Introduction to Data Visualization in Python

Practical, hands-on, straightforward guide to understanding and performing Data Visualization in Python using Plotnine, Matplotlib and Seaborn.

Duvérier DJIFACK ZEBAZE

Introduction to Data Visualization in Python

Duvérier DJIFACK ZEBAZE

Contents

Introduction 1

1 Data preparation 4

1.1 Importing data . . . 4

1.1.1 Test files . . . 4

1.1.2 Excel spreadsheets . . . 4

1.1.3 Statistical packages . . . 5

1.1.4 Databases . . . 5

1.2 Cleaning data . . . 5

1.2.1 Selecting variables . . . 6

1.2.2 Selecting observations . . . 7

1.2.3 Creating/recoding variables . . . 8

1.2.4 Summarizing data . . . 9

1.2.5 Using pipes . . . 10

1.2.6 Reshaping data . . . 10

1.2.7 Missing data . . . 11

2 Introduction to plotnine 14

2.1 A worked example . . . 14

2.1.1 ggplot . . . 14

2.1.2 geoms . . . 15

2.1.3 grouping . . . 16

2.1.4 scales . . . 17

2.1.5 facets . . . 18

2.1.6 labels . . . 19

2.1.7 themes . . . 20

2.2 Placing the `data` and `mapping` options . . . 20
2.3 Graphs as objects . . . 21

3 Univariate Graphs **24**
3.1 Categorical . . . 24
3.1.1 Bar chart . . . 24
3.1.2 Pie chart . . . 29
3.1.3 Tree map . . . 30
3.2 Quantitative . . . 32
3.2.1 Histogram . . . 32
3.2.2 Kernel Density plot . . . 34
3.2.3 Dot Chart . . . 36

4 Bivariate Graphs **38**
4.1 Categorical vs. Categorical . . . 38
4.1.1 Stacked bar chart . . . 38
4.1.2 Grouped bar chart . . . 39
4.1.3 Segmented bar chart . . . 40
4.1.4 Improving the color and labeling . . . 40
4.2 Quantitative vs. Quantitative . . . 42
4.2.1 Scatterplot . . . 42
4.2.2 Line plot . . . 45
4.3 Categorical vs. Quantitative . . . 47
4.3.1 Bar chart (on summary statistics) . . . 47
4.3.2 Grouped kernel density plots . . . 48
4.3.3 Box plots . . . 49
4.3.4 Violin plots . . . 50
4.3.5 Ridgeline plots . . . 51
4.3.6 Mean/SEM plots . . . 52
4.3.7 Strip plots . . . 54
4.3.8 Beeswarm plots . . . 57
4.3.9 Cleveland Dot Charts . . . 58

5 Multivariate Graphs **60**
5.1 Grouping . . . 60
5.2 Faceting . . . 63

6 Time - dependent graphs **67**
6.1 Time series . . . 67

6.2 Dumbbell charts . . . 71
6.3 Slope graphs . . . 73
6.4 Area Charts . . . 75

7 Statistics Models **78**
7.1 Correlation plots . . . 78
7.2 Linear Regression . . . 79
7.3 Logistic regression . . . 81
7.3.1 Setting a reference or base level for categorical variables . . . 81
7.4 Survival plots . . . 82
7.4.1 The Veterans' Administration Lung Cancer Trial . . . 82
7.4.2 Survival Data . . . 83
7.4.3 The Survival Function . . . 83
7.4.4 Considering other variables by stratification . . . 84
7.4.5 Survival functions by cell type . . . 85
7.4.6 Multivariate Survival Models . . . 86
7.5 Mosaic plots . . . 89

8 Other Graphs **90**
8.1 3-D scatterplot . . . 90
8.2 Biplots . . . 93
8.3 Bubble charts . . . 97
8.4 Flow diagrams . . . 98
8.4.1 Sankey diagrams . . . 98
8.4.2 What is Sankey Diagram? . . . 98
8.4.3 Alluvial diagrams . . . 101
8.5 Heatmaps . . . 103
8.6 Radar charts . . . 103
8.7 Scatterplot matrix . . . 107
8.8 Waterfall charts . . . 107
8.9 Word clouds . . . 109

9 Customizing graphs **112**
9.1 Axes . . . 112
9.1.1 Quantitative axes . . . 112
9.1.2 Categorical axes . . . 114
9.1.3 Date axes . . . 114
9.2 Colors . . . 116

9.2.1 Specifying colors manually . . . 116
9.2.2 Color palettes . . . 116
9.3 Points & Lines . . . 118
9.3.1 Points . . . 118
9.3.2 Lines . . . 118
9.4 Fonts . . . 118
9.5 Legends . . . 119
9.5.1 Legend location . . . 119
9.5.2 Legend title . . . 119
9.6 Labels . . . 120
9.7 Annotations . . . 121
9.7.1 Adding text . . . 121
9.7.2 Adding lines . . . 123
9.8 Themes . . . 124
9.8.1 Altering theme elements . . . 124

Conclusion **126**

References **127**

Introduction

About this book

This book is based on the book **Data visualization with R** by Rob Kabacoff who used the R platform to introduce data visualization. His book is available on https://rkabacoff.github.io/datavis/index.html.

How to use this book?

You don't need to read this book from start to finish in order to start building effective graphs. Feel free to jump to the section that you need and then explore others that you find interesting.

Graphs are organized by the number of variables to be plotted, the type of variables to be plotted and the purpose of the visualization.

Table 1: Overview

Chapter	Description
1	provides a quick overview of how to get your data into Python and how to prepare it for analysis
2	provides an overview of the `plotnine` package
3	describes graphs for visualizing the distribution of a single categorical (e.g. race) or quantitative (e.g. income) variable.
4	describes graphs that display the relationship between two variables.
5	describes graphs that display the relationships among 3 or more variables. It is helpful to read chapters 3 and 4 before this chapter.
6	describes graphs that display change over time.
7	describes graphs that can help you interpret the results of statistical models.
8	covers graphs that do not fit neatly elsewhere (every book needs a miscellaneous chapter).
9	describes how to customize the look and feel of your graphs. If you are going to share your graphs with others, be sure to skim this chapter.

Setup

This book is essentially based on three python libraries: plotnine, plydata and mizani.

About plotnine

When doing descriptive statistics we frequently need to partition the graphics based on categorical (i.e. qualitative) or ordinal variables. Doing such graphics may be particularly difficult when using classical Python graphical libraries (e.g matplotlib). The R software benefits from a very nice library for such a task, the ggplot2 package developed by Hadley Wickham (Wickham 2016) This package has been quickly became really popular in the bioinformatic field (here categories may be gene, groups of genes, species, signaling pathways, epigenetic marks…) and ordinal variables a discretized level of expression, for instance. The ggplot2 R package is an implementation of the graphical model proposed by Leland Wilkison in his book: The Grammar of Graphics (Wilkinson 2016). In this model, the graph is viewed as an entity composed of data, layers, scales, coordinate system and facets. One can create a graphic then add the various component using the + operator. Although the syntax may appear a little bit tricky for beginners, one can quickly understand the benefit of such an approach when composing complexe diagrams.

Several projects have proposed a port of ggplot2 under Python. The plotnine library is one of these projects that proposes a rather stable and exhaustive port of ggplot2 under Python.

A grammar of graphics for Python

plotnine is an implementation of a *grammar of graphics* in Python based on ggplot2. The grammar allows you to compose plots by explicitly mapping variables in a dataframe to the visual objects that make up the plot.

Plotting with a *grammar of graphics* is powerful. Custom (and otherwise complex) plots are easy to think about and build incremently, while the simple plots remain simple to create.

Installation

plotnine can be installed in a couple of ways, depending on purpose.

```
# Using pip
$ pip install plotnine             # 1. should be sufficient for most
$ pip install 'plotnine[extra]'    # 2. includes extra/optional packages
$ pip install 'plotnine[test]'     # 3. testing
$ pip install 'plotnine[doc]'      # 4. generating docs
$ pip install 'plotnine[dev]'      # 5. development (making releases)
$ pip install 'plotnine[all]'      # 6. everyting

# Or using conda
$ conda install -c conda-forge plotnine
```

About plydata

plydata is a library that provides a grammar for data manipulation. The grammar consists of verbs that can be applied to pandas dataframes or database tables. It is based on the R packages

dplyr, tidyr and forcats. plydata uses the `>>` operator as a pipe symbol, alternatively there is the `ply(data, *verbs)` function that you can use instead of `>>`.

At present the only supported data store is the pandas dataframe

Installation

plydata **only** supports Python 3

```
pip install plydata
```

For more about plydata package, read the plydata documentation.

About Mizani

Mizani is python library that provides the pieces necessary to create scales for a graphics system. It is based on the R Scales package.

Installation

mizani can be can be installed in a couple of ways depending on purpose.

```
pip install mizani
```

For more about mizani package, read the documentation.

1 Data preparation

Before you can visualize your data, you have to get it into python. This involves importing the data from external source and massaging it into a useful format.

1.1 Importing data

Python can import data from almost source, including text files, excel spreadsheets, statistical packages, and database management systems. We'll illustrate these techniques using the Salaries dataset, containing the 9 month academic salaries of college professors at a single institution in 2008-2009.

1.1.1 Test files

The pandas packages provides functions for importing delimited text files into python data frames.

```
import pandas as pd

# Import data frame from a comma delimited file
Salaries_csv = pd.read_csv("salaries.csv")

# Import data frame from a tab delimited file
Salaries_txt = pd.read_csv("salaries.txt", sep = "\t")
```

These function assume that the first line of data contains the variables names, values are separated by commas or tabs respectively, and that missing data are represented by blancs.

Options allow you to alter these assumptions. See the documentation for more details.

1.1.2 Excel spreadsheets

The pandas package can import data from Excel workbooks. Both `xls` and `xlsx` formats are supported.

```
# Import data from an Excel workbook
Salaries_excel = pd.read_excel("salaries.xlsx",sheet_name = 0)
```

Since workbooks can have more than one worksheet, you can specify the one you want with the `sheet_name` option. The default is `sheet_name = 0`.

1.1.3 Statistical packages

The pandas package provides functions for importing data from a variety of statistical packages.

```
# Import data from Stata
Salaries_stata = pd.read_stata("salaries.dta")

# Import data from SPSS
Salaries_spss = pd.read_spss("salaries.sav")

# Import data from SAS
Salaries_sas = pd.read_sas("salaries.sas7bdat")
```

1.1.4 Databases

Importing data from a database requires additional steps. Depending on the database containing the data, the following packages can help : cantools and pyodbc for DBC file, `pandas.read_sql` for SQL file.

1.2 Cleaning data

The processes of cleaning your data can be the most time-consuming part of any data analysis. The most important steps are considered below. While there are many approaches, those using dfply and plydata packages are some of the quickest and easiest to learn.

Table 1.1: Example function available in plydata

Function	Use
`arrange`	Sort rows by column variables
`create`	Create DataFrame with columns
`define`	Add column to DataFrame
`distinct`	Select distinct/unique riws
`do`	Do arbitrary operations on dataframe
`group_by`	Group dataframe by one or more columns/variables
`group_indices`	Generate a unique id for each group
`head`	Select the top n rows
`mutate`	Alias of plydata.one_table_verbs.define
`pull`	Pull a single column from the dataframe
`query`	Return rows with matching conditions
`rename`	Rename columns
`sample_frac`	Sample a fraction of rows from dataframe
`sample_n`	Sample nn rows from dataframe
`select`	Rows columns
`slice_rows`	Select rows
`summarize`	Summarise multiple values to a single value
`tail`	Select the bottom n rows
`transmute`	alias of plydata.one_table_verbs.create
`ungroup`	Remove the grouping variables for dataframe
`unique`	alias of plydata.one_table_verbs.distinct

You can download the plydata documentation.

For example, in this section will use the starwars dataset from the `datar` package. The dataset provides descriptions of 87 characters from the Starwars universe on 13 variables.

1.2.1 Selecting variables

The select function allows you to limit your dataset to specified variables (columns).

```
from plydata import select

# load data
from datar.datasets import starwars

# Keep the variables name, height and gender
newdata = select(starwars, "name", "height", "gender")
newdata.head()
```

```
##              name    height     gender
##          <object> <float64>   <object>
## 0  Luke Skywalker     172.0  masculine
## 1           C-3PO     167.0  masculine
## 2           R2-D2      96.0  masculine
## 3     Darth Vader     202.0  masculine
## 4     Leia Organa     150.0   feminine
```

To use column name in slice : `names = slice("col2", "col5")`.

```
# keep the variables name and all variables between mass and species inclusive
newdata = select(starwars,"name",slice("mass","species"))
newdata.head()
```

```
##              name      mass hair_color  ...     gender homeworld  species
##          <object> <float64>   <object>  ...   <object>  <object> <object>
## 0  Luke Skywalker      77.0      blond  ...  masculine  Tatooine    Human
## 1           C-3PO      75.0        NaN  ...  masculine  Tatooine    Droid
## 2           R2-D2      32.0        NaN  ...  masculine     Naboo    Droid
## 3     Darth Vader     136.0       none  ...  masculine  Tatooine    Human
## 4     Leia Organa      49.0      brown  ...   feminine  Alderaan    Human
##
## [5 rows x 10 columns]
```

You can exclude columns by prepending `-`.

```
# keep all variables except birth_year and gender
newdata = select(starwars,"-birth_year","-gender")
print(newdata.head())
```

```
##              name    height      mass  ...      sex homeworld  species
##          <object> <float64> <float64>  ... <object>  <object> <object>
```

```
## 0  Luke Skywalker     172.0      77.0  ...     male  Tatooine    Human
## 1            C-3PO     167.0      75.0  ...     none  Tatooine    Droid
## 2            R2-D2      96.0      32.0  ...     none     Naboo    Droid
## 3      Darth Vader     202.0     136.0  ...     male  Tatooine    Human
## 4      Leia Organa     150.0      49.0        female  Alderaan    Human
##
## [5 rows x 9 columns]
```

1.2.2 Selecting observations

The query function allows you to limit your dataset to observations (rows) meeting a specific criteria. Multiple criteria can be combined with the `&` (and) and `|` (or) symbols.

```
# Select females
from plydata import query

newdata = query(starwars, "gender == 'feminine'")
newdata.head()
```

```
##                    name    height      mass  ...    gender  homeworld  species
##                <object> <float64> <float64>  ...  <object>   <object> <object>
## 4           Leia Organa     150.0      49.0  ...  feminine   Alderaan    Human
## 6    Beru Whitesun lars     165.0      75.0  ...  feminine   Tatooine    Human
## 26           Mon Mothma     150.0       NaN  ...  feminine  Chandrila    Human
## 40       Shmi Skywalker     163.0       NaN  ...  feminine   Tatooine    Human
## 43          Ayla Secura     178.0      55.0       feminine     Ryloth  Twi'lek
##
## [5 rows x 11 columns]
```

```
# select females that are from Alderaan
newdata = query(starwars, "gender == 'feminine' & homeworld == 'Alderaan'")
newdata
```

```
##            name    height      mass  ...    gender homeworld  species
##        <object> <float64> <float64>  ...  <object>  <object> <object>
## 4  Leia Organa     150.0      49.0       feminine  Alderaan    Human
##
## [1 rows x 11 columns]
```

```
# select individuals that are from Alderaan, Coruscant, or Endor
newdata = query(starwars,
("homeworld == 'Alderaan' | homeworld == 'Coruscant' | homeworld == 'Endor'"))
newdata.head()
```

```
##                     name    height      mass  ...     gender  homeworld     species
##                 <object> <float64> <float64>  ...   <object>   <object>    <object>
## 4             Leia Organa     150.0      49.0  ...   feminine   Alderaan       Human
## 28  Wicket Systri Warrick      88.0      20.0  ...  masculine      Endor        Ewok
## 32          Finis Valorum     170.0       NaN  ...  masculine  Coruscant       Human
## 51             Adi Gallia     184.0      50.0  ...   feminine  Coruscant  Tholothian
```

```
## 64    Bail Prestor Organa     191.0       NaN       masculine   Alderaan       Human
##
## [5 rows x 11 columns]

# this can be written more succinctly as
newdata = query(starwars, 'homeworld in ["Alderaan", "Coruscant", "Endor"]')
newdata.head()
```

```
##                     name    height      mass  ...     gender  homeworld     species
##                 <object> <float64> <float64>  ...   <object>   <object>    <object>
## 4            Leia Organa     150.0      49.0  ...   feminine   Alderaan       Human
## 28  Wicket Systri Warrick      88.0      20.0  ...  masculine      Endor        Ewok
## 32          Finis Valorum     170.0       NaN  ...  masculine  Coruscant       Human
## 51             Adi Gallia     184.0      50.0  ...   feminine  Coruscant  Tholothian
## 64    Bail Prestor Organa     191.0       NaN       masculine   Alderaan       Human
##
## [5 rows x 11 columns]
```

1.2.3 Creating/recoding variables

The define function allows you to create new variables or transform existing ones.

```
from plydata import define

# convert height in centimeters to inches, and mass in kilograms to pounds
newdata = define(starwars, height = "height * 0.394", mass   = "mass * 2.205")
newdata.head()
```

```
##              name    height      mass  ...     gender homeworld  species
##          <object> <float64> <float64>  ...   <object>  <object> <object>
## 0  Luke Skywalker    67.768   169.785  ...  masculine  Tatooine    Human
## 1           C-3PO    65.798   165.375  ...  masculine  Tatooine    Droid
## 2           R2-D2    37.824    70.560  ...  masculine     Naboo    Droid
## 3     Darth Vader    79.588   299.880  ...  masculine  Tatooine    Human
## 4     Leia Organa    59.100   108.045        feminine  Alderaan    Human
##
## [5 rows x 11 columns]

# Using mutate
from plydata.one_table_verbs import mutate

newdata = mutate(starwars, height = "height * 0.394", mass   = "mass * 2.205")
newdata.head()
```

```
##              name    height      mass  ...     gender homeworld  species
##          <object> <float64> <float64>  ...   <object>  <object> <object>
## 0  Luke Skywalker    67.768   169.785  ...  masculine  Tatooine    Human
## 1           C-3PO    65.798   165.375  ...  masculine  Tatooine    Droid
## 2           R2-D2    37.824    70.560  ...  masculine     Naboo    Droid
## 3     Darth Vader    79.588   299.880  ...  masculine  Tatooine    Human
```

```
## 4     Leia Organa    59.100   108.045        feminine  Alderaan    Human
##
## [5 rows x 11 columns]
```

The `if_else` function can be used for recoding data. The format is `if_else(test, return if True, return if False)`.

```
from plydata import if_else

# if height is greater than 180
# then heightcat = "tall",
# otherwise heightcat = "short"

newdata = define(starwars,heightcat=if_else("height > 180",'"taill"','"short"'))
newdata.head()
```

```
##              name    height      mass  ... homeworld  species heightcat
##          <object> <float64> <float64>  ...  <object> <object>  <object>
## 0  Luke Skywalker     172.0      77.0  ...  Tatooine    Human     short
## 1           C-3PO     167.0      75.0  ...  Tatooine    Droid     short
## 2           R2-D2      96.0      32.0  ...     Naboo    Droid     short
## 3     Darth Vader     202.0     136.0  ...  Tatooine    Human     taill
## 4     Leia Organa     150.0      49.0        Alderaan    Human     short
##
## [5 rows x 12 columns]
```

1.2.4 Summarizing data

The summarize function can be used to reduce multiple values down to a single value (such as a mean).

```
from plydata import summarize
import numpy as np

# Calculate mean height and mass
newdata = summarize(starwars, mean_ht = "np.nanmean(height)",
                              mean_mass = "np.nanmean(mass)")
```

Table 1.2: mean height and mass

mean_ht	mean_mass
174.358	97.31186

It is often used in conjunction with the group_by function, to calculate statistics by group.

```
from plydata import group_by

# Calculate mean height and weight by gender
newdata = group_by(starwars, "gender")
newdata = summarize(newdata,mean_ht = "np.nanmean(height)",
                            mean_wt = "np.nanmean(mass)")
```

Table 1.3: mean height and mass by gender

gender	mean_ht	mean_wt
masculine	176.5161	106.14694
feminine	164.6875	54.68889

1.2.5 Using pipes

Packages like `plydata` allow you to write your code in a compact format using the pipe `>>` operator. Here is an example

```
# Calculate the mean height by gender
newdata = group_by(starwars, "gender")
newdata = summarize(newdata, mean_ht = "np.nanmean(height)",
                             mean_wt = "np.nanmean(mass)")
```

Table 1.4: mean height and mass by gender

gender	mean_ht	mean_wt
masculine	176.5161	106.14694
feminine	164.6875	54.68889

```
# This can be written as
newdata = (starwars >>
  group_by("gender") >>
  summarize(mean_ht = "np.nanmean(height)",mean_wt = "np.nanmean(mass)"))
```

Table 1.5: mean height and mass by gender

gender	mean_ht	mean_wt
masculine	176.5161	106.14694
feminine	164.6875	54.68889

The `>>` operator passes the result on the left to the first parameter of the function on the right.

1.2.6 Reshaping data

Some graphs require the data to be in wide format, while some graphs require the data to be in long format.

```
# Create a dataframe
import pandas as pd

df = pd.DataFrame({
  "id" : ["01", "02", "03"],
  "name" : ["Bill", "Bob", "Mary"],
  "sex" : ["Male", "Male", "Female"],
  "age" :  [22, 25, 18],
  "income" : [55000, 75000, 90000]
})
```

Table 1.6: Wide data

id	name	sex	age	income
01	Bill	Male	22	55000
02	Bob	Male	25	75000
03	Mary	Female	18	90000

You can convert a wide dataset to a long dataset using

```
# Long data
long_data = df.melt(id_vars=['id','name'],var_name='variable',value_name='value')
```

Table 1.7: Long data

id	name	variable	value
01	Bill	sex	Male
02	Bob	sex	Male
03	Mary	sex	Female
01	Bill	age	22
02	Bob	age	25
03	Mary	age	18
01	Bill	income	55000
02	Bob	income	75000
03	Mary	income	90000

Conversely, you can convert a long dataset to a wide dataset using

```
# Wide data
wide_data = (long_data
             .pivot(index=['id','name'], columns='variable', values='value')
             .reset_index())
```

Table 1.8: Wide data

id	name	age	income	sex
01	Bill	22	55000	Male
02	Bob	25	75000	Male
03	Mary	18	90000	Female

1.2.7 Missing data

Real data are likely contain missing values. There are three basic approaches to dealing with missing data : feature selection, listwise delection, and imputation. Let's see how each applies to the `msleep` dataset from `datar` package. The `msleep` dataset describes the sleep habits of mammals and contains missing values on several variables.

1.2.7.1 Feature selection

In feature selection, you delete variables (columns) that contain too many missing values.

```
from datar.datasets import msleep

# what is the proportion of missing data for each variable?
pctmiss = msleep.isnull().mean().to_frame("pct. of NA")
```

Table 1.9: Percentage of missing values

	pct. of NA
name	0.0000000
genus	0.0000000
vore	0.0843373
order	0.0000000
conservation	0.3493976
sleep_total	0.0000000
sleep_rem	0.2650602
sleep_cycle	0.6144578
awake	0.0000000
brainwt	0.3253012
bodywt	0.0000000

Sixty-one percent of the sleep_cycle values are missing. You may decide to drop it.

1.2.7.2 Listwise deletion

Listwise delection involves deleting observations (rows) that contain missing values on any of the variables of interest.

```
# Create a dataset containing genus, vore, and conservation.
# Delete any rows containing missing data.
newdata = select(msleep, "genus", "vore", "conservation")
newdata = newdata.dropna()
newdata.isnull().mean().round(2)
```

```
## genus           0.0
## vore            0.0
## conservation    0.0
## dtype: float64
```

1.2.7.3 Imputation

Imputation involves replacing missing values with "reasonable" guesses about what the values would have been if they had not been missing. There are several approaches, as detailed in sklearn.

```
# Set index
df = msleep.set_index('name')
# Select numerical variable
num_df = df.select_dtypes(include = np.number)
num_df.isnull().mean().round(2)
```

```
## sleep_total    0.00
## sleep_rem      0.27
```

```
## sleep_cycle    0.61
## awake          0.00
## brainwt        0.33
## bodywt         0.00
## dtype: float64
```

```
# Impute missing values using the 5 nearest neighbors
from sklearn.impute import KNNImputer

imputer = KNNImputer(n_neighbors=5)
imputer.set_output(transform="pandas");
num_df = imputer.fit_transform(num_df)
num_df.isnull().mean().round(2)
```

```
## sleep_total    0.0
## sleep_rem      0.0
## sleep_cycle    0.0
## awake          0.0
## brainwt        0.0
## bodywt         0.0
## dtype: float64
```

For categorical variables, we can use sklearn.impute.SimpleImputer using `strategy = "most_frequent"` : this will replace missing values using the most frequent value along each column; no matter if they are strings or numeric data.

```
# Select no numerical variable
cat_df = df.select_dtypes(exclude = np.number)
cat_df.isnull().mean().round(2)
```

```
## genus          0.00
## vore           0.08
## order          0.00
## conservation   0.35
## dtype: float64
```

```
# Replace missing value with most frequent
from sklearn.impute import SimpleImputer

imp = SimpleImputer(missing_values=np.nan, strategy='most_frequent')
imp.set_output(transform="pandas");
cat_df = imp.fit_transform(cat_df)
cat_df.isnull().mean().round(2)
```

```
## genus          0.0
## vore           0.0
## order          0.0
## conservation   0.0
## dtype: float64
```

2 Introduction to plotnine

This chapter provides a brief overview of how the plotnine package works. If you are simply seeking code to make a specific type of graph, feel free to skip this section. However, the material can help you understand how the pieces fit together.

2.1 A worked example

The functions in the `plotnine` package build up a graph in layers. We'll build a complex graph by starting with a simple graph and adding additionnal elements, one at a time.

The example uses data from 1985 Current Population Survey to explore the relationship between (`wage`) and (`exper`). It is an R dataset.

```
# load data
data(CPS85, package = "mosaicData")
```

In building a `plotnine` graph, only the first two functions described below are required. The other functions are optional and can appear in any order.

2.1.1 ggplot

The first function in building a graph is the `ggplot` function. It specifies the :

- data frame containing the data to be plotted
- the mapping of the variables to visual properties of the graph. The mappings are placed within the `aes` function (where *aes* stands for aesthetics).

```
# Specify dataset and mapping
from plotnine import *

print((ggplot(data =  r.CPS85,mapping = aes(x = "exper", y = "wage"))))
```

Why is the graph empty? We specified that the *expr* variable should be mapped to the x-axis and that the *wage* should be mapped to the y-axis, but we haven't yet specified what we wanted placed on the graph.

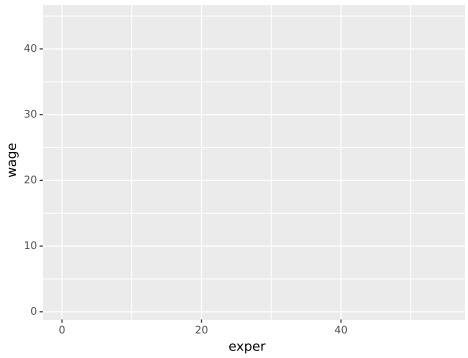


Figure 2.1: Map variables

2.1.2 geoms

Geoms are the geometric objects (points, lines, bars, etc.) that can be placed on a graph. They are added using functions that start with `geom_`. In this example, we'll add points using the `geom_point` function, creating a scatterplot.

In `plotnine` graphs, functions are chained together using the `+` sign to build a final plot.

```
# add points
print((ggplot(data =  r.CPS85,mapping = aes(x = "exper", y = "wage"))+
       geom_point()))
```

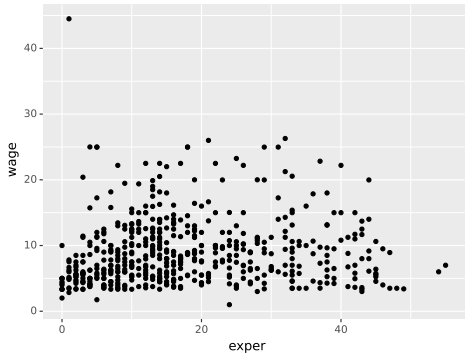


Figure 2.2: Add points

The graph indicates that there is an outlier. One individual has a wage much higher than the rest. We'll delete this case before continuing.

```
# delete outlier
from plydata import query
plotdata = query(r.CPS85,"wage < 40")

# redraw scatterplot
print((ggplot(data = plotdata, mapping = aes(x = "exper", y = "wage"))+
       geom_point()))
```

A number of parameters (options) can be specified in a `geom_` function.Options for the `geom_point` function include `color`, `size` and `alpha`. These control the point color, size

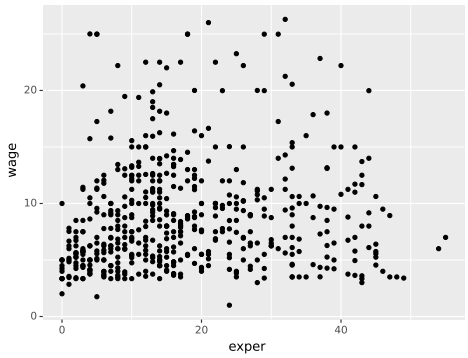


Figure 2.3: Remove outlier

and transparency, respectively. Transparency ranges from 0 (completely transparent) to 1 (completely opaque). Adding a degree of transparency can help visualize overlapping points.

```
# Make points blue, larger, and semi-transparent
print((ggplot(data = plotdata, mapping=aes(x = "exper", y = "wage"))+
       geom_point(color = "cornflowerblue", alpha = 0.7, size = 3)))
```

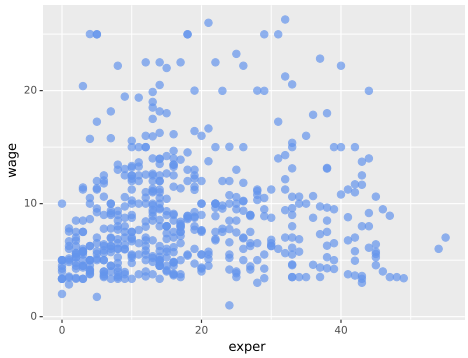


Figure 2.4: Modify point color, transparency, and size

Next, let's add a line of the best fit. We can do this with the `geom_smooth` function.Options control the type of line (linear, quadratic, non parametric), the thickness of the line, the line's color, and the presence or absence of a confidence interval. Here we request a linear regression (`method = lm`) line (where *lm* stands for linear model).

```
# add a line of best fit
print((ggplot(data = plotdata, mapping=aes(x = "exper", y = "wage"))+
       geom_point(color = "cornflowerblue", alpha = 0.7, size = 3)+
       geom_smooth(method = "lm")))
```

Wages appears to increase with experience.

2.1.3 grouping

In addition to mapping variables to the x and y axes, variables can be mapped to the color, shape, size, transparency, and other visual characteristics of geometric objects. This allows group of observations to be superimposed in a single graph.

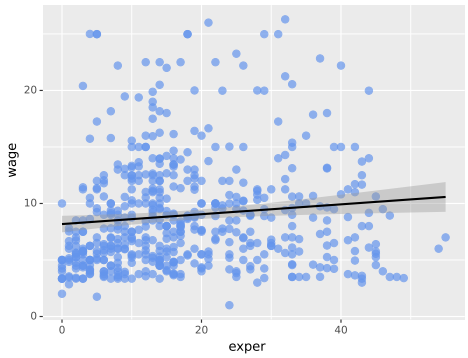


Figure 2.5: Add line of best fit

Let's add `sex` to the plot and represent it by color.

```
# Indcate sex using color
print((ggplot(data = plotdata,
              mapping = aes(x = "exper", y = "wage", color = "sex"))+
       geom_point(alpha = 0.7, size = 3)+
       geom_smooth(method = "lm", se = False, size = 1.5)+
       theme(legend_position=(0.85,0.8),legend_direction='vertical')))
```

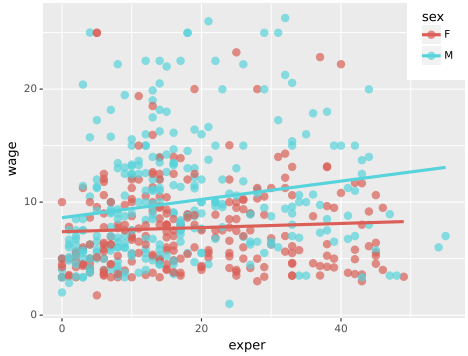


Figure 2.6: Include sex, using color

The `color ="sex"` option is placed in the `aes` function, because we are mapping a variable to an aesthetic. The `geom_smooth` option (`se = False`) was added to suppresses the confidence intervals.

It appears that men tend to make more money than women. Additionally, there may be a stronger relationship between experience and wages for men than for women.

