Le 22 Avril 2018

Challenge BearingPoint
SD 210 : Prédire la probabilité qu'une paire de chaussure soit retournée

Sommaire :

Auteurs :

Matthis Maillard

Aziz Rachdi

Benjamin Fargeton

Antoine Prat

I) Description des données

Les données étaient classées dans 4 tables différentes, Customers regroupant les caractéristiques des Clients avec comme clé “CustomerId”. La seconde Products avec les caractéristiques de tous les produits, avec comme clé “VariantId”. La troisième, X_train qui a comme clé “OrderNumber” qui permettait de relier les autres tables, et finalement y_train qui donne le nombre de retour (en quantité/produits).

En premier lieu, nous avons dû nettoyer les données. En effet la table product avait un souci : Des descriptions de “SizeAdvice” s’étaient mises dans la case “VariantId”. Nous avons donc fait ce code :

```
products = pd.read_csv("products.csv", sep = ";", encoding = 'iso-8859-1') # encoding in utf-8 produces errors
products.VariantId = products.VariantId.apply(pd.to_numeric, errors = 'coerce') # removes VariantId which are strings
products = products.dropna(subset=['VariantId'])
products = products[products.VariantId > 100] # removes wrong VariantId (VariantId < 100 are errors)
products.VariantId= products.VariantId.astype(int)
```

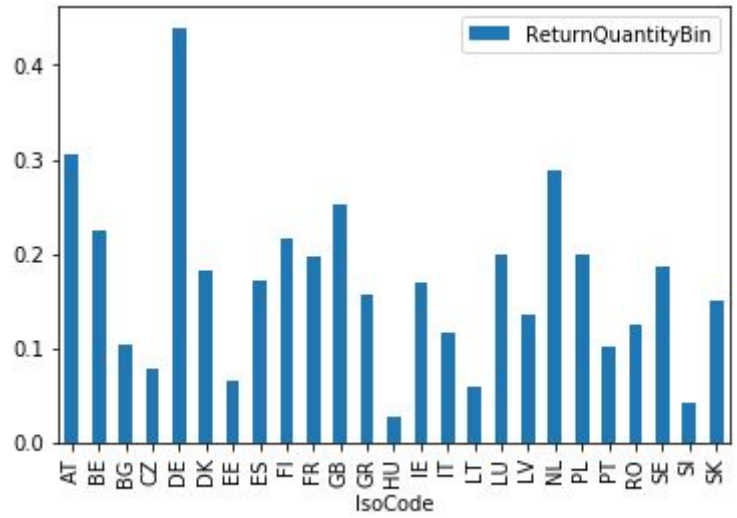
En dernière ligne, nous avons transformé les “VariantId” en Int (certains étaient en float) pour se mettre d’accord avec le type Int de la table X_train et ne pas rater la fusion des tables.

Pour classer les ventes, nous avons procédé en deux étapes. La première consistait en l’extraction des caractéristiques les plus intéressantes. En premier lieu, nous avons fait cela en regardant celles de la base de donnée “X_train”. Dans un second temps, nous avons fusionné les les bases de données “Customers” et “Products” à la base “X_train” et nous avons étudié les caractéristiques liées à ces deux bases. Lors de la deuxième étape, nous avons recherché le classifieur optimal.

II) Choix des caractéristiques

Pour évaluer l’importance d’une caractéristique, nous avons calculé le pourcentage de retour pour chacune de ses valeurs. S’il y avait beaucoup de disparités dans ces pourcentages, nous avons décidé de choisir cette caractéristique car il y a une grande probabilité qu’elle ait une influence sur les raisons de retour. A l’inverse, si les probabilités sont équivalentes, la caractéristique n’a probablement pas d’impact sur le retour.
Nous avons d’abord regardé ces pourcentages sur la base des commandes “Order” qui est en fait “X_train”. Parmi les features sélectionnées, nous avons “IsoCode” qui est l’identifiant du pays vers lequel la commande est expédiée. On peut voir les pourcentages de chaque valeur dans la figure ci-dessous. Nous avons remarqué que les disparités sont très fortes selon les valeurs. Il a fallu également s’assurer que le nombre d'échantillon par valeur est suffisant pour que cela soit significatif. Par exemple, la valeur ayant le pourcentage le plus élevé est “DE”. S’il n’y avait que 20 commandes qui avaient cette valeur, nous ne pouvions pas affirmer qu’elle avait un

