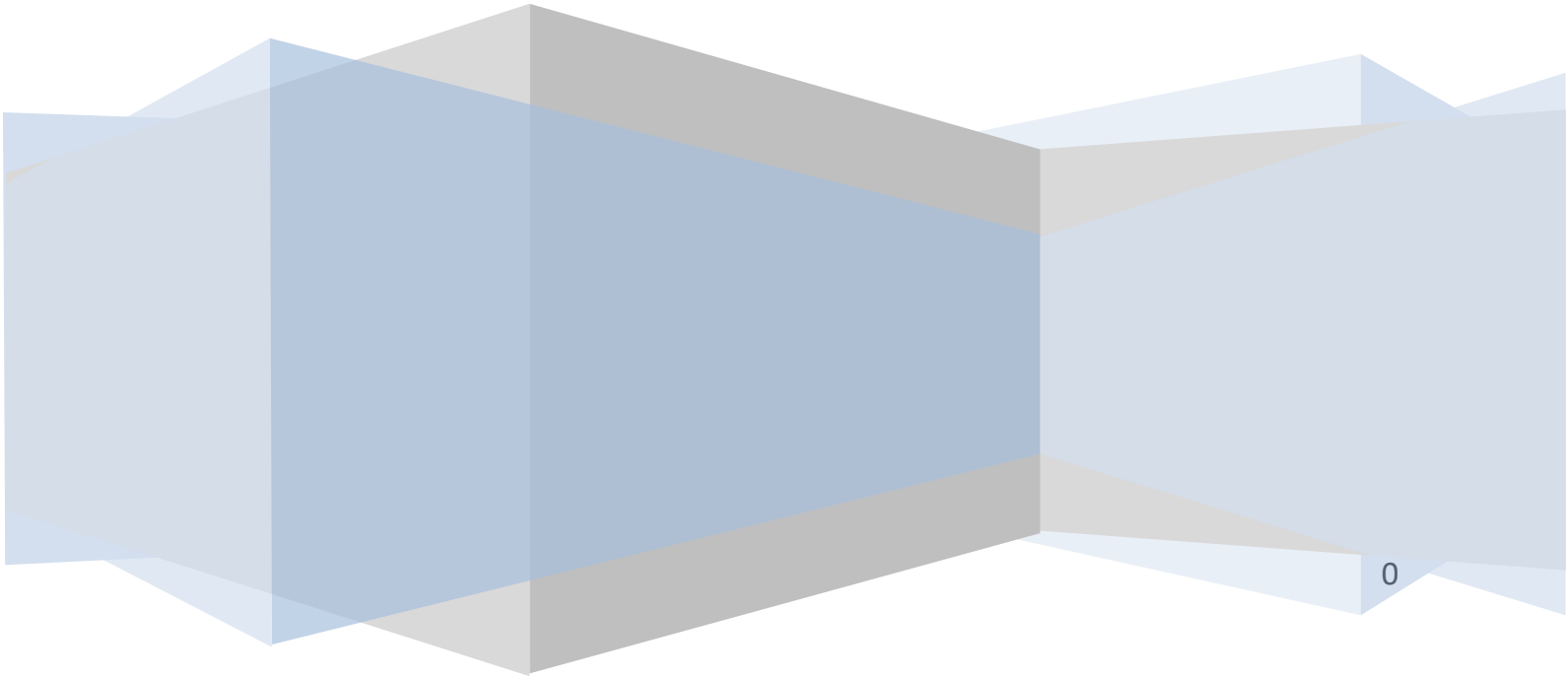
Ecole Des Sciences de L'Information

Rapport Du Projet Spark

Mohamed Amine Bihani



Sommaire

Résumé

Le projet que j'ai réalisé fonctionne sur une base de données de céréales qui inclut plusieurs informations et caractéristiques sur différentes céréales. On traite donc ces céréales selon leurs caractéristiques mais aussi en prenant en considération les notes donnés par plusieurs clients (On prend en considération juste une moyenne des notes).