2.1.4 scales

Scales control how variables are mapped to the visual characteristics of the plot. Scale functions (which start with `scale_`) allow you to modify this mapping. In the next plot, we'll change the x and y scaling, and the colors employed.

```
# modify the x and y axes and specify the colors to be used
from mizani.formatters import currency_format

print((ggplot(data = plotdata,
              mapping = aes(x = "exper", y = "wage",color = "sex")) +
       geom_point(alpha = .7,size = 3) +
       geom_smooth(method = "lm", se = False,size = 1.5) +
       scale_x_continuous(breaks = range(0,60,10)) +
       scale_y_continuous(breaks=range(0,30,5),labels=currency_format()) +
       scale_color_manual(values = ["indianred","cornflowerblue"])+
       theme(legend_position=(0.85,0.8),legend_direction='vertical')))
```

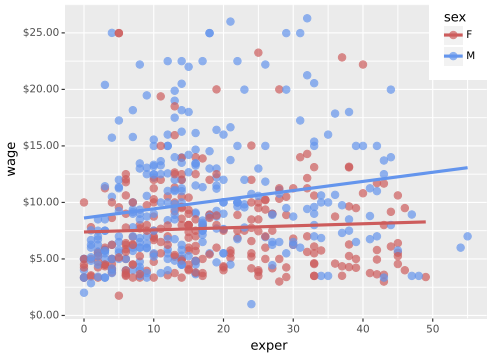


Figure 2.7: Change colors and axis labels

We're getting. The numbers on the x and y axes are better, the y axis uses dollar notation, and the colors are more attractive (IMHO).

Here is a question. Is the relationship between experience, wages and sex the same for each job sector? Let's repeat this graph once for each job sector in order to explore this.

2.1.5 facets

Facets reproduce a graph for each level a given variable (or combination of variables). Facets are created using functions that start with `facet_`. Here, facets will be defined by the eight levels of the sector variable.

```
# reproduce plot for each level of job sector
print((
  ggplot(data = plotdata,mapping = aes(x = "exper", y = "wage",color = "sex")) +
    geom_point(alpha = 0.7) +
    geom_smooth(method = "lm", se = False) +
    scale_x_continuous(breaks = range(0, 60, 10)) +
    scale_y_continuous(breaks = range(0, 30, 5),labels = currency_format()) +
    scale_color_manual(values = ["indianred", "cornflowerblue"]) +
    facet_wrap("sector")+
    theme(legend_position=(0.85,0.2),legend_direction='vertical')))
```

It appears that the differences between mean and women depend on the job sector under consideration.

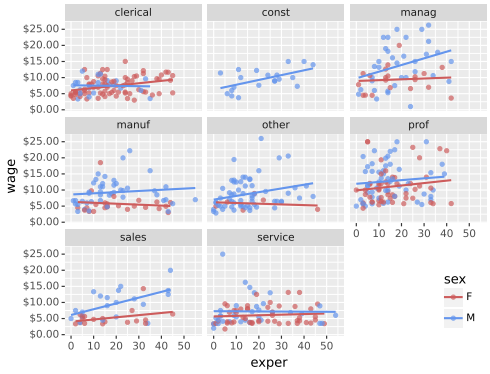


Figure 2.8: Add job sector, using faceting

2.1.6 labels

Graphs should be easy to interpret and informative labels are a key element in achieving this goal. The labs function provides customized labels for the axes and legends. Additionally, a custom title and caption can be added.

```
# add informative labels
print((
  ggplot(data = plotdata,mapping = aes(x = "exper", y = "wage",color = "sex")) +
    geom_point(alpha = 0.7) +
    geom_smooth(method = "lm", se = False) +
    scale_x_continuous(breaks = range(0, 60, 10)) +
    scale_y_continuous(breaks = range(0, 30, 5),labels = currency_format()) +
    scale_color_manual(values = ["indianred", "cornflowerblue"]) +
    facet_wrap("sector")+
    labs(title = "Relationship between wages and experience",
         caption = "source: http://mosaic-web.org/",
         x = " Years of Experience",y = "Hourly Wage",color = "Gender")+
    theme(legend_position=(0.85,0.2),legend_direction='vertical')
))
```

Figure 2.9: Add informative titles and labels.

2.1.7 themes

Finally, we can fine the appearance of the graph using themes. Themes functions (which start with `theme_`) control background colors, fonts, grid-lines, legend placement, and other non-data related features of the graph. Let's use a cleaner theme.

```
# use a minimalist theme
print((
  ggplot(data = plotdata,mapping = aes(x = "exper", y = "wage",color = "sex")) +
    geom_point(alpha = 0.7) +
    geom_smooth(method = "lm", se = False) +
    scale_x_continuous(breaks = range(0, 60, 10)) +
    scale_y_continuous(breaks = range(0, 30, 5),labels = currency_format()) +
    scale_color_manual(values = ["indianred", "cornflowerblue"]) +
    facet_wrap("sector")+
    labs(title = "Relationship between wages and experience",
         caption = "source: http://mosaic-web.org/",
         x = " Years of Experience",y = "Hourly Wage",color = "Gender")+
    theme_minimal()+
    theme(legend_position=(0.85,0.2),legend_direction='vertical')
))
```

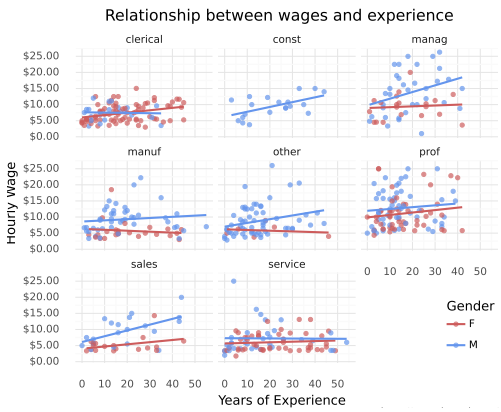


Figure 2.10: Use a simpler theme

Now we have something. It appears that men earn more than women in management, manufacturing, sales, and the "other" category. They are most similar in clerical, professional, and service positions. The data contain no women in the construction sector. For management positions, wages appear to be related to experience for men, but not for women (this may be the most interesting finding). This also appears to be true for sales.

Of course, these findings are tentative. They are based on a limited sample size and do not involve statistical testing to assess whether differences may be due to chance variation.

2.2 Placing the `data` and `mapping` options

Plots created with `plotnine` always start with the `ggplot` function. In the example above, the `data` and `mapping` options were placed in this function. In this case they apply to each `geom_` function that follows.

You can also place these options directly within a `geom`. In that case, they only apply only to that specific geom.

Consider the following graph.

```
# Create poly function
def poly(x,p):
  return x**p

# placing color mapping in the ggplot function
print((
  ggplot(plotdata,aes(x = "exper", y = "wage",color = "sex")) +
  geom_point(alpha = .7,size = 3) +
  geom_smooth(method = "lm",formula = "y ~ x+poly(x,2)", se = False,size = 1.5)+
  theme(legend_position=(0.85,0.8),legend_direction='vertical')))
```

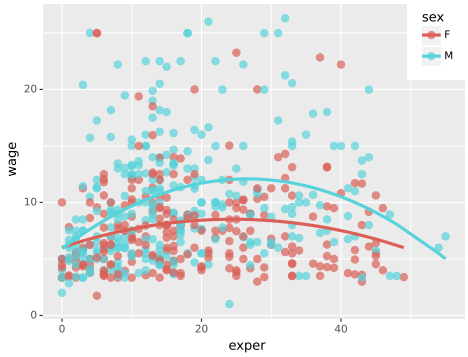


Figure 2.11: Color mapping in plotnine function

Since the mapping of sex to color appears in the `ggplot` function, it applies to both `geom_point` and `geom_smooth`. The color of the point indicates the sex, and a separate colored trend line is produce for men and women. Compare to this :

```
# placing color mapping in the geom_point function
print((
  ggplot(plotdata,aes(x = "exper", y = "wage")) +
  geom_point(aes(color = "sex"),alpha = .7,size = 3) +
  geom_smooth(method = "lm",formula = "y ~x+poly(x,2)",se = False,size = 1.5)+
  theme(legend_position=(0.85,0.8),legend_direction='vertical')))
```

Since the sex to color mapping only appears in the `geom_point` function, it is only used there. A single trend line is created for all observations.

Most of the examples in this book place the data and mapping options in the `ggplot` function. Additionally, the phrases *data* = and *mapping*= are omitted since the first option always refers to data and the second option always refers to mapping.

2.3 Graphs as objects

A `plotnine` graph can be saved as a named Python object (like a data frame), manipulated further, and then printed or saved to disk.

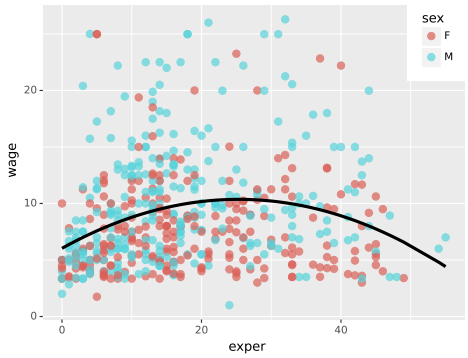


Figure 2.12: Color mapping in plotnine function

```
# Prepare data
plotdata = r.CPS85 >> query('wage < 40')

# Create scatterplot and save it
myplot = (ggplot(data = plotdata, mapping = aes(x = "exper", y = "wage"))+
          geom_point())

# print the graph
print(myplot)
```

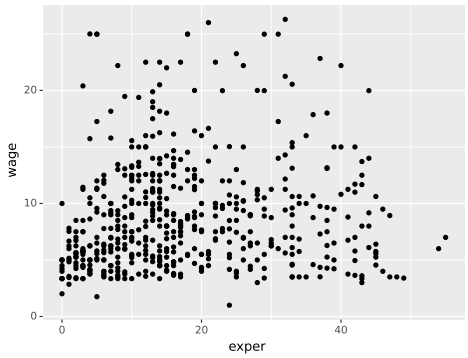


Figure 2.13: Basic scatterplot

```
# make the points larger and blue then print the graph
myplot = myplot + geom_point(size = 3, color = "blue")
print(myplot)


# print the graph with a title and line of best fit
# but don't save those changes
print((myplot + geom_smooth(method = "lm")+
                labs(title = "Mildly interesting graph")))
```

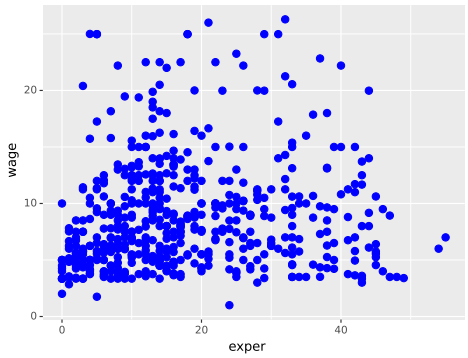


Figure 2.14: Add color abd size

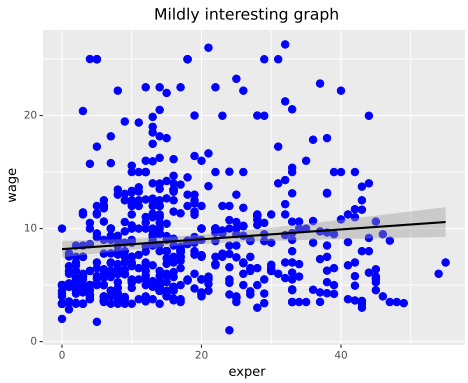


Figure 2.15: Add linear regression

```
# print the graph with a black and white theme
# but don't save those changes
print((myplot + theme_bw()))
```

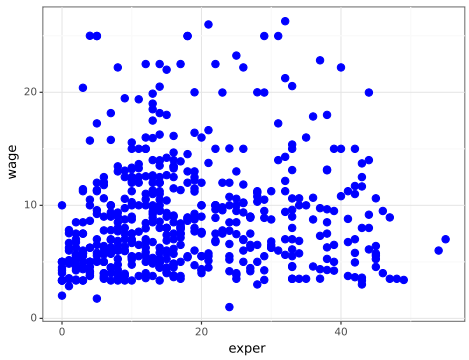


Figure 2.16: Add theme

Now it's time to try out other types of graphs.

3 Univariate Graphs

Univariate graphs plot the distribution of data from a single variable. The variable can be categorical (e.g., race, sex) or quantitative (e.g., age, weight).

3.1 Categorical

The distribution of a single categorical variable is typically plotted with a bar chart, a pie chart,or (less commonly) a tree map.

3.1.1 Bar chart

The Marriage dataset contains the marriage records of 98 individuals in Mobile County, Alabama. Below, a bar chart is used to display the distribution of wedding participants by race.

```
# load Marriage dataset
data(Marriage, package = "mosaicData")
```

```
# plot the distribution of race
from plotnine import *
print((r.Marriage >> ggplot(aes(x = "race"))+geom_bar()))
```

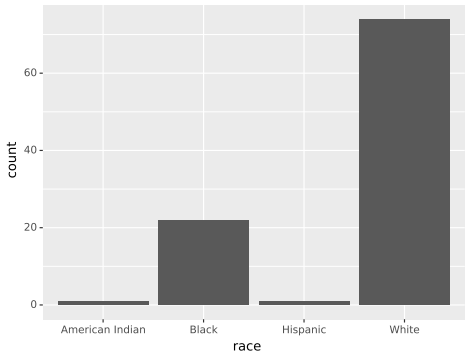


Figure 3.1: Simple barchart

The majority of participants are white, followed by black, with very few Hispanics or American Indians.

You can modify the bar fill and border colors, plot labels, and title by adding options to the geom_bar function.

```
# plot the distribution of race with modified colors and labels
print((
  r.Marriage >>
  ggplot(aes(x = "race")) +
  geom_bar(fill = "cornflowerblue",color="black") +
  labs(x = "Race",y = "Frequency",title = "Participants by race")
))
```

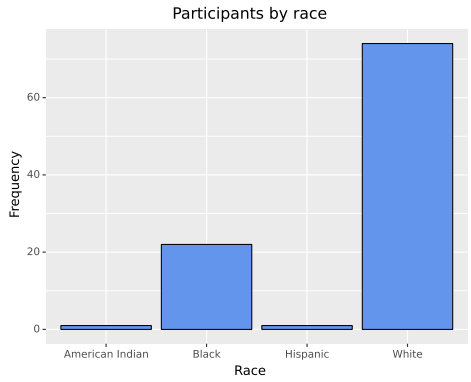


Figure 3.2: Barchart with modified colors, labels, and title

3.1.1.1 Percents

Bars can represent percents rather than counts. For bar charts, the `aes(x = "trans")` is actually a shortcut for `aes(x ="trans", y = "..count..")`, where `"..count.."` is a special variable representing the frequency within each category.

```
# plot the distribution as percentages
from mizani.formatters import percent_format

print((r.Marriage >> ggplot(aes(x = "race",y = "..count.. / sum(..count..)")) +
       geom_bar() +
       labs(x = "Race", y = "Percent",title  = "Participants by race") +
       scale_y_continuous(labels = percent_format())))
```

In the code above, the mizani package is used to add % symbols to the y - axis labels.

3.1.1.2 Sorting categories

It is often helpful to sort the bars by frequency. In the code below, the frequencies are calculated explicitly. Then the `reorder` function is used to sort the categories by the frequency. The option `stat="identity"` tells the plotting function not to calculate counts, because they are supplied directly.

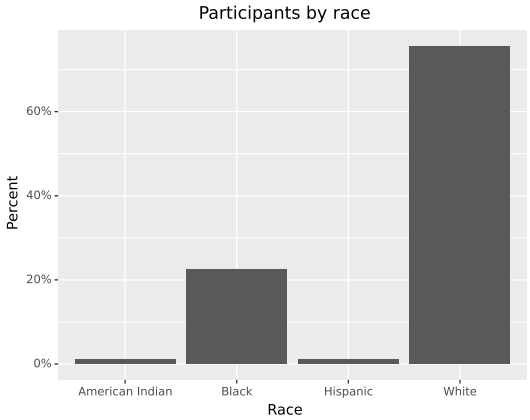


Figure 3.3: Barchart with percentages

```
# calculate number of participants in
# each race category
from plydata import *

plotdata = r.Marriage >> count("race")
```

The resulting dataset is give below.

Table 3.1: plotdata

race	n
White	74
Hispanic	1
Black	22
American Indian	1

The new dataset in then used to create the graph.

```
# plot the bars in ascending order
print((plotdata >> ggplot(aes(x='reorder(race,n)', y ="n")) +
       geom_bar(stat = "identity") +
       labs(x ="Race",y ="Frequency",title = "Participants by race")))
```

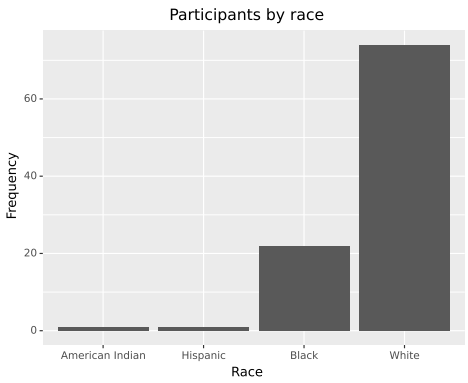


Figure 3.4: Sorted bar chart

The graph bars are sorted in ascending order. Use `reorder(race, -n)` to sort in descending order.

3.1.1.3 Labelling bars

Finally, you may want to label each bar with its numerical value.

```
# plot the bars with numeric labels
print((plotdata >> ggplot(aes(x = "race", y = "n")) +
       geom_bar(stat = "identity") +
       geom_text(aes(label = "n"),va= "center") +
       labs(x = "Race", y = "Frequency", title  = "Participants by race")))
```

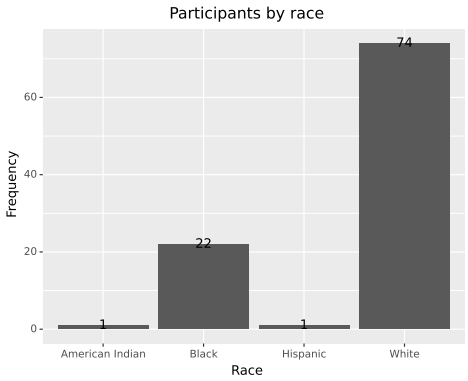


Figure 3.5: Bar chart with numeric labels

Here `geom_text` adds the labels, and `va` controls vertical alignment (one of *top*, *center*, *bottom*, *baseline*)

Putting these ideas together, you can create a graph like the one below. The minus sign in `reorder(race, -n)` is used to order the bars in descending order.

```
plotdata = r.Marriage >> count("race") >> mutate(pct = "n/sum(n)")

from mizani.formatters import percent_format

plotdata.loc[:,"pctlabel"] = percent_format()(plotdata['pct'])

print((ggplot(plotdata, aes(x = "reorder(race, -pct)", y = "pct"))+
  geom_bar(stat = "identity", fill ="indianred", color = "black")+
  geom_text(aes(label ="pctlabel"),va = "center") +
  scale_y_continuous(labels = percent_format()) +
  labs(x = "Race",y = "Percent", title = "Participants by race")))
```

3.1.1.4 Overlapping labels

Category labels may overlap if (1) there are many categories or (2) the labels are long. Consider

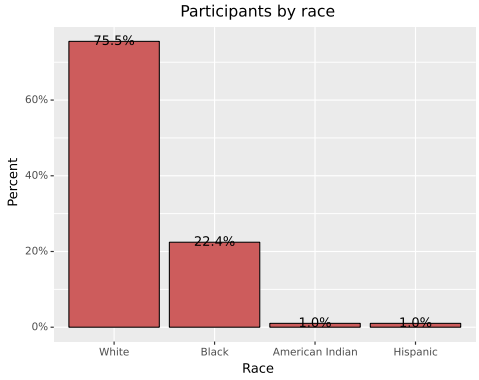


Figure 3.6: Sorted bar chart with percent labels

```
print((r.Marriage >>ggplot(aes(x = "officialTitle"))+geom_bar()+
  labs(x = "OfficialTitle", y = "frequency",title = "Marriages by officiate")))
```

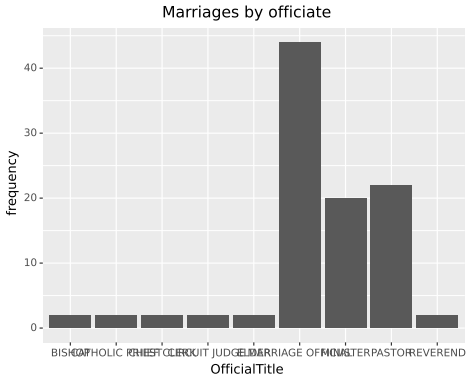


Figure 3.7: Barchart with problematic labels

In this case, you can flip the x and y axes.

```
print((r.Marriage >> ggplot(aes(x = "officialTitle"))+ geom_bar()+
      labs(x = "OfficialTitle", y = "Frequency",title="Marriages by officiate")+
      coord_flip()))
```

Alternatively, you can rotate the axis labels

```
# bar chart with rotated labels
print((r.Marriage >>ggplot(aes(x = "officialTitle"))+geom_bar()+
       labs(x = "", y = "Frequency", title = "Marriages by officiate")+
       theme(axis_text_x=element_text(angle=45, hjust = 1))))
```

Finally, you can try staggering the labels. The trick is to add a newline `\n` to every other label.

```
import textwrap

def str_wrap(x): return textwrap.wrap(x,20)
```

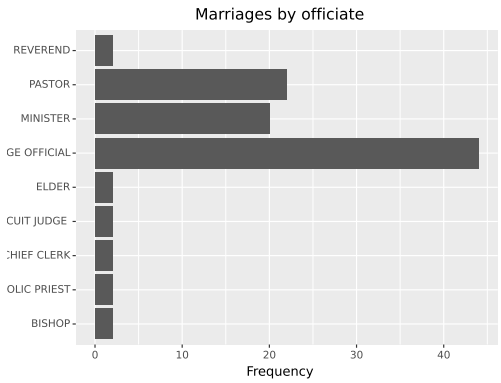


Figure 3.8: Horizontal barchart

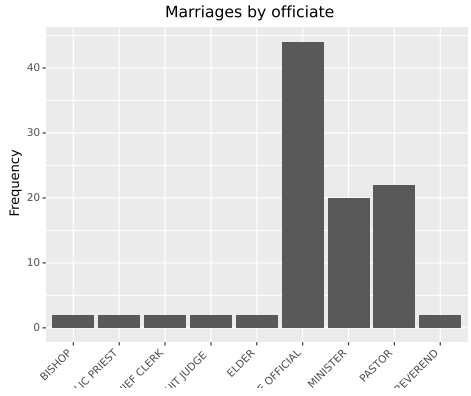


Figure 3.9: Barchart with rotated labels

```
print((r.Marriage >>ggplot(aes(x = "officialTitle"))+geom_bar()+
       labs(x = "", y = "Frequency", title = "Marriages by officiate")+
       theme(axis_text_x=element_text(angle=45, hjust = 1))))
```

3.1.2 Pie chart

Pie charts are controversial in statistics. If your goal is to compare the frequency of categories, you are better off with bar charts (humans are better at judging the length of bars than the volume of pie slices). If your goal is compare each category with the whole (e.g., what portion of participants are Hispanic compared to all participants), and the number of categories is small, then pie charts may work for you.

Unfortunately, plotnine, has not yet ported over the `coord_polar` ggplot layouts. Instead of using `plotnine`, we will used `matplotlib` package.

```
# Create a basic pie chart with matplotlib
import matplotlib.pyplot as plt

#create pie chart
fig, ax = plt.subplots(figsize=(4,3))
ax.pie(plotdata.pct,labels = plotdata.race,shadow = True);
ax.set_title("Marriages by officiate");
plt.show()
```

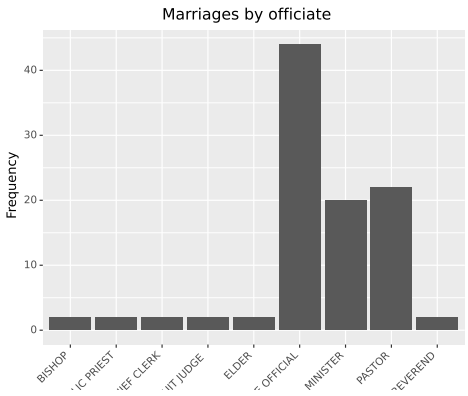


Figure 3.10: Barchart with staggered labels

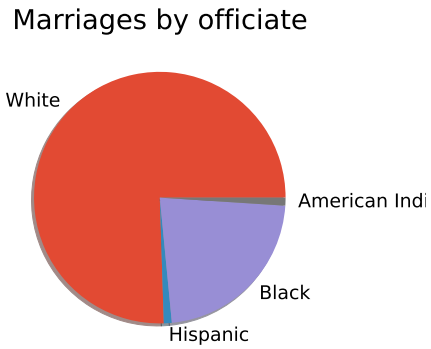


Figure 3.11: Basic pie chart

Now let's get fancy and add labels.

```
#create pie chart
fig, ax = plt.subplots(figsize=(4,3))
ax.pie(plotdata.pct,labels = plotdata.race,autopct='%.0f%%',shadow = True);
ax.set_title("Marriages by officiate");
plt.show()
```

The pie chart makes it easy to compare each slice with the whole.

3.1.3 Tree map

An alternative to a pie chart is a tree map. Unlike pie charts, it can handle categorical variables that have *many* levels.

Plotnine, has not yet ported over the `geom_treemap` ggplot layouts. Instead of using `plotnine`, we will used squarify package.

```
plotdata = r.Marriage >> count("officialTitle")
```

```
import squarify
```

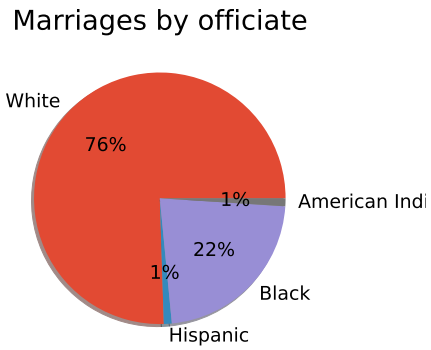


Figure 3.12: Pie chart with percent labels

Table 3.2: plotdata

officialTitle	n
CIRCUIT JUDGE	2
MARRIAGE OFFICIAL	44
MINISTER	20
PASTOR	22
CHIEF CLERK	2
REVEREND	2
ELDER	2
CATHOLIC PRIEST	2
BISHOP	2

```
squarify.plot(sizes=plotdata.n);
plt.title("Marriages by officiate");
plt.axis("off");
plt.show()
```

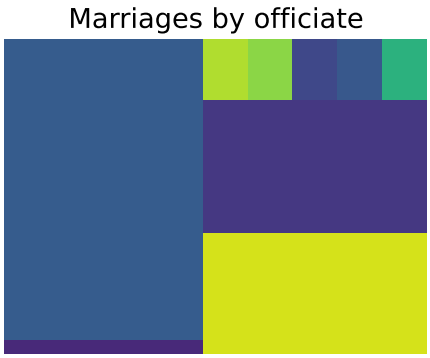


Figure 3.13: Basic treemap

Here is a more useful version with labels.

```
# Add labels
squarify.plot(sizes=plotdata.n,label=plotdata.officialTitle,pad=1,
              text_kwargs={'fontsize': 6});
plt.title("Marriages by officiate");
plt.axis("off");
```

```
plt.show()
```

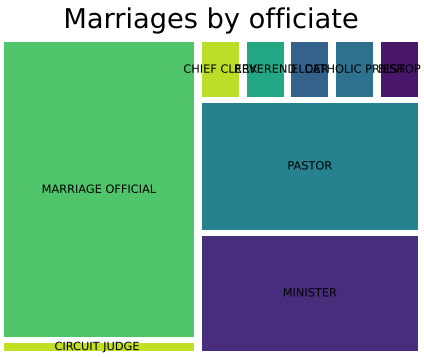


Figure 3.14: Treemap with labels

3.2 Quantitative

The distribution of a single quantitative variable is typically plotted with a histogram, kernel density plot, or dot plot.

3.2.1 Histogram

Using the Marriage dataset, let's plot the mpg.

```
# plot the age distribution using a histogram
print((r.Marriage >>ggplot(aes(x = "age"))+geom_histogram(bins=30)+
       labs(title = "Participants by age",x = "age")))
```

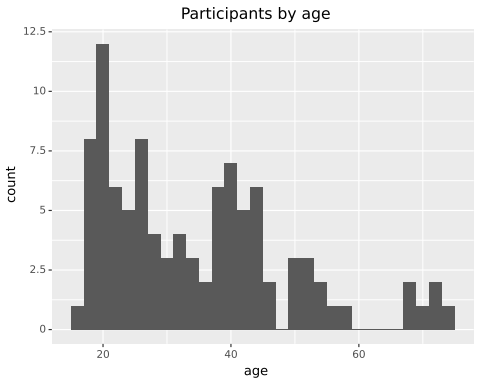


Figure 3.15: Basic histogram

Most participants appear to be in their early 20's with another group in their 40's, and a much smaller group in their later sixties and early seventies. This would be a *multimodal* distribution.

Histogram colors can be modified using two options :

- `fill` - fill color for the bars

- `color` - border color around the bars

```
# plot the histogram with blue bars and white borders
print((r.Marriage >>ggplot(aes(x = "age"))+
       geom_histogram(bins=30,fill = "cornflowerblue", color = "white")+
       labs(title = "Participants by age", x = "age")))
```

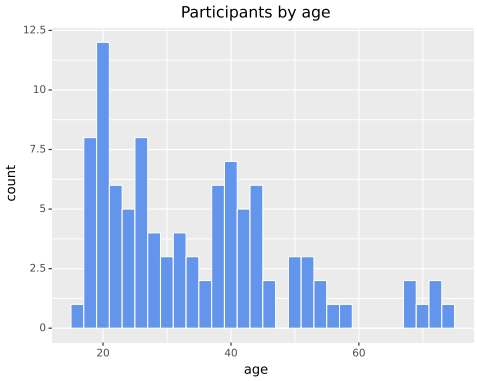


Figure 3.16: Histogram with specified fill and border colors.

3.2.1.1 Bins and bandwidths

One of the most important histogram options is `bins`, which controls the number of bins into which the numeric variable is divided (i.e., the number of bars in the plot). The default is 30, but it is hepful to try smaller and larger numbers to get a better impression of the shape of the distribution.

```
# plot the histogram with 10 bins
print((r.Marriage >> ggplot(aes(x = "age"))+
       geom_histogram(fill = "cornflowerblue", color = "white", bins = 20)+
       labs(title = "Participants by age  (number of bins = 20)", x = "Age")))
```

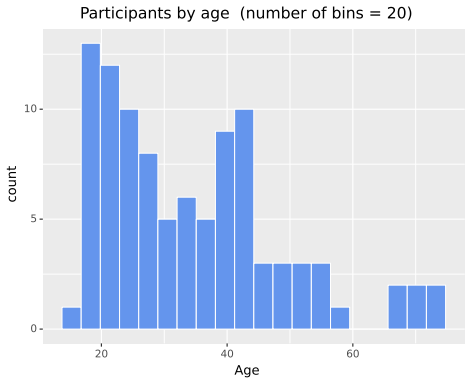


Figure 3.17: Histogram with a specified number of bins.

Alternatively, you can specify the `binwidth`, the width of the bins represented by the bars.

```
# plot the histogram with a binwidth of 5
print((r.Marriage>>ggplot(aes(x = "age"))+
       geom_histogram(fill = "cornflowerblue", color = "white", binwidth = 5)+
       labs(title = "Participants by age (binwidth = 5 years)",x="Age")))
```

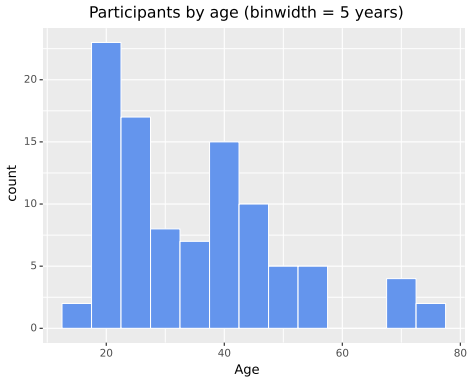


Figure 3.18: Histogram with specified a bin width

As with bar charts, the y-axis can represent counts or percent of the total.

```
# plot the histogram with percentages on the y-axis
print((r.Marriage >> ggplot(aes(x = "age", y = "..count.. / sum(..count..)"))+
    geom_histogram(fill = "cornflowerblue", color = "white", binwidth = 5)+
    labs(title = "Participants by age",y="Percent",x = "Age")+
    scale_y_continuous(labels = percent_format())))
```

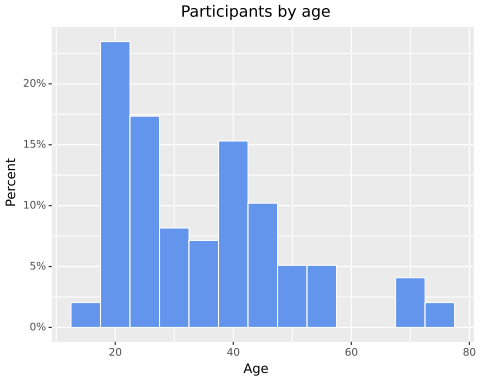


Figure 3.19: Histogram with percentages on the y - axis

3.2.2 Kernel Density plot

An alternative to a histogram is the kernel density plot. Technically, kernel density estimation is a non parametric method for estimating the probability density function of a continuous random variable. (What??) Basically, we are trying to draw a smoothed histogram, where the area under the curve equal one.

```
# Create a kernel density plot of age
print((r.Marriage >> ggplot(aes(x = "age"))+geom_density()+
       labs(title = "Participants by age")))
```

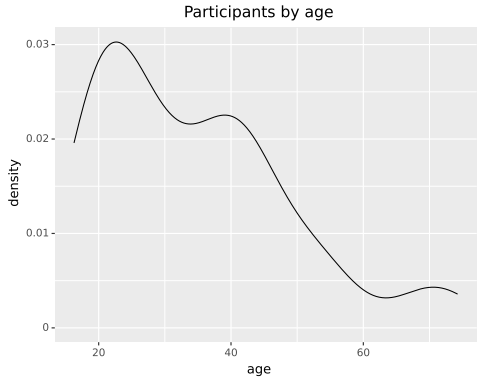


Figure 3.20: Basic kernel density plot

The graph shows the distribution of scores. For example, the proportion of cases between 20 and 40 years old would be represented by the area under the curve between 20 and 40 on the x-axis.

