Trafic de données avec Python-pandas

Résumé

*L'objectif de ce tutoriel est d'introduire Python pour la préparation (*data munging *ou* wrangling *ou trafic) de données massives, lorsqu'elles sont trop volumineuses pour la mémoire (RAM) d'un ordinateur. Cette étape est abordée par l'initiation aux fonctionnalités de la librairie* `pandas` *et à la classe* `DataFrame`*; lire et écrire des fichiers, gérer une table de données et les types des variables, échantillonner, discrétiser, regrouper des modalités, description élémentaires uni et bi-variées ; concaténation et jointure de tables.*

- *Python pour Calcul Scientifique*
- *Trafic de Données avec Python.*`Pandas`
- *Apprentissage Statistique avec Python.*`Scikit-learn`
- *Programmation élémentaire en Python*
- *Sciences des données avec* `Spark-MLlib`

1 Introduction

1.1 Objectif

Le *data munging* ou *wrangling* (traduit ici par *trafic*) de données est l'ensemble des opérations permettant de passer de données brutes à une table (*data frame*) correcte et adaptée aux objectifs à atteindre par des méthodes statistiques d'analyse, exploration, modélisation ou apprentissage.

En présence de données complexes, peu ou mal organisées, présentant des trous, trop massives pour tenir en mémoire... la qualité de cette étape est fondamentale (*garbage in garbage out*) pour la bonne réalisation d'une étude. Compte tenu de la diversité des situations envisageables, il serait vain de vouloir en exposer tous les outils et techniques qui peuvent s'avérer nécessaires. Tâchons néanmoins de résumer les problèmes qui peuvent être rencontrés.

Croissance du volume

Le volume des données et sa croissance occasionnent schématiquement trois situations.

1. Le fichier initial des données brutes peut être chargé intégralement en mémoire moyennant éventuellement de sauter quelques colonnes ou lignes du fichier (cf. section 3.1). C'est la situation courante, tout logiciel statistique peut réaliser les traitements, R est à favoriser mais, afin d'aborder la situation suivante de façon cohérente, l'usage de Python peut se justifier.

   C'est l'objet des sections 2 à 6.

2. Le fichier initial est très volumineux mais la table (`DataFrame`), qui résulte de quelques trafics (*munging*) appropriés, tient en mémoire. Cette situations nécessite : lecture, analyse, transformation, ré-écriture, séquentielles du fichier ligne à ligne ou par bloc. Il existe des astuces avec R mais il est préférable d'utiliser des outils plus adaptés. Tout langage de programmation (java, c, perl, ruby...) peut être utilisé pour écrire le ou les programmes réalisant ce travail. Néanmoins Python, et plus précisément la librairie `pandas`, offrent un ensemble d'outils efficaces pour accomplir ces tâches sans avoir à ré-inventer la roue et ré-écrire tout un ensemble de fonctionnalités relativement basiques. Remarque : les procédures `univariate` et `freq` et l'étape `data` de SAS sont adaptées car elles ne chargent pas les données en mémoire pour réaliser des traitements rudimentaires. Néanmoins pour tout un tas de raisons trop longues à exposer, notamment de coût annuel de location, SAS perd régulièrement des parts de marché sur ce créneau.

   Cette approche est introduite ci-dessous en section 7 et consiste à enchâsser dans une même structure itérative et séquentielle les étapes précédentes des sections 2 à 6.

3. Lorsque les données, très massives, sont archivées sur un système de données distribuées (HDFS *Hadoop*), trafic et prétraitement des données doivent tenir compte de cet environnement. Si, en résultat, la table résultante est trop volumineuse, à moins d'échantillonnage, il ne sera pas possible de mettre en œuvre des méthodes statistiques et d'apprentissage avancées en utilisant R ou la librairie `scikit-learn` de Python. Noter néanmoins que `scikit-learn` propose quelques méthodes incrémentales rudimentaires d'apprentissage. Pour prévenir des temps d'exécu-

tions nécessairement longs, il devient incontournable de mettre en œuvre les technologies appropriées de parallélisation : *Python / Spark / Hadoop* et la librairie `MLlib` de *Spark* (ou plus récentes !) pour *paralléliser l'analyse de données distribuées*.