Le clustering ce fait sans prendre en considération l'entreprise créatrice du céréal, laissant ainsi seulement les caractéristiques affecter les clusters.

La classification se fait en prenant compte de tous les critères des céréales ; elle permet de prédire l'entreprise à partir des caractéristiques de chaque céréale.

La régression se fait aussi pour la prédiction des entreprises.

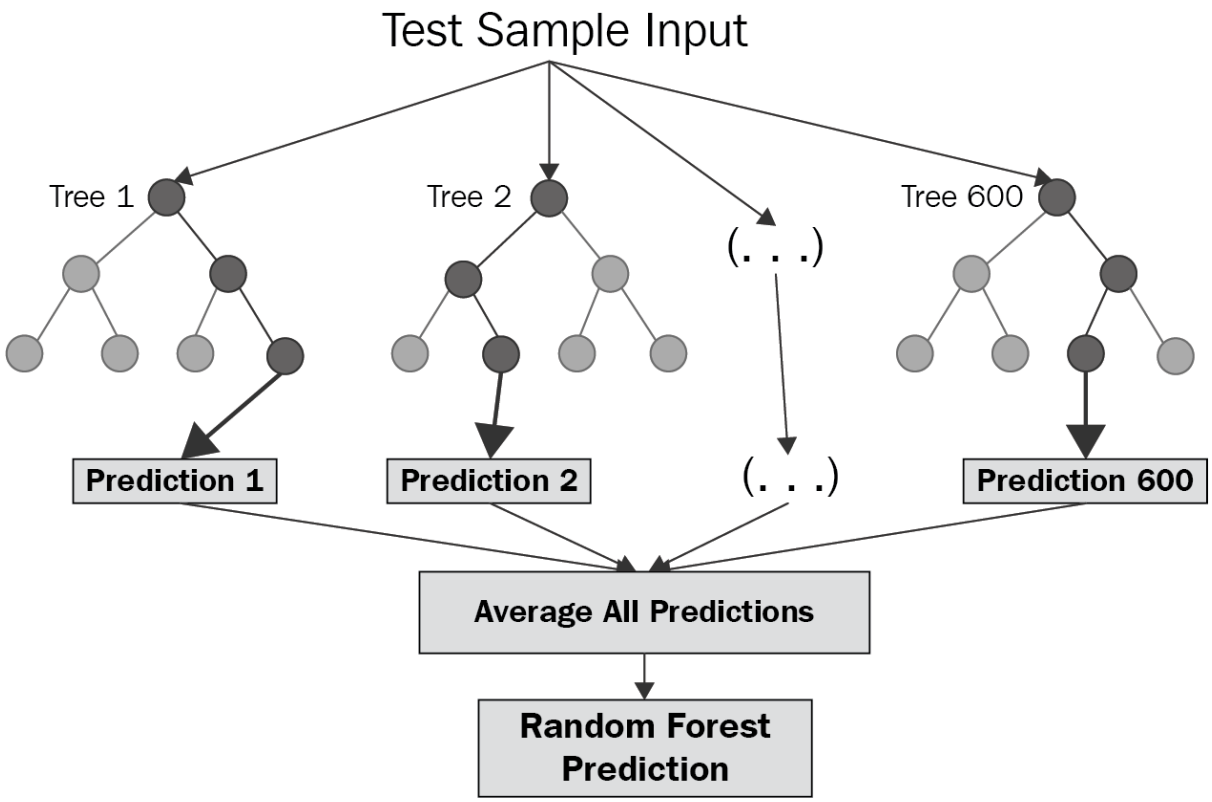
Le système de recommandation permet de prendre en considération un vecteur des caractéristiques et de recommander la céréale étant le plus proche de celui que vous préférez.

