Cours
Big Data Analytics

Cycle Ingénieur
INDIA, Semestre 5

Pr. Abderrahim El Qadi
Département Mathématique Appliquée et Génie Informatique
ENSAM, Université Mohammed V de Rabat

A.U. 2023/2024

Partie 2

4. Spark SQL
5. Machine Learning avec Spark
6. Spark Streaming

4. Spark SQL, DataFrames, et Datasets

- Spark SQL est le module Spark de traitement des données structurées.
- Dataset est une nouvelle interface ajoutée dans Spark SQL qui offre tous les avantages RDD avec le moteur d'exécution Spark SQL optimisé.
- DataFrame est un ensemble de données organisées en colonnes nommées, ce qui facilite l'interrogation.
- Le DataFrame équivaut à une table dans n'importe quelle base de données relationnelle.

4.1. DataFrames

- Les DataFrames peuvent être créés en utilisant des RDD existants, des tables Hive et d'autres sources de données telles que des fichiers texte et des bases de données externes.

```
# creation DataFrame en utilisant SparkSession
val df = spark.read.json("examples/src/main/resources/people.json")
```

Opérations des DataFrames

df.show()	Affichage des données
dfs.printSchema()	Affichage de la structure
dfs.select("column-name").show() dfs.select("name").show()	les noms et les colonnes de la liste des dataframes
dfs.filter(dfs("column-name") > value).show()	Filtrage
dfs.groupBy("column-name").count().show()	Compter
df.createOrReplaceTempView("people") sqlDF=spark.sql("select * from people") sqlDF.show()	fonction SQL sur une SparkSession
df.createGlobalTempView("people") park.sql("select * from global_temp.people").show() spark.newSession().sql("Select * from global_temp.people").show()	SQL lors d'une session Spark pour une vue temporaire globale

4.2. Dataset

Exemple : Création un jeu de données Spark :

```
val spark = SparkSession
    .builder()
   .appName("SparkDatasetExample")
   .enableHiveSupport()
   .getOrCreate()
```

- Création un jeu de données à l'aide d'une structure de données de base telle que Plage, Séquence, Liste, etc

```
val ds=spark.range(3)
ds.show()
```

id
0
1
2

- Utilisation de la séquence

```
val ds=Seq(10,11,12).toDS()
```

- Utilisation de List

```
val ds=List(10,11,12).toDS()
```


- Création du dataset à partir de Dataframe à l'aide de Case Class

```
// Seq[Book] -> RDD[Book] -> Dataframe -> Dataset[Book]
case class Book(name: String, cost: Int)
val bookSeq = Seq(Book("Scala", 400), Book("Spark", 500), Book("Kafka", 300))
val bookRDD = sc.parallelize(bookSeq)
val bookDF = bookRDD.toDF()
val bookDS = bookDF.as[Book]
bookDS.show()
```

```
▸ (3) Spark Jobs
▸ bookDF: org.apache.spark.sql.DataFrame = [name: string, cost: integer]
▸ bookDS: org.apache.spark.sql.Dataset[Book] = [name: string, cost: integer]
+-----+----+
| name|cost|
+-----+----+
|Scala| 400|
|Spark| 500|
|Kafka| 300|
+-----+----+
```


- Création un jeu de données à partir de RDD à l'aide de .toDS()

```
val rdd = sc.parallelize(Seq(("Spark",500), ("Scala",400),("Kafka",300)))
val integerDS = rdd.toDS()
integerDS.show()
```

```
▸ (3) Spark Jobs
▸ integerDS: org.apache.spark.sql.Dataset[(String, Int)] = [_1: string, _2: integer]
+-----+---+
|   _1| _2|
+-----+---+
|Spark|500|
|Scala|400|
|Kafka|300|
+-----+---+
```

