UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE

M2 TIDE – 2024-2025

Big Data Analytics

Rapport de Projet – Big Data Analytics

SAIDI Syrine, BOUZMANE Salma, BEAULIRE Stevensia

1. Exercice 1 : Analyse et Implémentation du Calcul des Amis en Commun via MapReduce et PySpark :

Partie 1:

L'objectif était d'identifier les amis en commun entre utilisateurs d'un réseau social. Nous avons choisi de nous inspirer du principe MapReduce, étudié en fin de chapitre 2.Cette version appelle aux étapes de MapReduce de manière séquentielle, sans traitement distribué. La démarche est expliquée comme suit:

1.1 Étape 1 : Lecture et pré-traitement des données

Nous possédons Le fichier amis.txt. Chaque ligne de ce fichier est composée d'un identifiant utilisateur suivi de la liste de ses amis, séparée par des virgules.

Nous avons défini une fonction **lire_fichier()** qui lit les données, filtre les lignes valides (*celles qui contiennent :*), puis les convertit en tuples de la forme (utilisateur, set(amis)). Ce format facilite les traitements ultérieurs.

1.2 Étape 2 : Phase MAP

Dans cette Phase de Map, nous avons essayé de recourir à la génération des couples sous forme(clé, valeur), comme c'est recommandé dans le cours.

À partir de chaque utilisateur et de ses amis, la fonction **mapper** génère des paires triées d'utilisateurs associées à la liste des autres amis. Cela permet ensuite de croiser les listes d'amis pour trouver celles en commun.

1.3 Étape 3 : Groupement par clé

A cette étape, nous avons été inspirés par la fonction fournie dans le chapitre 2 dans notre cours, celle de **groupByKey()** qui permet de regrouper toutes les valeurs associées à une même clé. Dans notre cas,

Les couples **(ami1, ami2)** sont regroupés grâce à la fonction **groupByKey()** qui regroupe les valeurs (listes d'amis) par paire d'utilisateurs.

1.4 Étape 4 : Réduction — intersection des listes

De même, dans cette phase Reduce, l'étape finale du modèle MapReduce, nous souhaitons combiner les valeurs ayant la même clé pour produire un résultat agrégé.

Nous avons été inspirés par la fonction fournie dans le chapitre 2 dans notre cours, celle de **reduceByKey(),**qui simule ce comportement en regroupant les paires

utilisateur–ami (préalablement structurées avec groupByKey) et en leur appliquant une fonction de réduction, ici appelée intersection_reducer.

La fonction **intersection_reducer** prend en entrée deux listes représentant les autres amis d'un utilisateur dans deux contextes différents (par exemple, les amis d'un 10 vus depuis 20, puis les amis de 20 vus depuis 10). Elle calcule leur intersection, c'est-à-dire les amis communs aux deux listes, en convertissant les deux listes en ensembles (set) puis en appliquant l'opérateur d'intersection.

Cette opération est répétée pour chaque paire d'utilisateurs : ainsi, le programme identifie, pour chaque couple d'amis, la liste des amis qu'ils ont en commun. Le résultat est une liste de tuples où chaque entrée indique une paire d'utilisateurs ainsi que la liste et le nombre de leurs amis en commun.

1.5 Étape 5 : Résultats finaux

Les résultats finaux affichent, pour chaque paire d'utilisateurs, le **nombre d'amis en commun** et la **liste des amis** concernés avec un affichage très clair, montrant tous les outputs aussi issus des fonctions de map et de reduce.

Partie 2 : Implémentation avec PySpark

Dans cette seconde partie, nous avons utilisé **Apache Spark** pour paralléliser et distribuer le traitement du calcul d'amis en commun entre utilisateurs d'un réseau social. Il a été demandé d'utiliser **PySpark.** Le but-nous pensons- c'est de bénéficier de la puissance de calcul distribuée pour traiter de grands volumes de données efficacement. Cette implémentation suit le paradigme MapReduce, adaptée aux RDD de Spark.

