TP Spark.MLib en python

L. Benyoussef & S. Derrode

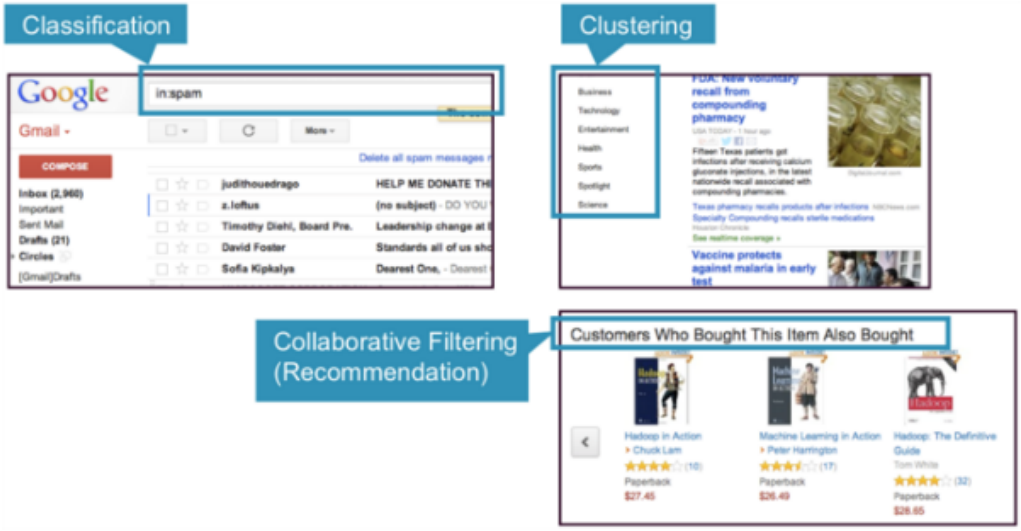
1. Mise a jour de votre depot github :

1. « *git add -A* » # Tous les fichiers que vous avez créés durant le 2ième TP sont ajoutés à *git*.
2. « *git commit -a -m "mon travail de TP Spark"* » # Enregistrement local de votre travail
3. « *git pull origin* » # récupération des nouveaux fichiers depuis le serveur github.

Si *git* vous informe de problème lié au *merge* de votre version avec celle du serveur, n'en tenez pas compte.

2. Introduction to Machine Learning

Trois catégories usuelles d'apprentissage artificiel sont : *Classification*, *Clustering* and *Collaborative Filtering*.



- ***Classification*:** Gmail uses a machine learning technique called classification to designate if an email is spam or not, based on the data of an email: the sender, recipients, subject, and message body. Classification takes a set of data with known labels and learns how to label new records based on that information.
- **Clustering:** Google News uses a technique called clustering to group news articles into different categories, based on title and content. Clustering algorithms discover groupings that occur in collections of data.
- ***Collaborative Filtering*:** Amazon uses a machine learning technique called collaborative filtering (commonly referred to as recommendation), to determine which products users will like based on their history and similarity to other users.

3. Prise en main de Spark.MLib

Spark.MLib[1] est une librairie de *machine learning* pour Spark qui contient tous les algorithmes et utilitaires d'apprentissage classiques, comme la classification, la régression, le *clustering*, le filtrage collaboratif, la réduction de dimensions, en plus des primitives d'optimisation sous-jacentes. L'utilisation de tables de données résilientes (RDD) au cœur de l'efficacité calculatoire de *Spark* rend *MLlib* dépendante de structures de données très contraignantes.