Opérations sur le jeu de données Spark

1. Exemple de comptage de mots

```
val linesDS = sc.parallelize(Seq("Spark is fast", "Spark has Dataset", "Spark Dataset is typesafe")).toDS()
val wordsDS = linesDS.flatMap(_.toLowerCase.split(" ")).filter(_ != "")
val groupedDS = wordsDS.groupBy("value")
val countsDS = groupedDS.count()
countsDS.show()
```

(5) Spark Jobs
linesDS: org.apache.spark.sql.Dataset[String] = [value: string]
wordsDS: org.apache.spark.sql.Dataset[String] = [value: string]
countsDS: org.apache.spark.sql.DataFrame = [value: string, count: long]

```
+--------+-----+
|   value|count|
+--------+-----+
|typesafe|    1|
|    fast|    1|
|      is|    2|
| dataset|    2|
|   spark|    3|
|     has|    1|
```


2. Convertir le jeu de données Spark en Dataframe

```
val countsDF = countsDS.toDF.orderBy($"count" desc)
countsDF.show()
```

(1) Spark Jobs
countsDF: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [value: string, count: long]

```
+--------+-----+
|   value|count|
+--------+-----+
|   spark|    3|
| dataset|    2|
|      is|    2|
|typesafe|    1|
|    fast|    1|
|     has|    1|
+--------+-----+
```


4.3. JDBC vers des bases de données externes

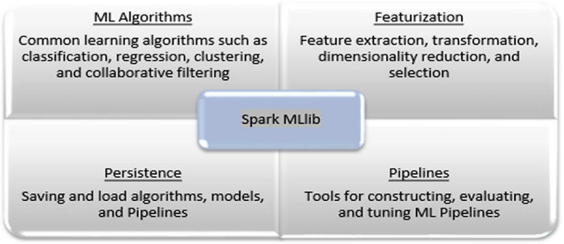
– Spark SQL permet aux utilisateurs de se connecter à des bases de données externes via JDBC.
– Les tables des bases de données peuvent être chargées comme tables temporaires DataFrame ou Spark SQL à l'aide de l'API Datasources.
– Les propriétés suivantes sont obligatoires pour se connecter à la base de données :
  - ***URL***: JDBC URL (e.g., jdbc:mysql://${jdbcHostname}:${jdbcPort}/${jdbcDatabase}).
  - ***Driver***: (e.g., com.mysql.jdbc.Driver, pour mysql database).
  - ***UserName et Password***.

Exemple : Création de dataframe à partir d'une table mysql

```
val jdbcDF = spark.read.format("jdbc")
      .option("url", "jdbc:mysql:localhost:3306/sampleDB")
      .option("dbtable", "sampleDB.bookDetailsTable ")
      .option("user", "<username>")
      .option("password", "<password>")
      .load()
// Enregistrement des données dans une source JDBC
jdbcDF.write.format("jdbc")
      .option("url", " jdbc:mysql:localhost:3306/sampleDB")
      .option("dbtable", "schema.tablename")
      .option("user", "username")
      .option("password", "password")
      .save()
// Spécification des types de données de colonne de table de création lors de l'écriture
jdbcDF.write
  .option("createTableColumnTypes", "name CHAR(64), comments VARCHAR(1024)")
  .jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties)
```

5. Spark Machine Learning

– Spark MLlib est la collection de bibliothèques d'apprentissage automatique (ML) de Spark, qui peut être utilisée en tant qu'API pour implémenter des algorithmes de ML.
– L'objectif global est de rendre le ML pratique évolutif et facile.



– Types de données : Les principales abstractions de données de MLlib sont Vector, LabeledPoint et Rating.

