
# Projet Data Mining

# Classement et ciblage

Léa REGAZZETTI
Chloé CARRIE

Master 1 Informatique
Promotion 2020/2021

## Table des matières

## Introduction

Ce dossier s'intègre dans le cadre du contrôle continu des connaissances de nos études en première année de master informatique, et plus particulièrement lors de notre cours de data mining. Dans ce dossier, nous retraçons nos démarches en utilisant la méthodologie CRISP-DM.

Pour répondre aux consignes de ce projet, nous avons dû produire un système de classement permettant de prédire le plus précisément possible les valeurs de la variable cible V200, et un système de scoring qui permet de cibler une modalité de la variable cible V200. Puis, la troisième partie consistait à construire une nouvelle variable cible en regroupant des modalités de V200 dans le but de créer des "super-modalités" puis reproduire la démarche afin de produire un système qui prédit cette variable avec le taux d'erreur le plus bas.
Nous avons choisi de réaliser ce projet en langage Python sur Jupyter Notebook.

## I. Présentation des données et prétraitements

Nous avons reçu un fichier texte de 494 021 individus et 200 variables, notées V1,V2,V3,....,V200. La variable V200 est la variable cible qui comporte 23 modalités, les 199 autres sont les variables explicatives. La majorité d'entre elles sont quantitatives sauf : V160, V161, V162.
Nous avons commencé par importer le fichier et vérifier si l'importation s'était bien réalisée en affichant les informations et les dimensions du fichier de données.

Puis, on a réalisé des prétraitements sur les variables qualitatives pour pouvoir effectuer des modèles de classement et de scoring. En effet, nous avons recodé les variables V160, V161 et V162 en 0/1. Le recodage diffère légèrement selon la méthode d'analyse utilisée. En effet, pour l'arbre de décision, nous sommes passés par un codage disjonctif complet, pour la régression logistique avec H2O aucun recodage n'a été effectué, tandis que pour les autres méthodes nous avons créé M-1 indicatrices, avec M le nombre de modalités de la variable qualitative, pour éviter des problèmes de colinéarité dans certains calculs. Dans le code suivant, seule l'option drop_first=True n'est pas présente pour le codage disjonctif complet.

```
#Création des dummies variables pour les modalités des variables qualitatives (V160,V161 et V162)
#liste des variables qualitatives
import numpy
lstQuali = [var for var in df.columns[:-1] if df[var].dtype == numpy.object_]
print(lstQuali)
```

```
['V160', 'V161', 'V162']
```

```
#Recodage des variables qualitatives
dfQualiEncoded = pandas.get_dummies(df[lstQuali],drop_first=True)

#liste des variables quantitatives
lstQuanti = [var for var in df.columns[:-1] if df[var].dtype != numpy.object_]

#réunir dummies et quantitatives dans le même data frame
dfNew = pandas.concat([dfQualiEncoded,df[lstQuanti]],axis=1)

#rajouter la variable cible
dfNew['V200'] = df['V200']
```

Ainsi, notre nouveau fichier de données comporte les variables quantitatives, les variables qualitatives recodées et la variable cible V200.

## II. Méthodes de classement

La première étape de ce projet est de produire un système de classement pour prédire les valeurs de la variable cible. Le modèle définitif doit s'écrire comme une combinaison linéaire basée sur les variables initiales ou à l'aide d'un système à base de règle. Afin de déterminer le meilleur modèle, nous en avons comparé plusieurs, dont notamment un arbre de décision, une analyse discriminante linéaire, et deux régressions logistiques avec des packages différents que nous allons décrire dans la sous-partie suivante. Nous voulons donc appliquer le schéma type de l'analyse prédictive , c'est-à-dire commencer par scinder les données en échantillons d'apprentissage et de test, puis développer un modèle sur l'échantillon d'apprentissage et évaluer ses performances sur l'échantillon test. Enfin, il s'agit de développer le modèle que l'on considère comme le meilleur.

### *A. Modélisations*

La première étape commune à toutes nos modélisations consiste à scinder les données en échantillon d'apprentissage et de test. Pour cela, nous utilisons la fonction train_test_split(). L'option random_state=1 nous permet de reproduire l'approche, et surtout permet de comparer les performances des différents modèles testés. Nous avons ensuite vérifié les dimensions des sous-échantillons.
A noter que pour le codage disjonctif complet, nous avons 277 variables au lieu de 274. L'échantillon d'apprentissage contient environ 70% des données soit 345 815 individus tandis que l'échantillon test en contient environ 30% soit 148 206 individus.

Ensuite, pour chaque échantillon, nous avons isolé les variables prédictives de la variable cible, excepté pour la régression logistique avec le package H2O dont nous donnerons les détails dans la partie consacrée.

```
#Isole les variables prédictives (x) de la variable cible (y) pour chaque échant
illon (test et train)

x_train=train.iloc[:,0:dfNew.shape[1]-1]
y_train=train.iloc[:,dfNew.shape[1]-1]
print(y_train.shape, x_train.shape)
```

```
x_test=test.iloc[:,0:dfNew.shape[1]-1]
y_test=test.iloc[:,dfNew.shape[1]-1]
print(y_test.shape, x_test.shape)
```

```
(345815,) (345815, 273)
(148206,) (148206, 273)
```

Il y a bien 273 variables prédictives (276 dans le cas du codage disjonctif complet).