Vector

Création de vecteurs denses contenant les valeurs nulles :

```
from pyspark.mllib.linalg import Vectors # vecteur "dense"
denseVec2=Vectors.dense([1.0,0.0,2.0,4.0,0.0])
```

Création de vecteurs creux, seules les valeurs non nulles sont identifiées et stockées. Il faut préciser la taille du vecteur et les coordonnées de ces valeurs non nulles. C'est défini par un dictionnaire ou par une liste d'indices et de valeurs :

```
sparseVec1 = Vectors.sparse(5, {0: 1.0, 2: 2.0, 3: 4.0})
sparseVec2 = Vectors.sparse(5, [0, 2, 3], [1.0,2.0, 4.0])
```

LabeledPoint

Ce type est spécifique aux algorithmes d'apprentissage et associe un "label", en fait un réel, et un vecteur. Ce "label" est

- soit la valeur de la variable Y quantitative à modéliser en régression,
- soit un code de classe : 0.0, 1.0... en classification supervisée ou discrimination.

Rating

Type de données (note d'un article par un client) spécifique aux systèmes de recommandation et donc à l'algorithme de factorisation (ALS : *Alternate Least Square*) disponible.

Model

La classe *Model* est le résultat d'un algorithme d'apprentissage qui dispose de la fonction *predict()* pour appliquer le modèle à une nouvelle observation ou à une table de données résiliente de nouvelles observations.

4. Utilisation rudimentaire de MLib

La préparation des données est une étape essentielle à la qualité des analyses et modélisations qui en découlent. Extraction, filtrage, échantillonnage, complétion des données manquantes, correction, détection d'anomalies ou atypiques, jointures, agrégations ou cumuls, transformations (recodage, discrétisation, réduction, "normalisation"...), sélection des variables ou *features*, recalages d'images de signaux... sont les principales procédures à mettre en œuvre et de façon itérative avec les étapes d'apprentissage visant les objectifs de l'étude. Par principe, la plupart de ces étapes se distribuent naturellement sur les nœuds d'un cluster en exécutant des fonctions *MapReduce*.
Utiliser si nécessaire la doc https://spark.apache.org/docs/latest/ml-guide.html pour trouver la signification des paramètres transmis aux transformations et aux actions.

Kmeans

L'algorithme de classification non supervisé le plus utilisé, car le plus facile à passer à l'échelle est *k-means*. Visualisez le fichier *kmeans_data.txt* dans le répertoire courant : il contient des lignes de valeurs (on peut considérer que chaque donnée est un point de coordonnée 3D). L'algorithme des *k-means* cherche à classer les 3 premières lignes dans la première classe et les 3 dernières dans la seconde classe.

[1] https://spark.apache.org/docs/latest/mllib-guide.html

Modifiez le fichier *MLib_Kmeans.py* pour pointer vers l'endroit où vous avez sauvegardé le fichier *kmeans_data.txt*. Par exemple : « file:///home/sderrode/tp-hadoop-python/.../kmeans_data.txt ». Lancez alors :

spark-submit --master=local[2] MLib_Kmeans.py

Interprétez le résultat affiché sur le Terminal à la vue du code.

Régression logistique

Un exemple trivial de fouille de textes est la détection de pourriels. Le fichier *spam.txt* contient un spam par ligne, le fichier *ham.txt* un message "normal" par ligne. Il s'agit ensuite de prévoir le statut "spam" ou non d'un message.

Observez le contenu du fichier *MLib_LogReg.py* :

- Tout d'abord, il y a vectorisation des messages en utilisant la classe HashingTF qui permet de construire des vecteurs creux identifiant la fréquence de chaque mot. Un objet de cette classe est déclaré pouvant contenir jusqu'à 10000 mots différents. Chaque message est ensuite découpé en mots et chaque mot associé à une variable. Ces actions sont traitées message par message, donc dans une étape *Map*.
- Ensuite, on crée une table de données de type *LabeledPoint* avec "1" pour désigner un Spam et "0" un message normal.
- Ensuite, on fusionne les deux tables en une seule qui devient résiliente car mise en "cache". C'est fort utile avant exécution d'un algorithme itératif.
- Enfin, on estime la régression logistique puis on prédit la classe de deux messages.

