Introduction à l'utilisation de MLlib de Spark avec l'API pyspark

Résumé

L'objectif de ce tutoriel est d'introduire les objets de la technologie Spark et leur utilisation à l'aide de commandes en Python, plus précisément en utilisant l'API `pyspark`*, puis d'exécuter des algorithmes d'apprentissage avec la librairie* `MLlib`*. Motivations à l'utilisation de cet environnement, description des principaux types de données et du concept de* Resilient Distributed Datasets *(RDD) ; exécution d'algorithmes d'apprentissage sur des données volumineuses.*

- *Python pour Calcul Scientifique – Statistique*
- *Trafic de Données avec Python.*`Pandas`
- *Apprentissage Statistique avec Python.*`Scikit-learn`
- *Sciences des données avec Spark.mllib*

1 Pourquoi Spark

Les données sont, même après préparation, encore trop volumineuses pour la mémoire (RAM, disque) et/ou les temps de calcul rédhibitoires. Plus précisément, la distribution des données avec une technologies *Hadoop* est déjà en place ou devient incontournable. La description d'Hadoop n'est pas détaillée, non plus que les principes contraignants des fonctionnalités *MapReduce* associées. Le tutoriel consacré à RHadoop en est une (très) brève introduction.

Rappelons le contexte :

- Les méthodes d'apprentissage statistique supervisée ou non déploient des algorithmes itératifs dont l'exécution sur des données gérées dans un cadre Hadoop provoquent des lectures / écritures à chaque itération. Les temps d'exécution s'en trouvent fortement pénalisés.
- La technologie Spark y remédie en intégrant le concept de *tables de données distribuées résilientes* (*Resilient Distributed Dataset* ou RDD de Zaharia et al ; 2012)[3]). Très schématiquement, chaque partition de données reste en mémoire sur son serveur de calcul entre deux itérations tout en gérant les principes de tolérance aux pannes.
- Les commandes spécifiques de Spark s'exécutent en Java, Scala et aussi pour certaines en Python. D'où l'intérêt de l'apprentissage de Python qui permet sans "trop" d'effort de franchir le changement d'échelle en volume, notamment par l'emploi de la librairie ou plutôt de l'interface de programmation (*Application Interface programmation* ou API) dédiée `PySpark` ; cette API propose une utilisation interactive et même celle des calepins d'IPython.
- Spark intègre deux principales librairies :
  - SQL pour du requêtage dans des données volumineuses et structurées,
  - `MLlib` avec les principaux algorithmes d'apprentissage et méthodes statistique.

  Deux autres librairies sont disponibles pour traiter des données en flux continu (*streaming*) ou celles de graphes et réseaux. Elles ne sont actuellement (version 1.4 de Spark) accessibles que par les langages Scala ou Java.

La principale motivation pour utiliser la pile Spark est que les mêmes programmes ou commandes sont utilisées pour exécuter des algorithmes d'apprentissage (librairie MLlib), que ce soit sur un poste isolé, sur un cluster, un ensemble de machines virtuelles sur un serveur dont Amazon, Google cloud..., sur des données stockées dans un fichier ou distribuées dans un système Hadoop.

Attention : Une solution qui consisterait à utiliser un plus gros ordinateur avec beaucoup de mémoire vive est souvent préférable en terme de temps de mise au point, d'exécution, à celle d'installer Hadoop pour un usage ciblé. L'utilisation de spark / Hadoop s'impose lorsque les données sont déjà dans cet environnement mais très souvent, l'extraction, l'échantillonnage, la préparation des données, en vue d'un problème spécifique suffit à se ramener à un volume compatible avec la mémoire d'un ordinateur.

Le principal *objectif* est d'introduire ces outils pour l'usage et les objectifs d'un statisticien ou maintenant *data scientist*. Ce tutoriel se focalise donc sur l'utilisation de la librairie MLlib.