Conception

Notre Dataset contient pour la plupart des données numériques, ce qui nous permettras d'utiliser une régression grâce random forest regressor qui utilise une technique de fonctionnement en parallèle du modèle pour la classification et pour la régression. Je l'ai choisi car ce qu'on cherche à trouver c'est comment les caractéristiques des céréales affectent le fabricant, donc on a besoin de plusieurs modèles pour qu'on puisse trouver cette optimalité dans la prédiction.

Du même principe pour la classification, on utilise RandomForestClassifier : comme son nom l'indique, RDF se compose d'un grand nombre d'arbres de décision individuels qui fonctionnent comme un ensemble. Chaque arbre individuel dans la forêt aléatoire crache une prédiction de classe et leur moyenne devient la prédiction de notre modèle. Ce modèle fonctionne aussi bien avec une basse corrélation (ce qui est notre cas). Ceci permet d'augmenter l'efficacité des features à décider de la prédiction.

Le modele RandomForest suit le principe suivant :



Puisque toutes nos données (presque) sont numériques, on va aussi changer les fabricants vers des valeurs indexés qui faciliteront le travail.

Le Clustering est très efficace en utilisant K-means, elle nous permettras de trouver les centres des clusters selon les features des céréales, par la suite on verras même l'appartenance de chaque céréale à un certain cluster indexé. J'ai choisit de limiter les clusters à 3 grâce à la silhouette qui a donné une valeur maximale dans le nombre de clusters : 3, puisqu'elle sert à trouver la meilleur répartition pour les clusters selon la moyenne des silhouettes des points, plus elle est grande, plus c'est mieux.

Bisecting k-means : j'ai choisit d'ajouter un autre clustering pour voir l'efficacité du premier clustering, et puisque celui la fonctionne beaucoup mieux sur un grand nombre de features. En plus il réduit le temps d'exécution et produit des clusters de taille similaire.

Réalisation

Préparation et Visualisation

On prépare l'environnement spark sur Jupyter Notebook dans windows, en plus de l'importation de quelques fonctions nécessaires.

Preparing environment

```
]: import findspark
   findspark.init()
```

```
]: from pyspark import StorageLevel
   import pyspark.sql.functions as sql_func
   from pyspark.sql.types import *
   from pyspark.ml.recommendation import ALS, ALSModel
   from pyspark.context import SparkContext
   from pyspark.sql.session import SparkSession

   sc = SparkContext('local')
   spark = SparkSession(sc)
```

Importation du dataset dans une dataframe Spark.

Dataset

https://www.kaggle.com/crawford/80-cereals

```
In [3]: data= spark.read.load(r'C:\Users\HP\Downloads\cereal.csv',
                              format='csv',
                              sep = ';',
                              header='true',
                              escape='"',
                              inferSchema='true')
```

Visualisation du schéma du Dataframe

```
data.printSchema()
```

```
root
 |-- name: string (nullable = true)
 |-- mfr: string (nullable = true)
 |-- type: string (nullable = true)
 |-- calories: string (nullable = true)
 |-- protein: string (nullable = true)
 |-- fat: string (nullable = true)
 |-- sodium: string (nullable = true)
 |-- fiber: string (nullable = true)
 |-- carbo: string (nullable = true)
 |-- sugars: string (nullable = true)
 |-- potass: string (nullable = true)
 |-- vitamins: string (nullable = true)
 |-- shelf: string (nullable = true)
 |-- weight: string (nullable = true)
 |-- cups: string (nullable = true)
 |-- rating: string (nullable = true)
```

On réalise une visualisation par pandas afin de voir le contenu du dataframe.

```
import pandas as pd
pd.DataFrame(data.take(10), columns= data.columns)
```

