```
# TP 3 : Machine learning avec Spark

<!-- TOC -->

- [TP 3 : Machine learning avec Spark](#tp-3--machine-learning-avec-spark)
    - [Introduction](#introduction)
    - [Chargement du DataFrame.](#chargement-du-dataframe)
    - [Utilisation des données textuelles](#utilisation-des-données-textuelles)
        - [Stage 1 : récupérer les mots des textes](#stage-1--récupérer-les-mots-des-textes)
        - [Stage 2 : retirer les stop words](#stage-2--retirer-les-stop-words)
        - [Stage 3 : computer la partie TF](#stage-3--computer-la-partie-tf)
        - [Stage 4 : computer la partie IDF](#stage-4--computer-la-partie-idf)
    - [Conversion des variables catégorielles en variables numériques](#conversion-des-variables-catégorielles-en-variables-numériques)
        - [Stage 5 : convertir *country2* en quantités numériques](#stage-5--convertir-country2-en-quantités-numériques)
        - [Stage 6 : convertir *currency2* en quantités numériques](#stage-6--convertir-currency2-en-quantités-numériques)
        - [Stages 7 et 8: One-Hot encoder ces deux catégories](#stages-7-et-8-one-hot-encoder-ces-deux-catégories)
    - [Mettre les données sous une forme utilisable par Spark.ML](#mettre-les-données-sous-une-forme-utilisable-par-sparkml)
        - [Stage 9 : assembler tous les features en un unique vecteur](#stage-9--assembler-tous-les-features-en-un-unique-vecteur)
        - [Stage 10 : créer/instancier le modèle de classification](#stage-10--créerinstancier-le-modèle-de-classification)
    - [Création du Pipeline](#création-du-pipeline)
    - [Entraînement, test, et sauvegarde du modèle](#entraînement-test-et-sauvegarde-du-modèle)
        - [Split des données en training et test sets](#split-des-données-en-training-et-test-sets)
        - [Entraînement du modèle](#entraînement-du-modèle)
        - [Test du modèle](#test-du-modèle)
    - [Réglage des hyper-paramètres (a.k.a. tuning) du modèle](#réglage-des-hyper-paramètres-aka-tuning-du-modèle)
        - [Grid search](#grid-search)
        - [Test du modèle](#test-du-modèle-1)
    - [Supplément](#supplément)

<!-- /TOC -->

## Introduction

Dans cette partie du TP, on veut créer un modèle de classification entraîné sur
les données qui ont été pré-traitées dans les TPs précédents. Pour que tout le
monde reparte du même point, téléchargez le dataset *prepared_trainingset* (ce
sont des fichiers parquet) situé dans le répertoire [*data*](data).

Pour cette partie du TP, veuillez coder dans l'objet *Trainer*, cela vous évitera
de refaire les preprocessings des TPs précédents à chaque run. Pour lancer
l'exécution du script Trainer, tappez dans un terminal à la racine du projet :
```bash
./build_and_submit.sh Trainer
```

Rappel: Il y a deux librairies de machine learning dans Spark: [*spark.ml*]
(https://spark.apache.org/docs/latest/api/scala/
index.html#org.apache.spark.ml.package), basée sur les DataFrames, et
[*spark.mllib*](https://spark.apache.org/docs/latest/api/scala/
index.html#org.apache.spark.mllib.package), basée sur les RDDs. Il est préférable
d'utiliser `spark.ml` comme le préconise la [doc](https://spark.apache.org/docs/
latest/ml-guide.html).

Le but de ce TP est de construire un [*Pipeline*](https://spark.apache.org/docs/
latest/ml-pipeline.html) de Machine Learning. Pour construire un tel Pipeline, on
```

commence par construire les *Stages* que l'on assemble ensuite dans un Pipeline.

Comme dans [*scikit-learn*](*https://scikit-learn.org/stable/*), un pipeline dans spark ML est une succession d'algorithmes et de transformations s'appliquant aux données. Chaque étape du pipeline est appelée "stage", et l'output de chaque stage est l'input du stage qui le suit. L'avantage des pipelines est qu'ils permettent d'encapsuler des étapes de preprocessing et de machine learning dans un seul objet qui peut être sauvegardé après l'entraînement puis chargé d'un seul bloc, ce qui facilite notamment la gestion des modèles et leur déploiement.

Nous allons utiliser les modules *spark.ml.feature*, *spark.ml.classification*, *spark.ml.evaluation*, *spark.ml.tuning*, et la classe *spark.ml.Pipeline*.

Chargement du DataFrame.

Charger le DataFrame obtenu à la fin du TP 2.

