# 3 Data Wrangling - Préparation des données

L'objectif de ce chapitre est de parler de la collecte des donnée. De décrire en quoi l'analyse descriptive des données est une étape indispensable pour bien préparer les données et enfin de donner les outils principaux de la préparation des bases de modélisation. Ce chapitre est subdivisé en quatres sections.

## 3.1 Les principales étapes de la préparation des données

Cette section est consacrée aux principales étapes de la préparation des données. C'est ce qu'on appelle le Data Wrangling.

### 3.1.1 Cycle de vie d'un projet de machine learning

Repartons tout d'abord du cycle de vie d'un projet de machine learning représenté par la figure 3.1 :

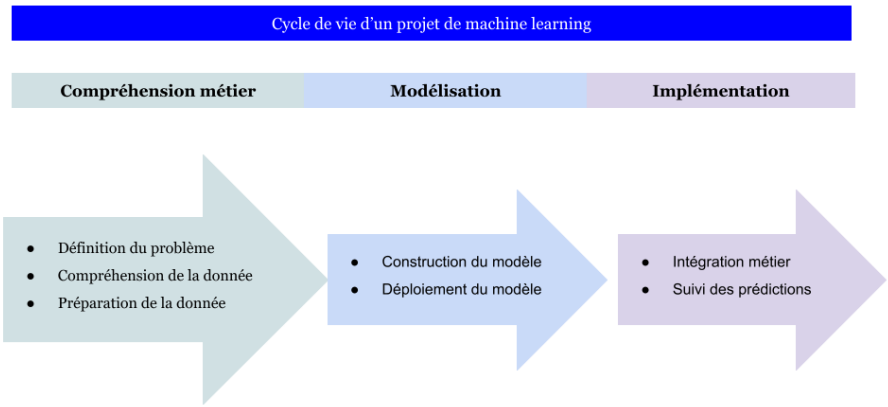


**Figure 3.1 –** Cycle de vie d'un projet de machine learning

Le cycle de vie intègre trois étapes : d'abord la compréhension métier et la préparation des données ; puis la modélisation et enfin l'implémentation.

#### 3.1.1.1 Compréhension métier

La compréhension métier consiste à définir proprement le problème qui sera posé ensuite aux Data Scientists qui lui apportera une réponse en terme de traitement de données afin de résoudre une problématique métierne l'occurence ici pour le scoring de crédit. L'objectif est de classifier les emprunteurs de la banque dans l'ordre croissant des niveaux de risque. Et le travail du Data Scientist est d'élaborer le cahier des charges en termes de collecte de données, de préparation, de modélisation et aussi d'implémentation.

Lors de la collecte des données, les Data Scientists vont comprendre les données. Ils vont les préparer pour passer à la deuxième phase qui est la phase de modélisation.

#### 3.1.1.2 Modélisation

Cette phase de modélisation comprend trois sous étapes : une première étape de selection des variables c'est - à - dire la constitution du jeu de features qui vont être utilisés afin de réaliser les modèles de prédiction qui consiste à éliminer les variables qui ont un faible pouvoir explicatif et à ne retenir que les variables les plus pertinentes. Ensuite, nous avons la construction du modèle de prédiction lui - même. Enfin, il y a le deploiement du modèle c'est - à - dire l'homologation du modèle, la validation par les corps internes et éventuellement par les régulateurs externes, les superviseurs et ensuite, il y a la phase d'industrialisation, c'est - à - dire de développement d'un outil.

#### 3.1.1.3 Implémentation

La troisième phase est l'integration du modèle dans les processus métiers : que ce soit l'octroi de crédit, la gestion des risques, la mesure des risques, le stress testing, etc. Il y a également une étape importante dans la vie du modèle qui est le suivi des prédiction c'est - à - dire le **back testing** : le suivi de la performance du modèle dans le temps. Est ce que le modèle performe bien, oui ou non ? Doit - il être recalibré ? Doit - on le refondre ?

### 3.1.2 Collecte et analyse des données

La préparation des données inclut la phase de compréhension métier et une partie de la prépartion des bases de données pour la modélisation. La compréhension du métier, bien évidemment, va orienter la collecte des données : va - t - on collecter dans les données du métier ? dans les données comptables de la banque ? ou dans les systèmes risques ? dans les systèmes liés aux différents business units ? ou en central dans le groupe ? ou encore au sein des systèmes d'information de chaque filiale ? Egalement, il faut spécifier précisément les données nécessaires car les systèmes d'information des banques contiennent des quantités de données astronomiques. Voilà dont toute la problématique de la collecte des données.

Vient ensuite la phase de compréhension de la donnée, c'est - à - dire l'analyse des sources, d'identification des problèmes de qualité, l'analyse exploratoire des données et par exemple, l'identification, s'il y en a, des problèmes de données manquantes.

Enfin arrive la phase préliminaire de préparation des données, c'est - à - dire le nettoyage des données. A titre d'exemple, s'assurer que les données soient toutes dans la

même devise ou bien que les dates soient dans un seul format et sont bien cohérentes (pas de date de naissance avant 1900 par exemple).

#### 3.1.2.1 Collecte des données

Faisons un zoom à présent sur la collecte des données et sur ses principales étapes. Avant tout, il s'agit de compréhension métier et c'est pour cela que nous ne réduisons pas le sujet de la collecte des données à un problème technique d'aller chercher les données dans des systèmes informatiques et de constituer des fichiers. Le problème est avant tout de demander aux acteurs métiers quels sont les indicateurs pertinents pour faire des prédictions. Cela va orienter, bien évidemment, l'univers des données qui sont utiles pour résoudre le problème.

Dexuièmement, l'analyse des processus métiers, les environnements techniques, les acteurs, les données disponibles et manquantes. L'objectif est de se donner une vision d'ensemble des données qu'on va utiliser mais aussi des données qu'on va récupérer.

Le troisième point c'est comment les décisions seront prises à partir des modèles prédictifs. Cela va orienter à la fois la façon de faire les bases de modélisation mais aussi de développer le modèle prédictif.

Enfin, l'identification des variables explicatives, i.e. l'ensemble de l'univers de toutes les features qui sont théoriquement possibles.

Nous mentionnons que nous ne sommes pas encore dans la phase de réduction du mombre de features pour développer le modèle. Nous sommes toujours dans la phase de collecte des données et l'objectif est bien d'établir la liste des features sur lesquelles nous avons besoin de collecter les données.

### 3.1.3 Feature engineering

Le « feature engineering » recouvre une partie de la préparation des données et de la modélisation, c'est - à - dire la mise au format des données : la préparation colonne par colonne de la matrice des features, s'assurer que les montants sont dans les mêmes dévises ou les mêmes unités. Une erreur que l'on retrouve fréquemment dans les bases de données est d'avoir des montants par exemple en millions d'euros et aussi des montants en euros dans le même champ de la base de données. Ces incohérences, ces erreurs sont fréquentes dans les bases de données et elles engendrent des biais important dans les modèles. Cette phase de préparation des données se situe à la troisième étape de la compréhension métier.

Nous avons également mentionner la sélection des variables, c'est - à - dire l'univers des features que nous récuperons dans la phase de collecte. Bien, il va falloir en déterminer un nombre réduit qui vont être intégré à la construction du modèle.

Ces deux phases de préparation des données et de la sélection des variables constitue ce qu'on appelle le « feature engineering ».

### 3.1.4 Elaboration des bases de modélisation

L'objectif est de préparer une base, i.e. un fichier type « csv » qui sera ensuite exploiter pour la conception des modèles. Ce fichier contient à la fois des observations de défaut et de survie (c'est ce qu'on appelle la réponse) mais aussi les variables explicatives

(les features). Très concrètement le fichier se présente comme une matrice appelée matrice des features et dont une des colonnes contient les réponses. Les lignes de cette matrice représentent les différentes observations. Si les données proviennent de tables différentes, il faut évidemment concevoir des règles dites « règles de jointures » et trouver par exemple les identifiants uniques pour pouvoir fusionner et joindre les bases. A titre d'exemple, lorsqu'on travaille sur des données de crédit, nous avons des données de transaction qui sont souvent dans les bases de transaction, puis des données clients qui sont dans des bases de referentiels clients. Il nous faut joindre ces deux types de bases et donc faire le lien entre chaque emprunteur et l'ensemble de ses transactions.

Le niveau de granularité doit être adapté à la situation métier. Bien évidemment, l'élaboration des bases de modélisation conduit à déterminer trois jeux de données : un jeu de données pour l'entraînement du modèle (*training set*) ; un jeu de données pour la cross - validation c'est - à - dire l'identification des hyperparamètres du modèle (*validation set*) et le jeu de test (*test set*) qui sert à valider le modèle.

**Exemple 3.1.** *Base de données Titanic*

Pour illustrer nos propos et montrer concrètement à quoi ressemble la matrice des features, voici un extrait d'une base de données extrêmement classique qui sert de jeu d'apprentissage pour tous les data scientists qui débutent dans le métier : la base de données des passagers du Titanic.

```
# Chargement de la base titanic
import seaborn as sns
titanic = sns.load_dataset("titanic")
# Informations
titanic.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 891 entries, 0 to 890
## Data columns (total 15 columns):
##  #   Column       Non-Null Count  Dtype
## ---  ------       --------------  -----
##  0   survived     891 non-null    int64
##  1   pclass       891 non-null    int64
##  2   sex          891 non-null    object
##  3   age          714 non-null    float64
##  4   sibsp        891 non-null    int64
##  5   parch        891 non-null    int64
##  6   fare         891 non-null    float64
##  7   embarked     889 non-null    object
##  8   class        891 non-null    category
##  9   who          891 non-null    object
##  10  adult_male   891 non-null    bool
##  11  deck         203 non-null    category
##  12  embark_town  889 non-null    object
##  13  alive        891 non-null    object
##  14  alone        891 non-null    bool
## dtypes: bool(2), category(2), float64(2), int64(4), object(5)
## memory usage: 80.7+ KB
```

**Table 3.1 –** Données titanic - 15 premières observations

| survived | pclass | sex | age | sibsp | parch | fare | embarked | class | who | adult_male | deck | embark_town | alive | alone |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 3 | male | 22 | 1 | 0 | 7.2500 | S | Third | man | TRUE | NA | Southampton | no | FALSE |
| 1 | 1 | female | 38 | 1 | 0 | 71.2833 | C | First | woman | FALSE | C | Cherbourg | yes | FALSE |
| 1 | 3 | female | 26 | 0 | 0 | 7.9250 | S | Third | woman | FALSE | NA | Southampton | yes | TRUE |
| 1 | 1 | female | 35 | 1 | 0 | 53.1000 | S | First | woman | FALSE | C | Southampton | yes | FALSE |
| 0 | 3 | male | 35 | 0 | 0 | 8.0500 | S | Third | man | TRUE | NA | Southampton | no | TRUE |
| 0 | 3 | male | NaN | 0 | 0 | 8.4583 | Q | Third | man | TRUE | NA | Queenstown | no | TRUE |
| 0 | 1 | male | 54 | 0 | 0 | 51.8625 | S | First | man | TRUE | E | Southampton | no | TRUE |
| 0 | 3 | male | 2 | 3 | 1 | 21.0750 | S | Third | child | FALSE | NA | Southampton | no | FALSE |
| 1 | 3 | female | 27 | 0 | 2 | 11.1333 | S | Third | woman | FALSE | NA | Southampton | yes | FALSE |
| 1 | 2 | female | 14 | 1 | 0 | 30.0708 | C | Second | child | FALSE | NA | Cherbourg | yes | FALSE |
| 1 | 3 | female | 4 | 1 | 1 | 16.7000 | S | Third | child | FALSE | G | Southampton | yes | FALSE |
| 1 | 1 | female | 58 | 0 | 0 | 26.5500 | S | First | woman | FALSE | C | Southampton | yes | TRUE |
| 0 | 3 | male | 20 | 0 | 0 | 8.0500 | S | Third | man | TRUE | NA | Southampton | no | TRUE |
| 0 | 3 | male | 39 | 1 | 5 | 31.2750 | S | Third | man | TRUE | NA | Southampton | no | FALSE |
| 0 | 3 | female | 14 | 0 | 0 | 7.8542 | S | Third | child | FALSE | NA | Southampton | no | TRUE |

Nous voyons en colonne 1, le vecteur de toutes les réponses, i.e. est - ce que le passager a survécu ou non. Chaque ligne représente une observation i.e. un passager. Viennent ensuite les caractéristiques de chaque passager représenté chacune dans une colonne. L'objectif est de prédire les survivants du titanic c'est - à - dire expliquer le contenu de la colonne 1 à partir des vecteurs des colonnes suivantes.

Toute l'information utile à la résolution de ce problème mathématique est donc contenu dans la matrice des features.