As with previous charts, we can use `fill` and `color` to specify the fill and border colors.

```
print((r.Marriage >> ggplot(aes(x = "age"))+geom_density(fill = "indianred")+
       labs(title = "Participants by age")))
```

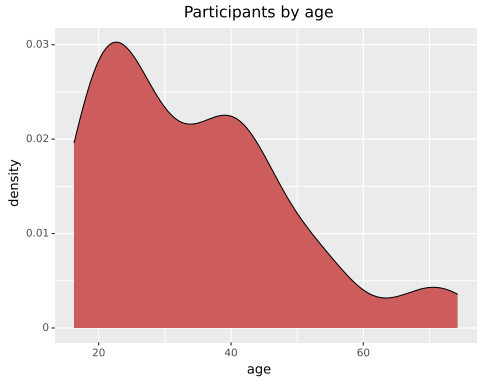


Figure 3.21: Kernel density plot with fill

3.2.2.1 Smoothing parameter

The degree of smoothness is controlled by the bandwidth parameter `bw`. To find the default value for a particular variable, use the `bw_nrd0` function.

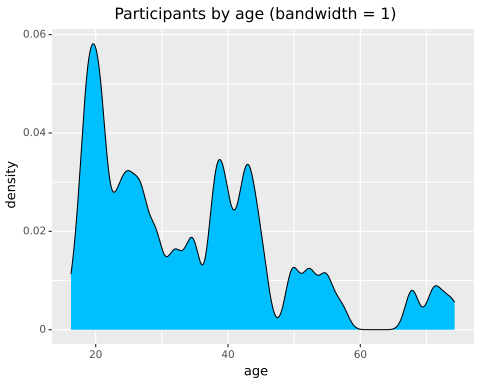
```
# Default bandwidth for the age variable
import numpy as np

def bw_nrd0(x):
```

```
    if len(x) < 2:
        raise(Exception("need at least 2 data points"))

    hi = np.std(x, ddof=1)
    q75, q25 = np.percentile(x, [75 ,25])
    iqr = q75 - q25
    lo = min(hi, iqr/1.34)

    if not ((lo == hi) or (lo == abs(x[0])) or (lo == 1)):
        lo = 1

    return 0.9 * lo *len(x)**-0.2

bw_nrd0(r.Marriage.age)
```

```
## 5.1819460066411365
```

Values that are larger will result in more smoothing, while values that are smaller will produce less smoothong.

```
# Create a kernel density plot of mpg
print((r.Marriage >> ggplot(aes(x = "age")) +
    geom_density(fill = "deepskyblue", bw = 1)+
    labs(title = "Participants by age (bandwidth = 1)")))
```



In this example, the default bandwidth for mpg is 5.18. Choosing a value of 1 resulted in more smoothing.

Kernel density plots allow you to easily see which scores are most frequent and which are relatively rare. However it can be difficult to explain the meaning of the y-axis to a non-statistician. (But it will make you look really smart at parties!).

3.2.3 Dot Chart

Another alternative to the histogram is the dot chart. Again, the quantitative variable is divided into bins, but rather than summary bars, each observation is represented by a dot. By default, the width of a dot corresponds to the bin width, and dots are stacked, with each dot representing one observation. This works best when the number of observations is small (say, less than 150).

```
# plot the age distribution using a dotplot
print((r.Marriage >> ggplot(aes(x = "age"))+geom_dotplot()+
       labs(title = "Participants by age", y = "Proportion", x = "Age")))
```

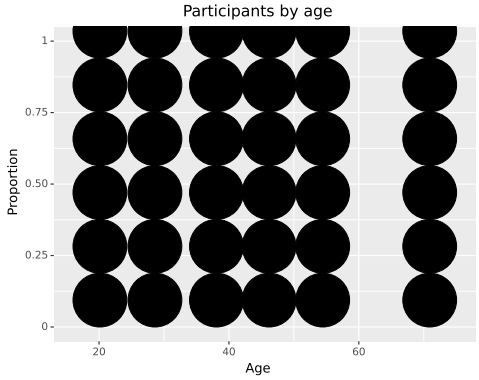


Figure 3.22: Basic dotplot

The `fill` and `color` options can be used to specify the fill and border color of each dot respectively.

```
# Plot ages as a dot plot using
# gold dots with black borders
print((ggplot(r.Marriage,aes(x = "age"))+
        geom_dotplot(fill = "gold", color = "black")+
        labs(title = "Participants by age", y = "Proportion", x = "Age")))
```

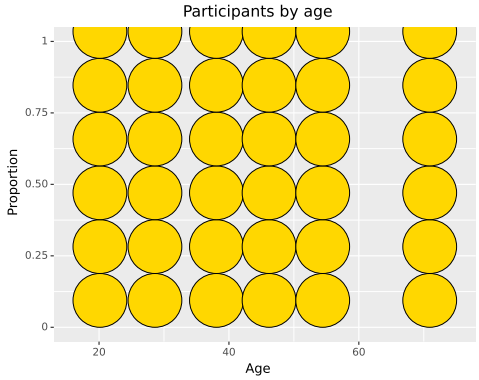


Figure 3.23: Dotplot with a specified color scheme.

The are many more options available. See the help for the details and examples.

4 Bivariate Graphs

Bivariate graphs display the relationship between two variables. The type of graph will depend on the measurement elevel of the variables (categorical or quantitative).

4.1 Categorical vs. Categorical

When plotting the relationship between two categorical variables, stacked, grouped, or segmented bar charts are typically used. A less common approach is the mosaic chart (7.5).

4.1.1 Stacked bar chart

Let's plot the relationship between automobile class and drive type (front-wheel, rear-wheel, or 4-wheel drive) for the automobiles in the Fule economy dataset.

```
# stacked bar chart
from plotnine.data import mpg
from plotnine import *

print((ggplot(mpg,aes(x="class", fill="drv"))+geom_bar(position="stack")+
       theme(legend_position=(0.55,0.75),legend_direction="vertical")))
```

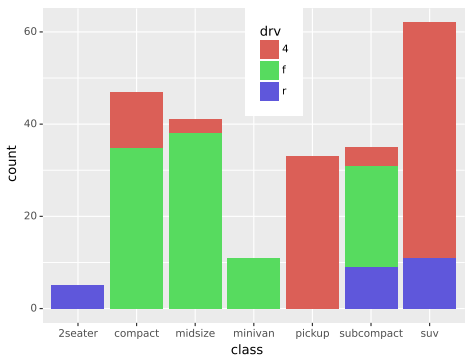


Figure 4.1: Stacked bar chart

From the chart, we can see for example, that the most common vehicle is the SUV. All 2seater cars are rear wheel drive, while most, but not all SUVs are 4-wheel drive.

Stacked is the default, so the last line could have also been written as geom_bar.

4.1.2 Grouped bar chart

Grouped bar charts place bars for the second categorical variable side-by-side. To create a grouped bar plot use the `position = "dodge"`.

```
# grouped bar plot
print((ggplot(mpg,aes(x="class",fill="drv"))+geom_bar(position="dodge")+
       theme(legend_position=(0.55,0.75),legend_direction="vertical")))
```

Figure 4.2: Side-by-side bar chart

Notice that all Minivans are front-wheel drive. By default, zero count bars dropped and the remaining bars are made wider. This may not be behavior you want. You can modify this using the `position = position_dodge(preserve = "single")` option.

```
# grouped bar plot preserving zero count bars
print((ggplot(mpg,aes(x = "class", fill = "drv"))+
       geom_bar(position = position_dodge(preserve ="single"))+
       theme(legend_position=(0.55,0.75),legend_direction="vertical")))
```

Figure 4.3: Side-by-side bar chart with zero count bars retained

4.1.3 Segmented bar chart

A segmented bar plot is stacked bar plot where each bar represents 100 percent. You can create a segmented bar chart using the `position = "fill"` option.

```
# bar plot, with each bar representing 100%
print((ggplot(mpg,aes(x="class",fill="drv"))+geom_bar(position="fill")+
       labs(y = "Proportion")))
```

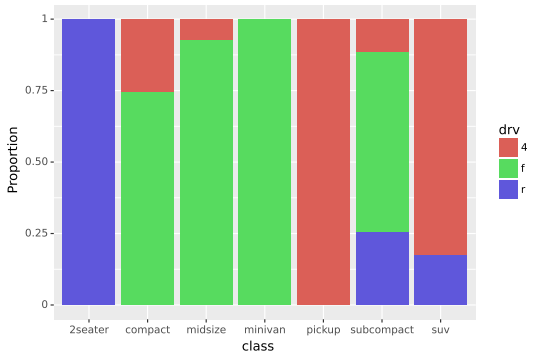


Figure 4.4: Segmented bar chart

This type of plot is particularly useful if the goal is to compare the percentage of a category in one variable across each level of another variable. For example, the proportion of front-wheel drive cars go up as you move from compact, to midsize, to minivan.

4.1.4 Improving the color and labeling

You can use additional options to improve color and labeling. In the graph below

- `reorder_categories` modifies the order of the categories for the class variable and both the order and the labels for the drive variable.
- `scale_y_continuous` modifies the y-axis tick mark labels
- `labs` provides a title and changed the labels for the x and the y axes and the legend
- `scale_y_brewer` changes the fill color scheme.
- `theme_minimal` removes the grey background an changed the grid color.

```
# Select class and drv
from plydata import *
import numpy as np
from mizani.formatters import percent_format

df = mpg >> select("class", "drv")

# change the order the levels for the categorical
df["class"] = (df["class"].cat.reorder_categories(["2seater","subcompact",\
                          "compact","midsize","minivan", "suv", "pickup"]))
df["drv"] = (df["drv"].cat.reorder_categories(["f", "r", "4"]).cat
                      .rename_categories(["front-wheel","rear-wheel","4-wheel"]))
```

```
print((ggplot(df,aes(x = "class",fill = "drv")) + geom_bar(position = "fill") +
    scale_y_continuous(breaks = np.arange(0,1.2,.2),labels = percent_format()) +
    scale_fill_brewer(type="qual",palette="Set2")+
    labs(y="Percent",fill="Drive Train",x="Class",
         title="Automobile Drive by Class") +theme_minimal()+
    theme(legend_position=(0.55,0.55),legend_direction="vertical")))
```

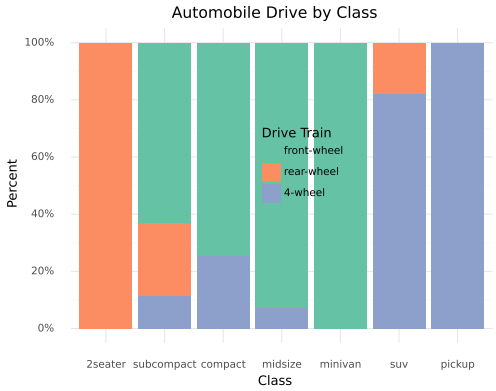


Figure 4.5: Segmented bar chart with improved labeling and color

In the graph above, the reorder_categories function was used to reorder and/or rename the levels of a categorical variable. You could also apply this to the original dataset, making these changes permanent. It would then apply to all future graphs using that dataset. For example:

```
# change the order the levels for the categorical variable "class"
mpg["class"] = (mpg["class"].cat.reorder_categories(["2seater","subcompact",\
                            "compact","midsize","minivan", "suv", "pickup"]))
```

Next, let's add percent labels to each segment. First, we'll create a summary dataset that has the necessary labels.

```
# create a summary dataset
plotdata = (mpg >> group_by("class", "drv") >> summarize(n = "n()") >>
                   group_by("class") >> mutate(pct = "n/sum(n)"))
plotdata.loc[:,"lbl"] = percent_format()(plotdata['pct'])
```

Table 4.1: Plot data

class	drv	n	pct	lbl
compact	f	35	0.7446809	74.5%
compact	4	12	0.2553191	25.5%
midsize	4	3	0.0731707	7.3%
suv	r	11	0.1774194	17.7%
2seater	r	5	1.0000000	100.0%
suv	4	51	0.8225806	82.3%
midsize	f	38	0.9268293	92.7%
minivan	f	11	1.0000000	100.0%
pickup	4	33	1.0000000	100.0%
subcompact	r	9	0.2571429	25.7%
subcompact	f	22	0.6285714	62.9%
subcompact	4	4	0.1142857	11.4%

Next, we'll use this dataset and the `geom_text` function to add labels to each bar segment.

```
# create segmented bar chart
# adding labels to each segment
plotdata["class"] = (plotdata["class"].cat
                     .reorder_categories(["2seater","subcompact",\
                         "compact","midsize","minivan", "suv", "pickup"]))
plotdata["drv"] = (plotdata["drv"].cat.reorder_categories(["f", "r", "4"])
                   .cat.rename_categories(["front-wheel","rear-wheel","4-wheel"]))
print((plotdata >>ggplot(aes(x = "class", y = "pct",fill = "drv"))+
       geom_bar(stat = "identity", position = "fill")+
       scale_y_continuous(breaks= np.arange(0,1.2,.2),labels=percent_format())+
    geom_text(aes(label="lbl"),size=10,position=position_stack(vjust=0.5))+
    scale_fill_brewer(type="qual",palette = "Set2") +
    labs(y = "Percent", fill = "Drive Train",x = "Class",
         title = "Automobile Drive by Class") +theme_minimal()+
    theme(legend_position=(0.55,0.55),legend_direction="vertical")))
```

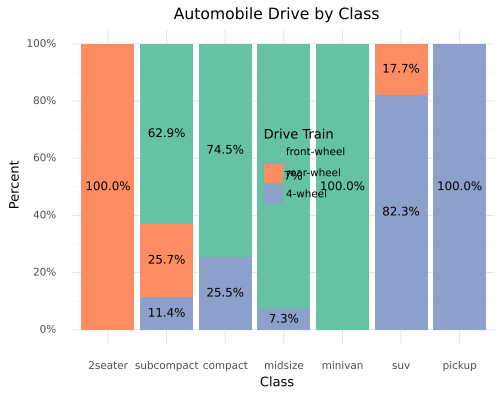


Figure 4.6: Segmented bar chart with value labeling

Now we have a graph that is easy to read and interpret.

4.2 Quantitative vs. Quantitative

The relationship between two quantitative variables is typically displayed using scatterplots and line graphs.

4.2.1 Scatterplot

The simplest display of two quantitative variables is a scatterplot, with each variable represented on an axis. For example, using Salaries

```
data(Salaries,package="carData")
```

```
# Simple scatterplot
print((r.Salaries >>ggplot(aes(x="yrs.since.phd",y="salary"))+geom_point()))
```

geom_point options can be used to change the :

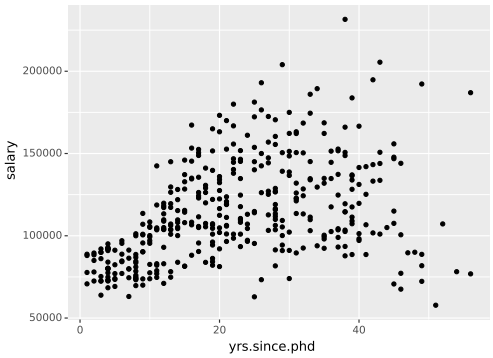


Figure 4.7: Simple scatterplot

- `color` - point color
- `size` - point size
- `shape` - point shape
- `alpha` - point transparency. Transparency ranges from 0 (transparency) to 1 (opaque), and is a useful parameter when points overlap.

The functions scale_x_continuous and scale_y_continuous control the scaling on x and y axes respectively.

```
from mizani.formatters import currency_format

print((r.Salaries >> ggplot(aes(x="yrs.since.phd",y = "salary")) +
    geom_point(color = "cornflowerblue", size = 2, alpha = 0.8)+
    scale_y_continuous(labels=currency_format(),limits=[50000,250000])+
    scale_x_continuous(breaks=np.arange(0,61,10),limits=[0,60])+
    labs(x = "Years Since PhD",y = "",
         title = "Experience vs. Salary (9-month salary for 2008-2009)")
))
```

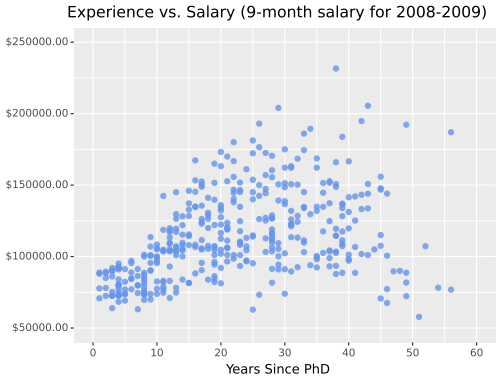


Figure 4.8: Scatterplot with color, transparency, and axis scaling

4.2.1.1 Adding best fit lines

It is often useful to summarize the relationship displayed in the scatterplot, using a best fit line. Many types of lines are supported, including linear, polynomial, and nonparametric (loess). By

default, 95% confidence limits for these lines are displayed.

```
# scatterplot with linear fit line
print((r.Salaries >>ggplot(aes(x="yrs.since.phd", y = "salary")) +
       geom_point(color = "steelblue")+geom_smooth(method = "lm")))
```

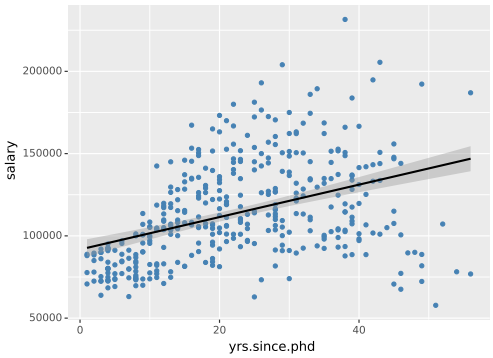


Figure 4.9: Scatterplot with linear fit line

Clearly, salary increases with experience. However, there seems to be a dip at the right end - professors with significant experience, earning lower salaries. A straight line does not capture this non-linear effect. A line with a bend will fit better here.

A polynomial regression line provides a fit line of the form

$$\hat{y} = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \beta_4 x^4 + \cdots$$

Typically either a quadratic (one bend), or cubic (two bends) line is used. It is rarely necessary to use a higher order(>3) polynomials. Applying a quadratic fit to the salary dataset produces the following result.

```
# create polynome
def poly(x,p):
  return x**p

# scatterplot with quadratic line of best fit
print((r.Salaries >>ggplot(aes(x="yrs.since.phd", y = "salary")) +
       geom_point(color = "steelblue")+
       geom_smooth(method = "lm",formula="y ~x+poly(x,2)",color="indianred")))
```

Finally, a smoothed nonparametric fit line can often provide a good picture of the relationship. The default in `plotnine` is a loess line which stands for for locally weighted scatterplot smoothing.

```
# scatterplot with loess smoothed line
print((ggplot(r.Salaries,aes(x = "yrs.since.phd", y = "salary")) +
       geom_point(color= "steelblue")+geom_smooth(color = "tomato")))
```

You can suppress the confidence bands by including the option `se = false`.

Here is a complete (and more attractive) plot

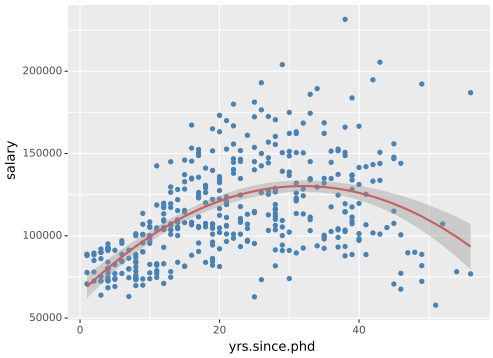


Figure 4.10: Scatterplot with quadratic fit line

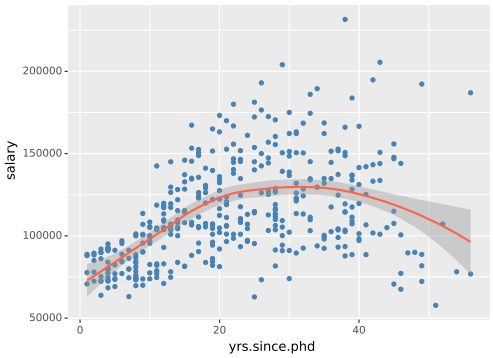


Figure 4.11: Scatterplot with nonparametric fit line

```
# scatterplot with loess smoothed line
# and better labeling and color
print((ggplot(r.Salaries,aes(x = "yrs.since.phd", y = "salary")) +
       geom_point(color="cornflowerblue", size = 2, alpha = .6) +
       geom_smooth(size = 1.5,color = "darkgrey") +
       scale_y_continuous(label = currency_format(),limits = [50000, 250000]) +
       scale_x_continuous(breaks = range(0, 61, 10),limits = [0, 60])+
       labs(x = "Years Since PhD",y = "",
       title = "Experience vs. Salary (9-month salary for 2008-2009)") +
       theme_minimal()))
```

4.2.2 Line plot

When one of the two variables represents time, a line plot can be an effective method of displaying relationship. For example, the code below displays the relationship between time (`year`) and life expectancy (`lifeExp`) in the United States between 1952 and 2007. The data comes from the `gapminder` dataset.

```
from gapminder import gapminder
```

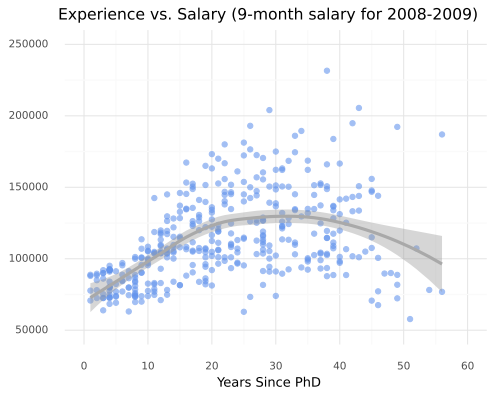


Figure 4.12: Scatterplot with non parametric fit line

```
# Select US cases
plotdata = gapminder >> query("country == 'United States'")

# Simple line plot
print((plotdata >> ggplot(aes(x = "year", y = "lifeExp"))+geom_line()))
```

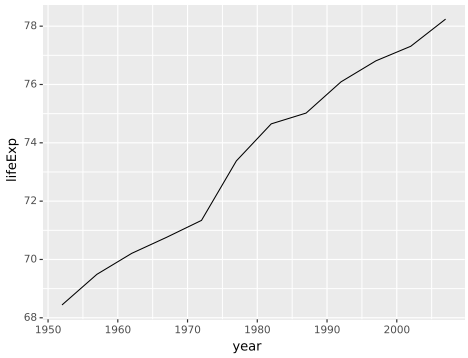


Figure 4.13: Simple line plot

It is hard to read individual values in the graph above. In the next plot, we'll add points as well.

```
# Line plot with points and improved labelling
print((plotdata >> ggplot(aes(x = "year", y = "lifeExp")) +
      geom_line(size = 1.5,color = "lightgrey") +
      geom_point(size = 3,color = "steelblue") +
      labs(y = "Life Expectancy (years)", x = "Year",
           title = "Life expectancy changes over time",
           caption = "Source: http://www.gapminder.org/data/")))
```

Time dependent data is covered in more detail under Time series. Customizing line graphs is covered in the Customizing graphs section

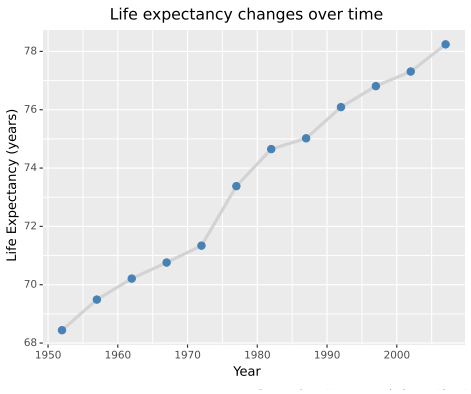


Figure 4.14: Line plot with points and labels

4.3 Categorical vs. Quantitative

When plotting the relationship between a categorical variable and a quantitative variable, a large number of graph types are available. These include bar charts using summary statistics, grouped kernel density plots, side-by-side box plots, side-by-side violin plots, mean/sem plots, ridgeline plots, and Cleveland plots.

4.3.1 Bar chart (on summary statistics)

In previous sections, bar charts were used to display the number of cases by category for a signe variable or for two variables. You can also use bar charts to display other summary statistics (e.g., means or medians) on a quantitative variable for each level of a categorical variable.

For example, the following graph displays the mean salary for a sample of university professors by their academic rank.

```
# Import starwars dataset
from datar.datasets import starwars

# calculate mean salary for each rank
plotdata = (r.Salaries>>group_by("rank")>>summarize(mean_salary="mean(salary)"))
```

Table 4.2: Plot data

rank	mean_salary
Prof	126772.11
AsstProf	80775.99
AssocProf	93876.44

```
# Plot mean salaries
print((plotdata >>ggplot(aes(x="rank",y="mean_salary"))+
       geom_bar(stat="identity")))
```

We can make it more attractive with some options.

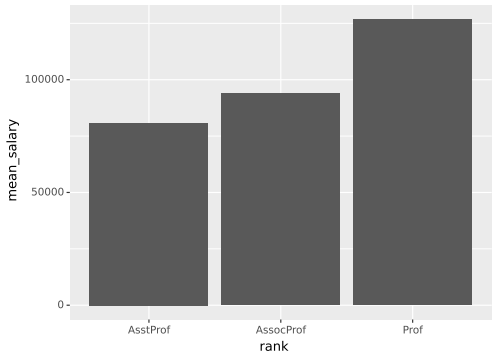


Figure 4.15: Bar chart displaying means

```
# plot mean salaries in a more attractive fashion
print((ggplot(plotdata,aes(x = "rank",y = "mean_salary")) +
       geom_bar(stat = "identity", fill = "cornflowerblue") +
       geom_text(aes(label=currency_format()(plotdata["mean_salary"])),va="center")+
       scale_y_continuous(breaks = range(0, 130000, 20000),
                          labels = currency_format()) +
     labs(title = "Mean Salary by Rank (9-month academic salary for 2008-2009)",
       x = "",y = "")+
     scale_x_discrete(breaks=["AsstProf","AssocProf","Prof"],
     labels = ["Assistant\nProfessor","Associate\nProfessor","Full\nProfessor"])))
```

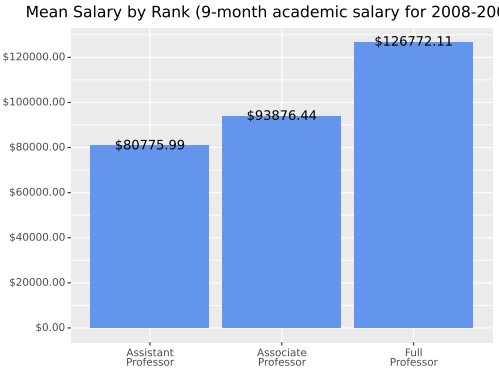


Figure 4.16: Bar chart displaying means

One limitation of such plots is that they do not display the distribution of the data - only the summary statistic for each group. The plots below correct this limitation to some extent.

4.3.2 Grouped kernel density plots

One can compare groups on a numeric variable by superimposing kernel density plots in a single graph.

```
# plot the distribution of salaries
# by rank using kernel density plots
print((ggplot(r.Salaries, aes(x = "salary", fill = "rank")) +
```

```
        geom_density(alpha = 0.4)+labs(title = "Salary distribution by rank")+
        theme(legend_position=(0.8,0.7),legend_direction='vertical')))
```

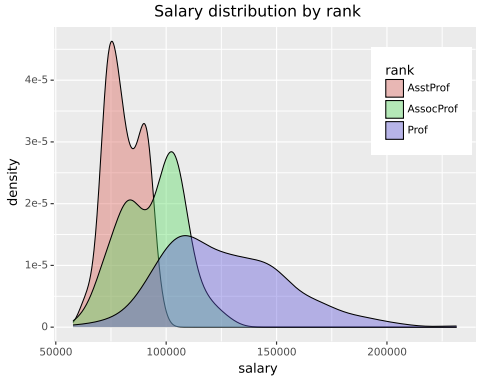


Figure 4.17: Grouped kernel density plots

The `alpha` option makes the density plots partially transparent, so that we can see what is happening under the overlaps. Alpha values range from 0 (transparent) to 1 (opaque). The graph makes clear that, in general, salary goes up with rank. However, the salary range for full professors is very wide.

4.3.3 Box plots

A boxplot display the 25^{th} percentile, median, and 75^{th} percentile of distribution. The whiskers (vertical lines) capture roughly 99% of a normal distribution, and observations outside this range are plotted as points representing outliers. Side-by-side box plots are very useful for comparing groups (i.e., the levels of categorical variable) on a numerical variable.

```
# plot the distribution of salaries by rank using boxplots
print((ggplot(r.Salaries,aes(x = "rank", y = "salary")) +geom_boxplot() +
       labs(title = "Salary distribution by rank")))
```

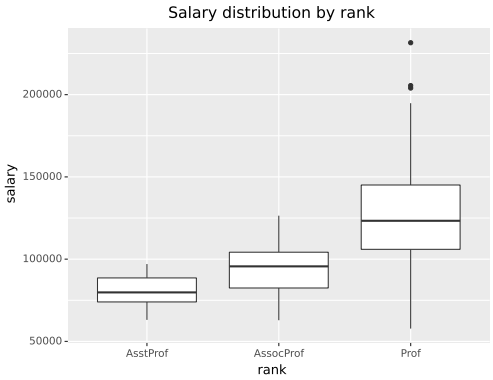


Figure 4.18: Side-by-side boxplots

Notched boxplots provide an approximate methode for visualizing whether groups differ. Although not a formal test, if the notches of two boxplots do not overlap, there is strong evidence (95% confidence) that the medians of the two groups differ.

```
# plot the distribution of salaries by rank using boxplots
print((ggplot(r.Salaries, aes(x = "rank", y = "salary")) +
       geom_boxplot(notch = True, fill = "cornflowerblue",alpha = 0.7) +
       labs(title = "Salary distribution by rank")))
```

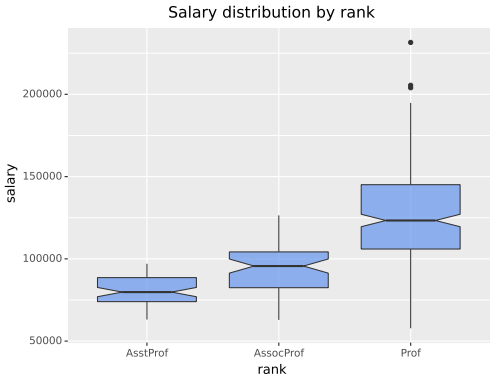


Figure 4.19: Side-by-side notched boxplots

In the example above, all three groups appear to differ.

One of the advantages of boxplots is that their widths are not usually meaningful. This allows you to compare the distribution of many groups in a single graph.