Nous avons ensuite effectué un premier traitement qui consiste à supprimer les colonnes de constantes dans les variables prédictives. Pour cela, nous avons utilisé l'algorithme de sélection de variables VarianceThreshold basé sur la variance. Puis nous avons réduit nos échantillons de test et d'apprentissage aux variables sélectionnées par l'algorithme.

```
#Première sélection de variables : pour enlever les colonnes de constantes
from sklearn.feature_selection import VarianceThreshold
selector = VarianceThreshold()
selector.fit(x_train)

#Liste des variables constantes
print(x_train.columns[selector.variances_ == 0])
```

```
Index(['V161_m45', 'V178', 'V179'], dtype='object')
```

```
#Réduction de la base d'apprentissage aux variables sélectionnées
x_train = x_train.iloc[:,selector.get_support()]
print(x_train.shape)
```

```
(345815, 270)
```

```
#réduction de la base test aux mêmes variables
x_test = x_test.iloc[:,selector.get_support()]
print(x_test.shape)
```

```
(148206, 270)
```

Nous disposons alors de 270 variables prédictives (273 dans le cas du codage disjonctif complet).

## 1. Arbre de décision

L'arbre de décision a été réalisé avec le package Sklearn. On a commencé par instancier la classe DecisionTreeClassifier() avec les paramètres par défaut puis on a modélisé l'arbre sur les données d'apprentissage.
Nous avons appliqué ce modèle sur les données de test afin de prédire la variable cible V200. Nous déterminons la matrice de confusion puis nous calculons le taux d'erreur qui est le nombre d'individus qui n'ont pas été correctement prédits par le modèle, toutefois comme nous allons affiner notre analyse, nous avons choisi de ne pas présenter ces résultats, vous les trouverez cependant le fichier Jupyter Notebook *Arbre_decision_modelisation.ipynb*.

Après avoir modélisé l'arbre de décision par défaut, nous avons décidé d'améliorer notre modèle en cherchant les paramètres optimaux du modèle. Pour l'arbre de décision, deux paramètres importants à fixer sont ccp_alpha et max_depth. Le paramètre ccp_alpha permet d'effectuer un post-élagage de l'arbre et max_depth fixe la profondeur de l'arbre.

Dans un premier temps, nous avons cherché la valeur du paramètre "ccp_alpha" la plus précise afin d'avoir un meilleur modèle. Cette recherche a été faite en combinant lignes de code et interprétation graphique. Nous avons stocké dans une variable "best_model" le meilleur modèle puis nous avons utilisé ce modèle pour prédire la variable cible sur les données de test afin de voir si le modèle est meilleur. Nous avons trouvé que la valeur du alpha optimale est 2.8917195610369708e-06. De plus, suite à une lecture graphique, nous avons fixé la profondeur maximale de l'arbre à 15. Nous avons donc refait les calculs de l'arbre de décision avec les paramètres optimaux. Nous obtenons 0.0769 % comme taux d'erreur, ce qui est faible.

Enfin, après la recherche des paramètres optimaux, nous avons procédé à la sélection des variables les plus pertinentes. Pour cela, nous avons utilisé le classement des variables selon leur importance dans l'arbre de décision. Les variables qui influencent le modèle sont celles dont l'importance est supérieure à 0 donc nous avons sélectionné ces variables ci.

```
#On souhaite obtenir l'importance de chaque variable dans l'arbre de décision
importance_variable={"Variable":x_train.columns,"Importance":dtree_opt.feature_i
mportances_}
imp=pandas.DataFrame(importance_variable).sort_values(by="Importance",ascending=
False)
```

```
#On veut seulement celles qui contribuent dans le modèle donc Importance > à 0
variables_importantes=imp.Variable[imp.Importance>0]
```

53 variables sont pertinentes pour l'arbre de décision. Nous pouvons donc sélectionner dans les fichiers d'apprentissage et de test uniquement ces variables afin d'appliquer de nouveau le modèle de l'arbre de décision.

La procédure est donc la même que précédemment. Nous avons appliqué sur les fichiers d'apprentissage et de test comportant uniquement les variables importantes l'arbre de décision avec les paramètres optimaux.
Nous obtenons un taux d'erreur de 0.0007826943578532752.

### 2. Analyse discriminante linéaire

Pour mener cette analyse, nous avons utilisé le package Sklearn. Tout d'abord, nous avons instancié la classe LinearDiscriminantAnalysis() avec les paramètres par défaut, puis nous avons lancé la modélisation en précisant la matrice des variables prédictives et le vecteur de la variable cible.
Nous nous sommes ensuite intéressés aux coefficients des fonctions de classement sans oublier les constantes. Nous avons ensuite effectué la prédiction et calculer les indicateurs de performance du modèle. Nous ne nous attardons pas sur ces résultats, que vous trouverez dans notre fichier Jupyter Notebook Analyse_Discriminante_Lineaire.ipynb, puisque nous avons par la suite affiner notre analyse.