C'est l'objet du tutoriel introduisant les fonctionnalités de Spark.

Quelques problèmes

Liste non exhaustive des problèmes pouvant être rencontrés et dont la résolution nécessite simultanément des compétences en Informatique, Statistique, Mathématiques et aussi "métier" du domaine de l'étude.

- Détermination des "individus" × "variables" (*instances*×*features* en langue informatique) de la table à mettre en forme à partir de bases de données variées ; *i.e.* logs d'un site web, listes d'incidents, localisations...
- Donnés atypiques (*outliers*) : correction, suppression, transformation des variables ou méthode statistique robuste ?
- Variable qualitative avec beaucoup de modalités dont certaines très peu fréquentes : suppression, modalité "`autres`", recodage aléatoire, regroupement "métier" ou méthode tolérante ?
- Distributions a-normales (log-normale, Poisson, multimodales...) et problèmes d'hétéroscédasticité : transformation, discrétisation ou méthodes tolérantes ?
- Données manquantes : suppressions (ligne ou colonne), imputation ou méthodes tolérantes ?
- Représentations (splines, Fourier, ondelettes) et recalage (*time warping*) de données fonctionnelles.
- Représentation de trajectoires, de chemins sur un graphe ?
- Choix d'une distance (quadratique, absolue, géodésique...) entre les objets étudiés.
- ...

Bien entendu les "bons" choix dépendent directement de l'objectif poursuivi et des méthodes mises en œuvre par la suite. D'où l'importance d'intégrer de façon précoce, dès la planification du recueil des données, les compétences nécessaires au sein d'une équipe.

1.2 Fonctionnalités de `pandas`

La richesse des fonctionnalités de la librairie `pandas` est une des raisons, si ce n'est la principale, d'utiliser Python pour extraire, préparer, éventuellement analyser, des données. En voici un bref aperçu.

Objets : les classes `Series` et `DataFrame` ou *table de données*.

Lire, écrire création et exportation de tables de données à partir de fichiers textes (séparateurs, `.csv`, format fixe, compressés), binaires (HDF5 avec `Pytable`), HTML, XML, JSON, MongoDB, SQL...

Gestion d'une table : sélection des lignes, colonnes, transformations, réorganisation par niveau d'un facteur, discrétisation de variables quantitatives, exclusion ou imputation élémentaire de données manquantes, permutation et échantillonnage aléatoire, variables indicatrices, chaînes de caractères...

Statistiques élémentaires uni et bivariées, tri à plat (nombre de modalités, de valeurs nulles, de valeurs manquantes...), graphiques associés, statistiques par groupe, détection élémentaire de valeurs atypiques...

Manipulation de tables : concaténations, fusions, jointures, tri, gestion des types et formats...

1.3 Références

Ce tutoriel élémentaire s'inspire largement du livre de référence (Mc Kinney, 2013)[1] et de la documentation en ligne à consulter sans modération. Cette documentation inclut également des tutoriels à exécuter pour compléter et approfondir la première ébauche d'un sujet relativement technique et qui peut prendre des tournures très diverses en fonction de la qualité et des types de données traitées.

1.4 Exemple

Les données choisies pour illustrer ce tutoriel sont issues d'une compétition du site Kaggle : Titanic:Machine learnic from Disaster. Le concours est terminé mais les données sont toujours disponibles sur le site avec des tutoriels utilisant Excel, Python ou R.

Une des raisons du drame, qui provoqua la mort de 1502 personnes sur les 2224 passagers et membres d'équipage, fut le manque de ca-

nots de sauvetage. Il apparaît que les chances de survie dépendaient de différents facteurs (sexe, âge, classe...). Le but du concours est de construire un modèle de prévision (classification supervisée) de survie en fonction de ces facteurs. Les données sont composées d'un échantillon d'apprentissage (891) et d'un échantillon test (418) chacun décrit par 11 variables dont la première indiquant la survie ou non lors du naufrage.

```
VARIABLE DESCRIPTIONS:
survival        Survival
                (0 = No; 1 = Yes)
pclass          Passenger Class
                (1 = 1st; 2 = 2nd; 3 = 3rd)
name            Name
sex             Sex
age             Age
sibsp           Number of Siblings/Spouses Aboard
parch           Number of Parents/Children Aboard
ticket          Ticket Number
fare            Passenger Fare
cabin           Cabin
embarked        Port of Embarkation
                (C = Cherbourg; Q = Queenstown; S = Southampton)
```

2 Les types `Series` et `DataFrame`

De même que la librairie `Numpy` introduit le type `array` indispensable à la manipulation de matrices en calcul scientifique, celle `pandas` introduit les classes `Series` (séries chronologiques) et `DataFrame` ou table de données indispensables en statistique.

