

# RCP 216 - INGÉNIERIE DE LA FOUILLE ET DE LA VISUALISATION DE DONNÉES MASSIVES

**PROJET** : Australian New South Wales Electricity Market Dataset

# Sommaire :

## I. Introduction

## II. Analyse exploratoire

### 1. Le Dataset

## III. Modélisation

### 1. Division du Dataset

### 2. Encodage

### 3. Normalisation

### 4. Création du modèle

### 5. Evaluation du modèle

### 6. Traitement des flux de données

## IV. Conclusion

# I. Introduction

Le projet consiste à réaliser , à partir de données recueillies auprès du marché australien de l'électricité de la Nouvelle-Galles du sud, un modèle capable de déterminer si le prix de l'électricité va fluctuer à la hausse ou à la baisse.

Sur ce marché, les prix ne sont pas fixes et sont affectés par la demande et l'offre du marché. Le jeu de données ELEC contient 45 312 instances. L'étiquette de classe identifie la variation du prix par rapport à une moyenne mobile des dernières 24 heures.

## II. Analyse exploratoire

### 1- Le Dataset

Nous avons un Dataset constitué de 9 variables ( dont 7 variables doubles et 2 variables catégoriques) et 45312 observations.

```
1 df.printSchema
```

```
root
 |-- _c0: integer (nullable = true)
 |-- date: double (nullable = true)
 |-- day: string (nullable = true)
 |-- period: double (nullable = true)
 |-- nswprice: double (nullable = true)
 |-- nswdemand: double (nullable = true)
 |-- vicprice: double (nullable = true)
 |-- vicedemand: double (nullable = true)
 |-- transfer: double (nullable = true)
 |-- label: string (nullable = true)
```

La première chose à faire est de charger les données puis déterminer le label c'est-à-dire la variable cible ou dépendante. Ici il s'agit de la variable « class ».

```
2
val df = spark.read.format("csv")
    .option("header", "true")
    .option("inferSchema", "true")
    .load("/FileStore/tables/electricite.csv").toDF()
    .withColumnRenamed("class","label")

display(df)
```

	_c0	date	day	period	nswprice	nswdemand	vicprice	vicdemand	transfer	label
1	0	0	b'2'	0	0.056443	0.439155	0.003467	0.422915	0.414912	b'UP'
2	1	0	b'2'	0.021277	0.051699	0.415055	0.003467	0.422915	0.414912	b'UP'
3	2	0	b'2'	0.042553	0.051489	0.385004	0.003467	0.422915	0.414912	b'UP'
4	3	0	b'2'	0.06383	0.045485	0.314639	0.003467	0.422915	0.414912	b'UP'
5	4	0	b'2'	0.085106	0.042482	0.251116	0.003467	0.422915	0.414912	b'DOWN'
6	5	0	b'2'	0.106383	0.041161	0.207528	0.003467	0.422915	0.414912	b'DOWN'
7	6	0	b'2'	0.12766	0.041161	0.171824	0.003467	0.422915	0.414912	b'DOWN'

Après avoir chargé les données, on vérifie s'il n'y a pas de données manquantes. Pour cela il faut compter les valeurs manquantes en additionnant la sortie de la méthode `isNull()`, après l'avoir convertie en type entier.

```
2 import org.apache.spark.sql.avro.functions._
3 df.select(df.columns.map(c => sum(col(c).isNull.cast("int")).alias(c)): _*).show
```

