

BENOÎT DEPAIRE, GERT JANSSENSWILLEN

# EXPLORATIEVE EN DESCRIPTIEVE DATA ANALYSE



# *Contents*

<i>Voorwoord</i>	7
<i>Hoe de lecture notes te gebruiken</i>	7
<i>Hoe de tutorials te gebruiken</i>	8
<i>Over de auteurs</i>	8
<b>1    <i>Introductiecollege</i></b>	<b>9</b>
<b>1.1    <i>Data science?</i></b>	<b>9</b>
<b>1.2    <i>Gastcollege Bart Van Proeyen - Kwarts</i></b>	<b>20</b>
<b>1.3    <i>Data &amp; Data types</i></b>	<b>20</b>
<b>1.4    <i>Referenties</i></b>	<b>24</b>
<b>2    <i>Data visualisatie</i></b>	<b>25</b>
<b>2.1    <i>Univariate visualisaties (1 variabele)</i></b>	<b>26</b>
<b>2.2    <i>Bivariate visualisatie (2 variabelen)</i></b>	<b>36</b>
<b>2.3    <i>Multivariate visualisaties (meer dan 2 variabelen)</i></b>	<b>50</b>
<b>2.4    <i>Visualisaties voor communicatie</i></b>	<b>53</b>
<b>2.5    <i>How charts lie</i></b>	<b>57</b>
<b>2.6    <i>Referenties</i></b>	<b>58</b>
<b>3    <i>Tutorial - Data vizualisatie</i></b>	<b>59</b>
<b>3.1    <i>Before you start</i></b>	<b>59</b>
<b>3.2    <i>Introduction</i></b>	<b>59</b>

4	benoît depaire, gert janssenswillen	
3.3	<i>Different geometrics</i>	61
3.4	<i>Enhancing the appearance of our plots</i>	75
3.5	<i>Advanced plots</i>	84
3.6	<i>Background material</i>	90
4	<i>Descriptieve statistieken</i>	91
4.1	<i>Beschrijvende statistieken versus exploratieve plots</i>	91
4.2	<i>Notatie</i>	91
4.3	<i>Data</i>	92
4.4	<i>Univariate statistieken</i>	92
4.5	<i>Bivariate statistieken</i>	96
	<i>Referenties</i>	101
5	<i>Tutorial - Descriptieve statistieken</i>	103
5.1	<i>Before you start</i>	103
5.2	<i>Introduction</i>	104
5.3	<i>Helpful dplyr functions</i>	104
5.4	<i>Univariate analysis of a continuous variable</i>	108
5.5	<i>Bivariate analysis of a continuous variable with respect to a categorical variable</i>	111
5.6	<i>Univariate analysis of a categorical variable</i>	114
5.7	<i>Bivariate analysis of a continuous variable with respect to another continuous variable</i>	
	127	
5.8	<i>Bivariate analysis of a categorical variable with respect to another categorical variable</i>	
	131	
5.9	<i>Background material</i>	141
6	<i>Storytelling - van data tot inzicht</i>	143
7	<i>Data voorbereiden</i>	145
7.1	<i>Beginnen bij het begin</i>	145
7.2	<i>Data inlezen</i>	146

7.3	Dataproblemen identificeren en corrigeren	157
7.4	Data opwaarderen	169
	Referenties	176
8	Tutorial - Data voorbereiden	177
8.1	Before you start	177
8.2	Introduction	178
8.3	Reading data	178
8.4	Cleaning Data	185
8.5	Transforming Data	210
8.6	Using transformations in visualizations.	217
8.7	Background Material	225
9	Tidy data	227
9.1	Inleiding	227
9.2	Case: NYC Vluchten 2013	227
9.3	Data in een lang formaat plaatsen (voor visuele analyses)	230
9.4	Data in een breed formaat plaatsen (voor overzichtelijke tabellen)	235
9.5	Referenties	237
10	Tutorial - Tidy data	239
10.1	Before you start	239
	Disclaimer	239
10.2	Merging data	239
10.3	Transforming data	252
10.4	[Case study]: WHO	260
11	Tijdsdata	273
11.1	Inleiding	273
11.2	Periode-data	274
11.3	Analyseren van tijdgerelateerde data	275
11.4	Referenties	279

<i>12 Tutorial - tijdsdata</i>	281
<i>12.1 Before we start</i>	281
<i>12.2 Introduction</i>	281
<i>12.3 What's in a date?</i>	282
<i>12.4 There's always time</i>	291
<i>12.5 My time is not your time</i>	292
<i>12.6 Extract information from dates</i>	294
<i>12.7 Time differences</i>	296
<i>12.8 Arithmetics</i>	301
<i>12.9 Interval</i>	302

# *Voorwoord*

Dit boek bevat de lecture notes en tutorials voor het opleidingsonderdeel “Exploratieve en Descriptieve Data Analyse” (1ste Ba Handelsingenieur/Handelsingenieur in de Beleidsinformatica) aan de Universiteit Hasselt. De lectures notes dienen ter begeleiding van de hoorcolleges, terwijl de tutorials telkens een vervolg zijn hierop ter voorbereiding van de werkzittingen.

## *Hoe de lecture notes te gebruiken*

Het idee van de lecture notes is om een begeleidende tekst aan te reiken ter ondersteuning van de slide-decks die gebruikt worden tijdens de hoorcolleges. Deze tekst is “bullet-point” gewijs opgebouwd en helpt het verhaal dat tijdens het hoorcollege wordt verteld terug op te roepen. Daarnaast zal er per hoofdstuk ook een *referentielijst* aangereikt worden met werken die de diverse topics in detail uitleggen.

- Neem de lecture notes mee naar het hoorcollege (digitaal of geprint), en gebruik deze om belangrijke aspecten tijdens het hoorcollege te markeren en korte nota's toe te voegen. Ga zeker niet de volledige uitleg van het hoorcollege noteren. Dit is vaak niet mogelijk en indien je er toch in slaagt zal je tijdens het hoorcollege niet in staat zijn geweest om een eerste keer te reflecteren over de leerstof.
- Bestudeer na de les de lecture notes samen met de notities. Controleer of je alles begrijpt en waar nodig noteer je aanvullingen. Probeer een overzicht te verkrijgen van de diverse concepten die je tijdens het hoorcollege bestudeerd hebt en tracht na te gaan hoe je deze inzichten kunt gebruiken voor exploratieve en descriptieve data analyse.
- (optioneel) Lees de bronnen in de referentielijst. Indien er elementen niet duidelijk zijn in je eigen notities of de lecture notes, dan ga je best gericht op zoek naar de antwoorden op je vragen in de referentiewerken.

### *Hoe de tutorials te gebruiken*

De tutorials zijn een logisch gevolg op de leerstof in het hoorcollege en bereiden je voor op de oefening in de werkzittingen. In de tutorials worden de concepten uit het hoorcollege geïllustreerd in R code. Het is niet enkel de bedoeling de tutorials te lezen, maar ook zelf de voorbeelden uit te proberen in Rstudio. Je vindt de nodige datasets hiervoor telkens terug op Blackboard. Zonder de tutorials grondig te bekijken heeft het geen zin om naar de werkzittingen te komen.

### *Over de auteurs*

Prof. dr. Benoît Depaire is hoofddocent Beleidsinformatica aan de Universiteit Hasselt en lid van de onderzoeksgroep Beleidsinformatica. Zijn onderzoeksinteresse situeert zich rond de topics data mining, data-gedreven procesanalyse en statistiek met een focus op de extractie van bedrijfskundige inzichten uit data. Als voorzitter van het onderwijsmanagementteam voor de opleiding Beleidsinformatica, alsook op basis van zijn jarenlange ervaring als docent, heeft hij een onderwijsexpertise uitgebouwd rond diverse topics zoals projectmanagement, business process management, data analyse en de rol van IT in de moderne bedrijfswereld. Daarnaast houdt hij zich ook bezig met dienstverlening naar de bedrijfswereld toe door middel van gastlezingen, adviesverstrekking en toegepaste onderzoeksprojecten.

dr. Gert Janssenswillen is academisch medewerker aan de faculteit Bedrijfseconomische Wetenschappen (BINF Business Informatics) van de Universiteit Hasselt. Na het verwerven van zijn diploma Handelsingenieur in de Beleidsinformatica in 2014, behaalde hij in 2019 een PhD in de Bedrijfseconomie aan de Universiteit Hasselt. Tijdens zijn doctoraat ontwikkelde hij de open-source R packages bupaR, welke wereldwijd wordt gebruikt door bedrijven en organisaties voor de analyse van bedrijfsprocessen. Hij spreekt regelmatig op business process management - conferenties, zoals BPM, ICPM, en SIMPDA, alsook R conferenties, zoals useR. Sinds 2019 is hij lid van het organisatie comité van de Europese R User Meeting.

# 1

## *Introductiecollege*

### *1.1 Data science?*

#### *1.1.1 Netflix*

- Netflix Prize (2006)
  - Wereldwijde open competitie voor de constructie van een nieuw algoritme dat moest voorspellen hoe goed een klant een film zou beoordelen op basis van zijn of haar filmvoorkeuren.
  - Winnaar was het team dat als eerste een verbetering van 10% kon realiseren ten opzichte van het algoritme van Netflix zelf.
  - Eerste prijs was 1 miljoen USD.
  - Hiervoor stelde Netflix een dataset ter beschikking met 100 miljoen filmbeoordelingen van 500 000 klanten met betrekking tot 18 000 films.
- Het kunnen voorspellen hoe hun klanten gaan reageren op specifieke films/series laat Netflix toe hun aanbod aan films en series te optimaliseren om het huidige klantenbestand te behouden en nieuwe klanten aan te trekken.
- De hoeveelheid data die door Netflix wordt verzameld is enorm.
  - In 2016 had Netflix 93.8 miljoen leden.
  - Netflix weet wanneer je pauzeert.
  - Netflix weet op welke dagen en welke uren je kijkt.
  - Netflix weet wat je kijkt.
  - Netflix weet van waar je kijkt.
  - Netflix weet op welk soort toestellen je kijkt.
  - Netflix weet wanneer je definitief stopt met het bekijken van een serie.
  - Netflix weet hoe snel je verschillende afleveringen van een serie achter elkaar kijkt.
  - Netflix weet welke titels je zoekt.

- Netflix komt op deze manier zeer veel te weten over het kijkgedrag van zijn klanten en kan op basis van deze inzichten betere beslissingen nemen. Bijvoorbeeld:
  - Netflix ontdekt uit haar data dat 40% van haar klanten een serie zijn beginnen te kijken die door het oorspronkelijke productiehuis is stopgezet.
  - Stel dat Netflix uit de data ook ontdekt dat 85% van deze klanten de serie volledig uitkijken zonder dat het tempo waartegen men afleveringen kijkt significant afneemt.
  - Op basis van deze inzichten kan Netflix eventueel beslissen om de rechten van de serie te kopen (die goedkoop zullen zijn aangezien de serie was stopgezet) en zelf een nieuw seizoen voor de serie te maken.
- House of Cards
  - Netflix deed het beste bod voor de serie House of Cards waardoor het won van kanalen zoals HBO.
  - Ze kochten initieel 2 seizoenen van de serie waar een prijskaartje aan vast hing van meer dan 100 miljoen dollar.
  - Deze beslissing was voor een groot stuk gebaseerd op data:
    - \* Netflix leerde uit haar data dat haar klanten geïnteresseerd waren in producties van regisseur David Fincher.
    - \* Netflix leerde uit haar data dat haar klanten geïnteresseerd waren in de oorspronkelijke Britse versie van House of Cards.
    - \* Netflix leerde uit haar data dat haar klanten geïnteresseerd waren in producties met Kevin Spacey.
  - Maar ook na de beslissing om deze serie te maken, bleef Netflix haar data gebruiken om slimme beslissingen te nemen.
    - \* Er werden verschillende trailers gemaakt en afhankelijk van je voorkeuren kreeg je een trailer op maat te zien.
    - \* Klanten die vooral graag Kevin Spacey zagen, kregen een trailer waar vooral Kevin Spacey in voorkwam.
    - \* Klanten die vooral geïnteresseerd waren in films van David Fincher, kregen een trailer te zien die de typische “look&feel” had van David Fincher.
    - \* Klanten die ook de Britse versie hadden gezien, kregen een trailer te zien die vooral op het verhaal focuste.

### *1.1.2 Waar komt data vandaan?*

- Data over een maatschappij
  - Het verzamelen van data is iets dat teruggaat tot in de oudheid.
  - Denk hierbij aan de volkstellingen die reeds plaatsvonden ten tijden van de Romeinen.

- Een volkstelling gaat alle inwoners van een bevolking registreren, samen met diverse kenmerken zoals burgerlijke status, leeftijd, geslacht, enzovoort.
  - Volkstellingen waren en zijn nog steeds belangrijk voor een overheid om de impact van haar openbaar beleid te kunnen inschatten.
- Scientific Management
    - Frederick Taylor
    - Eind 19de eeuw
    - Benaderde het organiseren van werk op een wetenschappelijke manier.
    - Ging data verzamelen om vervolgens te analyseren hoe men werk efficiënter kon organiseren.
    - Een van de eerste vormen van dataverzameling en -analyse om bedrijfswaarde (productiviteit) te creëren.
    - Beperkt in hoeveelheid data omdat registratie en analyse nog manueel gebeurde.
- Het ontstaan van het digitale tijdperk
    - Met de uitvinding van de computer tijdens en na de tweede wereldoorlog, is de mensheid het digitale tijdperk ingegaan.
    - De computer zorgt ervoor dat we data in een digitale vorm (als een reeks van één en nullen) opslaan. Dit biedt het voordeel dat exacte kopieën van de data gemaakt kunnen worden met één muisklik.
- Digitalisatie van de werkvloer
    - Computers op de werkvloer dateert terug tot midden vorige eeuw, maar de grote doorbraak komt er met de opkomst van de personal computer
      - \* 1977: Apple Home Computer II
      - \* 1981: IBM Personal Computer
      - \* Eind jaren 80, begin jaren 90 was de PC wijdverspreid op de werkvloer.
      - \* Dit liet toe meer data te registreren, maar deze was nog moeilijk te delen met andere computers.
    - Opkomst Internet / WWW in de bedrijfswereld
      - \* 1990: De technologie voor WWW werd publiek gedeeld door Tim Berners-Lee.
      - \* Dankzij WWW en internettechnologie werd het steeds een- voudiger om digitaal werk te delen.
    - Opkomst van e-commerce
      - \* 1995: Begin van dot-com bubble/hype.

- \* Opkomst van digitale ondernemingen (vb. Amazon, Netflix, Google, ...).
- \* Digitale handel maakt het eenvoudiger om gegevens hierover te registreren.

- Digitalisatie van mensen

- Opkomst Web 2.0 (begin 2000)
  - \* Inhoud van het web wordt nu gecreëerd door de bezoekers/gebruikers/klanten.
  - \* Websites worden dynamisch (passen zich aan de context en bezoeker aan).
- Opkomst sociale media
  - \* Gebruikers gaan spontaan hun leven digitaliseren.
  - \* Hiervoor worden diverse media gebruikt (foto, video, tekst, ...).
  - \* Facebook, Twitter, Instagram, Persoonlijke blogs, . . . .
  - \* Nog nooit heeft zo'n groot deel van de wereldbevolking informatie gecreëerd en gedeeld met de rest van de wereld.

- Digitalisatie van dingen

- Opkomst goedkope sensoren
- Steeds meer “dingen” (machines, auto’s, huishoudtoestellen, huizen, steden, . . . ) worden ‘intelligent’.
- Internet of Things (IoT): Al deze intelligente dingen worden via het Internet met elkaar verbonden.
- De hoeveelheid data die hiermee gegenereerd zal worden is ongezien.
- Volgens IDC studie waren in 2013 reeds 7% van de “verbindbare dingen” geconecteerd aan het Internet of Things.
- In dezelfde studie voorspellen ze dat dit zal stijgen tot 15% in 2020.
- In 2013 werd 2% van alle data in het digitaal universum geproduceerd door het IoT.
- Verwacht wordt dat dit zal stijgen tot 10% in 2020.

### *1.1.3 Hoeveel data is er beschikbaar?*

- De hoeveelheid data die de laatste decennia gegenereerd en opgeslagen wordt is enorm toegenomen.
- Deze groei is exponentieel (de groei gaat steeds sneller). Meer specifiek verdubbelt de hoeveelheid data in het digitaal universum iedere 2 jaar.
- Volgens een studie van IDC, bestond het digitaal universum in 2013 uit 4.4 Zetabytes data

- 1 Zetabyte = 1024 Exabytes
- 1 Exabyte = 1024 Petabytes
- 1 Petabyte = 1024 Terabytes
- 1 Terabyte = 1024 Gigabytes
- Volgens dezelfde studie zal het digitaal universum in 2020 uit 44 Zetabytes bestaan
- Echter, slechts 22% van deze data (in 2013) is geschikt voor analyse.
- Er wordt geschat dat dit zal stijgen tot 35% in 2020.
- Slechts 5% van de geschikte data voor analyse wordt feitelijk geanalyseerd (2013).

#### *1.1.4 Waar wordt data voor gebruikt in de bedrijfswereld?*

- Er zijn verschillende redenen waarom bedrijven data bijhouden. Deze kunnen we onderverdelen in volgende categorieën: Geschiedenis bijhouden, beslissingen nemen en voorspellingen maken.

#### *Geschiedenis bijhouden*

- Je registreert feiten zodat je achteraf met zekerheid kunt weten wat de realiteit in het verleden was.
- Dit is belangrijk als je wilt evalueren of een bedrijf goed beheerd wordt. Hiervoor heb je inzicht in het verleden nodig.
- De gegevens die worden bijgehouden in een boekhouding en jaarrekeningen zijn hier een typisch voorbeeld van.

#### *Dagelijkse werking*

- Opdat een bedrijf zijn dagelijkse werking kan uitvoeren, is het essentieel een up to date zicht te hebben van de werkelijkheid. Als een klant belt met een klacht over een levering, dan moet je als onderneming kunnen achterhalen wat de klant precies besteld heeft, of dit reeds geleverd is, of de klant al betaald heeft, enzovoort. Zonder deze informatie kan een onderneming haar dagelijkse werking niet garanderen.
- Om de dagelijkse werking te verzekeren, hebben bedrijven altijd al data bijgehouden. Denk maar aan informatie over aankoop- en verkooporders, de financiële gegevens in de boekhouding, de afschriften van een bank, de productieplanning, enzovoort.

#### *Beslissingen nemen*

- Een bedrijf neemt dagelijks talrijke beslissingen op verschillende niveaus

- Operationeel.
  - \* Vb: Moet ik een nieuwe bestelling plaatsen voor grondstof X of hebben we nog genoeg voorraad?
  - \* Dit zijn typisch zeer frequente beslissingen die nodig zijn om de dagelijkse werking te garanderen.
  - \* Deze beslissingen worden genomen door mensen op de werkvloer of door het (lager) management.
- Tactisch/Management.
  - \* Vb: Sluit ik best een exclusief contract af met 1 leverancier voor grondstof X voor een vaste periode en tegen een vaste verkoopsprijs of koop ik wanneer nodig tegen de marktprijs?
  - \* Deze beslissingen worden minder frequent genomen dan operationele beslissingen en zijn typisch nodig om de werking van de onderneming op middellange termijn te optimaliseren.
  - \* Deze beslissingen worden genomen door het management van een onderneming en hebben een aanzienlijke impact.
- Strategisch.
  - \* Vb: Zullen we grondstof X aankopen op de markt of beslissen we deze grondstof zelf te produceren?
  - \* Deze beslissingen hebben een zeer grote impact op de onderneming en worden niet frequent genomen. Ze vergen typisch ook lange voorbereidingstijd en bepalen de richting en toekomst van de onderneming op lange termijn.
  - \* Deze beslissingen worden genomen door het topmanagement van een onderneming.
- Data kan bedrijven helpen bij het nemen van beslissingen.
  - Dit betekent echter niet dat beslissingen enkel en alleen op data gebaseerd zijn.
  - Vaak wordt data gecombineerd met ervaring en expertise om een beslissing te nemen.
- Bij het nemen van beslissingen op basis van data, kunnen we zowel patronen in historische data gebruiken alsook voorspellingen op basis van data.

#### *1.1.5 Waarover verzamelen bedrijven data*

- Het ultieme doel van een onderneming is gegevens te verzamelen die hen toelaten om het gedrag van hun omgeving beter te begrijpen, alsook de werking van hun eigen onderneming.
- Onder omgeving verstaan we:
  - Klanten
  - Concurrenten

- Leveranciers
- Alternatieve markten
- Overheden
- Onder werking van eigen onderneming vertaan we o.a.:
  - Werknemers
  - Processen
  - Producten
  - Diensten

#### *1.1.6 Van data tot ‘actionable insights’*

- Data
  - Data verwijst typisch naar de gegevens die geregistreerd en opgeslagen worden.
  - Data beschrijft een heel klein aspect van een realiteit (bijvoorbeeld op welk exact tijdstip ben ik aflevering 2 van “House of Cards” beginnen te kijken).
  - Data op zich heeft echter heel weinig waarde.
- Informatie
  - Als we echter data gaan analyseren, dan kunnen we dit transformeren tot informatie.
  - Informatie beschrijft een realiteit en gaat typisch op zoek naar patronen in de data en afwijkingen op deze patronen.
  - Bijvoorbeeld: Ik kijk typisch House of Cards gedurende de week om 20u00 's avonds, maar stop meestal met kijken om 20u30, waardoor ik in de week zelden een aflevering in 1 keer uitkijk.
  - Informatie is beschrijvend en zegt ons WAT de realiteit is.
- Actionable Insights
  - Actionable Insights is informatie die ons niet enkel zegt WAT de realiteit is, maar ons ook het inzicht verschafft HOE we moeten handelen.
  - Niet alle informatie is actionable.
  - Op basis van actionable insights en in combinatie met onze eigen ervaringen en kennis die we reeds bezitten, komen we soms tot inzichten die beschrijven HOE we moeten handelen.

#### *1.1.7 Data Scientists*

- Nieuwe jobomschrijving.
- Verantwoordelijk om data te transformeren naar ‘actionable insights’ en hier iets mee te doen om bedrijfswaarde te creëren.
- Omschreven als meest ‘sexy job’ van de 21ste eeuw door HBR

- Opvolgers van de Wall Street ‘Quants’ uit de jaren 80 en 90.
- Vaardigheden
  - Bedrijfskunde
    - \* Productontwikkeling
    - \* Management
  - Machine Learning / Big Data
    - \* Ongestructureerde data
    - \* Gestructureerde data
    - \* Machine Learning
    - \* Big Data
  - Wiskunde en Operationeel Onderzoek
    - \* Optimalisatie
    - \* Wiskunde
    - \* Simulatie
  - Programmeren
  - Statistiek
    - \* Visualisatie
    - \* Tijdreeksanalyse
    - \* Wetenschappelijk onderzoek
    - \* Data Manipulatie
- 4 profielen van data scientists
  - Data Businessperson
    - \* Focust voornamelijk hoe data omzet kan genereren.
    - \* Vaak in een leidinggevende rol.
    - \* Werken zelf ook met data en beschikken over de nodige technische vaardigheden.
  - Data Creatives
    - \* Zijn in staat een volledige data analyse zelfstandig uit te voeren.
    - \* Hebben een hele brede bagage aan technische vaardigheden.
    - \* Beschikken in zekere mate over bedrijfskundige vaardigheden.
    - \* Gaan vaak innovatief om met data.
  - Data Developer
    - \* Is voornamelijk gefocust op de technische uitdagingen met betrekking tot het beheer van data.
    - \* Sterke programmeervaardigheden. Zijn in staat productie-code te schrijven.
    - \* Zijn sterk in het gebruik van machine learning technieken.
  - Data Researcher

- \* Vaak mensen met een wetenschappelijke achtergrond (doctoraat).
- \* Sterk in statistische vaardigheden en wetenschappelijk onderzoek.

#### *1.1.8 Verschillende soorten van data analyse*

- Er zijn verschillende manieren om data analyse taken te classificeren.
- De classificatie die we hier hanteren is gebaseerd op het doel van de data analyse.

#### *Descriptieve data analyse*

- Deze analyse focust zich op het beschrijven van de data.
- Deze analyse gaat over het samenvatten van de grote hoeveelheid data in enkele statistische cijfers en grafieken.
- Deze analyse wordt gebruikt als je een grote hoeveelheid data krijgt en je snel inzicht wilt krijgen in de data.
- Voorbeelden:
  - Je hebt een dataset met alle studieresultaten van de studenten van 1ste bachelor HI/BI en je wilt weten wat de gemiddelde score is per vak.
  - Je hebt de verkoopscijfers van het afgelopen jaar en je wil weten welke drie producten het beste verkochten (zowel in aantal als in omzet).
- Descriptieve data analyse zegt alleen iets over de realiteit die door de data is beschreven. Je kan **geen** conclusies trekken die verder reiken dan de geobserveerde data.
- Je kan een descriptieve data analyse vergelijken met het werk van een detective die als taak heeft een beschrijving te maken van de misdaadscene.

#### *Exploratieve data analyse*

- Exploratieve analyse focust op het verkennen van de data en het zoeken naar interessante patronen en afwijkingen van deze patronen.
- Net als bij descriptieve data analyse zal exploratieve analyse de beschikbare data beschrijven en zeggen de resultaten **niets** over ongeobserveerde feiten.
- In tegenstelling tot bij descriptieve data analyse, gaat exploratieve data analyse verder dan het louter beschrijven van de data en tracht men interessante patronen te ontdekken in de data.
- Voorbeelden:

- Zijn er specifieke kenmerken van studenten die sterk gerelateerd zijn aan hun studieresultaten.
- Zijn er opmerkelijke verschillen tussen vakken wat betreft de punten die behaald worden. Zo ja, wat zijn dan deze verschillen.
- Zijn er producten in ons gamma die gevoelig zijn voor seizoenseffecten?
- Je kan een exploratieve data analyse vergelijken met het werk van een detective die als taak heeft verbanden te ontdekken tussen verschillende bewijsstukken om zo inzicht te verschaffen wat er gebeurd is tijdens de misdaad.

#### *Confirmatorische data analyse*

- Confirmatorische analyse focust op het bevestigen of weerleggen van vermoedens die men heeft met behulp van de beschikbare data.
- In tegenstelling tot descriptieve en exploratieve data analyse zal men bij confirmatorische data analyse wel conclusies trekken die verder gaan dan de geobserveerde data.
- Omdat confirmatorische data analyses ook uitspraken doen over ongeobserveerde data, is er altijd een mate van onzekerheid over de correctheid van de resultaten.
- Voorbeelden:
  - Halen studenten met 8u Wiskunde achtergrond betere resultaten dan studenten met 6u Wiskunde achtergrond? In welke mate zijn we zeker dat dit voor alle studenten geldt en niet enkel voor de studenten waarover we data hebben?
  - Verkoopt product X beter bij mannen dan bij vrouwen? In welke mate zijn we zeker dat dit verschil niet een toevalligheid in de data is?
- Je kan een confirmatorische data analyse vergelijken met het werk van een rechter die op basis van het aangeboden bewijsmateriaal moet beslissen of er genoeg bewijs is om iemand te veroordelen van de misdaad.

#### *Predictieve data analyse*

- Het doel van predictieve analyse is om op basis van de beschikbare data voorspellingen te doen over de toekomst of over nieuwe of alternatieve situaties.
- Net als bij confirmatorische data analyse zal predictieve data analyse uitspraken doen die ook van toepassing zijn voor ongeobserveerde feiten/situaties.
- Bijgevolg is er net als bij confirmatorische data analyse dus een zekere onzekerheid over de conclusies die men trekt.

- Voorbeelden:
  - Zal een studente die met meer dan 80% haar diploma van het middelbaar onderwijs behaalt slagen in eerste zit voor het vak *Exploratieve en Descriptieve Data Analyse*?
  - Zullen de verkoopcijfers van product Y het komende jaar verder stijgen en met hoeveel procent?
- Je kan een predictieve data analyse vergelijken met het werk van een detective die op basis van het bewijsmateriaal op een misdaadscene moet voorspellen waar en wanneer de dader opnieuw zal toeslaan.

#### 1.1.9 De kunst van data analyse

- Data analyse is een kunst. Net als bij iedere kunst, kunnen we hierbij drie componenten onderscheiden: kennis en vaardigheden, ervaring en creativiteit.
- Kennis en vaardigheden
  - Als data analist moet je de juiste hulpmiddelen kunnen identificeren voor het voorgelegde probleem.
  - Deze diverse hulpmiddelen moet je zo goed mogelijk beheersen.
  - Bij (exploratieve) data analyse gaat het hierbij zowel over analyses als over datavaardigheden.
  - Dit aspect kun je leren en laat je reeds toe om correcte analyses uit te voeren.
- Ervaring
  - Hoe meer data je analyseert, hoe beter je er in wordt.
  - Ook laat ervaring toe om sneller vaste patronen in je werk te herkennen en efficiënter te worden in wat je doet.
  - Ervaring is ook essentieel om complexere uitdagingen beheersbaar te maken.
  - Dit deel kunnen we je niet ‘leren’, maar heb je wel volledig in de hand.
- Creativiteit
  - Een kunstenaar die over kennis, vaardigheden en ervaring beschikt, maar creativiteit ontbreekt, kan perfecte replica’s maken van een kustwerk, maar kan zelf geen nieuwe kunst creëren.
  - Creativiteit is in staat zijn op een nieuwe en onverwachte manier naar data te kijken en deze te visualiseren.
  - Het is niet zeker dat dit aspect aan te leren is. Maar dit hoeft niet te verhinderen dat je een goede data scientist wordt, zolang je maar voldoende aandacht besteedt aan de andere twee componenten.

### 1.1.10 De kracht van descriptieve en exploratieve data analyse

<https://www.youtube.com/watch?v=RUwS1uAdUcI>

## 1.2 Gastcollege Bart Van Proeyen - Kwarts

Zie bijhorende slidedeck (Blackboard).

## 1.3 Data & Data types

- Data is het resultaat van een meting van een attribuut van een specifiek object met een specifiek meetinstrument.
  - Het object verwijst naar wat je gaat meten.
    - \* vb.: Student “Karel Jespers”.
  - Een object hoort meestal tot een verzameling van objecten. Deze verzameling wordt ook wel de populatie genoemd.
    - \* vb.: Populatie “Studenten 1ste Ba HI/BT”.
  - Een specifiek object uit de populatie wordt ook wel element genoemd.
    - \* vb.: “Karel Jespers” is een element uit de populatie “Student 1ste Ba HI/BT”.
  - Je meet altijd een specifiek aspect van het object. Omdat de meetwaarde van dit aspect kan variëren tussen verschillende objecten (elementen) in je verzameling (populatie), worden zulke aspecten ook variabelen genoemd.
    - \* vb.: Lengte is een specifiek aspect (variabele) van de student “Karel Jespers” (element).
  - De meting gebeurt met behulp van een meetinstrument. Het is belangrijk te beseffen dat een meetinstrument altijd een zekere nauwkeurigheid heeft (tot hoeveel cijfers na de komma exact kan je meten?) en mogelijk ook onderhevig kan zijn aan willekeurige en/of systematische meetfouten.
    - \* vb.: Student “Karel Jespers” wordt gemeten met een meetlat bevestigd tegen de muur. De meetlat heeft een nauwkeurigheid van 1cm, dus we kunnen zijn lengte niet uitdrukken in millimeters. Verder is de meetlat 2cm te laag opgehangen. Bijgevolg is er een systematische meetfout van 2cm. Tenslotte wordt de meting geregistreerd door een arts die vluchtig kijkt waar de student uitkomt op de meetlat. Het is dus niet onmogelijk dat de werkelijke lengte (willekeurig) afwijkt van de geregistreerde lengte.
    - \* Tenzij anders vermeld wordt, gaan we in dit hoofdstuk uit van meetinstrumenten met oneindige nauwkeurigheid en zonder meetfouten.

- De uitkomst van een meting voor een specifiek element wordt de waarde genoemd.
  - \* vb.: 1m80 is de waarde van de variabele “lengte” voor element “student Karel Jespers”

### 1.3.1 Dataset

- Een dataset is een verzameling van data waarbij
  - Iedere rij één element uit de populatie voorstelt.
  - Iedere kolom een variabele is die gemeten wordt.
  - De verschillende rijen verschillende elementen uit dezelfde populatie voorstellen.
  - De waarde in een cel de meting is van de betreffende variabele voor het betreffend element.

luchthaven	maatschappij	datum	vertrek_vertraging	aankomst_vertraging	afstand	vliegtijd
EWR	United Air Lines	2013-01-01	2	11	1400	227
	Inc.	05:15:00				
LGA	United Air Lines	2013-01-01	4	20	1416	227
	Inc.	05:29:00				
JFK	American	2013-01-01	2	33	1089	160
	Airlines Inc.	05:40:00				
LGA	Delta Air Lines	2013-01-01	-6	-25	762	116
	Inc.	06:00:00				
EWR	United Air Lines	2013-01-01	-4	12	719	150
	Inc.	05:58:00				
EWR	JetBlue Airways	2013-01-01	-5	19	1065	158
		06:00:00				
LGA	ExpressJet	2013-01-01	-3	-14	229	53
	Airlines Inc.	06:00:00				
JFK	JetBlue Airways	2013-01-01	-3	-8	944	140
		06:00:00				
LGA	American	2013-01-01	-2	8	733	138
	Airlines Inc.	06:00:00				
JFK	JetBlue Airways	2013-01-01	-2	-2	1028	149
		06:00:00				

Table 1.1: Uitgaande vluchten NYC

### 1.3.2 Klassieke datatypologie

- Klassieke onderverdeling van data
  - Nominaal, Ordinaal, Interval en Ratio
  - Gebaseerd op de publicatie “On the Theory of Scales of Measurement” (1946)
    - \* Beschrijft een hiërarchie van ‘datatypes’
      - Alles wat ordinaal is, is ook nominaal, maar niet omgekeerd.
      - Alles wat interval is, is ook ordinaal, maar niet omgekeerd.
      - Alles wat ratio is, is ook interval, maar niet omgekeerd.
    - \* Identificeert geschikte statistische testen voor ieder type.
  - Ieder datatype voldoet aan één of meerdere van de volgende eigenschappen:

- Identiteit: Iedere waarde heeft een unieke betekenis.
- Grootorde: Er is een natuurlijke volgorde tussen de waarden.
- Gelijke intervals: Eenheidsverschillen zijn overal even groot. Dus het verschil tussen 1 en 2 is even groot als het verschil tussen 19 en 20.
- Absoluut nulpunt: De waarde 0 betekent dat er ook feitelijk niets aanwezig is van de variabele en is niet een arbitrair gekozen nulpunt.

### *Nominaal*

- Voorbeelden:
  - Geslacht: Man, Vrouw.
  - Ondernemingsvorm: vzw, bvba, nv.
- Voldoet enkel aan de eigenschap ‘identiteit’.
- Dit betekent dat we enkel concluderen of twee waardes gelijk zijn of niet. Er bestaat geen natuurlijke volgorde tussen de verschillende waardes.

### *Ordinaal*

- Voorbeeld:
  - Opleidingsniveau: Lager onderwijs, Middelbaar onderwijs, Hoger onderwijs.
  - Klantentevredenheid: Ontevreden, Matig tevreden, Tevreden, Zeer tevreden.
- Voldoet aan de eigenschappen ‘identiteit’ en ‘grootorde’.
- Dit betekent dat we niet alleen kunnen concluderen of twee waardes gelijk zijn of niet. Het is ook mogelijk te bepalen welke waarde ‘groter’ is.
- We kunnen echter niet zeggen hoeveel groter één waarde is dan de andere.

### *Interval*

- Voorbeeld:
  - Temperatuur (Celsius).
- Voldoet aan de eigenschappen ‘identiteit’, ‘grootorde’ en ‘gelijke intervals’.
- We kunnen nu twee waardes vergelijken, bepalen welke groter is alsook de verschillen tussen waardes met elkaar vergelijken.

- We kunnen dus stellen dat het verschil tussen 8 en 9 graden Celsius daadwerkelijk minder groot is dan het verschil tussen 12 en 20 graden Celsius.

### *Ratio*

- Voorbeeld:
  - Gewicht
  - Voldoet aan alle 4 de eigenschappen.
  - We kunnen verschillende gewichten met elkaar vergelijken, we kunnen bepalen wat zwaarder is en we kunnen gewichtsverschillen onderling vergelijken. Hierbij komt nu ook nog dat we kunnen zeggen hoeveel keer iets zwaarder is dan iets anders.
  - Dit is een gevolg van het feit dat de waarde 0 nu feitelijk betekent dat iets geen gewicht heeft.

#### *1.3.3 De klassieke datatypologie is misleidend*

- Voorbeeld:
  - Op een feestje wordt bij het binnengaan oplopende nummers toegewezen aan iedere gast, beginnend bij 1.
  - Tijdens het feestje wordt er een tombola georganiseerd en wie nummer 126 heeft, heeft gewonnen.
  - 1 gast vergelijkt dit nummer met haar kaartje en ziet dat ze gewonnen heeft. Zij beschouwde de waarde op haar ticket dus als een nominale variabele want het enige wat ze vergelijkt is of de waarde op haar ticket verschillend is van de winnende waarde.
  - Een andere gast kijkt naar zijn kaartje en ziet dat hij nummer 56 heeft. Hij concludeert dat hij te vroeg is binnengekomen en beschouwt de waarde op zijn kaartje dus als ordinaal.
  - Nog een andere gast heeft een kaartje met nummer 70 en beschikt over bijkomende data omtrent het ritme waarmee gasten zijn binnengekomen. Deze gast kan dus schatten hoeveel later hij had moeten binnengekomen om te winnen en interpreteert zijn nummer dus als een interval variabele.
- Dit voorbeeld illustreert dat het datatype niet een vaststaand kenmerk is van de data, maar afhankelijk is van de vraag die je tracht te beantwoorden en de extra informatie waarover je beschikt.

#### *1.3.4 Alternatieve datatypologie*

- Alternatieve taxonomie van data
  - Graden: vb. academische graad: “op voldoende wijze”, “onderscheiding”, “grote onderscheiding”, . . . (geordende labels)

- Rangordes: vb. plaats in voetbalklassemement: 1, 2, 3, . . . , 16  
(gehele getallen die beginnen bij 1)
- Fracties: vb. percentage opgenomen verlof: van 0% tot 100%  
(ligt tussen 0 en 1, als percentage uit te drukken).
- Aantallen: vb aantal kinderen: 0, 1, 2, . . . (niet-negatieve gehele waarden).
- Hoeveelheden: vb. inkomen (niet-negatieve reële waarden).
- Saldo: vb. winst (negatieve en positieve reële waarden).
- Voor deze cursus volstaat het meestal een onderscheid te maken tussen categorische en continue variabelen.
  - Categorisch: Nominaal + Ordinaal.
  - Continu: Interval + Ratio.

#### 1.4 Referenties

1. Data Scientist, the Sexiest Job of the 21st Centure
2. Netflix Prize
3. How Netflix Uses Analytics
4. The Digital Universe of Opportunities - website
5. The Digital Universe of Opportunities - videoclip
6. How the Computer Changed the Office Forever
7. History of Computers in the Workplace
8. Web 1.0, 2.0, 3.0
9. From Data to Understanding
10. Analyzing the Analyzers
11. Scales of Measurement
12. Nominal, Ordinal, Interval, and Ratio Typologies are Misleading

## 2

# Data visualisatie

- Vaak de eerste stap om zicht te krijgen op de data.
- Relatief eenvoudig om patronen te zien, maar minder geschikt om exacte waarden te zien.
- We moeten hierbij onderscheid maken tussen exploratieve visualisaties en informatieve visualisaties om een boodschap over te brengen.
  - Exploratieve visualisaties dienen om snel inzicht te krijgen in patronen in de data. Men besteedt hierbij veel minder aandacht aan de opmaak van de visualisatie. Vaak is deze visualisatie tijdelijk en niet bedoeld voor communicatie naar derden.
  - Communicatieve visualisaties dienen om een boodschap over te brengen aan derden. Hier dient men heel veel aandacht te besteden aan de opmaak zodat de boodschap duidelijk en helder gecommuniceerd wordt.
- We kunnen bij exploratieve visualisaties een onderscheid maken tussen univariate, bivariate en multivariate visualisaties.

De grafieken in dit hoofdstuk zijn gebaseerd op volgende dataset omrent vluchten vertrekkende uit New York.

```
## Rows: 329,174
## Columns: 7
## $ luchthaven      <fct> EWR, LGA, JFK, LGA, EWR, EWR, LGA, JFK, LGA, JF...
## $ maatschappij    <chr> "United Air Lines Inc.", "United Air Lines Inc...."
## $ datum           <dttm> 2013-01-01 05:15:00, 2013-01-01 05:29:00, 2013...
## $ vertrek_vertraging <dbl> 2, 4, 2, -6, -4, -5, -3, -3, -2, -2, -2, -2...
## $ aankomst_vertraging <dbl> 11, 20, 33, -25, 12, 19, -14, -8, 8, -2, -3, 7, ...
## $ afstand          <dbl> 1400, 1416, 1089, 762, 719, 1065, 229, 944, 733...
## $ vliegtijd         <dbl> 227, 227, 160, 116, 150, 158, 53, 140, 138, 149...
```

## 2.1 Univariate visualisaties (1 variabele)

- Als we slechts 1 variabele bestuderen, dan zijn we voornamelijk geïnteresseerd in de spreiding van de data. Dit wordt de verdeling van de data genoemd.
- Welke vragen kunnen we beantwoorden met dit soort visualisaties?
  - Wat is de meest voorkomende waarde van de data? Dit wordt ook de modus genoemd.
  - Bezit de data 1 modus, i.e. 1 waarde die duidelijk dominant is, of meerdere modi?
    - \* Indien er slechts 1 afgetekende modus is, dan wordt de verdeling unimodaal genoemd.
    - \* Indien er meerdere modi zijn (dominante waarden), dan wordt de verdeling multimodaal genoemd.
    - \* Een multimodale verdeling kan er op wijzen dat de objecten in je data niet allemaal van hetzelfde type zijn en dat je in feiten twee populaties in je data aanwezig hebt.
  - Is de data geconcentreerd rond de modus of eerder breed verspreid. Met andere woorden, wat is de spreiding? Dit geeft inzicht in de variabiliteit van de data.
  - Is de data gelijkmataig verdeeld aan weerszijden van de modus of zien we duidelijk meer data aan één zijde van de verdeling? Indien er meer data aan één zijde van de verdeling ligt (ten opzichte van de modus) dan zegt men dat de verdeling asymetrisch verdeeld is.
  - Zijn er waardes die opmerkelijk ver van de modus verwijderd zijn en geïsoleerd zijn van andere observaties? Dit worden extreme waarden of outliers genoemd. Deze verdienen meestal extra aandacht.

### 2.1.1 Categorische variabele

#### *Staafdiagram*

- Op de X-as staan de verschillende waarden van de categorische variabele. (Fig. 2.1)
- Bij iedere waarde tekenen we een verticale balk die aangeeft hoe vaak die waarde in de dataset voorkomt.
- Minder geschikt indien er veel waarden zijn. Dan wordt de X-as snel onleesbaar. (Fig. 2.2)
- Je kan natuurlijk de labels roteren. Maar dit kan nog steeds onhandig zijn om te lezen. (Fig. 2.3).

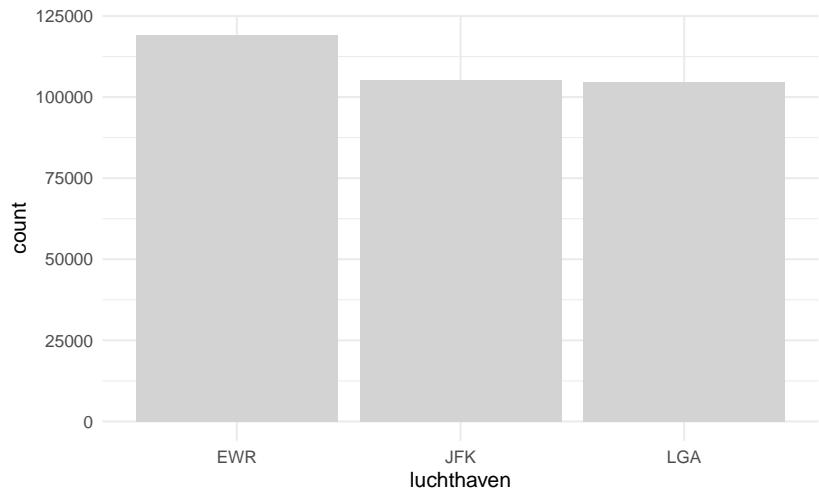


Figure 2.1: Staafdiagram luchthavens

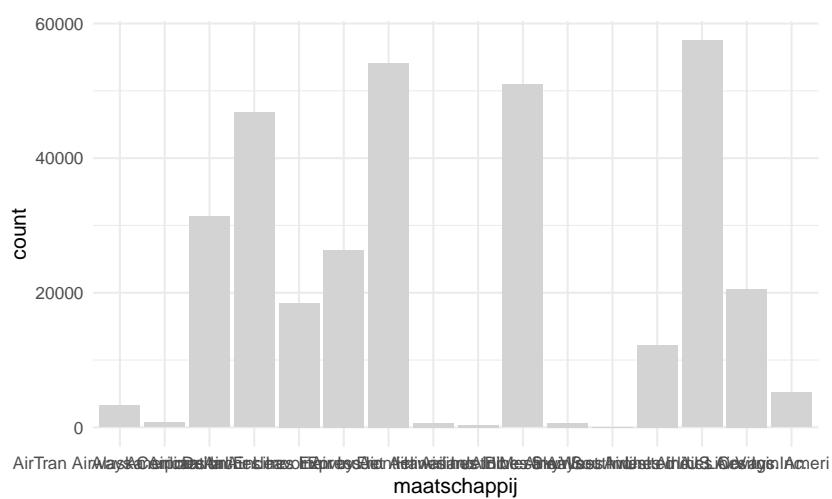


Figure 2.2: Staafdiagram maatschappijen

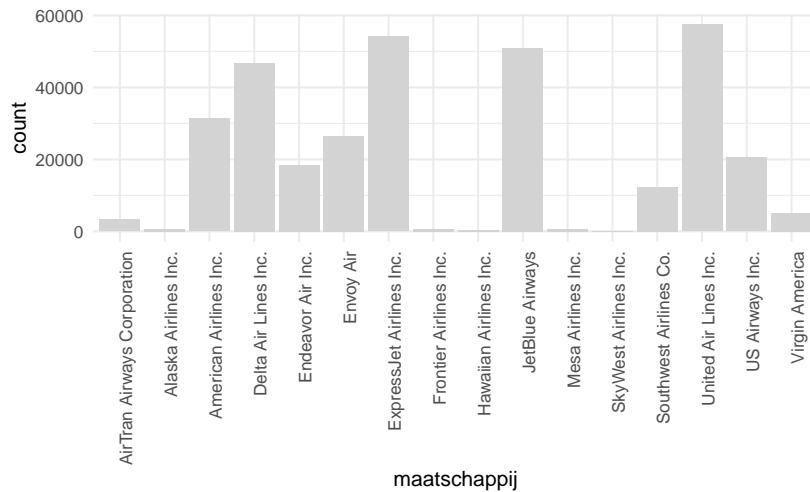


Figure 2.3: Staafdiagram met geroteerde labels

- In geval van een **nominale** variabele zijn er twee mogelijkheden om de waarden te rangschikken:
  - Alfabetisch. (standaard) Dit is handig om snel waarden terug te vinden.
  - Volgens frequentie. Dit is handig om snel te zien welke waarden vaak/weinig voorkomen en geeft ook een beter beeld van de verdeling van de waarden. (Fig. 2.4)

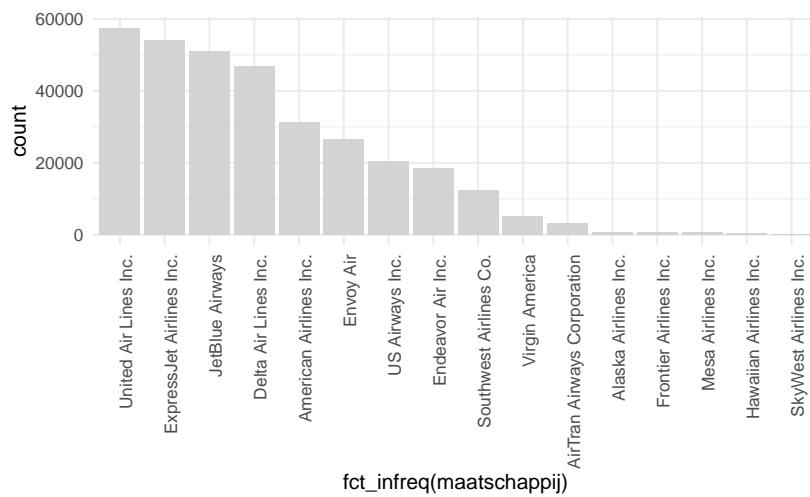


Figure 2.4: Staafdiagram gesorteerd op frequentie

- In het geval van een **ordinale** variabele houd je best de intrinsieke volgorde van de waarden aan.
- Je kan ook een horizontaal staafdiagram maken. (Fig. 2.5)

- Zelfde principe, maar dan met horizontale balken.
- Is handiger om de verschillende waarden te lezen, vooral indien dit er veel zijn.

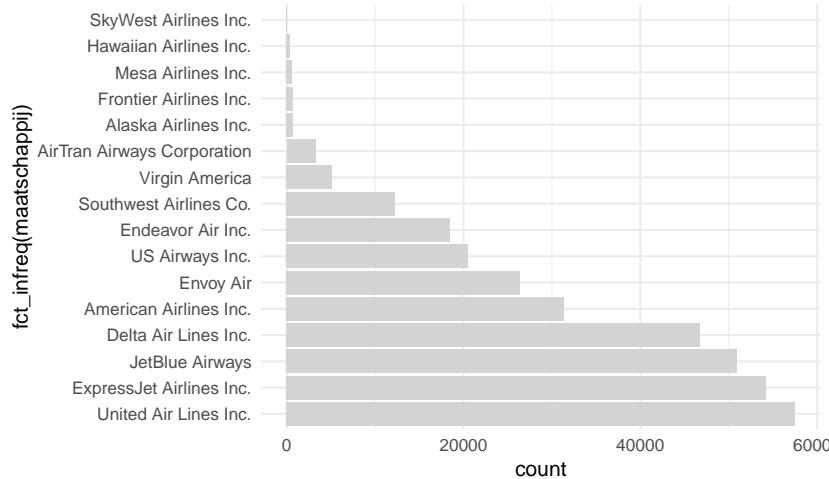


Figure 2.5: Verticaal staafdiagram gesorteerd op frequentie

### Dotplot

- In plaats van balken te gebruiken om de frequentie van een waarde aan te geven, kan je dit ook met punten doen. (Fig. 2.6)
- Een dotplot laat duidelijker zien waar de sprongen in de verdeling zit. Daarom is de dotplot vooral relevant als je de waarden ordent volgens frequentie.

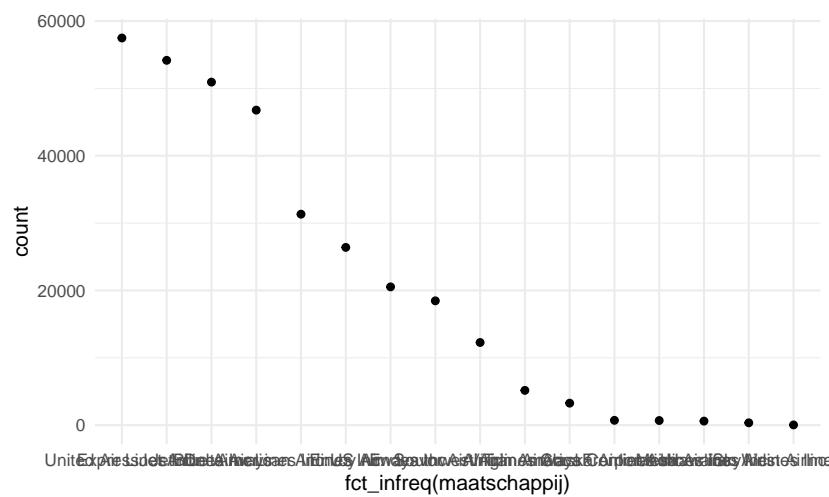


Figure 2.6: Dotplot maatschappij

- Net als de barplot kan je zowel een verticale als horizontale dotplot maken. (Fig. 2.7)

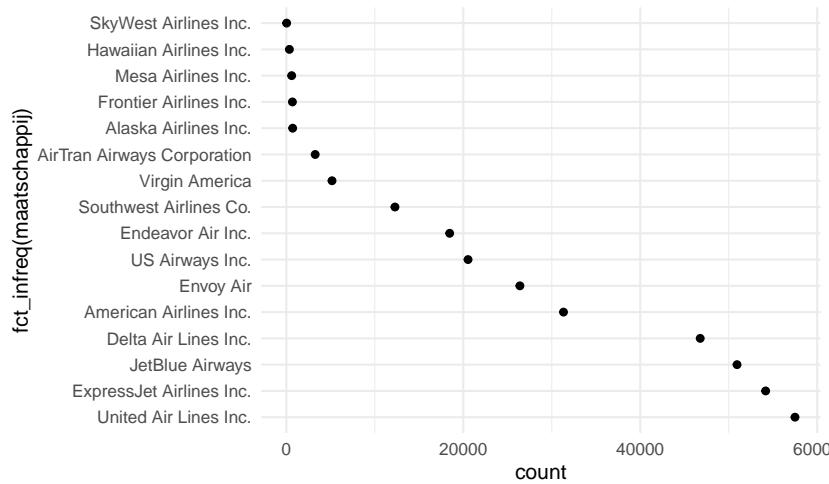


Figure 2.7: Verticale dotplot

### 'Stacked' staafdiagram

- We maken nu slechts 1 kolom. Iedere waarde is een andere kleur en neemt een deel van de balk in beslag. De volledige balk stelt 100% van de data voor. (Fig. 2.8)
- Kan nuttig zijn om data cumulatief te bestuderen.
- Hiermee kunnen we vragen beantwoorden zoals: "Welke waarden moeten we nemen om met zo weinig mogelijk waarden x% van de objecten te hebben?"
- We kunnen ook horizontale versies maken. (Fig. 2.9)
- Univariate stacked barcharts kunnen soms wat *raar* overkomen.  
Vaak komt een gewone barchart beter over.

### Andere soorten

- treemap: indelen van rechthoekige oppervlakte volgens categorische variabelen
- pie chart
  - Moeilijk te interpreteren.
  - Verschillen tussen waarden zijn enkel duidelijk bij grote verschillen, terwijl barplots en dotplots deze ook bij kleine verschillen kunnen tonen.
  - Voor cumulatieve analyses van de data zijn barplots beter omdat het hier eenvoudiger is om af te leiden waar x% zicht bevindt.

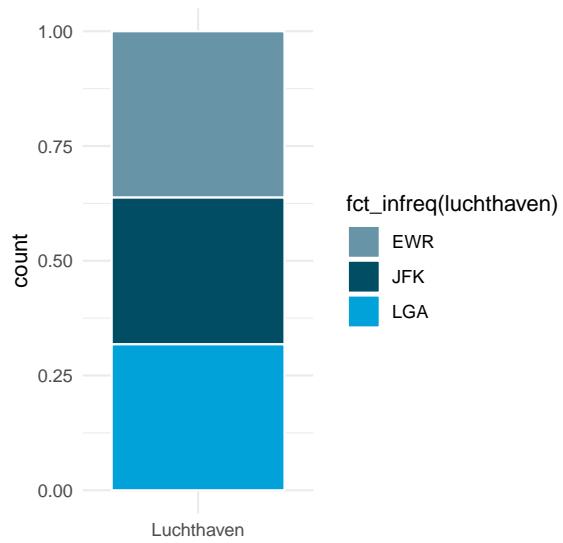


Figure 2.8: Stacked barplot

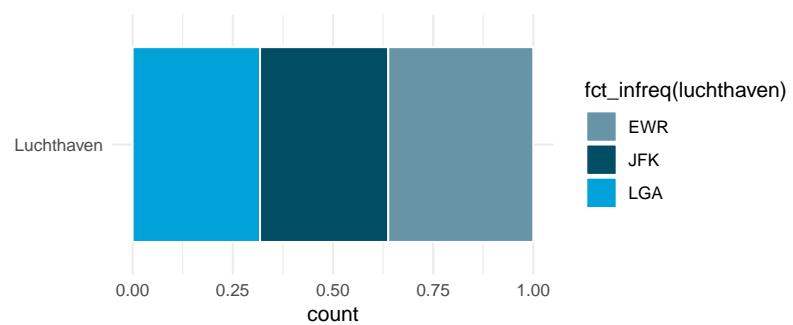


Figure 2.9: Horizontale stacked barplot

### 2.1.2 Continue variabele

#### Histogram

- Analoog met barplot, alleen gaan we hier eerst onze “categorieën” definiëren. (Fig. 2.10)
- Dit wordt ‘binning’ genoemd en wordt bepaald door een bin-breedte te kiezen.
- Je kan de binbreedte rechtstreeks kiezen of bepalen door vast te leggen hoeveel categorieën/bins je wenst.

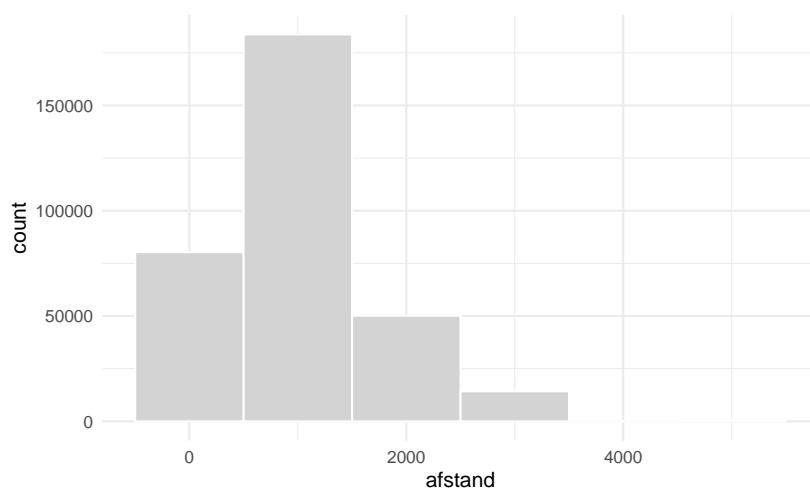


Figure 2.10: Histogram with binwidth 1000

- Voor de visualisatie, worden alle waarden gegroepeerd per ‘bin’.
- De binbreedte kan een enorme impact hebben op het uitzicht van de verdeling. (Fig. 2.11 - 2.12)
  - Hoe breder de bins, hoe minder modi je kan detecteren.
  - Hoe smaller de bins, hoe meer modi je gaat zien, hoewel dit niet altijd even betekenisvol is.
  - Hoe smaller de bins, hoe minder data er in iedere bin gaan zitten en dan kunnen patronen wel in jouw dataset bestaan maar louter ten gevolge van toeval.

#### Density

- Variant van histogram.
- In plaats van staven wordt er een curve getekend. (Fig. 2.13)
- De oppervlakte onder de curve is steeds gelijk aan 1
- Hoe hoger de curve, hoe meer observaties ter hoogte van deze waarde (hoe hoger de densiteit)

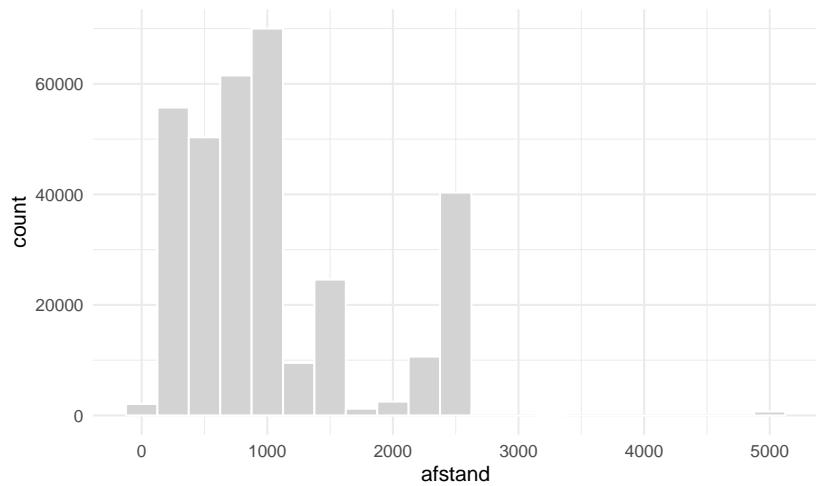


Figure 2.11: Histogram with binwidth 250

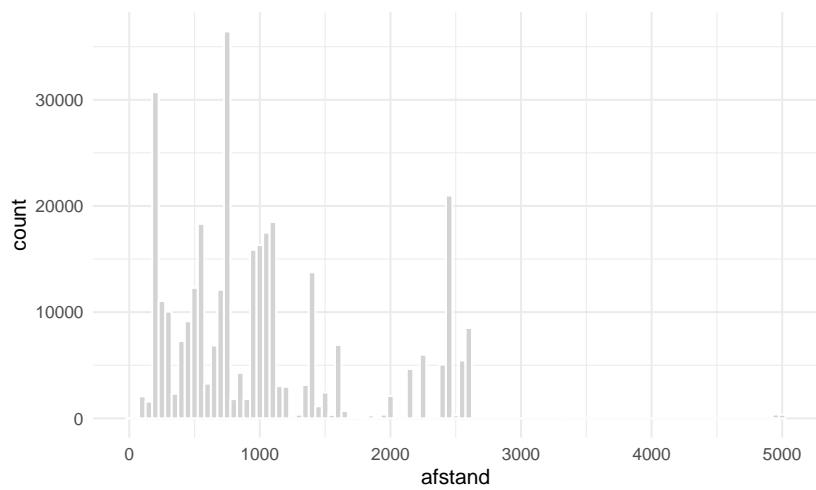


Figure 2.12: Histogram with binwidth 50

- De waarde van de y-as heeft geen directe betekenis.

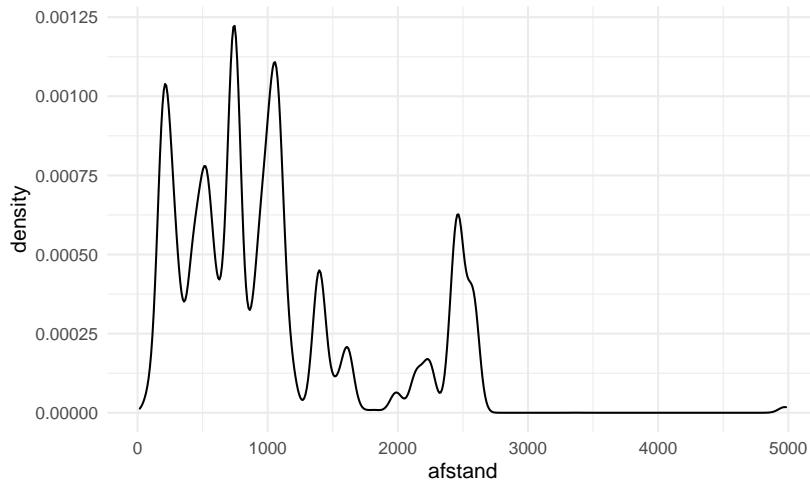


Figure 2.13: Density plot

### *Boxplot*

- De lijn in het midden duidt de mediaan aan. Dit betekent dat 50% van je data onder deze lijn ligt, terwijl 50% er boven ligt. (Fig. 2.14)
- De box in het midden duidt de middelste 50% van je data aan. Dit wordt ook de interkwartiel-box genoemd. Dit betekent dat 25% van je data onder deze box zit en nog eens 25% boven deze box ligt. Hoe groter de box, des te meer de data gespreid is.
- Indien de box aan één zijde van de mediaanlijn groter is dan aan de andere zijde, dan wijst dit er op dat de data meer gespreid is aan die kant.
- De “whiskers” geven de laatste datapunten aan die als “normaal” beschouwd worden. Datapunten buiten deze grenzen beschouwt een boxplot als outliers of extreme waarden.
- De grens waar data van normaal naar extreem overgaat wordt door de boxplot bepaald door anderhalf keer de grootte van de interkwartiel-box op te tellen (en af te trekken) van de bovenste (onderste) grens van de interkwartiel-box. Punten die hier buiten liggen zijn outliers en worden als aparte punten aangeduid. De uitersten van de whiskers duiden de laatste datapunten aan binnen deze grenzen.
- Het is niet abnormaal dat er outliers in je data aanwezig zijn.
- Bij normaal verdeelde data zal je gemiddeld 7 outliers per 1000 datapunten mogen verwachten.

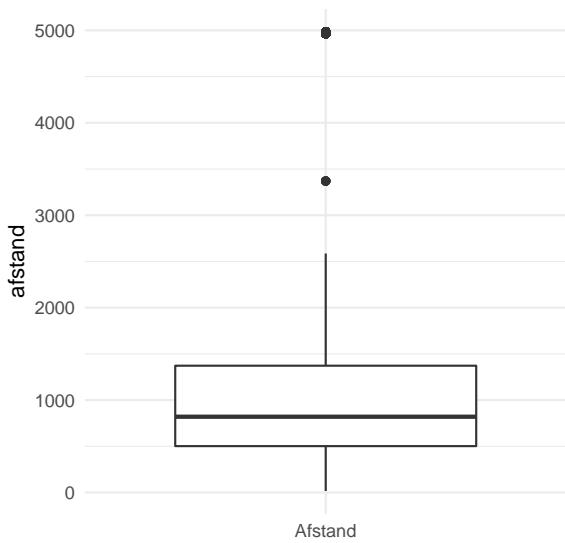


Figure 2.14: Verticale boxplot  
vertrekvertraging

- Een normale verdeling is een bepaalde manier waarop data waar-den verdeeld kunnen zijn die in de realiteit vaak voorkomt.
- Indien je echter veel meer outliers ziet op je boxplot visualisatie, dan is de kans reëel dat er meer aan de hand is:
  - Er zijn bijvoorbeeld systematische meetfouten
  - De objecten in je data zijn in feite op bepaalde aspecten signifi-cant verschillend waardoor je ze apart moet bestuderen.
- Je kan een boxplot ook roteren. (Fig 2.15)
- Boxplots komen beter tot hun recht bij bivariate analyses dan bij univariate analyses.

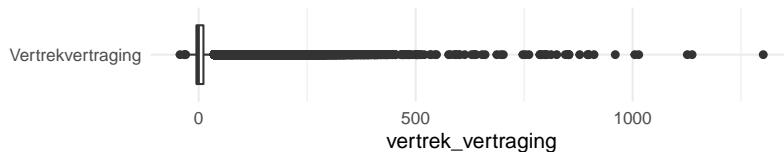


Figure 2.15: Horizontale boxplot  
vertrekvertraging

### Violin plot

- Een violin plot kan je beschouwen als een combinatie van een his-togram en een boxplot. (Fig. @ref(fig:2\_10a))
- Net als bij een boxplot wordt op verticale wijze getoond hoe de data verspreid is.
- Opnieuw kan je ervoor kiezen de grafiek te roteren. (Fig. @ref(fig:2\_10b))

- Net als bij een histogram kan je goed zien waar het volume (de massa) van de data zich bevindt.
- Net als bij een histogram kan je detecteren hoeveel modi de data bezit.
- In tegenstelling tot de boxplot, kan je bij een violinplot wel niet duidelijk zien waar bijvoorbeeld het ‘midden’ van je data is.

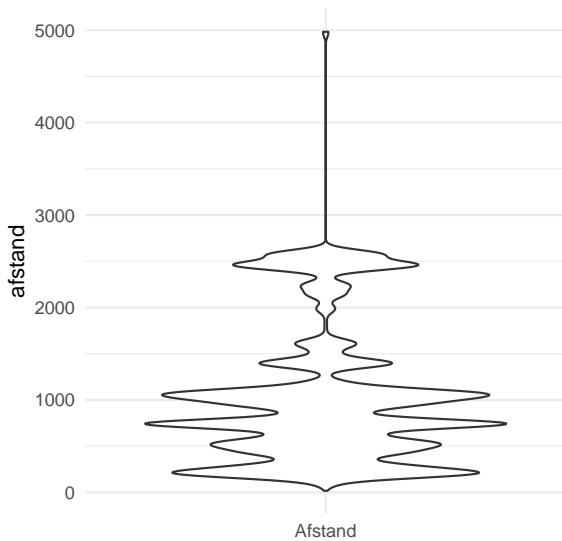


Figure 2.16: Verticale violin plot afstand

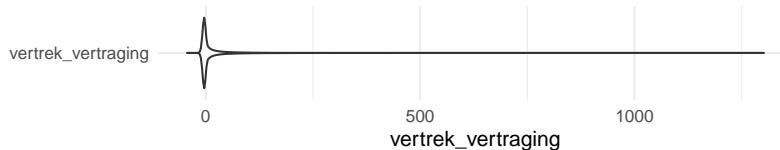


Figure 2.17: Horizontale violin plot vertrekvertraging

### Jitter plot

- puntenwolk waarbij willekeurige “noise” (ruis) wordt toegevoegd.
- de ruis zorgt ervoor dat datapunten niet overlappen, en dat het duidelijk is waar de massa zich bevindt.
- Fig. 2.18 toont een vergelijking van violin, boxplot, point en jitter plot.

## 2.2 Bivariate visualisatie (2 variabelen)

- Wanneer we de relatie tussen 2 variabelen bekijken is het eenvoudig te denken in *oorzaak* en gevolg *termen*.<sup>1</sup>

<sup>1</sup> Zie opmerking i.v.m. correlatie versus causaliteit, 2.5.

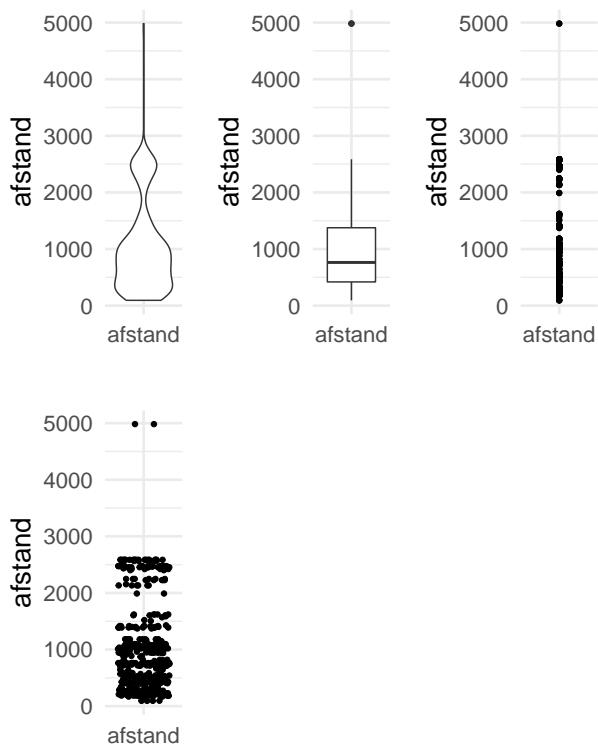


Figure 2.18: Violin, boxplot, point en jitter

- De variabele die we het label “oorzaak” geven, zullen we voortaan “onafhankelijke variabele” noemen.
- De variabele die we het label “gevolg” geven, zullen we voortaan “afhankelijke variabele” noemen.
- Waar we eigenlijk in geïnteresseerd zijn bij een visualisatie van 2 variabelen is de impact van de onafhankelijke variabele op de afhankelijke variabele weer te geven.
- Alle vragen die we kunnen stellen bij de visualisatie van één variabele, kunnen we nog steeds stellen, met telkens de bijkomende vraag of het waargenomen patroon verandert als de onafhankelijke variabele van waarde verandert.

### 2.2.1 Situatie 1: De onafhankelijke variabele is categorisch

Indien de afhankelijke variabele een continue variabele is kan je: \* meerdere boxplots op 1 grafiek visualiseren, met telkens 1 boxplot per waarde van de onafhankelijke variabele. (Fig. 2.19)

- meerdere violinplots op 1 grafiek tonen, met telkens 1 violinplot per waarde van de onafhankelijke variabele. (Fig. 2.20)

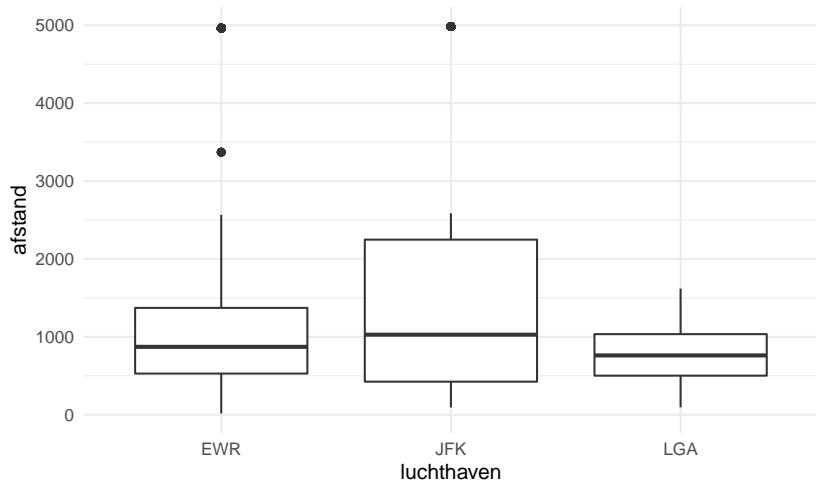


Figure 2.19: Bivariate boxplot

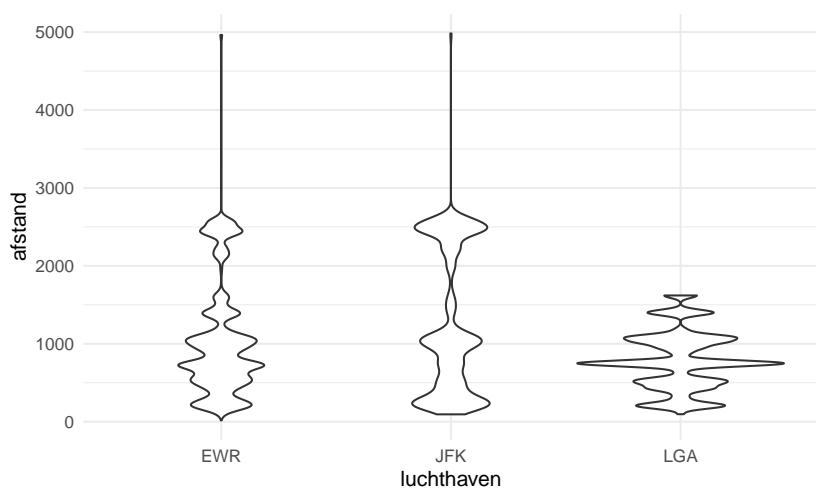


Figure 2.20: Bivariate violin plot

- meerdere histogrammen op 1 grafiek tonen
  - Hiervoor gebruiken we facetten: we tekenen voor elke waarde van de onafhankelijke variabele een apart assenstelsel. (Fig. 2.21)

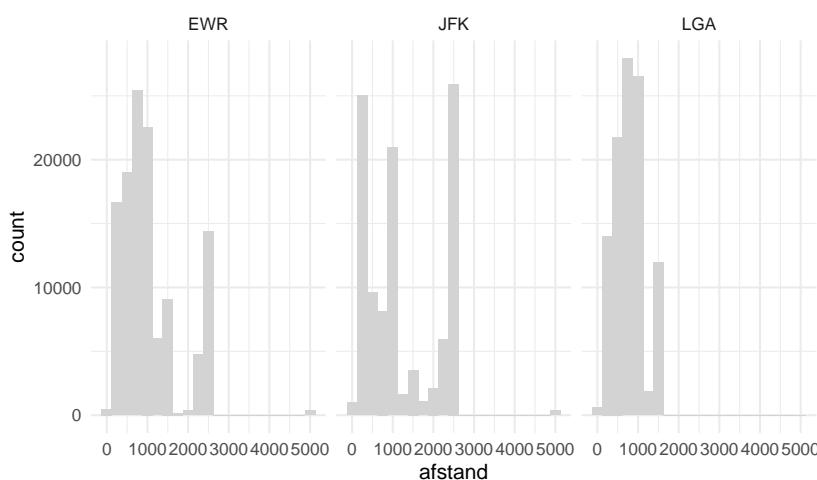


Figure 2.21: Bivariate histogram plot

- meerdere density plots
- Hiervoor kunnen we facetten gebruiken, ofwel de density plots over elkaar tekenen en onderscheiden met kleur. (Fig. 2.22-2.23)

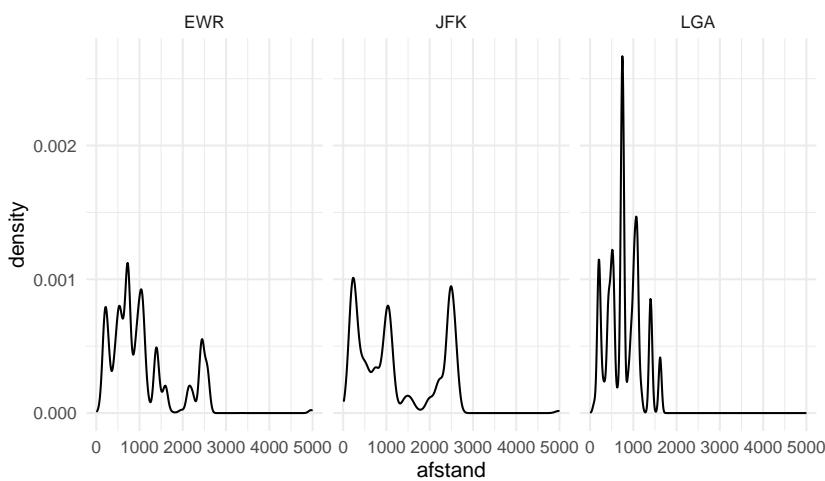


Figure 2.22: Bivariate density plot - apart

Indien de afhankelijke variabele een categorische variabele is:

- Kan je meerdere barplots op 1 grafiek visualiseren, met telkens de bars gegroepeerd per waarde van de onafhankelijke variabele.

