Data engineering

Contents

Vrijwel ieder data science project start met het vergaren van data. Deze data kan op verschillende manieren aangeboden worden en in verschillende formaten. De eerste taak van de data scientist is dus het manipuleren van de data zodat alles in het gewenste formaat komt te staan. Men noemt dit de data engineeringdata engineering fase van het project. Pas daarna kan de data verwerkt worden. Deze taak mag niet onderschat worden: meestal besteedt men meer dan 80% van de tijd aan data engineering! In dit hoofdstuk bespreken we een aantal veel voorkomende technieken, maar er zijn er nog veel meer!

Voor dit deeltje gebruiken we volgende bibliotheken in Python:

```
import pandas as pd
import numpy as np
import datetime as dt
import pytz
import math
import os
```

Om de data-engineering taak te illustreren, geven we hier een voorbeeld.

Voorbeeld

*In een bedrijf wil men een overzicht maken van waar de belangrijke en minder belangrijke klanten zich bevinden. De output zou een **kaart** moeten zijn waarop de klanten worden aangegeven met bollen die de belangrijkheid weergeven.*

*De belangrijkheid van een klant wordt bepaald aan de hand van de **totale prijs** van alle bestellingen die door die klant werden geplaatst (exclusief btw) gedurende het laatste jaar. Het bedrijf is een fusie van 3 bedrijven. Hierdoor is de data verspreid over verschillende systemen: een reeks **Excel-bestanden**, een **Oracle-database**, **COBOL-databestanden** en een **SAP-systeem**. De data moet dus uit de verschillende bronnen worden gehaald en gecombineerd. Als je de data bij elkaar plaatst, blijkt dat er voor sommige bedrijven informatie **ontbreekt**.*

*De data is op verschillende manier **geëncodeerd**. De COBOL-databestanden gebruiken EBCDIC-encodering voor karakters. Andere data is met UTF-8 geëncodeerd en weer andere data is ASCII. Sommige bestanden bevatten ook binaire data (dus niet in tekstformaat). Daar kan gebruik gemaakt worden van little of big endian. Alle data moet dus geconverteerd worden naar eenzelfde formaat.*

*Sommige klanten blijken verschillende adressen te hebben. Dat moet dus uitgeklaard worden. Welk adres wordt er gebruikt? Als we naar de facturen gaan kijken, blijken er heel wat verschillen te zijn. Sommige facturen hebben enkel een factuurdatum, anderen ook een besteldatum. Aangezien we bestellingen willen bekijken, kan dit een probleem vormen. De datums staan niet altijd in hetzelfde **formaat**. Soms wordt het Amerikaanse formaat gebruikt (eerst maand en dan dag) en soms het Europese formaat (eerst dag, dan maand). Sommige datums zijn nog anders geformatteerd: jaar-maand-dag. Dit moet allemaal omgezet worden naar hetzelfde formaat.*

*De bedragen op de facturen zijn soms inclusief en soms exclusief btw. Dit moet worden aangepast. Er zijn **verschillende btw-tarieven**. Dit moet dus ook mee opgenomen worden. Sommige klanten komen verschillende keren voor in de databanken. Daarbij zijn de gegevens soms licht verschillend (spellingsfouten in de naam en adres, telefoonnummers die veranderd zijn, ...). Klanten die heel erg hard op elkaar gelijken, zullen dus misschien als één klant moeten worden gezien.*

We zullen in dit hoofdstuk geen antwoord kunnen geven op alle problemen die zich in het voorbeeld voordoen, maar het illustreert wel hoe **complex** data-engineering kan worden. Er is dikwijls veel creativiteit nodig om alles in het juiste formaat te krijgen. Daarbij mag de data in de databanken meestal niet veranderd worden. Je hebt dus een script nodig dat alle bewerkingen doet zodat je de analyse later nog een keer kan uitvoeren. Een programmeertaal als Python is daar zeer geschikt voor.
We gaan in deze cursus data uit een bron halen en deze in een data frame bewaren. Om te begrijpen hoe je data frames leest en wegschrijft, is het belangrijk te weten welke bronnen er gebruikt worden. Er zijn heel wat bronnen mogelijk waaruit je data frames kan halen:

- relationele databanken
- excel files
- libre office calc files
- html tables
- csv files
- tabellen uit boeken, artikels, ...
- SPSS bestanden
- google docs
- ...