Il faut voir l'utilisation présente de MLlib comme une approche expérimentale afin d'en évaluer les performances au cœur de la problématique soulevée

par l'analyse de données massives : minimiser le terme d'erreur additionnel d'*optimisation* entre d'une part l'erreur par exemple due à l'échantillonnage et d'autre part l'erreur due à l'utilisation sous-optimale d'un algorithme souvent frustre de MLlib.

2 Présentation de Spark

2.1 Documentation

Les procédures d'installation, les concepts et les modes d'utilisation de Spark sont explicités dans la documentation en ligne et dans le livre de Karau et al. (2015)[1].

La librairie `MLlib` et son usage sont décrits par Pentreath (2015)[2] et dans la documentation en ligne pour la dernière version à jour.

D'autres ressources sont à consulter notamment pour :

- développer des compétences sur les bases de données distribuées avec un cours du CNAM,
- utiliser Spark avec la langage Scala plutôt que Python dans un autre cours du CNAM.

2.2 Installation

L'installation de Spark nécessite, entre autres, celle du gestionnaire de projet Maven et d'une version récente de Java. Cette opération qui est relativement complexe, surtout sous Windows, n'est pas détaillée ici mais consultable dans les documentations citées. Nous considérons que cela ne fait pas partie des compétences incontournables d'un statisticien qui se focalise sur les méthodes d'analyse / valorisation des données. Spark est donc supposé installé, accessible par l'intermédiaire de l'API `pyspark` et son interprète de commande ou encore par le biais d'un calepin IPython moyennant la bonne configuration de variables d'environnement.

2.3 Utilisation

Spark propose quatre types ou modes d'exécution :

Standalone local s'exécute dans un processus de machine virtuelle java sur un poste. C'est le mode utilisé pour tester un programme sur un petit ensemble de données et sur un poste de travail.

Standalone cluster est le cadre pour gérer en interne l'ordonnancement des tâches sur un cluster.

MESOS pour utiliser un cluster gérer par ces ressources du projet Apache.

YARN même chose pour un cluster utilisant la nouvelle génération des fonctionnalités MapReduce de Hadoop.

Spark exécute un programme en Java, Spark ou Python comme un langage interprété, à l'aide d'une console interactive, par le biais d'un notebook ou calepin IPython ou encore dans les environnements (nuages) loués comme par exemple à Amazon's EC2 (*Elastic Cloud Compute*).

2.4 Configuration

L'environnement utilisé est décrit par la commande `SparkContext` initialisant un objet `SparkConf` qui définit la configuration utilisée comme par exemple l'URL du nœud "maître" (*driver*) du cluster de calcul utilisé, le nombre de nœuds "esclaves" ou *workers*, leur espace mémoire réservé à chacun dans le cas de machines virtuelles.

En utilisation sur un poste seul, cet environnement est prédéfini à l'installation de Spark.

2.5 *Resilient Distributred Datasets (RDD)*

La notion de *table de données résiliente* est au cœur des fonctionnalités de Spark. Il s'agit d'un ensemble d'enregistrement ou objets d'un type spécifique, partitionnés ou distribués sur plusieurs nœuds du cluster. Cet objet est *tolérant aux pannes*, si un nœud est touché par une défaillance matérielle ou de réseau, la table résiliente est reconstruite automatiquement sur les autres nœuds et la tâche achevée.

Les opérations sur les RDD se déclinent "normalement" pour des données distribuées en étapes *Map* et *Reduce*.

La principale propriété des RDD de Spark est la possibilité de les *cacher* en mémoire (RAM) de chaque nœud. C'est ce qui permet d'économiser énormément d'accès disques qui sont le principal verrou, en terme de temps de calcul, lors de l'exécution d'algorithmes itératifs.

Une autre spécificité de Spark est la notion de variable *broadcast*. Ces va-

riables, en *lecture seule* et définies à partir du nœud maitre, sont connues et gardées en mémoire de tous les autres nœuds.

Un *accumulateur* est un cas particulier de variable *broadcast*. Un accumulateur n'est pas en "lecture seule", il peut être incrémenté ou plutôt chaque version locale peut être incrémenté par chaque nœud tandis que le nœud maître a accès au cumul global.