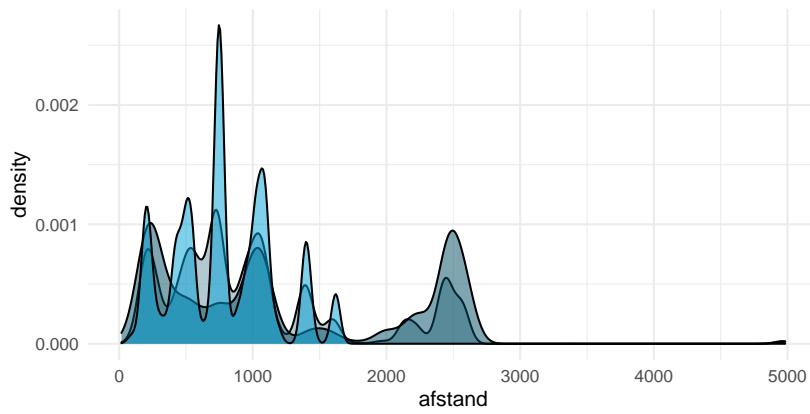


Figure 2.23: Bivariate density plot - overlappend

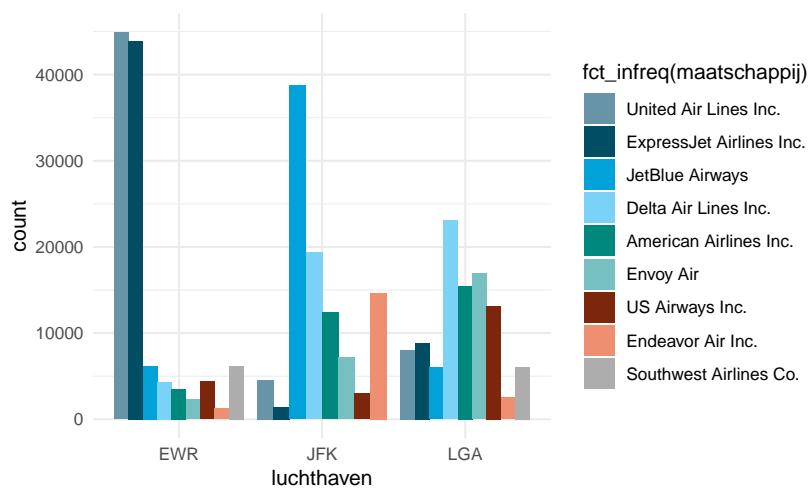
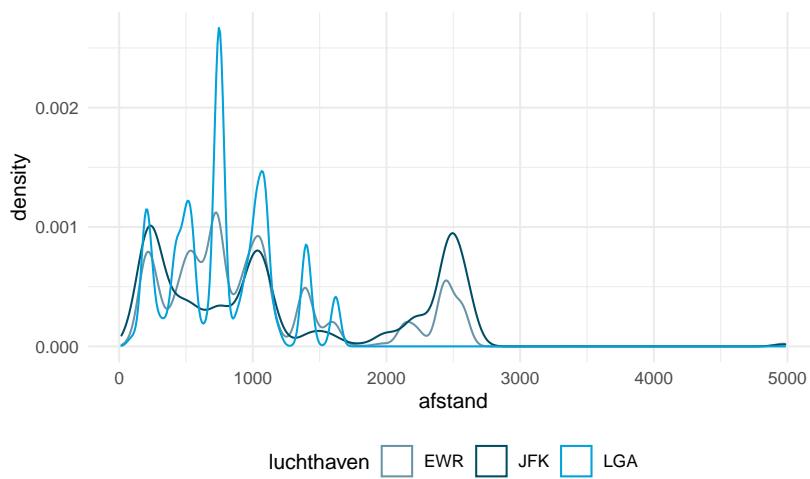


Figure 2.24: Bivariate barplot

- Kan je meerdere stacked barplots op 1 grafiek plaatsen, met telkens een volledige stack per waarde van de onafhankelijke variabele.

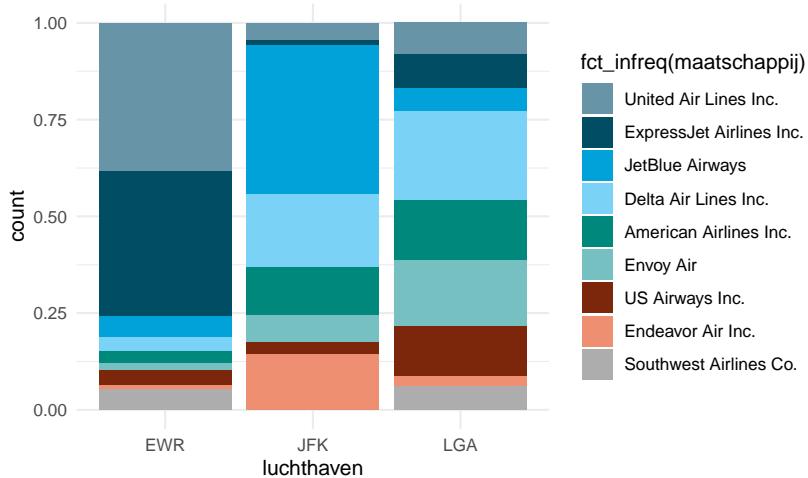
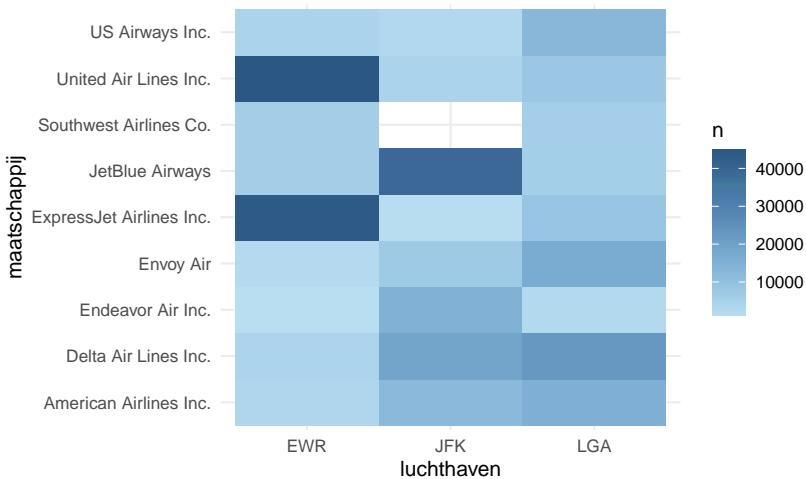


Figure 2.25: Bivariate stacked barplot

- Kan je een heatmap (of tile plot) gebruiken. Hier bij plaats je 2 categorische variabelen op de x-as en y-as, respectievelijk.
  - Voor elke combinatie van waarden is er een tegel die je kan inkleuren volgens de frequentie van de combinatie.



- Je kan bijkomende ook de exacte waarde in elke tegel plotten.



**Let op** wanneer beide variabelen categorisch zijn, is het nog steeds van belang welke je beschouwd als afhankelijke en welke als onafhankelijke. Technisch kan je ze omdraaien, maar de betekenis van je visualisatie is niet dezelfde!

Andere mogelijkheden:

- treemap
- mosaic plot

### 2.2.2 Situatie 2: De onafhankelijke variabele is continu

In dit geval kan je geen aparte plot per mogelijke waarde van de onafhankelijke variabele maken omdat er mogelijk oneindig veel waarden zijn.

Indien de afhankelijke variabele continu is, dan kan je een scatterplot maken.

- Iedere observatie is een punt in je grafiek, waarbij de x-waarde op de grafiek overeenkomt met de waarde van de onafhankelijke variabele en de y-waarde op de grafiek overeenkomt met de waarde van de afhankelijke variabele.
- Om patronen beter te herkennen kan je een “trend-lijn” toevoegen.
- Bij scatterplots is er gevaar voor overplotting
- Mogelijke oplossingen
  - 2D histogram: verdeel veld op in vierkante bins en tel per bin hoeveel data punten er zijn

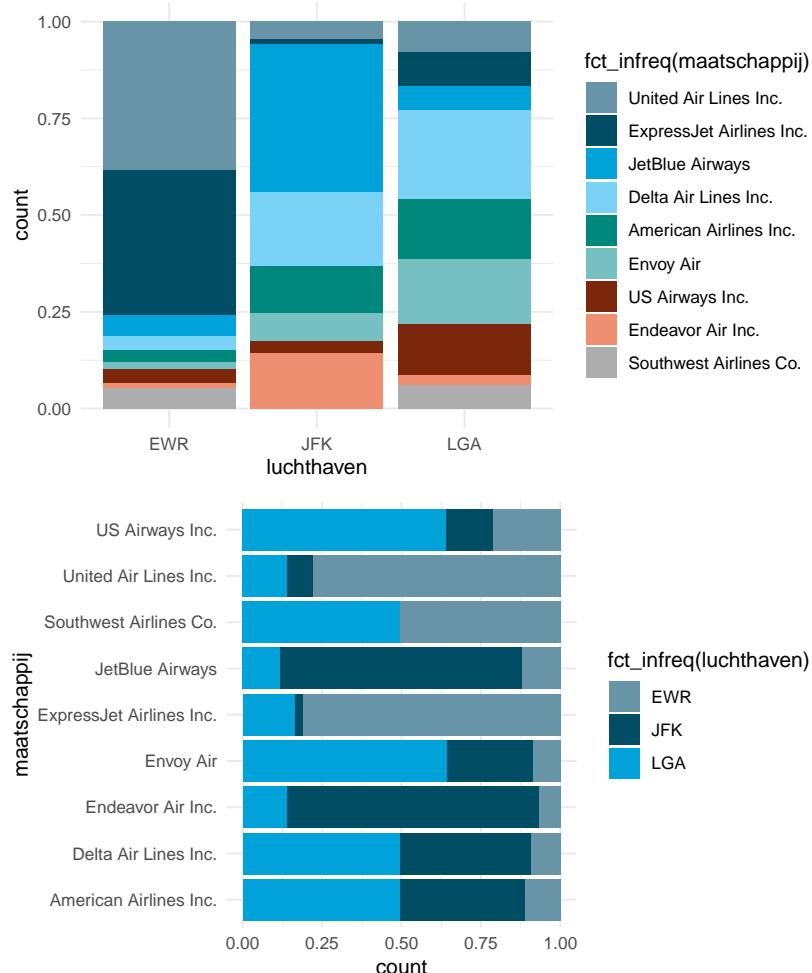


Figure 2.26: Twee verschillende stacked barcharts van luchthaven en maatschappij.

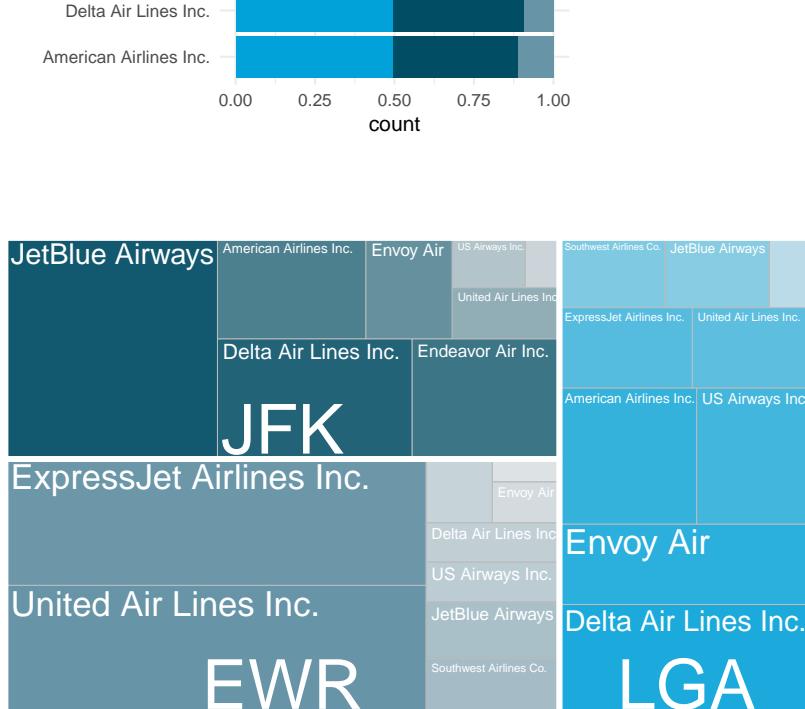


Figure 2.27: Treemap luchthaven en maatschappij.



Figure 2.28: Mosaic plot luchthaven en maatschappij.

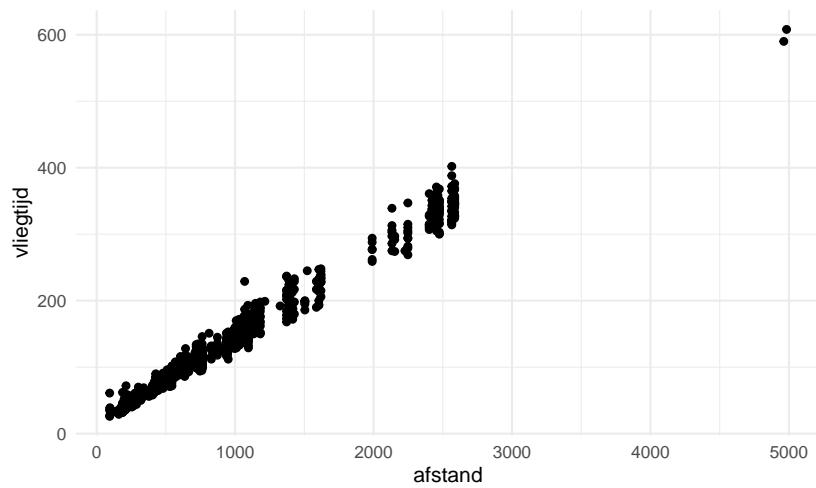


Figure 2.29: Scatterplot

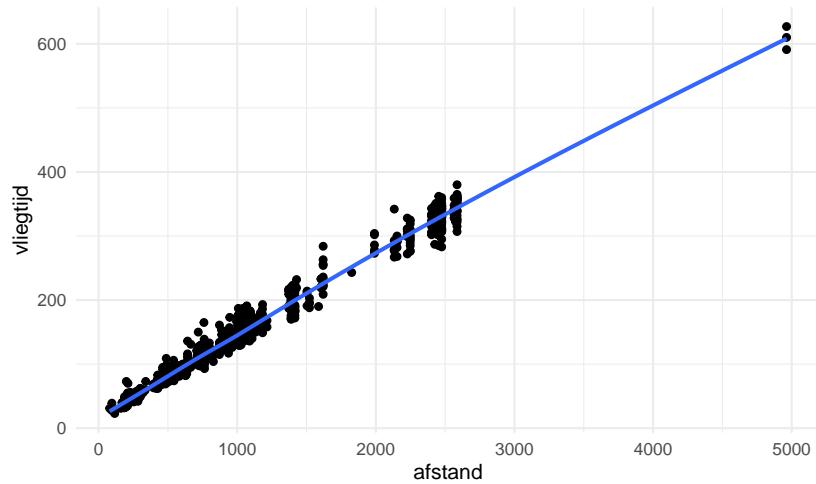


Figure 2.30: Scatterplot met trendlijn

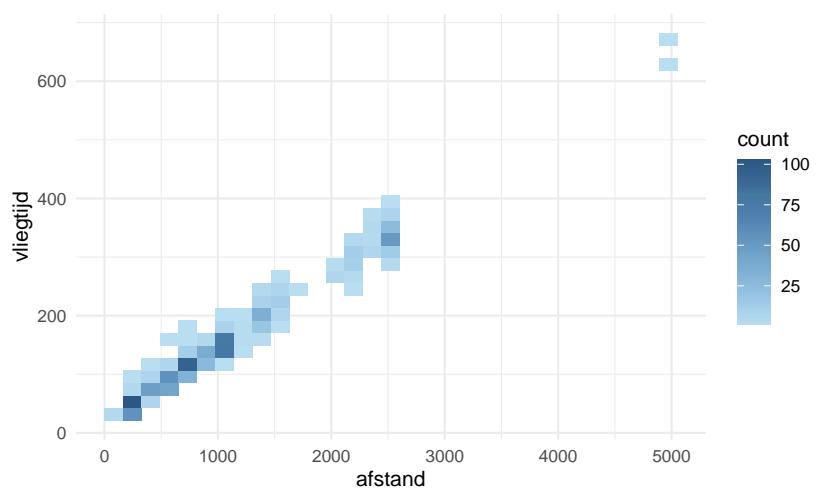


Figure 2.31: Scatterplot met trendlijn

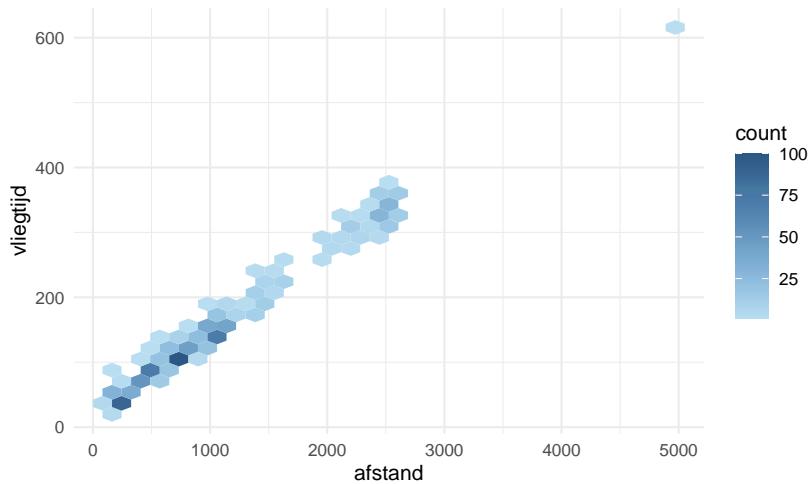


Figure 2.32: Hexplot met trendlijn

- Hexplot: analoog, maar gebruik zeshoekige bins ipv vierkanten. Voordeel: punten binnen elke zeshoek liggen dichter bij het middelpunt van de bin.

Indien de afhankelijke variabele categorisch is, dan kan je niet rechtstreeks een betekenisvolle plot maken omdat er waarschijnlijk te weinig datapunten zijn voor iedere mogelijke waarde van de onafhankelijke variabele.

- Wat je dan best kan doen, is de onafhankelijke continue variabele categorisch maken door deze in te delen in bins/intervallen. En dan ben je terug in de situatie waarbij de onafhankelijke variabele categorisch is. We komen hierop terug in het hoofdstuk over Data Voorbereiding.

### 2.2.3 Situatie 3: De onafhankelijke variabele is tijd

- Tijd kunnen we zien als continue variabele
  - Bijgevolg zelfde grafieken mogelijk als wanneer onafhankelijke variabele continu is
    - \* Tijd + continu afhankelijk -> scatterplot, 2D histograms, hex bins
    - \* Tijd + categorisch afhankelijk -> probleem: tijd categoriseren (zie verder).
- Wanneer we één enkele variabele voorstellen doorheen de tijd is er per tijdseenheid maar 1 data punt. Hieronder wordt de gemiddelde vertrekvertraging per dag getoond.

```
## `summarise()` ungrouping output (override with '.groups' argument)
```

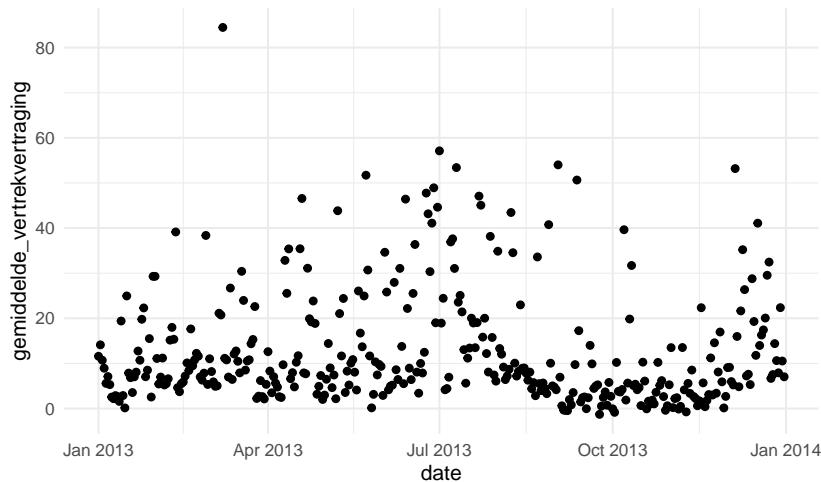


Figure 2.33: Puntenwolk met tijd op x-as

In dat geval is het beter om in plaats van punten een lijngrafiek te gebruiken.

```
## `summarise()` ungrouping output (override with `groups` argument)
```

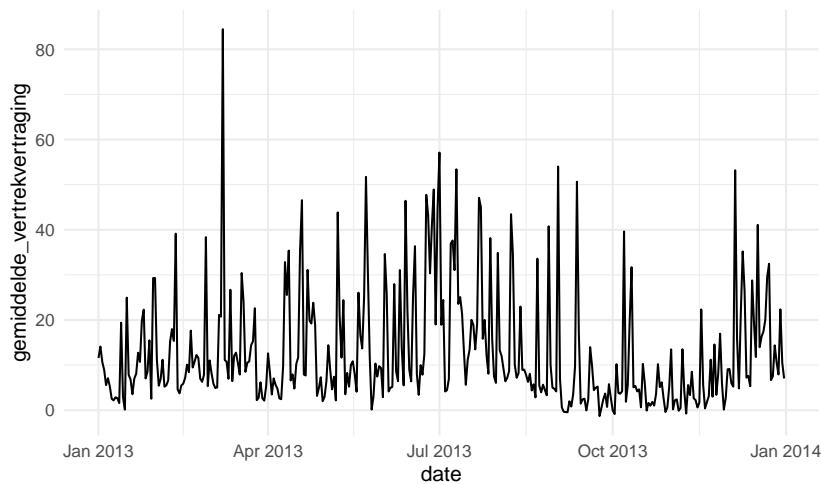


Figure 2.34: Lijngrafiek

Indien je een beperkt aantal punten hebt (hieronder bijvoorbeeld één maand van de vluchtgegevens) kan je ervoor kiezen om zowel punten als lijnen te tonen. Op die manier is het makkelijker individuele data punten af te lezen.

```
## `summarise()` ungrouping output (override with `groups` argument)
```

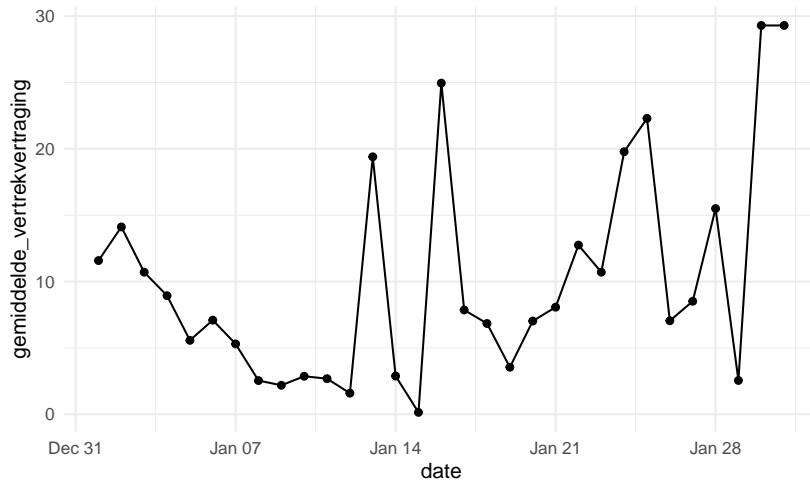


Figure 2.35: Lijn grafiek met punten

Indien we veel datapunten hebben, wat hier het geval is, kan een lijngrafiek zeer chaotisch worden. We kunnen daarom ervoor kiezen om onze tijd in te delen in categoriën. Bijvoorbeeld, in plaats van de dagelijkse gemiddelde vertrekvertraging, kunnen we de gemiddelde vertrekvertraging per maand berekenen en tonen.

```
## `summarise()` ungrouping output (override with `groups` argument)
```

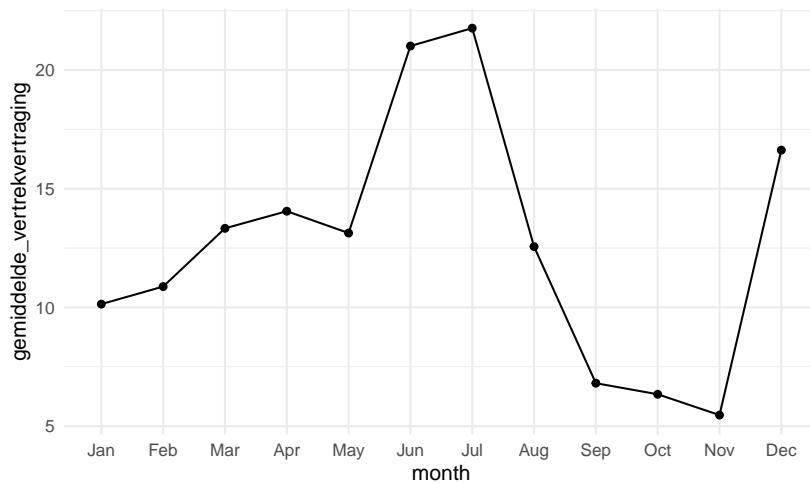


Figure 2.36: Lijngrafiek van gemiddelde vertrekvertraging per maand.

- Op dit moment verliezen we daardoor wel veel informatie. Maar we kunnen dit nu ook beschouwen als een visualisatie van een categorische variabele (maand) t.o.v. een continue. Waardoor we de technieken voor dit type bivariate visualizaties kunnen toepassen.

Bijvoorbeeld boxplots. We zien nu zowel de algemene trend als outliers. In februari was er bijvoorbeeld een dag waar de gemiddelde vertraging ver boven de normale trend lag.

```
## 'summarise()' regrouping output by 'date' (override with '.groups' argument)
```

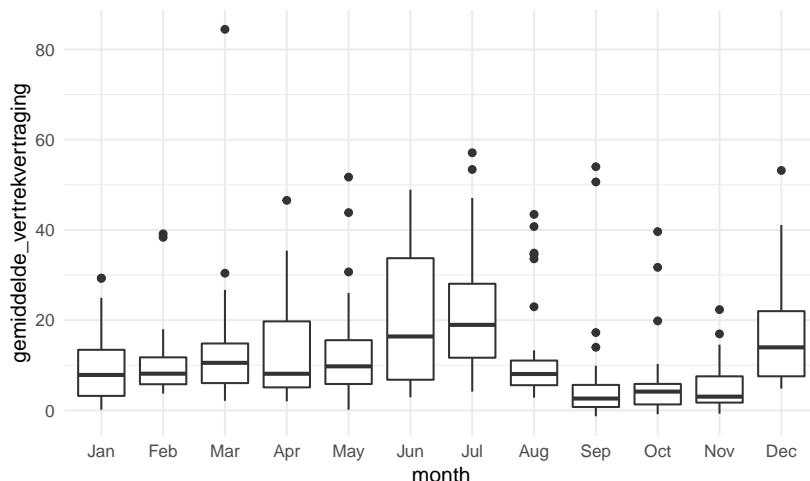


Figure 2.37: Boxplots van gemiddelde dagelijks vertrekvertraging voor elke maand.

- Wanneer we de tijd gecategoriseerd hebben kunnen we ook categorische variabelen weergeven als afhankelijke. Bijvoorbeeld, zijn er verschillen in het aantal vluchten per maatschappij doorheen de tijd. We kunnen hier dezelfde types grafieken als voor bivariate cat+cat visualisaties gebruiken, bijvoorbeeld stacked barcharts.

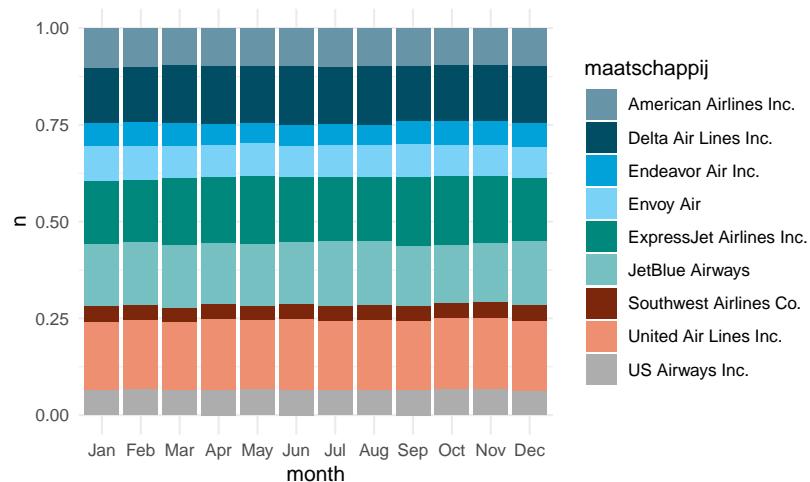


Figure 2.38: Verdeling van aantal vluchten over maatschappijen per maand.

- We kunnen categorizeren op maand, jaar, etc. Maar ook op tijdspecifieker kenmerken, zoals bijvoorbeeld de dag van de week

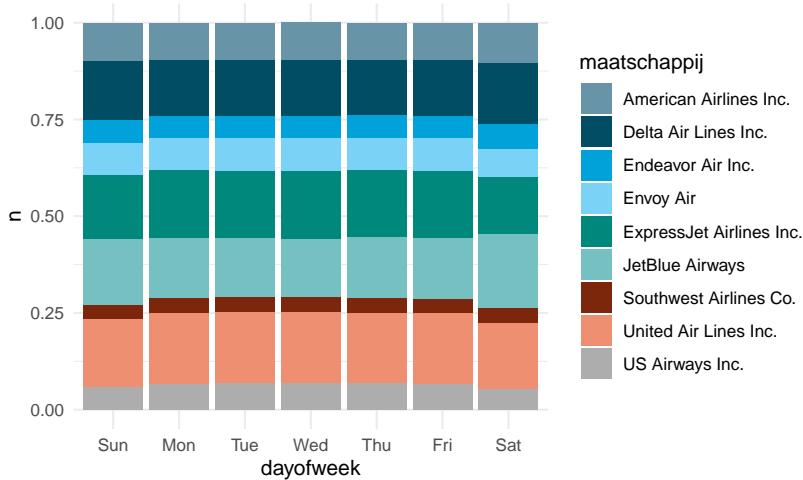


Figure 2.39: Verdeling van aantal vluchten over maatschappijen per dag van de week.

- Of het uur van de dag

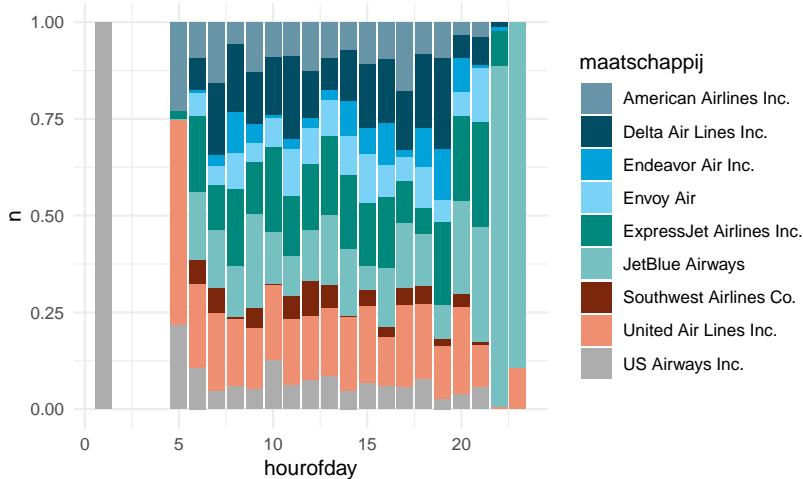


Figure 2.40: Verdeling van aantal vluchten over maatschappijen per vertrekuur.

### 2.3 Multivariate visualisaties (meer dan 2 variabelen)

- Datavisualisatie van patronen tussen meer dan 2 variabelen worden snel te complex om te interpreteren.
- Het basisprincipe is wel eenvoudig.

- Je hebt typisch 1 afhankelijke variabele (Y) en een aantal onafhankelijke variabelen (A, B, ...).
- Je visualiseert eerst Y en A (bivariaat)
- Je voegt dan de volgende variabelen (B, c, ...) stap voor stap toe aan de grafiek.
  - \* Door de bivariate grafiek te herhalen in verschillende facetten (een voor elke waarde van B).
  - \* Door verschillende kleuren te gebruiken voor elke waarde van B
- Bij multivariate visualisaties zijn er afhankelijk van de data types oneindig veel mogelijke grafieken die je kan maken.
  - Het is vaak afhankelijk van de data welke grafiek het “best past”
  - Enkel wanneer de onafhankelijk variabele continu is zijn de keuzes beperkt en ben je vaak genoodzaakt om deze om te zetten naar categoriëën.

### 2.3.1 Voorbeeld: In welke mate hangt de vertrek vertraging af van de luchthaven en de afstand?

Stap 1. Vertraging vs. afstand

- Beide continue: scatterplot

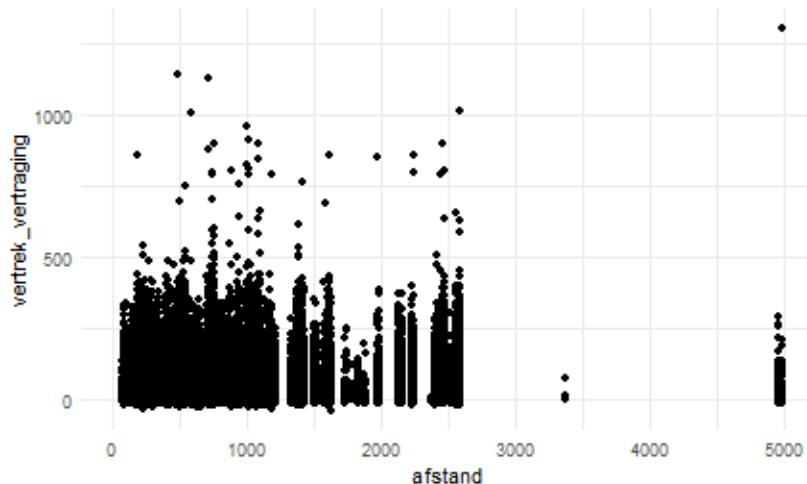


Figure 2.41: Vertrekvertraging vs afstand

Stap 2. Voeg invloed van luchthaven toe.

- Optie 1: gebruik kleur om de verschillende luchthavens te differenteren. Een trendlijn kan hier helpen.

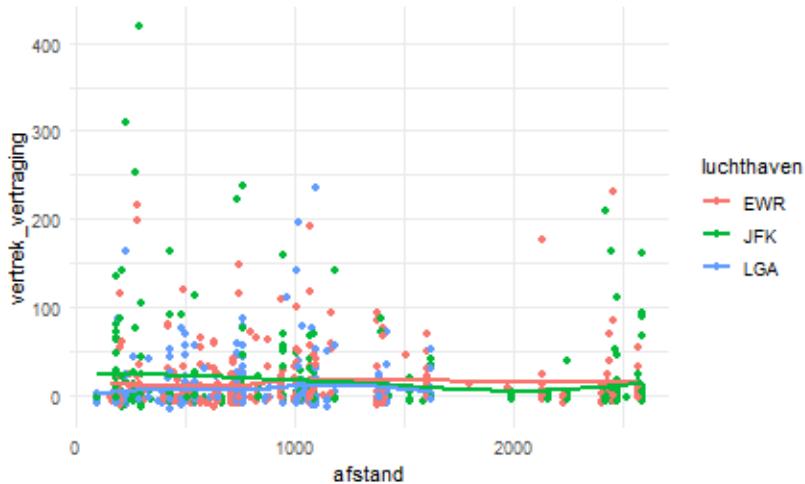


Figure 2.42: Vertrekvertraging vs afstand en luchthaven

- Geen geweldig resultaat in dit geval.
- Optie 2: Gebruik facetten voor de verschillende luchthavens.
- Optie 3: Facets, maar gebruik hex bins

### 2.3.2 Voorbeeld: multivariaat tijd

*Situatie 1: Variabelen hebben dezelfde eenheid.*

Voorbeeld: vertrekvertraging en aankomstvertraging. Je kan lijngrafieken tekenen met meerdere lijnen op hetzelfde assenstelsel.

```
## `summarise()` regrouping output by 'date' (override with '.groups' argument)
```

- Of je kan er voor kiezen elke lijn in een afzonderlijk paneel te tonen

```
## `summarise()` regrouping output by 'date' (override with '.groups' argument)
```

*Situatie 2: Variabelen hebben niet dezelfde eenheid*

Voorbeeld: de gemiddelde levensverwachting en gdp per capita doorheen de tijd. In dit geval ben je genoodzaakt 2 panelen te gebruiken.

```
## `summarise()` regrouping output by 'year' (override with '.groups' argument)
```

Optie 2: Maak een connected scatterplot. Toon een punt voor elke meting, waarbij x en y elk een variabele voorstellen. Verbindt dat elk punt in chronologische volgorde.

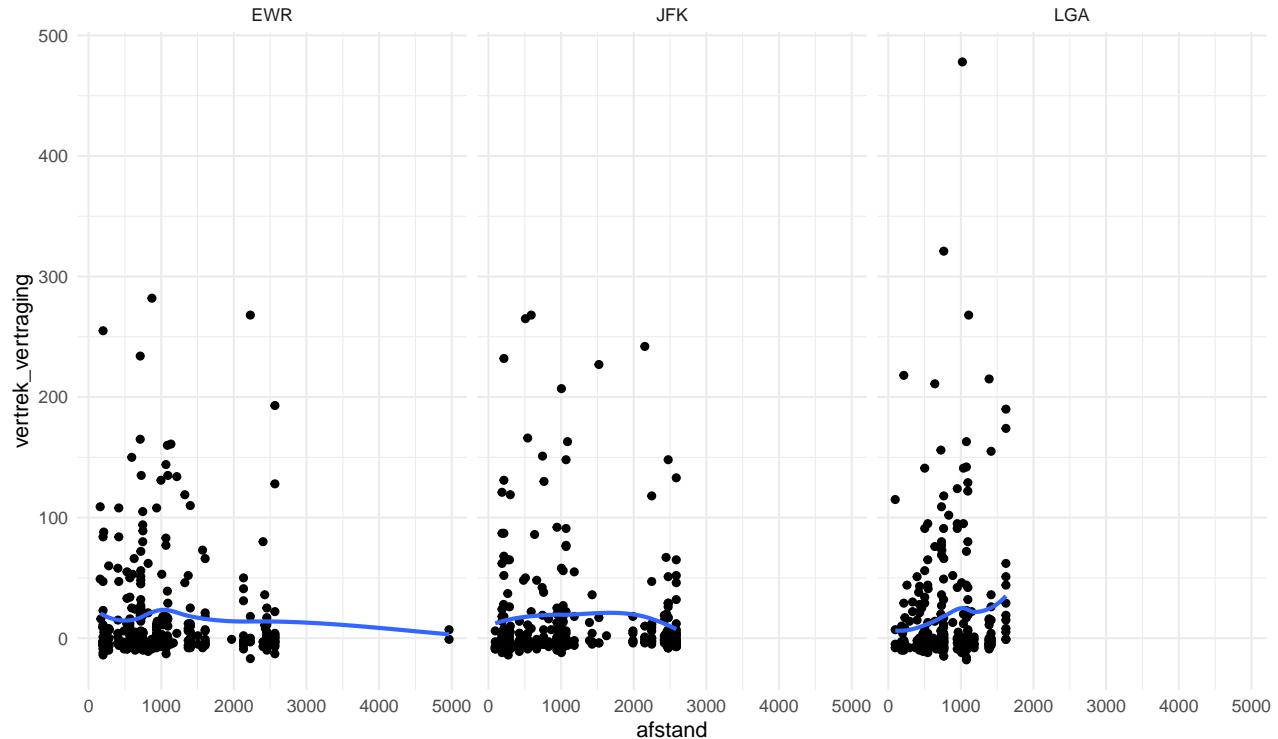


Figure 2.43: Vertrekvertraging vs afstand en luchthaven

```
## `summarise()` regrouping output by 'year' (override with '.groups' argument)
```

Variant, per continent:

```
## `summarise()` regrouping output by 'year', 'key' (override with '.groups' argument)
```

## 2.4 Visualisaties voor communicatie

Wanneer uiteindelijk beslist om een visualisatie te gebruiken om te communiceren, zorg ervoor dat

- de grafiek leesbaar is
- je kleur enkel gebruikt waar nodig.
- je correcte as-labels gebruikt
- je geen thema gebruikt dat te druk/overheersend is
- je een gepaste titel voorziet.

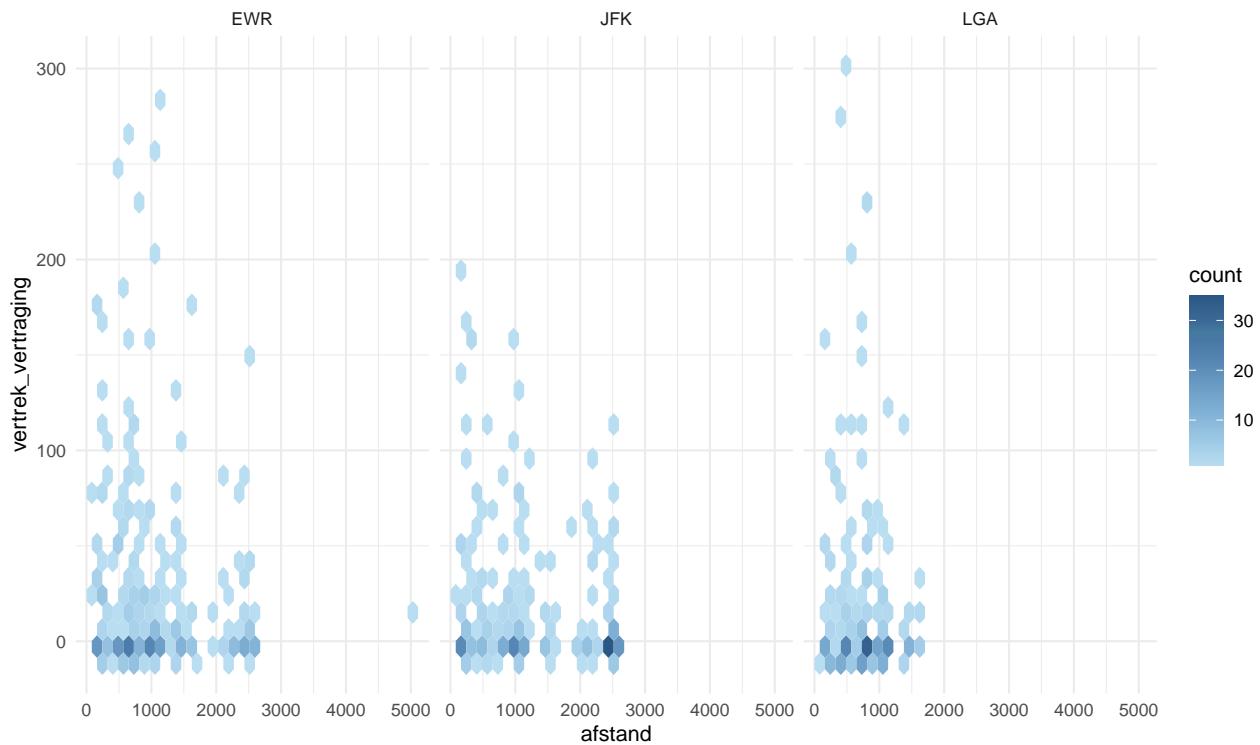


Figure 2.44: Vertrekvertraging vs afstand en luchthaven, hexbins

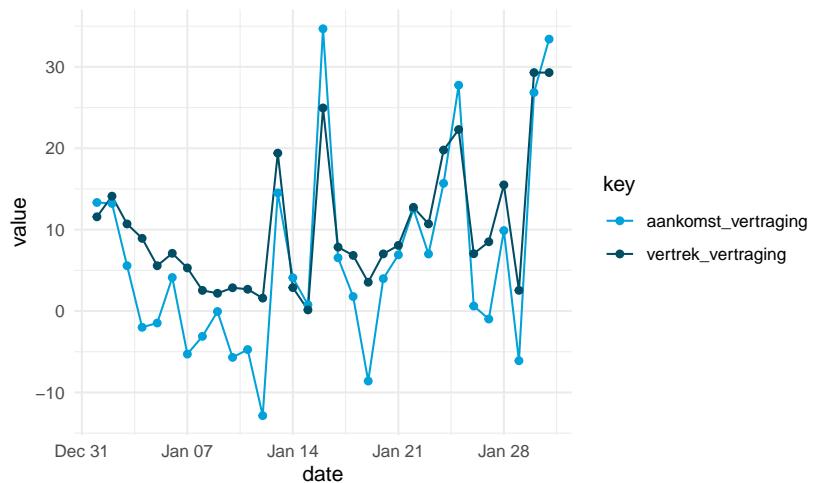


Figure 2.45: Evolutie van 2 variabelen over tijd in één grafiek (zelfde meeteenheid)

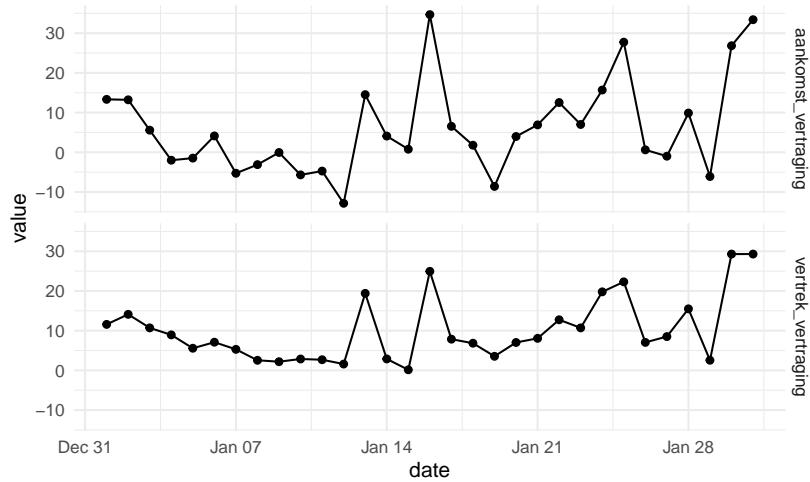


Figure 2.46: Evolutie van 2 variabelen over tijd in afzondelijke panels.

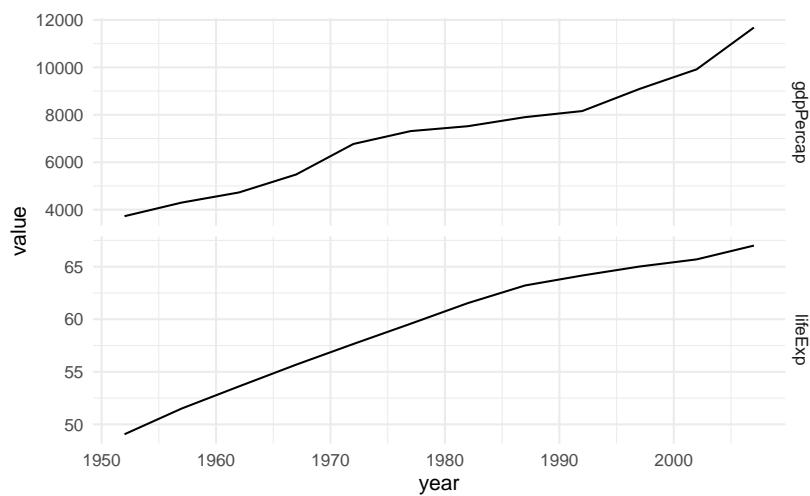


Figure 2.47: Evolutie van 2 variabelen met andere eenheden in afzonderlijke panels.

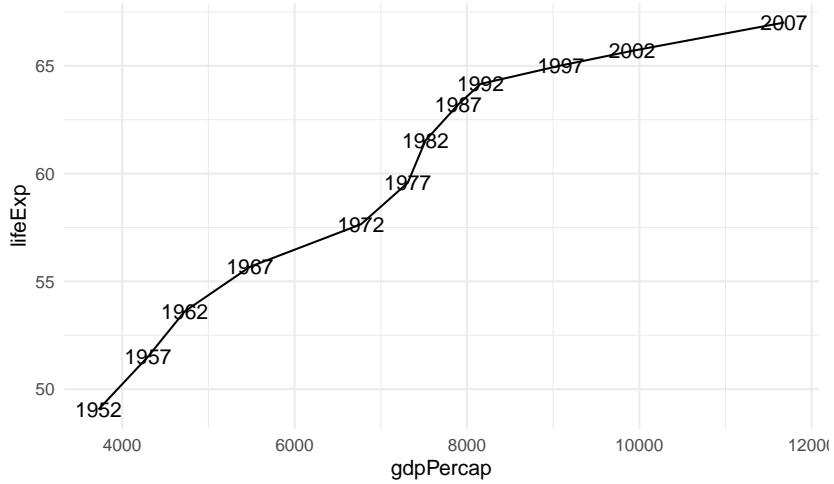


Figure 2.48: Evolutie van 2 variabelen (levensverwachting en inkomen per capita) aan de hand van connected scatterplot.

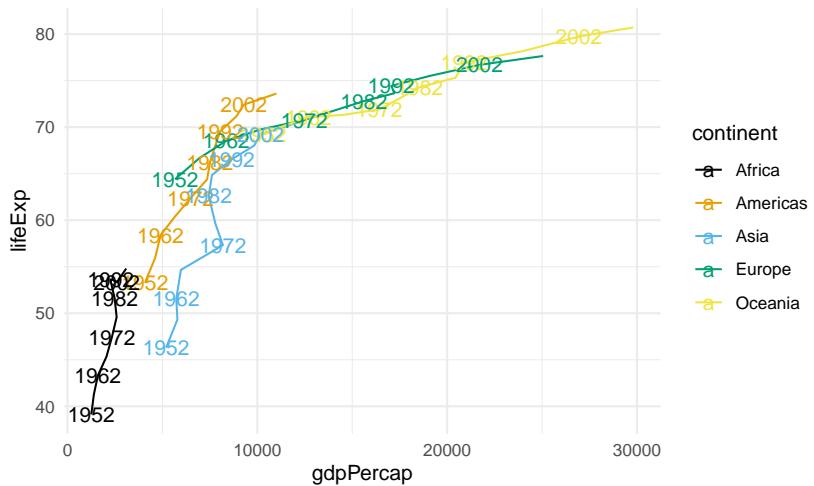
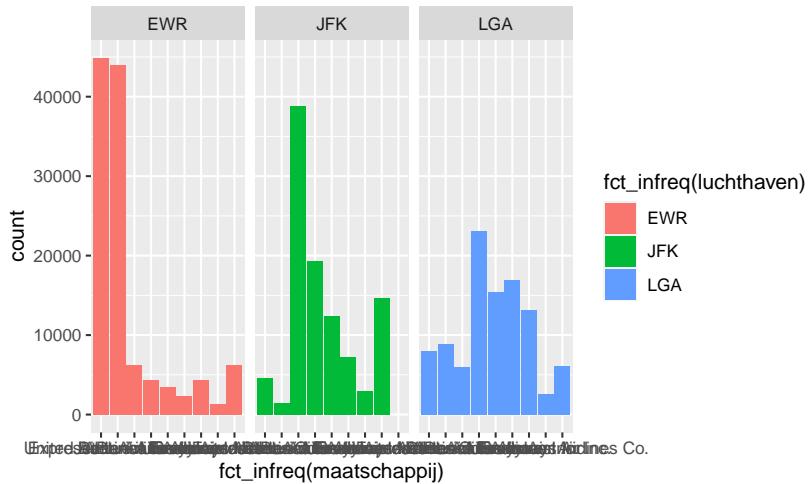
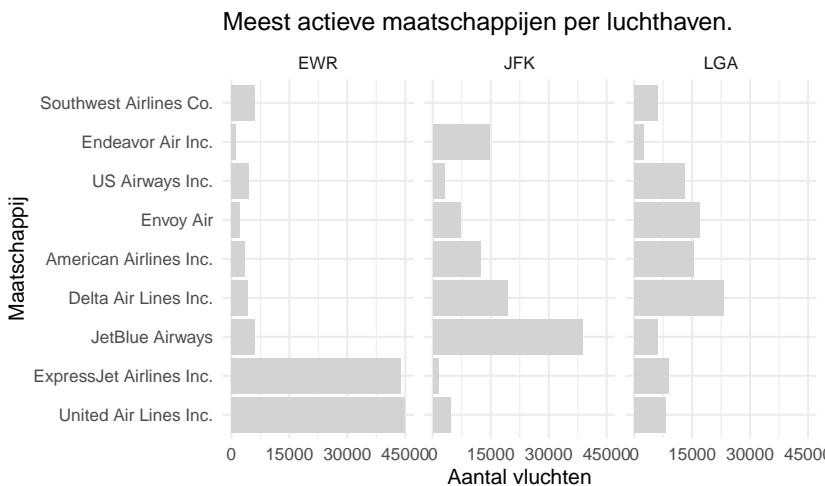


Figure 2.49: Evolutie van 2 variabelen aan de hand van connected scatterplot - verschillende groepen.

#### 2.4.1 Voorbeeld: voor ~ goed voor exploratie



#### 2.4.2 Voorbeeld: na ~ goed voor communicatie



Merk op: ver van alle grafieken getoond in dit hoofdstuk zijn goed voor communicatie zonder aanpassingen.

### 2.5 How charts lie

#### 2.5.1 Causaliteit vs correlatie

- Van zodra er twee (of meer) variabelen zijn, gaan we op zoek naar patronen in relaties tussen de variabelen.
- Het is belangrijk en essentieel te beseffen dat mensen een automatische reflex hebben om te denken in termen van oorzaak-gevolg als we kijken naar relaties tussen twee variabelen.
  - Het is echter niet omdat er een duidelijke relatie bestaat tussen

twee variabelen (correlatie), dat hier sprake is van een oorzaak-gevolg verband (causaliteit).

- Bijvoorbeeld: Indien in de zomer de verkoop van paraplu's sterk stijgt, dan zal de graanopbrengst in het najaar dalen. Dit betekent niet dat de verkoop van paraplu's een impact heeft op de graanopbrengst. Wat hier waarschijnlijk gebeurt, is dat door hevige regenval in de zomermaanden, de verkoop van paraplu's is toegenomen en de graanoogst tegenvalt.
  - \* Soms is het intuïtief zeer onwaarschijnlijk dat de waargenomen correlatie causaliteit impliceert. Kijk hiervoor maar eens naar de voorbeelden op <http://www.tylervigen.com/spurious-correlations>
  - \* Wanneer het echter plausibel is dat de waargenomen correlatie causaliteit voorstelt, is het belangrijk dat we tegen onze natuurlijke reflex in gaan en niet in termen van oorzaak-gevolg denken.
  - \* Het aantonen van causaliteit is nooit mogelijk met descriptieve en exploratieve data analyse!

## 2.6 Referenties

- Information is Beautiful
- Fundamentals of Data Visualization
- R Graph Gallery
- Data to viz
- Spurious correlations
- Misleading election map

# 3

## Tutorial - Data vizualisatie

### 3.1 Before you start

Before you start this tutorial, make sure to install the package `ggplot2`, if you haven't already done so. You can do this using the following line of code:

```
install.packages("ggplot2")
```

In any case, you need to load the package into your session.

```
library(ggplot2)
```

You also need two datasets, `movies` and `diamonds`. Both will be provided as a .RDS file with this tutorial.

```
movies <- readRDS("movies.RDS")
diamonds <- readRDS("diamonds.RDS")
```

### 3.2 Introduction

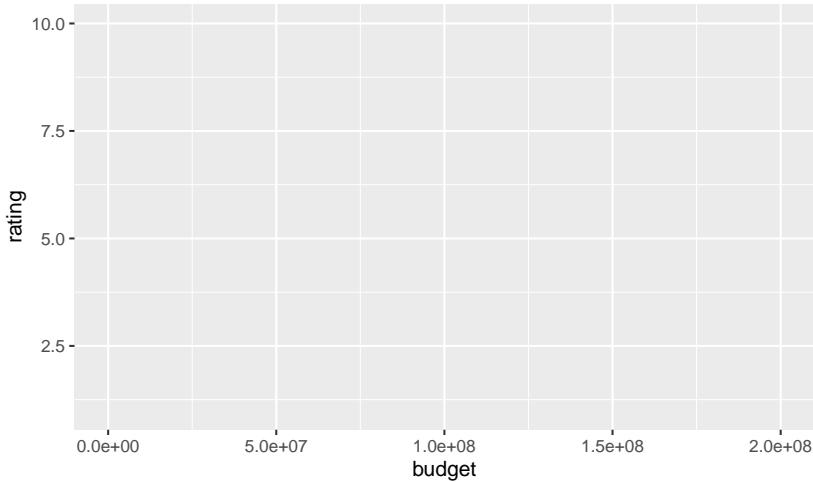
Creating a plot with `ggplot2` starts with the `ggplot()` function. The `ggplot` function has two important arguments

- **data**: this will define the dataset to be used for the plot. This must be a `data.frame`.
- **mapping**: the mapping will define how the variables are *mapped* onto the aesthetics<sup>1</sup> of the plot, as will be explained further on. This mapping must always be created using the `aes()` function .

For example, we might want to make a scatterplot of the movies in which we use the x-axis for their budget and the y-axis for their rating. We then call `ggplot` as follows:

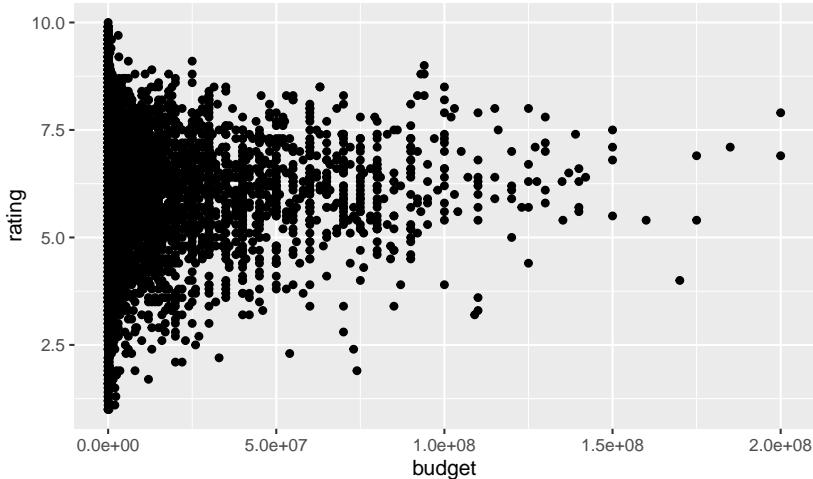
```
ggplot(data = movies, mapping = aes(x = budget, y = rating))
```

<sup>1</sup> Although it might make certain analyses more interesting when you are familiar with the meaning of the different variables, no specific knowledge about cars is required in order to complete this tutorial.



As you can see, this line of code creates a plot with the axes as defined. However, there is no data visualised. The reason therefore is that ggplot doesn't know yet how we want to visualise it. We have to add what is called *a geometric layer*. To create a scatterplot, which consists of *points*, we add `geom_point` to the plot.

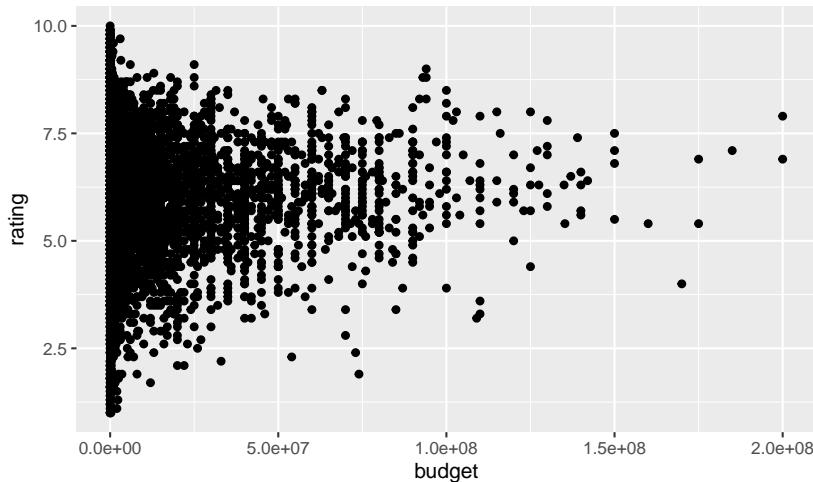
```
ggplot(data = movies, mapping = aes(x = budget, y = rating)) +
  geom_point()
```



This already looks more like it. Remark that the geometric layer was added to the plot by using the `+` symbol. In this way, multiple layers can be added to the same plot, as well as titles, labels, and configurations of the lay-out, as we will see further on.

Note that we can also place the mapping of the aesthetics in the geometric layer itself. This will make our code slightly more readable for now, as we place the geometric layer and its mapping on the same line.

```
ggplot(data = movies) +
  geom_point(mapping = aes(x = budget, y = rating))
```



We have now created our very first plot! In the following sections we will learn how to use different geometries and aesthetics and how to enhance the layout of our graphs.

### 3.3 Different geometries

Next to `geom_point`, many more different geometries exist to plot data in ggplot. You can check them out by typing '`geom_`' in the console and navigating through the auto-completion list. Each of the geom-layers comes with their own specific set of aesthetics that can (and sometimes need to) be mapped. In this tutorial, we will focus mostly on the following geometric layers:

- `geom_point`
- `geom_histogram`
- `geom_boxplot`
- `geom_violin`
- `geom_bar`
- `geom_col`

#### 3.3.1 `geom_point`

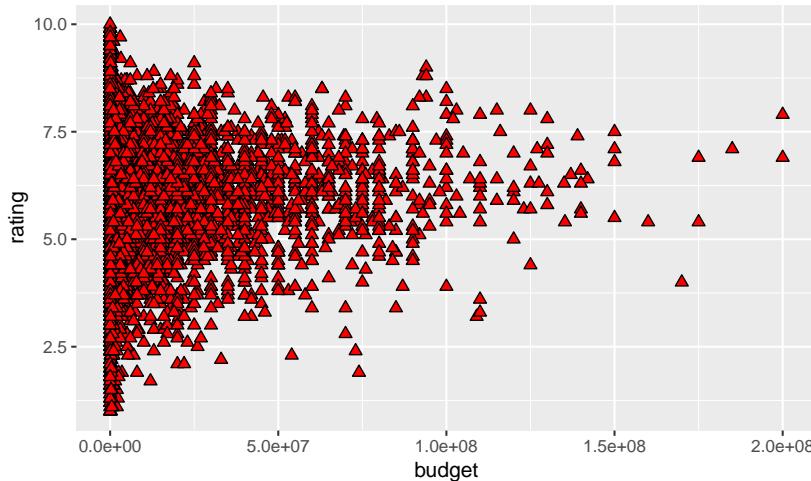
We already used `geom_point` to make our first graph, where we mapped two variables on the aesthetics `x` and `y`. However, there are some more aesthetics which can be set using this layer. Let's look at them in more detail.

- `x`: this will define the position of the points along the x-axis
- `y`: this will define the position of the points along the y-axis

- **color**: this will define the color of the points
- **shape**: this will define the type of points to be plotted<sup>2</sup>
- **fill**: this will define how the points are filled (for shapes 21-25)
- **size**: this will define the size of the points
- **stroke**: this will define the width of the border
- **alpha**: this will define the degree of transparency

For example, the graph below will plot black triangles filled with red with a size of 2. Note that the position of the triangles is exactly the same as the position of the points in the previous plot.

```
ggplot(data = movies) +
  geom_point(mapping = aes(x = budget, y = rating),
             shape = 24,
             fill = "red",
             color = "black",
             size = 2)
```

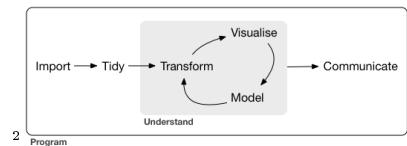


But wait, there is something important going on here! While the x and y-aesthetics were defined inside the aes-mapping, the other aesthetics were defined outside of it. Why's that?

Actually, aesthetics can be set in two different ways:

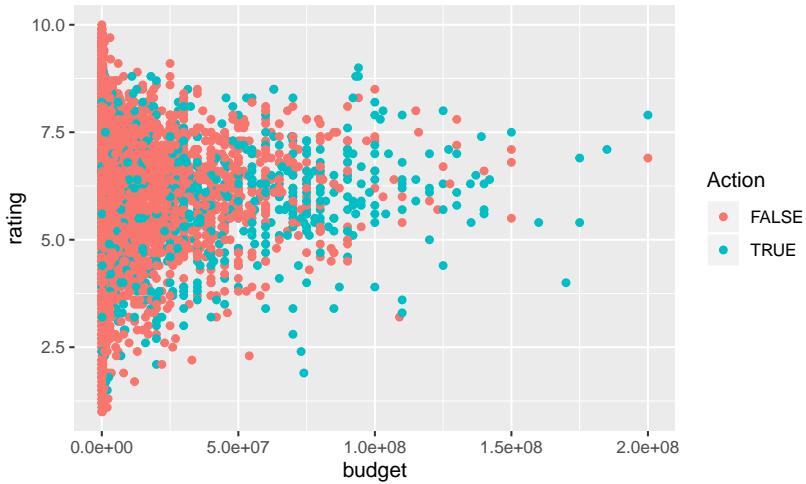
1. They can be *mapped* to a variable in the dataset.
2. They can be set to one fixed value.

In our example, x and y are mapped to two variables in the data, i.e. budget and rating, while the other aesthetics, shape, fill, color and size, are set to fixed values. Although some of the aesthetics typically are always mapped, such as x and y-positions, some others can be a fixed value as well as a mapped variable. For instance, what happens if we map the color of points to a variable, say the variable Action.



<sup>2</sup> The *tidyverse* is a set of packages for data science that work in harmony because they share common data representations and API design. The *tidyverse*-package is designed to make it easy to install and load core packages from the tidyverse in a single command. The tidyverse includes packages such as: ggplot2, dplyr, tidyr, readr, purrr, tibble, hms, stringr, lubridate, forcats, jsonlite, readxl, broom, and others. Hadley Wickham can be regarded as the founding father of the tidyverse. The best place to learn about these packages is by reading this book: R for Data Science, written by Hadley and Garret Grolemund.

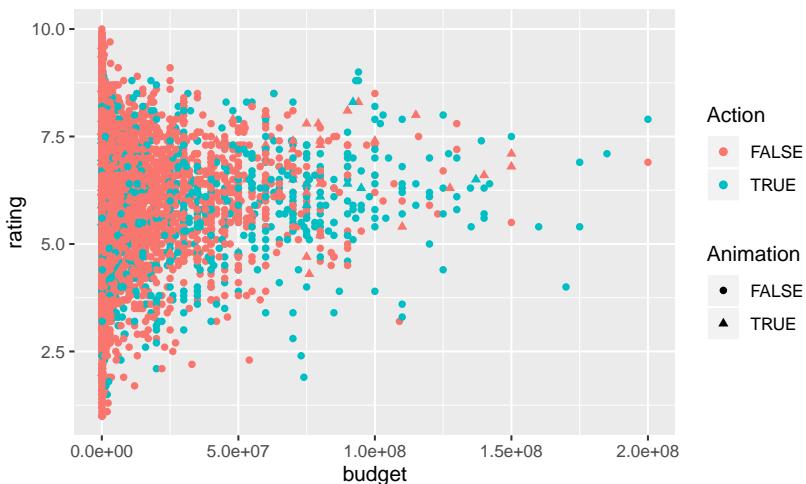
```
ggplot(data = movies) +
  geom_point(mapping = aes(x = budget,
                           y = rating,
                           color = Action))
```



We see that points are now colored with respect to the value in the variable Action. Action movies receive a green color, while other movies receive a red color, which we can see in the legend that appeared.

Also the other fixed aesthetics can be used in a mapping. The next example uses the variable Animation for the shape mapping.

```
ggplot(data = movies) +
  geom_point(mapping = aes(x = budget,
                           y = rating,
                           color = Action,
                           shape = Animation))
```



Great! We now completely understand the `geom_point` layer, and the working of the aesthetics-mapping. Now it's time to look at some other geometric layers. We start with histograms.

### 3.3.2 `geom_histogram`

The `geom_histogram` layer can be used to plot a histogram. As you should already know, a histogram represents the distribution of **one** continuous variable. As a result, there is only an `x`-aesthetic that should be set, and no `y`-aesthetic. The complete list of aesthetics is as follows:

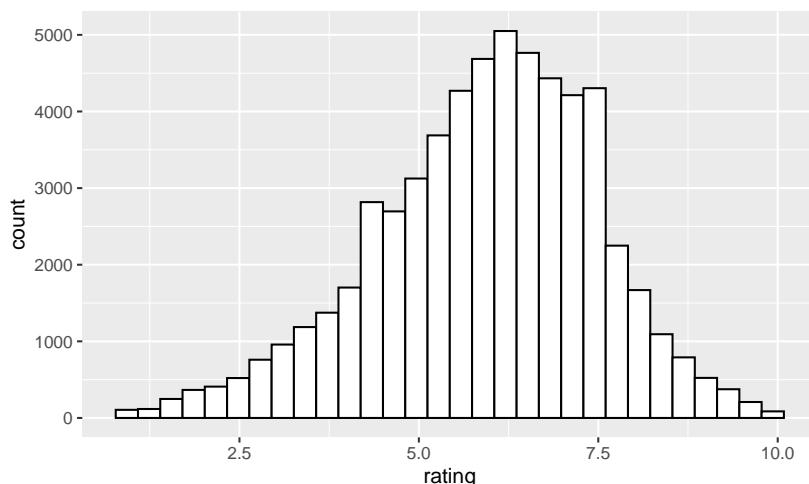
- `x`: this will define the variable to be used
- `color`: this will define the color of the borders
- `fill`: this will define how the histogram is filled
- `size`: this will define the size of the border
- `linetype`: this will define the type of the border <sup>3</sup>
- `alpha`: this will define the degree of transparency
- `weight`: this will define how observations should be weighted. By default, each observation is weighted as one.

<sup>3</sup> Note the capital letter I

Using our knowledge on how to use aesthetics from before, it is now very easy to make a histogram. Let's make a histogram for the rating of movies. We will give it a black border with a white fill. Ready to try?

Let's have a look.

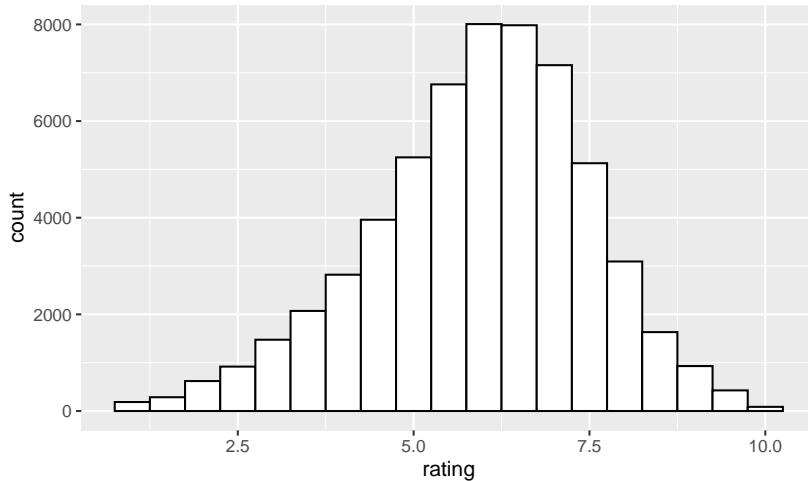
```
ggplot(movies) +
  geom_histogram(aes(rating), color = "black", fill = "white")
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Note that several things were omitted in these two lines of codes. In particular, the argument name *data* in `ggplot`, *mapping* in `geom_histogram` and *x* in `aes`. Since we know that these are the first arguments of these functions, we can safely omit them, as long as we follow the correct order of arguments. However, we cannot omit *color* and *fill*, as these are not the second and third argument of `geom_histogram`. Whenever in doubt, just play safe and write down the proper argument names.

However, that's not the only thing which is tricky here. Indeed, a warning pops up: `stat_bin()` using `bins = 30`. Pick better value with `binwidth`. This warning reminds us to the fact that a default value for the number of bins is chosen by `geom_histogram`, which is probably not suitable for our graph. We can change the binwidth by adding it as an argument to the `geom_histogram` call.

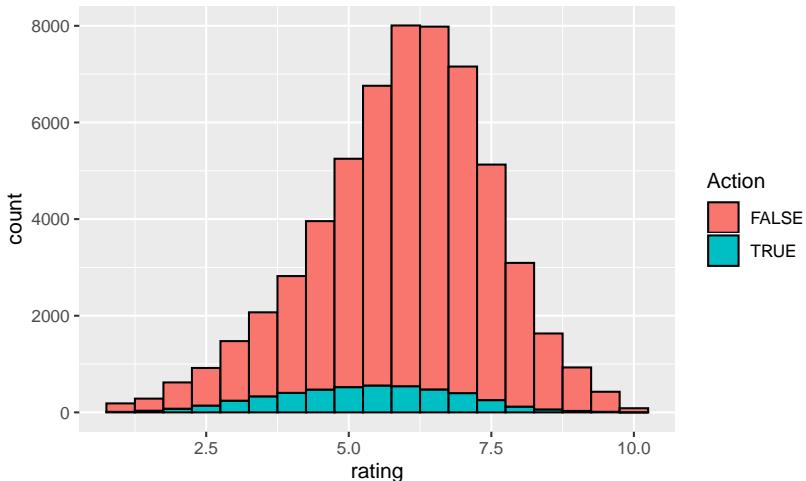
```
ggplot(movies) +
  geom_histogram(aes(rating), color = "black", fill = "white", binwidth = 0.5)
```



Often, the binwidth has an important impact on how the obtained histogram looks like. Carefully configuring this argument by experimenting with different values is therefore important.

In the last plot, we used a fixed color for filling the bars of the histogram. But as we already know by now, we can also map it to a variable in the data. Let's use the variable *Action* one more time.

```
ggplot(movies) +
  geom_histogram(aes(rating, fill = Action), color = "black", binwidth = 0.5)
```

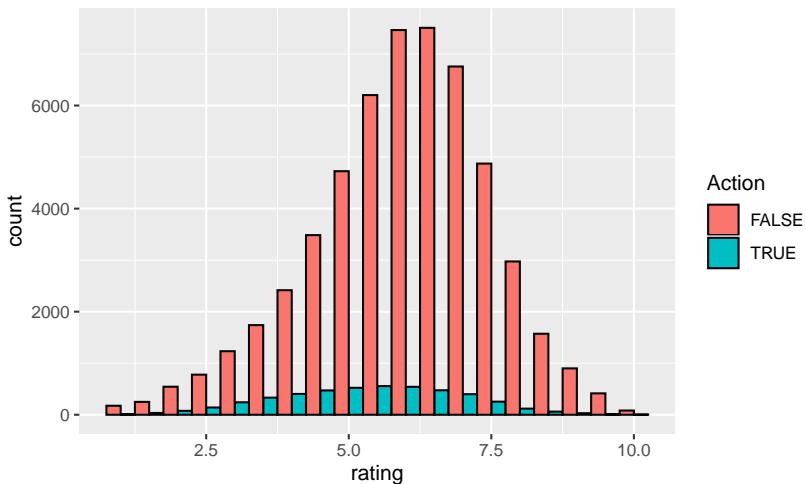


Note how we moved the `fill` argument inside the `aes` function. Now, each of the bars is filled with two colors: one part for Action movies, and the remainder for other movies. Although it is not very clear in this graph, it appears that the center of the histogram for action movies is slightly more to the left.

Note how the punctuation changes when moving to the `aes-mapping`: variable names are always used without quotes, i.e. *bare* variable names, while fixed aesthetics (colors, shapes, linetypes) are used with quotes (except for numbers). It is important to not mess this up!

By default, the histogram for the different *fills* are stacked, i.e. placed on top of each other. However, we can use the `position` argument of `geom_histogram` to put the bars next to each other, or *dodge*.

