A case study from BearingPoint's: Data & Analytics team

Rapport

Simon DAHAN, Louise LOESCH
Louis SALOME, Martin VERSTRAETE

22/04/2018

Chapitre 1

Démarche générale

Méthodologie Les premiers pas dans ce projet ont été facilités par le notebook Jupyter fourni avec le sujet, qui propose des fonctions de bases pour le traitement des données du projet. Une fois les données importées et stockées sous forme de DataFrame de la bibliothèque pandas de python, il était facile de se familiariser avec les différentes features proposées. Nous avons rapidement identifié plusieurs problèmes comme la taille des 3 tables (plusieurs millions de lignes pour certaines tables), les types des valeurs des colonnes, la nécessité d'un pré-traitement pour les données (clusterisation) ou la difficulté à regrouper les tables (merge) selon des clés parfois incompatibles.

Comme dans de nombreux problèmes de traitement de données et de machine learning, nous avons à notre disposition un jeu de données qui sert à l'apprentissage et à la validation et un jeu de données de test pour effectuer une prédiction sur ce nouveau jeu de données. Notre objectif est donc d'obtenir une table de données d'apprentissage/validation la plus pertinente possible à partir des informations brutes. Pour cela nous effectuons des pré-traitements des données pour les rendre les plus exploitables. Certaines données sont des bouts de phrases dont il faut récupérer l'information essentielle, d'autres sont des nombres trop précis (comme le prix) qui peuvent entraîner des phénomène d'overfitting. D'autres données sont à examiner avec attention et regard critique car des valeurs trop extrêmes peuvent représenter des données outlier dont il faut considérer l'impact sur le modèle prédictif.

Finalement, ayant effectué ces traitements sur les données, nous pouvons construire un modèle prédictif en utilisant les différents algorithmes à disposition, et en optimisant chacun des paramètres du modèle.

Dictionnaire Nous avions à notre disposition un dictionnaire des colonnes, qui correspondent aux caractéristiques des données (taille des chaussures, prix, âge du client etc.), pour donner un avis critique sur la pertinence de chaque feature. Certaines étaient très peu influentes, comme la taille maximale commercialisée du produit, ou la durée entre la commande et l'expédition du produit. Cependant, la plupart des features se valaient, aucune ne semblait être particulièrement décisive pour répondre à notre problème. Donc nous ne pouvions pas nous contenter de garder une dizaine ou une vingtaine de features. Il fallait traiter toutes les données. Un point intéressant à mentionner dès maintenant est l'importance d'avoir un avis critique sur certaines features qui semblent *a priori* importantes ou au contraire non importantes, et qui finalement se révèlent inutilisables ou au contraire importante. *Nous discuterons de la façon de juger la pertinence ou non des données plus loin dans ce rapport.*

Chapitre 2

Traitement des données

Comme expliqué en première partie, le traitement des données était une nécessité pour réduire le nombre de labels différents dans la table. Prenons un exemple : la feature *Season* dispose de deux labels : « printemps/été », « automne hiver ». Pour effectuer ce traitement sur les labels, on utilise la fonction *get_ dummies* fournie avec le premier notebook. Quand la feature avait des labels sous forme de flottant, cela ne posait pas de problème de mémoire outre mesure, la source de problème étant plutôt un risque d'overfitting. Quand en revanche le stockage se faisait avec le type string, la fonction *get_ dummies* créait une colonne par chaîne différente. Par exemple la combinaison de couleurs présentes sur le produit donnée par la feature *SupplierColor* comptait 20 000 couleurs et donc autant de colonnes à créer ! Ceci est bien sûr impossible à traiter avec nos moyens et peut être même que ce n'est pas judicieux.