impact sur les retours de commande. Ici, ce nombre est de 37 600 donc il est représentatif. C'est pour ces raisons que nous avons choisi cette caractéristique.



Les autres caractéristiques que nous avons sélectionnées sont :

- 'IsOnSale' : Le produit est-il soldé?
- 'IsoCode' : Pays d'expédition
- 'LineItem' : Index du produit dans la commande
- 'OrderNumCustomer' : Nombre de commandes du client
- 'OrderTypelabel' : Type de la commande
- 'PayementModeLabel' : Moyen de paiement
- 'PricingTypeLabel' : Type de prix
- 'Quantity': Quantité
- 'SeasonLabel' : Saison du produit
- 'TotalLineItems' : Nombre de produits dans la commande

Nous avons ensuite décidé de regarder (en liant les tables sur les clés proposées) l'impact d'une caractéristique sur le % de retour pour la table Products.

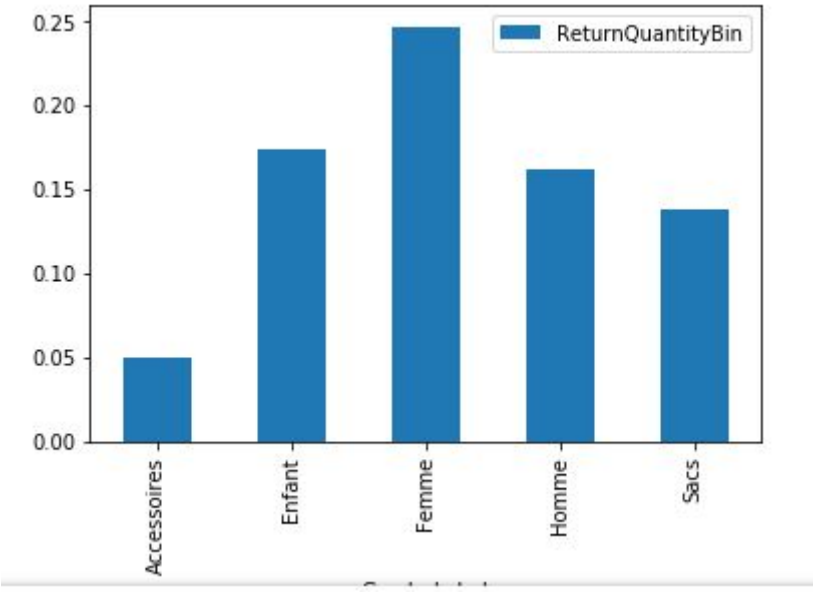
In [4]:
```
def add_universe_products(X,y,carac):
    #On crée la table Customer + X_train pour avoir accès à la BirthDate et OrderCreationDate
    right= products.loc[:,['VariantId',carac]]
    right = right.set_index('VariantId')
    # fusion de la table X avec la table right qui est une sous table de product
    result = pd.merge(X, right, left_on='VariantId', right_index=True, how="left", sort=False)
    # fusion de la table précédemment obtenue avec y qui contient la colonne des retours
    result = pd.merge(result, y, on=['OrderNumber', 'LineItem'])
    return (result)
```