In deze cursus zullen we ons in eerste instantie concentreren op CSV-bestanden. Dat formaat kan gelezen en geschreven worden door enorm veel softwarepakketten en vormt dus een ideale manier om gegevens uit te wisselen.

3.1 CSV bestanden

CSV staat voor Comma Separated Values. Het zijn tekstbestanden (meestal in ASCII, ISO 8859 of UTF-8 gecodeerd) waarin iedere lijn een record uit een tabel voorstelt. Een CSV-bestand kan maar 1 tabel bevatten. Alle waarden van een record worden gescheiden door komma's of door

een ander karakter. Dat karakter noemt men ook het **scheidingsteken**, de **delimiter** of de **separator**. Strings worden meestal tussen aanhalingstekens geplaatst, zeker wanneer er spaties in voorkomen (of de separator bijvoorbeeld).

De eerste regel van een CSV-bestand bevat meestal alle kolomnamen, maar dit kan ontbreken. Men noemt dit de **header** van het CSV-bestand.

Kommagetallen kunnen op twee manieren worden opgeslagen: met een `.` of met een `,`. Als je in een Nederlandstalige excel een bestand naar CSV exporteert, dan zul je zien dat er automatisch een `,` wordt gebruikt. In een Engelstalige excel zal dat een `.` zijn.

CSV-bestanden bevatten enkel data. Er is geen mogelijkheid om lay-out of kleuren of andere meta-informatie op te slaan. Maar dat hebben we ook niet nodig om berekeningen te doen.

Hier zie je een voorbeeld van een CSV-bestand (data_en.csv):

```
"id","gewicht","lengte"
1,43.2,154
2,55.1,160
3,45.7,148
4,61.4,161
5,51.1,165
```

In dit voorbeeld kan je zien dat de velden worden gescheiden door een `,`. Er is een header (de eerste regel bevat de namen van de kolommen) en er wordt een `.` gebruikt voor kommagetallen.

Dezelfde data zou ook zo opgeslagen kunnen worden (data_nl.csv):

```
"id";"gewicht";"lengte"
1;43,2;154
2;55,1;160
3;45,7;148
4;61,4;161
5;51,1;165
```

Hier werd een `;` als delimiter gebruikt en een `,` voor kommagetallen.

Alvorens een CSV bestand in te lezen, moet je dus ALTIJD eerst gaan kijken hoe dit is opgeslagen. Je kan dit doen met een tekst editor zoals notepad, notepad+, wordpad, gedit, geany, ...

3.2 CSV schrijven

Als je een data frame vanuit Python wil exporteren, dan kan je gemakkelijk via een CSV bestand doen. Eerst en vooral is het soms handig om de huidige directory te wijzigen. Dit kan met:

```
os.chdir("datasets")
```

We creëren eerst een data frame:

```
f = pd.DataFrame({"a": ["bla", "boe", "bla"], "b": [1.1, 2.2, 3.3]})
```

Het wegschrijven naar een CSV bestand gaat dan als volgt:

```
f.to_csv("testje_nl.csv", sep=";", decimal=",", index=False)
```

Het gegenereerde bestand kan je openen met excel of een ander spreadsheet programma. Pas desnoods de waarden van sep en decimal aan zodat dit overeenkomt met de taalinstellingen van je spreadsheet.