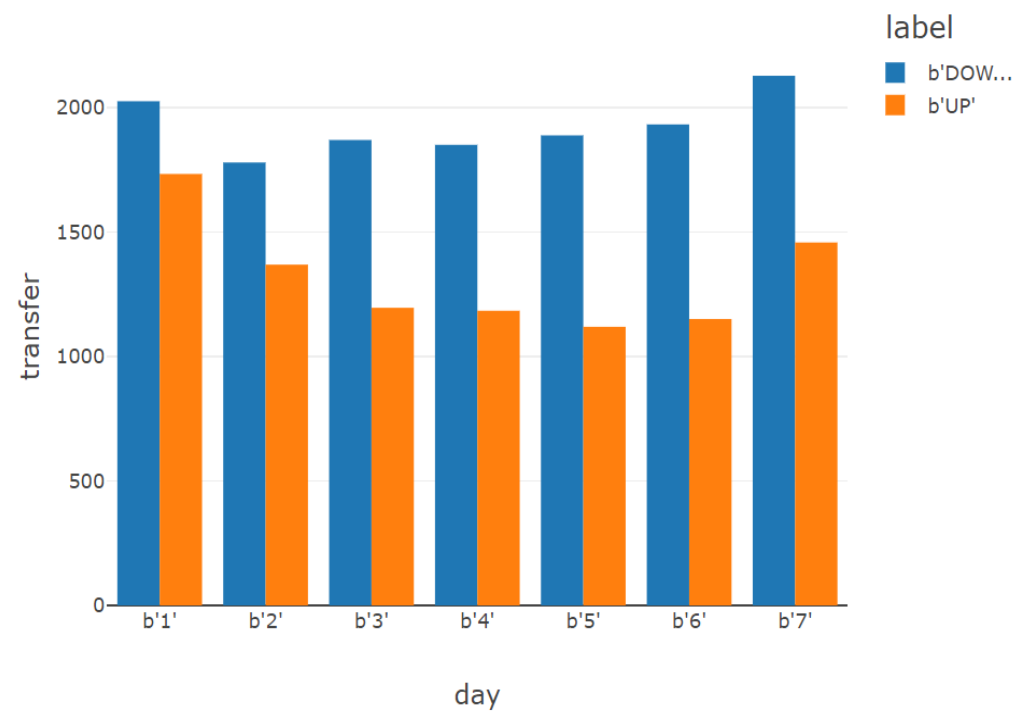
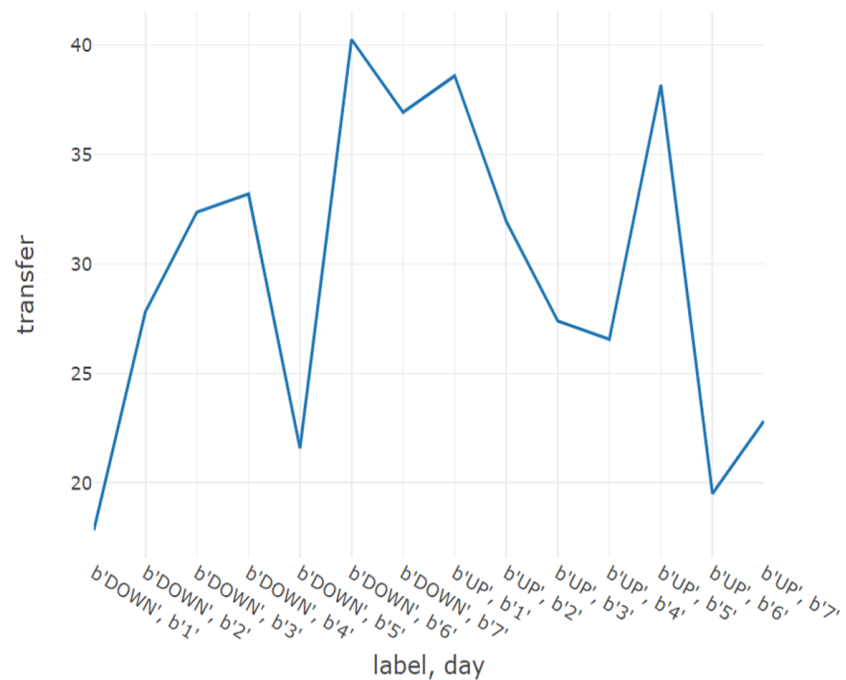
► (2) Spark Jobs

```
+---+-----+---+-----+-----+-----+-----+-----+-----+-----+
|_c0|date|day|period|nswprice|nswdemand|vicprice|vicdemand|transfer|label|
+---+-----+---+-----+-----+-----+-----+-----+-----+-----+
| 0|    0| 0|    0|      0|      0|    0|      0|      0|    0|
+---+-----+---+-----+-----+-----+-----+-----+-----+-----+
```

Nous constatons qu'il n'y a aucune valeur manquantes, ce qui diminue la complexité de traitement des données.