En effet, nous avons commencé par une recherche de paramètre optimaux grâce à la classe GridSearchCV(). Pour cela, une combinaison de paramètres à évaluer a été choisie. Celle-ci contient les différentes options pour les paramètres « solver » et « shrinkage ».

```
#Determination des meilleurs paramètres
#import de la classe
from sklearn.model_selection import GridSearchCV

#instanciation
adl = LinearDiscriminantAnalysis()

#combinaisons de paramètres à évaluer
parametres = [{'solver':['svd', 'lsqr', 'eigen'],'shrinkage':[None,'auto']}]

grid=GridSearchCV(estimator=adl,param_grid=parametres,scoring='accuracy')
grille = grid.fit(x_train,y_train)
```

```
#résultat pour chaque combinaison
print(pandas.DataFrame.from_dict(grille.cv_results_).loc[:,["params","mean_test_
score"]])

#meilleur paramétrage
print(grille.best_params_)

#meilleur performance - estimée en interne par validation croisée
print(grille.best_score_)


#prédiction avec le meilleur modèle
y_pred3 = grille.predict(x_test)

#succès en test
print(metrics.accuracy_score(y_test,y_pred3))
```

```
                                    params  mean_test_score
0      {'shrinkage': None, 'solver': 'svd'}         0.994864
1     {'shrinkage': None, 'solver': 'lsqr'}         0.990087
2    {'shrinkage': None, 'solver': 'eigen'}              NaN
3    {'shrinkage': 'auto', 'solver': 'svd'}              NaN
4   {'shrinkage': 'auto', 'solver': 'lsqr'}         0.994014
5  {'shrinkage': 'auto', 'solver': 'eigen'}         0.994011
{'shrinkage': None, 'solver': 'svd'}
0.9948643060595984
0.9950676760724937
```

Ainsi, le meilleur paramétrage correspond à celui par défaut de la classe LinearDiscriminantAnalysis(), mais notons toutefois que le paramétrage importe peu au regard du taux de succès.

Par la suite, nous avons choisi de réaliser une sélection de variables afin de ne garder que celles qui contribuent au modèle. Pour cela, nous avons utilisé l'algorithme RFE (Recursive Feature Elimination) appliqué au modèle avec les paramètres optimaux. Étant donné que nous avons gardé les paramètres par défaut de l'algorithme, celui-ci sélectionne la moitié du nombre de variables initialement présentes.

```
lda = LinearDiscriminantAnalysis(solver='svd', shrinkage=None, store_covariance=
True)
#algorithme de sélection de var.
from sklearn.feature_selection import RFE
selecteur = RFE(estimator=lda)

#lancer la recherche
sol = selecteur.fit(x_train,y_train)

#nombre de var. sélectionnées
print(sol.n_features_)
```

```
135
```

Ainsi, nous sélectionnons uniquement ces variables dans nos échantillons de test et d'apprentissage.

Nous pouvons donc développer notre modèle précédemment choisi sur les variables sélectionnées, puis réaliser la prédiction de la variable cible sur l'échantillon test et enfin en évaluer les performances. Nous nous sommes également intéressés aux coefficients des fonctions de classement sans oublier les constantes.
Nous obtenons alors un taux d'erreur de 0.006409996896212022.

### 3. Régression logistique avec Sklearn

Pour cette analyse, nous avons utilisé le package Sklearn de Python. Comme pour toutes les classes de ce package, la première étape a été d'instancier la classe LogisticRegression(). Au début, nous avons gardé le solver par défaut « lbfgs », et nous avons précisé la valeur « multinomial » pour le paramètre « multi_class » étant donné que la variable cible possède 23 modalités. Nous avons toutefois dû réaliser une normalisation des données grâce à StandardScaler().

```
#importation de l'outil
from sklearn import preprocessing
#instanciation
stds = preprocessing.StandardScaler()
#transformation
z_train = stds.fit_transform(x_train)
```

Nous avons appliqué cette même transformation sur l'échantillon de test.

```
#transformation de l'échantillon test
z_test = stds.transform(x_test)
```

Ensuite, nous avons cherché à déterminer les paramètres optimaux pour nos données. Pour cela, nous avons utilisé GridSearchCV, comme pour l'analyse discriminante, avec comme combinaison de paramètres, « sag », « saga » et « lbfgs » pour le solver, « ovr » et « multinomial » pour le paramètre multi_class. Nous avons ainsi obtenu comme meilleur paramétrage celui-ci :

```
{'multi_class': 'ovr', 'penalty': 'none', 'solver': 'lbfgs'}
```