2.1 `Series`

La classe `Series` est l'association de deux `arrays` unidimensionnels. Le premier est un ensemble de valeurs indexées par le 2ème qui est souvent une série temporelle. Ce type est introduit principalement pour des applications en Économétrie et Finance où Python est largement utilisé.

2.2 `DataFrame`

Cette classe est proche de celle du même nom dans le langage R, il s'agit d'associer avec le même index de lignes des colonnes ou variables de types différents (entier, réel, booléen, caractère). C'est un tableau bi-dimensionnel avec des index de lignes et de colonnes mais il peut également être vu comme une liste de `Series` partageant le même index. L'index de colonne (noms des variables) est un objet de type `dict` (dictionnaire). C'est la classe qui sera principalement utilisée dans ce tutoriel.

Comme pour le tutoriel précédent qui est un prérequis indispensable : Python pour Calcul Scientifique, exécuter les lignes de code une à une ou plutôt résultat par résultat, dans un calepin IPython ou Jupyter.

```
# Exemple de data frame
import pandas as pd
data = {"state": ["Ohio", "Ohio", "Ohio",
       "Nevada", "Nevada"],
     "year": [2000, 2001, 2002, 2001, 2002],
     "pop": [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = pd.DataFrame(data)
# ordre des colonnes
pd.DataFrame(data, columns=["year", "state", "pop"])
# index des lignes et valeurs manquantes (NaN)
frame2=pd.DataFrame(data, columns=["year", "state",
     "pop", "debt"],
     index=["one", "two", "three", "four", "five"])
# liste des colonnes
frame.columns
# valeurs d'une colonnes
frame["state"]
frame.year
# "imputation"
frame2["debt"] = 16.5
frame2
# créer une  variable
frame2["eastern"] = frame2.state == "Ohio"
frame2
```

```
# supprimer une  variable
del frame2[u"eastern"]
frame2.columns
```

2.3 Index

Les index peuvent être définis par emboîtement et beaucoup d'autres fonctionnalités sur la gestion des index sont décrites par Mac Kinney (2013)[1] (chapitre 5) :

`append` nouvel index par concaténation,

`diff` différence ensembliste,

`intersection` intersection ensembliste,

`union` union ensembliste

`isin` vrai si la valeur est dans la liste,

`delete` suppression de l'index i,

`drop` suppression d'une valeur d'index,

`is_monotonic` vrai si les valeurs sont croissantes,

`is_unique` vrai si toutes les valeurs sont différentes,

`unique` tableau des valeurs uniques de l'index.

3 Lire, écrire des tables de données

`Pandas` offre des outils efficaces pour lire écrire des fichiers selon différents formats (csv, texte, fixe, compressé, xml, html, hdf5) ou interagir avec des bases de données SQL, MongoDB, des APIs web. Ce document se contente de décrire les fonctions les plus utiles `read_csv` et `read_table` pour lire des fichiers textes et générer un objet de type `DataFrame`.

En principe ces fonctions font appel à un code écrit en C dont très rapide à l'exécution sauf pour l'emploi de certaines options (`skip_footer, sep` autre qu'un seul caractère), à éviter, qui provoquent une exécution en Python (`engine=Python`).

La réciproque pour l'écriture est obtenue par les commandes `data.to_csv` ou `_table` avec des options similaires.

3.1 Syntaxe

L'exemple de base est donné pour lire un fichier au format `.csv` dont les valeurs sont séparées par des "," et dont la première ligne contient le nom des variables.

```
# importation
import pandas as pd
data=pd.read_csv("fichier.csv")
# ou encore de façon équivalente
data=pd.read_table("fichier.csv", sep=",")
# qui utilise la tabulation comme
# séparateur par défaut
```

Il est important de connaître la liste des possibilités et options offertes par cette simple commande. Voici les principales ci-dessous et un lien à la liste complète.