Nous avons alors regroupé les données dans chaque feature pour éviter que ces problèmes ne se produisent. Nous avons choisi de regrouper des features quand l'un des cas suivant se présentait :

- *Similarité des labels* : Deux labels donnent deux informations très proches ou identiques comme par exemple des conseils indiquant de prendre une pointure plus petite mais avec des formulations différentes.
- *Trop de minorités* : 2 ou 3 labels suffisent à décrire 80% des cas et une longue traîne suit et multiplie le nombre de catégories. Très souvent, nous les avons regroupé sous le même label 'Autre'.
- *Valeurs bruitées* : Certaines valeurs sont énormes et donnent une fausse importance à une feature. Par exemple, nous avons estimé qu'il n'y avait pas de différence entre un objet commandé en 17ème ligne sur une liste d'achat et un autre sur la 28ème ligne. Ou encore lorsque les pointures de chaussures sont aberrantes. Nous avons décidé de majorer ou de binariser certaines features à valeurs numériques. Cela permet de limiter l'effet d'overfitting en ayant un jeu de

données plus régulier.

Pour détecter ces choix, nous observions nos données à l'aide de la méthode groupby.

products.groupby('SubtypeLabel').count()

Cela nous permettait de voir combien de labels différents constituaient une colonne, quelles étaient leur valeur et leur importance dans la colonne.

2.1 Les regroupements

Voici les regroupements que nous avons faits :

Table Order

- *LineItem* : Nous avons majoré à 5 la position du produit dans la liste d'achat.
- *TotalLineItems* : Nous avons majoré à 4 la taille de la liste d'achat.
- *Quantity* : Nous avons mis 1 pour un objet commandé 1 fois ou 2 pour un objet commandé plusieurs fois.
- *OrderNumCustomer* : Nous majorons par 3 le nombre de commandes déjà faites par le client
- *PayementModeLabel* : Nous ne conservons que les 2 moyens de paiement principaux, à savoir Carte Bancaire et PayPal, ainsi que le cas ou le produit est gratuit.
- *Isocode* : Nous indiquons seulement si le pays de facturation est la France ou non.
- *OrderCreationDate* : Nous remplaçons la date de la commande par le mois de l'année de la commande pour avoir un label qui regroupe des commandes sur plusieurs années par période de l'année commune.
- *OrderShipDate* : Nous remplaçons la date de l'expédition par le nombre d'heures écoulées entre le moment de la commande et le moment de l'expédition.
- *UnitPMPEUR* : Il fallait changer les virgules par des points et passer en flottant pour interpréter le prix correctement.

Table Products

- *ProductType* : Nous avons jugé pertinent de ne conserver que les 6 plus gros labels et de ranger le reste dans une catégorie ?Autre ?, ces autres labels étant largement minoritaires.
- *SizeAdviceDescription* : Nous avons créé 3 catégories différentes d'indication de taille de chaussure. Soit les chaussures chaussent petit et il faut prendre une taille au-dessus, soit la chaussure taille normal et est confortable, soit elle taille grand et il faut prendre une taille en dessous.
- *CalfTurn* : Nous avons créé 3 catégories différentes de tour de mollet, la distribution semblant être Gaussienne.
- *SubTypeLabel* : De même que pour ProductType, nous ne conservons que les 7 plus grosses catégories.
- *UpperMaterialLabel, LiningMaterialLabel et OutSoleMaterialLabel* : Nous ne conservons que les 3 plus gros labels, ceux-ci représentant la majorité des produits.
- *MarketTargetLabel* : Cette catégorie regroupe le public visé pour chaque type de produit. Nous avons conservé uniquement les 5 catégories les plus importantes, qui regroupent l'essentiel des données.
- *BrandId* : Cette catégorie semble a priori importante. En effet, il s'agit d'un identifiant qui indique la marque de chaussure. De ce fait cette feature permet de savoir quelles sont les marques le plus souvent re-expédiées. De très nombreuses marques étaient présentes et donc nous avons décidé de ne conserver que les plus représentées.
- *UniverseLabel* : Cette feature correspond à l'univers de la marque du produit. A nouveau nous décidons de ne conserver que les 5 labels les plus représentatifs.

Table Costumers

- *CountryISOCode* : Nous indiquons seulement si le pays de résidence du client est la France ou non.
- *BirthDate* : A partir de cette donnée, nous calculons l'âge de la personne qui est selon nous plus parlant que son année de naissance.

