Chapitre III Data Cleaning

Cours Master 1 IA - GL

Ilyas Bambrik

2024

Table des matières

Introduction

Le nettoyage des données est un élément clé de la science des données, mais il peut s'avérer extrêmement frustrant. Pourquoi certains valeurs sont-ils manquantes ? Que devez-vous faire à propos de ces valeurs manquantes ? Pourquoi vos dates ne sont-elles pas correctement formatées ? Comment pouvez-vous nettoyer rapidement les entrées de données incohérentes ? Dans ce cours, vous apprendrez comment résoudre ce type de problèmes.

Entrées manquantes

I

Regardons le nombre d'entrées vides du DataSet des jours de Transfermarket[3*]. L'instruction suivante donne le nombre d'entrées vides par colonne :

```
df=pd.read_csv('players.csv')
df.head()
```

	player_id	first_name	last_name	name	last_season	current_club_id	player_code	country_of_birth	city_of_birth
0	598	Timo	Hildebrand	Timo Hildebrand	2014	24	timo-hildebrand	Germany	Worms
1	670	Martin	Petrov	Martin Petrov	2012	714	martin-petrov	Bulgaria	Vratsa
2	1323	Martin	Amedick	Martin Amedick	2012	24	martin-amedick	Germany	Paderborn
3	3195	Jermaine	Pennant	Jermaine Pennant	2013	512	jermaine-pennant	England	Nottingham
4	3259	Damien	Duff	Damien Duff	2013	931	damien-duff	Ireland	Ballyboden

```
df.isnull().sum()
```

```
player_id                              0
first_name                          1965
last_name                              0
name                                   0
last_season                            0
current_club_id                        0
player_code                            0
country_of_birth                    2689
city_of_birth                       2203
country_of_citizenship               543
date_of_birth                         47
sub_position                         172
position                               0
foot                                2389
height_in_cm                        2098
market_value_in_eur                10919
highest_market_value_in_eur         1321
contract_expiration_date           11467
agent_name                         15361
image_url                              0
url                                    0
current_club_domestic_competition_id   0
current_club_name                      0
dtype: int64
```

Pour avoir le nombre total des cellules vides dans le DataFrame, il suffit de prendre la somme de ce résultat avec *.sum()* (df.isnull().sum().sum()). Dans cet exemple, *51174* entrées sont nulles. Par contre, comme mentionné dans le chapitre précédent, la méthode *count()* d'une colonne / Series renvoie le nombre de valeurs non-nulles dans celle-ci.

Les valeurs nulles sont plus faciles à trouver alors que les incohérences dans le DataFrame sont plus difficiles à reconnaître. Ceci nécessite d'analyser chaque colonne individuellement. Par exemple, regardant la colonne *height_in_cm*. La description statistique de cette colonne montre que la valeur minimale est égale à *18cm* ce qui est clairement une erreur. Ainsi, la méthode *describe()* est une méthode très facile à utiliser pour repérer des incohérences. Cependant, avant d'examiner les données, il nécessaire d'être sûr que les données sont dans le bon format et qu'ils sont traitées selon le type approprié. Par exemple, si une colonne comporte des valeurs numériques mais représentent une variable catégorique, celle-ci doit être convertie en une colonne catégorique. C'est le cas de la colonne *player_id*.

```
df.height_in_cm.describe()
```

```
count    28204.000000
mean       182.234577
std          6.833916
min         18.000000
25%        178.000000
50%        182.000000
75%        187.000000
max        207.000000
Name: height_in_cm, dtype: float64
```

Dans la prochaine étape, nous allons examiner de plus près certaines des colonnes avec des valeurs manquantes et si c'est possible de les remplacer.

C'est à ce moment-là que nous entrons dans la partie de la science liée à l'*intuition*, ce qui signifie « examiner réellement vos données et essayer de comprendre pourquoi elles sont ainsi et comment cela va affecter votre analyse ». Cela peut être une partie frustrante de la science des données, surtout si vous êtes nouveau dans le domaine et vous n'avez pas beaucoup d'expérience. Pour gérer les valeurs manquantes, vous devrez utiliser votre intuition pour comprendre pourquoi la valeur est manquante. L'une des questions les plus importantes que vous pouvez vous poser pour vous aider à comprendre cela est la suivante : *Cette valeur est-elle manquante parce qu'elle n'a pas été enregistrée ou parce qu'elle n'existe pas ?*

Si une valeur manque parce qu'elle n'existe pas (comme la taille de l'aîné d'une personne qui n'a pas d'enfants), cela n'a aucun sens d'essayer de deviner ce qu'elle pourrait être. Vous souhaitez probablement conserver ces valeurs sous forme de NaN. D'un autre côté, si une valeur est manquante parce que elle n'a pas été enregistrée, vous pouvez alors essayer de deviner de quoi elle aurait pu être en vous basant sur les autres valeurs de cette colonne et de cette ligne (prendre la moyenne, la médiane , le mode si la variable est de type catégorique, ou prédire la valeur en utilisant un modèle approprié).

Il est possible de simplement supprimer toutes les lignes ayant une cellule nulle avec la méthode *dropna()* :

```
df_1=df.dropna()
print(df_1.shape)
print(df.shape)
```

```
(8894, 23)
(30302, 23)
```

Après la suppression des lignes contenant des cellules nulles, environ de 20000 lignes ont été supprimés. Ceci représente une grande perte des données. Alternativement, il est possible de supprimer les colonnes contenant des valeurs nulles avec l'option axis=1. Avec l'instruction suivante, 12 des 23 colonnes initiales ont été supprimées. Par rapport à la suppression des lignes, plus de cellules ont été préservées avec la suppression des colonnes :

```
df_1=df.dropna(axis=1)
print(df_1.shape)
print(df.shape)
```

```
(30302, 11)
(30302, 23)
```