`path` chemin ou non du fichier ou URL.

`sep` délimiteur comme `,` `;` `|` `\t` ou `\s+` pour un nombre variable d'espaces.

`header` défaut 0, la première ligne contient le nom des variables ; si `None` les noms sont générés ou définis par ailleurs.

`index_col` noms ou numéros de colonnes définissant les index de lignes, index pouvant être hiérarchisés comme les facteurs d'un plan d'expérience.

`names` si `header=None`, liste des noms des variables.

`nrows` utile pour tester et limiter le nombre de ligne à lire.

`skiprow` liste de lignes à sauter en lecture.

`skip_footer` nombre de lignes à sauter en fin de fichier.

`na_values` définition du ou des codes signalant des valeurs manquantes. Ils peuvent être définis dans un dictionnaire pour associer variables et codes de valeurs manquantes spécifiques.

`usecols` sélectionne une liste des variable à lire pour éviter de lire des champs ou variables volumineuses et inutiles.

`skip_blank_lines` à `True` pour sauter les lignes blanches.

`converters` appliquer une fonction à une colonne ou variable.

`day_first` par défaut `False`, pour des dates françaises au format `7/06/2013`.

`chunksize` taille des morceaux à lire itérativement.

`verbose` imprime des informations comme le nombre de valeurs manquantes des variables non numériques.

`encoding` type d'encodage comme `"utf-8"` ou `"latin-1"`

`thousand` séparateur des miliers : `"."` ou `","`.

Remarques :

- De nombreuses options de gestion des dates et séries ne sont pas citées.
- `chunksize` provoque la lecture d'un gros fichiers par morceaux de même taille (nombre de lignes). Des fonctions (comptage, dénombrement...) peuvent ensuite s'appliquer itérativement sur les morceaux.

3.2 Exemple

Les données du naufrage du Titanic illustrent l'utilisation de `pandas`. Elles sont directement lues à partir de leur URL ou sinon les charger ici vers le répertoire de travail de Python.

```
# Importations
import pandas as pd
import numpy as np
# tester la lecture
path="http://www.wikistat.fr/data/"
df = pd.read_csv(path+"titanic-train.csv",nrows=5)
print(df)
df.tail()
# tout lire
df = pd.read_csv(path+"titanic-train.csv")
df.head()
type(df)
df.dtypes
# Des variables sont inexploitables
# Choisir les colonnes utiles
df=pd.read_csv(path+"titanic-train.csv",
    usecols=[1,2,4,5,6,7,9,11],nrows=5)
df.head()
```

À partir de la version 0.15, `pandas`, inclut un type `category` assez proche de celui `factor` de R. Il devrait normalement être déclaré dans un dictionnaire au moment par exemple de la lecture (`dtype={"Surv":pd.Categorical...}`) mais ce n'est pas le cas, c'est donc le type objet qui est déclaré puis modifié. Il est vivement recommandé de bien affecter les bons types à chaque variable ne serait-ce que pour éviter de faire des opérations douteuses, par exemple arithmétiques sur des codes de modalités.

```
df=pd.read_csv(path+"titanic-train.csv",skiprows=1,
  header=None,usecols=[1,2,4,5,9,11],
  names=["Surv","Classe","Genre","Age",
    "Prix","Port"],dtype={"Surv":object,
    "Classe":object,"Genre":object,"Port":object})
df.head()
df.dtypes
```

Redéfinition des bons types.

```
df["Surv"]=pd.Categorical(df["Surv"],ordered=False)
df["Classe"]=pd.Categorical(df["Classe"],
    ordered=False)
df["Genre"]=pd.Categorical(df["Genre"],
    ordered=False)
df["Port"]=pd.Categorical(df["Port"],ordered=False)
df.dtypes
```

Remarque : il est également possible de tout lire avant de laisser "tomber" les variable inexploitables. C'est le rôle de la commande

```
df = df.drop(["Name", "Ticket", "Cabin"], axis=1)
```