	name	mfr	type	calories	protein	fat	sodium	fiber	carbo	sugars	potass	vitamins	shelf	weight	cups	rating
0	String	Categorical	Categorical	Int	Int	Int	Int	Float	Float	Int	Int	Int	Int	Float	Float	Float
1	100% Bran	N	C	70	4	1	130	10	5	6	280	25	3	1	0.33	68.402973
2	100% Natural Bran	Q	C	120	3	5	15	2	8	8	135	0	3	1	1	33.983679
3	All-Bran	K	C	70	4	1	260	9	7	5	320	25	3	1	0.33	59.425505
4	All-Bran with Extra Fiber	K	C	50	4	0	140	14	8	0	330	25	3	1	0.5	93.704912
5	Almond Delight	R	C	110	2	2	200	1	14	8	-1	25	3	1	0.75	34.384843
6	Apple Cinnamon Cheerios	G	C	110	2	2	180	1.5	10.5	10	70	25	1	1	0.75	29.509541
7	Apple Jacks	K	C	110	2	0	125	1	11	14	30	25	2	1	1	33.174094
8	Basic 4	G	C	130	3	2	210	2	18	8	100	25	3	1.33	0.75	37.038562
9	Bran Chex	R	C	90	2	1	200	4	15	6	125	25	1	1	0.67	49.120253

On remarque que les types des colonnes est stocké dans la première ligne du dataframe, donc il faudra faire un pre processing sur les données afin d'attribuer le type de données et d'enlever cette ligne par la suite.

Pre-Processing

On commence notre pre-processing par l'attribution des types de données :

```
data = data \
    .withColumn("calories", col("calories").cast(IntegerType())) \
    .withColumn("protein", col("protein").cast(IntegerType())) \
    .withColumn("sodium", col("sodium").cast(IntegerType())) \
    .withColumn("sugars", col("sugars").cast(IntegerType())) \
    .withColumn("potass", col("potass").cast(IntegerType())) \
    .withColumn("shelf", col("shelf").cast(IntegerType())) \
    .withColumn("vitamins", col("vitamins").cast(IntegerType())) \
    .withColumn("weight", col("weight").cast(FloatType())) \
    .withColumn("cups", col("cups").cast(FloatType())) \
    .withColumn("rating", col("rating").cast(FloatType())) \
    .withColumn("fiber", col("fiber").cast(FloatType())) \
    .withColumn("carbo", col("carbo").cast(FloatType())) \
    .withColumn("sodium", col("sodium").cast(IntegerType())) \
    .withColumn("fat", col("fat").cast(DoubleType()))
```

On utilise la fonction cast pour modifier le type selon ce qui a été spécifié.

```
data.printSchema()
```

```
root
 |-- name: string (nullable = true)
 |-- mfr: string (nullable = true)
 |-- type: string (nullable = true)
 |-- calories: integer (nullable = true)
 |-- protein: integer (nullable = true)
 |-- fat: double (nullable = true)
 |-- sodium: integer (nullable = true)
 |-- fiber: float (nullable = true)
 |-- carbo: float (nullable = true)
 |-- sugars: integer (nullable = true)
 |-- potass: integer (nullable = true)
 |-- vitamins: integer (nullable = true)
 |-- shelf: integer (nullable = true)
 |-- weight: float (nullable = true)
 |-- cups: float (nullable = true)
 |-- rating: float (nullable = true)
```

Le nouveau schéma inclut les types de variables spécifiques à chaque colonne.

Maintenant on veut supprimer la ligne ayant les types, pour cela on réalise cette commande :

```
data= data.filter((data.name != 'String'))
```

Maintenant on a nos données qui sont prêts

```
pd.DataFrame(data.take(10), columns= data.columns)
```

