Spark Scala

1)Introduction :

Le cancer du sein est l'un des cancers les plus fréquents chez les femmes. Le diagnostic précis et la classification du type de cancer sont essentiels pour une prise en charge et un traitement appropriés. Le dataset "Breast Cancer Wisconsin" fournit des informations précieuses sur les caractéristiques des tumeurs mammaires et leur lien avec le diagnostic (bénin ou malin).

2)Preparation du spark scala et du dataset pour l'analyse des données :

Nous avons procédé à l'installation de Spark sur le cluster Hadoop utilisé dans le bilan 1., il suffit de lancer vos machines grâce aux commandes suivantes:

docker start hadoop-master hadoop-worker1 hadoop-worker2

copier le dataset (data.csv) dans le hadoop master :

```
C:\Users\Amen Khlifi>docker cp "C:\Users\Amen Khlifi\OneDrive\Documents\project data mining\data.csv" hadoop-m
aster:/hdfs/data.csv
Successfully copied 127kB to hadoop-master:/hdfs/data.csv
```

puis d'entrer dans le contenaire master:

docker exec -it hadoop-master bash

Lancer ensuite les démons yarn et hdfs:

root@hadoop-master:~# ./start-hadoop.sh

Vous pourrez vérifier que tous les démons sont lancés en tapant: jps. Un résultat semblable au suivant pourra être visible:

```
root@hadoop-master:~# jps
693 ResourceManager
198 NameNode
1017 Jps
444 SecondaryNameNode
```

-Créer un répertoire dans HDFS, appelé *dataset :*

Hadoop fs -mkdir -p dataset

Chargement du fichier purchases dans le répertoire input :

Hadoop fs -put data.csv dataset

```
root@hadoop-master:~# hadoop fs -put data.csv dataset
root@hadoop-master:~# ls
data.csv  mapper.py      reducer.py        start-hadoop.sh
hdfs      purchases.txt  run-wordcount.sh  start-kafka-zookeeper.sh
```

Lancer spark :

root@hadoop-master:~# spark-shell

On utilise cette commande pour lire un fichier CSV stocké dans HDFS et le charge dans un DataFrame Spark nommé df. Le DataFrame peut ensuite être utilisé pour des opérations d'analyse et de traitement de données à l'aide des API Spark.