5.1. Type de données : Vector

- Représente une collection indexée de valeurs de type Double avec un index de base zéro de type Int.
- Un vecteur de longueur n représente une observation avec n caractéristiques.

```
scala> val v1 = scala.collection.immutable.Vector.empty
scala>println(v1)
Vector()
scala>val v2 = v1 :+ 5
scala>println(v2)
Vector(5)
scala>val v3 = v2 :+ 10 :+ 20
scala>println(v3)
Vector(5,10,20)
```

– Les valeurs vectorielles peuvent être modifiées à l'aide de la méthode **updated ()** basée sur l'index de éléments.

```
scala> val v3_changed=v3.updated(2,100)
scala> print(v3_changed)
Vector(5, 10, 100)
```

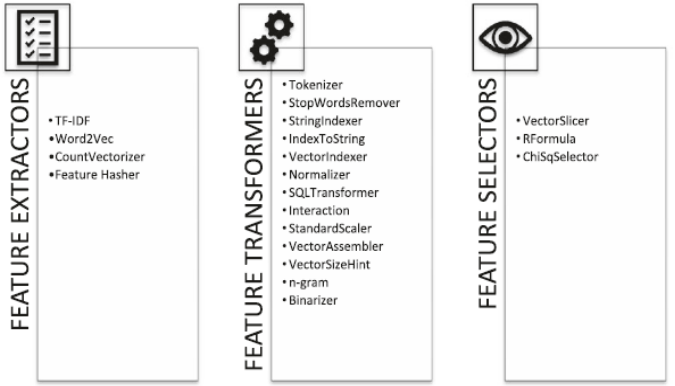

- Représentation vectorielle dans Spark :
  - **Exemple** : création du vecteur dense en important des vecteurs à partir du package spark.ml

```
scala> import org.apache.spark.ml.linalg.{Vector, Vectors}
scala> val densevector=Vectors.dense(1,2,0,0,5)
scala> print(densevector)
[1.0,2.0,0.0,0.0,5.0]
```

  - La même chose peut être créée en tant que vecteurs sparse en spécifiant la taille et les indices des éléments non nuls.
    Classe SparseVector : org.apache.spark.mllib.linalg.SparseVector
    SparseVector(int size, int[] indices, double[] values)

```
scala> val sparseVector=Vectors.sparse(5,Array(0,1,4),Array(1.0,2.0,5.0))
scala> print(sparseVector)
(5,[0,1,4],[1.0,2.0,5.0])
```


5.2. Extraction, transformation et sélection des attributs



5.2.1. Feature extraction

– **Term Frequency–Inverse Document Frequency (TF–IDF)**

- Fréquence du terme TF(t,d) : définie le nombre de fois où terme apparaît dans le document.
- Fréquence de document DF(t,D) : définie le nombre de documents contenant le terme.

TFIDF(t,d,D) = TF(t,d)*IDF(t,D)

IDF(t,D)=log((|D|+1) / DF(t,D)+1)

Exemple :

```
import org.apache.spark.ml.feature.HashingTF
import org.apache.spark.ml.feature.IDF
import org.apache.spark.ml.feature.Tokenizer

val rawData = spark.createDataFrame(Seq(
                (0.0, "This is spark book"),
                (0.0, "published by Apress publications"),
                (1.0, "Dharanitharan wrote this book"))).
                    toDF("label", "sentence")
```



```
val tokenizer = new Tokenizer()
                    .setInputCol("sentence")
                    .setOutputCol("words")
val wordsData = tokenizer.transform(rawData)
val hashingTF = new HashingTF().setInputCol("words")
                                .setOutputCol("rawFeatures")
                                .setNumFeatures(20)
val featurizedData = hashingTF.transform(wordsData)
val idf = new IDF().setInputCol("rawFeatures").setOutputCol("features")
val idfModel = idf.fit(featurizedData)
val rescaledData = idfModel.transform(featurizedData)
rescaledData.select("label", "features").show(false)
```

```
-----------------------+
|label|features                                                                 |
+-----+-------------------------------------------------------------------------+
|0.0  |(20,[6,9,13],[0.6931471805599453,0.28768207245178085,0.0])               |
|0.0  |(20,[0,1,13,15],[0.6931471805599453,0.6931471805599453,0.0,0.6931471805599453])|
|1.0  |(20,[9,13,19],[0.28768207245178085,0.0,0.6931471805599453])              |
+-----+-------------------------------------------------------------
```