Nous avons ensuite effectué une sélection de variables en lien avec les paramètres optimaux trouvés précédemment. Pour cela, nous avons utilisé l'algorithme SelectFromModel().

```
from sklearn.feature_selection import SelectFromModel
selecteur = SelectFromModel(estimator=lr)

#lancer la recherche
sol = selecteur.fit(z_train,y_train)
```

Ainsi, 65 variables ont été sélectionnées. Nous avons alors réduit nos échantillons de test et d'apprentissage aux variables sélectionnées. Puis nous avons construit notre modèle et effectué les prédictions sur le modèle réduit.

```
#réduction de la base d'app. aux var. sélectionnées
#en utilisant le filtre booléen sol.support_
z_new_train = z_train[:,sol.get_support()]
print(z_new_train.shape)
```

```
(345815, 65)
```

```
#construction du modèle sur les explicatives sélectionnées
modele_selLR = lrSkStd.fit(z_new_train,y_train)

#réduction de la base test aux mêmes variables
z_new_test = z_test[:,sol.get_support()]
print(z_new_test.shape)
```

```
(148206, 65)
```

```
#prédiction du modèle réduit sur l'éch. test
y_pred_selLR = modele_selLR.predict(z_new_test)

#évaluation
print(1.0 - metrics.accuracy_score(y_test,y_pred_selLR))
```

```
0.0011200626155486448
```

Pour cette méthode, nous avons obtenu un taux d'erreur en test de 0.0011200626155486448.

### 4. Régression logistique avec H2O

Cette deuxième régression logistique a été réalisé avec le package H2O. Pour l'utiliser, nous l'avons importé et initialiser afin d'utiliser toute la capacité du processus de nos machines.

```
#importation de la librairie H2O
import h2o
#démarrage de la machine "h2o"
h2o.init(nthreads = -1)
```

Avec ce package, nous avons légèrement modifié notre trame de subdivision des données en échantillon de test et d'apprentissage. En effet, après avoir scinder en échantillon de test et d'apprentissage comme décrit précédemment, nous avons typé nos data frames obtenus dans un format spécifique à H20.

```
train_h2o = h2o.H2OFrame(train)
test_h2o = h2o.H2OFrame(test)
```

Ensuite, nous avons importé la classe H2OGeneralizedLinearEstimator, puis nous l'avons instancié en précisant « multinomial » comme valeur du paramètre « family » étant donné que la variable cible comporte 23 modalités.

```
#importation de la régression
from h2o.estimators import H2OGeneralizedLinearEstimator


#instanciation - régression logistique multinomiale
modelH2o = H2OGeneralizedLinearEstimator(family="multinomial")
```

Ensuite, nous effectuons une sélection de variables grâce à la mesure de l'influence des variables dans le modèle que le package nous propose. Nous obtenons ainsi une sélection de 13 variables sur lesquelles nous réduisons nos échantillons d'apprentissage et de test. Nous lançons alors la modélisation, puis nous pouvons réaliser la prédiction sur l'échantillon test. Nous obtenons pour chaque prédiction également le score d'appartenance aux différentes classes.

```
#prédiction sur l'échantillon test - format H2o obtenu
new_predm_h2o = new_modelH2o.predict(new_test_h2o)
#convertir en Pandas
new_ppredm_h2o = new_predm_h2o.as_data_frame()
print(new_ppredm_h2o.head())
```

Nous pouvons alors calculer la matrice de confusion en test et déduire le taux d'erreur, qui est de 0.007347880652605165.

### *B. Évaluation*

Afin d'identifier le modèle définitif, nous avons comparé le taux d'erreur de chacune des analyses que nous avons réalisées. L'objectif étant de prédire le plus précisément possible les valeurs de la variable cible, nous cherchons le modèle avec le taux d'erreur le plus bas. Voici leur récapitulatif :

| **Modèle** | **Taux d'erreur** |
|---|---|
| Arbre de décision | 0.0007826943578532752 |
| Analyse discriminante | 0.006403249531058086 |
| Régression logistique avec Sklearn | 0.0011200626155486448 |
| Régression logistique avec H2O | 0.007347880652605165 |

Le modèle qui a le plus faible taux d'erreur est l'arbre de décision, c'est donc celui-ci que nous allons déployer pour prédire les valeurs de la variable V200.