```
ggplot(movies) +
  geom_histogram(aes(rating, fill = Action),
    color = "black", binwidth = 0.5, position = "dodge")
```



Using `position = "dodge"`, the bars for Action movies and non-

Action movies are placed next to each other, instead of on top of one another. We can go back to the original graph by using position = “stack”, or just omitting this argument. Later on we will see how to handle such things better by using grids of different plots, or so-called *facets*.

All good so far? Let’s look at another way to visualize the distribution of continuous variables, i.e. the boxplot.

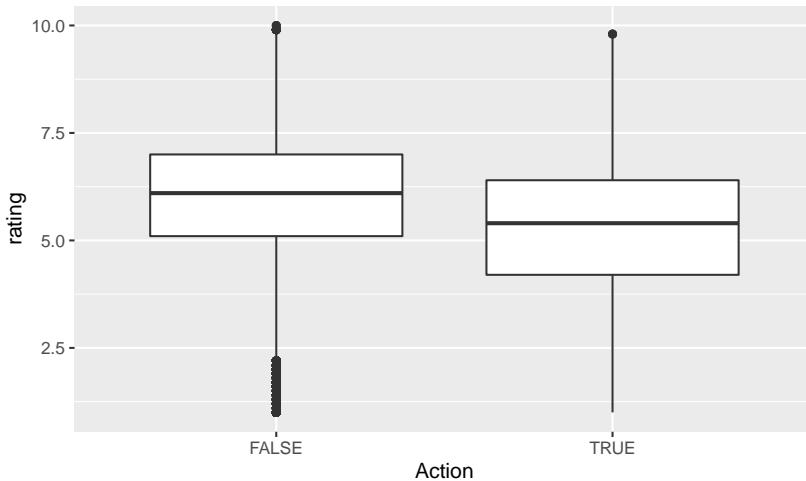
### 3.3.3 geom\_boxplot

A boxplot places the values of the variable along the y-axis. So, if we want to make a boxplot for ratings, we should use `aes(y = ratings)`. However, something tricky is going on here... The aesthetics for `geom_boxplot` are the following

- **x**: this will define the variable used for the x-axis
- **y**: this will define the variable used for the y-axis
- **color**: this will define the color of the borders
- **fill**: this will define how the boxplot is filled
- **size**: this will define the size of the border
- **linetype**: this will define the type of the border
- **alpha**: this will define the degree of transparency
- **weight**: this will define how observations should be weighted. By default, each observation is weighted as one.

Thus, the boxplot needs both an x-variable and an y-variable? That seems strange at first sight. The reason behind this is that in the philosophy of ggplot, always *something* must be plotted on both x and y-axes. Although only a x-variable is given to a histogram, it will compute frequencies to plot on the y-axis. However, there is no such thing going to happen for boxplots. As a result, the x-axis should be used to map different categories for which the distribution can then be compared. For instance, we can compare the rating for Action movies with those for other movies.

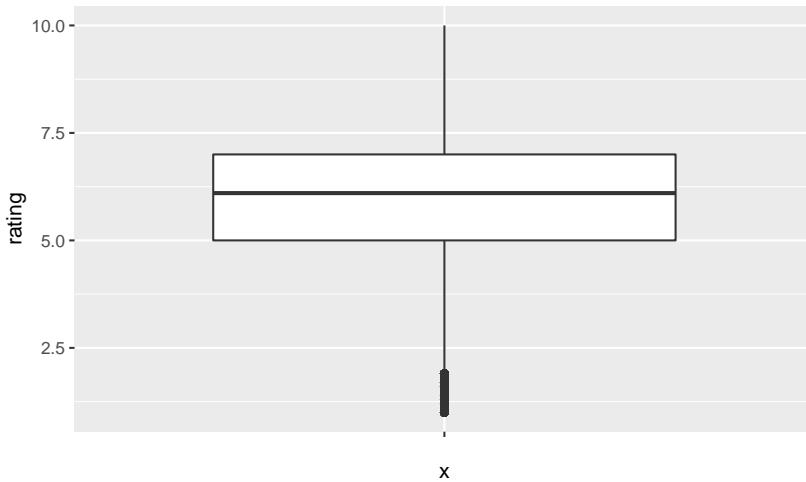
```
ggplot(movies) +
  geom_boxplot(aes(Action, rating))
```



Here we see that, as we already suspected, Action movies tend to have a lower rating compared to other movies. We can change the color and the fill of the boxplot just like before, as well as the linetype, the size of the border, or the transparency.

But, what if we just want to plot a boxplot of the overall rating, without having to specify a variable for the x-axis? A little workaround is needed here. One possibility is to use an *empty string* for the x-axis mapping.

```
ggplot(movies) +
  geom_boxplot(aes("", rating))
```



Note that this creates the label "x" for the x-axis, where we would normally find the name of the variable mapped onto it. Later we will see how to omit this label to make our graph a little nicer.

### 3.3.4 geom\_violin

The violin-plot is similar to the boxplot, although it will reflect in more detail where the mass of the values is located. Its aesthetics are the same as for a boxplot.

- **x**: this will define the variable used for the x-axis
- **y**: this will define the variable used for the y-axis
- **color**: this will define the color of the borders
- **fill**: this will define how the violin is filled
- **size**: this will define the size of the border
- **linetype**: this will define the type of the border
- **alpha**: this will define the degree of transparency
- **weight**: this will define how observations should be weighted. By default, each observation is weighed as one.

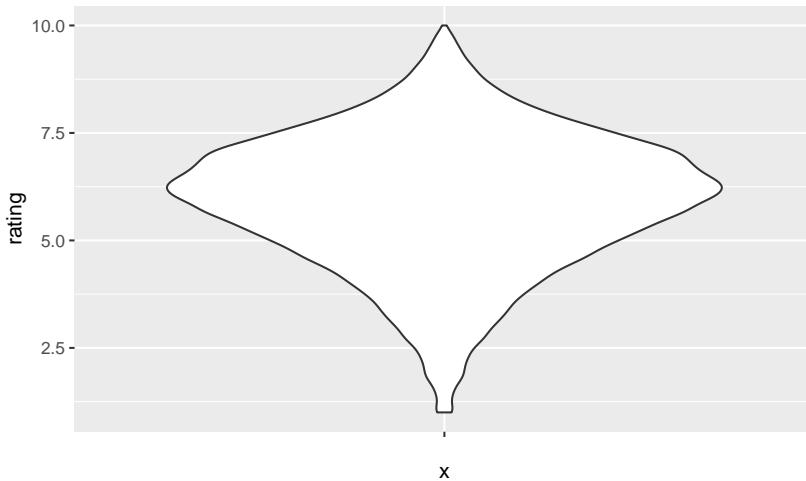
Let's make the same graphs, now using the violin-plot.

```
ggplot(movies) +
  geom_violin(aes(Action, rating))
```



It's probably clear now where this type of graph got its name from. As you can see, violin plots somehow sit in between boxplots and histograms. However, since their width is normalized, they can be used better for comparison. Also here, we can use the same workaround if we want to plot the overall distribution.

```
ggplot(movies) +
  geom_violin(aes("", rating))
```



So far, we have seen three different ways to analyse the distribution of continuous variables. Now, we will look into barplots, which can be used to represent categorical distribution.

### 3.3.5 *geom\_bar*

Just like a histogram, a barplot only needs an x variable. The difference is that this variable needs to be categorical, while for histograms it needs to be continuous. The full list of aesthetics is the following:

- **x**: this will define the variable used for the x-axis
- **color**: this will define the color of the borders
- **fill**: this will define how the bars are filled
- **size**: this will define the size of the border
- **linetype**: this will define the type of the border
- **alpha**: this will define the degree of transparency
- **weight**: this will define how observations should be weighted. By default, each observation is weighted as one.

We can make a simple bar chart that shows how many Action movies there are, and how many other movies, as follows.

```
ggplot(movies) +
  geom_bar(aes(Action))
```



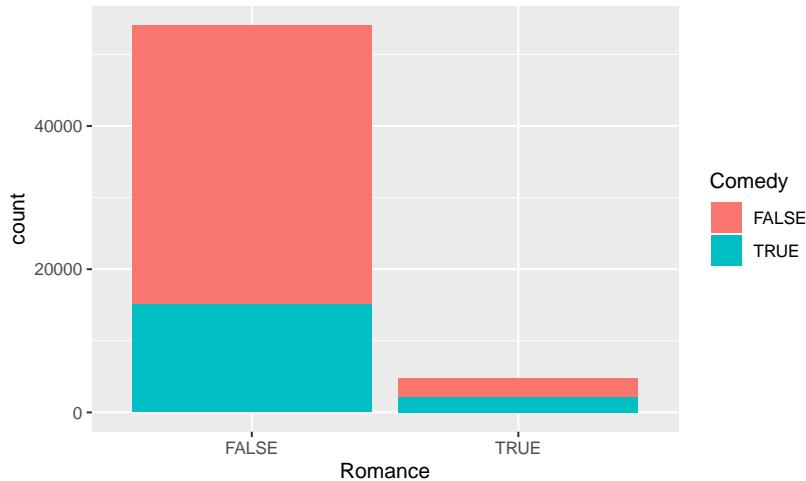
Furthermore, we could add colors to this, according to the number of Animation movies. We immediately see that there are almost no Action movies which are also animation movies.

```
ggplot(movies) +
  geom_bar(aes(Action, fill = Animation))
```



We can do the same for Romance and Comedy movies.

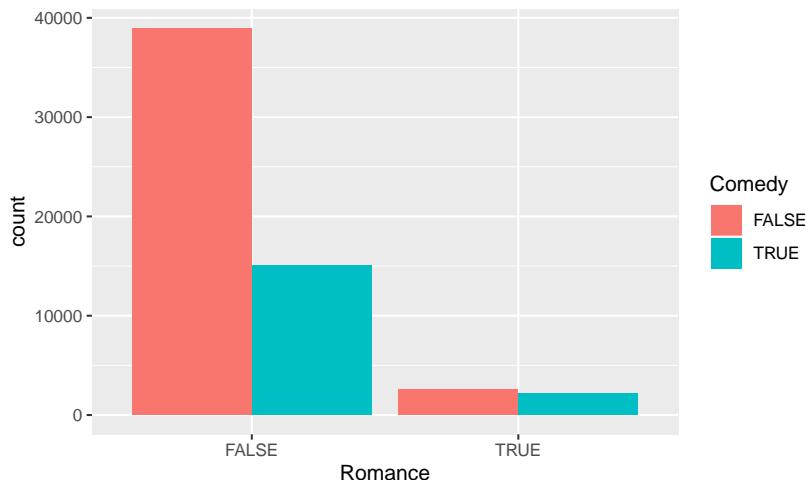
```
ggplot(movies) +
  geom_bar(aes(Romance, fill = Comedy))
```



In contrast, it can be seen here that approximately half of the Romantic movies are also comedy, which is more compared to non-romantic movies.

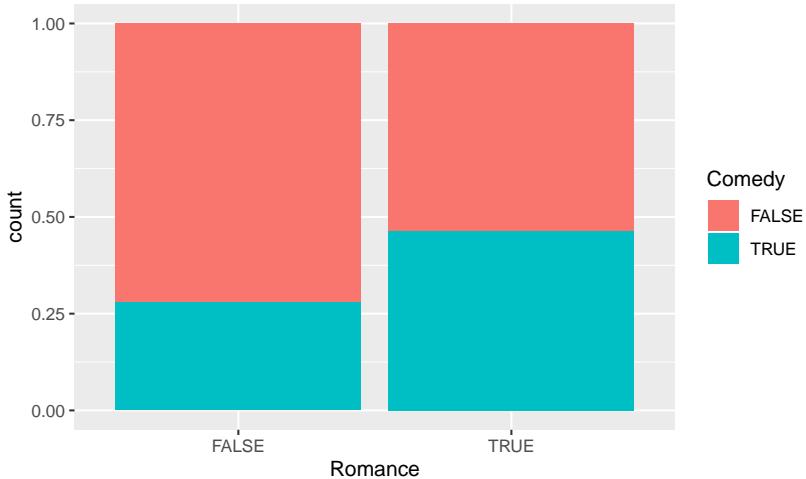
Remember that, in case of histograms, we could change the position to “dodge”, which placed the bars next to each other. The same can be done here.

```
ggplot(movies) +
  geom_bar(aes(Romance, fill = Comedy), position = "dodge")
```



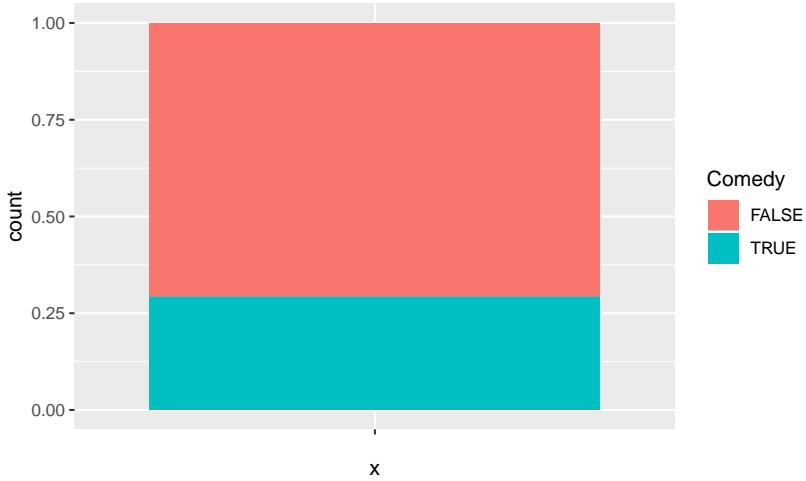
A third option that is available for the position is to extend the bars so that they have the same height. As a result, we will observe the distribution of values as a part of a whole. Instead of the count, the labels on the y-axis will now depict the percentage points.

```
ggplot(movies) +
  geom_bar(aes(Romance, fill = Comedy), position = "fill")
```



Finally, if we want to do this to show the distribution of one variable, we can use the same workaround as before and set the x-aesthetic to `" "`. The plot below will show the portion of all movies that are comedies.<sup>4</sup>

```
ggplot(movies) +
  geom_bar(aes(" ", fill = Comedy), position = "fill")
```



<sup>4</sup> Note that in cases like these, it perfectly makes sense to make the plot less wide, or flip it 90 degrees and make it less high. However, these are configurations related to the output devices used, and we will not consider them here.

### 3.3.6 `geom_col`

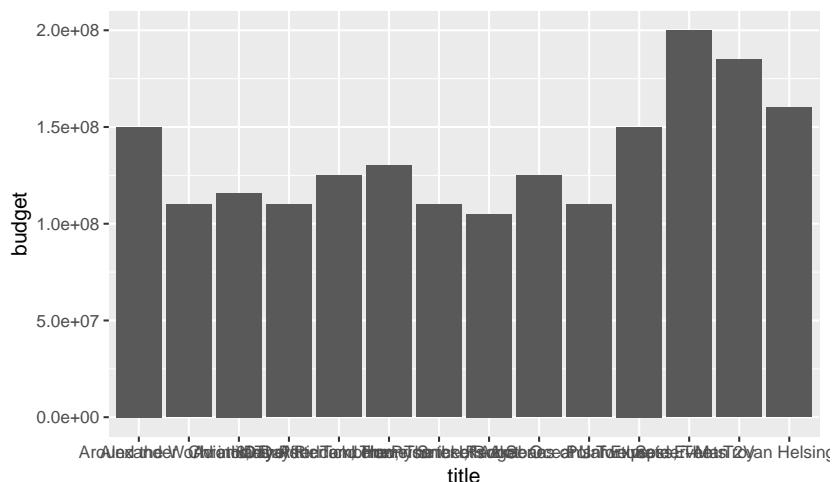
When we use `geom_bar`, the heights of the bars are computed by using the frequency of the categorical variable. However, sometimes we just want to plot a bar chart with values that are already in the data, or values which we computed ourselves. For instance, what if we wanted a barchart depicting the budget of a set of movies? In such a case, we can use `geom_col`. “Col” indicates that we want to use a column in the data to set the height of the bars. The aesthetics are

the same for `geom_bar`, only now we need to specify a variable for the y-axis, evidently.

- **x**: this will define the variable used for the x-axis
- **y**: this will define the variable used for the y-axis
- **color**: this will define the color of the borders
- **fill**: this will define how the bars are filled
- **size**: this will define the size of the border
- **linetype**: this will define the type of the border
- **alpha**: this will define the degree of transparency
- **weight**: this will define how observations should be weighed. By default, each observation is weighed as one.

Let's make a barplot depicting the budget of all movies from 2004 where the budget was higher than 100 million.

```
filter(movies, year == 2004, budget > 100000000) %>%
  ggplot() +
  geom_col(aes(title, budget))
```



Do you notice something strange in the code? Don't worry if you don't understand the first line. All you need to know is that we filtered movies from 2004 with a budget higher than 100 million. The strange looking `%>%` symbol will make sure this data is passed to `ggplot`. We will come back to this in another session.<sup>5</sup>

We have now plotted movie titles on the x-axis and budget on the y-axis. Great! Or not so? The values on the x-axis are somewhat cluttered and unreadable. It's time now to give some attention to the layout of our plots!



### 3.3.7 Other geometrics

So far, we used the most important geometrics to make simple visualizations: scatterplots, histograms, boxplots, violinplots and bar plots. However, we only covered the tip of the iceberg, as many more types exist, some simple and some more advanced. An overview of all geoms and their uses can be found in the ggplot Cheat Sheet, of which an extract is shown here. Don't be afraid to try something out!

<b>Geoms</b> - Use a geom to represent data points, use the geom's aesthetic properties to represent variables. Each function returns a layer.		
<b>Graphical Primitives</b>	<b>Two Variables</b>	<b>Continuous Bivariate Distribution</b>
a <- ggplot(seals, aes(x = long, y = lat)) b <- ggplot(economics, aes(date, unemploy))	c <- ggplot(mpg, aes(cty, hwy))   a + geom_blank() (Useful for expanding limits)   a + geom_curve(aes(end = lat + delta_lat, xend = long + delta_long, curvature = 2)) x, y, end, alpha, angle, color, curvature, linetype, size   b + geom_path(inend = "butt", linejoin = "round", linemiter = 1) x, y, alpha, color, group, linetype, size   b + geom_polygon(aes(group = group)) x, y, alpha, color, fill, group, linetype, size   a + geom_rect(aes(min = long, ymin = lat, xmax = long + delta_long, ymax = lat + delta_lat)) xmax, xmin, ymax, ymin, alpha, color, fill, linetype, size   b + geom_ribbon(aes(min = unemploy - 900, ymax = unemploy + 900)) x, ymax, ymin, alpha, color, fill, group, linetype, size   a + geom_segment(aes(end = lat + delta_lat, xend = long + delta_long)) x, yend, alpha, color, linetype, size   a + geom_spoke(aes(yend = lat + delta_lat, xend = long + delta_long)) x, y, angle, radius, alpha, color, linetype, size	h <- ggplot(diamonds, aes(carat, price))   h + geom_bin2d(binwidth = c(0.25, 50)) x, y, alpha, color, fill, linetype, size, weight   h + geom_density2d() x, y, alpha, color, group, linetype, size   h + geom_hex() x, y, alpha, color, fill, size
<b>One Variable</b>	<b>Continuous Function</b>	<b>Visualizing error</b>
Continuous c <- ggplot(mpg, aes(hwy))   c + geom_area(stat = "bin") x, y, alpha, color, fill, linetype, size   c + geom_density(kernel = "gaussian") x, y, alpha, color, fill, group, linetype, size, weight   c + geom_dotplot() x, y, alpha, color, fill   c + geom_freqpoly() x, y, alpha, color, group, linetype, size, weight   c + geom_histogram(binwidth = 5) x, y, alpha, color, fill, linetype, size, weight  Discrete d <- ggplot(mpg, aes(rf))   d + geom_bar() x, alpha, color, fill, linetype, size, weight	i <- ggplot(economics, aes(date, unemploy))   i + geom_area() x, y, alpha, color, fill, linetype, size   i + geom_line() x, y, alpha, color, group, linetype, size   i + geom_step(direction = "hv") x, y, alpha, color, group, linetype, size	
		j <- data.frame(grp = c("A", "B"), fit = 4:5, se = 1:2) j <- ggplot(df, aes(grp, fit, ymin = fit-se, ymax = fit+se))   j + geom_crossbar(fatten = 2) x, ymax, ymin, alpha, color, fill, group, linetype, size   j + geom_errorbar() x, ymax, ymin, alpha, color, group, linetype, size, width (also geom_errorbarh())   j + geom_linerange() x, ymax, ymin, alpha, color, group, linetype, size   j + geom_pointrange() x, ymin, ymax, alpha, color, fill, group, linetype, shape, size
	<b>Three Variables</b>	<b>Maps</b>
	seals\$z <- with(seals, sqrt(delta_long^2 + delta_lat^2)) l <- ggplot(seals, aes(long, lat))   l + geom_contour(aes(z = z)) x, y, z, alpha, colour, group, linetype, size, weight	data <- data.frame(murder = USArrests\$Murder, state = tolower(rownames(USArrests))) map <- map_data("state") k <- ggplot(data, aes(fill = murder)) k + geom_map(data = map, id = state), map = map) + expand_limits(x = map\$long, y = map\$lat) map_id, alpha, color, fill, linetype, size

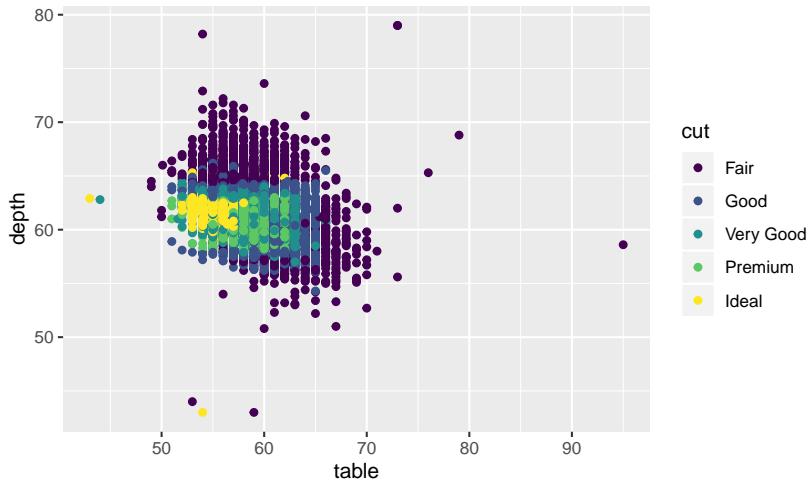
### 3.4 Enhancing the appearance of our plots

So far, we have mainly looked at different types of plots and how to map them on our data. In this section, we will focus on the presentation of the plot, e.g. titles, colors, axes, etc. The concepts introduced in this section can be applied for any type of plot, no matter which geometric object is used.

In this section, the dataset “diamonds” will be used. The plot below will be used as a starting point.<sup>6</sup>

```
ggplot(diamonds) +
  geom_point(aes(table, depth, color = cut))
```

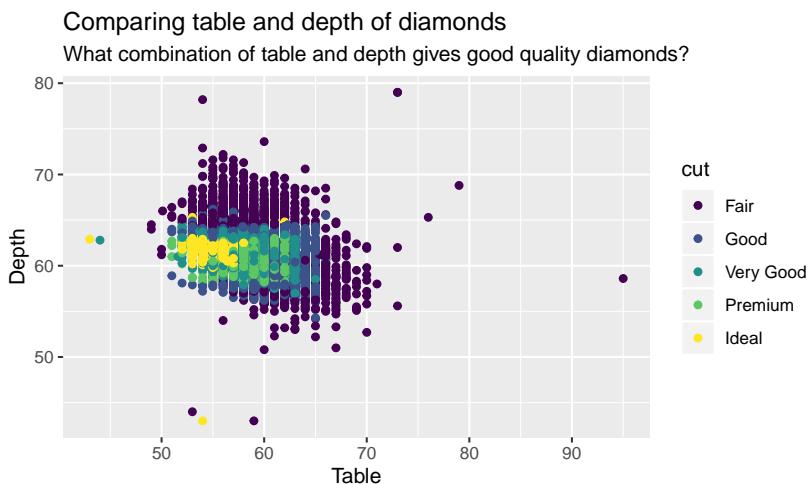
<sup>6</sup> The table of a diamond refers to the flat facet of the diamond which can be seen when the stone is faced up. The depth of a diamond is its height (in millimeters) measured from the culet to the table.



### 3.4.1 Titles

One of the most important things to add to our plot are titles. Titles are used to give meaning to the axis as well as the plot itself. The most straightforward way to add titles is to use the function `labs()`. In this function, 4 arguments can be set: the title, the subtitle, x for the x label and y for the y label. The `labs` function can just be added to the plot as an extra layer.

```
data("diamonds")
ggplot(diamonds) +
  geom_point(aes(table, depth, color = cut)) +
  labs(title = "Comparing table and depth of diamonds",
       subtitle = "What combination of table and depth gives good quality diamonds?",
       x = "Table",
       y = "Depth")
```



You'll see that our graph already looks much better when titles are added! However, there is much more to improve.

### 3.4.2 Theme

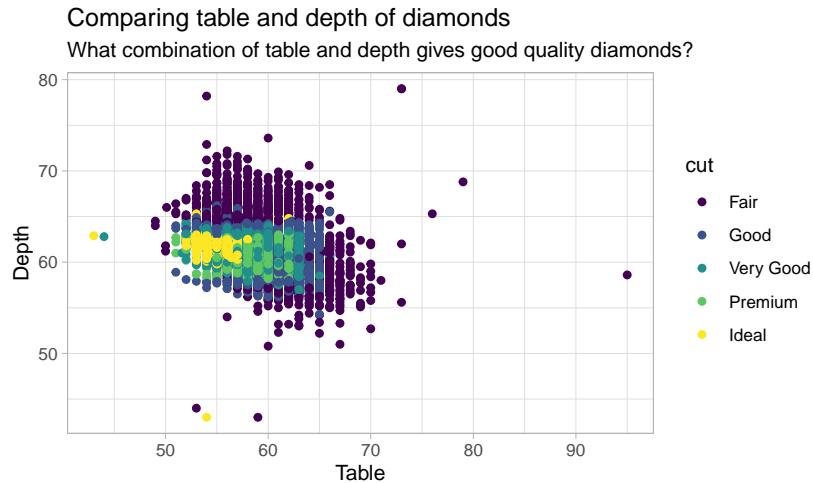
The *theme* of a plot defines its overall appearance: the gridlines, the background, the size of the text, titles and legend, the position of the legend, etc. The theme can be defined manually by adding a `theme()` layer to the plot and by setting the required arguments. (You can look at `?theme` to see the arguments which are available). However, this is a cumbersome approach. Fortunately, some predefined themes are provided in ggplot:

- `theme_gray`: the default theme (used so far)
- `theme_bw`: a theme for black-white plots
- `theme_dark`: a dark theme for contrast
- `theme_classic`: a minimal theme
- `theme_light`: another minimal theme
- `theme_linedraw`: yet another minimal theme
- `theme_minimal`: yet another minimal theme
- `theme_void`: an empty theme

Feel free to experiment with some of these themes. Preferably, you can use some of the minimal themes. Here, we used the `theme_light` theme.<sup>7</sup>

```
ggplot(diamonds) +
  geom_point(aes(table, depth, color = cut)) +
  labs(title = "Comparing table and depth of diamonds",
       subtitle = "What combination of table and depth gives good quality diamonds?",
       x = "Table",
       y = "Depth") +
  theme_light()
```

<sup>7</sup> For most of the layers, it is not important in which order they are added to a plot. However, if you make manual changes with `theme`, make sure to put them after any predefined theme, otherwise your changes will be overwritten.



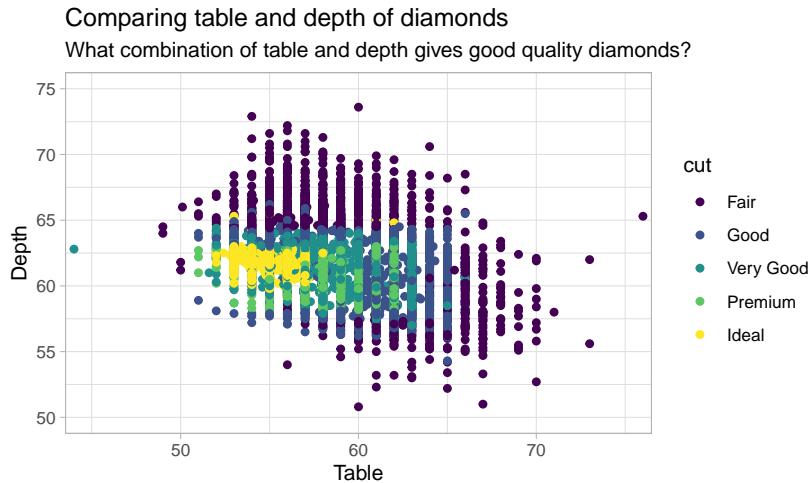
When you are not yet satisfied with any of these themes, you can install the package `ggthemes` to get hold of even more themes, such as the theme from *The Economist*, [fivethirtyeight.com](http://fivethirtyeight.com), or *Google Docs*.

### 3.4.3 Configuring the coordinate system

The appearance of the graph is not only defined by the titles and the graphics. Also the axes in the coordinate system need some attention. One thing to decide on are the limits of the coordinate system. This can be done using the function `coord_cartesian` and its arguments `xlim` and `ylim`. Both arguments expect a numerical vector of length two. Let's see how this works in our example.<sup>8</sup>

<sup>8</sup> A Cartesian coordinate system is a coordinate system that specifies each point uniquely in a plane by a pair of numerical coordinates, which are the signed distances to the point from two fixed perpendicular directed lines, measured in the same unit of length.

```
ggplot(diamonds) +
  geom_point(aes(table, depth, color = cut)) +
  labs(title = "Comparing table and depth of diamonds",
       subtitle = "What combination of table and depth gives good quality diamonds?",
       x = "Table",
       y = "Depth") +
  theme_light() +
  coord_cartesian(xlim = c(45, 75), ylim = c(50, 75))
```



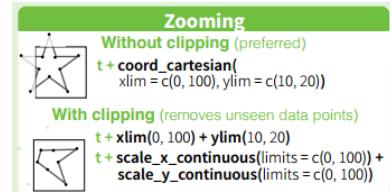
We have now limited the x-axis to the interval from 45 to 75, while we have limited the y-axis to the interval 50 to 75. There are some alternatives to the cartesian coordinate system which are less often used. Notable ones are:

- coord\_equal: a coordinate system where the x-axis and y-axis are scaled equally (i.e. ration = 1)
- coord\_fixed: a coordinate system with a fixed ratio
- coord\_polar: a coordinate system for polar plots, or pie charts
- coord\_map: a coordinate system for plotting geographical data.

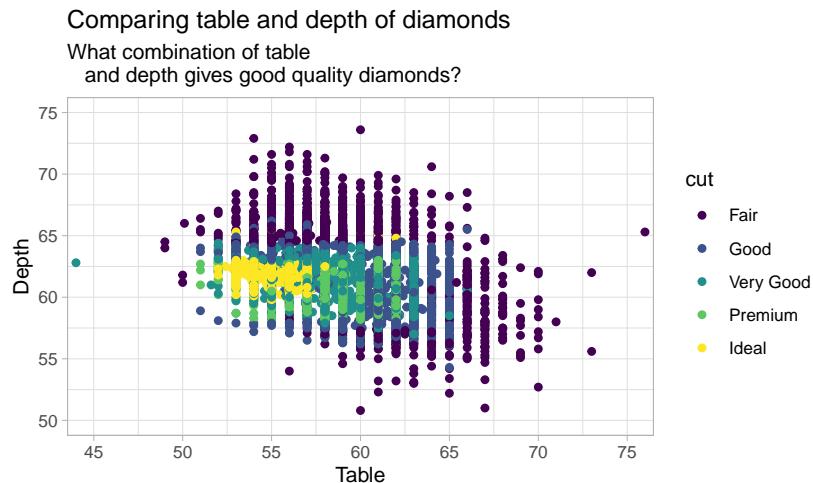
Next to setting the limits of the coordinate system, we can also set the breaks on the x-axis and y-axis. This can be done using the functions `scale_x_continuous` and `scale_y_continuous` respectively. Both functions have a *breaks* argument. This argument can be given a vector of values to be plot as labels on the axis.<sup>9</sup> We can use the function `seq` to create this vector: i.e. `seq(0,10,5)` will return a vector starting at 0 and increasing to ten with intervals of 5.<sup>10</sup>

```
ggplot(diamonds) +
  geom_point(aes(table, depth, color = cut)) +
  labs(title = "Comparing table and depth of diamonds",
       subtitle = "What combination of table and depth gives good quality diamonds?",
       x = "Table",
       y = "Depth") +
  theme_light() +
  coord_cartesian(xlim = c(45,75), ylim = c(50, 75)) +
  scale_x_continuous(breaks = seq(45,75,5)) +
  scale_y_continuous(breaks = seq(50,75,5))
```

<sup>9</sup> Note that the `scale_..._continuous` functions also have an argument `limits` to set the limits of the axes which can be used instead of `coord_cartesian`. However, there is a tricky difference. `Coord_cartesian` will zoom into the limits without throwing away other data points. Setting the limits within the scale functions, however, will throw away data points and can bias your visualisation.

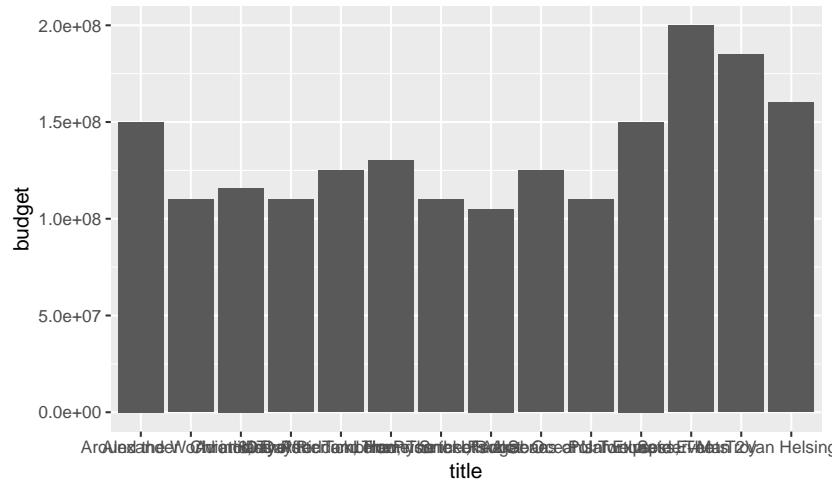


<sup>10</sup> The titles of the axes which we defined with the `labs` function can also be set in the scale functions with the argument `name`. As you get more familiar with using ggplot2, you will often find that there are multiple ways to reach the same goal.



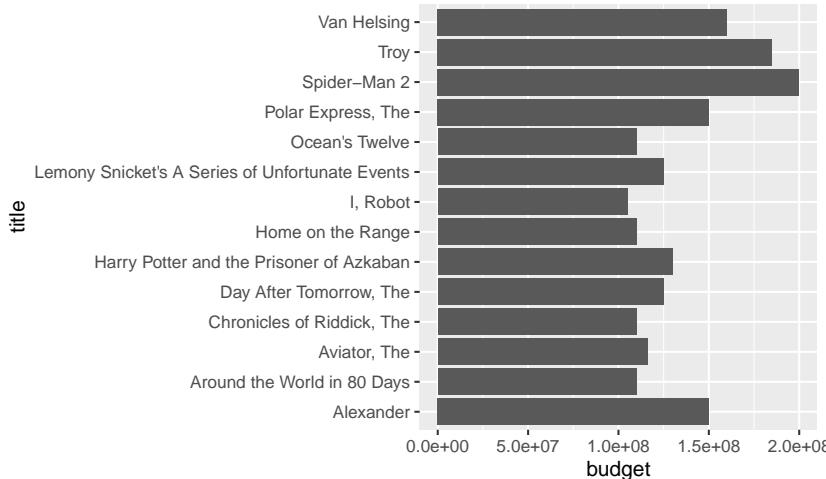
Another useful function is the `coord_flip` function, which we will illustrate with the following graph we saw earlier.

```
filter(movies, year == 2004, budget > 100000000) %>%
  ggplot() +
  geom_col(aes(title, budget))
```



As you may remember, the movie titles on the x-axis were overlapping and therefore unreadable. One way to fix this is to *flip* the entire graph, such that the labels of the x-axis are placed on the y-axis, and can be read horizontally.

```
filter(movies, year == 2004, budget > 100000000) %>%
  ggplot() +
  geom_col(aes(title, budget)) +
  coord_flip()
```



Another option is to keep the original orientation, but the change the orientation of the labels on the x-axis. You could rotate them by 45 or 90 degrees, for instance. This can be done with the `theme` function. Ready to experiment? Challenge yourself!

As our code contains more and more lines, our plot is getting nicer and nicer! Good job! The last thing on our list are colors.

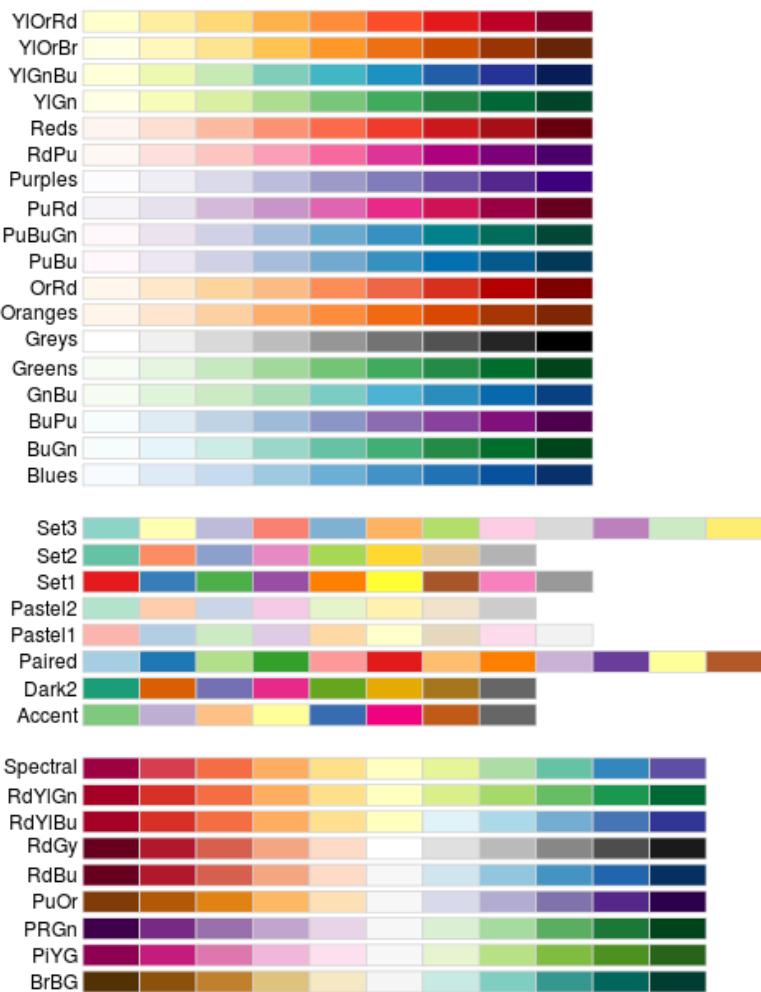
#### 3.4.4 Color scales

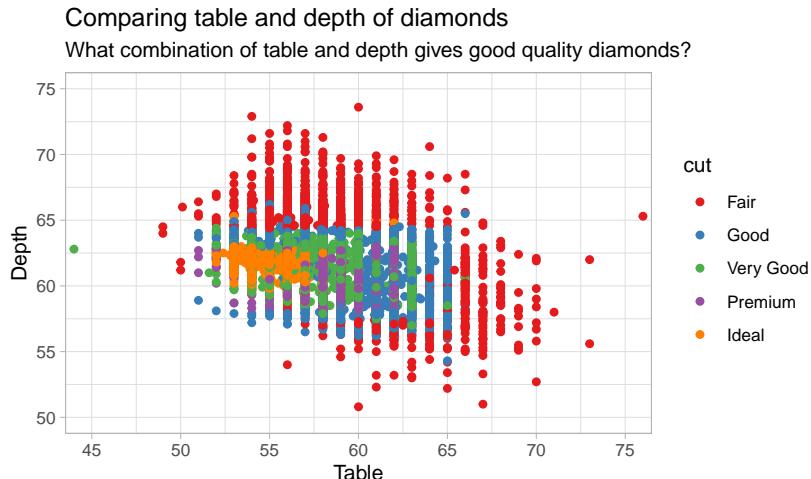
Often, we use color or fill to visualise categorical data, such as the quality of the cut in our graph about diamonds. By default, ggplot will use a rainbow theme. However, many more palettes are available. We can add these by using `scale_color_brewer` or `scale_fill_brewer`, depending on whether it concerns a color or fill-color. Both layers have a palette argument, of which the possible values can be found below.

For instance, let's use the Set1 palette.

```
ggplot(diamonds) +
  geom_point(aes(table, depth, col = cut)) +
  labs(title = "Comparing table and depth of diamonds",
       subtitle = "What combination of table and depth gives good quality diamonds?",
       x = "Table",
       y = "Depth") +
  theme_light() +
  coord_cartesian(xlim = c(45,75), ylim = c(50, 75)) +
  scale_x_continuous(breaks = seq(45,75,5)) +
  scale_y_continuous(breaks = seq(50,75,5)) +
  scale_color_brewer(palette = "Set1")
```

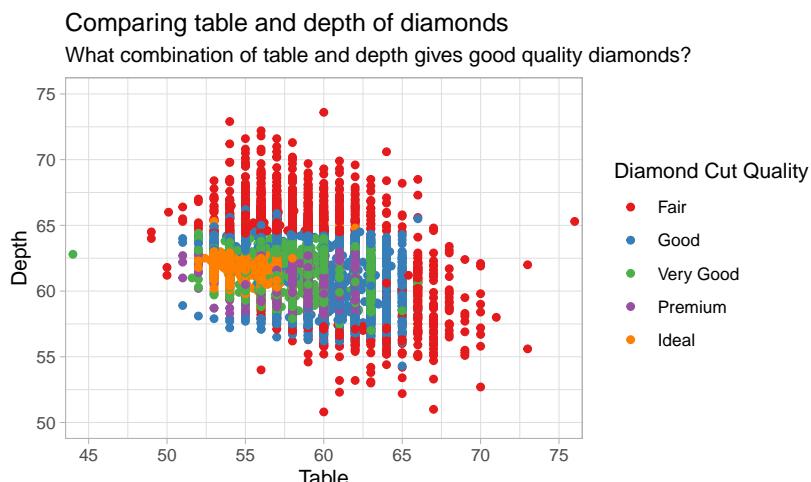
Figure 3.1: R color palettes





The scale\_.brewer functions also have the argument `name`, which we can use to set the name of the legend, and the argument `guide`, which will remove the legend if set to FALSE.

```
ggplot(diamonds) +
  geom_point(aes(table, depth, col = cut)) +
  labs(title = "Comparing table and depth of diamonds",
       subtitle = "What combination of table and depth gives good quality diamonds?",
       x = "Table",
       y = "Depth") +
  theme_light() +
  coord_cartesian(xlim = c(45,75), ylim = c(50, 75)) +
  scale_x_continuous(breaks = seq(45,75,5)) +
  scale_y_continuous(breaks = seq(50,75,5)) +
  scale_color_brewer(palette = "Set1", name = "Diamond Cut Quality")
```



Next to the standard color palettes available, many more can be found in the packages `ggthemes` and `ggsci`. They can be used by

adding `scale_color + name` of the palette. Feel free to explore some more!

### 3.5 Advanced plots

Often, you want to compare plots for different categories of a variable. In our example, we looked for which combinations of table and depth, the cut of the diamond was good. We now want to know whether there is a difference between the 8 different clarity levels in the data. One way would be to make 8 different plots, one for each of the levels. However, this would be cumbersome to put together. Fortunately, we can use the function `facet_grid` to create different plots within a plot.

#### 3.5.1 Using Facets

This function expects a formula in the form of `A ~ B` where A and B are two categorical variables. For each combination of values of A and B, a different plot will be constructed, and they will be arranged in a grid where the values of A each constitute a row and the values of B each constitute a column. Comparisons of more than 2 variables are possible using a formula of the form `A + B ~ C`. A comparison along one variable is possible by using a dot instead of a variable name, i.e. `. ~ A` or `A ~ .`

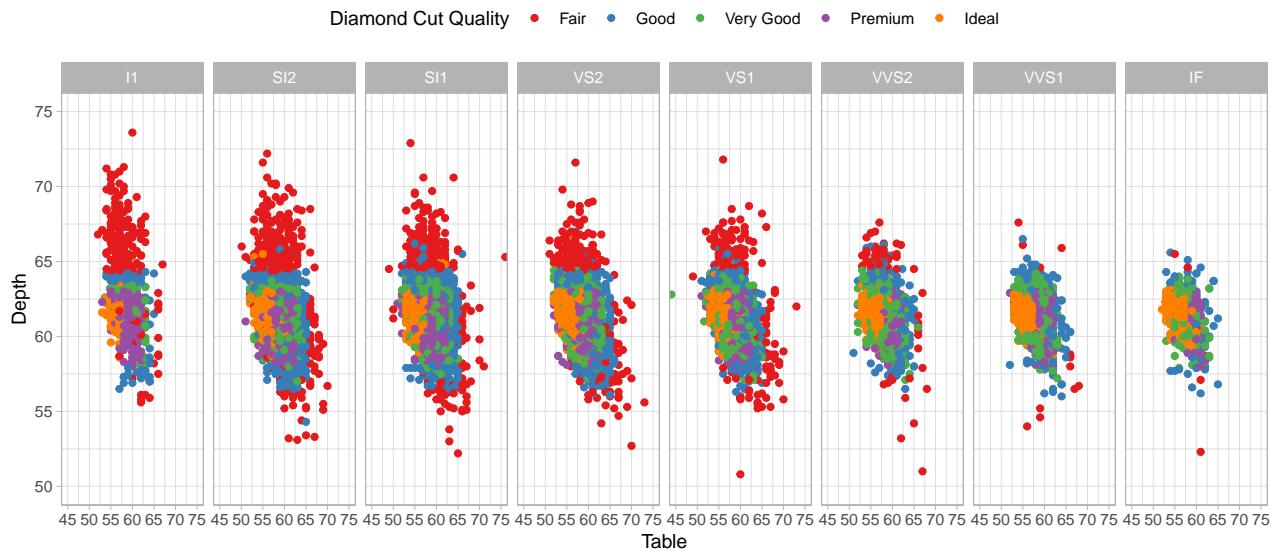
In the following plot, we use facets to redraw our plot for each of the clarity levels. Furthermore, the legend is placed on top to create more space.

```
ggplot(diamonds) +
  geom_point(aes(table, depth, col = cut)) +
  labs(title = "Comparing table and depth of diamonds",
       subtitle = "What combination of table and depth gives good quality diamonds?",
       x = "Table",
       y = "Depth") +
  theme_light() +
  coord_cartesian(xlim = c(45,75), ylim = c(50, 75)) +
  scale_x_continuous(breaks = seq(45,75,5)) +
  scale_y_continuous(breaks = seq(50,75,5)) +
  scale_color_brewer(palette = "Set1", name = "Diamond Cut Quality") +
  facet_grid(. ~ clarity) +
  theme(legend.position = "top")
```

An alternative, mostly suitable for comparisons along one variable, is `facet_wrap`. Instead of making one row (like `facet_grid` does), it will order the plots in a grid with a specified number of columns or rows. Below, we ordered them in 3 columns.

### Comparing table and depth of diamonds

What combination of table and depth gives good quality diamonds?



```
ggplot(diamonds) +
  geom_point(aes(table, depth, col = cut)) +
  labs(title = "Comparing table and depth of diamonds",
       subtitle = "What combination of table and depth gives good quality diamonds?",
       x = "Table",
       y = "Depth") +
  theme_light() +
  coord_cartesian(xlim = c(45,75), ylim = c(50, 75)) +
  scale_x_continuous(breaks = seq(45,75,5)) +
  scale_y_continuous(breaks = seq(50,75,5)) +
  scale_color_brewer(palette = "Set1", name = "Diamond Cut Quality") +
  facet_wrap(~ clarity, ncol = 3) +
  theme(legend.position = "top")
```

### 3.5.2 Combining multiple geometric layers

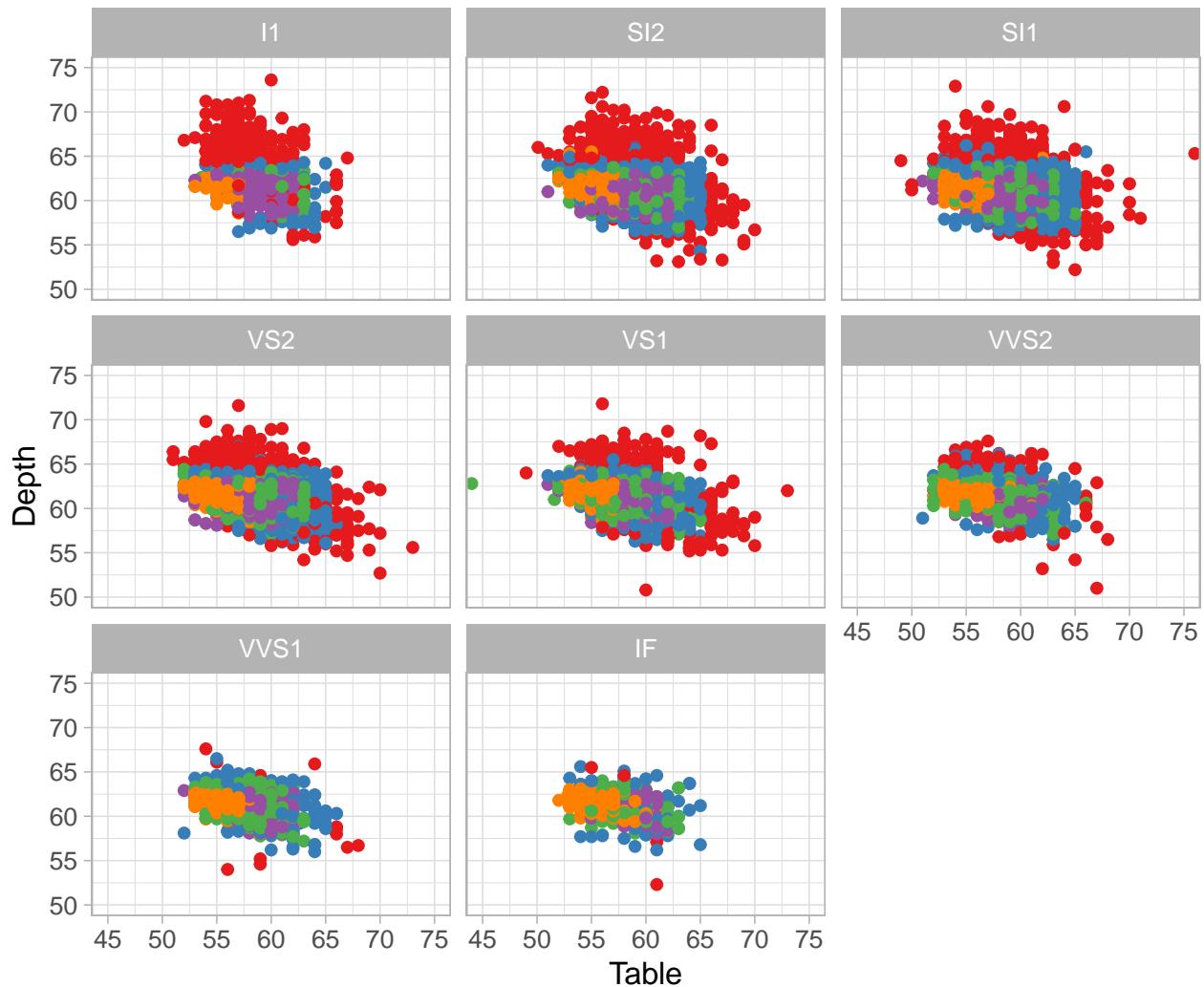
So far, we only used one geometric layer at a time. However, it is perfectly possible to combine different layers. For instance, we can use the `geom_text` label to add data labels to a bar plot. `Geom_text` is a geometric layer which we can use, indeed, to put text in a graph. We didn't see `geom_text` before, however, its working will be straightforward, based on all the things we already know. We build further upon the last graph, which we have given a slightly nicer appearance

```
filter(movies, year == 2004, budget > 100000000) %>%
  ggplot() +
```

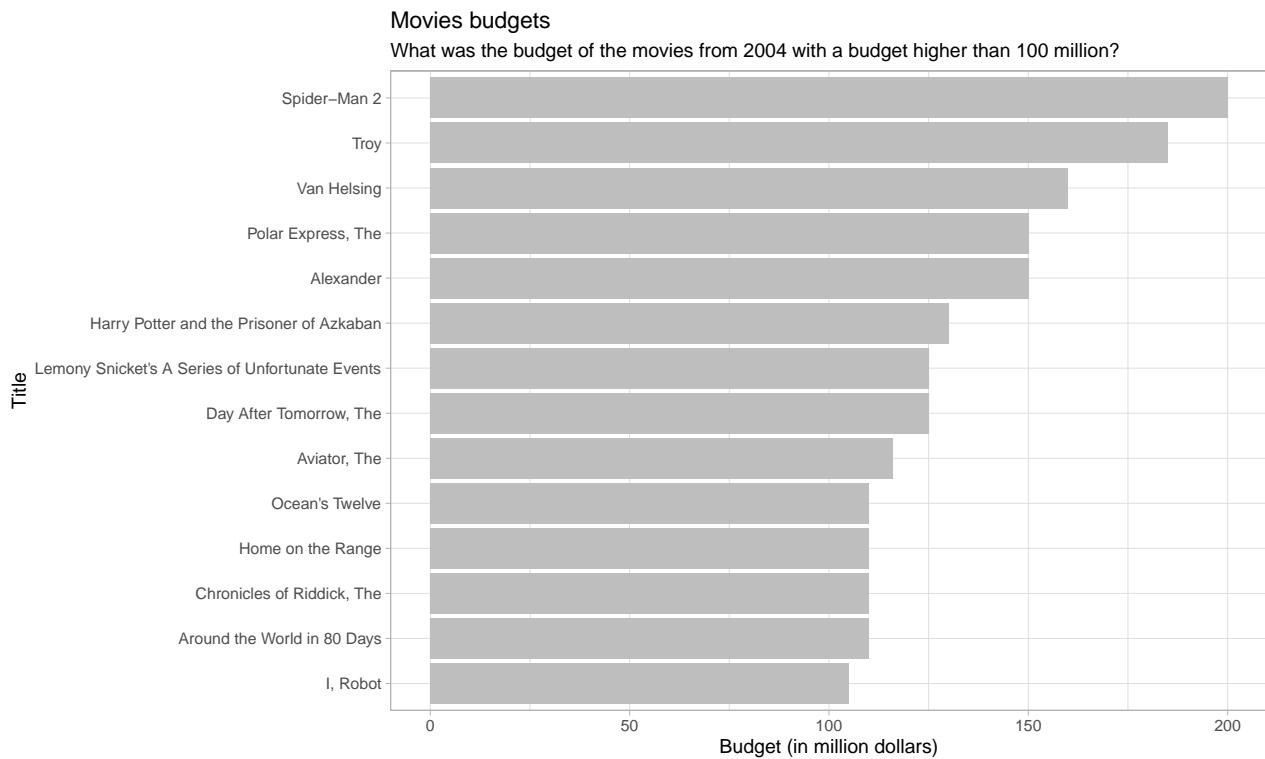
## Comparing table and depth of diamonds

What combination of table and depth gives good quality diamonds?

Diamond Cut Quality • Fair • Good • Very Good • Premium • Ideal



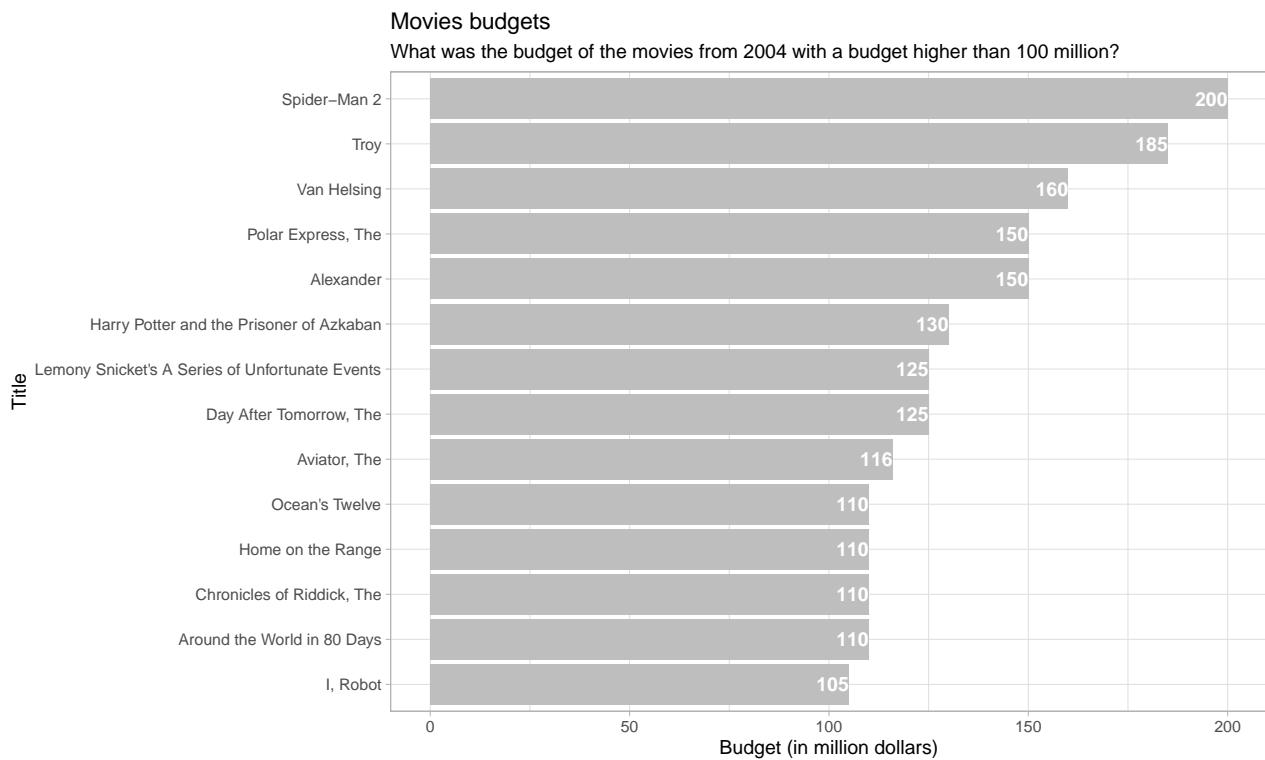
```
geom_col(aes(reorder(title, budget), budget/1000000), fill = "grey") +
coord_flip() +
labs(title = "Movies budgets",
subtitle = "What was the budget of the movies from 2004 with a budget higher than 100 million?",
x = "Title",
y = "Budget (in million dollars)") +
theme_light()
```



Note that we changed the y-aesthetic to budget/1000000, such that the values are in millions. Furthermore, it is important to remark that, since we used coord\_flip, our labels set in labs also switched. Thus the x-label appears on the y-axis and the y-label on the x-axis. This is exactly what we would prefer, as removing coord\_flip in the future won't mess up the labels.

We would now like to put the exact number of millions on top of the bars. In order to do this, we add geom\_text, and give it the same mapping for x and y as the geom\_col layer. Furthermore, we add a mapping for the label, i.e. the text to be displayed. We give the text a bold fontface and a white color. Finally, setting hjust to 1 will make sure that the textlabels are horizontally aligned to the left. This makes sure that the text is entirely within the bars, and doesn't fall out of them.

```
filter(movies, year == 2004, budget > 100000000) %>%
  ggplot() +
  geom_col(aes(reorder(title, budget), budget/1000000), fill = "grey") +
  geom_text(aes(reorder(title, budget), budget/1000000,
                label = budget/1000000), color = "white", fontface = "bold", hjust = 1) +
  coord_flip() +
  labs(title = "Movies budgets",
       subtitle = "What was the budget of the movies from 2004 with a budget higher than 100 million?",
       x = "Title",
       y = "Budget (in million dollars)") +
  theme_light()
```



The addition of the second geom layer seemed a little bit cumbersome, as we needed to repeat the mapping for x and y, which was made even worse because it involved the reorder function and the division by a million. Surely, there is a better way to do this? We call it aes-inheritance.

### 3.5.3 Aes-inheritance

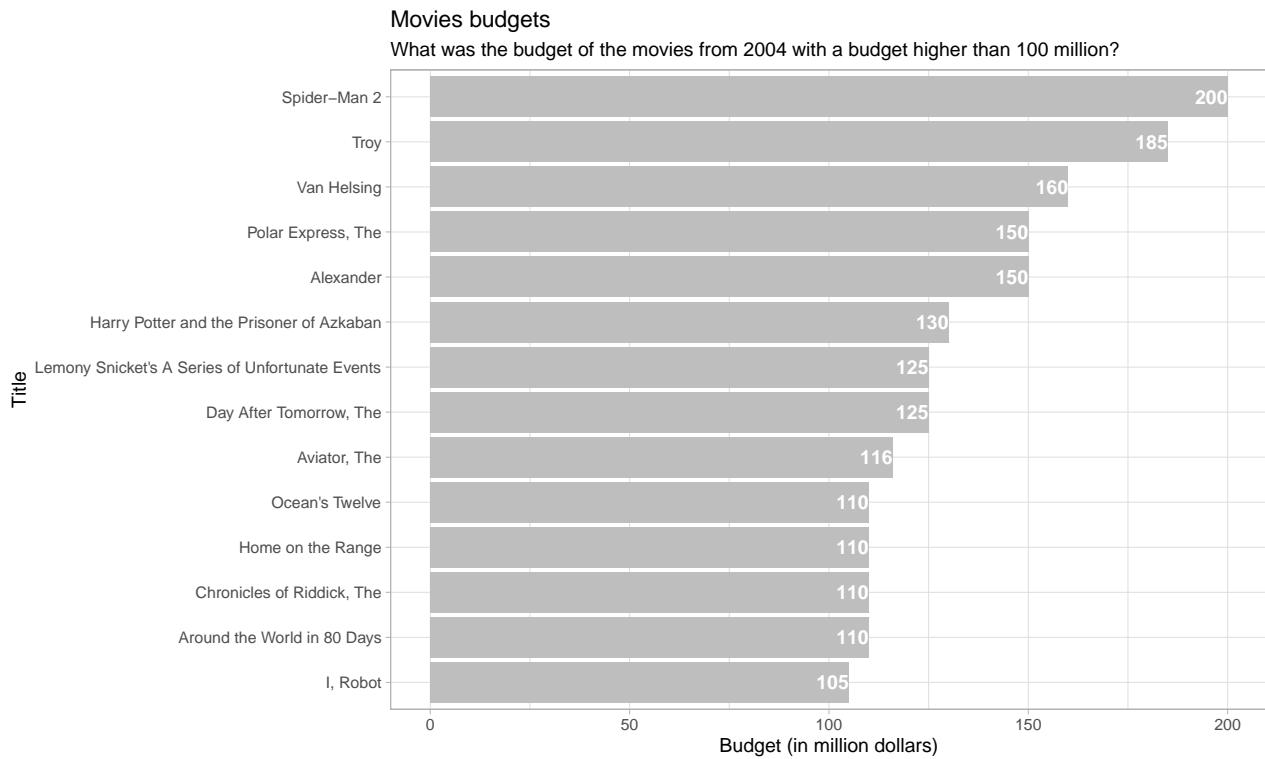
Aes-inheritance, or inheritance of the aes-mapping, means that each of the geom layers which is added to a ggplot call *inherits* the mapping

which is specified in that `ggplot` function call. As you may remember from the very beginning, an `aes-mapping` can be placed in both `geom` layers as in `ggplot`. In the end, we learn there is a little but importance difference.

When different layers have (part of) a mapping in common, it is best practice to move this part to the `ggplot` call. As such, you don't have to repeat this in any of the layers which use it. And, if any of the layers doesn't use this mapping, you can easily overwrite it by specifying a new mapping within that layer. Let's look at an example.

The previous plot we made can more easily be made as follows:

```
filter(movies, year == 2004, budget > 100000000) %>%
  ggplot(aes(reorder(title, budget), budget/1000000)) +
  geom_col(fill = "grey") +
  geom_text(aes(label = budget/1000000), color = "white", fontface = "bold", hjust = 1) +
  coord_flip() +
  labs(title = "Movies budgets",
       subtitle = "What was the budget of the movies from 2004 with a budget higher than 100 million?",
       x = "Title",
       y = "Budget (in million dollars)") +
  theme_light()
```



By moving the mapping for `x` and `y` to `ggplot`, there is no mapping

needed in geom\_col, and only a mapping for label in geom\_text.  
Both geom layers inherit the other part of the mapping for the ggplot  
function call. Truly, this seems far more efficient!

### 3.6 *Background material*

By now you already master a great deal of plotting with ggplot2. You have both learned how to use different geometric layers to represent data, how to combine them, how to use facets, how to make your code more efficient using aes-inheritance, and last but not least, how to give your plot a nice appearance. Congratulations!

If you are eager to learn even more, you might have a look at the following background materials:

- The ggplot Cheat Sheet, which provides an overview of all basic functionality in the ggplot2 package.
- The ggplot documentation
- An even more comprehensive ggplot2 tutorial

# 4

## *Descriptieve statistieken*

### *4.1 Beschrijvende statistieken versus exploratieve plots*

- Plots zijn vooral sterk om patronen in de data te visualiseren.
- Plots zijn minder geschikt om de ‘sterkte’ of ‘grootte’ van een patroon uit te drukken.
- Beschrijvende statistieken laten dit wel toe aangezien aspecten van de patronen in een exploratieve plot in exacte getallen worden gegoten.
- Er kunnen hoofdzakelijk 3 soorten beschrijvende statistieken worden onderscheiden:
  - Centrummaten
  - Spreidingsmaten
  - Associatiematen
- Centrummaten en spreidingsmaten zijn univariate statistieken en hebben als doel de verdeling van 1 variabele data samen te vatten in 2 cijfers.
- Associatiematen zijn typisch bivariate statistieken en hebben als doel de samenhang tussen twee variabelen samen te vatten.

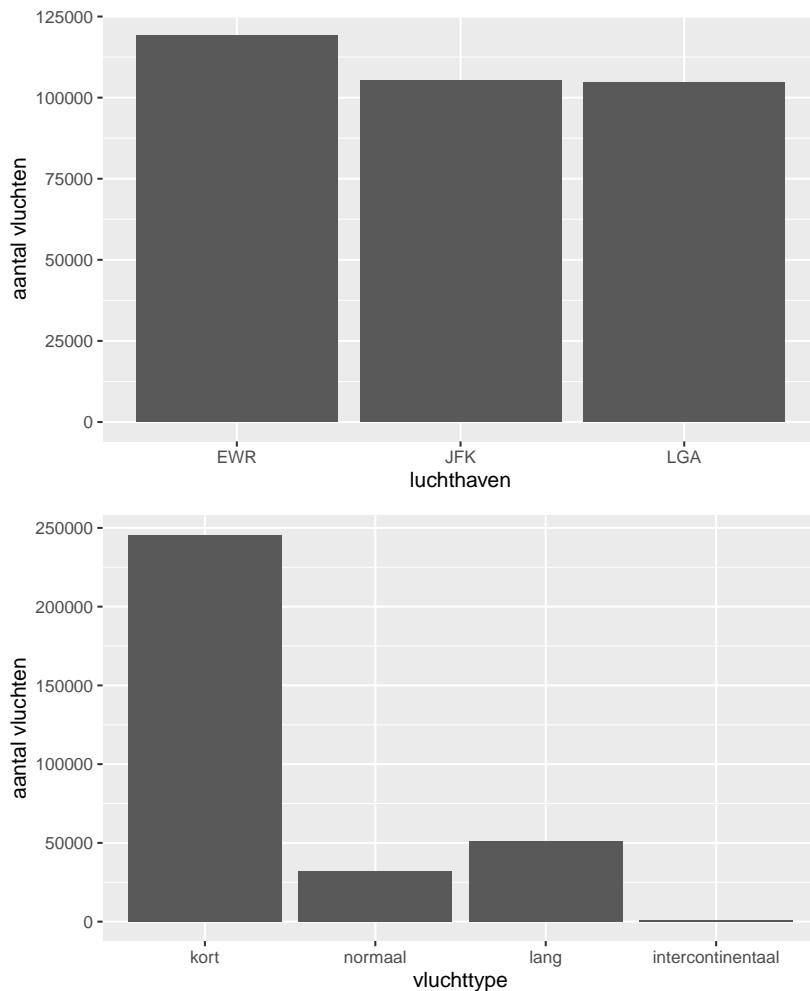
### *4.2 Notatie*

- $n$ : aantal observaties.
- $X, Y$ : variabelen.
- $x_i, y_i$ : de waarden voor variabelen  $X$  en  $Y$  voor observatie  $i$ .
- $x_{(i)}$ : de  $i$ -de waarde voor  $X$  na rangschikking van klein naar groot.

### 4.3 Data

### 4.4 Univariate statistieken

#### 4.4.1 Categorische variabele



#### Frequentietabel

- De absolute frequentie  $f$  geeft aan hoe vaak een waarde voorkomt.
- De relatieve frequentie  $f/n$  geeft aan welk aandeel deze frequentie heeft in het totaal aantal elementen  $n$ .
- De cumulatieve frequentie  $F_n(x)$  van een bepaalde waarde  $x$  geeft aan hoeveel observaties kleiner zijn dan of gelijk zijn aan  $x$ .
- De cumulatieve relatieve frequentie  $F_n(x)/n$  van een bepaalde waarde  $x$  geeft aan hoeveel percent van de observaties kleiner zijn dan of gelijk zijn aan  $x$ .
- Een frequentietabel laat voor alle mogelijke waarden van een cat-

egorische variabele de absolute en relatieve frequentie zien (zowel normaal als cumulatief).

- Een frequentietabel laat zien waar een bepaalde waarde zich precies in de verdeling bevindt en hoe uitzonderlijk het is een specifieke waarde in de data te zien (of een waarde groter/kleiner dan) .

luchthaven	freq	rel_freq	cum_freq	cum_rel_freq
EWR	119282	0.36	119282	0.36
JFK	105230	0.32	224512	0.68
LGA	104662	0.32	329174	1.00

Table 4.1: Aantal vluchten per luchthaven

### Centrummaten

- Modus
  - Meest voorkomende waarde.
  - Enige centrummaat voor nominale variabele.
  - Ook bruikbaar voor ordinale variabele.
  - Een variabele kan meerdere modi hebben.
  - De modus is robuust tegen uitschieters.
  - De modus kan je aflezen als de eerste rij in een frequentietabel als je deze ordent van de meest voorkomende tot de minst voorkomende waarde.
- Mediaan
  - De middelste waarde na rangschikken van de gegevens.
  - Voor ordinale variabelen definiëren we de mediaan aan de hand van de relatieve cumulatieve frequentie. De mediaan is de kleinste waarde waar 50% van de observaties kleiner dan of gelijk aan is.
  - De mediaan is robuust tegen uitschieters.

vluchtype	freq	rel_freq	cum_freq	cum_rel_freq
kort	245666	0.75	245666	0.75
normaal	31813	0.10	277479	0.85
lang	50980	0.15	328459	1.00
intercontinentaal	715	0.00	329174	1.00

Table 4.2: Aantal vluchten per vluchtype

variabele	mediaan
vluchtype	kort

Table 4.3: Centrummaten voor vluchtype

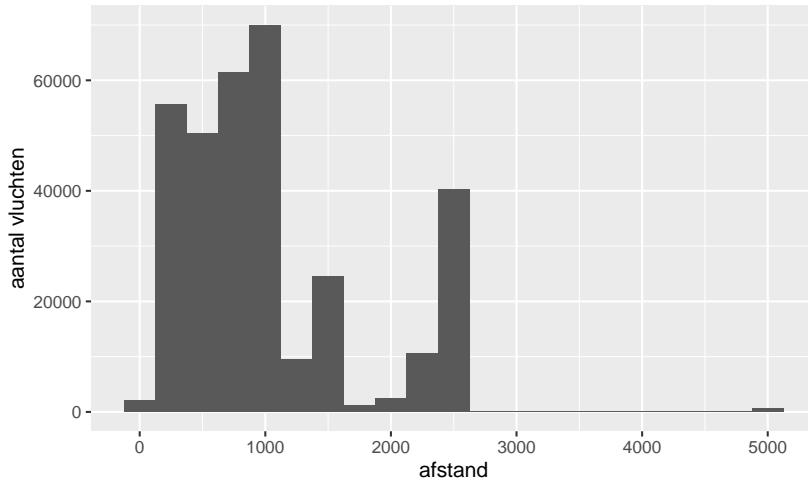
### *Spreidingsmaten*

- Kwantilen.
  - Kwantilen (of percentielen) zijn gebaseerd op de cumulatieve relatieve frequentie.
  - Het  $p\%$  kwantiel is de kleinste waarde waar  $p\%$  van de observaties kleiner dan of gelijk aan is.
  - Het 50% kwantiel komt overeen met de mediaan.
  - Veel voorkomende kwantilen om de spreiding van de data weer te geven zijn het 25% en 75% kwantiel.

Table 4.4: Kwantilen voor vluchtype

variabele	Q25	Q50	Q75
vluchtype	kort	kort	normaal

#### 4.4.2 Continue variabele



### *Centrummaten*

- Modus
  - Vaak minder bruikbaar bij een continue variabelen omdat iedere waarde zeer weinig voorkomt. Bijgevolg zijn er vaak zeer veel modi met telkens slechts enkele observaties.
- Mediaan
  - De middelste waarde na rangschikking van de gegevens.
  - In geval van een oneven aantal observaties, komt dit overeen met  $x_{\frac{(n+1)}{2}}$ .

- In geval van een even aantal observaties zijn er twee ‘middelste’ observaties en is de mediaan gelijk aan  $\frac{1}{2}(x_{\frac{n}{2}} + x_{\frac{n}{2}+1})$
- De mediaan is robuust tegen uitschieters.
- (Rekenkundig) Gemiddelde
  - $\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$
  - Het gemiddelde is gevoelig voor uitschieters.
  - Dit is de centrummaat die mensen intuïtief selecteren indien mogelijk.

Table 4.5: Afstand (centrummaten)

variabele	gemiddelde	mediaan
afstand	1026.98	820

### Spreidingsmaten

- Kwantilen
- Bereik
  - Dit is het verschil tussen de grootste en kleinste waarde.
  - Zeer gevoelig voor uitschieters.
  - Is slechts gebaseerd op 2 observaties en bevat dus weinig informatie. Hiermee bedoelen we dat de spreiding van 2 variabelen sterk kan verschillen terwijl ze toch hetzelfde bereik hebben.
- Interkwartielafstand (IQR)
  - Dit is het verschil tussen Q75 en Q25.
  - Zelfde principe als het bereik, maar minder gevoelig voor uitschieters.
  - IQR is ook slechts gebaseerd op 2 observaties.
- Gemiddelde absolute afwijking (average absolute deviation)
  - Dit is de gemiddelde afwijking ten opzichte van het gemiddelde over alle observaties.
  - $\frac{1}{n} \sum_{i=1}^n |x_i - \bar{x}|$ .
- Variantie
  - $s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$ .
  - Vergelijkbaar met gemiddelde absolute afwijking, maar nu wordt het kwadraat gebruikt om te voorkomen dat de verschillen ten opzichte van het gemiddelde elkaar opheffen.
  - Vanuit analytisch standpunt is deze spreidingsmaat interessanter (geen absolute waarden, waardoor afgeleiden bijvoorbeeld eenvoudiger worden om te berekenen).

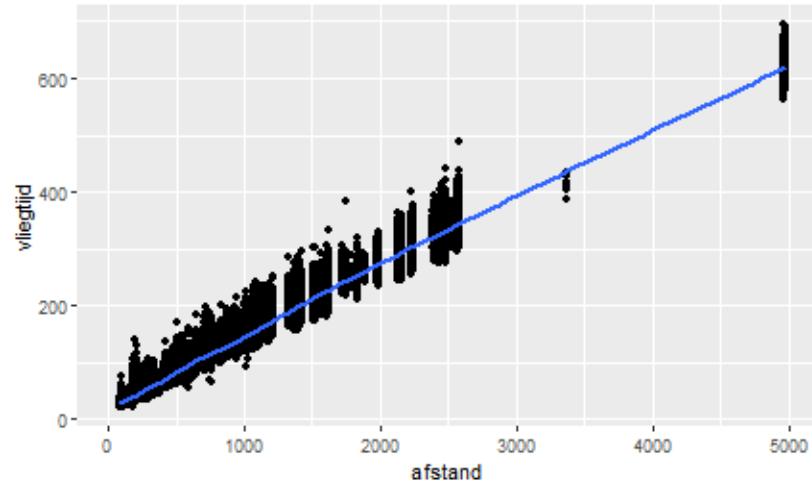
- Wel gevoelig voor uitschieters en door het kwadraat wordt het effect van deze uitschieters ook nog eens vergroot.
- De wortel van de variantie wordt de standaardafwijking genoemd. De standaardafwijking heeft het voordeel dat het dezelfde eenheid uitgedrukt wordt als de oorspronkelijke data.
- Median Absolute Deviation (MAD)
  - Dit is de middelste afwijking ten opzichte van de mediaan over alle observaties.
  - $MAD = \text{median}(|X_i - \text{median}(X)|)$ .
  - Deze maatstaf is robuster tegen outliers.

variabele	minimum	Q25	Q50	Q75	maximum	bereik	IQR	var	sd
afstand	17	502	820	1372	4983	4966	870	542630.2	736.6344

Table 4.6: Afstand (spreidingsmaten)

## 4.5 Bivariate statistieken

### 4.5.1 Continu versus Continu



### Correlatie

- Covariantie
  - $\text{cov}(x, y) = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$ .
  - Bij een positieve associatie tussen twee variabelen zal de covariante positief zijn.
  - Bij een negatieve associatie tussen twee variabelen zal de covariante negatief zijn.

- De covariantie is echter afhankelijk van de maateenheid van de variabelen, waardoor ze weinig bruikbaar is om de sterkte van de associatie weer te geven.
- Pearson correlatiecoëfficiënt
  - Herschaalt de covariantie naar de schaal  $[-1, 1]$
  - Laat toe om de sterkte van een associatie te evalueren.
  - $r(x, y) = \frac{\text{cov}(x, y)}{s_x s_y}$
  - $r(x, y) = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$
  - Meet **lineaire** associatie tussen 2 variabelen.
  - Twee variabelen kunnen positief geassocieerd zijn, maar in een niet-lineaire wijze, waardoor de correlatiecoëfficiënt naar nul gaat.
  - Meest gebruikelijke correlatiecoëfficiënt voor continue variabelen.
  - Daarom best altijd samen met een puntenwolk bekijken.
- Spearman's rangcorrelatiecoëfficiënt.
  - Zelfde principe als Pearson's, maar dan gebaseerd op de rangorde van de waarden in plaats van de waarden zelf.
  - $r_i$ : rangorde van waarde  $x_i$ . Bijvoorbeeld  $r_i = 4$  betekent dat de waarde  $x_i$  de vierde kleinste waarde is.
  - $s_i$ : rangorde van waarde  $y_i$ .
  - $\rho(x, y) = \frac{\sum_{i=1}^n (r_i - \bar{r})(s_i - \bar{s})}{\sqrt{\sum_{i=1}^n (r_i - \bar{r})^2} \sqrt{\sum_{i=1}^n (s_i - \bar{s})^2}}$
  - Meet associatie tussen 2 variabelen, dus niet specifiek lineaire associatie.
- Kendall's correlatiecoëfficiënt
  - Ook wel Kendall's tau genoemd.
  - De methode is gebaseerd door alle mogelijke observatieparen  $(x_i, y_i)$  en  $(x_j, y_j)$  te bestuderen.
  - Net als Spearman's aanpak gebaseerd op rangorde  $(r_i, s_i)$  en niet de feitelijke waarden.
  - Indien  $r_i > r_j$  en  $s_i > s_j$  (of  $r_i < r_j$  en  $s_i < s_j$ ) dan zijn observaties  $i$  en  $j$  concordant.
  - Indien  $r_i > r_j$  en  $s_i < s_j$  (of  $r_i < r_j$  en  $s_i > s_j$ ) dan zijn observaties  $i$  en  $j$  discordant.
  - Notatie:  $C$  en  $D$  zijn respectievelijk het aantal concordante en discordante paren.
  - $\tau = \frac{C-D}{\frac{1}{2}n(n-1)}$
  - Net als Spearman's correlatiecoëfficiënt, focust Kendall's tau op de associatie (positief of negatief) en niet specifiek op lineaire associatie.
  - Het nadeel van Kendall's tau is dat je alle observatieparen moet bestuderen en het aantal kan snel exploderen bij veel observaties.

Immers het aantal paren is  $\frac{n!}{2!(n-2)!}$ . Hierdoor kan je Kendall in de praktijk niet gebruiken als je veel observaties hebt.

variabelenpaar	pearson	spearman
afstand-vliegtijd	0.99	0.98

Table 4.7: Correlatie tussen afstand en vliegtijd

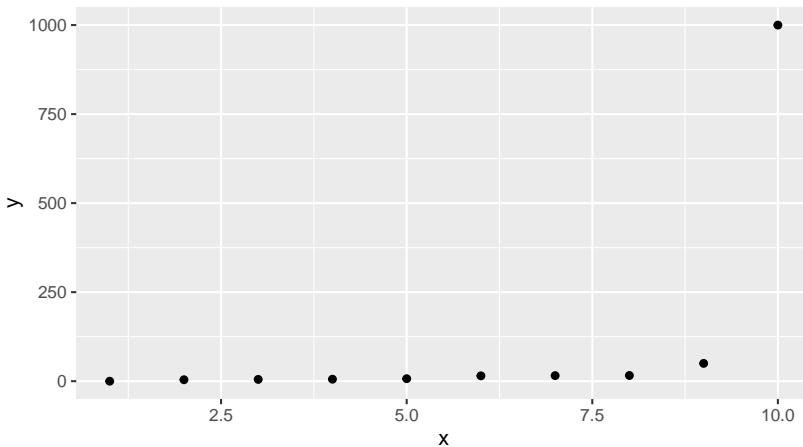
### Vergelijking correlatiecoëfficiënten

- Rangcorrelatiecoëfficiënten meten associatie, terwijl Pearson correlatiecoëfficiënt **lineaire** associatie meet!

Table 4.8: Fictieve dataset

x	y
1	0.0
2	4.0
3	5.0
4	5.5
5	7.0
6	15.0
7	15.6
8	16.0
9	50.0
10	1000.0

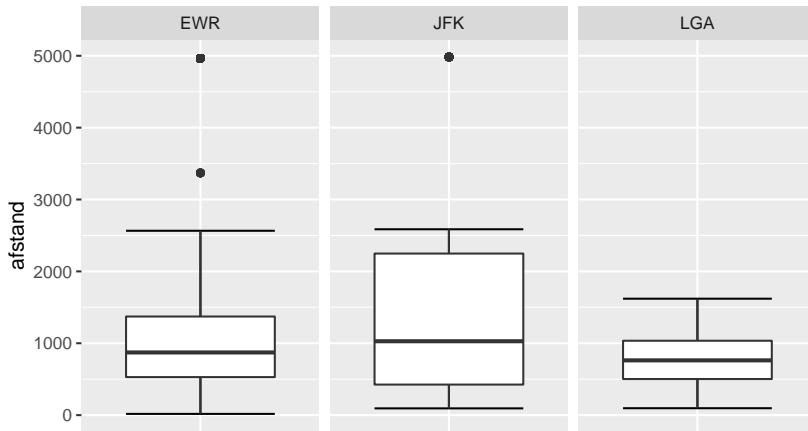
Fictieve dataset ter illustratie correlatiecoëfficiënten



variabelenpaar	pearson	spearman	kendall
x-y	0.55	1	1

Table 4.9: Correlatiecoëfficiënten fictieve dataset

#### 4.5.2 Categorisch versus Continu



#### Univariate statistieken per categoriewaarde

- Je toont de relevante centrum- en spreidingsmaten voor de afhankelijke continue variabele per waarde van de onafhankelijke categorische variabele.

	luchthaven	gemiddelde	mediaan
EWR	1049.58	872	
JFK	1247.16	1028	
LGA	779.84	762	

Table 4.10: Afstand-Luchthaven  
(centrummaten)

luchthaven	var	min	Q25	Q50	Q75	max	bereik	IQR	sd
EWR	536177.0	17	529	872	1372	4963	4946	843	732.2411
JFK	842460.4	94	425	1028	2248	4983	4889	1823	917.8564
LGA	138132.3	96	502	762	1035	1620	1524	533	371.6615

Table 4.11: Afstand-Luchthaven  
(spreidingsmaten)

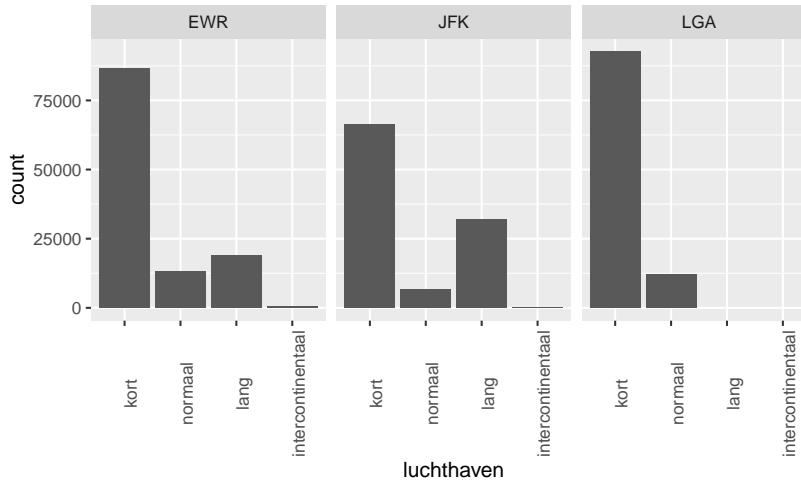
#### Correlatie

- Enkel toepasbaar als de categorische variabele ordinaal is.
- Pearson's correlatiecoëfficiënt kan je NIET toepassen.
- Spearman rangcorrelatiecoëfficiënt ( $\rho$ ).
- Kendall's rangcorrelatiecoëfficiënt ( $\tau$ ) kan theoretisch wel toegepast worden, maar is in de praktijk vaak niet haalbaar.

Table 4.12: Correlatie tussen vluchtype en vliegtijd

variabelenpaar	spearman
vluchtype-vliegtijd	0.76

#### 4.5.3 Categorisch versus Categorisch



#### Univariate statistieken per categoriewaarde

- Je toont de relevante centrum- en spreidingsmaten voor de afhankelijke categorische variabele per waarde van de onafhankelijke categorische variabele. Dit is enkel mogelijk indien de afhankelijke categorische variabele ordinaal is, waarbij je minimum, mediaan, maximum en kwantilen kan berekenen.

Table 4.13: Centrummaten voor vluchtype-luchthaven

luchthaven	variabele	mediaan
EWR	vluchtype	kort
JFK	vluchtype	kort
LGA	vluchtype	kort

Table 4.14: Kwantilen voor vluchtype-luchthaven

luchthaven	variabele	Q25	Q50	Q75
EWR	vluchtype	kort	kort	normaal
JFK	vluchtype	kort	kort	lang
LGA	vluchtype	kort	kort	kort

#### Contingentietabellen

Een andere mogelijkheid is het maken van 2-dimensionale frequentietabellen, ookal contingentietabellen genoemd. Meer info hierover lees

je in de tutorial.

### *Referenties*

1. Tekst Beleidsstatistiek: Hoofdstukken 1 en 2 en secties 4.2 en 4.3  
(Blackboard)
2. Spearman's rangcorrelatiecoëfficiënt
3. Kendall's rangcorrelatiecoëfficiënt
4. Spearman versus Kendall's correlatiecoëfficiënt



# 5

## *Tutorial - Descriptieve statistieken*

### *5.1 Before you start*

Before you start this tutorial, make sure to have installed the packages `dplyr`, `tidyr`, `ggcorrplot`, and `ggplot2`, **if you haven't already done so**. You can load these using the `library` function. If you need to install some of them, use `install.packages` first.