3.3 CSV inlezen

Om een CSV bestand in te lezen, kan je volgend commando gebruiken. We demonstreren de Engelse en de Nederlandse versie, maar sommige bestanden wijken ook hiervan af. Als je de bestanden wil inlezen, kan dit met:

```
data_en = pd.read_csv("data_en.csv", sep=",", decimal=".", header=0)
data_nl = pd.read_csv("data_nl.csv", sep=";", decimal=",", header=0)
```

Als er geen header is, kan je deze optie gebruiken: header=None. Als je CSV bestanden inleest, moet je zeker eerst nakijken welk formaat er gebruikt werd:

- is er een header? Op welke regel staat deze?
- wat is de separator (spatie, komma, puntkomma, tab, ...)?
- hoe werden kommagetallen bewaard (met een punt of komma)?
- begint het bestand met regels commentaar?
- met welke encodering zijn strings bewaard (utf_8, latin_1, iso8859_2, ...)? (meer info vind je op https://docs.python.org/3/library/codecs.html#standard-encodings)

Je kan het bestand dan inlezen met:

```
data = pd.read_csv('bestandsnaam.csv', sep='separator', decimal='komma',
skip=aantalCommentaarRegels, header=regelnummer, encoding='encodering')
```

Het kan gebeuren dat het inlezen van een CSV bestand fouten oplevert. Dat komt dan meestal omdat het bestand lijnen bevat die niet correct geformatteerd zijn. Een veel voorkomende fout is dat de separator ook voorkomt in de data. Er zijn dan op die lijn meer kolommen aanwezig dan in de andere lijnen. Hier zie je een klein voorbeeld (bestand1.csv):

```
id, slogan
1,Een voor allen
2,Allen Voor een
3,Aan allen die aanwezig zijn, Proficiat
4,Wait for it
```

Er zijn hier 2 kolommen, gescheiden door komma's. Maar in lijn 3 zitten er ook komma's in de waarde van de string. Normaal gezien zouden de strings tussen aanhalingstekens gezet moeten worden. Maar om een of andere reden is dat hier niet gebeurd. Hoewel dit niet zou mogen gebeuren, zul je het in de praktijk zeker tegenkomen.

Als we dit bestand proberen in te lezen, krijgen we dit:

```
tabel = pd.read_csv('bestand1.csv', sep=',')
```

Je krijgt volgende foutmelding:

> **Error**
>
> ParserError: Error tokenizing data. C error: Expected 2 fields in line 4, saw 3

Het is mogelijk om deze fouten te negeren. Dat kan als volgt:

```
tabel = pd.read_csv('bestand1.csv', sep=',', on_bad_lines='skip')
```

Maar nu werd de foute lijn gewoon overgeslagen. Hoewel dit heel gemakkelijk is, kan het zijn dat er op die manier te veel data verloren gaat. Het is ook niet altijd mogelijk om het bestand handmatig te corrigeren. Daarom zal men in de praktijk een (Python) script schrijven dat de

fouten eerst verbetert. Hierbij moet je soms heel wat creativiteit aan de dag leggen en moet je heel goed zijn in programmeren. Aangezien dit een basiscursus is, zullen deze meer geavanceerde technieken hier nog niet besproken worden.
Als je een bestand hebt ingelezen, dan wil je in veel gevallen de ingelezen data bestuderen. Je zou dit kunnen doen met:

```
data_nl
```

	id	gewicht	lengte
0	1	43.2	154
1	2	55.1	160
2	3	45.7	148
3	4	61.4	161
4	5	51.1	165

Maar als het data frame heel veel data bevat, is dit vrij nutteloos. Gelukkig kan je ook vragen om een deel te laten zien. Dit doe je met:

```
data_nl.head(n=10)
```

	id	gewicht	lengte
0	1	43.2	154
1	2	55.1	160
2	3	45.7	148
3	4	61.4	161
4	5	51.1	165

Dit werkt net zoals het `head` commando in Linux en toont dus de eerste 10 lijnen van de tabel.

In de meeste gevallen is het belangrijk te weten welke kolommen er zijn en welk datatype deze hebben. Je kan dit met het volgende commando zichtbaar maken:

```
data_nl.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   id       5 non-null      int64
 1   gewicht  5 non-null      float64
 2   lengte   5 non-null      int64
dtypes: float64(1), int64(2)
memory usage: 248.0 bytes
```

Dit print voor iedere kolom: de naam, het aantal geldige waarden en het type (dtype).