Regardons par exemple les colonnes *country_of_birth* et *city_of_birth* (2689 et 2203 respectivement). L'instruction suivante affiche les lignes avec une valeur nulle *country_of_birth* et une valeur non nulle de la colonne *city_of_birth*.

```
df[(df.country_of_birth.isnull()) & (df.city_of_birth.notnull())]
```

	player_id	first_name	last_name	name	last_season	current_club_id	player_code	country_of_birth	city_of_bi
289	67089	Juan	Cala	Juan Cala	2022	2687	juan-cala	NaN	Leb
346	88103	James	Rodríguez	James Rodríguez	2022	683	james-rodriguez	NaN	Cúc
386	99227	Andrea	Bertolacci	Andrea Bertolacci	2022	3205	andrea-bertolacci	NaN	Ro
422	116900	Simon	Hedlund	Simon Hedlund	2022	206	simon-hedlund	NaN	Trollhätt
469	131694	Olarenwaju	Kayode	Olarenwaju Kayode	2022	24245	olarenwaju-kayode	NaN	Ibac
...	...	...	...	...	...	...	...	...	
				Alperen			alperen-		

Il est évidant qu'il y a une relation entre la ville et le payé natale. Sachant le nom de la ville, il est possible de récupérer le pays avec plusieurs méthode. Le code suivant utilise Google pour récupérer les coordonnées de la ville. Ensuite, a partir des coordonnées , il est possible de récupérer le pays. Pour tester le code suivant, vous devez installer geopy (pip install geopy à partir de votre invité de commande Anaconda, ou bien installez le module à partir du navigateur)

```
from geopy.geocoders import Nominatim

def read_country(city):
    geolocator = Nominatim(user_agent="google")
    location = geolocator.geocode(city, language="en")
    country= location.address.split(',')[-1]
    return country
```

Alternativement, il est aussi possible de supprimer des lignes/colonnes avec la méthode *drop()*. Les arguments optionnels *columns* et/ou *index* permettent de spécifier les colonnes et/ou les lignes à supprimer. Comme d'autres méthodes vues précédemment, cette méthode revoie une copie modifiée du DataFrame (*inplace=True* modifier l'objet DataFrame invoquant).

L'exemple suivant supprime la ligne avec l'index 0 et 1 ainsi que les colonnes *country_of_citizenship, last_season* et *player_code*:

```
df.drop(index=[0,1],columns=["country_of_citizenship","last_season","player_code"],inplace=True)
df.head()
```

	player_id	first_name	last_name	name	current_club_id	country_of_birth	city_of_birth	date_of_birth	sub_position	position	foot	height_in_cm
2	1323	Martin	Amedick	Martin Amedick	24	Germany	Paderborn	1982-09-06	Centre-Back	Defender	NaN	NaN
3	3195	Jermaine	Pennant	Jermaine Pennant	512	England	Nottingham	1983-01-15	Right Winger	Attack	right	173.0
4	3259	Damien	Duff	Damien Duff	931	Ireland	Ballyboden	1979-03-02	Right Midfield	Midfield	left	177.0
5	3614	Tony	Hibbert	Tony Hibbert	29	England	Liverpool	1981-02-20	Right-Back	Defender	right	173.0
6	3804	Carlo	Nash	Carlo Nash	1123	England	Bolton	1973-09-13	Goalkeeper	Goalkeeper	right	197.0

Remplacement de valeurs manquantes

II

Une autre option consiste à essayer de remplir les valeurs manquantes. Nous pouvons utiliser la fonction fillna() de Panda pour remplacer les valeurs manquantes. Une option dont nous disposons est de spécifier par quoi nous voulons que les valeurs NaN soient remplacées. Dans l'instruction suivante, les valeurs manquantes de la colonne *height_in_cm* sont remplacées par la moyenne de celle-ci :

```
df.height_in_cm.fillna(df.height_in_cm.mean(),inplace=True)
df.height_in_cm.isnull().sum()
```

```
0
```

Il est aussi possible de spécifier les noms des colonnes avec les valeurs de remplacement de chacune comme un dictionnaire. Dans l'instruction suivante, les valeurs nulles de la colonne *first_name* sont remplacées par *Unkown* et celle de la colonne *foot* par *R*. Pour une colonne représentant une variable catégorique, les valeurs manquantes sont généralement remplacées par la valeur la plus fréquente (prendre la moyenne de la colonne n'a pas de sens) :

```
df.fillna({"first_name":"Unkown","foot":"R"})
```

Il existe aussi deux options de replacement permettant de remplacer une valeur nulle par la valeur non nulle précédente/suivante. Ceci est réalisable avec l'argument optionnel *method* égale à *bfill* (backward fill) *ffill* (forward fill). Le code suivant remplace les valeurs nulles de la colonne *city_of_birth* avec la valeur suivante non-nulle. Afin que l'application de cette méthode soit cohérent, les entrées doivent triés selon *country_of_birth et last_name* (il est plus probable que deux personnes ayant le même pays natal et prénom qu'ils soient de la même ville). Similairement, la méthode *bfill* remplace une valeur nulle avec la valeur non nulle précédente.

```
df.sort_values(["country_of_birth","last_name"]).city_of_birth.fillna(method=
 'bfill',inplace=True)
```

Alternativement, nous pouvons avoir une valeur non nulle que nous aimerions remplacer. Ceci est réalisable avec la méthode replace :

```
df.foot.replace("right","R").head(10)
```

```
0     NaN
1     NaN
2     NaN
3       R
4    left
5       R
6       R
7    left
8       R
9    both
```

```
11 Name: foot, dtype: object
```

La méthode *.replace()* permet aussi de remplacer plusieurs valeurs en définissant un dictionnaire contenant les valeurs à remplacer et comme elles sont remplacées. Le code suivant remplace "*right*" par "*R*", "*left*" par "*L*" et "*both*" par "*RL*":

```
1 df.foot.replace({"right":"R","left":"L","both":"RL"}).head(10)
```

```
1 0    NaN
2 1    NaN
3 2    NaN
4 3      R
5 4      L
6 5      R
7 6      R
8 7      L
9 8      R
10 9    RL
11 Name: foot, dtype: object
```