	name	mfr	type	calories	protein	fat	sodium	fiber	carbo	sugars	potass	vitamins	shelf	weight	cups	rating
0	100% Bran	N	C	70	4	1.0	130	10.0	5.0	6	280	25	3	1.00	0.33	68.402969
1	100% Natural Bran	Q	C	120	3	5.0	15	2.0	8.0	8	135	0	3	1.00	1.00	33.983681
2	All-Bran	K	C	70	4	1.0	260	9.0	7.0	5	320	25	3	1.00	0.33	59.425507
3	All-Bran with Extra Fiber	K	C	50	4	0.0	140	14.0	8.0	0	330	25	3	1.00	0.50	93.704910
4	Almond Delight	R	C	110	2	2.0	200	1.0	14.0	8	-1	25	3	1.00	0.75	34.384842
5	Apple Cinnamon Cheerios	G	C	110	2	2.0	180	1.5	10.5	10	70	25	1	1.00	0.75	29.509541
6	Apple Jacks	K	C	110	2	0.0	125	1.0	11.0	14	30	25	2	1.00	1.00	33.174095
7	Basic 4	G	C	130	3	2.0	210	2.0	18.0	8	100	25	3	1.33	0.75	37.038563
8	Bran Chex	R	C	90	2	1.0	200	4.0	15.0	6	125	25	1	1.00	0.67	49.120255
9	Bran Flakes	P	C	90	3	0.0	210	5.0	13.0	5	190	25	3	1.00	0.67	53.313812

On sauvegarde nos données non numériques

```
columns_to_drop = ['name', 'mfr', 'type']
cat = data.select(*columns_to_drop)
```

Mais on souhaite aussi prendre que les valeurs numériques pour les étapes suivantes, donc :

```
columns_to_drop = ['name', 'mfr', 'type']
data = data.drop(*columns_to_drop)
```

Maintenant on a notre Dataframe qui est prête.

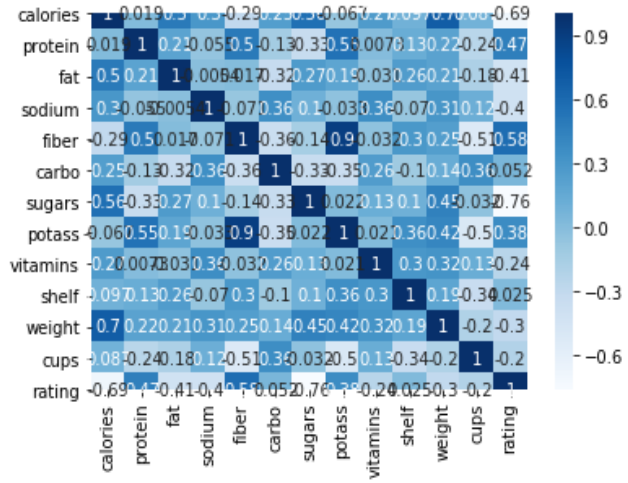
Corrélation

On utilise pyplot pour l'affichage des données, et on transforme nos données numériques vers pandas pour pouvoir utiliser un affichage des corrélations (et non seulement des valeurs)

```
import matplotlib.pyplot as plt
import seaborn as sns
```

```
corr = data.select("*").toPandas()
```

```
cor = corr.corr()
sns.heatmap(cor, annot=True, cmap='Blues')
plt.show()
```



Classification

Pour la classification, on utilisera les noms d'entreprise donc il faudra changer ces noms vers des numéros indexés :

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer


indexer = StringIndexer().setInputCol("mfr").setOutputCol("mfrIndex").fit(dt)
indexed = indexer.transform(dt)
```

Ensuite on créera notre vecteur des caractéristiques (dense Features Vector)

```
from pyspark.ml.feature import VectorAssembler


train_columns = [col for col in indexed.columns if col not in ['name', 'mfr', 'type']]


assembler = VectorAssembler().setInputCols(train_columns).setOutputCol("features")
train_mod01 = assembler.transform(indexed)
```

On prend les features et les noms d'entreprises(mfrindex)

```
train_new= train_mod01.select("features", "mfrindex")
train_new.limit(5).toPandas()
```

	features	mfrindex
0	[70.0, 4.0, 1.0, 130.0, 10.0, 5.0, 6.0, 280.0,...	5.0
1	[120.0, 3.0, 5.0, 15.0, 2.0, 8.0, 8.0, 135.0, ...	3.0
2	[70.0, 4.0, 1.0, 260.0, 9.0, 7.0, 5.0, 320.0, ...	0.0
3	[50.0, 4.0, 0.0, 140.0, 14.0, 8.0, 0.0, 330.0,...	0.0
4	[110.0, 2.0, 2.0, 200.0, 1.0, 14.0, 8.0, -1.0,...	4.0

Et on utilise RandomForestClassifier : comme son nom l'indique, RDF se compose d'un grand nombre d'arbres de décision individuels qui fonctionnent comme un ensemble. Chaque arbre individuel dans la forêt aléatoire crache une prédiction de classe et la classe avec le plus de votes devient la prédiction de notre modèle. Ce modèle fonctionne aussi bien avec une basse corrélation (ce qui est notre cas).