4.3.4 Violin plots

Violin plots are similar to kernel density plots, but are mirrored and rotated 90°.

```
#a plot the distribution of salaries
# by rank using violin plots
print((ggplot(r.Salaries,aes(x = "rank",y = "salary")) +geom_violin() +
       labs(title = "Salary distribution by rank")))
```

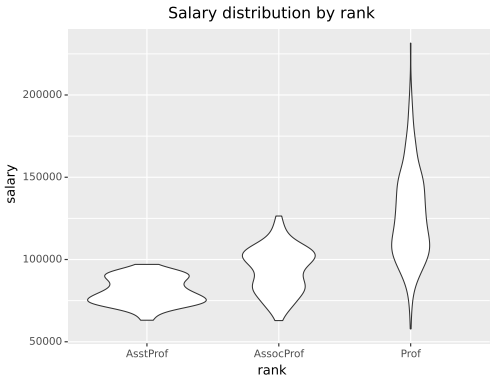


Figure 4.20: Side-by-side violin plots

A useful variation is to superimpose boxplots on violin plots

```
# Plot the distribution using violin and boxplots
print((ggplot(r.Salaries,aes(x = "rank", y = "salary"))+
      geom_violin(fill = "cornflowerblue")+
      geom_boxplot(width = 0.2,fill = "orange", outlier_color = "orange",
                   outlier_size = 2)+
      labs(title = "Salary distribution by rank")))
```

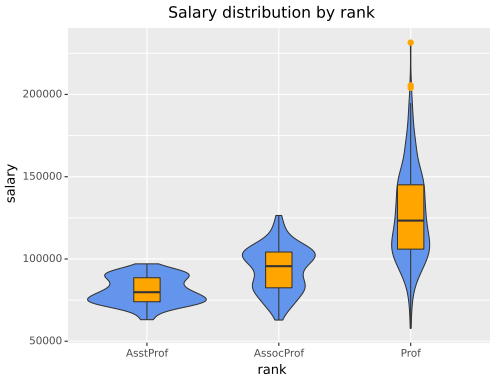


Figure 4.21: Side-by-side violin/box plots

4.3.5 Ridgeline plots

A ridgeline plot (also called a joyplot) displays the distribution of a quantitative variable for several groups. They're similar to kernel density plots with vertical faceting, but take up less room. Ridgeline plots are created with the seaborn package.

Using the Fuel economy dataset, let's plot the distribution of city driving miles per gallon by car class.

```
from joypy import joyplot
import matplotlib.pyplot as plt

from matplotlib import cm
joyplot(mpg,by = "class",column = "cty", colormap=cm.autumn, fade = True);
plt.show()
```

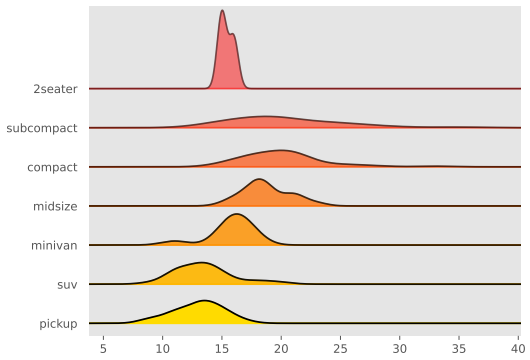


Figure 4.22: Ridgeline graph with color fill

4.3.6 Mean/SEM plots

A popular method for comparing groups on a numeric variable is the mean plot with error bars. Error bars can represent standard deviations, standard error of the mean, or confidence intervals. In this section, we'll plot means and standard errors.

```
# Calculate means, standard deviation, standard errors, and 95% confidence intervals
from math import *
import scipy.stats as st
from plotnine.data import mtcars

plotdata = (r.Salaries >>
    group_by("rank")>>
    summarize(
      n = "n()",
      mean = "mean(salary)",
      sd = "std(salary)",
      se =  "std(salary)/sqrt(n())",
      ci = "st.t.isf(0.975,n()-1)*std(salary)/sqrt(n())"))
```

The resulting dataset is given below.

Table 4.3: Plot data

rank	n	mean	sd	se	ci
Prof	266	126772.11	27666.523	1696.3434	-3340.026
AsstProf	67	80775.99	8112.882	991.1463	-1978.888
AssocProf	64	93876.44	13723.214	1715.4018	-3427.957

```
# Plot the means and standard errors
print((plotdata >> ggplot(aes(x = "rank",  y = "mean", group = 1))+
    geom_point(size=3)+geom_line()+
    geom_errorbar(aes(ymin = "mean- se", ymax = "mean + se"), width = 0.1)))
```

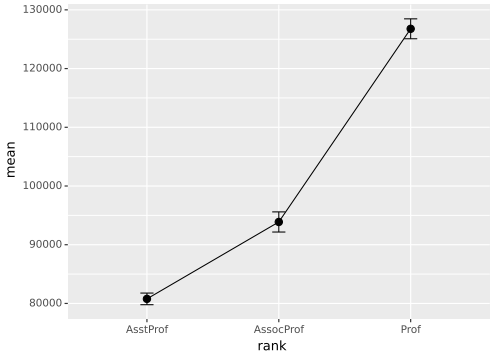


Figure 4.23: Mean plots with standard error bars

Although we plotted error bars representing the standard error, we could have plotted standard deviations or 95% confidence intervals. Simply replace `se` with `sd` or `error` in the `aes` option.

We can use the same technique to compare salary across rank and sex. (Technically, this is not bivariate since we're plotting rank, sex, and salary, but it seems to fit here)

```
# calculate means and standard errors by rank and sex
plotdata = (r.Salaries >>
      group_by("rank", "sex") >>
      summarize(n = "n()",
                mean = "mean(salary)",
                sd = "std(salary)",
                se = "std(salary)/sqrt(n())"))
```

Table 4.4: Plot data

rank	sex	n	mean	sd	se
Prof	Male	248	127120.82	28156.868	1787.963
AsstProf	Male	56	81311.46	7830.477	1046.392
AssocProf	Male	54	94869.70	12770.900	1737.899
Prof	Female	18	121967.61	19066.807	4494.090
AssocProf	Female	10	88512.80	17043.367	5389.586
AsstProf	Female	11	78049.91	8935.848	2694.259

```
# plot the means and standard errors by sex
print((plotdata >>ggplot(aes(x="rank",y="mean",group="sex",color = "sex"))+
    geom_point(size = 3)+geom_line(size=1)+
    geom_errorbar(aes(ymin = "mean - se", ymax = "mean + se"), width = 0.1)+
    theme(legend_position=(0.2,0.7),legend_direction='vertical')))
```

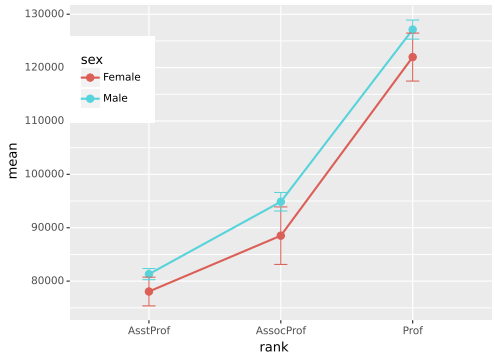


Figure 4.24: Mean plots with standard error bars by sex

Unfortunately, the error bars overlap. We can dodge the horizontal positions a bit to overcome this.

```
# plot the means and standard errors by sex (dodged)
pd = position_dodge(0.2)
print((ggplot(plotdata, aes(x="rank",y="mean", group="sex", color="sex")) +
       geom_point(position = pd, size = 3) +
       geom_line(position = pd,size = 1) +
       geom_errorbar(aes(ymin = "mean - se", ymax = "mean + se"), width = 0.1,
                     position= pd)+
       theme(legend_position=(0.2,0.7),legend_direction='vertical')))
```

Finally, lets add some options to make the graph more attractive.

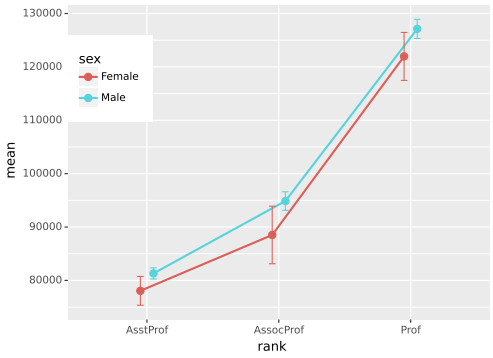


Figure 4.25: Mean plots with standard error bars (dodged)

```
# improved means/standard error plot
pd = position_dodge(0.2)
print((ggplot(plotdata,aes(x="rank",y="mean", group="sex",color="sex")) +
       geom_point(position=pd, size = 3) +
       geom_line(position = pd, size = 1) +
       geom_errorbar(aes(ymin = "mean - se", ymax = "mean + se"), width = 0.1,
                     position = pd, size = 1) +
       scale_y_continuous(labels = currency_format()) +
       scale_color_brewer(type="qual",palette="Set1") +
       theme_minimal() +
       labs(title = "Mean salary by rank and sex (mean +/- standard error)",
            x = "",y = "",color = "Gender")+
       theme(legend_position=(0.2,0.7),legend_direction='vertical')))
```

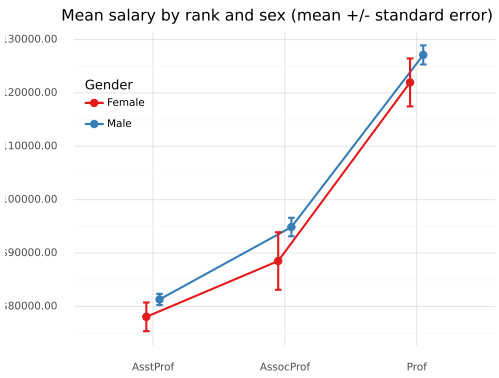


Figure 4.26: Mean/se plot with better labels and colors

4.3.7 Strip plots

The relationship between a grouping variable and a numeric variable can be displayed with a scatter plot. For example

```
# plot the distribution of salaries
# by rank using strip plots
```

```
print((ggplot(r.Salaries, aes(y = "rank", x = "salary")) +
       geom_point() + labs(title = "Salary distribution by rank")))
```

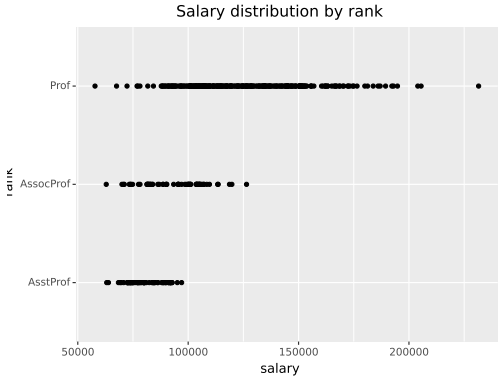


Figure 4.27: Categorical by quantitative scatterplot

These one-dimensional scatterplots are called strip plots. Unfortunately, overprinting of points makes interpretation difficult. The relationship is easier to see if the points are jittered. Basically a small random number is added to each y-coordinate.

```
# plot the distribution of salaries
# by rank using jittering
print((ggplot(r.Salaries, aes(y = "rank", x = "salary")) +geom_jitter() +
       labs(title = "Salary distribution by rank")))
```

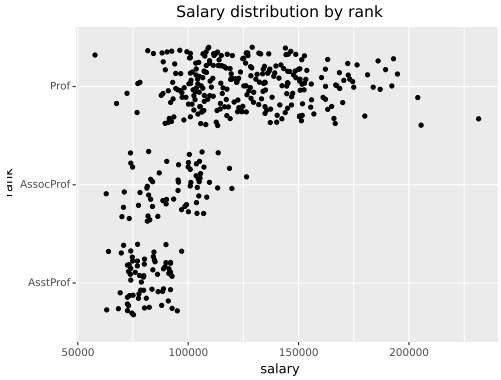


Figure 4.28: Jittered plot

It is easier to compare groups if we use color.

```
# plot the distribution of salaries
# by rank using jittering
print((ggplot(r.Salaries,aes(y = "rank", x = "salary", color = "rank")) +
       geom_jitter(alpha = 0.7,size = 1.5) +
       scale_x_continuous(labels = currency_format()) +
       labs(title = "Academic Salary by Rank (9-month salary for 2008-2009)",
            x = "",y = "") +
       theme_minimal() +
```

```
      theme(legend_position = "none")+
      scale_y_discrete(breaks=["AsstProf","AssocProf","Prof"],
  labels = ["Assistant\nProfessor","Associate\nProfessor","Full\nProfessor"])))
```

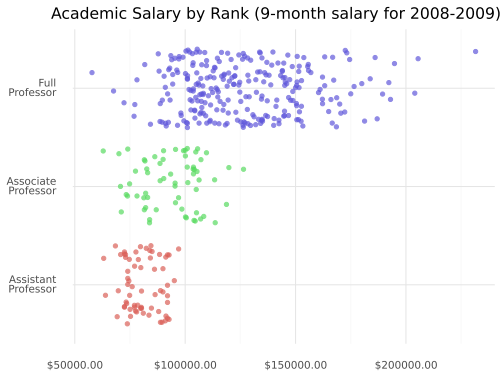


Figure 4.29: Fancy jittered plot

The option `legend_position = "none"` is used to suppress the legend (which is not needed here). Jittered plots work well when the number of points in not overly large.

4.3.7.1 Combining jitter and boxplots

It may be easier to visualize distributions if we add boxplots to the jitter plots.

```
# plot the distribution of salaries
# by rank using jittering
print((ggplot(r.Salaries, aes(x = "rank", y = "salary", color = "rank")) +
  geom_boxplot(size=1,outlier_shape=1,outlier_color="black",outlier_size=3) +
  geom_jitter(alpha = 0.5,width=0.2) +
  scale_y_continuous(labels = currency_format()) +
  labs(title = "Academic Salary by Rank (9-month salary for 2008-2009)",
       x = "",y = "") +
  theme_minimal() +theme(legend_position = "none") +
  scale_x_discrete(breaks=["AsstProf","AssocProf","Prof"],
  labels = ["Assistant\nProfessor","Associate\nProfessor","Full\nProfessor"])+
  coord_flip()))
```

Several options were added to create this plot.

For the boxplot

- `size = 1` makes the lines thicker
- `outlier_color = "black"` makes outliers black
- `outlier_shape = 1` specifies circles for outliers
- `outlier_size = 3` increases the size of the outlier symbol

For the jitter

- `alpha = 0.5` makes the points more transparent

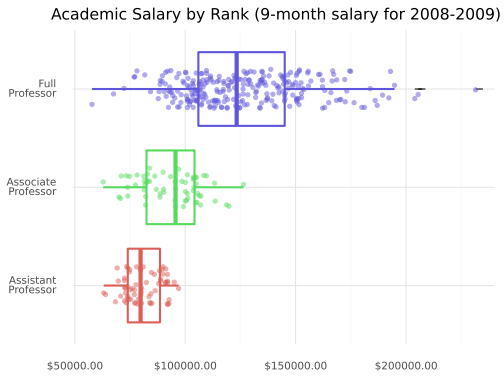


Figure 4.30: Jitter plot with superimposed box plots

- `width = 0.2` decreases the amount of jitter (0.4 is the default)

Finally, the x and y axes are revered using the `coord_flip` function (i.e. the graph is turned on the side).

4.3.8 Beeswarm plots

Beeswarm plots (also called violin scatter plots) are similar to jittered scatterplots, in that they display the distribution of a quantitative variable by plotting points in ways that reduces overlap. In addition, they also help display the density of the data at each point (in a manner that is similar to a violin plot).

```
# plot the distribution of salaries
# by rank using beewarm-syle plots
import seaborn as sns
ax =  sns.swarmplot(x="rank", y="salary", hue = "rank", data=r.Salaries)
ax.set_xticklabels(["Assistant\nProfessor","Associate\nProfessor",
                    "Full\nProfessor"]);
ax.set(xlabel="",ylabel="",
       title="Academic Salary by Rank (9-month salary for 2008-2009)");
plt.show()
```

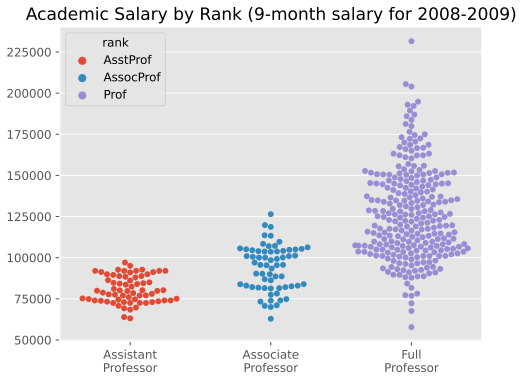


Figure 4.31: Beeswarm plot

4.3.9 Cleveland Dot Charts

Cleveland plots are useful when you want to compare a numeric statistic for a large number of groups.For example, say that you want to compare the 2007 life expectancy for Asian country using the gapminder dataset.

```
# Suset Asian countries in 2007
plotdata = gapminder >> query("continent=='Asia' & year == 2007")
# Basic Cleveland plot of life expectancy by country
print((plotdata >>ggplot(aes(x = "lifeExp", y = "country"))+geom_point()))
```

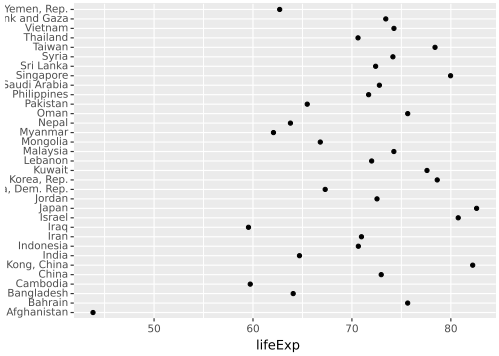


Figure 4.32: Basic Cleveland dot plot

Comparisons are usually easier if the y-axis is sorted.

```
# Sorted Cleveland plot
print((plotdata >> ggplot(aes(x = "lifeExp", y = "reorder(country, lifeExp)"))+
       geom_point()))
```

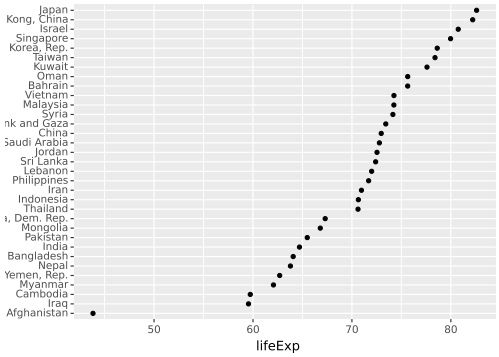


Figure 4.33: Sorted Cleveland dot plot

Finaly, we can use options to make the graph more attractive.

```
# Fancy Cleveland plot
print((plotdata >> ggplot(aes(x = "lifeExp", y = "reorder(country, lifeExp)"))+
      geom_point(color = "blue", size = 2)+
      geom_segment(aes(x = 40,xend = "lifeExp",y = "reorder(country, lifeExp)",
```

```
                    yend = "reorder(country, lifeExp)"), color= "lightgrey")+
labs(x = "Life Expectancy (years)",y = "",
     title = "Life Expectancy by Country")+
theme_minimal()))
```

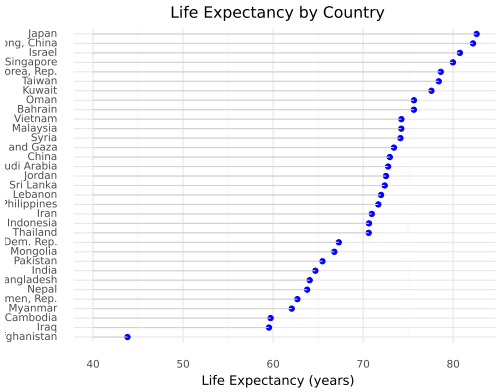


Figure 4.34: Fancy Cleveland plot

Japan clearly has the highest life expectancy, while Afghanistan has the lowest by far. This last plot is also called a lollipop graph.

5 Multivariate Graphs

Multivariate graphs display the relationships amons three or more variables. There are two common methods for accomodating multiple variables : grouping and faceting.

5.1 Grouping

In grouping, the values of the first two variables are mapped to the x and y axes. Then additional variables are mapped to other visual characteristics such as color, shape, size, line type, and transparency. Grouping allows you to plot the data for multiple groups in a single graph.

Using the Salaries dataset, let's display the relationship between yrs.since.phd and salary.

```
data(Salaries, package="carData")
```

```
from plotnine import *
print((r.Salaries >>ggplot(aes(x = "yrs.since.phd", y = "salary"))+
    geom_point()+labs(title = "Academic salary by years since degree")))
```

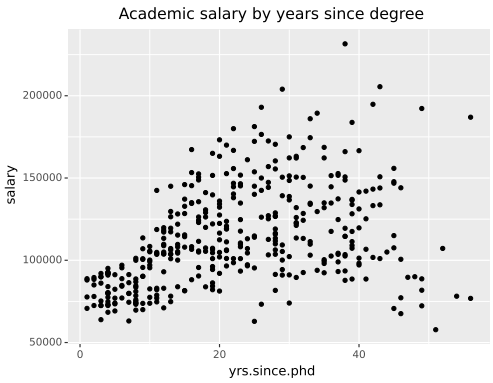


Figure 5.1: Simple scatterplot

Next, let's include the rank of the professor, using color.

```
# plot experience vs. salary (color represents rank)
print((r.Salaries >>ggplot(aes(x="yrs.since.phd",y ="salary",color = "rank"))+
       geom_point()+labs(title="Academic salary by rank and years since degree")))
```

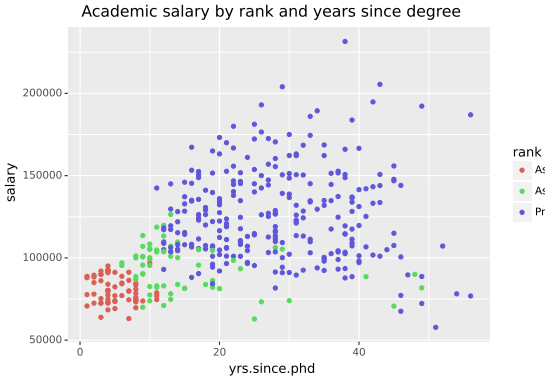


Figure 5.2: Scatterplot with color mapping

Finally, let's add the gender of professor, using the shape of the points to indicate sex. We'll increase the point size and add transparency to make the individual points clearer.

```
# plot experience vs. salary
# (color represents rank, shape represents sex)
print((r.Salaries >>
  ggplot(aes(x = "yrs.since.phd",y = "salary",color = "rank",shape = "sex")) +
  geom_point(size = 3,alpha = .6) +
  labs(title = "Academic salary by rank, sex, and years since degree")+
  theme(legend_position=(0.28,0.75),legend_direction='vertical')))
```

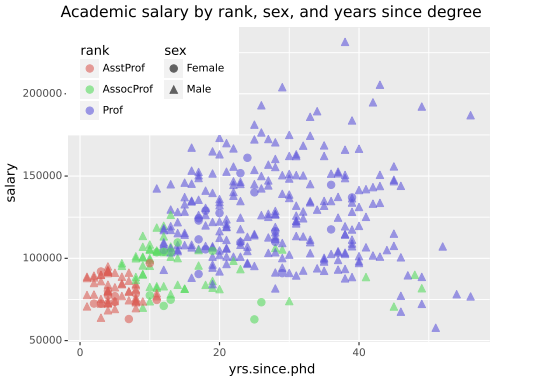


Figure 5.3: Scatter plot with color and shape mapping

I can't say that this is a great graphic. It is very busy, and it can be difficult to distinguish male from female professors. Faceting (described in the next section) would probably be a better approach.

> Notice the difference between specifying a constant value (such as `size = 3`) and a mapping of a variable to a visual charactreistic (e.g., `color = "rank"`). Mappings are always placed within the `aes` function, while the assignment of a constant value appear outside of the `aes` function.

Here is a cleaner example. We'll graph the relationship between years since Ph.D. and salary using the size of the points to indicate years of service. This is called a bubble plot.

```
# plot experience vs. salary
# (color represents rank and size represents service)
print((r.Salaries >>
  ggplot(aes(x = "yrs.since.phd",y="salary",color="rank",size="yrs.service")) +
  geom_point(alpha = 0.6) +
  labs(title = "Academic salary by rank, years of service,and years since degree")+
  theme(legend_position=(0.28,0.65),legend_direction='vertical')))
```

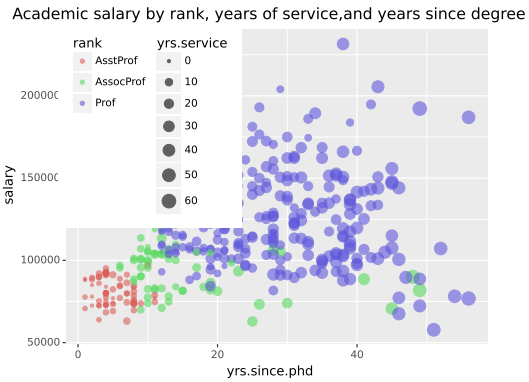


Figure 5.4: Scatterplot with size and color mapping

There is obviously a strong positive relationship between years since Ph.D. and year of service. Assistant Professors fall in the 0-11 years since Ph.D. and 0-10 years of service range. Clearly highly experienced professionals don't stay at the Assistant Professor level (they are probably promoted or leave the University). We don't find the same time demarcation between Associate and Full Professors.

Bubble plots are described in more detail in a later chapter.

As a final example, let's look at the yrs.since.phd vs salary and add sex using color and quadratic best fit lines.

```
# plot experience vs. salary with
# fit lines (color represents sex)
from mizani.formatters import currency_format

def poly(x,p):
  return x**p

print((
  r.Salaries >>
  ggplot(aes(x = "yrs.since.phd",y = "salary", color = "sex")) +
  geom_point(alpha = .4,size = 3) +
  geom_smooth(se=False,method = "lm",formula = "y~x+poly(x,2)",size = 1.5) +
  labs(x = "Years Since Ph.D.",title = "Academic Salary by Sex and Years Experience",
       y = "",color = "Sex") +
  scale_y_continuous(labels = currency_format()) +
  scale_color_brewer(type="qual", palette="Set1") +
```

```
  theme_minimal()
))
```

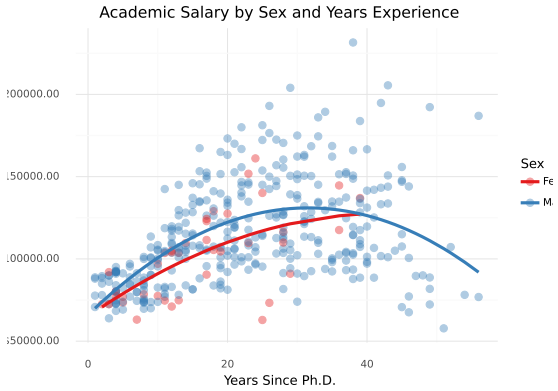


Figure 5.5: Scatterplot with color mapping and quadratic fit lines

To insert qualitative palette using plotnine, you have to include the `type = "qual"` argument.

5.2 Faceting

Grouping allows you to plot multiple variables in a single graph, using visual characteristics such as color, shape, and size.

In faceting, a graph consists of several separate plots or `small multiples`, one for each level of a third variable, or combination of variables. It is easiest to understand this with an example.

```
# Plot disp histogram by gear
print((r.Salaries >>ggplot(aes(x = "salary"))+
      geom_histogram(fill = "cornflowerblue", color = "white")+
      facet_wrap("~rank", ncol = 1)+
      labs(title = "Salary histograms by rank")))
```

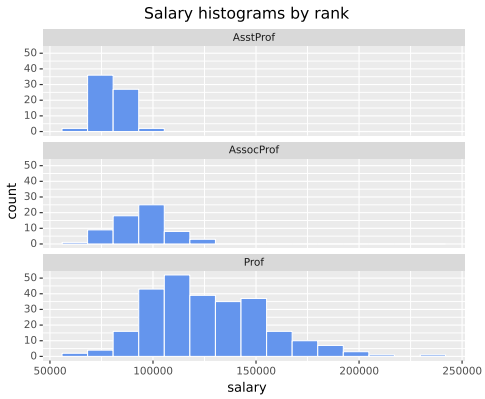


Figure 5.6: Salary distribution by rank

The `facet_wrap` function creates a separate graph for each level of `rank``. The`col‘ option controls the numbers of columns.

In the next example, two variables are used to define the facets.

```
# plot salary histograms by rank and sex
print((
  r.Salaries >>
    ggplot(aes(x = "salary/1000"))+
      geom_histogram(color= "white", fill = "cornflowerblue")+
      facet_grid("sex~rank")+
      labs(title = "Salary histograms by sex and rank", x = "Salary ($1000)")
))
```

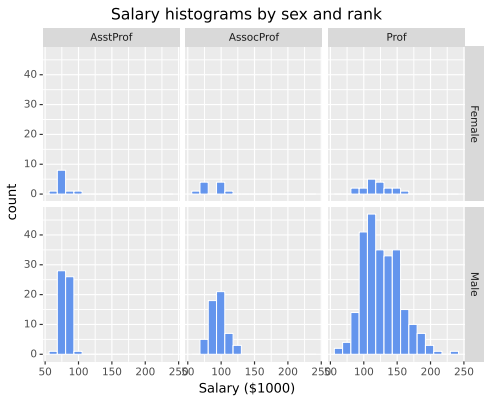


Figure 5.7: Salary distribution by rank and sex

The format of the `facet_grid` function is `facet_grid("row variable(s) ~ column variable(s)"`. Here, the function assigns `sex` to the rows and 'rank" to the columns, creating a matrix of 6 plots in one graph.

We can also combine grouping and faceting. Let's use Mean/SE plots and faceting to compare the salaries of male and female professors, within rank and discipline. We'll use color to distinguish sex and faceting to create plots for rank by discipline combinations.

```
# calculate means and standard erroes by sex,
# rank and discipline
from math import *
from plydata import *

plotdata = (r.Salaries >> group_by("sex","rank", "discipline")>>
            summarize(n = "n()",mean = "mean(salary)",sd = "std(salary)",
                      se =  "std(salary)/sqrt(n())"))


# create better labels for discipline
plotdata.discipline = (plotdata.discipline.cat
                       .rename_categories({'A': 'Theoretical', 'B': 'Applied'}))


# Create
print((plotdata >>ggplot(aes(x = "sex",y = "mean",color = "sex")) +
  geom_point(size = 3) +
  geom_errorbar(aes(ymin = "mean - se",ymax = "mean + se"),width = 0.1) +
  scale_y_continuous(breaks = range(70000, 140000, 10000),
                     label = currency_format()) +
```

Table 5.1: Plot data

sex	rank	discipline	n	mean	sd	se
Male	Prof	B	125	133518.36	26408.024	2362.005
Male	AsstProf	B	38	84647.08	6808.895	1104.549
Male	AssocProf	B	32	101621.53	9456.576	1671.702
Female	Prof	B	10	131836.20	16606.012	5251.282
Male	Prof	A	123	120619.26	28388.765	2559.729
Female	Prof	A	8	109631.88	14119.695	4992.066
Female	AssocProf	A	4	72128.50	5544.914	2772.457
Female	AsstProf	B	5	84189.80	8758.338	3916.848
Female	AssocProf	B	6	99435.67	12859.134	5249.719
Male	AssocProf	A	22	85048.86	10367.901	2210.444
Male	AsstProf	A	18	74269.61	4451.082	1049.130
Female	AsstProf	A	6	72933.33	4987.206	2036.018

Table 5.2: Plot data

sex	rank	discipline	n	mean	sd	se
Male	Prof	Applied	125	133518.36	26408.024	2362.005
Male	AsstProf	Applied	38	84647.08	6808.895	1104.549
Male	AssocProf	Applied	32	101621.53	9456.576	1671.702
Female	Prof	Applied	10	131836.20	16606.012	5251.282
Male	Prof	Theoretical	123	120619.26	28388.765	2559.729
Female	Prof	Theoretical	8	109631.88	14119.695	4992.066
Female	AssocProf	Theoretical	4	72128.50	5544.914	2772.457
Female	AsstProf	Applied	5	84189.80	8758.338	3916.848
Female	AssocProf	Applied	6	99435.67	12859.134	5249.719
Male	AssocProf	Theoretical	22	85048.86	10367.901	2210.444
Male	AsstProf	Theoretical	18	74269.61	4451.082	1049.130
Female	AsstProf	Theoretical	6	72933.33	4987.206	2036.018

```
facet_grid(". ~ rank + discipline") +
theme_bw() +
theme(legend_position = "none",panel_grid_major_x = element_blank(),
      panel_grid_minor_y = element_blank()) +
labs(x="", y="",
     title="Nine month academic salaries by gender, discipline, and rank") +
scale_color_brewer(type = "qual",palette="Set1")))
```

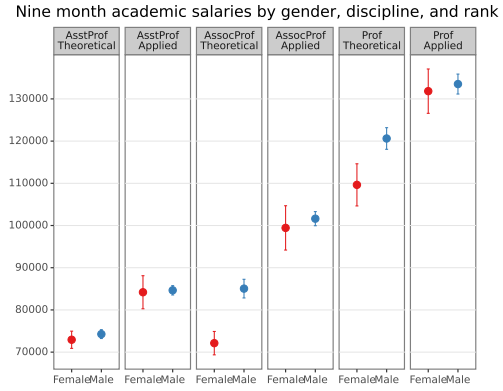


Figure 5.8: Salary by sex, rank, and discipline

The statement `facet_grid(".~ rank + discipline")` specifies no row variable (.) and columns defined by the combination of rank and discipline.