5.2.2. Feature transformation

– **Tokenizer les données textuelles :** segmentation d'une phrase complète en mots individuels

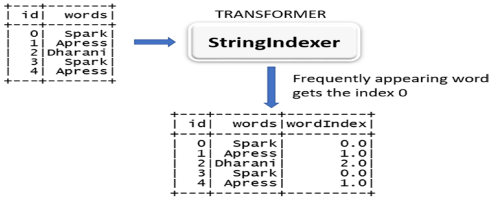
```
scala> import org.apache.spark.ml.feature.Tokenizer
import org.apache.spark.ml.feature.Tokenizer

scala> import org.apache.spark.sql.functions._
import org.apache.spark.sql.functions._

scala> val rawData = spark.createDataFrame(Seq(
     |   (0.0, "This is spark book"),
     |   (0.0, "published by Apress publications"),
     |   (1.0, "Dharanitharan wrote this book")
     |   )).toDF("label", "sentence")
rawData: org.apache.spark.sql.DataFrame = [label: double, sentence: string]

scala> val tokenizer = new Tokenizer().
     |   setInputCol("sentence").
     |   setOutputCol("words")
tokenizer: org.apache.spark.ml.feature.Tokenizer = tok_4ae4da6d943f

scala> val countTokens = udf { (words: Seq[String]) => words.length }
countTokens: org.apache.spark.sql.expressions.UserDefinedFunction

scala> val tokenized = tokenizer.transform(rawData)
tokenized: org.apache.spark.sql.DataFrame

scala> tokenized.select("sentence", "words").
     |   withColumn("tokens", countTokens(col("words"))).
     |   show(false)
+-----------------------------+---------------------------------------+------+
|sentence                     |words                                  |tokens|
+-----------------------------+---------------------------------------+------+
|This is spark book           |[this, is, spark, book]                |4     |
|published by Apress publications|[published, by, apress, publications]|4     |
|Dharanitharan wrote this book|[dharanitharan, wrote, this, book]     |4     |
+-----------------------------+---------------------------------------+------+
```


– **StopWordsRemover :** suppression des mots vides dans les données textuelles

```
scala> import org.apache.spark.ml.feature.StopWordsRemover
import org.apache.spark.ml.feature.StopWordsRemover

scala>

scala> val wordsRemover = new StopWordsRemover().
     |   setInputCol("rawData").
     |   setOutputCol("filtered")
wordsRemover: org.apache.spark.ml.feature.StopWordsRemover = stopWords_a2c7ec9

scala>

scala> val input = spark.createDataFrame(Seq(
     |   (0, Seq("This","is","spark", "book")),
     |   (1, Seq("published","by", "Apress", "publications")))).
     |   toDF("id", "rawData")
input: org.apache.spark.sql.DataFrame = [id: int, rawData: array<string>]

scala> wordsRemover.transform(input).show(false)
+---+---------------------------------------+-------------------------------+
|id |rawData                                |filtered                       |
+---+---------------------------------------+-------------------------------+
|0  |[This, is, spark, book]                |[spark, book]                  |
|1  |[published, by, Apress, publications]|[published, Apress, publications]|
+---+---------------------------------------+-------------------------------+
```

– **StringIndexer :** encode les étiquettes d'une colonne de chaîne dans une colonne d'indices d'étiquettes.
Les indices sont en [0, numLabels), classés par fréquences d'étiquettes, donc l'étiquette la plus fréquente obtient l'indice 0

```
scala> val input = spark.createDataFrame(Seq(
      (0, "Spark"),(1, "Apress"),(2, "Dharani"),(3, "Spark"),
      (4,"Apress"))).toDF("id", "words")
scala> import org.apache.spark.ml.feature.StringIndexer
scala> val indexer = new StringIndexer().
      setInputCol("words").
      setOutputCol("wordIndex")
```



– **Normalizer :** normalise un vecteur pour avoir une norme unitaire.