Ajout de feature

- Nous avons décidé d'ajouter une feature NbDay qui correspondait à la différence en jours entre la date d'inscription du client et sa date d'achat. Cette feature a permis d'augmenter le score de prédiction.
- Nous avons voulu ajouter une feature NbReturns qui compte le nombre d'articles retournés par un client. En effet si un client a pour habitude de renvoyer ses articles, il va continuer. Pour cela, nous avons utilisé la feature ReturnQuantity de la table ?returns'. Puis à l'aide d'un dictionnaire qui compte le nombre d'articles retournés par client,

nous complétons la nouvelle feature. Cependant beaucoup de clients apparaissent pour la première fois dans la table de test. Cette feature n'était donc pas pertinente.

2.2 Le merge des features

Pour effectuer le regroupement entre les différentes tables de données, il suffisait d'opérer un left join, un peu comme en SQL, entre le DataFrame de base et les deux autres associés aux clients et aux produits. A chaque clé présente dans le premier étaient associées des features correspondant au bon produit ou au bon client. Seulement, certaines clés n'étaient présentes que dans le premier DataFrame, et la ligne en question était alors remplie de « NaN ». Il a alors fallu procéder à une intersection entre les clés présentes entre les tables pour s'acquitter de ces « NaN ». Seulement le DataFrame des features n'était plus de même taille que celui de la table « returns ». Pour remédier à cela, nous avons procédé à un merge entre les deux, en utilisant le couple unique [« OrderNumber », « LineItem »], le numéro de commande pouvant être répété pour différents articles achetés en même temps. Il est important de noter que nous n'avons appliqué ce traitement qu'aux seules données d'entraînement. En effet, en faisant de même pour df_test, celui-ci n'aurait plus eu le même nombre de lignes et nous n'aurions alors été dans l'incapacité de faire des soumissions pour le Challenge.

Dans sa version finale, le table X_train (mise à jour pendant le challenge) contenait plus d'un million de commandes, ainsi qu'une cinquantaine de paramètres différents. Ainsi, le moindre traitement demandait un temps de calcul considérable. Le premier obstacle a été la fonction get_dummies. Cette fonction crée en effet une colonne à chaque nouveau label rencontré pour une feature donnée. Sans pré-traitement, une erreur de mémoire se produisait lorsque nous voulions appliquer get_dummies à l'ensemble de la table. Deux solutions s'offraient à nous : sélectionner un petit nombre de features ou regrouper plusieurs labels d'un même paramètre en grosses catégories.

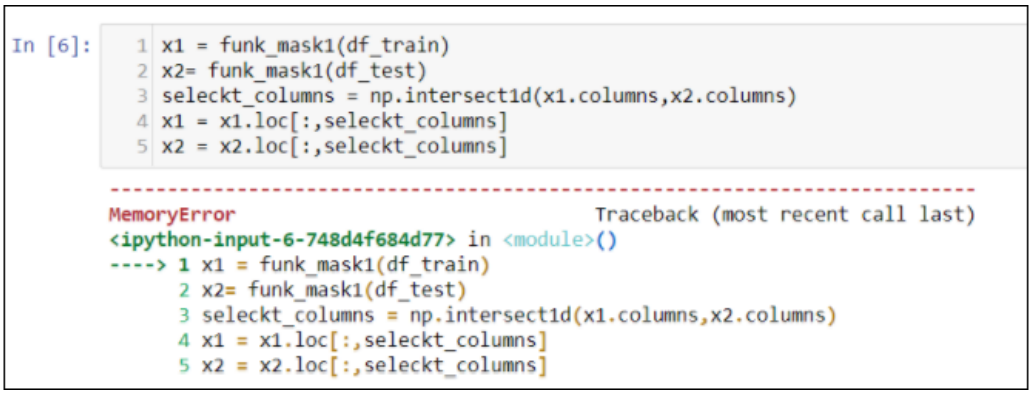

FIGURE 2.1 – Memory Error lors de l'application du mask

2.3 Feature selection

Afin de déterminer l'importance relative des features dans nos algorithmes de prédictions, on utilise la méthode *feature_ importance* de *ExtraTreesClassifier*. Ainsi en fonction du jeu de features que l'on donne en entrée, on arrive à estimer de façon empirique, l'impact de la présence de certains features, sur la prédiction globale et sur l'importance relative par rapport aux autres.