```
library(dplyr)  
library(tidyr)  
library(ggplot2)
```

Additionally, you might also want to use the following packages. They will appear useful at some point in this tutorial, but are not strictly necessary.

```
library(pander)  
library(forcats)
```

This tutorial will use the `mpg` data set, containing information on 234 different cars. The `mpg.RDS` file is distributed with this tutorial.

```
mpg <- readRDS("mpg.RDS")
```

The data set contains the following variables:<sup>1</sup>

Variable	Description
manufacturer	The manufacturer
model	The model name
displ	Engine displacement, in liters
year	Year of manufacture
cyl	Number of cylinders
trans	Type of transmission ( <i>koppeling</i> )
drv	f = front-wheel drive, r = rear wheel drive, 4 = 4wd

<sup>1</sup> Although it might make certain analyses more interesting when you are familiar with the meaning of the different variables, no specific knowledge about cars is required in order to complete this tutorial.

Variable	Description
cty	City miles per gallon of fuel
hwy	Highway miles per gallon of fuel
fl	Fuel type (c,d,e,p,r)
class	Type of car (2seater, compact, midsize, minivan, pickup, subcompact, suv)

## 5.2 Introduction

The aim of this tutorial is to perform both univariate and bivariate analysis with the `dplyr` package. Furthermore, the `tidyverse` and `ggcorrplot` packages will be used in a limited number of cases for extra support.

The univariate analysis will be done for both categorical and continuous variables. The bivariate analysis will be done for each of the following pairs: continuous-categorical, continuous-continuous and categorical-categorical. Next to a *numerical* analysis using functions from the aforementioned packages, the analyses will be accompanied by appropriate graphs made with `ggplot2`. A basic understanding of `ggplot2` is required.

The tutorial is structured as follows:

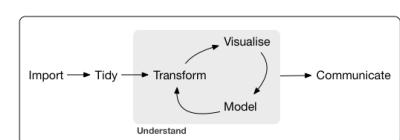
- Some general `dplyr` functions which are helpful
- Univariate analysis of a continuous variable
- Bivariate analysis of a continuous variable with respect to a categorical variable
- Univariate analysis of a categorical variable
- Bivariate analysis of a continuous variable with respect to another continuous variable
- Bivariate analysis of a categorical variable with respect to another categorical variable

## 5.3 Helpful `dplyr` functions

Before turning to analyzing our data set, there are a few helpful auxiliary functions contained in `dplyr` which we can use to handle and analyze our data. The first one is `tbl_df`, pronounced as *tibble data.frame*.

### 5.3.1 Tibbles

Given a `data.frame` `df`, `tbl_df(df)` will turn it into a *tibble*. A tibble is a special kind of `data.frame` used by `dplyr` and other packages of the `tidyverse`.<sup>2</sup> When a `data.frame` is turned into a tibble its class will change.



<sup>2</sup> The `tidyverse` is a set of packages for data science that work in harmony because they share common data representations and API design. The `tidyverse`-package is designed to make it easy to install and load core packages from the `tidyverse` in a single command. The `tidyverse`

```

class(mpg)
## [1] "data.frame"

mpg <-tbl_df(mpg)
class(mpg)

## [1] "tbl_df"     "tbl"        "data.frame"

```

You can see that the `mpg` object now has three different `class` labels. It is still a `data.frame`, but a special kind of `data.frame`, i.e. a `tibble` `data.frame`.

The difference between an ordinary `data.frame` and a `tibble` `data.frame` is most noticeable when printing (large) `data.frames`. Just try to print the data set with the following two lines in the console.

```

#These lines are not executed here
#you can try them in the console.
as.data.frame(mpg)
mpg

```

Do you notice the difference? When printing an ordinary `data.frame`, an abundance of observations will be printed in the console. As a result you'll have to scroll back up to see the variable names. Even worse, when the columns don't fit on a single page, each observation will be scattered among different lines, making the output quite unreadable.<sup>3</sup>

However, when printing a `tibble` `data.frame`, only the first 10 rows will be printed by default, and variables that don't fit within the width of the page or console will be hidden. This makes the printed data set much more readable and our console less messed up.

The downside of this approach is that, sometimes, you want to see more observations or more variables, without turning your `tibble` back into a `data.frame`. You can solve this by explicitly using the `print` function and setting the arguments `n` and `width`. The following line will print all rows and columns, by setting both arguments to `Inf`, which stands for infinity.<sup>4</sup> You can also use other values to print varying numbers of columns and rows. Later we will see how we can print specific rows and columns.

```

#This line is not executed here - you can try it in the console.
print(mpg, n = Inf, width = Inf)

```

Except for printing, `tibbles` allow for some easier manipulations compared the a normal `data.frame` when you install the package `tibble`. However, this falls outside of the scope of this tutorial. Furthermore, you will see that the output of `dplyr` functions discussed in

<sup>3</sup> Since we turned `mpg` into a `tibble` `data.frame`, we can only print it as if it were a normal `data.frame` by using the `as.data.frame` function. However, we don't update the object since we don't store it using the `<-`.

<sup>4</sup> Note the capital letter I

the remainder of this tutorial are automatically tibbles. This means that often you do not explicitly need to use `tbl_df`. So, for now, it suffices to understand the difference.

### 5.3.2 Glimpse

A second useful function is the `glimpse` function. This function is the `dplyr`-alternative to the well-known `str` function for base R, and is thus helpful for a first inspection of the data set at hand. It will give a slightly different output compared to `str`, and is by some perceived as more neat.

```
str(mpg)

## Classes 'tbl_df', 'tbl' and 'data.frame': 234 obs. of 11 variables:
##   $ manufacturer: Factor w/ 15 levels "audi","chevrolet",...: 1 1 1 1 1 1 1 1 1 1 ...
##   $ model       : Factor w/ 38 levels "4runner 4wd",...: 2 2 2 2 2 2 2 3 3 3 ...
##   $ displ        : num  1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
##   $ year         : int  1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
##   $ cyl          : Ord.factor w/ 4 levels "4"<"5"<"6"<"8": 1 1 1 1 3 3 3 1 1 1 ...
##   $ trans        : Factor w/ 10 levels "auto(av)","auto(13)",...: 4 9 10 1 4 9 1 9 4 10 ...
##   $ drv          : Factor w/ 3 levels "4","f","r": 2 2 2 2 2 2 2 1 1 1 ...
##   $ cty          : int  18 21 20 21 16 18 18 18 16 20 ...
##   $ hwy          : int  29 29 31 30 26 26 27 26 25 28 ...
##   $ fl           : Factor w/ 5 levels "c","d","e","p",...: 4 4 4 4 4 4 4 4 4 4 ...
##   $ class        : Factor w/ 7 levels "2seater","compact",...: 2 2 2 2 2 2 2 2 2 2 ...

glimpse(mpg)

## Observations: 234
## Variables: 11
##   $ manufacturer <fct> audi, audi, audi, audi, audi, audi, audi, audi, audi, ...
##   $ model       <fct> a4, a4, a4, a4, a4, a4, a4 quattro, a4 quattro, a4...
##   $ displ        <dbl> 1.8, 1.8, 2.0, 2.0, 2.8, 2.8, 3.1, 1.8, 1.8, 2.0, 2.0, ...
##   $ year         <int> 1999, 1999, 2008, 2008, 1999, 1999, 2008, 1999, 1999, ...
##   $ cyl          <ord> 4, 4, 4, 4, 6, 6, 6, 4, 4, 4, 6, 6, 6, 6, 6, 8, ...
##   $ trans        <fct> auto(15), manual(m5), manual(m6), auto(av), auto(15), ...
##   $ drv          <fct> f, f, f, f, f, f, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, ...
##   $ cty          <int> 18, 21, 20, 21, 16, 18, 18, 18, 16, 20, 19, 15, 17, 17...
##   $ hwy          <int> 29, 29, 31, 30, 26, 26, 27, 26, 25, 28, 27, 25, 25, 25...
##   $ fl           <fct> p, ...
##   $ class        <fct> compact, compact, compact, compact, compact, compact, ...
```

### 5.3.3 %>%

Another helpful function is a very special one, and is called *the piping symbol*.<sup>5</sup> The piping symbol consist of a greater-than sign, preceded



and followed by a %-sign. The piping symbol can be used to *pipe* different statements together. You can compare it with pipes in a waterworks system. Within a water purification station, pipes will transport the water from one treatment station to the next and eventually to the households. Here, we use the piping symbol to bring our data from one manipulation to the next. Consider this very simple example.

```
#This line is not executed here - you can try it in the console.
head(mpg)
```

We can paraphrase this line as: “Take the head (i.e. the first 6 rows) of the data set `mpg`.”

When we use the piping symbol, the first argument of a function can be placed *before* the function call instead of inside it. The last line of code is therefore equivalent to the following line.<sup>6</sup>

```
#This line is not executed here - you can try it in the console.
mpg %>% head()
```

We can now say: “We take the data set `mpg` and then take the head of it.” This sentence seems more natural, as it is very easy to extend it with the further steps you are going to take.

As pipes bring water from one place to the next in a waterworks system, the piping symbol brings our data from one place to the next, in this case the `head` function. At this point, it might seem ridiculous to do this, but as we will see very soon, this symbol comes in very handy.

As another example, the next two hypothetical statements are equivalent

```
f(x,y,z)
x %>% f(y,z)
```

And when we have two functions, say `f` and `g` which we called in a nested way, i.e. one within the other, we can do the following.

```
f(g(x,y),z)
# Bring the first argument of f to the front
g(x,y) %>% f(z)
# Bring the first argument of g to the front
x %>% g(y) %>% f(z)
```

Thus, this command takes `x`, performs function `g` with argument `y` and then the result is given to function `f` with second argument `z`. This is much more easy to read than the original line, were we performed function `f` on the result of function `g` on `x` and `y`, and using `z` as second argument to `f`.

However, that’s enough of abstract concepts. It’s time to do something with our data and put this symbol to good use.

<sup>6</sup> You can even omit the brackets of the functions when there are no arguments left. As such `mpg %>% head` would work just fine. However, do not confuse this with adding layers to a `ggplot` call. Here, you need would need to keep the brackets of empty function calls. E.g. don’t add `coord_flip` to a `ggplot` object, but add `coord_flip()`. Be aware of this difference.

## 5.4 Univariate analysis of a continuous variable

We start with a uniform analysis of continuous variables. Examples of continuous variables are age, distance, speed, weight, etc. For these kind of variables we can measure the centrality and the spread. Measures of centrality are mean, median. Measures of spread are standard deviation, quantiles, min and max, range, interquartile range.

All these measures have one thing in common: they *summarise* a continuous variable by way of *one* number, *one* value. This can be performed using the `summarise` function of `dplyr`.<sup>7</sup>

As an example, suppose we want to compute the mean and the standard deviation of the `cty`.

```
summarize(mpg, mean_cty = mean(cty), st_dev_cty = sd(cty))

## # A tibble: 1 x 2
##   mean_cty    st_dev_cty
##       <dbl>        <dbl>
## 1     16.9        4.26
```

The first argument of `summarize` is the data argument. All the following arguments will become new columns in the resulting tables. `mean_cty` and `st_dev_cty` are the names of the new columns. These names can be whatever that you want them to be.<sup>8</sup> The parts after the `=`-sign will become the contents of the columns, i.e. the mean (computed by the `mean` function) and the standard deviation (computed by the `sd` function).

We can rewrite the line above by using the `%>%` symbol. Notice that after this symbol, we used *enter*, so that `summarize` begins on a new line, with an indentation. The indentation points to the fact that these lines actually form one statement (i.e. one line cannot be executed without the other).<sup>9</sup>

```
mpg %>%
  summarize(mean_cty = mean(cty), st_dev_cty = sd(cty)) %>%
  pander()
```

mean_cty	st_dev_cty
16.86	4.256

All functions used within the `summarize` function should return *one* and only *one* value, i.e. the mean, the median, the interquartile range, etc. We call these functions *summary functions*. An (incomplete) list of summary functions is included here:

<sup>7</sup> Both the British English *summarise* as the American English *summarize* can be used.

<sup>8</sup> Instead of providing names without quotation marks (e.g. `mean_cty`), you can also add quotation marks. This allows you to use spaces. When your output is final and not intended for further manipulation, you might want to do this, to have nice column headers. For example `mpg %>% summarize("Mean cty" = mean(cty))`. However, in general, spaces should be avoided.

<sup>9</sup> Remember that we use `pander` to improve the layout of our tables, and you shouldn't pay attention to that.

- `min`
- `max`
- `mean`
- `median`
- `first` (first element of vector)
- `last` (last element of vector)
- `n` (number of values)<sup>10</sup>
- `nth` (n-th value of a vector)
- `n_distinct` (number of distinct values in a vector)
- `IQR` (interquartile range)
- `var` (variance)
- `sd` (standard deviation)
- `quantile`
- `sum`

<sup>10</sup> WARNING: only use `n` within `summarize`. This function cannot be used on normal vectors.

Note that (except `n`) all these functions can also be used on normal vectors, i.e. not inside the `summarize` function. The other way round, these are not the only functions that can be used within `summarize`. In general, all function that return a single value can be used. Moreover, we can also include columns of which we set the value manually. For instance, we can update the last example and include that we used the `cty` variable as a character column. In this case, we don't need a summary function, but only a value, e.g. "cty".

```
mpg %>%
  summarize(variable = "cty", mean_cty = mean(cty), st_dev_cty = sd(cty)) %>%
  pander
```

variable	mean_cty	st_dev_cty
cty	16.86	4.256

Some summary functions need additional arguments, such as the `nth` function which need a value for `n`, and possibly specify an ordering. The `quantile` function needs a `probs` argument, which stands for probability. For instance, to calculate the first 10% percentile, we set `probs = 0.1`.

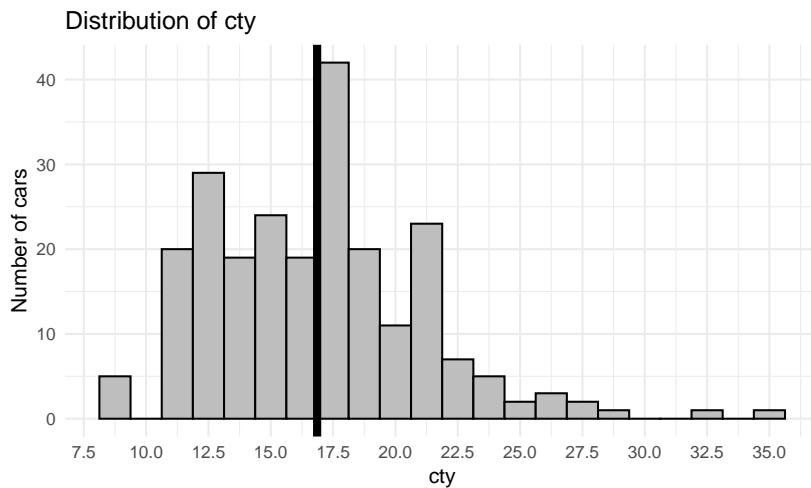
```
mpg %>%
  summarize(variable = "cty",
            q0.2 = quantile(cty, 0.2),
            q0.4 = quantile(cty, 0.4),
            q0.6 = quantile(cty, 0.6),
            q0.8 = quantile(cty, 0.8)) %>%
  pander
```

variable	q0.2	q0.4	q0.6	q0.8
cty	13	15	18	20

We can see that 20% of the *cty* values is less than or equal to 13, and 20% of the *cty* values is greater than or equal to 20. In the last block of code, we have put each variable on a new line. Rstudio will automatically indent all lines, such that it is clear that they are arguments of the **summarize** function. The **pander** function came again on a new line, using the same indentation as **summarize**.

If we want to support our results with graphical output, we can plot a box plot or histogram using **ggplot2**. For an introduction to **ggplot2**, we refer to our **ggplot2** tutorial. Below, we have plotted a histogram. We have also added a vertical line to indicate the mean of *cty*, using **geom\_vline**.

```
mpg %>%
  ggplot(aes(cty)) +
  geom_histogram(binwidth = 1.25, color = "black", fill = "grey") +
  geom_vline(xintercept = mean(mpg$cty), lwd = 2) +
  labs(title = "Distribution of cty",
       x = "cty",
       y = "Number of cars") +
  theme_minimal() +
  scale_x_continuous(breaks = seq(7.5, 35, 2.5))
```



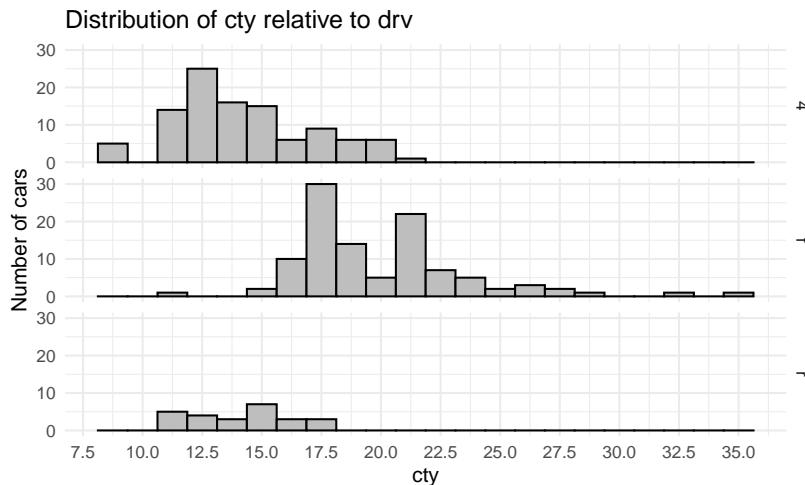
With the **summarize** function we can perform a univariate analysis of the spread and centrality of any continuous variable. Great! Now, it is time to go one step further and start with bivariate analysis of continuous variables in combination with categorical variables.

## 5.5 Bivariate analysis of a continuous variable with respect to a categorical variable

Before we start our calculations, let's continue in a graphical way.

Before, we have analyzed the distribution of cty, as shown with the histogram above. Now we want to analyse it for different types of drivings (e.g. 4-wheel drive, front wheel drive or rear wheel drive), as recorded by the variable drv. Graphically, we can plot different histograms for each of these three categories using facets.

```
mpg %>%
  ggplot(aes(cty)) +
  geom_histogram(binwidth = 1.25, color = "black", fill = "grey") +
  labs(title = "Distribution of cty relative to drv",
       x = "cty",
       y = "Number of cars") +
  theme_minimal() +
  scale_x_continuous(breaks = seq(7.5, 35, 2.5)) +
  facet_grid(drv~.)
```



Clearly, there are some differences among these categories. Let's try to put these into numbers. What we effectively want to do is calculate the centrality and spread for each of these categories, i.e. for each *group* of cars. We therefore introduce a new dplyr-function called `group_by`. The first argument of this function is needed for the data, all other arguments (which should normally be categorical) will be used to group the data.

```
group_by(mpg, drv)

## # A tibble: 234 x 11
## # Groups:   drv [3]
```

```

##   manufacturer model      displ  year cyl trans   drv     cty   hwy fl class
##   <fct>        <fct>    <dbl> <int> <ord> <fct>   <fct> <int> <int> <fct> <fct>
## 1 audi         a4       1.8  1999  4 auto(l~ f     18    29 p   comp~
## 2 audi         a4       1.8  1999  4 manual~ f    21    29 p   comp~
## 3 audi         a4       2    2008  4 manual~ f    20    31 p   comp~
## 4 audi         a4       2    2008  4 auto(a~ f     21    30 p   comp~
## 5 audi         a4       2.8  1999  6 auto(l~ f     16    26 p   comp~
## 6 audi         a4       2.8  1999  6 manual~ f    18    26 p   comp~
## 7 audi         a4       3.1  2008  6 auto(a~ f     18    27 p   comp~
## 8 audi         a4 quat~  1.8  1999  4 manual~ 4    18    26 p   comp~
## 9 audi         a4 quat~  1.8  1999  4 auto(l~ 4    16    25 p   comp~
## 10 audi        a4 quat~  2    2008  4 manual~ 4   20    28 p   comp~
## # ... with 224 more rows

```

The output of this line just returns us a tibble, with no remarkable modifications. But don't be fooled, because there are! On the second line printed we read “Groups: *drv* [3]”. Thus, this tibble is grouped on the variable *drv*, and there are three different groups. Just for the sake of example, let us also add *trans* as a group.

```

group_by(mpg, drv, trans)

## # A tibble: 234 x 11
## # Groups:   drv, trans [24]
##   manufacturer model      displ  year cyl trans   drv     cty   hwy fl class
##   <fct>        <fct>    <dbl> <int> <ord> <fct>   <fct> <int> <int> <fct> <fct>
## 1 audi         a4       1.8  1999  4 auto(l~ f     18    29 p   comp~
## 2 audi         a4       1.8  1999  4 manual~ f    21    29 p   comp~
## 3 audi         a4       2    2008  4 manual~ f    20    31 p   comp~
## 4 audi         a4       2    2008  4 auto(a~ f     21    30 p   comp~
## 5 audi         a4       2.8  1999  6 auto(l~ f     16    26 p   comp~
## 6 audi         a4       2.8  1999  6 manual~ f    18    26 p   comp~
## 7 audi         a4       3.1  2008  6 auto(a~ f     18    27 p   comp~
## 8 audi         a4 quat~  1.8  1999  4 manual~ 4    18    26 p   comp~
## 9 audi         a4 quat~  1.8  1999  4 auto(l~ 4    16    25 p   comp~
## 10 audi        a4 quat~  2    2008  4 manual~ 4   20    28 p   comp~
## # ... with 224 more rows

```

We can just put *trans* after *drv* in the `group_by` function. Now, we can see that the tibble is grouped on these two variables, and there are 24 different groups<sup>11</sup>. We can also check the grouping of `data.frame` by using the `groups` function. This saves us from printing the entire data to check the grouping. Note that below, we again use the piping symbol effectively.

```
mpg %>%
```

<sup>11</sup> In particular, there are 3 *drv* values and 10 *trans* values. Thus, there could be maximum 30 different groups. The fact that we see only 24 groups, means that not all possible combinations of *drv* and *trans* values occur in the data.

```
group_by(drv, trans) %>%
  groups()

## [[1]]
## drv
##
## [[2]]
## trans
```

Ok, but wait a minute. If nothing else changes, why are we doing this? Once a data.frame is grouped, the next operations will be performed for each group separately. Isn't that just what we needed?

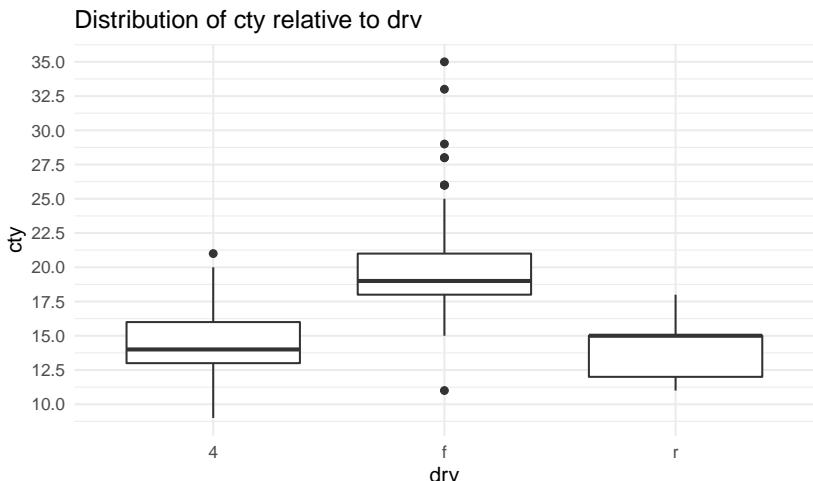
Indeed. For each *drv* group, we wanted to compute the centrality and spread. Let's give it a try.

```
mpg %>%
  group_by(drv) %>%
  summarize(mean_cty = mean(cty), sd_cty = sd(cty)) %>%
  pander
```

drv	mean_cty	sd_cty
4	14.33	2.874
f	19.97	3.627
r	14.08	2.216

Great! Calling summarize will no longer give us 1 row of values. Instead, it will return 1 row for each group. We can see that the mean cty is much larger for cars with a front wheel drive than for other cars, as was already suspected based on the histogram. Another way of visualizing this, without using facets, is to use box plots.

```
mpg %>%
  ggplot(aes(drv, cty)) +
  geom_boxplot() +
  labs(title = "Distribution of cty relative to drv",
       x = "drv",
       y = "cty") +
  theme_minimal() +
  scale_y_continuous(breaks = seq(7.5, 35, 2.5))
```



We can now analyse the centrality and spread of a continuous variable, both univariately and bivariately with respect to a categorical variable. To do this, we have learned to use two new functions: `summarize` and `group_by`. Summarizing an ungrouped `data.frame` will return a `data.frame` with 1 row. Summarizing a grouped `data.frame` will return a `data.frame` where the number of rows is equal to the number of groups.

The next thing on our list is the univariate analysis of a categorical variable. For this, we will compute frequency tables and learn yet some additional new exciting functions: `mutate`, `arrange` and `slice`. Let's get on with it!

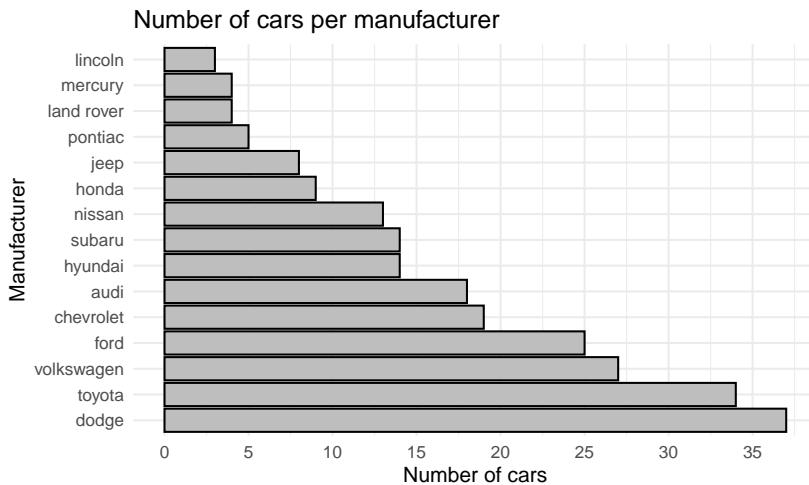
## 5.6 Univariate analysis of a categorical variable

In our data, there are 15 manufacturers. Assume we want to know more about these, i.e. which manufacturer has a lot of cars in our data, and which one has less. Graphically, we can already do this using a bar chart.<sup>12</sup>

```
mpg %>%
  ggplot() +
  geom_bar(aes(fct_infreq(manufacturer)), color = "black", fill = "grey") +
  coord_flip() +
  labs(title = "Number of cars per manufacturer",
       x = "Manufacturer",
       y = "Number of cars") +
  scale_y_continuous(breaks = seq(0, 40, 5)) +
  theme_minimal()
```

<sup>12</sup>

Did you notice the function `fct_infreq` wrapped around the `manufacturer` variable within the `geom_bar` mapping? This function makes sure that the bars are ordered from infrequent to frequent. This function is one of the helpful functions for factors from the package `forcats`, which is an anagram of `factors` and it symbolizes the fact that many people in the R community are cat lovers (seriously). If you want, you



Now, we want to analyze this numerically. The most obvious way to do this is with a frequency table. Below, we see the end product of this analysis. Subsequently, we will start constructing it step by step.

nr	manufacturer	frequency	relative_frequency	cumulative_relative_frequency
1	dodge	37	15.81	15.81
2	toyota	34	14.53	30.34
3	volkswagen	27	11.54	41.88
4	ford	25	10.68	52.56
5	chevrolet	19	8.12	60.68
6	audi	18	7.69	68.38
7	hyundai	14	5.98	74.36
8	subaru	14	5.98	80.34
9	nissan	13	5.56	85.9
10	honda	9	3.85	89.74
11	jeep	8	3.42	93.16
12	pontiac	5	2.14	95.3
13	land rover	4	1.71	97.01
14	mercury	4	1.71	98.72
15	lincoln	3	1.28	100

Let's start. The first thing to do is to count the number of observations for each value of the categorical variable. Thus, how many cars are there for each manufacturer? We can do this by grouping the data on *manufacturer* and then counting the number of rows using `n`.

```
mpg %>%
  group_by(manufacturer) %>%
  summarize(frequency = n()) %>%
  pander
```

manufacturer	frequency
audi	18
chevrolet	19
dodge	37
ford	25
honda	9
hyundai	14
jeep	8
land rover	4
lincoln	3
mercury	4
nissan	13
pontiac	5
subaru	14
toyota	34
volkswagen	27

The function `n()` does not require any argument. It will just calculate the number of rows there are in a `data.frame`, or, as in this case, in each group. The next step is to order these manufacturers according to the number of cars. In order to *arrange* rows, we use the function `arrange` from `dplyr`. Since we want to arrange the manufacturers by decreasing frequency, we use the function `desc` (descending). In particular, the following will do the trick:

```
mpg %>%
  group_by(manufacturer) %>%
  summarize(frequency = n()) %>%
  arrange(desc(frequency)) %>%
  pander
```

manufacturer	frequency
dodge	37
toyota	34
volkswagen	27
ford	25
chevrolet	19
audi	18
hyundai	14
subaru	14
nissan	13
honda	9
jeep	8

manufacturer	frequency
pontiac	5
land rover	4
mercury	4
lincoln	3

The first argument of `arrange` is again the data, which is given to it using the piping symbol. The other arguments will then be used to order the data. Note that more than one variable can be specified. The data will then be arranged firstly using the first variable. Subsequently, the next variables will be used to break ties. When `desc` is placed around a variable name, this variable will be arranged decreasingly. Just try out some different orderings yourself!

Now, we need to add a new column for the relative frequencies. To do this we use the function `mutate` from `dplyr`, which literally means *to change*. The `mutate` function call is similar to that of `summarize`: the first argument is the data, the others are of the form `variable_name = value`. The variable name is the name that will appear as name of the new column. The value is the value for the new column. This can either be function which returns a vector with a length equal to that of the data.frame, or it can be a simple computation using other variables in the data.frame. The following statement will compute the relative frequencies, by dividing each frequency, with the sum of all frequencies.

```
mpg %>%
  group_by(manufacturer) %>%
  summarize(frequency = n()) %>%
  arrange(desc(frequency)) %>%
  mutate(relative_frequency = frequency/sum(frequency)) %>%
  pander
```

manufacturer	frequency	relative_frequency
dodge	37	0.1581
toyota	34	0.1453
volkswagen	27	0.1154
ford	25	0.1068
chevrolet	19	0.0812
audi	18	0.07692
hyundai	14	0.05983
subaru	14	0.05983
nissan	13	0.05556
honda	9	0.03846

manufacturer	frequency	relative_frequency
jeep	8	0.03419
pontiac	5	0.02137
land rover	4	0.01709
mercury	4	0.01709
lincoln	3	0.01282

Here, `sum(frequency)` refers to the sum of the frequency column, while `frequency` refers to the specific values in each row. Next, we will add the cumulative relative frequency. We can add this to the same `mutate` call.

```
mpg %>%
  group_by(manufacturer) %>%
  summarize(frequency = n()) %>%
  arrange(desc(frequency)) %>%
  mutate(relative_frequency = frequency/sum(frequency),
         relative_cumulative_frequency = cumsum(relative_frequency)) %>%
  pander
```

manufacturer	frequency	relative_frequency	relative_cumulative_frequency
dodge	37	0.1581	0.1581
toyota	34	0.1453	0.3034
volkswagen	27	0.1154	0.4188
ford	25	0.1068	0.5256
chevrolet	19	0.0812	0.6068
audi	18	0.07692	0.6838
hyundai	14	0.05983	0.7436
subaru	14	0.05983	0.8034
nissan	13	0.05556	0.859
honda	9	0.03846	0.8974
jeep	8	0.03419	0.9316
pontiac	5	0.02137	0.953
land rover	4	0.01709	0.9701
mercury	4	0.01709	0.9872
lincoln	3	0.01282	1

While the computation of relative frequency was a simple formula, the cumulative frequency is computed using the `cumsum`. This is what is called a *window function*. While summary functions always return one value, window functions return the same number of values as the vector used as input. Thus, our 15 relative frequencies will result in 15

cumulative frequencies. Other window functions are listed below.

### Cumulative functions

- cumsum: The cumulative sum of a vector
- cummax: The cumulative maximum of a vector
- cummin: The cumulative minimum of a vector
- cumprod: The cumulative product of a vector
- cummean: A cumulative, or rolling, mean
- cumany: For logical values, a cumulative “or”
- cumall: For logical values, a cumulative “and”
- cume\_dist: Cumulative distribution

### Element-wise function of more than one variable

- pmax: Element/pair-wise maximum of a vector
- pmin: Element/pair-wise minimum of a vector

### Ranking function

- percent\_rank: ranks rescaled to [0,1]
- row\_number: Rank which breaks ties by taking elements by first occurrence (mostly alphabetically)
- min\_rank: Rank which breaks ties by giving them the same rank and omitting the next rank
- dense\_rank: Same as min\_rank, but without omission

### Shifting functions

- lead(n): shift values n places to the front, add n NA's as last values
- lag(n): shift values n places to the back, add n NA's as first values

### Other

- between(a,b): Lie the values of a vector between a and b? Returns logical
- ntile(x,n): Use variable x to arrange the observations and put them in n equally sized groups.

That's a long list, and not all of them are important. You will use some of them a lot (like cumsum), and others only in rare instances. In case of need, come back to this list.

The **pmin** function returns the pairwise minimum of x and y, while the **pmax** function will return the pairwise maximum of x and y. Although the p in these functions points to *pair* it can be used on more than two vectors.

Percent rank will assign a value with between 0 and 1. The smallest value will get zero and the highest value will be one (when there

are no ties). But, when two observations are equal, they get the same value, and there will be a *gap* between subsequent values. Row number will break ties randomly. As such, the highest value will always be equal to the number of observations. Minimum rank assigns the minimum rank equally among ties, and also includes *gaps*, like percent rank. Finally, dense rank will also assign ties the same rank, but will not allow gaps in the ranking.

```
data.frame(x = sample(1:10)) %>%
  mutate(lead = lead(x),
        lag = lag(x, 2),
        between(x, 4, 8),
        ntile(x, 5)) %>%
  pander
```

x	lead	lag	between(x, 4, 8)	ntile(x, 5)
6	1	NA	TRUE	3
1	10	NA	FALSE	1
10	2	6	FALSE	5
2	9	1	FALSE	1
9	4	10	FALSE	5
4	7	2	TRUE	2
7	5	9	TRUE	4
5	3	4	TRUE	3
3	8	7	FALSE	2
8	NA	5	TRUE	4

An example of the last window functions is show above. The lead and lag functions shift values up and down. By default, they are shifted with one place. The between function tests whether values are between certain bound (including the boundary values). Finally, the ntile function will create a variable that groups observations in equal bins, in this case 5, according to a specific variable. Thus, values 1 and 2 of x are grouped in bin 1, values 3 and 4 in bin 2, etc. Note that, although the ntile function returns numbers, it is actually a categorical variable!

They may not all seem very obvious, but sometimes these window function can come in quite powerful and handy during an analysis. Make sure to remember their existence. But, let's get back to our frequency table now, as we have gotten a little bit of the beaten track. Below, we show again our last result.

```
mpg %>%
  group_by(manufacturer) %>%
```

```

summarize(frequency = n()) %>%
arrange(desc(frequency)) %>%
mutate(relative_frequency = frequency/sum(frequency),
       relative_cumulative_frequency = cumsum(relative_frequency)) %>%
pander

```

manufacturer	frequency	relative_frequency	relative_cumulative_frequency
dodge	37	0.1581	0.1581
toyota	34	0.1453	0.3034
volkswagen	27	0.1154	0.4188
ford	25	0.1068	0.5256
chevrolet	19	0.0812	0.6068
audi	18	0.07692	0.6838
hyundai	14	0.05983	0.7436
subaru	14	0.05983	0.8034
nissan	13	0.05556	0.859
honda	9	0.03846	0.8974
jeep	8	0.03419	0.9316
pontiac	5	0.02137	0.953
land rover	4	0.01709	0.9701
mercury	4	0.01709	0.9872
lincoln	3	0.01282	1

We were quite finished, it seems, but we can still improve the table. One way to do so is to turn the relative (cumulative) frequencies into values between 0 and 100, and to round them to 2 decimals. That last thing can be done using the function `round` and specifying the number of decimals.

```

mpg %>%
group_by(manufacturer) %>%
summarize(frequency = n()) %>%
arrange(desc(frequency)) %>%
mutate(relative_frequency = frequency/sum(frequency),
       relative_cumulative_frequency = cumsum(relative_frequency),
       relative_frequency = round(100*relative_frequency,2),
       relative_cumulative_frequency = round(100*relative_cumulative_frequency,2)) %>%
pander

```

manufacturer	frequency	relative_frequency	relative_cumulative_frequency
dodge	37	15.81	15.81
toyota	34	14.53	30.34

manufacturer	frequency	relative_frequency	relative_cumulative_frequency
volkswagen	27	11.54	41.88
ford	25	10.68	52.56
chevrolet	19	8.12	60.68
audi	18	7.69	68.38
hyundai	14	5.98	74.36
subaru	14	5.98	80.34
nissan	13	5.56	85.9
honda	9	3.85	89.74
jeep	8	3.42	93.16
pontiac	5	2.14	95.3
land rover	4	1.71	97.01
mercury	4	1.71	98.72
lincoln	3	1.28	100

What's important to note here is that we have put 4 different statements in `mutate`. In these statements, it is possible to use variables created before **within the same mutate call**: the second statement uses the variable in the first. Also, it is possible to overwrite variables, i.e. statement 3 and 4 overwrite the variables created in statement 1 and 2.

You might wonder why we use 4 statements instead of 2, and why didn't we round immediately? We could have done that for the cumulative frequency, but not for the relative frequency. Can you figure out why?

Finally, we may want to add a ranking, i.e. a row number, to the frequency table. Here, we use the `row_number` function introduced above.

Before we used `desc` to arrange decreasingly in `arrange`, and now we use the minus sign to allocate the row numbers according to decreasing frequency. Is `dplyr` really this inconsistent? Fortunately, the answer is no. Using the minus is a little trick to shorten your code, but can only be used for numerical variables. This is, because sorting a numerical variable decreasingly, is the same as sorting its negative mirror increasingly (Convince yourself!). As such, we could also have used the minus sign within `arrange`, and we could have used `desc` within `row_number`. **But**, never use the minus sign to sort categorical values (they don't have a negative counter part, typically). It is best practice to be consistent in using - or `desc`, but as this is a tutorial, we show you different ways to do it.

```
mpg %>%
  group_by(manufacturer) %>%
  summarize(frequency = n()) %>%
```

```

arrange(desc(frequency)) %>%
mutate(relative_frequency = frequency/sum(frequency),
       relative_cumulative_frequency = cumsum(relative_frequency),
       relative_frequency = round(100*relative_frequency,2),
       relative_cumulative_frequency = round(100*relative_cumulative_frequency,2),
       nr = row_number(-frequency)) %>%
pander(split.table = 120)

```

manufacturer	frequency	relative_frequency	relative_cumulative_frequency	nr
dodge	37	15.81	15.81	1
toyota	34	14.53	30.34	2
volkswagen	27	11.54	41.88	3
ford	25	10.68	52.56	4
chevrolet	19	8.12	60.68	5
audi	18	7.69	68.38	6
hyundai	14	5.98	74.36	7
subaru	14	5.98	80.34	8
nissan	13	5.56	85.9	9
honda	9	3.85	89.74	10
jeep	8	3.42	93.16	11
pontiac	5	2.14	95.3	12
land rover	4	1.71	97.01	13
mercury	4	1.71	98.72	14
lincoln	3	1.28	100	15

Mutate has added the number at the end of the table.<sup>13</sup> That's what we want in most of the cases, but not in this one. To change the order of the variables, we can use the `select` function from `dplyr`. This function can be used to *select* variables, and will place them in the specified order. Thus, what we can do is the following.

```

mpg %>%
  group_by(manufacturer) %>%
  summarize(frequency = n()) %>%
  arrange(desc(frequency)) %>%
  mutate(relative_frequency = frequency/sum(frequency),
         relative_cumulative_frequency = cumsum(relative_frequency),
         relative_frequency = round(100*relative_frequency,2),
         relative_cumulative_frequency = round(100*relative_cumulative_frequency,2),
         nr = row_number(-frequency)) %>%
  select(nr, manufacturer, frequency, relative_frequency, relative_cumulative_frequency) %>%
  pander(split.table = 120)

```

<sup>13</sup> For the eager students who were wondering where the `split.table = 120` argument came from in the `pander` call: `pander` by default splits tables which are more than 80 characters wide. It will then add superfluous columns below the table in a *second* table. In order to avoid this splitting, and since this document has a quite big side margin for the table to continue, I increased this parameter to 120 characters.

nr	manufacturer	frequency	relative_frequency	relative_cumulative_frequency
1	dodge	37	15.81	15.81
2	toyota	34	14.53	30.34
3	volkswagen	27	11.54	41.88
4	ford	25	10.68	52.56
5	chevrolet	19	8.12	60.68
6	audi	18	7.69	68.38
7	hyundai	14	5.98	74.36
8	subaru	14	5.98	80.34
9	nissan	13	5.56	85.9
10	honda	9	3.85	89.74
11	jeep	8	3.42	93.16
12	pontiac	5	2.14	95.3
13	land rover	4	1.71	97.01
14	mercury	4	1.71	98.72
15	lincoln	3	1.28	100

Using the `select` statement does the trick. But I am still kind of lazy, and I don't want to write out all the variables, just to put one in the first position. We had only 5 variables here, but imagine what a waste of time it would be when we had more, like 6 for instance. Can't we just say, put `nr` in the front, and then add all the other variables in their original order? Dplyr to the rescue!

Fortunately we can. We can just add the function `everything` to `select`. This will add all the columns after `nr`, without repeating `nr` for a second time. Great, isn't it?

```
mpg %>%
  group_by(manufacturer) %>%
  summarize(frequency = n()) %>%
  arrange(desc(frequency)) %>%
  mutate(relative_frequency = frequency/sum(frequency),
         relative_cumulative_frequency = cumsum(relative_frequency),
         relative_frequency = round(100*relative_frequency,2),
         relative_cumulative_frequency = round(100*relative_cumulative_frequency,2),
         nr = row_number(-frequency)) %>%
  select(nr, everything()) %>%
  pander(split.table = 120)
```

nr	manufacturer	frequency	relative_frequency	relative_cumulative_frequency
1	dodge	37	15.81	15.81
2	toyota	34	14.53	30.34
3	volkswagen	27	11.54	41.88

nr	manufacturer	frequency	relative_frequency	relative_cumulative_frequency
4	ford	25	10.68	52.56
5	chevrolet	19	8.12	60.68
6	audi	18	7.69	68.38
7	hyundai	14	5.98	74.36
8	subaru	14	5.98	80.34
9	nissan	13	5.56	85.9
10	honda	9	3.85	89.74
11	jeep	8	3.42	93.16
12	pontiac	5	2.14	95.3
13	land rover	4	1.71	97.01
14	mercury	4	1.71	98.72
15	lincoln	3	1.28	100

Select, arrange, mutate, summarize, group\_by. We are really mastering dplyr already. Not so difficult after all, isn't it? Of course there are still other functions, but these 5 are really some of most important functions for data manipulation. Only filter misses in the dplyr hall of fame, but we leave that one for another time. It is always a good idea to have something to look forward to, don't you agree?

Before we continue with bivariate analysis, there are two more things we should do. Look at the slice function, and evaluate our piping symbol.

The slice function can be used to slice rows from a data.frame. Suppose we have a very long frequency table, and we are only interested in the top 10 values. We can then slice the first 10 rows of this table as follows.

```
mpg %>%
  group_by(manufacturer) %>%
  summarize(frequency = n()) %>%
  arrange(desc(frequency)) %>%
  mutate(relative_frequency = frequency/sum(frequency),
         relative_cumulative_frequency = cumsum(relative_frequency),
         relative_frequency = round(100*relative_frequency,2),
         relative_cumulative_frequency = round(100*relative_cumulative_frequency,2),
         nr = row_number(-frequency)) %>%
  select(nr, everything()) %>%
  slice(1:10) %>%
  pander(split.table = 120)
```

nr	manufacturer	frequency	relative_frequency	relative_cumulative_frequency
1	dodge	37	15.81	15.81
2	toyota	34	14.53	30.34
3	volkswagen	27	11.54	41.88
4	ford	25	10.68	52.56
5	chevrolet	19	8.12	60.68
6	audi	18	7.69	68.38
7	hyundai	14	5.98	74.36
8	subaru	14	5.98	80.34
9	nissan	13	5.56	85.9
10	honda	9	3.85	89.74

That's slicing. Nothing more, nothing less. We have build up some *large* block of code, but our piping symbol neatly strings it together, doesn't it? Imagine for a moment that we didn't had this symbol. In such a case, we should put each first argument again in the function it is followed by. Thus, mpg should go inside group\_by. The group\_by should go inside summarize, etc. The result would be the following.

```
pander(
  slice(
    select(
      mutate(
        arrange(
          summarize(
            group_by(mpg,
                      manufacturer),
            frequency = n(),
            desc(frequency)),
            relative_frequency = frequency/sum(frequency),
            relative_cumulative_frequency = cumsum(relative_frequency),
            relative_frequency = round(100*relative_frequency,2),
            relative_cumulative_frequency = round(100*relative_cumulative_frequency,2),
            nr = row_number(-frequency)),
            nr,
            everything(),
            1:10)
  )
```

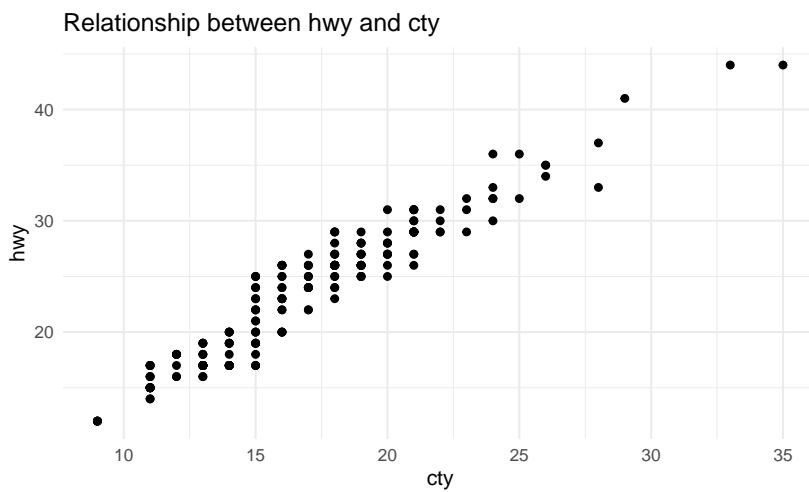
That's quite a mess, isn't it? All functions got separated from their arguments, and we cannot really make out what we were doing at all. Surely, the piping symbol does a great job at making our code readable and understandable. Make sure to use it wisely!

Now we continue with the bivariate analysis of two continuous variables.

### 5.7 Bivariate analysis of a continuous variable with respect to another continuous variable

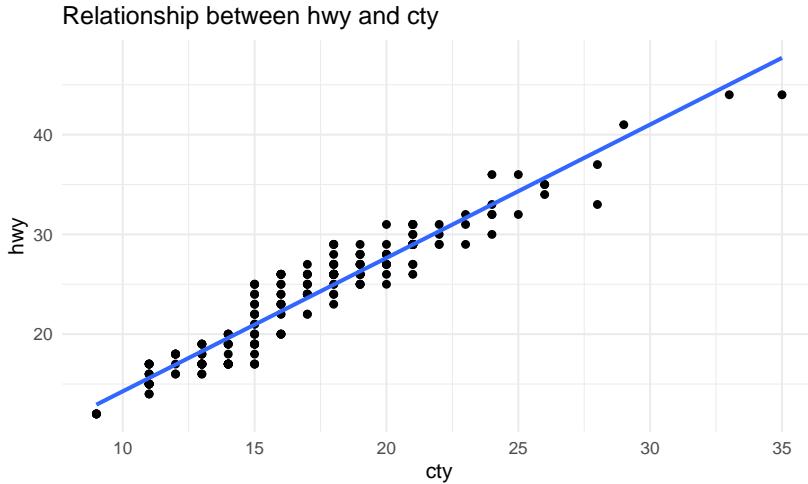
The relationship between two continuous variables can be easily shown graphically with a scatter plot. Let's look at *cty* compared with *hwy*.

```
mpg %>%
  ggplot(aes(cty, hwy)) +
  geom_point() +
  theme_minimal() +
  labs(title = "Relationship between hwy and cty")
```



There is a very clear relation between these variables, and it appears to be quite linear. Let's draw a linear line through the scatter plot.

```
mpg %>%
  ggplot(aes(cty, hwy)) +
  geom_point() +
  theme_minimal() +
  labs(title = "Relationship between hwy and cty") +
  geom_smooth(method = "lm", se = F)
```



A linear relationship as this can be very effectively measured with the correlation coefficient. Computing correlations can be done with the base R function `cor`. However, this function expects a `data.frame` with only continuous variables, as it will compute all possible correlations between these. Correlations cannot be computed between categorical variables (except for ordinal ones). Thus, first, we select the continuous variables using the `select` function from `dplyr`. The variables we are interested in are `displ`, `year`, `cty` and `hwy`.

```
mpg %>%
  select(displ, year, cty, hwy) %>%
  cor %>%
  pander
```

	displ	year	cty	hwy
displ	1	0.1478	-0.7985	-0.766
year	0.1478	1	-0.03723	0.002158
cty	-0.7985	-0.03723	1	0.9559
hwy	-0.766	0.002158	0.9559	1

The `select` call might become large when selecting numeric variables in a large data set. We can rewrite it using the `select_if` function, which doesn't expect variable names, but it requires a function to test whether a variable should be included. We can use the `is.numeric` function to test whether a vector is numeric. Thus,<sup>14</sup>

```
mpg %>% select_if(is.numeric) %>%
  cor %>%
  pander
```

<sup>14</sup> Note how we specify the `is.numeric` function within `select_if`: without quotation marks and without brackets. This is important!

	displ	year	cty	hwy
displ	1	0.1478	-0.7985	-0.766
year	0.1478	1	-0.03723	0.002158
cty	-0.7985	-0.03723	1	0.9559
hwy	-0.766	0.002158	0.9559	1

Here we see the very strong correlation between *cty* and *hwy*. We also see strong negative correlations between *displ* on the one hand and *cty* and *hwy* on the other hand. Can you try to visualize these as we did for *cty* and *hwy*?

We could also visualize the correlations computed by `cor` using `ggplot`, but something's amiss. The data returned by the `cor` function isn't very tidy. There are variable names in the columns as well as in the rows?<sup>15</sup> Even worse, it isn't a `data.frame`!

```
mpg %>%
  select_if(is.numeric) %>%
  cor %>%
  class

## [1] "matrix"
```

Matrix? The movie? No, matrix is another type of object in R that you might not have heard of yet. While a `data.frame` is a collection of vectors, you can think of a matrix as a two dimensional vector. This means that, like in a vector, all elements in a matrix should have the same type. Like a vector can have names, also the rows and columns of a matrix can have names, like in our example.

But, as we know, `ggplot2` can only work with `data.frames`. We are completely doomed! But, we could possibly turn a matrix into a `data.frame`... And then we can reshape it such that all variables are on columns... We could try...

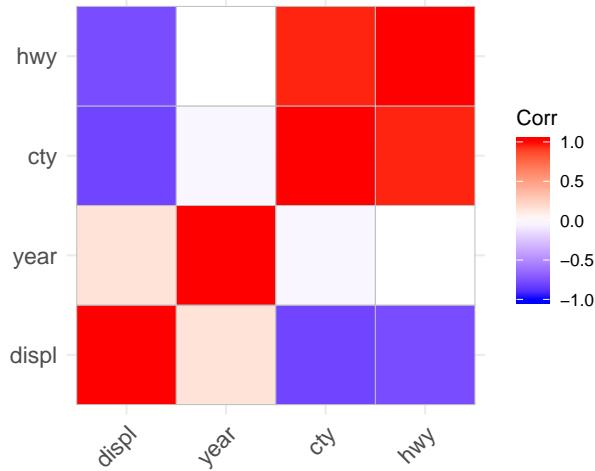
But, that seems a lot of work. And we are still lazy. Luckily, most R-users are lazy to some extent, and someone must have done this work someday, and must have been so generous to put it into a package. `ggcorrplot` is the answer to all our needs. The `ggcorrplot` package has a single function that is useful for us: `ggcorrplot`. Note that both the package and its main function has been given the same name. How generous! But be careful: `cor` had one r, `ggcorrplot` has two.

```
library(ggcorrplot)

mpg %>%
  select_if(is.numeric) %>%
```

<sup>15</sup> Remember the tidyverse? There are different meanings related to “tidy”. One thing is tidy functions, i.e. functions that neatly work together with, for instance, the piping symbol. Another thing is tidy data. However, tidy data is out of the scope of this tutorial.

```
cor %>%
ggcorrplot()
```



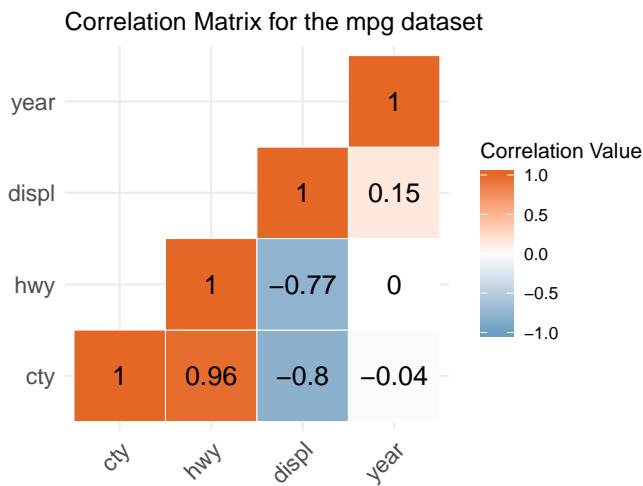
What we get is a visual matrix. The color of the squares indicate the direction of the relationship (by default, red is positive and blue is negative), That's nice. However, there are many different options to make `ggcorrplot` just as we like it. Lets look at the most important ones:

- method: “square” or “circle”: the shape of the elements in the matrix
- lab: if TRUE, it will show the correlation values on top of the squares (or circles)
- lab\_col and lab\_size: allow us to modify how the values are printed
- outline\_color: the color of the borders of the squares
- type: “full”,“lower” or “upper”: a correlation matrix is symmetric, so we can chose to show only the lower or upper half.
- ggtheme: you can use any one of the default ggplot2 themes: theme\_grey, theme\_minimal, theme\_classic. Specify them without brackets or quotation marks, just as we did with `is.numeric` a few minutes ago.
- title
- legend.title
- colors, a vector of three colors for the low, mid and high values.  
Default: `c("blue","white","red")`
- hc.order: if set to TRUE we can order the variables to show which variables are most related
- show.diag: Show the diagonal if TRUE (in case type is equal to “lower” or “upper”)

An example of a slightly modified version of our plot is the follow-

ing: <sup>16</sup>

```
mpg %>%
  select_if(is.numeric) %>%
  cor %>%
  ggcorrplot(type = "lower", ggtheme = theme_minimal, colors = c("#6D9EC1", "white", "#E46726"),
             show.diag = T,
             lab = T, lab_size = 5,
             title = "Correlation Matrix for the mpg dataset",
             legend.title = "Correlation Value",
             outline.color = "white",
             hc.order = T)
```



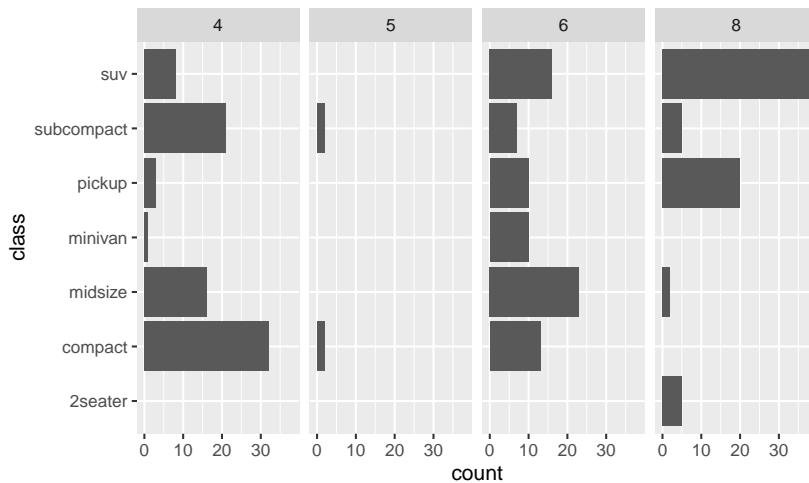
<sup>16</sup> More information and settings for these plots can be found in this manual

We have already traveled a long and exciting way through the tidyverse! The final thing we want to achieve, is to do a bivariate analysis of two categorical variables. For this, we will learn to construct contingency tables. The end is really near now.

## 5.8 Bivariate analysis of a categorical variable with respect to another categorical variable

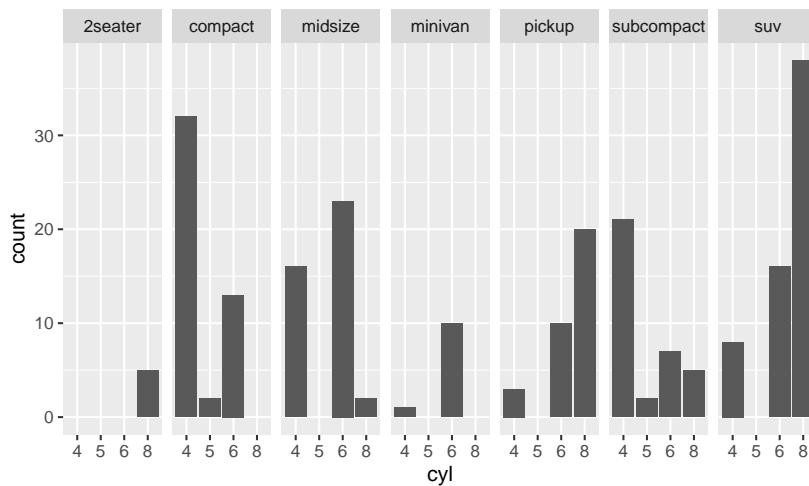
We start again by looking at a graph. Suppose we want to look at the relationship between *class* and *cyl*. We could make faceted bar charts.

```
mpg %>%
  ggplot(aes(class)) +
  geom_bar() +
  facet_grid(~cyl) +
  coord_flip()
```



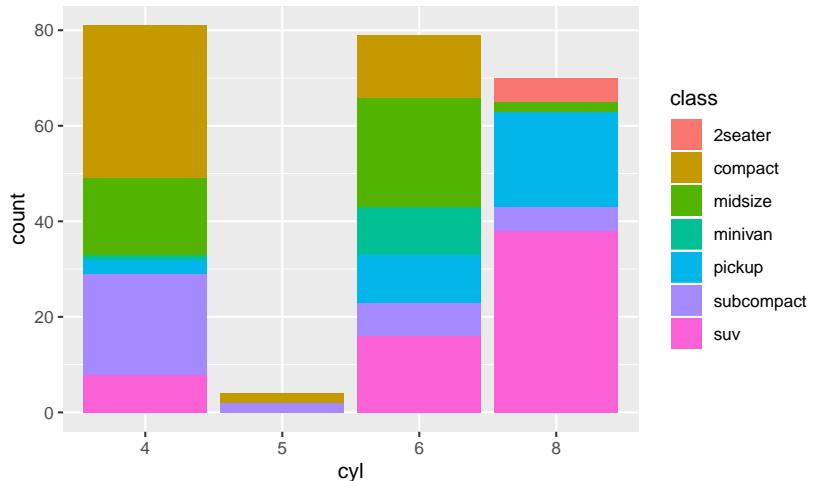
Here we see that the distributions of class differs when the number of cylinder changes. For 4 cylinders, most cars are compact, while for 8 cylinders, most cars are suv's. We could also look at it from another perspective however.

```
mpg %>%
  ggplot(aes(cyl)) +
  geom_bar() +
  facet_grid(~class)
```



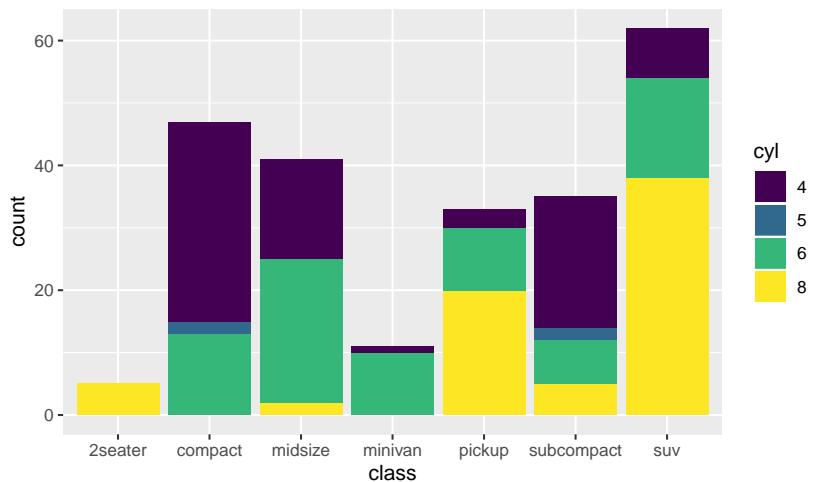
And yet another

```
mpg %>%
  ggplot(aes(cyl)) +
  geom_bar(aes(fill = class))
```



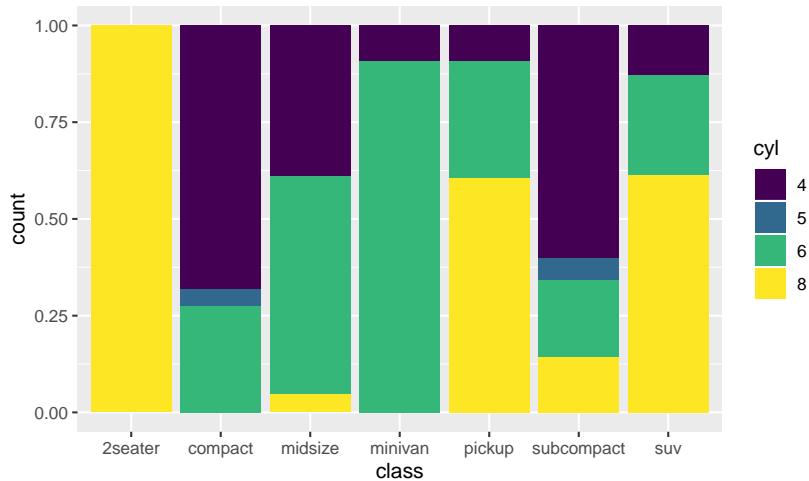
And yet another

```
mpg %>%
  ggplot(aes(class)) +
  geom_bar(aes(fill = cyl))
```



And yet another

```
mpg %>%
  ggplot(aes(class)) +
  geom_bar(aes(fill = cyl), position = "fill")
```



Indeed, there are many ways to look at the relationship between two categorical variables. The many graphs that can be made indicate that there can also be multiple contingency tables. Everything really depends on the question you want to answer. Fortunately, we already know a lot of important functions by now, which we can put to good use. We start by counting how many cars there are for each cyl-class combination. Let's roll.

```
mpg %>%
  group_by(cyl, class) %>%
  summarize(frequency = n()) %>%
  pander
```

cyl	class	frequency
4	compact	32
4	midsize	16
4	minivan	1
4	pickup	3
4	subcompact	21
4	suv	8
5	compact	2
5	subcompact	2
6	compact	13
6	midsize	23
6	minivan	10
6	pickup	10
6	subcompact	7
6	suv	16
8	2seater	5
8	midsize	2

cyl	class	frequency
8	pickup	20
8	subcompact	5
8	suv	38

Great, that went perfect! To make a contingency table of this, we want to put one of the two variables on the columns. This will create a matrix-like structure. Here, we need a new functions, which is called **spread**. This function is from the **tidyverse** package, which is used to tidy data. However, the other functions from this packages won't be discussed here.

The function **spread** has 2 arguments, **key** and **value** (apart from data, of course). The key refers to the variable of which we want to put the values as new columns, in this case *class*. The value refers to the variable of which we want to place the values in the new columns, in this case, our frequency. No idea what is going to happen? Let's look at an example.

```
mpg %>%
  group_by(cyl, class) %>%
  summarize(frequency = n()) %>%
  spread(class, frequency) %>%
  pander
```

cyl	2seater	compact	midsize	minivan	pickup	subcompact	suv
4	NA	32	16	1	3	21	8
5	NA	2	NA	NA	NA	2	NA
6	NA	13	23	10	10	7	16
8	5	NA	2	NA	20	5	38

Did you see what has happened? Just compare the last two tables. What happens if you use *cyl* as key?

Now, why do some of the values in this `data.frame` show NA, which stands for not available? That's because not all class-cyl combinations exists, i.e., our list before only showed us 19 existing combinations. There are no cars for the other 9 combinations. However, we don't like NA's very much, so let's change them into zero's. Fortunately, this is a feature of the **spread** function. The argument **fill** will be used to "fill" the empty cells. We just need to set it to zero.<sup>17</sup>

```
mpg %>%
  group_by(cyl, class) %>%
  summarize(frequency = n()) %>%
```

<sup>17</sup> We have already encountered `fill` in different contexts. It is used to specify the fill-color in plots, it can be used as a position in `geom_bar` to fill the bars to 100%, and now we can use it to fill empty spaces in the contingency table. Don't mess things up here!

```
spread(class, frequency, fill = 0) %>%
pander
```

cyl	2seater	compact	midsized	minivan	pickup	subcompact	suv
4	0	32	16	1	3	21	8
5	0	2	0	0	0	2	0
6	0	13	23	10	10	7	16
8	5	0	2	0	20	5	38

Great. This is what I would call a “Contingency table with absolute frequencies”. It shows us the absolute number of cars for each combination of cyl and class, and it tells us that the combination 8 and SUV is the most prominent in our data. This contingency table goes very well with our first two graphs.

Another question we could ask is: If we look at cars with 4 cylinders, what is the specific distribution of classes? In this case, we would like to have relative frequencies. Let’s try this.

```
mpg %>%
group_by(cyl, class) %>%
summarize(frequency = n()) %>%
mutate(relative_frequency = frequency/sum(frequency)) %>%
pander
```

cyl	class	frequency	relative_frequency
4	compact	32	0.3951
4	midsized	16	0.1975
4	minivan	1	0.01235
4	pickup	3	0.03704
4	subcompact	21	0.2593
4	suv	8	0.09877
5	compact	2	0.5
5	subcompact	2	0.5
6	compact	13	0.1646
6	midsized	23	0.2911
6	minivan	10	0.1266
6	pickup	10	0.1266
6	subcompact	7	0.08861
6	suv	16	0.2025
8	2seater	5	0.07143
8	midsized	2	0.02857
8	pickup	20	0.2857

cyl	class	frequency	relative_frequency
8	subcompact	5	0.07143
8	suv	38	0.5429

That's exactly what we wanted to have. But, don't you notice something strange?

Within each cyl group, the relative frequencies add up to one. For instance look at cyl = 5, there are 50% compact cars and 50% subcompact. That is what we wanted, but... this is not what happened before when making frequency tables. What has changed?

The answer is subtle, tricky and important.

Every time a summarize is done of a grouped data.frame, the summarize function removes the last grouping variable. It is said that the summarize function *peels* of group levels. In the case of our frequency table earlier:

```
mpg %>%
  group_by(manufacturer) %>%
  summarize(frequency = n()) %>%
  groups

## NULL
```

After the summarize, there are no grouping variables left. There was only one, and it has been removed. The summarize function implicitly assumes that the grouping has become useless after the summarize. This means that, when we used `sum(frequency)` in the next line to compute relative frequencies, it would compute the sum of the frequencies in the whole table.

Now, back to our example. After the frequencies are computed, the data is only grouped by cyl. Thus, the `sum(frequency)` will now compute the sum of the frequencies for each cyl-group. Indeed, remember, for a grouped data.frame, all operations happen separately for each group. That is exactly why the relative frequencies add to 1 within each cyl group.

```
mpg %>%
  group_by(cyl, class) %>%
  summarize(frequency = n()) %>%
  groups

## [[1]]
## cyl
```

When we change the order of grouping levels in the `group_by` function, we also change the relative frequencies which will get computed.

If we put class first, the relative frequencies will add up to 1 for each of the classes.