2.6 Programme Spark en Python

Voici un exemple rudimentaire de programme utilisant l'API `pyspark` donc en Python pour exécuter du "MapReduce" sur une installation Spark.

Créer un ficher texte avec le contenu suivant :

```
Albert,chocolat,3.27
Albert,cassoulet,2.45
Julie,coca,3.20
Dominique,tarte,1.50
Paul,cassoulet,5.40
```

Le sauver sous le nom de fichier `HistorCommande.csv`.

Lancer, selon votre environnement et dans le même répertoire, l'interprète de commande `pyspark`. Le contexte (`master=local[*]`) étant en principe prédéfini dans cet environnement, ne pas le redéfinir.

Exécuter dans l'environnement actuel (serveur d'enseignement) de l'INSAT :

```
pyspark.sh
```

Puis exécuter les commandes suivantes.

```
# lecture et distribution du fichier
data = sc.textFile("HistorCommande.csv").map(lambda
   line: line.split(",")).map(lambda record:
   (record[0], record[1], record[2]))
# nombre total de commandes
NbCommande=data.count()
print("Nb de commandes: %d" % NbCommande)
# Nombre de clients uniques
ClientUnique = data.map(lambda record:
   record[0]).distinct().count()
print("Nb clients: %d" % ClientUnique)
# Total des commandes
TotalCom = data.map(lambda record:
   float(record[2])).sum()
print("Total des commandes: %2.2f" % TotalCom)
# Produit le plus commandé
produits = data.map(lambda record:
   (record[1], 1.0)).reduceByKey(
   lambda a, b: a + b).collect()
plusFreq = sorted(produits, key=lambda x: x[1],
   reverse=True)[0]
print "Produit le plus populaire: %s avec
   %d commandes" %  (plusFreq[0],plusFreq[1])
```

Nous ne nous arrêterons pas sur la syntaxe spécifique d'exécution des commandes MapReduce en Python pour nous focaliser sur l'utilisation de la librairie `MLlib`.

Pentreath (2015)[2] explicite sur un exemple le lancement d'un cluster Spark dans l'environnement EC2 d'Amazon.

3 présentation de MLlib

3.1 Fonctionnalités

Cette librairie est en plein développement ; seule la documentation en ligne de la dernière version est à jour concernant la liste des méthodes disponibles. Voici un rapide aperçu de MLlib correspondant à la version 1.4.0 de Spark.

Statistique de base : Univariée, corrélation, échantillonnage stratifié, tests d'hypothèse, générateurs aléatoires, transformation (standardisation, quantification de textes avec TF-IDF et vectorisation), sélection (chi2) de variables (*features*).

Exploration multidimensionnelle Classification non-supervisée (k-means avec version en ligne, modèles de mélanges gaussiens, LDA ou *Latent*

Derichlet Allocation, réduction de dimension (SVD et ACP mais en java ou scala pas en python), factorisation non négative (NMF) par moindres carrés alternés (ALS).

Apprentissage Méthodes linéaires : SVM, régression gaussienne et binomiale ou logistique avec pénalisation L1 ou L2 ; estimation par gradient stochastique, ou L-BFGS ; classifieur bayésien naïf, arbre de décision, forêts aléatoires, boosting (*gradient boosting machine*).

3.2 Types de données

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. Une fonction `map()` est utilisée pour crée les objets de ces types.

Vector

Des vecteurs numériques sont distribuables sur les nœuds sous deux formats : dense ou creux. dans le dernier cas, seules les coordonnées non nulles sont enregistrées.

Créations de vecteurs denses contenant les valeurs nulles :

```
from numpy import array
from pyspark.mllib.linalg import Vectors
# vecteur "dense"
# à partir de numpy
denseVec1=array([1.0,0.0,2.0,4.0,0.0])
# en utilisant la classe Vectors
denseVec2=Vectors.dense([1.0,0.0,2.0,4.0,0.0])
```

Créations 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) 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.