Je kan ook al een aantal afgeleide waarden zien per kolom met het commando:

```
data_nl.gewicht.describe()
```

```
count     5.000000
mean     51.300000
std       7.305135
min      43.200000
25%      45.700000
50%      51.100000
75%      55.100000
max      61.400000
Name: gewicht, dtype: float64
```

Als de kolom getallen bevat, dan zal dit commando het aantal waarden laten zien, samen met het gemiddelde, de standaardafwijking, het minimum, de kwartielen en het maximum.

Als de kolom strings bevat of een Categorical is, dan toont dit commando het aantal waarden, het aantal unieke waarden, de meest voorkomende waarde en het aantal keer dat die voorkomt. In veel gevallen zal je merken dat niet alle waarden van een tabel gekend zijn. In een databank worden deze waarden dikwijls als NULL genoteerd. In Python kan een ontbrekende waarde op verschillende manieren worden voorgesteld:

- np.nan
- None (dit is hetzelfde als een null pointer)
- pd.NA
- math.nan
- ...

Dat kan soms verwarrend zijn. De pandas bibliotheek biedt een aantal methoden aan om snel te kunnen zien welke waarden in een Series ontbreken. We maken daarvoor eerst een Series en steken er een aantal ontbrekende waarden in:

```
data = pd.Series([1, np.nan, 2, None, pd.NA])
print(data)
```

```
0       1
1     NaN
2       2
3    None
4    <NA>
dtype: object
```

Zoals je ziet worden de verschillende mogelijkheden voor ontbrekende waarden hier gebruikt.

Als je wil weten welke waarden ontbreken, dan kan je dat als volgt doen:

```
data.isna()
```

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

Als we willen weten hoeveel ontbrekende waarden er zijn, dan kan dat als volgt:

```
data.isna().sum()
```

```
3
```

Tenslotte kunnen we alle ontbrekende waarden schrappen met:

```
data.dropna()
```

```
0    1
2    2
dtype: object
```

Deze methode geeft een nieuwe Series terug zonder de ontbrekende waarden. De oorspronkelijke Series blijft onveranderd. Je kan de oorspronkelijke data wel veranderen met volgende optie:

```
data.dropna(inplace=True)
```

Deze methoden kan je ook uitvoeren op een data frame:

```
df.isna().sum().sum() # totaal aantal ontbrekende waarden
df.dropna() # wis alle rijen met ontbrekende waarden
```

5.1 Ontstaan van ontbrekende waarden

Ontbrekende waarden kunnen op verschillende manieren ontstaan. We bespreken hier een paar mogelijkheden.

5.1.1 Bij het inlezen

Bij het inlezen van data kan het zijn dat ontbrekende waarden met een bepaalde string werden voorgesteld. Python herkent deze string niet altijd en zal de hele kolom als strings inlezen. Hier zie je een voorbeeld databestand (`bestand2.csv`):

```
id;lengte;gewicht
1;173;70.1
2;174;60.3
3;175,3;NA
4;Missing;60
```

Zoals je ziet, gebruikt de kolom `lengte` een ',' en de kolom `gewicht` een '.' voor decimale getallen. Waarschijnlijk komt de data van 2 plaatsen en werd deze bij elkaar gezet. We kiezen in eerste instantie voor ',':

```
tabel = pd.read_csv('bestand2.csv', sep=';', decimal=',')
```

Met het info() commando kan je nagaan hoe het bestand ingelezen werd:

```
tabel.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   id       4 non-null      int64
 1   lengte   4 non-null      object
 2   gewicht  3 non-null      object
dtypes: int64(1), object(2)
memory usage: 224.0+ bytes
```