In [5]:
```
# liste des colonnes que l'on voudrait tester
columns = ['GenderLabel', 'MarketTargetLabel', 'SeasonalityLabel', 'UniverseLabel', 'TypeBrand',
           'ProductType', 'SupplierColor', 'MinSize', 'MaxSize', 'CalfTurn', 'UpperHeight', 'HeelHeight',
           'PurchasePriceHT', 'IsNewCollection', 'SubtypeLabel', 'UpperMaterialLabel', 'LiningMaterialLabel',
          'OutSoleMaterialLabel', 'RemovableSole', 'SizeAdviceDescription']
# liste des colonnes dont le nombre de valeurs possibles est inférieur à 40
small_columns = [elt for elt in columns if len(products[elt].unique())<=40]
long_columns = [elt for elt in columns if elt not in small_columns]
```

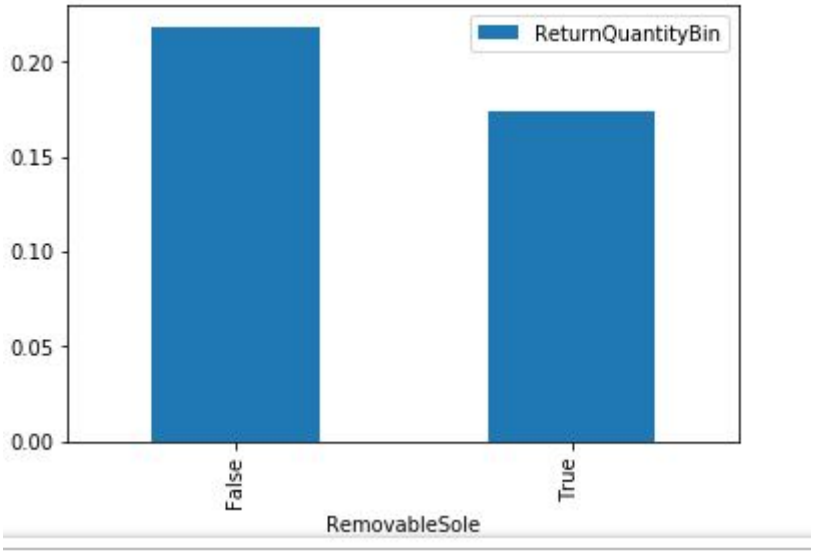
In [6]:
```
for carac in columns:
    # on fusionne X_train avec y_train et la colonne de products carac
    X0=add_universe_products(X_train, y_train,carac)
    # on ne sélectionne la colonne des retours et carac
    join = X0[['ReturnQuantityBin', carac]]
    #on calcule le nombre de retours par valeur divisé par le nombre de commandes ayant cette valeur
    join = join.groupby(carac).sum()/join.groupby(carac).count()
    #on affiche les pourcentages obtenus
    join.plot(kind ='bar')
```

Nous avons séparé en “small_columns” et “long_columns” pour permettre de visualiser les plus petites facilement et rapidement. Grâce à ce plot qui nous renvoie le pourcentage de retour pour chaque élément dans une catégorie, nous avons pu décider d’ajouter ou non cette caractéristique dans notre table X_train. Voici un exemple de deux caractéristiques, une que nous trouvons important et une autre qui ne l’est pas.

GenderLabel qui est intéressante



et RemovableSole qui l’est beaucoup moins.



Nous avons donc décidé de garder en caractéristiques brutes (sans modification) : Product_Type, IsNewCollection, UpperMaterialLabel, GenderLabel, BrandId, SizeAdvice et SubTypeLabel.

Nous avons ajouté une caractéristique : “IsSameSeason” qui se sert de la saison de l’objet et de la date d’envoie (“OrderShipDate”) de notre objet. La caractéristique se forme de cette manière, si l’objet est envoyé dans le bon mois (Avril à Septembre inclu) pour Printemps/Été et le reste de l’année pour Automne/Hiver alors notre caractéristique valait sinon 1, 0. Voici le code pour la créer :

```
def add_is_same_season(d):
    right = products.loc[:,['VariantId','SeasonLabel']]
    right = right.set_index('VariantId')
    merged = pd.merge(d.loc[:,['VariantId', 'OrderShipDate']] , right, left_on = 'VariantId',
                      right_index=True ,how='left', sort = False)
    merged.OrderShipDate = pd.to_datetime(merged.OrderShipDate).dt.month
    hiver = (merged.OrderShipDate > 9)*1 + (merged.OrderShipDate < 4)*1
    same_season = (hiver.astype(bool) == (merged.SeasonLabel == 'Automne/Hiver'))
    d = d.assign(SameSeason = same_season*1)
    return(d)
```

NB: Nous ne mettrons pas le code pour les caractéristiques brutes, en effet c’est une simple fusion des tables grâce aux clés.