Nous avons ainsi procédé à ces étapes suivantes:

2.1 Étape 1 : Création d'une session Spark

Avant tout traitement, nous avons créé une **SparkSession**, point d'entrée pour l'application Spark. Elle instancie un **SparkContext**, nécessaire pour manipuler les **RDD** (Resilient Distributed Datasets).

2.2 Étape 2 : Lecture et pré-traitement des données

Cette étape est similaire à l'étape de lecture des données et de pré-traitement de la première partie. Donc, nous avons procédé au même traitement de fichier.

2.3 Étape 3 : Phase MAP

Nous avons recouru dans cette phase à la fonction appelée **flatMap**, qui permet de générer plusieurs paires clé-valeur à partir d'un seul enregistrement.

Cette propriété est particulièrement utile ici, car un utilisateur peut avoir plusieurs amis, et pour chaque ami, on veut générer une paire représentant une relation d'amitié (utilisateur, ami), accompagnée de la liste des autres amis.

Concrètement, pour chaque utilisateur, la fonction **mapper** retourne une liste de couples **(pair, autres_amis)**. L'utilisation de **flatMap** permet alors de "déplier" ces listes directement dans le RDD résultant, sans créer de structures imbriquées.

Cela simplifie la suite du traitement, notamment le **groupByKey**, car toutes les paires (ami1, ami2) sont directement accessibles comme des éléments simples du RDD. Cette opération est donc à la fois pratique et efficace dans le cadre du traitement distribué typique de Spark.

2.4 Étape 4 : Reduce

Les couples générés sont ensuite regroupés par clé avec **groupByKey()**, propre à l'API Spark.

Une fois les données regroupées, l'étape de réduction est assurée par **reduceByKey(intersection_reducer).**
Cette fonction applique la logique d'intersection sur les listes d'amis associées à chaque couple d'utilisateurs, grâce à la fonction définie **intersection_reducer()** (expliqué précédemment le but de cette fonction dans la partie 1)

L'utilisation de reduceByKey, contrairement à groupByKey, optimise les performances en réduisant les données en cours de shuffle, ce qui est plus efficace en termes de mémoire et de bande passante.

Grâce à la symétrie des couples (les paires sont triées), les relations (10, 20) et (20, 10) sont considérées équivalentes, évitant ainsi les doublons et assurant un calcul correct du nombre d'amis communs entre chaque paire unique d'utilisateurs.

2.5 Étape 5 : Affichage des résultats

Nous avons ensuite trié et affiché les résultats sous format lisible. Pour chaque paire d'utilisateurs, nous indiquons le nombre d'amis en commun ainsi que leur liste.Et nous avons obtenu le même résultat que celui de la partie 1.

2.6 Étape finale : Arrêt de la session Spark

Enfin, nous avons stoppé proprement la session Spark en appelant **spark.stop()** pour libérer les ressources système.

Résultat de l'Ex 1:

N.B: dans le code, nous avons affiché les résultat après les phases reduce et map pour mettre en évidence l'output des fonctions recourues.

```
Résultats finaux (amis en commun) :
('10', '20'): 2 ami(s) en commun → ['30', '40']
('10', '30'): 2 ami(s) en commun → ['20', '40']
('10', '40'): 2 ami(s) en commun → ['30', '20']
('20', '30'): 2 ami(s) en commun → ['10', '40']
('20', '40'): 2 ami(s) en commun → ['10', '30']
('30', '40'): 3 ami(s) en commun → ['50', '10', '20']
('30', '50'): 1 ami(s) en commun → ['40']
('40', '50'): 1 ami(s) en commun → ['30']
```

2. Exercice 2 : Extraction d'informations depuis des pages web via une approche MapReduce :

L'exercice 2 avait pour objectif d'extraire les adresses e-mail et les mots les plus fréquents depuis un ensemble de sites web, en appliquant une logique inspirée du paradigme MapReduce. À l'origine, l'idée était d'utiliser Apache Spark pour profiter de son efficacité dans le traitement distribué de données. Cependant, en raison de problèmes d'installation de Spark sur mon environnement Windows, j'ai opté pour une solution plus simple et accessible, basée sur Jupyter Notebook.

