Lab 1

<u>See current working directory and its files:</u>
os.getcwd()
os.listdir()

Basic Pandas operations cfr. ‘cheat sheet’.

Lab 2

Univariate analysis

Nominal categories have NO inherent order, eg. eye color. Binary classifications are also nominal.

Ordinal values are categories with inherent order eg. education level, Likert scale.

<u>Analyse numeric columns:</u>
numeric_columns.describe() OR you can also just calculate one of the values returned by the .describe() method

<u>Use NumPy to analyse data:</u>
list = [np.mean, ,np.max, np.median ...]

OR just do

list = [“mean”, “max”, “median]

numeric_values.aggregate(list)

<u>Analyse categorical columns:</u>
single_categorical_column.value_counts()
→ get count per category

single_categorical_column.mode()
→ get most frequent value

Bivariate analysis

categorical + continuous	group by aggregation function
categorical + categorical	frequency table
numeric + numeric	correlation

Correlation:

dataframe[[“value1”, “value2”]].corr()

→ you can add as many numeric values as you wish, it will just enlarge the correlation matrix

Group by aggregation:

dataframe[[“numeric1”, “numeric2”, category1”].groupby[“category1”].aggregate()

→ aggregate functions cfr. supra

Frequency table:

pd.crosstab(dataframe[“category1”], dataframe[“category2”])

Lab 3

Format data to read and explore

pd.options.display.float_format = '{:.2f}'.format

→ this gives no loss of precision of underlying data

Data exploration with Pandas

Large difference between mean and median? Think about outliers!

Cleaning data

Make use of boolean indexing to filter out rows.

boolean_mask = dataframe[‘column’] == ‘value’

dataframe.loc[boolean_mask]

→ all rows with true on mask are kept, others are discarded

DataFrames and Series have indices!

dataframe.index

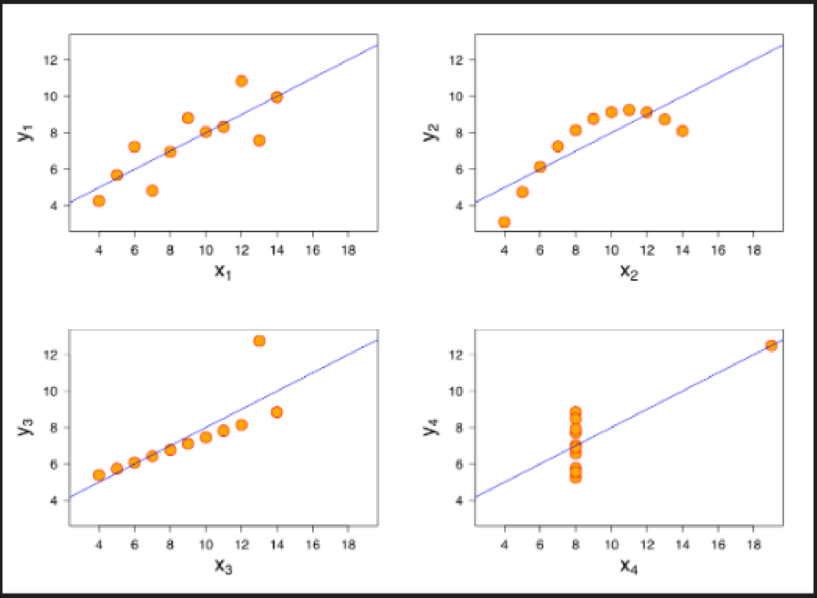
Always copy your data before you do any changes.

Use .loc to filter on multiple elements:

dataframe.loc[mask, [“column1”, “column2”]]

Low correlation values don’t always mean no relationship. Because the latter can be non-linear! Thus, always visualise data!