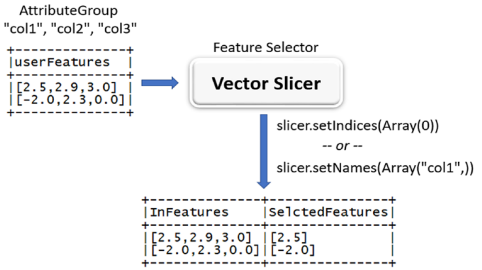
```
scala> import org.apache.spark.ml.feature.Normalizer
scala> val dataFrame = spark.createDataFrame(Seq(
          |      (0, Vectors.dense(1.0, 0.5, -1.0)),
          |      (1, Vectors.dense(2.0, 1.0, 1.0)),
          |      (2, Vectors.dense(4.0, 10.0, 2.0))
          |    )).toDF("id", "features")
scala>  val normalizer = new Normalizer()
                        .setInputCol("features")
                        .setOutputCol("normFeatures")
                        .setP(1.0)
scala>val l1NormData = normalizer.transform(dataFrame)
scala>println("Normalized using L^1 norm")
scala> l1NormData.show()
```

```
| id|     features|     normFeatures|
+---+-------------+-----------------+
|  0|[1.0,0.5,-1.0]|   [0.4,0.2,-0.4]|
|  1|[2.0,1.0,1.0]|   [0.5,0.25,0.25]|
|  2|[4.0,10.0,2.0]|  [0.25,0.625,0.125]|
```


5.2.3. Feature selection

– **VectorSlicer :** accepte une colonne vectorielle avec des indices spécifiés, puis génère une nouvelle colonne vectorielle avec des valeurs sélectionnées via les indices.
Il existe deux types d'indices.
• Indice entier : représente les indices dans le vecteur. Il est représenté par setIndices().
• Indice de chaîne : représente les noms des entités dans le vecteur et représenté par setNames().

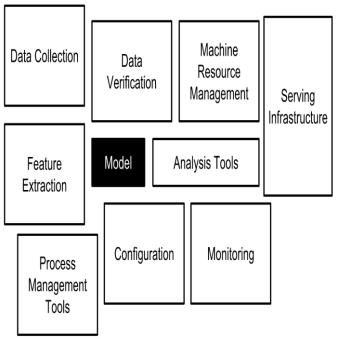
Exemple :




```
scala> val data = Arrays.asList(
     |   Row(Vectors.dense(2.5, 2.9, 3.0)),
     |   Row(Vectors.dense(-2.0, 2.3, 0.0))
     | )
data: java.util.List[org.apache.spark.sql.Row] = [[[2.5,2.9,3.0]], [[-2.0,2.3,0.0]]]

scala>

scala> val defaultAttr = NumericAttribute.defaultAttr
defaultAttr: org.apache.spark.ml.attribute.NumericAttribute = {"type":"numeric"}

scala>

scala> val attrs = Array("col1", "col2", "col3").map(defaultAttr.withName)
attrs: Array[org.apache.spark.ml.attribute.NumericAttribute]
me":"col3"}}

scala>

scala> val attrGroup = new AttributeGroup("InFeatures", attrs.asInstanceOf[Array[Attribute]])
attrGroup: org.apache.spark.ml.attribute.AttributeGroup
3"}]},"num_attrs":3}}

scala>

scala> val dataset = spark.createDataFrame(data, StructType(Array(attrGroup.toStructField())))
dataset: org.apache.spark.sql.DataFrame = [InFeatures: vector]

scala>

scala> val slicer = new VectorSlicer().setInputCol("InFeatures").setOutputCol("SelctedFeatures")
slicer: org.apache.spark.ml.feature.VectorSlicer = vectorSlicer_05c9263c062f

scala>

scala> slicer.setIndices(Array(0))
res54: slicer.type = vectorSlicer_05c9263c062f

scala>

scala> val output = slicer.transform(dataset)
output: org.apache.spark.sql.DataFrame = [InFeatures: vector, SelctedFeatures: vector]

scala> output.show(false)
+--------------+---------------+
|InFeatures    |SelctedFeatures|
+--------------+---------------+
|[2.5,2.9,3.0] |[2.5]          |
|[-2.0,2.3,0.0]|[-2.0]         |
+--------------+---------------+
```