The `theme_` functions create a black and white theme and eliminates vertical grid lines and

minor horizontal grid lines. The `scale_color_brewer` function changes the color scheme for the points and errors bars.

At first glance, it appears that there might be gender differences in salaries for associate and full professors in theoretical fields. I say "might" because we haven't done any formal hypothesis testing yet (ANCOVA in this case).

See the Customizing section to learn more about customizing the appearance of a graph.

As a final example, we'll shift to a new dataset and plot the change in life expectancy over time for countries in the `America`. The data comes from the gapminder dataset in the `gapminder` package. Each country appears in its own facet. The theme functions are used to simplify the background color, rotate the x-axis text, and make the font size smaller.

```
# Plot life expectancy by year separately for each country in the America
from gapminder import gapminder

# Select the America data
plotdata = gapminder >> query("continent == 'Americas'")

# Plot life expectancy by year, for each country
print((
  plotdata >>
      ggplot(aes(x = "year", y = "lifeExp"))+
        geom_line(color = "grey")+
        geom_point(color= "blue")+
        facet_wrap("~country")+
        theme_minimal(base_size = 9)+
        theme(axis_text_x = element_text(angle = 45, hjust = 1))+
        labs(title = "Changes in Life Expectancy", x = "Year", y = "Life Expectancy")
))
```

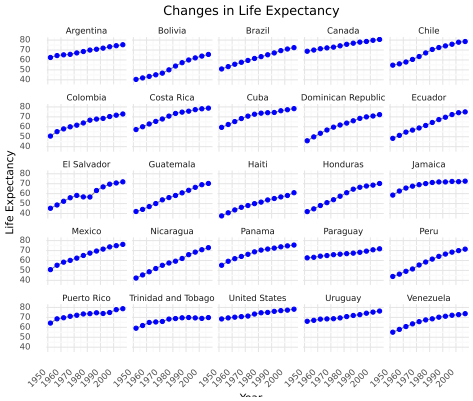


Figure 5.9: Changes in life expectancy by country

We can see that life expectancy is increasing in each country, but that Haiti is lagging behing.

6 Time - dependent graphs

A graph can be a powerful vehicle for displaying change over time. The most common time-dependent graph is the time series line graph. Other options include the dumbbell charts and the slope graph.

6.1 Time series

A time series is a set of quantitative values obtained at successive time points. The intervals between time points (e.g., hours, days, weeks, months, or years) are usually equal.

Consider the Economics time series that come with the `plotnine` package. It contains US monthly economic data collected from July 1967 to April 2015. Let's plot personal savings rate (`psavert`). We can do this with a simple line plot.

```
# Load economics data
from plotnine import *
from plotnine.data import economics

print((economics >> ggplot(aes(x = "date", y = "psavert"))+geom_line()+
       labs(title="Personal Savings Rate",x="Date",y="Personal Savings Rate")))
```

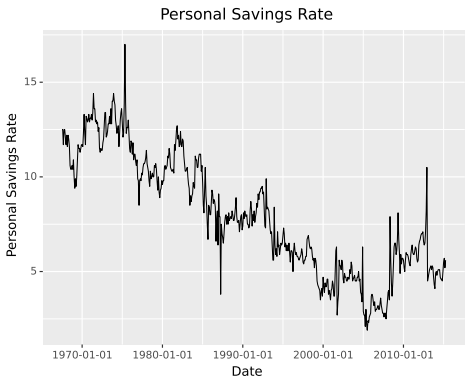


Figure 6.1: Simple time series

The calculated breaks are awful, we need to intervene. We do so using the date_breaks and date_format functions from mizani.

Set breaks every 5 years.

```
# Date breaks
from mizani.breaks import date_breaks
from plotnine.stats import stat_smooth

print((economics >> ggplot(aes(x = "date", y = "psavert"))+
      geom_line(color = "indianred",size=1)+
      geom_smooth(stat=stat_smooth(method="loess"),se=True)+
      scale_x_datetime(breaks = date_breaks('5 years'))+
      labs(title = "Personal Savings Rate (1967 to 2015)", x = "",
          y = "personal Saving Rate")))
```

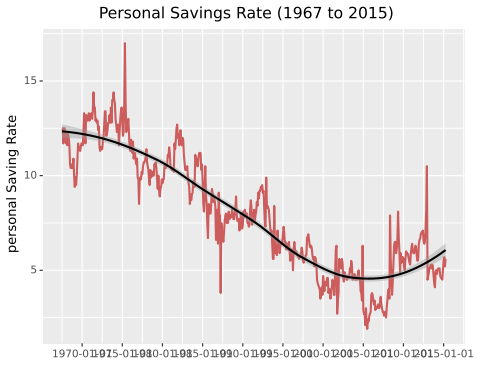


Figure 6.2: Simple time series with loess regression and modified date breaks

That is better. Since all the breaks are at the beginning of the year, we can omit the month and day. Using `date_format` we override the format string. For more on the options for the format string see the strftime behavior.

```
# Date format
from mizani.formatters import date_format

print((economics >> ggplot(aes(x = "date", y = "psavert"))+geom_line()+
      scale_x_datetime(breaks=date_breaks('10 years'),labels=date_format("%Y"))+
      labs(title="Personal Savings Rate",x="Date",y="personal Saving Rate")))
```

We can achieve the same result with a custom formatting function.

```
def custom_date_format1(breaks):
    """
    Function to format the date
    """
    return [x.year if x.month==1 and x.day==1 else "" for x in breaks]

print((economics >> ggplot(aes(x = "date", y = "psavert"))+geom_line()+
      scale_x_datetime(breaks=date_breaks('10 years'),labels=custom_date_format1)+
      labs(title = "Personal Savings Rate",x="Date",y="personal Saving Rate")))
```

We can use a custom formatting function to get results that are not obtainable with the `date_format` function. For example if we have monthly breaks over a handful of years we

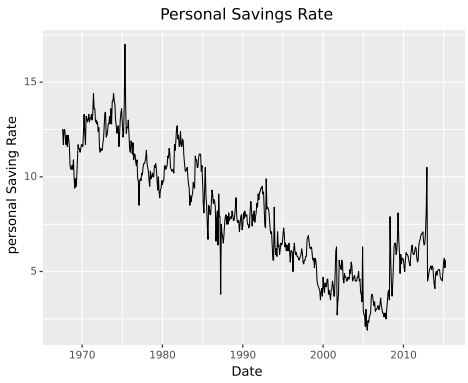


Figure 6.3: Simple time series with modeified date format

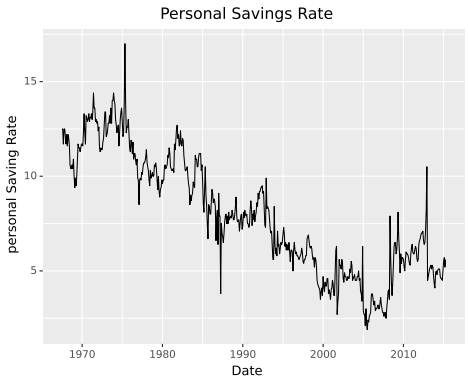


Figure 6.4: Simple time series with customized date format - one

can mix date formats as follows; specify beginning of the year and every other month. Such tricks can be used reduce overcrowding.

```
from datetime import date

def custom_date_format2(breaks):
    """
    Function to format the date
    """
    res = []
    for x in breaks:
        # First day of the year
        if x.month == 1 and x.day == 1:
            fmt = '%Y'
        # Every other month
        elif x.month % 2 != 0:
            fmt = '%b'
        else:
            fmt = ''

        res.append(date.strftime(x, fmt))

    return res
```

```
print((economics.loc[40:60,:] >> ggplot(aes(x="date",y="psavert"))+geom_line()+
   scale_x_datetime(breaks=date_breaks('1 months'),labels = custom_date_format2,
            minor_breaks = [])+
   labs(title = "Personal Savings Rate",x="Date",y="personal Saving Rate")))
```

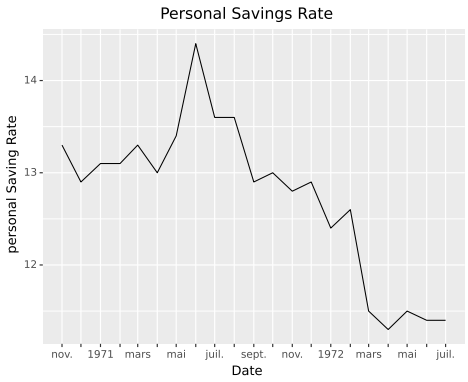


Figure 6.5: Simple time series with customized date format - two

We removed the labels but not the breaks, leaving behind dangling ticks for the skipped months. We can fix that by wrapping `date_breaks` around a filtering function.

```
def custom_date_format3(breaks):
    """
    Function to format the date
    """
    res = []
    for x in breaks:
        # First day of the year
        if x.month == 1:
            fmt = '%Y'
        else:
            fmt = '%b'

        res.append(date.strftime(x, fmt))

    return res


def custom_date_breaks(width=None):
    """
    Create a function that calculates date breaks

    It delegates the work to `date_breaks`
    """
    def filter_func(limits):
        breaks = date_breaks(width)(limits)
        # filter
        return [x for x in breaks if x.month % 2]

    return filter_func
```

```
print((economics.loc[40:60,:] >> ggplot(aes(x="date",y="psavert"))+geom_line()+
      scale_x_datetime(breaks = custom_date_breaks('1 months'),
                       labels = custom_date_format3)+
      labs(title="Personal Savings Rate",x="Date",y="personal Saving Rate")))
```

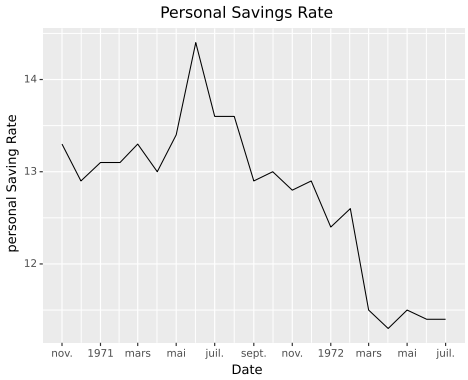


Figure 6.6: Simple time series with customized date format - three and date breaks

When plotting time series, be sure that the date variable is class `date` and not class `category`.

6.2 Dumbbell charts

Dumbbell charts are useful for dissplaying change between time points for several groups or observations.

Using the gapminder dataset let's plot the change in life expectancy from 1952 to 2007 in the Aericas. The dataset is in long format. We willneed to convert it to wide format in order to create the dumbbell plot.

```
# Subset data
from gapminder import gapminder
from plydata import *

plotdata_long =  (gapminder >>
    query("continent == 'Americas' & year in [1952,2007]") >>
    select("country", "year", "lifeExp"))

# Convert data to wide format
plotdata_wide = plotdata_long.pivot(index='country', columns='year', values='lifeExp')
# index to columns
plotdata_wide = plotdata_wide.rename_axis("country").reset_index()
plotdata_wide = plotdata_wide.rename({1952:"y1952", 2007 : "y2007"},axis='columns')

import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(16,8), facecolor = "white")
ax.grid(which="major", axis='both', color='#758D99', alpha=0.6, zorder=1);
ax.hlines(y=plotdata_wide["country"],xmin=plotdata_wide["y1952"],
          xmax=plotdata_wide["y2007"],color='#758D99',zorder=2,linewidth=2,
```

```
          label='_nolegend_', alpha=.8);
ax.scatter(plotdata_wide["y1952"],plotdata_wide["country"],label='y1952', s=60,
           color='#DB444B', zorder=3);
ax.scatter(plotdata_wide["y2007"],plotdata_wide["country"],label='y2007', s=60,
           color='#006BA2', zorder=3);
ax.xaxis.set_tick_params(labeltop=False,labelbottom=True,bottom=True,
                         labelsize=11,pad=-1);
ax.set_xlabel("y1952");
ax.set_yticks(plotdata_wide["country"]);
ax.set_yticklabels(plotdata_wide["country"],ha = 'left');
ax.yaxis.set_tick_params(pad=120,labelsize=11);
ax.legend(['1952', '2007'], loc=(0,0.9),ncol=1, frameon=False,
          handletextpad=-0.1, handleheight=1);
plt.show()
```

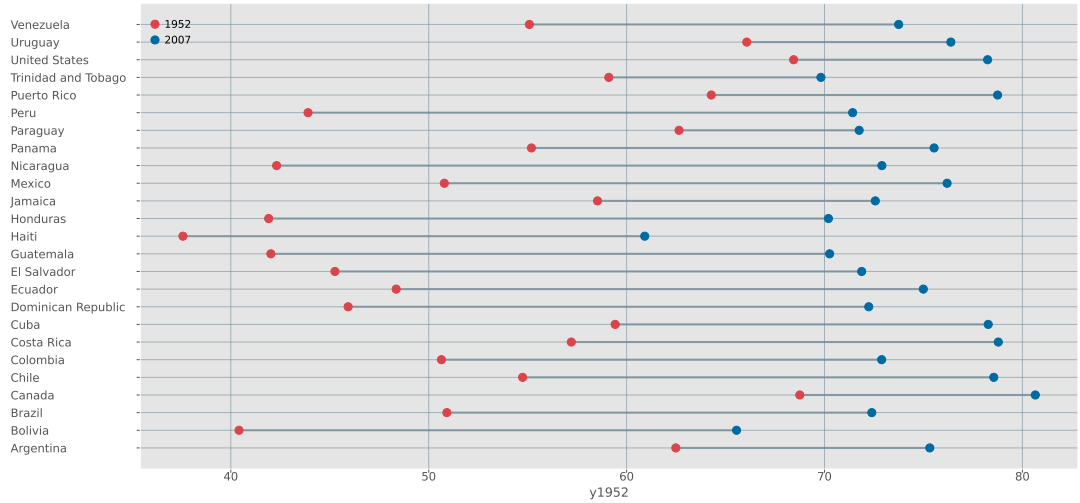


Figure 6.7: Simple dumbbell chart

The graph will be easier to read if the countries are sorted and the points are sized and colored. In the next graph, we'll sort by 1952 life expectancy, and modify the line and point size, color the points, add titles and labels, and simplify the theme.

```
# Sort values by
plotdata_wide = plotdata_wide.sort_values(by= "y1952")

# create dumbbell plot
fig, ax = plt.subplots(figsize=(16,8), facecolor = "white")
ax.grid(which="major", axis='both', color='#758D99', alpha=0.6, zorder=1);
ax.hlines(y=plotdata_wide["country"],xmin=plotdata_wide["y1952"],
          xmax=plotdata_wide["y2007"],color='#758D99',zorder=2,linewidth=2,
          label='_nolegend_', alpha=.8);
ax.scatter(plotdata_wide["y1952"],plotdata_wide["country"],label='y1952', s=60,
           color='#DB444B', zorder=3);
ax.scatter(plotdata_wide["y2007"],plotdata_wide["country"],label='y2007', s=60,
```

```
            color='#006BA2', zorder=3);
ax.xaxis.set_tick_params(labeltop=False,labelbottom=True,bottom=True,
                         labelsize=11,pad=-1);
ax.set_xlabel("Life Expectancy (years)");
ax.set_yticks(plotdata_wide["country"]);
ax.set_yticklabels(plotdata_wide["country"],ha = 'left');
ax.yaxis.set_tick_params(pad=120,labelsize=11);
ax.legend(['1952', '2007'], loc=(0,0.9), ncol=1, frameon=False,
          handletextpad=-0.1, handleheight=1);
ax.set_title("Change in Life Expectancy",ha='right',weight='bold',fontsize=11);
ax.text(x=-0.04,y=1.01,s="1952 to 2007",ha='right',fontsize=11);
plt.show()
```

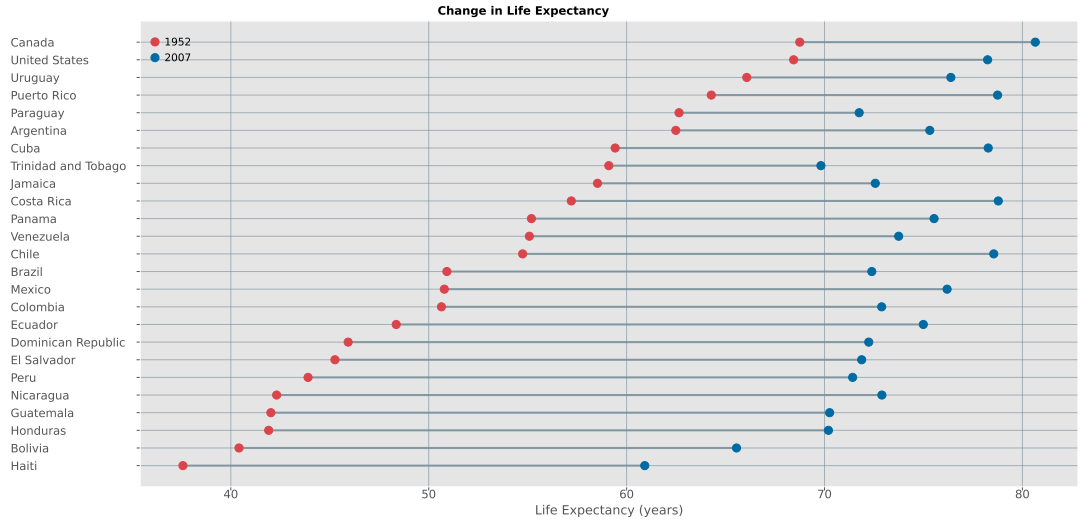


Figure 6.8: Sorted, colored dumbbell chart

It is easier to discern patterns here. For example Haiti started with the lowest life expectancy in 1952 and still has the lowest in 2007. Paraguay started relatively high by has made few gains.

6.3 Slope graphs

When there are several groups and several time points, a slope graph can be helpful. Let's plot life expectancy for six Central American countries in 1992, 1997, 2002 and 2007. Again we'll use the gapminder data.

```
# Filter
condition = '''
        year in [1992, 1997, 2002, 2007] and \
        country in ['Panama','Costa Rica',
       'Nicaragua','Honduras','El Salvador','Guatemala','Belize']
'''
df = gapminder >> query(condition.replace('\n', ''))
```

```
colors = ["green","red","yellow","blue","black","darkmagenta","purple"]

#plot the chart
import numpy as np
fig, ax = plt.subplots()
for i, name in enumerate(np.unique(df.country)):
    temp = df[df['country'] == name]
    plt.plot(temp.year, temp.lifeExp,color=colors[i],marker='o',
             markersize=5);
    # start label
    plt.text(temp.year.values[0]-0.4,temp.lifeExp.values[0],name,
             color=colors[i],ha='right', va='center');
    # end label
    plt.text(temp.year.values[-1]+0.4,temp.lifeExp.values[-1],name,
             color=colors[i],ha='left', va='center');
ax.set_title("Life Expectancy by Country",fontweight="bold");
fig.tight_layout()
fig.patch.set_visible(False);
ax.axis('off');
plt.show()
```

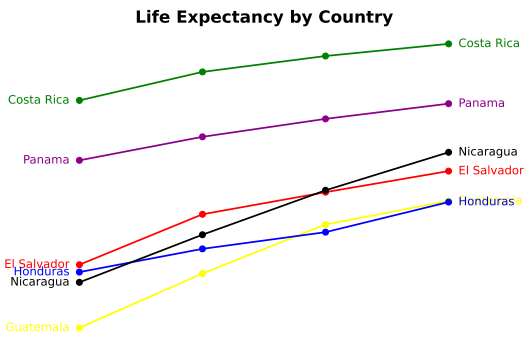


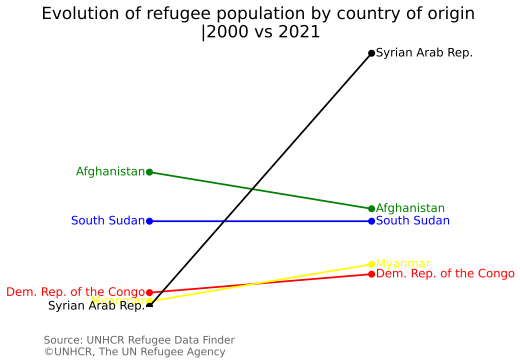
Figure 6.9: slope graph

In the graph above, Costa Rica has the highest life expectancy across the range of years studied. Guatemala has the lowest, and caught up with Honduras (also low at 69) in 2002.

Slope charts can be used to show change in a 'before and after' story by comparing their values at different points in time. It is a great way to show change or difference when there are only two data points. It works well with both continuous data and categorical data.

```
#load and reshape the data
import pandas as pd

df = pd.read_excel("./donnee/slope_2.xlsx")
#set a list of country names
countries=['Afghanistan','Dem. Rep. of the Congo','Myanmar',
           'South Sudan','Syrian Arab Rep.']
#plot the chart
fig, ax = plt.subplots()
for i, name in enumerate(countries):
```

```
    temp = df[df['country_origin'] == name]
    plt.plot(temp.year, temp.refugees_number,color=colors[i],marker='o',
             markersize=5);
    # start label
    plt.text(temp.year.values[0]-0.4,temp.refugees_number.values[0], name,
             color=colors[i],ha='right', va='center');
    # end label
    plt.text(temp.year.values[1]+0.4,temp.refugees_number.values[1], name,
             color=colors[i],ha='left', va='center');
ticks = plt.xticks([2000, 2021])
xl = plt.xlim(1990,2031)
yl = plt.ylim(0, 7*1e6)
ax.set_title('Evolution of refugee population by country of origin \n|2000 vs 2021')
ax.set_ylabel('Number of people (millions)');
def number_formatter(x, pos):
    if x >= 1e6:
        s = '{:1.0f}M'.format(x*1e-6)
    elif x < 1e6 and x > 0:
        s = '{:1.0f}K'.format(x*1e-3)
    else:
        s = '{:1.0f}'.format(x)
    return s
ax.yaxis.set_major_formatter(number_formatter);
plt.annotate('Source: UNHCR Refugee Data Finder', (0,0), (0, -25),
             xycoords='axes fraction', textcoords='offset points',
             va='top', color = '#666666', fontsize=9);
plt.annotate('©UNHCR, The UN Refugee Agency', (0,0), (0, -35),
             xycoords='axes fraction', textcoords='offset points', va='top',
             color = '#666666', fontsize=9);
fig.tight_layout();
plt.grid(True);
fig.patch.set_visible(False);
ax.axis('off');
plt.show()
```



6.4 Area Charts

A simple area chart is basically a line graph, with a fill from the line to the x-axis.

```
# basic area chart
print((economics >> ggplot(aes(x = "date", y = "psavert")) +
  geom_area(fill="lightblue", color="black") +
  labs(title = "Personal Savings Rate",x = "Date",y = "Personal Savings Rate")))
```

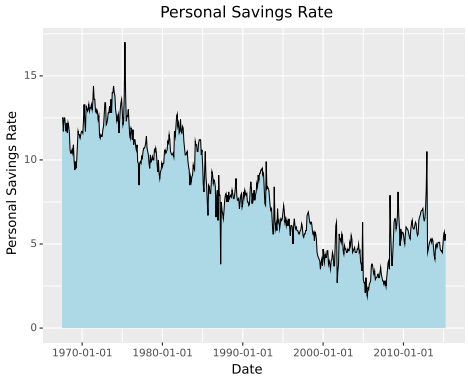


Figure 6.10: Basic area chart

A stacked area chart can be used to show differences between group over time. Consider the uspopage dataset from the gcookbook package. We'll plot the age distribution of the US population from 1900 and 2002.

```
# Load uspopage dataset
data(uspopage, package = "gcookbook")
```

```
# stacked area chart
print((ggplot(r.uspopage, aes(x = "Year",y = "Thousands", fill = "AgeGroup")) +
       geom_area() + labs(title = "US Population by age",x = "Year",
                          y = "Population in Thousands")+
       theme(legend_position = (0.2, 0.675),legend_direction='vertical')))
```

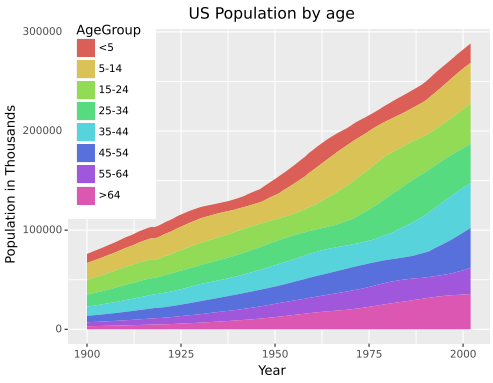


Figure 6.11: Stacked area chart

It is best to avoid scientific notation in your graphs. How likely is it that the average reader will know that 3e+05 means 300,000,000? It is easy to change the scale in plotnine. Simply divide the Thousands variable by 1000 and report it as Millions. While we are at it, let's

- create black borders to highlight the difference between groups

- reverse the order the groups to match increasing age
- improve labeling
- choose a different color scheme
- choose a simpler theme.

The levels of the AgeGroup variable can be reversed using the reorder_categories function in the pandas package.

```
r.uspopage["AgeGroup"] = (r.uspopage["AgeGroup"].cat
.reorder_categories([">64","55-64","45-54","35-44","25-34","15-24","5-14","<5"]))

# Stacked are chart
print((ggplot(r.uspopage, aes(x = "Year",y = "Thousands/1000",fill = "AgeGroup")) +
       geom_area(color = "black") +
       labs(title = "US Population by age (1900 to 2002)",
            caption = "source: U.S. Census Bureau, 2003, HS-3",x = "Year",
             y = "Population in Millions",fill = "Age Group") +
  scale_fill_brewer(type="qual",palette = "Set2") +theme_minimal()+
  theme(legend_position =(0.2, 0.675),legend_direction='vertical')))
```

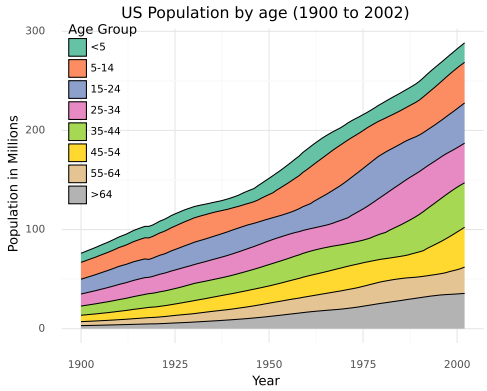


Figure 6.12: Stacked area chart with simpler scale

Apparently, the number of young children have not changed very much in the past 100 years.

Stacked area charts are most useful when interest is on both (1) group change over time and (2) overall change over time. Place the most important groups at the bottom. These are the easiest to interpret in this type of plot.

7 Statistics Models

A statistical model describes the relationship between one or more explanatory variables and one or more response variables. Graphs can help to visualize these relationships. In this section we'll focus on modles that have a single response variable that is either quantitative (a number) or binary (yes/no).

7.1 Correlation plots

Correlation plots help you to visualize the pairwise relationships between a set of quantitative variables by displaying their correlations using color or shading.

Consider the Saratoga Houses dataset, which contains the sale price and characteristics of Saratoga County, NY homes in 2006. In order to explore the relationships among the quantitative variables, we can calculate the Pearson Product-Moment correlation coefficients.

```
# Load datatset
data(SaratogaHouses, package="mosaicData")
```

```
# select numeric variables
from plydata import *
df = (r.SaratogaHouses >> select_if("is_numeric"))
```

```
# calulate the correlations
corr =  df.corr(method="pearson")
```

Table 7.1: Correlation matrix

	price	lotSize	age	landValue	livingArea	pctCollege	bedrooms	fireplaces	bathrooms	rooms
price	1.00	0.16	-0.19	0.58	0.71	0.20	0.40	0.38	0.60	0.53
lotSize	0.16	1.00	-0.02	0.06	0.16	-0.03	0.11	0.09	0.08	0.14
age	-0.19	-0.02	1.00	-0.02	-0.17	-0.04	0.03	-0.17	-0.36	-0.08
landValue	0.58	0.06	-0.02	1.00	0.42	0.23	0.20	0.21	0.30	0.30
livingArea	0.71	0.16	-0.17	0.42	1.00	0.21	0.66	0.47	0.72	0.73
pctCollege	0.20	-0.03	-0.04	0.23	0.21	1.00	0.16	0.25	0.18	0.16
bedrooms	0.40	0.11	0.03	0.20	0.66	0.16	1.00	0.28	0.46	0.67
fireplaces	0.38	0.09	-0.17	0.21	0.47	0.25	0.28	1.00	0.44	0.32
bathrooms	0.60	0.08	-0.36	0.30	0.72	0.18	0.46	0.44	1.00	0.52
rooms	0.53	0.14	-0.08	0.30	0.73	0.16	0.67	0.32	0.52	1.00

The ggcorrplot function in the ggcorrplot package can be used to visualize these correlations. By default, it creates a `plotnine` graph were darker red indicates stronger positive correlations, darker blue indicates stronger negative correlations and white indicates no correlation.

```
from ggcorrplot import *
print(ggcorrplot(df))
```

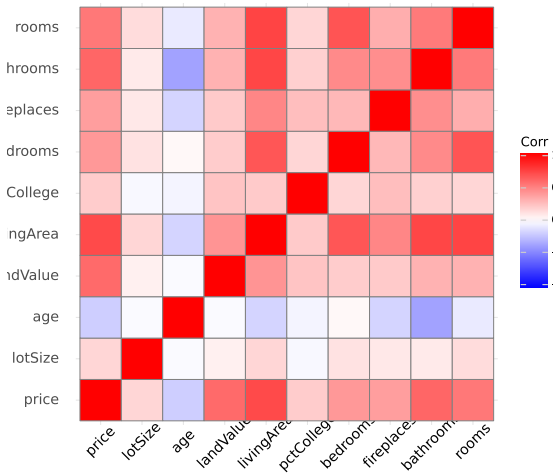


Figure 7.1: Correlation matrix

From the graph, an increase in number of bathrooms and living area are associated with increased price, while older homes tend to be less expensive. Older homes also tend to have fewer bathrooms.

The ggcorrplot function has a number of options for customizing the output. For example

- `hc_order = True` reorders the variables, placing variables with similar correlation patterns together.
- `type= "lower"` plots the lower portion of the correlation matrix.
- `lab = True` overlays the correlation coefficients (as text) on the plot.

```
print(ggcorrplot(df,hc_order = True, type = "lower",lab = True))
```

This, and other options, can make the graph easier to read and interpret.

7.2 Linear Regression

Linear regression allows us to explore the relationship between a quantitative response variable and an explanatory variable while other variables are held constant.

Consider the prediction of home prices in the Saratoga dataset from lot size (square feet), age (years), land value (1000s dollars), living area (square feet), number of bedrooms and bathrooms and whether the home is on the waterfront or not.

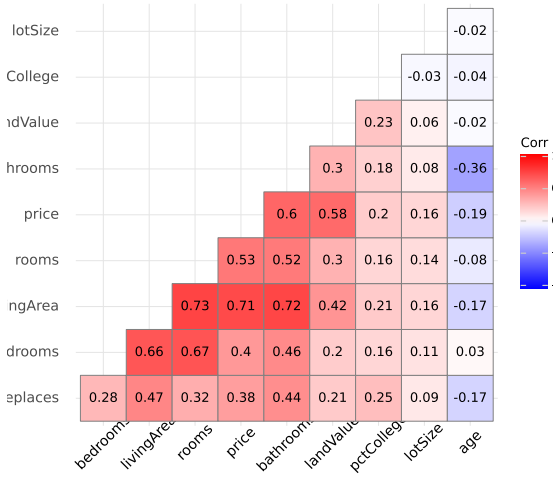


Figure 7.2: Sorted lower triangel correlation matrix with options

```
# Linear regression
from statsmodels.api import OLS

formul="price~lotSize+age+landValue+livingArea+bedrooms+bathrooms+waterfront"
lm_model = OLS.from_formula(formula = formul, data = r.SaratogaHouses).fit()
lm_res = lm_model.summary2().tables[1]
```

Table 7.2: Linear regression coefficients

	Coef.	Std.Err.	t	P>|t|	[0.025	0.975]
Intercept	139878.80	16472.93	8.49	0.00	107569.72	172187.88
waterfront[T.No]	-120726.62	15600.83	-7.74	0.00	-151325.21	-90128.03
lotSize	7500.79	2075.14	3.61	0.00	3430.74	11570.85
age	-136.04	54.16	-2.51	0.01	-242.26	-29.82
landValue	0.91	0.05	19.84	0.00	0.82	1.00
livingArea	75.18	4.16	18.08	0.00	67.02	83.33
bedrooms	-5766.76	2388.43	-2.41	0.02	-10451.30	-1082.22
bathrooms	24547.11	3332.27	7.37	0.00	18011.38	31082.83

From the results, we can estimate that an increase of one square foot of living area is associated with a home price increase of $75, holding the other variables constant. Additionally, waterfront home cost approximately $120,726 more than non-waterfront home, again controlling for the other variables in the model.