Remarque

Comme *sort_values, set_index* et d'autres méthodes précédentes, *fillna*/*dropna* retournent une copie modifiée du DataFrame/Series et peuvent être utilisée avec l'option *inplace=True.*

Traitement des données de type Date

En observant, la colonne *date_of_birth*, celle-ci contient des dates mais elle est considérée comme une chaîne de caractères. Le problème posé c'est qu'il n'est pas possible de trier selon l'ordre chronologique des dates dans cette colonne.

```
df.date_of_birth.head()
```

```
0    1979-04-05
1    1979-01-15
2    1982-09-06
3    1983-01-15
4    1979-03-02
Name: date_of_birth, dtype: object
```

La fonction *pandas.to_datetime()* offre un moyen pour convertir les cellules d'un objet *Series* ou *DataFrame* en objet date. L'instruction suivante renvoie une série avec des valeurs *datetime64* :

```
pd.to_datetime(df.date_of_birth).head()
```

```
0   1979-04-05
1   1979-01-15
2   1982-09-06
3   1983-01-15
4   1979-03-02
Name: date_of_birth, dtype: datetime64[ns]
```

Par ailleurs, il est possible de spécifier comment extraire la date avec l'argument optionnel *format*. Cet argument est une chaîne de caractère spécifiant la position et le format de l'année (%Y pour une année de 4 chiffres,%y pour 2 chiffres), le mois avec le %m et le jour avec %d. Pour l'exemple précédant, le format suivant est correcte (%Y-%m-%d):

```
pd.to_datetime(df.date_of_birth.head(),format="%Y-%m-%d")
```

En outre, il est facile de spécifier si le format de la date commence par le jour (avec *dayfirst=True*) ou bien commence avec l'année (avec *yearfirst=True*).

En cas d'erreur de conversion, le programme affichera une exception. La gestion des erreurs peut être personnalisée avec le paramètre optionnel *errors*. Dans l'exemple suivant, la série comporte une date erronée (troisième valeur "20x-12-2018"). Pour spécifier que les dates erronées doivent être remplacés par *NaT* (*Note a Timestamp*), *errors='coerce'* est utilisé :

```
pd.to_datetime(pd.Series(["16/11/2020","2022-10-6","20x-12-2018"]), errors=
 'coerce')
```

```
0   2020-11-16
```

```
1   2022-10-06
2          NaT
dtype: datetime64[ns]
```

Une dernière option intéressante est l'argument *infer_datetime_format=True* permettant de déduire le format selon toutes les valeurs de la colonne avant de procéder à la conversion. Cette option est utile pour détecter automatiquement le format de la date dans la colonne mais nécessite plus de temps d'exécution.

Après la conversion, il est possible d'accéder aux propriétés de l'objet datetime64. Pour ceci, nous utilisons l'attribut *dt* (*datetime*) de l'objet Series avant d'accéder aux propriétés et méthodes de chaque cellule comme étant un objet *datetime*.

Dans le code suivant, la série est convertie en une série *datetime64* et affectée à la variable *s* (*ligne 1*). Puis pour accéder à la propriété *day* (jour du mois) de chaque cellule, *.dt.day* est utilisé.

L'attribut *dt* de l'objet Series est appelée *accesseur* et permet d'accéder à des méthodes/attributs spécifiques aux objets de types datetime.

```
s=pd.to_datetime(pd.Series(["16/11/2020","06/09/2022","20/12/2018"]) )
s.dt.day<10
```

```
0    False
1     True
2    False
dtype: bool
```

Nous pouvons accéder à plusieurs attributs tels que le jours de la semaine et le nombre de jours dans le mois désigné par la date.

```
s.dt.day_name()
```

```
0      Monday
1    Thursday
2    Thursday
dtype: object
```

```
s.dt.days_in_month
```

```
0    30
1    30
2    31
dtype: int64
```

Accesseurs

IV

Comme l'accesseur *dt*, les chaînes de caractères ont un accesseur *str* permettant d'accéder aux méthodes de traitement d'une chaîne de caractères ordinaire python. Regardons la colonne *name* de *palyers.csv*. Par exemple, la méthode *lower()* de l'accesseur *str* transforme la chaîne de caractères en texte minuscule.

```
df.name
```

```
0           Timo Hildebrand
1             Martin Petrov
2            Martin Amedick
3          Jermaine Pennant
4               Damien Duff
                ...
30297            Jaka Bijol
30298    Semuel Pizzignacco
30299         Festy Ebosele
30300        Nicolò Cocetta
30301         Axel Guessand
Name: name, Length: 30302, dtype: object
```

```
df.name.str.lower()
```

```
0           timo hildebrand
1             martin petrov
2            martin amedick
3          jermaine pennant
4               damien duff
                ...
30297            jaka bijol
30298    semuel pizzignacco
30299         festy ebosele
30300        nicolò cocetta
30301         axel guessand
Name: name, Length: 30302, dtype: object
```

Comme lower(), upper() transforme le texte en majuscule.

```
df.name.str.upper()
```

```
0           TIMO HILDEBRAND
1             MARTIN PETROV
2            MARTIN AMEDICK
3          JERMAINE PENNANT
4               DAMIEN DUFF
                ...
30297            JAKA BIJOL
30298    SEMUEL PIZZIGNACCO
30299         FESTY EBOSELE
```

```
30300        NICOLÒ COCETTA
30301         AXEL GUESSAND
Name: name, Length: 30302, dtype: object
```

Les méthodes suivantes sont aussi importantes :