**En résumé**

— La préparation des données commence par une phase de compréhension des données, d'analyse descriptive et de nettoyage.
— La deuxième phase s'appelle le « feature engineering » : mise au format, imputation,imputation, sélection des features.
— La base de modélisation est réalisée par jointure des différentes bases de données utiles pour la modélisation. Trois bases sont réalisées : la base de calibrage, la base de cross - validation et la base de test.

## 3.2 Illustration sur la base German Credit Data

Nous avons vu dans la précédente section les principales étapes pour la préparation des données et la mise en place de la base de modélisation. Dans cette section, nous allons illustrer sur la base de German Credit Data la phase d'exploitation des données.

### 3.2.1 Description de la base de données German Credit Data

German Credit Data est une base de données de crédits à la consommation allemands. Il y a plusieurs versions de cette base de données : kaggle, UCI database et MIT open source repository.

La base que nous avons utiliser pour cette section est celle du MIT https ://ocw.mit.edu. Un exemple d'analyse de données de cette base est disponible sur le lien https ://online.stat.psu.edu/stat857/node/215/.

German Credit Data est une base de petite taille : 1000 lignes, i.e. 1000 emprunteurs. Il y a 32 colonnes dont 30 colonnes pour les features, 1 colonne pour la fonction réponse (defaut ou survie) et une colonne d'identifiant pour chacun des emprunteurs mais qui n'aura aucune utilité dans le modèle de score.

#### 3.2.1.1 Chargement des données

Nous avons chargé la base de données disponible le site du MIT. Cette base contient 32 colonnes et 1000 lignes.

Pour charger les données, nous utilisons la fonction pd.read_excel qui permet d'obtenir un dataframe que nous appellons « data_german ».

```
# Chargement des données
import pandas as pd
data_german = pd.read_excel("./donnee/GermanCreditMIT_full.xls")
print(data_german.info())
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 1000 entries, 0 to 999
## Data columns (total 32 columns):
##  #   Column            Non-Null Count  Dtype
## ---  ------            --------------  -----
##  0   OBS#              1000 non-null   int64
##  1   CHK_ACCT          1000 non-null   int64
##  2   DURATION          1000 non-null   int64
##  3   HISTORY           1000 non-null   int64
##  4   NEW_CAR           1000 non-null   int64
##  5   USED_CAR          1000 non-null   int64
##  6   FURNITURE         1000 non-null   int64
##  7   RADIO/TV          1000 non-null   int64
##  8   EDUCATION         1000 non-null   int64
##  9   RETRAINING        1000 non-null   int64
##  10  AMOUNT            1000 non-null   int64
##  11  SAV_ACCT          1000 non-null   int64
##  12  EMPLOYMENT        1000 non-null   int64
##  13  INSTALL_RATE      1000 non-null   int64
##  14  MALE_DIV          1000 non-null   int64
##  15  MALE_SINGLE       1000 non-null   int64
##  16  MALE_MAR_or_WID   1000 non-null   int64
##  17  CO-APPLICANT      1000 non-null   int64
##  18  GUARANTOR         1000 non-null   int64
##  19  PRESENT_RESIDENT  1000 non-null   int64
##  20  REAL_ESTATE       1000 non-null   int64
##  21  PROP_UNKN_NONE    1000 non-null   int64
##  22  AGE               1000 non-null   int64
##  23  OTHER_INSTALL     1000 non-null   int64
##  24  RENT              1000 non-null   int64
##  25  OWN_RES           1000 non-null   int64
##  26  NUM_CREDITS       1000 non-null   int64
##  27  JOB               1000 non-null   int64
##  28  NUM_DEPENDENTS    1000 non-null   int64
##  29  TELEPHONE         1000 non-null   int64
##  30  FOREIGN           1000 non-null   int64
##  31  RESPONSE          1000 non-null   int64
## dtypes: int64(32)
## memory usage: 250.1 KB
```

```
## None
```

La dimension de ce dataframe est donnée par `data.shape` qui retourne une liste.

```
# Ddimension du dataset
data_german.shape
```

```
## (1000, 32)
```

Le nombre de lignes est donné par le premier élément de la liste

```
# nombre d'observations
data_german.shape[0]
```

```
## 1000
```

#### 3.2.1.2 Nombre de modalités

Pour avoir le nombre d'élements distincts dans chaque colonne, on utilise la fonction nunique de pandas.

```
# Nombre de modalités par colonnes
data_german.nunique()
```

```
## OBS#                1000
## CHK_ACCT               4
## DURATION              33
## HISTORY                5
## NEW_CAR                2
## USED_CAR               2
## FURNITURE              2
## RADIO/TV               2
## EDUCATION              2
## RETRAINING             2
## AMOUNT               921
## SAV_ACCT               5
## EMPLOYMENT             5
## INSTALL_RATE           4
## MALE_DIV               2
## MALE_SINGLE            2
## MALE_MAR_or_WID        2
## CO-APPLICANT           2
## GUARANTOR              2
## PRESENT_RESIDENT       4
## REAL_ESTATE            2
## PROP_UNKN_NONE         2
## AGE                   53
## OTHER_INSTALL          2
## RENT                   2
```

```
## OWN_RES                2
## NUM_CREDITS            4
## JOB                    4
## NUM_DEPENDENTS         2
## TELEPHONE              2
## FOREIGN                2
## RESPONSE               2
## dtype: int64
```

#### 3.2.1.3 Sélection des variables

L'attribut `.columns` permet d'obtenir les noms des différentes features correspondantes à chaque colonne du dataframe.

```
# nom des colonnes
data_german.columns
```

```
## Index(['OBS#', 'CHK_ACCT', 'DURATION', 'HISTORY', 'NEW_CAR', 'USED_CAR',
##        'FURNITURE', 'RADIO/TV', 'EDUCATION', 'RETRAINING', 'AMOUNT',
##        'SAV_ACCT', 'EMPLOYMENT', 'INSTALL_RATE', 'MALE_DIV', 'MALE_SINGLE',
##        'MALE_MAR_or_WID', 'CO-APPLICANT', 'GUARANTOR', 'PRESENT_RESIDENT',
##        'REAL_ESTATE', 'PROP_UNKN_NONE', 'AGE', 'OTHER_INSTALL', 'RENT',
##        'OWN_RES', 'NUM_CREDITS', 'JOB', 'NUM_DEPENDENTS', 'TELEPHONE',
##        'FOREIGN', 'RESPONSE'],
##       dtype='object')
```

Nous pouvons faire une sélection dans ces colonnes si on ne veut pas toutes les garder pour la modélisation. A titre d'exemple, on définit un vecteur avec les numéros des colonnes que l'on souhaite garder.

```
# sélection des colonnes
c = [1,2,3,10,11,13,31]
data_german2 = data_german.iloc[:,c]
```

Pour rétirer la dernière colonne au dataframe, on applique :

```
# Retier la dernière colonne
data_german3 = data_german.iloc[:,:-1]
data_german3.columns
```

```
## Index(['OBS#', 'CHK_ACCT', 'DURATION', 'HISTORY', 'NEW_CAR', 'USED_CAR',
##        'FURNITURE', 'RADIO/TV', 'EDUCATION', 'RETRAINING', 'AMOUNT',
##        'SAV_ACCT', 'EMPLOYMENT', 'INSTALL_RATE', 'MALE_DIV', 'MALE_SINGLE',
##        'MALE_MAR_or_WID', 'CO-APPLICANT', 'GUARANTOR', 'PRESENT_RESIDENT',
##        'REAL_ESTATE', 'PROP_UNKN_NONE', 'AGE', 'OTHER_INSTALL', 'RENT',
##        'OWN_RES', 'NUM_CREDITS', 'JOB', 'NUM_DEPENDENTS', 'TELEPHONE',
##        'FOREIGN'],
##       dtype='object')
```

On remarque que la colonne « RESPONSE » a disparu.

#### 3.2.1.4 Signification des variables

Nous pouvons vérifier la liste des colonnes sélectionner dans le deuxième dataframe

```
# nom des colonnes - dataset2
data_german2.columns
```

```
## Index(['CHK_ACCT', 'DURATION', 'HISTORY', 'AMOUNT', 'SAV_ACCT', 'INSTALL_RATE',
##        'RESPONSE'],
##       dtype='object')
```

Signification des variables :

- `CHK_ACCOUNT` - situation du compte courant : 4 modalités (découvert, $< 200$ DM, $> 200$ DM, pas de compte)
- `DURATION` - durée du crédit en mois
- `HISTORY` - historique crédit de l'emprunteur : 5 modalités (No crédit taken, all crédits paid back duly,...,critical account)
- `AMOUNT` - Montant de l'encours de crédit
- `SAV_ACCT` - Solde moyen du compte d'épargne : 5 modalités ($< 100$ DM, $< 500$ DM, $< 1000$ DM, $> 1000$ DM, pas de compte d'épargne)
- `INTALL_RATE` - Ratio d'endettement ($> 35\%$, $25 - 35\%$, $20 - 25\%$, $< 20\%$).

Pour plus d'informations sur la signification des variables, visitez le site https ://archive.ics.uci.edu/ml/datasets/statlog+(german+credit+data).

#### 3.2.1.5 Défauts (variable de réponse)

La base d'observation est quant à elle relativement équilibrée c'est - à - dire que la variable `RESPONSE` indique qu'il y a eu un defaut (`RESPONSE=0`) ou non (`RESPONSE=1`).

```
# Repartition des defauts et non défauts
n_k = data_german.RESPONSE.value_counts().to_frame()
```

**Table 3.2 –** Effectifs - Creditability

| | RESPONSE |
|---|---|
| 1 | 700 |
| 0 | 300 |

Le taux de défaut est de $30\%$.

```
# Taux de défauts
p_k = data_german.RESPONSE.value_counts(normalize=True).to_frame()
```

**Table 3.3 –** Proportions - Creditability

| | RESPONSE |
|---|---|
| 1 | 0.7 |
| 0 | 0.3 |

### 3.2.2 Exploration des données

Voici à présent les grandes étapes de l'exploration des données. Il y a l'analyse des valeurs manquantes, le calcul et l'analyse des grandeurx statistiques (moyenne, médiane, extrêmes). L'objectif est de déterminer si ces distributions sont symétriques ou au contraire ont des queues epaisses (skew des données). C'est aussi d'identifier les doublons dans la base des données ou d'identifier encore les outliers, les valeurs abérrantes pour pouvoir les corriger.

Les doublons, on doit les supprimer. Les outliers, on doit les supprimer ou les corriger.

#### 3.2.2.1 Premières lignes de la base

La fonction head() permet de voir ce qui se passe dans les premières lignes de la base de données.

```
# 5 premières observations
print(data_german2.head())
```

```
##    CHK_ACCT  DURATION  HISTORY  AMOUNT  SAV_ACCT  INSTALL_RATE  RESPONSE
## 0         0         6        4    1169         4             4         1
## 1         1        48        2    5951         0             2         0
## 2         3        12        4    2096         0             2         1
## 3         0        42        2    7882         0             2         1
## 4         0        24        3    4870         0             3         0
```

#### 3.2.2.2 Statistiques élémentaires

La fonction describe permet d'avoir les premiers indicateurs statistiques pour chacune des variables de notre dataset.

```
# Indicateurs statistiques
stats =  data_german2.describe(include="all").T
```

**Table 3.4 –** Statistiques descriptives

| | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| CHK_ACCT | 1000 | 1.577 | 1.2576377 | 0 | 0.0 | 1.0 | 3.00 | 3 |
| DURATION | 1000 | 20.903 | 12.0588145 | 4 | 12.0 | 18.0 | 24.00 | 72 |
| HISTORY | 1000 | 2.545 | 1.0831196 | 0 | 2.0 | 2.0 | 4.00 | 4 |
| AMOUNT | 1000 | 3271.258 | 2822.7368760 | 250 | 1365.5 | 2319.5 | 3972.25 | 18424 |
| SAV_ACCT | 1000 | 1.105 | 1.5800226 | 0 | 0.0 | 0.0 | 2.00 | 4 |
| INSTALL_RATE | 1000 | 2.973 | 1.1187147 | 1 | 2.0 | 3.0 | 4.00 | 4 |
| RESPONSE | 1000 | 0.700 | 0.4584869 | 0 | 0.0 | 1.0 | 1.00 | 1 |

Pour la variable AMOUNT, nous voyons ici qu'elle est comprise entre 250 et 18424 DM. Cela ne revèle pas d'anomalie a priori en terme de qualité de données sur cette variable. Ces ordres de grandeurs sont valables pour du crédit à la consommation, même si une analyse plus détaillée est nécessaire pour conclure formellement. Nous pouvons lire la médiane cette variable qui est de 2319.5 DM et la moyenne de 3271 DM.