```
mpg %>%
  group_by(class, cyl) %>%
  summarize(frequency = n()) %>%
  mutate(relative_frequency = frequency/sum(frequency)) %>%
  pander
```

class	cyl	frequency	relative_frequency
2seater	8	5	1
compact	4	32	0.6809
compact	5	2	0.04255
compact	6	13	0.2766
midsize	4	16	0.3902
midsize	6	23	0.561
midsize	8	2	0.04878
minivan	4	1	0.09091
minivan	6	10	0.9091
pickup	4	3	0.09091
pickup	6	10	0.303
pickup	8	20	0.6061
subcompact	4	21	0.6
subcompact	5	2	0.05714
subcompact	6	7	0.2
subcompact	8	5	0.1429
suv	4	8	0.129
suv	6	16	0.2581
suv	8	38	0.6129

Admittedly, this is confusing, but at the same time it is very helpful, as it is already something useful in the case at hand. However, you should thus pay attention to the order of variables in the `group_by` function, and to the number of `summarizes` used after grouping.

Now that we have the relative frequencies, we can again reshape those with `spread` to create a matrix-like table. This time, we set `relative_frequency` as the value.

```
mpg %>%
  group_by(class, cyl) %>%
  summarize(frequency = n()) %>%
  mutate(relative_frequency = frequency/sum(frequency)) %>%
  spread(cyl, relative_frequency, fill = 0)    %>%
  pander
```

class	frequency	4	5	6	8
2seater	5	0	0	0	1
compact	2	0	0.04255	0	0
compact	13	0	0	0.2766	0
compact	32	0.6809	0	0	0
midsize	2	0	0	0	0.04878
midsize	16	0.3902	0	0	0
midsize	23	0	0	0.561	0
minivan	1	0.09091	0	0	0
minivan	10	0	0	0.9091	0
pickup	3	0.09091	0	0	0
pickup	10	0	0	0.303	0
pickup	20	0	0	0	0.6061
subcompact	2	0	0.05714	0	0
subcompact	5	0	0	0	0.1429
subcompact	7	0	0	0.2	0
subcompact	21	0.6	0	0	0
suv	8	0.129	0	0	0
suv	16	0	0	0.2581	0
suv	38	0	0	0	0.6129

Oh... that's not what we had expected, is it? We expected to have one row for each class, but now we have more than one. The reason is simple. For each combination of class and cyl, we have two values: frequency and relative frequency. When we want to spread the relative frequencies onto one line, there is no more place for the frequencies to go. As a result, the lines cannot *collapse* into one line, just as they did before. However, we can simply solve this by first getting rid of the frequencies.

```
mpg %>%
  group_by(class, cyl) %>%
  summarize(frequency = n()) %>%
  mutate(relative_frequency = frequency/sum(frequency)) %>%
  select(-frequency) %>%
  spread(cyl, relative_frequency, fill = 0) %>%
  pander
```

class	4	5	6	8
2seater	0	0	0	1
compact	0.6809	0.04255	0.2766	0
midsize	0.3902	0	0.561	0.04878
minivan	0.09091	0	0.9091	0

class	4	5	6	8
pickup	0.09091	0	0.303	0.6061
subcompact	0.6	0.05714	0.2	0.1429
suv	0.129	0	0.2581	0.6129

This looks more like it. Note that instead of indicating the variables we want to keep with `select`, we indicate the variable we want to delete with a minus sign, which is much shorter. Do not confuse it with the minus sign when arranging data, because that is something quite different.

On each row we can now see the distribution of different cylinders for a specific class of cars. You could also turn it the other way around, and show the distribution of different classes for a specific number of cylinders. Just as we had many graphs to look at these two variables, we can make many contingency tables. Can you give it a try?

Finally, suppose we want a contingency table with overall relative frequencies. I.e. we want to answer the question: what percentage of the cars has 5 cylinders and is of the class compact. We can recycle the last piece of code, and we only have to add one line.

```
mpg %>%
  group_by(class, cyl) %>%
  summarize(frequency = n()) %>%
  ungroup() %>%
  mutate(relative_frequency = frequency/sum(frequency)) %>%
  select(-frequency) %>%
  spread(cyl, relative_frequency, fill = 0) %>%
  pander
```

class	4	5	6	8
2seater	0	0	0	0.02137
compact	0.1368	0.008547	0.05556	0
midsize	0.06838	0	0.09829	0.008547
minivan	0.004274	0	0.04274	0
pickup	0.01282	0	0.04274	0.08547
subcompact	0.08974	0.008547	0.02991	0.02137
suv	0.03419	0	0.06838	0.1624

The `ungroup` function which we have added removes all the group. As a result, the sum of the frequency will be calculated over the complete data.frame. Thus, only the relative frequencies of all combina-

tions will add to one (you can do the math).

### *5.9 Background material*

We have performed 5 different analyses and learned a lot of new helpful functions along the way.

If you want to learn more, you can consult these materials

- R for Data Science, chapter 5
- dplyr Introduction



# 6

## *Storytelling - van data tot inzicht*

Bij dit hoorcollege zijn geen lecture notes beschikbaar. We verwijzen hiervoor naar de slides van het hoorcollege, incl. referenties



# 7

## *Data voorbereiden*

### *7.1 Beginnen bij het begin*

- Alvorens we aan een exploratieve data analyse kunnen beginnen, moeten we eerst onze data voorbereiden.
- Er kunnen drie grote fases geïdentificeerd worden tijdens de datavoorbereiding.
  - Correct inlezen van de data.
  - Identificeren van problemen in de data en deze corrigeren van de data.
  - Opwaarderen van de data.
- Data kan in diverse formaten aangeleverd worden en de eerste stap is ervoor zorgen dat de data ingeladen is in R. Hierbij zijn er twee specifieke elementen om aandacht aan te schenken:
  - De data analist moet ervoor zorgen dat de data correct ingeladen wordt en dat de inhoud na het inladen overeenkomt met de inhoud toen deze data de laatste keer werd opgeslagen.
  - De data analist moet ervoor zorgen dat de verschillende variabelen het juiste data type hebben.
- De volgende fase is het opkuisen van de data. Dit betekent dat men fouten gaat identificeren en deze ‘oplost’ alvorens verder te gaan.
  - Er zijn verschillende soorten fouten die in de data kunnen sluipen. Enkele mogelijke fouten zijn:
    - \* Sommige waarden ontbreken (geen waarde voor bepaalde variabele bij bepaalde observaties).
    - \* Sommige waarden zijn fout. Bijvoorbeeld: voor een deel observaties is de afstand in km opgeslagen ipv mijl of is er een typfout in de waarde van een categorische variabele.
    - \* Sommige observaties staan meerdere keren in de dataset.
  - Het opkuisen van data gebeurt in principe in 2 stappen:

- \* Eerst moeten we de data bestuderen en fouten identificeren.
- \* Vervolgens moeten we de fouten in de data ‘corrigeren’ (indien mogelijk).
- Het opwaarderen van de data betreft een reeks transformaties met als doel de data bruikbaarder te maken voor exploratieve data analyse. Er zijn verschillende manieren om dit te bereiken:
  - Bestaande variabelen transformeren naar nieuwe variabelen die geschikter zijn om patronen in de data bloot te leggen. Bijvoorbeeld het transformeren van een continue variabele naar een categorische variabele of het creëren van een nieuwe variabele ‘gemiddelde snelheid’ op basis van de variabelen ‘reistijd’ en ‘totaal afgelegde afstand’.
  - Het opsplitsen van de dataset in meerdere datasets die apart bestudeerd worden. Dit is vooral zinvol indien de dataset verschillende soorten observaties bevat of een deel observaties met uitzonderlijke waarden.

## 7.2 Data inlezen

### 7.2.1 Uitdagingen bij het correct inlezen van data

- Data kan in verschillende formaten aangeleverd worden. Afhankelijk van het formaat, zal je andere functies moeten gebruiken om de data correct in te lezen
- Maar zelfs als je de juiste functie gebruikt, kan het inlezen fout gaan omdat een computer data altijd als een reeks van 1 en 0’en opslaat en er daarom een soort vertaalsleutel nodig is van 1 en 0’en naar leesbare tekst. Deze vertaalslag wordt gerealiseerd door encoderingschema’s en je moet er voor zorgen dat bij het inlezen van data je het juiste encoderingschema hanteert.
- Eenmaal de data is ingeladen, moet je er voor zorgen dat R de datatypes juist identificeert. De meeste dataformaten houden geen informatie bij van welk datatype een specifieke variabele is en dus moet R dit ‘raden’. Omdat dit wel eens fout kan gaan, moet je als analist dit controleren en corrigeren waar nodig.

### 7.2.2 Dataformaten

#### *Flat-file databestanden*

- Lees de bron over delimited en fixed-width bestanden.
- Flat-file databestanden bevatten data die in een tabelvorm passen:
  - Iedere rij is een observatie.
  - Iedere kolom stelt een variabele voor.

- Alle items in een kolom zijn van dezelfde soort.
- Cellen van de tabel bevatten enkelvoudige gegevens (dus niet een kolom hobby's met hierin meerdere hobby's in 1 cel).
- De volgorde van de kolommen is niet van belang.
- De volgorde van de rijen is niet belang.
- Volgende data kan dus in een flat-file bestand opgeslagen worden:

naam	voornaam
Nelissen	Rob
Franssen	Ann

- De twee meest gebruikte formaten voor flat-file databestanden zijn delimited en fixed-width bestanden.
- Een delimited bestand (vaak ook wel csv-bestand of comma-separated values bestand genoemd):
  - gebruikt voor iedere rij een nieuwe regel,
  - splitst de kolommen op met behulp van een specifiek splitsingsteeken (vaak de komma of de puntkomma),
  - kan het begin en het einde van een karakterstring met behulp van een specifiek quote-teken (vaak ' of ") aanduiden.
- Bovenstaande tabel kan als een delimited databestand opgeslagen worden en ziet er dan als volgt uit:

```
naam;voornaam
Nelissen;Rob
Franssen;Ann
```

- Een fixed-width bestand gebruikt eveneens een aparte regel per rij, maar gebruikt een vast aantal karakters per kolom en heeft dus geen splitsingstekens, noch quote-teken nodig.
- Bovenstaande tabel kan als een fixed-width databestand opgeslagen worden en ziet er dan als volgt uit:

```
naam      voornaam
Nelissen Rob
Franssen Ann
```

- Het nadeel van een delimited bestand is dat je het splitsings- en quote-teken niet kunt gebruiken in je data.
- Het voordeel van een delimited bestand is dat een veld niet meer ruimte in beslag neemt dan nodig.
- Bestudeer hoofdstuk Data Import van het boek 'R for Data Science' om te weten hoe je in R data uit flat-file bestanden kunt inlezen.

### *Hiërarchische databestanden*

- Het nadeel van flat-file data bestanden is dat de data in een tabelvorm moet passen.
- Indien data een complexere (vaak hiërarchische) structuur heeft, dan is dit niet evident om correct in een tabelvorm te gieten.
  - Je hebt bijvoorbeeld data over de studenten en van iedere student heb je naamgegevens en de resultaten van de verschillende afgelegde vakken.
  - Het aantal afgelegde vakken verschilt echter van student tot student.
  - Ook per student kan het aantal opgenomen kansen per vak verschillen van vak tot vak.
  - Het aantal scores per student kan hierdoor sterk variëren.
- Voor hiërarchische databestanden wordt daarom vaak gebruikt gemaakt van XML-bestanden of JSON-bestanden.
- XML- en JSON-bestanden zijn ook zeer populair om gegevens via het web uit te wisselen.

### *XML-bestanden*

- Bestudeer de bron over XML (tot en met XML attributes) om te begrijpen hoe een XML-bestand is opgebouwd.
- Een XML-bestand bestaat uit XML-elementen.
  - De naam van het XML-element wordt bepaald door het openings- en sluitingslabel.
  - Het openingslabel volgt het formaat `<element-naam>`.
  - Het sluitingslabel heeft dezelfde naam en volgt het formaat `</element-naam>`.
  - Tussen het openings- en sluitingslabel plaatsen we de inhoud van het XML-element.
- Voorbeeld: `<student>Rob Nelissen</student>`.
  - Hier wordt het XML-element student gedefinieerd.
  - De inhoud van dit XML-element is Rob Nelissen.
- De inhoud van een XML-element kan ook bestaan uit andere XML-elementen. Op deze manier kan je volledige tabellen in XML opslaan. Onderstaande voorbeeld is de vertaling van voorgaande data in tabelvorm naar XML.
- Voorbeeld:

```
<studenten>
  <student>
    <naam>Nelissen</naam>
```

```

<voornaam>Rob</naam>
</student>
<student>
  <naam>Franssen</naam>
  <voornaam>Ann</voornaam>
</student>
</studenten>

```

- Zoals blijkt uit de vergelijking tussen de XML-representatie en voorgaande flat-file representaties, bevat een XML-bestand redelijk veel overhead om tabelvorm-data op te slaan.
- De kracht van XML ten opzichte van de flat-file bestanden is echter dat je veel complexere datastructuren kunt opslaan. Onderstaand voorbeeld opslaan in een tabelvorm (en dus flat-files) is allesbehalve evident.
- Voorbeeld:

```

<studenten>
  <student>
    <naam>Nelissen</naam>
    <voornaam>Rob</voornaam>
    <vakken>
      <vak>
        <naam>Exploratieve en Descriptieve Data Analyse</naam>
        <academiejaar>20162017</academiejaar>
        <score_kans1>8</score_kans1>
        <score_kans2>12</score_kans2>
      </vak>
      <vak>
        <naam>Macro-economie</naam>
        <academiejaar>20152016</academiejaar>
        <score_kans1>8</score_kans1>
        <score_kans2>7</score_kans2>
      </vak>
      <vak>
        <naam>Macro-economie</naam>
        <academiejaar>20162017</academiejaar>
        <score_kans1>14</score_kans1>
      </vak>
      ...
    </vakken>
  </student>
  <student>
    <naam>Franssen</naam>
    <voornaam>Ann</voornaam>
  </student>
</studenten>

```

```

<vakken>
  <vak>
    <naam>Exploratieve en Descriptieve Data Analyse</naam>
    <academiejaar>20162017</academiejaar>
    <score_kans1>15</score_kans1>
  </vak>
  <vak>
    <naam>Macro-economie</naam>
    <academiejaar>20162017</academiejaar>
    <score_kans1>16</score_kans1>
  </vak>
  ...
</vakken>
</student>
...
</studenten>

```

#### *JSON-bestanden*

- Bestudeer de JSON Tutorial om te begrijpen hoe een JSON-bestand is opgebouwd.
- JSON is een ander formaat dat steeds populairder wordt om hiërarchische data op te slaan en uit te wisselen.
  - In vergelijking met XML is JSON korter en eenvoudiger te lezen.
- Een JSON-bestand bestaat voornamelijk uit JSON-objecten en JSON-lijsten.
- JSON-objecten komen typisch overeen met een observatie (rij) in een dataset.
  - Een JSON-object wordt omsloten door accolades.
  - De inhoud van een JSON-object bestaat uit key-value paren.
    - \* De key-value paren zijn van elkaar gescheiden door middel van een komma.
    - \* De key is een string omsloten door dubbele aanhalingsstekens en geeft aan wat de value voorstelt.
    - \* Key en value zijn van elkaar gescheiden door middel van een dubbelpunt.
- Voorbeelden van een JSON-object dat de student Rob Nelissen voorstelt:
  - {"naam": "Nelissen", "voornaam": "Rob"}
- Een JSON-lijst (array genoemd) bestaat uit een lijst van waarden gescheiden door een komma en omgeven door rechte haakjes.

- De waarden van de verschillende elementen in een JSON-lijst moeten van hetzelfde type zijn.
- Toegelaten waarden zijn o.a. strings, getallen, objecten, andere arrays (lijsten).
- Indien je data in tabelvorm wenst voor te stellen, zal je iedere rij als een JSON-object voorstellen en de volledige tabel als een lijst van deze JSON-objecten.
- Onderstaand voorbeeld is de vertaling van voorgaande studentendata in tabelvorm naar JSON.

```
[  
  {"naam": "Nelissen",  
   "voornaam": "Rob"  
  },  
  {"naam": "Franssen",  
   "voornaam": "Ann"  
  }  
]
```

- Net als bij XML kan je met JSON complexe datastructuren voorstellen.
- Voorbeeld:

```
[  
  {"naam": "Nelissen",  
   "voornaam": "Rob",  
   "vakken":  
     [  
       {"naam": "Exploratieve en Descriptieve Data Analyse",  
        "academiejaar": "20162017",  
        "score_kans1": 8,  
        "score_kans2": 12  
       },  
       {"naam": "Macro-economie",  
        "academiejaar": "20152016",  
        "score_kans1": 8,  
        "score_kans2": 7  
       },  
       {"naam": "Macro-economie",  
        "academiejaar": "20162017",  
        "score_kans1": 14  
       }  
     ]  
  },  
  {"naam": "Franssen",  
   "voornaam": "Ann",  
   "vakken":  
     [  
       {"naam": "Exploratieve en Descriptieve Data Analyse",  
        "academiejaar": "20162017",  
        "score_kans1": 10,  
        "score_kans2": 10  
       },  
       {"naam": "Macro-economie",  
        "academiejaar": "20152016",  
        "score_kans1": 7,  
        "score_kans2": 7  
       },  
       {"naam": "Macro-economie",  
        "academiejaar": "20162017",  
        "score_kans1": 14  
       }  
     ]  
  }]
```

```

"vakken": [
    {
        "naam": "Exploratieve en Descriptieve Data Analyse",
        "academiejaar": "20162017",
        "score_kans1": 15
    },
    {
        "naam": "Macro-economie",
        "academiejaar": "20162017",
        "score_kans1": 16
    }
]
}
]

```

### *Applicatie-specifieke dataformaten*

- Naast deze standaard dataformaten, waarbij data als tekstbestanden worden opgeslagen, bestaan er verschillende applicatie-specifieke dataformaten.
- Het voordeel van applicatie-specifieke dataformaten is dat deze extra informatie over de data kunnen opslaan die specifiek is voor de applicatie.
  - Zo zal een Excel databestand ook informatie bevatten over de formules en opmaak van de data (vet, cursief, ...).
- Het nadeel van deze applicatie-specifieke dataformaten is dat ze niet altijd leesbaar zijn door andere applicaties.

### *R-packages voor het inladen van diverse dataformaten*

- Standaard R heeft een aantal functies om delimited bestanden in te lezen, namelijk `read.csv()` en `read.csv2()`. Het verschil tussen beide functies heeft betrekking op de standaardwaarden voor specifieke parameters, zoals welk teken als delimiter gebruikt wordt (`read.csv` veronderstelt het komma-teken als delimiter, terwijl `read.csv2` er van uitgaat dat de puntkomma gebruikt wordt om waarden van elkaar te scheiden).
- Er is ook het R pacakge ‘`readr`’ dat ook twee soortgelijke functies aanbiedt - `read_csv()` en `read_csv2()` - die performanter zijn dan de standaardfuncties.
- Om xml-bestanden in te lezen, wordt typisch het R package ‘`xml`’ gebruikt. Voor JSON-bestanden kan gebruik gemaakt worden van de packages ‘`rjson`’ of ‘`jsonlite`’.
- Voor een aantal applicatie-specifieke formaten zijn ondertussen ook al R-packages ontwikkeld. Zo is er bijvoorbeeld het R-package

‘readxl’ dat het inladen van Excel bestanden relatief eenvoudig maakt.

### 7.2.3 Data-encoding

#### *Binair, decimaal en hexadecimaal rekenstelsel*

- Een computer kan slechts 2 waarden opslaan, typisch voorgesteld als 0 en 1.
- Iedere opslaglocatie op een computer kan dus slechts 2 verschillende waarden opslaan en wordt een bit genoemd.
  - De afkorting van bit is de kleine letter ‘b’.
- Een rekenstelsel waarbij iedere locatie slechts 2 waarden kan voorstellen noemen we een binair rekenstelsel.
  - Indien we 2 opslaglocaties (2 bits) gebruiken, kunnen we 4 verschillende waarden opslaan: 00, 01, 10 en 11.
  - Indien we 3 bits gebruiken zijn er 8 mogelijke waarden, bij 4 bits zijn er 16 mogelijke waarden.
  - Het aantal waarden dat men met  $n$  bits kan opslaan is gelijk aan  $2^n$ .
- Merk op dat in het rekenstelsel dat door mensen gebruikt wordt iedere opslaglocatie 10 verschillende waarden gebruikt kunnen worden (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).
  - Dit noemen we het decimaal rekenstelsel.
  - Met 2 opslaglocaties kunnen we in het decimaal rekenstelsel 100 ( $= 10^2$ ) waarden opslaan: van 00 tot 99.
- Een ander rekenstelsel dat vaak gebruikt wordt binnen computerwetenschappen is het hexadecimaal rekenstelsel.
  - In dit rekenstelsel kan iedere opslaglocatie 16 waardes opslaan, nl. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E en F.
  - Het hexadecimaal rekenstelsel is interessant omdat 1 locatie overeenstemt met 4 bit (4 locaties in het binair rekenstelsel).
- Om in computerprogramma’s aan te geven dat iets voorgesteld wordt in het hexadecimaal stelsel, laten we het voorafgaan door een 0x. Om aan te geven dat iets voorgesteld wordt in het binair stelsel, gebruiken we prefix 0b. Zonder prefix verwijzen we typisch naar het decimaal rekenstelsel.

decimaal	hexadecimaal	binair
0	0x0	0b0000
1	0x1	0b0001

decimaal	hexadecimaal	binair
2	0x2	0b0010
3	0x3	0b0011
4	0x4	0b0100
5	0x5	0b0101
6	0x6	0b0110
7	0x7	0b0111
8	0x8	0b1000
9	0x9	0b1001
10	0xA	0b1010
11	0xB	0b1011
12	0xC	0b1100
13	0xD	0b1101
14	0xE	0b1110
15	0xF	0b1111

Tabel: conversietabel decimaal, hexadecimaal en binair.

- 8 bits worden ook een byte genoemd wat afgekort wordt met de hoofdletter B. 1B bestaat dus uit 8b en kan dus 256 ( $= 2^8$ ) waarden opslaan.

### Tekst opslaan

- Bestudeer de bron over Encoding tot sectie ‘Encodings en PHP’.
- Lees de bron over de geschiedenis van ASCII (enkel sectie ‘A Historical Perspective’).
- Aangezien computers enkel bits kunnen opslaan, hebben we een conversieschema nodig om tekst op te slaan. Ieder letterteken zal moeten omgezet worden naar een string van bits. Dit conversieschema wordt een ‘encoding scheme’ of encoderingsschema genoemd.
- Ieder encoderingsschema voorziet de vertaling van een specifieke set van karakters naar bijhorende bitstrings. Deze sets van karakters noemen we ‘character sets’ of karaktersets.
- Er bestaan zeer veel encoderingsschema’s.
  - ASCII is een van de oudste encoderingsschema’s en is voor- namelijk bruikbaar voor Engelstalige tekst.
    - \* De ASCII karakterset bestaat uit 128 lettertekens en bevat o.a. de cijfers 0 tot 9, de letters a-z en A-Z.
    - \* ASCII gebruikt 1 byte per letterteken en kan dus in principe 256 verschillende lettertekens opslaan.

- \* Aangezien ASCII slechts een karakterset van 128 tekens heeft, gebruikt het dus slechts 7 bit van de beschikbare byte ( $2^7 = 128$ ).
- \* Omdat de ASCII tekenset gemaakt was voor de Engelse taal ontbreken er verschillende tekens voor andere talen.
- De ANSI-standaard nam de ASCII tekenset over, maar voegt hier vervolgens 128 tekens aan toe door de volledige byte te gebruiken.
  - \* Met welke tekens de karakterset wordt uitgebreid ligt echter niet vast binnen de ANSI-standaard, maar is afhankelijk van de gekozen codepage (of karakterset).
  - \* Er bestaan zeer veel ANSI codepages (die eigenlijk Windows codepages genoemd moeten worden). Voor de eerste 128 tekens maakt de specifieke codepage niet uit, maar voor de laatste 128 code pages is dit wel belangrijk.
- Voor talen waar 1 byte per letterteken onvoldoende is, werden dan weer nieuwe encoderingsschema's gebruikt die 2 bytes gebruiken en zo 65536 lettertekens kunnen voorstellen.
- Unicode is een poging om tot 1 karakterset te komen voor alle tekens die gebruikt worden in tekst.
  - \* Unicode is een standaard en definieert zelf geen encoding. Ze vertaalt dus zelf geen lettertekens naar bitstrings.
  - \* Unicode legt wel codepoints vast, wat een mapping is tussen lettertekens en een hexadecimaal getal. Zo is de letter ‘A’ gekoppeld aan het codepoint 0x0041.
  - \* Het Unicode systeem bevat in totaal meer dan 1 miljoen codepoints en omvat niet enkel cijfers en letters, maar ook emoji’s. Zo heeft de ‘lachend gezicht’-emoji codepoint 0x1F642.
  - \* De feitelijke omzetting van de codepoints naar een bitstring gebeurt door een specifieke encoding, waarbij UTF-8 de meest voorkomende is.
- Het gevolg is dat we een grote waaier aan encoderingsschema's hebben.
  - Als je dus een tekst opslaat volgens het ene encoderingsschema en vervolgens terug inleest volgens een ander encoderingsschema, dan kan het zijn dat delen van de tekst geen steek meer houden.
  - Als je bijvoorbeeld de letter ‘i’ opslaat volgens de Windows-1252 codepage dan zal dit binair als 11101111 opgeslagen worden (0xEF). Als je deze reeks van 8 bits echter later weer inleest volgens de Windows-1257 (Windows-Baltic), dan zal de binaire reeks 11101111 geïnterpreteerd worden als ‘l’.
- Het R-package ‘readr’ gaat er van uit dat tekst geëncodeerd is in UTF-8.

### 7.2.4 Datatypes controlleren en corrigeren

- We zullen de datavoorbereidingsfase illustreren aan de hand van de vluchtgegevens van de drie luchthavens in New York City.
- Voor we kunnen beginnen is het altijd verstandig een snel overzicht te maken van de dataset, zodat we weten welke variabelen we voor handen hebben alsook hun datatypes.
- Met de *glimpse* functie krijg je snel een overzicht van de verschillende variabelen en van welk type ze zijn.

```
glimpse(df)
```

```
## # Observations: 329,174
## # Variables: 7
## $ luchthaven      <fct> EWR, LGA, JFK, LGA, EWR, LGA, JFK, LGA, JF...
## $ maatschappij    <fct> United Air Lines Inc., United Air Lines Inc., A...
## $ vertrek_vertraging <dbl> 2, 4, 2, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2...
## $ aankomst_vertraging <dbl> 11, 20, 33, -25, 12, 19, -14, -8, 8, -2, -3, 7, ...
## $ afstand          <chr> "1400", "1416", "1089", "762", "719", "1065", "...
## $ vliegtijd         <dbl> 227, 227, 160, 116, 150, 158, 53, 140, 138, 149...
## $ vluchtttype       <ord> normaal, normaal, kort, kort, kort, kort, ...
```

- Deze output bevat al een opmerkelijk resultaat. Zo zien we dat de variabele ‘afstand’ als tekst is opgeslagen in plaats van als een continue (numerieke) variabele.
- Soms gebeurt het dat R niet het juiste variablename herkent. Zo kan het zijn dat een categorische variabele als numerieke variabele wordt beschouwd omdat de categorieën gehele getallen zijn (vb. aantal cylinders: 4, 6 of 8).
- In onze dataset hebben we opgemerkt dat de variabele ‘afstand’ niet als numerieke variabele wordt geïnterpreteerd, maar als een ‘tekst’-variabele. Dit kan verschillende oorzaken hebben.
  - Zo kan het zijn dat er voor 1 van de observaties een waarde geregistreerd is met een niet-numeriek teken (vb. 1OO ipv 100). In dat geval zal je eerst deze waarden moeten corrigeren naar de juiste waarde.
  - Een andere vaak voorkomende oorzaak is dat R een punt als decimaalteken verwacht, terwijl dat een komma is in de dataset. Dit valt vaak op te lossen door de data met andere opties in te lezen in R.
- Indien de fouten zijn gecorrigeerd, dan moeten we nog altijd de data omzetten van een ‘tekst’-variabele naar een numerieke variabele (in ons geval). Dit doen we door middel van de ‘mutate’-functie en de ‘as.numeric’-functie.

```
df <- df %>%
  mutate(afstand = as.numeric(afstand))
```

- Merk op dat als er toch nog een waarde aanwezig is die niet kan omgezet worden naar het nieuwe variabeletype, R een waarschuwing zal geven en de waarde zal vervangen door ‘NA’. In dat geval ga je eerst moeten zoeken naar de oorzaak van de waarschuwing, deze aanpakken en dan de variabele transformeren naar het nieuwe type.
- De belangrijkste datatypes in R en de bijhorende transformatiefuncties zijn:
  - numeric (decimale getallen) - as.numeric()
  - integer (gehele getallen) - as.integer()
  - character (tekst) - as.character()
  - factor (nominale variabele) - as.factor()
  - ordered factor (ordinale variabele) - as.ordered()

## 7.3 Dataproblemen identificeren en corrigeren

### 7.3.1 Overzicht

- We onderscheiden drie soorten problemen die kunnen opduiken met data en die best op voorhand gecorrigeerd worden:
  - Foutieve waarden.
  - Ontbrekende waarden.
  - Inconsistente waarden.

### 7.3.2 Foutieve waarden

#### Categorische variabele

- Coderingsfouten bij categorische variabelen uiten zich typisch in redundante categorielabels. Dit zijn labels met een typfout die door R als een aparte categorie worden beschouwd, maar dit niet zijn.
- Om dit soort coderingsfouten te detecteren, moet je de verschillende labels van een categorische variabele bestuderen.
  - Omdat deze foute categorielabels meestal uitzonderlijk zijn, kan je best de verschillende categorielabels bekijken volgens stijgende frequentie.
  - Een andere aanpak is de categorielabels alfabetisch te ordenen.
- Eenmaal men deze coderingsfouten gedetecteerd heeft, kan men ze manueel corrigeren door gebruik te maken van de functies *mutate* (dplyr) en *fct\_recode* (forcats).

*Case: Vluchtdata NYC*

- Als we de categorische variabele *maatschappij* analyseren op foutieve labels dan zien we dat 1 vlucht foutief het label ‘Xpress-Jet Airlines Inc.’ heeft gekregen in plaats van ‘ExpressJet Airlines Inc.’.

```
df %>%
  group_by(maatschappij) %>%
  tally() %>%
  arrange(n)
```

maatschappij	n
XpressJet Airlines Inc.	1
Envoi Air	19
SkyWest Airlines Inc.	32
Hawaiian Airlines Inc.	342
Mesa Airlines Inc.	601
Frontier Airlines Inc.	685
Alaska Airlines Inc.	714
AirTran Airways Corporation	3260
Virgin America	5162
Southwest Airlines Co.	12275
Endeavor Air Inc.	18460
US Airways Inc.	20536
Envoy Air	26378
American Airlines Inc.	31327
Delta Air Lines Inc.	46779
JetBlue Airways	50940
ExpressJet Airlines Inc.	54172
United Air Lines Inc.	57491

Table 7.3: Maatschappijen geordend volgens stijgende frequentie.

- Indien we de labels van de categorische variabele *maatschappij* alfabetisch ordenen dan zien we ook dat er een aantal vluchten foutief gecodeerd zijn als ‘Envoi Air’ in plaats van ‘Envoy Air’.

```
df %>%
  group_by(maatschappij) %>%
  tally() %>%
  arrange(maatschappij)
```

- We kunnen deze foutieve labels corrigeren met behulp van de functie `fct_recode` uit het `forcats` package.

Table 7.4: Maatschappij alfabetisch geordend.

maatschappij	n
AirTran Airways Corporation	3260
Alaska Airlines Inc.	714
American Airlines Inc.	31327
Delta Air Lines Inc.	46779
Endeavor Air Inc.	18460
Envoi Air	19
Envoy Air	26378
ExpressJet Airlines Inc.	54172
Frontier Airlines Inc.	685
Hawaiian Airlines Inc.	342
JetBlue Airways	50940
Mesa Airlines Inc.	601
SkyWest Airlines Inc.	32
Southwest Airlines Co.	12275
United Air Lines Inc.	57491
US Airways Inc.	20536
Virgin America	5162
XpressJet Airlines Inc.	1

```
df %>%
  mutate(maatschappij = fct_recode(maatschappij,
                                    "ExpressJet Airlines Inc." = "XpressJet Airlines Inc.",
                                    "Envoy Air" = "Envoi Air")) -> df

df %>%
  group_by(maatschappij) %>%
  tally() %>%
  arrange(maatschappij)
```

### Ordinale variabelen

- Bij ordinale variabelen kunnen dezelfde coderingsfouten voorkomen als bij categorische variabelen. Deze worden op dezelfde manier gedetecteerd en gecorrigeerd.
- Er is echter nog een bijkomende coderingsfout voor ordinale variabelen, namelijk wanneer de voorgedefinieerde volgorde tussen de labels fout is.
- Om dit te detecteren, moet je de verschillende labels ('levels') opvragen met behulp van de *unique*-functie.
- Indien we de ordinale variabele *vluchtype* analyseren dan zien we dat de voorgedefinieerde volgorde van de labels foutief is.

```
unique(df$vluchtype)
## [1] normaal      kort        lang       intercontinentaal
```

```
## Levels: lang < kort < normaal < intercontinentaal

• We kunnen de volgorde tussen de labels van een ordinale variabele corrigeren met behulp van de functies mutate (dplyr) en fct_relevel (forcats).

df %>%
  mutate(vluchttype = fct_relevel(df$vluchttype, "lang", after = 2)) -> df

unique(df$vluchttype)

## [1] normaal          kort            lang           intercontinentaal
## Levels: kort < normaal < lang < intercontinentaal
```

### *Continue variabelen*

- Foutieve waarden bij een continue variabelen detecteren is een stuk moeilijker omdat een foutieve waarde nog steeds een geldige waarde kan zijn (nog steeds een getal).
- Ook de aanpak om naar weinig voorkomende waarden te kijken, zoals bij categorische variabelen, werkt niet goed omdat bij een continue variabele vaak veel waarden zijn zeer weinig voorkomen.
- De meest voor de hand liggende aanpak is de waarden te bestuderen die opmerkelijk hoog of laag zijn in vergelijking met de andere waarden van de variabele.
- Het is belangrijk te beseffen dat niet iedere extreme waarde per definitie een foutieve waarde is. Uitzonderlijk hoge of lage waardes zijn natuurlijk altijd mogelijk.
- Daarom moet men altijd voorzichtig te werk gaan bij het bepalen of iets een foutieve waarde is (meetfout, ingavefout) of een uitzonderlijke doch correcte waarde. Domeinkennis kan hierbij helpen.
- Indien je door te kijken naar de uiterste waarden mogelijke problemen hebt gedetecteerd, moet je deze observaties van nabij bestuderen om te achterhalen of het meetfouten kunnen zijn of niet. Ga hiervoor steeds naar de volledige observatie kijken en niet enkel naar de waarde voor de continue variabele.
- Een andere manier om te detecteren of een continue variabele uitzonderlijke waarden bevat, is door middel van een boxplot. Uitzonderlijk grote/kleine waarden vallen buiten de ‘whiskers’ en worden door punten aangeduid in een boxplot. Let wel op, de filosofie achter uitzonderlijke waarden is gebaseerd op een normale verdeling van de data. Indien de data werkelijk normaal verdeeld is, dan is de kans op een uitzonderlijke waarde slechts 0.7%. Dit betekent echter dat men best op voorhand het histogram bekijkt om te controleren of de data enigszins normaal verdeeld is, alvorens de boxplot te hanteren.

- Bij foutieve waarden van een continue variabele is het vaak niet mogelijk om de correcte waarde af te leiden (zoals bij een categorische variabele). Daarom is de enige juiste correctie deze foutieve waarden te vervangen met “missing values”.
- We zullen de variabele *vliegtijd* analyseren op foutieve waarden.
- We zullen eerst de kleinste waarden bestuderen. Hiervoor selecteren we de 10 vluchten met de kortste vliegtijd en rangschikken deze volgens stijgende vliegtijd. We kijken hierbij niet alleen naar de vliegtijd, maar ook naar de luchthaven, de maatschappij en de afgelegde afstand.

```
df %>%
  arrange(vliegtijd) %>%
  select(luchthaven, maatschappij, afstand, vliegtijd) %>%
  filter(row_number()<11)
```

luchthaven	maatschappij	afstand	vliegtijd
JFK	Endeavor Air Inc.	94	0.5833333
EWR	JetBlue Airways	1065	2.7666667
JFK	Delta Air Lines Inc.	2248	5.1500000
EWR	ExpressJet Airlines Inc.	116	20.0000000
EWR	ExpressJet Airlines Inc.	116	20.0000000
EWR	ExpressJet Airlines Inc.	116	21.0000000
EWR	ExpressJet Airlines Inc.	80	21.0000000
EWR	ExpressJet Airlines Inc.	116	21.0000000
EWR	ExpressJet Airlines Inc.	80	21.0000000
LGA	US Airways Inc.	184	21.0000000

Table 7.5: Vluchten met kortste vliegtijd.

- Deze analyse doet vermoeden dat de eerste drie vluchten waarschijnlijk meetfouten zijn. Het betreffen hier drie vluchten van minder dan 6 minuten wat zeer onwaarschijnlijk is, zeker wanneer we zien dat de tweede en derde vlucht lange vluchten zijn.
- De overige vluchten zijn vluchten van 20 minuten of meer, maar aangezien het hier om korte vluchten gaan is dit mogelijk correct. We zullen enkel de eerste drie observaties (met een vliegtijd kleiner dan 6) als foutief beschouwen.
- We bestuderen vervolgens de vluchten met de grootste vliegtijd.

```
df %>%
  arrange(-vliegtijd) %>%
  select(luchthaven, maatschappij, afstand, vliegtijd) %>%
  filter(row_number()<11)
```

luchthaven	maatschappij	afstand	vliegtijd
EWR	United Air Lines Inc.	4963	695
JFK	Hawaiian Airlines Inc.	4983	691
JFK	Hawaiian Airlines Inc.	4983	686
JFK	Hawaiian Airlines Inc.	4983	686
JFK	Hawaiian Airlines Inc.	4983	683
JFK	Hawaiian Airlines Inc.	4983	679
EWR	United Air Lines Inc.	4963	676
JFK	Hawaiian Airlines Inc.	4983	676
JFK	Hawaiian Airlines Inc.	4983	675
EWR	United Air Lines Inc.	4963	671

Table 7.6: Vluchten met langste vliegtijd.

- Deze resultaten doen vermoeden dat het hier NIET om meetfouten gaat. Het gaat hier immers om zeer verre vluchten en de maatschappijnaam doet vermoeden dat het hoofdzakelijk vluchten naar Hawaï zijn. We hebben daarom via Google opgezocht hoe lang een vlucht van New York naar Hawaï duurt en dit komt overeen met de vliegtijden van 11 tot 12u in deze dataset. Daarom besluiten we dat deze waarden geen foutieve waarden zijn.
- Vervolgens zullen we voor de drie vluchten met een vliegtijd van minder dan 6 minuten de waarde van de vliegtijd vervangen door een ontbrekende waarde. In R wordt dit aangegeven door de waarde *NA* dat voor ‘not available’ staat.

```
df <- df %>%
  mutate(vliegtijd = ifelse(vliegtijd<6,NA,vliegtijd))
```

```
df %>%
  arrange(vliegtijd) %>%
  select(luchthaven, maatschappij, afstand, vliegtijd) %>%
  filter(row_number()<11)
```

luchthaven	maatschappij	afstand	vliegtijd
EWR	ExpressJet Airlines Inc.	116	20
EWR	ExpressJet Airlines Inc.	116	20
EWR	ExpressJet Airlines Inc.	116	21
EWR	ExpressJet Airlines Inc.	80	21
EWR	ExpressJet Airlines Inc.	116	21
EWR	ExpressJet Airlines Inc.	80	21
LGA	US Airways Inc.	184	21
JFK	Endeavor Air Inc.	94	21
EWR	ExpressJet Airlines Inc.	116	21
EWR	ExpressJet Airlines Inc.	116	21

Table 7.7: Vluchten met kortste vliegtijd.

### 7.3.3 Ontbrekende waarden

- Soms gebeurt het dat voor bepaalde observaties waarden ontbreken voor een specifieke variabele. In zulke gevallen spreken we van ontbrekende waarden of missing values.
- Het detecteren van ontbrekende waarden is relatief eenvoudig, omdat deze normaal als *NA* gecodeerd zijn in een dataset (*NA* = ‘not available’).
- We kunnen onderscheid maken tussen drie soorten van ontbrekende waarden.
  - Missing completely at random (MCAR): Indien het ontbreken van waarden voor een specifieke variabele volledig willekeurig is, dan spreekt men over MCAR.
  - Missing at random (MAR): Indien het ontbreken van waarden voor variabele  $X_1$  niet willekeurig is, maar afhankelijk van de waarden van andere variabelen  $X_2, X_3, \dots$ , dan spreekt men over MAR.
  - Not missing at random (NMAR): Indien het ontbreken van waarden voor variabele  $X_1$  niet willekeurig is, maar afhankelijk is van de waarde van  $X_1$  of van de waarden van ongeobserveerde variabelen, dan spreekt men over NMAR.
- Om te bepalen of data al dan niet MCAR is, moet men achterhalen of het ontbreken van waarden voor variabele  $X_1$  gecorreleerd is met de waarden van een andere variabele  $X_2$ . Een mogelijkheid is om de dataset in twee te splitsen, i.e. alle observaties met een waarde voor  $X_1$  en alle observaties met een missing value voor  $X_1$ . Vervolgens kijken we naar de verdeling van variabele  $X_2$ . Indien deze hetzelfde is voor beide datasets, dan suggereert dit dat er geen relatie bestaat tussen de waarde van  $X_2$  en het al dan niet ontbreken van de waarde voor  $X_1$ . Indien deze verdeling van  $X_2$  sterkt verschilt tussen beide datasets, dan is er mogelijk wel een relatie en dan is de data niet MCAR.
- Het soort ontbrekende waarde heeft belangrijke implicaties hoe je correct met ontbrekende waarden kan omgaan in het kader van confirmatorische data analyse. Zo zal het ‘weglaten’ van observaties met ontbrekende waarden enkel in het geval van MCAR geen vertrekking geven in de resultaten van een confirmatorische data analyse.
- In het kader van een descriptieve of exploratieve data analyse, zijn de implicaties eerder beperkt, omdat men toch enkel uitspraken wenst te doen voor de beschikbare data.
- Wel kan het identificeren van het type ontbrekende waarden op zich interessante inzichten geven. Zo kan het het patroon dat het jaarsalaris voornamelijk ontbreekt bij mensen die hogere studies

gevolgd hebben op zich ook interessant zijn voor verdere interpretatie.

- In descriptieve en exploratieve analyses zijn er 3 manieren om met ontbrekende waarden om te gaan:
  - We verwijderen de variabele waarvoor we missing values hebben.
  - We verwijderen de observaties met missing values.
  - We beschouwen de missing values als een aparte waarde.
- Het verwijderen van de variabele zelf is een drastische maatregel.  
Dit betekent immers dat we de variabele volledig buiten beschouwing laten in onze analyse. Dit is vaak het laatste redmiddel en wordt enkel toegepast als er een te hoog percentage ontbrekende waarden is.
- Bij het verwijderen van observaties moet men met de nodige aandacht te werk gaan. Indien de ontbrekende waarden MAR zijn (en niet MCAR), dan gaat men mogelijk waardevolle patronen tussen andere variabelen ook verwijderen. Een mogelijke manier om dit te omzeilen is de observaties met ontbrekende waarden te negeren bij analyses van de variabelen waarvoor de waarden ontbreken. Dit zorgt ervoor dat deze observaties wel nog beschikbaar zijn voor de analyse van andere variabelen.
- Indien de data suggereert dat de ontbrekende waarden MCAR zijn, dan kan men overwegen deze observaties te verwijderen. Indien dit niet het geval is, dan is het beter deze NA-waarden als een aparte categorie te beschouwen.
  - Omdat R de waarde ‘NA’ anders behandelt dan reguliere waarden, is het vaak aangeraden om deze waarde te transformeren (indien je de ontbrekende waarden als een aparte categorie wenst te beschouwen).
  - In geval van een categorische variabele, kan je de ‘NA’ waarde transformeren naar een aparte categorie (vb ‘waarde ontbreekt’).
  - In geval van een continue variabele, is het aangeraden een nieuwe categorische variabele aan te maken die aangeeft of er wel of niet een waarde aanwezig was voor de continue variabele.
- De eerste stap is na te gaan welke variabelen ontbrekende waarden hebben en hoe vaak deze variabelen ontbrekende waarden hebben.  
Dit functie summary() is hiervoor zeker nuttig.

```
summary(df)
```

```
##   luchthaven           maatschappij vertrek_vertraging
##   EWR:119282    United Air Lines Inc. :57491   Min.   : -43.00
##   JFK:105230    ExpressJet Airlines Inc.:54173   1st Qu.:  -5.00
##   LGA:104662    JetBlue Airways       :50940   Median : -2.00
```

```

##          Delta Air Lines Inc.    :46779   Mean    : 12.71
##          American Airlines Inc. :31327   3rd Qu.: 11.00
##          Envoy Air            :26397   Max.    :1301.00
##          (Other)              :62067   NA's     :8214
##  aankomst_vertraging      afstand      vliegtijd                  vluchtype
##  Min.    : -86.000   Min.    : 17   Min.    : 20.0   kort             :245666
##  1st Qu.: -17.000   1st Qu.: 502  1st Qu.: 81.0   normaal        : 31813
##  Median  : -5.000   Median  : 820  Median  :127.0   lang            : 50980
##  Mean    :  6.987   Mean    :1027  Mean    :149.6   intercontinentaal:  715
##  3rd Qu.: 14.000   3rd Qu.:1372  3rd Qu.:184.0
##  Max.    :1272.000  Max.    :4983  Max.    :695.0
##  NA's    :9365       NA's    :9368  NA's    :9368

```

- Uit deze analyse blijkt dat de variabelen *vertrek\_vertraging*, *aankomst\_vertraging* en *vliegtijd* last hebben van ontbrekende waarden.
- Om vervolgens te analyseren of deze ontbrekende waarden MCAR zijn of niet, zullen we voor ieder van de drie continue variabelen een nieuwe categorische variabele maken die aangeeft of de waarde ontbreekt of niet.

```

df <- df %>%
  mutate(vertrek_vertraging_na = is.na(vertrek_vertraging),
        aankomst_vertraging_na = is.na(aankomst_vertraging),
        vliegtijd_na = is.na(vliegtijd))

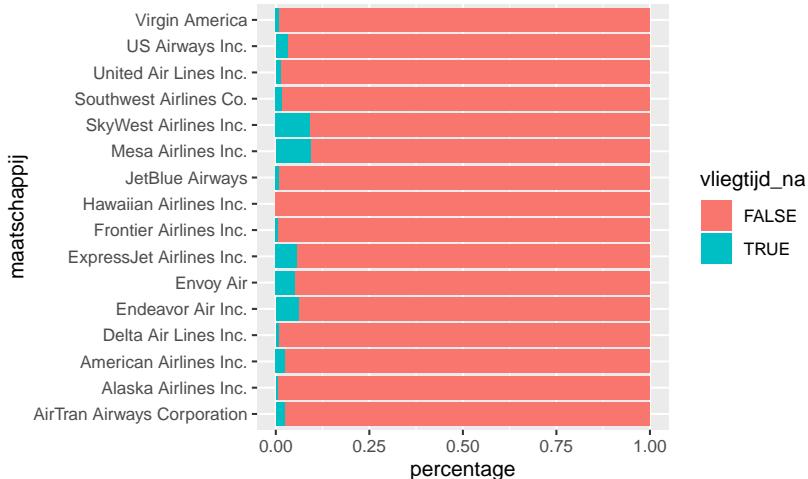
```

- Nu kunnen we met ggplot achterhalen of de andere variabelen zich ‘anders’ gedragen als er voor één van deze drie variabelen een waarde ontbreekt.
- We illustreren voor ‘afstand’ (continu) en voor ‘maatschappij’ (categorisch).
- Indien we dit bestuderen voor ‘maatschappij’ gaan we voor iedere maatschappij laten zien welk percentage cases een ontbrekende waarde voor *vliegtijd* heeft. Indien er geen verband is, dan zouden we geen grote verschillen mogen zien tussen de maatschappijen.

```

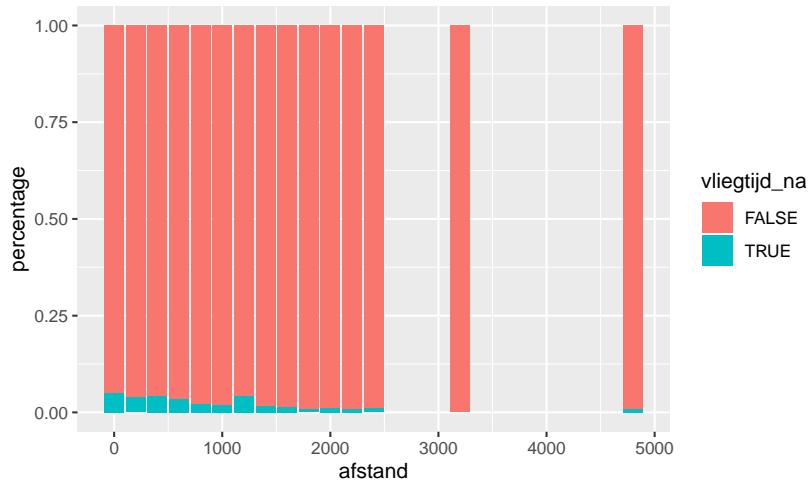
df %>%
  group_by(maatschappij, vliegtijd_na) %>%
  summarise(aantal = n())%>%
  ungroup() %>%
  group_by(maatschappij) %>%
  mutate(totaal = sum(aantal), percentage = aantal/totaal) %>%
  ggplot(aes(x=maatschappij, y=percentage)) +
  coord_flip() +
  geom_col(aes(fill=vliegtijd_na), position = "stack")

```



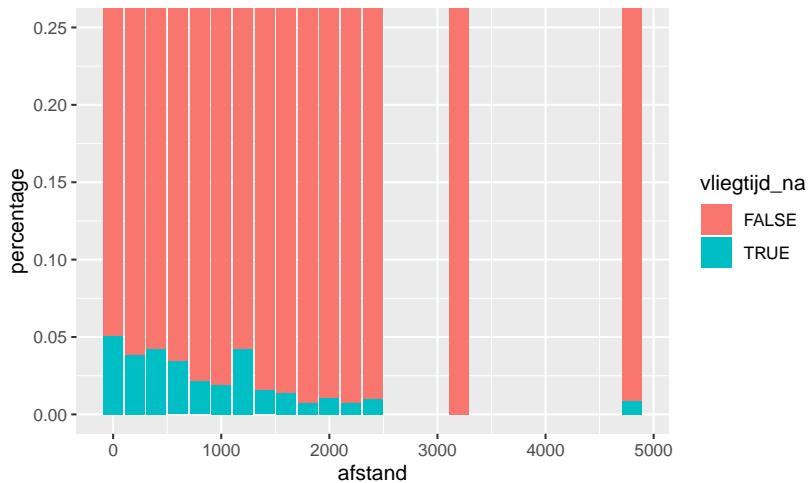
- Uit deze resultaten blijkt dat voor sommige maatschappijen een aanzienlijk hoger percentage ontbrekende waarden bij vliegtijd voorkomt (SkyWest, Mesa en ook ExpressJet, Envoy en Endeavor).
- We kunnen een soortgelijke analyse ook uitvoeren voor de variabele *afstand*. Hiervoor zullen we eerst de variabele *afstand* omvormen tot een categorische variabele.

```
binwidth <- 200
df %>%
  mutate(afstand_cat = afstand %% binwidth) %>%
  group_by(afstand_cat, vliegtijd_na) %>%
  summarise(aantal = n()) %>%
  ungroup() %>%
  group_by(afstand_cat) %>%
  mutate(totaal = sum(aantal), rel_aantal = aantal/totaal) %>%
  ggplot(aes(x=afstand_cat*binwidth, y=rel_aantal)) +
  geom_col(aes(fill=vliegtijd_na), position = "stack") +
  xlab("afstand")+
  ylab("percentage")
```



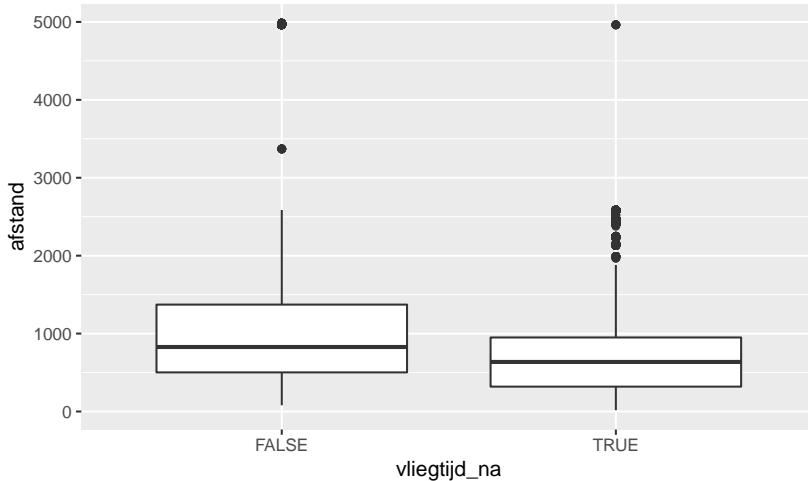
- Deze resultaten lijken te suggereren dat naarmate de vlucht langer wordt, het percentage ontbrekende waarden bij *vliegtijd* afneemt (met een uitzonderlijke piek bij vluchten rond 1200 mijl).
- Omdat het percentage ontbrekende waarden eerder klein is, is het moeilijk om het patroon duidelijk te zien. We kunnen ook dezelfde plot maken, maar de y-as laten stoppen bij een waarde van 0.25. Op deze manier wordt het patroon duidelijker.

```
binwidth <- 200
df %>%
  mutate(afstand_cat = afstand %% binwidth) %>%
  group_by(afstand_cat, vliegtijd_na) %>%
  summarise(aantal = n()) %>%
  ungroup() %>%
  group_by(afstand_cat) %>%
  mutate(totaal = sum(aantal), rel_aantal = aantal/totaal) %>%
  ggplot(aes(x=afstand_cat*binwidth, y=rel_aantal)) +
  geom_col(aes(fill=vliegtijd_na), position = "stack") +
  xlab("afstand")+
  ylab("percentage")+
  coord_cartesian(ylim = c(0, 0.25))
```



- Tenslotte kunnen we ook nog op een andere manier het verband tussen de *afstand* en het voorkomen van ontbrekende waarden bij *vliegtijd* bestuderen, nl. via 2 boxplots.

```
df %>%
  ggplot(aes(x=vliegtijd_na, y=afstand)) +
  geom_boxplot()
```



- Hier zien we dat vluchten waarvoor de vliegtijd ontbreekt vaak kortere vluchten zijn dan waarvoor we de vliegtijd wel hebben. Dit komt overeen met de vorige bevinding.
- Op basis van deze resultaten kunnen we dus stellen dat het ontbreken van de vliegtijd niet willekeurig is, maar vaker voorkomt bij bepaalde maatschappijen en eerder bij kortere dan bij langere vluchten.
- We zouden nog verder kunnen onderzoeken of deze maatschappijen eerder langere of kortere vluchten organiseren.

- Soortgelijke analyses kunnen we ook uitvoeren voor de variabelen *vertrek\_vertraging* en *aankomst\_vertraging*.

#### 7.3.4 Inconsistente waarden

- Data is inconsistent als het niet voldoet aan een aantal regels/beperkingen die horen te gelden op basis van domeinkennis.
- De vorm van inconsistenties waar we ons op focussen, betreft in-record inconsistenties. Dit zijn tegenstrijdigheden die aanwezig zijn binnen één enkele observatie. Enkele voorbeelden zijn:
  - De gemiddelde snelheid van een vlucht ligt hoger dan de maximale theoretische snelheid van het vliegtuig.
  - Het aankomsttijdstip van een vlucht vindt plaats voor het vertrektijdstip.
  - De aankomsttijdstip komt niet overeen met het vertrektijdstip + vertrekvertraging + vluchtduur.
- Het identificeren van inconsistenties kan door middel van diverse dplyr-functies, waarbij je voor iedere observatie test of deze voldoen aan de opgelegde beperkingsregel.
- Daarnaast is er ook het editrules package dat nuttige functies biedt om op een gestructureerdere manier consistentie te evalueren.

#### 7.4 Data opwaarderen

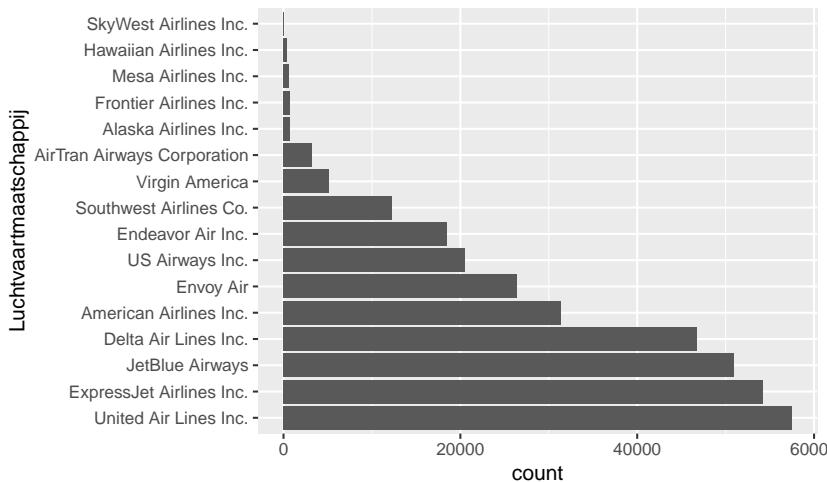
- Van zodra de data geen foutieve en/of ontbrekende waarde meer bevat, kunnen we een aantal technieken toepassen om de data bruikbaarder te maken voor exploratieve analyses. We onderscheiden hierbij 2 technieken:
  - Transformatie van bestaande variabelen.
  - Selectie van observaties.

#### Categorische variabelen

- Soms is het beter om de categorieën van een categorische variablen te wijzigen door sommige categorieën samen te nemen. Er zijn verschillende situaties waarbij dit het overwegen waard is, zoals:
  - De labels van een categorische variabele is op een te gedetailleerd niveau gedefinieerd, met als gevolg dat de exploratieve analyse al snel complex wordt door de vele categorieën. In zulke gevallen kan het zinvol zijn om het aantal categorieën te verminderen door categorieën die inhoudelijk bij elkaar horen samen te nemen.

- Een categorische variabele bestaat uit een beperkt aantal categorieën met veel observaties en een groot aantal categorieën met zeer weinig observaties. In zulke gevallen kan het zinvol zijn om de categorieën met weinig observaties samen te nemen in 1 categorie “Overige”.
- Om te bepalen welke categorieën men kan samenvoegen, kan een frequentietabel of barplot gemaakt worden.
- Het herdefiniëren van de labels gebeurt vervolgens met de functie `fct_recode` (forcats). Hierbij heeft men steeds de keuze om de oorspronkelijke variabele te vervangen of een nieuwe variabele aan te maken.
- Laten we eens aan de hand van een barplot naar de variabele ‘luchtvaartmaatschappij’ kijken. We zien hierbij dat er relatief veel luchtvaartmaatschappijen (categorieën) in onze data zijn en dat er een aantal verwaarloosbaar weinig vluchten bevatten.

```
df %>%
  ggplot(aes(x=fct_infreq(maatschappij))) +
  geom_bar() +
  coord_flip() +
  xlab("Luchtvaartmaatschappij")
```



- We kunnen de exacte aantallen achterhalen met behulp van een frequentietabel.

```
df %>%
  group_by(maatschappij) %>%
  summarise(n = n()) %>%
  arrange(n)
```

maatschappij	n
SkyWest Airlines Inc.	32
Hawaiian Airlines Inc.	342
Mesa Airlines Inc.	601
Frontier Airlines Inc.	685
Alaska Airlines Inc.	714
AirTran Airways Corporation	3260
Virgin America	5162
Southwest Airlines Co.	12275
Endeavor Air Inc.	18460
US Airways Inc.	20536
Envoy Air	26397
American Airlines Inc.	31327
Delta Air Lines Inc.	46779
JetBlue Airways	50940
ExpressJet Airlines Inc.	54173
United Air Lines Inc.	57491

Table 7.8: Luchtvaartmaatschappijen geordend volgens stijgend aantal vluchten.

- Op basis van deze analyse beslissen we om de luchtvaartmaatschappijen met minder dan 10000 vluchten samen te voegen in een nieuwe categorie met het label “Overige”. We opteren ervoor de oorspronkelijke variabelen te vervangen.

```
df %>%
  mutate(maatschappij = fct_lump(maatschappij,
                                   n = 9,
                                   other_level = "Overige")) -> df
```

- De nieuwe frequentietabel toont het resultaat.

```
df.complete %>%
  group_by(maatschappij) %>%
  summarise(n = n()) %>%
  arrange(n)
```

### Continue variabelen

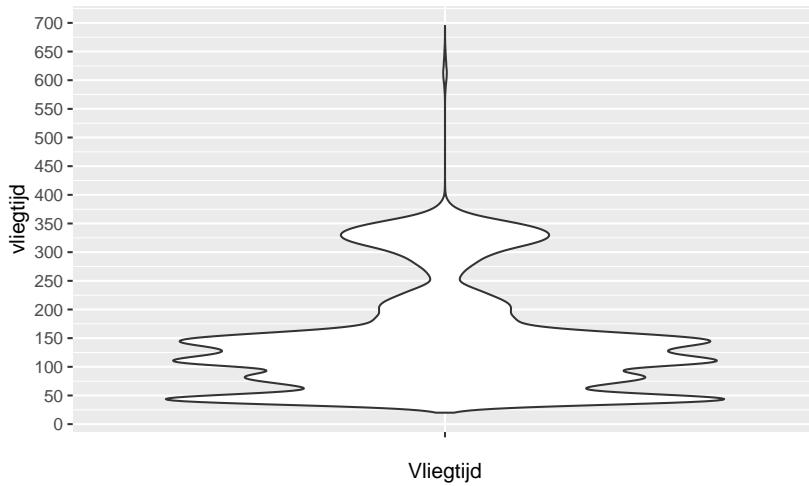
- Bij continue variabelen zijn er verschillende transformaties die regelmatig uitgevoerd worden:
  - De transformatie van een continue variabele naar een categorische variabele.
  - Het herschalen van de continue variabele.
  - De creatie van een nieuwe variabele op basis van bestaande continue variabelen.

maatschappij	n
Overige	10796
Southwest Airlines Co.	12275
Endeavor Air Inc.	18460
US Airways Inc.	20536
Envoy Air	26397
American Airlines Inc.	31327
Delta Air Lines Inc.	46779
JetBlue Airways	50940
ExpressJet Airlines Inc.	54173
United Air Lines Inc.	57491

Table 7.9: Luchtvaartmaatschappijen geordend volgens stijgend aantal vluchten.

- Ook hier hebben we weer steeds de mogelijkheid om de bestaande variabele te vervangen of een nieuwe variabele aan te maken.
- Laten we de variabele vliegtijd eens onder de loep nemen. We beginnen met een visuele analyse aan de hand van een violinplot.

```
df %>%
  ggplot(aes(x="", y=vliegtijd)) +
  geom_violin() +
  xlab("Vliegtijd") +
  scale_y_continuous(breaks=seq(0,800,50))
```



- Op basis van deze plot beslissen we een nieuwe categorische variabele “vliegtijd\_fct” aan te maken, waarbij ‘kort’ overeenkomt met een vlucht die minder dan een uur duurt, ‘normaal’ overeenkomt met een vlucht tussen 1 en 4 uur (60-240) en ‘lang’ overeenkomt met een vlucht van meer dan 4 uur. Hiervoor maken we gebruik van de functie *cut*.

```
df %>%
```

```
mutate(vliegtijd_fct = cut(vliegtijd, c(-Inf,60,240,Inf),
                           labels=c('kort','normaal','lang'))) -> df
```

- Aan de hand van een frequentietabel kunnen we nu het resultaat bekijken.

```
df %>%
  group_by(vliegtijd_fct) %>%
  tally()
```

vliegtijd_fct	n
kort	53220
normaal	210758
lang	55828
NA	9368

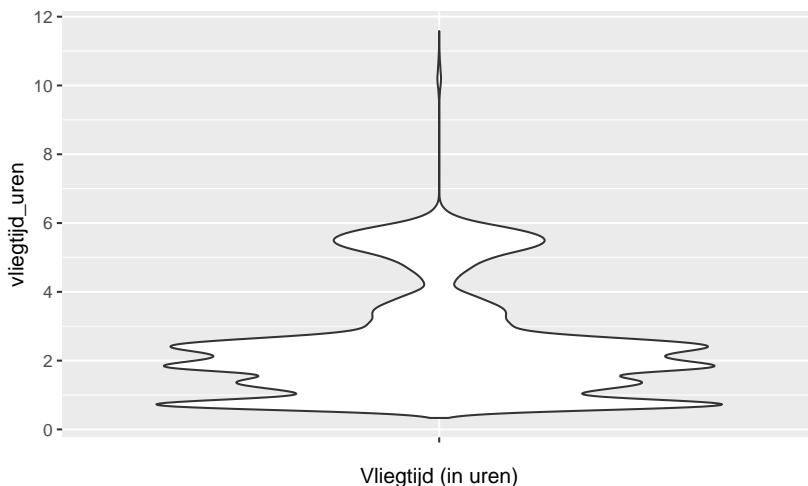
Table 7.10: Frequentietabel vliegtijd (factor)

- Vervolgens beslissen we een nieuwe variabele te maken die de vliegtijd uitdrukt in uren in plaats van minuten.

```
df %>%
  mutate(vliegtijd_uren = vliegtijd/60) -> df
```

- We kunnen het resultaat bekijken met een violinplot.

```
df %>%
  ggplot(aes(x="", y=vliegtijd_uren)) +
  geom_violin() +
  xlab("Vliegtijd (in uren)") +
  scale_y_continuous(breaks=seq(0,12,2))
```

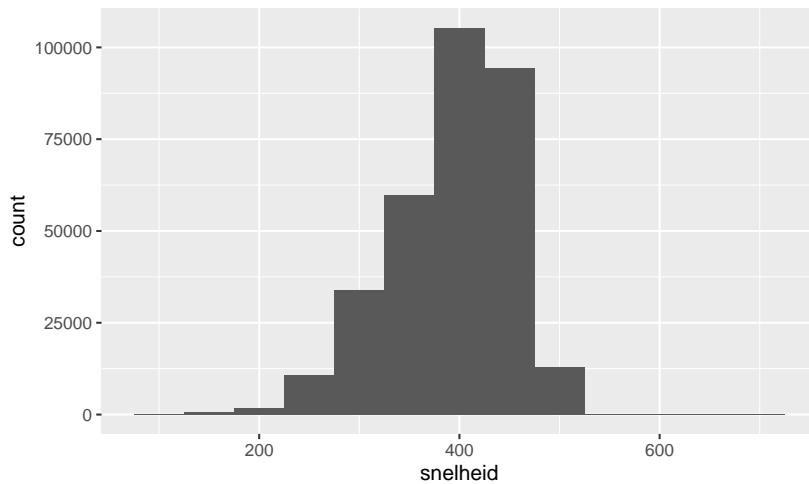


- Tenslotte maken we een nieuwe variabele die de gemiddelde snelheid van het vliegtuig uitdrukt door de afstand te delen door de vliegtijd.

```
df %>%
  mutate(snelheid = afstand / vliegtijd_uren) -> df
```

- Laten we het resultaat aan de hand van een histogram bekijken.

```
df %>%
  ggplot(aes(x=snelheid)) +
  geom_histogram(binwidth = 50) +
  scale_x_continuous(breaks = seq(0,3000, 200))
```



#### 7.4.1 Sampling

- Soms is een dataset zo groot, dat analyses veel tijd in beslag nemen. In zulke gevallen kan het nuttig zijn om een random sample te nemen van de oorspronkelijke data om een eerste exploratieve analyse op uit te voeren.
- Zolang de sample willekeurig getrokken wordt en de nieuwe dataset niet te klein wordt, is de kans dat je patronen ontdekt in de sample die niet voorkomen in de volledige dataset eerder klein.
- Na een eerste exploratieve data analyse op de beperkte sample, kan men vervolgens gerichter de volledige dataset analyseren.
- Laten we een sample van 10000 vluchten nemen uit de oorspronkelijke dataset.

```
df.10000 <- df %>% sample_n(10000)
```

- We kunnen nu een eerste blik op deze sample werpen met behulp van de *summary* functie.

```
summary(df.10000)

##   luchthaven                  maatschappij  vertrek_vertraging
##   EWR:3499   United Air Lines Inc.    :1663   Min.   :-20.00
##   JFK:3259   ExpressJet Airlines Inc.:1613   1st Qu.: -5.00
##   LGA:3242   JetBlue Airways       :1563   Median  : -2.00
##               Delta Air Lines Inc.   :1427   Mean    : 12.09
##               American Airlines Inc. : 976   3rd Qu.: 11.00
##               Envoy Air            : 824   Max.   :533.00
##               (Other)             :1934   NA's    :239
##   aankomst_vertraging      afstand      vliegtijd          vluchtype
##   Min.   :-71.00      Min.    : 94   Min.   : 21.0   kort      :7406
##   1st Qu.:-17.00      1st Qu.: 502  1st Qu.: 82.0   normaal   :1011
##   Median  :-5.00      Median  : 828  Median  :128.0   lang      :1560
##   Mean    : 6.68      Mean    :1033  Mean    :150.8   intercontinentaal: 23
##   3rd Qu.: 14.00      3rd Qu.:1372  3rd Qu.:188.0
##   Max.   :499.00      Max.    :4983  Max.   :661.0
##   NA's    :275        NA's    :275   NA's    :275
##   vertrek_vertraging_na aankomst_vertraging_na vliegtijd_na  vliegtijd_fct
##   Mode :logical      Mode :logical      Mode :logical   kort   :1602
##   FALSE:9761         FALSE:9725       FALSE:9725    normaal:6400
##   TRUE :239          TRUE :275        TRUE :275     lang   :1723
##                           NA's   : 275
##
##   vliegtijd_uren      snelheid
##   Min.   : 0.350  Min.   :117.5
##   1st Qu.: 1.367  1st Qu.:356.1
##   Median  : 2.133  Median :402.6
##   Mean    : 2.513  Mean   :392.2
##   3rd Qu.: 3.133  3rd Qu.:435.8
##   Max.   :11.017  Max.   :519.2
##   NA's    :275    NA's   :275
```

- Als we dit vergelijken met de volledige dataset, dan zien we relatief weinig verschillen wat betreft de centrummaten en de robuste spreidingsmaten.
- Merk op dat minima's en maxima's wel sterk kunnen verschillen.  
Dit is omdat dit geen robuste maatstaven zijn.

```
summary(df)

##   luchthaven                  maatschappij  vertrek_vertraging
##   EWR:119282   United Air Lines Inc.    :57491   Min.   :-43.00
##   JFK:105230   ExpressJet Airlines Inc.:54173   1st Qu.: -5.00
```

```

##   LGA:104662    JetBlue Airways      :50940  Median : -2.00
##                               Delta Air Lines Inc.   :46779  Mean   : 12.71
##                               American Airlines Inc. :31327  3rd Qu.: 11.00
##                               Envoy Air        :26397  Max.   :1301.00
##                               (Other)          :62067  NA's   :8214
##   aankomst_vertraging    afstand       vliegtijd           vluchtype
##   Min.   : -86.000   Min.   : 17   Min.   : 20.0   kort            :245666
##   1st Qu.: -17.000   1st Qu.: 502  1st Qu.: 81.0   normaal         : 31813
##   Median : -5.000    Median : 820  Median :127.0   lang             : 50980
##   Mean   :  6.987    Mean   :1027  Mean   :149.6   intercontinentaal:  715
##   3rd Qu.: 14.000    3rd Qu.:1372  3rd Qu.:184.0
##   Max.   :1272.000   Max.   :4983  Max.   :695.0
##   NA's   :9365        NA's   :9368  NA's   :9368
##   vertrek_vertraging_na aankomst_vertraging_na vliegtijd_na   vliegtijd_fct
##   Mode :logical      Mode :logical      Mode :logical   kort   : 53220
##   FALSE:320960        FALSE:319809        FALSE:319806  normaal:210758
##   TRUE :8214          TRUE :9365          TRUE :9368    lang   : 55828
##   NA's   :9368        NA's   :9368  NA's   :9368
##
##
##
##   vliegtijd_uren      snelheid
##   Min.   : 0.333   Min.   : 76.8
##   1st Qu.: 1.350   1st Qu.:356.3
##   Median : 2.117   Median :402.6
##   Mean   : 2.493   Mean   :392.1
##   3rd Qu.: 3.067   3rd Qu.:436.2
##   Max.   :11.583   Max.   :703.4
##   NA's   :9368        NA's   :9368

```

## Referenties

1. Encoding
2. De geschiedenis van ASCII
3. Windows code pages
4. Data importeren in R
5. Delimited en fixed-width bestanden
6. XML
7. JSON Tutorial
8. Unique-functie
9. Factors
10. Fct\_relevel
11. From continuous to categorical

# 8

## Tutorial - Data voorbereiden

### 8.1 Before you start

During this tutorial, we'll use several r-packages. Make sure to install and load them, if needed.

```
library(ggplot2)
library(dplyr)
library(forcats)
library(mice)
```

The `forcats` package contains a series of functions to easily manipulate factors (of which `forcats` is an anagram). The `mice` package can be used to analyze the occurrence of missing values (*MICE* stands for *Multiple Imputation by Chained Equations*, refering to a technique to estimate missing values).

We also assume that you are familiar with the content of both our `ggplot2` and `dplyr` tutorial. You can read the `survey`<sup>1</sup> dataset provided with this tutorial if you want to try things out yourself.[^try]

If you want to try things yourself, make sure to follow the tutorial step by step. The incremental nature of the cleaning and transformation process does not allow to perform parts of this tutorial in isolation.

```
survey <- readRDS("survey.RDS")
glimpse(survey)

## Observations: 21,483
## Variables: 9
## $ year      <chr> "2000", "2000", "2000", "2000", "2000", "2000", "2000", "20...
## $ marital   <chr> "Never married", "Divorced", "Widowed", "Never married", "D...
## $ age       <chr> "26", "48", "67", "39", "25", "25", "36", "44", "44", "47",...
## $ race      <chr> "White", "White", "White", "White", "White", "White", "White", "Whit...
## $ rincome   <chr> "$8000 to 9999", "$8000 to 9999", "Not applicable", "Not ap...
```

<sup>1</sup> The `survey` data used comes from the General Social Survey, and contains general information about social aspects of the American citizens. In particular, the `survey` data.frame contains a sample of categorical attributes.

```
## $ partyid <chr> "Ind,near rep", "Not str republican", "Independent", "Ind,n...
## $ relig    <chr> "Protestant", "Protestant", "Protestant", "Orthodox-christi...
## $ denom    <chr> "Southern baptist", "Baptist-dk which", "No denomination", ...
## $ tvhours  <chr> "12", NA, "2", "4", "1", NA, "3", NA, "0", "3", "2", NA, "1...
```

## 8.2 Introduction

This tutorial on cleaning and transformation is divided into three different sections.

- Reading
- Cleaning
- Transforming

**Reading.** The first task is to read the data provided in a particular format into R. While we will not cover this topic exhaustively, we will give useful pointers to appropriate R-packages to do so.

**Cleaning.** We *clean* the data by: removing mistakes, duplicates, etc.

**Transforming.** Here we do not target the removal of mistakes and inconsistencies, but rather try to make the data easier to analyse: creating discrete representations of continuous variables, adding calculated variables, or recoding the labels of categorical variables.

These steps in general take place before we do any of the analysis or visualizations which we saw in the ggplot2 and dplyr tutorials, although often multiple iterations are needed. Until now, we always received our data in a fairly clean state, but that is rarely the case in reality. Now, it's time to do our own cleaning. Let's go ahead!

## 8.3 Reading data

For our purpose, we will not discuss different data formats and how to read them at length.<sup>2</sup> Mostly we will use the `readRDS` function, which you probably have seen before. For example,

```
survey <- readRDS("survey.RDS")
```

A .RDS-file stores a **single** R-object in a serialized way. (RDS can be thought of a R Data Serialized). We can create an .RDS-file using `saveRDS` and read one using `readRDS`. Information systems will **never** export data as .RDS files – all .RDS-files are created within R. All the .RDS-files you have been using in the exercises and tutorials have been prepared by us. So, in which type of files can data be found in the wild? Let's give you a quick tour of common file formats.

<sup>2</sup> The section on reading data should be seen as background material for when you need it. You are only expected to be familiar with the functions and formats discussion in the classes. However, if you need to import a particular data file in your future career, you can use this as a starting point. The contents of this section can therefore partly be skipped, **except** for the part on converting variables.

### 8.3.1 CSV and TSV

CSV-files are probably thé most common type of data files. CSV stand for Comma Separated Values. These files can be seen as ordinary text files, where each line is an observation, i.e. a row, and columns are separated with commas (therefore its name). The first row can contain the names of the column, although this is not necessary. TSV is a much less common variant, which stands for Tab Separated Values. As its name suggests, values in this files are not seperated with commas but with tabs.

For CSV-files, there are two varying import functions: `read.csv` and `read.csv2`. The first is for regular comma separated files, while the latter is for semicolon seperated files. Otherwise, the usage is similar to `readRDS`. The functions for TSV are similar – just with a T instead of a C.