3.3 Échantillonnage simple

Comme dans R, le type `DataFrame` de Python est chargé en mémoire. Si, malgré les options précédentes permettant de sélectionner, les colonnes, les

types des variables... le fichier est encore trop gros, il reste possible, avant de chercher une configuration matérielle lourde et en première approximation, de tirer un échantillon aléatoire simple selon une distribution uniforme. Un tirage stratifié demanderait plus de travail. Cela suppose de connaître le nombre de ligne du fichier ou une valeur inférieure proche.

```
# pour les données titanic:
N=891  # taille du fichier
n=200  # taille de l'échantillon
lin2skipe=[0] # ne pas lire la première ligne
# ne pas lire N-n lignes tirées aléatoirement
lin2skipe.extend(np.random.choice(np.arange(1,N+1),
    (N-n),replace=False))
df_small=pd.read_csv(path+"titanic-train.csv",
    skiprows=lin2skipe,header=None,
    usecols=[1,2,4,5,9,11],
    names=["Surv","Classe","Genre","Age",
         "Prix","Port"])
print(df_small)
```

4 Gérer une table de données

4.1 Discrétisation d'une variable quantitative

Pour la discrétisation d'une variable quantitative. Il est d'un bon usage de définir les bornes des classes à des quantiles, plutôt qu'également espacées, afin de construire des classes d'effectifs sensiblement égaux. Ceci est obtenu par la fonction `qcut`. La fonction `cut` propose par défaut des bornes équiréparties à moins de fournir une liste de ces bornes.

```
df["AgeQ"]=pd.qcut(df.Age,3,labels=["Ag1","Ag2",
   "Ag3"])
df["PrixQ"]=pd.qcut(df.Prix,3,labels=["Pr1","Pr2",
   "Pr3"])
df["PrixQ"].describe()
```

4.2 Modifier / regrouper des modalités

Le recodage des variables qualitatives ou renommage en clair des modalités est obtenu simplement.

```
df["Surv"]=df["Surv"].cat.rename_categories(
    ["Vnon","Voui"])
df["Classe"]=df["Classe"].cat.rename_categories(
    ["Cl1","Cl2","Cl3"])
df["Genre"]=df["Genre"].cat.rename_categories(
    ["Gfem","Gmas"])
df["Port"]=df["Port"].cat.rename_categories(
    ["Pc","Pq","Ps"])
```

Il est possible d'associer recodage et regroupement des modalités en définissant un dictionnaire de transformation.

```
data = pd.DataFrame({"food":["bacon","pulled pork",
   "bacon", "Pastrami",
   "corned beef", "Bacon", "pastrami", "honey ham",
   "nova lox"],
   "ounces": [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
data
```

Définition de la transformation :

```
meat_to_animal = {
    "bacon": "pig",
    "pulled pork": "pig",
    "pastrami": "cow",
    "corned beef": "cow",
    "honey ham": "pig",
    "nova lox": "salmon"
}
# Eviter les mélanges de majuscules minuscules
# en mettant tout en minuscule
data["animal"] = data["food"].map(
    str.lower).map(meat_to_animal)
data
```

Ou avec une seule fonction :

```
data["food"].map(lambda x: meat_to_animal[x.lower()])
```

4.3 Variables indicatrices

Générer des indicatrices des modalités ou *dummy variables*.

```
dfs = pd.DataFrame({"key": ["b", "b", "a", "c",
    "a", "b"],"data1": range(6)})
pd.get_dummies(dfs["key"])
```

```
dummies = pd.get_dummies(df['key'], prefix='key')
df_with_dummy = dfs[['data1']].join(dummies)
df_with_dummy
```

4.4 Permutation et tirage aléatoires

Permutation aléatoire :

```
dfs = pd.DataFrame(np.arange(5 * 4).reshape(5, 4))
sampler = np.random.permutation(5)
sampler
dfs
dfs.take(sampler)
```

Tirage aléatoire avec remplacement ou *bootstrap* ; celui sans remplacement est traité section 3.3.

```
bag = np.array([5, 7, -1, 6, 4])
sampler = np.random.randint(0, len(bag), size=10)
draws = bag.take(sampler)
draws
```

4.5 Transformations, opérations

Les opérations arithmétiques entre `Series` et `DataFrame` sont possibles au même titre qu'entre `array`. Si les index ne correspondent pas, des valeurs manquantes (NAN) sont créées à moins d'utiliser des méthodes d'arithmétique *flexible* (`add, sub, div, mul`) autorisant la complétion par une valeur par défaut, généralement 0.

