W6 Joining data

- **Join/Merge**
- **Concat**

One-to-one

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

C	D
C1	D1
C2	D2
C3	D3

One-To-One = Every row in the left table is related to only one row in the right table

One-to-many

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

C	D
C1	D1
C1	D2
C1	D3
C2	D4

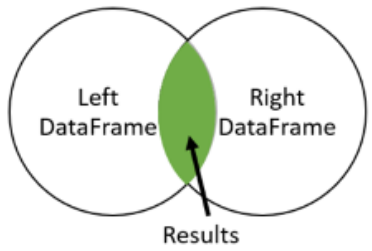
One-To-Many = Every row in left table is related to one or more rows in the right table

When you merge tables that have a one-to-many relationship, the number of rows returned will likely be different than the number in the left table.

Inner join

```
wards_census = wards.merge(census, on='ward')
print(wards_census.head(4))
```

Inner join



Suffixes

```
wards_census = wards.merge(census, on='ward', suffixes=('_ward','_cen'))
print(wards_census.head())
print(wards_census.shape)
```

Theoretical merge

```
grants_licenses = grants.merge(licenses, on='zip')
print(grants_licenses.loc[grants_licenses['business']=="REGGIE'S BAR & GRILL",
                          ['grant','company','account','ward','business']])
```

	grant	company	account	ward	business
0	136443.07	CEDARS MEDIT...	307071	3	REGGIE'S BAR...
1	39943.15	DARRYL & FYL...	307071	3	REGGIE'S BAR...
2	31250.0	JGF MANAGEMENT	307071	3	REGGIE'S BAR...
3	143427.79	HYDE PARK AN...	307071	3	REGGIE'S BAR...
4	69500.0	ZBERRY INC	307071	3	REGGIE'S BAR...

```
grants.merge(licenses, on=['address','zip'])
```

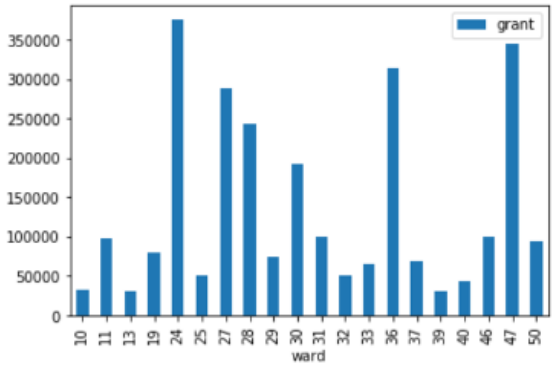
	address	zip	grant	company	account	ward	aid	business
0	1020 N KOLMA...	60651	68309.8	TRITON INDUS...	7689	37	929	TRITON INDUS...
1	10241 S COMM...	60617	33275.5	SOUTH CHICAG...	246598	10	nan	SOUTH CHICAG...
2	11612 S WEST...	60643	30487.5	BEVERLY RECO...	3705	19	nan	BEVERLY RECO...
3	1600 S KOSTN...	60623	128513.7	CHARTER STEE...	293825	24	nan	LEELO STEEL,...
4	1647 W FULTO...	60612	5634.0	SN PECK BUIL...	85595	27	673	S.N. PECK BU...

Merging multiple tables

```
grants_licenses_ward = grants.merge(licenses, on=['address','zip']) \
                        .merge(wards, on='ward', suffixes=('_bus','_ward'))
grants_licenses_ward.head()
```

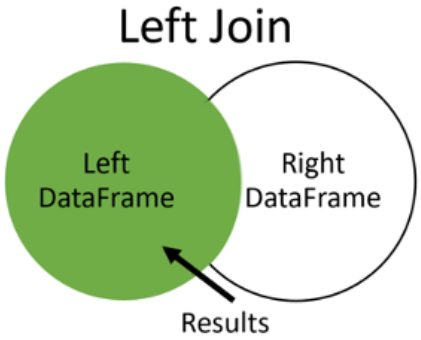
	address_bus	zip_bus	grant	company	account	ward	aid	business	alderma
0	1020 N KOLMA...	60651	68309.8	TRITON INDUS...	7689	37	929	TRITON INDUS...	Emma M.
1	10241 S COMM...	60617	33275.5	SOUTH CHICAG...	246598	10	nan	SOUTH CHICAG...	Susan S
2	11612 S WEST...	60643	30487.5	BEVERLY RECO...	3705	19	nan	BEVERLY RECO...	Matthew
3	3502 W 111TH ST	60655	50000.0	FACE TO FACE...	263274	19	704	FACE TO FACE	Matthew
4	1600 S KOSTN...	60623	128513.7	CHARTER STEE...	293825	24	nan	LEELO STEEL,...	Michael

```
grant_licenses_ward.groupby('ward').agg('sum').plot(kind='bar', y='grant')
plt.show()
```



Left Join

A left join returns all rows of data from the left table and only those rows from the right table where key columns match.



Left Table

A	B	C
A2	B2	C2
A3	B3	C3
A4	B4	C4

Right Table

C	D
C1	D1
C2	D2
C4	D4
C5	D5

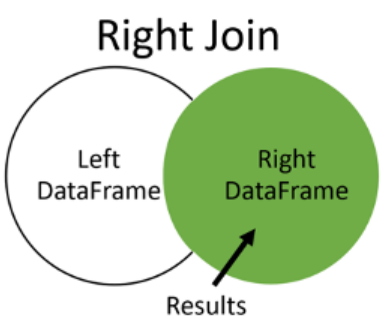
Result Table

A	B	C	D
A2	B2	C2	D2
A3	B3	C3	
A4	B4	C4	D4

Here we have two tables named left and right. We want to use a left join to merge them on key column C. A left join returns all of the rows from the left table and only those rows from the right table where column C matches in both. Notice the second row of the merged table. The columns from the left table are filled in, while the column from the right table is not since there wasn't a match found for that row in the right table.

Right Join

The right join. It will return all of the rows from the right table and includes only those rows from the left table that have matching values. It is the mirror opposite of the left join.



Right join

Left Table

A	B	C
A2	B2	C2
A3	B3	C3
A4	B4	C4

Right Table

C	D
C1	D1
C2	D2
C4	D4
C5	D5

Result Table

A	B	C	D
		C1	D1
A2	B2	C2	D2
A4	B4	C4	D4
		C5	D5

Merge with right join

```
tv_movies = movies.merge(tv_genre, how='right',
                         left_on='id', right_on='movie_id')
print(tv_movies.head())
```

```
   id      title           popularity  release_date  movie_id  genre
0 153397  Restless        0.812776    2012-12-07    153397    TV Movie
1 10947   High School ...  16.536374   2006-01-20    10947     TV Movie
2 231617  Signed, Seal...  1.444476    2013-10-13    231617    TV Movie
3 78814   We Have Your...  0.102003    2011-11-12    78814     TV Movie
4 158150  How to Fall ...  1.923514    2012-07-21    158150    TV Movie
```

Outer Join

Our last type of join is called an outer join. An outer join will return all of the rows from both tables regardless if there is a match between the tables.

Datasets for outer join

```
m = movie_to_genres['genre'] == 'Family'
family = movie_to_genres[m].head(3)
```

```
   movie_id  genre
0  12        Family
1  35        Family
2  105       Family
```

```
m = movie_to_genres['genre'] == 'Comedy'
comedy = movie_to_genres[m].head(3)
```

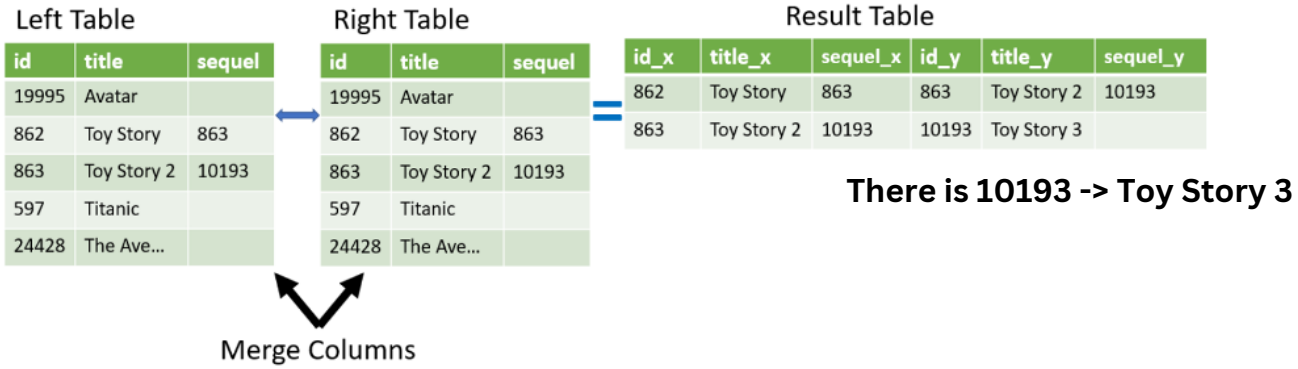
```
   movie_id  genre
0  5         Comedy
1  13        Comedy
2  35        Comedy
```

Merge with outer join

```
family_comedy = family.merge(comedy, on='movie_id', how='outer',
                             suffixes=('_fam', '_com'))
print(family_comedy)
```

```
   movie_id  genre_fam  genre_com
0  12        Family     NaN
1  35        Family     Comedy
2  105       Family     NaN
3  5         NaN        Comedy
4  13        NaN        Comedy
```

Merging a table to itself



There is 10193 -> Toy Story 3

Merging a table to itself

```
original_sequels = sequels.merge(sequels, left_on='sequel', right_on='id',
                                 suffixes=('_org','_seq'))
print(original_sequels.head())
```

```
   id_org  title_org        sequel_org  id_seq  title_seq        sequel_seq
0  862     Toy Story        863         863     Toy Story 2      10193
1  863     Toy Story 2      10193       10193   Toy Story 3      NaN
2  675     Harry Potter...  767         767     Harry Potter...  NaN
3  121     The Lord of ...  122         122     The Lord of ...  NaN
4  120     The Lord of ...  121         121     The Lord of ...  122
```

Merging a table to itself with left join

```
original_sequels = sequels.merge(sequels, left_on='sequel', right_on='id',
                                 how='left', suffixes=('_org','_seq'))
print(original_sequels.head())
```

```
   id_org  title_org     sequel_org  id_seq  title_seq    sequel_seq
0  19995   Avatar        NaN         NaN     NaN          NaN
1  862     Toy Story     863         863     Toy Story 2  10193
2  863     Toy Story 2   10193       10193   Toy Story 3  NaN
3  597     Titanic       NaN         NaN     NaN          NaN
4  24428   The Avengers  NaN         NaN     NaN          NaN
```

Concatenate two tables vertically

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

A	B	C
A4	B4	C4
A5	B5	C5
A6	B6	C6

- pandas `.concat()` method can concatenate both vertical and horizontal.
  - `axis=0` , vertical

```
pd.concat([inv_jan, inv_feb, inv_mar])
```

	iid	cid	invoice_date	total
0	1	2	2009-01-01	1.98
1	2	4	2009-01-02	3.96
2	3	8	2009-01-03	5.94
0	7	38	2009-02-01	1.98
1	8	40	2009-02-01	1.98
2	9	42	2009-02-02	3.96
0	14	17	2009-03-04	1.98
1	15	19	2009-03-04	1.98
2	16	21	2009-03-05	3.96

Ignoring the index

```
pd.concat([inv_jan, inv_feb, inv_mar],
          ignore_index=True)
```

	iid	cid	invoice_date	total
0	1	2	2009-01-01	1.98
1	2	4	2009-01-02	3.96
2	3	8	2009-01-03	5.94
3	7	38	2009-02-01	1.98
4	8	40	2009-02-01	1.98
5	9	42	2009-02-02	3.96
6	14	17	2009-03-04	1.98
7	15	19	2009-03-04	1.98
8	16	21	2009-03-05	3.96

```
# df.groupby(level=0) It specifies the first index of the Dataframe.
avg_inv_by_month_group = avg_inv_by_month.groupby(level=0).agg(
    {"total": "mean"}
)
avg_inv_by_month_group
```

	total
aug	5.660000
jul	5.431429
sep	5.945455

```
avg_inv_by_month_group.plot(kind="bar")
```

Make sure that ignore_index argument is False, since you can't add a key and ignore the index at the same time.

Setting labels to original tables

```
pd.concat([inv_jan, inv_feb, inv_mar],
          ignore_index=False,
          keys=['jan','feb','mar'])
```

		iid	cid	invoice_date	total
jan	0	1	2	2009-01-01	1.98
	1	2	4	2009-01-02	3.96
	2	3	8	2009-01-03	5.94
feb	0	7	38	2009-02-01	1.98
	1	8	40	2009-02-01	1.98
	2	9	42	2009-02-02	3.96
mar	0	14	17	2009-03-04	1.98
	1	15	19	2009-03-04	1.98
	2	16	21	2009-03-05	3.96

Concatenate tables with different columns.

The concat method by default will include all of the columns in the different tables it's combining. The sort argument, if true, will alphabetically sort the different column names in the result. We can see in the results that the billing country for January invoices is NaN. However, there are values for the February invoices.

Table: `inv_jan`

	iid	cid	invoice_date	total
0	1	2	2009-01-01	1.98
1	2	4	2009-01-02	3.96
2	3	8	2009-01-03	5.94

Table: `inv_feb`

	iid	cid	invoice_date	total	bill_ctry
0	7	38	2009-02-01	1.98	Germany
1	8	40	2009-02-01	1.98	France
2	9	42	2009-02-02	3.96	France

```
pd.concat([inv_jan, inv_feb],
          sort=True)
```

	bill_ctry	cid	iid	invoice_date	total
0	NaN	2	1	2009-01-01	1.98
1	NaN	4	2	2009-01-02	3.96
2	NaN	8	3	2009-01-03	5.94
0	Germany	38	7	2009-02-01	1.98
1	France	40	8	2009-02-01	1.98
2	France	42	9	2009-02-02	3.96

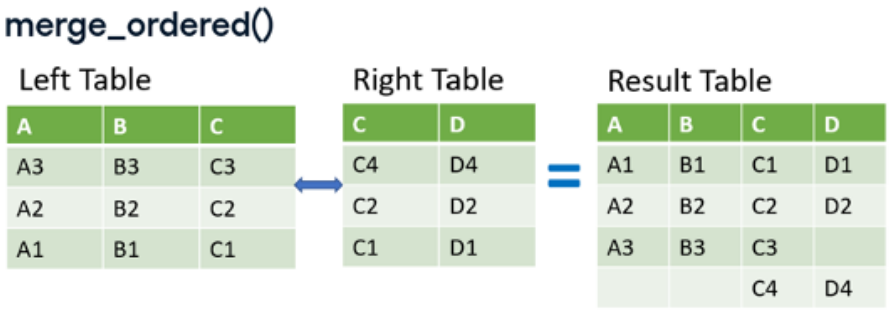
If we only want the matching columns between tables, we can set the join argument to "inner". Its default value is equal to "outer", which is why concat by default will include all of the columns. Additionally, the sort argument has no effect when join equals "inner". The order of the columns will be the same as the input tables. Now the bill country column is gone and we're only left with the columns the tables have in common.

```
pd.concat([inv_jan, inv_feb],
          join='inner')
```

iid	cid	invoice_date	total
1	2	2009-01-01	1.98
2	4	2009-01-02	3.96
3	8	2009-01-03	5.94
7	38	2009-02-01	1.98
8	40	2009-02-01	1.98
9	42	2009-02-02	3.96

Merge_ordered() in Pandas

The merge_ordered method will allow us to merge the left and right tables shown here. We can see the output of the merge when we merge on the "C" column. The results are similar to the standard merge method with an outer join, but here that the results are sorted. The sorted results make this a useful method for ordered or time-series data.



