Exercice 1

On souhaite afficher pour les utilisateurs d'un réseau social le nombre d'amis en commun avec un autre utilisateur quand il visite la page de ce dernier.

1. Ecrivez sous python, un programme de type MapReduce qui calcule le nombre d'amis en communs pour chaque paire d'utilisateurs sachant que le fichier amis.txt contient les identifiants des utilisateurs suivis des identifiants de leurs amis.

Tout d'abord, nous avons commencé par importer notre fichier « amis.txt » dans un dictionnaire. Chaque ligne de notre dictionnaire se présentent sous la forme : {Id_utilisateur : {id_ami1, id_ami2, … }. Il est important de préciser que chaque id est au format « int » et que nous avons importé les id_ami en set() afin de faciliter les calculs qui auront lieu par la suite.

MapReduce est un modèle de programmation et un framework open-source pour le traitement distribué de données volumineuses. Il est utilisé pour traiter des ensembles de données trop volumineux pour être stockés et traités sur un seul ordinateur. Ce modèle s'articule en deux phases distinctes :

Étape Map :

Nous avons créé une fonction `map_cree(dico)`. Cette fonction a pour but de générer des paires d'utilisateurs où chaque paire représente deux utilisateurs amis. Pour chaque paire, la fonction associe la liste des amis de l'utilisateur initial. Il est important de préciser que chaque paire est triée, en mettant l'id le plus faible puis l'id le plus élevé, afin d'éviter de la redondance. Cette fonction va permettre d'obtenir une liste de tuples. La clé est la paire d'utilisateur (triée de manière à éviter la duplication des paires). La valeur est la liste des amis de l'utilisateur initial dans la paire.

Étape Reduce :

Ainsi, la fonction `map_cree(…)` sert d'étape préliminaire à l'étape de réduction. En effet, à présent il nous suffit de regrouper les paires identiques et compter le nombre d'amis en commun. Si plusieurs paires concernent les mêmes utilisateurs, les listes d'amis sont regroupées. Pour ce faire, Nous avons réutilisé deux fonctions vues en TP.
Pour commencer, nous avons repris la fonction `groupByKey(data)`. Cette fonction prend en entrée une liste de paires (clé, valeur) et regroupe les valeurs associées à chaque clé. Par exemple, si plusieurs utilisateurs ont les mêmes amis, leurs amis seront regroupés sous une même clé.
Ensuite, nous avons réutilisé la fonction `reduceByKey(f, data)`. Cette dernière prend en entrée (f) une fonction d'intersection. Dans notre cas nous avons créé une fonction d'intersection nommée `intersec(set1, set2)` qui calcule les identifiants communs aux deux sets. Ainsi, la fonction `reduceByKey` renvoie une liste où chaque paire sera associé à ses amis en communs.

Grace à cette méthodologie suivie, ainsi qu'à nos fonctions crées, nous avons pu obtenir le nombre d'amis en communs pour chaque paire d'utilisateurs du fichier amis.txt

Hurtado, Daddio, Sopguombue

2. Ré-écrivez votre programme MapReduce en utilisant Pyspark.

À présent, nous allons nous appuyer sur ce que nous avons fait dans la question une pour reproduire la même sortie mais en utilisant PySpark.
Pour ce faire, nous avons commencé par initialiser un Spark contexte « amis » en local. Nous avons ensuite transformé notre dictionnaire « dico » (contenant les informations du fichier amis.txt) en RDD afin de le rendre lisible par Spark. Une fois nos étapes préliminaires faites nous avons, comme précedement, agit en deux étapes :

Étape Map :
Cette étape ce base exactement sur le même principe que pour la question une : « générer des paires d'utilisateurs où chaque paire représente deux utilisateurs amis. Pour chaque paire, la fonction associe la liste des amis de l'utilisateur initial.». Pour ce faire nous avons créé la fonction `map_cree_spark(ami_entry)`. Cette fonction repose sur le même principe que `map_cree`, mais adaptée pour être utilisée avec PySpark.