Globalement le prix de l'électricité est volatile d'une semaine à l'autre. Par exemple nous remarquons une pique entre le premier et le quatrième jour jusqu'à +33%, une chute entre le quatrième et le cinquième jour jusqu'à -21%, puis une montée en flèche entre le cinquième et le sixième jour jusqu'à +40%. Par contre semaine suivante, nous constatons une descente progressive entre le premier et le quatrième jour jusqu'à -26% et une remontée entre le quatrième et le cinquième jour jusqu'à +38%, puis une chute à partir du cinquième jour pour atteindre son niveau le plus bas -19%. Les schémas suivants illustrent cette évolution.

## Evolution du prix de l'électricité de la Nouvelle-Galles du sud australien





# III-Modélisation

$\pi$

## 1. Division du Dataset

Avant construction du modèle, nous devons diviser le Dataset en données d'entraînement et en données de test. Le modèle doit être entraîné sur une partie des données et testé sur l'autre pour vérifier la généralisation et la performance, c'est-à-dire le rapport entre les valeurs prédites et les valeurs réelles. Dans ce projet, les données d'entraînement représentent 80% et les données de test 20%.

```
// Découpage en données d'entraînement et de test  
val Array(trainSet, testSet) = df.randomSplit(Array(0.8, 0.2), seed = 1L)
```

## 2. Encodage

Le Dataset est constitué de variables quantitatives et de variables qualitatives. Pour modéliser nous devons transformer les variables qualitatives en variables quantitatives. Pour se faire nous utilisons le transformer de fonctionnalité **StringIndexer**, qui encode une colonne de chaîne d'étiquettes en une colonne d'indices d'étiquettes. Nous transformons les deux variables catégoriques « **day** » et « **label** » en variables numériques. Le code suivant montre comment réaliser ces transformations:

```
import org.apache.spark.ml.feature.StringIndexer

// Indexation de la variable day qui est en string au lieu de double
val dayIndexed = new StringIndexer().setInputCol("day").setOutputCol("dayIndexed")
// Indexation de la variable class qui est en string au lieu de double
val labelIndexed = new StringIndexer().setInputCol("label").setOutputCol("labelIndexed")
```

```
//l'exécution de fit() et transform() sera effectuée par le pipeline,  
//ceci est montré pour expliquer le fonctionnement de fit et transform  
var strIndModel = dayIndexed.fit(trainSet).transform(trainSet)  
var strIndModel_1 = classIndexed.fit(trainSet).transform(trainSet)
```

Par la suite nous utilisons **VectorAssembler**, qui est un transformer qui combine une liste de données de colonnes en une seule colonne vectorielle. Il est utile pour combiner des caractéristiques brutes et des caractéristiques générées par différents transformateurs de caractéristiques en un seul vecteur de caractéristiques, afin de former des modèles ML comme la régression logistique et les arbres de décision. VectorAssembler accepte les types de colonne d'entrée suivants : tous les types numériques, le type booléen et le type vectoriel. Dans chaque ligne, les valeurs des colonnes d'entrée seront concaténées dans un vecteur dans l'ordre spécifié.

Toutes les variables du Dataset sont assemblées en vecteur sauf le label.

```
val assemblerVec = new VectorAssembler().setInputCols(Array("date", "dayIndexed", "period", "nswprice",  
"nswdemand", "vicprice", "vicdemand", "transfer")).setOutputCol("features_assembler")
```

### 3-Normalisation

À présent, il s'agit maintenant de normaliser le Dataset pour que toutes les données soit à la même échelle. Pour cela, on utilise le transformer **Normalizer** qui transforme un ensemble de données de vecteur lignes, normalisant chaque vecteur pour avoir une norme unitaire. Il prend le paramètre  $p$ , qui spécifie la  $p$ -norme utilisée pour la normalisation. ( $p = 2$  par défaut.) Cette normalisation peut améliorer le comportement des algorithmes d'apprentissage.