5.3. ML Pipelines

– Le pipeline implique une séquence d'algorithmes pour traiter et construire le modèle en apprenant à partir des données.



Par exemple, un pipeline de traitement de document texte simple peut impliquer les étapes suivantes.

1. segmenter le texte du document en mots.
2. Convertir chaque mot du document en une caractéristique numérique vecteur.
3. Apprendre des données et construire un modèle de prédiction en utilisant les vecteurs de caractéristiques et les étiquettes.

Ces étapes sont les étapes du pipeline. Chaque étape peut être un **transformateur ou un estimateur**.

Exemple1 : Algorithme de Régression linéaire (y = a + b(x)))

– La régression linéaire est une approche linéaire pour modéliser la relation entre la variable dépendante (y) et une ou plusieurs variables indépendantes (x1, x2, ...)

```
scala> import org.apache.spark.ml.regression.LinearRegression
```

1. Création du DataFrame avec des étiquettes de colonne

```
scala> val data = List(
            (2.0, Vectors.dense(1.0)),
            (4.0, Vectors.dense(3.0)),
            (6.0, Vectors.dense(5.0)),
            (8.0, Vectors.dense(7.0))
        )
scala> val inputToModel = data.toDF("label","features")
```

où label est la variable dépendante (c'est-à-dire la valeur à prédire) et les caractéristiques sont les variables indépendantes.

2. Construction du modèle en utilisant LinearRegression()

```
scala> val linearReg = new LinearRegression()
scala> val lrModel = linearReg.fit(inputToModel)
```

3. Les coefficients du model

```
scala> println(s"Coefficients:${lrModel.coefficients}Intercept:${lrModel.intercept}")
```

4. Résumé du modèle

```
scala>   val trainingSummary = linearRegModel.summary
scala> println(s"numIterations: ${trainingSummary.totalIterations}")
scala> println(
     | s"objectiveHistory:[${trainingSummary.objectiveHistory.
     | mkString(",")}]"
     | )
scala> trainingSummary.residuals.show()
    |          residuals|
    +-------------------+
    |-2.66453525910037...|
    |-1.77635683940025...|
    |-1.77635683940025...|
    |                0.0|
```



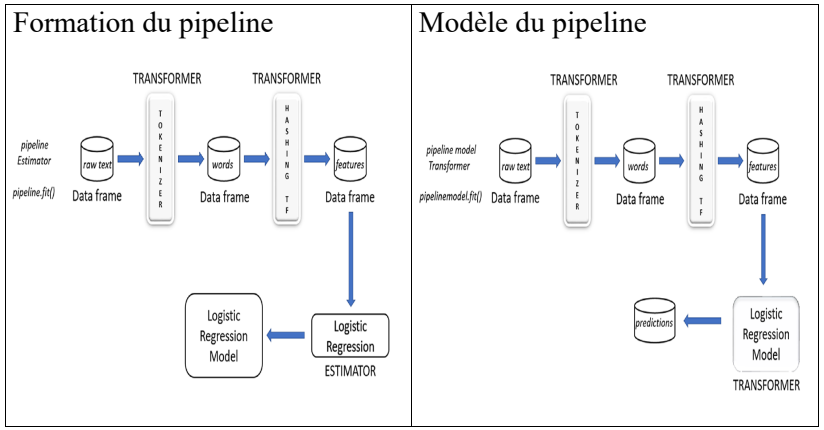
```
scala> println(s"RMSE: ${trainingSummary.rootMeanSquaredError}")
RMSE: 1.831026719408895E-15
scala> println(s"r2: ${trainingSummary.r2}")
r2: 1.0
```