- *contains() :* pour tester l'existence d'une chaîne de caractère à l'intérieur de la chaîne de la cellule ;
- *startswith()/endswith() :* teste si la cellule commence/se termine par la chaîne de caractères en paramètre ;
- *strip() :* supprimer les espaces au début et à la fin de la chaîne (*lstrip* pour supprimer seulement les espaces au début, *rstrip* pour supprimer à la fin);
- *replace() :* remplace les occurrences d'une chaîne par une autre dans le texte;
- *split() :* découpe une chaîne selon un séparateur et retourne une liste ;
- *title() :* transforme le texte tel que chaque mot commence par une majuscule ;

```
df.name.str.split(" ")
```

```
0           [Timo, Hildebrand]
1             [Martin, Petrov]
2            [Martin, Amedick]
3          [Jermaine, Pennant]
4               [Damien, Duff]
                ...
30297            [Jaka, Bijol]
30298    [Semuel, Pizzignacco]
30299          [Festy, Ebosele]
30300        [Nicolò, Cocetta]
30301         [Axel, Guessand]
Name: name, Length: 30302, dtype: object
```

```
df.name.str.replace(" ","-")
```

```
0           Timo-Hildebrand
1             Martin-Petrov
2            Martin-Amedick
3          Jermaine-Pennant
4               Damien-Duff
                ...
30297            Jaka-Bijol
30298    Semuel-Pizzignacco
30299          Festy-Ebosele
30300        Nicolò-Cocetta
30301         Axel-Guessand
Name: name, Length: 30302, dtype: object
```

Si on souhaite de transformer la chaîne de caractères en minuscule avec *lower()* avant d'utiliser la méthode *split()* , nous devons utiliser l'accesseur chaque fois que la méthode spécifique aux chaînes de caractères est invoquée :

```
df.name.str.lower().str.split()
```

```
0           [timo, hildebrand]
1             [martin, petrov]
2            [martin, amedick]
3          [jermaine, pennant]
4               [damien, duff]
                ...
30297            [jaka, bijol]
30298    [semuel, pizzignacco]
```

```
30299          [festy, ebosele]
30300        [nicolò, cocetta]
30301          [axel, guessand]
Name: name, Length: 30302, dtype: object
```

Dans le DataCleaning, les méthodes de l'accesseur *str* sont particulièrement utiles pour uniformiser la façon avec la quelle les valeurs sont écrites.

Les données catégoriques obtenu avec *astype("category")* sont très similaires aux données obtenues avec *astype (str)*. Cependant, une série de type *category* possède l'accesseur *.cat*, au lieu de *.str*, qui permet d'utiliser à une suite de méthodes complètement différentes.

Prenant le DataSet *wr2023.csv* en ignorant les universités avec la colonne Location égale à nulle. Prenant seulement les 30 premières lignes :

```
df=pd.read_csv("wr2023.csv")
df=df[df.Location.notnull()].iloc[:30,:]
```

Initialement, les valeurs de la colonne *Location* sont traitées comme des chaînes de caractères. Transformant cette colonne en colonne de type *category* pour explorer les méthodes proposées par l'accesseur de celle-ci :

```
df.Location=df.Location.astype("category")
df.Location
```

```
0     United Kingdom
1      United States
2     United Kingdom
3      United States
4      United States
5      United States
6      United States
7      United States
8      United States
9     United Kingdom
10     United States
11       Switzerland
12     United States
13     United States
14     United States
17            Canada
19     United States
20     United States
21    United Kingdom
22     United States
23     United States
24     United States
25     United States
26     United States
27     United States
28    United Kingdom
31     United States
33         Australia
34    United Kingdom
35         Singapore
Name: Location, dtype: category
Categories (6, object): [Australia, Canada, Singapore, Switzerland, United
Kingdom, United States]
```

Maintenant, l'accesseur *.cat* donne accès aux valeurs uniques de la catégorie avec l'attribut *.categories* :

```
df.Location.cat.categories
```

```
Index(['Australia', 'Canada', 'Singapore', 'Switzerland', 'United Kingdom',
       'United States'],
      dtype='object')
```

La transformation suivante prend la listes des catégories de la colonne Location et retourne une liste contenant seulement les deux premiers caractères du nom de chaque pays :

```
def abreviation(nom_pays):
    if len(nom_pays.split())==1: return nom_pays[:2]
    return "".join([*map(lambda x:x[0],nom_pays.split())])
pays=[*map(abreviation,list(df.Location.cat.categories))]
pays
```

```
['Au', 'Ca', 'Si', 'Sw', 'UK', 'US']
```

Pour remplacer les valeurs d'une colonne nous avons déjà explorer la méthode *replace*. Dans le cas d'une colonne catégorique, une meilleur façon pour remplacer toutes les valeurs est d'utiliser la méthode *.cat.rename_categories*. Celle-ci prend en paramètre les nouvelles nominations des catégories selon l'ordre dans lequel ils sont listés dans *.cat.categories*. Le code suivant remplace les noms des catégories par les abréviations dans la liste "*pays*" :

```
df.Location.cat.rename_categories(pays,inplace=True)
df.Location.head()
```

```
0     UK
1     US
2     UK
3     US
4     US
Name: Location, dtype: category
Categories (6, object): [Au, Ca, Si, Sw, UK, US]
```

La méthode *rename_categories()* offre aussi une syntaxe similaire à replace() qui prend un dictionnaire comme paramètre définissant les catégories à remplacer :

```
df.Location.cat.rename_categories({"UK":"ENG","US":"USA"}).head()
```

```
0    ENG
1    USA
2    ENG
3    USA
4    USA
Name: Location, dtype: category
Categories (6, object): [Ca, Si, Sw, USA, ENG, Au]
```

L'ordre dans lequel les catégories sont placées dans *.cat.categories* définie comment ils sont classées lors de la comparaison/trie. Dans l'exemple précédant, une valeur de catégorie *"UK*" est plus petite que la valeur de catégorie *"US"*. Après le trie de la colonne Location, le résultat obtenu sera le suivant :

```
df.Location.sort_values(ascending=False).head()
```

```
14    US
3     US
10    US
```

```
6     US
12    US
Name: Location, dtype: category
Categories (6, object): [Au, Ca, Si, Sw, UK, US]
```