Let's plot conditional relationships.

```
# Store endog variables and livingArea
df = pd.DataFrame({
  "price": lm_model.model.endog,"livingArea" : r.SaratogaHouses.livingArea})

from plotnine import *

# Plot
print((ggplot(data = df,mapping = aes(x = "livingArea", y = "price"))+
    geom_point() +
```

```
    geom_smooth(method = "lm",se = True,color = "blue",fill = "gray")+
    ylim(0, 800000)))
```

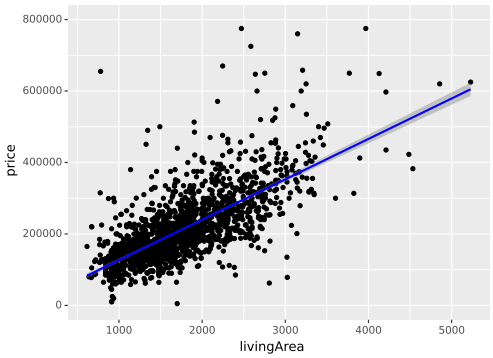


Figure 7.3: Conditional plot of living area and price

The graph suggests that, after controlling for lot size, age, living area, number of bedrooms and bathrooms, and waterfront location, sales price increases with living area in a linear fashion.

7.3 Logistic regression

Logistic regression can be used to explore the relationship between a binary response variable and an explanatory variable while other variables are held constant. Binary response variables have two levels (yes/no, lived/died, pass/fail, malignant/benign).

```
# Load Titanic dataset
import seaborn as sns

titanic = sns.load_dataset("titanic")
titanic = titanic >> select("survived","pclass","sex","age","embark_town")
titanic = titanic.dropna()

# fit logistic model for predicting survived died/alive
import statsmodels.formula.api as smf
log_reg=smf.logit("survived~sex+age+embark_town",data=titanic).fit(disp=False)
log_res = log_reg.summary2().tables[1]
```

Table 7.3: Logistic regression coefficients

	Coef.	Std.Err.	z	P>|z|	[0.025	0.975]
Intercept	2.20	0.32	6.85	0.00	1.57	2.84
sex[T.male]	-2.48	0.19	-12.98	0.00	-2.85	-2.10
embark_town[T.Queenstown]	-1.82	0.54	-3.39	0.00	-2.86	-0.77
embark_town[T.Southampton]	-1.01	0.24	-4.25	0.00	-1.47	-0.54
age	-0.01	0.01	-1.23	0.22	-0.02	0.00

7.3.1 Setting a reference or base level for categorical variables

With Categorical Variables, you'll sometimes wan to set the reference category to be a specific value. This ca, help make the results more interpretable.

In the Titanic Dataset used above, we could examine how likely survival was for first-class passengers relative to third-class. We can do this with Patsy's **categorical treatments**.

In the Titanic dataset, the `pclass` column gets interpreted as an integer.We change this by wrapping it in an uppercase `C` and parentheses `()`.

```
# Fit logistic regression between survived and pclass
log_reg = smf.logit("survived ~ C(pclass)",data=titanic).fit(disp=False)
log_res = log_reg.summary2().tables[1]
```

Table 7.4: Logistic regression with categorical variables

	Coef.	Std.Err.	z	P>|z|	[0.025	0.975]
Intercept	0.63	0.15	4.06	0	0.33	0.93
C(pclass)[T.2]	-0.71	0.22	-3.27	0	-1.14	-0.28
C(pclass)[T.3]	-1.78	0.20	-8.99	0	-2.17	-1.40

Notice, though, this only signals to Patsy to treat `pclass` as categorical. The reference level hasn't changed. To set the reference level, we include a `Treatment` argument with a `reference` set to the desired value.

```
# Fit logistic regression between survived and pclass with treatment
formula =  "survived~C(pclass,Treatment(reference=3))"
log_reg=smf.logit(formula,data=titanic).fit(disp=False)
log_res = log_reg.summary2().tables[1]
```

Table 7.5: Logistic regression with categorical variables

	Coef.	Std.Err.	z	P>|z|	[0.025	0.975]
Intercept	-1.16	0.12	-9.29	0	-1.40	-0.91
C(pclass, Treatment(reference=3))[T.1]	1.78	0.20	8.99	0	1.40	2.17
C(pclass, Treatment(reference=3))[T.2]	1.07	0.20	5.47	0	0.69	1.46

For more on categorical treatments, see here and here from the Patsy docs.

7.4 Survival plots

In many research settings, the response variable is the time to an event. This is frequently true in healthcare research, where we are interested in time to recovery, time to death, or time to relapse.

If the event has not occurred for an observation (either because the study ended or the patient dropped out) the observation is said to be `censored`.

7.4.1 The Veterans' Administration Lung Cancer Trial

The Veterans' Administration Lung Cancer Trial is a randomized trial of two treatment regimens for lung cancer. The data set (Kalbfleisch J. and Prentice R, (1980) The Statistical Analysis of Failure Time Data.New York: Wiley) consists of 137 patients and 8 variables, which are described below:

- `Treatment` : denotes the type of lung cancer treatment; `standard` and `test` drug.

- `Celltype` : denotes the type of cell involved; `squamous`, `small cell`, `adeno`, `large`
- `Karnofsky_score` : is the Karnofsky score
- `Diag` : is the time since diagnosis in months.
- `Age` : is the age of years.
- `Prior_Therapy` : denotes any prior therapy; `none` or `yes`
- `Status` : denotes the status of the patient as dead or alive; `dead` or `alive`.
- `Survival_in_days` : is the survival time in days since the treatment.

7.4.2 Survival Data

Survival times are subject to right-censoring, therefore, we need to consider an individual's status in addirion to survival time. To be fully compatible with scikit-learn, `Status` and `Survival_in_days` need to be stored as a structured array with the first field indicating whether the actual survival time was observed or if was censored, and the second field denoting the observed survival time, which corresponds to the time of death (if `Status ==dead`, $\delta = 1$) or the last time that person was contacted (if `Status == alive`, $\delta = 0$).

```
from sksurv.datasets import load_veterans_lung_cancer

data_x, data_y = load_veterans_lung_cancer()
pd.DataFrame(data_y)
```

```
##       Status  Survival_in_days
##       <bool>         <float64>
## 0       True              72.0
## 1       True             411.0
## 2       True             228.0
## 3       True             126.0
## ..       ...               ...
## 4       True             118.0
## 132     True             133.0
## 133     True             111.0
## 134     True             231.0
## 135     True             378.0
## 136     True              49.0
##
## [137 rows x 2 columns]
```

We can easily see that only a few survival times are right censored (status is false), i.e., most veteran's died during the study period (Status is True).

7.4.3 The Survival Function

A key quantity in survival analysis is the so-called survival function, which relates time to the probability of surviving beyond a given time point.

> Let T denote a continuous non-negative random variable corresponding to a patient's survival time. The survival function $S(t)$ returns the probability of survival beyond time t and is defined as
>
> $$S(t) = \mathbb{P}(T > t)$$

If we observed the exact survival time of all subjects, i.e., everyone died before the study ended, the survival function at time can simply be estimated by the ratio of patients surviving beyond time and the total number of patients:

$$\widehat{S}(t) = \frac{\text{number of patients surviving beyong } t}{\text{total number of patients}}$$

In the presence of censoring, this estimator cannot be used, because the numerator is not always defined. For instance, consider the following set of patients:

```
import pandas as pd

pd.DataFrame.from_records(data_y[[11, 5, 32, 13, 23]], index=range(1, 6))
```

```
##    Status  Survival_in_days
##    <bool>         <float64>
## 1    True               8.0
## 2    True              10.0
## 3    True              20.0
## 4   False              25.0
## 5    True              59.0
```

Using the formula from above, we can compute $\widehat{S}(t = 11) = \frac{3}{5}$, but not $\widehat{S}(t = 30)$, because we don't know whether the 4th patient is still alive at $t = 30$, all we know is that when we last checked at $t = 25$,the patient was still alive.

An estimator, similar to the one above, that is valid if survival times are right-censored is the Kaplan-Meier estimator.

```
import matplotlib.pyplot as plt
from sksurv.nonparametric import kaplan_meier_estimator

time, survival_prob = kaplan_meier_estimator(data_y["Status"],
                                             data_y["Survival_in_days"])
fig, ax = plt.subplots(figsize=(16,6))
ax.step(time, survival_prob, where="post");
ax.set_ylabel("est. probability of survival $\widehat{S}(t)$");
ax.set_xlabel("time $t$");
ax.grid(True);
plt.show()
```

The estimated curve is a step function, with steps occurring at time points where one or more patients died. From the plot we can see that most patients died in the first 200 days, as indicated by the steep slope of the estimated survival function in the first 200 days.

7.4.4 Considering other variables by stratification

7.4.4.1 Survival functions by treatment

Patients enrolled in the Veterans' Administration Lung Cancer Trial were randomized to one of two treatments: `standard` and a new `test` drug. Next, let's have a look at how many patients underwent the standard treatment and how many received the new drug.

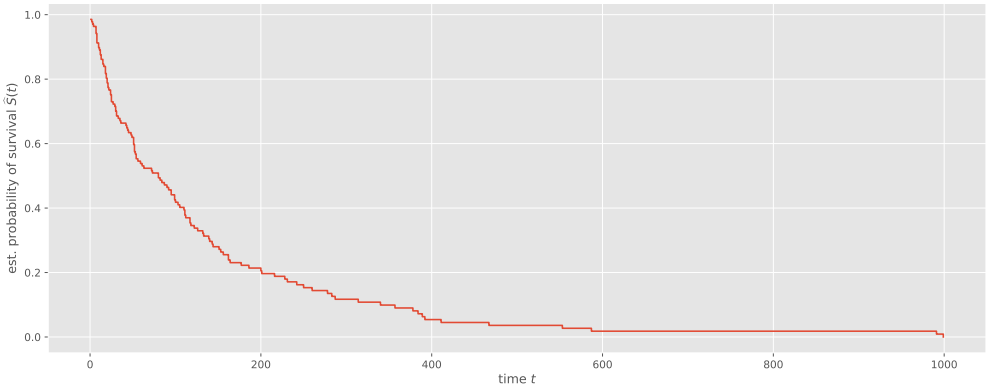


Figure 7.4: est. probability of survival $\widehat{S}(t)$

```
data_x["Treatment"].value_counts()
```

```
## standard    69
## test        68
## Name: Treatment, dtype: int64
```

Roughly half the patients received the alternative treatment.

The obvious questions to ask is: *> Is there any difference in survival between the two treatment groups?*

As a first attempt, we can estimate the survival function in both treatment groups separately.

```
fig, ax = plt.subplots(figsize=(16,6))
for treatment_type in ("standard", "test"):
    mask_treat = data_x["Treatment"] == treatment_type
    time_treatment, survival_prob_treatment = kaplan_meier_estimator(
        data_y["Status"][mask_treat],
        data_y["Survival_in_days"][mask_treat])
    ax.step(time_treatment, survival_prob_treatment, where="post",
            label="Treatment = %s" % treatment_type);
ax.set_ylabel("est. probability of survival $\widehat{S}(t)$");
ax.set_xlabel("time $t$");
ax.legend(loc="best");
ax.grid(True);
plt.show()
```

Unfortunately, the results are inconclusive, because the difference between the two estimated survival functions is too small to confidently argue that the drug affects survival or not.

7.4.5 Survival functions by cell type

Next, let's have a look at the cell type, which has been recorded as well, and repeat the analysis from above.

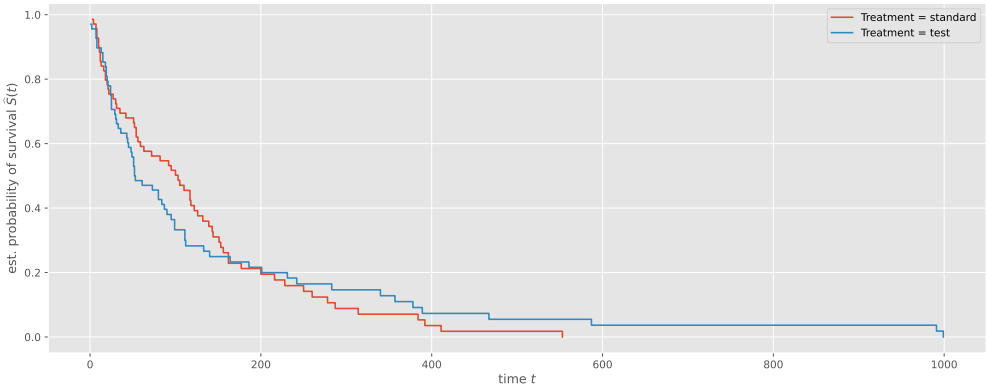


Figure 7.5: est. probability of survival $\widehat{S}(t)$

```
fig, ax = plt.subplots(figsize=(16,6))
for value in data_x["Celltype"].unique():
    mask = data_x["Celltype"] == value
    time_cell,survival_prob_cell = kaplan_meier_estimator(data_y["Status"][mask],
                                                          data_y["Survival_in_days"][mask])
    ax.step(time_cell, survival_prob_cell, where="post",
            label="%s (n = %d)" % (value, mask.sum()))
ax.set_ylabel("est. probability of survival $\widehat{S}(t)$");
ax.set_xlabel("time $t$");
ax.legend(loc="best");
ax.grid(True);
plt.show()
```

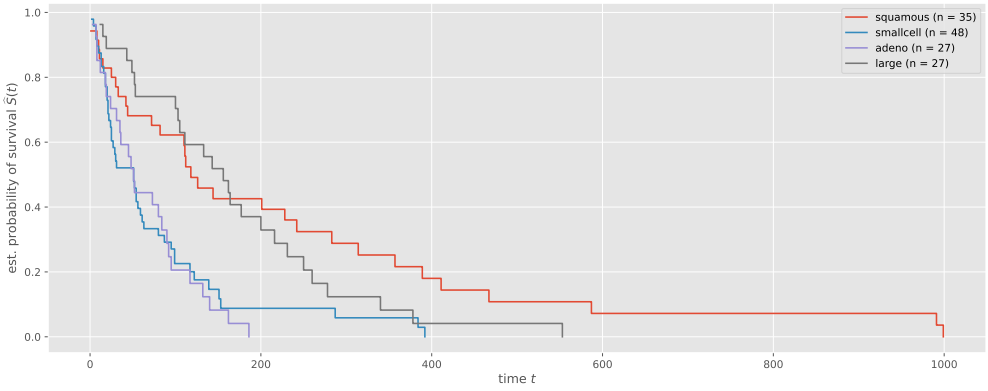


Figure 7.6: est. probability of survival $\widehat{S}(t)$

In this case, we observe a pronounced difference between two groups. Patients with squamous or large cells seem to have a better prognosis compared to patients with small or adeno cells.

7.4.6 Multivariate Survival Models

In the Kaplan-Meier approach used above, we estimated multiple survival curves by dividing the dataset into smaller sub-groups according to a variable. If we want to consider more than 1

or 2 variables, this approach quickly becomes infeasible, because subgroups will get very small. Instead, we can use a linear model, Cox's proportional hazard's model, to estimate the impact each variable has on survival.

First however, we need to convert the categorical variables in the data set into numeric values.

```
from sksurv.preprocessing import OneHotEncoder

sc = OneHotEncoder()
sc.set_output(transform="pandas");
data_x_numeric = sc.fit_transform(data_x)
data_x_numeric.head()
```

```
##    Age_in_years  Celltype=large  ...  Prior_therapy=yes  Treatment=test
##       <float64>       <float64>  ...          <float64>       <float64>
## 0          69.0             0.0  ...                0.0             0.0
## 1          64.0             0.0  ...                1.0             0.0
## 2          38.0             0.0  ...                0.0             0.0
## 3          63.0             0.0  ...                1.0             0.0
## 4          65.0             0.0                     1.0             0.0
##
## [5 rows x 8 columns]
```

Survival models in scikit-survival follow the same rules as estimators in scikit-learn, i.e., they have a fit method, which expects a data matrix and a structured array of survival times and binary event indicators.

```
from sklearn import set_config
from sksurv.linear_model import CoxPHSurvivalAnalysis

set_config(display="text")  # displays text representation of estimators

estimator = CoxPHSurvivalAnalysis()
res = estimator.fit(data_x_numeric, data_y)
```

The result is a vector of coefficients, one for each variable, where each value corresponds to the log hazard ratio.

```
# Coefficients
coef = pd.Series(res.coef_, index=data_x_numeric.columns).to_frame("coefficients")
```

Table 7.6: Coefficients

	coefficients
Age_in_years	-0.0085
Celltype=large	-0.7887
Celltype=smallcell	-0.3318
Celltype=squamous	-1.1883
Karnofsky_score	-0.0326
Months_from_Diagnosis	-0.0001
Prior_therapy=yes	0.0723
Treatment=test	0.2899

Using the fitted model, we can predict a patient-specific survival function, by passing an appropriate data matrix to the estimator's *predict_survival_function* method.

```
x_new = pd.DataFrame.from_dict({
    1: [65, 0, 0, 1, 60, 1, 0, 1],
    2: [65, 0, 0, 1, 60, 1, 0, 0],
    3: [65, 0, 1, 0, 60, 1, 0, 0],
    4: [65, 0, 1, 0, 60, 1, 0, 1]},
     columns=data_x_numeric.columns, orient='index')
```

Table 7.7: New data

Age_in_years	Celltype=large	Celltype=smallcell	Celltype=squamous	Karnofsky_score	Months_from_Diagnosis	Prior_therapy=yes	Treatment=test
65	0	0	1	60	1	0	1
65	0	0	1	60	1	0	0
65	0	1	0	60	1	0	0
65	0	1	0	60	1	0	1

Similar to kaplan_meier_estimator, the predict_survival_function method returns a sequence of step functions, which we can plot.

```
# Prediction
pred_surv = estimator.predict_survival_function(x_new)
time_points = np.arange(1,1000)
fig, ax = plt.subplots(figsize=(16,6))
for i, surv_func in enumerate(pred_surv):
    ax.step(time_points, surv_func(time_points), where="post",
            label="Sample %d" % (i + 1));
ax.set_ylabel("est. probability of survival $\widehat{S}(t)$");
ax.set_xlabel("time $t$");
ax.legend(loc="best");
ax.grid(True);
plt.show()
```

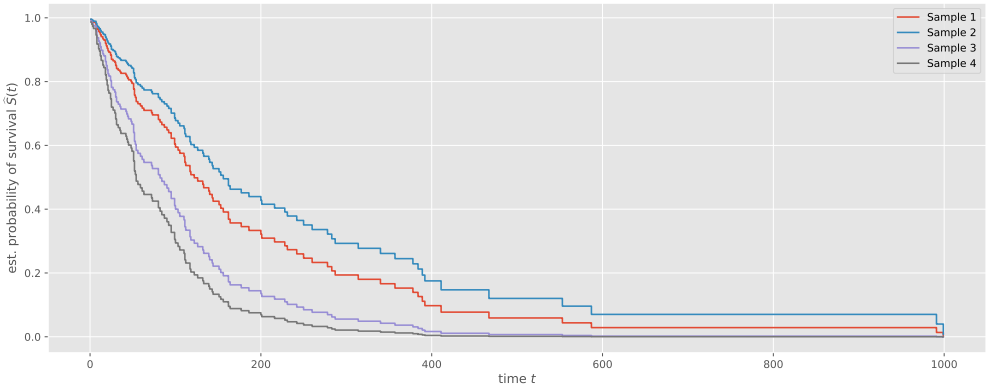


Figure 7.7: est. probability of survival $\widehat{S}(t)$

For more click here.

7.5 Mosaic plots

Mosaic charts can display the relationship between categorical variables using rectangles whose areas represent the proportion of cases for any given combination of levels. The color of the tiles can also indicate the degree relationship among the variables.

the `plotnine` package can't create a mosaic plot. However, you can create a mosaic plot with the mosaic function in the `statsmodels` package.

People are fascinated with the Titanic (or is it with Leo?). In the Titanic disaster, what role did sex and class play in survival? We can visualize the relationship between these three categorical variables using the code below.

```
# Mosaic plot
from statsmodels.graphics.mosaicplot import mosaic

fig = mosaic(data=titanic, index=['survived', 'pclass', 'sex'])
fig[0]
```

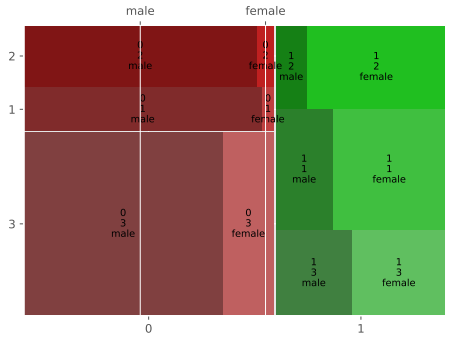


Figure 7.8: Basic mosaic plot

8 Other Graphs

Graphs in this chapter can be very useful, but don't fit in easily within the other chapters.

8.1 3-D scatterplot

The `plotnine` package can't create a 3-D plot. However, you can create a 3-D scatterplot with the `scatter` function in the `matplotlib` package.

Let's say that we want to plot automobile mileage vs. engine displacement vs. car weight using the data in the `mtcars` dataframe.

```
import matplotlib.pyplot as plt
from plotnine.data import mtcars

fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(projection='3d')
ax.scatter(mtcars.disp, mtcars.wt, mtcars.mpg, marker= "o",color="black");
ax.set_xlabel('disp');
ax.set_ylabel('wt')
ax.set_zlabel('mpg')
ax.set_title("3-D Scatterplot Example 1")
plt.show()
```

Now lets, modify the graph by replacing the points with filled blue circles, add drop lines to the x-y plane, and create more meaningful labels.

```
import numpy as np
fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(projection='3d')
ax.scatter(mtcars.disp, mtcars.wt, mtcars.mpg, marker= "o",color="blue");
z2 = np.ones(shape=mtcars.mpg.shape[0])*min(mtcars.mpg)
for i, j, k, h in zip(mtcars.disp,mtcars.wt,mtcars.mpg,z2):
  ax.plot([i,i],[j,j],[k,h],color="blue");
ax.set_xlabel("Displacement (cu. in.)");
ax.set_ylabel("Weight (lb/1000)")
ax.set_zlabel("Miles/(US) Gallon")
```

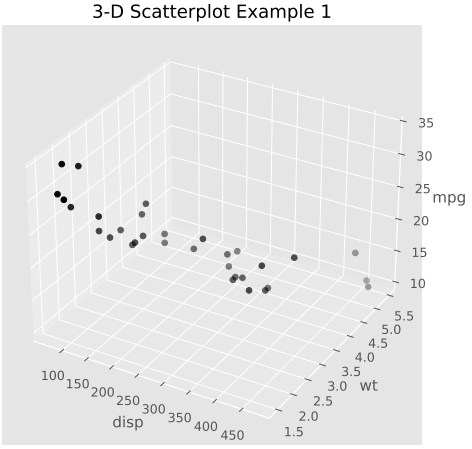


Figure 8.1: Basic 3-D scatterplot

```
ax.set_title("3-D Scatterplot Example 2")
plt.show()
```

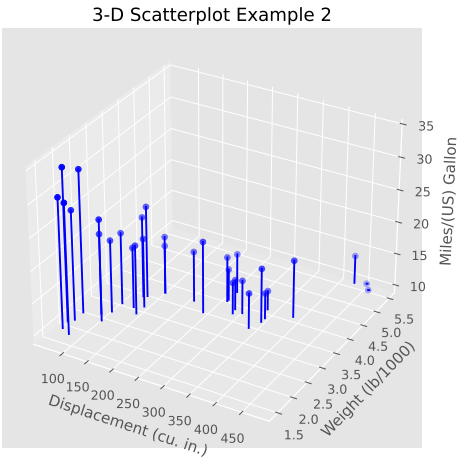


Figure 8.2: 3-D scatterplot with vertical lines

Next, let's label the points.

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(projection='3d')
ax.scatter(mtcars.disp, mtcars.wt, mtcars.mpg,marker= "o",color="blue");
z2 = np.ones(shape=mtcars.mpg.shape[0])*min(mtcars.mpg)
for i, j, k, h in zip(mtcars.disp,mtcars.wt,mtcars.mpg,z2):
  ax.plot([i,i],[j,j],[k,h],color="blue");
for i, name in enumerate(mtcars.name):
  ax.text(mtcars.disp[i],mtcars.wt[i],mtcars.mpg[i],name,color="black",fontsize=7);
```

```
ax.set_xlabel("Displacement (cu. in.)");
ax.set_ylabel("Weight (lb/1000)");
ax.set_zlabel("Miles/(US) Gallon");
ax.set_title("3-D Scatterplot Example 3");
plt.show()
```

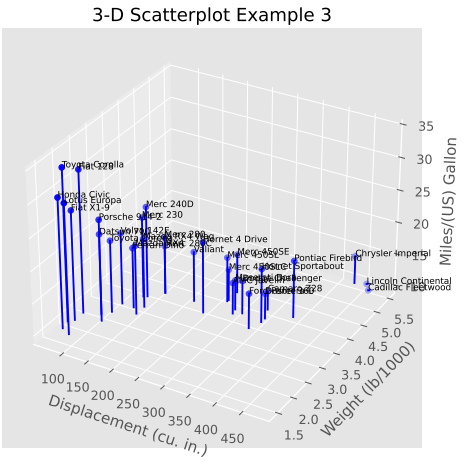


Figure 8.3: 3-D scatterplot with vertical lines and point labels

Almost there. As a final step, we will add information on the number of cylinders in each car. To do this, we'll add a column to the `mtcars` dataframe indicating the color for each point.

```
# create column indicating point color
from plydata import *
df = (mtcars >>
         define(pcolor = if_else('cyl == 4','"red"',
                                 if_else('cyl == 6','"blue"','"darkgreen"'))))
fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(projection='3d')
labels = np.unique(df["pcolor"])
for lbl, col in zip(labels,labels):
  temp = df[df["pcolor"] == lbl]
  ax.scatter(temp.disp,temp.wt,temp.mpg,marker='o',color = col,label = lbl)
  z2 = np.ones(shape=temp.mpg.shape[0])*min(temp.mpg)
  for i, j, k, h in zip(temp.disp,temp.wt,temp.mpg,z2):
    ax.plot([i,i],[j,j],[k,h],color=col,linestyle="dashed");
for i, name in enumerate(df.name):
  ax.text(df.disp[i],df.wt[i],df.mpg[i],name,color="black",fontsize=7);
ax.set_xlabel("Displacement (cu. in.)");
ax.set_ylabel("Weight (lb/1000)");
ax.set_zlabel("Miles/(US) Gallon");
ax.set_title("3-D Scatterplot Example 4");
handles, previous_labels = ax.get_legend_handles_labels()
ax.legend(handles=handles, labels=[6,8,4],title='Number of Cylinders');
plt.show()
```

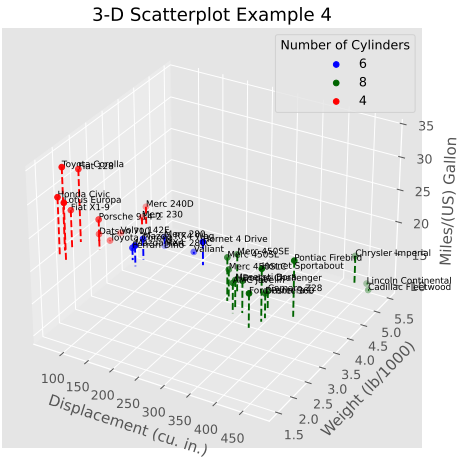


Figure 8.4: 3-D scatterplot with vertical lines and point labels and legend

We can easily see that the car with the highest mileage (Toyota Corolla) has low engine displacement, low weight, and 4 cylinders.

8.2 Biplots

A biplot is a specialized graph that attempts to represent the relationship between observations, between variables, and between observations and variables, in a low (usually two) dimensional space.

```
# Principal Components Analysis - PCA
from scientisttools.decomposition import PCA
from scientisttools.extractfactor import summaryPCA
import pandas as pd

## perform PCA
X =  mtcars.drop(columns=['name'])
res = PCA(normalize= True,row_labels=mtcars.name, col_labels = X.columns).fit(X)
summaryPCA(res,to_markdown=False)
```

```
##                      Principal Component Analysis - Results
##
## Importance of components
##                              Dim.1     Dim.2  ...    Dim.10    Dim.11
##                          <float64> <float64>  ... <float64> <float64>
## Variance                     6.608     2.650  ...     0.052     0.022
## Difference                   3.958     2.023  ...     0.030       NaN
## % of var.                   60.076    24.095  ...     0.473     0.200
## Cumulative of % of var.     60.076    84.172         99.800   100.000
##
## [4 rows x 11 columns]
##
## Individuals (the 10 first)
```

```
##
##                      d(i,G)      p(i)    I(i,G)  ...     Dim.3       ctr      cos2
##                                                  ...
## name               <float64> <float64> <float64>  ... <float64> <float64> <float64>
## Mazda RX4              2.234     0.031     0.156  ...    -0.601     1.801     0.072
## Mazda RX4 Wag          2.081     0.031     0.135  ...    -0.382     0.728     0.034
## Datsun 710             2.987     0.031     0.279  ...    -0.241     0.290     0.007
## Hornet 4 Drive         2.521     0.031     0.199  ...    -0.136     0.092     0.003
## Hornet Sportabout      2.456     0.031     0.189  ...    -1.134     6.412     0.213
## Valiant                3.014     0.031     0.284  ...     0.164     0.134     0.003
## Duster 360             3.187     0.031     0.317  ...    -0.363     0.656     0.013
## Merc 240D              2.841     0.031     0.252  ...     0.944     4.439     0.110
## Merc 230               3.733     0.031     0.435  ...     1.797    16.094     0.232
## Merc 280               1.907     0.031     0.114          1.493    11.103     0.613
##
## [10 rows x 12 columns]
##
## Continues variables
##
##          Dim.1       ctr      cos2  ...     Dim.3       ctr      cos2
##      <float64> <float64> <float64>  ... <float64> <float64> <float64>
## mpg     -0.932    13.143     0.869  ...    -0.179     5.096     0.032
## cyl      0.961    13.981     0.924  ...    -0.139     3.073     0.019
## disp     0.946    13.556     0.896  ...    -0.049     0.378     0.002
## hp       0.848    10.894     0.720  ...     0.111     1.960     0.012
## drat    -0.756     8.653     0.572  ...     0.128     2.598     0.016
## wt       0.890    11.979     0.792  ...     0.271    11.684     0.073
## qsec    -0.515     4.018     0.266  ...     0.319    16.255     0.102
## vs      -0.788     9.395     0.621  ...     0.340    18.388     0.115
## am      -0.604     5.520     0.365  ...    -0.163     4.234     0.027
## gear    -0.532     4.281     0.283  ...     0.229     8.397     0.053
## carb     0.550     4.580     0.303          0.419    27.936     0.175
##
## [11 rows x 9 columns]
```

```
# Individuals plots
from scientisttools.pyplot import plotPCA
fig, axe = plt.subplots(figsize=(6,6))
plotPCA(res,choice = "ind",repel=True,ax=axe)
plt.show()
```

```
# Variables plots
fig, axe = plt.subplots(figsize=(6,6))
plotPCA(res,choice = "var",repel=True,ax=axe)
plt.show()
```

It's easiest to see how this works with an example. Let's create a biplot for the `mtcars` dataset

```
# Biplots
fig = plt.figure(figsize=(6,6))
```

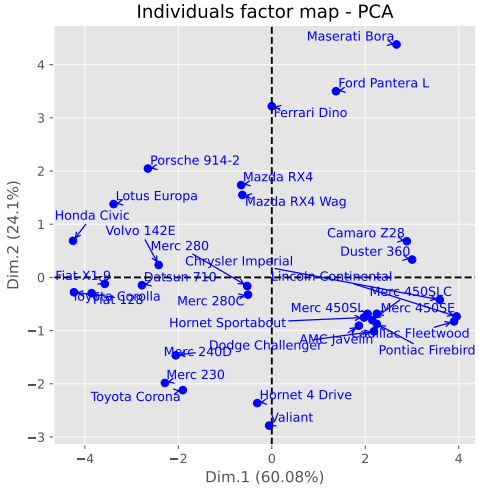


Figure 8.5: Individual plot PCA - mtcars data

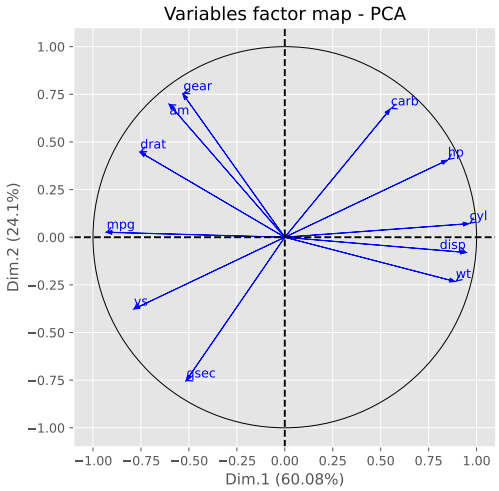


Figure 8.6: Variables plot PCA - mtcars data

```
axe1 = fig.add_subplot(111)
axe2 = axe1.twiny()
axe2 = axe2.twinx()
axe1.set_title("Biplot of mtcars data");
axe1.set_xlabel("Dim1 (60.1%)");
axe1.set_ylabel("Dim2 (24.1%)");
plt.axhline(0,color = "blue", linestyle = "--", linewidth = 0.5);
plt.axvline(0,color = "blue", linestyle = "--", linewidth = 0.5);
axe1.scatter(res.row_coord_[:,0],res.row_coord_[:,1],c = "black",alpha = 1);
# Add row labels
for i,name in enumerate(mtcars.name):
  axe1.text(res.row_coord_[i,0],res.row_coord_[i,1],name,color = "black",
            fontsize= 10);
# Add columns
for j in range(X.shape[1]):
```

```
  axe2.arrow(0,0,res.col_coord_[j,0],res.col_coord_[j,1],head_width = 0.02,
             head_length = 0.02,color = "blue", linestyle = "--");
# Add columns labels
for j,col in enumerate(X.columns):
  axe2.text(res.col_coord_[j,0],res.col_coord_[j,1],col,color = "blue",
            fontsize= 10);
plt.show()
```

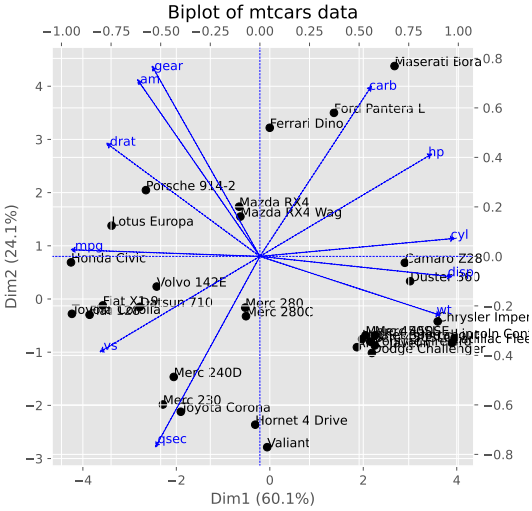


Figure 8.7: Basic biplot

Dim1 and Dim2 are the first two principal components - linear combinations of the original p variables.

$$\begin{cases} PC_1 &= a_{10} + a_{11}x_1 + a_{12}x_2 + \cdots + a_{1p}x_p \\ PC_1 &= a_{20} + a_{21}x_1 + a_{22}x_2 + \cdots + a_{2p}x_p \end{cases}$$

The weights of these linear combinations (a_{ij}) are chosen to maximize the variance accounted for in the original variables. Additionally, the principal components (PCs) are constrained to be uncorrelated with each other.

In this graph, the first PC accounts for 60% of the variability in the original data. The second PC accounts for 24%. Together, they account for 84% of the variability in the original $p = 11$ variables.

As you can see, both the observations (cars) and variables (car characteristics) are plotted in the same graph.

- Points represent observations. Smaller distances between points suggest similar values on the original set of variables. For example, the `Toyota Corolla` and `Honda Civic` are similar to each other, as are the `Chrysler Imperial` and `Lincoln Continental`. However, the `Toyota Corolla` is very different from the `Lincoln Continental`.
- The vectors (arrows) represent variables. The angle between vectors are proportional to the correlation between the variables. Smaller angles indicate stronger correlations. For example, `gear` and `am` are positively correlated, `gear` and `qsec` are uncorrelated (90 degree angle), and `am` and `wt` are negatively correlated (angle greater then 90 degrees).

- The observations that are are farthest along the direction of a variable's vector, have the highest values on that variable. For example, the `Toyoto Corolla` and `Honda Civic` have higher values on `mpg`. The `Toyota Corona` has a higher `qsec`. The `Duster 360` has more cylinders.

Care must be taken in interpreting biplots. They are only accurate when the percentage of variance accounted for is high. Always check your conclusion with the original data.

See the article by Forrest Young to learn more about interpreting biplots correctly.

8.3 Bubble charts

A bubble chart is basically just a scatterplot where the point size is proportional to the values of a third quantitative variable.

Using the mtcars dataset, let's plot car weight vs. mileage and use point size to represent horsepower.