5. Prédiction de l'étiquette de data 9.0

```
scala> val toPredict = List((0.0,Vectors.dense(9.0)),(0.0,Vectors.dense(11.0)))
scala> val toPredictDF = toPredict.toDF("label","features")
scala> val predictions=linearRegModel.transform(toPredictDF)
scala> predictions.select("prediction").show()
    |        prediction|
    +------------------+
    |              10.0|
    |11.999999999999998|
```

Exemple2 : Algorithme de classification : Régression logistique

– La régression logistique spark.ml peut être utilisée pour prédire un résultat binaire (0 ou 1) en utilisant la régression logistique binomiale.



1. Importation des API de pipeline à partir du package spark.ml

```
import org.apache.spark.ml.{Pipeline, PipelineModel}
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.feature.{HashingTF, Tokenizer}
import org.apache.spark.ml.linalg.Vector
import org.apache.spark.sql.Row
```

2. Préparation de données : Schéma: ("id", "text", "label")

```
val training = spark.createDataFrame(Seq(
        (0L, "This is spark book", 1.0),
        (1L, "published by Apress publications", 0.0),
        (2L, "authors are Dharanitharan", 1.0),
        (3L, "and Subhashini", 0.0))).toDF("id", "text", "label")
```

3. Création du pipeline avec : Tokenizer, HashingTF, et l'algorithme de régression logistique

```
val tokenizer = new Tokenizer().setInputCol("text").setOutputCol("words")
val hashingTF = new HashingTF().setNumFeatures(1000)
                    .setInputCol(tokenizer.getOutputCol)
                    .setOutputCol("features")
```



```
val logitreg = new LogisticRegression().setMaxIter(10).setRegParam(0.001)
val pipeline = new Pipeline().setStages(Array(tokenizer, hashingTF,
logitreg))
```

4. Evaluation du pipeline sur les données d'apprentissage

```
val model = pipeline.fit(training)
```

5. Création des documents de test, qui ne sont pas étiquetés. Nous prédisons ensuite l'étiquette en fonction des vecteurs de caractéristiques

```
val test = spark.createDataFrame(Seq(
        (4L, "spark book"),
        (5L, "apress published this book"),
        (6L, "Dharanitharan wrote this book")))
        .toDF("id", "text")
```


6. Réalisation de la prédiction

```
model.transform(test)
    .select("id", "text", "probability", "prediction")
    .collect()
    .foreach {
        case Row(id: Long, text: String, prob: Vector, prediction:
                Double)=>
                println(s"($id, $text) --> prob=$prob, prediction=$prediction")
    }
```

```
(4, spark book) --> prediction=1.0
(5, apress published this book) --> prediction=0.0
(6, Dharanitharan wrote this book) --> prediction=1.0
```

Exemple 3 : Algorithme de clustering (K-Means)

kmeans_data.txt

```
0 1:0.0 2:0.0 3:0.0
1 1:0.1 2:0.1 3:0.1
2 1:0.2 2:0.2 3:0.2
3 1:9.0 2:9.0 3:9.0
4 1:9.1 2:9.1 3:9.1
5 1:9.2 2:9.2 3:9.2
```

```
import org.apache.spark.ml.clustering.KMeans
// Load the dataset in "libsvm" format
scala> val dataset = spark.read.format("libsvm").load("kmeans_data.txt")
// Trains a k-means model by setting the number of clusters as 2.
scala> val kmeans = new KMeans().setK(2).setSeed(1L)
scala> val model = kmeans.fit(dataset)// Make predictions
scala> val predictions = model.transform(dataset)// print the result.
scala> model.clusterCenters.foreach(println)
[9.1,9.1,9.1]
[0.1,0.1,0.1]
```