#### 3.2.2.3 Valeurs manquantes

Regardons à présent les valeurs manquantes. La fonction apply() sert à appliquer un traitement global à toute la base des données, c'est - à - dire de compter toutes les données manquantes de chacune des colonnes de la base. Nous utilisons pour cela la fonction is.na() qui renvoie un booléen lorsque la valeur est manquante et en faisant la somme de tout cela, nous pouvons compter le nombre de valeurs manquantes.

```
# Valeurs manquantes
print(data_german.apply(lambda x : x.isna().sum(),axis=0))
```

```
## OBS#                0
## CHK_ACCT            0
## DURATION            0
## HISTORY             0
## NEW_CAR             0
## USED_CAR            0
## FURNITURE           0
## RADIO/TV            0
## EDUCATION           0
## RETRAINING          0
## AMOUNT              0
## SAV_ACCT            0
## EMPLOYMENT          0
## INSTALL_RATE        0
## MALE_DIV            0
## MALE_SINGLE         0
## MALE_MAR_or_WID     0
## CO-APPLICANT        0
## GUARANTOR           0
## PRESENT_RESIDENT    0
## REAL_ESTATE         0
## PROP_UNKN_NONE      0
## AGE                 0
## OTHER_INSTALL       0
## RENT                0
## OWN_RES             0
## NUM_CREDITS         0
## JOB                 0
## NUM_DEPENDENTS      0
## TELEPHONE           0
## FOREIGN             0
## RESPONSE            0
## dtype: int64
```

Ce que nous voyons c'est qu'il n'y a pas de valeurs manquantes dans la base de données. C'est un cas plutôt exceptionel car les bases de données sont souvent imparfaites. Elles contiennent des erreurs et des données manquantes.

#### 3.2.2.4 Distribution univariée des variables

La visualisation des distributions univariées est souvent utile pour découvrir le contenu des bases de données et pour comprendre la signification de chacune des données.

```
# Représentation graphique
import matplotlib.pyplot as plt
fig, axe = plt.subplots(figsize=(16,10))
data_german[["INSTALL_RATE","AMOUNT","DURATION"]].hist(color="steelblue",ax=axe);
fig.tight_layout();
plt.show()
```

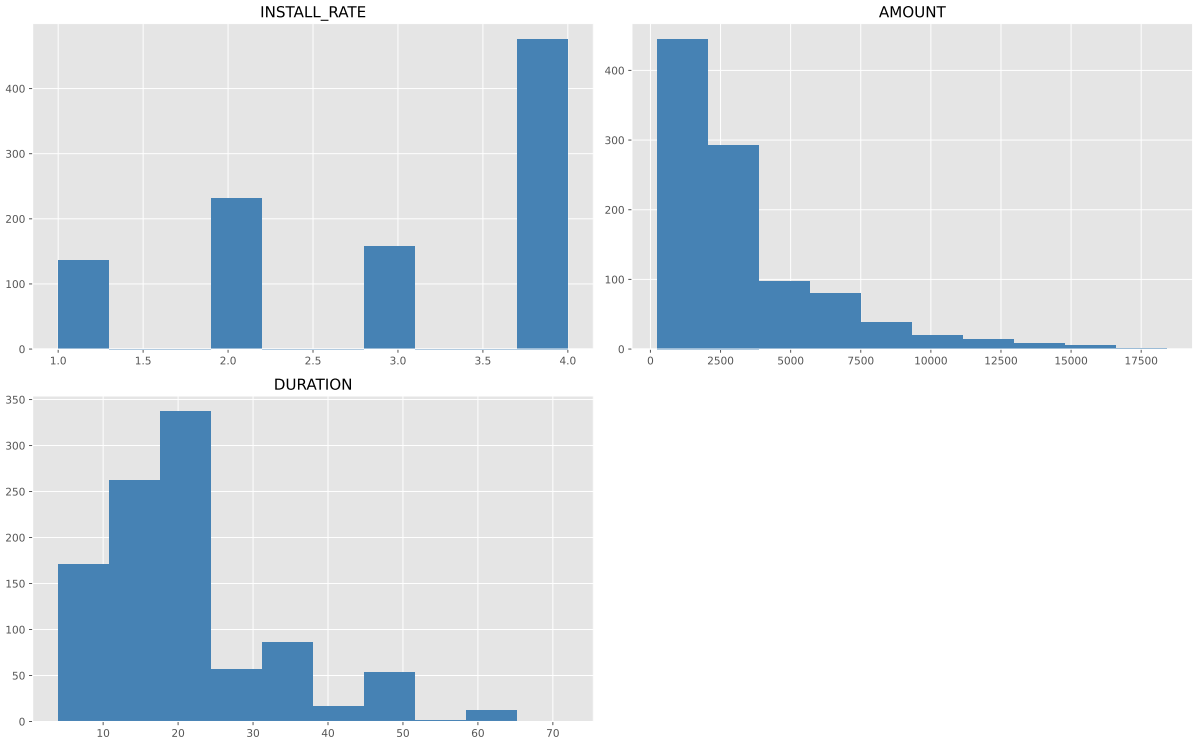


**Figure 3.2 –** Distribution univariées des variables

Voici les distributions univariées sur trois des variables de la base : la variable `INSTALL_RATE` pour laquelle il y a 4 modalités, la variable `AMOUNT` et la variable `DURATION`. Ce type d'histogramme permet de prendre connaissance des répartitions des populations selon les différentes caractéristiques du portefeuille et de vérifier de visu s'il y a des anomalies dans la base de données.

Nous pouvons faire des distributions conditionnelles c'est - à - dire conditionnées à la survue et au défaut de ces mêmes variables pour déterminer si oui ou non il y a une différence visible dans la distribution des variables conditionnellement au défaut ou conditionnellement à la survie.

```
from plotnine import *
Data = data_german.copy()
# INSTALL_RATE
Data["RESPONSE"] = Data["RESPONSE"].map({0:"Défauts",1:"Survivants"})
```

```
p1 = (ggplot(Data,aes(x="INSTALL_RATE"))+
        geom_histogram(fill="steelblue")+facet_wrap("RESPONSE"))
print(p1)
```

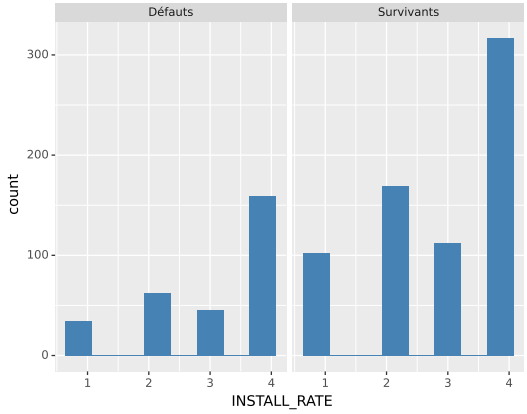


**Figure 3.3 –** Distribution univariée conditionelle de INSTALL RATE | RESPONSE

Concernant la première variable INSTALL_RATE, il semble que c'est car même assez proche entre le défaut et la survie. Visuellement, il semble que quatrième modalité soit légèrement surreprésenté par rapport aux autres modalités sur les prêts en défaut.

```
# AMOUNT
p2 = (ggplot(Data,aes(x="AMOUNT"))+
        geom_histogram(fill="steelblue")+facet_wrap("RESPONSE"))
print(p2)
```

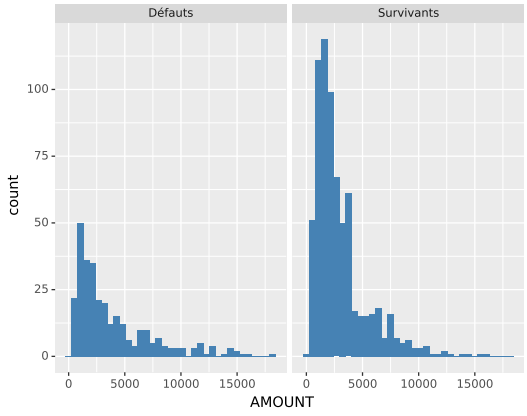


**Figure 3.4 –** Distribution univariée conditionnelle de AMOUNT | RESPONSE

Concernant la variable AMOUNT, on voit plus clairement dans les queues de distribution sur ces histogrammes, que les montants très élevés correspondent plus souvent à des situations de défaut qu'à des situations de survie.

```
# DURATION
p3 = (ggplot(Data,aes(x="DURATION"))+
        geom_histogram(fill="steelblue")+facet_wrap("RESPONSE"))
print(p3)
```

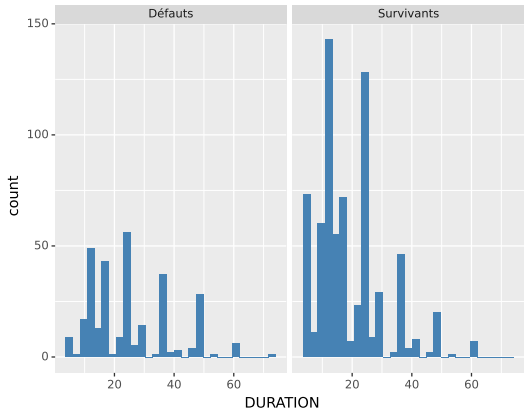


**Figure 3.5 –** Distribution univariée conditionelle de DURATION | RESPONSE

Et concernant la durée du crédit, DURATION, on voit aussi que les durées élevées vont engendrer probablement plus de défaut que les durées faibles. Et nous le voyons aussi ici sur la première modalité, sur les horizons de maturité court.

#### 3.2.2.5 Corrélations

Nous pouvons aussi représenter les variables dans des diagrammes à deux dimensions afin de visualiser les éventuelles corrélations entre les variables du dataset.

```
# Représentation graphique
pscatter = (ggplot(Data,aes(x="AMOUNT",y="DURATION"))+
              geom_point(aes(colour ="RESPONSE"))+
              scale_colour_manual(values =["blue","red"])+
              theme(legend_direction="vertical",legend_position=[0.8,0.33]))
print(pscatter)
```

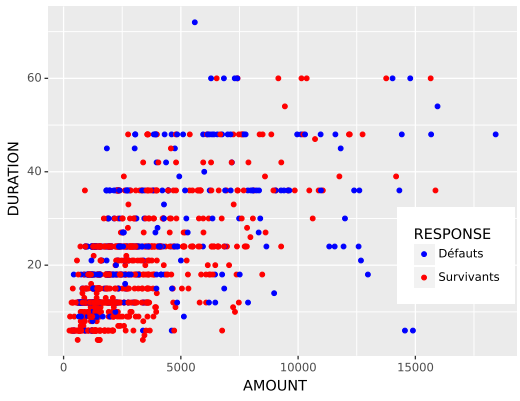


**Figure 3.6 –** Nuage de point entre AMOUNT et DURATION (coloré par RESPONSE)

Nous avons fait un graphique ici des populations de la base des données selon les axes « DURATION » et « AMOUNT ». Nous avons représenté en bleu les événements de défaut et en rouge les événements de survie. Pour les durées élevées, il y a en effet plus de bleu que de rouge par rapport aux durées courtes et également pour des montants

élevés par rapport aux montants moins élevés. Ainsi, durée de crédit élevé ou montant de crédit élevé sont des facteurs du risque de défaut.

**En résumé**

— Collecte et exploration des données sont les premières étapes fondamentales pour pouvoir construire un projet de machine learning et de modélisation.
— Bien évidemment, les statistiques univariées (nombre de modalités, intervalles de valeurs, distributions des valeurs, etc.) permettent d'explorer les données.
— Les liens de corrélation entre les variables avec la valeur de la réponse à modéliser sont aussi un élément très important dans l'appréciation et dans l'identification des features que nous allons également garder dans le modèle lorsqu'on va concevoir le dispositif de prédiction.

## 3.3 Traitement des données manquantes

Dans la section précédente, nous avons décrit les outils d'exploration des données. A présent dans cette section nous allons nous focaliser sur les traitements des données manquantes.

### 3.3.1 La base Give Me Somme Credit en bref

Nous allons pour cela utiliser la base `Give Me Somme Credit`. C'est une base de données de crédits à la consommation issus d'un concours kaggle datant de 2011. Nous pouvons trouver le jeu de données directement sur le lien suivant https ://www.kaggle.com/c/GiveMeSomeCredit

La taille de cette base est de 150 000 lignes pour le training et une dizaine de colonnes de features plus une colonne de réponse (défaut ou pas défaut) ainsi qu'une colonne d'identifiant, soit au total 12 colonnes.