1.1 Étape 1 : Collecte des données

J'ai constitué un fichier texte (URLs.txt) regroupant une série de liens vers des pages web de restaurants, salons de coiffure et hôtels situés à Paris.

1.2 Étape 2 : Téléchargement et traitement du contenu HTML

Pour chaque URL, le script envoie une requête HTTP et télécharge le contenu HTML. Un en-tête de type *User-Agent* a été ajouté afin d'éviter que certaines pages ne bloquent la requête.

1.3 Étape 3 : Nettoyage du contenu

Le texte brut est extrait des pages en supprimant les balises HTML inutiles (scripts, styles) grâce à la bibliothèque BeautifulSoup. Cela permet de récupérer uniquement le contenu visible et lisible.

1.4 Étape 4 : Extraction des e-mails

Pour identifier les adresses e-mail présentes dans le contenu des pages web, le script utilise une expression régulière adaptée aux formats standards d'e-mails. Cette expression ([A-Za-z0-9\._\-+]+@[A-Za-z0-9\._\-+]+\.[a-zA-Z]{2,6}) permet de détecter automatiquement les adresses comportant des lettres, chiffres, points, tirets ou symboles usuels, suivis d'un signe @ et d'un nom de domaine valide. Cette détection

est effectuée sur le texte brut extrait des pages HTML, après nettoyage du contenu avec BeautifulSoup. Ainsi, toutes les adresses e-mail visibles dans la page sont collectées de manière automatique et fiable.

1.5 Étape 5 : Comptage des mots fréquents (style MapReduce)

J'ai reproduit une version simplifiée de MapReduce en Python :

> Phase Map : chaque mot est converti en paire (mot,1).
> Phase Shuffle/Group : les paires sont regroupées par mot.
> Phase Reduce : les occurrences de chaque mot sont additionnées.
> Ensuite, seuls les mots les plus fréquents sont conservés (top 10).

1.6 Étape 6 : Sauvegarde des résultats

Les résultats (emails et mots les plus fréquents) sont enregistrés dans un fichier texte (RESULTATS.txt) pour chaque site visité.

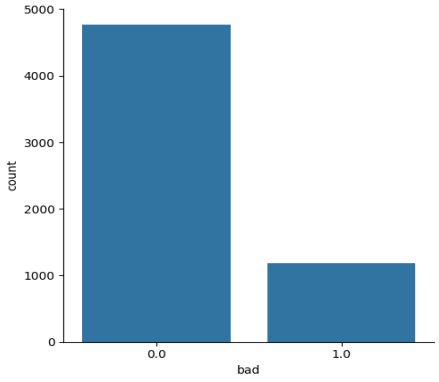
3. Exercice 3 : Modélisation d'un Score de Crédit Hypothécaire Interprétable à l'Aide de SageMaker :

L'objectif de cet exercice était de construire un modèle de score de crédit hypothécaire, à la fois performant et interprétable, afin d'automatiser la prise de décision pour l'octroi de crédit tout en respectant les principes de l'Equal Credit Opportunity Act. Le dataset HMEQ contient les données de 5 960 emprunteurs récents, avec une variable cible binaire BAD indiquant s'il y a eu défaut de remboursement.
Pour ce faire, trois modèles de prédiction ont été entraînés en local sur les données, et le meilleur d'entre eux a ensuite été déployé sur SageMaker pour être appliqué aux données qui ont été préalablement stockées dans Amazon S3.

1.1 Étape 1 : Analyse exploratoire et traitement des valeurs manquantes

Nous avons commencé par une exploration des données, identifiant les variables manquantes et les possibles valeurs aberrantes.
Cette analyse préliminaire nous a permis de détecter que les variables debtinc, derog, delinq, mortdue, value, entre autres, comportaient un taux non négligeable de valeurs manquantes et que la distribution de la variable cible était déséquilibrée (20 % de défaut).

