TP 2 : Début du projet et pre-processings

<!-- TOC -->

<!-- /TOC -->

Nous allons commencer le projet guidé de data-science, sur lequel nous travaillerons jusqu'à la fin de ce module. Dans ce tp nous allons travailler sur le preprocessing des données. Le preprocessing occupe une place importante dans un projet de data science : il s'agit de faire en sorte que les données soient correctement formatées pour l'entraînement du modèle, et parfois de l'aider en ajoutant de l'information (feature-engineering).

Introduction

Objectif: Réaliser un modèle capable de prédire si une campagne Kickstarter va atteindre son objectif ou non.

Données: Les données sont disponibles sur [kaggle](*https://www.kaggle.com/codename007/funding-successful-projects*).

Commencez par lire la description des données et la description de chaque colonne en cliquant sur **train.csv** dans la partie **Data Sources**.
Téléchargez le dataset pré-cleané **train_clean.csv** présent dans le dossier *data* (pour jouer en mode "ultra nightmare" utilisez plutôt le fichier **train.csv**).

Nous allons utiliser l'IDE IntelliJ qui permet de développer des projets (en particulier de data science) en Spark/scala de façon plus efficace qu'avec le spark-shell. Le shell reste toutefois utile pour tester des lignes de codes ou pour interagir rapidement avec les données.

Setup

Installation de IntelliJ

Suivre la section [Setup TP 2](*setup.md#setup-tp-2-installation-de-sbt-intellij-et-démarrage-du-projet*) dans le fichier `setup.md`.

Charger le template du projet dans IntelliJ

Téléchargez le template du projet Spark [*spark_project_kickstarter_2019_2020*](*https://github.com/Flooorent/spark_project_kickstarter_2019_2020*) puis importez-le dans IntelliJ :
- Ouvrez IntelliJ, dans la page d'accueil cliquez sur "import project".
- Sélectionner le chemin vers le projet décompressé.
- Sélectionner "import project from external model", et sélectionner SBT
- Click next
- Si "project sdk" est vide cliquez sur new, intelliJ devrait directement ouvrir l'arborescence vers votre installation de java.
- Sélectionner "use auto import" (Si cette option n'est pas disponible,

puisqu'elle a été retirée de la dernière version d'IntelliJ, continuer sans rien cocher)
- Click finish
- SBT project data to import : vérifiez que les deux dossiers sont bien sélectionnés
- Click OK
- Attendre qu'intelliJ charge le projet et ses dépendances

Soumettre un script à Spark

Pour exécuter un script Spark/scala, il faut le compiler et en faire un "jar" i.e. un fichier exécutable sur la machine virtuelle java. La compilation d'un script en scala peut se faire avec SBT (équivalent de maven pour java). L'exécutable doit ensuite être lancé sur le cluster Spark via la commande spark-submit.

Pour simplifier les choses un script bash **build_and_submit.sh** est fourni dans le template de projet Spark. Pour que ce script fonctionne, vous devez avoir le dossier spark-2.3.4-bin-hadoop2.7 dans votre répertoire HOME.

(Pour voir plus en détail la procédure complète pour soumettre un Job à un cluster Spark, reportez-vous à la section [HOW TO: lancer un job Spark](*setup.md#how-to-lancer-un-job-spark*) du fichier `setup.md`)

Assurez-vous que le fichier **build_and_submit.sh** soit exécutable en entrant dans un terminal
```bash
cd /chemin/vers/projet/spark
chmod +x build_*and*_submit.sh
```

Il vous suffit ensuite pour lancer votre job spark d'ouvrir un terminal et de faire:
```bash
cd /chemin/vers/projet/spark  # sauf si vous êtes déjà dans le répertoire où se trouve le script
./build_*and*_submit.sh Preprocessor
```
La dernière commande compile le code, construit le jar, puis exécute le job **Preprocessor** sur une instance Spark temporaire sur votre machine (créée juste pour l'exécution du script). Les outputs de votre script (les **println**, **df.show()**, etc.) sont affichés dans le terminal.

Début du TP

Allez dans l'arborescence du projet : **src/main/scala/paristech**. Vous devez voir deux objets: **Preprocessor** et **Trainer**. Nous allons coder dans **Preprocessor**, la partie **Trainer** servira pour le TP 3. Le but du TP 2 est de préparer le dataset, c'est-à-dire essentiellement nettoyer les données, créer de nouveaux features et traiter les valeurs manquantes.

Vous continuerez de coder dans la fonction **main** de l'objet **Preprocessor**.

Pour compiler et lancer le script, tapez dans un terminal à la racine du projet:
```bash
./build_*and*_submit.sh Preprocessor
```

Chargement des données

L'ensemble des ressources nécessaires pour les prochaines questions se trouvent dans la documentation de Spark.