La fonction read_csv permet de charger le jeu de données dans un dataframe que nous appelons `data_GMSC`.

```
# Chargement des données
data_GMSC = pd.read_csv("./donnee/give-me-some-credit.csv",sep=",")
# Inspection
data_GMSC.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 150000 entries, 0 to 149999
## Data columns (total 12 columns):
##  #   Column                                Non-Null Count   Dtype
## ---  ------                                --------------   -----
##  0   Unnamed: 0                            150000 non-null  int64
##  1   SeriousDlqin2yrs                      150000 non-null  int64
##  2   RevolvingUtilizationOfUnsecuredLines  150000 non-null  float64
##  3   age                                   150000 non-null  int64
```

```
##  4   NumberOfTime30-59DaysPastDueNotWorse  150000 non-null  int64
##  5   DebtRatio                             150000 non-null  float64
##  6   MonthlyIncome                         120269 non-null  float64
##  7   NumberOfOpenCreditLinesAndLoans       150000 non-null  int64
##  8   NumberOfTimes90DaysLate               150000 non-null  int64
##  9   NumberRealEstateLoansOrLines          150000 non-null  int64
##  10  NumberOfTime60-89DaysPastDueNotWorse  150000 non-null  int64
##  11  NumberOfDependents                    146076 non-null  float64
## dtypes: float64(4), int64(8)
## memory usage: 13.7 MB
```

```
# Dimension du tableau
data_GMSC.shape
```

```
## (150000, 12)
```

La fonction `shape`, appliquée à ce dataframe, permet de montrer qu'il y a bien 150000 lignes et 12 colonnes dans ce dataframe.

```
# Nombre d'observations du tableau
data_GMSC.shape[0]
```

```
## 150000
```

La fonction `columns` permet d'avoir la liste des noms de colonnes dans cette base de données.

```
# Nom des colonnes
data_GMSC.columns
```

```
## Index(['Unnamed: 0', 'SeriousDlqin2yrs',
##        'RevolvingUtilizationOfUnsecuredLines', 'age',
##        'NumberOfTime30-59DaysPastDueNotWorse', 'DebtRatio', 'MonthlyIncome',
##        'NumberOfOpenCreditLinesAndLoans', 'NumberOfTimes90DaysLate',
##        'NumberRealEstateLoansOrLines', 'NumberOfTime60-89DaysPastDueNotWorse',
##        'NumberOfDependents'],
##       dtype='object')
```

La première colonne correspond à l'identifiant, la deuxième colonne correspond à la colonne `reponse`, c'est - à - dire défaut ou survit. Et enfin, les 10 colonnes suivantes sont les *features* de notre base données.

#### 3.3.1.1 Visualisation des données manquantes sur la base Give Me Some Credit

La visualisation des données manquantes s'obtient en appliquant la ligne de code suivante :

```
# Nombre de valeurs manquantes
data_GMSC.apply(lambda x : x.isna().sum(),axis=0)
```

```
## Unnamed: 0                                0
## SeriousDlqin2yrs                          0
## RevolvingUtilizationOfUnsecuredLines      0
## age                                       0
## NumberOfTime30-59DaysPastDueNotWorse      0
## DebtRatio                                 0
## MonthlyIncome                         29731
## NumberOfOpenCreditLinesAndLoans           0
## NumberOfTimes90DaysLate                   0
## NumberRealEstateLoansOrLines              0
## NumberOfTime60-89DaysPastDueNotWorse      0
## NumberOfDependents                     3924
## dtype: int64
```

Nous observons que pour la variable *MonthlyIncome*, il y a 29731 valeurs manquantes. Et pour la variable *NumberOfDependents*, il y a 3924. Sur les autres colonnes de ce dataset, il n'y a aucune valeur manquante.

#### 3.3.1.2 Suppression d'une colonne

Une des solutions pour traiter le problème des données manquantes, c'est de ne pas utiliser la *feature* sur laquelle il y a des données manquantes, c'est - à - dire de supprimer purement et simplement la colonne de la base de données.

Cette ligne de code permet de résoudre le problème.

```
# Suppression de colonnes
data_GMSC_2 = data_GMSC
data_GMSC_2 = data_GMSC_2.drop(columns=["MonthlyIncome"])
```

La donnée `MonthlyIncome` n'est plus dans la base :

```
# Noms des colonnes
data_GMSC_2.columns
```

```
## Index(['Unnamed: 0', 'SeriousDlqin2yrs',
##        'RevolvingUtilizationOfUnsecuredLines', 'age',
##        'NumberOfTime30-59DaysPastDueNotWorse', 'DebtRatio',
##        'NumberOfOpenCreditLinesAndLoans', 'NumberOfTimes90DaysLate',
##        'NumberRealEstateLoansOrLines', 'NumberOfTime60-89DaysPastDueNotWorse',
##        'NumberOfDependents'],
##       dtype='object')
```

Nous voyons que la variable *MonthlyIncome* ne fait plus partie de la liste des variables du dataframe.

#### 3.3.1.3 Suppression des lignes contenant une valeur manquante

Nous pouvons également supprimer des lignes qui contiennent des valeurs manquantes. Ceci est réalisé par sélection des lignes sur lesquelles il n'y a pas de données manquantes via la fonction notna de pandas notamment sur la variable *MonthlyIncome* et la variable *NumberOfDependents*.

```
data_GMSC_3 = data_GMSC[data_GMSC["MonthlyIncome"].notna()]
data_GMSC_4 = data_GMSC[data_GMSC["NumberOfDependents"].notna()]
data_GMSC_5 = data_GMSC_3[data_GMSC_3["NumberOfDependents"].notna()]
```

Dans notre exemple, `data_GMSC_3` est le sous-ensemble du dataset initial pour lequel il n'y a pas de données manquantes sur la variable *MonthlyIncome*.

Il en est de même sur le sous - dataset `data_GMSC_4`. Il s'agit d'un sous - dataset du dataset initial sur lequel il n'y a pas de valeurs manquantes sur la variable *NumberOfDependents*.

Enfin, le dataset `data_GMSC_5` est issu du dataset `data_GMSC_3`, c'est - à - dire celui sur lequel nous avons déjà filtré toutes les lignes qui contenaient des valeurs manquantes sur *MonthlyIncome*, et nous filtrons ensuite les lignes qui contiennent des données manquantes sur *NumberOfDependents*.

Nous voyons que le dataset initial a 150000.

```
# nombre d'observations
data_GMSC.shape[0]
```

```
## 150000
```

Le dataset `data_GMSC_3` a 120269. L'écart par rapport au dataset initial étant le nombre de valeurs manquantes sur *MonthlyIncome*.

```
# Nombre d'observations
data_GMSC_3.shape[0]
```

```
## 120269
```

Le dataset `data_GMSC_4` a 146076 L'écart par rapport au dataset initial étant le nombre de données manquantes sur *NumberOfDependents*.

```
# nombre d'observations
data_GMSC_4.shape[0]
```

```
## 146076
```

Enfin le dataset `data_GMSC_5` est issu du dataset `data_GMSC` en supprimant toutes les lignes qui ont à la fois des données manquantes sur *NumberOfDependents* et *MonthlyIncome*.

```
# nombre d'observations
data_GMSC_5.shape[0]
```

```
## 120269
```

Nous constatons que les datasets `data_GMSC_3` et `data_GMSC_5` ont le même nombre de lignes, ce qui signifie que toutes les lignes sur lesquelles on a des données manquantes sur *NumberOfDependents* correspondent aussi à des données manquantes sur la variable *MonthlyInciome*. Peut être qu'il existe un lien de cause à effet entre les données manquantes sur ces deux variables.

#### 3.3.1.4 Traitement des données manquantes

A présent, comment peut - on vérifier qu'après traitement et suppression des lignes sur lesquelles il y a des données manquantes, il n'y a plus de données manquantes ?

```
# nombre de valeurs manquantes
import numpy as np
data_GMSC_5.apply(lambda x : np.sum(np.isnan(x)),0)
```

```
## Unnamed: 0                            0
## SeriousDlqin2yrs                      0
## RevolvingUtilizationOfUnsecuredLines  0
## age                                   0
## NumberOfTime30-59DaysPastDueNotWorse  0
## DebtRatio                             0
## MonthlyIncome                         0
## NumberOfOpenCreditLinesAndLoans       0
## NumberOfTimes90DaysLate               0
## NumberRealEstateLoansOrLines          0
## NumberOfTime60-89DaysPastDueNotWorse  0
## NumberOfDependents                    0
## dtype: int64
```

C'est bien le cas lorsque nous regardons sur la base `data_GMSC_5`. Nous trouvons $0$ sur toutes les colonnes.

### 3.3.2 Remplacer les valeurs manquantes

Passons à présent à l'imputation. Il s'agit d'une méthode qui consiste à remplacer les valeurs manquantes par un nombre issu soit d'une moyenne, soit d'une médiane, soit une valeur par défaut, soit encore une valeur modélisée. L'imputation est généralement préférable à la suppression des lignes et des colonnes telle qu'illustrée précédemment.

#### 3.3.2.1 Remplacer les valeurs manquantes par la moyenne de la colonne

Dans le dataset `data_GMSC_6`, nous remplaçons les valeurs manquantes

```
# Remplacement des valeurs manquantes par la moyenne
data_GMSC_6 = data_GMSC
data_GMSC_6["MonthlyIncome"] = data_GMSC_6["MonthlyIncome"].fillna(
                               value = np.nanmean(data_GMSC_6["MonthlyIncome"]))
data_GMSC_6.MonthlyIncome.mean()
```

```
## 6670.221237392847
```

```
# Compter les valeurs manquantes
data_GMSC_6.isna().sum()
```

```
## Unnamed: 0                                0
## SeriousDlqin2yrs                          0
## RevolvingUtilizationOfUnsecuredLines      0
## age                                       0
## NumberOfTime30-59DaysPastDueNotWorse      0
## DebtRatio                                 0
## MonthlyIncome                             0
## NumberOfOpenCreditLinesAndLoans           0
## NumberOfTimes90DaysLate                   0
## NumberRealEstateLoansOrLines              0
## NumberOfTime60-89DaysPastDueNotWorse      0
## NumberOfDependents                     3924
## dtype: int64
```

Ainsi, après imputation, lorsque nous calculons le nombre de données manquantes sur *MonthyIncome*, nous devons obtenir 0, ce qui est bien le cas.

SUr la variable *NumberOfDependents*, nous n'avons effectué aucun traitement, par conséquent, le nombre de données manquantes reste inchangé.

#### 3.3.2.2 Remplacer les valeurs manquantes par la médiane de la colonne

Plutôt que de faire l'imputation par la moyenne, nous pouvons également imputer par un autre indicateur, la médiane de la colonne par exemple.

```
# remplacer les valeurs manquantes par la médiane
data_GMSC_7 = data_GMSC
data_GMSC_7["MonthlyIncome"] = data_GMSC_7["MonthlyIncome"].fillna(
                               value = np.nanmedian(data_GMSC_7["MonthlyIncome"]))
data_GMSC_7.MonthlyIncome.median()
```

```
## 6600.0
```

```
# compter les valeurs manquantes
data_GMSC_7.isna().sum()
```

```
## Unnamed: 0                                0
```

```
## SeriousDlqin2yrs                         0
## RevolvingUtilizationOfUnsecuredLines     0
## age                                      0
## NumberOfTime30-59DaysPastDueNotWorse     0
## DebtRatio                                0
## MonthlyIncome                            0
## NumberOfOpenCreditLinesAndLoans          0
## NumberOfTimes90DaysLate                  0
## NumberRealEstateLoansOrLines             0
## NumberOfTime60-89DaysPastDueNotWorse     0
## NumberOfDependents                    3924
## dtype: int64
```

Ici encore le nombre de *MonthlyIncome* manquant est égal à $0$ après imputation.

#### 3.3.2.3 Remplacer les valeurs manquantes par une modalité dédiée

Une autre méthode consiste aussi à remplacer les valeurs manquantes par une modalité dédiée, soit une valeur par défaut autre la moyenne ou la médiane, soit de dire qu'il s'agit d'une modalité différente des autres. Par exemple, nous appelons cette modalité $-1$ ici. Ainsi, nous transformons la variable *MonthlyIncome* en une variable catégorielle en ajoutant une modalité qui est la valeur manquante.

```
# remplacer les valeurs manquantes par -1
data_GMSC_8 = data_GMSC
data_GMSC_8["MonthlyIncome"] = data_GMSC_8["MonthlyIncome"].fillna(value=-1)
data_GMSC_8.isna().sum()
```

```
## Unnamed: 0                               0
## SeriousDlqin2yrs                         0
## RevolvingUtilizationOfUnsecuredLines     0
## age                                      0
## NumberOfTime30-59DaysPastDueNotWorse     0
## DebtRatio                                0
## MonthlyIncome                            0
## NumberOfOpenCreditLinesAndLoans          0
## NumberOfTimes90DaysLate                  0
## NumberRealEstateLoansOrLines             0
## NumberOfTime60-89DaysPastDueNotWorse     0
## NumberOfDependents                    3924
## dtype: int64
```

Là encore après imputation, nous obtenons bien $0$ valeurs manquantes sur *MonthlyIncome*.