Par exemple on obtient, la figure 2.2, lorsque les features ont une importance relativement identique.

```
Feature ranking:
1. feature 0 (0.199936)
2. feature 1 (0.116649)
3. feature 4 (0.115314)
4. feature 6 (0.098115)
5. feature 11 (0.092092)
6. feature 9 (0.087815)
7. feature 3 (0.085624)
8. feature 10 (0.072319)
9. feature 2 (0.040694)
10. feature 7 (0.034102)
11. feature 14 (0.021056)
12. feature 13 (0.012046)
13. feature 12 (0.011639)
14. feature 5 (0.007420)
15. feature 8 (0.005179)
```

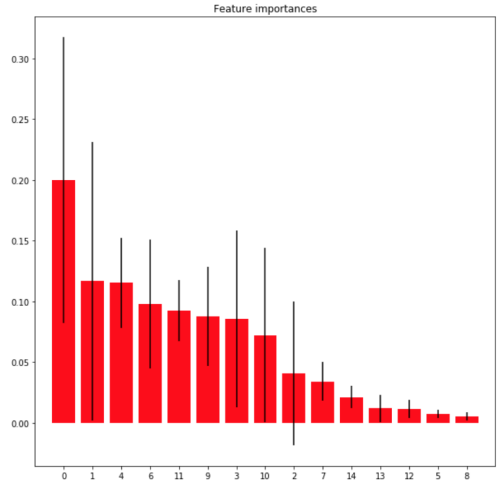

FIGURE 2.2 – Feature importance

Alors qu'on obtient le type de résultat de la figure 2.3 lorsque deux features ont une importance prépondérante sur les autres.

