Frequenties

Contents

Samenvatting

In dit deeltje spreken we over frequenties. Het berekenen van frequenties is dikwijls een van de eerste stappen die men onderneemt na het verzamelen en opkuisen van ruwe gegevens.

De ruwe gegevens

We zullen in dit deel vertrekken van een voorbeeld. Stel dat we inzicht willen krijgen in de laptops die gebruikt worden in een bedrijf. Er zijn 857 werknemers in dit bedrijf met een laptop. Van iedere laptop noteren we volgende gegevens: CPU-generatie en type, RAM-geheugen (in GB), geformatteerde harde schijfruimte (in GB) en het merk.

We lezen de tabel in als volgt. Dat leidt tot volgende tabel (met `head()` tonen we enkel de eerste 5 lijnen):

```
import pandas as pd

laptops = pd.read_csv('datasets/laptops.csv', sep=';', decimal=',')
laptops.head()
```

	cpuGeneration	cpuType	RAM	diskspace	brand
0	Kabylake	i7	4.0	232.5	Toshiba
1	Kabylake	i5	2.0	992.5	Acer
2	Haswell	i7	16.0	495.6	Dell
3	Skylake	i7	4.0	217.2	Toshiba
4	Broadwell	i5	4.0	245.8	Acer

In deze ruwe data zijn er 5 variabelen. Als eerste stap, bepalen we de meetniveau's. Deze zijn als volgt:

- cpuGeneration: ordinaal
- cpuType: ordinaal
- RAM: ratio
- diskspace: ratio, continu
- brand: nominaal

Voor de ordinale gegevens, moeten we de categorieën in de juiste volgorde zetten. We doen dit als volgt:

De tabel met ruwe data is heel erg lang en het is dus ook heel moeilijk om uit deze tabel direct conclusies te trekken. Daarom dat we de data verder gaan verwerken. We controleren even of de datatypes in orde zijn:

```
# ordinaal niveau maken
cpuGenerationLevels = ['SandyBridge', 'IvyBridge', 'Haswell', 'Broadwell', 'Skylake',
'Kabylake']
laptops.cpuGeneration = pd.Categorical(laptops.cpuGeneration, ordered=True,
categories=cpuGenerationLevels)

# ordinaal niveau maken
cpuTypeLevels = ['i3', 'i5', 'i7']
laptops.cpuType = pd.Categorical(laptops.cpuType, ordered=True,
categories=cpuTypeLevels)

# optioneel: nominaal niveau maken (bespaart geheugen)
laptops.brand = pd.Categorical(laptops.brand)
```

De tabel met ruwe data is heel erg lang en het is dus ook heel moeilijk om uit deze tabel direct conclusies te trekken. Daarom dat we de data verder gaan verwerken. We controleren even of de datatypes in orde zijn:

```
pd.DataFrame(laptops.dtypes, columns=['dtype'])
```

Absolute frequenties

Een heel eenvoudige techniek kan al iets meer informatie opleveren: voor iedere variabele tel je hoeveel maal elke waarde voorkomt. Dit noemt men de (absolute) frequentie. Je kan dit in Python heel gemakkelijk berekenen met de functie `value_counts()`. Hier zie je hoe dat werkt (voor de kolom cpuType):

```
laptops.cpuType.value_counts()
```

```
i5    556
i3    213
i7     84
Name: cpuType, dtype: int64
```

We berekenden hier de absolute frequenties van de variabele “cpuType”. Het type

- `i5` komt 556 keer voor,
- `i3` komt 213 keer voor en
- `i7` komt 84 keer voor.

Je kan op dezelfde manier de frequenties berekenen van de andere variabelen. Het verkregen resultaat is een pandas Series. Als je bijvoorbeeld een lijst wil met alle mogelijke waarden, zonder frequenties, dan kan je dit als volgt doen:

```
laptops.cpuType.unique()
```

```
['i7', 'i5', 'i3', NaN]
Categories (3, object): ['i3' < 'i5' < 'i7']
```

Als je nu de som berekent van alle frequenties, dan vind je 853. Dat komt niet overeen met het aantal lijnen in de tabel (857)! Dit komt omdat `value_counts()` de ontbrekende waarden (NaN) niet meetelt. Dat is jammer, want soms zijn er ontbrekende waarden die je niet snel ziet (omdat er bijvoorbeeld heel veel waarden zijn). Daarom dat het dikwijls beter is om die wel mee te tellen. Hier zie je hoe dat gaat:

```
laptops.cpuType.value_counts(dropna=False)
```

```
i5     556
i3     213
i7      84
NaN      4
Name: cpuType, dtype: int64
```

In dit geval zijn er dus vier ontbrekende-waarden.