Method comparison

`.merge()` **method:**

- Column(s) to join on
  - `on` , `left_on` , and `right_on`
- Type of join
  - `how` *(left, right, inner, outer)* {{@}}
  - **default** inner
- Overlapping column names
  - `suffixes`
- Calling the method
  - `df1.merge(df2)`

`merge_ordered()` **method:**

- Column(s) to join on
  - `on` , `left_on` , and `right_on`
- Type of join
  - `how` *(left, right, inner, outer)*
  - **default** outer
- Overlapping column names
  - `suffixes`
- Calling the function
  - `pd.merge_ordered(df1, df2)`

Merging stock data

```
import pandas as pd
pd.merge_ordered(appl, mcd, on='date', suffixes=('_aapl','_mcd'))
```

```
  date        close_aapl  close_mcd
0 2007-01-01  NaN         44.349998
1 2007-02-01  12.087143   43.689999
2 2007-03-01  13.272857   45.049999
3 2007-04-01  14.257143   48.279999
4 2007-05-01  17.312857   50.549999
5 2007-06-01  17.434286   NaN
```

Filling missing values with forward fill.

Forward fill

Forward fill example

```
pd.merge_ordered(appl, mcd, on='date',
                 suffixes=('_aapl','_mcd'),
                 fill_method='ffill')
```

```
  date        close_aapl  close_mcd
0 2007-01-01  NaN         44.349998
1 2007-02-01  12.087143   43.689999
2 2007-03-01  13.272857   45.049999
3 2007-04-01  14.257143   48.279999
4 2007-05-01  17.312857   50.549999
5 2007-06-01  17.434286   50.549999
```

```
pd.merge_ordered(appl, mcd, on='date',
                 suffixes=('_aapl','_mcd'))
```

```
  date        close_AAPL  close_mcd
0 2007-01-01  NaN         44.349998
1 2007-02-01  12.087143   43.689999
2 2007-03-01  13.272857   45.049999
3 2007-04-01  14.257143   48.279999
4 2007-05-01  17.312857   50.549999
5 2007-06-01  17.434286   NaN
```

When to use merge_ordered()?

- Ordered data / time series
- Filling in missing values

```
taxi_own_veh = taxi_owner.merge(
    taxi_vehicles, on="vid", suffixes=("_own", "_veh")
)
taxi_own_veh.head()
```

	rid	vid	owner_own	address	zip	make	model	year	fuel_type	owner_veh
0	T6285	6285	AGEAN TAXI LLC	4536 N. ELSTON AVE.	60630	NISSAN	ALTIMA	2011	HYBRID	AGEAN TAXI LLC
1	T4862	4862	MANGIB CORP.	5717 N. WASHTENAW AVE.	60659	HONDA	CRV	2014	GASOLINE	MANGIB CORP.
2	T1495	1495	FUNRIDE, INC.	3351 W. ADDISON ST.	60618	TOYOTA	SIENNA	2015	GASOLINE	FUNRIDE, INC.
3	T4231	4231	ALQUSH CORP.	6611 N. CAMPBELL AVE.	60645	TOYOTA	CAMRY	2014	HYBRID	ALQUSH CORP.
4	T5971	5971	EUNIFFORD INC.	3351 W. ADDISON ST.	60618	TOYOTA	SIENNA	2015	GASOLINE	EUNIFFORD INC.

```
taxi_own_veh[["fuel_type"]].value_counts()
```

```
fuel_type
HYBRID                   2792
GASOLINE                  611
FLEX FUEL                  89
COMPRESSED NATURAL GAS     27
Name: count, dtype: int64
```

value_conts() count how many time does a value occur

```
licenses_owners = licenses.merge(biz_owners, on="account")
licenses_owners.head(10)
```

	account	ward	aid	business	address	zip	first_name	last_name	title
0	307071	3	743	REGGIE'S BAR & GRILL	2105 S STATE ST	60616	ROBERT	GLICK	MEMBER
1	10	10	829	HONEYBEERS	13200 S HOUSTON AVE	60633	PEARL	SHERMAN	PRESIDENT
2	10	10	829	HONEYBEERS	13200 S HOUSTON AVE	60633	PEARL	SHERMAN	SECRETARY
3	10002	14	775	CELINA DELI	5089 S ARCHER AVE	60632	WALTER	MROZEK	PARTNER
4	10002	14	775	CELINA DELI	5089 S ARCHER AVE	60632	CELINA	BYRDAK	PARTNER
5	10005	12	NaN	KRAFT FOODS NORTH AMERICA	2005 W 43RD ST	60609	IRENE	ROSENFELD	PRESIDENT
6	10005	12	NaN	KRAFT FOODS NORTH AMERICA	2005 W 43RD ST	60609	CAROL	WARD	SECRETARY
7	10044	44	638	NEYBOUR'S TAVERN & GRILLE	3651 N SOUTHPORT AVE	60613	JESSICA	DEVOS	SECRETARY
8	10044	44	638	NEYBOUR'S TAVERN & GRILLE	3651 N SOUTHPORT AVE	60613	CURTIS	JENNETTE	PRESIDENT
9	10044	44	638	NEYBOUR'S TAVERN & GRILLE	3651 N SOUTHPORT AVE	60613	BRIAN	HAINES	VICE PRESIDENT

```
"""
Count column by .agg({'account':'count'}) after groupby
"""
counted_df = licenses_owners.groupby("title").agg({"account": "count"})
counted_df.head()
```

	account
title	
ASST. SECRETARY	111
BENEFICIARY	4
CEO	110
DIRECTOR	146
EXECUTIVE DIRECTOR	10

```
sorted_df = counted_df.sort_values(by="account", ascending=False)
sorted_df.head()
```

	account
title	
PRESIDENT	6259
SECRETARY	5205
SOLE PROPRIETOR	1658
OTHER	1200
VICE PRESIDENT	970

```
"""
// IMPORATNT
"""
filter_criteria = (
    (ridership_cal_stations["month"] == 7)
    & (ridership_cal_stations["day_type"] == "Weekday")
    & (ridership_cal_stations["station_name"] == "Wilson")
)
filter_criteria
```

```
0       False
1       False
2       False
3       False
4       False
        ...
3280    False
3281    False
3282    False
3283    False
3284    False
Length: 3285, dtype: bool
```

```
ridership_cal_stations.loc[filter_criteria].head()
```

	station_id	year	month	day	rides	day_type	station_name	location
1641	40540	2019	7	1	6464	Weekday	Wilson	(41.964273, -87.657588)
1642	40540	2019	7	2	6491	Weekday	Wilson	(41.964273, -87.657588)
1643	40540	2019	7	3	6639	Weekday	Wilson	(41.964273, -87.657588)
1645	40540	2019	7	5	4794	Weekday	Wilson	(41.964273, -87.657588)
1648	40540	2019	7	8	6351	Weekday	Wilson	(41.964273, -87.657588)

```
ridership_cal_stations.loc[filter_criteria, "rides"].head()
```

```
1641    6464
1642    6491
1643    6639
1645    4794
1648    6351
Name: rides, dtype: int64
```

```
ridership_cal_stations.loc[filter_criteria, "rides"].sum()
```

```
140005
```

cal and ridership must have year, month, data

```
ridership_cal_stations = ridership.merge(cal, on=["year", "month", "day"])
ridership_cal_stations.head()
```

	station_id	year	month	day	rides	day_type
0	40010	2019	1	1	576	Sunday/Holiday
1	40080	2019	1	1	1839	Sunday/Holiday
2	40770	2019	1	1	2724	Sunday/Holiday
3	40120	2019	1	1	754	Sunday/Holiday
4	40540	2019	1	1	2175	Sunday/Holiday

```
licenses_zip_ward.groupby("alderman").agg({"income": "median"}).head()
```

	income
alderman	
Ameya Pawar	66246.0
Anthony A. Beale	38206.0
Anthony V. Napolitano	82226.0
Ariel E. Reyboras	41307.0
Brendan Reilly	110215.0

```
number_of_missing_fin = movies_financials["budget"].isnull().sum()
number_of_missing_fin
```

```
1574
```

```
scifi_only = action_scifi[action_scifi["genre_act"].isnull()]
scifi_only
```

	movie_id	genre_act	genre_sci
2	19	NaN	Science Fiction
3	38	NaN	Science Fiction
4	62	NaN	Science Fiction
5	68	NaN	Science Fiction
6	74	NaN	Science Fiction
...	...	...	...
529	333371	NaN	Science Fiction
530	335866	NaN	Science Fiction
531	347548	NaN	Science Fiction
532	360188	NaN	Science Fiction
534	371690	NaN	Science Fiction

```
boolean_filter = (crews_self_merged["job_dir"] == "Director") & (
    crews_self_merged["job_crew"] != "Director"
)
crews_self_merged[boolean_filter].head()
```

	id	department_dir	job_dir	name_dir	department_crew	job_crew	name_crew
156	19995	Directing	Director	James Cameron	Editing	Editor	Stephen E. Rivkin
157	19995	Directing	Director	James Cameron	Sound	Sound Designer	Christopher Boyes
158	19995	Directing	Director	James Cameron	Production	Casting	Mali Finn
160	19995	Directing	Director	James Cameron	Writing	Writer	James Cameron
161	19995	Directing	Director	James Cameron	Art	Set Designer	Richard F. Mays

```
gdp_sp500 = pd.merge_ordered(
    sp500, gdp, how="left", right_on="year", left_on="date"
)
gdp_sp500.head()
```

	date	returns	country code	year	gdp
0	2008	-38.49	NaN	NaN	NaN
1	2009	23.45	NaN	NaN	NaN
2	2010	12.78	USA	2010.0	1.500000e+13
3	2011	0.00	USA	2011.0	1.550000e+13
4	2012	13.41	USA	2012.0	1.620000e+13

```
gdp_sp500 = pd.merge_ordered(
    gdp, sp500, how="left", left_on="year", right_on="date", fill_method="ffill"
)
gdp_sp500
```

	country code	year	gdp	date	returns
0	USA	2010	1.500000e+13	2010	12.78
1	USA	2011	1.550000e+13	2011	0.00
2	USA	2012	1.620000e+13	2012	13.41
3	USA	2012	1.620000e+13	2012	13.41
4	USA	2013	1.680000e+13	2013	29.60
5	USA	2014	1.750000e+13	2014	11.39
6	USA	2015	1.820000e+13	2015	-0.73
7	USA	2016	1.870000e+13	2016	9.54
8	USA	2017	1.950000e+13	2017	19.42
9	USA	2018	2.050000e+13	2017	19.42

```
inflation_unemploy[["unemployment_rate", "cpi"]].corr()
```

	unemployment_rate	cpi
unemployment_rate	1.000000	-0.868388
cpi	-0.868388	1.000000

W7 statistics

What is statistics?

- The field of statistics – the practice and study of collecting and analyzing data.
- A summary statistics – a fact about or summary of some data.
- What can statistics do?
  - How likely is someone to purchase a product? Are people more likely to purchase it if they can use a different payment system?
  - How many occupants will your hotel have? How can you optimize occupancy?
  - How many sizes of jeans need to be manufactured so they can fit 95% of the population? Should the same number of each size be produced?
  - A/B tests: Which ad is more effective in getting people to purchase a product?

What can't statistics do?

- Why is Game of Thrones so popular?

Instead…

- Are series with more violent scenes viewed by more people?

But…

- Even so, this can't tell us if more violent scenes lead to more views.

Types of statistics

Descriptive statistics

- *Describe* and summarize data

- 50% of friends drive to work
- 25% take the bus
- 25% bike

Inferential statistics

- Use a sample of data to make *inferences* about a larger population

What percent of people drive to work?

41

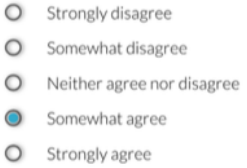
Types of data

Numeric (Quantitative)

- **Continuous (Measured)**
  - Airplane speed
  - Time spent waiting in line
- **Discrete (Counted)**
  - Number of pets
  - Number of packages shipped

Categorical (Qualitative)

- **Nominal (Unordered)**
  - Married/unmarried
  - Country of residence
- **Ordinal (Ordered)**



Categorical data can be represented as numbers

Nominal (Unordered)

- Married/unmarried (`1` / `0`)
- Country of residence (`1`, `2`, ...)

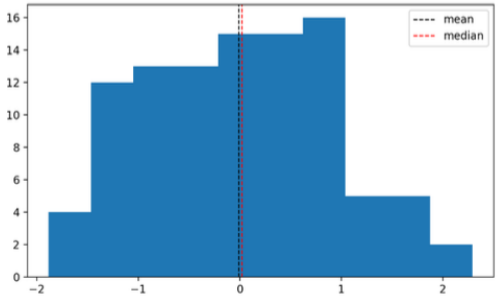
Ordinal (Ordered)

- Strongly disagree (`1`)
- Somewhat disagree (`2`)
- Neither agree nor disagree (`3`)
- Somewhat agree (`4`)
- Strongly agree (`5`)

Being able to identify data types is important since the type of data you're working with will dictate what kinds of summary statistics and visualizations make sense for your data, so this is an important skill to master. For numerical data, we can use summary statistics like mean, and plots like scatter plots, but these don't make a ton of sense for categorical data.

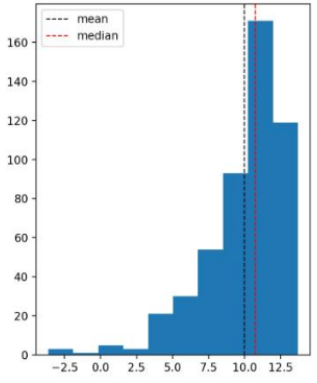
Which measure (mean vs. median) to use?

Measures of center
mean, median, mode?

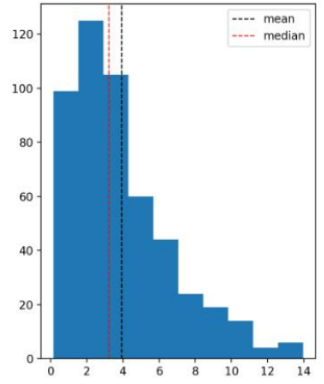


When data is skewed, the mean and median are different. The mean is pulled in the direction of the skew, so it's lower than the median on the left-skewed data, and higher than the median on the right-skewed data. Because the mean is pulled around by the extreme values, it's better to use the median since it's less affected by outliers.