Dans tous ces exemples, les lignes de code sont toutes très similaires et assez explicites.

#### 3.3.2.4 Remplacer les valeurs manquantes par un plus proche voisin

Evoquons les méthodes plus sophistiquées pour imputer les données. Nous pouvons utilisés les modèles et notamment chercher une valeur qui serait assez proche de ce que les voisins de notre emprunteur, dans l'espace des paramètres et des *features*, auraient. Quelle serait la valeur qu'on pourrait imputer ?

Le tableau suivant présente les quelques méthodes avec leurs avantages et leurs inconvénients :

**Table 3.5 –** Avantages et inconvénients de quelques méthodes d'imputation

| Méthodes | Avantage | Inconvénient |
|---|---|---|
| Régression | Simplicité | Variance réduite |
| Inputation KNN | Impute les données catégoriques et numériques | Performance sur les ensembles de données volumineux |
| Imputation multiple | Réduit l'erreur due à l'imputation | Complexité |

**En résumé**

— Les données peuvent être manquantes pour des raisons aléatoires ou non aléatoires. Le traitement des données manquantes inclut des approches sommaires et plus sophistiquées
— Suppression de features, de lignes
— Imputation basique (remplacement par une valeur par défaut)
— Imputation plus sophistiquée (création d'une modalité dédiée, imputation à l'aide d'un modèle)

## 3.4 Préparation de la base de modélisation

Dans les deux sections précédentes, nous avons parlé de l'exploration des données et du traitement des données manquantes. A présent, nous abordons la dernière phase : la préparation de la base de modélisation.

### 3.4.1 Retirer les caractères spéciaux

La préparation de la base de modélisation consiste à préparer les données afin qu'elle soit directement exploitable par un modèle de machine learning, c'est - à - dire par des routines informatiques effectuant des calculs mathématiques.

**Exemple 3.2.** *Passer de "8,684 EUR" à 8684*

L'exemple suivant est celui du retrait des caractères spéciaux. Si la base de données contient des chaînes de caractère désignant des montants avec leur dévise, il faut les traduire en nombre exploitable par un algorithme ou par un modèle. Cela se passe en trois étapes :

— **Etape 1 :** Suppression de la virgule

Cela est effectué par la fonction replace pour rechercher les virgules. Cette ligne de code recherche la virgule dans la chaîne de caractère x et la remplace par un vide.

```
# Suppression de la virgule
x = "8,684 EUR"
y = x.replace(',', '')
y
```

```
## '8684 EUR'
```

Nous avons une chaîne de caractère dans laquelle nous avons retiré la virgule.

— **Extape 2 :** Suppression du blanc et de la dévise.

Nous effectuons le même traitement que précédemment en supprimant le blanc et la dévise et remplaçant par le vide.

```
# supprimer l'espace
z = y.replace(" EUR","")
z
```

```
## '8684'
```

Nous obtenons une chaînes de charactères avec un nombre.

— **Etape 3 :** Transformer la chaîne de charactères en nombre

Elle consiste à appliquer la fonction float (ou int) afin d'obtenir un nombre réel (ou un entier).

```
# convertion en numéric
float(z)
```

```
## 8684.0
```

### 3.4.2 Encoder une variable catégorielle en "factor"

Une autre opération indispensable également lorsqu'on fait un modèle, c'est de s'assurer que les variables catégorielles sont encodées dans le type `category`.

Pourquoi le faire ? Parce que si une variable catégorielle à trois modalités, par exemple les valeurs 1, 2 et 3, alors la valeur 3 n'a pas de sens intrinsèque autre que de représenter la modalité numéro 3. En particulier, cela ne signifie pas que c'est 3 fois plus que 1. Cela signifie que la modalité, c'est la modalité numéro 3. On peut très bien imaginer que la modalité 1 signifie « couple », la modalité 2 signifie « femme seule » et la modalité 3 signifie « homme seul ». L'objectif est donc de transformer les nombres en modalité.

La fonction dtype renvoit le type de la variable contenue dans le dataset. Ainsi, pour vérifier s'il s'agit du type « category », on exécute la ligne de code suivante qui vérifie l'égalité entre le type de la variable contenue dans le dataset et le type souhaité.

```
# Vérifier si category
data_german.RESPONSE.dtype == "category"
```

```
## False
```

Si la réponse est `False`, il faut l'encoder avec la fonction astype. Cette fonction permet d'encoder la colonne `RESPONSE` en un type categoriel avec deux modalités.

```
# Encodage en "category"
data_german.RESPONSE =  data_german.RESPONSE.astype("category")
```

#### Encoder plusieurs variables

Nous pouvons également par ce code encoder plusieurs variables.

```
# Encodage plusieurs colonnes
colonnes = ["RESPONSE","INSTALL_RATE"]
data_german.loc[:,colonnes] = (data_german.loc[:,colonnes]
                                        .apply(lambda x : x.astype("category")))
```

### 3.4.3 One Hot Encoding

Une autre opération consiste, lorsqu'il y a plusieurs modalités sur une variable (c'est le cas de la variable *INSTALL_RATE* qui a $4$ modalités), à transformer cette variable catégorielle en une colonne binaire $0/1$ pour chaque modalité.

— On crée une nouvelle colonne pour chaque modalité de la variabe catégorielle

```
# One Hot Encoder
Data2 = data_german
for modalite in np.unique(data_german["INSTALL_RATE"]):
  data_german[f"INSTALL_RATE_{modalite}"]=np.where(
    data_german["INSTALL_RATE"]==modalite,1,0)
data_german.columns
```

```
## Index(['OBS#', 'CHK_ACCT', 'DURATION', 'HISTORY', 'NEW_CAR', 'USED_CAR',
##        'FURNITURE', 'RADIO/TV', 'EDUCATION', 'RETRAINING', 'AMOUNT',
##        'SAV_ACCT', 'EMPLOYMENT', 'INSTALL_RATE', 'MALE_DIV', 'MALE_SINGLE',
##        'MALE_MAR_or_WID', 'CO-APPLICANT', 'GUARANTOR', 'PRESENT_RESIDENT',
##        'REAL_ESTATE', 'PROP_UNKN_NONE', 'AGE', 'OTHER_INSTALL', 'RENT',
##        'OWN_RES', 'NUM_CREDITS', 'JOB', 'NUM_DEPENDENTS', 'TELEPHONE',
##        'FOREIGN', 'RESPONSE', 'INSTALL_RATE_1', 'INSTALL_RATE_2',
##        'INSTALL_RATE_3', 'INSTALL_RATE_4'],
##       dtype='object')
```

— Ne pas oublier de supprimer une colonne générée par le One Hot Encoding

```
# Supprimer la colonne supplémentaire
data_german = data_german.drop(columns=["INSTALL_RATE_4"])
```

**Remarque 3.1.** *Si nous disposons de plusieurs variables pour lesquelles nous souhaitons effectuer du One Hot Enconding, il est plus pratique d'utiliser la fonction get_dummies de pandas.*

### 3.4.4 Amélioration de la qualité des données

La préparation des données consiste aussi à améliorer la qualité et à identifier des problèmes notamment des outliers, c'est - à - dire des valeurs abérrantes.

```
# Filter sur Debt Ratio supérieur à 10 000
DebtHist = (ggplot(data_GMSC[data_GMSC.DebtRatio>=10000],aes(x="DebtRatio"))+
               geom_histogram(fill="steelblue")+
               labs(x="DebtRatio_above_10000)",y="Frequency",
                    title = "Histogram of DebtRatio_above_10000"))
print(DebtHist)
```

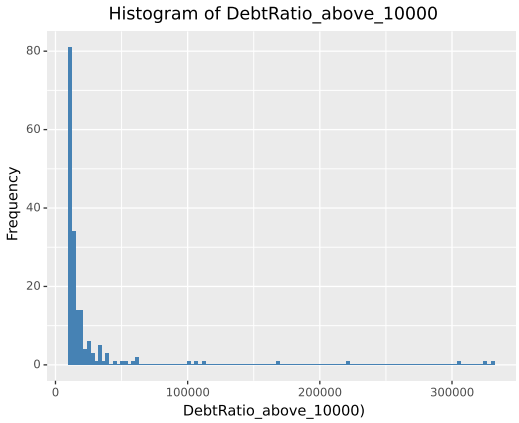


**Figure 3.7 –** Ratio d'endettement supérieure supérieur ou égal à 10000.

Si nous regardons par exemple la variable *DebtRatio* qui doit être inférieure à 100 en général, nous voyons que ses valeurs peuvent monter jusqu'à 10000. Il y a donc ici un problème. Toutes ces valeurs représentées dans le graphique sont abérrantes. Nous avons représenté ici l'histogramme du ratio de dettes supérieur à 10000 qui sont toutes des valeurs abérrantes.

### 3.4.5 Mettre les variables à la même échelle

Une autre étape également importante est de mettre les variables, toutes, à la même échelle. Les variables n'ont pas la même dispersion. En conséquence, une variable qui a beaucoup moins de dispersion qu'une autre variable, a des chances de moins influer sur le modèle.

Pour prendre cet effet en compte, nous considérons les variables *AGE* et *AMOUNT*.

```
# Sélection des variables "AGE" et "AMOUNT"
a = ["AGE","AMOUNT"]
b = data_german[a]
b.head()
```

```
##    AGE  AMOUNT
## 0   67    1169
## 1   22    5951
```

```
## 2   49    2096
## 3   45    7882
## 4   53    4870
```

Nous voyons que la variable *AGE* est comprise entre 20 et 80 ans, c'est - à - dire varie d'une catégorie de 1 à 4. Le montant du crédit (*AMOUNT*) était compris entre 250 et 18000 DM, donc la dispersion est dans un ratio de 1 à 100. Il y a une différence de variabilité entre ces 2 variables et par conséquent, il faut remettre à l'échelle les deux variables.

Cette opération est réalisée par la fonction StandardScaler de sklearn qui pour chaque variable retranche la moyenne de cette variable et divise par la standard deviation de cette variable :

$$z = \frac{x - \overline{x}}{\sigma_x}$$

```
# Centrage - réduction
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
sc.set_output(transform="pandas");
c = sc.fit_transform(b)
print(c.head())
```

```
##         AGE    AMOUNT
## 0  2.766456 -0.745131
## 1 -1.191404  0.949817
## 2  1.183312 -0.416562
## 3  0.831502  1.634247
## 4  1.535122  0.566664
```

Ainsi, on standardise chacune des variables et donc on redéfinit une variable *AGE* et une variable *AMOUNT* qui sont recentrées et remises sur une même échelle.

Nous voyons que les valeurs des différentes variables ont été modifiées. Elles ne veulent plus dire grand chose pour nous mais elles seront plus exploitables par le calcul mathématique.

### 3.4.6 Discrétiser les variables continues

La discrétisation des variables continues, notamment lorsqu'on traite dans un modèle à la fois des données catégorielles et des données continues, est aussi une étape importante. Pour nous aider à faire cela, les statistiques descriptives univariées et la visualisation des données sont des outils précieux. La discrétisation permet aussi de normaliser les variables entre elles.

— Traitement des données catégorielles et continues
— Statistiques descriptives univariées et multivariées : data visualisation
— Recentrage et normalisation

```
# Transformation en un nombre
col_list = ["AGE","DURATION","AMOUNT"]
data_german.loc[:,col_list] = data_german.loc[:,col_list].apply(
                                        lambda x : x.astype("float"),axis=0)

# Variable AGE
data_german["AGE"] = np.where(data_german["AGE"]<=25,"0-25",
                     np.where(data_german["AGE"]<=35,"25-35",
                     np.where(data_german["AGE"]<=45,"35-45","45-")))

# Variable DURATION
data_german["DURATION"] = np.where(data_german["DURATION"]<=12,"0-12",
                          np.where(data_german["DURATION"]<=20,"12-20",
                          np.where(data_german["DURATION"]<=30,"20-30","30-")))

# Variable Amount
data_german["AMOUNT"] = np.where(data_german["AMOUNT"]<=2500,"0-2500",
                        np.where(data_german["AMOUNT"]<=5000,"2600-5000","5000+"))
```

Nous reordonnons l'ordre des modalités.

```
# Transformation en catégorie
for col in col_list :
  data_german[col] = data_german[col].astype("category")
# Ordonnons les modalités
data_german["AGE"].cat.categories = ["0-25","25-35","35-45","45-"]
data_german["DURATION"].cat.categories = ["0-12","12-20","20-30","30-"]
data_german["AMOUNT"].cat.categories = ["0-2500","2600-5000","5000+"]
```

Le choix des frontières et du nombre de modalités font partie du travail de modélisation, de la préparation de la base de données et c'est un élément important dans la performance des modèles. Donc, il faut bien sélectionner le nombre de catégories, le nombre de modalités et les frontières pour ces modalités afin d'avoir des modèles

### 3.4.7 Identification des variables ayant le meilleur pouvoir explicatif

Passons à présent à l'identification des variables ayant le meilleur pouvoir explicatif. Nous avons en parlé dans l'une des sections précédentes et cela se fait par la Data visualisation. Cela peut également se faire par des méthodes statistiques classiques en évaluant le caractère discriminant des variables lorsqu'on prend une, puis une deuxième, puis on en rajoute successivement une troisième, une quatrième, etc...Ceci est une méthode dit **forward** où on ajoute successivement les variables par ordre incrémentale d'importance de la plus importante à la moins importante en se donnant un critère d'arrêt.