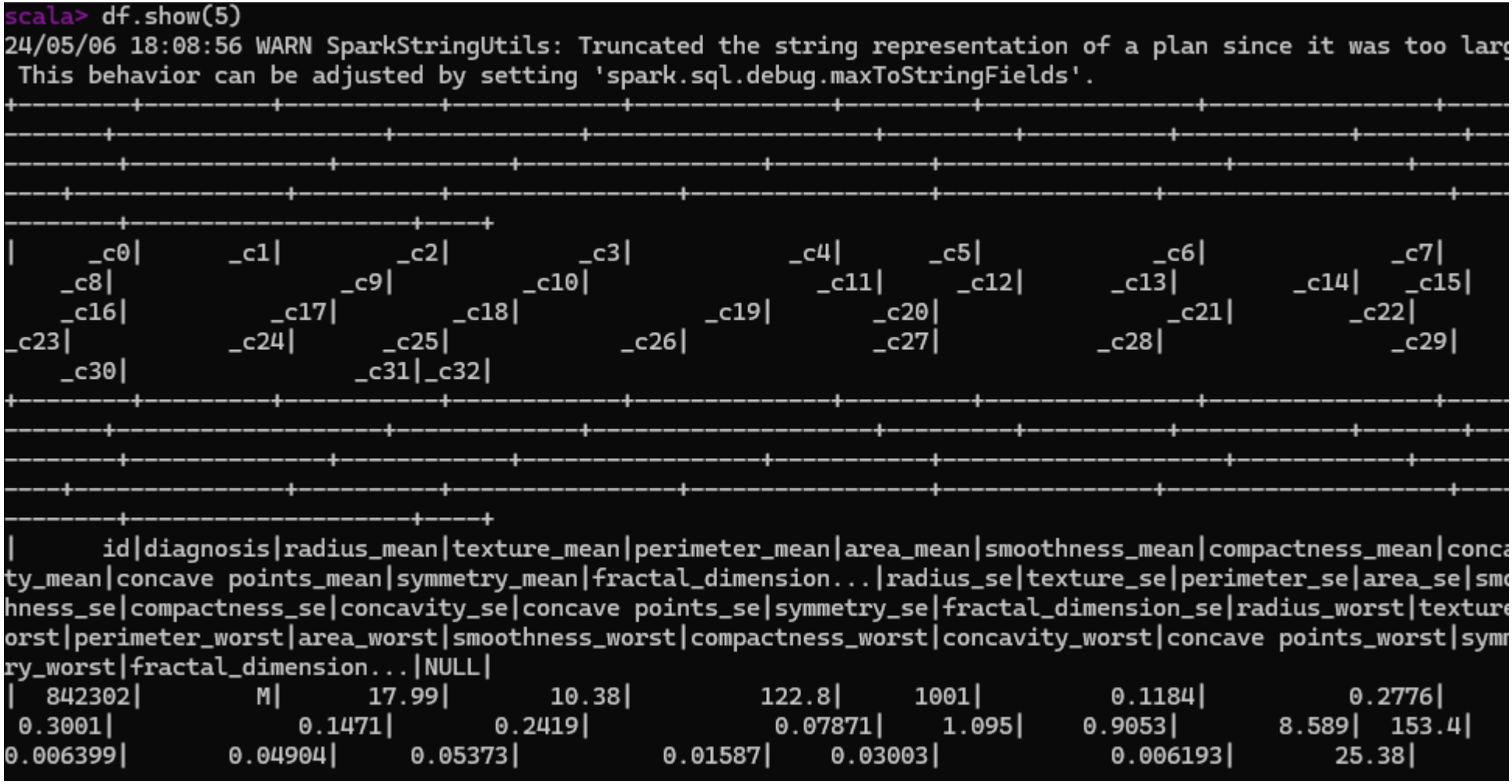
```
val df =spark.read.csv("hdfs:///user/hadoop/dataset/data.csv")
```

3)Analyses descriptives :

Les analyses descriptives permettront de résumer les caractéristiques du dataset, d'identifier les tendances et de découvrir des informations préliminaires sur les relations entre les variables. Cela inclura des mesures telles que la moyenne, la médiane, l'écart-type, la distribution des valeurs et les corrélations entre les variables.

top 5 rows :

df.show(5)


scala> df.show(5)
24/05/06 18:08:56 WARN SparkStringUtils: Truncated the string representation of a plan since it was too larg
This behavior can be adjusted by setting 'spark.sql.debug.maxToStringFields'.
|_c0| _c1| _c2| _c3| _c4| _c5| _c6| _c7|
_c8| _c9| _c10| _c11| _c12| _c13| _c14| _c15|
_c16| _c17| _c18| _c19| _c20| _c21| _c22|
_c23| _c24| _c25| _c26| _c27| _c28| _c29|
_c30| _c31|_c32|
|id|diagnosis|radius_mean|texture_mean|perimeter_mean|area_mean|smoothness_mean|compactness_mean|conca
ty_mean|concave points_mean|symmetry_mean|fractal_dimension...|radius_se|texture_se|perimeter_se|area_se|smo
hness_se|compactness_se|concavity_se|concave points_se|symmetry_se|fractal_dimension_se|radius_worst|texture
orst|perimeter_worst|area_worst|smoothness_worst|compactness_worst|concavity_worst|concave points_worst|symm
ry_worst|fractal_dimension...|NULL|
| 842302| M| 17.99| 10.38| 122.8| 1001| 0.1184| 0.2776|
0.3001| 0.1471| 0.2419| 0.07871| 1.095| 0.9053| 8.589| 153.4|
0.006399| 0.04904| 0.05373| 0.01587| 0.03003| 0.006193| 25.38|

retourne la 1ere ligne :