Left-skewed



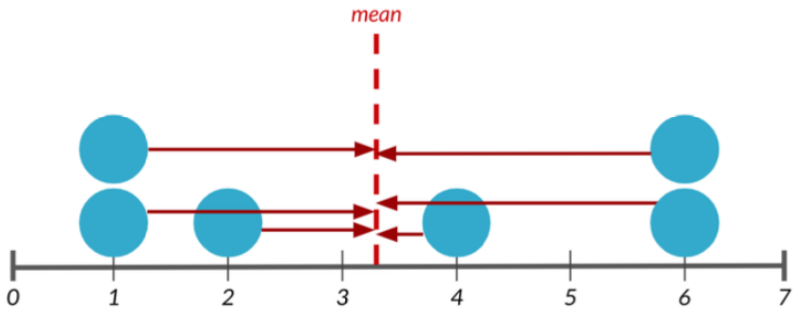
Right-skewed



Measures of spread

- What is spread?
  - Spread is just what it sounds like - it describes how spread apart or close together the data points are. Just like measures of center, there are a few different measures of spread.
- Variance - measures the average distance from each data point to the data's mean.

Average distance from each data point to the data's mean



Calculating variance

1. Subtract mean from each data point

```
dists = msleep['sleep_total'] -
        np.mean(msleep['sleep_total'])
print(dists)
```

```
0     1.666265
1     6.566265
2     3.966265
3     4.466265
4    -6.433735
      ...
```

2. Square each distance

```
sq_dists = dists ** 2
print(sq_dists)
```

```
0      2.776439
1     43.115837
2     15.731259
3     19.947524
4     41.392945
      ...
```

Calculating variance

3. Sum squared distances

```
sum_sq_dists = np.sum(sq_dists)
print(sum_sq_dists)
```

```
1624.065542
```

4. Divide by number of data points - 1

```
variance = sum_sq_dists / (83 - 1)
print(variance)
```

```
19.805677
```

We can calculate the variance in one step using np.var, setting the ddof argument to 1. If we don't specify ddof equals 1, a slightly different formula is used to calculate variance that should only be used on a full population, not a sample.

Use `np.var()`

```
np.var(msleep['sleep_total'], ddof=1)
```

```
19.805677
```

*Without `ddof=1`, **population variance** is calculated instead of **sample variance**:*

```
np.var(msleep['sleep_total'])
```

```
19.567055
```

Standard deviation

```
np.sqrt(np.var(msleep['sleep_total'], ddof=1))
```

```
4.450357
```

```
np.std(msleep['sleep_total'], ddof=1)
```

```
4.450357
```

Quartiles:

```
np.quantile(msleep['sleep_total'], [0, 0.25, 0.5, 0.75, 1])
```

```
array([ 1.9 ,  7.85, 10.1 , 13.75, 19.9 ])
```

```
np.quantile(msleep['sleep_total'], 0.5)
```

```
10.1
```

0.5 quantile = median

`np.linspace(start, stop, num)`

```
np.quantile(msleep['sleep_total'], np.linspace(0, 1, 5))
```

```
array([ 1.9 ,  7.85, 10.1 , 13.75, 19.9 ])
```

Interquartile range (IQR)

Height of the box in a boxplot

```
np.quantile(msleep['sleep_total'], 0.75) - np.quantile(msleep['sleep_total'], 0.25)
```

```
5.9
```

```
from scipy.stats import iqr
iqr(msleep['sleep_total'])
```

```
5.9
```

Outliers

Outlier: data point that is substantially different from the others

How do we know what a substantial difference is? A data point is an outlier if:

- $\text{data} < \text{Q1} - 1.5 \times \text{IQR}$ or
- $\text{data} > \text{Q3} + 1.5 \times \text{IQR}$

Identifying thresholds

```
# 75th percentile
seventy_fifth = salaries["Salary_USD"].quantile(0.75)

# 25th percentile
twenty_fifth = salaries["Salary_USD"].quantile(0.25)

# Interquartile range
salaries_iqr = seventy_fifth - twenty_fifth

print(salaries_iqr)
```

```
from scipy.stats import iqr
iqr = iqr(msleep['bodywt'])
lower_threshold = np.quantile(msleep['bodywt'], 0.25) - 1.5 * iqr
upper_threshold = np.quantile(msleep['bodywt'], 0.75) + 1.5 * iqr
```

```
msleep[(msleep['bodywt'] < lower_threshold) | (msleep['bodywt'] > upper_threshold)]
```

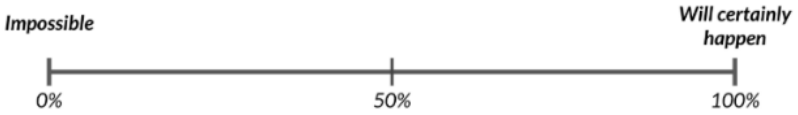
	name	vore	sleep_total	bodywt
4	Cow	herbi	4.0	600.000
20	Asian elephant	herbi	3.9	2547.000
22	Horse	herbi	2.9	521.000

What's the probability of an event?

$$P(\text{event}) = \frac{\#\text{ ways event can happen}}{\text{total }\#\text{ of possible outcomes}}$$

Example: a coin flip

$$P(\text{heads}) = \frac{1\text{ way to get heads}}{2\text{ possible outcomes}} = \frac{1}{2} = 50\%$$



$$P(\text{Brian}) = \frac{1}{4} = 25\%$$

```
np.random.seed(10)
sales_counts.sample()
```

You can put number here to do n time

```
sales_counts.sample(5, replace = True)
```

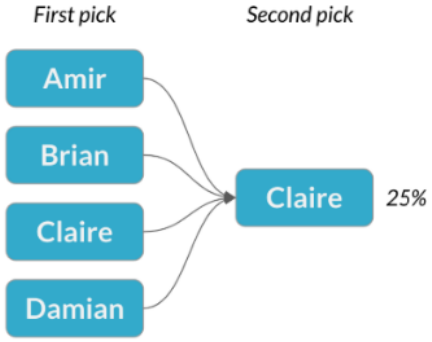
```
    name  n_sales
1  Brian      128
```

Independent events

*Two events are **independent** if the probability of the second event **isn't** affected by the outcome of the first event.*

Sampling with replacement = each pick is independent

Sampling with Replacement

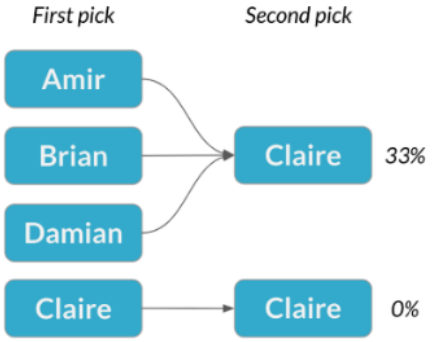


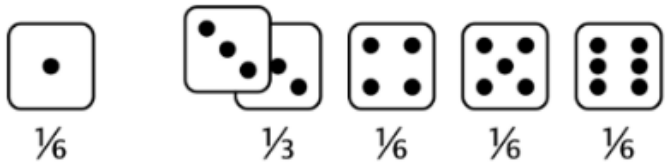
Dependent events

*Two events are **dependent** if the probability of the second event **is** affected by the outcome of the first event.*

Sampling without replacement = each pick is dependent

Sampling without Replacement

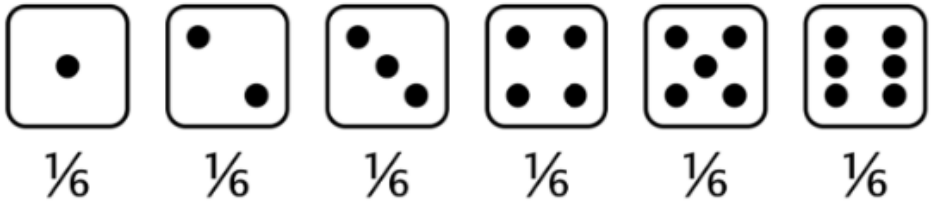




Expected value of uneven die roll =

$$(1 \times \tfrac{1}{6}) + (2 \times 0) + (3 \times \tfrac{1}{3}) + (4 \times \tfrac{1}{6}) + (5 \times \tfrac{1}{6}) + (6 \times \tfrac{1}{6}) = 3.67$$

Describes the probability of each possible outcome in a scenario

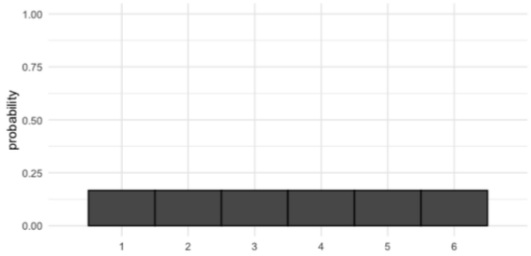


Expected value: mean of a probability distribution

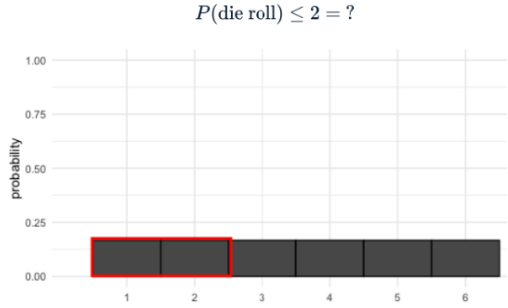
Expected value of a fair die roll =

$$(1 \times \tfrac{1}{6}) + (2 \times \tfrac{1}{6}) + (3 \times \tfrac{1}{6}) + (4 \times \tfrac{1}{6}) + (5 \times \tfrac{1}{6}) + (6 \times \tfrac{1}{6}) = 3.5$$

Visualizing a probability distribution



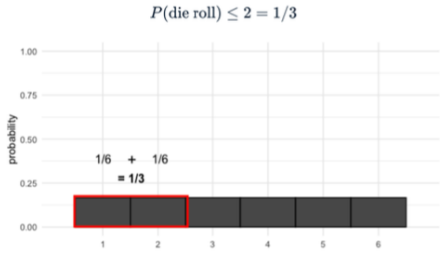
Probability = area



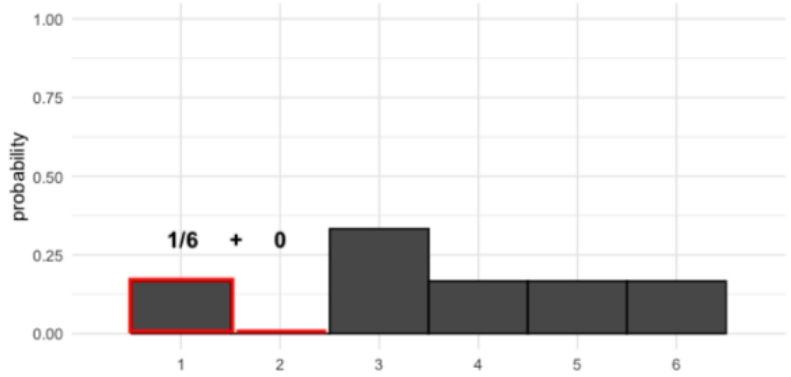
We can calculate probabilities of different outcomes by taking areas of the probability distribution. For example, what's the probability that our die roll is less than or equal to 2?

Each bar has a width of 1 and a height of one sixth, so the area of each bar is one sixth. We'll sum the areas for 1 and 2, to get a total probability of one third.

Probability = area

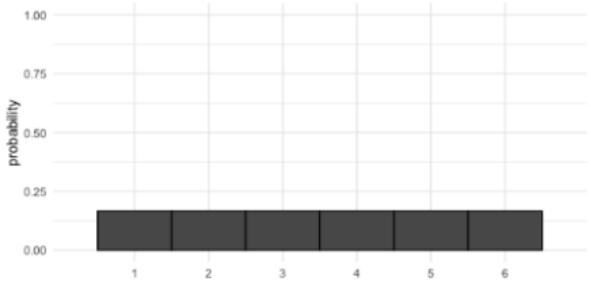


$$P(\text{uneven die roll}) \leq 2 = 1/6$$

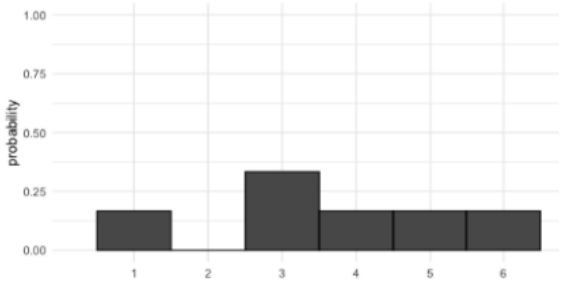


Describe probabilities for discrete outcomes

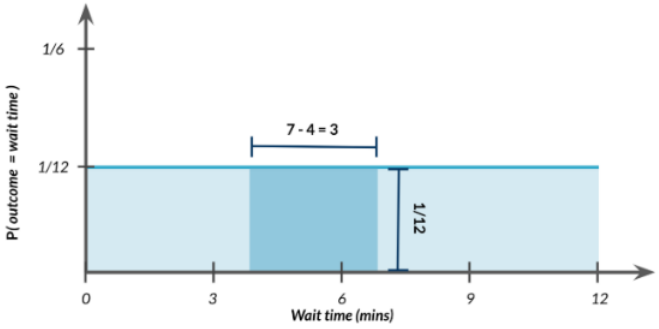
Fair die



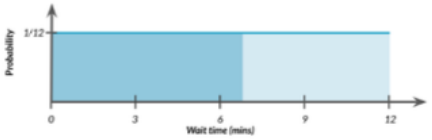
Uneven die



$$P(4 \leq \text{wait time} \leq 7) = 3 \times 1/12 = 3/12$$



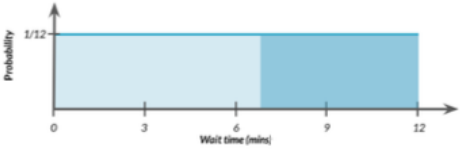
$$P(\text{wait time} \leq 7)$$




```
from scipy.stats import uniform
uniform.cdf(7, 0, 12)
```

```
0.5833333
```

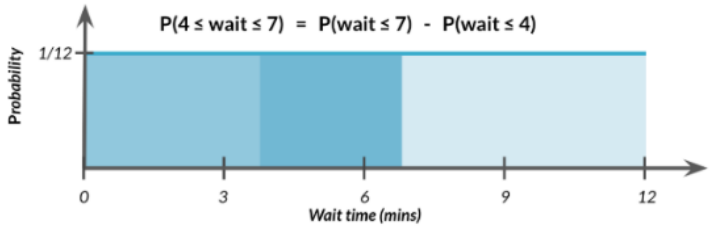
$$P(\text{wait time} \geq 7) = 1 - P(\text{wait time} \leq 7)$$




```
from scipy.stats import uniform
1 - uniform.cdf(7, 0, 12)
```

```
0.4166667
```

$$P(4 \leq \text{wait time} \leq 7)$$

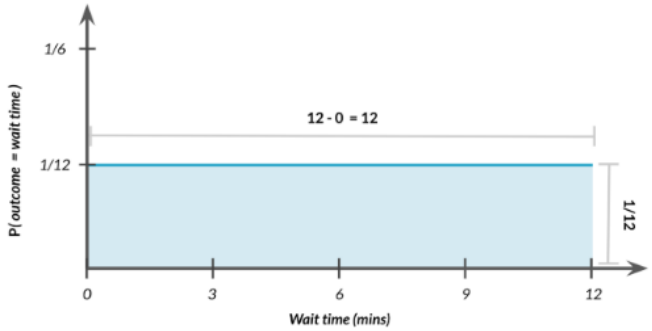



```
from scipy.stats import uniform
uniform.cdf(7, 0, 12) - uniform.cdf(4, 0, 12)
```

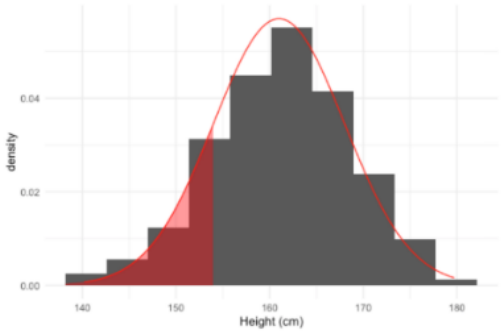
```
0.25
```

Total area = 1

$$P(0 \leq \text{outcome} \leq 12) = 12 \times 1/12 = 1$$



What percent of women are shorter than 154 cm?