Utilisation des données textuelles

Les textes ne sont pas utilisables tels quels par les algorithmes parce qu'ils ont besoin de données numériques, en particulier pour les calculs d'erreurs et d'optimisation. On veut donc convertir la colonne "text" en données numériques. Une façon très répandue de faire cela est d'appliquer l'algorithme [TF-IDF](*https://spark.apache.org/docs/latest/ml-features.html#tf-idf*).

Stage 1 : récupérer les mots des textes

La première étape est de séparer les textes en mots (ou tokens) avec un tokenizer. Construire le premier stage du pipeline de la façon suivante :

```scala
val tokenizer = new RegexTokenizer()
  .setPattern("\\W+")
  .setGaps(true)
  .setInputCol("text")
  .setOutputCol("tokens")
```

Stage 2 : retirer les stop words

On veut retirer les [*stop words*](*https://en.wikipedia.org/wiki/Stop_words*) pour ne pas encombrer le modèle avec des mots qui ne véhiculent pas de sens. Créer le 2ème stage avec la classe *StopWordsRemover*.

Stage 3 : computer la partie TF

La partie TF de TF-IDF est faite avec la classe *CountVectorizer*. Lire la [doc](*https://spark.apache.org/docs/latest/ml-features.html#tf-idf*) pour plus d'info sur TF-IDF et son implémentation.

Stage 4 : computer la partie IDF

Implémenter la partie IDF avec en output une colonne *tfidf*.

Conversion des variables catégorielles en variables numériques

Les colonnes *country2* et *currency2* sont des variables catégorielles (qui ne prennent qu'un ensemble limité de valeurs, ces valeurs n'ayant, ici, aucune notion d'ordre entre elles), par opposition aux variables continues comme *goal* ou *hours_prepa* qui peuvent prendre n'importe quelle valeur réelle positive. Ici les catégories sont indiquées par une chaîne de charactères, e.g. "US" ou "EUR". On veut convertir ces classes en quantités numériques.

Stage 5 : convertir *country2* en quantités numériques

On veut les résultats dans une colonne *country_indexed*.

Stage 6 : convertir *currency2* en quantités numériques

On veut les résultats dans une colonne **currency_indexed**.

Stages 7 et 8: One-Hot encoder ces deux catégories

Transformer ces deux catégories avec un "one-hot encoder" en créant les colonnes **country_onehot** et **currency_onehot**. Une [page Quora](*https://www.quora.com/What-is-one-hot-encoding-and-when-is-it-used-in-data-science*) sur le one-hot encoding.

Mettre les données sous une forme utilisable par Spark.ML

La plupart des algorithmes de machine learning dans Spark requièrent que les colonnes utilisées en input du modèle (les features du modèle) soient regroupées dans une seule colonne qui contient des vecteurs. On veut donc passer de

Feature A	Feature B	Feature C	Label
0.5	1	3.5	0
0.6	1	1.2	1

à

Features	Label
(0.5, 1, 3.5)	0
(0.6, 1, 1.2)	1

Stage 9 : assembler tous les features en un unique vecteur

Assembler les features **tfidf**, **days_campaign**, **hours_prepa**, **goal**, **country_onehot**, et **currency_onehot** dans une seule colonne **features**.

Stage 10 : créer/instancier le modèle de classification

Le classifieur que nous utilisons est une [régression logistique](
http://spark.apache.org/docs/latest/api/scala/
index.html#org.apache.spark.ml.classification.LogisticRegression) avec une régularisation dans la fonction de coût qui permet de pénaliser les features les moins fiables pour la classification.

On la définit de la façon suivante :
```scala
val lr = new LogisticRegression()
  .setElasticNetParam(0.0)
  .setFitIntercept(true)
  .setFeaturesCol("features")
  .setLabelCol("final_status")
  .setStandardization(true)
  .setPredictionCol("predictions")
  .setRawPredictionCol("raw_predictions")
  .setThresholds(Array(0.7, 0.3))
  .setTol(1.0e-6)
  .setMaxIter(20)
```

Création du Pipeline

Enfin, créer le [pipeline](*https://spark.apache.org/docs/latest/ml-pipeline.html#ml-pipelines*) en assemblant les 10 stages définis précédemment, dans le bon ordre.

Entraînement, test, et sauvegarde du modèle

Split des données en training et test sets

On veut séparer les données aléatoirement en un training set (90% des données) qui servira à l'entraînement du modèle et un test set (10% des données) qui servira à tester la qualité du modèle sur des données que le modèle n'a jamais vues lors de son entraînement. Cette phase est nécessaire pour avoir des résultats non-biaisés sur la pertinence du modèle obtenu.