Voici un extrait de ce modèle (l'expression complète du modèle se trouve dans le fichier *Arbre_decision_modelisation.ipynb*) :

```
from sklearn.tree import export_text
tree_rules = export_text(dtree_opt,feature_names = list(x_train.columns),show_we
ights=True)
print(tree_rules)
```

```
|--- V182 <= 315.50
|   |--- V187 <= 0.33
|   |   |--- V193 <= 0.15
|   |   |   |--- V163 <= 0.50
|   |   |   |   |--- V194 <= 0.06
|   |   |   |   |   |--- V188 <= 0.93
|   |   |   |   |   |   |--- V194 <= 0.05
|   |   |   |   |   |   |   |--- V188 <= 0.56
|   |   |   |   |   |   |   |   |--- weights: [0.00, 74338.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.
00, 0.00, 0.00, 0.00, 0.00] class: m10
|   |   |   |   |   |   |   |   |--- V188 >  0.56
|   |   |   |   |   |   |   |   |   |--- V196 <= 0.51
|   |   |   |   |   |   |   |   |   |   |--- weights: [0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 2.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.
00, 0.00, 0.00, 0.00, 0.00] class: m16
|   |   |   |   |   |   |   |   |   |--- V196 >  0.51
|   |   |   |   |   |   |   |   |   |   |--- weights: [0.00, 28.00, 0.00, 0.00, 0.00
, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00, 0.00] class: m10
|   |   |   |   |   |   |   |--- V194 >  0.05
|   |   |   |   |   |   |   |   |--- V36 <= 2.50
|   |   |   |   |   |   |   |   |   |--- weights: [0.00, 0.00, 0.00, 0.00, 0.00, 0.0
0, 0.00, 1.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00] class: m16
```

Les variables qui composent ce modèle sont les suivantes, classées par ordre de leur importance : V182, V187, V171, V193, V195, V163, V197, V166, V162_m5, V161_m17, V168, V194, V161_m44, V161_m11, V162_m6, V161_m12, V185, V198, V160_m1, V183, V169, V180, V161_m20, V190, V161_m39, V181, V196, V164, V191, V162_m1, V188, V16, V54, V64, V36, V175, V170, V37, V78, V162_m4, V177, V117, V29, V159, V95, V84, V109, V149, V132, V155, V189, V99, V5.

### *C. Déploiement et performance*

Après avoir identifié l'arbre de décision comme le modèle définitif, nous avons sauvegardé ce modèle avec les paramètres optimaux grâce au package pickle. Nous avons également sauvegardé notre sélection de variables dans un fichier texte.

```
#Export de la liste des variables qui contribuent pour le déploiement après
variables_importantes.to_csv('variables_importantes_arbre.txt', header=False, in
dex=False)

#librairie pour sauvegarde du modèle
import pickle

#référence du fichier - ouverture en écriture binaire
f = open("modele_arbre_decision.sav","wb")

#sauvegarde dans le fichier référencé
pickle.dump(dtree_opt, f)

#fichier qu'il faut fermer
f.close()
```

Ainsi, dans notre programme de déploiement, la première étape consiste à importer la base de déploiement. Pour cela, il faut changer le répertoire de travail dans la commande os.chdir() ainsi que le nom du fichier dans la variable name_data.

```
##### Importation du fichier de données et de la liste des variables sélectionné
es
import os
os.chdir('nom du répertoire ‘)
name_data = 'nom_du_fichier.txt'
import pandas
Newdf = pandas.read_csv(name_data, sep = '\t')
```

On importe ensuite la liste des variables sélectionnées lors de la modélisation. Vous trouverez cette liste dans le fichier intitulé « variables_importantes_arbre.txt ».

```
liste_var=pandas.read_csv('variables_importantes_arbre.txt', header=None)
liste_var=liste_var[0].values
```

Comme précédemment décrit dans la section I, nous recodons les variables qualitatives de la base de déploiement et nous réunissons les variables quantitatives et les variables qualitatives recodées dans un data frame intitulé « data_New ». Puis nous effectuons la sélection des variables.

Viens ensuite le chargement du modèle, toujours effectué avec le package pickle. Nous affichons ses paramètres par mesure de vérification.

```
#librairie chargement du modèle
import pickle

#ouverture en lecture binaire
f = open("modele_arbre_decision.sav","rb")

#et chargement
mdl = pickle.load(f)

#fermeture du fichier
f.close()

#type de l'objet chargé
print(type(mdl))
print(mdl)
```

```
<class 'sklearn.tree._classes.DecisionTreeClassifier'>
DecisionTreeClassifier(ccp_alpha=2.8917195610369708e-06, max_depth=15,
                       random_state=1)
```

Nous pouvons alors réaliser nos prédictions, et les exporter dans un fichier texte après les avoir mis sous forme de data frame.

```
#prédiction
pred = mdl.predict(data_selection)

#transformation en data frame
dfPred = pandas.DataFrame(pred,columns=['prediction'])

#exportation
dfPred.to_csv("predictions.txt",index=False)
```

Concernant le nombre d'erreurs de classement sur une base de déploiement comportant 4 898 424 observations, en multipliant ce nombre par le taux d'erreur obtenu dans notre modèle définitif, à savoir 0.0007826943578532752*4 898 424, on obtient 3 834 individus mal classés.

# III. Méthodes de scoring

La deuxième étape de ce projet consistait à produire un système de scoring. Le scoring permet d'attribuer à un individu un score. Ce score se traduit en la probabilité que l'individu ait la modalité positive, ici m16. Nous avons réalisé le scoring à partir de deux modèles : l'analyse discriminante linéaire et la régression logistique.