Standaard zal Python de waarden sorteren volgens hun frequentie. Als je dat niet wil, dan kan je dit oplossen met `sort_index()`:

```
laptops.cpuType.value_counts().sort_index()
```

```
i3    213
i5    556
i7     84
Name: cpuType, dtype: int64
```

In dit geval worden de waarden gesorteerd volgens de volgorde die we zelf opgaven. Dit is vooral handig voor variabelen die minstens ordinaal meetniveau hebben.

Klassen

Merk op dat je absolute frequenties kan berekenen vanaf nominaal meetniveau. Je kan ze met andere woorden altijd berekenen. Er is echter een probleem bij continue variabelen of variabelen met heel veel mogelijke waarden. Om dat duidelijk te maken, bekijken we de kolom diskspace. Deze laat zien hoeveel vrije diskruimte er op de schijf was nadat deze geformatteerd werd. Als we nu gaan tellen hoeveel iedere waarde voorkomt, krijgen we het volgende:

```
laptops.diskspace.value_counts()
```

```
510.6     5
488.9     4
482.9     4
494.1     4
231.3     4
         ..
493.6     1
996.7     1
482.6     1
1003.6    1
234.0     1
Name: diskspace, Length: 592, dtype: int64
```

Op het eerste zicht is er niet veel aan de hand, maar de lijst is enorm lang (592 lijnen!) en de meeste frequenties zijn 1 of 2. Het resultaat is dus helemaal niet interessant en geeft zeker niet veel informatie over de data.

Een oplossing bestaat erin om waarden bij elkaar te nemen in zogenaamde “klassen” en dan de frequentie per klasse te bepalen. In Python kan je klassen maken met het `cut()`-commando:

```
cutpoints = range(0, 1200, 100)
klassen = pd.cut(laptops.diskspace, bins=cutpoints)
klassen
```

```
0       (200.0, 300.0]
1      (900.0, 1000.0]
2       (400.0, 500.0]
3       (200.0, 300.0]
4       (200.0, 300.0]
            ...
852                NaN
853     (200.0, 300.0]
854     (200.0, 300.0]
855     (200.0, 300.0]
856     (200.0, 300.0]
Name: diskspace, Length: 857, dtype: category
Categories (11, interval[int64, right]): [(0, 100] < (100, 200] < (200, 300] < (300,
400] ... (700, 800] < (800, 900] < (900, 1000] < (1000, 1100]]
```

We kunnen nu de absolute frequenties van de **klassen** berekenen:

```
klassen.value_counts().sort_index()
```

```
(0, 100]         29
(100, 200]       52
(200, 300]      407
(300, 400]        0
(400, 500]      227
(500, 600]       96
(600, 700]        0
(700, 800]        0
(800, 900]        0
(900, 1000]      19
(1000, 1100]     21
Name: diskspace, dtype: int64
```

Eerst bepaal je de grenzen van de klassen (in dit voorbeeld in de variabele "cutpoints"). Het commando `pd.cut()` zal dan iedere waarde vervangen door een string die het interval voorstelt waarbinnen de waarde valt. De variabele wordt dus omgezet van interval meetniveau naar ordinaal meetniveau! De intervallen beginnen in dit geval altijd met een rond haakje en ze eindigen met een recht haakje. Dit geeft aan dat de linkergrens niet inbegrepen is. De schijfruimte 200 zal dus niet in $(200, 300]$ zitten, maar wel in $(100, 200]$. Als je dit andersom wil doen, dan kan dat als volgt:

```
klassen = pd.cut(laptops.diskspace, bins=cutpoints, right=False)
klassen.value_counts().sort_index()
```

```
[0, 100)         29
[100, 200)       52
[200, 300)      407
[300, 400)        0
[400, 500)      227
[500, 600)       96
[600, 700)        0
[700, 800)        0
[800, 900)        0
[900, 1000)      19
[1000, 1100)     21
Name: diskspace, dtype: int64
```

In dit geval geeft dit dezelfde frequenties omdat de randgevallen nooit voorkomen. Als dat wel zo is, dan kunnen beide commando's verschillende resultaten geven.