```
train_rf, test_rf = train_new.randomSplit([0.8, 0.2], seed=10000)
```

```
from pyspark.ml.classification import RandomForestClassifier
rf = RandomForestClassifier(featuresCol = 'features',labelCol = "mfrindex", numTrees = 100)
```

```
rfModel = rf.fit(train_rf)
predictions_rf = rfModel.transform(test_rf)
evaluator_rf = MulticlassClassificationEvaluator(labelCol='mfrindex', predictionCol='prediction', metricName="accuracy")
accuracy_rf = evaluator_rf.evaluate(predictions_rf)
print('rf '+ str(accuracy_rf))
```

```
rf 1.0
```

Et on visualise notre prédiction :

```
rf 1.0
```

In [61]:
```
predictions_rf.select("features", "probability", "prediction","mfrindex").limit(5).toPandas()
```

Out[61]:

	features	probability	prediction	mfrindex
0	[50.0, 2.0, 0.0, 0.0, 1.0, 10.0, 0.0, 50.0, 0....	[0.07, 0.00125, 0.056428571428571425, 0.578333...	3.0	3.0
1	[90.0, 2.0, 0.0, 0.0, 2.0, 15.0, 6.0, 110.0, 2...	[0.5980054945054945, 0.0359230769230769 2, 0.13...	0.0	0.0
2	[90.0, 2.0, 0.0, 15.0, 3.0, 15.0, 5.0, 90.0, 2...	[0.04454822954822954, 0.01808760683760684, 0.2...	5.0	5.0
3	[90.0, 3.0, 0.0, 0.0, 3.0, 20.0, 0.0, 120.0, 0...	[0.0, 0.0, 0.010000000000000002, 0.06809523809...	5.0	5.0
4	[90.0, 3.0, 0.0, 170.0, 3.0, 18.0, 2.0, 90.0, ...	[0.713167277167277, 0.052641636141636146, 0.10...	0.0	0.0

Clustering

Pour le clustering on utiliseras Kmeans et pour son évaluation le ClusteringEvaluator.

```
from pyspark.ml.clustering import KMeans
from pyspark.ml.evaluation import ClusteringEvaluator


# Trains a k-means model.
kmeans = KMeans().setK(3).setSeed(1)
model = kmeans.fit(clustering)

# Make predictions
predictions = model.transform(clustering)

# Evaluate clustering by computing Silhouette score
evaluator = ClusteringEvaluator()

silhouette = evaluator.evaluate(predictions)
print("Silhouette with squared euclidean distance = " + str(silhouette))

# Shows the result.
centers = model.clusterCenters()
print("Cluster Centers: ")
for center in centers:
    print(center)
```

```
Silhouette with squared euclidean distance = 0.6570607412705105
Cluster Centers:
[112.14285714   3.35714286   1.42857143 172.5          5.60714286
  12.03571429   9.14285714 217.14285714  30.35714286   2.92857143
   1.16000001   0.66142858  46.01531901   1.5        ]
[93.75        2.5625      0.75       24.375       1.85625    13.25
  4.625      78.6875     12.5         2.          0.926875    0.829375
 54.03677905  2.9375    ]
[109.78723404   2.29787234   0.9787234  201.91489362   1.22340426
  15.81914894   7.04255319  65.93617021  32.9787234    2.06382979
   1.02574468   0.86574468  37.79694371   1.36170213]
```

Le modèle trouve les clusters comme spécifié selon la distance carrée euclidienne du centre de chaque cluster.

Un autre essai avec le Bisecting K-means va nous donner le même résultat (avec une petite différence dans les distances)