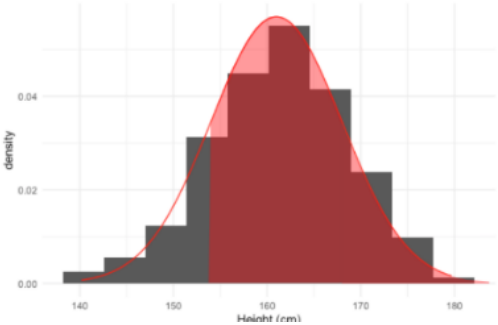

```
from scipy.stats import norm
norm.cdf(154, 161, 7)
```

```
0.158655
```

7 is std

16% of women in the survey are shorter than 154 cm

What percent of women are taller than 154 cm?




```
from scipy.stats import norm
1 - norm.cdf(154, 161, 7)
```

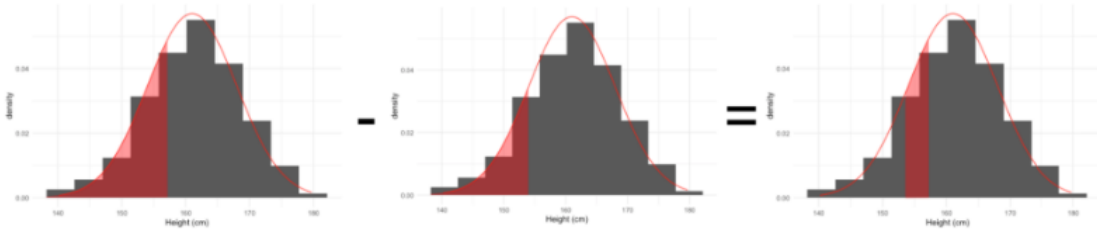
```
0.841345
```

Generating random numbers according to uniform distribution

```
from scipy.stats import uniform
uniform.rvs(0, 5, size=10)
```

```
array([1.89740094, 4.70673196, 0.33224683, 1.0137103 , 2.31641255,
       3.49969897, 0.29688598, 0.92057234, 4.71086658, 1.56815855])
```

What percent of women are 154-157 cm?




```
norm.cdf(157, 161, 7) - norm.cdf(154, 161, 7)
```

```
0.1252
```

Generating random numbers

```
# Generate 10 random heights
norm.rvs(161, 7, size=10)
```

```
array([155.5758223 , 155.13133235, 160.06377097, 168.33345778,
       165.92273375, 163.32677057, 165.13280753, 146.36133538,
       149.07845021, 160.5790856 ])
```

The central limit theorem

Rolling the dice 5 times

```
die = pd.Series([1, 2, 3, 4, 5, 6])
# Roll 5 times
samp_5 = die.sample(5, replace=True)
print(samp_5)
```

```
array([3, 1, 4, 1, 1])
```

```
np.mean(samp_5)
```

```
2.0
```

```
# Roll 5 times and take mean
samp_5 = die.sample(5, replace=True)
np.mean(samp_5)
```

```
4.4
```

```
samp_5 = die.sample(5, replace=True)
np.mean(samp_5)
```

```
3.8
```

Rolling the dice 5 times 10 times

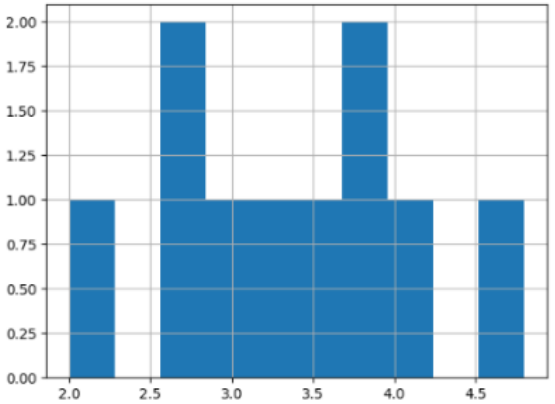
Repeat 10 times:

- Roll 5 times
- Take the mean

```
sample_means = []
for i in range(10):
  samp_5 = die.sample(5, replace=True)
  sample_means.append(np.mean(samp_5))
print(sample_means)
```

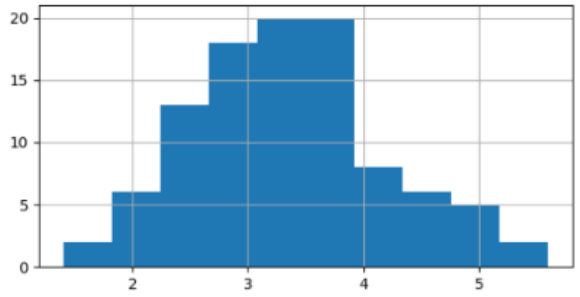
```
[3.8, 4.0, 3.8, 3.6, 3.2, 4.8, 2.6,
3.0, 2.6, 2.0]
```

Sampling distribution of the sample mean



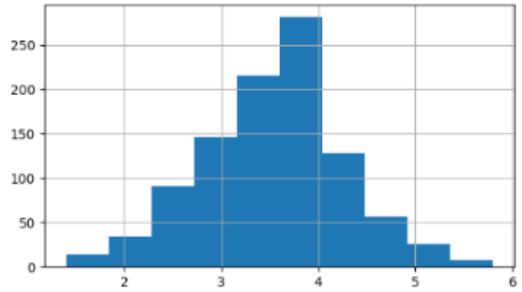
100 sample means

```
sample_means = []
for i in range(100):
    sample_means.append(np.mean(die.sample(5, replace=True)))
```



1000 sample means

```
sample_means = []
for i in range(1000):
    sample_means.append(np.mean(die.sample(5, replace=True)))
```



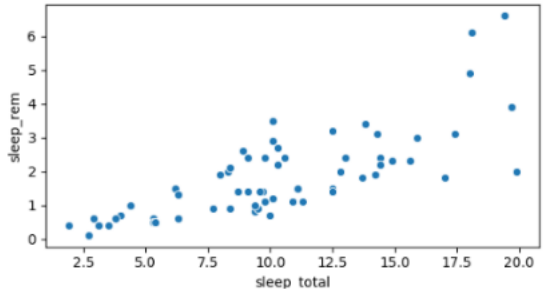
Computing correlation

```
msleep['sleep_total'].corr(msleep['sleep_rem'])
```

```
0.751755
```

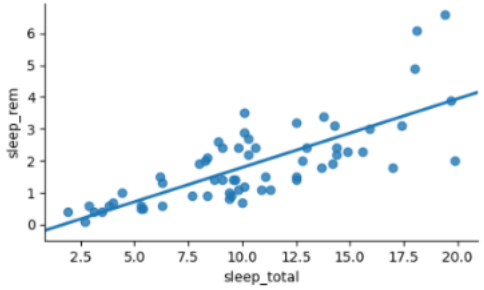
Visualizing relationships

```
import seaborn as sns
sns.scatterplot(x="sleep_total", y="sleep_rem", data=msleep)
plt.show()
```



Adding a trendline

```
import seaborn as sns
sns.lmplot(x="sleep_total", y="sleep_rem", data=msleep, ci=None)
plt.show()
```

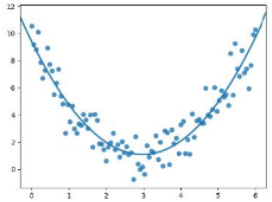


Correlation caveats

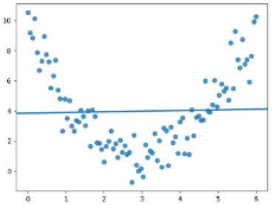
- Consider the graph. There is clearly a relationship between x and y, but when we calculate the correlation, we get 0.18.
- This is because the relationship between the two variables is a quadratic relationship, not a linear relationship. The correlation coefficient measures the strength of linear relationships, and linear relationships only.

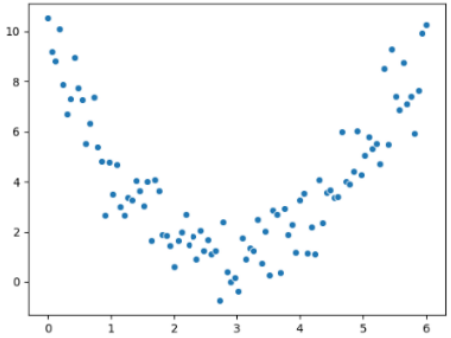
Non-linear relationships

What we see:



What the correlation coefficient sees:





$r = 0.18$

Log transformation

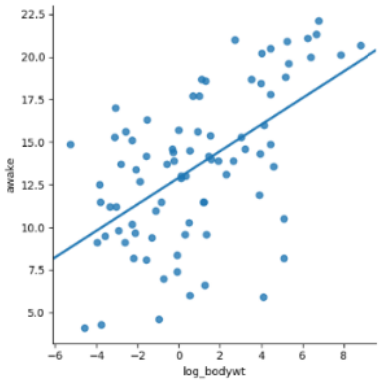
- When data is highly skewed like this, we can apply a log transformation.
- We'll create a new column called log_bodywt which holds the log of each body weight. We can do this using np.log. If we plot the log of bodyweight versus awake time, the relationship looks much more linear than the one between regular bodyweight and awake time. The correlation between the log of bodyweight and awake time is about 0.57, which is much higher than the 0.3 we had before.

Log transformation

```
msleep['log_bodywt'] = np.log(msleep['bodywt'])

sns.lmplot(x='log_bodywt',
           y='awake',
           data=msleep,
           ci=None)
plt.show()
```

```
msleep['log_bodywt'].corr(msleep['awake'])
```

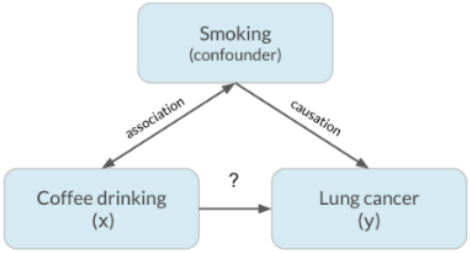
```
0.5687943
```



When our original continuous data do not follow the bell curve, we can log transform this data to make
it as “normal” as possible so that the statistical analysis results from this data become more valid.
In other words, the log transformation reduces or removes the skewness of our original data.

A phenomenon called confounding can lead to spurious correlations. Let's say we want to know if drinking coffee causes lung cancer. Looking at the data, we find that coffee drinking and lung cancer are correlated, which may lead us to think that drinking more coffee will give you lung cancer.

Confounding

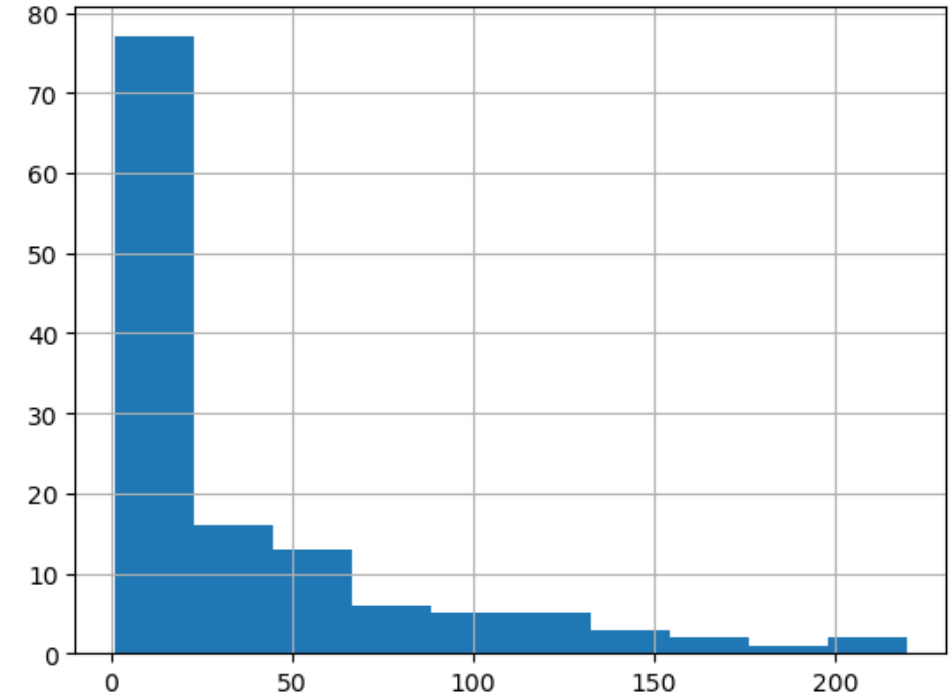


However, there is a third, hidden variable at play, which is smoking. It is also known that smoking causes lung cancer.