Le normalisateur prendra la colonne créée par le VectorAssembler « **feature\_assembler** », la normalisera et produira une nouvelle colonne appelée « **features** ».

```
import org.apache.spark.ml.feature.Normalizer
import org.apache.spark.ml.linalg.Vectors

val normalizer = new Normalizer().setInputCol("features_assembler").setOutputCol("features")
```

## 4- Création du modèle

Après avoir testé plusieurs modèles, celui qui a donné le meilleur résultat est le **GBTClassificationModel**, qui sont des ensembles d'arbres de décision. Les GBT forment itérativement des arbres de décision afin de minimiser une fonction de perte. Comme les arbres de décision, les GBT gèrent les caractéristiques catégorielles, s'étendent au paramètre de classification multi classe, ne nécessitent pas de mise à l'échelle des caractéristiques et sont capables de capturer les non-linéarités et les interactions des caractéristiques.

L'amplification de gradient entraîne de manière itérative une séquence d'arbres de décision. À chaque itération, l'algorithme utilise l'ensemble actuel pour prédire l'étiquette de chaque instance d'entraînement, puis compare la prédiction avec la véritable étiquette. L'ensemble de données est réétiqueté pour mettre davantage l'accent sur les instances d'entraînement avec de mauvaises prédictions. Ainsi, à la prochaine itération, l'arbre de décision aidera à corriger les erreurs précédentes. Dans ce cas, au-delà de 20 itérations la performance n'évolue plus, il y a une convergence.

```
import org.apache.spark.ml.classification.{GBTClassificationModel, GBTClassifier}

// Train a GBT model.
val gbt = new GBTClassifier()
    .setLabelCol("labelIndexed")
    .setFeaturesCol("features_assembler")
    .setMaxIter(20)
    .setFeatureSubsetStrategy("auto")
```

Il s'agit maintenant de construire un pipeline dans lequel on mettra les variables «dayIndexed», «labelIndexed», «assemblerVec », «normalizer» et « gbt ».

Le Pipeline est spécifié comme une séquence d'étapes, et chaque étape est soit un Transformer soit un Estimator. Ces étapes sont exécutées dans l'ordre et l'entrée DataFrame est transformée au fur et à mesure qu'elle passe par chaque étape. Pour transformer les étapes, la méthode transform() est appelée sur le DataFrame. Pour estimer les étapes, la méthode fit() est appelée pour produire un Transformer(qui devient une partie du Modèle Pipeline), et cette méthode transform() est appelée sur le DataFrame.

```
import org.apache.spark.ml.Pipeline

val pipeline = new Pipeline().setStages(Array(dayIndexed, labelIndexed, assemblerVec, normalizer, gbt))
```

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

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

```
1 //Nous sélectionnons l'étiquette réelle, la probabilité et les prédictions
2 predictions.select("labelIndexed","probability","prediction").show(5)
```

► (1) Spark Jobs

labelIndexed	probability	prediction
0.0	[0.92702959050635...	0.0
0.0	[0.94176035793038...	0.0
0.0	[0.93984486026627...	0.0
0.0	[0.93984486026627...	0.0
0.0	[0.92717544781612...	0.0



## 5- Evaluation du modèle

Il s'agit maintenant d'évaluer la performance du modèle. Pour cela, nous allons utiliser **BinaryClassificationEvaluator** pour la classification binaire, qui attend des colonnes d'entrée « **Prediction** », « **labelIndexed** » et une colonne de poids facultative. La colonne « Prediction » peut être de type double (prédiction binaire 0/1 ou probabilité d'étiquette 1) ou de type vecteur (vecteur longueur-2 de prédictions brutes, scores ou probabilités d'étiquette).

```
1 import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
2
3 val binaryClassificationEvaluator = new BinaryClassificationEvaluator()
4   .setLabelCol("labelIndexed")
5   .setRawPredictionCol("prediction")
6
7 def printlnMetric(metricName: String): Unit = {
8   println(metricName + " = " + binaryClassificationEvaluator.setMetricName(metricName).evaluate(predictions))
9 }
10
11 printlnMetric("areaUnderROC")
12 printlnMetric("areaUnderPR")
```

► (7) Spark Jobs

areaUnderROC = 0.8192462149214805

areaUnderPR = 0.7736517668051821

Pour le modèle nous obtenons une performance de 81%, ce qui n'est pas mal.