3.3 Préparation des données

Comme cela est déjà largement expliqué dans le tutoriel consacré au trafic (*munging*) des données, leur préparation 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 œuve 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 comme dans l'exemple initial, en utilisant les fonctions spécifiques des librairies `SparKSQL` et `MLlib`.

Cependant, ces traitements étant exclusifs et spécifiques à un jeu de données, ils sont développés dans les ateliers.

4 Utilisations rudimentaire de MLlib

Les exemples proposés le sont sur des jeux de données "jouet" de la documentation en ligne afin de tester MLlib et introduire la syntaxe des programmes écrits en python. Les ateliers développent des utilisations sur des données plus réalistes.

4.1 k-means

L'algorithme de classification non supervisé le plus utilisé, car le plus facile à passer à l'échelle "volume", est k-means ou plutôt la version dite de Forgy d'un algorithme de réallocation dynamique.

Générer le fichier ci-dessous dans le répertoire courant :

```
0.0 0.0 0.0
0.1 0.1 0.1
0.2 0.2 0.2
9.0 9.0 9.0
9.1 9.1 9.1
9.2 9.2 9.2
```

Lancer pyspark dans un calepin Ipython à partir du même répertoire. Sur les serveurs d'enseignements de l'INSAT :

```
pyspark_notebook.sh
```

Entrer ensuite les lignes ci-dessous dans des cellules du calepin en privilégiant une sortie par cellule.

```
# Déclaration des fonctions
from pyspark.mllib.clustering import KMeans
from numpy import array
from math import sqrt
# Lire et "distribuer" les données
data = sc.textFile("kmeans_data.txt")
parsedData = data.map(lambda line:
   array([float(x) for x in line.split(' ')]))
parsedData.collect()
```

```
# Recherche des 2 classes
clusters = KMeans.train(parsedData, 2,
   maxIterations=10, runs=10,
   initializationMode="random")
# Qualité de la classification
def error(point):
    center = clusters.centers[
       clusters.predict(point)]
    return sum([x**2 for x in (point - center)])
Inert = parsedData.map(lambda point:
   error(point)).reduce(lambda x, y: x + y)
varIntra=Inert/parsedData.count()
print("Variance intraclasse = " + str(varIntra))
```

La fonction de "prévision" est simplement celle d'affectation d'une observation à une classe.

```
clusters.predict([ 9.,  9.,  9.])
clusters.predict([ 0.1,  0.1,  0.1])
```

Cette fonction permet également d'afficher les classes des points.

```
# fonction lambda dans map pour "prédire"
# tous les vecteurs
parsedData.map(lambda point:
   clusters.predict(point)).collect()
```

4.2 Régression logistique

Un exemple trivial (Karau et al. 2015)[1] de fouille de textes pour la détection de pourriels . Le fichier `spam.txt` contient un spam par ligne, celui `ham.txt` un message "normal" par ligne. Il s'agit ensuite de prévoir le statut "spam" ou non d'un message.

Importation des fonctions utilisées et lecture / distribution des fichiers. Ceux-ci sont disponible dans le répertoire wikistat.fr/data.

```
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.feature import HashingTF
from pyspark.mllib.classification
```

```
   import LogisticRegressionWithSGD
spam = sc.textFile("spam.txt")
normal = sc.textFile("ham.txt")
```

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

Vectorisation des messages en utilisant la classe `HashingTF` qui permet de construire des vecteurs creux identifiant la fréquence de chaque mot. Un object 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.

```
tf = HashingTF(numFeatures = 10000)
spamFeatures = spam.map(lambda email:
   tf.transform(email.split(" ")))
normalFeatures = normal.map(lambda email:
   tf.transform(email.split(" ")))
```

Création de table de données de type `LabeledPoint` avec "1" pour désigner un Spam et "0" un message normal.

```
posExamples = spamFeatures.map(lambda features:
   LabeledPoint(1, features))
negExamples = normalFeatures.map(lambda features:
   LabeledPoint(0, features))
```