Chargez le fichier **train_clean.csv** dans un **DataFrame**. La première ligne du fichier donne le nom de chaque colonne (aka le header), on veut que cette ligne soit utilisée pour nommer les colonnes du dataFrame.

```scala
val df: DataFrame = spark
  .read
  .option("header", true) // utilise la première ligne du (des) fichier(s) comme
header
  .option("inferSchema", "true") // pour inférer le type de chaque colonne (Int,
String, etc.)
  .csv("/Users/flo/Documents/github/cours-spark-telecom/data/train_clean.csv"")
```

Affichez le nombre de lignes et le nombre de colonnes dans le DataFrame :
```scala
println(s"Nombre de lignes : ${df.count}")
println(s"Nombre de colonnes : ${df.columns.length}")
```

Affichez un extrait du DataFrame sous forme de tableau :
```scala
df.show()
```

Affichez le schéma du DataFrame, à savoir le nom de chaque colonne avec son type :
```scala
df.printSchema()
```

Assignez le type *Int* aux colonnes qui vous semblent contenir des entiers :
```scala
val dfCasted: DataFrame = df
  .withColumn("goal", $"goal".cast("Int"))
  .withColumn("deadline" , $"deadline".cast("Int"))
  .withColumn("state_changed_at", $"state_changed_at".cast("Int"))
  .withColumn("created_at", $"created_at".cast("Int"))
  .withColumn("launched_at", $"launched_at".cast("Int"))
  .withColumn("backers_count", $"backers_count".cast("Int"))
  .withColumn("final_status", $"final_status".cast("Int"))

dfCasted.printSchema()
```

Cleaning

Certaines opérations sur les colonnes sont déjà implémentées dans Spark, mais il est souvent nécessaire de faire appel à des fonctions plus complexes. Dans ce cas on peut créer des *UDFs* (*User Defined Functions*) qui permettent d'implémenter de nouvelles opérations sur les colonnes. Voir la partie [User Defined Functions](spark_notes#user-defined-functions) du fichier `spark_notes.md` pour comprendre comment ça fonctionne.

Affichez une description statistique des colonnes de type *Int* :
```scala
dfCasted
  .select("goal", "backers_count", "final_status")
  .describe()
  .show
```

Observez les autres colonnes, posez-vous les bonnes questions : quel cleaning faire pour chaque colonne ? Y a-t-il des colonnes inutiles ? Comment traiter les valeurs manquantes ? A-t-on des données dupliquées ? Quelles sont les valeurs de mes colonnes ? Des répartitions intéressantes ? Des "fuites du futur" (vous entendrez souvent le terme *data leakage*) ??? Proposez des cleanings à faire sur les données : des *groupBy-count*, des *show*, des *dropDuplicates*, etc.
```scala
dfCasted.groupBy("disable_communication").count.orderBy($"count".desc).show(100)
dfCasted.groupBy("country").count.orderBy($"count".desc).show(100)
```

```
dfCasted.groupBy("currency").count.orderBy($"count".desc).show(100)
dfCasted.select("deadline").dropDuplicates.show()
dfCasted.groupBy("state_changed_at").count.orderBy($"count".desc).show(100)
dfCasted.groupBy("backers_count").count.orderBy($"count".desc).show(100)
dfCasted.select("goal", "final_status").show(30)
dfCasted.groupBy("country", "currency").count.orderBy($"count".desc).show(50)
```

Enlevez la colonne *disable_communication*. Cette colonne est très largement majoritairement à *false*, il n'y a que 322 *true* (négligeable), le reste est non-identifié :
```scala
val df2: DataFrame = dfCasted.drop("disable_communication")
```

Les fuites du futur

Dans les datasets construits a posteriori des évènements, il arrive que des données ne pouvant être connues qu'après la résolution de chaque évènement soient insérées dans le dataset. On a des fuites depuis le futur ! Par exemple, on a ici le nombre de "backers" dans la colonne *backers_count*. Il s'agit du nombre total de personnes ayant investi dans chaque projet, or ce nombre n'est connu qu'après la fin de la campagne.

Il faut savoir repérer et traiter ces données pour plusieurs raisons :
- pendant l'entraînement (si on ne les a pas enlevées) elles facilitent le travail du modèle puisqu'elles contiennent des informations directement liées à ce qu'on veut prédire. Par exemple, si `backers_count = 0` on est sûr que la campagne a raté.
- au moment d'appliquer notre modèle, les données du futur ne sont pas présentes (puisqu'elles ne sont pas encore connues). On ne peut donc pas les utiliser comme input pour un modèle.

Ici, pour enlever les données du futur on retire les colonnes *backers_count* et *state_changed_at* :
```scala
val dfNoFutur: DataFrame = df2.drop("backers_count", "state_changed_at")
```

Colonnes *currency* et *country*

On pourrait penser que les colonnes *currency* et *country* sont redondantes, auquel cas on pourrait enlever une des colonnes. Mais c'est oublier par exemple que tous les pays de la zone euro ont la même monnaie ! Il faut donc garder les deux colonnes.