Taux de valeurs manquantes	
debtinc	0,212548
derog	0,118772
delinq	0,097299
mortdue	0,086898
yoj	0,086395
ninq	0,085556
clage	0,051669
job	0,046804
reason	0,042275
clno	0,037242
value	0,018789



L'approche adoptée fut une conservation maximale des données, nous avons privilégié l'imputation à la suppression. Pour les variables numériques, présentant un nombre considérable d'outliers, nous avons opté pour une imputation par la médiane, méthode plus robuste face aux valeurs extrêmes.
Avant d'appliquer l'imputation, nous avons ajouté pour chaque variable concernée une variable indicatrice (ex : value_missing) signalant la présence ou non d'une valeur manquante. Ce choix permet au modèle de détecter les éventuels schémas d'absence d'information, comme lorsqu'un client omet volontairement de déclarer certaines données sensibles. Ce qui peut révéler des motifs importants dans les données.
Concernant les variables qualitatives (job, reason), nous avons appliqué un encodage one-hot, qui consiste à créer une colonne binaire par modalité. Les valeurs manquantes y ont été traitées comme une catégorie à part entière.

1.2. Étape 2 : Préparation des jeux d'entraînement et de test

Les données ont ensuite été divisées en X_train et X_test avec une stratification sur la variable BAD, On a ainsi procédé afin de conserver la proportion de défaillants (≈ 20 %) dans chaque sous-ensemble de test et d'entrainement.

1.3. Étape 3 : Application des algorithmes de prédictions en local sur les données

Trois modèles ont été testés en local, la régression logistique (modèle de référence interprétable), le forêt aléatoire (RandomForest), et XGBoost. Chaque modèle a été évalué à l'aide d'une validation croisée avec stratification (StratifiedKFold) pour garantir une évaluation robuste.
Nous avons procédé à la standardisation des variables numériques pour l'algorithmes de la régression logistique, et appliqué l'option class_weight='balanced' aux modèles compatibles (régression logistique et forêt aléatoire) pour gérer le déséquilibre des classes de la variable BAD.

Les performances ont été comparées à l'aide de métriques standard : AUC, F1-score, précision, rappel.

```
 Évaluation des modèles (CV sur X_train) :

                       AUC  Précision  Rappel  F1-score
Logistic Regression  0.9059     0.6053  0.7981    0.6884
Random Forest        0.9567     0.8381  0.6719    0.7458
XGBoost              0.9584     0.8675  0.7476    0.8031
```

- La régression logistique détecte beaucoup de défaillants (rappel : 0.7981), mais a une faible précision (0.6053) il alerte à tort la banque.
- Le Random Forest offre un excellent compromis entre précision et rappel (F1-score = 0.7458).
- Le modele XGBoost pour sa part donne un meilleur compromis que le Random Forest et la régression logistique (F1-score: 0.8031). Il distingue mieux les clients à risque, ceci en limitant les faux positifs qui font perdre du profit a la banque. C'est le meilleur modèle en termes de performance globale.

Le modèle XGBoost est donc retenu comme modèle final et sera déployé dans SageMaker pour prédiction sur l'ensemble de test en utilisant les hyperparamètres trouvés par la methode Grid Search.

1.4. Étape 4 : Entraînement et déploiement sur SageMaker

On a commencé par diviser le jeu de donnees X_train en train.csv et validation.csv, ces fichiers ont été envoyé vers Amazon S3 vers S3, après qu'une session SageMaker ait été initialisée dans Notebook sur vsCode.

Nous avons ensuite configuré l'environnement SageMaker dans AWS pour pouvoir accéder à S3, créer des endpoints etc. Pour cela on a identifié la région AWS dans laquelle nous travaillons sur AWS, défini le rôle IAM permettant en fait d'exécuter les opérations SageMaker lecture/écriture sur S3, déploiement de modèle, etc.

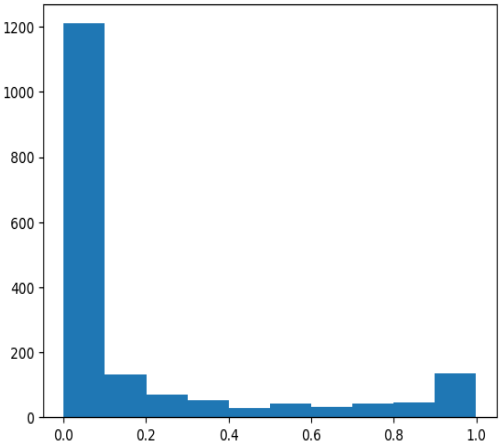
Après la configuration de l'environnement SageMaker, nous avons construit un chemin menant vers un dossier de sortie qu'on a nommé 'xgboost_model' sur Amazon S3 où le modèle entraîné sera stocké, une fois que l'entraînement SageMaker terminé.