Cependant pour le “BrandId”, nous avons aussi modifié les catégories. Effectivement, nous avons préféré que chacune ait un minimum de données à l’intérieur, ceci est fait grâce à la fonction suivante.

```
def pivot(d,feat):
    temp = pd.merge(d, y_train, on=['OrderNumber', 'LineItem'])
    pivot = temp[[feat, 'ReturnQuantityBin']].groupby([feat], as_index=False).agg(['mean', 'count'])
    return pivot.sort_values(by=[('ReturnQuantityBin','mean')], ascending=True)

# Dictionary from pivot
def dict_from_pivot(rate):
    return dict(zip(np.array(rate.index.values),np.arange(1,int(len(rate))+1)))

# Converting BrandId, #t1 is X_train with all the new features
pvtBrand = pivot(t1,'BrandId')
pvtBrand = pvtBrand[(pvtBrand['ReturnQuantityBin','count'] >= 1000)]
convert_BrandId = dict(zip(np.insert(np.array(pvtBrand.index.values),0,0),np.arange(0,int(len(pvtBrand)))))
#t2 is X_test with all the new features
t2['BrandId'] = t2['BrandId'].map(convert_BrandId)
```

En ce qui concerne customers, cette table est petite, nous avons extrait “Gender” mais aussi “BirthDate” que nous avons transformé en classe, nous connaissons désormais l’âge de chaque customer à la date d’achat du produit.

```
def add_age(d):
    right = customers.loc[:,['CustomerId','BirthDate']]
    right = right.set_index('CustomerId')
    merged = pd.merge(d , right, left_on = 'CustomerId', right_index=True ,how='left', sort = False)
    order_date = pd.to_datetime(merged['OrderCreationDate'])
    merged=merged.drop(merged.loc[lambda df: df.BirthDate > str(2016), :].index)
    birth_date = pd.to_datetime(merged['BirthDate'])
    age = order_date - birth_date
    age = age.dt.days/365.25

    age_final = (age<14)*1
    age_final += (age<20)*1
    age_final += (age<30)*1
    age_final += (age<40)*1
    age_final += (age<60)*1
    age_final += (age<90)*1
    age_final -= (age<5)*6

    return (d.assign(CustomerAge = age))
```

Nous avons fait plusieurs tests en ajoutant age et age_final, nous avons obtenu des meilleurs résultats avec age, il faut penser que catégoriser une caractéristique déjà numérique n'est pas une bonne idée.

III) La prédiction et les Classifieurs.

Les algorithmes de classification fonctionnent sur des valeurs numériques, or, dans certaines de nos caractéristiques, nous avons des "string". Nous avons remédié à ce problème avec la fonction mask.
Celle-ci était déjà définie dans le notebook initial mais reste très importante pour notre challenge. Elle fait principalement deux choses, la première est de masquer les caractéristiques que l'on ne choisit pas, nous avons décidé de "cacher" les dates mais aussi les identifiants et "BillingPostalCode" qui prend énormément de place avec la fonction "get_dummies". Cette fonction agit comme une indicatrice, pour chaque valeur de la caractéristique. Ainsi, si une colonne contient N valeurs différentes alors il y aura N colonnes qui seront créées. Il s'agit de la deuxième fonctionnalité de la fonction Mask : transformer les caractéristiques non numériques en nombres. Enfin, les valeurs nulles ("Nan") sont remplacées par la valeur moyenne de la colonne.