Opdelen in klassen lijkt het probleem op te lossen, maar er blijft nog 1 vraag over: hoeveel klassen moet je maken? Als je te veel klassen maakt, dan krijg je het oorspronkelijke probleem weer. Maar als je te weinig klassen maakt, gaat er veel informatie verloren. Er zijn een aantal manieren om het aantal klassen (m) te bepalen:

- Sturges: $m = \lceil 1 + log_2(n) \rceil$
- Scott: $b = 3.5 \cdot \frac{s_X}{\sqrt[3]{n}}$, $m = \lceil \frac{(\max(X) - \min(X))}{b} \rceil$
- MS Excel: $m = \lceil \sqrt{n} \rceil$

Hierbij is $\lceil x \rceil$ het kleinste geheel getal dat groter is dan x (je rondt x dus gewoon af naar boven). Deze functie komt overeen met de math.ceil() functie in Python. Het getal n is het aantal waarden (de lengte) van de variabele x en s_X is de standaardafwijking van de variabele (zie het deeltje over spreidingsmaten). Het aantal klassen is m. De regel van Scott berekent eerst de klassenbreedte b en aan de hand daarvan het aantal klassen m.

In Python kan je deze waarden als volgt berekenen (we gebruiken variabelen `n` en `diskspace` en verwijderen eerst de NaN-waarden):

```
diskspace = laptops.diskspace.dropna()
n = len(diskspace)
```

```
import math

# Sturges
m = math.ceil(1 + math.log2(n))
print(f'm = {m}')
```

```
m = 11
```

```
from statistics import stdev

# Scott
b = 3.5 * stdev(diskspace) / (n ** (1 / 3))
m = math.ceil((diskspace.max() - diskspace.min()) / b)
print(f'm = {m}')
```

```
m = 13
```

```
# Excel
m = math.ceil(math.sqrt(n))
print(f'm = {m}')
```

```
m = 30
```

Het lijkt hier dus aangewezen om 11 tot 13 klassen te maken (30 is echt wel teveel). Je kan handmatig natuurlijk ook een ander aantal klassen maken, afhankelijk van je doel. Algemeen wordt wel aangenomen dat het aantal klassen steeds tussen 5 en 20 moet liggen om een duidelijk beeld te kunnen geven.

Als je weet hoeveel klassen je wil maken, kan je de frequenties ook heel snel als volgt berekenen (we maken hier 11 klassen):

```
diskspace.value_counts(bins=11).sort_index()
```

```
(87.363, 173.345]      81
(173.345, 258.391]    407
(258.391, 343.436]      0
(343.436, 428.482]      0
(428.482, 513.527]    323
(513.527, 598.573]      0
(598.573, 683.618]      0
(683.618, 768.664]      0
(768.664, 853.709]      0
(853.709, 938.755]      0
(938.755, 1023.8]      40
Name: diskspace, dtype: int64
```

Relatieve frequenties

Absolute frequenties tellen hoeveel keer een bepaalde waarde voorkomt. Maar deze waarde heeft eigenlijk niet zo veel betekenis. Als je veel metingen doet, zullen de absolute frequenties ook groter zijn. We zijn dikwijls meer geïnteresseerd hoeveel een waarde voorkomt ten opzichte

van het totaal. Dan kunnen we dit ook in procenten uitdrukken en het geeft ineens ook een mogelijkheid om dit te vergelijken met andere studies waarin het aantal metingen verschillend is. We zoeken dus hoeveel procent van de waarden gelijk is aan een bepaalde waarde.

Dit noemen we relatieve frequenties. In Python bereken je ze als volgt:

```
laptops.brand.value_counts(normalize=True)
```

```
HP          0.227804
Dell        0.188084
Toshiba     0.151869
Acer        0.151869
Lenovo      0.091121
Asus        0.075935
Apple       0.075935
Medion      0.037383
Name: brand, dtype: float64
```

Als je dit in procenten wil zien, dan vermenigvuldig je deze cijfers met 100:

```
(laptops.brand.value_counts(normalize=True) * 100).round(1)
```

```
HP          22.8
Dell        18.8
Toshiba     15.2
Acer        15.2
Lenovo       9.1
Asus         7.6
Apple        7.6
Medion       3.7
Name: brand, dtype: float64
```

De functie `round()` wordt hier enkel gebruikt om de output iets leesbaarder te maken. Je kan deze ook weg laten. Van de onderzochte laptops is dus bijvoorbeeld 15.2% van het merk **Acer**.