Une fonction quelconque (`lambda`) peut être appliquée avec une même commande qu'`apply` de R.

```
# la table de données
frame = pd.DataFrame(np.random.randn(4,3),
    columns=list("bde"),
    index=["Utah", "Ohio", "Texas", "Oregon"])
# une fonction
f = lambda x: x.max() - x.min()
frame.apply(f, axis=1)
```

4.6 Tri et rangs

Trier une table selon les valeurs d'une variable ou d'un index.

```
frame = pd.DataFrame(np.arange(8).reshape((2,4)),
        index=["three", "one"],
        columns=["d", "a", "b", "c"])
frame.sort_index()
frame.sort_index(axis=1)
frame.sort_index(axis=1, ascending=False)
frame.sort_index(by="b")
frame.sort_index(by=["a", "b"])
```

La commande `rank` remplace les valeurs par leur rang dans l'ordre des lignes ou des colonnes.

```
frame = pd.DataFrame({"b": [4.3, 7, -3, 2],
    "a": [0, 1, 0, 1],"c": [-2, 5, 8, -2.5]})
frame.rank(axis=1)
frame.rank(axis=0)
```

5 Statistiques descriptives élémentaires

Continuer l'étude des données sur le naufrage du Titanic. Pour un utilisateur habitué à R, il est évidemment plus facile de passer directement à ce logiciel en

transformant (voir ci-dessous) le `DataFrame` Python en un `data frame` R. Néanmoins, l'analyse préalable nécessite éventuellement quelques manipulations supplémentaires et relectures avant de passer à R. Les commandes ci-dessous permettent des premiers diagnostics sur la qualité des données.

5.1 Description univariée

```
import pylab as P
# description univariée
df.dtypes
df.describe()
df["Age"].hist()
show()
df.boxplot("Age")
show()
df["Age"].plot(kind="box")
show()
df["Prix"].hist()
show()
df["Prix"].plot(kind="hist")
show()
# qualitatif
df["Surv"].value_counts()
df["Classe"].value_counts()
df["Genre"].value_counts()
df["Port"].value_counts()
```

5.2 Description bivariée

```
# Corrélationdf.corr()
# Nuage
df.plot(kind="scatter",x="Age",y="Prix")
show()
# afficher une sélection
df[df["Age"]>60][["Genre","Classe","Age","Surv"]]
# parallèle boxplots
df.boxplot(column="Age",by="Classe")
show()
df.boxplot(column="Prix",by="Surv")
show()
# table de contingence
table=pd.crosstab(df["Surv"],df["Classe"])
print(table)
# Mosaics plots
from statsmodels.graphics.mosaicplot import mosaic
mosaic(df,["Classe","Genre"])
show()
mosaic(df,["Surv","Classe"])
show()
```

5.3 Imputation de données manquantes

La gestion des données manquantes est souvent un point délicat. De nombreuses stratégies ont été élaborées, les principales sont décrites dans une vignette. Nous ne décrivons ici que les plus élémentaires à mettre en œuvre avec `pandas`.

Il est ainsi facile de supprimer toutes les observations présentant des données manquantes lorsque celles-ci sont peu nombreuses et majoritairement regroupées sur certaines lignes ou colonnes.

```
# les individus ou lignes
df = df.dropna(axis=0)
# les variables ou colonnes
df=df.dropna(axis=1)
```

`Pandas` permet également de faire le choix pour une variable qualitative de considérer `np.nan` comme une modalité spécifique ou d'ignorée l'observation correspondante.

Autres stratégies :

- Cas quantitatif : une valeur manquante est imputée par la moyenne ou la médiane.
- Cas d'une série chronologique : imputation par la valeur précédente ou suivante ou par interpolation linéaire, polynomiale ou encore lissage spline.

- Cas qualitatif : modalité la plus fréquente ou répartition aléatoire selon les fréquences observées des modalités.

C'est encore une fois le cas du type `Series` qui est le plus développé.