Nous avons donc décidé de cacher les colonnes non numériques coûteuse en place après l'application de "get_dummies" ou qui ne sont pas pertinentes :
"OrderCreationDate", "OrderNumber", "VariantId", "CustomerId", "OrderCreationDate", "OrderShipDate", "BillingPostalCode", "DeviceTypeLabel", "CustomerTypeLabel" et "OrderStatusLabel"

Logistic Regression
Pour ce classifieur simple, nous avons cherché à trouver le paramètre de pénalisation en utilisant "GridSearchCV". C'est à dire que pour chaque paramètre nous avons fait une validation croisée et le paramètre choisi finalement correspond à celui pour lequel nous avons eu le meilleur score. Nous avons trouvé un paramètre de 0.01. Les scores obtenus étaient relativement faible, le maximum que l'on ait eu est de 0.67. Nous avons ensuite essayé de réduire la dimension du problème en utilisant la PCA. Pour cela, nous avons utilisé la fonction Pipeline et GridSearchCV qui permet de calculer les score de validation croisée lorsque la PCA est appliquée au données. Malheureusement, cela n'a pas permis d'avoir des meilleurs scores. Nous avons donc décidé de choisir d'autres classifieurs car cet algorithme est trop simple pour les données complexes que nous avons.

XGBoost

Pour le classifieur XGBoost, performant sur des grands dataset. C'est grâce à lui que nous avons obtenu le meilleur score avec les paramètres suivants, obtenus grâce à un GridSearch puis une Bayesian Optimization de notre XGBClassifier. Cet algorithme est une implémentation du GradientBoosting :

```
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
```

```
stds = StandardScaler()
x1_scaled = stds.fit_transform(x1)
```

```
model = XGBClassifier(booster = 'gbtree', learning_rate= 0.1, n_estimators=300,
                      objective = 'binary:logistic', silent = False,
                      alpha=2.2566, colsample_bytree = 0.7527, reg_gamma = 2.3014,
                       max_depth = 5, min_child_weight=17.1129, subsample = 0.7853)
model.fit(x1_scaled, y_train.ReturnQuantityBin)
y_tosubmit = model.predict_proba(stds.transform(x2))
np.savetxt('y_pred.txt', y_tosubmit[:,1], fmt='%f') #bestscore
```

Fonctionnement : C'est un algorithme d'optimisation qui travaille sur les arbre de classification. Il combine comme les algorithmes d'ensembles des modèles faibles pour créer une bonne estimation. De plus, à chaque itération il crée un arbre à partir de l'arbre calculé précédemment pour minimiser l'erreur résiduelle ce qui donne la meilleur prédiction possible. Il minimise avec comme critère la perte convexe et le terme de pénalité (L2 et/ou L1).

Perceptron Multicouche

Nous avons utilisé la fonction 'MLPClassifier' de sklearn pour donner une probabilité. Notre motivation à utiliser ce classifieur vient du très grand nombre de données et la grande dimension de l'ensemble de Test. En effet, ce modèle permet de trouver des relations non-linéaires entre les données. Nous n'avions pas besoin de centrer et réduire les données pour ce classifieur. Enfin, la recherche de la solution optimale du Perceptron Multicouche fonctionne par minimisation d'énergie, il est donc possible que la solution retournée soit un minimum local et non global. Pour contourner ce problème, nous avons fait un Bagging de MLPClassifier, c'est à dire que nous en calculant plusieurs. Ainsi, il y a moins de chance que tous les classifieurs atteignent le même minimum local et il y a une plus forte probabilité qu'un grand nombre atteignent le minimum global. De plus, faire un bagging réduit la variance du problème. Avec cette méthode de fonctionnement, nous avons obtenu un score très légèrement inférieur à celui utilisant XGBoost. Le principal inconvénient de cette méthode est le temps de calcul puisque le fait de calculer un grand nombre de perceptron est très coûteux.