In reality, it turns out that coffee does not cause lung cancer and is only associated with it, but it appeared causal due to the third variable, smoking. This third variable is called a confounder, or lurking variable. This means that the relationship of interest between coffee and lung cancer is a spurious correlation.

```python
# rice_consumption['co2_emission'].plot(kind="hist")
# plt.grid()
rice_consumption["co2_emission"].hist()
```

```
<Axes: >
```




```python
food_category = food_consumption.groupby("food_category")["co2_emission"].agg(
    [np.var, np.std]
)
food_category
```

food_category	var	std
beef	88748.408132	297.906710
dairy	17671.891985	132.935669
eggs	21.371819	4.622966
fish	921.637349	30.358481
lamb_goat	16475.518363	128.356996
nuts	35.639652	5.969895
pork	3094.963537	55.632396
poultry	245.026801	15.653332
rice	2281.376243	47.763754
soybeans	0.879882	0.938020
wheat	71.023937	8.427570

```python
# Mean is greater than median
food_consumption[food_consumption["food_category"] == "beef"][
    "co2_emission"
].hist()
```

```python
total = amir_deals["product"].value_counts().sum()

(counts / total) * 100
# amir_deals.shape[0] gets number of rows
```

```
product
Product B    34.831461
Product D    22.471910
Product A    12.921348
Product C     8.426966
Product F     6.179775
Product H     4.494382
Product I     3.932584
Product E     2.808989
Product N     1.685393
Product G     1.123596
Product J     1.123596
Name: count, dtype: float64
```

```python
food_consumption[food_consumption["food_category"] == "beef"][
    "co2_emission"
].mean()
```

```
.102
```

```python
food_consumption[food_consumption["food_category"] == "beef"][
    "co2_emission"
].median()
```

```
restaurant_groups.hist(bins=[2, 3, 4, 5, 6])
```

```
size_dist = (
    restaurant_groups["group_size"].value_counts() / restaurant_groups.shape[0]
)
size_dist
```

```
size_dist = (
    restaurant_groups["group_size"].value_counts() / restaurant_groups.shape[0]
)

size_dist = size_dist.reset_index()
size_dist.columns = ["group_size", "prob"]
print(size_dist)

expected_value = np.sum(size_dist["group_size"] * size_dist["prob"])
print("The expected value is", expected_value)

groups_4_or_more = size_dist[size_dist["group_size"] >= 4]

prob_4_or_more = np.sum(groups_4_or_more["prob"])
print("The probability is ", prob_4_or_more)
```

```
from scipy.stats import uniform

min_time = 0
max_time = 30

prob_less_than_5 = uniform.cdf(4, min_time, max_time)
print(prob_less_than_5)

prob_more_than_5 = 1 - prob_less_than_5
print(prob_more_than_5)

prob_between_10_and_20 = uniform.cdf(20, min_time, max_time) - uniform.cdf(
    10, min_time, max_time
)
print(prob_between_10_and_20)
```

W8

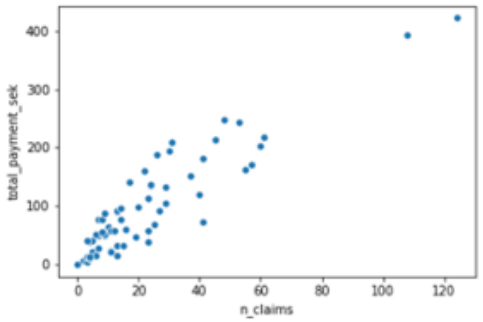
What is regression?

- Statistical models to explore the relationship between a response variable and some explanatory variables.
- Given values of explanatory variables, you can predict the values of the response variable.
- **Response variable** (a.k.a. dependent variable) The variable that you want to predict.
- **Explanatory variables** (a.k.a. independent variables) The variables that explain how the response variable will change.

n_claims	total_payment_sek
108	3925
19	462
13	157
124	4222
40	1194
200	**???**

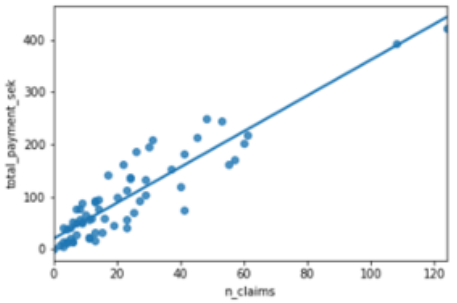
Visualizing pairs of variables

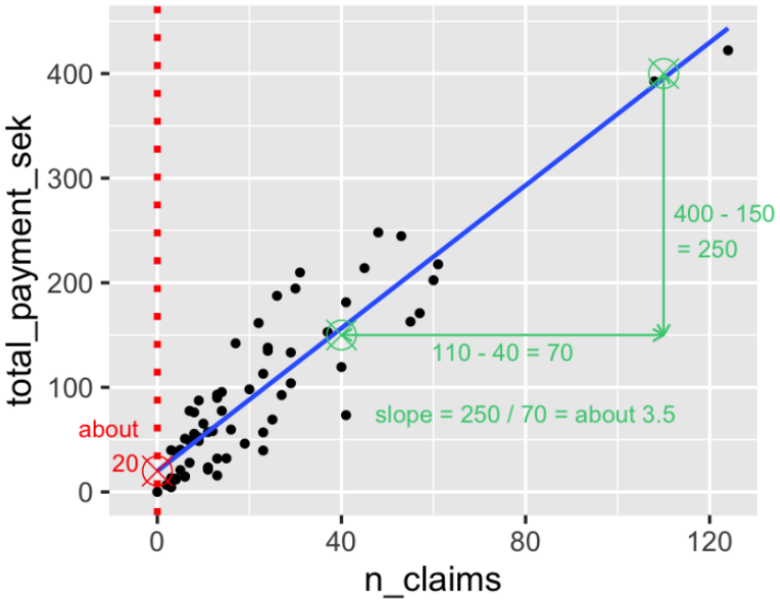
```
import matplotlib.pyplot as plt
import seaborn as sns

sns.scatterplot(x="n_claims",
                y="total_payment_sek",
                data=swedish_motor_insurance)

plt.show()
```



Adding a linear trend line

```
sns.regplot(x="n_claims",
            y="total_payment_sek",
            data=swedish_motor_insurance,
            ci=None)
```





Straight lines are defined by two things

Intercept

The y value at the point when x is zero.

Slope

The amount the y value increases if you increase x by one.

Equation

$$y = \text{intercept} + \text{slope} * x$$

Running a model

```
from statsmodels.formula.api import ols
mdl_payment_vs_claims = ols("total_payment_sek ~ n_claims",
                            data=swedish_motor_insurance)

mdl_payment_vs_claims = mdl_payment_vs_claims.fit()
print(mdl_payment_vs_claims.params)
```

```
Intercept    19.994486
n_claims      3.413824
```

Interpreting the model coefficients

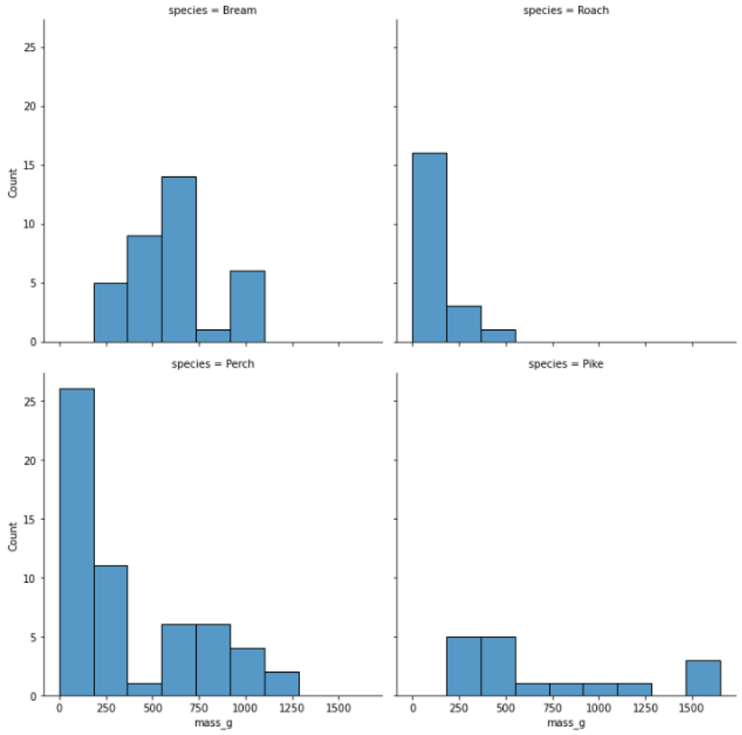
```
Intercept    19.994486
n_claims      3.413824
dtype: float64
```

Equation

$$\text{total\_payment\_sek} = 19.99 + 3.41 * \text{n\_claims}$$

Visualizing 1 numeric and 1 categorical variable

```
import matplotlib.pyplot as plt
import seaborn as sns

sns.displot(data=fish,
            x="mass_g",
            col="species",
            col_wrap=2,
            bins=9)

plt.show()
```



Plot histogram using displot in seaborn.

Summary statistics: mean mass by species

```
summary_stats = fish.groupby("species")["mass_g"].mean()
print(summary_stats)
```

```
species
Bream    617.828571
Perch    382.239286
Pike     718.705882
Roach    152.050000
```

```python
from statsmodels.formula.api import ols
mdl_mass_vs_species = ols("mass_g ~ species", data=fish).fit()
print(mdl_mass_vs_species.params)
```

```
Intercept           617.828571
species[T.Perch]   -235.589286
species[T.Pike]     100.877311
species[T.Roach]   -465.778571
```

This time we have four values: an intercept, and one coefficient for three of the fish species. A coefficient for bream is missing, but the number for the intercept looks familiar. The intercept is the mean mass of the bream that you just calculated.

```
species
Bream    617.828571
Perch    382.239286
Pike     718.705882
Roach    152.050000
Name: mass_g, dtype: float64
```

Model with or without an intercept

From previous slide, model with intercept

```python
mdl_mass_vs_species = ols(
  "mass_g ~ species", data=fish).fit()
print(mdl_mass_vs_species.params)
```

```
Intercept           617.828571
species[T.Perch]   -235.589286
species[T.Pike]     100.877311
species[T.Roach]   -465.778571
```

The coefficients are relative to the intercept: $617.83 - 235.59 = 382.24$!

Model without an intercept

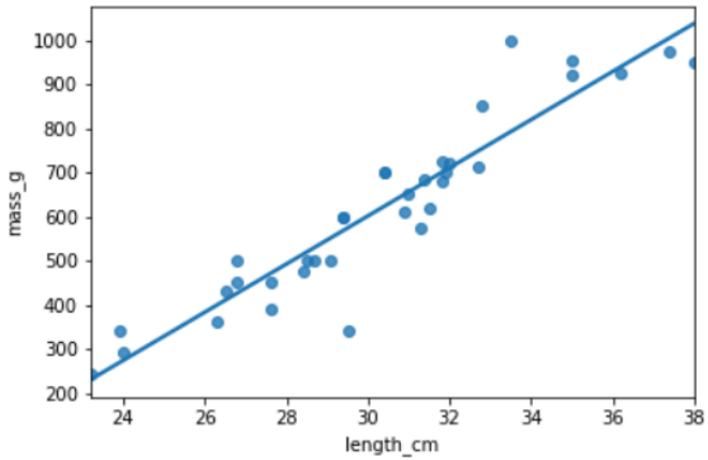
```python
mdl_mass_vs_species = ols(
  "mass_g ~ species + 0", data=fish).fit()
print(mdl_mass_vs_species.params)
```

```
species[Bream]    617.828571
species[Perch]    382.239286
species[Pike]     718.705882
species[Roach]    152.050000
```

In case of a single, categorical variable, coefficients are the means.

Plotting mass vs. length

```
sns.regplot(x="length_cm",
            y="mass_g",
            data=bream,
            ci=None)

plt.show()
```



Running the model

```
mdl_mass_vs_length = ols("mass_g ~ length_cm", data=bream).fit()
print(mdl_mass_vs_length.params)
```

mass as the response variable and species as the explanatory variable

```
Intercept   -1035.347565
length_cm      54.549981
dtype: float64
```

To view the coefficients of the model, we use the .params attribute.

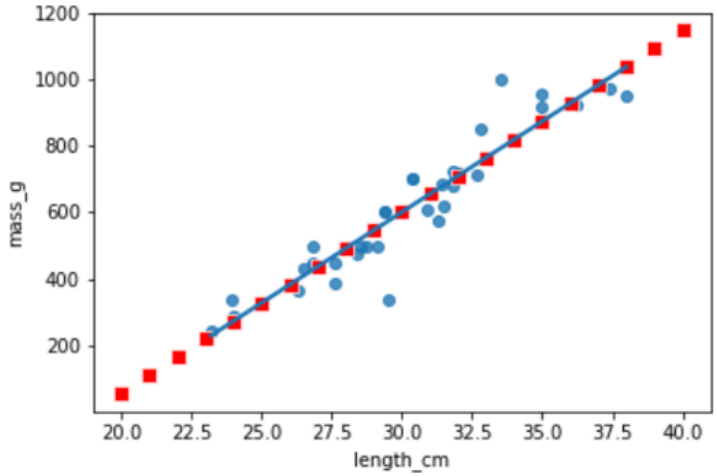
Predicting inside a DataFrame

```
explanatory_data = pd.DataFrame(
  {"length_cm": np.arange(20, 41)}
)
prediction_data = explanatory_data.assign(
    mass_g=mdl_mass_vs_length.predict(explanatory_data)
)
print(prediction_data)
```

	length_cm	mass_g
0	20	55.652054
1	21	110.202035
2	22	164.752015
3	23	219.301996
4	24	273.851977
..	...	...
16	36	928.451749
17	37	983.001730
18	38	1037.551710
19	39	1092.101691
20	40	1146.651672

Showing predictions

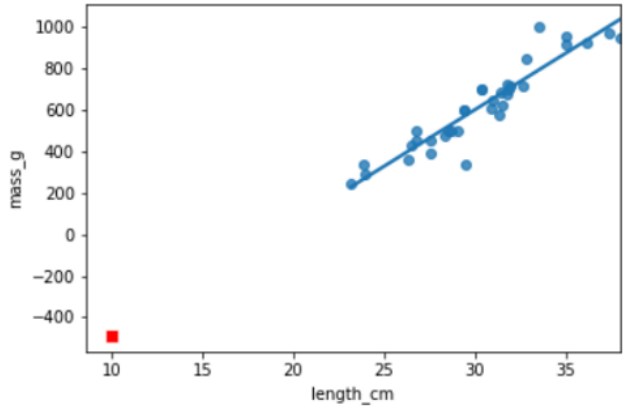
```
import matplotlib.pyplot as plt
import seaborn as sns
fig = plt.figure()
sns.regplot(x="length_cm",
            y="mass_g",
            ci=None,
            data=bream,)
sns.scatterplot(x="length_cm",
                y="mass_g",
                data=prediction_data,
                color="red",
                marker="s")
plt.show()
```



Extrapolating means making predictions outside the range of observed data.

```
little_bream = pd.DataFrame({"length_cm": [10]})

pred_little_bream = little_bream.assign(
    mass_g=mdl_mass_vs_length.predict(little_bream))

print(pred_little_bream)
```

```
   length_cm      mass_g
0         10 -489.847756
```



An explanatory variable is what you manipulate or observe changes in (e.g., caffeine dose), while a response variable is what changes as a result (e.g., reaction times).