Je kan zien dat de kolommen `lengte` en `gewicht` als strings ingelezen werden (het type is `object`). Je kan er dus ook niet mee rekenen. Bij de kolom `lengte` komt dat omdat de string `Missing` niet herkend wordt als ontbrekende waarde. Daardoor werden alle waarden als strings ingelezen en niet als getallen. De string `NA` in de kolom `gewicht` werd daarentegen wel herkend (kijk maar na!), maar aangezien de kommagetallen een '.' gebruiken in die kolom, werden de waarden ook als strings ingelezen. Je kan dit ook in de tabel zelf zien:

```
print(tabel)
```

```
   id   lengte gewicht
0   1      173    70.1
1   2      174    60.3
2   3    175,3     NaN
3   4  Missing      60
```

Je kan de kolom `lengte` omzetten naar getallen door de komma te vervangen door een '.' en dan te parsen naar getallen. Dit kan met het volgende commando's:

```
tabel.lengte = tabel.lengte.str.replace(',', '.', regex=False)
tabel.lengte = pd.to_numeric(tabel.lengte, errors='coerce')
print(tabel.lengte)
```

```
0    173.0
1    174.0
2    175.3
3      NaN
Name: lengte, dtype: float64
```

Alle strings die getallen bevatten worden hierdoor omgezet naar getallen en alle andere strings worden vervangen door NaN.

Bemerk dat de laatste kolom ook als strings werd ingelezen (ook al werd NA wel herkend). Dat is omdat daar een punt werd gebruikt voor decimale getallen (bij het inlezen gebruikten we ',' voor kommagetallen). Die kolom moeten we dus ook nog omzetten naar getallen:

```
tabel.gewicht = pd.to_numeric(tabel.gewicht, errors='coerce')
tabel.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   id       4 non-null      int64
 1   lengte   3 non-null      float64
 2   gewicht  3 non-null      float64
dtypes: float64(2), int64(1)
memory usage: 224.0 bytes
```

De types van alle kolommen staan nu juist en de ontbrekende waarden staan mooi op NaN.

Er bestaat echter ook een veel kortere manier om de data correct in te lezen. Als je op voorhand weet welke string(s) er werd(en) gebruikt voor ontbrekende waarden, kan je deze ook opgeven bij het inlezen. Het vorige bestand konden we bijvoorbeeld veel beter inlezen met:

```
tabel = pd.read_csv('bestand2.csv', sep=';', decimal='.', header=0, na_values=
['Missing', 'NA'])
```

Nu staat de waarde `Missing` al onmiddellijk op NaN. Door decimal='.' te gebruiken is de laatste kolom ook direct in orde en hoeven we dus enkel de tweede kolom nog naar getallen om te zetten. Het loont dus zeker om eerst naar de data te kijken vooraleer die in te lezen!

5.1.2 Bij het omzetten

Na het inlezen van data, zullen we steeds alle nominale en ordinale variabelen omzetten naar een **Categorical**. Het voordeel is dan dat de opslagruimte beperkt wordt en dat ordinale variabelen hierdoor een nummer krijgen dat we kunnen gebruiken in bepaalde algoritmes.

Bij het omzetten van een variabele naar een **Categorical** kunnen er echter ook ontbrekende waarden ontstaan. Hier zie je een voorbeeld:

```
kolom = pd.Series(['a', 'b', 'a', 'c', 'a', 'b'])
cats = pd.Categorical(kolom, categories=['a', 'b'])
print(cats)
```

```
['a', 'b', 'a', NaN, 'a', 'b']
Categories (2, object): ['a', 'b']
```

Doordat we niet alle mogelijke strings hebben opgesomd, zijn de waarden die niet voorkomen in de categorieën, vervangen door NaN. Het is dus belangrijk om steeds te weten welke unieke waarden voorkomen in een variabele. Je kan dit doen met de functie `unique()`:

```
kolom.unique()
```

```
array(['a', 'b', 'c'], dtype=object)
```

Je kan de ontbrekende waarden echter ook opzettelijk doen ontstaan. De waarde 'c' is misschien niet toegelaten. Op die manier worden alle illegale waarden automatisch naar NaN omgezet.