Dfh.first()

```
scala> dfh.first()
res1: org.apache.spark.sql.Row = [842302,M,17.99,10.38,122.8,1001,0.1184,0.2776,0.3001,0.1471,0.2419,0.07871,1.095
373,0.01587,0.03003,0.006193,25.38,17.33,184.6,2019,0.1622,0.6656,0.7119,0.2654,0.4601,0.1189]
```

Afficher les statistiques descriptives pour une colonne: radius_mean :

df.describe("_c2").show()

```
scala> df.describe("_c2").show()
+-------+------------------+
|summary|               _c2|
+-------+------------------+
|  count|               570|
|   mean|14.127291739894563|
| stddev|3.5240488262120793|
|    min|             10.03|
|    max|       radius_mean|
+-------+------------------+
```

Calculer la moyenne , l'ecart type , le min , le max pour la colonne radius_mean :

```
scala> df.select(avg("_c2"), stddev("_c2"), min("_c2"), max("_c2")).show()
+-----------------+-----------------+-------+----------+
|         avg(_c2)|       stddev(_c2)|min(_c2)|   max(_c2)|
+-----------------+-----------------+-------+----------+
|14.127291739894563|3.5240488262120793|   10.03|radius_mean|
+-----------------+-----------------+-------+----------+
```

Callculer le nombre de cas benins et malins :

```
val benignCount = df.filter($"_c1" === "B").count()
val malignantCount = df.filter($"_c1" === "M").count()
```

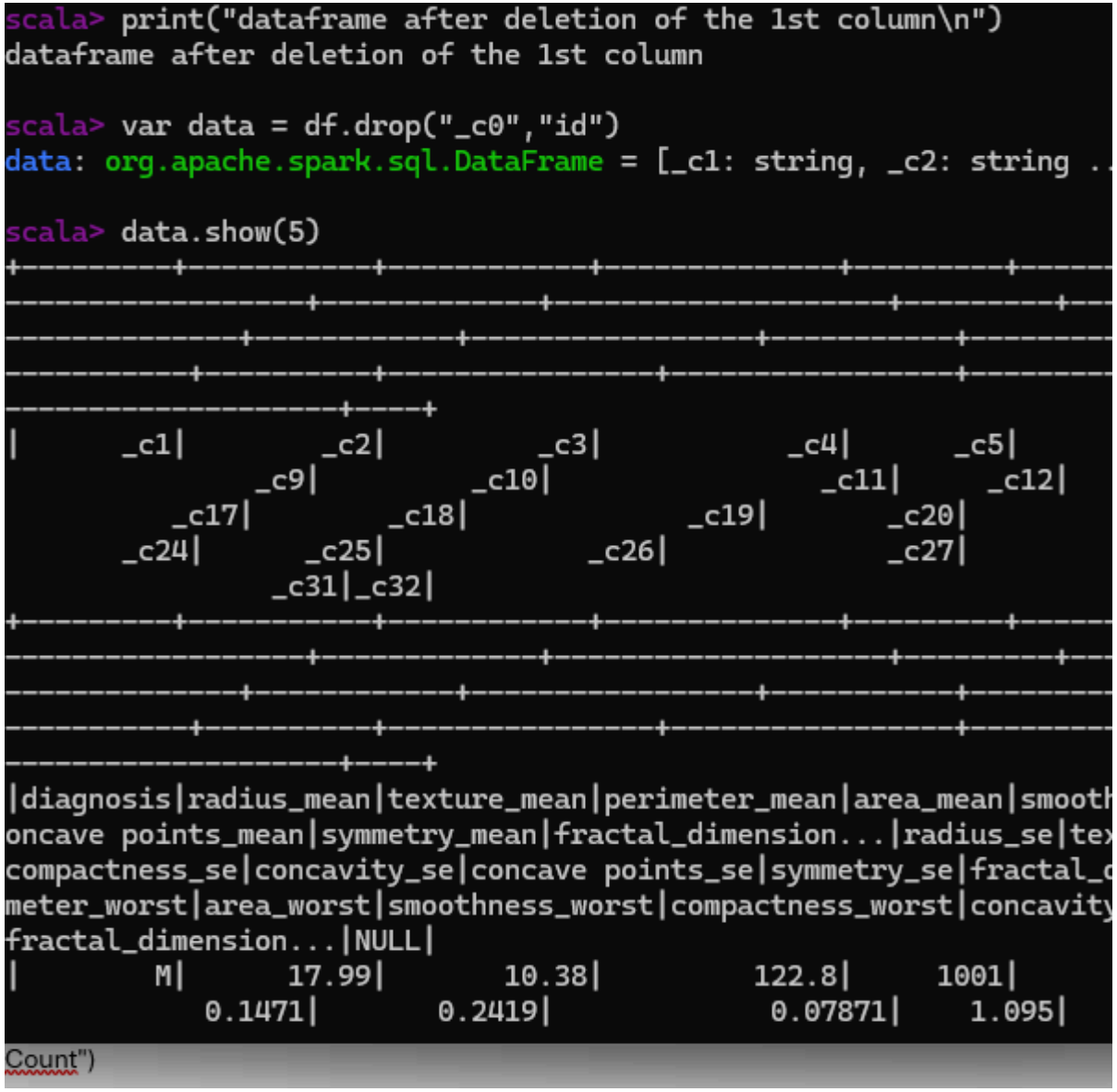
println(s"Nombre de cas benins : $benignCount")