.fittedvalues attribute

Fitted values: predictions on the original dataset

```
print(mdl_mass_vs_length.fittedvalues)
```

or equivalently

```
explanatory_data = bream["length_cm"]

print(mdl_mass_vs_length.predict(explanatory_data))
```

```
0      230.211993
1      273.851977
2      268.396979
3      399.316934
4      410.226930
     ...
30     873.901768
31     873.901768
32     939.361745
33    1004.821722
34    1037.551710
Length: 35, dtype: float64
```

.resid attribute

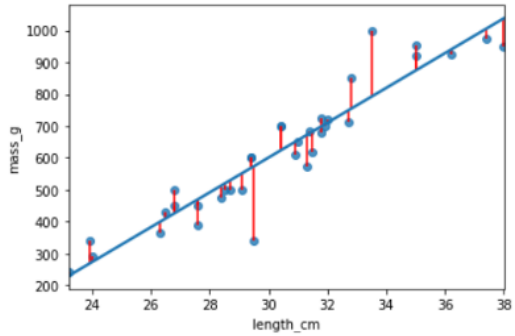
Residuals: actual response values minus predicted response values

```
print(mdl_mass_vs_length.resid)
```

or equivalently

```
print(bream["mass_g"] - mdl_mass_vs_length.fittedvalues)
```

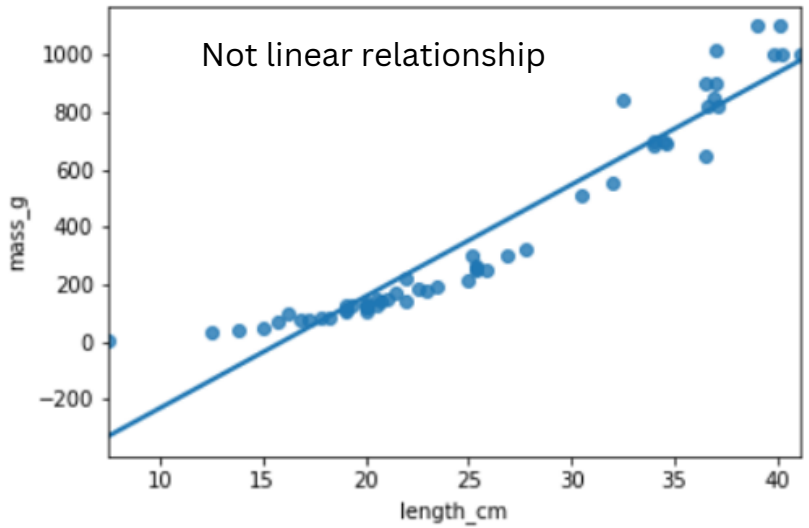
```
0    11.788007
1    16.148023
2    71.603021
3   -36.316934
4    19.773070
   ...
```



"Residuals" are a measure of inaccuracy in the model fit, and are accessed with the .resid attribute. Like fitted values, there is one residual for each row of the dataset. Each residual is the actual response value minus the predicted response value. In this case, the residuals are the masses of breams, minus the fitted values. Here we illustrated the residuals as red lines on the regression plot. Each vertical line represents a single residual.

Transforming variables

- Sometimes, the relationship between the explanatory variable and the response variable may not be a straight line. To fit a linear regression model, you may need to transform the explanatory variable or the response variable, or both of them.

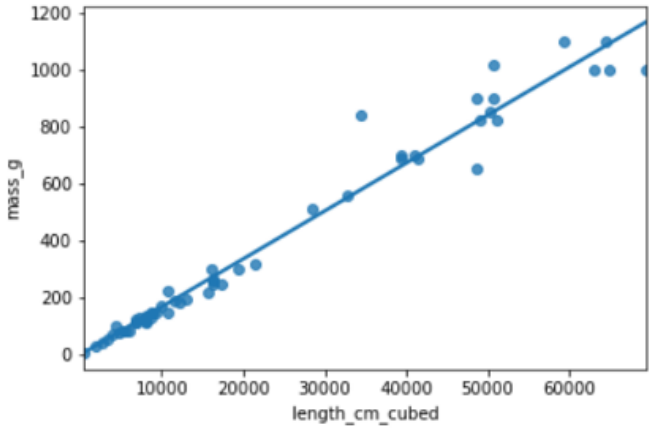


Plotting mass vs. length cubed

```
perch["length_cm_cubed"] = perch["length_cm"] ** 3
```

```
sns.regplot(x="length_cm_cubed",
            y="mass_g",
            data=perch,
            ci=None)
plt.show()
```

cube to make llinear



Modeling mass vs. length cubed

```
perch["length_cm_cubed"] = perch["length_cm"] ** 3

mdl_perch = ols("mass_g ~ length_cm_cubed", data=perch).fit()
mdl_perch.params
```

```
Intercept         -0.117478
length_cm_cubed    0.016796
dtype: float64
```

To model this transformation, we replace the original length variable with the cubed length variable. We then fit the model and extract its coefficients.

Predicting mass vs. length cubed

```
explanatory_data = pd.DataFrame({"length_cm_cubed": np.arange(10, 41, 5) ** 3,
                                 "length_cm": np.arange(10, 41, 5)})
```

```
prediction_data = explanatory_data.assign(
  mass_g=mdl_perch.predict(explanatory_data))
print(prediction_data)
```

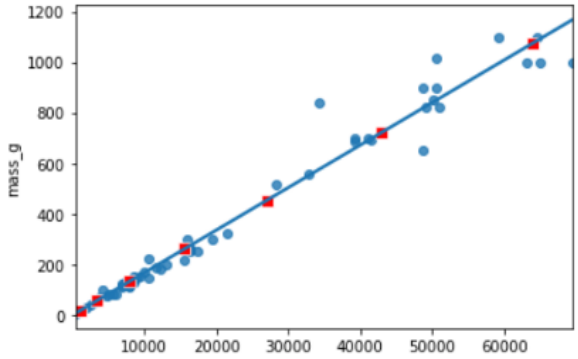
```
   length_cm_cubed  length_cm       mass_g
0             1000         10    16.678135
1             3375         15    56.567717
2             8000         20   134.247429
3            15625         25   262.313982
4            27000         30   453.364084
5            42875         35   719.994447
6            64000         40  1074.801781
```

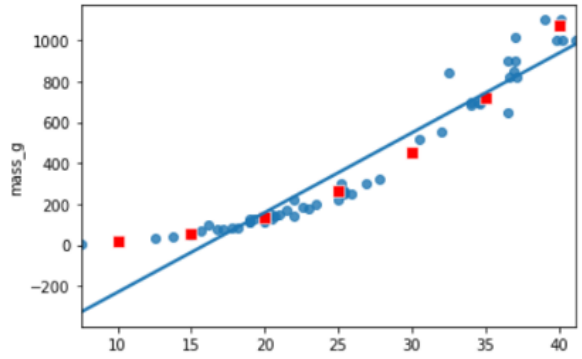
We create the explanatory DataFrame in the same way as usual. Notice that you specify the lengths cubed. We can also add the untransformed lengths column for reference. The code for adding predictions is the same assign and predict combination as you've seen before.

Plotting mass vs. length cubed

```
fig = plt.figure()
sns.regplot(x="length_cm_cubed", y="mass_g",
            data=perch, ci=None)
sns.scatterplot(data=prediction_data,
                x="length_cm_cubed", y="mass_g",
                color="red", marker="s")
```

```
fig = plt.figure()
sns.regplot(x="length_cm", y="mass_g",
            data=perch, ci=None)
sns.scatterplot(data=prediction_data,
                x="length_cm", y="mass_g",
                color="red", marker="s")
```





The predictions have been added to the plot of mass versus length cubed as red points. As you might expect, they follow the line drawn by regplot. It gets more interesting on the original x-axis. Notice how the red points curve upwards to follow the data. Your linear model has non-linear predictions, after the transformation is undone.

Coefficient of Determination

- Sometimes called "r-squared" or "R-squared".
- The proportion of the variance in the response variable that is predictable from the explanatory variable
  - 1 means a perfect
  - 0 means the worst possible

.summary()

Look at the value titled "R-Squared"

```
mdl_bream = ols("mass_g ~ length_cm", data=bream).fit()
```

```
print(mdl_bream.summary())
```

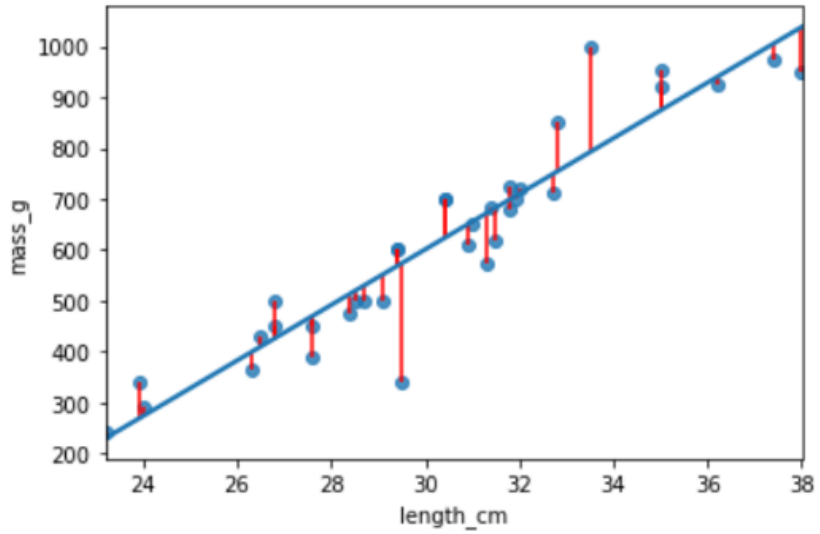
```
# Some lines of output omitted

                            OLS Regression Results
Dep. Variable:                   mass_g   R-squared:                       0.878
Model:                             OLS   Adj. R-squared:                  0.874
Method:                  Least Squares   F-statistic:                     237.6
```

.rsquared attribute

```
print(mdl_bream.rsquared)
```

```
0.8780627095147174
```

Residual standard error (RSE)



- A "typical" difference between a prediction and an observed response
- It has the same unit as the response variable.
- MSE = RSE^2

.mse_resid attribute

```
mse = mdl_bream.mse_resid
print('mse: ', mse)
```

```
mse:  5498.555084973521
```

```
rse = np.sqrt(mse)
print("rse: ", rse)
```

```
rse:  74.15224261594197
```

Interpreting RSE

`mdl_bream` has an RSE of `74`.

> The difference between predicted bream masses and observed bream masses is typically about 74g.

```
summary_fish = fish.groupby("species")["mass_g"].mean()
print(summary_fish)
```

```
species
Bream    617.828571
Perch    382.239286
Pike     718.705882
Roach    152.050000
Name: mass_g, dtype: float64
```

```
# Mean is same the intercept

# If you plus 617.828571 + (-235.589286) = 382.239286
mdl_mass_vs_species = ols("mass_g ~ species", data=fish)

mdl_mass_vs_species = mdl_mass_vs_species.fit()
print(mdl_mass_vs_species.params)
```

```
Intercept           617.828571
species[T.Perch]   -235.589286
species[T.Pike]     100.877311
species[T.Roach]   -465.778571
dtype: float64
```

```
little_bream = pd.DataFrame({"length_cm": [10]})

pred_little_bream = little_bream.assign(
    mass_g=mdl_mass_vs_length.predict(little_bream)
)

print(pred_little_bream)
```

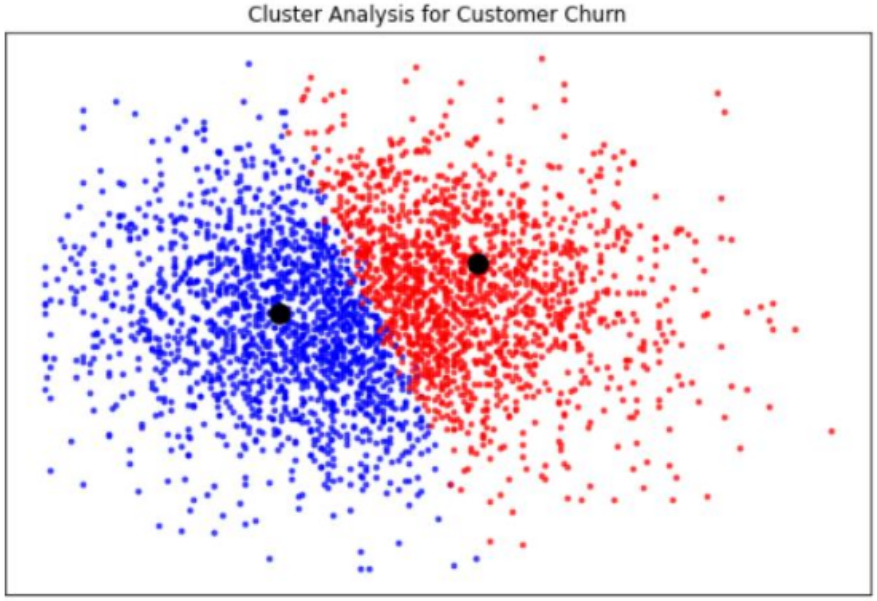
```
   length_cm      mass_g
0         10 -489.847756
```

W9
What is machine learning?

- Machine learning is the process whereby:
  - Computers are given the ability to learn to make decisions from data without being explicitly programmed.