5.2 Omgaan met ontbrekende waarden

Je kan op verschillende manieren omgaan met ontbrekende waarden, afhankelijk van de context.

5.2.1 Niets doen

In eerste instantie is het misschien niet eens nodig om iets met ontbrekende waarden te doen. Heel wat algoritmes kunnen er perfect mee omgaan. Kijk dus altijd eerst of die waarden een probleem vormen of niet.

5.2.2 Rijen of kolommen verwijderen

Een tweede mogelijkheid is het verwijderen van alle rijen in een tabel waar een ontbrekende waarde in voorkomt. Dat kan met volgend commando:

```
tabel = tabel.dropna(axis='rows')
```

Dit verwijdert volledige rijen. Er gaat dus informatie verloren en is daarom niet altijd de beste keuze. Kijk goed na of dit een probleem kan zijn. Je kan de rijen verwijderen als er niet te veel zijn met NaN waarden. Als de tabel heel veel ontbrekende waarden heeft, dan is het geen goed idee om de rijen te verwijderen omdat er dan teveel informatie verloren gaat.

Je kan natuurlijk ook kolommen met ontbrekende waarden verwijderen, maar dat is zelden gewenst. Het gaat als volgt:

```
tabel = tabel.dropna(axis='columns')
```

5.2.3 Vervangen

Een derde mogelijkheid is de ontbrekende waarden vervangen door een geldige waarde. Een voorbeeldje zie je hier:

```
waarden = ['geslaagd', 'geslaagd', np.nan, 'niet geslaagd']
scores = pd.Series(waarden)
```

De ontbrekende waarde betekent hier eigenlijk dat de student niet heeft deelgenomen. We kunnen de waarde dus even goed vervangen door de string `niet deelgenomen`. Dat kan als volgt:

```
scores.fillna("niet deelgenomen", inplace=True)
```

Alle ontbrekende waarden worden hierdoor vervangen door de gegeven string. De optie `inplace=True` zorgt ervoor dat de data ook direct bewaard wordt in de series. Als je die weglaat, dan zal de methode `fillna()` een nieuwe series teruggeven met de nieuwe waarden. Daarbij wordt de oorspronkelijke data niet aangepast.

Let goed op bij het vervangen van waarden! Dit is niet altijd zinvol. Bij getallen heeft men soms de neiging om ontbrekende waarden te vervangen door 0. Maar dat kan grote verschillen geven bij berekeningen. Soms is het beter om de ontbrekende waarden dan te vervangen door het gemiddelde, de mediaan of de modus van de kolom, maar ook dat moet in de context bekeken worden.

5.2.4 Interpoleren

Stel dat de waarden in een lijst metingen zijn die op regelmatige basis werden uitgevoerd. In die lijst ontbreken er echter waarden omdat de meting niet werd uitgevoerd. In dat geval kan het soms goed zijn om de ontbrekende waarden te schatten door naar de omliggende waarden te kijken. We noemen dit **interpoleren**.

Een voorbeeldje kan dit duidelijker maken: we noteerden ieder uur de temperatuur in een kamer, maar af en toe vergaten we het. Het resultaat zetten we in een Pandas Series:

```
waarden = [18.4, math.nan, 19.5, 20.5, 20.5, 19.5, math.nan, 17.8]
temps = pd.Series(waarden)
print(temps)
```

```
0    18.4
1     NaN
2    19.5
3    20.5
4    20.5
5    19.5
6     NaN
7    17.8
dtype: float64
```

We kunnen de ontbrekende temperaturen nu automatisch laten schatten door te interpoleren:

```
new_temps = temps.interpolate()
print(new_temps)
```

```
0    18.40
1    18.95
2    19.50
3    20.50
4    20.50
5    19.50
6    18.65
7    17.80
dtype: float64
```