println(s"Nombre de cas malins : $malignantCount")

```
scala> val benignCount = df.filter($"_c1" === "B").count()
benignCount: Long = 357

scala> val malignantCount = df.filter($"_c1" === "M").count()
malignantCount: Long = 212

scala>

scala> println(s"Nombre de cas benins : $benignCount")
Nombre de cas benins : 357

scala> println(s"Nombre de cas malins : $malignantCount")
Nombre de cas malins : 212
```

On n’a pas besoin de la 1ere colonne donc on peut la supprimer par la commande :

Var data = df.drop(“_c0”,”id”)


scala> print("dataframe after deletion of the 1st column\n")
dataframe after deletion of the 1st column
scala> var data = df.drop("_c0","id")
data: org.apache.spark.sql.DataFrame = [_c1: string, _c2: string ..
scala> data.show(5)
|diagnosis|radius_mean|texture_mean|perimeter_mean|area_mean|smooth
oncave points_mean|symmetry_mean|fractal_dimension...|radius_se|tex
compactness_se|concavity_se|concave points_se|symmetry_se|fractal_d
meter_worst|area_worst|smoothness_worst|compactness_worst|concavity
fractal_dimension...|NULL|
| M| 17.99| 10.38| 122.8| 1001|
0.1471| 0.2419| 0.07871| 1.095|
Count")

Types des colonne :

```
data.printSchema()
```

```
scala> data.printSchema()
root
 |-- _c1: string (nullable = true)
 |-- _c2: string (nullable = true)
 |-- _c3: string (nullable = true)
 |-- _c4: string (nullable = true)
 |-- _c5: string (nullable = true)
 |-- _c6: string (nullable = true)
 |-- _c7: string (nullable = true)
 |-- _c8: string (nullable = true)
 |-- _c9: string (nullable = true)
 |-- _c10: string (nullable = true)
 |-- _c11: string (nullable = true)
 |-- _c12: string (nullable = true)
 |-- _c13: string (nullable = true)
 |-- _c14: string (nullable = true)
 |-- _c15: string (nullable = true)
 |-- _c16: string (nullable = true)
 |-- _c17: string (nullable = true)
 |-- _c18: string (nullable = true)
 |-- _c19: string (nullable = true)
 |-- _c20: string (nullable = true)
 |-- _c21: string (nullable = true)
 |-- _c22: string (nullable = true)
 |-- _c23: string (nullable = true)
 |-- _c24: string (nullable = true)
 |-- _c25: string (nullable = true)
 |-- _c26: string (nullable = true)
```