Étape Reduce :
Comme précédemment, la fonction `map_cree_spark(ami_entry)` sert d'étape préliminaire à l'étape de réduction. Nous cherchons encore une fois à regrouper les paires identiques et compter le nombre d'amis en commun. Pour cela, nous allons créer une fonction `intersec_spark(…)`. Tout comme la fonction `intersec(set1, set2)`, cette fonction calcule l'intersection de sets. Cependant pour s'adapter au contexte de parallélisation avec PySpark, cette nouvelle fonction permet de traiter plusieurs sets plutôt que seulement deux. En effet, elle s'assure que l'intersection se fait progressivement entre deux sets à la fois, ce qui est nécessaire lorsque les données sont distribuées et traitées de manière parallèle.
Enfin, tout comme dans la première question, nous avons utilisé la fonction vue en TP `groupByKey()`. Nous avons groupé les éléments de dico_rdd par paires d'utilisateurs, ainsi que le listes d'amis associées à chaque paire d'utilisateurs. Après cela, on applique la fonction `intersec_spark()` à chaque liste de sets d'amis (par paire d'utilisateurs).

Grace à cette méthode on obtient les mêmes résultats qu'a la question précédant à un détail près : la sortie ici est un RDD.

Exercice 2

Nous souhaitons connaître les k mots le plus fréquents dans des site web. Ce résultat sera mené à être gardé dans le disque dur.

Dans un premier temps, en nous basant sur le cours, nous avons décidé de télécharger les packages partagés. Ainsi, nous nous sommes servis des packages suivants :

- Beautiful Soup
- Url.Request
- Functools

Hurtado, Daddio, Sopguombue

Afin que le code soit le plus lisible possible, nous avons décidé de procéder par des fonctions remplissant des taches intermédiaires pour les mettre ensemble à la fin et que cela génère un programme pouvant réaliser la totalité de ce qui nous est demandé et que le fait de modifier une partie soit plus simple.

La première des fonctions est celle qui nous permettra d'accéder à des sites web. Cela nous permet surtout de lire des URL au format HTML et UTF-8. Dans cette étape nous obtiendrions le texte brut qui sera exploité par la suite.

Pour donner suite à l'extraction du texte brut, nous avons encodé notre texte en suivant l'expression régulière suivante "[a-zA-Z]\w{4,}". Elle permet notamment de garder les chaines de caractères d'une longueur supérieure ou égal à 4. Ce choix a été fait afin d'éviter la prédominance des articles qui ne communiquent pas une information pertinente du sujet de l'URL.

Cette façon d'encoder ne permettait pas d'avoir les accents qui sont propres à la langue française ou d'autres langues latines. Ainsi, l'expression régulière a évolué. L'expression finale est : « [A-Za-zÀ-ÖØ-öœŒÇç]+\w{4,} ». Elle permet d'accéder à des caractères tels que les accents, la cédille et la ligature.

Nous nous sommes servis de ce qui a été montré dans le cours de « map » et « reduce » afin de réaliser le WordCount. L'étape « map » nous permet d'avoir la liste de mots et le nombre d'apparitions de ces mots en termes de 1. L'étape « reduce » fera la somme de ces 1 et nous aurons la fréquence de chaque mot à la fin.

Afin d'obtenir les éléments les plus fréquents, Nous avons trouvé un moyen de trier un dictionnaire de façon ascendante et faire ressortir les K dernier mots de ce dictionnaire. L'exportation dans le disque dur a été fait à travers d'un fichier «.txt ». L'utilisateur trouvera ce document dans le même dossier où le « script » Python a été enregistré.

Finalement, toutes ces fonctions sont regroupées dans la fonction WebMax qui aura en output le fichier don nous avons parlé ci-dessus. Elle montrera dans l'environnement Python le dictionnaire et les mots correspondants. Ces mots seront triés de moins à plus fréquents.

À titre d'exemple nous avons testé le site Wikipédia de la Formule 1, sport automobile. Les mots plus fréquents sont Formule, Grand, Modifier, Pilotes et Monde.