La méthode *reorder_categories()* permet de réordonner les catégories selon l'ordre des valeurs dans la liste en paramètre. Le code suivant réordonne les valeurs de la catégorie pour que *"UK"* représente une catégorie supérieure à *"US"* et "Au" soit considérée comme la plus grande valeur dans la catégorie:

```
df.Location.cat.reorder_categories([ "Ca", "Si", "Sw", "US", "UK","Au"],inplace=
True)
df.Location.sort_values(ascending=False).head()
```

```
33    Au
0     UK
28    UK
2     UK
21    UK
Name: Location, dtype: category
Categories (6, object): [Ca, Si, Sw, US, UK, Au]
```

Pour afficher l'ordre actuelle des catégories, la méthodes *as_ordered()* est utilisée (voir la dernière ligne de l'affichage suivant) :

```
df.Location.cat.as_ordered().head()
```

```
0    UK
1    US
2    UK
3    US
4    US
Name: Location, dtype: category
Categories (6, object): [Ca < Si < Sw < US < UK < Au]
```

Pour supprimer une ou plusieurs catégories d'une colonne, la méthode *remove_categories()* est utilisée qui prend en paramètre la liste des catégories à supprimer. Les cellules ayant une valeur comprise dans les catégories supprimées deviennent nulles. L'exemple suivant supprime la catégorie "*UK*" de la colonne :

```
df.Location.cat.remove_categories(["UK"]).head()
```

```
0    NaN
1     US
2    NaN
3     US
4     US
Name: Location, dtype: category
Categories (5, object): [Ca, Si, Sw, US, Au]
```

La méthode *add_categories()* permet d'ajouter de nouvelles catégories :

```
df.Location.cat.add_categories(["Rus","UAE"],inplace=True)
df.Location
```

Pour clôturer cette section, la méthode *remove_unused_categories()* permet de supprimer les catégories non utilisées dans la colonne :

```
df.Location.cat.remove_unused_categories(inplace=True)
df.Location.head()
```

```
0    UK
1    US
2    UK
3    US
4    US
Name: Location, dtype: category
Categories (6, object): [Ca, Si, Sw, US, UK, Au]
```

Uniformisation des entrées texte incohérentes

V

Dans cette partie, nous allons apprendre à nettoyer les entrées de texte incohérentes. Prenons le DataFrame qui représente le capitale intellectuelle du Pakistan[4*] :

```
df=pd.read_csv("pakistan_intellectual_capital.csv")
df.head()
```

	Unnamed: 0	S#	Teacher Name	University Currently Teaching	Department	Province University Located	Designation	Terminal Degree	Graduated from	Country	Year	Area of Specialization/Research Interests	Other Information
0	2	3	Dr. Abdul Basit	University of Balochistan	Computer Science & IT	Balochistan	Assistant Professor	PhD	Asian Institute of Technology	Thailand	NaN	Software Engineering & DBMS	NaN
1	4	5	Dr. Waheed Noor	University of Balochistan	Computer Science & IT	Balochistan	Assistant Professor	PhD	Asian Institute of Technology	Thailand	NaN	DBMS	NaN
2	5	6	Dr. Junaid Baber	University of Balochistan	Computer Science & IT	Balochistan	Assistant Professor	PhD	Asian Institute of Technology	Thailand	NaN	Information processing, Multimedia mining	NaN
3	6	7	Dr. Maheen Bakhtyar	University of Balochistan	Computer Science & IT	Balochistan	Assistant Professor	PhD	Asian Institute of Technology	Thailand	NaN	NLP, Information Retrieval, Question Answering...	NaN
4	24	25	Samina Azim	Sardar Bahadur Khan Women's University	Computer Science	Balochistan	Lecturer	BS	Balochistan University of Information Technolo...	Pakistan	2005.0	VLSI Electronics DLD Database	NaN

Supposons que nous souhaitions nettoyer la colonne '*Country*' pour nous assurer qu'elle ne contient aucune incohérence dans la saisie des données.

```
sorted(df.Country.unique())
```

```
[' Germany', ' New Zealand', ' Sweden', ' USA', 'Australia', 'Austria', 'Canada',
 'China', 'Finland', 'France', 'Greece', 'HongKong', 'Ireland', 'Italy', 'Japan',
 'Macau', 'Malaysia', 'Mauritius', 'Netherland', 'New Zealand', 'Norway', 'Pakistan',
 'Portugal', 'Russian Federation', 'Saudi Arabia', 'Scotland', 'Singapore', 'South
Korea', 'SouthKorea', 'Spain', 'Sweden', 'Thailand', 'Turkey', 'UK', 'USA', 'USofA',
 'Urbana', 'germany']
```

En regardant les valeurs uniques de la colonne, des problèmes dus à la saisie de données sont bien visibles (du texte écrit en minuscule et d'autres commençant par un espace) et cela entraîne des erreurs lors de la sélection et traitement des données.

La première chose à faire est de transformer le texte de toutes les cellules en minuscules et de supprimer les espaces blancs au début et à la fin des cellules. Les incohérences dans les majuscules et les espaces blancs à la fin/début sont très récurrentes dans les données texte et vous pouvez corriger 80 % de vos incohérences de saisie en procédant ainsi. Il est aussi complètement correcte de transformer le texte de la colonne en majuscule, ou en minuscule puis en titlecase.

```
df.Country=df.Country.str.lower().str.strip()
sorted(df.Country.unique())
```

```
['australia', 'austria', 'canada', 'china', 'finland', 'france', 'germany',
'greece', 'hongkong', 'ireland', 'italy', 'japan', 'macau', 'malaysia', 'mauritius',
'netherland', 'new zealand', 'norway', 'pakistan', 'portugal', 'russian federation',
'saudi arabia', 'scotland', 'singapore', 'south korea', 'southkorea', 'spain',
'sweden', 'thailand', 'turkey', 'uk', 'urbana', 'usa', 'usofa']
```