```
data <- read.csv("path/to/data/file.csv")
```

```
data <- read.csv2("path/to/data/file.csv")
```

Both these functions are base-R function, and have many additional arguments to fine-tune the resulting data.frame based on peculiarities in the data file. However, they have become less used since the `readr` package from the tidyverse introduced faster functions which better defaults. These functions are `read_csv` and `read_csv2`, i.e. with an underscore instead of a period.

### 8.3.2 Excel

While we do not like Excel very much, many people unfortunately still do. As such, it will be probable that you have to read an Excel file sooner or later. Reading Excel files can be done using the special `readxl` package. This package contains the `read_excel` function.

```
data <- read_excel("path/to/excel/file.xlsx")
```

Again, just like for csv, there are many additional arguments in `read_excel`. For example, you can set the sheet in the excel file you want to read, you can configure the types of the variables, and you can even specify a range in the excel file that you want to read, i.e. B3:G8.

### 8.3.3 JSON and XML

JSON - or JavaScript Object Notation - and XML - eXtensible Markup Language - are much more complex data notations compared with CSV. We will not discuss these formats here, but instead just mention the packages you can use if you every encounter these types.

- For JSON, the most common R-package is `jsonlite`, which contains the `fromJSON` function.
- For XML, multiple options exists, but we advise the `xml2` package. For `xml` files, there is not a single function, but you'll typically to combine many functions to get the data in the right format.

#### 8.3.4 Other statistical packages

Sometimes you will need to read data which comes from other commercial data analysis and statistical software used by less R-savvy co-workers. Often you need this because the analysis at hand cannot be done by the commercial packages and R needs to rescue you. In particular, files can come from SPSS, STATA and SAS. For each of these files, the `haven` package contains a read-function.

```
# SPSS
read_spps("file")

# Stata
read_dta("file")

#sas
read_sas("file")
```

#### 8.3.5 Databases

Finally, it is also possible to analyse data which is stored in a data.base. The way to go here will depend on the type of database. One of the useful packages in `DBI`, but you will need a specific databased backend, such as `RMySQL`, `RSQLite`, `RPostgreSQL`). Also useful is `dbplyr`, which enables many `dplyr` functions to be used directly on a data base, such that heavy computations don't have to be done by your pc.

#### 8.3.6 Background material

You can find more information on data import in Chapter 11 of the R for Data Science book, and on the help pages of mentioned packages and functions.

< This is the end of the optional reading data section >

#### 8.3.7 Converting variables

Often an integral part of reading data from files, is making sure that all the variables in our data are correctly stored. Let factors be factors, and numbers be numbers. So, let's have a look at the dataset.

```
glimpse(survey)
```

```

## Observations: 21,483
## Variables: 9
## $ year      <chr> "2000", "2000", "2000", "2000", "2000", "2000", "20...
## $ marital   <chr> "Never married", "Divorced", "Widowed", "Never married", "D...
## $ age       <chr> "26", "48", "67", "39", "25", "25", "36", "44", "44", "47",...
## $ race      <chr> "White", "White", "White", "White", "White", "White", "Whit...
## $ rincome   <chr> "$8000 to 9999", "$8000 to 9999", "Not applicable", "Not ap...
## $ partyid   <chr> "Ind,near rep", "Not str republican", "Independent", "Ind,n...
## $ relig     <chr> "Protestant", "Protestant", "Protestant", "Orthodox-christi...
## $ denom     <chr> "Southern baptist", "Baptist-dk which", "No denomination", ...
## $ tvhours   <chr> "12", NA, "2", "4", "1", NA, "3", NA, "0", "3", "2", NA, "1...

```

That does not seem very right. Due to some evil forces, all the variables are stored as characters, which isn't really correct. The `year` and `age` variables certainly should be numeric, while `marital`, for example, is clearly a nominal variable, and should this be stored as factor.

The type of variables can be changed with the following functions:

<sup>3</sup>

- `as.numeric` -> for numeric variables
- `as.integer` -> for integer variables
- `as.factor` -> for nominal variables
- `as.ordered` -> for ordinal variables
- `as.character` -> for character variables

In order to fix this, let's use an old acquaintance from `dplyr`: `mutate`. We already learned that `mutate` can be used to add new variables to a datasets, but we can just as well use it to *overwrite* existing ones.

```

survey %>%
  mutate(year = as.integer(year),
        marital = as.factor(marital),
        age = as.integer(age),
        race = as.factor(race),
        rincome = as.factor(rincome),
        partyid = as.factor(partyid),
        relig = as.factor(relig),
        denom = as.factor(denom),
        tvhours = as.numeric(tvhours)) %>%
  glimpse

## Observations: 21,483
## Variables: 9
## $ year      <int> 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, ...

```

<sup>3</sup> Note that these conversions are not by default without danger. For example, a variable can only made numeric if all its values can be treated as numeric values. If it finds values which cannot be correctly converted, such as text, it will insert a missing value instead (NA, for Not Available, as we will see below). Insertions of NA's will always lead to a warning. Such a warning will generally alert you that you did something wrong (did you convert a wrong variable?) or that there are errors in the data.

```
## $ marital <fct> Never married, Divorced, Widowed, Never married, Divorced, ...
## $ age      <int> 26, 48, 67, 39, 25, 25, 36, 44, 44, 47, 53, 52, 52, 51, 52, ...
## $ race     <fct> White, White, White, White, White, White, White, White, Whi...
## $ rincome <fct> $8000 to 9999, $8000 to 9999, Not applicable, Not applicabl...
## $ partyid <fct> "Ind,near rep", "Not str republican", "Independent", "Ind,n...
## $ relig    <fct> Protestant, Protestant, Protestant, Orthodox-christian, Non...
## $ denom   <fct> Southern baptist, Baptist-dk which, No denomination, Not ap...
## $ tvhours <dbl> 12, NA, 2, 4, 1, NA, 3, NA, 0, 3, 2, NA, 1, NA, 1, 7, NA, 3...
```

That already looks better! However, observe that we did not yet store the result of our efforts. In fact, we want to use this opportunity to give all variables an easy name and understandable name. For this, we can use the `rename` function. `rename` is a dplyr function with a very clear task: renaming variables. You can use it by giving it a list of new names connected to old names: `new_name = old_name`.

```
survey %>%
  mutate(year = as.integer(year),
        marital = as.factor(marital),
        age = as.integer(age),
        race = as.factor(race),
        rincome = as.factor(rincome),
        partyid = as.factor(partyid),
        relig = as.factor(relig),
        denom = as.factor(denom),
        tvhours = as.numeric(tvhours)) %>%
  rename(reported_income = rincome,
         party = partyid,
         religion = relig,
         denomination = denom,
         tv_hours = tvhours) %>%
  glimpse

## # Observations: 21,483
## # Variables: 9
## $ year              <int> 2000, 2000, 2000, 2000, 2000, 2000, 2000, 200...
## $ marital           <fct> Never married, Divorced, Widowed, Never married, Di...
## $ age               <int> 26, 48, 67, 39, 25, 25, 36, 44, 44, 47, 53, 52, 52, ...
## $ race              <fct> White, White, White, White, White, White, White, White, Wh...
## $ reported_income <fct> $8000 to 9999, $8000 to 9999, Not applicable, Not a...
## $ party             <fct> "Ind,near rep", "Not str republican", "Independent"...
## $ religion          <fct> Protestant, Protestant, Protestant, Orthodox-christ...
## $ denomination      <fct> Southern baptist, Baptist-dk which, No denomination...
## $ tv_hours          <dbl> 12, NA, 2, 4, 1, NA, 3, NA, 0, 3, 2, NA, 1, NA, 1, ...
```

Certainly, there is no right answer in naming variables. Just make

sure their names are understandable, easy to use and somewhat uniformly typesetted.

Furthermore, note that what we just did is not the only possible way. For instance, we could also directly create the new variable names with mutate, although we will have to remove the old names afterwards.

```
survey %>%
  mutate(year = as.integer(year),
        marital = as.factor(marital),
        age = as.integer(age),
        race = as.factor(race),
        reported_income = as.factor(rincome),
        party = as.factor(partyid),
        religion = as.factor(relig),
        denomination = as.factor(denom),
        tv_hours = as.numeric(tvhours)) %>%
  select(-rincome:-tvhours) %>%
  glimpse

## Observations: 21,483
## Variables: 9
## $ year           <int> 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 200...
## $ marital        <fct> Never married, Divorced, Widowed, Never married, Di...
## $ age            <int> 26, 48, 67, 39, 25, 25, 36, 44, 44, 47, 53, 52, 52, ...
## $ race           <fct> White, White, White, White, White, White, White, Wh...
## $ reported_income <fct> $8000 to 9999, $8000 to 9999, Not applicable, Not a...
## $ party          <fct> "Ind,near rep", "Not str republican", "Independent"...
## $ religion        <fct> Protestant, Protestant, Protestant, Orthodox-christ...
## $ denomination    <fct> Southern baptist, Baptist-dk which, No denomination...
## $ tv_hours        <dbl> 12, NA, 2, 4, 1, NA, 3, NA, 0, 3, 2, NA, 1, NA, 1, ...
```

The result is the same, but the code is slightly shorter. If you really want to master this, you might be interested to know that there are plenty variants on mutate which might make your life even more easier (or confused).

- transmute: this will **only** keep the *new* variables your list in it
- mutate\_if: this will work in the same way as select\_if, e.g. applying a function on a certain type of columns
- mutate\_at: this will apply a function on a certain set of columns you specify.

Don't worry. You'll come a long way if you can use select, mutate and rename. But don't be afraid to challenge yourself and check out the more advanced stuff.

Now, let's continue. Before we do, we copy the last part of code, this time storing the result again as `survey`, thereby overwriting the old version. You can do this in two ways: either put `survey <-` before the piece of code, or put `-> survey` after the piece of code. Again, there is no wrong or right way. Personally, I prefer the later option, as it nicely fits our narrative we created with the `%>%` symbol: we take a dataset, we perform some steps, and then we store it.

```
survey %>%
  mutate(year = as.integer(year),
        marital = as.factor(marital),
        age = as.integer(age),
        race = as.factor(race),
        reported_income = as.factor(rincome),
        party = as.factor(partyid),
        religion = as.factor(relig),
        denomination = as.factor(denom),
        tv_hours = as.numeric(tvhours)) %>%
  select(-rincome:-tvhours) -> survey
```

This is a good place to pay attention to work-flow aspects. Before, during the analysis of data, different pieces of code rarely depended on each other. For example, if we made graph A and then table B, both could be made independent from each other. We never stored the results we created to be used later (apart from a sample of data we sometimes took.) However, now that we will be cleaning and transforming the data, we will always update the `data.frame`, typically under the same name. Indeed, we don't want to end up with a list of `survey`, `survey2`, `survey3`, `survey4`, without remembering their differences. So, at each step, we update the previous version of `survey`.

However, there is a risk. If we make a mistake, our data could be broken. For example, if we erroneously converted `race` to numeric, the `as.numeric` function will fail to do so and create a column full of NAs instead. We can then quickly correct our mistake in the code, but this won't bring the original `race` variable back – it was gone the moment we mistakenly converted it.

To right our wrongs, we will need to reload the data, and all the transformations we already applied before. Just correcting the code is no longer going to be sufficient, **we need to correct our data**. When working in R Markdown, this is easiest done with the central button in an R-chunk, as this will rerun all the previous R-chunks, bringing our data in the state it was in before.

These dependencies in our workflow also mean that exercises will more depend on each other, and we must always be sure to save our updated `data.frame`. Not updating the data (or running the code) will

be a frequent source of errors later on. Be aware. (You are warned)

## 8.4 Cleaning Data

Now that we have imported the data and made sure that all the variables at least have to appropriate type, it's time to start cleaning the data. In particular, we will cover the following topics

- Duplicates observations
- Cleaning of categorical variables
- Cleaning of continuous variables
- Checking Data inconsistencies

Furthermore, we will also spend some time discussing missing values. That's a lot of concepts to cover, so let's get started!

### 8.4.1 Duplicates Removal

Sometimes, it might happen that some rows were accidentally included multiple times in the dataset. There is an easy way to find these, and to remove them.

The `duplicated` function (a base R function), returns a logical vector indicating duplicate rows in a dataset. The vector will have the same length as the number of rows in the dataset, and will be `TRUE` for rows which are not unique, and `FALSE` otherwise.

```
survey %>%
  duplicated %>%
  summary

##      Mode   FALSE     TRUE
##  logical  21220     263
```

It seems that there are 263 in our data which are not unique. We can take a look at these by using the output of `duplicated` as an input of `filter.[^point]`

```
survey %>%
  filter(duplicated(.))

## # A tibble: 263 x 9
##       year marital    age race reported_income party religion denomination
##       <int> <fct>    <int> <fct> <fct>        <fct> <fct>    <fct>
## 1  2000 Married     48 White $25000 or more Ind.^ Catholic Not applica^
## 2  2000 Never      39 Other $25000 or more  Not ^ Catholic Not applica^
## 3  2000 Married     36 Other $25000 or more  Not ^ Catholic Not applica^
## 4  2000 Never      30 White $25000 or more  Not ^ Catholic Not applica^
```

```

## 5 2000 Never ~ 19 White $1000 to 2999 Not ~ Catholic Not applica~
## 6 2000 Married 47 White $25000 or more Not ~ Catholic Not applica~
## 7 2000 Married 29 White $25000 or more Inde~ Catholic Not applica~
## 8 2000 Married 39 White $10000 - 14999 Not ~ Catholic Not applica~
## 9 2000 Widowed 80 White Not applicable Stro~ Protest~ Southern ba~
## 10 2002 Married 43 White Not applicable Inde~ Catholic Not applica~
## # ... with 253 more rows, and 1 more variable: tv_hours <dbl>

```

Notice the `.` within the `duplicated` function? The point has a special significance if used together with the piping symbol. Internally, it will be replaced with the input coming through the piping symbol. As such, `survey %>% filter(duplicated(.))` is equal to `filter(survey, duplicated(survey))`. It's very convenient if you need to refer to the piping input multiple times, not only as the first argument of the function.

Right now we selected the duplicate rows and we can have a look at them. If we only want to retain the unique rows, we can add a `!`-symbol to negate the selection.

```

survey %>%
  filter(!duplicated(.))

## # A tibble: 21,220 x 9
##   year marital age race reported_income party religion denomination
##   <int> <fct>  <int> <fct> <fct> <fct> <fct> <fct>
## 1 2000 Never ~  26 White $8000 to 9999 Ind,~ Protest~ Southern ba~
## 2 2000 Divorc~ 48 White $8000 to 9999 Not ~ Protest~ Baptist-dk ~
## 3 2000 Widowed 67 White Not applicable Inde~ Protest~ No denomina~
## 4 2000 Never ~  39 White Not applicable Ind,~ Orthodo~ Not applica~
## 5 2000 Divorc~ 25 White Not applicable Not ~ None     Not applica~
## 6 2000 Married  25 White $20000 - 24999 Stro~ Protest~ Southern ba~
## 7 2000 Never ~  36 White $25000 or more Not ~ Christi~ Not applica~
## 8 2000 Divorc~ 44 White $7000 to 7999 Ind,~ Protest~ Lutheran-mo~
## 9 2000 Married  44 White $25000 or more Not ~ Protest~ Other
## 10 2000 Married 47 White $25000 or more Stro~ Protest~ Southern ba~
## # ... with 21,210 more rows, and 1 more variable: tv_hours <dbl>

```

However, this is a little bit verbose. Therefore, `dplyr` contains a very handy short cut: the `distinct` function.

```

survey %>%
  distinct

## # A tibble: 21,220 x 9
##   year marital age race reported_income party religion denomination
##   <int> <fct>  <int> <fct> <fct> <fct> <fct> <fct>

```

```

## 1 2000 Never ~ 26 White $8000 to 9999 Ind,~ Protest~ Southern ba~
## 2 2000 Divorc~ 48 White $8000 to 9999 Not ~ Protest~ Baptist-dk ~
## 3 2000 Widowed 67 White Not applicable Inde~ Protest~ No denomina~
## 4 2000 Never ~ 39 White Not applicable Ind,~ Orthodo~ Not applica~
## 5 2000 Divorc~ 25 White Not applicable Not ~ None Not applica~
## 6 2000 Married 25 White $20000 - 24999 Stro~ Protest~ Southern ba~
## 7 2000 Never ~ 36 White $25000 or more Not ~ Christi~ Not applica~
## 8 2000 Divorc~ 44 White $7000 to 7999 Ind,~ Protest~ Lutheran-mo~
## 9 2000 Married 44 White $25000 or more Not ~ Protest~ Other
## 10 2000 Married 47 White $25000 or more Stro~ Protest~ Southern ba~
## # ... with 21,210 more rows, and 1 more variable: tv_hours <dbl>

```

Whether we want to remove duplicate rows or not really depends on the data. In our cases, it is not at all surprising that some of these rows are the same. It just happens that some people are very much alike: the same age, income, religion etc.

However, in other cases, such duplicate rows would be impossible. For instance, if there are variables which would per definition make each row unique, such as a national id number. In such cases, duplicate rows clearly need further consideration and removing them might be the right solution. But for now, let's continue.

#### 8.4.2 Cleaning Categorical Variables

For the cleaning of categorical variables, we consider the following changes

- Recoding values
- Reordering values

#### 8.4.3 Recode Categorical Variables

Sometimes, categorical variables, i.e. factors, have weird or even wrong labels. In that case, we would like to *recode* these values. Finding wrong labels isn't always easy, and often these mistakes will surface later during the analysis, in which case you have to take a step back and correct them afterwards. Nonetheless, looking at frequency tables in alphabetical order, or ordered from least to most frequent, can point to some mistakes.<sup>4</sup> Let's take the party variable as an example.

```

survey %>%
  count(party)

## # A tibble: 10 x 2
##   party           n
##   <fct>        <int>

```

<sup>4</sup> The `count` function used here is a `dplyr` short hand for `group_by(party) %>% summarize(n = n())`. Feel free to use it to save you from a lot of typing. It also has a `sort` argument for sorting on descending frequencies. As such, `count(party, sort = T)` is equal to `group_by(party) %>% summarize(n = n()) %>% arrange(-n)`. However, be aware the `count` will remove the entire grouping of a `data.frame` afterward, unlike `summarize`.

```
## 1 Don't know           1
## 2 Ind,near dem       2499
## 3 Ind,near rep       1791
## 4 Independent         4119
## 5 No answer           154
## 6 Not str democrat   3690
## 7 Not str republican 3032
## 8 Other party         393
## 9 Strong democrat    3490
## 10 Strong republican  2314
```

While the values for party are not really wrong, they are not every uniform: Str and Strong, Ind and Independent. Let's change them. We can recode factor levels using the `fct_recode` function from `forcats`. As arguments, we need to say which variable to recode, and which levels to changes.

```
data %>%
  mutate(<factor_name> = fct_recode(<factor_name>,
                                      "<new_level1>" = "<old_level1>",
                                      "<new_level2>" = "<old_level2>")
```

We can recode as many old levels into new levels as we want. Furthermore, you can replace several old levels by the same new level. Any level not mentioned will be left unchanged. Let's create some uniformity in the political affiliations.

```
survey %>%
  mutate(party = fct_recode(party,
                            "Republican, strong" = "Strong republican",
                            "Republican, weak" = "Not str republican",
                            "Independent, near rep" = "Ind,near rep",
                            "Independent, near dem" = "Ind,near dem",
                            "Democrat, weak" = "Not str democrat",
                            "Democrat, strong" = "Strong democrat"
  )) -> survey
```

Don't forget to update the dataset!

#### 8.4.4 Reorder Categorical Variables

Another possibility, especially for ordinal factors, is that the values are not really wrong, but they are in the wrong order. For instance, take a look at the reported income.

```
survey %>%
  count(reported_income)
```

```
## # A tibble: 16 x 2
##   reported_income     n
##   <fct>             <int>
## 1 $1000 to 2999     395
## 2 $10000 - 14999    1168
## 3 $15000 - 19999    1048
## 4 $20000 - 24999    1283
## 5 $25000 or more    7363
## 6 $3000 to 3999     276
## 7 $4000 to 4999     226
## 8 $5000 to 5999     227
## 9 $6000 to 6999     215
## 10 $7000 to 7999    188
## 11 $8000 to 9999    340
## 12 Don't know       267
## 13 Lt $1000          286
## 14 No answer         183
## 15 Not applicable   7043
## 16 Refused           975
```

The value “Lt \$1000” - meaning Limited, or less than \$1000 - should be shown first, but instead it is in the wrong place. Here, we need anotherforcats function, namely `fct_relevel`. This function can be used in two different ways to put a level into a different place.

**Option 1:** Move one (or more) level(s) to the front

```
data %>%
  mutate(factor_name = fct_relevel(factor_name,
                                    "level1_to_move", "level2_to move", "..."))
```

**Option 2:** Insert one (or more) level(s) after a number N of levels

```
data %>%
  mutate(factor_name = fct_relevel(factor_name,
                                    "level1_to_move", "level2_to move", "...", after = N))
```

So, let's move the LT \$1000 level to the first place.

```
survey %>%
  mutate(reported_income = fct_relevel(reported_income,
                                         "Lt $1000")) -> survey
```

We can check the results by using count on the updated `survey` data.frame.

```
survey %>%
  count(reported_income)
```

```
## # A tibble: 16 x 2
##   reported_income     n
##   <fct>             <int>
## 1 Lt $1000            286
## 2 $1000 to 2999        395
## 3 $10000 - 14999      1168
## 4 $15000 - 19999      1048
## 5 $20000 - 24999      1283
## 6 $25000 or more      7363
## 7 $3000 to 3999       276
## 8 $4000 to 4999       226
## 9 $5000 to 5999       227
## 10 $6000 to 6999      215
## 11 $7000 to 7999      188
## 12 $8000 to 9999      340
## 13 Don't know          267
## 14 No answer           183
## 15 Not applicable      7043
## 16 Refused              975
```

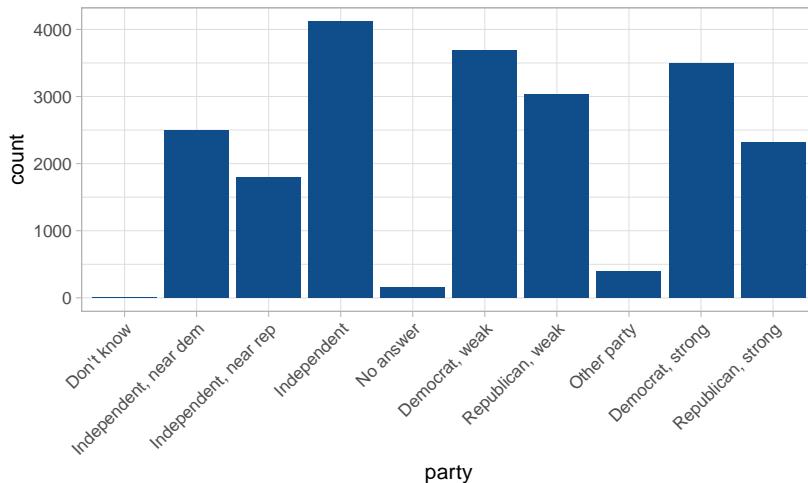
Changing the order of levels of a categorical variable is useful for both nominal and ordinal data. For ordinal data, it is logical that we want to order of the levels to be the correct one. But also for nominal data we might want to change the order. For example, there are often catch-all values such as “Other” or “Various”. It is good practice to treat these values differently compared to the regular values in a nominal, by putting them last. As such, they will shown up on one end of a graph or table, and not in between the other values. Let’s take a look at party.

```
survey %>%
  count(party)

## # A tibble: 10 x 2
##   party                 n
##   <fct>               <int>
## 1 Don't know            1
## 2 Independent, near dem 2499
## 3 Independent, near rep 1791
## 4 Independent            4119
## 5 No answer              154
## 6 Democrat, weak        3690
## 7 Republican, weak      3032
## 8 Other party             393
## 9 Democrat, strong       3490
## 10 Republican, strong     2314
```

Deciding whether a factor is ordinal or not is not always that straightforward. If we look at the reported income, it is clear that there is an order. However, we didn't define `party` as an ordered factor. There is no "best" or "superior" political party, so explicitly program this variable as a ordinal factor would be one bridge too far – we would have to decide which end of the political spectrum is the "lowest" and which the "highest". However, this is somewhat undesirable if we make graphs.

```
survey %>%
  ggplot(aes(party)) +
  geom_bar(fill = "dodgerblue4") +
  theme_light() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



When we don't have an ordinal factor, the order of labels will be often be alphabetical. In this case, because we recoded the labels before, they are not even in alphabetical order any more.<sup>5</sup>. The resulting graph is difficult to read, as the x-axis is scrambled. A natural reflex would be to order the bar chart according to frequency, but that would not really improve the readability in this special case. Instead, we can apply a more logical order, without necessary considering `party` as an ordinal variable. Such a logical order is readily available for the current variable, as we often speak of left and right-wing politics. We can leave the alternative answers (Don't know, No Answer, Other Party), either at the start or end of the order. By not noticing them in the code below, the latter will happen.

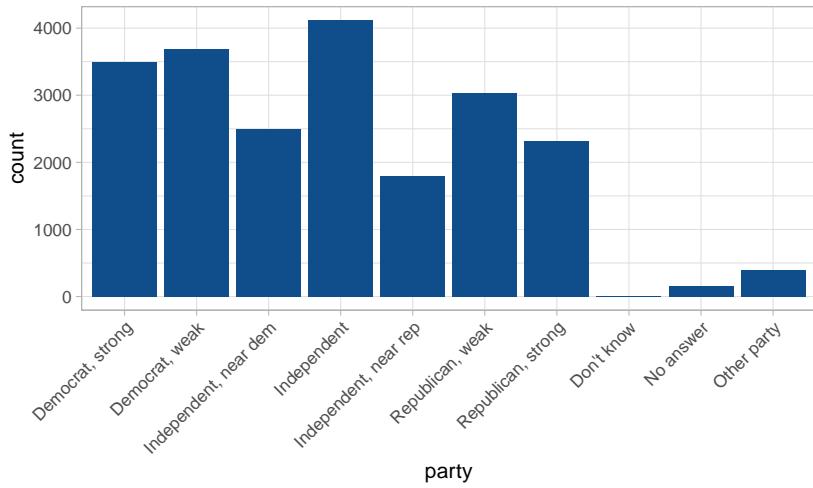
```
survey %>%
  mutate(party = fct_relevel(party,
    "Democrat, strong",
    "Democrat, weak",
```

<sup>5</sup> When we first did `as.factor(party)`, the levels were placed in alphabetical order. However, we then recoded some levels, but this didn't update the order. Thus "Strong Republican" became "Republican Strong" and "Strong Democrate" became "Democrat Strong", but both remained the last levels because they originally started with S. It's not expected that you are familiar with all the side-effects of some transformations, but this shows you the complexities when our actions start to depend on earlier actions, and how we are really working in an iterative context.

```
"Independent, near dem",
"Independent",
"Independent, near rep",
"Republican, weak",
"Republican, strong")) -> survey
```

Our graph now looks as follows. Better, isn't it? <sup>6</sup>

```
survey %>%
  ggplot(aes(party)) +
  geom_bar(fill = "dodgerblue4") +
  theme_light() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



<sup>6</sup> Of course, *better* is subjective and depends on what you like to show your audience. If the data would be about Flemish political parties, it's more difficult to apply a logical order. (I.e. who is more “left”, Groen or Sp.a?) Because there are not the same nuances in these parties, such as “weak” or “strong”, it would make more sense to order them according to frequencies. (Which is what happens in times of Flemish elections, but not necessarily in times of US elections.) In the end, none two situations are the same and much is left to your decisions and assumptions as data analyst.

In conclusion we use

- `fct_recode` for recoding values of a categorical variable, and
- `fct_relevel` for manually reordering values of a categorical variable.

Later on, we will seen more specific functions for recoding and reordering categorical variables when transforming data. Make sure you don't lose the overview! First, we look at cleaning continuous data.

#### 8.4.5 Cleaning continuous data

For continuous data, the range of possible values is infinite, and it is therefore more difficult to find *wrong* values. Without information on the context of the data, finding wrong continuous entries is extremely difficult.

#### 8.4.6 Errors

In the `survey` data.frame, there are three continuous variables: year, age and (daily) tv\_hours. For each of these variables, we have a certain idea about the possible range of values. The age will probably be somewhere between 20 and 100 (knowing that the dataset contains information on adults). The number of tv hours should be between 0 and 24, as there are 24 hours in a day. Also for year we more or less know what to expect. Let's have a look at each.<sup>7</sup>

```
summary(survey$year)

##   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
##   2000    2002    2006    2007    2010    2014
```

For year, everything seems allright. We converted it already to an integer variable before, so we don't have to check for erroneous decimal years. Furthermore, the minimum and maximum seems quite alright. When year contains errors, we most likely observe it at these extremes: 2102 instead of 2012, 1099 instead of 1999, or 9999 indicating that it is actually missing.

Let's look at age then.

```
summary(survey$age)

##   Min. 1st Qu. Median   Mean 3rd Qu.   Max.   NA's
##   18.00  33.00  46.00  47.18  59.00  89.00     76
```

On first sight, there does not seem to be problems with age. There are 76 missing values, but the values present are situated between 18 and 89 years, which again is a logical range. Also age was converted without problems to an integer variable before, so all values are integer numbers.

If we look at tv hours, we see something peculiar.

```
summary(survey$tv_hours)

##   Min. 1st Qu. Median   Mean 3rd Qu.   Max.   NA's
##   0.000  1.000  2.000  3.004  4.000  84.000 10146
```

We see there are 10146 missing values, which is high but not necessarily wrong (unless we accidentally removed some values, which we didn't). There are no negative values, as the minimum number of hours someone watched tv is zero. However, on the other side, we see that the number of tv hours goes up to 84 hours a day - this is clearly wrong. We all have just 24 hours in a day.

We can further look at the records for which the tv hours are more than 24.

<sup>7</sup> When inspecting categorical data, we can do a quick count. While this might work for some continuous variables, such as year, it is not suited for most continuous variables, as they are often unique for each observation. Instead, we look at the summary of the variables.

```
survey %>%
  filter(tv_hours > 24)

## # A tibble: 5 x 9
##   year marital age race reported_income party religion denomination tv_hours
##   <int> <fct>  <int> <fct> <fct>      <fct> <fct> <fct>      <dbl>
## 1 2000 Married    84 White Not applicable Inde~ Protest~ No denomina~     28
## 2 2000 Widowed    68 White Not applicable Inde~ Protest~ Other luther~     84
## 3 2010 Married    69 White Not applicable Demo~ Protest~ Baptist-dk ~    35
## 4 2010 Divorc~    57 White Not applicable Demo~ Catholic Not applica~    56
## 5 2014 Separa~    30 White Not applicable Inde~ None      Not applica~    28
```

There seem to be 5 observations for which the number of tv hours is clearly wrong, and we need to correct them. However, we don't have a clue how to correct them in our case. The only thing we can do, is to delete these values, and make them missing.<sup>8</sup> Observe, we do not delete the entire observations, just the tvhours variable for these observations. The other variables can still be used for these 5 rows.

We can do this by using the `ifelse` function. This function is a very generic function which returns a value dependent on a logical test.<sup>9</sup> The function can be used as follows.

```
if_else( <condition>, <value_a>, <value_b> )
```

Suppose we have a vector `score` containing student scores. We can use the `ifelse` function to create a vector `grade` with values `FAILED` and `PASSED`.

```
grade <- ifelse(score >= 10, "PASSED", "FAILED")
```

Now, let's use the function to update the `tv_hours` variable.

```
survey %>%
  mutate(tv_hours = ifelse(tv_hours > 24, NA, tv_hours)) -> survey
```

So, what happens? The `tv_hours` variable is updated using `mutate`. If it is larger than 24, the new value will be `NA`, i.e. not available. Otherwise, the new value will just be the old value of `tv_hours`. After the column is updated, the new data.frame is again stored as `survey`.

Checking for errors, both categorical and continuous, can be a *street without ending*. Usually, you do the best you can by using the tricks discussed above for each variable. If this seems like a lot of work, remember that 70% of a typical data project goes to cleaning and transforming data, and only 30% to actual analysis and interpretation.

And even then, despite all your time and efforts, it can happen that you discover data errors in the analysis phase. Not surprisingly, as you will really look at the data in detail at that point. When this

<sup>8</sup> Errors such as these can often be avoided by proper data collection. I.e., if you create a survey or a data form online, make sure that all fields are reasonably checked: age cannot be negative, a zip code consists of 4 numbers, and one does not have more than 24 hours in a day. While there are still things which can go wrong later, at least the data cannot be wrongly inputted by respondents. In real-life, many data collection happens without much thought, unfortunately.

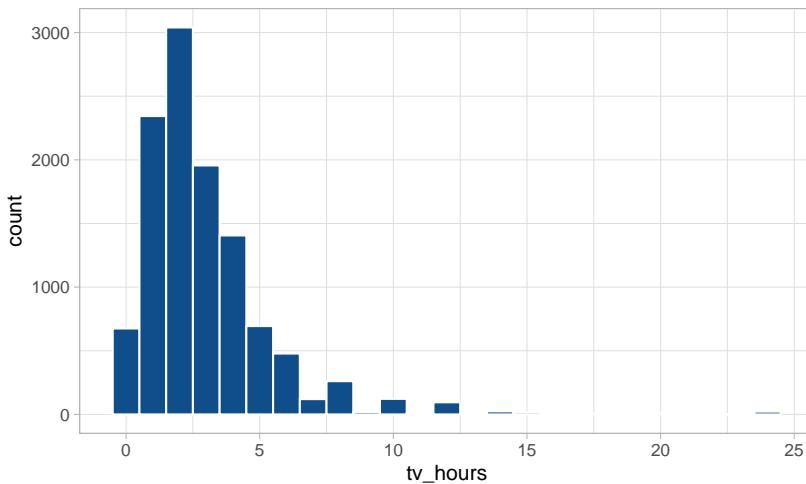
<sup>9</sup> Maybe you are familiar with the IF function in Excel? Its use is exactly the same.

happens, you go back to your cleaning and transformation scripts and modify them where needed. As such, it is important that you store the original data as well as all the changes you performed. Even more important is to make sure that all your analysis are stored as a script of RMarkdown, so that they can quickly be rerun after correcting the error. This is called *reproducible research* and will save you a lot of time and mistakes. Have a look at this video for an illustration of reproducible work-flows.

#### 8.4.7 Outliers

It also happens that some continuous values are not necessarily wrong, but *exceptionally* high or low. These exceptions we don't call error, but instead we call outliers – a data point that is distant from other observations. For example, consider tv hours. Before we removed values greater than 24 hours, because there are logically errors. Let's have a look at the remaining values.

```
survey %>%
  ggplot(aes(tv_hours)) +
  geom_histogram(binwidth = 1,
                 color = "white", fill = "dodgerblue4") +
  theme_light()
```



It seems that while most people watch television between 0 and 5 hour a day, there are some exceptionally high values. It's not clear whether they are mistakes like the ones we removed before, or just abnormal tv-addicts. Let's have a look at the 25 highest values, and compare them with some related attributes, such as age, income and marital status (comparing them with religion or party might be regarded as politically incorrect, so let's steer clear from that danger zone).<sup>10</sup>

<sup>10</sup> Remember that we use `pander` to improve the layout of our tables, and you shouldn't pay attention to that.

```
survey %>%
  arrange(-tv_hours) %>%
  slice(1:25) %>%
  select(tv_hours, marital, age, reported_income) %>%
  pander()
```

tv_hours	marital	age	reported_income
24	Never married	30	Not applicable
24	Separated	45	Not applicable
24	Never married	33	\$6000 to 6999
24	Divorced	53	Not applicable
24	Divorced	50	No answer
24	Never married	44	Not applicable
24	Never married	21	Don't know
24	Widowed	71	Not applicable
24	Widowed	62	Not applicable
24	Widowed	52	Refused
24	Never married	56	Not applicable
24	Divorced	51	Not applicable
24	Divorced	75	Not applicable
24	Separated	49	\$8000 to 9999
24	Divorced	65	Not applicable
24	Never married	27	Not applicable
24	Married	71	Not applicable
24	Never married	27	\$8000 to 9999
24	Separated	63	Not applicable
24	Divorced	31	\$5000 to 5999
24	Separated	37	Not applicable
24	Married	46	Not applicable
23	Never married	32	Not applicable
22	Divorced	69	Not applicable
22	Married	63	Not applicable

It seems that many of the respondents watch the maximum number of 24 hours tv each day, which is surprising. We can use the other

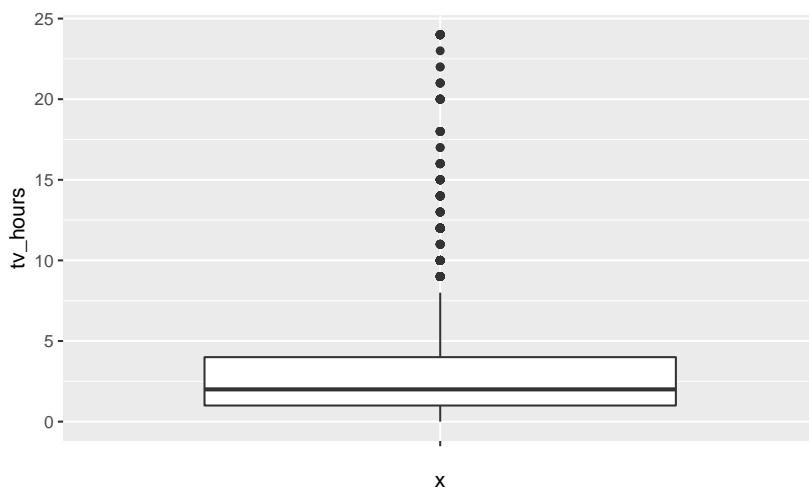
variables to get a better idea about these observations. We see that many of them are single (never married, divorced or separated). The age seems to be relatively high, but not very remarkable (average age for the dataset was about 47). Finally, many did not report their income.

This information can be interpreted in multiple ways:

- These are single, lazy people with an income who watch tv all day
- These are people who weren't very honest when filling out their information (also not reporting any income).

Deciding whether something is wrong or exceptional is not trivial and requires certain domain knowledge and assumptions. In this case, the logical assumption is that these numbers are incorrect (even lazy, tv-binging people need to sleep now and then). The more difficult question is at what point something becomes incorrect. 20 hours? 16? For continuous variables it can help to look at a boxplot and see until where the whiskers go. However, this is no exact science, especially when variables are not distributed symmetrically.

```
survey %>%
  ggplot(aes("", tv_hours)) +
  geom_boxplot()
```



For the current case, we can say the people who watch more than 8 hours tv either filled in an incorrect number, or are exceptional tv-watchers. As such, we remove all values higher than 8.

```
survey %>%
  mutate(tv_hours = ifelse(tv_hours > 8, NA, tv_hours)) -> survey
```

#### 8.4.8 Data inconsistencies

Another approach to check for errors is to look at more than one variable at the same time, and check for obvious inconsistencies. This can be thought of as “rules” which are violated. In the current dataset, we could check that all married people should be at least 18 years old. Since all observations concern people which are at least 18 years old, we know that this rule is not violated.

In other cases, example rules can be:

- the departure of a flight should take place before it can arrive.  
(Taking into account different time zones, obviously.)
- someone whose job status is “unemployed” cannot have a reported income (unless unemployment benefits are taken into account).
- etc.

Often, you will not have time to check all possible rules you can think of. Furthermore, some rules will be based on certain assumptions which you need to check. For example, the age at which one can marry depends on the country, and thus be lower or higher than 18.

#### 8.4.9 Missing values

In a real-life setting, data typically comes with missing values. The values might be missing from the offset, or they might be missing because we removed them as outliers or wrong values. In subsequent paragraphs we will see how to analyse missing values – e.g. are they missing at random or not? – and how to handle them during your analysis. We are obliged to say that there are also techniques which can be used to *guess* missing values based on the values for other attributes and other observations with similar values. This is called missing value *imputation* and as a separate field on itself. Due to the complexities involved, we will not talk about it here, but the interested reader is referred to this manual of the mice-package.

#### 8.4.10 Analyzing missing data

There are three different ways in which missing data can occur (see theory).

- Missing Completely at Random (MCAR)
- Missing at Random (MAR)
- Not Missing at Random (NMAR)

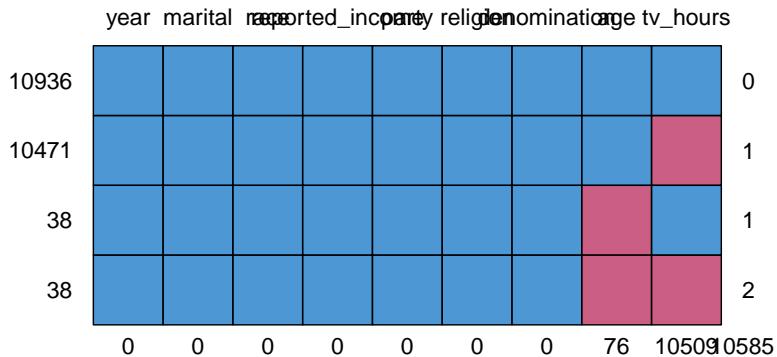
Below, we illustrate some techniques to analyse the missing values in your data. The most obvious way to check whether your data contains missing values is by looking at the summary.

```
summary(survey)
```

```
##      year          marital        age         race
##  Min.   :2000   Divorced    : 3383   Min.   :18.00  Black: 3129
##  1st Qu.:2002  Married     :10117   1st Qu.:33.00  Other: 1959
##  Median :2006  Never married: 5416   Median :46.00  White:16395
##  Mean   :2007  No answer    :    17   Mean   :47.18
##  3rd Qu.:2010  Separated    :  743   3rd Qu.:59.00
##  Max.   :2014  Widowed     :1807   Max.   :89.00
##                               NA's    :76
##      reported_income           party       religion
## $25000 or more:7363  Independent      :4119  Protestant:10846
## Not applicable:7043  Democrat, weak   :3690  Catholic   : 5124
## $20000 - 24999:1283  Democrat, strong  :3490  None       : 3523
## $10000 - 14999:1168  Republican, weak  :3032  Christian  : 689
## $15000 - 19999:1048  Independent, near dem:2499  Jewish     : 388
## Refused       : 975   Republican, strong  :2314  Other      : 224
## (Other)        :2603  (Other)            :2339  (Other)    : 689
##      denomination      tv_hours
## Not applicable :10072   Min.   :0.000
## Other          : 2534   1st Qu.:1.000
## No denomination: 1683   Median :2.000
## Southern baptist: 1536   Mean   :2.659
## Baptist-dk which: 1457   3rd Qu.:4.000
## United methodist: 1067   Max.   :8.000
## (Other)         : 3134   NA's    :10509
```

This tells us for which variables there are missing data, and how many. However, it does not tell us anything about the relationships between missing values. In order to look at *patterns* of missing data, we can use the `md.pattern` function (missing data patterns) from the package `mice`.

```
md.pattern(survey)
```



```
##          year marital race reported_income party religion denomination age
## 10936      1       1     1                  1       1       1           1   1
## 10471      1       1     1                  1       1       1           1   1
## 38         1       1     1                  1       1       1           1   0
## 38         1       1     1                  1       1       1           1   0
##          0       0     0                  0       0       0           0  76
##          tv_hours
## 10936      1       0
## 10471      0       1
## 38         1       1
## 38         0       2
##          10509 10585
```

The output of `md.pattern` is a bit cryptic, but let's have a closer look. Each column refers to one of the variables, as is indicated. Each row is a *pattern* consisting of 1s (data is not missing) and 0s (data is missing). The first row, where all variables have a 1, is a pattern where none of the variables is missing. This is also denoted by the zero in the last column. In the second rows, age is indicated with a zero, meaning that for this pattern, the variable age is missing. The last column thus indicates 1 missing value. The last pattern (the penultimate row) is one with 2 missing values as indicated in the last column. In particular, age and tv hours are missing.

The numbers in the first (unnamed) column indicate how many observations of each pattern there are. As such, the first pattern has the most observations – 10936 persons without missing data. The last pattern (tv hours and age missing) occurs 38 times. The final row equals the number of missing values for each variable (the same information summary gave us). Finally, the number in the lower-right corner is the total number of missing values.

The output of `md.patterns` can show us whether the occurrence of missing values are related. For example, if age is missing, then tv hours is also missing. The latter is not the case, as there are as many observations where age is missing and tv hours not, as there are observations where age is missing and tv hours also.

Next to `md.pattern` we can also check whether the occurrence of missing values is related to the value for other variables. As such, we can ask ourselves whether people from certain religions or political affiliations are more or less likely to report their age over the number of hours they watch tv. These patterns can be checked using `ggplot/dplyr`, by creating a new variable that indicates whether an observation has a missing value.

Let's consider age. We add a variable to indicate that age is or is not missing. Note that to check this, we need a special function. We cannot use `age == NA` to see if age is missing. In the latter condition, we are comparing age with NA, i.e. we are comparing age with something we don't have. We can never know whether age is equal to something we don't have, so the result of that is always NA, regardless of the value of age. As an example, consider the variables `a` and `b`, of which `a` is missing and `b` is not.

```
a <- NA
b <- 1

a == NA
## [1] NA

b == NA
## [1] NA
```

So, how do we check if something is NA? We use the special function `is.na` for that.

```
is.na(a)
## [1] TRUE

is.na(b)
## [1] FALSE
```

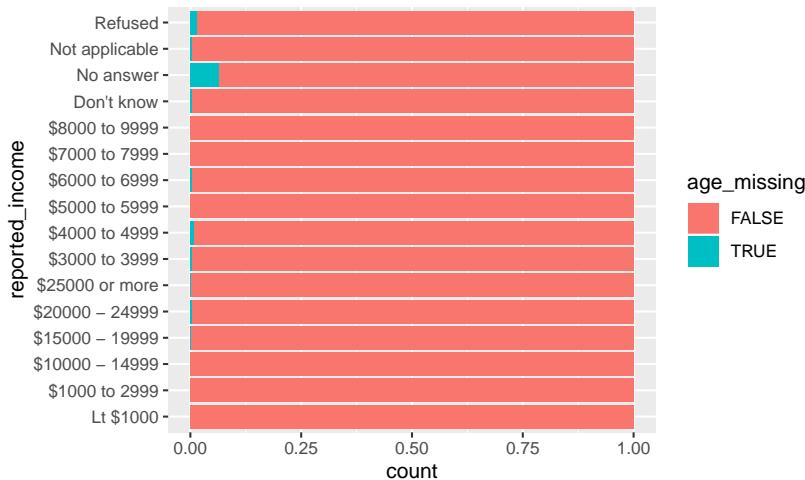
Thus, for the survey data:

```
survey %>%
  mutate(age_missing = is.na(age),
        tv_missing = is.na(tv_hours)) -> survey_md
```

Note that we store the data.frame with the additional variables under another name, since we will not need these variables in the eventual analysis stage, just for looking at the missing data for the moment.

When we compare the missing/not missing variable with a categorical variable, we have in fact 2 categorical variables. Thus, we can use a graph or table which is appropriate for comparing categorical variables. As we all (should) know by now, we can use a bar chart.

```
survey_md %>%
  ggplot(aes(reported_income, fill = age_missing)) +
  geom_bar(position = "fill") +
  coord_flip()
```

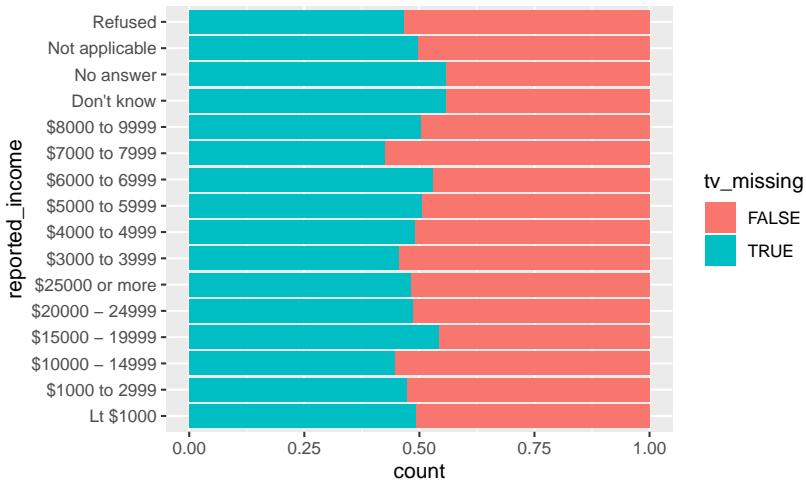


The bar chart shows that people who didn't report their income, were more likely to not report their age also.<sup>11</sup>

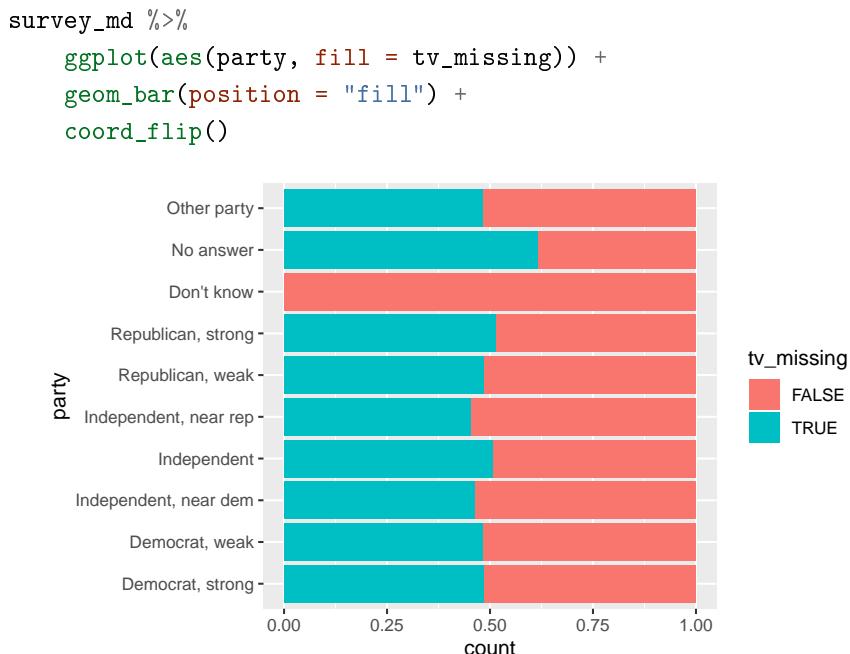
We can do the same for tv-hours. We then see that the percentage of missing values varies – being somewhat higher or lower for certain incomes – but no clear patterns exist.

<sup>11</sup> One could argue that we also coded the “No answer” value for reported income as NA. However, in this case, it was decided not to, in order to retain a clear distinction between the special categories (Refused, Not applicable, No answer and Don't know).

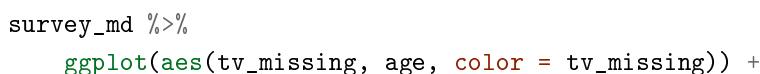
```
survey_md %>%
  ggplot(aes(reported_income, fill = tv_missing)) +
  geom_bar(position = "fill") +
  coord_flip()
```



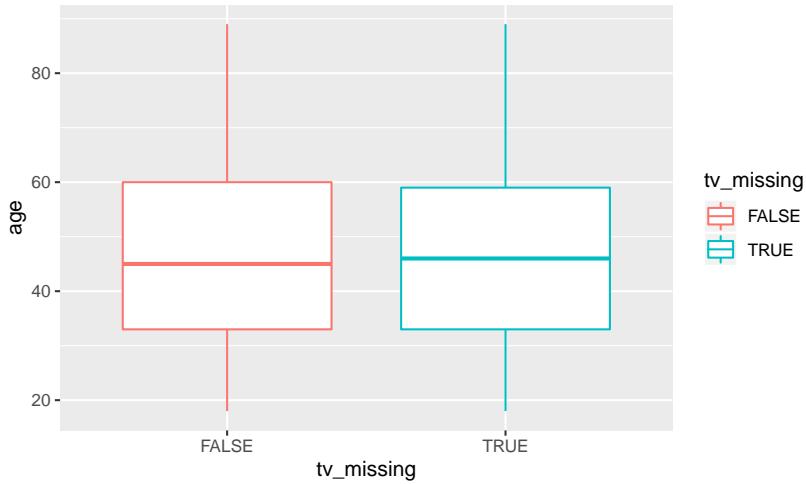
Of course, the occurrence of missing values can be compared with more than one variable, not just income. Below we show the comparison of tv-hours missing with political affiliation. Here we can clearly see that there is one answer for political party which was only used for observations which did not include the tv hours.



As a final example, we can compare the missing of tv hours with a continuous variable, let's say age. Are people who didn't report the number of hours they watch tv generally older or younger? The boxplots below don't show any difference.



```
geom_boxplot()
```



How many comparisons you make for each variable where data is missing is up to you – but you have to gather reasonable evidence whether your missing values are MAR, MCAR or NMAR.

#### 8.4.11 Working around missing data

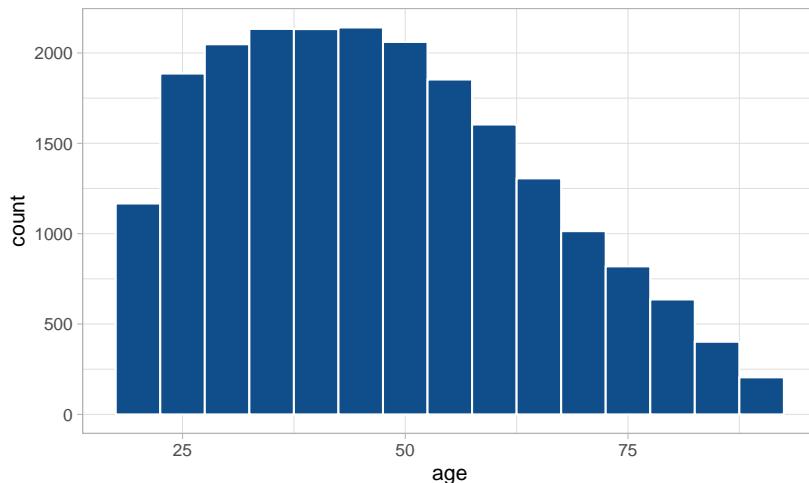
While we will not see how to impute – *guess* – missing values, it is quite important to know how to work with them.

#### 8.4.12 Visualizations

First, consider visualizations. When working with ggplot, missing values will often be ignored automatically. For instance, if we try to create a histogram of the age, ggplot will tell us that it ignored some missing values through a warning.

```
survey %>%
  ggplot(aes(age)) +
  geom_histogram(binwidth = 5, fill = "dodgerblue4", color = "white") +
  theme_light()

## Warning: Removed 76 rows containing non-finite values (stat_bin).
```



The warning

```
## Warning: Removed 76 rows containing non-finite values (stat_bin).
```

tells us when and how many missing values are ignored.

In case that a categorical variable has missing values, NA will appear as a separate category. For example, consider the dataset survey2 (which we will just use here as example), which has missing values for several variables.

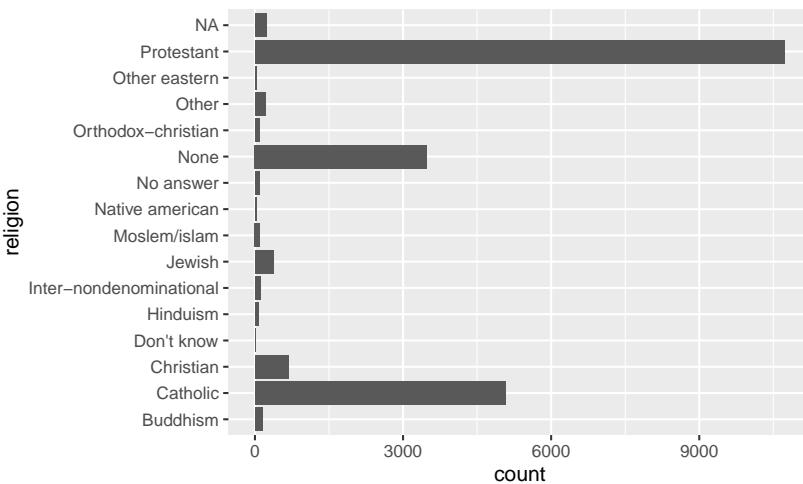
```
summary(survey2)
```

```
##      age           denomination        marital
##  Min.   :18.00   Not applicable :9960   Divorced    :3351
##  1st Qu.:33.00   Other          :2508   Married     :9999
##  Median :46.00   No denomination:1665   Never married:5359
##  Mean   :47.18   Southern baptist:1521  No answer    :  17
##  3rd Qu.:59.00   Baptist-dk which:1439  Separated    : 738
##  Max.   :89.00   (Other)         :4159   Widowed    :1788
##  NA's   :280    NA's            : 231   NA's       : 231
##                  party          race        religion
##  Independent     :4080   Black: 3098   Protestant:10733
##  Democrat, weak :3658   Other: 1941   Catholic  : 5066
##  Democrat, strong:3453   White:16226  None     : 3485
##  Republican, weak:2997   NA's :  218   Christian :  685
##  Independent, near dem:2479                      Jewish    :  384
##  (Other)          :4604                           (Other)   :  902
##  NA's            : 212                           NA's     : 228
##      reported_income   tv_hours        year
##  $25000 or more:7294   Min.   :0.000   Min.   :2000
##  Not applicable:6973   1st Qu.:1.000   1st Qu.:2002
##  $20000 - 24999:1274   Median :2.000   Median :2006
```

```
##   $10000 - 14999:1159   Mean     :2.659   Mean     :2006
##   $15000 - 19999:1034   3rd Qu.:4.000   3rd Qu.:2010
##   (Other)      :3544    Max.    :8.000   Max.    :2014
##   NA's        : 205    NA's    :10596   NA's    :206
```

Let's say we make a bar chart of the religion variable. The NA value then appears as a separate category and gets its own bar. Automatically, this category will be plotted at the side of the others, not between them.

```
survey2 %>%
  ggplot(aes(religion)) +
  geom_bar() +
  coord_flip()
```



As such, ggplot handles missing values without problem and quite transparent. The same is not true when we go to numeric computations.

#### 8.4.13 Numerical analysis

Let's start with the good news. If we create a frequency table of a categorical value with missing values, the NA's will be considered just as any other value. For example, we can make a frequency table to support the graph above for survey2.

```
survey2 %>%
  group_by(religion) %>%
  summarize(frequency = n()) %>%
  mutate(relative_frequency = frequency/sum(frequency)) %>%
  arrange(-frequency) %>%
  pander
```

religion	frequency	relative_frequency
Protestant	10733	0.4996
Catholic	5066	0.2358
None	3485	0.1622
Christian	685	0.03189
Jewish	384	0.01787
NA	228	0.01061
Other	220	0.01024
Buddhism	147	0.006843
Inter-	107	0.004981
nondenominational		
Moslem/islam	103	0.004794
Orthodox-christian	94	0.004376
No answer	92	0.004282
Hinduism	70	0.003258
Other eastern	32	0.00149
Native american	22	0.001024
Don't know	15	0.0006982

Unfortunately, the same is not true when we start to compute metrics for centrality or spread. By design, when you apply a function on a vector containing missing values, the function will return a missing value. Consider the vector below, of which we want to know the mean and sum

```
x <- c(5, 6, 12, NA, 43)
mean(x)
```

```
## [1] NA
```

```
sum(x)
```

```
## [1] NA
```

Just as the warning we got when using ggplot with missing values, this result is warning. It warns us that we want to compute something about a vector which partly misses. However, the warning is quite strong here; it doesn't give us anything we can use.

In order to circumvent this from happening, each of these functions has a argument which is called `na.rm` – meaning “NA remove”. If we say `na.rm = T`, missing values will be removed and ignored, and the function will compute the result with the remaining value. Thus,

```
mean(x, na.rm = T)
```

```
## [1] 16.5
```

```
sum(x, na.rm = T)
## [1] 66
```

The fact that we have to explicitly state that we want to ignore missing values is R's safety mechanism which prevents us from ignoring missing values by accident.

Thus, if we want to compute measures of centrality and spread for, let's say, age, we will need to use this argument if we want to come up with anything.<sup>12</sup>

```
survey %>%
  summarize(min = min(age, na.rm = T),
            mean = mean(age, na.rm = T),
            max = max(age, na.rm = T),
            iqr = IQR(age, na.rm = T))

## # A tibble: 1 x 4
##       min     mean    max     iqr
##   <int> <dbl> <int> <dbl>
## 1     18    47.2    89    26
```

Finally, what happens when we want to compute correlations? For instance, how is age and tv hours correlated. Both of these have missing values. We could try the following:

```
survey %>%
  select(age, tv_hours) %>%
  cor(na.rm = T)

## Error in cor(., na.rm = T): unused argument (na.rm = T)
```

Oops, that was wishful thinking from my part. It seems that the `na.rm` argument doesn't exist for the `cor` function. So far for consistency in base R functions.

In order to compute correlations, we need to ensure that we only consider rows without missing values. We can do this with the `na.omit` function. This function removes – omits – all rows which have one or more missing values.

```
survey %>%
  select(age, tv_hours) %>%
  na.omit() %>%
  summary()

##      age          tv_hours
##  Min. :18.00    Min. :0.000
##  ...
```

<sup>12</sup> Yes, we need to repeat this for each function we want to use, and no, there is no easier way.

```

##   1st Qu.:33.00   1st Qu.:1.000
##   Median :45.00   Median :2.000
##   Mean    :47.13   Mean    :2.659
##   3rd Qu.:60.00   3rd Qu.:4.000
##   Max.    :89.00   Max.    :8.000

```

Then, we can compute the correlation.

```

survey %>%
  select(age, tv_hours) %>%
  na.omit() %>%
  cor()

##           age   tv_hours
## age     1.0000000 0.1895392
## tv_hours 0.1895392 1.0000000

```

The `na.omit` function can be somewhat dangerous and should only be used in exceptional cases as these. It can make a huge difference if we do it before select. Consider again the `survey2` dataset, where all variables have some missing values.

This dataset has the following number of rows.

```

survey2 %>% nrow()

## [1] 21483

```

If we select age and tv hours, and remove rows with missing values, we are left with the following number of rows

```

survey2 %>%
  select(age, tv_hours) %>%
  na.omit() %>%
  nrow()

## [1] 10732

```

But, if we first remove rows with missing values, and then select the two variables, we are only left with the following number of rows.

```

survey2 %>%
  na.omit() %>%
  select(age, tv_hours) %>%
  nrow()

## [1] 9974

```

Do you see what's different? Have a good look. If we perform the removal before the select, it will take into account missing values for ALL variables. If we do the select first, it will only check for the variables we retain. This nuance can make a huge difference in practice. Using the `na.omit` function should always raise an alert in your mind that directs you to be cautious.

In conclusion, it is advised to be cautious all the way when working with missing value. And with those wise words, we end our cleaning efforts, and proceed to transformations.

## 8.5 Transforming Data

Whereas cleaning data is focussed on finding errors, transforming data is about making it more easier to analyse. There are many ways to do so.

Firstly, we can remove the number of levels in a categorical variable, preventing the plotting of too many different categories. Secondly, we can also create new variables based on existing ones. The latter is also called enriching data. Moreover, we can adjust variables to make them easier to interpret. For example, using distances in km instead of miles (or the other way around when you are British or American). Furthermore, we might want to discretize continuous variables – turning them into categorical ones – thereby enabling us to use other analyses or visualizations. Finally, to conclude out tutorial, we will also discuss some transformations which are useful when we are visualizing data, most notably to order variables.

Many many things to do, so time to start.

### 8.5.1 Discretization of continuous variables

Turning a continuous variable into a categorical one is called discretization. Several functions to do this are provided by `ggplot`.

- `cut_width`: creating intervals of a specific width
- `cut_interval`: creating a specific number of intervals of equal width
- `cut_number`: creating a specific number of interval with an equal number of observations.

For example, we can *cut* tv hours into interval of width 4.<sup>13</sup>

```
survey %>%
  mutate(tv_hours = cut_width(tv_hours, width = 4, boundary = 0)) %>%
  count(tv_hours)

## # A tibble: 3 x 2
```

<sup>13</sup> The `boundary` argument is an optional argument to define where the interval should start.

```

##   tv_hours     n
##   <fct>     <int>
## 1 [0,4]      9421
## 2 (4,8]     1553
## 3 <NA>      10509

```

Alternatively, we can *cut* tv hours into 4 intervals of equal width.

```

survey %>%
  mutate(tv_hours = cut_interval(tv_hours, n = 4)) %>%
  count(tv_hours)

## # A tibble: 5 x 2
##   tv_hours     n
##   <fct>     <int>
## 1 [0,2]      6059
## 2 (2,4]     3362
## 3 (4,6]     1172
## 4 (6,8]      381
## 5 <NA>      10509

```

Or, we can *cut* tv hours into three intervals which contain an equal number of observations.

```

survey %>%
  mutate(tv_hours = cut_number(tv_hours, n = 3)) %>%
  count(tv_hours)

## # A tibble: 4 x 2
##   tv_hours     n
##   <fct>     <int>
## 1 [0,2]      6059
## 2 (2,3]     1956
## 3 (3,8]     2959
## 4 <NA>      10509

```

Here, it should be noted that the intervals do not contain the equal amount of observations at all. This is because there are many observations with a unique value which cannot be split further. But nevertheless, *cut\_number* will try its best, which will turn out better if the values of the variable are more unique. For example, age lends itself better for this:

```

survey %>%
  mutate(age = cut_number(age, n = 5)) %>%
  count(age)

```

```
## # A tibble: 6 x 2
##   age      n
##   <fct>    <int>
## 1 [18,31]  4656
## 2 (31,41]  4305
## 3 (41,51]  4207
## 4 (51,63]  4149
## 5 (63,89]  4090
## 6 <NA>     76
```

Each of the discretization functions allows us to modify the names of the categories – instead of the default interval notation by providing a vector of names to the `labels` argument. For example

```
survey %>%
  mutate(age_category = cut_number(age, n = 3, labels = c("Young","Middle-aged","Old"))) %>%
  group_by(age_category) %>%
  summarize(min = min(age), max = max(age), frequency = n())

## # A tibble: 4 x 4
##   age_category  min   max frequency
##   <fct>        <dbl> <dbl>     <int>
## 1 Young         18    37      7234
## 2 Middle-aged   38    54      7117
## 3 Old           55    89      7056
## 4 <NA>          NA    NA       76
```

In the last example we stored the discretized variable under a different name. This is most advised, in order to not lose the original, detailed data.

### 8.5.2 Rescaling continuous variables

When we have continuous variables, we can also adjust there scale. For example, we can compute the percentage of tv hours per day by dividing tv hours by 24.

```
survey %>%
  mutate(tv_per_day = tv_hours/24) %>%
  summary

##      year            marital          age          race
##  Min.   :2000   Divorced      : 3383   Min.   :18.00   Black: 3129
##  1st Qu.:2002  Married      :10117   1st Qu.:33.00   Other: 1959
##  Median :2006  Never married: 5416   Median :46.00   White:16395
##  Mean   :2007  No answer    :    17   Mean   :47.18
##  3rd Qu.:2010 Separated    :  743   3rd Qu.:59.00
```

```

##   Max.    :2014   Widowed      : 1807   Max.    :89.00
##                               NA's     :76
##   reported_income           party          religion
## $25000 or more:7363   Independent       :4119   Protestant:10846
## Not applicable:7043   Democrat, weak    :3690   Catholic   : 5124
## $20000 - 24999:1283   Democrat, strong   :3490   None       : 3523
## $10000 - 14999:1168   Republican, weak  :3032   Christian  : 689
## $15000 - 19999:1048   Independent, near dem:2499   Jewish     : 388
## Refused      : 975   Republican, strong  :2314   Other      : 224
## (Other)       :2603   (Other)          :2339   (Other)    : 689
##   denomination        tv_hours        tv_per_day
## Not applicable :10072   Min.    :0.000   Min.    :0.000
## Other          : 2534   1st Qu.:1.000   1st Qu.:0.042
## No denomination: 1683   Median   :2.000   Median   :0.083
## Southern baptist: 1536   Mean     :2.659   Mean     :0.111
## Baptist-dk which: 1457   3rd Qu.:4.000   3rd Qu.:0.167
## United methodist: 1067   Max.    :8.000   Max.    :0.333
## (Other)         : 3134   NA's    :10509   NA's    :10509

```

Other common transformations are:

- different currencies (euro vs dollar, etc.)
- different measurement scales (miles vs km, inch vs cm, etc.)
- different time zones or time units (on which more later).

### 8.5.3 Adding calculated variables

The addition of calculated variables is similar to rescaling variables. The only difference is that rescaling only relates to one variable, while calculated variables can concern different variables.

For example, we can compute the year of birth for people in our data.

```

survey %>%
  mutate(year_of_birth = year - age) %>%
  select(year, age, year_of_birth) %>%
  summary

##   year        age      year_of_birth
##   Min.    :2000   Min.    :18.00   Min.    :1911
##   1st Qu.:2002   1st Qu.:33.00   1st Qu.:1947
##   Median  :2006   Median  :46.00   Median  :1960
##   Mean    :2007   Mean    :47.18   Mean    :1959
##   3rd Qu.:2010   3rd Qu.:59.00   3rd Qu.:1973
##   Max.    :2014   Max.    :89.00   Max.    :1996
##   NA's    :76     NA's    :76     NA's    :76

```

### 8.5.4 Recode Categorical Variables

When transforming categorical variables, our goal is often to reduce the number of values. This can be done in different ways:

- combining values which are similar into a single value
- combining infrequent values into a “Various” or “Other” value.

The first way can be done with `fct_collapse`, which collapse factor levels into a new level. The second can be achieved with `fct_lump`, which “lumps” together infrequent values.

### 8.5.5 Collapsing factors

Collapsing a factor can be done as shown below. For each group of levels that you want to collapse, you create a vector. Subsequently, you can give each group a new name.<sup>14</sup> All levels which you do not mentioned will be left untouched.

```
fct_collapse(factor,
  new_group_1 = c("old_level_1", "old_level_2", "..."),
  new_group_2 = c("old_level_a", "old_level_b", "..."),
  ...)
```

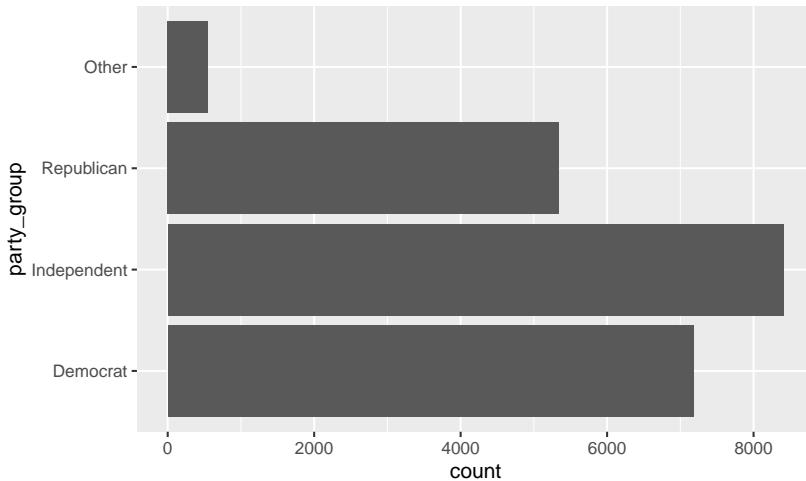
In our survey data, we can collapse the party variable into different groups: “Democrat”, “Republican”, “Independent” and “Other”. We save the new variable as `party_group`.

```
survey %>%
  mutate(party_group = fct_collapse(party,
    Other = c("No answer", "Don't know", "Other party"),
    Republican = c("Republican, strong", "Republican, weak"),
    Independent = c("Independent, near rep", "Independent", "Independent, strong"),
    Democrat = c("Democrat, weak", "Democrat, strong"))
)) -> survey
```

Transformations as these can be useful in there own right: reducing the number of categories as in this plot.

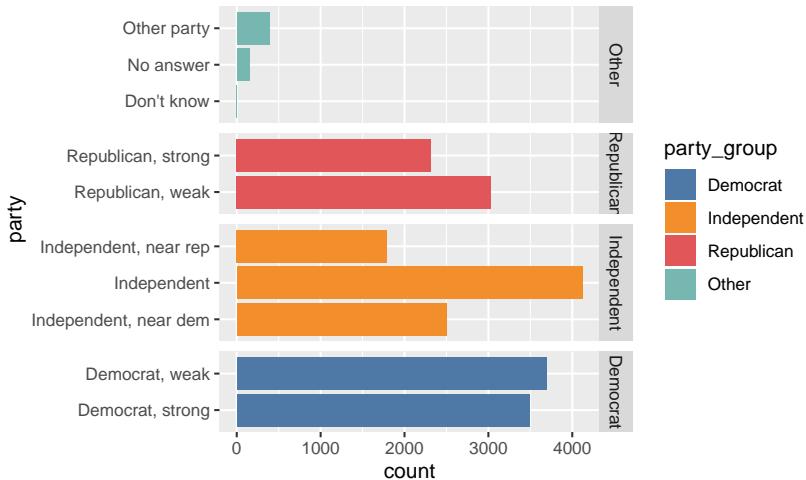
```
survey %>%
  ggplot(aes(party_group)) +
  geom_bar() +
  coord_flip()
```

<sup>14</sup> The attentive reader might have noticed a similarity between `fct_collapse` and `fct_recode`. Indeed, you could obtain the same result with the latter function, but you would have more typing work.



But they can as well be used in combination with the original levels. For example to improve visuals:<sup>15</sup>

```
library(ggthemes)
survey %>%
  ggplot(aes(party, fill = party_group)) +
  geom_bar() +
  facet_grid(fct_rev(party_group)~., scales = "free", space = "free") +
  coord_flip() +
  scale_fill_tableau()
```



Alternatively, we can use fct\_lump to create an other category. For example, take a look at the religions.

```
survey %>%
  count(religion)

## # A tibble: 15 x 2
```

<sup>15</sup> In this example, we use the scale\_fill\_tableau color scale from the package ggthemes to have a color scale where Democrats are blue and Republicans are red. Furthermore, we use fct\_rev to reverse the order of the factors. fct\_rev will be discussed in a few moments.

```

##   religion      n
##   <fct>      <int>
## 1 Buddhism     147
## 2 Catholic    5124
## 3 Christian    689
## 4 Don't know    15
## 5 Hinduism      71
## 6 Inter-nondenominational 109
## 7 Jewish        388
## 8 Moslem/islam   104
## 9 Native american  23
## 10 No answer     93
## 11 None         3523
## 12 Orthodox-christian  95
## 13 Other          224
## 14 Other eastern    32
## 15 Protestant    10846

```

Let's say we want to keep only the 5 most frequent religions. We can do this as follows.

```

survey %>%
  mutate(religion = fct_lump(religion, n = 5)) %>%
  count(religion)

## # A tibble: 6 x 2
##   religion      n
##   <fct>      <int>
## 1 Catholic    5124
## 2 Christian    689
## 3 Jewish        388
## 4 None         3523
## 5 Protestant   10846
## 6 Other         913

```

The "Other" label can be changed as you like.

```

survey %>%
  mutate(religion = fct_lump(religion, n = 5, other_level = "Other religions")) %>%
  count(religion)

## # A tibble: 6 x 2
##   religion      n
##   <fct>      <int>
## 1 Catholic    5124
## 2 Christian    689

```

```

## 3 Jewish          388
## 4 None            3523
## 5 Protestant       10846
## 6 Other religions  913

```

Instead of specifying the number of levels to retain, you can also specify a minimal relative frequencies using the `prop` argument.

```

survey %>%
  mutate(religion = fct_lump(religion, prop = 0.02)) %>%
  count(religion)

## # A tibble: 5 x 2
##   religion     n
##   <fct>     <int>
## 1 Catholic    5124
## 2 Christian   689
## 3 None        3523
## 4 Protestant  10846
## 5 Other       1301

```

Even more so than cleaning, all the transformations we have seen are very iterative in nature, and their use can be for specific analyses only. For example, we might want to lump infrequent religions to make a bar chart without too many bars – but we probably don’t want to remove infrequent religions all the way. Transformations will thus often happen in the build-up towards a chart or table, and not permanently saved in the data. This is especially true for the last functions we will discuss here: reordering functions.

## *8.6 Using transformations in visualizations.*

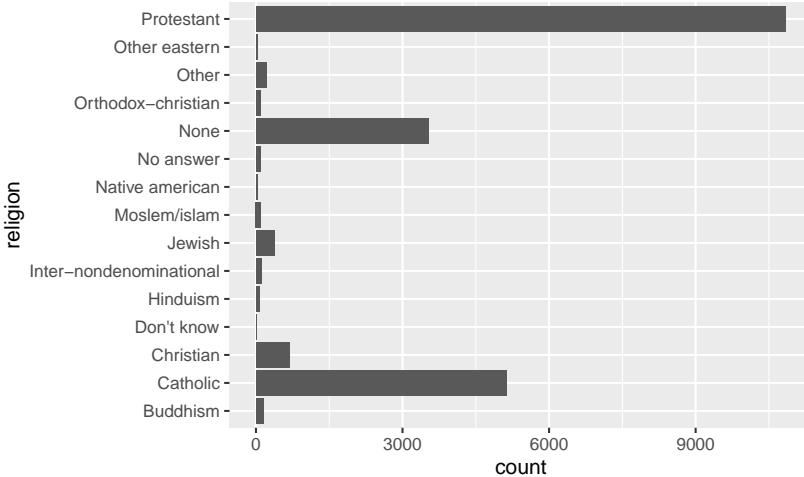
When visualizing categorical variables, we often want to change the order of levels according to frequency or based on another variable. We already saw `fct_relevel` to manually reorder levels, but it’s not very suited to automatically reorder levels. For this, we will use two new functions: `fct_infreq` and `fct_reorder`.

We start with the following graph.