La variable âge contient de nombreuses données manquantes. La fonction `fillna` présente plusieurs options d'imputation.

```
# Remplacement par la médiane d'une variable
# quantitative
df=df.fillna(df.median())
df.describe()
# par la modalité "médiane" de AgeQ
df.info()
df.AgeQ=df["AgeQ"].fillna("Ag2")
# par le port le plus fréquent
df["Port"].value_counts()
df.Port=df["Port"].fillna("Ps")
df.info()
```

Ces imputations sont pour le moins très rudimentaires et d'autres approches sont à privilégier pour des modélisations plus soignées mais ces méthodes font généralement appel à R.

5.4 Autres

D'autres fonctions (Mac Kinney, 2013)[1] sont proposées pour supprimer les duplicatas (`drop_duplicates`), modifier les dimensions, traquer des atypiques unidimensionnels selon un modèle gaussien ou par rapport à des quantiles

6 Manipuler des tables de données

6.1 Jointure

Il s'agit de "jointer" deux tables partageant la même clef ou encore de concaténer horizontalement les lignes en faisant correspondre les valeurs d'une variable clef qui peuvent ne pas être uniques.

```
# tables
df1 = pd.DataFrame({"key": ["b", "b", "a", "c",
     "a","a", "b"],"data1": range(7)})
df2 = pd.DataFrame({"key": ["a", "b", "d"],
    "data2": range(3)})
pd.merge(df1,df2,on="key")
```

La gestion des clefs manquantes est en option : entre autres, ne pas introduire de ligne (ci-dessus), insérer des valeurs manquantes ci-dessous.

```
# valeurs manquantes
pd.merge(df1,df2,on="key", how="outer")
```

L'index de ligne peut être utilisé comme clef.

6.2 Concaténation selon un axe

Concaténation verticale (axis=0) ou horizontales (axis=1) de tables. La concaténation horizontale est similaire à la jointure (option `outer`).

```
# tables
df1 = pd.DataFrame({"key": ["b", "b", "a", "c",
    "a", "a", "b"],"var": range(7)})
df2 = pd.DataFrame({"key": ["a", "b", "d"],
    "var": range(3)})
# concaténation verticales
pd.concat([df1,df2],axis=0)
# concaténation horizontale
pd.concat([df1,df2],axis=1)
```

7 Trafic séquentiel de gros fichiers

Étape suivante associée de la croissance du volume : les fichiers des données brutes ne tiennent pas en mémoire. Il "suffit" d'intégrer ou enchâsser les étapes des sections précédentes dans la lecture séquentielle d'un gros fichier. En apparence, simple d'un point de vue méthodologique, cette étape peut consommer beaucoup de temps par tests et remises en cause incessantes des choix de sélection, transformation, recodage... des variables. Il est crucial de se doter d'outils efficaces.

Il s'agit donc de lire les données par morceau (nombre fixé de lignes) ou

ligne à ligne, traiter chaque morceau, le ré-écrire dans un fichier de format binaire plutôt que texte ; le choix du format HDF5 semble le plus efficace du point de vue technique et pour servir d'interface à d'autres environnements : C, java, Matlab... et R car une librairie (`rhdf5` de Bioconductor) gère ce format.

La procédure est comparable à une étape `Data` de SAS, qui lit/écrit les tables ligne à ligne.

Deux librairies :`h5py` et `PyTables` gèrent le format HDF5 en Python. Pour simplifier la tâche, `pandas` intègre une classe `HDFStore` utilisant `PyTables` qui doit donc être installée.

Attention, ce format n'est pas adapté à une gestion *parallélisée*, notamment en écriture.