Fusion des deux tables en une seule qui devient résiliente car mise en "cache". C'est fort utile avant exécution d'un algorithme itératif.

```
trainingData = posExamples.union(negExamples)
trainingData.cache()
```

Estimation de la régression logistique puis prévision de la classe de deux messages.

```
model=LogisticRegressionWithSGD.train(trainingData)
posTest = tf.transform("O M G GET cheap stuff
   by sending money to ...".split(" "))
negTest = tf.transform("Hi Dad, I started studying
   Spark the other ...".split(" "))
```

Résultats.

```
print "Prediction for positive test example:
   %g" % model.predict(posTest)
print "Prediction for negative test example:
   %g" % model.predict(negTest)
```

4.3 Arbre de discrimination

MLlib propose la construction d'arbres de régression ou classification mais comme dans `scikit-learn`, la fonction n'inclut pas de paramètre de pénalisation et l'élagage ne se prête pas à une bonne optimisation.

```
# Importation des fonctions
from numpy import array
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.tree import DecisionTree
# Construction de l'échantillon d'apprentissage
# Format de vecteur avec étiquette
data = [
    LabeledPoint(0.0, [0.0]),
    LabeledPoint(1.0, [1.0]),
    LabeledPoint(1.0, [2.0]),
    LabeledPoint(1.0, [3.0])]
sc.parallelize(data).collect()
# estimation de l'arbre
model = DecisionTree.trainClassifier(
   sc.parallelize(data), 2, {})
# "affichage" de l'arbre
print(model)
print model.toDebugString()
# Prévision d'une observation de R2
model.predict(array([1.0]))
```

```
# Prévision de toute une table
rdd = sc.parallelize([[1.0], [0.0]])
model.predict(rdd).collect()
```

4.4 Forêt aléatoire

L'algorithme des forêts aléatoires devenu le "couteau suisse" de l'apprentissage est implémenté dans toutes les librairies. Deux paramètres importants apparaissent avec les données massives : `maxBins` (32 par défaut)ou nombre maximale de modalités d'une variables qualitatives ou de classes des variables quantitatives qui sont systématiquement discrétisées et `subsamplingRate` ou taux d'échantillonnage.

Ces deux paramètres règlent le compromis entre temps de calcul et qualité de l'optimisation. L'autre paramètre important est le nombre `featureSubsetStrategy` (`mtry en R`) de variables candidates à la construction d'un nœud et tirées aléatoirement.

Les résultats sont nettement moins moins détaillés qu'avec R ou même `scikit-learn`, avec notamment les absences du calcul de l'erreur *out of bag* et de celles des importances des variables.

```
# iportations des fonctions
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.tree import RandomForest
# construction des données
data = [
     LabeledPoint(0.0, [0.0]),
     LabeledPoint(0.0, [1.0]),
     LabeledPoint(1.0, [2.0]),
     LabeledPoint(1.0, [3.0])]
# distribution de la table
trainingData=sc.parallelize(data)
trainingData.collect()
# Estimation du modèle
model = RandomForest.trainClassifier(
   trainingData, 2, {}, 3, seed=42)
model.numTrees()
model.totalNumNodes()
# "Affichage" de la forêt
print model.toDebugString()
# Prévision d'u échantillon
rdd = sc.parallelize([[3.0], [1.0]])
model.predict(rdd).collect()
```

5 Détection de Spams

p219 The code and data files are available in the book ?s Git repository

Références

[1] H. Karau, A. Konwinski, P. Wendell et M. Zaharia, *Learning Spark*, O'reilly, 2015.

[2] N. Pentreath, *Machine Learning with Spark*, Packt puplishing, 2015.

[3] M. Zaharia, M. Chowdhury, D. Das, A. Dave, J. Ma, M. McCauly, S. Franklin, M. J.and Shenker et I. Stoica, *Resilient Distributed Datasets : A Fault-Tolerant Abstraction for In-Memory Cluster Computing*, Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), USENIX, 2012, p. 15–28.