Exercice 3

Etape 1 : Analyse des données

On dispose de données bancaires en lien avec l'octroi de prêt hypothécaires, le jeu de données contient 5961 lignes pour 13 colonnes, l'enjeu ici est donc de construire à partir de ces données un modèle permettant de prédire la probabilité de défaut d'un individu souhaitant disposer d'un prêt hypothécaire. Pour se faire il y a parmi ces 13 colonnes, 12 qui serviront de variables et la variable « Bad » qui permet de déterminer si le prêt n'a pas

été remboursé ou s'il y a un grand retard de paiement, ces variables sont décrites ci-dessous :

- **Bad (qualitative) :** la personne a remboursé son crédit sans incident(s) (Bad=0) avec incident(s) (Bad=1).
- **Loan (quantitative) :** montant de la demande de prêt.
- **Mortdue (quantitative) :** montant dû sur l'hypothèque.
- **Value (quantitative) :** valeur de la propriété.
- **Reason (qualitative) :** motif du prêt ; consolidation financière : Debtcon et amélioration habitat : Homelmp.
- **Job (qualitative) :** profession Mgr,Office, Other, ProfXexe, Sales, Self, manqué.
- **Yoj (quantitative) :** nombre d'années dans le travail actuel.
- **Derog (quantitative) :** nombre de demande de report d'échéances de prêt.
- **Deling (quantitative) :** nombre de litiges.
- **Clage (quantitative) :** Age du plus ancien crédit en mois.
- **Ninq (quantitative) :** nombre de demandes récentes de crédit.
- **Clno (quantitative) :** nombre de crédits dans la banque.
- **Debtinc (quantitative) :** Ratio dette sur revenu.

On peut constater un déséquilibre au niveau de nos classes à prédire puisque sur ces 5000 lignes, seulement 1189 concernent des individus en défaut de paiement du crédit, cet élément est à prendre en compte dans notre problème. La plupart des prêts contracté concernent des consolidations de dettes (reason), les individus rattachés aux différents prêts sont issus de secteurs de travail variés (job). En regardant les histogrammes de nos variables, on remarque que la majorité des contractants n'ont pas d'antécédents en termes d'incidents avec de précédents crédits (derog et delinq), près de la moitié n'ont pas de demande de crédit récente (ninq), mis à part ces points là on a une population relativement variée sur les autres postes. Avec nos boxplots on constate la présence de beaucoup de valeurs aberrantes surtout sur les variables continue, sachant que lorsque l'on sépare ces boîtes à moustaches selon le statut de remboursement du prêt on voit qu'il y a un peu plus de valeurs aberrantes du côté des personnes n'ayant pas remboursé leur prêt. En ce qui concerne les tableaux de contingence entre les variables job et reason par rapport à bad, il ne semble pas y avoir de tendance particulière, les effectifs des différents postes rejoignent la répartition de la variable bad. Enfin on distingue un grand nombre de valeurs manquantes au sein du jeu de données, réparties sur presque toutes les variables si ce n'est bad et loan et s'étalant sur près de 2600 lignes (dont plus de 800 correspondent aux individus en défaut).

Etape 2 : Pre-processing

On décide de séparer nos données en train set, validation set, et test set avec une séparation 70/15/15 afin de disposer d'un nombre décent d'observation dans l'ensemble de validation et de test. On applique 2 traitements essentiels sur nos données, le premier concerne les valeurs manquantes, et le second concerne l'encodage des variables non-numériques. Pour les valeurs manquantes on privilégie une imputation par k-plus-proches voisins pour les variables numériques avec 5 voisins en mettant plus de poids sur les voisins les plus proches, cette imputation utilisant les distances entre points implique au préalable de normaliser nos données d'où l'utilisation d'un MinMaxScaler au

préalable ; du côté des variables non-numériques on a simplement remplacé les valeurs manquantes par une valeur « Unknown » qui sera encodée après. De ce fait, en raison de la présence de variables non numériques il nous est nécessaire de les encoder afin de les passer au format numérique pour assurer le bon fonctionnement de nos algorithmes, c'est pourquoi on fait appel à un Catboost encoding qui est une forme altérée de Target encoding permettant d'obtenir une valeur comprise entre 0 et 1 dérivée de la target en limitant le risque de target leakage.