```
# Create a bubble plot
from plotnine import *
print((mtcars >>ggplot(aes(x = "wt", y = "mpg", size = "hp"))+geom_point()+
      theme(legend_position = (0.85, 0.7),legend_direction='vertical')))
```

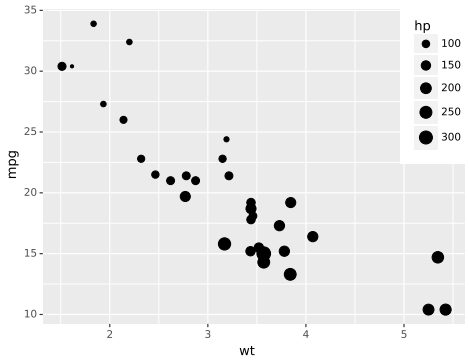


Figure 8.8: Basic bubble plot

We can improve the default appearance by increasing the size of the bubbles, choosing a different point shape and color, and adding some transparency.

```
# create a bubble plot with modifications
print((mtcars >> ggplot(aes(x = "wt", y = "mpg", size = "hp")) +
  geom_point(alpha = 0.5, fill="cornflowerblue", color="black",shape="o") +
  scale_size_continuous(range = (1, 14))+
  labs(title = "Auto mileage by weight and horsepower", x = "Weight (1000 lbs)",
       y = "Miles/(US) gallon",size = "Gross horsepower")+
  theme(legend_position = (0.8, 0.65),legend_direction='vertical')))
```

The `range` parameter in the `scale_size_continuous` function specifies the minimum and maximum size of the plotting symbol. The default is `range = (1, 6)`.

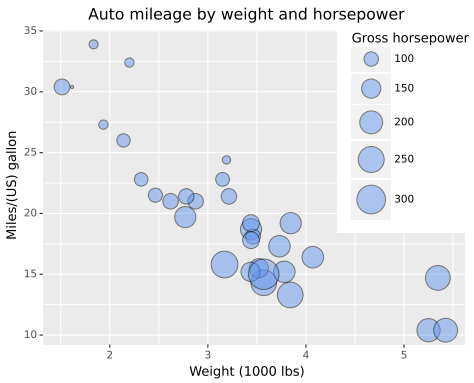


Figure 8.9: Bubble plot with modifications

The `shape` option in the geom_point function specifies an circle with a border color and fill color.

Clearly, miles per gallon decreases with increased car weight and horsepower. However, there is one car with low weight, high horsepower, and high gas mileage. Going back to the data, it's the Lotus Europa.

Bubble charts are controversial for the same reason that pie charts are controversial. People are better at judging length than volume. However, they are quite popular.

8.4 Flow diagrams

A flow diagram represents a set of dynamic relationships. It usually captures the physical or metaphorical flow of people, materials, communications, or objects through a set of nodes in a network.

8.4.1 Sankey diagrams

8.4.2 What is Sankey Diagram?

Sankey Diagram is used to display the flow of some property from various sources to destinations.

The simple diagram has a few **source nodes** plotted at the beginning and a few **destination nodes** at the end of the diagram. Then, there are various **arrows/links** representing the flow of property from sources to destinations. There is one arrow/link per source and destination combination. The **width** of an arrow/link is proportional to the amount of property flowing from source to destination.

The Sankey diagrams can have **intermediate nodes** (plotted between source and destination nodes) as well when the path from **source** to **destination** involves multiple **intermediate** nodes (E.g., Journey of users on website pages).

8.4.2.1 Applications of Sankey Diagram

Sankey diagrams and its variation like **Alluvial Diagrams** are commonly used for purposes like analyzing population migration, website user journey, the flow of energy, the flow of other

properties (oil, gas, etc.), research paper citations, etc. Google Analytics uses an alluvial diagram to show users' Journey on a website (sessions).

We'll use holoviews packages to build a Sankey diagrams. **Holoviews** is a high-level library that let us specify chart metadata and then creates charts using one of its back end (Bokeh, Matplotlib or Plotly). It takes pandas dataframe as a dataset and lets us create charts from it.

We'll be using the New Zealand migration dataset for our plotting purpose. It's available on kaggle for download. Dataset has information about a number of people who departed from and arrived in New Zealand from all continents and countries of the world from 1979 till 2016. We'll be aggregating this data in various ways to create different Sankey diagrams. We suggest that you download this dataset to follow along with us.

```
# Load dataset
nz_migration = pd.read_csv("./donnee/migration_nz.csv")
```

Table 8.1: New Zealand Population Migration Dataset

Measure	Country	Citizenship	Year	Value
Arrivals	Oceania	New Zealand Citizen	1979	11817
Arrivals	Oceania	Australian Citizen	1979	4436
Arrivals	Oceania	Total All Citizenships	1979	19965
Arrivals	Antarctica	New Zealand Citizen	1979	10
Arrivals	Antarctica	Australian Citizen	1979	0
Arrivals	Antarctica	Total All Citizenships	1979	13

After loading the dataset, we have performed a few steps of data cleaning and aggregation as mentioned below.

- We'll remove entries other than arrival and departure.
- We'll remove entries where the proper country name is not present.
- We'll then group the dataframe by **Measure & Country** attributes and sum up all entries.

After performing the above steps, we'll have a dataset where we'll have information about arrivals and departure counts from each country and continent of all time.

```
## Removing Entries related to "Net Total"
nz_migration = nz_migration >> query('Measure != "Net"')
## Removing entries with No Details or all countries
nz_migration = nz_migration[~nz_migration["Country"].isin(["Not stated",\
                                                           "All countries"])]
nz_migration_grouped = (nz_migration.groupby(by=["Measure","Country"])
                                    .sum()[["Value"]])
nz_migration_grouped = nz_migration_grouped.reset_index()


# import holoviews as hv
# hv.extension('bokeh')
```

8.4.2.2 Population Migration between "New Zealand" & Various "Continents"

For our first Sankey diagram, we need to filter entries of the dataframe to keep only entries where the count for each continent is present. Below we are filtering the dataset based on continent names to remove all other entries.

Table 8.2: New Zealand Population Migration Dataset (grouped)

Measure	Country	Value
Arrivals	Afghanistan	1644
Arrivals	Africa and the Middle East	149784
Arrivals	Albania	178
Arrivals	Algeria	143
Arrivals	American Samoa	2412
Arrivals	Americas	267137
Arrivals	Andorra	84
Arrivals	Angola	71
Arrivals	Anguilla	17
Arrivals	Antarctica	245

```
# Population Migration between "New Zealand" & Various "Continents"
continents = ["Asia", "Australia","Africa and the Middle East","Europe", \
              "Americas", "Oceania"]
continent_wise_migration = nz_migration_grouped[(nz_migration_grouped.Country
                                                 .isin(continents))]
```

Table 8.3: Continent wise migration

	Measure	Country	Value
1	Arrivals	Africa and the Middle East	149784
5	Arrivals	Americas	267137
14	Arrivals	Asia	795697
15	Arrivals	Australia	1057127
74	Arrivals	Europe	1044693
166	Arrivals	Oceania	1331987
252	Departures	Africa and the Middle East	63555
256	Departures	Americas	245915
265	Departures	Asia	317603
266	Departures	Australia	2325398
325	Departures	Europe	877240
417	Departures	Oceania	2534100

8.4.2.2.1 Simple Sankey Diagram We can plot a Sankey diagram very easily using holoviews by passing it above the dataframe. Holoviews needs a dataframe with at least three columns. It'll consider the first column as **source**, the second as **destination**, and the third as **property flow value** from source to destination. It'll plot links between each combination of source and destination.

```
# Run all this once
# sankey = hv.Sankey(continent_wise_migration)
# Save figure - run it also once
# hv.save(sankey, './figure/sankey_one.png')
```

We can notice from the above chart that all links have a gray color. We can color them in two ways.

1. **Source Node Color** - It'll color links with the same color as the source node from which they originated. This helps us better understand the flow from source to destination.
2. **Destination Node Color** - It'll color links with the same color as the destination node to which they are going. This helps us better understand reverse flow from destination to source.

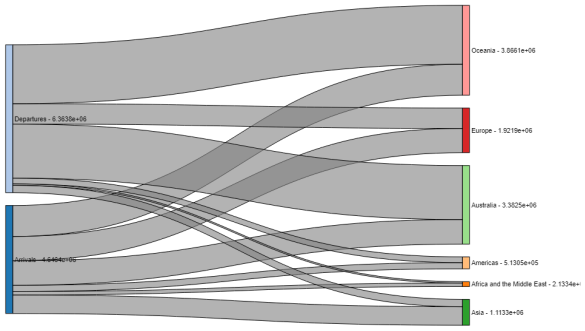


Figure 8.10: Basic Sankey diagrams

You can color links according to your need. We have explained in our tutorial how we can color links using both ways.

8.4.2.2.2 Coloring Edges as Per Destination Node Color Below we are creating the same Sankey plot again, but this time specifying which columns from the dataframe to take as **source** and **destination** in **kdims** parameter and which column to use to generate property flow arrow sizes using **vdims** parameter.

We have also specified various parameters as a part of **opts()** method called on the Sankey plot object which helped us further improve the styling of the diagram further.

We have included colormap to use for nodes & edges, label position in the diagram, column to use for edge color, edge line width, node opacity, graph width & height, graph background color, and title attributes which improves the styling of the graph a lot and makes it aesthetically pleasing.

```
## Providing column names to use as Source, Destination and Flow Value.
# sankey1 = hv.Sankey(continent_wise_migration, kdims=["Measure", "Country"],
#                     vdims=["Value"])
#
# ## Modifying Default Chart Options
# sankey1.opts(cmap='Colorblind',
#              label_position='left',
#              edge_color='Country', edge_line_width=0,
#              node_alpha=1.0, node_width=40, node_sort=True,
#              width=800, height=600, bgcolor="snow",
#              title="Population Migration between New Zealand and Other Continents")
# # Save
# hv.save(sankey1, './figure/sankey_two.png')
```

8.4.3 Alluvial diagrams

Alluvial diagrams are a subset of Sankey diagrams, and are more rigidly defined. A discussion of the differences can be found here

```
# Alluvial
import alluvial
from matplotlib import colormaps
```

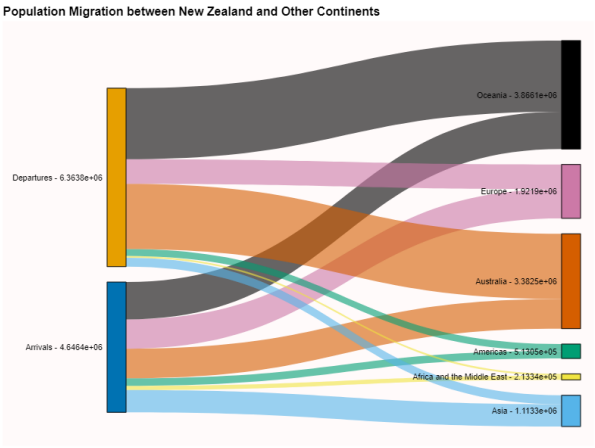


Figure 8.11: Sankey diagrams - Coloring Edges as Per Destination Node Color

```
import numpy as np

# Generating the input_data:
seed=7
np.random.seed(seed)
def rand_letter(num): return chr(ord('A')+int(num*np.random.rand()))

input_data = [[rand_letter(15), rand_letter(5)*2] for _ in range(50)]

# Plotting:
cmap = colormaps['jet']
ax = alluvial.plot(
    input_data,  alpha=0.4, color_side=1, rand_seed=seed, figsize=(7,5),
    disp_width=True, wdisp_sep=' '*2, cmap=cmap, fontname='Monospace',
    labels=('Capitals', 'Double Capitals'), label_shift=2);
ax.set_title('Utility display', fontsize=14, fontname='Monospace');
plt.show()
```

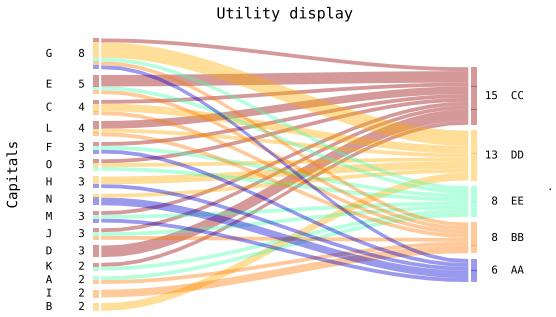


Figure 8.12: Basic Alluvial diagrams

Full alluvial code availiable https://github.com/vinsburg/alluvial_diagram.

8.5 Heatmaps

A heatmap displays a set of data using colored tiles for each variable value within each observation.

First, let's create a heatmap for the `mtcars` dataset. The `mtcars` dataset contains information on 32 cars measured on 11 variables.

```
# Basic heatmap
import seaborn as sns
mtcars2 = mtcars.set_index('name')
mtcars2 = (mtcars2 >> select_if("is_numeric"))
def scale(x):
  return ((x - x.mean())/x.std(ddof=0))
fig, axe = plt.subplots(figsize=(16,6))
sns.heatmap(mtcars2.transform(scale),ax=axe)
plt.show()
```

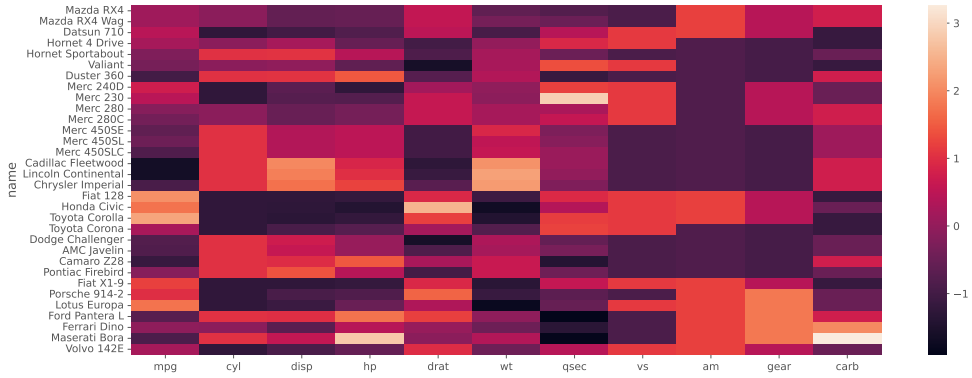


Figure 8.13: Basic heatmap

```
# Clustermap
sns.clustermap(mtcars2,figsize=(16,6),row_cluster=True,
               dendrogram_ratio=(.1, .2),cbar_pos=(0, .2, .03, .4))
```

8.6 Radar charts

A radar chart (also called a spider or star chart) displays one or more groups or observations on three or more quantitative variables.

```
# Radar charts
from matplotlib.patches import Circle, RegularPolygon
from matplotlib.path import Path
from matplotlib.projections.polar import PolarAxes
from matplotlib.projections import register_projection
from matplotlib.spines import Spine
from matplotlib.transforms import Affine2D
```

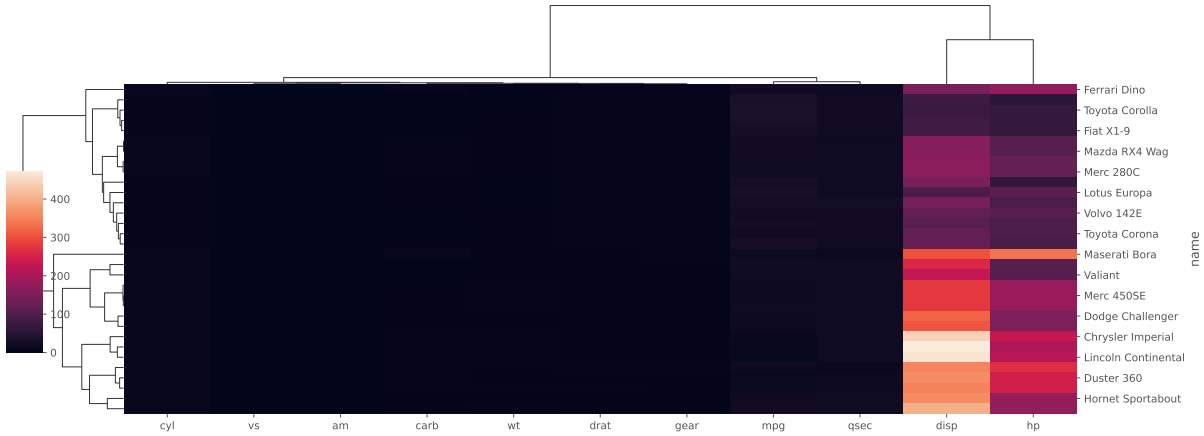


Figure 8.14: Basic clustermap

```
def radar_factory(num_vars, frame='circle'):
    """Create a radar chart with `num_vars` axes.

    This function creates a RadarAxes projection and registers it.

    Parameters
    ----------
    num_vars : int
        Number of variables for radar chart.
    frame : {'circle' | 'polygon'}
        Shape of frame surrounding axes.

    """
    # calculate evenly-spaced axis angles
    theta = np.linspace(0, 2*np.pi, num_vars, endpoint=False)

    class RadarAxes(PolarAxes):

        name = 'radar'

        def __init__(self, *args, **kwargs):
            super().__init__(*args, **kwargs)
            # rotate plot such that the first axis is at the top
            self.set_theta_zero_location('N')

        def fill(self, *args, closed=True, **kwargs):
            """Override fill so that line is closed by default"""
            return super().fill(closed=closed, *args, **kwargs)

        def plot(self, *args, **kwargs):
            """Override plot so that line is closed by default"""
            lines = super().plot(*args, **kwargs)
            for line in lines:
                self._close_line(line)
```

```
        def _close_line(self, line):
            x, y = line.get_data()
            # FIXME: markers at x[0], y[0] get doubled-up
            if x[0] != x[-1]:
                x = np.concatenate((x, [x[0]]))
                y = np.concatenate((y, [y[0]]))
                line.set_data(x, y)

        def set_varlabels(self, labels):
            self.set_thetagrids(np.degrees(theta), labels)

        def _gen_axes_patch(self):
            # The Axes patch must be centered at (0.5, 0.5) and of radius 0.5
            # in axes coordinates.
            if frame == 'circle':
                return Circle((0.5, 0.5), 0.5)
            elif frame == 'polygon':
                return RegularPolygon((0.5, 0.5), num_vars,
                                      radius=.5, edgecolor="k")
            else:
                raise ValueError("unknown value for 'frame': %s" % frame)

        def draw(self, renderer):
            """ Draw. If frame is polygon, make gridlines polygon-shaped """
            if frame == 'polygon':
                gridlines = self.yaxis.get_gridlines()
                for gl in gridlines:
                    gl.get_path()._interpolation_steps = num_vars
            super().draw(renderer)

        def _gen_axes_spines(self):
            if frame == 'circle':
                return super()._gen_axes_spines()
            elif frame == 'polygon':
                # spine_type must be 'left'/'right'/'top'/'bottom'/'circle'.
                spine = Spine(axes=self,
                              spine_type='circle',
                              path=Path.unit_regular_polygon(num_vars))
                # unit_regular_polygon gives a polygon of radius 1 centered at
                # (0, 0) but we want a polygon of radius 0.5 centered at (0.5,
                # 0.5) in axes coordinates.
                spine.set_transform(Affine2D().scale(.5).translate(.5, .5)
                                    + self.transAxes)

                return {'polar': spine}
            else:
                raise ValueError("unknown value for 'frame': %s" % frame)

    register_projection(RadarAxes)
    return theta
```

```
data = [['Sulfate', 'Nitrate', 'EC', 'OC1', 'OC2', 'OC3', 'OP', 'CO', 'O3'],
        ('Basecase', [
            [0.88, 0.01, 0.03, 0.03, 0.00, 0.06, 0.01, 0.00, 0.00],
            [0.07, 0.95, 0.04, 0.05, 0.00, 0.02, 0.01, 0.00, 0.00],
            [0.01, 0.02, 0.85, 0.19, 0.05, 0.10, 0.00, 0.00, 0.00],
            [0.02, 0.01, 0.07, 0.01, 0.21, 0.12, 0.98, 0.00, 0.00],
            [0.01, 0.01, 0.02, 0.71, 0.74, 0.70, 0.00, 0.00, 0.00]])]

N = len(data[0])
theta = radar_factory(N, frame='polygon')

spoke_labels = data.pop(0)
title, case_data = data[0]

fig, ax = plt.subplots(figsize=(4,3), subplot_kw=dict(projection='radar'));
fig.subplots_adjust(top=0.85, bottom=0.05)
ax.set_rgrids([0.2, 0.4, 0.6, 0.8]);
ax.set_title(title,  position=(0.5, 1.1), ha='center');
for d in case_data:
    line = ax.plot(theta, d);
    ax.fill(theta, d,  alpha=0.25);
ax.set_varlabels(spoke_labels);
plt.show()
```

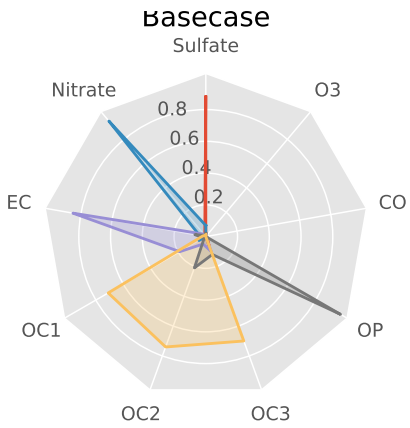


Figure 8.15: Basic radar chart

```
# Using plotly
from plotnine.data import msleep

# Min - Max scaler
def minmax(x): return (x - np.min(x))/(np.max(x)- np.min(x))

plotdata = (msleep >> query('name in ["Cow","Dog","Pig"]')>>
                      select("name", "sleep_total", "sleep_rem",\
                              "sleep_cycle", "brainwt", "bodywt") >>
                      rename(group = "name")>>
                      mutate_at("-group",minmax))
```

Table 8.4: Plot data

	group	sleep_total	sleep_rem	sleep_cycle	brainwt	bodywt
4	Cow	0.0000000	0.0000000	1.0	1.0000000	1.0000000
8	Dog	1.0000000	1.0000000	0.0	0.0000000	0.0000000
73	Pig	0.8360656	0.7727273	0.5	0.3116147	0.1232935

```
# Radar chart using plotly
import plotly.graph_objects as go

categories= ["sleep_total","sleep_rem","sleep_cycle", "brainwt", "bodywt"]

fig = go.Figure()
for g in ["Cow","Dog","Pig"]:
  df = plotdata[plotdata["group"]== g].drop(columns="group").values[0]
  fig.add_trace(go.Scatterpolar(r=df,theta=categories,fill='toself',name=g))
fig.update_layout(
  polar=dict(radialaxis=dict(visible=True,range=[0, 1])),
  showlegend=True)
# # Save figure - run once
fig.write_image("./figure/radar.png")
```

8.7 Scatterplot matrix

A scatterplot matrix is a collection of scatterplots organized as a grid. It is similar to a correlation plot but instead of displaying correlations, displays the underlying data.

You can create a scatterplot matrix using the `pairplot` function in the `seaborn` package.

```
# Create scatterplot matrix
from math import log
df = (msleep >>
      mutate(log_brainwt="np.log(brainwt)",log_bodywt="np.log(bodywt)") >>
      select("log_brainwt", "log_bodywt", "sleep_total", "sleep_rem"))
sns.pairplot(df,height=2, aspect=1)
```

8.8 Waterfall charts

A waterfall chart illustrates the cumulative effect of a sequence of positive and negative values.

```
# Create waterfall chart
def waterfall(df, x, y):
    # calculate running totals
    df['tot'] = df[y].cumsum()
    df['tot1']=df['tot'].shift(1).fillna(0)
    # lower and upper points for the bar charts
    lower = df[['tot','tot1']].min(axis=1)
    upper = df[['tot','tot1']].max(axis=1)
    # mid-point for label position
```

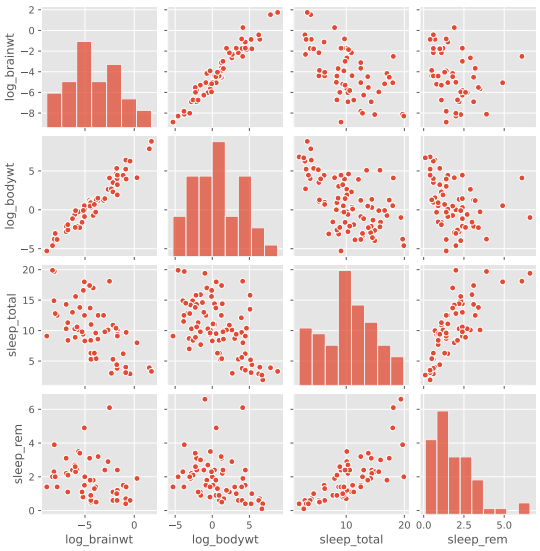


Figure 8.16: Scatterplot matrix

```
    mid = (lower + upper)/2
    # positive number shows green, negative number shows red
    df.loc[df[y] >= 0, 'color'] = 'green'
    df.loc[df[y] < 0, 'color'] = 'red'
    # calculate connection points
    connect= df['tot1'].repeat(3).shift(-1)
    connect[1::3] = np.nan
    fig,ax = plt.subplots()
    # plot first bar with colors
    bars = ax.bar(x=df[x],height=upper, color =df['color'])
    # plot second bar - invisible
    plt.bar(x=df[x], height=lower,color='white');
    # plot connectors
    plt.plot(connect.index,connect.values, 'k' );
    # plot bar labels
    for i, v in enumerate(upper):
        plt.text(i-.15, mid[i], f"{df[y][i]:,.0f}");
    return plt.show()

# Create dataset
income = pd.DataFrame({
  'category':["Sales", "Services", "Fixed Costs", "Variable Costs", "Taxes"],
       'num':[101000, 52000, -23000, -15000, -10000]})
```

Table 8.5: Company income statement

category	num
Sales	101000
Services	52000
Fixed Costs	-23000
Variable Costs	-15000
Taxes	-10000

Now we can visualize this with a waterfall chart, using the `waterfall` function.