```

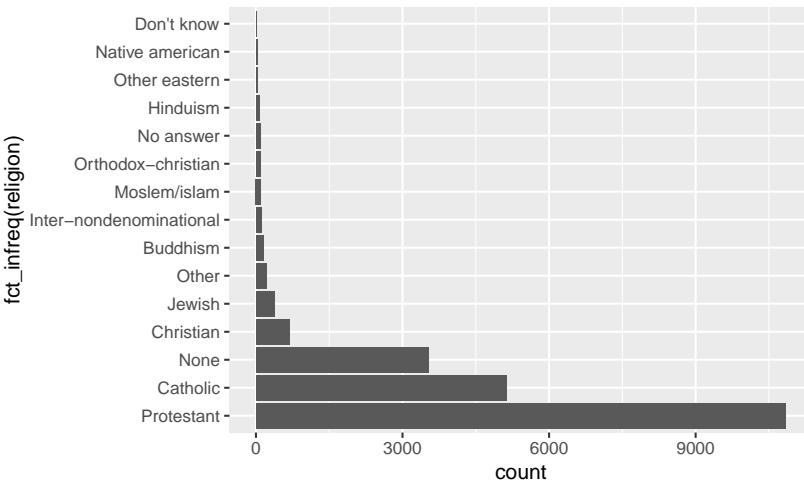
survey %>%
  ggplot(aes(religion)) +
  geom_bar() +
  coord_flip()

```



A factor can be ordered based on the (in)frequency of each level using the `fct_infreq` function. Its use is simple. We can choose to add the function directly within ggplot, or to update the religion variable using `mutate` in advance of plotting.

```
survey %>%
  ggplot(aes(fct_infreq(religion))) +
  geom_bar() +
  coord_flip()
```



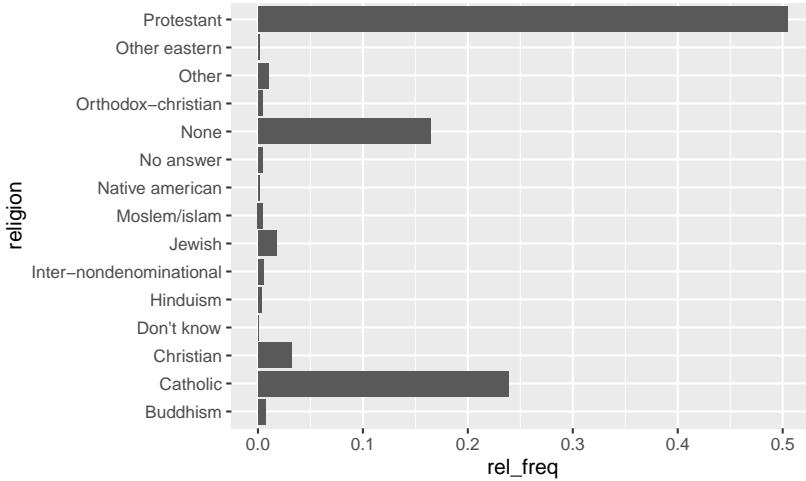
`fct_infreq` will count for each of the levels – religions in this case – how often it occurs, and reorder the levels accordingly. Now, suppose that we don't want absolute frequencies like in the last plot, but relative. In that case, we need to compute them ourselves and use `geom_col`.

```
survey %>%
  group_by(religion) %>%
```

```

summarise(freq = n()) %>%
mutate(rel_freq = freq/sum(freq)) %>%
ggplot(aes(x = religion, y = rel_freq)) +
geom_col() +
coord_flip()

```

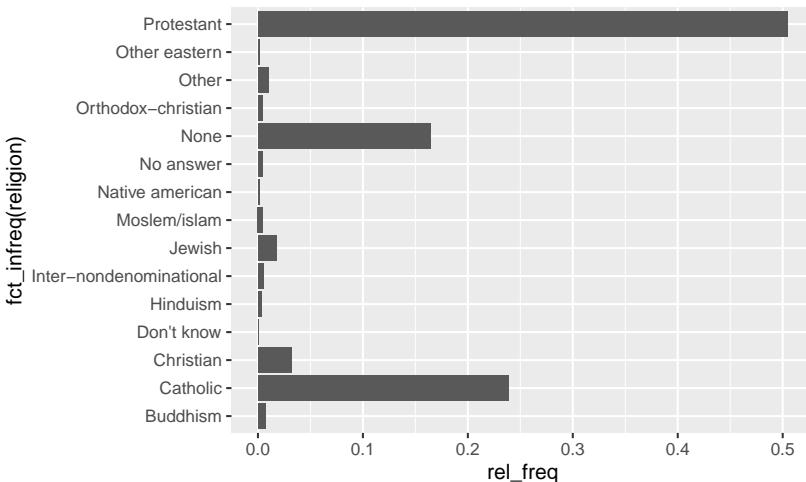


We now can see read the relative frequencies from the chart. Now let's order the bars once more.

```

survey %>%
group_by(religion) %>%
summarise(freq = n()) %>%
mutate(rel_freq = freq/sum(freq)) %>%
ggplot(aes(x = fct_infreq(religion), y = rel_freq)) +
geom_col() +
coord_flip()

```



Oops, that didn't seem to work. Why not? Let's look at the data we gave to ggplot.

```

survey %>%
  group_by(religion) %>%
  summarise(freq = n()) %>%
  mutate(rel_freq = freq/sum(freq))

## # A tibble: 15 x 3
##   religion           freq  rel_freq
##   <fct>            <int>    <dbl>
## 1 Buddhism          147  0.00684
## 2 Catholic          5124 0.239
## 3 Christian         689  0.0321
## 4 Don't know        15   0.000698
## 5 Hinduism          71   0.00330
## 6 Inter-nondenominational 109  0.00507
## 7 Jewish             388  0.0181
## 8 Moslem/islam      104  0.00484
## 9 Native american    23   0.00107
## 10 No answer         93   0.00433
## 11 None              3523 0.164
## 12 Orthodox-christian 95   0.00442
## 13 Other              224  0.0104
## 14 Other eastern      32   0.00149
## 15 Protestant         10846 0.505

```

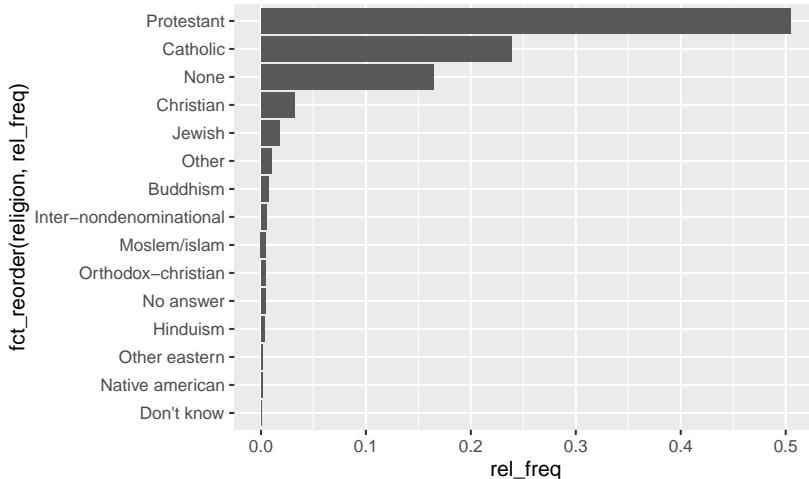
The data – a frequency table – contains one row for each religion. When we use `fct_infreq` on this table, all religions appear once, so there are all equally frequent. `fct_infreq` implicitly tries to compute frequencies, but we already did that. As a result, the ordering failed. It is similar to using `geom_bar` based on a frequency table – we are trying to compute the frequencies twice, which results in unintended plots.

So, what can we do instead? Well, we want to order the religions based on frequency. That shouldn't be hard, because the frequency is already there. The function `fct_reorder` can help us. In contrast to `fct_infreq` it will use a second variable we provide to order the factor. Thus:

```

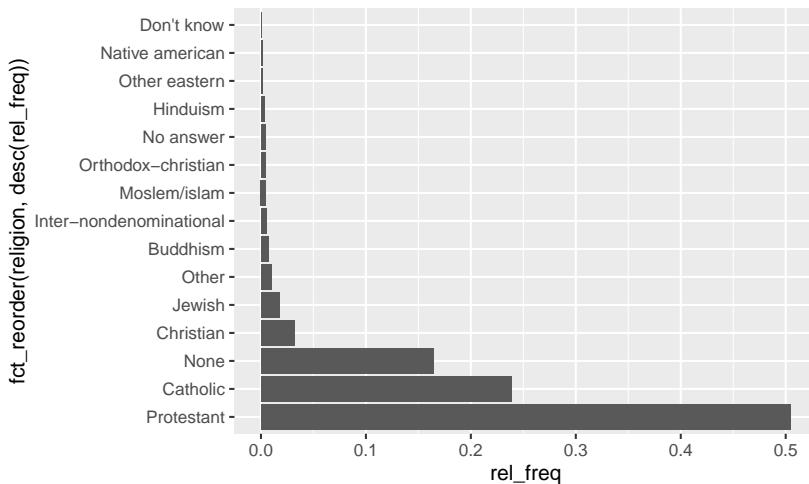
survey %>%
  group_by(religion) %>%
  summarise(freq = n()) %>%
  mutate(rel_freq = freq/sum(freq)) %>%
  ggplot(aes(x = fct_reorder(religion, rel_freq), y = rel_freq)) +
  geom_col() +
  coord_flip()

```



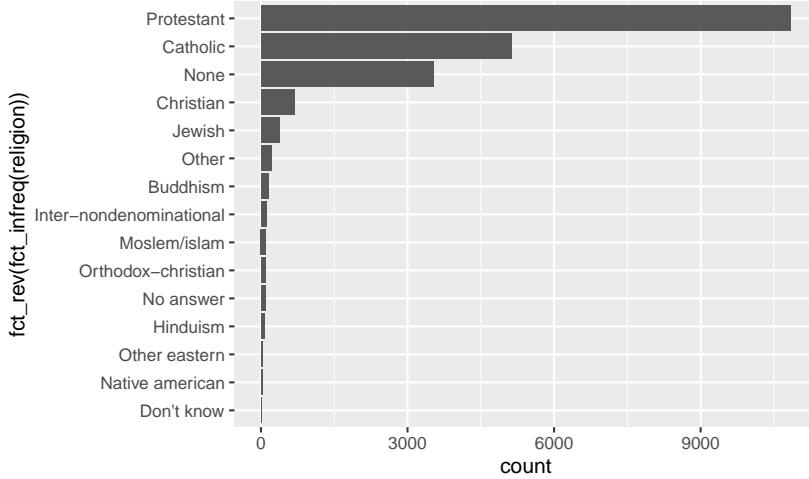
Note how we refer to `rel_freq` twice: once to order, and once to use as y-axis. Also note that the order of the bars is reversed: `fct_infreq` will always order from most to least frequent, while our current configuration with `fct_reorder` is ordering from least to most frequent. We can simply change it using `desc()`, as we have done before.

```
survey %>%
  group_by(religion) %>%
  summarise(freq = n()) %>%
  mutate(rel_freq = freq/sum(freq)) %>%
  ggplot(aes(x = fct_reorder(religion, desc(rel_freq)), y = rel_freq)) +
  geom_col() +
  coord_flip()
```



Alternatively, we can use the `fct_rev` function: this function will reverse the order of a factor. For example, we can reverse the order made by `fct_infreq`.

```
survey %>%
  ggplot(aes(fct_rev(fct_infreq(religion)))) +
  geom_bar() +
  coord_flip()
```



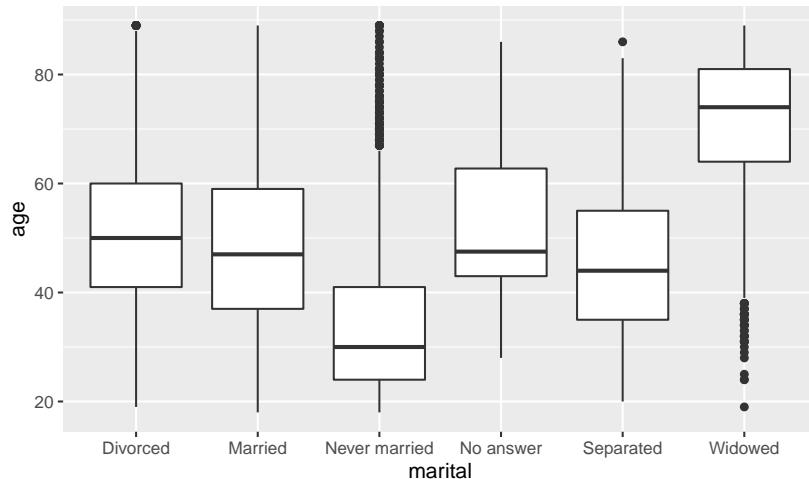
Thus, summarizing:

- `fct_infreq(f)`: reorder the levels of factor f from most to least frequent
- `fct_reorder(f, x)`: reorder the levels of factor f according to variable x
- `fct_rev(f)`: reverse the order of the levels of factor f

In general, you should use `fct_infreq` only on the original data, and when you have a frequency table as input to `ggplot`, you should use `fct_reorder`.

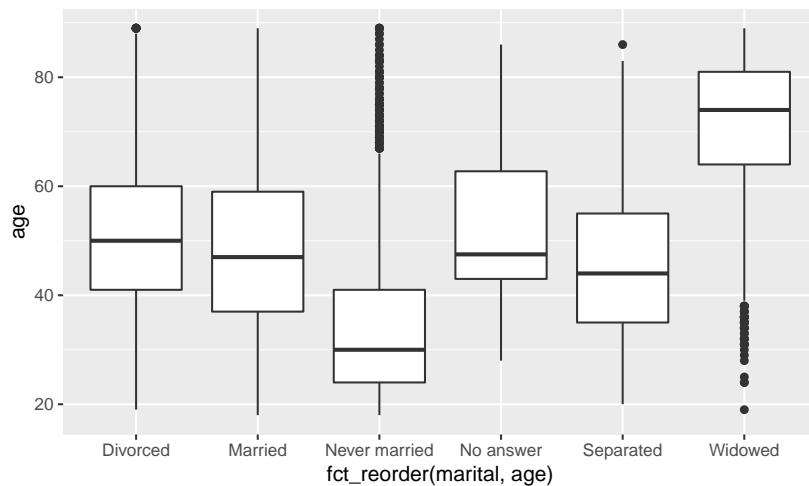
However, there is one more thing we need to cover. `fct_reorder(f, x)` can reorder on any variable x, not just frequency. For example, look at the following plot.

```
survey %>%
  ggplot(aes(marital, age)) +
  geom_boxplot()
```



This plot shows the distribution of age for different marital statuses. Let's say we want to sort these boxplots according to age. We use `fct_reorder` just as we did before.

```
survey %>%
  ggplot(aes(fct_reorder(marital, age), age)) +
  geom_boxplot()
```



Again, the results are not as we would expect. What's different compared to the previous use with frequencies?

There are actually two differences.

1. Before, we had a single frequency for each religion, making it easy to sort them.

Now, for each marital status, we have many persons with different ages. We need to summarize them in a single value, such as the mean or median. This can be done by adding the `.fun` argument in factor reorder.

```
fct_reorder(marital, age, .fun = median)
```

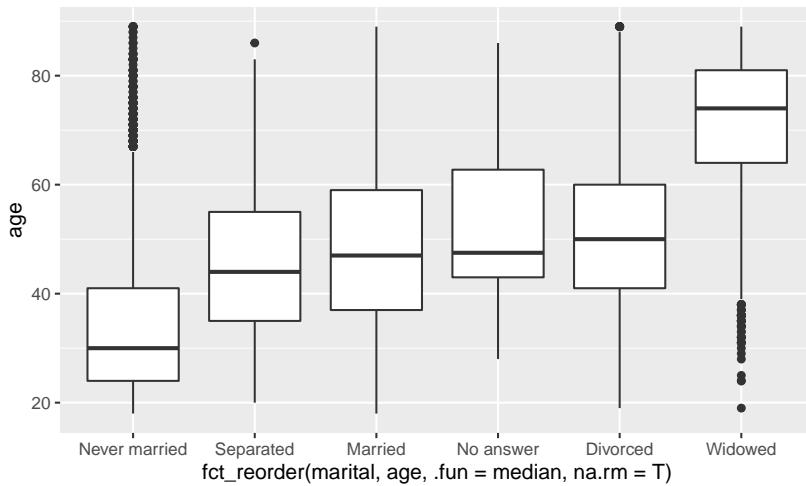
2. Some persons don't have an age, but a missing value.

Computing any function when there are missing values leads to a missing value. We have to make sure that missing values are ignored. Any argument we add to fct\_reorder after the .fun argument will be considered as a argument to this function. Thus, the following will correctly sort the marital variable

```
fct_reorder(marital, age, .fun = median, na.rm = T)
```

Putting it to the test:

```
survey %>%
  ggplot(aes(fct_reorder(marital, age, .fun = median, na.rm = T), age)) +
  geom_boxplot()
```



This looks better. So, let's summarize once again.

- **fct\_infreq(f)**: reorder the levels of factor f from most to least frequent
- **fct\_reorder**: reorder the levels of factor f according to variable x
  - **fct\_reorder(f, x)** when we are sure there is a single x-value for every f-level
  - **fct\_reorder(f, x, .fun = summarize\_function)** when there can be more than one x-value for some f-level. The **summarize\_function** will be used to combine multiple values. This can be mean, median, sum, length, ... any function that returns a single value.
  - If we need to give additional arguments to the summarize function, such as na.rm = T, we can do this as follows: **fct\_reorder(f, x, .fun = summarize\_function, na.rm = T)**.
- **fct\_rev(f)**: reverse the order of the levels of factor f

### *8.7 Background Material*

- More information on cleaning and transformation can be found in Prof. dr. Depaire's Lecture Notes
- More information onforcats can be read in Chapter 15 of R for Data Science



# 9

## *Tidy data*

### *9.1 Inleiding*

- In werkelijkheid komt data niet altijd in het geschikte formaat om de gewenste analyses op uit te voeren.
  - Vaak is data verspreid over meerdere datasets en moeten we hier 1 dataframe van maken voor onze analyses.
  - Soms stelt een rij niet de observatie voor die willen bestuderen (bv: één rij stelt de gegevens van één auto betrokken in een ongeval voor, terwijl we willen dat iedere rij een ongeval voorstelt met de gegevens van alle betrokken voertuigen).
- Het manipuleren van de data opdat het in het juiste formaat staat, wordt ook wel de creatie van ‘tidy data’ genoemd.
- **Bestudeer secties 12.1 tot en met 12.4 en 12.6 in ‘R for Data Science’ van Grolemund en Wickham!**
- **Bestudeer hoofdstuk 13 in ‘R for Data Science’ van Grolemund en Wickham!**

### *9.2 Case: NYC Vluchten 2013*

#### *9.2.1 Datasets samenvoegen*

- We vertrekken van een dataset met vluchten opgestegen vanuit NYC in 2013. Hieronder een overzicht van de variabelen in de dataset.

```
glimpse(df)

## # Observations: 319,809
## # Variables: 13
## $ id                  <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ...
## $ vertrekvluchthaven <chr> "EWR", "LGA", "JFK", "LGA", "EWR", "EWR", "LGA"...
## $ aankomstvluchthaven <chr> "George Bush Intercontinental", "George Bush In...
```

```

## $ maatschappij      <chr> "United Air Lines Inc.", "United Air Lines Inc...."
## $ tijdstip_aankomst <dttm> 2013-01-01 08:30:00, 2013-01-01 08:50:00, 2013...
## $ vertrek_vertraging <dbl> 2, 4, 2, -6, -4, -5, -3, -2, -2, -2, -2, -2...
## $ aankomst_vertraging <dbl> 11, 20, 33, -25, 12, 19, -14, -8, 8, -2, -3, 7, ...
## $ afstand           <dbl> 1400, 1416, 1089, 762, 719, 1065, 229, 944, 733...
## $ tijdstip_vertrek   <dttm> 2013-01-01 05:17:00, 2013-01-01 05:33:00, 2013...
## $ weekdag_vertrek    <ord> Tue, Tue, Tue, Tue, Tue, Tue, Tue, Tue, Tu...
## $ week_vertrek       <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ maand_vertrek      <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ maanddag_vertrek    <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...

```

- We beschikken nu ook over een tweede dataset met de gegevens van de luchthavens. Hieronder een overzicht van de variabelen in deze dataset.

```
glimpse(airports)
```

```

## Observations: 1,458
## Variables: 8
## $ faa    <chr> "04G", "06A", "06C", "06N", "09J", "0A9", "0G6", "0G7", "0P2"...
## $ name   <chr> "Lansdowne Airport", "Moton Field Municipal Airport", "Schaum...
## $ lat    <dbl> 41.13047, 32.46057, 41.98934, 41.43191, 31.07447, 36.37122, 4...
## $ lon    <dbl> -80.61958, -85.68003, -88.10124, -74.39156, -81.42778, -82.17...
## $ alt    <dbl> 1044, 264, 801, 523, 11, 1593, 730, 492, 1000, 108, 409, 875, ...
## $ tz     <dbl> -5, -6, -6, -5, -5, -5, -5, -5, -8, -5, -6, -5, -5, -5, -...
## $ dst    <chr> "A", "A", "A", "A", "A", "A", "U", "A", "A", "U", "...
## $ tzone  <chr> "America/New_York", "America/Chicago", "America/Chicago", "Am...

```

- Als we deze datasets vergelijken zien we een mogelijke relatie tussen beiden.
  - In de oorspronkelijke dataset stelt iedere rij een vlucht voor en wordt de vertrekvluchthaven voorgesteld door een 3-letterige code.
  - In de airports-dataset stelt iedere rij een luchthaven voor en vinden we een 3-letterige code terug in de kolom ‘faa’.
- We willen nu graag deze twee datasets aan elkaar koppelen door de gegevens van de vertrekvluchthavens uit de airports-dataset te halen en toe te voegen aan iedere vlucht.
- Alvorens we dit kunnen doen, moeten we eerst controleren of de faa-code in de airports-dataset uniek is.
  - Dit is een essentiële vereiste om de gegevens van de airports-dataset te kunnen toevoegen aan de oorspronkelijke dataset.
  - Indien er bijvoorbeeld 2 luchthavens in de airports-dataset zouden zitten met faa-code ‘EWR’, dan zou R niet kunnen achterhalen van welke luchthaven de gegevens moeten worden toegevoegd aan de vluchten met als vertrekvluchthaven ‘EWR’.

- In zulke gevallen gaat R de vlucht duplicerend en iedere kopie (van de vlucht) koppelen aan een andere luchthaven uit de airports-dataset met faa-code EWR.

```

airports %>%
  count(faa) %>%
  filter(n>1)

## # A tibble: 0 x 2
## # ... with 2 variables: faa <chr>, n <int>

• Uit bovenstaande analyse blijkt dat er geen twee rijen zijn in de airports-dataset met dezelfde faa-code.

• We kunnen nu de gegevens van de airports-dataset toevoegen aan het oorspronkelijk dataframe. We doen dit met behulp van een left_join() en geven aan via welke variabelen de link gelegd moet worden.

df <- df %>% left_join(airports, by=c("vertrek_luchthaven"="faa"))
glimpse(df)

## # Observations: 319,809
## # Variables: 20
## $ id                  <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ...
## $ vertrek_luchthaven <chr> "EWR", "LGA", "JFK", "LGA", "EWR", "EWR", "LGA"...
## $ aankomst_luchthaven <chr> "George Bush Intercontinental", "George Bush In...
## $ maatschappij        <chr> "United Air Lines Inc.", "United Air Lines Inc.....
## $ tijdstip_aankomst   <dttm> 2013-01-01 08:30:00, 2013-01-01 08:50:00, 2013...
## $ vertrek_vertraging  <dbl> 2, 4, 2, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2...
## $ aankomst_vertraging <dbl> 11, 20, 33, -25, 12, 19, -14, -8, 8, -2, -3, 7, ...
## $ afstand              <dbl> 1400, 1416, 1089, 762, 719, 1065, 229, 944, 733...
## $ tijdstip_vertrek    <dttm> 2013-01-01 05:17:00, 2013-01-01 05:33:00, 2013...
## $ weekdag_vertrek     <ord> Tue, Tue, Tue, Tue, Tue, Tue, Tue, Tu...
## $ week_vertrek         <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ maand_vertrek        <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ maanddag_vertrek     <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ name                 <chr> "Newark Liberty Intl", "La Guardia", "John F Ke...
## $ lat                  <dbl> 40.69250, 40.77725, 40.63975, 40.77725, 40.6925...
## $ lon                  <dbl> -74.16867, -73.87261, -73.77893, -73.87261, -74...
## $ alt                  <dbl> 18, 22, 13, 22, 18, 18, 22, 13, 22, 13, 13, ...
## $ tz                   <dbl> -5, -5, -5, -5, -5, -5, -5, -5, -5, -5, -5, ...
## $ dst                  <chr> "A", "A", "A", "A", "A", "A", "A", "A", "A", ...
## $ tzone                <chr> "America/New_York", "America/New_York", "Americ...

```

- Bovenstaande output laat zien dat 7 kolommen zijn toegevoegd aan de oorspronkelijke dataset.

- Merk op dat de faa-kolom van het airports-dataframe niet is toegevoegd.  
Dit is niet nodig aangezien we in de join-functie hadden aangegeven dat deze kolom overeenkwam met de kolom vertrekvluchthaven uit de oorspronkelijke dataset.
- Controleer ook altijd of het aantal observaties niet gewijzigd is, daar dit vaak wijst op een fout in de join. In dit geval is het aantal observaties niet veranderd.
- In een volgende stap verwijderen we een aantal kolommen die we verder niet nodig gaan hebben en veranderen we de kolom ‘name’ in ‘vertrekvluchthaven’. Zoals je in het resultaat kan zien bevat onze nieuwe dataset nu de volledige naam van de vertrekvluchthaven en niet enkel de faa-code.

```
df <- df %>%
  select(-vertrekvluchthaven, -lat, -lon, -alt, -tz, -dst, -tzone) %>%
  rename(vertrekvluchthaven = name)
glimpse(df)

## Observations: 319,809
## Variables: 13
## $ id              <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ...
## $ aankomstluchthaven <chr> "George Bush Intercontinental", "George Bush In...
## $ maatschappij      <chr> "United Air Lines Inc.", "United Air Lines Inc.....
## $ tijdstip_aankomst <dttm> 2013-01-01 08:30:00, 2013-01-01 08:50:00, 2013...
## $ vertrek_vertraging <dbl> 2, 4, 2, -6, -4, -5, -3, -2, -2, -2, -2, -2...
## $ aankomst_vertraging <dbl> 11, 20, 33, -25, 12, 19, -14, -8, 8, -2, -3, 7, ...
## $ afstand           <dbl> 1400, 1416, 1089, 762, 719, 1065, 229, 944, 733...
## $ tijdstip_vertrek   <dttm> 2013-01-01 05:17:00, 2013-01-01 05:33:00, 2013...
## $ weekdag_vertrek    <ord> Tue, Tue, Tue, Tue, Tue, Tue, Tue, Tu...
## $ week_vertrek       <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ maand_vertrek      <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ maanddag_vertrek   <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ vertrekvluchthaven <chr> "Newark Liberty Intl", "La Guardia", "John F Ke...
```

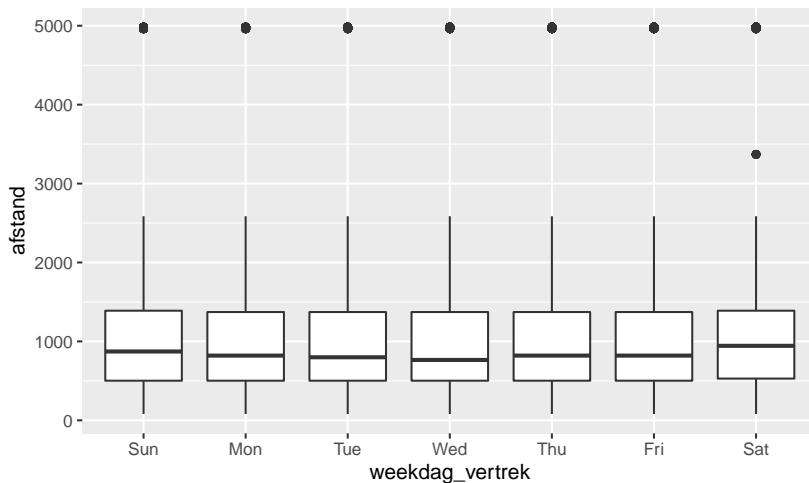
### 9.3 Data in een lang formaat plaatsen (voor visuele analyses)

- Bij een bivariate visualisatie heb je steeds het basisprincipe dat je de relatie tussen twee variabelen wenst weer te geven.
- Bij een multivariate visualisatie ga je vaak weergeven hoe deze relatie verandert in functie van een derde variabele.
- Deze derde variabele is vaak categorisch en de verschillende categorieën stellen hierbij groeperingen van de observaties voor waarvoor je de relatie tussen X en Y wenst weer te geven.
  - Je wil bijvoorbeeld initieel de relatie tussen weekdag en afstand van de vluchten weergeven. Hiervoor kan je een bivariate plot

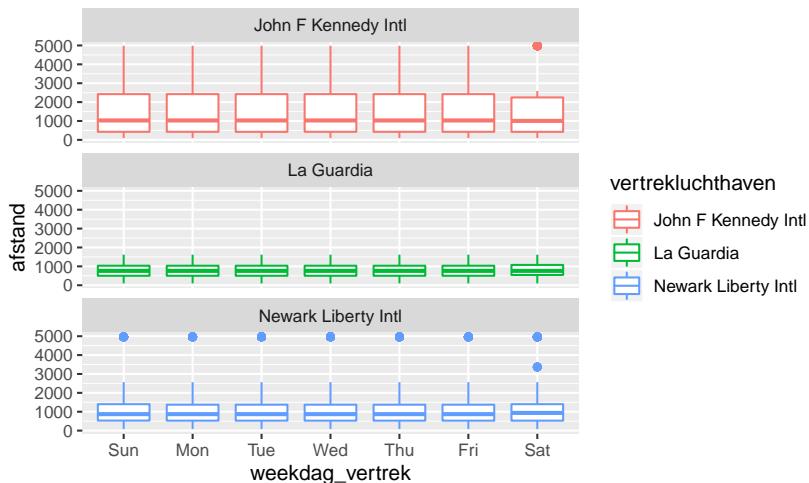
maken waarbij X categorisch is en Y continu. Een mogelijkheid hiervoor is een boxplot.

- In een volgende stap kan je de relatie tussen afstand en weekdag opsplitsen per luchthaven. Je wil dus weten hoe deze relatie verschilt tussen diverse luchthavens. Hiervoor gebruik je de categorische variabele ‘vertrekluchthaven’ en kan je bijvoorbeeld de kleur van de boxplot koppelen aan de vertrekluchthaven of aparte ‘facets’ maken voor iedere luchthaven.
- Hieronder zie je de bijhorende plots.

```
df %>%
  ggplot(aes(x=weekdag_vertrek, y=afstand)) +
  geom_boxplot()
```



```
df %>%
  ggplot(aes(x=weekdag_vertrek, y=afstand, colour=vertrekluchthaven)) +
  geom_boxplot() +
  facet_wrap(~vertrekluchthaven, ncol=1)
```



- Stel nu dat je het effect wenst te weten van de weekdag van vertrek op de vertraging van een vlucht, maar je wil hierbij onderscheid maken tussen vertrek- en aankomstvertraging.
- Volgens bovenstaande aanpak zou je dan een Y-variabele moeten hebben die de vertraging meet en een Z-variabele die het type van vertraging aangeeft (aankomst of vertrek).
- Onze dataset is echter anders opgebouwd. In de beschikbare data is de vertraging van een vlucht opgeslagen met behulp van twee aparte variabelen, namelijk vertrek- en aankomstvertraging. Dit blijkt uit onderstaande tabel.

```
df_temp <- df %>%
  select(id, vertrekvluchthaven, vertrek_vertraging, aankomst_vertraging, weekdag_vertrek)
df_temp

## # A tibble: 319,809 x 5
##       id vertrekvluchthaven  vertrek_vertraging aankomst_vertraging weekdag_vertrek
##   <int> <chr>                <dbl>            <dbl> <ord>
## 1     1 Newark Liberty Int~        2             11 Tue 
## 2     2 La Guardia              4             20 Tue 
## 3     3 John F Kennedy Int~      2             33 Tue 
## 4     4 La Guardia             -6            -25 Tue 
## 5     5 Newark Liberty Int~     -4             12 Tue 
## 6     6 Newark Liberty Int~     -5             19 Tue 
## 7     7 La Guardia             -3            -14 Tue 
## 8     8 John F Kennedy Int~     -3             -8 Tue 
## 9     9 La Guardia             -2              8 Tue 
## 10    10 John F Kennedy Int~    -2             -2 Tue 
## # ... with 319,799 more rows
```

- We moeten de data dus omzetten zodat het type vertraging niet

gecodeerd wordt als aparte variabelen, maar door middel van 1 categorische variabele.

- Hiervoor kunnen we de *gather()* functie hanteren. Deze functie zal een set van variabelen (in dit geval ‘vertrek\_vertraging’ en ‘aankomst\_vertraging’) transformeren naar 2 variabelen, namelijk een key-variabele en een value-variabele.
  - De key-variabele is een categorische variabele en de categorieën komen overeen met de variabelennamen in onze set van variabelen die we wensen te transformeren. In ons geval zijn dit dus de categorieën ‘vertrek\_vertraging’ en ‘aankomst\_vertraging’.
  - De value-variabele bevat de bijhorende waarde uit de oorspronkelijke dataset.
- De *gather()* functie bestaat uit 3 delen.
  - Eerst vermeld je alle variabelen die je wenst te vervangen.
  - Vervolgens geef je de naam van de nieuwe key-variabele.
  - Tenslotte geef je de naam van de nieuwe value-variabele.

```
df_long <- df_temp %>%
  gather(vertrek_vertraging, aankomst_vertraging, key="type_vertraging", value="vertraging") %>%
  arrange(id)
df_long

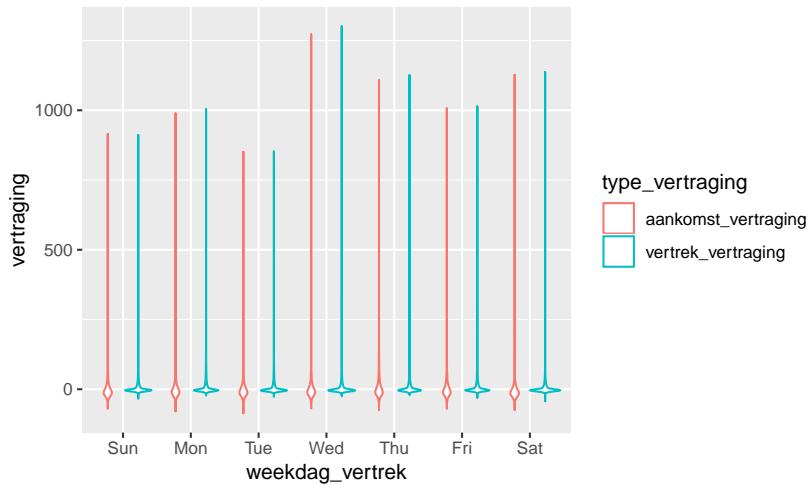
## # A tibble: 639,618 x 5
##       id vertrekluchthaven weekdag_vertrek type_vertraging    vertraging
##   <int> <chr>           <ord>          <chr>            <dbl>
## 1     1 Newark Liberty Intl Tue  vertrek_vertraging      2
## 2     1 Newark Liberty Intl Tue  aankomst_vertraging    11
## 3     2 La Guardia           Tue  vertrek_vertraging      4
## 4     2 La Guardia           Tue  aankomst_vertraging    20
## 5     3 John F Kennedy Intl Tue  vertrek_vertraging      2
## 6     3 John F Kennedy Intl Tue  aankomst_vertraging    33
## 7     4 La Guardia           Tue  vertrek_vertraging     -6
## 8     4 La Guardia           Tue  aankomst_vertraging   -25
## 9     5 Newark Liberty Intl Tue  vertrek_vertraging     -4
## 10    5 Newark Liberty Intl Tue  aankomst_vertraging    12
## # ... with 639,608 more rows
```

- Merk op dat het aantal rijen nu verdubbeld is. Dit komt omdat je nu voor zowel vertrek- als aankomstvertraging een aparte rij hebt gecreëerd.
  - Hierdoor krijg je een andere definitie van de observatie die in een rij staat. In de oorspronkelijke dataset was iedere rij (observatie) een vlucht vanuit NYC in 2013. In de nieuwe dataset stelt iedere

rij het vertrek of de aankomst van een vlucht vanuit NYC in 2013 voor!

- Indien je dus de `gather()` functie hanteert gaat het aantal rijen toenemen. Het aantal kolommen zal afnemen indien de variabelenset, die je wenst te transformeren, uit meer dan 2 variabelen bestaat.
- Hierdoor krijg je een dataset die minder breed is en vooral langer. Daarom wordt dit het lange formaat genoemd.
- Data in een lang formaat zijn voornamelijk nuttig om visualisaties te realiseren met ggplot.
- Met dit lange formaat kunnen we de relatie tussen weekdag van vertrek en de vertraging, uitgesplitst volgens vertrek- of aankomstvertraging, visualiseren.

```
df_long %>%
  ggplot(aes(x=weekdag_vertrek, y= vertraging, colour=type_vertraging)) + geom_violin()
```



- Indien we de relatie tussen we weekdag en de gemiddelde vertraging, uitgesplitst volgens vertragingstype, wensen te visualiseren, moeten we eerst de gemiddelde vertraging berekenen.

```
df_long_summary <- df_long %>%
  group_by(vertrekluchthaven, type_vertraging, weekdag_vertrek) %>%
  summarise(gem_vertraging = mean(vertraging))
df_long_summary

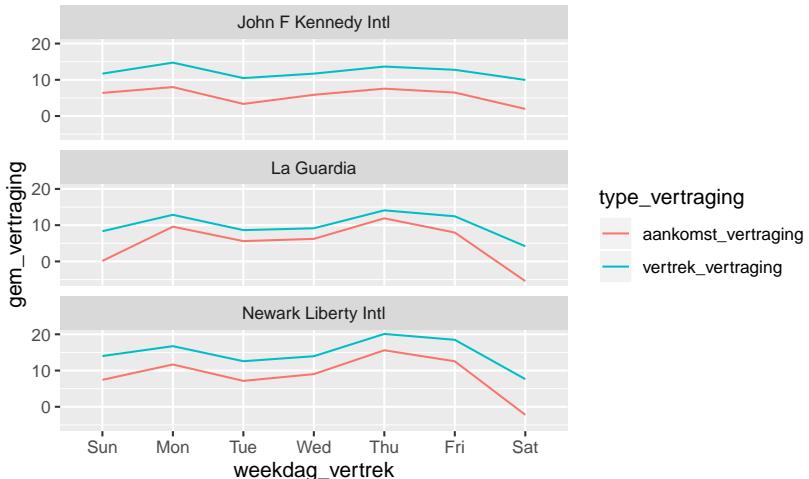
## # A tibble: 42 x 4
## # Groups:   vertrekluchthaven, type_vertraging [6]
##       vertrekluchthaven   type_vertraging   weekdag_vertrek   gem_vertraging
##       <chr>           <chr>           <ord>           <dbl>
## 1 John F Kennedy Intl aankomst_vertraging Sun        6.39
```

```

## 2 John F Kennedy Intl aankomst_vertraging Mon 7.99
## 3 John F Kennedy Intl aankomst_vertraging Tue 3.34
## 4 John F Kennedy Intl aankomst_vertraging Wed 5.86
## 5 John F Kennedy Intl aankomst_vertraging Thu 7.56
## 6 John F Kennedy Intl aankomst_vertraging Fri 6.49
## 7 John F Kennedy Intl aankomst_vertraging Sat 1.96
## 8 John F Kennedy Intl vertrek_vertraging Sun 11.7
## 9 John F Kennedy Intl vertrek_vertraging Mon 14.7
## 10 John F Kennedy Intl vertrek_vertraging Tue 10.5
## # ... with 32 more rows

df_long_summary %>%
  ggplot(aes(x=weekdag_vertrek,
             y=gem_vertraging,
             colour=type_vertraging,
             group=interaction(type_vertraging,vertrekluchthaven))) +
  geom_line() + facet_wrap(~ vertrekluchthaven, ncol=1)

```



#### 9.4 Data in een breed formaat plaatsen (voor overzichtelijke tabellen)

- Voor de laatste visualisatie hebben we een dataset gecreëerd met gemiddelde vertragingen per vertrekluchthaven, weekdag van vertrek en type vertraging.
- Om snel verbanden te zoeken en te evalueren is dit formaat niet erg handig. Voor zulke situaties kan je best voor een breed formaat opteren.
  - Hierbij moet je 2 variabelen selecteren: de key-variabele en de value-variabele.

Table 9.1: Gemiddelde vertraging  
(lang formaat)

weekdag_vertrek	vertrekluchthaven	type_vertraging	gem_vertraging
Sun	John F Kennedy Intl	aankomst_vertraging	6.39
Sun	John F Kennedy Intl	vertrek_vertraging	11.70
Sun	La Guardia	aankomst_vertraging	0.15
Sun	La Guardia	vertrek_vertraging	8.33
Sun	Newark Liberty Intl	aankomst_vertraging	7.44
Sun	Newark Liberty Intl	vertrek_vertraging	14.01
Mon	John F Kennedy Intl	aankomst_vertraging	7.99
Mon	John F Kennedy Intl	vertrek_vertraging	14.74
Mon	La Guardia	aankomst_vertraging	9.58
Mon	La Guardia	vertrek_vertraging	12.86
Mon	Newark Liberty Intl	aankomst_vertraging	11.67
Mon	Newark Liberty Intl	vertrek_vertraging	16.73
Tue	John F Kennedy Intl	aankomst_vertraging	3.34
Tue	John F Kennedy Intl	vertrek_vertraging	10.47
Tue	La Guardia	aankomst_vertraging	5.60
Tue	La Guardia	vertrek_vertraging	8.63
Tue	Newark Liberty Intl	aankomst_vertraging	7.15
Tue	Newark Liberty Intl	vertrek_vertraging	12.57
Wed	John F Kennedy Intl	aankomst_vertraging	5.86
Wed	John F Kennedy Intl	vertrek_vertraging	11.71
Wed	La Guardia	aankomst_vertraging	6.23
Wed	La Guardia	vertrek_vertraging	9.15
Wed	Newark Liberty Intl	aankomst_vertraging	9.02
Wed	Newark Liberty Intl	vertrek_vertraging	13.95
Thu	John F Kennedy Intl	aankomst_vertraging	7.56
Thu	John F Kennedy Intl	vertrek_vertraging	13.65
Thu	La Guardia	aankomst_vertraging	11.89
Thu	La Guardia	vertrek_vertraging	14.10
Thu	Newark Liberty Intl	aankomst_vertraging	15.60
Thu	Newark Liberty Intl	vertrek_vertraging	20.10
Fri	John F Kennedy Intl	aankomst_vertraging	6.49
Fri	John F Kennedy Intl	vertrek_vertraging	12.76
Fri	La Guardia	aankomst_vertraging	7.97
Fri	La Guardia	vertrek_vertraging	12.45
Fri	Newark Liberty Intl	aankomst_vertraging	12.55
Fri	Newark Liberty Intl	vertrek_vertraging	18.49
Sat	John F Kennedy Intl	aankomst_vertraging	1.96
Sat	John F Kennedy Intl	vertrek_vertraging	9.97
Sat	La Guardia	aankomst_vertraging	-5.44
Sat	La Guardia	vertrek_vertraging	4.19
Sat	Newark Liberty Intl	aankomst_vertraging	-2.22
Sat	Newark Liberty Intl	vertrek_vertraging	7.63

- De key-variabele is altijd een categorische variabele en de value-variabele kan zowel categorisch als continu zijn.
- Voor ieder level van de categorische key-variabele zal er een aparte kolom aangemaakt worden.
- Je kan een dataset van lang naar breed formaat omzetten met behulp van de *spread()* functie.

```
df_long_summary %>%
  spread(key=weekdag_vertrek, value=gem_vertraging) %>%
  arrange(vertrekluchthaven, type_vertraging)
```

vertrekluchthaven	type_vertraging	Table 9.2: Gemiddelde vertraging (breed-formaat)						
		Sun	Mon	Tue	Wed	Thu	Fri	Sat
John F Kennedy Intl	aankomst_vertraging	6.39	7.99	3.34	5.86	7.56	6.49	1.96
John F Kennedy Intl	vertrek_vertraging	11.70	14.74	10.47	11.71	13.65	12.76	9.97
La Guardia	aankomst_vertraging	0.15	9.58	5.60	6.23	11.89	7.97	-5.44
La Guardia	vertrek_vertraging	8.33	12.86	8.63	9.15	14.10	12.45	4.19
Newark Liberty Intl	aankomst_vertraging	7.44	11.67	7.15	9.02	15.60	12.55	-2.22
Newark Liberty Intl	vertrek_vertraging	14.01	16.73	12.57	13.95	20.10	18.49	7.63

## 9.5 Referenties

1. ‘R for Data Science’ van Grolemund en Wickham



# 10

## *Tutorial - Tidy data*

### *10.1 Before you start*

During this tutorial, we'll use several r-packages. Make sure to install and load them, if needed.

```
library(dplyr)
library(tidyr)
library(stringr)
```

Our old friend dplyr will provide us with some functions to combined different datasets into one. We will use tidyr to transform datasets, and stringr to do some manipulations of character variables.

This tutorial consist of two major parts:

1. Merging datasets
2. Transforming datasets

Thereafter, in an additional part, we will go through a case study as an example.

### *Disclaimer*

1. There are many datasets used in this tutorial. A loadscript has been provided to create all datasets for you. Just run the script and you are good to go.

Let's get started!

### *10.2 Merging data*

We can merge different datasets by **joining** or **binding**.

- We **join** different datasets which contain different information about the same observations. For example, we can have 1) a dataset

of countries with their population and 2) a dataset of countries with their life expectancy. These we can *join* together.

```
countries_population
```

```
## # A tibble: 114 x 2
##   country      pop
##   <fct>     <int>
## 1 Botswana    1639131
## 2 Greece      10706290
## 3 South Africa 43997828
## 4 Ethiopia     76511887
## 5 Zimbabwe     12311143
## 6 Yemen, Rep.  22211743
## 7 Nepal        28901790
## 8 Netherlands   16570613
## 9 United States 301139947
## 10 New Zealand  4115771
## # ... with 104 more rows
```

```
countries_lifeExp
```

```
## # A tibble: 85 x 2
##   country    lifeExp
##   <fct>     <dbl>
## 1 United States  78.2
## 2 Argentina     75.3
## 3 Korea, Dem. Rep. 67.3
## 4 Bulgaria      73.0
## 5 Chile          78.6
## 6 Croatia        75.7
## 7 Canada         80.7
## 8 Honduras       70.2
## 9 Liberia        45.7
## 10 Mexico         76.2
## # ... with 75 more rows
```

- We **bind** different datasets which contain the same information on different observations. For example, we can have 1) a dataset of European countries with their population and 2) a dataset of African countries with their population. We can *bind* these two together.<sup>1</sup>

```
population_africa
```

```
## # A tibble: 52 x 2
```

<sup>1</sup> There are actually other cases in which we can bind datasets together, but please don't bother about that for now. Just remind: bind different observations, join different information]

```

##   country          pop
##   <fct>           <int>
## 1 Algeria        33333216
## 2 Angola         12420476
## 3 Benin          8078314
## 4 Botswana       1639131
## 5 Burkina Faso  14326203
## 6 Burundi         8390505
## 7 Cameroon       17696293
## 8 Central African Republic 4369038
## 9 Chad            10238807
## 10 Comoros        710960
## # ... with 42 more rows

population_europe

## # A tibble: 30 x 2
##   country          pop
##   <fct>           <int>
## 1 Albania        3600523
## 2 Austria         8199783
## 3 Belgium        10392226
## 4 Bosnia and Herzegovina 4552198
## 5 Bulgaria       7322858
## 6 Croatia         4493312
## 7 Czech Republic 10228744
## 8 Denmark         5468120
## 9 Finland         5238460
## 10 France         61083916
## # ... with 20 more rows

```

Let's see how we can join data.

### 10.2.1 Joining data

Remember, we join datasets if they contain different information on the same observations. This means that there needs to be a way to *link* the datasets. These links we call *ids* or *keys*.

If we have population and life expectancy data about countries, than the name, code or abbreviation of the country is our key to link both datasets.

Note that, when both datasets use different keys, for example one uses the name (Belgium) and the other the code (BE), we cannot join them. In such a case, we would need to recode one of the variables or find another datasets which can serve as an intermediary link (i.e. one

that contains both the names and the codes. There exist many different country codes, so this is a common problem. But we are good to go in our case)

The join functions we will introduce in a second will always look for variables with the same names in both tables and uses these as the keys to link them. You can explicitly set the keys using the by argument. This is especially useful if

- a) The keys have a different name in both datasets. For example  
country vs ctry
- b) Not all common variables are actually keys.

For now, we will always let the keys be chosen by the functions. A message will tell us which keys they used.

Now, there are 4 ways to join datasets.

- inner\_join
- left\_join
- right\_join
- full\_join

Why four? Well, if we want to join two datasets, it typically happens that they don't contain information on *exactly* the same observations. Have a closer look at the population and life expectancy data. The first one contains information on 114 countries and the second one contains information on 85 countries. So they can impossibly contain information on the same set of countries. The different joins will tackles this problem differently.

### 10.2.2 Inner join

Inner join means: I only keep information about keys that occur in both tables. So, if I don't have the population of country A, I don't want its life expectancy.

```
inner_join(countries_population, countries_lifeExp)

## Joining, by = "country"

## # A tibble: 73 x 3
##   country          pop  lifeExp
##   <fct>        <int>   <dbl>
## 1 Botswana      1639131   50.7
## 2 South Africa  43997828   49.3
## 3 Ethiopia      76511887   52.9
## 4 Zimbabwe      12311143   43.5
## 5 Yemen, Rep.  22211743   62.7
```

```

## 6 Netherlands      16570613    79.8
## 7 United States 301139947   78.2
## 8 Kuwait          2505559     77.6
## 9 Colombia        44227550    72.9
## 10 Austria         8199783     79.8
## # ... with 63 more rows

```

This join gives us 73 observations, which is the subset of countries on which we have both types of information. Also note how the `inner_join` tells you which key it used.

### 10.2.3 Left join

Left join means: I keep all information in my first (left) table. So, even if I don't have the life expectancy, still give me the population. The missing part of the new observation (i.e. the life expectancy), is now NA.

```

left_join(countries_population, countries_lifeExp)

## Joining, by = "country"

## # A tibble: 114 x 3
##       country           pop  lifeExp
##       <fct>        <int>   <dbl>
## 1 Botswana      1639131   50.7
## 2 Greece        10706290    NA
## 3 South Africa  43997828   49.3
## 4 Ethiopia       76511887   52.9
## 5 Zimbabwe      12311143   43.5
## 6 Yemen, Rep.   22211743   62.7
## 7 Nepal          28901790    NA
## 8 Netherlands    16570613   79.8
## 9 United States 301139947   78.2
## 10 New Zealand   4115771    NA
## # ... with 104 more rows

```

This join gives us 114 observations, which is the number of countries for which we have information on the population. Also note how it inserts NA's for the lifeExp variable.

```

left_join(countries_population, countries_lifeExp) %>%
  summary()

## Joining, by = "country"

##       country           pop  lifeExp
##       <fct>        <int>   <dbl>
## 1 Botswana      1639131   50.7
## 2 Greece        10706290   49.3
## 3 South Africa  43997828   43.5
## 4 Ethiopia       76511887   52.9
## 5 Zimbabwe      12311143   43.5
## 6 Yemen, Rep.   22211743   62.7
## 7 Nepal          28901790   50.7
## 8 Netherlands    16570613   79.8
## 9 United States 301139947   78.2
## 10 New Zealand   4115771    NA
## # ... with 104 more rows

```

```

## Algeria   : 1    Min.   :1.996e+05    Min.   :42.59
## Angola    : 1    1st Qu.:4.120e+06    1st Qu.:59.44
## Australia : 1    Median  :1.009e+07    Median :71.88
## Austria   : 1    Mean    :4.751e+07    Mean   :67.78
## Bahrain   : 1    3rd Qu.:2.885e+07    3rd Qu.:76.44
## Bangladesh: 1    Max.    :1.319e+09    Max.   :82.21
## (Other)    :108

```

#### 10.2.4 Right join

Right join means: the opposite of left join. I keep all information in my second (right) table.

```
right_join(countries_population, countries_lifeExp)

## Joining, by = "country"

## # A tibble: 85 x 3
##   country           pop  lifeExp
##   <fct>        <int>    <dbl>
## 1 United States    301139947    78.2
## 2 Argentina          NA     75.3
## 3 Korea, Dem. Rep.  23301725    67.3
## 4 Bulgaria         7322858    73.0
## 5 Chile             16284741    78.6
## 6 Croatia          4493312     75.7
## 7 Canada            33390141    80.7
## 8 Honduras           NA     70.2
## 9 Liberia            NA     45.7
## 10 Mexico           108700891    76.2
## # ... with 75 more rows
```

This join gives us 85 observations, which is the number of countries for which we have information on the life expectancy.

### 10.2.5 Full join

Full join means: I want to keep all information I have. So also populations for countries without life expectancy and vice versa remain in the dataset. All missing information is filled in as NA.

```
full_join(countries_population, countries_lifeExp)

## Joining, by = "country"

## # A tibble: 126 x 3
##   country           pop lifeExp
```

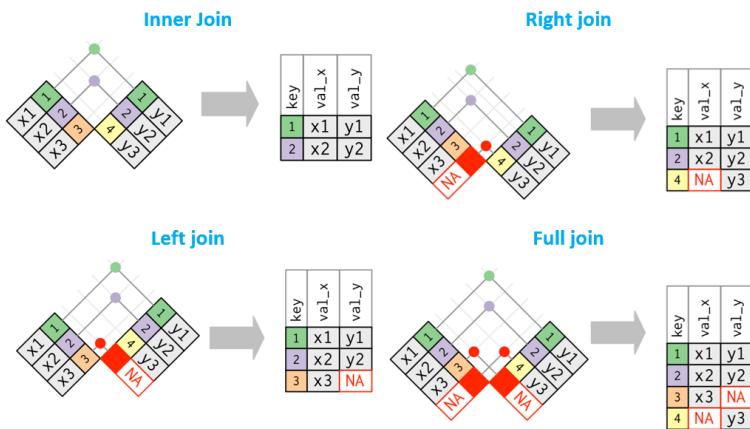
```

## <fct>          <int>    <dbl>
## 1 Botswana      1639131   50.7
## 2 Greece         10706290   NA
## 3 South Africa  43997828  49.3
## 4 Ethiopia       76511887  52.9
## 5 Zimbabwe       12311143  43.5
## 6 Yemen, Rep.   22211743  62.7
## 7 Nepal          28901790   NA
## 8 Netherlands    16570613  79.8
## 9 United States  301139947 78.2
## 10 New Zealand   4115771   NA
## # ... with 116 more rows

```

This join gives us 126 observations, which is the total number of countries for which we have at least one piece of information.

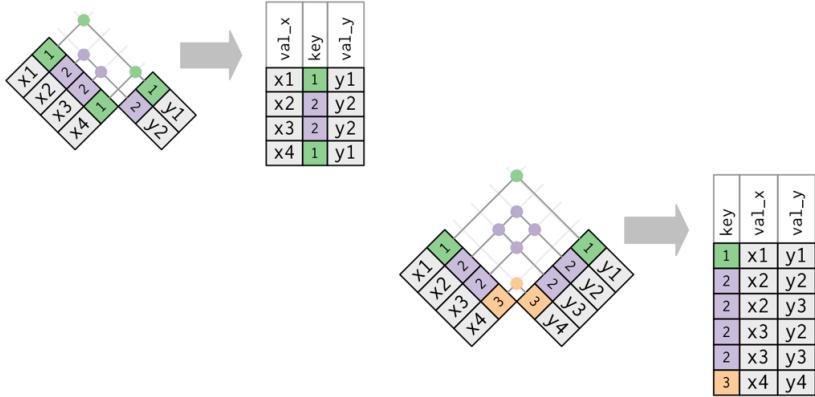
A schematical overview of the four types can be seen below. The coloured numbers represent the keys (countries in our example) while the x and y values represent the values (population and life expectancy in our example). Of course, there can be as many values as there are, it doesn't just have to be one. We will see other examples soon enough.



### 10.2.6 Duplicates

Sometimes one or both datasets contain duplicate keys: for example, we have information of the population in each country for more than a single year, so for each country we have more than one observation. In such cases, each observation will be joined multiple times, as in the figure below.<sup>2</sup>

<sup>2</sup> Of course, if we have both population data about multiple years and life expectancy data about multiple years, we should just include the *year* as a key variable. We don't want them to mix up. In that case, each observation is defined by both country and year.



### 10.2.7 An example

The package `nycflights13` contains different datasets about flights from NYC in 2013.

```
library(nycflights13)
```

One of the datasets is called `flights`

```
flights %>%
  glimpse

## # Observations: 336,776
## # Variables: 19
## # $ year              <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013...
## # $ month             <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
## # $ day               <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
## # $ dep_time           <int> 517, 533, 542, 544, 554, 555, 557, 557, 558, 55...
## # $ sched_dep_time    <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 60...
## # $ dep_delay          <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, ...
## # $ arr_time           <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 8...
## # $ sched_arr_time     <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 8...
## # $ arr_delay          <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, ...
## # $ carrier            <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6"...
## # $ flight              <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301...
## # $ tailnum             <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N...
## # $ origin              <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LG...
## # $ dest                <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IA...
## # $ air_time             <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149...
## # $ distance             <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 73...
## # $ hour                 <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6...
## # $ minute                <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 0, 59...
## # $ time_hour            <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-0...
```

Another one is airlines; with more information on the airlines, evidently.

```
airlines %>%
  glimpse

## # Observations: 16
## # Variables: 2
## $ carrier <chr> "9E", "AA", "AS", "B6", "DL", "EV", "F9", "FL", "HA", "MQ",...
## $ name      <chr> "Endeavor Air Inc.", "American Airlines Inc.", "Alaska Airl...
```

You can see they have the carrier variable in common, which contains a code for each airline. We can add the name of the airline to the flights

```
flights %>%
  inner_join(airlines)

## # Joining, by = "carrier"

## # A tibble: 336,776 x 20
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
## 1 2013     1     1      517            515       2     830          819
## 2 2013     1     1      533            529       4     850          830
## 3 2013     1     1      542            540       2     923          850
## 4 2013     1     1      544            545      -1    1004         1022
## 5 2013     1     1      554            600      -6     812          837
## 6 2013     1     1      554            558      -4     740          728
## 7 2013     1     1      555            600      -5     913          854
## 8 2013     1     1      557            600      -3     709          723
## 9 2013     1     1      557            600      -3     838          846
## 10 2013    1     1      558            600      -2     753          745
## # ... with 336,766 more rows, and 12 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
## #   name <chr>
```

Note that we did an inner join and our number of flights didn't decrease. This means that every carrier in flights is also available in airlines. In other words, for all carriers we have seen flights of, we know the name of the airline.

For a more advanced example, let's look at weather.

```
weather %>%
  glimpse
```

```

## Observations: 26,115
## Variables: 15
## $ origin      <chr> "EWR", "EWR", "EWR", "EWR", "EWR", "EWR", ...
## $ year        <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 20...
## $ month       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ day         <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ hour        <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 1...
## $ temp         <dbl> 39.02, 39.02, 39.02, 39.92, 39.02, 37.94, 39.02, 39.92, ...
## $ dewp         <dbl> 26.06, 26.96, 28.04, 28.04, 28.04, 28.04, 28.04, 28.04, ...
## $ humid        <dbl> 59.37, 61.63, 64.43, 62.21, 64.43, 67.21, 64.43, 62.21, ...
## $ wind_dir     <dbl> 270, 250, 240, 250, 260, 240, 240, 250, 260, 260, 260, 3...
## $ wind_speed   <dbl> 10.35702, 8.05546, 11.50780, 12.65858, 12.65858, 11.5078...
## $ wind_gust    <dbl> NA, ...
## $ precip       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
## $ pressure     <dbl> 1012.0, 1012.3, 1012.5, 1012.2, 1011.9, 1012.4, 1012.2, ...
## $ visib        <dbl> 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, ...
## $ time_hour    <dttm> 2013-01-01 01:00:00, 2013-01-01 02:00:00, 2013-01-01 03...

```

It contains information on place and time: the same we also have for flights, and it contains several variables about the weather (wind, temperature, precipitation, etc.)

Let's join the flights data with the weather.

```

flights %>%
  inner_join(airlines) %>%
  inner_join(weather) -> flights

## Joining, by = "carrier"

## Joining, by = c("year", "month", "day", "origin", "hour", "time_hour")

flights %>%
  glimpse

## Observations: 335,220
## Variables: 29
## $ year        <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013...
## $ month       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ day         <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ dep_time    <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 55...
## $ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 60...
## $ dep_delay   <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -2, -2, -2, -2, ...
## $ arr_time    <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 8...
## $ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 8...
## $ arr_delay   <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, ...
## $ carrier     <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6"...

```

```

## $ flight           <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301...
## $ tailnum         <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N...
## $ origin          <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LG...
## $ dest            <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IA...
## $ air_time        <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149...
## $ distance        <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 73...
## $ hour            <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6...
## $ minute          <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 59...
## $ time_hour       <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-0...
## $ name            <chr> "United Air Lines Inc.", "United Air Lines Inc.", "A...
## $ temp            <dbl> 39.02, 39.92, 39.02, 39.02, 39.92, 39.02, 37.94, 39...
## $ dewp            <dbl> 28.04, 24.98, 26.96, 26.96, 24.98, 28.04, 28.04, 24...
## $ humid            <dbl> 64.43, 54.81, 61.63, 61.63, 54.81, 64.43, 67.21, 54...
## $ wind_dir         <dbl> 260, 250, 260, 260, 260, 240, 260, 260, 260, 26...
## $ wind_speed       <dbl> 12.65858, 14.96014, 14.96014, 14.96014, 16.11092, 12...
## $ wind_gust        <dbl> NA, 21.86482, NA, NA, 23.01560, NA, NA, 23.01560, NA...
## $ precip           <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
## $ pressure         <dbl> 1011.9, 1011.4, 1012.1, 1012.1, 1011.7, 1011.9, 1012...
## $ visib            <dbl> 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, ...

```

Note that the second join used variables year, month, origin, hour and time\_hour to join the weather of the correct place and time to each flight.

#### 10.2.8 Binding data

The data we joined above were always different pieces of information which we somehow linked (same country, same, time, same place, same airline, etc.) Sometimes we have dataset on separate objects which are not linked, but contain the same information. Recall the datasets on African and European countries.

`population_africa`

```

## # A tibble: 52 x 2
##   country           pop
##   <fct>             <int>
## 1 Algeria          33333216
## 2 Angola           12420476
## 3 Benin            8078314
## 4 Botswana         1639131
## 5 Burkina Faso    14326203
## 6 Burundi          8390505
## 7 Cameroon         17696293
## 8 Central African Republic 4369038
## 9 Chad              10238807

```

```

## 10 Comoros                      710960
## # ... with 42 more rows

population_europe

## # A tibble: 30 x 2
##   country           pop
##   <fct>          <int>
## 1 Albania        3600523
## 2 Austria        8199783
## 3 Belgium       10392226
## 4 Bosnia and Herzegovina 4552198
## 5 Bulgaria      7322858
## 6 Croatia       4493312
## 7 Czech Republic 10228744
## 8 Denmark        5468120
## 9 Finland        5238460
## 10 France       61083916
## # ... with 20 more rows

```

These observations are not linked (there is no link between an African country and a European one), but they contain the same pieces of information (i.e. population).

We can **bind** these **rows** together.

```

bind_rows(population_africa, population_europe)

## # A tibble: 82 x 2
##   country           pop
##   <fct>          <int>
## 1 Algeria        33333216
## 2 Angola         12420476
## 3 Benin          8078314
## 4 Botswana       1639131
## 5 Burkina Faso  14326203
## 6 Burundi         8390505
## 7 Cameroon       17696293
## 8 Central African Republic 4369038
## 9 Chad            10238807
## 10 Comoros       710960
## # ... with 72 more rows

```

Note that we had 52 African countries and 30 European countries. Together, this makes for 82 countries.

For bind\_rows, it is not necessary to have exactly the same information. Suppose that we have life expectancy for African countries, but not for European. Consider the dataset **information\_africa**.

```
information_africa

## # A tibble: 52 x 3
##   country           pop  lifeExp
##   <fct>     <int>    <dbl>
## 1 Algeria      33333216  72.3
## 2 Angola       12420476  42.7
## 3 Benin        8078314  56.7
## 4 Botswana     1639131  50.7
## 5 Burkina Faso 14326203  52.3
## 6 Burundi      8390505  49.6
## 7 Cameroon     17696293  50.4
## 8 Central African Republic 4369038  44.7
## 9 Chad          10238807  50.7
## 10 Comoros      710960  65.2
## # ... with 42 more rows
```

And we bind these two datasets.

What we could have expected did indeed happen: the 30 European countries received an NA for life expectancy. However, be wary: if both datasets have different information, maybe bind\_rows is not what you are looking for, and maybe you need a join? Be sure that you understand how your datasets related to one another and how you should combine them.

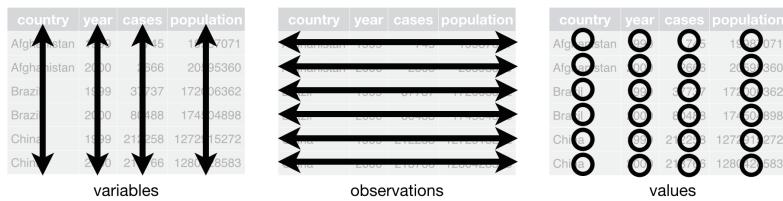
That said, one more remark on merging data. If there is a bind\_rows, there must surely be a bind\_cols for binding columns? Yes, there is. However, we will not use this function (hurray!). bind\_cols can do as it says: binding columns together just like bind\_rows binds rows together. However, binding columns together means that we have 2 sets of information about the same observations? That sounds a lot like it needs a join, doesn't it? Indeed! The main difference between bind\_rows and joins is that joins will combine rows that have the same key. However, bind\_rows will combine rows by position, i.e. the

first row of dataset A will be combined with first row of dataset B. It won't be looking at any keys. So if dataset A and B are in a different order, you have messed up your data. So, just forget about bind\_cols. Bind\_rows and joins should be able to get you where you want to be.

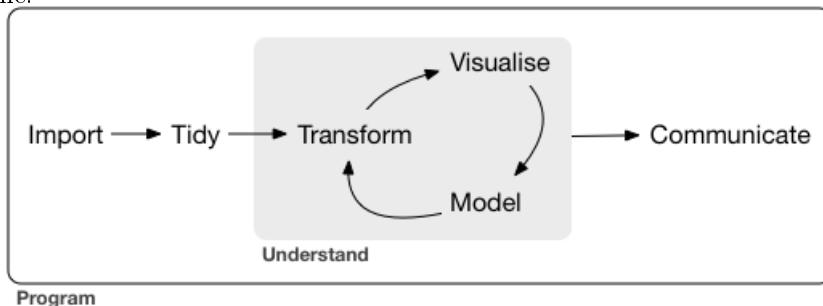
### 10.3 Transforming data

Next to merging data, we will also learn how to transform data. The difference? For merging we need two datasets, for transforming, we will only use a single one.

The main goal of transforming our data is to make sure it is *tidy*. This means: every row is an observation, and every column is a variable.



Now, tidying is primarily important in the initial phase of your project, as shown in the figure below. However, it can also be useful during analyses. For some graphs, it might happen that you need to transform your data - change what your observations are. This makes data transformation both essential and difficult. It is very important to understand what the current shape of your data is, and in which shape you need it to be for your analysis. This requires practice and time.



We will discuss four different transformations <sup>3</sup>.

There are 2 easy transformations:

1. Combine variables
2. Split variables

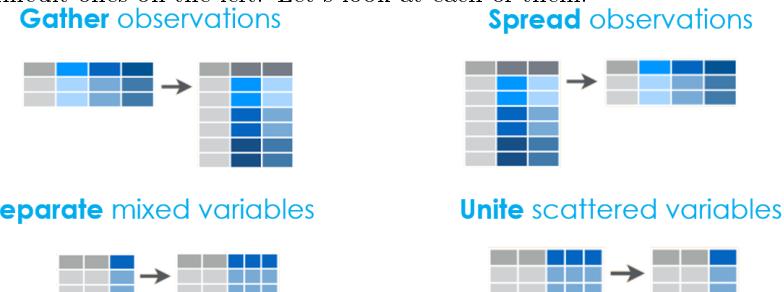
and 2 difficult ones

3. Spread a dataset

<sup>3</sup> Note that we used the term *transformation* for different things. We have used it before to transform *variables* (recode factors, rescale numerics, etc). At this moment we use it to transform *data*, which means that we are talking about multiple variable or complete datasets. The word choice is not to confuse you, we are actually doing the same thing, but at different levels.

#### 4. Gather a dataset

Below we show the schematically - the easy ones on the right, and the difficult ones on the left. Let's look at each of them.<sup>4</sup>



<sup>4</sup> Note that all the join and transformation functions discussed here are included on the cheatsheet of Data Manipulation. Make sure you can use it during exercises and exams!

##### 10.3.1 Unite variables

We use the function `unite` when we have several variables that we want to combine into a single one. The syntax for `unite` is as follows. Suppose we have information about students, with a `first_name` and `last_name`, and we want a single “name” variable.

`students`

```
## # A tibble: 10 x 2
##   first_name last_name
##   <chr>      <chr>
## 1 Kemb~     Raylin
## 2 Orean~    Elisha
## 3 Kirstyn~  Francico
## 4 Amparo~   Theoplis
## 5 Belen~    Ashea
## 6 Rayshaun~ Angela
## 7 Brazil~   Essie
## 8 Chaston~  Allyn
## 9 Reyn~     Tanita
## 10 Ogechi~  Sherriann

students %>%
  unite(col = name, first_name, last_name)

## # A tibble: 10 x 1
##   name
##   <chr>
## 1 Kemb~_Raylin
## 2 Orean~_Elisha
## 3 Kirstyn~_Francico
## 4 Amparo~_Theoplis
```

```
## 5 Belen_Ashea
## 6 Rayshaun_Angela
## 7 Brazil_Essie
## 8 Chaston_Allyn
## 9 Reyn_Tanita
## 10 Ogechi_Sherriann
```

We first specify the name for the new column (which here is just name), then we list all columns we want to unite. Note that by default, unite will put a `_` between the columns. We can change this with the argument `sep`.

```
students %>%
  unite(col = name, first_name, last_name, sep = " ")

## # A tibble: 10 x 1
##   name
##   <chr>
## 1 Kemba Raylin
## 2 Orean Elisha
## 3 Kirstyn Francico
## 4 Amparo Theoplis
## 5 Belen Ashea
## 6 Rayshaun Angela
## 7 Brazil Essie
## 8 Chaston Allyn
## 9 Reyn Tanita
## 10 Ogechi Sherriann
```

Sometimes we also prefer to keep the original variables. We can ask not to remove them as follows.

```
students %>%
  unite(col = name, first_name, last_name, sep = " ", remove = F)

## # A tibble: 10 x 3
##   name      first_name last_name
##   <chr>     <chr>     <chr>
## 1 Kemba Raylin    Kemba     Raylin
## 2 Orean Elisha    Orean     Elisha
## 3 Kirstyn Francico Kirstyn   Francico
## 4 Amparo Theoplis Amparo    Theoplis
## 5 Belen Ashea     Belen     Ashea
## 6 Rayshaun Angela Rayshaun  Angela
## 7 Brazil Essie    Brazil    Essie
## 8 Chaston Allyn   Chaston   Allyn
## 9 Reyn Tanita     Reyn     Tanita
## 10 Ogechi Sherriann Ogechi   Sherriann
```

### 10.3.2 Separate variables

Separate works the other way around: it separates a single variable into multiple ones. Suppose we have a list of students (students2) with their full names, and we want to separate them.<sup>5</sup>

```
students_2
```

```
## # A tibble: 10 x 1
##   name
##   <chr>
## 1 Vashawn Heathr
## 2 Hans Musab
## 3 Shihab Mahogany
## 4 Daden Braun
## 5 Shiloh Billi
## 6 Hashir Magdalena
## 7 Latangela Lois
## 8 Lydon Aliha
## 9 Garcelle Ziah
## 10 Jaleal Nancy
```

We can use separate in a similar way. First tell which column you want separated. Then tell them into which columns you want to put the pieces.<sup>6</sup>

```
students_2 %>%
  separate(col = name, into = c("first_name", "last_name"))

## # A tibble: 10 x 2
##   first_name last_name
##   <chr>      <chr>
## 1 Vashawn    Heathr
## 2 Hans        Musab
## 3 Shihab     Mahogany
## 4 Daden       Braun
## 5 Shiloh      Billi
## 6 Hashir      Magdalena
## 7 Latangela   Lois
## 8 Lydon       Aliha
## 9 Garcelle    Ziah
## 10 Jaleal     Nancy
```

Default, separate will split the columns on any character which is not alphanumerical: anything except numbers and letters. So, he correctly used spaces, which with we are perfectly happy. If you want to changes this, you can again set the sep argument. For example,

<sup>5</sup> Note how you spell separate. An e, followed by an a, another a, and another e. Can you remember that? Congratulations, you have just avoided a series of very common mistakes!

<sup>6</sup> Note that the col argument in unite is the new column, the col argument in separate is the existing column! Also note that the new columns created by separate should be given as a character vector, not as a list of unquoted names like we did in unite.

when there is a combined surname like Janssen-Swilden (let's say such a ridiculous name actually exists), it would be split on the - sign. We don't want that, so we should tell separate to split only on spaces, i.e. sep = " ".

Separate will create exactly as many columns as the number of names you provide in into. If he finds more or less pieces than that number for any observation, he will warn you about this. If there are less, NA will appear, if there are more, the last ones will be discarded. Also, you can use remove = F to keep the original variables.

Now let's get ready for those difficult ones!

### 10.3.3 Spread data

We can use spread to take a pair of variables - a *key* and a *value* - and spread them over different columns: one for each *key* with the corresponding *value* in it.



If at this moment you hear it thundering in Keulen, it might be time for you to revise earlier tutorials. Because we have actually already seen spread before (Did we?) (Yes we did.)

The following example might refresh things a bit.

```
library(ggplot2)
diamonds %>%
  count(color, clarity)

## # A tibble: 56 x 3
##   color clarity     n
##   <ord> <ord> <int>
## 1 D      I1        42
## 2 D      SI2       1370
## 3 D      SI1       2083
## 4 D      VS2       1697
## 5 D      VS1        705
## 6 D      VVS2       553
## 7 D      VVS1       252
## 8 D      IF         73
```

```

##   9 E      I1       102
## 10 E     SI2      1713
## # ... with 46 more rows

diamonds %>%
  count(color, clarity) %>%
  spread(clarity, n)

## # A tibble: 7 x 9
##   color    I1    SI2    SI1    VS2    VS1    VVS2    VVS1    IF
##   <ord> <int> <int> <int> <int> <int> <int> <int>
## 1 D        42   1370   2083   1697    705    553    252    73
## 2 E       102   1713   2426   2470   1281    991    656   158
## 3 F       143   1609   2131   2201   1364    975    734   385
## 4 G       150   1548   1976   2347   2148   1443    999   681
## 5 H       162   1563   2275   1643   1169    608    585   299
## 6 I        92   912    1424   1169    962    365    355   143
## 7 J        50   479    750    731    542    131     74    51

```

When we *spread* data, we go from a *long* dataset to a *wide* dataset. Just look back at the example and the schematic figure. Make sure to remember this.

#### 10.3.4 Gather data

If we already knew *spread*, *gather* is a piece of cake. It does the opposite of *spread*. How straightforward! So, with *gather* we go from a *wide* dataset to a *long* dataset, by *gathering* several observations into a single one.

Just look at this figure.



Let's look at an example.

The dataset below shows the population for every country on earth after each 5 year interval, starting in 1952, ending in 2007.

```
yearly_population
```

```
## # A tibble: 142 x 14
```

```

##   country continent `1952` `1957` `1962` `1967` `1972` `1977` `1982` `1987`
##   <fct>    <fct>     <int>   <int>   <int>   <int>   <int>   <int>
## 1 Afghan~ Asia      8.43e6 9.24e6 1.03e7 1.15e7 1.31e7 1.49e7 1.29e7 1.39e7
## 2 Albania Europe    1.28e6 1.48e6 1.73e6 1.98e6 2.26e6 2.51e6 2.78e6 3.08e6
## 3 Algeria Africa    9.28e6 1.03e7 1.10e7 1.28e7 1.48e7 1.72e7 2.00e7 2.33e7
## 4 Angola Africa     4.23e6 4.56e6 4.83e6 5.25e6 5.89e6 6.16e6 7.02e6 7.87e6
## 5 Argent~ Americas  1.79e7 1.96e7 2.13e7 2.29e7 2.48e7 2.70e7 2.93e7 3.16e7
## 6 Austra~ Oceania   8.69e6 9.71e6 1.08e7 1.19e7 1.32e7 1.41e7 1.52e7 1.63e7
## 7 Austria Europe    6.93e6 6.97e6 7.13e6 7.38e6 7.54e6 7.57e6 7.57e6 7.58e6
## 8 Bahrain Asia     1.20e5 1.39e5 1.72e5 2.02e5 2.31e5 2.97e5 3.78e5 4.55e5
## 9 Bangla~ Asia     4.69e7 5.14e7 5.68e7 6.28e7 7.08e7 8.04e7 9.31e7 1.04e8
## 10 Belgium Europe   8.73e6 8.99e6 9.22e6 9.56e6 9.71e6 9.82e6 9.86e6 9.87e6
## # ... with 132 more rows, and 4 more variables: `1992` <int>, `1997` <int>,
## #   `2002` <int>, `2007` <int>

```

Pretty well-arranged table, isn't it? Let's make a line plot of the evolution. We would need time (years) on the x-axis and population on the y-axis. But...? Well, f\*ck me! Those variables don't exist?! How can I make my line plot?

Let's *gather* the data into those two variables.

- The key argument is the **new** variable in which we want old variable **names** to go. In our case, we want all the years as a *time* variable, so we can use them, instead of being scattered over 12 variables.
- The value argument is the **new** variable in which the **values** of the old variables go. Thus, these would be the population numbers.
- After that, we specify all the columns we want to gather. In our case all years. So, we can just say that we don't want to gather country and continent instead.<sup>7</sup>

Let's see what happens.