## *A. Modélisations*

Les premières étapes sont similaires à la partie précédente puisque nous scindons toujours le fichier en deux échantillons : apprentissage et test. La variable cible reste la même ,V200, avec comme modalité positive m16.

```
#Recodage de la variable cible pour le scoring
import numpy
df['V200_recod'] = numpy.where(df['V200']=='m16', 'positive', 'negative')

print(df['V200_recod'].value_counts())
```

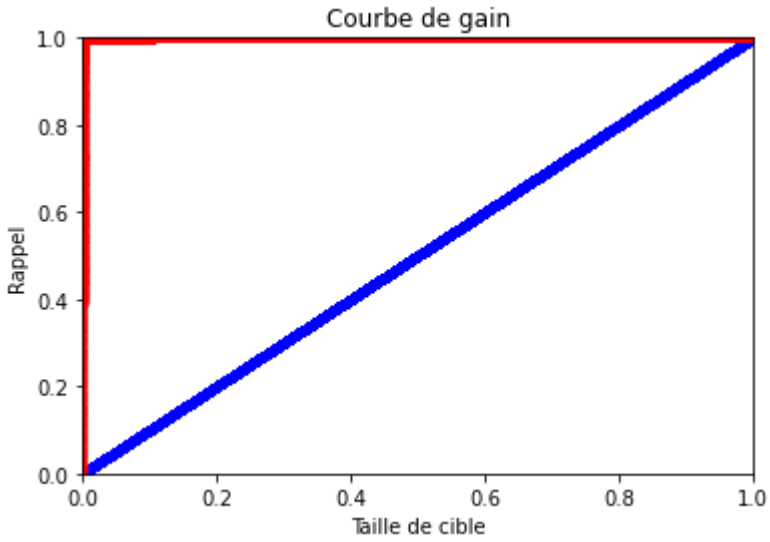
```
negative    492981
positive      1040
Name: V200_recod, dtype: int64
```

Nous supprimons les variables qui sont constantes dans les fichiers. Les modèles de l'analyse discriminante linéaire et de la régression logistique ont été détaillées précédemment, aussi nous allons expliquer uniquement les étapes spécifiques au scoring ici.

### 1. Analyse discriminante linéaire

Dans un premier temps, nous avons instancié le modèle de l'analyse discriminante linéaire avec les paramètres par défaut ceux-ci jouant peu sur les résultats. Et nous avons lancé la sélection de variables sur les données d'apprentissage afin de ne garder que celles qui contribuent dans notre modèle. Après avoir enlevé dans les fichiers d'apprentissage et de test les variables qui ne contribuent pas dans notre modèle, nous avons modélisé sur les données d'apprentissage.

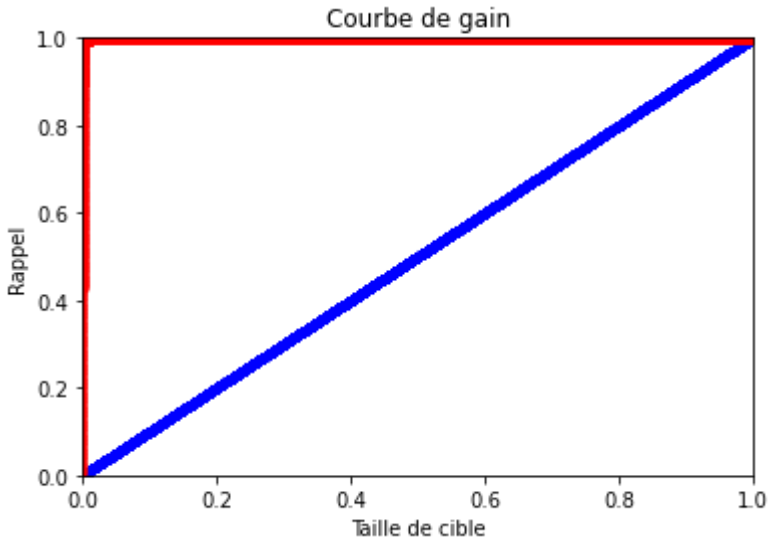
Ce modèle est ensuite appliqué sur les données de test afin de calculer les scores. Nous trions ces scores dans l'ordre décroissant. Pour tracer la courbe de gain, nous avons calculé le rappel et la taille de la cible. La taille de la cible est le numéro de l'individu divisé par le nombre total d'individus. On calcule cpos qui est le nombre de positifs, par exemple, si nous avons 5 individus et que les 3 premiers sont positifs. Nous avons : cpos=1/5 ,cpos=2/5, cpos = 3/5, cpos=3/5,cpos=3/5. Les deux derniers individus sont négatifs donc cpos reste 3/5 car il y a 3 individus avant eux positifs. Le rappel correspond donc à cpos divisé par le nombre total de positifs sur la base. Nous obtenons la courbe de gain suivante :




2. Régression logistique

Pour la régression logistique, nous avons effectué les même étapes, toutefois nous avons instancié la classe avec la valeur « ovr » pour le paramètre « multi_class » étant donné que la variable cible possède seulement 2 modalités, et la valeur « liblinear » pour le paramètre « solver ». Nous avons lancé la régression logistique sur les données d'apprentissage. Nous avons ensuite sélectionné les variables pertinentes. Les fichiers d'apprentissage et de test sont donc uniquement composés des variables sélectionnées. La régression logistique a été relancée sur le fichier d'apprentissage avec uniquement les variables sélectionnées . Ce modèle a été appliqué sur le fichier de test afin de calculer le score de présence et le trier dans l'ordre décroissant.