Etape 3 : Modélisation et entraînement

On entre maintenant dans la partie venant répondre au problème posé, on décide donc de tester 3 modèles qui selon nous peuvent convenir à notre problème de classification, ces modèles sont : la régression logistique, l'arbre de décision, et les k-plus-proches voisins. L'avantage de ces modèles est qu'ils sont explicables, ce qui est une obligation dans le secteur bancaires afin de justifier ou non la raison d'un refus d'octroi de crédit. On retrouve de nouveau la petite subtilité pour le KNN où l'on normalise nos données d'entrées afin d'assurer une bonne performance. Pour l'entraînement on passe par SageMaker en utilisant son module sci-kit learn pour pouvoir créer nos propres modèles. Ce point implique la création d'un script d'entraînement (Train_sm) pour permettre d'envoyer nos modèles depuis VScode vers SageMaker, et un script d'inférence (Inference_sm) afin de déployer le modèle dans VScode et faire nos prévisions. Ci-dessous les différents parmètres de nos modèles :

```
Paramètres Régression Logistique:
 {'penalty': 'none', 'solver': 'newton-cholesky', 'random_state': 25}

Paramètres Arbre de décision:
 {'criterion': 'gini', 'splitter': 'best', 'max_depth': 20, 'min_samples_split': 5, 'random_state': 25}

Paramètres K-plus-proches voisins:
 {'n_neighbors': 5, 'weights': 'distance'}
```

Etape 4 : Résultats

On évalue la qualité de la classification à l'aide du rapport de classification et de la matrice de confusion disponible avec sci-kit learn. On s'intéresse à la performance sur le set de validation mais aussi sur celui de test, une bonne performance sur ces 2 ensembles permet de conclure à une bonne généralisation du modèle. Ainsi avec les différents modèles construit on voit que celui performant le moins bien est la régression logistique, on peut voir que peu importe le modèle la performance entre ensemble de validation et de test est assez proche. En outre, si l'on s'attarde seulement sur l'accuracy de l'arbre de décision et du KNN (validation : 0.89 contre 0.9, et test : 0.88 contre 0.91) on pourrait penser que le dernier performe un peu mieux le rendant préférable, cependant en regardant de plus près les autres métriques on se rend compte que le KNN s'avère bien plus optimiste que l'arbre de décision en prédisant très souvent une capacité de rembourser pour les individus, ce qui donne une prévision parfaite de la classe 0 en raison de sa dominance mais peine à distinguer clairement les personnes qui sont en réalité en défaut (d'où la présence d'une precision proche de 1 et d'un recall aux alentours de 0.5 pour la classe 1) étant la classe sous-représentée. Il faut aussi préciser une chose c'est que dans notre problème ce que l'on cherche réellement à éviter ce sont

les faux négatifs, soit des personnes à qui on a octroyé un crédit alors qu'elles ne sont pas en capacités de le rembourser, et sur ce point-là l'arbre de décision se débrouille mieux, il surapprend moins la classe 0 que le KNN. Voici les résultats obtenus avec le l'arbre de décision, modèle que l'on a retenu :

```
Validation set Report:
              precision    recall  f1-score   support

         0.0       0.91      0.96      0.94       717
         1.0       0.79      0.64      0.71       178

    accuracy                           0.89       895
   macro avg       0.85      0.80      0.82       895
weighted avg       0.89      0.89      0.89       895

--------------------------------------------------------
Test set Report:
              precision    recall  f1-score   support

         0.0       0.91      0.94      0.93       716
         1.0       0.74      0.64      0.69       179

    accuracy                           0.88       895
   macro avg       0.83      0.79      0.81       895
weighted avg       0.88      0.88      0.88       895
```