```python
from pyspark.ml.clustering import BisectingKMeans


# Trains a bisecting k-means model.
bkm = BisectingKMeans().setK(3).setSeed(1)
model = bkm.fit(clustering)

# Evaluate clustering.
cost = model.computeCost(clustering)
print("Within Set Sum of Squared Errors = " + str(cost))

# Shows the result.
print("Cluster Centers: ")
centers = model.clusterCenters()
for center in centers:
    print(center)
```

```
Within Set Sum of Squared Errors = 429598.4602857565
Cluster Centers:
[109.72222222   2.05555556   0.88888889 193.88888889   0.875
  15.625        7.52777778  54.41666667  31.25         2.11111111
   1.00833333   0.87305556  36.32414352   1.36111111]
[93.75         2.5625      0.75        24.375        1.85625    13.25
  4.625       78.6875     12.5          2.           0.926875    0.829375
 54.03677905  2.9375    ]
[111.2          3.24         1.36       197.          4.18
  13.98         7.52       167.2         34.          2.48
   1.12600001   0.74080001  44.52006615   1.44      ]
```

Régression

Pour la régression on utilisera random forest regressor qui utilise le même principe du classifier pour la régression.

Pendant la régression on calculera l'erreur (RMSE) qui mesure la difference entre la prédiction et la réalité.

```
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator


# Split the data into training and test sets (30% held out for test
(trainingData, testData) = train_reg.randomSplit([0.8, 0.2])

# Train a RandomForest model.
rf = RandomForestRegressor(featuresCol="features",labelCol = "mfrin


# Train model.  This also runs the indexer.
model = rf.fit(trainingData)
# Make predictions.
predictions = model.transform(testData)

# Select example rows to display.
predictions.select("prediction", "mfrindex", "features").show(5)

# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(
    labelCol="mfrindex", predictionCol="prediction", metricName="rm
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

```
+------------------+--------+--------------------+
|        prediction|mfrindex|            features|
+------------------+--------+--------------------+
| 3.9118421052631582|     4.0|[90.0,2.0,1.0,200...|
|              4.85|     5.0|[90.0,3.0,0.0,0.0...|
| 2.0353269537480063|     2.0|[90.0,3.0,0.0,210...|
| 1.7600000000000002|     2.0|[100.0,2.0,0.0,45...|
|0.22472826086956524|     0.0|[100.0,2.0,0.0,29...|
+------------------+--------+--------------------+
only showing top 5 rows

Root Mean Squared Error (RMSE) on test data = 0.464004
```

On distingue que la prédiction est très exacte si on applique une approximation.

Recommendation Engine

Je n'ai pas pu réaliser un système de recommendation classique car mes données n'avait pas un User ID ou des avis par utilisateurs pour la méthode collaborative filtering ; ni des données textuels de description pour la méthode content based recommendation engine.

Donc j'ai utilisé une approche semblable à celle du content-based method.

J'ai pris l'input de l'utilisateur sur son céréal préféré, puis j'ai comparé le vecteur des features de ce céréal avec tous les autres céréals, et j'ai pris le céréal ayant la moindre différence (qui est différente de 0), donc qui est le plus proche du céréal choisit.