Calculer la moyenne et l'ecart type de la texture pour les cas malins :

```
scala> val meanTexture = data.filter($"_c1" === "M").agg(avg("_c3")).first().getDouble(0)
meanTexture: Double = 21.60490566037735

scala> val stdTexture = data.filter($"_c1" === "M").agg(stddev("_c3")).first().getDouble(0)
stdTexture: Double = 3.7794699207763 42

scala> println(s"Moyenne de la texture pour les cas malins : $meanTexture")
Moyenne de la texture pour les cas malins : 21.60490566037735

scala> println(s"Ecart type de la texture pour les cas malins : $stdTexture")
```

```
scala> println(s"Ecart type de la texture pour les cas malins : $stdTexture")
Ecart type de la texture pour les cas malins : 3.7794699207763 42
```

Trouver les valeurs unique dans une colonne speciifique :

```
val uniqueValues = data.select("_c2").distinct().collect().map(_(0))

println("Valeurs uniques dans la colonne 'radius_mean':")

uniqueValues.foreach(println)
```

```
scala> val uniqueValues = data.select("_c2").distinct().collect().map(_(0))
uniqueValues: Array[Any] = Array(17.42, 13.87, 20.64, 8.618, 6.981, 15.49, 12.85, 9.668, 8.734, 10.97, 9.683,
28.11, 12.8, 14.2, 16.6, 11.42, 8.888, 11.62, 9.029, 15, 13.73, 13.27, 11.29, 11.61, 21.09, 10.03, 11.64, 13.6
6, 18.63, 21.56, 12.31, 15.53, 11.75, 12.4, 11.87, 18.49, 8.726, 17.57, 12.03, 12.34, 15.7, 15.66, 21.16, 11.2
5, 13.48, 12.68, 22.01, 12.94, 10.95, 14.96, 9.042, 14.48, 12.86, 11.46, 15.13, 17.6, 14.47, 15.75, 11.54, 19.
68, 12.88, 14.78, 15.73, 12.75, 10.96, 11.04, 18.82, 13.43, 12.65, 8.95, 12.76, 25.73, 11.63, 14.81, 16.13, 16
.16, 17.93, 12.25, 14.8, 14.4, 17.99, 12.36, 14.03, 11.68, 14.26, 14.87, 19.59, 11.95, 13.38, 20.29, 18.31, 13
.65, 16.25, 11.67, 14.58, 10.9, 11.13, 12.22, 19.55, 17.35, 16.24, 19.4, 13.71, 20.2, 12.18, 24.25, 10.8, 17.1
9...

scala> println("Valeurs uniques dans la colonne 'radius_mean':")
Valeurs uniques dans la colonne 'radius_mean':

scala> uniqueValues.foreach(println)
17.42
13.87
20.64
8.618
6.981
15.49
12.85
9.668
8.734
10.97
9.683
```