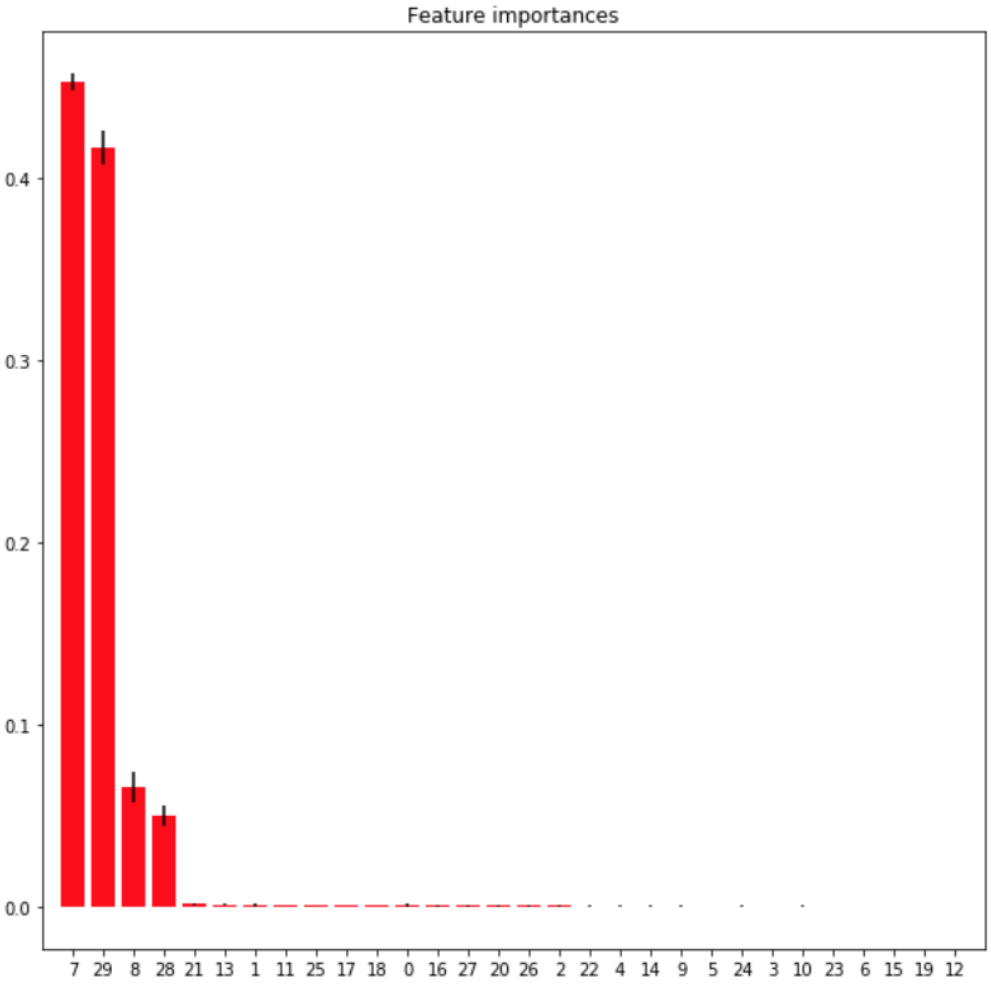

FIGURE 2.3 – Feature importance

Toutes ces informations nous permettent d'une part d'évaluer l'importance des features, mais aussi d'estimer si le pré traitement effectué au niveau du label des colonnes est judicieux.

Chapitre 3

Choix du classifieur

Pour traiter un grand nombre de données, le cours de SD210 a spécifié que les méthodes de *Bagging*, *Boosting* et *Random Forest* étaient les mieux adaptées. Nous avons également discuté avec d'autres groupes pour savoir quels classifieurs marchaient le mieux pour eux et tous ont confirmés l'efficacité de ces méthodes. Après quelques tests rapides de séparateurs de plans comme *LinearDicriminantClassifier*, *QuadraticDiscriminantClassifier* ou *SVM*, nous nous sommes concentrés uniquement sur les *Random Forest*, *Boosting* et *Bagging*.

Pour le choix des paramètres, nous avons utilisé la méthode *GridSearchCV* qui offre différents scoring et une gestion efficace des paramètres à tester. Dans certains cas précis nous faisions une Cross-Validation mais cela était souvent trop long étant donné que couper le jeu de données en 8 nous demande de calculer 8 fois le fitting. Il était parfois plus rapide de directement submit nos résultats sur le site.

Une majorité de nos tests ont étaient faits sur des *Random Forests*. Nous avons pris un nombre de features de l'ordre de la racine carrée du nombre de features disponibles (100 features, m = 10). Il était efficace d'avoir des petits arbres en grand nombre ($max\_depth = 8$, $n\_estimators = 300$) plutôt que des grands arbres en petit nombre, permettant ainsi de réduire l'overtfitting.

Après les fonctions de sklearn, nous avons installé d'autres boosters comme *xgboost* puis *lightgbm* qui sont des classifiers très fréquemment utilisés sur ce type de challenge, comme ceux sur Kaggle par exemple. *XGBoost* (Extreme Gradient Boosting) nous a permis d'établir de nouveaux records après les random forests. Les estimateurs avaient des profondeurs encore plus courtes ($max\_depth = 5$ ou $max\_depth = 6$). En revanche, le temps d'exécution était très long. Nous pouvions également jouer sur le *learning_rate* et les meilleurs résultats étaient donnés par des valeurs de 0.05 à 0.1, donc une exécution lente. Enfin *LightGBM* est basé sur le même principe que *XGBoost* mais a la particularité d'être très rapide à l'exécution. Cela a pu nous

permettre de faire des test rapides comme « le score baisse-t-il si j'enlève cette feature ? ». C'est avec des affinements de ce type que nous avons réussi à atteindre le score de 0.7023. De manière surprenante cependant, il semblerait qu'en règle générale un nombre important de features permet d'obtenir des scores satisfaisants. Autrement dit, en enlevant des paramètres que nous jugions peu cohérents, nous avons vu le score baisser significativement.