Unsupervised learning

- Uncovering hidden patterns from unlabeled data
- Example:
  - Grouping customers into distinct categories (Clustering)



Supervised learning

- The predicted values are known
- Aim: Predict the target values of unseen data, given the features

	Features					Target variable
	points_per_game	assists_per_game	rebounds_per_game	steals_per_game	blocks_per_game	position
0	26.9	6.6	4.5	1.1	0.4	Point Guard
1	13	1.7	4	0.4	1.3	Center
2	17.6	2.3	7.9	1.00	0.8	Power Forward
3	22.6	4.5	4.4	1.2	0.4	Shooting Guard

Types of supervised learning

- Classification: Target variable consists of categories

- Regression: Target variable is continuous

Naming conventions

- Feature = predictor variable = independent variable
- Target variable = dependent variable = response variable

Before you use supervised learning

- Requirements:
  - No missing values
  - Data in numeric format
  - Data stored in pandas DataFrame or NumPy array
- Perform Exploratory Data Analysis (EDA) first

Classifying labels of unseen data

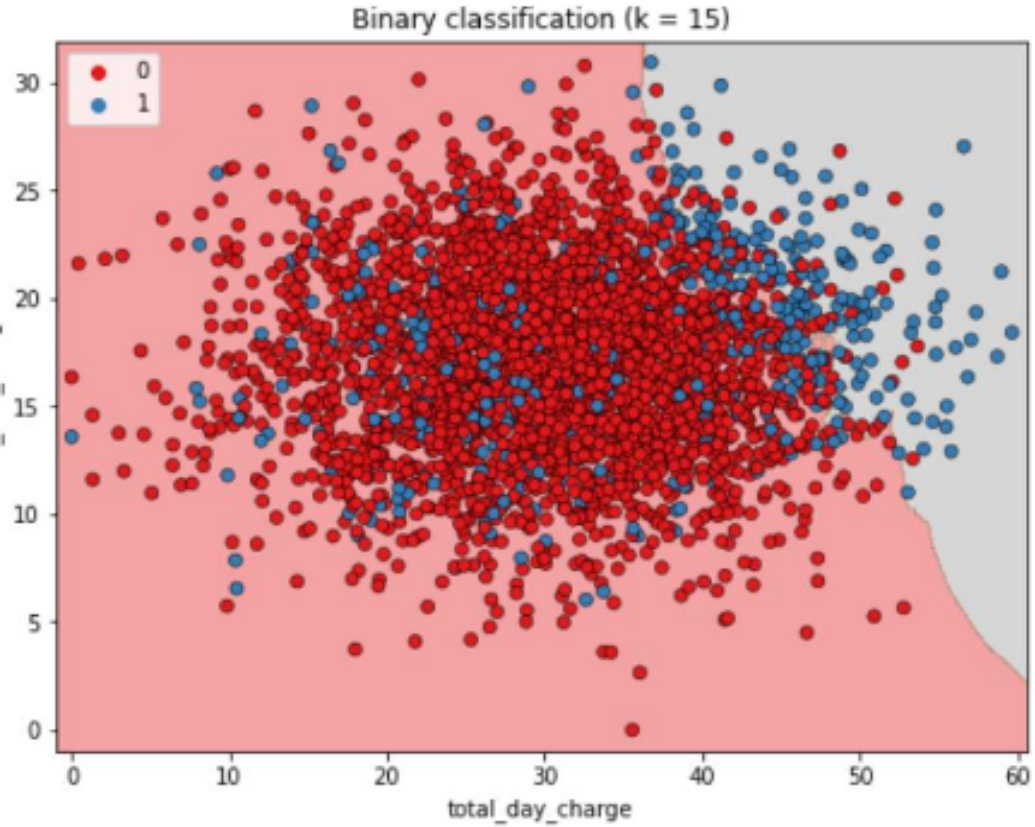
1. Build a model
2. Model learns from the labeled data we pass to it
3. Pass unlabeled data to the model as input
4. Model predicts the labels of the unseen data

- Labeled data = training data

k-Nearest Neighbors

- Predict the label of a data point by
  - Looking at the `k` closest labeled data points
  - Taking a majority vote

KNeighborsClassifier uses alogorithm to set boundary to predict



gray background are predicted to churn

red background are predicted to not churn

This boundary would be used to make predictions on unseen data.

Predicting on unlabeled data

```
X_new = np.array([[56.8, 17.5],
                  [24.4, 24.1],
                  [50.1, 10.9]])
print(X_new.shape)
```

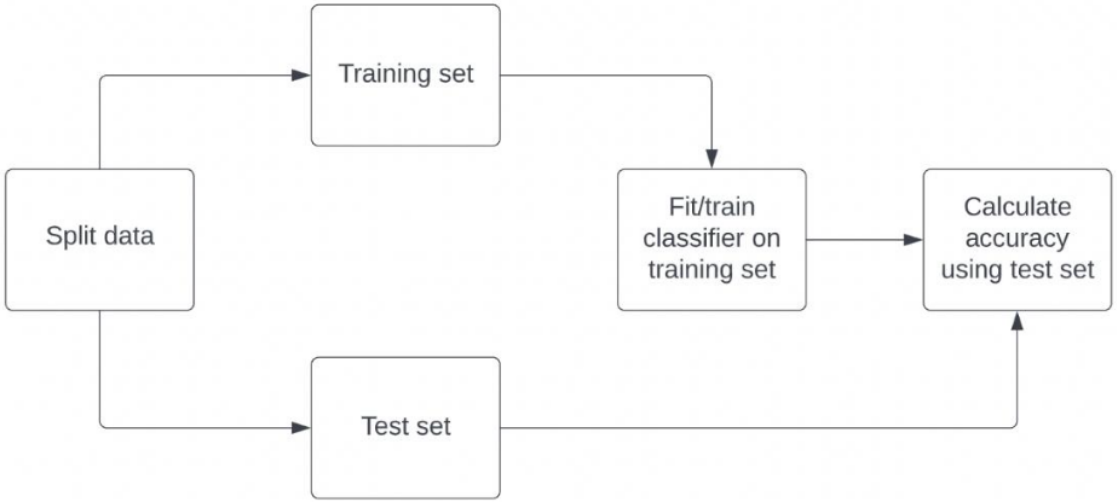
```
(3, 2)
```

```
predictions = knn.predict(X_new)
print('Predictions: {}'.format(predictions))
```

```
Predictions: [1 0 0]
```

1 corresponds to churn and 0 corresponds to no churn

Computing accuracy

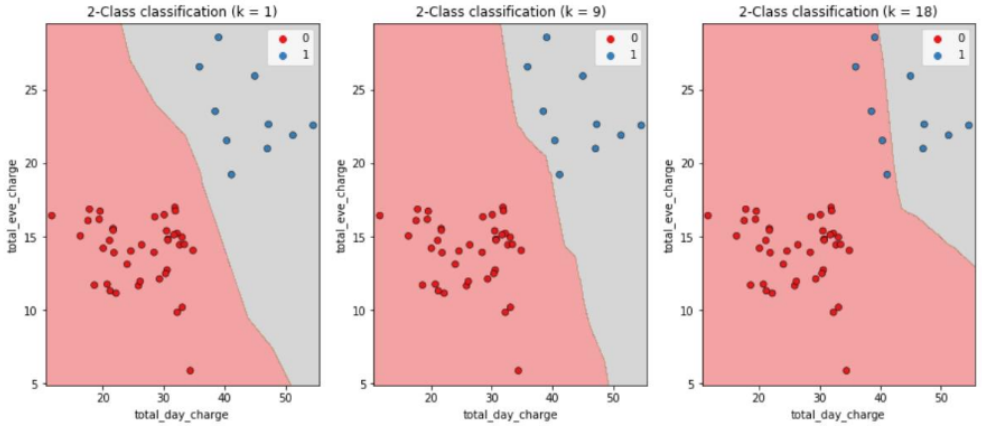



```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.3,
                                                    random_state=21,
                                                    stratify=y)

knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(X_train,y_train)
print(knn.score(X_test, y_test))
```

It is best practice to ensure our split reflects the proportion of labels in our data. So if churn occurs in 10% of observations, we want 10% of labels in our training and test sets to represent churn. We achieve this by setting stratify equal to y.

Model complexity

- Larger k = less complex model = can cause underfitting
- Smaller k = more complex model = can lead to overfitting



Recall that we discussed decision boundaries, which are thresholds for determining what label a model assigns to an observation. In the image shown, as k increases, the decision boundary is less affected by individual observations, reflecting a simpler model. Simpler models are less able to detect relationships in the dataset, which is known as underfitting. In contrast, complex models can be sensitive to noise in the training data, rather than reflecting general trends. This is known as overfitting.

Creating feature and target arrays

```
X = diabetes_df.drop("glucose", axis=1).values
y = diabetes_df["glucose"].values
print(type(X), type(y))
```

```
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
```

Making predictions from a single feature

```
X_bmi = X[:, 3]
print(y.shape, X_bmi.shape)
```

```
(752,) (752,)
```

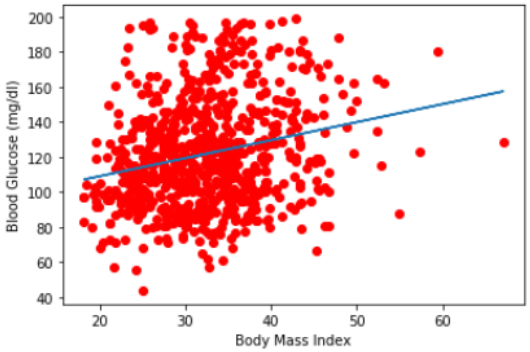
```
X_bmi = X_bmi.reshape(-1, 1)
print(X_bmi.shape)
```

```
(752, 1)
```

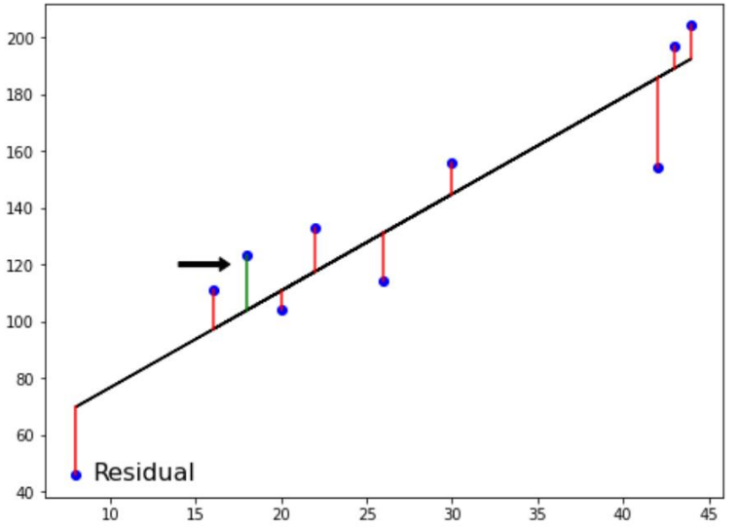
To start, let's try to predict blood glucose levels from a single feature: body mass index. To do this, we slice out the BMI column of X, which is the fourth column, storing as the variable X_bmi. Checking the shape of y and X_bmi, we see that they are both one-dimensional arrays. This is fine for y, but our features must be formatted as a two-dimensional array to be accepted by scikit-learn. To convert the shape of X_bmi we apply NumPy's .reshape method, passing minus one followed by one. Printing the shape again shows X_bmi is now the correct shape for our model.

Fitting a regression model

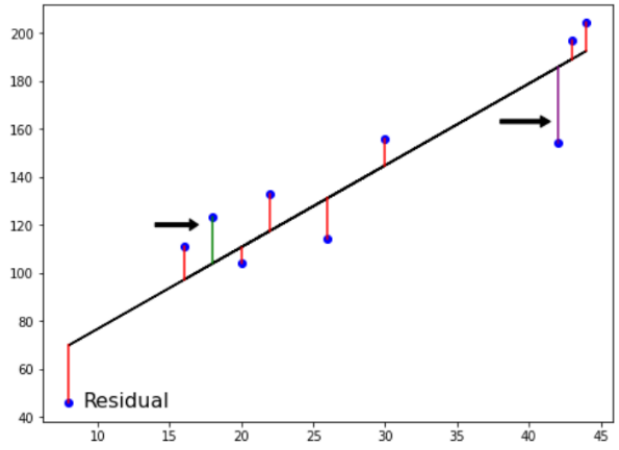
```
from sklearn.linear_model import LinearRegression
reg = LinearRegression()
reg.fit(X_bmi, y)
predictions = reg.predict(X_bmi)
plt.scatter(X_bmi, y)
plt.plot(X_bmi, predictions)
plt.ylabel("Blood Glucose (mg/dl)")
plt.xlabel("Body Mass Index")
plt.show()
```



The loss function



Ordinary Least Squares



$$RSS = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Ordinary Least Squares (OLS): minimize RSS

The distance is called a residual. We could try to minimize the sum of the residuals, each negative residual. To avoid this, we square the residuals. By adding all the squared residuals, we calculate the residual sum of squares, or RSS. This type of linear regression is called Ordinary Least Squares, or OLS, where we aim to minimize the RSS.

Measuring model performance

- In classification, accuracy is a commonly used metric
- Accuracy:

$$\frac{correct\ predictions}{total\ observations}$$

- How do we measure accuracy?
- Could compute accuracy on the data used to fit the classifier
- NOT indicative of ability to generalize

Linear regression in higher dimensions

$$y = a_1 x_1 + a_2 x_2 + b$$

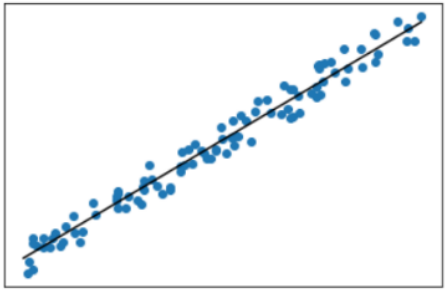
- To fit a linear regression model here:
  - Need to specify 3 variables: $a_1,\ a_2,\ b$
- In higher dimensions:
  - Known as multiple regression
  - Must specify coefficients for each feature and the variable b

$$y = a_1 x_1 + a_2 x_2 + a_3 x_3 + \ldots + a_n x_n + b$$

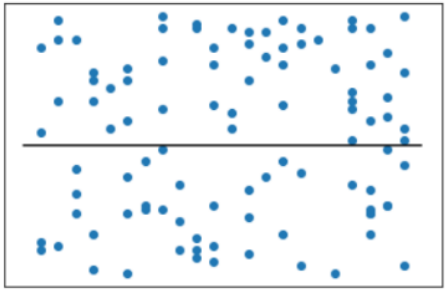
- scikit-learn works exactly the same way:
  - Pass two arrays: features and target

R-squared

- R^2: quantifies the variance in target values explained by the features
  - Values range from 0 to 1
- High R^2:



- Low R^2:



The coefficient of determination, or R2 , is a measure that provides information about the goodness of fit of a model.

R-squared in scikit-learn

```
reg_all.score(X_test, y_test)
```

```
0.356302876407827
```

To compute R-squared, we call the model's .score() method, passing the test features and targets. Here the features only explain about 35 percent of blood glucose level variance.

RMSE in scikit-learn

```
from sklearn.metrics import mean_squared_error
mean_squared_error(y_test, y_pred, squared=False)
```

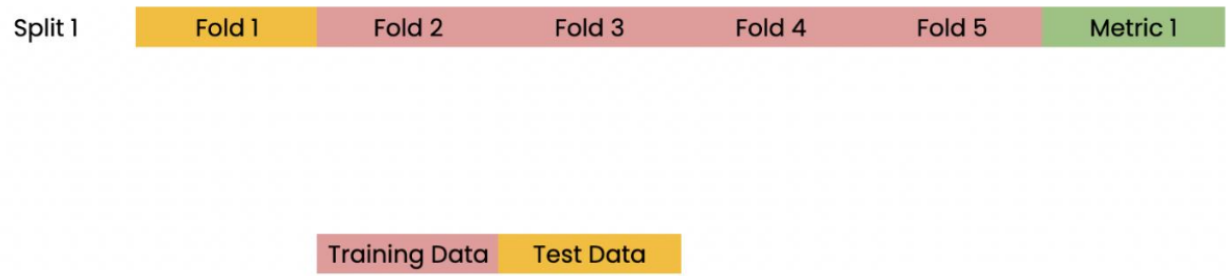
```
24.028109426907236
```

The model has an average error for blood glucose levels of around 24 milligrams per deciliter.

Cross-validation motivation

- Model performance is dependent on the way we split up the data
- Not representative of the model's ability to generalize to unseen data
- Solution: Cross-validation!

