

Chapitre III Data Cleaning

MI - IA / GL

Ilyas Bambrik

Table des matières



Introduction	3
I - Entrées manquantes	4
II - Remplacement de valeurs manquantes	7
III - Traitement des données de type Date	9
IV - Accesseurs de chaîne de caractères	11
V - Uniformisation des entrées texte incohérentes	14
VI - Encodage de caractères	16
VII - Transformation de données catégoriques	18

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 Transfermarkt. L'instruction suivante donne le nombre d'entrées vides par colonne :

```
1 df=pd.read_csv('players.csv')
2 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

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

```
1 player_id                0
2 first_name              1965
3 last_name                0
4 name                    0
5 last_season              0
6 current_club_id         0
7 player_code             0
8 country_of_birth        2689
9 city_of_birth           2203
10 country_of_citizenship  543
11 date_of_birth           47
12 sub_position            172
13 position                0
14 foot                   2389
15 height_in_cm            2098
16 market_value_in_eur    10919
17 highest_market_value_in_eur 1321
18 contract_expiration_date 11467
19 agent_name              15361
20 image_url               0
21 url                     0
22 current_club_domestic_competition_id 0
23 current_club_name       0
24 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.

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 est nécessaire d'être sûr que les données sont dans le bon format et qu'elles 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 pour *player_id*).

```
1 df.height_in_cm.describe()

1 count      28204.000000
2 mean        182.234577
3 std         6.833916
4 min         18.000000
5 25%         178.000000
6 50%         182.000000
7 75%         187.000000
8 max         207.000000
9 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 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 qu'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.

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

```
1 df_1=df.dropna()
2 print(df_1.shape)
3 print(df.shape)
```

```
1 (8894, 23)
2 (30302, 23)
```

Après la suppression des lignes contenant des cellules nulles, environ de 20000 lignes ont été supprimées. 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 :

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

```
2 print(df_1.shape)
3 print(df.shape)
```

```
1 (30302, 11)
2 (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`.

```
1 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	Rodriguez	James Rodriguez	2022	683	james-rodriguez	NaN	Cúci
386	99227	Andrea	Bertolacci	Andrea Bertolacci	2022	3205	andrea-bertolacci	NaN	Roi
422	116900	Simon	Hedlund	Simon Hedlund	2022	206	simon-hedlund	NaN	Trollhätt
469	131694	Olarenwaju	Kayode	Olarenwaju Kayode	2022	24245	olarenwaju-kayode	NaN	Ibac
...
20004	427040	Alperen	Alperen	Alperen	2022	500	alperen-	NaN	Dalija

Il est évident qu'il y a une relation entre la ville et le pays natal. Sachant le nom de la ville, il est possible de récupérer le pays avec plusieurs méthodes. Le code suivant utilise Google pour récupérer les coordonnées de la ville. Ensuite, à 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)

```
1 from geopy.geocoders import Nominatim
2
3 def read_country(city):
4     geolocator = Nominatim(user_agent="google")
5     location = geolocator.geocode(city, language="en")
6     country = location.address.split(',')[ -1]
7     return country
```

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 :

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

```
1 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) :

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

Il existe aussi deux options de remplacement 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 être 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 `ffill` remplace une valeur nulle avec la valeur non nulle précédente.

```
1 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` :

```
1 df.foot.replace("right", "R")
```

```
1 0      NaN
2 1      NaN
3 2      NaN
4 3        R
5 4     left
6      ...
7 30297    R
8 30298    R
9 30299    R
10 30300   NaN
```

```
11 30301    left  
12 Name: foot, Length: 30302, 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.

```
1 df.date_of_birth.head()

1 0    1979-04-05
2 1    1979-01-15
3 2    1982-09-06
4 3    1983-01-15
5 4    1979-03-02
6 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* :

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

1 0    1979-04-05
2 1    1979-01-15
3 2    1982-09-06
4 3    1983-01-15
5 4    1979-03-02
6 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`):

```
1 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 :

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

```
1 0    2020-11-16
```

```

2 1    2022-10-06
3 2            NaT
4 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 ²²plus 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`.

```

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

```

1 0    False
2 1     True
3 2    False
4 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.

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

```

1 0    Monday
2 1    Thursday
3 2    Thursday
4 dtype: object
    
```

```
1 s.dt.days_in_month
```

```

1 0    30
2 1    30
3 2    31
4 dtype: int64
    
```

Accesseurs de chaîne de caractères

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.

```
1 df.name
```

```
1 0          Timo Hildebrand
2 1          Martin Petrov
3 2          Martin Amedick
4 3          Jermaine Pennant
5 4          Damien Duff
6          ...
7 30297       Jaka Bijol
8 30298       Semuel Pizzignacco
9 30299       Festy Ebosele
10 30300       Nicolò Cocetta
11 30301       Axel Guessand
12 Name: name, Length: 30302, dtype: object
```

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

```
1 0          timo hildebrand
2 1          martin petrov
3 2          martin amedick
4 3          jermaine pennant
5 4          damien duff
6          ...
7 30297       jaka bijol
8 30298       semuel pizzignacco
9 30299       festy ebosele
10 30300       nicolò cocetta
11 30301       axel guessand
12 Name: name, Length: 30302, dtype: object
```

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

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

```
1 0          TIMO HILDEBRAND
2 1          MARTIN PETROV
3 2          MARTIN AMEDICK
4 3          JERMAINE PENNANT
5 4          DAMIEN DUFF
6          ...
7 30297       JAKA BIJOL
```