Il y a une approche inverse, **backward**, qui consiste à prendre toutes les variables et à supprimer successivement les variables tant qu'elles n'ont pas d'impact ou un impact faible sur le pouvoir discriminant.

Il y a aussi l'utilisation d'un modèle en mode boîte noire, c'est - à - dire que nous prenons toutes les variables que l'on recupère dans les bases de données, nous faisons un

modèle de score et nous regardons ce qu'on appelle la *feature importance*, c'est - à dire la contribution de chaque variable à la performance du modèle et nous gardons enfin dans le modèle final seulement les variables qui ont un impact sur la significativité du modèle.

Nous devons faire attention dans le choix des variables au *leakage*, c'est - à - dire lorsqu'une variable en input est en réalité un output fortement corrélé de manière un peu trivial à la variable *RESPONSE*. C'est l'exemple dans les modèles de crédit, un modèle de défaut, lorsqu'on utilise le retard de paiement à 90 jours comme variable explicative. Dans ce cas, la variable explicative est elle même l'observation.

### 3.4.8 Table de croisement

La table de croisement est aussi un outil important dans la sélection des variables et c'est la fonction CrossTable de la librairie gmodels sous R qui permet d'obtenir l'équivalent des tables croisés dynamiques que nous connaissons.

Voici donc le code qui permt d'obtenir la table de croisement.

```
# Table de croisement
library(gmodels)
gmodels::CrossTable(py$data_german$RESPONSE,py$data_german$CHK_ACCT,digits = 1,
                    prop.r = F,prop.t = F,prop.chisq = F,chisq = T)
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Col Total |
## |-------------------------|
##
##
## Total Observations in Table:  1000
##
##
##                          | py$data_german$CHK_ACCT
## py$data_german$RESPONSE |         0 |         1 |         2 |         3 | Row Total |
## ------------------------|-----------|-----------|-----------|-----------|-----------|
##                       0 |       135 |       105 |        14 |        46 |       300 |
##                         |       0.5 |       0.4 |       0.2 |       0.1 |           |
## ------------------------|-----------|-----------|-----------|-----------|-----------|
##                       1 |       139 |       164 |        49 |       348 |       700 |
##                         |       0.5 |       0.6 |       0.8 |       0.9 |           |
## ------------------------|-----------|-----------|-----------|-----------|-----------|
##            Column Total |       274 |       269 |        63 |       394 |      1000 |
##                         |       0.3 |       0.3 |       0.1 |       0.4 |           |
## ------------------------|-----------|-----------|-----------|-----------|-----------|
##
##
## Statistics for All Table Factors
##
```

```
##
## Pearson's Chi-squared test
## ------------------------------------------------------------
## Chi^2 =  123.7209     d.f. =  3     p =  1.218902e-26
##
##
##
```

Voici ce qu'on obtient lorsqu'on croise la variable *RESPONSE* avec la varible *CHK_ACCT*. Ce qui figure dans chacune des cellules, c'est le nombre d'observation et le ratio entre le nombre d'observation et le nombre total d'observations.

### 3.4.9 Echantillonnage des données

Concernant l'échantillonage des données, il s'agit d'un sujet important notamment lorsqu'on constitue l'ensemble de données d'entraînement et de test. Ainsi :

- Pour un data set uniformément distribé, l'échantillonnage aléatoire est suffisant
- pour des data sets en grande dimension, les techniques de stratified sampling sont plus adaptées et permettent de réduire les erreurs de modèles (biais et variance)
- Data sets déséquilibrés : fréquent dans le risque de cérdit (probabilités de défaut souvent faibles, de l'ordre de quelques pourcents) ; calibrage sur un data set équilibré et test sur le data set réel
- Attention aux biais dans les données (CSP,...)

**En résumé**

- La mise au format des données est une étape indispensable à une correcte modélisation
- Cela inclut le choix du type de chaque variable, le traitement des valeurs abérrantes, le mise à l'échelle des variables les unes par rapport aux autres.
- L'échantillonage des bases de calibrage, cross validation et test est la dernière étape de la préparation des bases avant développement des modèles.

# 4 Data management

Ce chapitre a pour finalité de faire appliquer les notions vues au chapitre précédent et de permettre de monter progressivement en compétences. Il est destiné à vous donner les bases en matière de data wrangling et de prépration des données.

Commençons d'abord par lire la base de données sur laquelle on va travailler :

```
# Chargement des données
import pandas as pd
base = pd.read_csv("./donnee/cs-training.csv",sep=",",index_col=0)
base.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## Int64Index: 150000 entries, 1 to 150000
## Data columns (total 11 columns):
##  #   Column                                Non-Null Count   Dtype
## ---  ------                                --------------   -----
##  0   SeriousDlqin2yrs                      150000 non-null  int64
##  1   RevolvingUtilizationOfUnsecuredLines  150000 non-null  float64
##  2   age                                   150000 non-null  int64
##  3   NumberOfTime30-59DaysPastDueNotWorse  150000 non-null  int64
##  4   DebtRatio                             150000 non-null  float64
##  5   MonthlyIncome                         120269 non-null  float64
##  6   NumberOfOpenCreditLinesAndLoans       150000 non-null  int64
##  7   NumberOfTimes90DaysLate               150000 non-null  int64
##  8   NumberRealEstateLoansOrLines          150000 non-null  int64
##  9   NumberOfTime60-89DaysPastDueNotWorse  150000 non-null  int64
##  10  NumberOfDependents                    146076 non-null  float64
## dtypes: float64(4), int64(7)
## memory usage: 13.7 MB
```

Cette base de données comporte 11 variables dont une variable qualitative (qui est la variable dépendante) et 10 variables quantitatives qui sont nos variables exogènes.

Nous allons commencer par quelques exemples sur les manipulations essentielles en termes de gestion de données à travers les fonctionnalités offertes par :

— Python base
— le package datatable

— le package plydata

Supposons que les données ont été enregistrées dans un tableau appelé tab.

```
# Stockage
tab = base
```

## 4.1 Le data wrangling avec Python

### 4.1.1 Sélection par position dans la table

Sélection des lignes et des colonnes par position

#### Sélection première ligne (toutes les colonnes)

```
# Sélection première ligne (toutes les colonnes)
Selection = tab.iloc[0,:]
```

#### Sélection première colonne (toutes les lignes)

```
# Sélection première colonne (toutes les lignes)
Selection = tab.iloc[:,0]
```

Plus de solutions pour sélectionner les observations par position :

#### Sélection groupe de lignes (continue)

```
# Sélection groupe de lignes (continue)
Selection = tab.iloc[0:4,:]
```

ou

```
# Sélection groupe de lignes (continue)
Selection = tab.iloc[slice(0,4),:]
```

#### Sélection groupe de lignes - sélection discontinue

```
# Sélection groupe de lignes - Sélection discontinue
Selection = tab.iloc[[0,4,7],:]
```

#### Premières lignes

```
# 5 premières lignes
Selection = tab.head()

# 10 premières lignes
Selection = tab.head(10)

# 100 premières lignes
Selection = tab.head(100)
```

#### Dernières lignes

```
# 5 dernières lignes
Selection = tab.tail()

# 10 dernières lignes
Selection = tab.tail(10)

# 100 dernières lignes
Selection = tab.tail(100)
```

Plus de solutions pour sélectionner les variables par position :

#### Sélection groupe de colonnes (continue)

```
# Sélection groupe de colonnes (continue)
Selection = tab.iloc[:,0:4]
```

ou

```
# Sélection groupe de colonnes (continue)
Selection = tab.iloc[:,slice(0,4)]
```

#### Sélection groupe de colonnes - sélection discontinue

```
# Sélection groupe de colonnes - sélection discontinue
Selection = tab.iloc[:,[0,5,7]]
```

Sélection simultanée lignes et colonnes :

#### Sélection discontinue lignes et continue colonnes

```
# Sélection discontinue lignes et continue colonnes
Selection = tab.iloc[[0,2],:8]
```

#### Sélection discontinue colonnes et continue lignes

```
# Sélection discontinue colonnes et continue lignes
Selection = tab.iloc[:8,[0,2]]
```

#### Sélection mixte de chaque côté

```
# Sélection mixte de chaque côté
Selection = tab.iloc[[1,4,5],[0,2,6]]
```

### 4.1.2 Filtrer les observations par valeur (condition logique)

#### Une seule condition

```
# Une seule condition
Selection = tab[tab.age== 34]
```

#### Conditions multiples (1)

```
# Conditions multiples (1)
Selection = tab[(tab.age == 34) | (tab.age == 36)]
```

#### Conditions multiples (2)

```
# Conditions multiples (2)
Selection = tab[(tab.age == 34) | (tab.age == 36) | (tab.age == 57)]
```

#### Conditions multiples (3) - écriture compacte avec `isin`

```
# Conditions multiples (3) - écriture compacte avec isin
Selection = tab[tab.age.isin([34,43,57,59,61,62])]
```

Avec la fonction query

#### Une seule condition

```
# Une seule condition
Selection = tab.query("age == 34")
```

#### Conditions multiples

```
# Conditions multiples
Selection = tab.query("age == 34 | age == 36")
```

Si nous voulions faire des filtrages par test logique sur des variables de type `object` ou `category` (autrement dit des textes, aussi appelés `string` ou chaines de caractères), il faudrait que chaque valeur à filtrer soit mise entre quotes.

Ces quotes peuvent être, au choix :

— quotes doubles, les guillemets classiques : ” ”
— quotes simples, en fait le signe apostrophe : ’ ’

```
## Filtrage par test logique
# sélection simple
Selection = tab.query("nom == 'Alexandre'")

# Sélection multiple
Selection = tab[(tab.nom == 'Alexandre') | (tab.nom == 'Sophie')]

# Sélection multiple avec isin
Selection = tab[tab.nom.isin(['Alexandre','Sophie','Jean','Sarah'])]
```

Il est clair que les variables de type character peuvent contenir des chiffres... En fait, vous pouvez parfaitement créer ou récupérer un tableau avec une colonne qui donne l’âge en chiffres sur toutes les lignes, sans aucune lettre, mais qui a été typée `object` ou `category` ! Ce serait sans doute une erreur, mais cela peut arriver.

Dans ce cas :

```
# Syntaxe appropriée
Selection = tab[tab.age == "34"]
```

est la syntaxe appropriée, et :

```
# Syntaxe inappropriée
Selection = tab[tab.age == 34]
```

ne peut pas marcher.

Si vous voulez revenir à un mode de sélection plus cohérent (sélectionner des chiffres sans les quotes) il ne reste qu’à modifier le format de la colonne en question. C’est-à-dire exécuter le code suivant :

```
# typage de la variable - réel
tab.age = tab.age.astype("float")
```

dans le cas des nombres réels, ou

```
# typage de la variable - réel
tab.age = tab.age.astype("int")
```

dans le cas des nombres entiers naturels.

### 4.1.3 Sélectionner ou non les valeurs manquantes/nulles

#### Seulement les valeurs manquantes pour la colonne ”NumberOfDependents”

```
# Filtrer uniquement les lignes NA sur NumberOfDependents
Selection = tab[tab["NumberOfDependents"].isna()]
```

#### Seulement les valeurs qui ne sont pas manquantes pour la colonne ”NumberOfDependents”

```
# Les lignes notna
Selection = tab[tab["NumberOfDependents"].notna()]
```

On peut également utiliser la fonction dropna du package pandas.

```
# Filtrer uniquement sur les lignes non NA NumberOfDependents
Selection = tab.dropna(subset=["NumberOfDependents"])
```

#### Seulement les valeurs qui ne sont pas manquantes, quelle que soit la colonne

```
# Seulement les valeurs qui ne sont pas manquantes
Selection = tab.dropna(how="any")
```

### 4.1.4 Sélectionner les variables par leur nom

#### Sélection une colonne

```
# Sélection une colonne
Selection = tab["age"]
```

ou

```
# Sélection une colonne
Selection = tab.loc[:,"age"]
```

#### Sélection plusieurs colonnes

```
# Sélection plusieurs colonnes
Selection = tab[["age","DebtRatio"]]
```

#### Sélection suite continue de colonnes (toutes les colonnes entre ”age” et ”DebtRatio”)

```
# Sélection suite continue de colonnes
Selection = tab.loc[:,"age":"DebtRatio"]
```

**NB :**

L'exemple ci-dessus est assez important. Ce n'est pas une manipulation si courante, mais on verra que faire appel à des packages particuliers peut rendre inopérantes des manipulations de Python base. C'est le cas pour l'instruction ci-dessus si on utilise un format de tableau appelé datatable, qui vient du package du même nom.