Pour créer la courbe de gain, nous avons besoin de deux variables : la taille de la cible et le rappel. Nous avons donc calculé ces deux variables puis nous avons tracé la courbe de gain.




### *B. Évaluation*

Pour comparer les deux systèmes de scoring, nous avons calculé pour chaque modèle le nombre de positifs parmi les 10 000 observations qui présentent les scores les plus élevés dans l'échantillon d'apprentissage. Pour le scoring de l'analyse discriminante, nous trouvons qu'il y a 310 positifs parmi les 10 000 individus ayant le score le plus élevé. En utilisant la régression logistique, nous avons obtenu un nombre de 311 positifs. Le modèle qui donne le plus grand nombre de positifs parmi les 10 000 individus ayant les scores les plus élevés est la régression logistique, c'est donc ce modèle que nous avons choisi de déployer.

Voici un extrait de l'expression du modèle :

```
#affichage des coefficients
print(pandas.DataFrame({"var":x_new_train.columns,"coef":lr.coef_[0]}))
#et de la constante
print(lr.intercept_)
```

```
         var      coef
0    V160_m2  0.448881
1    V160_m3 -1.716388
2   V161_m12  0.109859
3   V161_m20 -2.279058
4   V161_m39 -2.964152
..       ...       ...
64      V195 -1.086432
65      V196 -2.745613
66      V197 -0.030499
67      V198 -1.542917
```

```
68      V199  1.428344
[-1.8299586]
```

Les variables qui le composent sont les suivantes :

V160_m2, V160_m3, V161_m12, V161_m20, V161_m39, V162_m10, V162_m11, V162_m2, V162_m3, V162_m4, V162_m5, V162_m6, V2, V8, V10, V11, V12, V39, V43, V44, V56, V57, V60, V66, V72, V75, V76, V81, V86, V88, V89, V90, V94, V96, V99, V100, V101, V108, V113, V116, V120, V123, V124, V125, V139, V143, V147, V148, V150, V153, V157, V158, V164, V168, V182, V183, V184, V185, V186, V187, V188, V192, V193, V194, V195, V196, V197, V198 et V199.

### *C. Déploiement et performance*

Après avoir identifié la régression logistique comme le modèle définitif, nous avons sauvegardé ce modèle avec les paramètres optimaux grâce au package pickle. Nous avons également sauvegardé notre sélection de variables dans un fichier texte.

Ainsi, dans notre programme de déploiement, la première étape consiste à importer la base de déploiement. Pour cela, il faut changer le répertoire de travail dans la commande os.chdir() ainsi que le nom du fichier dans la variable name_data.

```
import os
os.chdir('…')
name_data = nom_du_fichier.txt'

import pandas
Newdf = pandas.read_csv(name_data, sep = '\t')
```

On importe ensuite la liste des variables sélectionnées lors de la modélisation. Vous trouverez cette liste dans le fichier intitulé « variables_importantes_scoring.txt ».

```
liste_var=pandas.read_csv('variables_importantes_scoring.txt', header=None)
liste_var=liste_var[0].values
```

Comme précédemment décrit dans la section I, nous recodons les variables qualitatives de la base de déploiement et nous réunissons les variables quantitatives et les variables qualitatives recodées dans un data frame intitulé « data_New ». Puis nous effectuons la sélection des variables.

Viens ensuite le chargement du modèle, toujours effectué avec le package pickle. Nous affichons ses paramètres par mesure de vérification.

Nous pouvons alors réaliser nos calculs des scores, et les exporter dans un fichier texte après les avoir mis sous forme de data frame.

```
#calcul des probas d'affectation
probas = mdl.predict_proba(data_selection)

#score de 'presence'
score = probas[:,1]

#transformation en data frame
dfScore = pandas.DataFrame(score,columns=['score'])

#exportation
dfScore.to_csv("scores.txt",index=False)
```

Concernant le nombre de positifs parmi les 10 000 observations qui représentent les scores les plus élevés dans une base de 4 898 424 observations, on obtiendrait une valeur de 9960.

# IV. Regroupement des modalités de la variable cible

La dernière étape de ce projet avait pour but de créer une nouvelle variable cible, étant le regroupement des modalités initiales de la variable cible. Pour cela, nous avons réalisé une classification ascendante hiérarchique. Ensuite, il s'agissait de construire un modèle permettant de prédire les nouvelles modalités de cette nouvelle variables cible.

## *A. Classification ascendante hiérarchique*

Avant de commencer notre classification ascendante hiérarchique, nous avons dû effectuer des traitements sur notre fichier de données final présenté en section I. Dans un premier temps, nous avons effectué une sélection de variables grâce à la méthode SelectFwe() avec un seuil de 0.01.