```

8 30298 SEMUEL PIZZIGNACCO
9 30299 FESTY EBOSELE
10 30300 NICOLÒ COCETTA
11 30301 AXEL GUESSAND
12 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 espace 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 ;

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

```

1 0          [Timo, Hildebrand]
2 1          [Martin, Petrov]
3 2          [Martin, Amedick]
4 3          [Jermaine, Pennant]
5 4          [Damien, Duff]
6          ...
7 30297      [Jaka, Bijol]
8 30298      [Semuel, Pizzignacco]
9 30299      [Festy, Ebosele]
10 30300     [Nicolò, Cocetta]
11 30301     [Axel, Guessand]
12 Name: name, Length: 30302, dtype: object

```

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

```

1 0          Timo-Hildebrand
2 1          Martin-Petrov
3 2          Martin-Amedick
4 3          Jermaine-Pennant
5 4          Damien-Duff
6          ...
7 30297      Jaka-Bijol
8 30298      Semuel-Pizzignacco
9 30299      Festy-Ebosele
10 30300     Nicolò-Cocetta
11 30301     Axel-Guessand
12 Name: name, Length: 30302, dtype: object

```

Si l'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 :

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

```

1 0          [timo, hildebrand]
2 1          [martin, petrov]
3 2          [martin, amedick]
4 3          [jermaine, pennant]
5 4          [damien, duff]
6          ...

```

```
7 30297      [jaka, bijol]
8 30298    [semuel, pizzignacco]
9 30299      [festy, ebosele]
10 30300    [nicolò, cocetta]
11 30301    [axel, guessand]
12 Name: name, Length: 30302, dtype: object
```

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

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 :

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

Unnamed: 0	S#	Teacher Name	University Currently Teaching	Department	Province University Located	Designation	Terminal Degree	Graduated from	Country	Year	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 souhaitons nettoyer la colonne '*Country*' pour nous assurer qu'elle ne contient aucune incohérence dans la saisie des données.

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

```
1 [' 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 erreurs.

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 de fin sont très courantes dans les données texte et vous pouvez corriger 80 % de vos incohérences de saisie de données texte en procédant ainsi. Il est aussi complètement correcte de transformer le texte de la colonne en majuscule, ou en minuscule puis en titlecase.

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

```
1 ['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 :

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

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

```
1 [('south korea', 100),
  2 ('southkorea', 48),
  3 ('saudi arabia', 43),
  4 ('norway', 35),
  5 ('austria', 33),
  6 ('ireland', 33),
  7 ('pakistan', 32),
  8 ('portugal', 32),
  9 ('scotland', 32),
  10 ('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-ÿâ—â€-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 codés 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 :

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

```
1 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 % que ce soit exact, mais c'est généralement plus rapide que d'essayer de deviner.

Nous allons tester les dix mille 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.

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

```
1 {'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 pour lire le fichier.


```
1 df=pd.read_csv('ks-projects-201612.csv',encoding='Windows-1252')
2 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 qu'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*).

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

```
1 player_id           int64
2 first_name          object
3 last_name           object
4 name                object
5 last_season         int64
6 current_club_id     int64
7 player_code         object
8 country_of_birth    object
9 city_of_birth       object
10 country_of_citizenship object
11 date_of_birth      object
12 sub_position       object
13 position           object
14 foot              object
15 height_in_cm       float64
16 market_value_in_eur float64
17 highest_market_value_in_eur float64
18 contract_expiration_date object
19 agent_name         object
20 image_url          object
21 url                object
22 current_club_domestic_competition_id object
23 current_club_name  object
24 dtype: object
```

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

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

```
1 count    30302.000000
2 mean      4366.055574
3 std      10056.373140
4 min         3.000000
5 25%       403.000000
6 50%      1071.000000
```

```

7 75%      3008.000000
8 max      83678.000000
9 Name: current_club_id, dtype: float64

```

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

```

1 df.current_club_id=df.current_club_id.astype('category')
2 df.current_club_id.describe()

```

```

1 count      30302
2 unique      424
3 top        2553
4 freq        177
5 Name: current_club_id, dtype: int64

```

En outre, il est parfois nécessaire de diviser des valeurs numériques sur des bacs / intervalles pour la transformé 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 ;

```

1 df.height_in_cm.describe()

```

```

1 count      28204.000000
2 mean        182.234577
3 std         6.833916
4 min         18.000000
5 25%        178.000000
6 50%        182.000000
7 75%        187.000000
8 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 :

```

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

```

```

1 0          NaN
2 1          NaN
3 2          NaN
4 3      (17.999, 178.0]
5 4      (17.999, 178.0]
6      ...
7 30297     (187.0, 207.0]
8 30298     (187.0, 207.0]
9 30299     (178.0, 182.0]
10 30300          NaN
11 30301     (182.0, 187.0]
12 Name: height_in_cm, Length: 30302, dtype: category

```

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

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

```
1 (17.999, 178.0]      8528
2 (182.0, 187.0]      7642
3 (187.0, 207.0]      6423
4 (178.0, 182.0]      5611
5 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 :

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

```
1 (179.0, 185.0]      9583
2 (17.999, 179.0]     9516
3 (185.0, 207.0]     9105
4 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 :

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

```
1 (159.75, 207.0]     28200
2 (17.811, 65.25]      2
3 (112.5, 159.75]      2
4 (65.25, 112.5]       0
5 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.

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

```
1 0      NaN
2 1      NaN
3 2      NaN
4 3      0.0
5 4      0.0
6      ...
7 30297   3.0
8 30298   3.0
9 30299   1.0
10 30300  NaN
11 30301   2.0
12 Name: height_in_cm, Length: 30302, dtype: float64
```