Créer un DataFrame nommé **training** et un autre nommé **test** à partir du DataFrame chargé initialement de façon à le séparer en training et test sets dans les proportions 90%, 10% respectivement.

Entraînement du modèle

Entraîner le modèle via le pipeline puis le sauvegarder.

Test du modèle

Appliquer le modèle aux données de test. Mettre les résultats dans le DataFrame `dfWithSimplePredictions`.

Afficher
```scala
dfWithSimplePredictions.groupBy("final_status", "predictions").count.show()
```

Afficher le [*f1-score*](*https://en.wikipedia.org/wiki/F1_score*) du modèle sur les données de test (cette métrique s'obtient via [MulticlassClassificationEvaluator](*https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator*)).

Réglage des hyper-paramètres (a.k.a. tuning) du modèle

La façon de procéder présentée plus haut permet rapidement d'entraîner un modèle et d'avoir une mesure de sa performance. Mais que se passe-t-il si l'ont souhaite utiliser 300 itérations au maximum plutôt que 50 (i.e. la ligne `.setMaxIter(50)`) ? Si l'on souhaite modifier le paramètre de régularisation du modèle ? Si l'on souhaite modifier le paramètre **minDF** de la classe **CountVectorizer** (qui permet de ne prendre que les mots apparaissant dans au moins minDF documents) ? Il faudrait à chaque fois modifier le(s) paramètre(s) à la main, ré-entraîner le modèle, re-calculer la performance du modèle obtenu sur l'ensemble de test, puis finalement choisir le meilleur modèle (i.e. celui avec la meilleure performance sur les données de test) parmi tous ces modèles entraînés. C'est ce qu'on appelle le réglage des hyper-paramètres ou encore tuning du modèle. Et c'est fastidieux.

La plupart des algorithmes de machine learning possèdent des hyper-paramètres, par exemple le nombre de couches et de neurones dans un réseau de neurones, le nombre d'arbres et leur profondeur maximale dans les random forests, etc. Qui plus est, comme mentionné précédemment avec le paramètre **minDF** de la classe **CountVectorizer**, on peut également se retrouver avec des hyper-paramètres au niveau des stages de préprocessing. L'objectif est donc de trouver la meilleure combinaison possible de tous ces hyper-paramètres.

Grid search

Une des techniques pour régler automatiquement les hyper-paramètres est la **grid search** qui consiste à :
- créer une grille de valeurs à tester pour les hyper-paramètres
- en chaque point de la grille
    - séparer le training set en un ensemble de training (70%) et un ensemble de validation (30%)
    - entraîner un modèle sur le training set
    - calculer l'erreur du modèle sur le validation set
- sélectionner le point de la grille (<=> garder les valeurs d'hyper-paramètres de ce point) où l'erreur de validation est la plus faible i.e. là où le modèle a le mieux appris

Pour la régularisation de notre régression logistique on veut tester les valeurs de 10e-8 à 10e-2 par pas de 2.0 en échelle logarithmique (on veut tester les valeurs 10e-8, 10e-6, 10e-4 et 10e-2).
Pour le paramètre minDF de CountVectorizer on veut tester les valeurs de 55 à 95 par pas de 20.
En chaque point de la grille on veut utiliser 70% des données pour l'entraînement et 30% pour la validation.
On veut utiliser le *f1-score* pour comparer les différents modèles en chaque point de la grille.

Préparer la grid-search pour satisfaire les conditions explicitées ci-dessus puis lancer la grid-search sur le dataset "training" préparé précédemment.

Test du modèle

On a vu que pour évaluer de façon non biaisée la pertinence du modèle obtenu, il fallait le tester sur des données qu'il n'avait jamais vues pendant son entraînement. Ça vaut également pour les données utilisées pour sélectionner le meilleur modèle de la grid search (training et validation)! C'est pour cela que nous avons construit le dataset de test que nous avons laissé de côté jusque là.

Appliquer le meilleur modèle trouvé avec la grid-search aux données de test. Mettre les résultats dans le DataFrame `dfWithPredictions`. Afficher le f1-score du modèle sur les données de test.

Afficher
```scala
dfWithPredictions.groupBy("final_status", "predictions").count.show()
```

Sauvegarder le modèle entraîné pour pouvoir le réutiliser plus tard.

Supplément

Pour plus d'information sur la façon dont est parallélisée la méthode de Newton (pour trouver le maximum de la fonction de coût définissant la régression logistique, qui est le log de la vraisemblance) :
- http://www.slideshare.net/dbtsai/2014-0620-mlor-36132297
- http://www.research.rutgers.edu/~lihong/pub/Zinkevich11Parallelized.pdf
- https://arxiv.org/pdf/1605.06049v1.pdf