jurong p...|       0.0|[go, until, juron...|[go, jurong, poin...|(13423,[7,11,31,6...|
|  ham|Ok lar... Joking ...|       0.0|[ok, lar..., joki...|[ok, lar..., joki...|(13423,[0,24,301,...|
+-----+--------------------+----------+--------------------+--------------------+--------------------+
```

In this dataset, the **raw_features** represents the text. But the different words don't have an importance indicator. That is why we will use the TDIDF transformer, to get the most important words in spam messages.

```
idf = IDF(inputCol="raw_features", outputCol="features")
```

Finally, we merges multiple columns into a vector column using **VectorAssembler**

```
assembler = VectorAssembler(inputCols=["features"],
outputCol="features_vector")
```

All the predictions will be done with this **features_vectors**

B. Training 4 models

After the data preparation, we obtain a vector with numericals values. This vector is in a good format to make predictions.

We will use 4 differents models : a **Logistic Regression**, a Decision Tree, a **Random Forest**, and a **Naive Bayes** to see which one is the best

```
#Creating pipelines for the different classifiers

lr = LogisticRegression(featuresCol="features_vector",
labelCol="labelIndex")
pipeline_lr = Pipeline(stages=[assembler, lr])
```

```
dt = DecisionTreeClassifier(featuresCol="features_vector",
labelCol="labelIndex")
pipeline_dt = Pipeline(stages=[assembler, dt])

rf = RandomForestClassifier(featuresCol="features_vector",
labelCol="labelIndex")
pipeline_rf = Pipeline(stages=[assembler, rf])

nb = NaiveBayes(featuresCol="features_vector",
labelCol="labelIndex")
pipeline_nb = Pipeline(stages=[assembler, nb])
```

For all of these models, we use a pipeline, they are used to combine several data processing steps (the assembler and the model creation) into one well organized sequence.

Then, we need to train our models. So we separate the whole dataset into a train and a test part.

```
#Separate the data into a train and a test data
training, testing = df.randomSplit([0.8, 0.2])

#Training the models using the pipelines
lr_model = pipeline_lr.fit(training)
dt_model = pipeline_dt.fit(training)
rf_model = pipeline_rf.fit(training)
nb_model = pipeline_nb.fit(training)
```

Then we define an evaluator that will give a score for each model

```
#Evaluating the models on the testing data
evaluator =
MulticlassClassificationEvaluator(predictionCol="prediction",
labelCol="labelIndex", metricName="accuracy")

#Calculating the accuracies
lr_accuracy = evaluator.evaluate(lr_model.transform(testing))
dt_accuracy = evaluator.evaluate(dt_model.transform(testing))
rf_accuracy = evaluator.evaluate(rf_model.transform(testing))
nb_accuracy = evaluator.evaluate(nb_model.transform(testing))
```

```
Logistic Regression Accuracy: 0.9836065573770492
Decision Tree Accuracy: 0.9326047358834244
Random Forest Accuracy: 0.8752276867030966
Naive Bayes Accuracy: 0.9153005464480874
```

We can see that the best model is the Logistic Regression and the least is the random forest.

So, for the next part, we will try to improve the random forest model.

C. Tuning the Random Forest model

In order to improve the random forest model, we will change some hyperparameters. We try some tree numbers values (10,20,50) and some maxDepth(5,10,15).

And we use a **CrossValidator** to test the different hyperparameters.

```
param_grid = (ParamGridBuilder().addGrid(rf.numTrees, [10, 20,
50]).addGrid(rf.maxDepth, [5, 10, 15]).build())

#CrossValidator
crossval = CrossValidator(estimator=pipeline_rf,
estimatorParamMaps=param_grid, evaluator=evaluator, numFolds=4)
rf_model_tuned = crossval.fit(training)
```

The new random forest accuracy is

```
Tuned Random Forest Accuracy: 0.9180327868852459
```

The final results are :

Logistic Regression Accuracy	0.9836065573770492
Decision Tree Accuracy	0.9326047358834244
Tuned Random Forest Accuracy	0.9180327868852459
Naive Bayes Accuracy	0.9153005464480874
Random Forest Accuracy	0.8752276867030966

By tuning the random forest, we get a 5% improvement.