### 4.1.5 Réorganiser le tableau (ordre des colonnes)

**Par position**

```
# Sélection par position
Selection = tab.iloc[:,[0,1,5,4,3,2]]
```

**On peut inverser totalement l'ordre des colonnes**

```
# Inverser l'ordre des colonnes
Selection = tab[tab.columns[::-1]]
```

ou

```
# Inverser l'ordre des colonnes
Selection = tab.iloc[:,::-1]
```

**Par nom**

```
# Nom des colonnes
tab.columns.values
```

```
## array(['SeriousDlqin2yrs', 'RevolvingUtilizationOfUnsecuredLines', 'age',
##        'NumberOfTime30-59DaysPastDueNotWorse', 'DebtRatio',
##        'MonthlyIncome', 'NumberOfOpenCreditLinesAndLoans',
##        'NumberOfTimes90DaysLate', 'NumberRealEstateLoansOrLines',
##        'NumberOfTime60-89DaysPastDueNotWorse', 'NumberOfDependents'],
##       dtype=object)
```

```
# Inverser par nom
Selection = tab[["SeriousDlqin2yrs","MonthlyIncome","age"]]
```

### 4.1.6 Ordonner les observations

**On peut ranger une variable par valeurs croissantes/décroissantes (ou par ordre alphabétique/anti-alphabétique si la variable est de type string)**

```
# Ordonner
Selection = tab.sort_values(by="age")
# Ordonner
Selection = tab.sort_values(by="age",ascending=False)
# Ordonner
Selection = tab.sort_values(by="MonthlyIncome")
# Ordonner
Selection = tab.sort_values(by="MonthlyIncome",ascending=False)
```

**Créer une nouvelle variable**

```
# Variable log - salaire
import numpy as np
tab.loc[:,"log_salaire"] = np.log(tab["MonthlyIncome"])
# Variable indicatrice_dette
tab.loc[:,"indicatrice_dette"] = np.where(tab["DebtRatio"]>1,1,0)
```

On retourne à la table d'origine sans "log_salaire" et "indicatrice_dette" :

```
# Base initiale
tab = base
```

### 4.1.7 Opération par groupe et fonctions d'agrégtion

**On veut le salaire moyen par âge**

```
# Salaire moyen par âge
Selection = tab.pivot_table(values=["MonthlyIncome"],columns=["age"],
                            aggfunc=np.mean,dropna=False)
Selection
```

```
## age                0            21            22   ...   105  107  109
## MonthlyIncome  6000.0  1128.123077  1312.592705   ...   NaN  NaN  NaN
##
## [1 rows x 86 columns]
```

Par défaut, la fonction pivot_table supprime les colonnes contenant les NA.

Une autre approche est d'utiliser « groupby ». Elle est plus réaliste lorsque des colonnes ou des lignes contiennent des NA.

```
# Salaire moyen par âge
Selection = tab[["MonthlyIncome","age"]].groupby("age").mean()
Selection
```

```
##      MonthlyIncome
## age
## 0      6000.000000
## 21     1128.123077
## 22     1312.592705
## 23     1728.003883
## 24     2061.717791
## ..             ...
## 102    3358.500000
## 103     800.500000
## 105            NaN
## 107            NaN
## 109            NaN
##
## [86 rows x 1 columns]
```

Les résultats montrent un échec : pour un âge donné, c'est la valeur manquante « NA » qui s'affiche à la place de la vraie moyenne de MonthlyIncome.

Pour les fonctions d'agrégats (mean, max, min...), la valeur NA se communique en effet à toute la variable en argument. Par exemple, si on fait la moyenne de 100 valeurs numériques dont 1 valeur est NA, la moyenne est NA, même si les 99 autres valeurs sont bien renseignées.

Il faut donc utiliser la fonction `np.nanmean`.

```
# Salaire moyen par âge
Selection = tab.pivot_table(values=["MonthlyIncome"],columns=["age"],
                            aggfunc=np.nanmean)
Selection
```

```
## age                0            21            22   ...     101     102    103
## MonthlyIncome  6000.0  1128.123077  1312.592705  ...  2274.5  3358.5  800.5
##
## [1 rows x 83 columns]
```

**Remarque :**

L'autre solution aurait été de travailler sur une table jumelle, que nous aurions spécialement modifiée pour se débarrasser de toutes les lignes avec NA sur la colonne-argument. L'option « dropna » permet justement de se passer de ce genre de traitement.

**Attention !!!**

La valeur « `True` » est réservée dans Python. On ne peut pas créer d'objet qui s'appelle True. Cette chaine de caractère est strictement réservée pour signifier qu'un test lo-

gique est évalué à « vrai ». Bien sûr, il en est de même pour « `False` », qui évalue à « faux ».

Cela n'est cependant vrai qu'avec « `True` » et « `False` ». Ceci signifie :

— qu'on peut créer des objets comme TRUE, true, FaLSe…
— et que de fait donner une valeur comme TRUE à l'option « dropna » ne marchera pas. Toute chaine de lettres autre que `dropna=True` et `dropna=False` ne sera pas comprise par Python .

### 4.1.8 Plus d'informations sur les fonctions d'agrégation

On appelle **fonctions d'agrégation** des fonctions qui retournent une valeur unique (agrégée) à partir d'une liste de plusieurs valeurs (en agrégeant ces valeurs).

**Attention !!!**

Avec ces fonctions d'agrégation vous pouvez obtenir un message d'avertissement, si les colonnes du tableau sont au format `category` (facteur). Il est en effet impossible de calculer des fonctions d'agrégation sur des facteurs. Dans ce cas, on peut retirer ce type "category" en convertissant ces variables au format numeric.

## 4.2 Le data wrangling avec datatable

### 4.2.1 Présentation du datatable

#### Quelles sont les caractéristiques d'un datatable ?

```
# Type
type(tab)
```

```
## <class 'pandas.core.frame.DataFrame'>
```

```
# Inspection
tab.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## Int64Index: 150000 entries, 1 to 150000
## Data columns (total 13 columns):
##  #   Column                                Non-Null Count   Dtype
## ---  ------                                --------------   -----
##  0   SeriousDlqin2yrs                      150000 non-null  int64
##  1   RevolvingUtilizationOfUnsecuredLines  150000 non-null  float64
##  2   age                                   150000 non-null  int64
##  3   NumberOfTime30-59DaysPastDueNotWorse  150000 non-null  int64
##  4   DebtRatio                             150000 non-null  float64
##  5   MonthlyIncome                         120269 non-null  float64
##  6   NumberOfOpenCreditLinesAndLoans       150000 non-null  int64
```

```
##  7   NumberOfTimes90DaysLate               150000 non-null  int64
##  8   NumberRealEstateLoansOrLines          150000 non-null  int64
##  9   NumberOfTime60-89DaysPastDueNotWorse  150000 non-null  int64
##  10  NumberOfDependents                    146076 non-null  float64
##  11  log_salaire                           120269 non-null  float64
##  12  indicatrice_dette                     150000 non-null  int32
## dtypes: float64(5), int32(1), int64(7)
## memory usage: 15.4 MB
```

```
# Nombre de lignes
len(tab)
```

```
## 150000
```

```
# Dimension du table
tab.shape
```

```
## (150000, 13)
```

**On se dote du package datatable**

```
# pip install datatable
import datatable as dt
```

**On transforme notre tableau en datatable**

```
# Transformation en datatable
tab = dt.Frame(tab)
```

Regardons ses caractéristiques :

```
# Caractéristiques
type(tab)
```

```
## <class 'datatable.Frame'>
```

```
# Nom des colonnes
for i in tab.names:
  print(i)
```

```
## SeriousDlqin2yrs
## RevolvingUtilizationOfUnsecuredLines
## age
## NumberOfTime30-59DaysPastDueNotWorse
## DebtRatio
## MonthlyIncome
```

```
## NumberOfOpenCreditLinesAndLoans
## NumberOfTimes90DaysLate
## NumberRealEstateLoansOrLines
## NumberOfTime60-89DaysPastDueNotWorse
## NumberOfDependents
## log_salaire
## indicatrice_dette

# Typage des colonnes
for i in tab.types:
  print(i)


## Type.int64
## Type.float64
## Type.int64
## Type.int64
## Type.float64
## Type.float64
## Type.int64
## Type.int64
## Type.int64
## Type.int64
## Type.float64
## Type.float64
## Type.int32

# Dimension
tab.shape


## (150000, 13)
```

On verra néanmoins un exemple où ce qu'on pouvait faire avec un DataFrame n'est plus possible avec un objet de type datatable. On aura alors besoin de retourner à un objet dont la classe est UNIQUEMENT dataframe. Comment faire ?

Comme ceci :

```
# Du datatable au dataframe
tab = tab.to_pandas()
```

### 4.2.2 Syntaxe spécifique

Maintenant étudions l'intérêt d'un datatable. La syntaxe est spécifique. Elle est du type :

```
DT [ i, j , by ]
```

— $i$ les lignes à sélectionner
— $j$ les colonnes
— $by$ la variable de regroupement s'il y a lieu

### 4.2.3 Sélection des lignes

On se dote d'un datatable

```
# Transformation en tableau datatable
tab = dt.Frame(tab)
```

#### Une seule condition

```
# Une seule condition (1)
Selection = tab[dt.f.age==34,:]
```

#### Conditions multiples (1)

```
# Conditions multiples (1)
Selection = tab[(dt.f.age==34)|(dt.f.age==36),:]
```

#### Conditions multiples (2)

```
# Conditions multiples (2)
Selection = tab[(dt.f.age==34)|(dt.f.age==43)|(dt.f.age==57)|(dt.f.age==59),:]
```

#### Conditions multiples (3)

```
# Conditions multiples (3)
Selection = tab[[dt.f.age == i for i in [34,43,57,59,61,62]],:]
```

### 4.2.4 Sélection des colonnes

On passe a pandas dataframe.

```
# Convertion to pandas DataFrame
tab = tab.to_pandas()
type(tab)
```

```
## <class 'pandas.core.frame.DataFrame'>
```

On reprend la manière de faire classique avec Python base pour sélectionner des colonnes :

#### Sélection d'une colonne

```
# Sélection d'une colonne
Selection = tab["age"]
```

#### Sélection de plusieurs colonnes

```
# Sélection de plusieurs colonnes
Selection = tab[["age","DebtRatio"]]
```

#### Sélection suite continue de colonnes (toutes les colonnes entre ”age” et ”DebtRatio”

```
# Selection age:DebtRatio
Selection = tab.loc[:,"age":"DebtRatio"]
```

### Comparaison avec datatable

On passe au datatable

```
# Conversion au tableau datatable
tab = dt.Frame(tab)
type(tab)
```

```
## <class 'datatable.Frame'>
```

#### Sélection avec les listes

```
# Selection des colonnes age et DebtRatio
Selection = tab[:,["age","DebtRatio"]]
```

#### Sélection via le slicing

```
# Sélection via le slicing
Selection = tab[:,"age":"DebtRatio"]
```

On peut également utiliser la fonction « slice »

```
# Sélection via la fonction slice
Selection = tab[:,slice("age","DebtRatio")]
```

#### Sélection via une expression

```
# Sélection via une express
Selection = tab[:,[dt.f.age,dt.f.DebtRatio]]
# Sélection
Selection = tab[:,dt.f["age","DebtRatio"]]
```

### 4.2.5 Fonctions d'agrégation

**Une fonction d'agrégation : la somme de toutes les lignes de la colonne MonthlyIncome :**

```
# Somme
tab[:,dt.sum(dt.f.MonthlyIncome)]
```

```
##    | MonthlyIncome
##    |       float64
## -- + -------------
##  0 |   8.02221e+08
## [1 row x 1 column]
```

On peut appliquer une fonction d'agrégation à deux colonnes à la fois, de sorte qu'elle retourne un résultat unique. Evidemment, il faut que les deux colonnes mesurent des choses comparables, et même qu'elles aient la même unité. Pour le besoin de l'exemple, on considère qu'on peut mettre sur le même plan deux de nos variables :

- `NumberOfTimes90DaysLate` : c'est le nombre de fois où l'emprunteur a eu plus de 90 jours de retard de paiement sur une échéance de son crédit
- `NumberOfTime60-89DaysPastDueNotWorse` : c'est le nombre de fois où l'emprunteur a eu un retard de paiement compris ente 60 et 89 jours.