Cross-validation basics



We begin by splitting the dataset into five groups or folds. Then we set aside the first fold as a test set, fit our model on the remaining four folds, predict on our test set, and compute the metric of interest, such as R-squared.

Cross-validation and model performance

- 5 folds = 5-fold CV
- 10 folds = 10-fold CV
- k folds = k-fold CV
- More folds = More computationally expensive

Class imbalance is a common problem in machine learning that occurs when the distribution of examples within a dataset is skewed or biased. This can lead to a bias in the trained model, which can negatively impact its performance.

Cross-validation in scikit-learn

```
from sklearn.model_selection import cross_val_score, KFold
kf = KFold(n_splits=6, shuffle=True, random_state=42)
reg = LinearRegression()
cv_results = cross_val_score(reg, X, y, cv=kf)
```

Evaluating cross-validation peformance

```
print(cv_results)
```

```
[0.70262578, 0.7659624, 0.75188205, 0.76914482, 0.72551151, 0.73608277]
```

```
print(np.mean(cv_results), np.std(cv_results))
```

```
0.7418682216666667 0.023330243960652888
```

Why regularize?

- Recall: Linear regression minimizes a loss function
- It chooses a coefficient, a, for each feature variable, plus b
- Large coefficients can lead to overfitting
- Regularization: Penalize large coefficients

Lasso regression

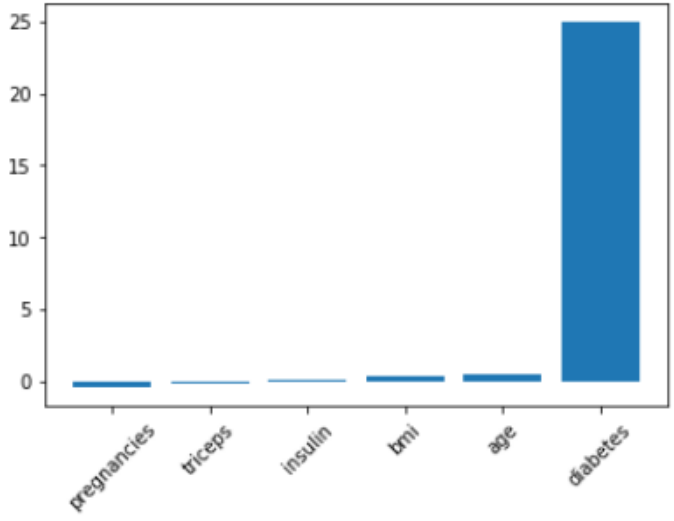
- Loss function = OLS loss function +

$$\alpha * \sum_{i=1}^{n} |a_i|$$

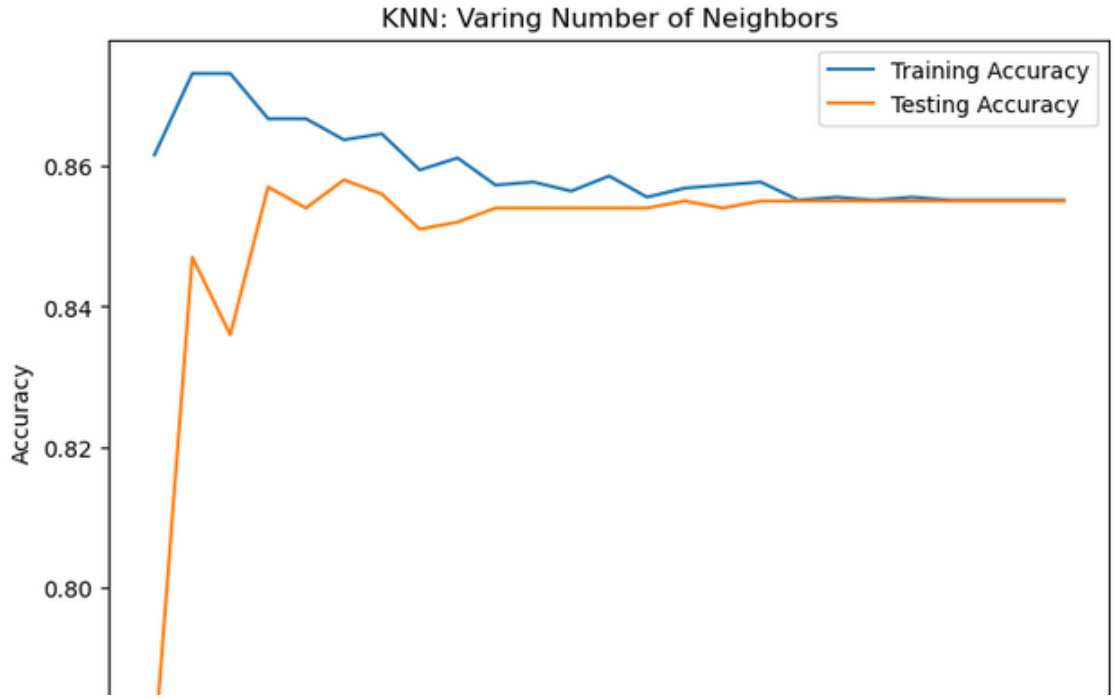
- α: parameter we need to choose
- Picking α is similar to picking `k` in KNN
- Hyperparameter: variable used to optimize model parameters
- α controls model complexity
  - α = 0 = OLS (Can lead to overfitting)
  - Very high α: Can lead to underfitting

Lasso regression for feature selection

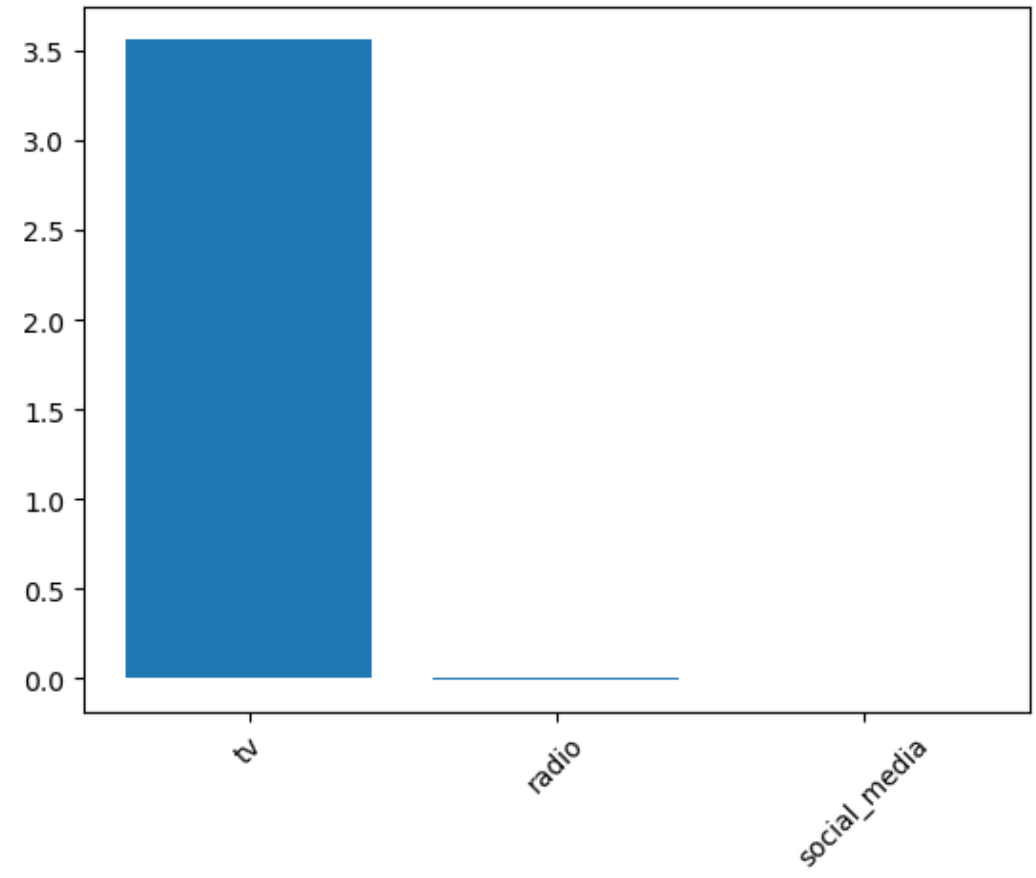
- Lasso can select important features of a dataset
- Shrinks the coefficients of less important features to zero
- Features not shrunk to zero are selected by lasso



```python
train_accuracies = {}
test_accuracies = {}
neighbors = np.arange(1, 26)

for neighbor in neighbors:
    knn = KNeighborsClassifier(n_neighbors=neighbor)
    knn.fit(x_train, y_train)
    train_accuracies[neighbor] = knn.score(x_train, y_train)
    test_accuracies[neighbor] = knn.score(x_test, y_test)

my_train = list(train_accuracies.values())
my_test = list(test_accuracies.values())

plt.figure(figsize=(8, 6))

plt.title("KNN: Varing Number of Neighbors")
plt.plot(neighbors, my_train, label="Training Accuracy")
plt.plot(neighbors, my_test, label="Testing Accuracy")

plt.legend()
plt.xlabel("Number of Neighbors")
plt.ylabel("Accuracy")
plt.show()

# Around k=6 is good
```




```python
sales_df = pd.read_csv("data_sets/sales_df.csv", index_col=0)

X = sales_df.drop("sales", axis=1).values
y = sales_df["sales"].values
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

names = sales_df.drop("sales", axis=1).columns
lasso = Lasso(alpha=0.1)
lasso_coef = lasso.fit(X, y).coef_
plt.bar(names, lasso_coef)
plt.xticks(rotation=45)
plt.show()
```



W10

How good is your model?

- Classification metrics
  - Measuring model performance with accuracy:
    - Fraction of correctly classified samples
    - Not always a useful metric

Class imbalance

- Classification for predicting fraudulent bank transactions
  - 99% of transactions are legitimate; 1% are fraudulent
- Could build a classifier that predicts NONE of the transactions are fraudulent
  - 99% accurate!
  - But terrible at actually predicting fraudulent transactions
  - Fails at its original purpose
- Class imbalance: Uneven frequency of classes
- Need a different way to assess performance

Confusion matrix for assessing classification performance

- Confusion matrix

	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

The false negatives are the number of legitimate transactions incorrectly labeled

The false positives are the number of transactions incorrectly labeled as fraudulent.

The true negatives are the number of legitimate transactions correctly labeled

The true positives are the number of fraudulent transactions correctly labeled

Assessing classification performance

	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

- Accuracy:

$$\frac{tp + tn}{tp + tn + fp + fn}$$

Firstly, we can retrieve accuracy: it's the sum of true predictions divided by the total sum of the matrix.

Precision

	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

- Precision

$$\frac{true\ positives}{true\ positives + false\ positives}$$

- High precision = lower false positive rate
- High precision: Not many legitimate transactions are predicted to be fraudulent

Usually, the class of interest is called the positive class. As we aim to detect fraud, the positive class is an illegitimate transaction. So why is the confusion matrix important? There are other important metrics we can calculate from the confusion matrix. Precision is the number of true positives divided by the sum of all positive predictions. It is also called the positive predictive value. In our case, this is the number of correctly labeled fraudulent transactions divided by the total number of transactions classified as fraudulent. High precision means having a lower false positive rate.

F1 score

- F1 Score: $2 * \frac{precision * recall}{precision + recall}$

The F1-score is the harmonic mean of precision and recall. This metric gives equal weight to precision and recall, therefore it factors in both the number of errors made by the model and the type of errors. The F1 score favors models with similar precision and recall, and is a useful metric if we are seeking a model which performs reasonably well across both metrics.

Recall

	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

- Recall

$$\frac{true\ positives}{true\ positives + false\ negatives}$$

- High recall = lower false negative rate
- High recall: Predicted most fraudulent transactions correctly

Recall is the number of true positives divided by the sum of true positives and false negatives. This is also called sensitivity. High recall reflects a lower false negative rate. For our classifier, it means predicting most fraudulent transactions correctly.

Confusion matrix in scikit-learn

```
from sklearn.metrics import classification_report, confusion_matrix
knn = KNeighborsClassifier(n_neighbors=7)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,
                                                    random_state=42)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
```

```
print(confusion_matrix(y_test, y_pred))
```

```
[[1106   11]
 [ 183   34]]
```

True Negative	False Positive
False Negative	True Positive

Classification report in scikit-learn

```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.86      0.99      0.92      1117
           1       0.76      0.16      0.26       217

    accuracy                           0.85      1334
   macro avg       0.81      0.57      0.59      1334
weighted avg       0.84      0.85      0.81      1334
```

This report includes precision and recall by class (churn: 1 or no churn: 0), point-seven-six and point-one-six for the churn class respectively, which highlights how poorly the model's recall is on the churn class. Support represents the number of instances for each class within the true labels. (Class imbalance)

Hyperparameter tuning

- Lasso regression: choosing alpha
- KNN: choosing n_neighbors
- Hyperparameters: parameters we specify before fitting the model

Choosing the correct hyperparameters

1. Try lots of different hyperparameter values
2. Fit all of them separately
3. See how well they perform
4. Choose the best performing values

- This is called **hyperparameter tuning**
- It is essential to use cross-validation to avoid overfitting to the test set
- We can still split the data and perform cross-validation on the training set
- We withhold the test set for final evaluation

Grid search cross-validation

	11		
	8		
	5		
n_neighbors	2		
		euclidean	manhattan
		metric	

We perform k-fold cross-validation for each combination of hyperparameters. The mean scores for each combination are shown here. By default, KNN uses Euclidean distance but Manhattan distance can be selected by setting p = 1.

For example, `.KNeighborsClassifier`(*n_neighbors=5*, *p=1*)

Grid search cross-validation

	11	0.8716	0.8692
	8	0.8704	0.8688
	5	0.8748	0.8714
n_neighbors	2	0.8634	0.8646
		euclidean	manhattan
		metric	

Imputing values

- Imputation - use subject-matter expertise to replace missing data with educated guesses
- Common to use the mean
- Can also use the median, or another value
- For categorical values, we typically use the most frequent value - the mode
- Must split our data first, to avoid *data leakage*

scikit-learn requirements

- Numeric data
- No missing values

- With real-world data:
  - This is rarely the case
  - We will often need to preprocess our data first

Dealing with categorical features

- scikit-learn will not accept categorical features by default
- Need to convert categorical features into numeric values
- Convert to binary features called dummy variables
  - 0: Observation was NOT that category
  - 1: Observation was that category

The Mean Squared Error measures how close a regression line is to a set of data points. It is a risk function corresponding to the expected value of the squared error loss. Mean square error is calculated by taking the average, specifically the mean, of errors squared from data as it relates to a function.

Dummy variables

genre
Alternative
Anime
Blues
Classical
Country
Electronic
Hip-Hop
Jazz
Rap
Rock

Alternative	Anime	Blues	Classical	Country	Electronic	Hip-Hop	Jazz	Rap	Rock
1	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	1

- pandas: `get_dummies()`