Je ziet dat de ontbrekende waarden vervangen zijn door getallen. Deze werden bepaald door rechte lijnen te trekken tussen de gekende waarden en te kijken welke waarden ertussen liggen. Dit is niet altijd een hele goede benadering. Een populaire manier om een betere waarden te bekomen, bestaat erin om een kromme lijn tussen de gekende waarden te tekenen. Men gebruikt hier dikwijls een **spline**. Als je dit wil doen, dan kan dat op de volgende manier (we gebruiken hier een **cubic spline**):

```
new_temps = temps.interpolate(method='spline', order=3)
print(new_temps)
```

```
0    18.400000
1    19.214414
2    19.500000
3    20.500000
4    20.500000
5    19.500000
6    19.037199
7    17.800000
dtype: float64
```

Soms heb je geen ontbrekende waarden, maar verkeerde waarden in een kolom staan. Als kolommen illegale waarden bevatten, kan het nodig zijn om deze te vervangen door geldige waarden of door NaN. Verkeerde waarden kunnen ontstaan omdat er geen validatie was bij het ingeven ervan of bij typefouten.

6.1 Vervangen

Stel dat je het geslacht van een aantal mensen opgeslagen hebt in een kolom van een tabel:

```
geslacht = pd.Series(['v', 'm', 'x', 'f', 'v', 'm', 'v', 'v'])
tabel = pd.DataFrame({'geslacht': geslacht})
print(tabel)
```

```
  geslacht
0        v
1        m
2        x
3        f
4        v
5        m
6        v
7        v
```

De bedoeling was om enkel ‘m’, ‘v’ of ‘x’ toe te laten. Maar iemand heeft ‘f’ ingevuld en dat moet dus waarschijnlijk ‘v’ zijn. Je kan dit veranderen met:

```
tabel.loc[tabel.geslacht == 'f', 'geslacht'] = 'v'
print(tabel)
```

```
  geslacht
0        v
1        m
2        x
3        v
4        v
5        m
6        v
7        v
```

6.2 Delen vervangen

Soms wil je in een hele reeks strings een deel vervangen door iets anders. Dit werd hierboven al kort gedemonstreerd. Stel dat je een kolom hebt in een tabel waarin alle spaties vervangen moeten worden door underscores. Dan kan dat met:

```
kolom = pd.Series(['dit ', 'en dat', 'is een testje '])
kolom = kolom.str.replace(' ', '_', regex=False)
print(kolom)
```

```
0             dit_
1           en_dat
2    is_een_testje_
dtype: object
```

Je kan ook reguliere expressies gebruiken. Dit doe je zo:

```
kolom = kolom.str.replace('d.t', 'dot', regex=True)
print(kolom)
```

```
0              dot_
1           en_dot
2    is_een_testje_
dtype: object
```

Dit vervangt alle patronen 'd.t' door 'dot'.

Bemerk dat je steeds `.str` moet schrijven om de string functies te kunnen gebruiken. Data-engineering is niet eenvoudig in een stappenplan te gieten. Je moet steeds kijken welke data je voorhanden hebt en welke data je nodig hebt. Python biedt enorm veel functionaliteit om data te transformeren en te manipuleren. Deze cursus laat slechts het tipje van de ijsberg zien.

Volgend stappenplan kan een leidraad zijn om eenvoudige CSV-bestanden in te lezen en klaar te zetten om te verwerken. Soms zul je stappen moeten herhalen omdat je onderweg fouten tegen komt die je in een vorige stap kon verhelpen. Het is dus geen stappenplan dat je zonder denken mag uitvoeren!