Convertir la colonne diagnosis :_c1 en des valeurs numerique B=0, M=1 pour pouvoir appliquer Bucketizer (acceptant que les colonnes numériques) :

```
scala> val df = data.withColumn("_c1", when($"_c1" === "B", 0).otherwise(when($"_c1" === "M", 1).otherwise($"_
c1")))
df: org.apache.spark.sql.DataFrame = [_c1: string, _c2: string ... 30 more fields]

scala> df.show()
+--------+----------+----------+-------------+--------+--------------+--------------+-------------+-
----------------+------------+------------------+--------+---------+-----------+------+------------+
-------------+-----------+---------------+----------+------------------+-----------+-----------+----
---------+---------+---------------+----------------+--------------+-------------+
--------------------+----+
|      _c1|        _c2|         _c3|           _c4|      _c5|           _c6|           _c7|          _c8|
          _c9|         _c10|              _c11|     _c12|      _c13|       _c14|   _c15|         _c16|
        _c17|        _c18|             _c19|       _c20|              _c21|       _c22|        _c23|
       _c24|      _c25|            _c26|            _c27|          _c28|               _c29|          _c30|
             _c31|_c32|
+--------+----------+----------+-------------+--------+--------------+--------------+-------------+-
----------------+------------+------------------+--------+---------+-----------+------+------------+
-------------+-----------+---------------+----------+------------------+-----------+-----------+----
```

Supprimer les valeurs nan s'ils existent :

```
scala> data=dfc.na.drop()
data: org.apache.spark.sql.DataFrame = [_c1: double, _c2: string ... 30 more fields]
```

4)Requêtes SQL :

Les requêtes SQL seront utilisées pour interroger le dataset et extraire des informations spécifiques. Cela permettra de répondre à des questions précises sur les caractéristiques des tumeurs mammaires et leur lien avec le diagnostic.

cette commande permet l'exécution de requêtes SQL sur le DataFrame data en utilisant le nom de la vue temporaire "breast_cancer" au lieu du nom du DataFrame.

```
scala> data.createOrReplaceTempView("breast_cancer")
```

Exécuter une requête SQL pour filtrer les enregistrements avec une taille de perimeter_mean supérieur à 100 :

```
scala> val result3 = spark.sql("SELECT * FROM breast_cancer WHERE _c4 > 100")
result3: org.apache.spark.sql.DataFrame = [_c1: double, _c2: string ... 30 more fields]

scala> println("R??sultat de la requ??te :")
R??sultat de la requ??te :

scala> result3.show()
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|_c1|_c2|_c3|_c4|_c5|_c6|_c7|_c8|_c9|_c10|_c11|_c12|_c13|_c14|_c15|_c16|_c17|_c18|_c19|_c20|_c21|_c22|_c23|_c24|_c25|_c26|_c27|_c28|_c29|_c30|_c31|_c32|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

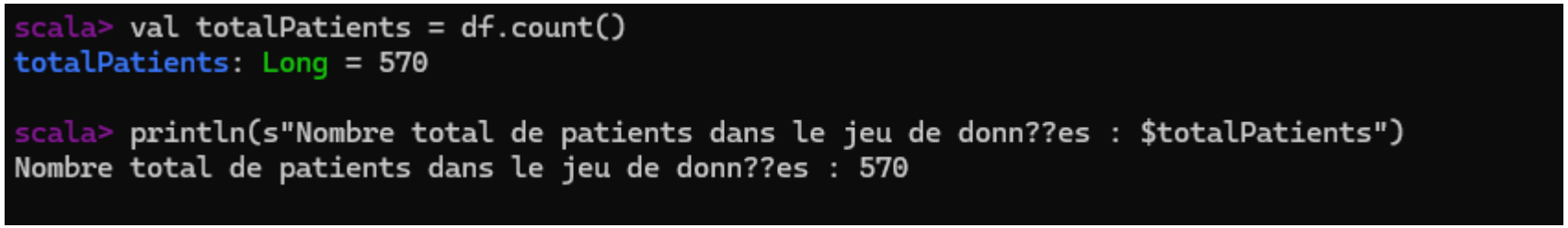
Cela affichera les premières lignes du résultat de la requête, avec les données triées par ordre décroissant de la colonne _c5