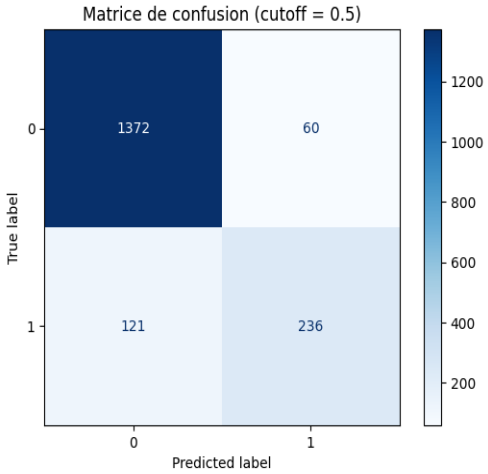
Ensuite, le modèle XGBoost a été entraîné sur une seule instance SageMaker à partir d'une image Docker préconfigurée (version 1.2-1). Cet entrainement s'est fait sur des donnees preparees au format requis par SageMaker à l'aide de la fonction TrainingInput(...). Une fois entraîné, il a été déployé sur un endpoint via une instance ml.t2.medium.

Afin de ne pas dépasser les limites de requêtes ou la capacité mémoire du modèle en ligne, les prédictions sur X_test ont été réalisées par paquets de 1 000 lignes qui ont été envoyé au modèle via le endpoint déployé, lequel a retourné une chaîne de probabilités au format texte CSV (par exemple : '0.85,0.34,...'), correspondant aux probabilités estimées pour la classe BAD = 1.

Les différentes chaînes ont été concaténées, nettoyées puis converties en un tableau NumPy contenant l'ensemble des scores de probabilité. Ces derniers ont été transformés en prédictions binaires (0 ou 1) à l'aide d'un seuil de décision (par exemple 0.5) avant d'être évalués à l'aide des métriques classiques (AUC, F1-score, précision, rappel, matrice de confusion).

1.5. Etape 5 : Interprétabilité et évaluation du modèle retenu





Évaluation du modele sur X_test :

	precision	recall	f1-score	support
0	0.92	0.96	0.94	1432
1	0.80	0.66	0.72	357
accuracy			0.90	1789
macro avg	0.86	0.81	0.83	1789
weighted avg	0.89	0.90	0.90	1789