7.1 Bepaal het bestandsformaat

Als je een CSV-bestand moet inlezen, start je best met het bestand te openen met een text editor of een hex editor. Hierdoor kan je volgende zaken bepalen:

- welke encodering werd er gebruikt? Dit is niet altijd zo eenvoudig te weten. Maar als het bestand met `ï»¿` (EF BB BF) begint, dan is dit een UTF-8 bestand. Je kan onder Linux het commando `file -i filename` gebruiken om de encodering te weten te komen. Als er veel 0-bytes in een bestand voorkomen, is het meestal met UTF-16 geëncodeerd. In andere gevallen zul je de encodering moeten raden en eventueel aanpassen als je ziet dat er foute data ingelezen werd.
- wat is de separator? Dit is meestal een komma of punt-komma, maar tabs en spaties komen ook voor.
- hoe werden kommagetallen genoteerd? Dit is meestal met een punt of een komma
- is er een header? Op welke regel staat deze?
- zijn er regels commentaar in het begin? Dit gebeurt soms en deze lijnen moeten overgeslagen worden bij het inlezen
- zijn er woorden die gebruikt worden om ontbrekende waarden voor te stellen?

7.2 Lees het bestand in

Probeer nu het bestand in te lezen met read_csv(). Zet hierbij alle parameters op de waarde die je in de vorige stap bepaalde:

```
data = pd.read_csv('bestandsnaam.csv', sep='...', decimal='...',
                   skip=aantalCommentaarRegels, header=regelnummer, encoding='...',
                   na_values=[...])
```

Als je fouten krijgt bij het inlezen, los deze dan op of negeer lijnen met fouten door de optie `on_bad_lines` te gebruiken.

7.3 Check de data

Gebruik volgende commando's om na te gaan of de data goed ingelezen werd:

```
data.info()
data.head()
data.describe()
```

Bepaal het datatype van iedere kolom en vergelijk dit met het gewenste datatype.

7.4 Datatypes juist zetten

Zet de data om zodat alle kolommen het juiste datatype hebben.

Je kan hierbij volgende leidraad gebruiken:

- als de kolom gehele getallen moet bevatten en de data bestaat uit strings, gebruik dan `to_numeric()` om de data om te zetten. Kijk waar er NaN waarden ontstaan en welke string daar stond. Misschien heb je een ontbrekende waarden verkeerd opgegeven bij het inlezen of komen er illegale waarden voor.
- als de kolom kommagetallen moet bevatten en de data bestaat uit strings, gebruik dan `to_numeric()` om de data om te zetten. Kijk waar er NaN waarden ontstaan en welke string daar stond. Misschien heb je de verkeerde waarde voor `decimal` gebruikt of heb je ontbrekende waarden verkeerde opgegeven bij het inlezen. Het kan ook zijn dat er illegale waarden voorkomen.
- als de kolom strings moet bevatten, bepaal dan het aantal unieke waarden met `unique()`. Als dit er heel veel zijn, kan je de kolom zo laten. Maar als er niet te veel verschillende waarden zijn, zet je de kolom best om naar een Categorical. Als de kolom ordinale gegevens bevat, gebruik je de juiste volgorde van de categorieën om de Categorical te maken
- als je ziet dat er verkeerde waarden in een kolom voorkomen, dan kan je ze vervangen door juiste waarden. Dit kan bijvoorbeeld met de `replace()` functie.
- je kan in numerieke kolommen kijken naar het minimum en maximum (met `describe()`). Dit kan aangeven of er illegale waarden aanwezig zijn.

7.5 Ontbrekende waarden

Met het commando:

```
data.isna().sum().sum()
```

kun je nagaan hoeveel ontbrekende waarden er voorkomen in het data frame.

Als je wil weten op welke rijen die ontbrekende waarden voorkomen, dan kan je dat met het volgende commando te weten komen:

```
rowsWithNaN = data[data.isna(). any(axis=1)]
print(rowsWithNan)
```

Je kan ook te weten komen welke kolommen ontbrekende waarden bevatten. Dit kan met het volgende commando:

```
columnsWithNaN = data[data.columns[data.isna().any(axis=0)]]
```

Als er nog ontbrekende waarden zijn, kan je bepalen wat je ermee wil doen. Je kan dit ook laten afhangen van het algoritme dat je wil gebruiken (zie later).

By Wouter Deketelaere, Kris Demuynck, Wim Dekeyser, Jan Van Overveldt
© Copyright 2023.