Pour vérifier le fonctionnement, commencez par modifier le chemin d'accès aux fichiers *spam.txt* (exemples de spams) et *ham.txt* (exemples de courriels normaux) dans le fichier *MLib_LogReg.py*. Puis, lancez la commande :

spark-submit --master=local[2] MLib_LogReg.py

Remarque : un traitement préalable supprimant les mots charnières (articles, conjonctions...) et résumant chaque mot à sa racine aurait été nécessaire pour une application opérationnelle.

Forêt aléatoire

Système de classification qui passe l'échelle : les forêts aléatoires[2].

spark-submit --master=local[2] MLib_ForetA.py

Analysez le code. Notez que les données utilisées ne sont pas extraites d'un fichier mais directement codées dans le fichier source.

5. ORGANISATION DES MOYENS DE LUTTE CONTRE LE FEU DE FORET

L'objectif de ce TP est d'analyser des données pour permettre d'organiser au mieux les moyens humains et matériels pour réagir le plus rapidement en cas de départ de feu en forêt[3]. Pour cela, on cherche à regrouper les feux de forêt ayant eu lieu les années précédentes en cluster géographiques afin de disposer de moyens de lutte aux endroits les mieux adaptés.

Les données de départ de feu proviennent des services forestiers américains (*USDA forest service*)[4].

1- Depuis le répertoire « tp-hadoop-pyton/MLib/Data », lancez le script *bash* de récupération des fichiers *shapefile* (GIS – System d'information Géographique) sur le serveur de données.

>> *./getshapefiles.sh # cela peut prendre 30 secondes*

Le téléchargement réalisé correspond à 4 années de donnée ; d'autres données sont disponibles sur le même serveur pour des années antérieures.

[2] https://fr.wikipedia.org/wiki/Forêt_d%27arbres_décisionnels

[3] Exemple extrait et adapté de https://mapr.com/blog/predicting-forest-fires-with-spark-machine-learning/.

[4] https://fsapps.nwcg.gov/afm/gisdata.php.

2- Depuis le répertoire « tp-hadoop-python/Data », lancez ensuite le script python de transformation des fichiers *.dbf* en fichiers *.csv* (avec *python2.x* obligatoirement) :
   >> *python2 convert_shp_to_csv.py*

Votre travail consiste à entrainer un classifieur k-means en Spark-python avec les données de longitude et de latitude pour

- D'une part, obtenir le centre des clusters de départ de feu, permettant au pompier de disposer des moyens humains et matériels pour réagir au plus vite.
- D'autre part, informer les pompiers des moyens à utiliser en cas d'un nouveau départ de feu (dont on connaît la longitude et la latitude), par prédiction du cluster le plus proche.

Bien sûr, les réponses que vous apporterez à ces questions sont nécessairement simplistes, car bien d'autres facteurs entrent en considération dans la vraie vie (notamment la morphologie du terrain) !

Avant de poursuivre, quelques remarques :

- Commencez à travailler avec un seul fichier de données, puis étendez à tous les fichiers lorsque vous avez validé votre algo.
- Attention à la conversion des chaines de caractèresen nombre → float()
- Pour le kmeans, utilisez les fonctions décrites dans l'API : https://spark.apache.org/docs/2.2.0/mllib-clustering.html.
- Le fichier *train_Kmeans.py* contient la lecture d'un fichier et l'affichage des coordonnées des 10 premiers feux. Vous pouvez le tester en lançant :
  >> *spark-submit --master local[2] train_Kmeans.py*
  Changez le chemin d'accès au fichier de données pour l'adapter à votre compte.
  N'oubliez pas de supprimer la première ligne qui contient les entêtes des colonnes.
- Ne conservez que les données dont la latitude est comprise entre 42 et 50, et dont la longitude est comprise entre -124 et -110
- Pour l'apprentissage du classifieur, prenez un échantillon de 50% des données lues avec la méthode *randomSplit(…)*, l'autre sera utilisé pour la prédiction.
  https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.randomSplit