```
scala> val result = spark.sql("SELECT * FROM breast_cancer ORDER BY _c5 DESC")
result: org.apache.spark.sql.DataFrame = [_c1: double, _c2: string ... 30 more fields]
```

On peut continuer à exécuter des requêtes SQL afin de fournir des insights sur les caractéristiques des tumeurs, leur distribution, leurs relations, et d'autres facteurs pouvant aider à comprendre la nature des tumeurs malignes et bénignes, ce qui peut être utile pour la prise de décision médicale et la recherche en oncologie.

le nombre total de patients dans le jeu de données :

```
val totalPatients = df.count()
println(s"Nombre total de patients dans le jeu de données : $totalPatients")
```



Nombre de tumeurs malignes et bénignes :

```
val diagnosisDistribution = df.groupBy("diagnosis")
  .count()
  .withColumnRenamed("count", "nombre")
diagnosisDistribution.show
```

Moyenne de la texture des tumeurs :

```
val meanTexture = df.agg(avg("texture_mean")).first().getDouble(0)
println(s"Moyenne de la texture des tumeurs : $meanTexture")
```

Corrélation entre le rayon moyen et la texture moyenne des tumeurs :

```
val correlation = df.stat.corr("radius_mean", "texture_mean")
println(s"Corrélation entre le rayon moyen et la texture moyenne des tumeurs : $correlation")
```

Moyenne de la surface des tumeurs bénignes :

```
val meanAreaForBenignTumors = df.filter($"diagnosis" === "B").agg(avg("area_mean")).first().getDouble(0)
println(s"Moyenne de la surface des tumeurs bénignes : $meanAreaForBenignTumors")
```

Nombre de tumeurs malignes avec un périmètre supérieur à la moyenne :

```
val avgPerimeter = df.agg(avg("perimeter_mean")).first().getDouble(0)
val countMalignantTumorsAboveAvgPerimeter = df.filter($"diagnosis" === "M" && $"perimeter_mean" > avgPerimeter).count()
println(s"Nombre de tumeurs malignes avec un périmètre supérieur à la moyenne : $countMalignantTumorsAboveAvgPerimeter")
```

5 patients avec le rayon moyen le plus élevé :

```
val top5PatientsByRadius = df.orderBy(desc("radius_mean")).select("id", "radius_mean").limit(5)
top5PatientsByRadius.show()
```

Moyenne de la compacité des tumeurs pour chaque diagnostic :

```
val avgCompactnessByDiagnosis = df.groupBy("diagnosis").agg(avg("compactness_mean").alias("avg_compactness"))
avgCompactnessByDiagnosis.show()
```

Corrélation entre la concavité et la concavité des points des tumeurs :

```
val correlationConcavity = df.stat.corr("concavity_mean", "concave points_mean")
println(s"Corrélation entre la concavité et la concavité des points des tumeurs : $correlationConcavity")
```

5)modeles de classification :

finalement ou peut même prédire le type de cancer à l'aide d'un modèle de classification:

-- Entraîner un modèle de classification (ex: LogisticRegression)

```
val model = LogisticRegression.train(
  labelCol = "diagnosis",
  featuresCol = "features",
  data = trainingData
)
```

-- Prédire le type de cancer pour de nouveaux patients

```
val predictions = model.transform(testData)
```

Cette requête utilise une bibliothèque d'apprentissage automatique Scala (ex: MLlib) pour entraîner un modèle de classification qui prédit le type de cancer pour de nouveaux patients. Les résultats du modèle peuvent être évalués en utilisant des mesures telles que la précision et le rappel.

6)Conclusion Globale:

ce document fournit un guide pas à pas pour installer et utiliser Spark Scala sur un cluster Hadoop, ainsi qu'une approche détaillée pour l'analyse du dataset "Breast Cancer Wisconsin". Il met en lumière l'importance des analyses descriptives, des requêtes SQL et des analyses avancées pour explorer les données, découvrir des informations précieuses sur les caractéristiques des tumeurs mammaires et leur lien avec le diagnostic, dans le but de contribuer à la lutte contre le cancer du sein.