```
# Waterfall chart
waterfall(income,'category','num')
```

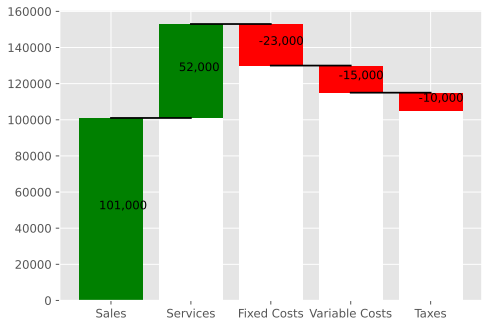


Figure 8.17: Basic waterfall chart

8.9 Word clouds

A word cloud (also called a tag cloud), is basically an infographic that indicates the frequency of words in a collection of text (e.g., tweets, a text document, a set of text documents). Here is the text :

> "Data Science : definition, problématiques et cas d'usage. La Data Science ou science des données est un vaste champ multi-disciplinaire visant à donner du sens aux données brutes. Data Science : définition, champs d'applications et limites actuelles, découvrez tout ce que vous devez savoir sur ce domaine complexe, devenu un enjeu prioritaire dans les entreprises de toutes les industries. Pour définir la Data Science de la plus simple des façons, il s'agit de l'extraction d'informations exploitables à partir de données brutes. Ce champ multi-disciplinaire a pour but principal d'identifier des tendances, des motifs, des connexions et des corrélations dans les larges ensembles de données. La science des données englobe une large variété d'outils et de techniques telles que la programmation informatique, l'analyse prédictive, les mathématiques, les statistiques ou l'intelligence artificielle. Désormais, la Data Science inclut aussi les algorithmes de Machine Learning. De nos jours, presque toutes les entreprises affirment pratiquer la Data Science sous une forme ou une autre. Cependant, les méthodes et approches employées peuvent varier d'une organisation à l'autre. Il devient donc très compliqué d'offrir une définition précise de la Data Science. D'autant que de nouvelles technologies apparaissent sans cesse et transforment continuellement ce domaine. Ainsi, pour définir la science des données, la meilleure question à se poser est: pourquoi ?. Pourquoi la science des données ? Si la Data Science connaît un essor fulgurant dans tous les secteurs d'activité, c'est parce que l'humanité génère de plus en plus de données. Entre 2011 et 2013, en seulement deux ans, le volume mondial de données a été multiplié par 9. Et cette explosion du Big Data n'a pas ralenti depuis. D'ici la fin de l'année 2020, le volume total de données à l'échelle de la planète devrait atteindre 44 zettabytes contre moins de 5 zettabytes en 2013. Comment expliquer ce phénomène ? Plusieurs technologies émergentes génèrent des données. C'est le cas des objets connectés, des réseaux sociaux, des smartphones, ou des moteurs de recherche web. Or, toutes ces données offrent des opportunités inouïes pour les entreprises de toutes les industries, les institutions de recherche ou

le secteur public. C'est la raison pour laquelle les données sont souvent considérées comme le pétrole du XXIème siècle. En s'appuyant sur ces découvertes, il est possible de créer de nouveaux produits et services innovants, de résoudre des problèmes concrets, d'améliorer ses performances comme jamais auparavant. La Data Science permet de prendre des décisions basées sur les données, plutôt que sur une simple intuition. Ainsi, elle révolutionne notre quotidien et nous permet de s'ouvrir à de nouveaux horizons. En bref, la data science représentera une science incontournable du monde demain ! Comment fonctionne la data science ? La Data Science couvre une large variété de disciplines et de champs d'expertise. Son but reste toutefois de donner du sens aux données brutes. Pour y parvenir, les Data Scientists doivent posséder des compétences en ingénierie des données, en mathématiques, en statistique, en informatique et en Data Visualization. Ces compétences leur permettront de parcourir les vastes ensembles de données brutes pour en dégager les informations les plus pertinentes et les communiquer aux décideurs de leurs organisations. Les Data Scientists exploitent également l'intelligence artificielle, et plus particulièrement le Machine Learning et le Deep Learning. Ces technologies sont utilisées pour créer des modèles et réaliser des prédictions en utilisant des algorithmes et diverses techniques. Dans un premier temps, les données doivent être collectées, extraites à partir de différentes sources. Il s'agit ensuite de les entreposer dans une Data Warehouse, de les nettoyer, de les transformer afin qu'elles puissent être analysées. L'étape suivante est celle du traitement des données, par le biais du Data Mining (forage de données), du clustering, de la classification ou de la modélisation. Les données sont ensuite analysées à l'aide de techniques comme l'analyse prédictive, la régression ou le text mining. Enfin, la dernière étape consiste à communiquer les informations dégagées par le biais du reporting, du dashboarding ou de la Data Visualization. Les cas d'usage et applications Les cas d'usage de la Data Science sont aussi nombreux que variés. Cette technologie est utilisée pour assister la prise de décision en entreprise, mais permet aussi l'automatisation de certaines tâches. Elle est utilisée à des fins de détection d'anomalies ou de fraude. La science des données permet aussi la classification, par exemple pour trier automatiquement les emails dans votre boîte. Elle permet aussi la prédiction, par exemple pour les ventes ou les revenus. En l'utilisant, il est possible de détecter des tendances ou des patterns dans les ensembles de données. La Data Science se cache aussi derrière les technologies de reconnaissance faciale, vocale ou textuelle. Elle alimente aussi les moteurs de recommandations capables de vous suggérer des produits ou du contenu en fonction de vos préférences. D'un secteur d'activité à l'autre, la Data Science est exploitée de différentes manières. Dans le domaine de la santé, les données permettent aujourd'hui de mieux comprendre les maladies, de recourir à la médecine préventive, d'inventer de nouveaux traitements ou d'accélérer les diagnostics. En logistique, la Data Science aide à optimiser les itinéraires et les opérations internes en temps réel en tenant compte de facteurs comme la météo ou le trafic. Dans la finance, elle permet d'automatiser le traitement des données d'accords de crédit grâce au Traitement Naturel du Langage (Vous n'êtes pas familier avec ce concept, découvrez le NLP dans notre article dédié) ou de détecter la fraude grâce au Machine Learning. Les entreprises de retail l'utilisent pour le ciblage publicitaire et le marketing personnalisé. Les moteurs de recommandations, basés sur l'analyse des préférences du consommateur, sont utilisés par Google pour son moteur de recherche web, par les plateformes de streaming comme Netflix ou Spotify, et par les entreprises de e-commerce comme Amazon. Les entreprises de cybersécurité se tournent vers l'IA et la science des données pour découvrir de nouveaux malwares au quotidien. Même les voitures autonomes reposent sur la Data Science et l'analyse prédictive pour ajuster

leur vitesse, éviter les obstacles et les changements de voie dangereux ou choisir l'itinéraire le plus rapide."

and word to exclure

'd', 'du', 'de', 'la', 'des', 'le', 'et', 'est', 'elle', 'une', 'en', 'que', 'aux', 'qui', 'ces', 'les', 'dans', 'sur', 'l', 'un', 'pour', 'par', 'il', 'ou', 'à', 'ce', 'a', 'sont', 'cas', 'plus', 'leur', 'se', 's', 'vous', 'au', 'c', 'aussi', 'toutes', 'autre', 'comme

```
from wordcloud import WordCloud
from PIL import Image

wordcloud = (
  WordCloud(background_color = 'white', stopwords = exclure_mots, max_words = 50)
  .generate(text)
)
plt.imshow(wordcloud);
plt.axis("off");
plt.show()
```

Figure 8.18: Word cloud

9 Customizing graphs

Graph defaults are fine for quick data exploration, but when you want to publish your results to a blog, paper, article or poster, you'll probably want to customize the results. Customization can improve the clarity and attractiveness of a graph.

This chapter describes how to customize a graph's axes, gridlines, colors, fonts, labels, and legend. It also describes how to add annotations (text and lines).

9.1 Axes

The x-axis and y-axis represent numeric, categorical, or date values. You can modify the default scales and labels with the functions below.

9.1.1 Quantitative axes

A quantitative axis is modified using the `scale_x_continuous` and `scale_y_continuous` function.

Options include :

- `breaks` - a numeric vector of positions
- `limits` - a numeric vector with the min and max for the scale.

```
# customize numerical x and y axes
from plotnine import *
from plotnine.data import mpg

print((
  mpg >> ggplot(aes(x="displ", y="hwy")) +
  geom_point() +
  scale_x_continuous(breaks = range(1, 8, 1),limits=[1, 7]) +
  scale_y_continuous(breaks = range(10, 46, 5),limits=[10, 45])
))
```

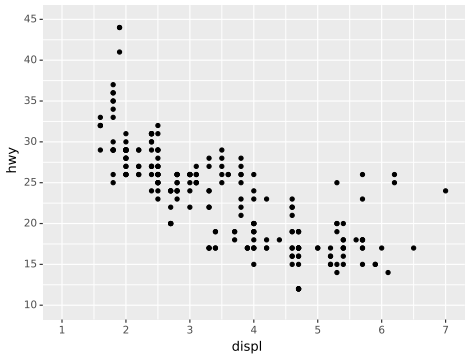


Figure 9.1: Customized quantitative axes

9.1.1.1 Numeric formats

The `mizani` package provides a number of functions for formatting numeric labels. Some of the most useful are :

- `currency_format`
- `comma_format`
- `percent_format`

Let's demonstrate these functions with some synthetic data.

```
# Create some date
import numpy as np
import pandas as pd
from plydata import *

np.random.seed(1234)
df = pd.DataFrame({
  "xaxis" : np.random.normal(loc=100000, scale=50000, size=50),
  "yaxis" : np.random.uniform(low=0.0, high=1.0, size=50),
  "pointsize": np.random.normal(loc = 1000, scale=1000, size=50)
})
```

Table 9.1: Dataset simulate - head

xaxis	yaxis	pointsize
123571.76	0.7905241	1079.84231
40451.22	0.9920815	600.03542
171635.35	0.9588018	-27.85056
84367.41	0.7919641	415.28179
63970.56	0.2852510	1816.59393
144358.15	0.6249167	918.05295

```
from mizani.formatters import percent_format, comma_format, currency_format

# Plot the axes and legend with formats
print(( df >> ggplot(aes(x = "xaxis", y = "yaxis", size = "pointsize"))+
```

```
    geom_point(color = "steelblue", alpha = 0.6)+
    scale_x_continuous(labels = comma_format())+
    scale_y_continuous(labels = percent_format())+
    scale_size_continuous(range = [1,10], labels = currency_format())+
    theme(legend_position=(0.9,0.3),legend_direction='vertical',
          legend_box_spacing=0.25)))
```

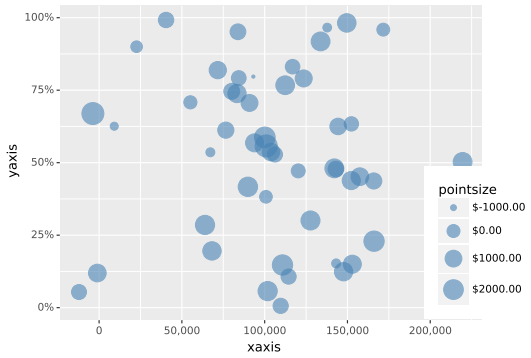


Figure 9.2: Formatted axes

9.1.2 Categorical axes

A categorical axis is modified using the `scale_x_discrete` or `scale_y_discrete` function. Options include :

- `limits` - a character vector (the levels of the quantitative variable in the desired order)
- `labels` - a character vector of labels (optional labels for these levels)

```
# Customize categorical x axis
print(( mpg >> ggplot(aes(x = "class"))+geom_bar(fill = "steelblue")+
    scale_x_discrete(limits = ["pickup", "suv", "minivan",
                               "midsize", "compact", "subcompact",
                               "2seater"],
                     labels = ["Pickup\nTruck", "Sport Utility\nVehicle",
                               "Minivan", "Mid-size", "Compact",
                               "Subcompact", "2-Seater"])))
```

9.1.3 Date axes

A date axis is modified using the `scale_x_date` or `scale_y_date` function. Options include :

- `date_breaks` - a string giving the distance between breaks likes “2 weeks” or “10 years”
- `date_labels` - A string giving the formatting specification for the labels.

The table below gives the formatting specifications for date values :

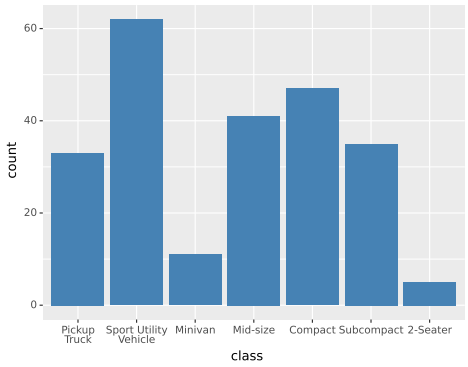


Figure 9.3: Customized categorical axis

Table 9.2: Formatting specifications for date values

Symbol	Meaning	Example
%d	day as a number (0-31)	01-31
%a	abbreviated weekday	Mon
%A	unabbreviated weekday	Monday
%m	month(00 - 12)	00-12
%b	abbreviated month	Jan
%B	unabbreviated month	January
%y	2-digit year	07
%Y	4-digit year	2007

```
# customize date scale on x axis
from mizani.breaks import date_breaks
from mizani.formatters import date_format
from plotnine.data import economics

print((economics >> ggplot(aes(x = "date", y = "unemploy"))+
    geom_line(color = "darkgreen")+
    scale_x_datetime(breaks = date_breaks("5 years"),
                     labels= date_format("%b-%y"))))
```

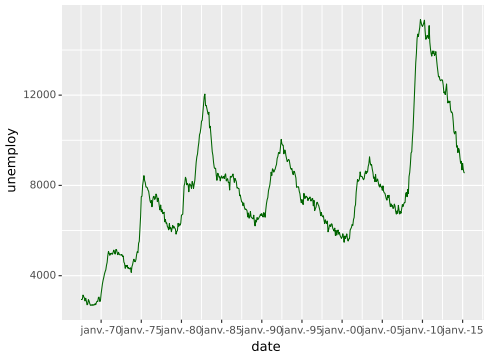


Figure 9.4: Customized date axis

9.2 Colors

The default colors in `plotnine` graphs are functional, but often not as visually appealing as they can be. Happily this is easy to change.

Specific colors can be :

- specified for points, lines, bars, areas, and text, or
- mapped to the levels of a variable in the dataset.

9.2.1 Specifying colors manually

To specify a color for points, lines, or text, use the `color = "colorname"` option in the appropriate geom. To specify a color for bars and areas, use the `fill = "colorname"` option.

Examples :

- `geom_point(color = "blue")`
- `geom_bar(fill = "steelblue")`

Colors can be specified by `name` or `hex code`.

To assign colors to the levels of a variable, use the `scale_color_manual` and `scale_fill_manual` functions. The former is used to specify the colors for points and lines, while the later is used for bars and areas.

Here is an example, using the `diamonds` dataset that ships with `plotnine`. The dataset contains the prices and attributes of 54,000 round cut diamonds.

```
# specify fill color manually
from plotnine.data import diamonds

print((diamonds >>ggplot(aes(x = "cut", fill = "clarity"))+
    geom_bar()+
    scale_fill_manual(values = ["darkred", "steelblue",
                                "darkgreen", "gold",
                                "brown", "purple",
                                "grey", "khaki"])+
    theme(legend_position=(0.2,0.6),legend_direction='vertical')))
```

9.2.2 Color palettes

There are many predefined color palettes availlable in python.

9.2.2.1 ColorBrewer

The most popular alternative palettes are probably the ColorBrewer palettes.

You can specify these palettes with the `scale_color_brewer` and `scale_fill_brewer` functions.

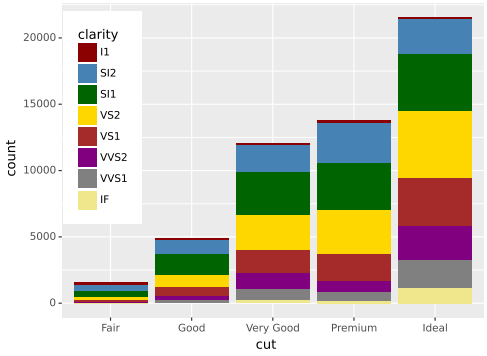
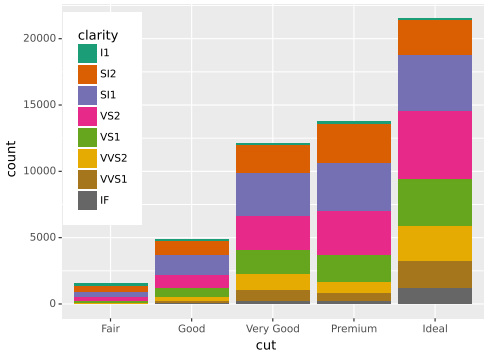


Figure 9.5: Manual color selection

```
# use an ColorBrewer fill palette
print((diamonds >> ggplot(aes(x = "cut", fill = "clarity"))+
    geom_bar()+scale_fill_brewer(type="qual", palette="Dark2")+
    theme(legend_position=(0.2,0.6),legend_direction='vertical')))
```



9.2.2.2 Viridis

The viridis palette is another popular choice.

```
# Use a viridis fill palette
import matplotlib as mpl
import matplotlib.pyplot as plt

def discrete_cmap_hex(N, base_cmap=None):
  base = plt.cm.get_cmap(base_cmap)
  color_list = list(base(np.linspace(0, 1, N)))
  color_list_list = [list(x) for x in color_list]
  hex_list = [mpl.colors.to_hex(x, keep_alpha=True) for x in color_list_list]
  return hex_list
print((diamonds >> ggplot(aes(x="cut",fill="clarity"))+
       geom_bar()+scale_fill_manual(discrete_cmap_hex(8,'viridis'))))
```

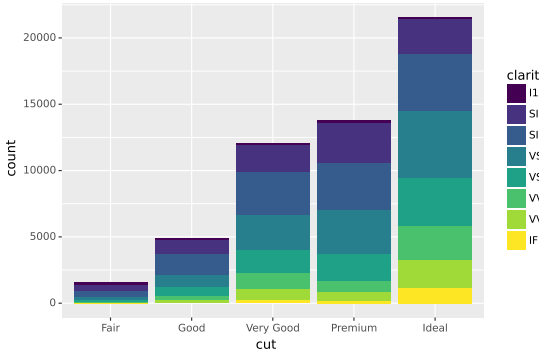


Figure 9.6: Using the viridis palette

9.3 Points & Lines

9.3.1 Points

For `plotnine` graphs, the default point is a filled circle. To specify a different shape, use the `shape = #` option in the `geom_point` function. To map shapes to the levels of a categorical variable use the `shape = variablename` option in the `aes` function.

Examples :

- `geom_point(shape = 1)`
- `geom_point(aes(shape=sex))`

Shapes from 21 through 26 provide for both a fill color and a border color.

9.3.2 Lines

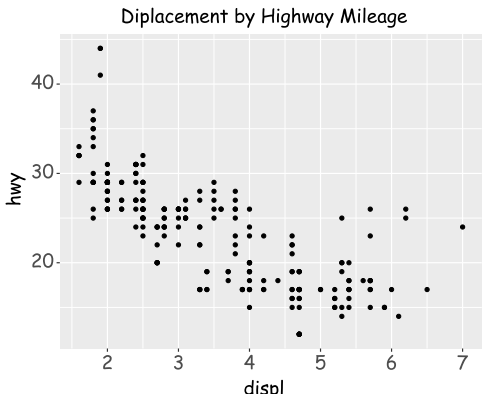
The default line type is a solid line. To change the linetype, use the `linetype = #` option in the `geom_line` function. To map linetypes to the levels of a categorical variable use the `linetype = variablename` option in the `aes` function.

Examples :

- `geom_line(linetype=1)`
- `geom_line(aes(linetype=sex))`

9.4 Fonts

```
# Specify new font
print((mpg >> ggplot(aes(x = "displ", y = "hwy"))+
    geom_point()+labs(title = "Diplacement by Highway Mileage")+
    theme(text = element_text(size=16, family = "Comic Sans MS"))
))
```



9.5 Legends

In `plotnine`, legends are automatically created when variables are mapped to color, fill, linetype, shape, size, or alpha.

You have a great deal of control over the look and feel of these legends. Modifications are usually made through the `theme` function and/or the `labs` function. Here are some of the most sought after.

9.5.1 Legend location

The legend can appear anywhere in the graph. By default, it's placed on the right. You can change the default with

```
theme(legend_position=position)
```

where

Table 9.3: Legends position

Position	Location
"top"	above the plot area
"right"	right of the plot area
"bottom"	below the plot area
"left"	left of the plot area
(x, y)	The x and y values must range between 0 and 1
"None"	suppress the legend

For example, to place the legend at the top, use the following code.

```
# Place legend on top
print((mpg >> ggplot(aes(x = "displ", y = "hwy", color = "class"))+
    geom_point(size=4)+labs(title = "Diplacement by Highway Mileage")+
    theme_minimal()+
    theme(legend_position=(0.6,0.8))))
```

9.5.2 Legend title

You can change the legend title through the `labs` function. Use `color`, `fill`, `size`, `shape`, `linetype`, and `alpha` to give new titles to the corresponding legends.

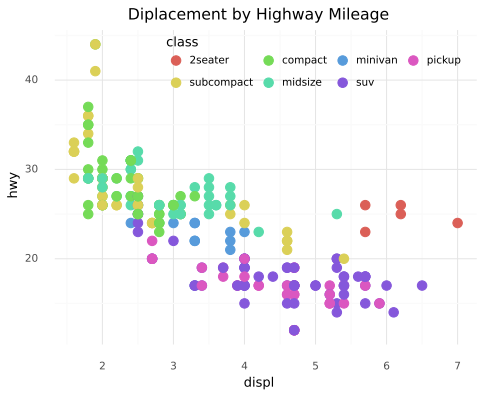


Figure 9.7: Moving the legend

The alignment of the legend title is controlled through the `legend_title_align` option in the `theme` function.

```
# Change the default legend title
print((mpg >>ggplot(aes(x = "displ", y = "hwy", color = "class"))+
  geom_point(size=4)+
  labs(title = "Diplacement by Highway Mileage",
       color = "Automobile\nClass")+
  theme_minimal()+
  theme(legend_title_align = 'center')+theme(legend_position=(0.6,0.8))))
```

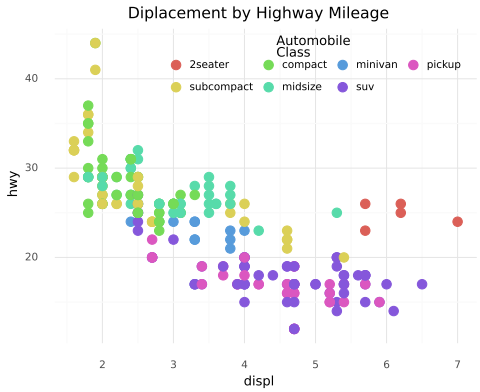


Figure 9.8: Changing the legend title

9.6 Labels

Labels are a key ingredient in rendering a graph understanble.They're are added with the `labs` function. Available options are given below :

For example

```
# add plot labels
print((
  mpg >> ggplot(aes(x = "displ", y="hwy", color = "class",shape="factor(year)")) +
  geom_point(size = 3,alpha = 0.5) +
```

Table 9.4: Labs option

option	Use
title	main title
caption	caption (bottom right by default)
x	horizontal axis
y	vertical axis
color	color legend title
fill	fill legend title
size	size legend title
linetype	linetype legend title
shape	shape legend title
alpha	transparency legend title
size	size legend title

```
  labs(title = "Mileage by engine displacement",
       caption = "Source: EPA (http://fueleconomy.gov)",
       x = "Engine displacement (litres)",
       y = "Highway miles per gallon",
       color = "Car Class",shape = "Year") +
  theme_minimal()+theme(legend_position=(0.6,0.8))
))
```

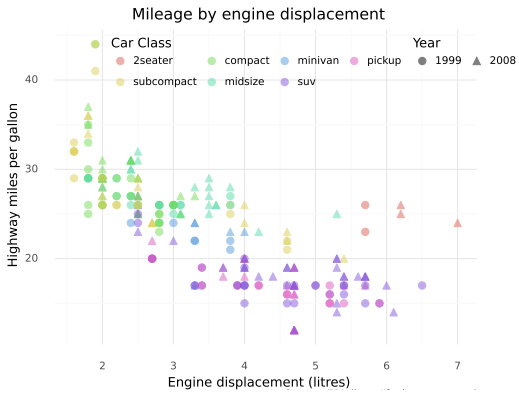


Figure 9.9: Graph with labels

9.7 Annotations

Annotations are addition information added to a graph to highlight important points.

9.7.1 Adding text

There are two primary reasons to add text to a graph.

One is to identify the numeric qualities of a geom. For example, we may want to identify points with labels in a scatterplot, or label the heights of bars in a bar chart.

Another reason is to provide additional information. We may want to add notes about the data, point out outliers, etc.

9.7.1.1 Labeling values

Consider the following scatterplot, based on the car data in the `mtcars` dataset.

```
# Basic scatterplot
from plotnine.data import mtcars
print((mtcars >>ggplot(aes(x = "wt", y = "mpg"))+geom_point()))
```

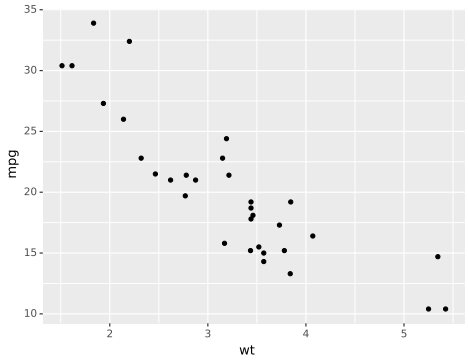


Figure 9.10: Simple scatterplot

Let's label each point with the name of the car it represents.

```
# Scatterplot with labels
print((mtcars >>ggplot(aes(x = "wt", y = "mpg"))+
    geom_point()+geom_text(label = mtcars.name)))
```

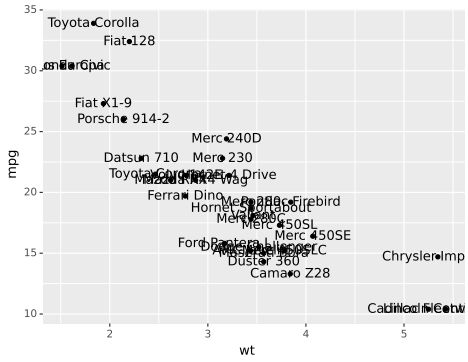


Figure 9.11: Scatterplot with labels

Change label size

```
# scatterplot with non-overlapping labels
print((mtcars >> ggplot(aes(x = "wt", y = "mpg"))+geom_point()+
    geom_text(label = mtcars.name, size = 8,
    adjust_text={'arrowprops': {'arrowstyle': '->','color': 'black','lw':1.0}})
))
```

9.7.1.2 Adding additional information

We can place text anywhere on a graph using the `annotate` function. The format is

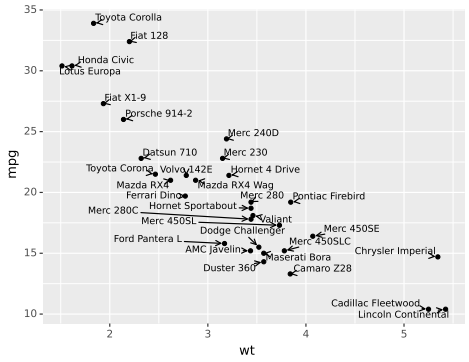


Figure 9.12: Scatterplot with non-overlapping labels

```
annotate("text",x, y, label = "Some text", color = "colorname",size=textsize)
```

where x and y are the coordinates on which to place the text. The `color` and `size` parameters are optional.

```
txt = """
The relationship between car weight \n
and mileage appears to be roughly linear
"""
print((mtcars >>ggplot(aes(x = "wt", y = "mpg"))+geom_point(color = "red")+
    geom_text(label = mtcars.name, size = 8,
    adjust_text={'arrowprops': {'arrowstyle': '->','color': 'black','lw':1.0}})+
    annotate("text", x = 4, y = 30, label = txt,color = "red",size=8)+
    theme_bw()))
```

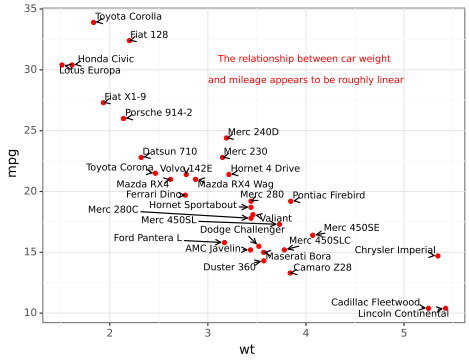


Figure 9.13: Scatterplot with arranged labels

9.7.2 Adding lines

Horizontal and vertical lines can be added using :

- `geom_hline(yintercept=a)`
- `geom_vline(xintercept=b)`

Where `a` is a number on the y-axis and `b` is a number on the x-axis respectively. Other option include `linetype` and `color`.

```
# Add annotation line and text label
min_cty = np.min(mpg.cty)
mean_hwy = np.mean(mpg.hwy)

print((mpg >>ggplot(aes(x = "cty", y = "hwy", color = "drv"))+
  geom_point(size = 3) +
  geom_hline(yintercept = mean_hwy,color = "darkred",linetype = "dashed") +
  annotate("text", x = min_cty,y = mean_hwy + 1,label = "Mean",color = "darkred") +
  labs(title = "Mileage by drive type",x = "City miles per gallon",
      y = "Highway miles per gallon", color = "Drive")))
```

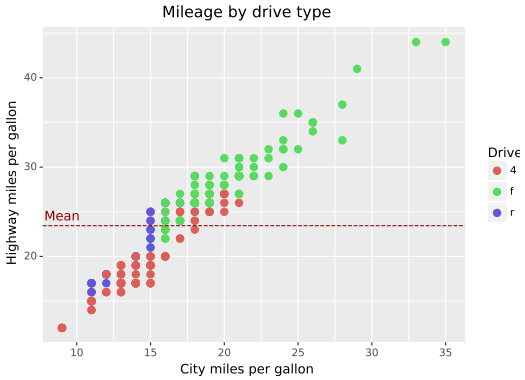


Figure 9.14: Graph with line annotation

We could add a vertical line for the mean city miles per gallon as well. In any case, always label annotation lines in some way. Otherwise the reader will not know what they mean.

9.8 Themes

`plotnine` themes control the appearance of all non-data related components of a plot. You can change the look and feel of a graph by altering the elements of its theme.

9.8.1 Altering theme elements

The `theme` function is used to modify individual components of a theme.

```
data(Salaries,package="carData")

# Create graph
p = (ggplot(r.Salaries, aes(x = "rank", fill = "sex")) +geom_bar() +
  facet_wrap("~discipline") +
  labs(title = "Academic Rank by Gender and Discipline",
       x = "Rank",
       y = "Frequency",
       fill = "Gender"))
print(p)
```

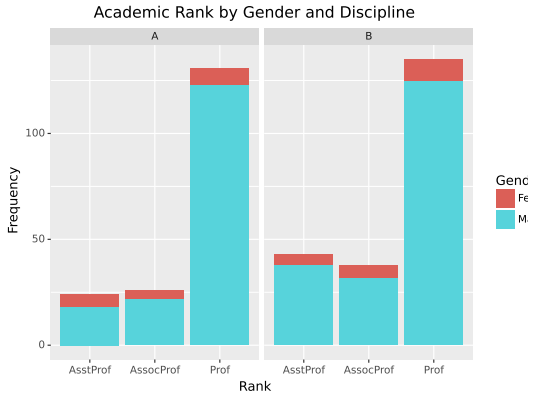


Figure 9.15: Graph with default theme

Let's make some changes to the theme.

- Change label text from black to navy blue
- Change the panel background color from grey to white
- Add solid grey lines for major y-axis grid lines
- Add dashed grey lines for minor y-axis grid lines
- Eliminate x-axis grid lines
- Change the strip background color to white with a grey border

Using the cheat sheet gives us

```
p = (p +
  theme(text = element_text(color = "navy"),
        panel_background = element_rect(fill = "white"),
        panel_grid_major_y = element_line(color = "grey"),
        panel_grid_minor_y = element_line(color = "grey",
                                          linetype = "dashed"),
        panel_grid_major_x = element_blank(),
        panel_grid_minor_x = element_blank(),
        strip_background = element_rect(fill = "white", color="grey")))
print(p)
```

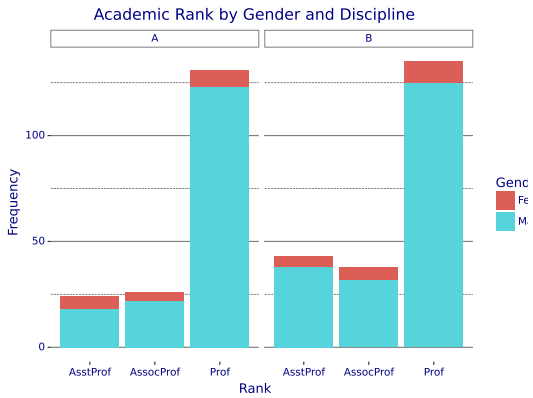


Figure 9.16: Graph with modified theme

Conclusion

There are multiple packages in Python for Data visualization. Here, we have a list of 12 interdisciplinary Python data visualization libraries, from the weel - known to the obscure.

- Matplotlib
- Seaborn
- Plotnine
- Bokeh
- pygal
- Plotly
- geoplotlib
- Gleam
- missingno
- Leather
- Altair
- Folium

For more example with plotnine, visit https://dputhier.github.io/jgb53d-bd-prog_github/practicals/intro_ggplot/intro_ggplot.html.

References

Gómez-Rubio, Virgilio. 2017. "Ggplot2-Elegant Graphics for Data Analysis." *Journal of Statistical Software* 77: 1–3.

Harris, Charles R., K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, et al. 2020. "Array Programming with NumPy." *Nature* 585: 357–62. https://doi.org/10.1038/s41586-020-2649-2.

Harris, Charles R., K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, et al. 2020. "Array Programming with NumPy." *Nature* 585 (7825): 357–62. https://doi.org/10.1038/s41586-020-2649-2.

McKinney, Wes et al. 2010. "Data Structures for Statistical Computing in Python." In *Proceedings of the 9th Python in Science Conference*, 445:51–56. Austin, TX.

Seabold, Skipper, and Josef Perktold. 2010. "Statsmodels: Econometric and Statistical Modeling with Python." In *9th Python in Science Conference*.

team, The pandas development. 2020. *Pandas-Dev/Pandas: Pandas* (version latest). Zenodo. https://doi.org/10.5281/zenodo.3509134.

Wickham, Hadley. 2006. "An Introduction to Ggplot: An Implementation of the Grammar of Graphics in r." *Statistics*, 1–8.

Wilkinson, Leland. 2012. *The Grammar of Graphics*. Springer.

Introduction to Data Visualization in Python

Data visualization is a field in data analysis that deals with visual representation of data. It graphically plots data and is an effective way to communicate inferences from data.

Using data visualization, we can get a visual summary of our data. With pictures, maps and graphs, the human mind has an easier time processing and understanding any given data. Data visualization plays a significant role in the representation of both small and large data sets, but it is especially useful when we have large data sets, in which it is impossible to see all of our data, let alone process and understand it manually.

plotnine is an implementation of a grammar of graphics in Python based on ggplot2. The grammar allows you to compose plots by explicitly mapping variables in a dataframe to the visual objects that make up the plot.

Plotting with a grammar of graphics is powerful. Custom (and otherwise complex) plots are easy to think about and build incrementally, while the simple plots remain simple to create.