La plus part des noms des payés semblent correctement formulés mis à part 'south korea' et 'southkorea'. Pour rectifier ceci, nous allons utiliser le module *fuzzywuzzy* permettant de repérer des chaînes de cratères similaires à un texte recherché. Afin d'utiliser *fuzzywuzzy* , vous devez installer le module dans votre environnement Anaconda :

```
pip install git+git://github.com/seatgeek/fuzzywuzzy.git@0.18.0#egg=fuzzywuzzy
```

```
fuzzywuzzy.process.extract("south korea", df.Country, limit=10, scorer=fuzzywuzzy.
fuzz.token_sort_ratio)
```

```
[('south korea', 100),
 ('southkorea', 48),
 ('saudi arabia', 43),
 ('norway', 35),
 ('austria', 33),
 ('ireland', 33),
 ('pakistan', 32),
 ('portugal', 32),
 ('scotland', 32),
 ('australia', 30)]
```

- L'argument *limit* de la fonction *extract* permet de spécifier le nombre de valeurs à afficher. Par ailleurs, l'argument *scorer* spécifie la fonction de comparaison de similitude entre le texte en paramètre et les valeurs de la série. Le résultat est une liste classée selon le score de similarité compris entre 0-100 où une valeur plus large indique une plus grand similarité.
- Plusieurs fonctions de mesure de similarité entre textes sont fournis par la bibliothèque (*fuzz.partial_ratio*, *fuzz.token_sort_ratio et fuzz.token_set_ratio*) basées sur l'algorithme Edit Distance (Distance de Levenshtein).

Encodage de caractères

VI

Un encodage de caractères (ASCII, UTF-8, UTF-16) est un ensemble spécifique de règles permettant de mapper des chaînes d'octets binaires brutes aux caractères qui composent un texte lisible par l'être humain. Il existe de nombreux encodages différents, et si vous essayez de lire un texte avec un encodage différent de celui dans lequel il a été écrit à l'origine, vous vous retrouvez avec un texte brouillé appelé '*mojibake*'. Voici un exemple de mojibake :

æ–‡å—åŒ–ã??

Vous pourriez également vous retrouver avec des caractères "inconnus". C'est ce qui est imprimé lorsqu'il n'y a pas de mappage entre une suite de bits particulière et un caractère dans l'encodage que vous utilisez pour lire votre chaîne d'octets et le résultat ressemble à ceci :

□□□□□□□□□□

La plupart des fichiers que vous rencontrerez seront probablement encodés en UTF-8. C'est ce que Python attend par défaut, donc la plupart du temps vous n'aurez pas de problème. Cependant, vous obtiendrez parfois une erreur comme celle-ci[5*] :

```
pd.read_csv('ks-projects-201612.csv')
```

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x99 in position 7955:
 invalid start byte
```

L'exception *UnicodeDecodeError* indique que le contenu du fichier n'est pas encodé avec *UTF-8*. Cependant, nous ne savons pas de quel encodage il s'agit réellement. Une façon de découvrir l'encodage est d'essayer de tester des encodages de caractères différents et de voir si l'un d'entre eux fonctionne. Une meilleure façon, cependant, consiste à utiliser le module *charset_normalizer* pour essayer de deviner automatiquement quel est le bon encodage. Le résultat n'est pas garanti à 100%, mais c'est généralement plus rapide que d'essayer de deviner.

Nous allons tester les dix milles premiers octets de ce fichier. Cela suffit généralement pour avoir une bonne idée de l'encodage et car c'est beaucoup plus rapide que d'essayer de regarder l'intégralité du fichier. Surtout avec un fichier volumineux, cela peut être très lent. Une autre raison pour examiner les 1000 premiers octets du fichier est que nous pouvons voir, en regardant le message d'erreur, que le premier problème est le *7955*em caractère. La méthode *charset_normalizer.detect* renvoie un dictionnaire qui représente l'encodage détecté.

```
import charset_normalizer
with open("ks-projects-201612.csv", 'rb') as sequence_octets:
    resultat= charset_normalizer.detect(sequence_octets.read(10000))
print(resultat)
```

```
{'encoding': 'windows-1250', 'language': 'English', 'confidence': 1.0}
```

Lors de la lecture avec la méthode *read_csv*, l'option *encoding* permet de spécifier l'encodage lors de la lecture du fichier.

```
df=pd.read_csv('ks-projects-201612.csv',encoding='Windows-1252')
df.head()
```

	ID	name	category	main_category	currency	deadline	goal	launched	pledged	state	backers	country	usd pledged	Unnamed: 13
0	1000002330	The Songs of Adelaide & Abullah	Poetry	Publishing	GBP	2015-10-09 11:36:00	1000	2015-08-11 12:12:28	0	failed	0	GB	0	NaN
1	1000004038	Where is Hank?	Narrative Film	Film & Video	USD	2013-02-26 00:20:50	45000	2013-01-12 00:20:50	220	failed	3	US	220	NaN
2	1000007540	ToshiCapital Rekordz Needs Help to Complete Album	Music	Music	USD	2012-04-16 04:24:11	5000	2012-03-17 03:24:11	1	failed	1	US	1	NaN
3	1000011046	Community Film Project: The Art of Neighborhoo...	Film & Video	Film & Video	USD	2015-08-29 01:00:00	19500	2015-07-04 08:35:03	1283	canceled	14	US	1283	NaN
4	1000014025	Monarch Espresso Bar	Restaurants	Food	USD	2016-04-01 13:38:27	50000	2016-02-26 13:38:27	52375	successful	224	US	52375	NaN

Transformation de données catégoriques

VII

Parfois, la colonne de notre DataFrame contient des valeurs numériques qui encodent des catégories. Pour spécifier que une colonne est de type catégorique, il suffit de convertir ses valeurs en type '*category*'. Par exemple, regardons le DataSet *players.csv*. La colonne *current_club_id* représente une catégorie même si elle contient des valeurs numériques (*int64*).