7.1 Lecture séquentielle

L'exemple est ici donné pour lire un fichier texte mais beaucoup d'autres formats (excel, hdf, sql, json, msgpack, html, gbq, stata, clipboard, pickle) sont connus de `pandas`.

```
# importations
import pandas as pd
import numpy as np
# lire tout le fichier par morceaux
# avec l'option chunksize
Partition=pd.read_csv(path+"titanic-train.csv",
  skiprows=1,header=None,usecols=[1,2,4,5,9,11],
  names=["Surv","Classe","Genre","Age",
    "Prix","Port"],dtype={"Surv":object,
    "Classe":object,"Genre":object,"Port":object},
    chunksize=100)
# ouverture du fichier HDF5
stock=pd.HDFStore("titan.h5")
# boucle de lecture
for Part in Partition:
    # "nettoyage" préliminaire des données
    # ... autres opérations
    # création de la table "df" dans "stock" puis
    # extension de celle-ci par chaque "Part"
    stock.append("df",Part)
# dernier morceau lu et ajouté
Part.head()
# Il est généralement utile de fermer le fichier
stock.close()
```

Attention aux types implicites des variables. Si, par exemple, une donnée manquante n'apparaît pas dans une colonne du 1er morceau mais dans le 2ème, cela peut engendre un conflit de type. Expliciter systématiquement les types et noms des variables dans un dictionnaire en paramètre.

NB. Il est possible de stocker plusieurs tables ou objets dans le même fichier HDF5 puis d'en faire des recherches par sélection.

7.2 Analyses élémentaires séquentielles

Beaucoup de traitements nécessitent une connaissance statistique plus approfondies des variables, de leur distribution... Ce n'est pas réalisable en une seule passe mais gérable facilement en mettant en œuvre les fonctionnalités de *sélection* (tables, index lignes ou colonnes...) de `pandas` pour réaliser des analyses / transformations élémentaires sans charger la table en mémoire.

Objectif : aboutir à *LA* table qui servira à la modélisation.

Utilisation d'une table HDF5

```
# Ouverture du fichier
Archiv=pd.HDFStore("titan.h5")
# sélection de la table et affichage de l'entête
Archiv.select("df").head()
```

Cette partie est à développer pour illustrer les fonctionnalités de `pandas` permettant d'interroger / requêter (*querying* notamment SQL) une table archivée dans un fichier HDF5. Consulter la documentation en ligne à ce sujet.

En voici une première utilisation.

Échantillon aléatoire simple

Le fichier créé au format HDF5 peut être encore très volumineux. Par souci d'efficacité, son raffinement, son exploitation, voire même son analyse pour modélisation, peuvent ou même doivent être opérés sur un simple échantillon

aléatoire.

```
# extraction du nombre de lignes  / individus
nrows = Archiv.get_storer("df").nrows
# génération des index aléatoires
r = np.random.randint(0,nrows,size=10)
print(r)
# extraction des lignes d'index fixés
df_ech=Archiv.select("df",where=pd.Index(r))
df_ech
```

Il "suffit" alors d'appliquer les outils des sections 4 à 6 précédentes.

7.3 Échanges entre R et Pyhton

Les données ayant été préparées, nettoyées, le transfert de la table dans R permet de déployer toute la richesse des librairies développées dans cet environnement plus familier au statisticien. Il est possible d'appeler des commandes R à partir de Python avec la librairie `rpy2` et réciproquement d'appeler des commandes Python de R avec la librairie `rpython`.

La librairie `rpy2` définit la commande `load_data` qui charge un `data.frame` de R dans un `DataFrame` tandis que celle `convert_to_r_dataframe` génère un objet R.

Le plus efficace mais sans doute pas le plus simple, consisterait à lire directement, à partir de R, le fichier intermédiaire précédent au format binaire HDF5 en utilisant la librairie `rhdf5` de Bioconductor. Cette démarche pose des problèmes pour la gestions des variables qualitatives et plus généralement celle de la classe `DataFrame`. Une alternative simple consiste à construire un fichier au format classique `.csv`.

À suivre...

Ces traitements font appel à de très nombreuses opérations de lectures / écritures sur un seul ordinateur, un seul disque au regard du volume des calculs ; ils ne sont pas adaptés à une parallélisation sur un ordinateur multiprocesseur. La gestion et l'analyse de plus gros volumes de données nécessite une distribution de celles-ci sur plusieurs serveurs / disques. D'autres technologies doivent être utilisées ; c'est actuellement le couple *Spark/Hadoop* le plus en vogue.

Intérêt, *Spark* est utilisable avec java, Scala et aussi Python. L'investissement dans ce langage est donc rentable.

Références

[1] W. Mac Kinney, *Python for Data Analysis*, O'Reilly, 2013, http://it-ebooks.info/book/1041/.