J'ai créé cette fonction par moi-même en 3 étapes :

Fonction qui donne la ligne selon la valeur :

```
def getrow(value):
    row=df_subset.loc[df_subset['name']== value]
    return row
```

Fonction qui calcule la moyenne du vecteur :

```
def average(a):

    # Find sum of array element
    sum = 0
    n=13
    for i in range(n):
        sum += a[i]

    return sum/n;
```

Et enfin la fonction qui donne le résultat final, utilisant les fonctions précédentes, la fonction crée une nouvelle colonne qui contient la différences des vecteurs et puis choisit la ligne correspondante selon la valeur minimale non nulle de la moyenne des features.

```
def recom(value):

    t=getrow(value)
    newdf = pd.DataFrame(np.repeat(t.values,77,axis=0))
    vecsub = lambda x, y: np.subtract(x, y)
    df_subset['dV'] = list(map(vecsub, df_subset['features'], newdf[1]))
    df_subset['dV']=df_subset['dV'].abs()
    for i in range(len(df_subset)) :
        df_subset.loc[i, "val"] = average(df_subset.loc[i, "dV"])

    for i in range(len(df_subset)) :
        x =df_subset.loc[df_subset['val'] > 0, 'val'].min()

    result=df_subset.loc[df_subset['val'] == x, 'name']
    print('On vous recommende', result)
```

Enfin pour le test du système de recommendation :

```
recom('Almond Delight')
```

```
On vous recommende 25    Frosted Flakes
```

Tuning

Le premier Tuning sur la classification RandomForestClassifier va nous donner une prédiction moins performante, mais renderas le modèle plus performant sur le long-terme sur d'autres données.

```
In [60]: paramGrid = (ParamGridBuilder()
                      .addGrid(rf.maxDepth,[7,9,12,14,17])
                      .addGrid(rf.maxBins,[20,80])
                      .build())
         cv = CrossValidator(estimator=rf, estimatorParamMaps=paramGrid, evaluator=evaluator_rf, numFolds=5)
         cvModel = cv.fit(train_rf)
         predictions = cvModel.transform(test_rf)
         accuracy=evaluator_rf.evaluate(predictions)
         print('tunning '+ str(accuracy))

         tunning 0.9444444444444444
```

Ensuite on utilise le MultiClass Classification Evaluator pour évaluer notre classification

```
train_rf, test_rf = train_new.randomSplit([0.8, 0.2], seed=12345)
```

```
Model = crossval.fit(train_rf)
```

```
predictions = Model.transform(test_rf)
```

```
predictions.select("features", "probability", "prediction","mfrindex").limit(5).toPandas()
```

	features	probability	prediction	mfrindex
0	[90.0, 2.0, 0.0, 0.0, 2.0, 15.0, 6.0, 110.0, 2...	[0.5141221592028044, 0.0775617818561367, 0.114...	0.0	0.0
1	[100.0, 3.0, 1.0, 230.0, 3.0, 17.0, 3.0, 115.0...	[0.07828571712917061, 0.23065945244930058, 0.1...	4.0	4.0
2	[100.0, 3.0, 2.0, 140.0, 2.5, 10.5, 8.0, 140.0...	[0.21552956272311846, 0.5386297903210544, 0.11...	1.0	1.0
3	[100.0, 5.0, 2.0, 0.0, 2.700000047683716, -1.0...	[0.06588827838827839, 0.0529050949050949, 0.14...	3.0	3.0
4	[110.0, 1.0, 1.0, 180.0, 0.0, 12.0, 13.0, 55.0...	[0.07094088322536504, 0.8100710648188786, 0.05...	1.0	1.0

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

evaluator = MulticlassClassificationEvaluator(labelCol="mfrindex", predictionCol="prediction", metricName="accuracy")

evaluator.evaluate(predictions)
```

```
1.0
```

Conclusion

Le résultat final du projet permet donc de prédire sur des nouvelles données de céréale leur fabricant potentiel, et de recommander une céréale basée sur votre céréale préférée, selon tous les critères de ressemblances possibles.

Ce projet peut aussi être facilement modifié pour prédire sur n'importe quel autre critère, comme les calories ou les protéines, en se basant sur les autres données existantes.

Webographie

https://towardsdatascience.com/how-to-build-a-simple-recommender-system-in-python-375093c3fb7d

https://spark.apache.org/docs/2.2.0/ml-classification-regression.html

https://spark.apache.org/docs/latest/ml-clustering.html

https://towardsdatascience.com/random-forest-and-its-implementation-71824ced454f