```
df=pd.read_csv("players.csv")
df.dtypes
```

```
player_id                                int64
first_name                              object
last_name                               object
name                                    object
last_season                              int64
current_club_id                          int64
player_code                             object
country_of_birth                        object
city_of_birth                           object
country_of_citizenship                  object
date_of_birth                           object
sub_position                            object
position                                object
foot                                    object
height_in_cm                           float64
market_value_in_eur                    float64
highest_market_value_in_eur            float64
contract_expiration_date                object
agent_name                              object
image_url                               object
url                                     object
current_club_domestic_competition_id    object
current_club_name                       object
dtype: object
```

Ainsi, si nous souhaitons avoir un résumé statistique avec *describe()* de *current_club_id*, nous aurons la moyenne et la distribution des valeurs selon les centiles.

```
df.current_club_id.describe()
```

```
count    30302.000000
mean      4366.055574
std      10056.373140
min          3.000000
25%        403.000000
50%       1071.000000
```

```
75%       3008.000000
max      83678.000000
Name: current_club_id, dtype: float64
```

Par contre, en convertissant la colonne en type '*category*', la méthode *describe()* produira une sortie plus convenable :

```
df.current_club_id=df.current_club_id.astype('category')
df.current_club_id.describe()
```

```
count     30302
unique      424
top        2553
freq        177
Name: current_club_id, dtype: int64
```

En outre, il est parfois nécessaire de diviser des valeurs numériques sur des bacs / intervalles pour transformer celle-ci en colonne catégorique. Pour ceci, deux méthodes existent : *qcut* et *cut*. Pour expliquer ces deux méthodes, prenons la colonnes *height_in_cm*. L'affichage de la méthode *describe()*, indique que :

- La moyenne de cette colonne est égale à 182.234577 ;
- 25% d'échantillons dans cette colonne sont inférieurs à 178.0 ;
- 50% d'échantillons dans cette colonne sont inférieurs à 182.0 ;
- 75% d'échantillons dans cette colonne sont inférieurs à 187.0 ;
- 100% d'échantillons dans cette colonne sont inférieurs ou égales au maximum, 207.0 ;

```
df.height_in_cm.describe()
```

```
count    28204.000000
mean       182.234577
std          6.833916
min         18.000000
25%        178.000000
50%        182.000000
75%        187.000000
max        207.000000
```

La méthode *qcut* permet de diviser une colonne de valeurs numériques sur plusieurs intervalles tel que chaque intervalle contient presque le même nombre d'échantillons. Par exemple, si nous souhaitons diviser les valeurs de la colonne sur 4 intervalles contenant presque le même nombre d'échantillons (comme dans le résultat de *describe()*), ci-dessous le code avec *pd.qcut* pour le faire. L'argument *q* permet de spécifier le nombre d'intervalles :

```
pd.qcut(df.height_in_cm,q=4)
```

```
0                    NaN
1                    NaN
2                    NaN
3        (17.999, 178.0]
4        (17.999, 178.0]
              ...
30297     (187.0, 207.0]
30298     (187.0, 207.0]
30299     (178.0, 182.0]
30300                NaN
30301     (182.0, 187.0]
Name: height_in_cm, Length: 30302, dtype: category
```

```
Categories (4, interval[float64, right]): [(17.999, 178.0] < (178.0, 182.0] <
(182.0, 187.0] < (187.0, 207.0]]
```

```
pd.qcut(df.height_in_cm,q=4).value_counts()
```

```
(17.999, 178.0]    8528
(182.0, 187.0]     7642
(187.0, 207.0]     6423
(178.0, 182.0]     5611
Name: height_in_cm, dtype: int64
```

Il est aussi possible de spécifier les centiles des rangés avec le paramètre *q*. Le code suivant divise les valeurs *height_in_cm* tel que chaque intervalle comporte presque 33% des échantillons :

```
pd.qcut(df.height_in_cm,q=[0,.33,.66,1]).value_counts()
```

```
(179.0, 185.0]     9583
(17.999, 179.0]    9516
(185.0, 207.0]     9105
Name: height_in_cm, dtype: int64
```

La méthode *cut* en revanche est similaire à *qcut* mais divise l'intervalle entière des échantillons (*[min,max]*) sur des intervalles de même taille. Le paramètre *bins*, comme *q* pour *qcut*, spécifie le nombre d'intervalles :

```
pd.cut(df.height_in_cm,bins=4).value_counts()
```

```
(159.75, 207.0]    28200
(17.811, 65.25]        2
(112.5, 159.75]        2
(65.25, 112.5]         0
Name: height_in_cm, dtype: int64
```

Similairement au paramètre *q* de *pcut*, il est possible de définir les frontières des intervalles avec le paramètre *bins* :

```
pd.cut(df.height_in_cm,bins=[df.height_in_cm.min(), 175,185,205]).value_counts()
```

```
(175.0, 185.0]    14179
(185.0, 205.0]     9100
(18.0, 175.0]      4918
Name: height_in_cm, dtype: int64
```

Le résultat de *qcut* et *cut* est la liste des intervalles dans les quelles les valeurs appartiennent. Il est possible d'encoder les intervalles numériquement dans le résultat avec l'argument *labels=False*. Ceci indique que le résultat doit être encodé numériquement au lieu d'avoir l'intervalle correspondante.