Cumulatieve frequenties

Wanneer een variabele ordinaal meetniveau heeft, kunnen we nog andere frequenties bepalen. Neem bijvoorbeeld de generatie van de processor die er in de laptop aanwezig is (cpuGeneration). Deze variabele is ordinaal want de processoren worden steeds beter in de volgende volgorde: Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Skylake, Kabylake. Men nummert deze generaties ook dikwijls van 2 tot en met 7.

We kunnen ons nu afvragen: hoeveel laptops hebben een Haswell processor? Dit kunnen we doen door alle absolute frequenties van Sandy Bridge, Ivy Bridge en Haswell op te tellen. Op dezelfde manier kunnen we tellen hoeveel laptops van een andere generatie zijn of minder. We moeten dus telkens de absolute frequenties optellen. Dit noemt men de cumulatieve som.

In Python kan je deze berekenen met de functie `cumsum()`. Deze functie levert een lijst van waarden waarvan de eerste waarde gelijk is aan die van de oorspronkelijke lijst. De tweede waarde is gelijk aan de eerste twee waarden opgeteld, de derde waarde is gelijk aan de eerste drie waarden opgeteld, enzovoort.

Hier zie je een voorbeeldje om de werking duidelijk te maken:

```
getallen = pd.Series(range(1, 6), name='getallen')
cumulatieve_som = pd.Series(getallen.cumsum(), name='cumsum')
pd.concat([getallen, cumulatieve_som], axis=1)
```

	getallen	cumsum
0	1	1
1	2	3
2	3	6
3	4	10
4	5	15

Hier zie je hoe je de cumulatieve frequenties in Python kan berekenen:

```
laptops.cpuGeneration.value_counts().sort_index().cumsum()
```

```
Broadwell       218
Haswell         384
Ivy Bridge      491
Kabylake        634
Sandy Bridge    697
Skylake         852
Name: cpuGeneration, dtype: int64
```

Er zijn dus 336 laptops die een Haswell processor hebben of minder. Bemerk dat de laatste cumulatieve frequentie altijd gelijk moet zijn aan n. We hebben de `NaN` waarden weggelaten. Het heeft toch geen zin om te weten hoeveel computers `NaN` of minder hebben als processor. Wat ook belangrijk is, is dat we de waarden uitdrukkelijk niet sorteren volgens frequentie, maar wel volgens de volgorde die we zelf bepaalden (`sort_index()`). Dit is zeer belangrijk omdat de volgorde nu bepaalt hoe we de frequenties optellen.

Cumulatieve percentages

De cumulatieve frequenties hebben weer niet zoveel zin op zich. Het is handiger om ze uit te drukken als percentages. Dit doe je als volgt:

```
(laptops.cpuGeneration.value_counts(normalize=True).sort_index().cumsum() *
100).round(1)
```

```
Broadwell        25.6
Haswell          45.1
Ivy Bridge       57.6
Kabylake         74.4
Sandy Bridge     81.8
Skylake         100.0
Name: cpuGeneration, dtype: float64
```

Dit noemt men **cumulatieve percentages**. De functie `round()` dient weer gewoon om de getallen leesbaarder te maken. We zien dus bijvoorbeeld dat 83,2% van de laptops een Skylake processor heeft of minder (waarbij we de `NaN` waarden dus niet meegeteld hebben).

Men noemt de cumulatieve percentages ook wel *percentielscores*. Dat gebeurt meestal in een context waarin de mogelijke waarden scores zijn. Een voorbeeldje moet dit duidelijk maken. Stel dat je behaalde punten op een examen in een variabele 'scores' hebt. Deze scores zijn gehele waarden tussen 0 en 10. Als je de percentielscores berekent, vind je bijvoorbeeld volgende waarden:

	percentiel score
0	1.1
1	10.3
2	11.4
3	33.6
4	55.4
5	65.1
6	84.2
7	98.1
8	100.0
9	100.0
10	100.0

Hier zie je dus dat 84,2 procent van de studenten 6 of minder haalde. Dat wil dus zeggen dat de toets heel moeilijk was. 55,4 procent was ook gebuisd (4/10 of minder). Om te compenseren voor deze moeilijke test, zou je dus de percentielscore kunnen gebruiken in plaats van de oorspronkelijke score. Je krijgt dan een score ten opzichte van de groep. In dit geval zou iemand met 4/10 dan geslaagd zijn. Het nadeel van deze aanpak is dat je dan op voorhand weet dat 50 procent van de deelnemers zal slagen, ongeacht het kennisniveau.

By Wouter Deketelaere, Kris Demuynck, Wim Dekeyser, Jan Van Overveldt