

MS BGD: Spark TP 3 (2018-2019)

Machine learning avec Spark

Dans cette partie du TP, on veut créer un modèle de classification entraîné sur les données qui ont été pré-traitées dans les TP précédents. Pour que tout le monde reparte du même point, téléchargez ce [dataset](#) (à dézipper une fois télécharger).

Pour cette partie du TP, veuillez **codez dans l'objet "Trainer"**, cela vous évitera de refaire les préprocessings des TP précédents à chaque run. Pour lancer l'exécution du script Trainer faites `./build_and_submit.sh Trainer` dans un terminal à la racine du projet.

Rappel: Il y a deux librairies de machine learning dans Spark: `spark.ml` et `spark.mllib`. Il est préférable **d'utiliser `spark.ml`** qui est plus récente, marche avec les `DataFrames` et permet de construire des Pipelines. `Mllib` ne sera plus développée et ne fonctionne qu'avec des `RDD`.

Le but de ce TP est de construire un Pipeline de Machine Learning:

<https://spark.apache.org/docs/latest/ml-pipeline.html>. Pour construire un tel Pipeline, on commence par construire les "Stages" que l'on assemble ensuite dans un Pipeline.

Comme dans `scikit-learn`, 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 de 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`. Pour avoir plus de détails sur ce que contiennent ces modules allez sur la page de la doc:

<https://spark.apache.org/docs/latest/api/scala/index.html#package>. Dans la barre de recherche, entrez "ml.feature" par exemple. Essayez aussi avec les autres modules mentionnés plus-haut.

1. Charger le dataframe.

Chargez le `DataFrame` obtenu à la fin du TP 2: [dataset](#)

2. Utiliser les données textuelles

Les textes ne sont pas utilisables tels quels par les algorithmes parce qu'ils ont besoin de données numériques en particulier pour faire 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://fr.wikipedia.org/wiki/TF-IDF>

- a. 1er stage: La première étape est séparer les textes en mots (ou tokens) avec un tokenizer. Vous allez donc construire le premier Stage du pipeline en faisant:

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

- b. 2e stage: On veut retirer les 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.
- c. 3e stage: La partie TF de TF-IDF est faite avec la classe CountVectorizer.
- d. 4e stage: Trouvez la partie IDF. On veut écrire l'output de cette étape dans une colonne "tfidf".
http://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction

3. Convertir les catégories en données numériques

Les colonnes "country2" et "currency2" sont des variables catégorielles (qui ne prennent qu'un ensemble limité de valeurs), par opposition aux variables continues comme "goal" ou "hours_prepa" qui peuvent prendre n'importe quelle valeur dans \mathbb{R} . Ici les catégories sont indiquées par une chaîne de caractères, e.g. "US" ou "EUR". On veut convertir ces classes en quantités numériques:


- e. 5e stage: Convertir la variable catégorielle "country2" en quantités numériques. On veut les résultats dans une colonne "country_indexed".
- f. 6e stage: Convertir la variable catégorielle "currency2" en quantités numériques. On veut les résultats dans une colonne "currency_indexed".
- g. 7e stage & 8e stage: transformer ces deux catégories avec un "one-hot encoder" en créant les colonnes "currency_onehot" et "country_onehot"

(<https://www.quora.com/What-is-one-hot-encoding-and-when-is-it-used-in-data-science>)

4. 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:

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



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

- h. 9e stage: Assembler les features "tfidf", "days_campaign", "hours_prepa", "goal", "country_onehot", "currency_onehot" dans une seule colonne "features".
- i. 10e stage: Le modèle de classification, il s'agit d'une régression logistique que vous définirez de la façon suivante:

```
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(300)
```

- j. Enfin, créer le pipeline en assemblant les 10 stages définis précédemment, dans le bon ordre.

5. Entraînement et tuning du modèle

Splitter les données en Training Set et Test Set

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.

- k. 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 classifieur et réglage des hyper-paramètres de l'algorithme

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>

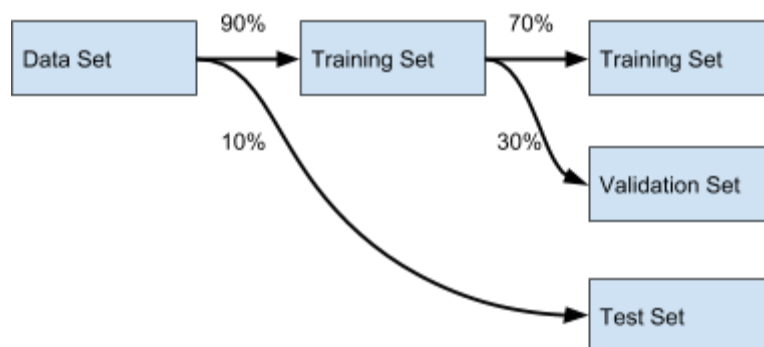
(que vous verrez en cours cette année) avec une régularisation dans la fonction de coût qui permet de pénaliser les features ayant peu d'impact sur la classification.

L'importance de la régularisation est contrôlée par un hyper-paramètre du modèle qu'il faut régler à la main. La plupart des algorithmes de machine learning possèdent des hyper-paramètres, par exemple le nombre de neurones dans un réseau de neurones, le nombre d'arbres et leur profondeur maximale dans les random forests, etc.

Par ailleurs le stage `countVectorizer` a un paramètre "minDF" qui permet de ne prendre que les mots apparaissant dans au moins le nombre spécifié par minDF de documents. C'est aussi un hyperparamètre du modèle que nous voulons régler.

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 et calculer l'erreur du modèle sur le validation set.

- Sélectionner le point de la grille où l'erreur de validation est la plus faible i.e. là où le modèle a le mieux appris. On garde ensuite les valeurs d'hyper-paramètres de ce point.
- Pour la régularisation de la 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 (https://en.wikipedia.org/wiki/F1_score). Cherchez dans *ml.evaluation*.
- l. Préparer la grid-search pour satisfaire les conditions explicitées ci-dessus puis lancer la grid-search sur le dataset "training" préparé précédemment.

Tester le modèle obtenu sur les données test

Pour évaluer la pertinence du modèle obtenu de façon non biaisée, il faut le tester sur des données que le modèle n'a jamais vu pendant son entraînement, et sur des données qui n'ont pas servi pour sélectionner le meilleur modèle de la grid search. C'est pour cela que nous avons construit le dataset "test" que nous avons laissé de côté jusque là.

- m. Appliquer le meilleur modèle trouvé avec la grid-search aux données test. Mettre les résultats dans le dataframe `df_WithPredictions`. Afficher le f1-score du modèle sur les données de test.
- n. Afficher `df_WithPredictions.groupby("final_status", "predictions").count.show()`

Sauvegarder le modèle entraîné pour pouvoir le réutiliser plus tard

Pour pouvoir réutiliser le modèle entraîné dans un autre projet (celui que vous allez faire en groupe par exemple), vous allez le sauvegarder sur vos machines.

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>