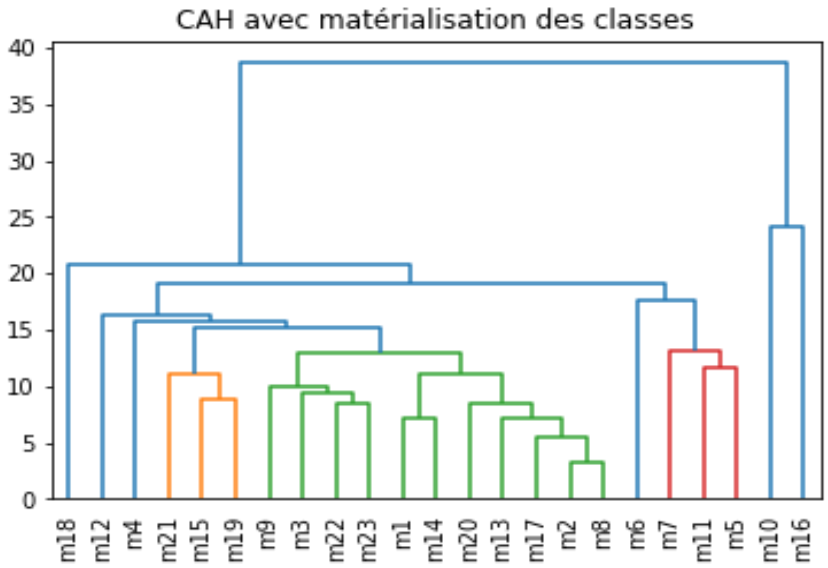
```
#Selection de variables
from sklearn.feature_selection import SelectFwe,f_classif
select=SelectFwe(f_classif,alpha=0.01)
select.fit(data_New[data_New.columns[0:276]],data_New.V200)
data_New.columns[0:276][select.pvalues_ < 0.01]
```

Nous avons ainsi un nouveau fichier de données ne contenant que 116 variables.

Ensuite, étant donné que le nombre d'individus reste conséquent, nous avons choisi de calculer la moyenne de chaque variable regroupée par modalité de la variable cible V200. Nous obtenons alors un fichier ne comportant que 23 « individus ».

```
#On crée un dataframe composé de la moyenne de chacune des variables pour chaque
modalité de V200
df = d.groupby('V200').mean()
```

Les données étant encore très hétérogènes d'une variable à l'autre, nous avons choisi de les centrer-réduire, puis nous avons construit la classification ascendante hiérarchique. Nous pouvons alors afficher le dendrogramme avec la matérialisation des classes.




Celui-ci suggère un découpage en 4 classes. Nous pouvons alors déterminer les groupes et les modalités qui les composent.

```
#affichage des observations et leurs groupes
res = pandas.DataFrame(df.index[idg],groupes_cah[idg])
print(res)
```

```
  V200
1  m21
1  m15
1  m19
2   m1
2   m3
2  m23
2  m22
2  m20
2   m8
2   m2
2  m17
2  m14
2  m13
2   m9
3   m4
4  m12
5  m11
5   m5
5   m7
6   m6
7  m18
8  m10
9  m16
```

Par la suite, nous aurions aimé réaliser différentes modélisations grâce à des systèmes à base de règles ou de combinaisons linéaires permettant de prédire les nouveaux groupes de la variable cible. Le modèle le plus performant aurait été déployé, permettant ainsi d'effectuer le regroupement des modalités, de prédire les nouveaux groupes sur une base de déploiement et de sauvegarder ces prédictions dans un fichier.

## Conclusion

Pour conclure, nous pouvons dire que nous avons testé plusieurs modèles pour prédire le plus précisément possible les valeurs de la variable cible V200. Nous avons choisi l'arbre de décision pour le modèle de classement et la régression logistique pour le système de scoring. Ces deux modèles étaient ceux qui présentaient les meilleurs résultats de prédiction avec le taux d'erreur le plus faible. Nous avons créé un programme de déploiement pour chacun de ces deux modèles et exporté nos prédictions dans un fichier texte. Pour tester, nous avons dupliqué notre base pour obtenir 5 millions d'individus et appliquer les programmes de déploiement dessus. Concernant la partie 3 du projet, nous avons réalisé une classification ascendante hiérarchique mais la modélisation et le déploiement n'ont pas pu être réalisé. En effet, nous avons eu quelques difficultés pour réaliser cette partie.

Pour finir, ce travail nous a permis de mettre en pratique tout ce que nous avons appris durant les séances de TD et de travailler en autonomie. De plus, nous avons appris à travailler avec de gros volumes de données ce qui était la principale difficulté de ce projet et c'était nouveau pour nous dans cette discipline. C'était très intéressant de partir d'un fichier de données et réaliser le projet de A à Z en choisissant le logiciel, les modèles, évaluer la performance et produire un programme de déploiement avec le meilleur modèle. Nous avons approfondi nos connaissances et compétences en Data Mining.