Il semble y avoir des inversions entre ces deux colonnes et du nettoyage à faire. On remarque en particulier que lorsque `country = "False"` le country à l'air d'être dans currency. On le voit avec la commande
```scala
df.filter($"country" === "False")
  .groupBy("currency")
  .count
  .orderBy($"count".desc)
  .show(50)
```

Créez deux udfs nommées *udf_country* et *udf_currency* telles que :
- *cleanCountry* : si `country = "False"` prendre la valeur de currency, sinon si country est une chaîne de caractères de taille autre que 2 remplacer par *null*, et sinon laisser la valeur country actuelle. On veut les résultat dans une nouvelle colonne *country2*.
- *cleanCurrency* : si `currency.length != 3` currency prend la valeur *null*, sinon laisser la valeur currency actuelle. On veut les résultats dans une nouvelle colonne *currency2*.
```scala
```

```
def cleanCountry(country: String, currency: String): String = {
  if (country == "False")
    currency
  else
    country
}

def cleanCurrency(currency: String): String = {
  if (currency != null && currency.length != 3)
    null
  else
    currency
}

val cleanCountryUdf = udf(cleanCountry _)
val cleanCurrencyUdf = udf(cleanCurrency _)

val dfCountry: DataFrame = dfNoFutur
  .withColumn("country2", cleanCountryUdf($"country", $"currency"))
  .withColumn("currency2", cleanCurrencyUdf($"currency"))
  .drop("country", "currency")

// ou encore, en utilisant sql.functions.when:
dfNoFutur
  .withColumn("country2", when($"country" === "False",
$"currency").otherwise($"country"))
  .withColumn("currency2", when($"country".isNotNull && length($"currency") =!= 3,
null).otherwise($"currency"))
  .drop("country", "currency")
```

On a montré ici l'utilisation d'udfs, mais de façon générale toujours privilégier les fonctions déjà codées dans Spark car elles sont optimisées.

Pour une classification, l'équilibrage entre les différentes classes cibles dans les données d'entraînement doit être contrôlé (et éventuellement corrigé). Affichez le nombre d'éléments de chaque classe (colonne **final_status**). Conservez uniquement les lignes qui nous intéressent pour le modèle, à savoir lorsque `final_status` vaut 0 (Fail) ou 1 (Success). Les autres valeurs ne sont pas définies et on les enlève. On pourrait toutefois tester en mettant toutes les autres valeurs à 0 en considérant que les campagnes qui ne sont pas un Success sont un Fail.

Ajouter et manipuler des colonnes

Il est parfois utile d'ajouter des **features** (colonnes dans un DataFrame) pour aider le modèle lors de son apprentissage. Ici nous allons créer de nouvelles features à partir de celles déjà présentes dans les données. Dans certains cas on peut ajouter des features en allant chercher des sources de données supplémentaires.

Les dates ne sont pas directement exploitables par un modèle sous leur forme initiale dans nos données : il s'agit de timestamps Unix (nombre de secondes depuis le 1er janvier 1970 0h00 UTC). Nous allons traiter ces données pour en extraire des informations pour aider les modèles. Nous allons, entre autres, nous servir des fonctions liées aux dates de l'objet [functions](*https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$*).

Ajoutez une colonne **days_campaign** qui représente la durée de la campagne en jours (le nombre de jours entre **launched_at** et **deadline**).

Ajoutez une colonne **hours_prepa** qui représente le nombre d'heures de préparation de la campagne entre **created_at** et **launched_at**. On pourra arrondir le résultat à 3 chiffres après la virgule.

Supprimez les colonnes **launched_at**, **created_at**, et **deadline**, elles ne sont pas exploitables pour un modèle.

Pour exploiter les données sous forme de texte, nous allons commencer par réunir toutes les colonnes textuelles en une seule. En faisant cela, on rend indiscernable le texte du nom de la campagne, de sa description et des keywords, ce qui peut avoir des conséquences sur la qualité du modèle. Mais on cherche à construire ici un premier benchmark de modèle, avec une solution simple qui pourra servir de référence pour des modèles plus évolués.

Mettre les colonnes **name**, **desc**, et **keywords** en minuscules.

Ajoutez une colonne **text**, qui contient la concaténation des Strings des colonnes **name**, **desc**, et **keywords**. ATTENTION à bien mettre des espaces entre les chaînes de caractères concaténées, car on fera par la suite un split en se servant des espaces entre les mots.

Valeurs nulles

Il y plusieurs façons de traiter les valeurs nulles pour les rendre exploitables par un modèle. Nous avons déjà vu que parfois les valeurs nulles peuvent être comblées en utilisant les valeurs d'une autre colonne (parce que le dataset a été mal préparé). On peut aussi décider de supprimer les exemples d'entraînement contenant des valeurs nulles, mais on risque de perdre beaucoup de données. On peut également les remplacer par la valeur moyenne ou médiane de la colonne. On peut enfin leur attribuer une valeur particulière, distincte des autres valeurs de la colonne.

Remplacez les valeurs nulles des colonnes **days_campaign**, **hours_prepa**, et **goal** par la valeur -1 et par **"unknown"** pour les colonnes **country2** et **currency2**.

Sauvegarder un DataFrame

Sauvegarder le DataFrame final au format parquet sur votre machine :
```scala
monDataFrameFinal.write.parquet("/path/ou/les/donnees/seront/sauvegardees")
```

Attention ! Lorsqu'on sauvegarde un output en Spark, le résultat est toujours un répertoire contenant un ou plusieurs fichiers. Cela est dû à la nature distribuée de Spark.