## 6- Traitement des flux de données

Dans ce projet, nous allons décomposer les données de test en 10 fichiers différents et les écrivons dans un fichier csv. Par la suite on crée un répertoire « ElecTest » où seront stockées les différentes partitions de flux.

```
val testData = testSet.repartition(10)
```

```
testData.write.format("CSV").option("header","true").save("FileStore/tables/ElecTest/")
```

- amazon.csv
- electricite.csv
- medsamp2016a.xml
- medsamp2016b.xml
- medsamp2016c.xml
- medsamp2016d.xml
- medsamp2016e.xml
- medsamp2016f.xml
- medsamp2016g.xml
- medsamp2016h.xml

- ElectTest

- stream\_csv

- \_SUCCESS
- \_committed\_7118909543047996685
- \_started\_7118909543047996685
- part-00000-tid-711890954304799...
- part-00001-tid-711890954304799...
- part-00002-tid-711890954304799...
- part-00003-tid-711890954304799...
- part-00004-tid-711890954304799...
- part-00005-tid-711890954304799...
- part-00006-tid-711890954304799...
- part-00007-tid-711890954304799...
- part-00008-tid-711890954304799...
- part-00009-tid-711890954304799...

On redéfinit la structure ou schéma du Dataset avant de créer une variable de source de lecture du flux de données.

```
import org.apache.spark.sql.types.{StructType, StructField, StringType, DoubleType, IntegerType}

val schema = StructType(
  Array(StructField("date", DoubleType),
    StructField("day", StringType),
    StructField("period", DoubleType),
    StructField("nswprice", DoubleType),
    StructField("nswdemand", DoubleType),
    StructField("vicprice", DoubleType),
    StructField("vicedemand", DoubleType),
    StructField("transfer", DoubleType),
    StructField("class", StringType)
  ))
```


```
val sourceStream = spark.readStream.format("csv")  
    .option("header","true")  
    .schema(schema)  
    .option("ignoreLeadingWhiteSpace","true")  
    .option("mode","dropMalformed")  
    .option("maxFilesPerTrigger",1)  
    .load("/FileStore/tables/ElectTest")  
    .withColumnRenamed("class","label")
```

Enfin on applique le modèle au flux pour prédire la fluctuation à la hausse ou à la baisse du prix de l'électricité de la nouvelle Galle du sud en Australie. Comme nous pouvons le voir, le feu vert est allumé, ce qui indique que nous diffusons des données invisibles, qui proviennent des données de test qui ont été repartitionnées pour reproduire la simulation à des fins de diffusion.

```
7  val streamingElec = model.transform(sourceStream).select("label","probability","prediction")
8
9  display(streamingElec)
```

Cancel

▶ (1) Spark Jobs 

▶  display\_query\_4 (id: 6c5c6b74-403c-4dcc-8ca0-1f792d17b6e6) *Last updated: 5 seconds ago*

## IV- Conclusion

Le projet a été réalisé avec un algorithme de classification GBTClassificationModel, avec lequel nous avons obtenu une performance de 81%. Les autres algorithmes comme la régression logistique , l'arbre de décision et la forêt aléatoire ont donné une performance de 76%. On constate globalement une volatilité des prix de l'électricité à la Nouvelle-Galles du sud en Australie. Cela n'est pas surprenant car il y a toujours une spéculation sur les énergies. Les producteurs mondiaux augmentent ou baisse leurs productions pour jouer sur les prix. Ce phénomène n'a jamais été aussi d'actualité qu'aujourd'hui avec la guerre en Ukraine. La solution à ce problème ne viendrai t'il pas des consommateurs ? La sobriété est-il un moyen pour juguler ce phénomène?



# Références

1. <https://moa.cms.waikato.ac.nz/datasets/>
2. <https://spark.apache.org/docs/latest/ml-guide.html>