On peut alors écrire simplement

```
# Moyenne
tab[:,dt.mean(dt.f["NumberOfTimes90DaysLate",
                   "NumberOfTime60-89DaysPastDueNotWorse"])]
```

```
##    | NumberOfTimes90DaysLate  NumberOfTime60-89DaysPastDueNotWorse
##    |                 float64                                float64
## -- + ----------------------  -----------------------------------
##  0 |                0.265973                               0.240387
## [1 row x 2 columns]
```

### 4.2.6 Fonctions d'agrégation : stocker le résultat

Stocker le résultat : pour le nom de la colonne à créer, on le met dans un dictionnaire

```
# Créer une colonne
tab[:,{"Ecart_Type_Income" : dt.sd(dt.f.MonthlyIncome)}]
```

```
##    | Ecart_Type_Income
##    |           float64
## -- + -----------------
##  0 |           14384.7
## [1 row x 1 column]
```

**Calculer d'un coup plusieurs agrégats et les stocker à chaque fois avec un nom :**

```
# Calculer d'un coup plusieurs agrégats et
tab2 = tab[:,{"Max_Income" : dt.max(dt.f.MonthlyIncome),
              "Min_Income" : dt.min(dt.f.MonthlyIncome),
              "Moy_Income" : dt.mean(dt.f.MonthlyIncome),
              "Ecart_Type_Income" : dt.sd(dt.f.MonthlyIncome),
              "Somme_Income" : dt.sum(dt.f.MonthlyIncome)}]
```

On se donne la table originale

```
# Tableau originale
tab = dt.Frame(base)
```

**On peut faire en sorte que le résultat créé constitue une nouvelle colonne de la table d'origine :**

```
# Stockage
tab2 = tab[:,{"Ecart_Type_Income" : dt.sd(dt.f.MonthlyIncome)}]
tab["Ecart_Type_Income"]=tab2
```

On se donne la table originale :

```
# Transformation
tab = dt.Frame(base)
```

### 4.2.7 Autres calculs

On peut évidemment avoir recours à d'autres fonctions, qui ne sont pas des fonctions d'agrégation. Logiquement le nombre initial de lignes du tableau est conservé :

```
# Autres calculs
tab[:,{"Log_Income" : dt.log(dt.f.MonthlyIncome)}]
```

```
##        | Log_Income
##        |    float64
## ------ + ----------
##      0 |    9.11823
##      1 |    7.86327
##      2 |    8.02027
##      3 |    8.10168
```

```
##      4 |   11.0602
##      5 |    8.16052
##      6 |   NA
##      7 |    8.16052
##      8 |   NA
##      9 |   10.0726
##     10 |    7.82405
##     11 |    8.77971
##     12 |    9.4298
##     13 |    9.52515
##     14 | -inf
##    ... |  ...
## 149995 |    7.64969
## 149996 |    8.62766
## 149997 |   NA
## 149998 |    8.65102
## 149999 |    9.00675
## [150000 rows x 1 column]
```

### 4.2.8 Créer/modifier une variable

Sauf manipulation ad hoc, les syntaxes précédentes ne vont pas ajouter de nouvelles variables au tableau d'origine. Donc comment enrichir le tableau par de nouvelles colonnes ?

**Doter la table d'une colonne supplémentaire :**

```
# Doter la table d'une colonne supplémentaire :
tab["Log_Income"] = tab[:,{"Log_Income" : dt.log(dt.f.MonthlyIncome)}]
for i in tab.names:
  print(i)
```

```
## SeriousDlqin2yrs
## RevolvingUtilizationOfUnsecuredLines
## age
## NumberOfTime30-59DaysPastDueNotWorse
## DebtRatio
## MonthlyIncome
## NumberOfOpenCreditLinesAndLoans
## NumberOfTimes90DaysLate
## NumberRealEstateLoansOrLines
## NumberOfTime60-89DaysPastDueNotWorse
## NumberOfDependents
## log_salaire
## indicatrice_dette
## Log_Income
```

ou

```
tab = dt.Frame(tab)
tab[:,dt.update(Log_Income = dt.log(dt.f.MonthlyIncome))]
for i in tab.names:
  print(i)
```

```
## SeriousDlqin2yrs
## RevolvingUtilizationOfUnsecuredLines
## age
## NumberOfTime30-59DaysPastDueNotWorse
## DebtRatio
## MonthlyIncome
## NumberOfOpenCreditLinesAndLoans
## NumberOfTimes90DaysLate
## NumberRealEstateLoansOrLines
## NumberOfTime60-89DaysPastDueNotWorse
## NumberOfDependents
## log_salaire
## indicatrice_dette
## Log_Income
```

### 4.2.9 Opération par groupe

On veut comme plus haut la moyenne du revenu par âge. C'est ici qu'on utilise le troisième indice (deuxième virgule) qui existe dans la syntaxe de datatable :

```
# Salaire moyen par âge
tab = dt.Frame(tab)
Selection = tab[:, dt.mean(dt.f.MonthlyIncome), dt.by("age")]
Selection
```

```
##     |   age  MonthlyIncome
##     | int64        float64
## --- + -----  -------------
##   0 |     0        6000
##   1 |    21        1128.12
##   2 |    22        1312.59
##   3 |    23        1728
##   4 |    24        2061.72
##   5 |    25        2529.14
##   6 |    26        2890.9
##   7 |    27        3247.63
##   8 |    28        3605.2
##   9 |    29        3934.36
##  10 |    30        4259.56
##  11 |    31        4676.3
##  12 |    32        4707.36
##  13 |    33        5277.78
##  14 |    34        5407.67
## ... |   ...         ...
##  81 |   102        3358.5
```

```
##  82 |   103         800.5
##  83 |   105          NA
##  84 |   107          NA
##  85 |   109          NA
## [86 rows x 2 columns]
```

## 4.3 Le data wrangling avec plydata et datar

On va maintenant étudier le package plydata, qui est probablement le package de wrangling le plus populaire sous Python.

### 4.3.1 Le format tibble avec datar

datar s'utilise souvent avec un nouveau format de tableau, là encore distinct de dataframe, le tibble.

```
# Format tibble
from datar.tibble import tibble
tab = tibble(base)
type(tab)
```

```
## <class 'datar.core.tibble.Tibble'>
```

### 4.3.2 le pipe >>

Un des grands intérêts du plydata, datar et dfplyr est de pouvoir utiliser le symbôle « >> ». Ce symbôle, appelé « le pipe », est d'une grande importance pour écrire des instructions de façon compacte. Le pipe >> rend le code plus lisible.

Le pipe >> fonctionne sur le principe suivant : >> passe l'objet se trouvant à gauche de ce symbole comme premier argument de la fonction se trouvant à droite. On peut ajouter d'autres pipes à la suite si on veut enchainer les actions.

**Exemple 4.1.** *Quel est l'âge pour lequel le revenu moyen est le plus élevé ?*

```
# Avec plydata
from plydata import *
Selection = (
  tab >>
  group_by("age") >>
  summarize(Moyenne='np.mean(MonthlyIncome)')>>
  arrange("-Moyenne")
)
Selection
```

```
##        age       Moyenne
##    <int64>     <float64>
## 76      94  13134.428571
## 27      52   8783.422230
```

```
## 46      44   8390.311348
## 6       57   7910.240089
## ..     ...           ...
## 31      68   7876.006200
## 72      21   1128.123077
## 80     103    800.500000
## 82     107           NaN
## 83     105           NaN
## 85     109           NaN
##
## [86 rows x 2 columns]
```

Dans la suite, on verra quelques exemples d'utilisation du pipe, pour comparer avec la syntaxe "classique".

### 4.3.3 Filtrer les observations par condition logique

La sélection des lignes par condition logique s'effectue avec `query`.

#### Condition logique classique

```
# Une seule condition
Selection = query(tab,"age > 22")
# Conditions multiples
Selection = query(tab,"age > 22 & age < 67")
```

Les 1000 revenus les plus bas (attention ne renvoie pas 1000 observations, mais potentiellement plus s'il y a des individus ex aequo).

Si on réécrit l'une de ces instructions avec le pipe :

```
# Utilisation de pipe
Selection = tab >> query("age > 22 & age < 67")
```

### 4.3.4 Sélectionner les lignes par position

La sélection des lignes par position est cependant toujours possible, toujours avec une fonction qui allège la syntaxe :

```
# Sélection par position
Selection = slice_rows(tab,500,800)
```

avec le pipe

```
# Sélection par position  - avec pipe
Selection = tab >> slice_rows(500,800)
```

### 4.3.5 Sélection d'un échantillon aléatoire

On sélectionne aléatoirement une fraction des observations, ou bien en proportion du nombre d'individus, ou bien en nombres d'individus :

```
#Sélection selon le pourcentage
Selection = sample_frac(tab, 0.35, replace = True)
# Sélection selon le nombre
Selection = sample_n(tab, 30, replace = True)
```

### 4.3.6 Sélection des variables

La fonction `select` allège la syntaxe et n'oblige même pas à recourir à [] si on sélectionne plus d'une seule variable :

```
# Sélection des variables
Selection = select(tab,"age","MonthlyIncome")
```

**Remarque :**

Le package plydata prévoit l'utilisation de nombreuses fonctions complémentaires avec select.

On peut sélectionner les variables.

**Si elles contiennent une chaîne de caractères**

```
# Contient une chaîne de caractères
Selection = select(tab,contains="days")
```

**si elles contiennent une expression régulière**

```
# Contiennent une expression régulière
Selection = select(tab,matches="D...")
```

**si elles se finissent par une chaine de caractères**

```
# Sélection se terminant par "late"
Selection = select(tab,endswith="late")
```

**si elles sont comprises entre deux colonnes (attention donc à l'ordre des variables dans le tableau)**

```
# Slicing
Selection = select(tab,slice("age","MonthlyIncome"))
```

Avec le pipe

```
# Avec le pipe
Selection = tab >> select("age","MonthlyIncome")
```

### 4.3.7 Renommer les variables

La fonction “rename” va renommer les variables, sans supprimer pour autant les variables non-renommées :

```
# Renommer
tab2 = rename(tab,deliquency="SeriousDlqin2yrs", Debt="DebtRatio",
              Income="MonthlyIncome")
```

Avec le pipe

```
# Renommer avec le pipe
tab2 = tab >> rename(deliquency="SeriousDlqin2yrs", Debt="DebtRatio",
                     Income="MonthlyIncome")
```

### 4.3.8 Réorganiser l’ordre des variables

Cette fois-ci préciser toutes les variables à inclure :

```
# Selectionner des colonnes
tab = select(tab, "MonthlyIncome", "age", "DebtRatio")
```

On se redonne notre table d’origine :

```
# Transformer en tibble
tab = tibble(base)
```

### 4.3.9 Ordonner les observations

#### Ordre croissant

```
# Ordre croissant
Selection = arrange(tab, "age")
```

#### Ordre décroissant

```
# Ordre décroissant
Selection = arrange(tab, "-age")
```

### 4.3.10 Fonctions d'agrégation "summarize"

On peut résumer l'information d'une variable sur laquelle s'applique une fonction d'agrégation en une seule ligne avec la fonction `summarise` (ou la fonction `summarize` avec l'orthographe américaine).

La fonction `summarize` s'utilise donc en couple avec d'autres fonctions sur le modèle :

```
# Application
summarize(tab,nom_resultat=fonction_de_resume(variable))
```

Parmi ces fonctions de résumé, on trouve :

- `first` : affiche la première valeur d'un vecteur
- `last` : affiche la dernière valeur d'un vecteur
- `nth` : affiche la n-ième valeur d'un vecteur
- `n` : affiche le nombre de valeurs d'un vecteur (c'est le nombre d'observations)
- `n_distinct` : affiche le nombre de valeurs distinctes d'un vecteur
- `min` : valeur minimum d'un vecteur
- `max` : valeur maximum d'un vecteur
- `mean` : moyenne d'un vecteur
- `median` : médiane d'un vecteur

```
# Moyenne de la dette
Selection = summarize(tab,average="np.mean(DebtRatio)")
```

### 4.3.11 Créer/remplacer des variables

#### Créer de nouvelles variables

On utilise `mutate`

```
# Mutate
tab = mutate(tab,log_age="np.log(age)")
```

Avec le pipe

```
# En utilisant le pipe
tab = tibble(base)
Selection = tab >> mutate(log_age="np.log(age)")
```

#### Actualiser le tableau en mentionnant les variables à garder

On utilise `transmute`.

```
# Transmute
tab2 = transmute(tab, "SeriousDlqin2yrs", "DebtRatio", "MonthlyIncome",
                 log_age="np.log(age)")
```

Avec la pipe

```
# Transmute avec le pipe
Selection = tab >> transmute("SeriousDlqin2yrs", "DebtRatio", "MonthlyIncome",
                             log_age="np.log(age)")
```