```

yearly_population %>%
  gather(key = time, value = population, -country, -continent)
## # A tibble: 1,704 x 4
##   country   continent time  population
##   <fct>     <fct>    <chr>   <int>
## 1 Afghanistan Asia     1952     8425333
## 2 Albania     Europe   1952     1282697
## 3 Algeria     Africa   1952     9279525
## 4 Angola      Africa   1952     4232095
## 5 Argentina   Americas 1952     17876956
## 6 Australia   Oceania  1952     8691212
## 7 Austria     Europe   1952     6927772

```

<sup>7</sup> Actually, there is a more important reason we want to use -country and -continent instead of listing all years, apart from being lazy. Remember that all object and variable names in R need to start with a letter, not a number? Well, the year columns clearly don't. Selecting them would need a special technique. Just saying 1952:2007 would unfortunately not work. But, luckily, that's a story for another time.

```

## 8 Bahrain      Asia      1952      120447
## 9 Bangladesh   Asia      1952      46886859
## 10 Belgium     Europe    1952      8730405
## # ... with 1,694 more rows

```

Well, exactly the opposite of spread, isn't it? A bunch of old variables (1952, 1957, 1962, etc.) are *gathered* into a single new variable time. While the contents of those old variables are placed next to them in the population variable.

Note how we went from a dataset with 13 columns and 142 rows (= WIDE) to a dataset with only 3 columns but 1704 rows (= LONG).

So, let's wrap this up.

- For gather, key and value are *new* columnnames. You can choose them as you like (just like I chose time and population)
- For spread, key and value are *existing* columns. The ones you want to spread out.
- With gather, you provide a list of *existing* columns which you want to gather/combine. You can also say which you don't want using -. In fact, you can use all the select-tricks here. If you don't tell it anything except for key and value, all columns will be gathered.
- With gather, only key and value are necessary arguments.

Easy, isn't it?

Unfortunately, no. It isn't.

Spread and gather are probably the least intuitive functions you will learn in this course. Try to read this section several times, and look very good at the examples. Try to see what's happening. Things can get very complicated with spread and gather, as they change the structure of your data entirely. Combining them with joins only increases the difficulty. So, don't go easy on this. Spend some time in trying to understand the functions, and learn how to use the cheatsheet. The functions are not easy at all, but you will need them sooner than you think. Let's see them at work in other example. We will use some real-life data of the World Health Organisation WHO!

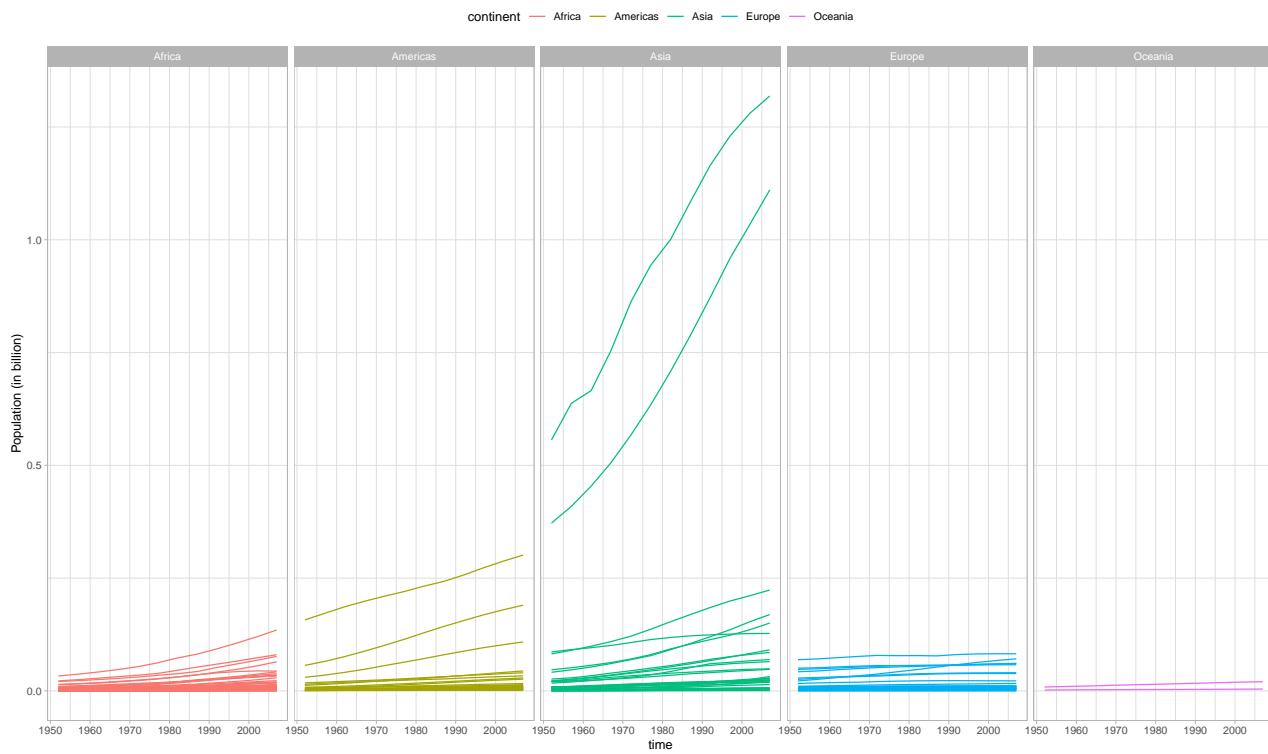
Oh, I almost forgot! We would make a line plot of the population data. Well, you see, once we have gather, it gets easy. We can almost directly go to ggplot.

```

yearly_population %>%
  gather(key = time, value = population, -country, -continent) %>%
  mutate(time = as.numeric(time)) %>%
  ggplot(aes(time, population/(10^9), group = country, color = continent)) +
  geom_line() +
  facet_grid(.~continent) +
  theme_light() +

```

```
  labs(y = "Population (in billion)") +
  theme(legend.position = "top")
```



Can you tell which countries are the two soaring lines in Asia?  
(Please tell me you can.)

So, let's study some health!

#### 10.4 [Case study]: WHO

We gathered (pun intended) data about the number of (new) Tuberculosis cases broken down by

- year
- country
- age (7 groups)
- gender
- type of TB
  - new/old -> (all new in our case)
  - diagnosis method
    - \* rel: relapse
    - \* sp: smear positive
    - \* sn: smear negative

\* ep: extrapulmonary

(No need to know the different diagnosis methods.)

The data looks as follows.

```
who %>%
  glimpse()

## # Observations: 7,240
## # Variables: 60
## $ country      <chr> "Afghanistan", "Afghanistan", "Afghanistan", "Afghanis...
## $ iso2         <chr> "AF", "AF", "AF", "AF", "AF", "AF", "AF", "AF", ...
## $ iso3         <chr> "AFG", "AFG", "AFG", "AFG", "AFG", "AFG", "AFG", "AFG"...
## $ year        <int> 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, ...
## $ new_sp_m014 <int> NA, NA...
## $ new_sp_m1524 <int> NA, NA...
## $ new_sp_m2534 <int> NA, NA...
## $ new_sp_m3544 <int> NA, NA...
## $ new_sp_m4554 <int> NA, NA...
## $ new_sp_m5564 <int> NA, NA...
## $ new_sp_m65   <int> NA, NA...
## $ new_sp_f014  <int> NA, NA...
## $ new_sp_f1524 <int> NA, NA...
## $ new_sp_f2534 <int> NA, NA...
## $ new_sp_f3544 <int> NA, NA...
## $ new_sp_f4554 <int> NA, NA...
## $ new_sp_f5564 <int> NA, NA...
## $ new_sp_f65   <int> NA, NA...
## $ new_sn_m014  <int> NA, NA...
## $ new_sn_m1524 <int> NA, NA...
## $ new_sn_m2534 <int> NA, NA...
## $ new_sn_m3544 <int> NA, NA...
## $ new_sn_m4554 <int> NA, NA...
## $ new_sn_m5564 <int> NA, NA...
## $ new_sn_m65   <int> NA, NA...
## $ new_sn_f014  <int> NA, NA...
## $ new_sn_f1524 <int> NA, NA...
## $ new_sn_f2534 <int> NA, NA...
## $ new_sn_f3544 <int> NA, NA...
## $ new_sn_f4554 <int> NA, NA...
## $ new_sn_f5564 <int> NA, NA...
## $ new_sn_f65   <int> NA, NA...
## $ new_ep_m014  <int> NA, NA...
## $ new_ep_m1524 <int> NA, NA...
## $ new_ep_m2534 <int> NA, NA...
```

```
## $ new_ep_m3544 <int> NA, NA...
## $ new_ep_m4554 <int> NA, NA...
## $ new_ep_m5564 <int> NA, NA...
## $ new_ep_m65   <int> NA, NA...
## $ new_ep_f014  <int> NA, NA...
## $ new_ep_f1524 <int> NA, NA...
## $ new_ep_f2534 <int> NA, NA...
## $ new_ep_f3544 <int> NA, NA...
## $ new_ep_f4554 <int> NA, NA...
## $ new_ep_f5564 <int> NA, NA...
## $ new_ep_f65   <int> NA, NA...
## $ newrel_m014 <int> NA, NA...
## $ newrel_m1524 <int> NA, NA...
## $ newrel_m2534 <int> NA, NA...
## $ newrel_m3544 <int> NA, NA...
## $ newrel_m4554 <int> NA, NA...
## $ newrel_m5564 <int> NA, NA...
## $ newrel_m65   <int> NA, NA...
## $ newrel_f014  <int> NA, NA...
## $ newrel_f1524 <int> NA, NA...
## $ newrel_f2534 <int> NA, NA...
## $ newrel_f3544 <int> NA, NA...
## $ newrel_f4554 <int> NA, NA...
## $ newrel_f5564 <int> NA, NA...
## $ newrel_f65   <int> NA, NA...
```

To be honest: quite a mess. We don't really need 60 variables for the data we just described, do we? What's going on?

It seems that for each country and each year, the data contains one row. Let's verify.

```
who %>%
  count(country, year)

## # A tibble: 7,240 x 3
##   country     year     n
##   <chr>      <int> <int>
## 1 Afghanistan 1980     1
## 2 Afghanistan 1981     1
## 3 Afghanistan 1982     1
## 4 Afghanistan 1983     1
## 5 Afghanistan 1984     1
## 6 Afghanistan 1985     1
## 7 Afghanistan 1986     1
## 8 Afghanistan 1987     1
```

```
##  9 Afghanistan 1988      1
## 10 Afghanistan 1989      1
## # ... with 7,230 more rows
```

We see mostly ones. Let's check for sure.

```
who %>%
  count(country, year) %>%
  filter(n > 1)

## # A tibble: 0 x 3
## # ... with 3 variables: country <chr>, year <int>, n <int>
```

Ok. So, each year, each country, one row. We have 7240 rows because we have

```
who %>% count(year) %>% nrow

## [1] 34
```

34 years, and

```
who %>% count(country) %>% nrow()

## [1] 219
```

219 countries.

Thus we expect this many rows:

```
219*34

## [1] 7446
```

It seems we are missing 206 rows. I.e. there are countries for which we don't have all years, or vice versa. It is not really important here, but these are the kind of things a good data analyst checks.

Let's go back to our problem.

Of the 60 variables, the first 3 all depict country (Remember that I told you that there are different ways to abbreviate a country), and the 4th contains the year. So, there remain 56 variables.

Well, we have information on 7 age groups, 2 genders, and 4 diagnosis methods. 7 times 2 times 4 equals 56. Aha! All the different cases are putted in a different variable. That's not really easy to work with.

Why not, I heard you think?

Let's try to solve the following questions.

- How many women older, 25 or older in Belgium were diagnosed with TB in 2000? How many of those had a relaps?
- What is the total number of TB cases in Belgium in each year?

- Can you graphically show the evolution of the number of cases for different genders and age groups?

No, you can't. At least, not without a lot of work, or without tidy-ing our data. So, let's start.

It is often helpful to think about the format we would like our data to be in, without getting lost in transformation. Ideally, we would like to have the following variables:

- country
- year
- is\_new
- diagnosis
- gender
- age
- cases (the number of TB cases)

First of all, let's go to a dataset in a long format, by gathering all the different types of diagnosis and cases into a long list. We will not gather the first 4 columns. The old columns will be a variable "type", and the numbers will be called "cases".

```
who %>%
  gather(key = type, value = cases, -country:-year)

## # A tibble: 405,440 x 6
##   country     iso2   iso3   year type      cases
##   <chr>       <chr>  <chr>  <int> <chr>      <int>
## 1 Afghanistan AF    AFG    1980 new_sp_m014     NA
## 2 Afghanistan AF    AFG    1981 new_sp_m014     NA
## 3 Afghanistan AF    AFG    1982 new_sp_m014     NA
## 4 Afghanistan AF    AFG    1983 new_sp_m014     NA
## 5 Afghanistan AF    AFG    1984 new_sp_m014     NA
## 6 Afghanistan AF    AFG    1985 new_sp_m014     NA
## 7 Afghanistan AF    AFG    1986 new_sp_m014     NA
## 8 Afghanistan AF    AFG    1987 new_sp_m014     NA
## 9 Afghanistan AF    AFG    1988 new_sp_m014     NA
## 10 Afghanistan AF   AFG    1989 new_sp_m014     NA
## # ... with 405,430 more rows
```

See what happened? Take a good look.

Had you figured out that we first needed to gather the data? If yes, congratulations, you start to get what data transformation is and which transformations you need where. If no, don't worry. Remember that I told you this is a hard skill. Furthermore, there are probably different ways to do this.

We can get rid of iso2 and iso3. Note that they might be useful for joining the data with other data about countries, but we have no plans to do so. Just, let's get them out of our way.

```
who %>%
  gather(key = type, value = cases, -country:-year) %>%
  select(-iso2, -iso3)

## # A tibble: 405,440 x 4
##   country     year type      cases
##   <chr>       <int> <chr>    <int>
## 1 Afghanistan 1980 new_sp_m014     NA
## 2 Afghanistan 1981 new_sp_m014     NA
## 3 Afghanistan 1982 new_sp_m014     NA
## 4 Afghanistan 1983 new_sp_m014     NA
## 5 Afghanistan 1984 new_sp_m014     NA
## 6 Afghanistan 1985 new_sp_m014     NA
## 7 Afghanistan 1986 new_sp_m014     NA
## 8 Afghanistan 1987 new_sp_m014     NA
## 9 Afghanistan 1988 new_sp_m014     NA
## 10 Afghanistan 1989 new_sp_m014    NA
## # ... with 405,430 more rows
```

Now, there is a lot of information in the type variable. Actually, there are more variables in this single variable. let's separate them. (See how that thought process goes?)

First, let's look at the different levels by doing a quick count.

```
who %>%
  gather(key = type, value = cases, -country:-year) %>%
  select(-iso2, -iso3) %>%
  count(type) %>%
  print(n = Inf) # I want to see all of them

## # A tibble: 56 x 2
##   type        n
##   <chr>    <int>
## 1 new_ep_f014  7240
## 2 new_ep_f1524  7240
## 3 new_ep_f2534  7240
## 4 new_ep_f3544  7240
## 5 new_ep_f4554  7240
## 6 new_ep_f5564  7240
## 7 new_ep_f65    7240
## 8 new_ep_m014   7240
## 9 new_ep_m1524  7240
```

```
## 10 new_ep_m2534 7240
## 11 new_ep_m3544 7240
## 12 new_ep_m4554 7240
## 13 new_ep_m5564 7240
## 14 new_ep_m65 7240
## 15 new_sn_f014 7240
## 16 new_sn_f1524 7240
## 17 new_sn_f2534 7240
## 18 new_sn_f3544 7240
## 19 new_sn_f4554 7240
## 20 new_sn_f5564 7240
## 21 new_sn_f65 7240
## 22 new_sn_m014 7240
## 23 new_sn_m1524 7240
## 24 new_sn_m2534 7240
## 25 new_sn_m3544 7240
## 26 new_sn_m4554 7240
## 27 new_sn_m5564 7240
## 28 new_sn_m65 7240
## 29 new_sp_f014 7240
## 30 new_sp_f1524 7240
## 31 new_sp_f2534 7240
## 32 new_sp_f3544 7240
## 33 new_sp_f4554 7240
## 34 new_sp_f5564 7240
## 35 new_sp_f65 7240
## 36 new_sp_m014 7240
## 37 new_sp_m1524 7240
## 38 new_sp_m2534 7240
## 39 new_sp_m3544 7240
## 40 new_sp_m4554 7240
## 41 new_sp_m5564 7240
## 42 new_sp_m65 7240
## 43 newrel_f014 7240
## 44 newrel_f1524 7240
## 45 newrel_f2534 7240
## 46 newrel_f3544 7240
## 47 newrel_f4554 7240
## 48 newrel_f5564 7240
## 49 newrel_f65 7240
## 50 newrel_m014 7240
## 51 newrel_m1524 7240
## 52 newrel_m2534 7240
## 53 newrel_m3544 7240
```

```
## 54 newrel_m4554 7240
## 55 newrel_m5564 7240
## 56 newrel_m65    7240
```

Oh crap. The first 42 levels are nicely separated by 2 underscores. But the last are not. It's all "newrel" instead of "new\_rel". Separate will not be able to split that...

So, let's pull together a neat trick. We are going to replace all the little "newrel" part with "new\_rel". How? Using the stringr package for string manipulation. It has a useful function `str_replace`. Here we go.

```
who %>%
  gather(key = type, value = cases, -country:-year) %>%
  select(-iso2, -iso3) %>%
  mutate(type = str_replace(type, "newrel", "new_rel")) %>%
  count(type) %>%
  print(n = Inf)

## # A tibble: 56 x 2
##   type             n
##   <chr>        <int>
## 1 new_ep_f014    7240
## 2 new_ep_f1524   7240
## 3 new_ep_f2534   7240
## 4 new_ep_f3544   7240
## 5 new_ep_f4554   7240
## 6 new_ep_f5564   7240
## 7 new_ep_f65     7240
## 8 new_ep_m014    7240
## 9 new_ep_m1524   7240
## 10 new_ep_m2534  7240
## 11 new_ep_m3544  7240
## 12 new_ep_m4554  7240
## 13 new_ep_m5564  7240
## 14 new_ep_m65    7240
## 15 new_rel_f014  7240
## 16 new_rel_f1524 7240
## 17 new_rel_f2534 7240
## 18 new_rel_f3544 7240
## 19 new_rel_f4554 7240
## 20 new_rel_f5564 7240
## 21 new_rel_f65    7240
## 22 new_rel_m014   7240
## 23 new_rel_m1524  7240
```

```

## 24 new_rel_m2534 7240
## 25 new_rel_m3544 7240
## 26 new_rel_m4554 7240
## 27 new_rel_m5564 7240
## 28 new_rel_m65    7240
## 29 new_sn_f014   7240
## 30 new_sn_f1524 7240
## 31 new_sn_f2534 7240
## 32 new_sn_f3544 7240
## 33 new_sn_f4554 7240
## 34 new_sn_f5564 7240
## 35 new_sn_f65    7240
## 36 new_sn_m014   7240
## 37 new_sn_m1524 7240
## 38 new_sn_m2534 7240
## 39 new_sn_m3544 7240
## 40 new_sn_m4554 7240
## 41 new_sn_m5564 7240
## 42 new_sn_m65    7240
## 43 new_sp_f014   7240
## 44 new_sp_f1524 7240
## 45 new_sp_f2534 7240
## 46 new_sp_f3544 7240
## 47 new_sp_f4554 7240
## 48 new_sp_f5564 7240
## 49 new_sp_f65    7240
## 50 new_sp_m014   7240
## 51 new_sp_m1524 7240
## 52 new_sp_m2534 7240
## 53 new_sp_m3544 7240
## 54 new_sp_m4554 7240
## 55 new_sp_m5564 7240
## 56 new_sp_m65    7240

```

That's better, isn't it? By the way, do you see how we at each point build on what we did before? This way we can easily change mistakes if we make some. Only when our data is correctly transformed, we save it, and put the code in our loadscript.

But now, we can separate the data. The first part will become the `is_new` variable, the second part the diagnosis variable, and the last part... well, it contains both the gender (f/m) and the age category. Let's just call it `age_gender`, and tackle that problem later.

```

who %>%
  gather(key = type, value = cases, -country:-year) %>%

```

```

select(-iso2, -iso3) %>%
mutate(type = str_replace(type, "newrel", "new_rel")) %>%
separate(type, into = c("is_new", "diagnosis", "gender_age"))

## # A tibble: 405,440 x 6
##   country      year is_new diagnosis gender_age cases
##   <chr>       <int> <chr>    <chr>     <chr>      <int>
## 1 Afghanistan 1980 new     sp        m014       NA
## 2 Afghanistan 1981 new     sp        m014       NA
## 3 Afghanistan 1982 new     sp        m014       NA
## 4 Afghanistan 1983 new     sp        m014       NA
## 5 Afghanistan 1984 new     sp        m014       NA
## 6 Afghanistan 1985 new     sp        m014       NA
## 7 Afghanistan 1986 new     sp        m014       NA
## 8 Afghanistan 1987 new     sp        m014       NA
## 9 Afghanistan 1988 new     sp        m014       NA
## 10 Afghanistan 1989 new    sp        m014       NA
## # ... with 405,430 more rows

```

Cool, that worked! We didn't even need to tell separate how to split. He decided this automagically. What a smart boy!

Now, let's split age\_gender. But on what? There is no character to split on. However, separate is so smart, we can tell him to split after the *first* character - 'cause that one is the gender, the remainder is the age. We could actually do this for any character. We just need to set sep = n, where n is our number. In this case 1. Let's try!

```

who %>%
gather(key = type, value = cases, -country:-year) %>%
select(-iso2, -iso3) %>%
mutate(type = str_replace(type, "newrel", "new_rel")) %>%
separate(type, into = c("is_new", "diagnosis", "gender_age")) %>%
separate(gender_age, into = c("gender", "age"), sep = 1)

## # A tibble: 405,440 x 7
##   country      year is_new diagnosis gender age   cases
##   <chr>       <int> <chr>    <chr>     <chr> <chr> <int>
## 1 Afghanistan 1980 new     sp        m     014       NA
## 2 Afghanistan 1981 new     sp        m     014       NA
## 3 Afghanistan 1982 new     sp        m     014       NA
## 4 Afghanistan 1983 new     sp        m     014       NA
## 5 Afghanistan 1984 new     sp        m     014       NA
## 6 Afghanistan 1985 new     sp        m     014       NA
## 7 Afghanistan 1986 new     sp        m     014       NA
## 8 Afghanistan 1987 new     sp        m     014       NA
## 9 Afghanistan 1988 new     sp        m     014       NA

```

```
## 10 Afghanistan 1989 new     sp      m    014      NA
## # ... with 405,430 more rows
```

I don't know about you, but I think this is exactly how we wanted the data to be! Let's save it now.

```
who %>%
  gather(key = type, value = cases, -country:-year) %>%
  select(-iso2, -iso3) %>%
  mutate(type = str_replace(type, "newrel", "new_rel")) %>%
  separate(type, into = c("is_new", "diagnosis", "gender_age")) %>%
  separate(gender_age, into = c("gender", "age"), sep = 1) -> tidy_who
```

And just for fun, let us solve the questions we had earlier.

- How many women older, 25 or older in Belgium were diagnosed with TB in 2000? How many of those had a relaps?

```
tidy_who %>%
  filter(gender == "f", !(age %in% c("014", "1524")), country == "Belgium", year == 2000) %>%
  group_by(diagnosis) %>%
  summarize(n_cases = sum(cases, na.rm = T))

## # A tibble: 4 x 2
##   diagnosis n_cases
##   <chr>      <int>
## 1 ep          0
## 2 rel         0
## 3 sn          0
## 4 sp          78
```

According to this data, there were 78 cases, and none of them were relapses.

- What is the total number of TB cases in Belgium in each year?

```
tidy_who %>%
  filter(country == "Belgium") %>%
  group_by(year) %>%
  summarize(n_cases = sum(cases, na.rm = T))

## # A tibble: 34 x 2
##   year n_cases
##   <int>   <int>
## 1 1980     0
## 2 1981     0
## 3 1982     0
```

```

## 4 1983      0
## 5 1984      0
## 6 1985      0
## 7 1986      0
## 8 1987      0
## 9 1988      0
## 10 1989     0
## # ... with 24 more rows

```

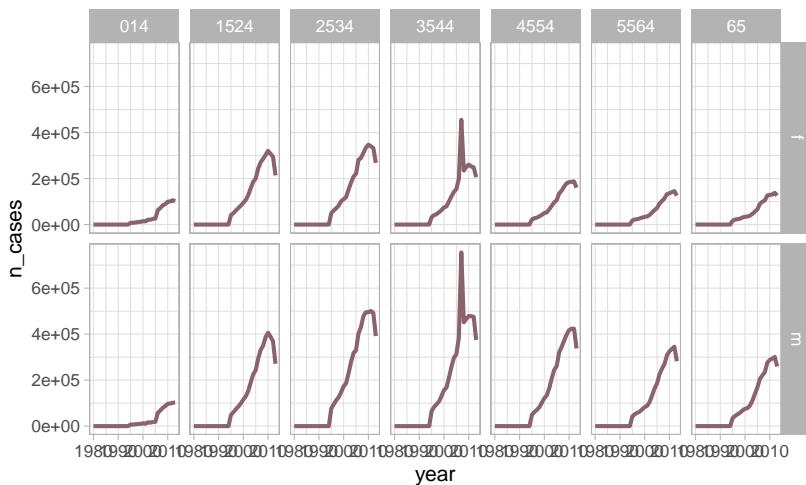
(It seems there were no cases of TB in Belgium before 1995. Or we are just missing data? That's the thing na.rm can do. You must be careful.)

- Can you graphically show the evolution of the number of cases for different genders and age groups?

```

tidy_who %>%
  group_by(year, age, gender) %>%
  summarize(n_cases = sum(cases, na.rm = T)) %>%
  ggplot(aes(year, n_cases)) +
  geom_line(color = "pink4", lwd = 1) +
  facet_grid(gender~age) +
  theme_light()

```





# 11

## *Tijdsdata*

### *11.1 Inleiding*

#### *11.1.1 Tijdstippen versus periodes*

- We kunnen tijdgerelateerde data in twee categorieën onderverdelen:  
tijdstippen en periodes.
- Tijdstip.
  - Verwijst naar een specifiek moment in de tijd.
  - 3 varianten:
    - \* datum (“01-01-2017”) verwijst naar een specifieke dag.
    - \* datum-tijdstip (“01-01-2017 13:54”) verwijst naar een specifiek moment op een specifieke dag.
    - \* tijdstip (“13:54”) verwijst naar een specifiek moment op een ongedefinieerde dag.
- Periode.
  - Verwijst naar een periode en wordt typisch uitgedrukt aan de hand van de duur van de periode.
    - \* Bijvoorbeeld: Een periode van “3605 seonden” of een periode van “2 maanden en 1 dag”.
  - Soms wordt een periode specifiek gedefinieerd aan de hand van twee specifieke tijdstippen die het begin en het einde van de periode aangeven.
    - \* Bijvoorbeeld: De periode van 01-01-2017 tot 03-01-2017.
- **Bestudeer hoofdstuk 16 van het boek ‘R for Data Science’ van Grolemund en Wickham !**

#### *11.1.2 Afronden van tijdstippen*

- Ieder tijdstip heeft een zekere nauwkeurigheid. Sommige tijdstippen zijn tot op de seconde gedefinieerd terwijl andere slechts een nauwkeurigheid hebben van weken of maanden.

- Soms kan het voor visualisaties of analyses zinvol zijn om tijdstippen minder nauwkeurig te maken en deze af te ronden.

## 11.2 Periode-data

- We kunnen 3 soorten van periodes onderscheiden, waarbij het eerste type (interval) naar een specifieke periode tussen 2 tijdstippen verwijst en de 2 andere types (*duration* en *period*) naar een periode van een specifieke duur verwijzen maar telkens onafhankelijk van het specifieke tijdstip.

### 11.2.1 Interval

- Een interval is een periode die bepaald wordt door twee specifieke tijdstippen.
- Intervals worden weinig gebruikt om rechtstreeks te analyseren, maar kunnen als tussenstap gebruikt worden om de duurtijd van specifieke periodes te bepalen.

### 11.2.2 Duration

- Duration is de duur van een periode uitgedrukt als het exact aantal seconden die feitelijk verstrekken zijn tussen twee tijdstippen.
- Tussen ‘26 maart 2017 02:00:00’ en ‘26 maart 2017 03:00:01’ is slechts 1 seconde feitelijk verstrekken omdat we van 2u naar 3u zijn overgeschakeld op het zomertijd.
- Durations gebruik je voornamelijk als je de werkelijke tijd tussen twee tijdstippen wenst te berekenen of wanneer je een aantal seconden wenst toe te voegen bij of af te trekken van een specifiek tijdstip.

### 11.2.3 Period

- De tijd die verstrekken ‘lijkt’ te zijn (op een klok) tussen twee tijdstippen.
- Dus tussen ‘26 maart 2017 02:00:00’ en ‘26 maart 2017 03:00:01’ zit een period van 1 uur en 1 seconde.
- Periods gebruik je voornamelijk als je periodes wilt toevoegen aan tijdstippen zonder rekening te moeten houden met onverwachte sprongen in de tijd (zomertijd/wintertijd, schrikkeljaren, . . . ).
  - Dus als je bij ieder tijdstip 1 dag (24u) wenst toe te voegen, kan je beter een period gebruiken dan een duration, omdat je anders rekening moet houden met de dag waarop we van zomer- naar winteruur gaan en omgekeerd.

### 11.3 Analyseren van tijdgerelateerde data

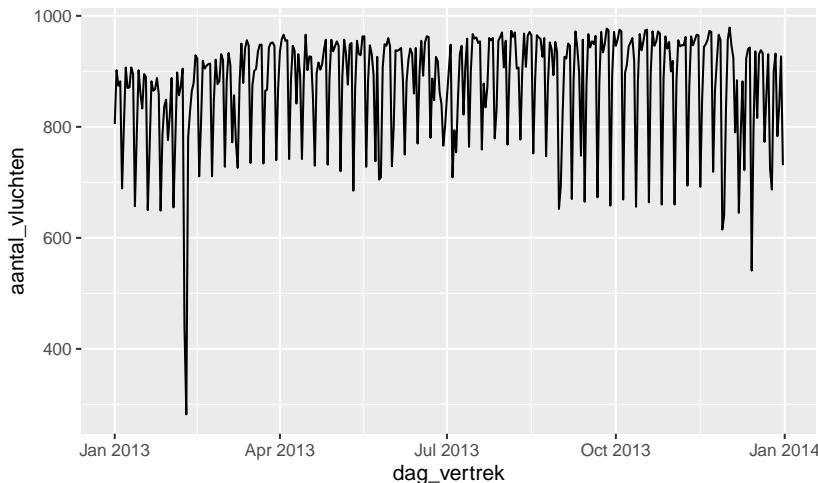
- Een eerste stap om inzicht te krijgen in de tijdgerelateerde data is met behulp van de *summary()* functie. Het is vooral nuttig om naar de minima en maxima te kijken. Dit geeft vaak aan of de tijdsperiode waarvoor de data verzameld is overeenkomt met de verwachte periode. In onderstaand geval blijkt dit in orde te zijn.

```
##  vertrek_luchthaven  aankomst_luchthaven maatschappij
##  Length:319809      Length:319809      Length:319809
##  Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character
##
##
##
##  vertrek_gepland          vertrek_werkelijk          vertrek_vertraging
##  Min.   :2013-01-01 05:17:00  Min.   :2013-01-01 05:19:00  Min.   :-43.00
##  1st Qu.:2013-04-05 09:07:00  1st Qu.:2013-04-05 09:10:00  1st Qu.: -5.00
##  Median :2013-07-04 13:43:00  Median :2013-07-04 13:47:00  Median : -2.00
##  Mean   :2013-07-03 21:27:29  Mean   :2013-07-03 21:40:07  Mean   : 12.62
##  3rd Qu.:2013-10-01 20:37:00  3rd Qu.:2013-10-01 20:39:00  3rd Qu.: 11.00
##  Max.   :2013-12-31 23:32:00  Max.   :2014-01-01 00:19:00  Max.   :1301.00
##
##  aankomst_gepland          aankomst_werkelijk
##  Min.   :2013-01-01 07:02:00  Min.   :2013-01-01 06:55:00
##  1st Qu.:2013-04-05 11:22:00  1st Qu.:2013-04-05 11:21:00
##  Median :2013-07-04 15:42:00  Median :2013-07-04 15:36:00
##  Mean   :2013-07-03 23:42:08  Mean   :2013-07-03 23:49:08
##  3rd Qu.:2013-10-01 22:27:00  3rd Qu.:2013-10-01 22:12:00
##  Max.   :2014-01-01 01:10:00  Max.   :2014-01-01 01:53:00
##
##  aankomst_vertraging      afstand      weekdag_vertrek  week_vertrek
##  Min.   :-86.000      Min.   : 80  Sun:44396        Min.   : 1.00
##  1st Qu.:-17.000      1st Qu.: 502 Mon:48246        1st Qu.:14.00
##  Median : -5.000      Median : 828 Tue:48084        Median :27.00
##  Mean   :  6.987      Mean   :1035 Wed:47597        Mean   :26.77
##  3rd Qu.: 14.000      3rd Qu.:1372 Thu:47378        3rd Qu.:40.00
##  Max.   :1272.000     Max.   :4983 Fri:47455        Max.   :53.00
##
##                                Sat:36653
##  maand_vertrek      dag_vertrek          maanddag_vertrek
##  Min.   : 1.000      Min.   :2013-01-01 00:00:00  Min.   : 1.00
##  1st Qu.: 4.000      1st Qu.:2013-04-05 00:00:00  1st Qu.: 8.00
##  Median : 7.000      Median :2013-07-04 00:00:00  Median :16.00
##  Mean   : 6.569      Mean   :2013-07-03 07:43:47  Mean   :15.74
```

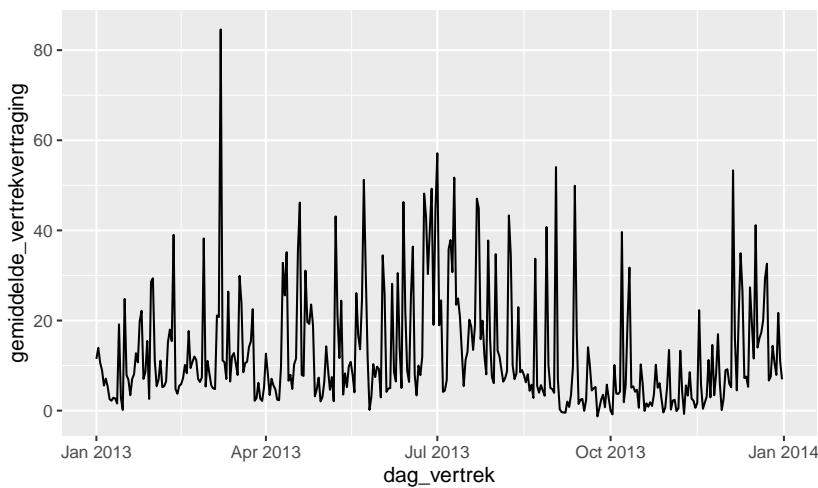
```
## 3rd Qu.:10.000 3rd Qu.:2013-10-01 00:00:00 3rd Qu.:23.00
## Max. :12.000 Max. :2013-12-31 00:00:00 Max. :31.00
##
```

### Analyse visuele tijdreeks patronen

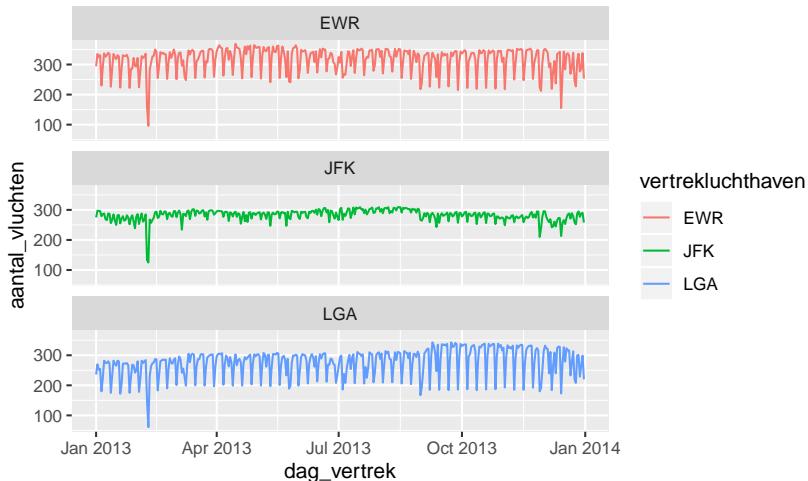
- Eén van de meest voorkomende exploratieve visuele analysetechnieken voor tijdgerelateerde data is het zoeken naar patronen hoe een variabele doorheen de tijd verandert.
- De eerste stap is hierbij telkens de tijdreeks patronen te visualiseren. Om dit te doen kan je volgend stappenplan toepassen.
  - Bepaal over welke tijdsdimensie je patronen wenst te bestuderen.  
Dit is je **X**-variabele. De **X**-variabele bepaalt de granulariteit van je visualisatie. Wens je op niveau van dagen te visualiseren, dan is je tijdsdimensie ‘dag’, en dan ga je gedetailleerder naar de patronen kijken, dan wanneer je op niveau van bijvoorbeeld ‘maand’ naar de data kijkt.
  - Bepaal welke variabele je doorheen de tijd wenst te bestuderen.  
Dit is je **Y**-variabele.
  - Je gaat voor iedere **X** waarde 1 **Y** waarde moeten hebben. Vaak betekent dit dat je deze **Y**-variabele nog moet aanmaken. Mogelijke **Y** variabelen zijn *het aantal observaties* per tijdsseenheid of de centrummaat (bv. mediaan) van een specifieke variabele.
  - Je R-code vertrekt steeds van de oorspronkelijke dataset, groepeert vervolgens op de tijdsdimensie, berekent de gewenste samenvattende statistiek (*summarise()*) en visualiseert vervolgens via *ggplot() + geom\_line()*.
- We willen bijvoorbeeld de evolutie zien van het aantal vluchten per dag. De tijdsdimensie is dus *dag\_vertrek* en de **Y**-variabele wordt gemaakt door het aantal rijen per dag te tellen.
- De analyse van onderstaande grafiek toont een aantal opvallende zaken:
  - Er is een zware en niet-wederkerende daling tussen januari en april. Hier moet iets uitzonderlijks gebeurd zijn.
  - We zien een terugkerend patroon, waarbij om de aantal dagen een daling is in het aantal vluchten.
  - De schommelingen en met name de daling op het einde van ieder terugkerend patroon wordt groter op het einde van het jaar.



- We kunnen een soortgelijke analyse doen voor de gemiddelde vertrekvertraging.



- Een volgende stap is vaak om de tijdreeks patronen apart te visualiseren voor de verschillende waarden van een categorische variabele.
- Dit kan op eenvoudige wijze door in onze R-code deze categorische variabele op te nemen in het `group_by()` gedeelte en vervolgens aparte plots te creëren met behulp van `facet_wrap()`.
- Laten we de evolutie van het aantal vluchten per dag bijvoorbeeld uitsplitsen per luchthaven.
- Uit onderstaande analyse blijkt dan dat het aantal vluchten vanuit JFK veel minder sterk schommelt dan EWR en LGA. Wel valt op dat alle drie de luchthavens een sterke uitzonderlijke daling kenden in de eerste helft van het jaar.



### *Identificeren van opmerkelijke gebeurtenissen in een tijdreeks*

- In de evolutie van het aantal vluchten valt op dat er een uitzonderlijke daling plaatsvond in de periode tussen januari en april.
- In zulke gevallen is het best te achterhalen wat hier precies de oorzaak is.
- De eerste stap is dan ook het exacte tijdstip te identificeren.
- We kunnen dit doen door de data te filteren op die dagen dat er zeer weinig vluchten zijn.

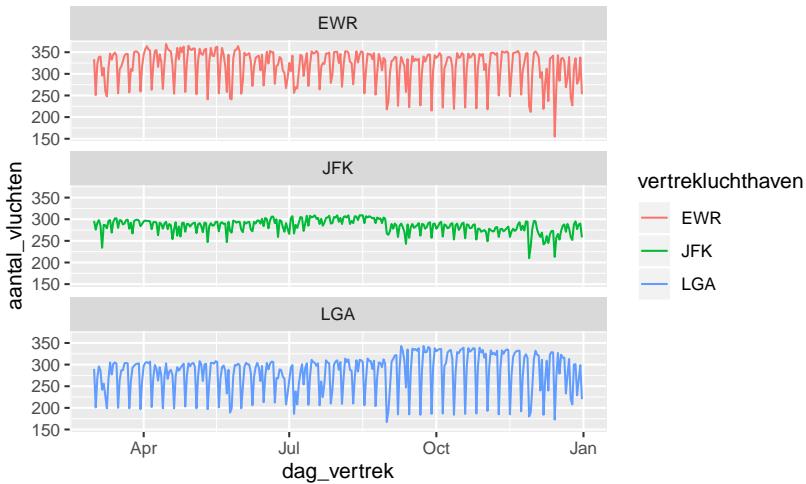
```
## # A tibble: 7 x 3
## # Groups:   dag_vertrek [3]
##   dag_vertrek     vertrekvluchthaven aantal_vluchten
##   <dttm>           <chr>                  <int>
## 1 2013-02-08 00:00:00 EWR                   159
## 2 2013-02-08 00:00:00 JFK                   133
## 3 2013-02-08 00:00:00 LGA                   148
## 4 2013-02-09 00:00:00 EWR                   96
## 5 2013-02-09 00:00:00 JFK                  125
## 6 2013-02-09 00:00:00 LGA                   61
## 7 2013-12-14 00:00:00 EWR                  155
```

- Uit deze analyse blijkt dat de daling plaatsvond op 8 en 9 februari 2013. Na enig opzoekwerk blijkt dat New York toen geteisterd werd door een hevige sneeuwstorm waardoor zeer veel vluchten geannuleerd moesten worden.
- Omdat dit moment niet representatief is voor een normaal jaar, beslissen we om enkel met de tijdgerelateerde data van maart tot en met december verder te gaan.

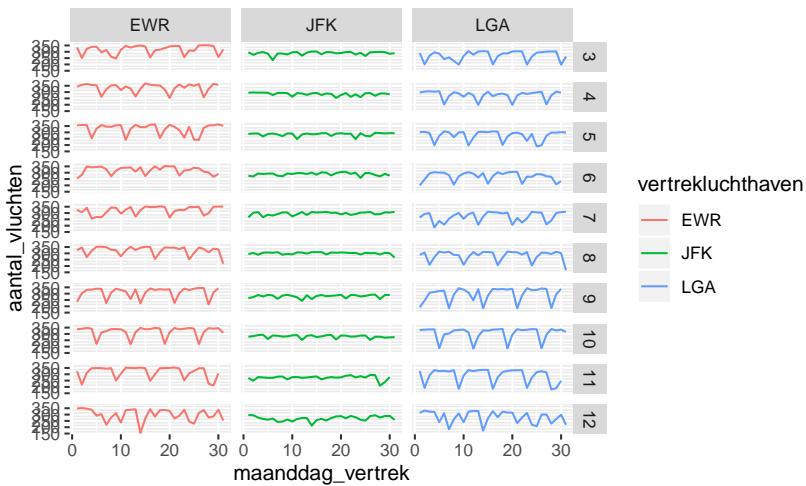
```
df %>%
  filter(vertrek_gepland >= ymd(130301)) -> df_mardec
```

- We kunnen de tijdreeks van de nieuwe periode opnieuw visualiseren.

```
df_mardec %>%
  group_by(dag_vertrek, vertrekvluchthaven) %>%
  summarise(aantal_vluchten=n()) %>%
  ggplot(aes(x = dag_vertrek, y= aantal_vluchten, colour=vertrekvluchthaven)) +
  geom_line() + facet_wrap(~ vertrekvluchthaven, ncol = 1)
```



- We kunnen verder inzoomen in de data door naar de tijdreekspatronen te kijken per maand en per luchthaven.



#### 11.4 Referenties

1. ‘R for Data Science’ van Grolemund en Wickham



# 12

## *Tutorial - tijdsdata*

### *12.1 Before we start*

```
## Last updated on 2020-02-24
```

```
library(dplyr)
library(lubridate)
```

Congratulations on reaching the final tutorial of this course. In this tutorial we will be playing with dates and times. To do this in an easy way, we will use the package lubridate, which you might need to install. We will also load dplyr for when we will use time in a data.frame context. We will not need any dataset, we will make all data ourselves (How cool is that?).

### *12.2 Introduction*

In current *times*, time data is very important, so it is crucial that you know how to work with it. If you think about it, there are so many types of data which have a time component. Just look at the different datasets we have been using.

- order data (each order is created, packed, shipped etc at a certain moment)
- terrorism data (each terrorism attack happened at a certain point in time, sadly)
- gapminder data (we had information on several years)
- nydeaths (deaths happen at a specific moment)
- flights depart and arrive at a specific time (and are sometimes delayed)
- concerts (concerts take place at a specific moment, tickets are bought at a specific moment)

Time is everywhere!

Except... well, yeah... you got me there: diamonds had nothing to do with time. But trust me, you'll sooner find order data in your future career than diamond data.

In essence, time and dates behave a lot like a continuous variable. You can compare two dates and see which one was later. You can compute the time difference between two dates. Most things that work for numbers will in a way also work for times or dates.

Allright, continuous data. So what's all the fuss about?

Well, time is a bit more complex. For starters, it has different components. Let's start with looking at some dates.

### 12.3 What's in a date?

Easy question: days, month, and year.

Of course, the tricky part is the order. Some first write the year, some the day. Some will write 1998, other will say 98. There is really no order you can think of which you cannot use. And then there are different separators like spaces, -, ., /. Let's look at this completely random date, which we have noted down in different formats.

- 1991-06-28
- 28 06 1991
- 28 June 91
- June 28, 1991
- 280691
- 1991/06/28

The possibilities are endless. However, thanks to lubridate, we don't need to bother much. We only need to know the order of the components Day, Month and Year.

Consider the following code. We create a character containing the date above.

```
date <- "1991-06-28"
```

Let's say I want to know the difference between that day and today. The following will obviously not work.

```
## "2020-02-24" - "1991-06-28"
```

I don't know about you, but I have learned that we cannot subtract two words from each other. What about if we remove the "'s?

```
2019-04-29 - 1991-06-28
```

```
## [1] -39
```

Unless we can travel it time, 28-06-1991, happened almost 28 years before 29-04-2019, not 39 years in the other direction. The difference is also not 39 days or 39 months. I assume you see what's going on here.

What we need is a new type of object. Not character, not numeric, not integer, not factor. We need a date!

We can make a date with the aptly-named ‘make\_date’ function in lubridate, where we can give year, month and date as an argument.

```
date <- make_date(year = 1991, month = 6, day = 28)
```

```
date
```

```
## [1] "1991-06-28"
```

Printing it doesn't really change a thing. Let's look at the class.<sup>1</sup>

```
class(date)
```

```
## [1] "Date"
```

Well, that seems alright.

Let's make a bunch of dates - let's say each day in april. Do you still remember the code blow? We saw it to create data.frames a long time ago. We use `tbl_df` to transform the `data.frame` to the easier to print tibble format.

```
dates_of_april <- data.frame(year = 2019,
                               month = 4,
                               day = 1:30) %>%
 tbl_df()
dates_of_april

## # A tibble: 30 x 3
##       year   month   day
##   <dbl> <dbl> <int>
## 1 2019     4     1
## 2 2019     4     2
## 3 2019     4     3
## 4 2019     4     4
## 5 2019     4     5
## 6 2019     4     6
## 7 2019     4     7
## 8 2019     4     8
## 9 2019     4     9
## 10 2019    4    10
## # ... with 20 more rows
```

<sup>1</sup> The words date and data are too different things. It's again a common source of typos.

Using mutate, we can now add the date as a separate variable.

```
dates_of_april %>%
  mutate(date = make_date(year, month, day)) -> dates_of_april
dates_of_april

## # A tibble: 30 x 4
##       year   month   day   date
##     <dbl>   <dbl>   <int> <date>
## 1    2019      4      1 2019-04-01
## 2    2019      4      2 2019-04-02
## 3    2019      4      3 2019-04-03
## 4    2019      4      4 2019-04-04
## 5    2019      4      5 2019-04-05
## 6    2019      4      6 2019-04-06
## 7    2019      4      7 2019-04-07
## 8    2019      4      8 2019-04-08
## 9    2019      4      9 2019-04-09
## 10   2019      4     10 2019-04-10
## # ... with 20 more rows
```

Look again at the type of this variable. It's called a Date.

```
dates_of_april %>%
  glimpse

## # Observations: 30
## # Variables: 4
## $ year   <dbl> 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2...
## $ month  <dbl> 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4...
## $ day    <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18...
## $ date   <date> 2019-04-01, 2019-04-02, 2019-04-03, 2019-04-04, 2019-04-05, ...
```

Of course, this only solves part of the problem. How do we go from something as “28-06-1991” to 28 and 6 and 1991 to a date? Suppose we have a table of dates where each date is stored as a character, like below. [^unite]

[^unite]; You can actually make this table yourself by uniting the year, month, day column of dates\_of\_april using tidyR’s unite.

```
dates_of_april_text

## # A tibble: 30 x 1
##       date
##     <chr>
## 1 2019-4-1
## 2 2019-4-2
```

```

## 3 2019-4-3
## 4 2019-4-4
## 5 2019-4-5
## 6 2019-4-6
## 7 2019-4-7
## 8 2019-4-8
## 9 2019-4-9
## 10 2019-4-10
## # ... with 20 more rows

```

Look closely, this date column is not a ‘date’, it is a character.

```

dates_of_april_text %>%
  glimpse()

## # Observations: 30
## # Variables: 1
## $ date <chr> "2019-4-1", "2019-4-2", "2019-4-3", "2019-4-4", "2019-4-5", "2...

```

However, using separate, we could separate these characters, and then applying make\_date.

```

library(tidyr)
dates_of_april_text %>%
  separate(date, into = c("year", "month", "day")) %>%
  mutate(date = make_date(year, month, day))

## # A tibble: 30 x 4
##   year month day   date
##   <chr> <chr> <chr> <date>
## 1 2019  4     1     2019-04-01
## 2 2019  4     2     2019-04-02
## 3 2019  4     3     2019-04-03
## 4 2019  4     4     2019-04-04
## 5 2019  4     5     2019-04-05
## 6 2019  4     6     2019-04-06
## 7 2019  4     7     2019-04-07
## 8 2019  4     8     2019-04-08
## 9 2019  4     9     2019-04-09
## 10 2019 4    10    2019-04-10
## # ... with 20 more rows

```

Of course, that’s a lot of work, which we don’t like. However, we can use lubridate. Not only can it make dates from its components, it can also detect these components from a character! We only need it to tell the order of the component.

Since year is written down first, then month, then day, the order of components is *ymd*. Well, there is a function in lubridate with that name. Let's try with our earlier date.

```
date <- ymd("1991-06-28")
date
```

```
## [1] "1991-06-28"
```

Let's make a second date for today.<sup>2</sup>

```
## date2 <- ymd("2020-02-24")
```

```
date2
```

```
## [1] "2020-02-24"
```

```
class(date2)
```

```
## [1] "Date"
```

There is actually a very useful function in lubridate for getting the date of today. It's just to `today()`

```
today()
```

```
## [1] "2020-02-24"
```

When the date components are in a different order, you just have to change the order of the letters y, m and d. The following functions are all provided by lubridate.

- dmy
- dym
- mdy
- myd
- ymd
- ydm

Here are some examples.

```
ymd("2017-03-25")
```

```
## [1] "2017-03-25"
```

```
dmy("25-03-2017")
```

```
## [1] "2017-03-25"
```

```
mdy("03-25-2017")
```

<sup>2</sup> Of course, this is not the day when you are reading the tutorial, but the last day that the tutorial was updated.

```
## [1] "2017-03-25"

ymd("17/03/25")

## [1] "2017-03-25"

ymd("20170325")

## [1] "2017-03-25"

ymd("170325")

## [1] "2017-03-25"
```

Note that lubridate will automagically find the components as long as it knows the order in which to look for - you don't have to worry about how it is separated, whether month and days are written with or without a leading zero (i.e. 1/4/2019 or 01/04/2019), or whether years are written with 2 or 4 numbers. Lubridate is even so smart it will try to guess the correct century. Suppose we define the date we saw earlier, 28th June 1991, but only give lubridate year 91.

```
ymd("910628")

## [1] "1991-06-28"
```

It turns 91 into 1991. But if we give it year 19 (i.e. the same day in 2019), it will turn 19 in 2019.

```
ymd("190628")

## [1] "2019-06-28"
```

This is very helpful of course (unless we are looking for the year 1919). However, after the Y2K bug, we will typically find yyyy-dates.<sup>3</sup>

So, let's go back to our table.

```
dates_of_april_text

## # A tibble: 30 x 1
##   date
##   <chr>
## 1 2019-4-1
## 2 2019-4-2
## 3 2019-4-3
## 4 2019-4-4
## 5 2019-4-5
## 6 2019-4-6
```

<sup>3</sup> Fun fact for geeks: lubridate will try to find a date within 50 years from now. Thus, all numbers from 69 to 99 will be regarded as 1999 to 1999, all other numbers will be regarded as years in the 21st century.

```
## 7 2019-4-7
## 8 2019-4-8
## 9 2019-4-9
## 10 2019-4-10
## # ... with 20 more rows
```

Luckily, ymd works with vectors, as well as with singular dates. So, instead of this

```
dates_of_april_text %>%
  separate(date, into = c("year", "month", "day")) %>%
  mutate(date = make_date(year, month, day))
```

we can now just do this

```
dates_of_april_text %>%
  mutate(date = ymd(date))

## # A tibble: 30 x 1
##   date
##   <date>
## 1 2019-04-01
## 2 2019-04-02
## 3 2019-04-03
## 4 2019-04-04
## 5 2019-04-05
## 6 2019-04-06
## 7 2019-04-07
## 8 2019-04-08
## 9 2019-04-09
## 10 2019-04-10
## # ... with 20 more rows
```

which obviously is a lot easier.

Of course, there are always times when lubridate can make mistakes. Consider the funny first day of april. In some parts of the world, they write dates in a month-day-year order, especially in the United States.

Thus, the first day of april is written down as 04-01-2019. If we would see this day without any context, we would wrongly think that 4 is the day, 1 the month and 2019 the year. Thus,

```
dmy("04-01-2019")
## [1] "2019-01-04"
```

Lubridate will not see any problems because this is not an incorrect date. But, we are no longer talking about the first of April. We are

talking about the 4 of January. The second of april will become the 4th of February using this code. Lubridate will sense that something is going wrong when we get to the 13th of april.

```
dmy("04-13-2019")
```

```
## [1] NA
```

Now lubridate will fail to see this as a correct dmy-date, and return an NA. However, it is good to know that lubridate doesn't learn very good from its mistakes. Suppose we have again a whole data set of dates in this format.<sup>4</sup>

```
dates_of_april_us
```

```
## # A tibble: 30 x 1
##   date
##   <chr>
## 1 4-1-2019
## 2 4-2-2019
## 3 4-3-2019
## 4 4-4-2019
## 5 4-5-2019
## 6 4-6-2019
## 7 4-7-2019
## 8 4-8-2019
## 9 4-9-2019
## 10 4-10-2019
## # ... with 20 more rows
```

When we try to convert is in a date using the wrong order, dmy. We get the following.

```
## # A tibble: 30 x 1
##   date
##   <date>
## 1 2019-01-04
## 2 2019-02-04
## 3 2019-03-04
## 4 2019-04-04
## 5 2019-05-04
## 6 2019-06-04
## 7 2019-07-04
## 8 2019-08-04
## 9 2019-09-04
## 10 2019-10-04
```

<sup>4</sup> You can create this table yourself by uniting the columns dates\_of\_april table in the month, day, year order.

```

## 11 2019-11-04
## 12 2019-12-04
## 13 NA
## 14 NA
## 15 NA
## 16 NA
## 17 NA
## 18 NA
## 19 NA
## 20 NA
## 21 NA
## 22 NA
## 23 NA
## 24 NA
## 25 NA
## 26 NA
## 27 NA
## 28 NA
## 29 NA
## 30 NA

dates_of_april_us %>%
  mutate(date = dmy(date))

```

Note that all dates till the 13th are (wrongly) converted. The remaining ones are not converted and replaced with NA, of which lubridate warns us. However, he's not so smart to correct our mistake and replace the first 12 days with NA.<sup>5</sup>

Let's return to our dates we defined earlier

```

date

## [1] "1991-06-28"

date2

## [1] "2020-02-24"

```

We can now calculate with these dates.

```

date2 - date

## Time difference of 10468 days

```

Problem solved. Let's add some other problems.

<sup>5</sup> Not that lubridate date doesn't learn retroactively (replacing the already converted dates with NA), but is also doesn't learn for the future. If we shuffle the data such that we'll see a wrong date faster, it will still wrongly convert all of the 12 first days of april. So he will always do his best, even if he has already had problems with some dates. Moral of the story: the data scientist should never be asleep and blindly trust tools.

## 12.4 There's always time

Instead of dates, we can also have times. In this tutorial, we will only consider the combination of date and time, which we call a *datetime*. We can look at an example using the function `now`. While `today` gives us the date of today, `now` gives us the date time of, well, now.

```
current_time <- now()
current_time

## [1] "2020-02-24 17:04:11 CET"

class(current_time)

## [1] "POSIXct" "POSIXt"
```

The class of a datetime is no longer a “date”, it is a “POSIXct” and “POSIXt”.<sup>6</sup>

Similarly to dates, we can create datetimes in two ways.

1. Using `make_datetime`, and giving all components (hours, minutes, seconds and timezone).
2. Using `ymd_hms`, and giving a character.

Next to `ymd_hms`, there's also `dmy_hms`, `mdy_hms`, etc. However, hours, minutes and second should always be in the same order. There is no `ymd_smh` for instances. But, we will never find a datetime in that format either, so no worries. There *is* a `ymd_hm` and a `ymd_h` (and this for all `ymd` orders), which we can use if we just have the hour, or the hour and the minutes, but not the more detailed parts.

You will mostly use `ymd_hms`. Let's see some examples. Note that the third notations, with a T between the date and time part, is very common in datasets, and `lubridate` takes care without problems.

```
dmy_hms("25-03-2017 10:30:00")

## [1] "2017-03-25 10:30:00 UTC"

dmy_hms("25032017103000")

## [1] "2017-03-25 10:30:00 UTC"

dmy_hms("25-03-2017T10:30:00")

## [1] "2017-03-25 10:30:00 UTC"

ymd_hms("2017-03-25 10.30.00")
```

<sup>6</sup> What the hell does that mean? POSIXct is the number of seconds sinds the start of the first of January in 1970. R uses this number of seconds to store a datetime. Another way to store it is by storing, separately the seconds, minutes, hours, etc. Then it is a POSIXlt. The difference between those two will not change for the user, only how the date is stored. POSIXt is a parent class which makes sure that most functions can work with both types of storage. There's no need to remember these names. Just know that everything starting with POSI is a datetime, not just a date.

```
## [1] "2017-03-25 10:30:00 UTC"
dmy_hms("25-03-2017 10h30m00s")
## [1] "2017-03-25 10:30:00 UTC"
dmy_hm("25-03-2017 10:30")
## [1] "2017-03-25 10:30:00 UTC"
```

Again, you can use this on entire columns using `mutate`. Also, make sure to check what the actual order of components is. We already saw that `lubridate` can make mistakes also.

Once we have datetimes, we can again do calculations with them.

```
datetime <- dmy_hms("25-03-2017 10h30m00s")
now() - datetime

## Time difference of 1066.232 days
```

Or compare them.

```
now() < datetime

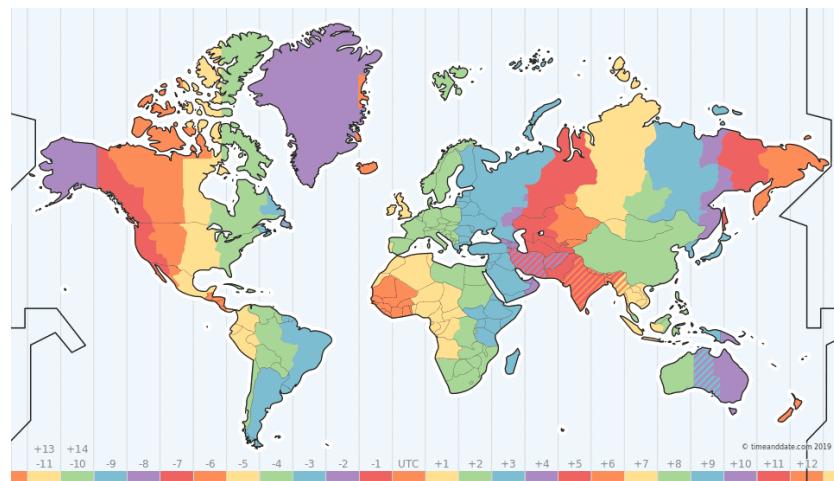
## [1] FALSE
```

We'll learn a lot more what we can do with dates, but there is still one more hurdle to take.

## 12.5 My time is not your time

So, handling time is not much more difficult from handling dates. Just add a few additional letters to the `ymd` functions. This is mainly true if you stay in the same place, like a tree. Once you start moving to different countries, time gets more difficult. Just crossing the North Sea channel and we have to change our clocks.

Below, you can see a map of all different time zones.



When you graduate from this program you'll most certainly become successful international businessleaders, so having a basic understanding about time zones is necessary. Let's again look at the current time.

```
now()
```

```
## [1] "2020-02-24 17:04:12 CET"
```

The last part of this time tells us the timezone we are in. At my pc, it currently says CEST, which is the Central European Summer Time<sup>7</sup>. You also check the time zone you are in using Sys.timezone().

```
Sys.timezone()
```

```
## [1] "Europe/Paris"
```

At this point, my pc tell me it is Europe/Paris instead of CEST. Indeed, there are multiple names for the same zone. It's all a great master scheme to get you confused.

The function olson\_time\_zones gives you a list of all time zones according to the Olson time database. This is notation of time zones using Continent/City. It is named after Arthur Olson

```
olson_time_zones()
```

Just try this in your console. It's a long list, and we don't want to spoil the environment with our paper usage.

When we are creating a date with a ymd\_hms-variant, it will automatically use UTC as a time zone.

```
ymd_hms("2017-03-25 10.30.00")
```

```
## [1] "2017-03-25 10:30:00 UTC"
```

UTC is the Universal Coordinated Time, i.e. the standard, base-line time zone. CEST is UTC + 2, i.e. two hours ahead of UTC. We can change that by setting the tz argument within the ymd\_hms function. Let's first take a more meaningful datetime.

```
ymd_hms("2017-01-20 17:30:00")
```

```
## [1] "2017-01-20 17:30:00 UTC"
```

At time a sad event happened in Washington DC. So let's define the time zone appropriately.

```
ymd_hms("2017-01-20 11:30:00", tz = "EST")
```

```
## [1] "2017-01-20 11:30:00 EST"
```

<sup>7</sup> Note that if you read the online version of this tutorial, the time zones you see here may be different because it might be compiled on a server in a different time zone.

Note that EST is the Eastern Standard time, the time zone used in winter in DC. You can see that the time doesn't really change, only the indication of the timezone.

We can now see what time it was back in Belgium, by showing this time in CET (Central European Time, that is, our winter time).

First, we save the time as "doomsday"

```
doomsday <- ymd_hms("2017-01-20 11:30:00", tz = "EST")
```

Using with\_tz, we can show a point in time in a different timezone.

```
with_tz(doomsday, "CET")
## [1] "2017-01-20 17:30:00 CET"
```

So, at 11.30 in Washington, it was 17.30 here in Belgium when we were watching the terrifying events occur live on your tv screens.

But, both refer to the same point in time. They are just different representations of that same point.

```
doomsday_in_belgium <- with_tz(doomsday, "CET")
```

```
doomsday_in_belgium - doomsday
## Time difference of 0 secs
```

If we really want to change the time zone, i.e. refer to 11.30 in Belgium, we can use force\_tz instead.

```
force_tz(doomsday, "CET")
## [1] "2017-01-20 11:30:00 CET"
```

At this point, we are no longer talking about the same point in time, but we are talking about the time 6 hours before doomsday.

```
force_tz(doomsday, "CET") - doomsday
## Time difference of -6 hours
```

We'll come back to time zones in a minute, because even in Belgium we change our clocks now and again (at least for now). First, let's have a look at the things we can do with dates and times.

## 12.6 Extract information from dates

When we have a date (or datetime), we can easily extract information from it. For example, consider doomsday again.

We can see which day it was.

```
day(doomsday)
## [1] 20

or at which hour

hour(doomsday)
## [1] 11

or even which day of the week

wday(doomsday)
## [1] 6
```

It was the 6th day of the week. But if you remember doomsday as well as I, you will have it printed on your eyelids that it was a terrible friday?

Indeed, lubridate will start counting on sunday - another American quirk. So, sunday is the first day of the week. Saturday the seventh. And friday the sixth.

Luckily, the wday function has an argument label, which we can set to TRUE when we forgot the exact meaning.

```
wday(doomsday, label = TRUE)
## [1] Fri
## Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

Friday it was. Not that it is automatically an ordered factor, which is helpful. Also note that it might not be in English on your pc. Time is a little different for each of us.

We can also ask for the month of doomsday as both a number and a label.

```
month(doomsday)
## [1] 1

month(doomsday, label = T)
## [1] Jan
## 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
```

Or course, the first month of the year is Januari. Even in the US. (Might not be so in China though).

Using these functions we can look at time data from very different angles. We can compare months, days of the week, hours, etc. It shows how complex time is, but also in how different ways we can use it to analyse something. That's why it's so important to know.

## 12.7 Time differences

There are three different ways to express a time difference.

- A period
- A duration
- An interval

A period is expressed in components: hours, days, months, years. It is how we *perceive* time

A duration is expressed as an amount of seconds. It is like the a machine or clock counts time.

An interval is a *exact* window of time, with a specific start date(time) and a specific end date(time). It is something totally different from periods and durations. Let's look at those first.

### 12.7.1 Period

We can create a period with the period function, by giving it a number of components.

```
period(days = 2, minutes = 4, seconds = 70)
## [1] "2d 0H 4M 70S"
```

Note that it will just give us the period in easy to understand chunks of time. 2 hours, 4 minutes and 70 seconds. It won't even change it to 2 hours, 5 minutes and 10 seconds, because that's not what we said. Periods have a human touch to it.

If our can be easily expressed in a single component, we can use the follow that components day (in plural).

```
days(3)
## [1] "3d 0H 0M 0S"

weeks(3)
## [1] "21d 0H 0M 0S"

months(2)
## [1] "2m 0d 0H 0M 0S"

years(4)
## [1] "4y 0m 0d 0H 0M 0S"
```

We need to use the plural (days, not day), because we already used day to extract the day from a date (remember?). When we use weeks, it will translate each week to 7 days. When we use months or years, it will created additional components in its output.

We can combine them.

```
days(1) + hours(1)
```

```
## [1] "1d 1H 0M 0S"
```

We can also make negative periods.

```
days(-10)
```

```
## [1] "-10d 0H 0M 0S"
```

Or we can make vectors of periods.

```
period(days = 1:5, hours = 3:7)
```

```
## [1] "1d 3H 0M 0S" "2d 4H 0M 0S" "3d 5H 0M 0S" "4d 6H 0M 0S" "5d 7H 0M 0S"
```

```
days(0:6)
```

```
## [1] "0S"           "1d 0H 0M 0S" "2d 0H 0M 0S" "3d 0H 0M 0S" "4d 0H 0M 0S"
## [6] "5d 0H 0M 0S" "6d 0H 0M 0S"
```

We can than also substract or add these periods to dates. Thus, if today is

```
today()
```

```
## [1] "2020-02-24"
```

the next 6 days are

```
today() + days(1:6)
```

```
## [1] "2020-02-25" "2020-02-26" "2020-02-27" "2020-02-28" "2020-02-29"
```

```
## [6] "2020-03-01"
```

### 12.7.2 Durations

Instead of using these human-understandable components, durations translate all time difference to seconds. Just like `period`, we can use `duration`.

```
duration(days = 2, minutes = 4, seconds = 70)
```

```
## [1] "173110s (~2 days)"
```

As you see, it will always try to give an approximation of the time differences in understandable terms for us, humans.

Indeed, we can also do the math

```
2*24*3600+4*60+70
```

```
## [1] 173110
```

Similarly, we can use ddays, dweek, dyears, etc.<sup>8</sup>

```
ddays(3)
```

```
## [1] "259200s (~3 days)"
```

```
dweeks(3)
```

```
## [1] "1814400s (~3 weeks)"
```

```
dyears(4)
```

```
## [1] "126144000s (~4 years)"
```

We can combine them.

```
ddays(1) + dhours(1)
```

```
## [1] "90000s (~1.04 days)"
```

We can not make negative durations, but it wont really give an error.

```
ddays(-10)
```

```
## [1] "864000s (~1.43 weeks)"
```

Note that there is no such thing as a negative second.

We can make vectors of periods.

```
duration(days = 1:5, hours = 3:7)
```

```
## [1] "97200s (~1.12 days)" "187200s (~2.17 days)" "277200s (~3.21 days)"  
## [4] "367200s (~4.25 days)" "457200s (~5.29 days)"
```

```
ddays(0:6)
```

```
## [1] "0s" "86400s (~1 days)" "172800s (~2 days)"  
## [4] "259200s (~3 days)" "345600s (~4 days)" "432000s (~5 days)"  
## [7] "518400s (~6 days)"
```

We can than also substract or add these durations to dates. Thus, if today is

<sup>8</sup> Evidently, this has nothing to do with D-Day. It's just dat day and days were already taken, and d stands for duration.

```
today()
## [1] "2020-02-24"

the next 6 days are

today() + ddays(1:6)

## [1] "2020-02-25" "2020-02-26" "2020-02-27" "2020-02-28" "2020-02-29"
## [6] "2020-03-01"
```

Note that we cannot compute dmonths!

```
dmonths(1)

## Error in dmonths(1): could not find function "dmonths"
```

There is no fixed length for months in number of seconds, of course.

9

Now, you wonder, why do we have both periods and durations? Well, there are some important differences.

### 12.7.3 Periods are not durations

As I already mentioned, durations are machine-interpretable seconds, and periods are human-interpretable days.

If we, humans, say “next month” on the 3th of february, we mean the 3th of march. A machine wouldn’t know what a month is.

Let’s look at some examples too make things clear.

Let’s take the first day of 2019, which is not a leap year.

```
ymd("2019-01-01")

## [1] "2019-01-01"

If we add a period-year

ymd("2019-01-01") + years(1)

## [1] "2020-01-01"

or a duration-year

ymd("2019-01-01") + dyears(1)

## [1] "2020-01-01"
```

both are the same, because how we perceive the length of 2019 (i.e. 365 days), is exactly the duration of a year.

However, if we add 2 years, there is a difference, because 2020 is a leapyear and how we perceive it (366 days) is not the same as the duration of a default year.

<sup>9</sup> You might wonder that there is also no fixed length for years, depending on whether we have a leap year or not. True, but dyears will implicitly use the common conventions that a year has 365 days, which is true more than 75% of the time. We don’t have such a convention for months.

```
ymd("2019-01-01")
## [1] "2019-01-01"
```

If we add two period-years

```
ymd("2019-01-01") + years(2)
## [1] "2021-01-01"
or two duration-years
ymd("2019-01-01") + dyears(2)
## [1] "2020-12-31"
```

“Next year” on New Year 2020 for us humans means New Year 2021, even though it is a leapyear. For durations, it means New Years Eve, because it assumes a year has 365 days.

Let’s take another example. On 31 march we changed from our winter time (CET) to summer time. What does it mean to add “a day” to 4pm on saturday.

```
ymd_hms("2019-03-30 16:00:00", tz = "CET") + days(1)
## [1] "2019-03-31 16:00:00 CEST"
ymd_hms("2019-03-30 16:00:00", tz = "CET") + ddays(1)
## [1] "2019-03-31 17:00:00 CEST"
```

If we add a human period, it will say 4pm on sunday, even though that’s only 23 hours later. If we a machine-duration, it will tell us that “a day later” is sunday at 5pm, because only then have 24 hours passed.

Of course, there is no difference on any other day of the year.

```
ymd_hms("2019-04-30 16:00:00", tz = "CET") + days(1)
## [1] "2019-05-01 16:00:00 CEST"
ymd_hms("2019-04-30 16:00:00", tz = "CET") + ddays(1)
## [1] "2019-05-01 16:00:00 CEST"
```

So, the difference is only in special cases, but a difference still.

## 12.8 Arithmetics

We've already seen quite a lot of computations we can do. Here are some other examples.

```
doomsday >= now()

## [1] FALSE

now() - doomsday

## Time difference of 1129.982 days

as.period(now()-doomsday)

## [1] "1129d 23H 34M 15.2572860866785S"

doomsday + period(1,"week")

## [1] "2017-01-27 11:30:00 EST"

leap_year(2012)

## [1] TRUE

leap_year(now())

## [1] TRUE

now() + weeks(0:2)

## [1] "2020-02-24 17:04:15 CET" "2020-03-02 17:04:15 CET"
## [3] "2020-03-09 17:04:15 CET"
```

Sometimes, calculations can go wrong. Especially with months.

```
ymd(20190331) + months(1)

## [1] NA
```

Well, the 31st of April doesn't really exist. (I suppose we all would like to have an extra 24 hours, but we don't.)

You can use the special `%m+%` operator to make sure that your additions never go over the months end.

```
ymd(20190331) %m+% months(1)

## [1] "2019-04-30"
```

Below, we have tried both to create a list of every months last day this year.

```

ymd(20190131) + months(0:11)

## [1] "2019-01-31" NA          "2019-03-31" NA          "2019-05-31"
## [6] NA          "2019-07-31" "2019-08-31" NA          "2019-10-31"
## [11] NA         "2019-12-31"

ymd(20190131) %m+% months(0:11)

## [1] "2019-01-31" "2019-02-28" "2019-03-31" "2019-04-30" "2019-05-31"
## [6] "2019-06-30" "2019-07-31" "2019-08-31" "2019-09-30" "2019-10-31"
## [11] "2019-11-30" "2019-12-31"

```

## 12.9 Interval

Finally, let's look at interval. Durations and periods only have a length. A period or a duration of a specific length can occur at any time.

An interval is different, as it refers to on specific window in time. It is not defined by its length, but rather by its start and end point.

Victor Hugo lived from February 26, 1802 until May 22, 1885. So we can define an interval using these two dates which represents his lifetime

```

hugo <- interval(mdy(02261802), mdy(05221885))
hugo

## [1] 1802-02-26 UTC--1885-05-22 UTC

class(hugo)

## [1] "Interval"
## attr(,"package")
## [1] "lubridate"

```

Note that we can of course also use datetimes to define an interval. (We just don't have any specific information on Hugo's hour of birth or death.)

Oscar Wilde lived from October 16, 1854 until November 30, 1900. Thus,

```

wilde <- interval(mdy(10161854), mdy(11301900))
wilde

## [1] 1854-10-16 UTC--1900-11-30 UTC

```

We can also create an interval with the special `%-%` operator. Thus the code above is equivalent to

```
wilde <- mdy(10161854) %--% mdy(11301900)
wilde
## [1] 1854-10-16 UTC--1900-11-30 UTC
```

It's shorter to write, but using the function is somewhat more readable.

We can check whether a date falls in an interval using the `%within%` function. (Not to be confused with the `%in%` function).

Charles Dickens was born on the 7th of februari 1812.

```
dickens_birthday <- dmy(7021812)

dickens_birthday %within% hugo

## [1] TRUE

dickens_birthday %within% wilde

## [1] FALSE
```

Hugo lived on that day, as this day fall within the interval of his life. Wilde didn't roam the earth on that day.

We can check whether two intervals overlap. I.e. did Hugo and Wilde both live at a certain point in history.

```
int_overlaps(hugo, wilde)

## [1] TRUE
```

Of course they did. We can also create the exact time interval both lived.

```
intersect(hugo, wilde)

## [1] 1854-10-16 UTC--1885-05-22 UTC
```

Or the total time interval that at least one of them lived

```
union(hugo, wilde)

## [1] 1802-02-26 UTC--1900-11-30 UTC
```

**Important:** note that intersect and union only works if you did `library(lubridate)` after `library(dplyr)`. This is because both packages have those functions, and the package that was loaded last is found first by R. It is like the book on top in your library, if you like. If you are not sure which you loaded first, you can solve this problem by placing the package name and two `:`'s before the function.

```
lubridate::union(hugo, wilde)
## [1] 1802-02-26 UTC--1900-11-30 UTC

lubridate::intersect(hugo, wilde)
## [1] 1854-10-16 UTC--1885-05-22 UTC
```

Note that when we explicitly use the dplyr functions, we do not get anything useful for intervals.

```
dplyr::union(hugo, wilde)
## [1] 2626646400 1455494400

dplyr::intersect(hugo, wilde)
## numeric(0)
```

Great, you have now learned so many functions that different packages are beginning to be in conflict with each other. You have thereby entered the realm of a true R-programmers and data scientists, congratz!

— The End —