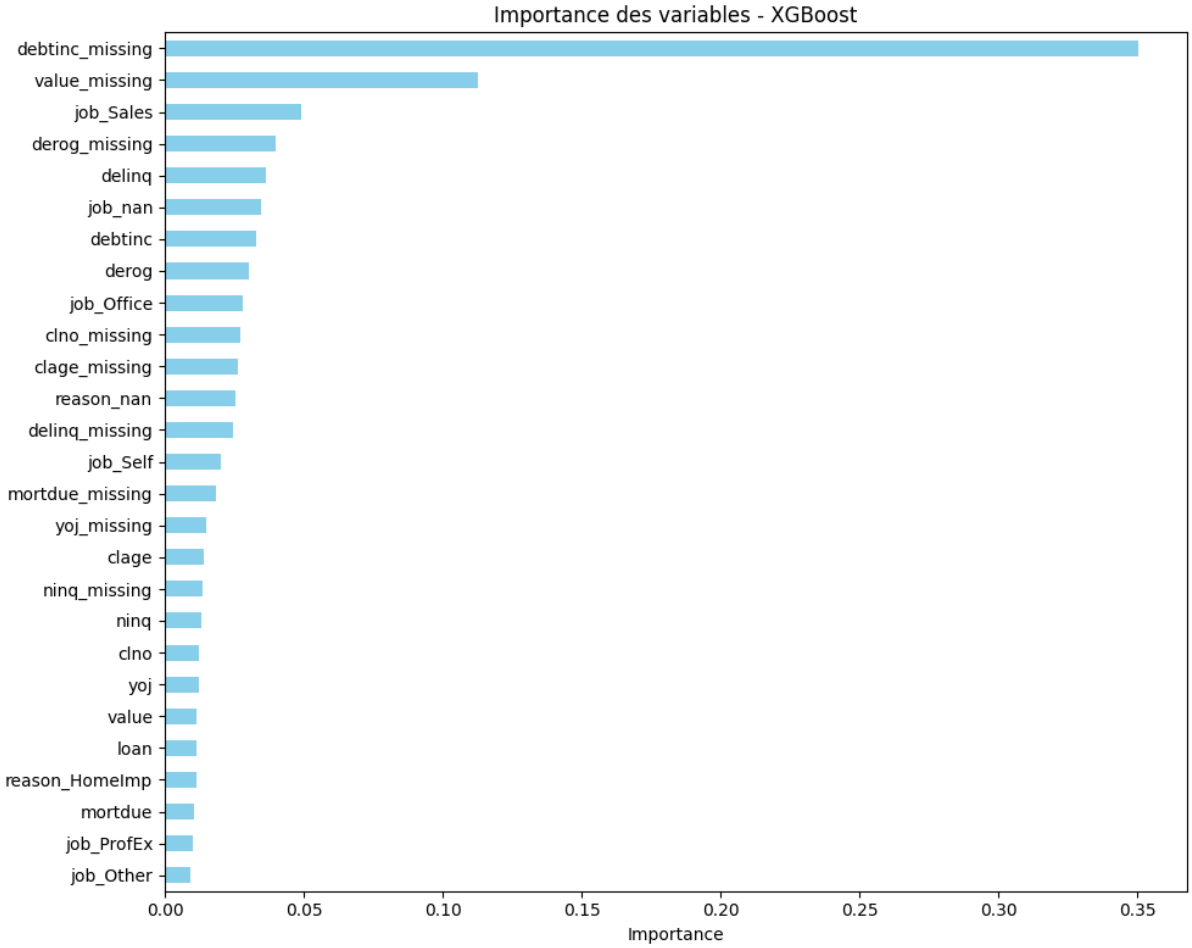
On voit qu'avec un taux de bonne classification global (accuracy) est 90 %, le modèle est très bon pour détecter les non-défaillants (classe 0), precision et recall 0.92 et 0.96

respectivement. 92 % des prédits "non-défaillants" sont corrects. 96 % des vrais "non-défaillants" sont bien détectés
Il est raisonnablement bon pour les défaillants, la précision est de 0.80, sur 100 clients prédits "défaillants", 80 le sont réellement, le rappel est de 0.66, le modèle détecte 66 % des vrais défaillants et donc 1/3 échappent à la détection.
Le modèle est plus prudent qu'agressif : il préfère rater des défauts que classer à tort un bon client comme mauvais.

L'importance des variables a été extraite via feature_importances_ de XGBoost afin de mieux comprendre les facteurs ayant guidé les décisions du modèle.

Les principales variables déterminantes sont les indicateurs de valeurs manquantes comme value_missing et debtinc_missing, le simple fait qu'une information soit absente révèlent un profil à risque, le nombre de litiges passés (delinq) qui est un indicateur direct d’un historique de crédit problématique, certaines professions (job_Sales, etc.), reflètent eux aussi des niveaux de stabilité financière variables.

Cette interprétation globale du modèle permet de justifier les décisions individuelles de manière transparente, tout en mettant en évidence les dimensions sensibles du profil client.


Importance des variables - XGBoost
debtinc_missing
value_missing
job_Sales
derog_missing
delinq
job_nan
debtinc
derog
job_Office
clno_missing
clage_missing
reason_nan
delinq_missing
job_Self
mortdue_missing
yoj_missing
clage
ninq_missing
ninq
clno
yoj
value
loan
reason_HomeImp
mortdue
job_ProfEx
job_Other
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
Importance