```
pd.qcut(df.height_in_cm,q=4, labels=False)
```

```
0        NaN
1        NaN
2        NaN
3        0.0
4        0.0
        ...
30297    3.0
30298    3.0
30299    1.0
30300    NaN
30301    2.0
```

```
12 Name: height_in_cm, Length: 30302, dtype: float64
```

Il est aussi possible de sélectionner les colonnes d'un DataFrame selon le type avec la méthode *select_dtypes()* et le paramètre optionnel *include*. Après la conversion des colonnes *current_club_id* et *date_of_birth* en variables catégorique et date respectivement, l'exemple suivant sélectionne toutes les colonnes de type *category* et *datetime* DataFrame *players.csv*:

```
1 df.current_club_id=df.current_club_id.astype('category')
2 df.date_of_birth=pd.to_datetime(df.date_of_birth)
3 df.select_dtypes(include=["datetime","category"]).head()
```

	current_club_id	date_of_birth
0	24	1979-04-05
1	714	1979-01-15
2	24	1982-09-06
3	512	1983-01-15
4	931	1979-03-02

Inversement au paramètre *include*, *exlude* permet d'omettre des colonnes de types spécifiques. Le code suivant exclu les colonnes de type objet :

```
1 df.select_dtypes(exclude=["object"]).head()
```

	player_id	last_season	current_club_id	height_in_cm	market_value_in_eur	highest_market_value_in_eur
0	598	2014	24	NaN	NaN	10000000.0
1	670	2012	714	NaN	NaN	12000000.0
2	1323	2012	24	NaN	NaN	2750000.0
3	3195	2013	512	173.0	NaN	10500000.0
4	3259	2013	931	177.0	NaN	17000000.0

Lecture de données depuis des source alternatives

VIII

Lors de la lecture d'un DataFrame, *read_csv()* permet de spécifier le séparateur utilisé pour séparer les valeurs avec le paramètre optionnel *sep*. En outre, au lieu d'utiliser le chemin vers le fichier à lire, il est possible de lire un fichier à partir d'un URL http.

La fonction *pandas.read_excel*() permet de lire un DataFrame à partir d'un fichier excel :

```
df_can = pd.read_excel("Canada.xlsx")
```

Comme *read_csv()*, la méthode *read_excel()* permet aussi de lire un DataFrame depuis un lien http en passant le l'URI de celle-ci. Inversement, la méthode *DataFrame.to_excel()* permet d'enregistrer un DataFrame dans un *fichier xlsx*.

```
df=pd.read_excel("'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.
cloud/IBMDeveloperSkillsNetwork-DV0101EN-SkillsNetwork/Data%20Files/Canada.xlsx")
```

Pandas permet aussi de créer un DataFrame à partir d'une table Sql. Pour commencer, il est nécessaire d'installer le pilote Sql et le SGBD approprié. Dans les exemples suivants nous allons utiliser MySql. Le connecteur[6*] nécessaire pour interfacer avec MySql depuis python est installé avec la commande suivante :

```
pip install mysql-connector-python
```

- Le code suivant commence par importer le connecteur (*mysql.connector*) et puis configure une connexion à la base de données.
- La méthode *cursor()* retourne un objet de la classe *cursor* permettant d'exécuter des requêtes Sql et de collecter les résultats.
- La méthode *fetchall()* retourne toutes les enregistrements après l'exécution d'une requête SELECT.

```
import mysql.connector

bdd = mysql.connector.connect(
  host="localhost",
  user="root",
  password="root",
  database="table1"
)
curseur = bdd.cursor()
curseur .execute("SELECT * FROM student_score")
resultat = curseur .fetchall()
for ligne in resultat:
  print(ligne)
```

```
(1, 'Tomiwa', 'Microbiology', 65)
(2, 'Yusuf', 'Biochemistry', 70)
(3, 'Habeebah', 'Microbiology', 80)
(4, 'Gbadebo', 'Computer Science', 80)
(5, 'Muritadoh', 'Biochemistry', 85)
(7, 'Taiwo', 'Microbiology', 76)
(8, 'Joel', 'Computer Science', 90)
(9, 'Nurain', 'Biochemistry', 80)
(10, 'Mustapha', 'Industrial Chemistry', 78)
(11, 'Ibrahim', 'Computer Science', 80)
(12, 'Tolu', 'Computer Science', 67)
```

Au lieu d'utiliser le curseur pour récupérer les enregistrements, la méthode *pandas.read_sql()* prend en paramètre la requête Sql et la connexion à la base de donnée, et retourne un DataFrame (*Ligne 9*).

```
import mysql.connector
bdd = mysql.connector.connect(
  host="localhost",
  user="root",
  password="root",
  database="table1"
)
requete = "Select * from student_score;"
df= pd.read_sql(requete,bdd)
df.head()
```

	student_id	student_name	dep_name	score
0	1	Tomiwa	Microbiology	65
1	2	Yusuf	Biochemistry	70
2	3	Habeebah	Microbiology	80
3	4	Gbadebo	Computer Science	80
4	5	Muritadoh	Biochemistry	85

Pour écrire un DataFrame comme une table dans une base de donnée, le module *sqlalchemy*[7**] est utilisé. Le code suivant importe la méthode *create_engine* de *sqlalchemy* (Ligne 1). En suite, une connexion est créée avec la méthode *create_engine*(). Pour terminer, la méthode *DataFrame.to_sql()* est invoquée avec comme paramètre le nom de la table dans la base de données. Le paramètre *con* prend comme paramètre la connexion créée par *create_engine*() et il est aussi possible d'ajouter les lignes du DataFrame à la fin de la table si celle-ci existe déjà avec le paramètre optionnel *if_exists='append'*.

```
from sqlalchemy import create_engine
engine = create_engine("mysql+mysqlconnector://root:root@localhost/table1")
df.to_sql('nouvelle_table_etudiants', con=engine, if_exists='append', index=False)
```

Références

[03] Transfermarkt Football Data 2023, https://www.kaggle.com/datasets/davidcariboo/player-scores

[04] Pakistan Intellectual Capital 2023, https://www.kaggle.com/datasets/omarmagdy711/pakistan-intellectual-capital

[05] Kick Starters Projects 2023, https://www.kaggle.com/code/rtatman/data-cleaning-challenge-character-encodings/input?select=ks-projects-201612.csv

[06] SQL Connector 2024, https://dev.mysql.com/doc/connector-python/en/

[07] sqlalchemy 2024, https://www.sqlalchemy.org/

[07] SQLAlchemy 2024, https://www.sqlalchemy.org/