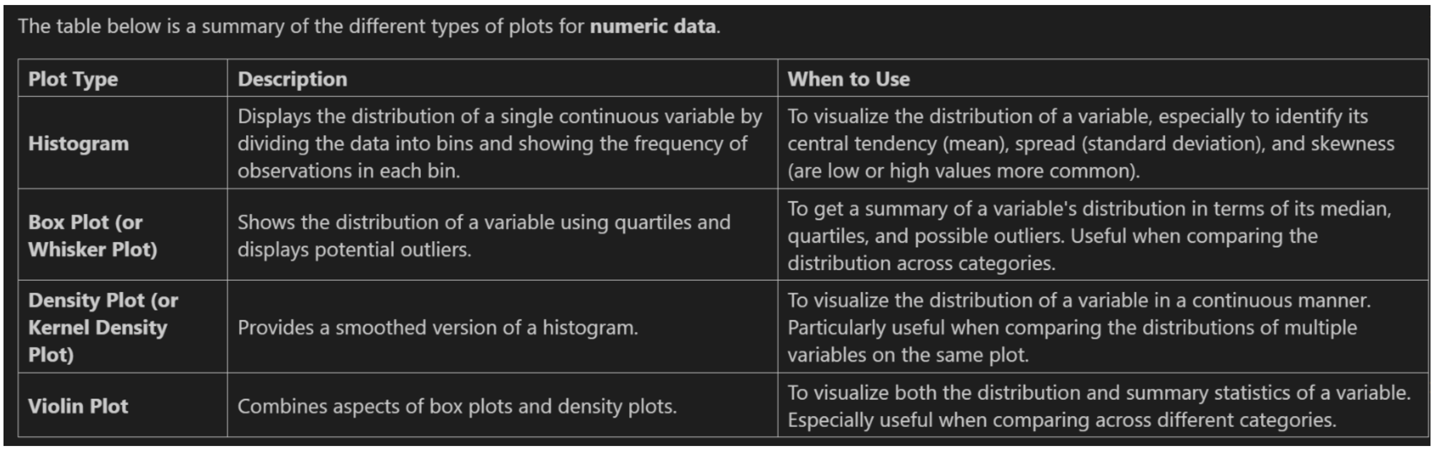
Anscomnbe's quarter: four datasets with an equal correlation.



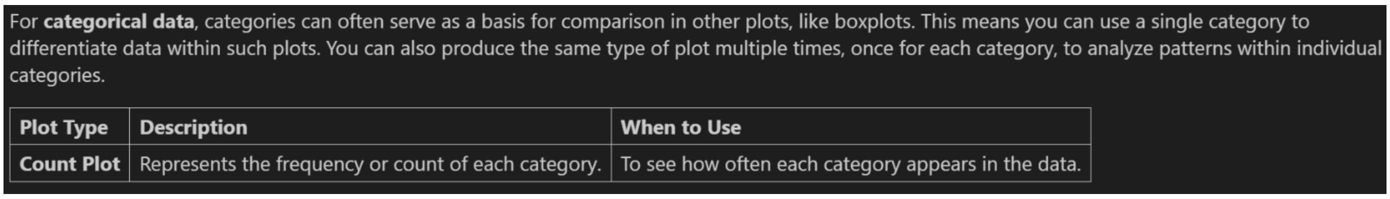
Visualise data

Use Matplotlib!

The syntax for plotting is generally `plt.<plotType>(x, y)`.


The table below is a summary of the different types of plots for **numeric data**.

Plot Type	Description	When to Use
Histogram	Displays the distribution of a single continuous variable by dividing the data into bins and showing the frequency of observations in each bin.	To visualize the distribution of a variable, especially to identify its central tendency (mean), spread (standard deviation), and skewness (are low or high values more common).
Box Plot (or Whisker Plot)	Shows the distribution of a variable using quartiles and displays potential outliers.	To get a summary of a variable's distribution in terms of its median, quartiles, and possible outliers. Useful when comparing the distribution across categories.
Density Plot (or Kernel Density Plot)	Provides a smoothed version of a histogram.	To visualize the distribution of a variable in a continuous manner. Particularly useful when comparing the distributions of multiple variables on the same plot.
Violin Plot	Combines aspects of box plots and density plots.	To visualize both the distribution and summary statistics of a variable. Especially useful when comparing across different categories.



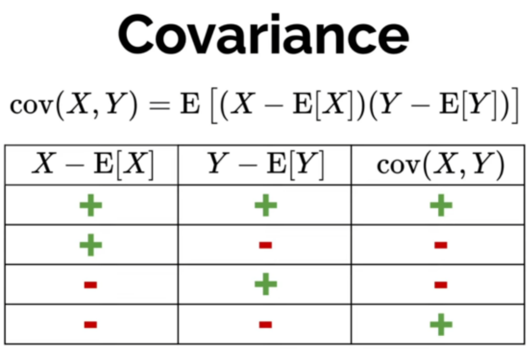
For **categorical data**, categories can often serve as a basis for comparison in other plots, like boxplots. This means you can use a single category to differentiate data within such plots. You can also produce the same type of plot multiple times, once for each category, to analyze patterns within individual categories.

Plot Type	Description	When to Use
Count Plot	Represents the frequency or count of each category.	To see how often each category appears in the data.

Lab 4

Covariance and correlation

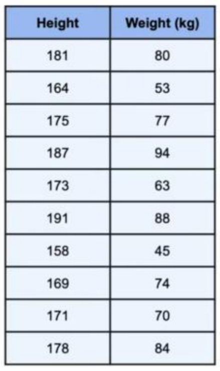
Covariance indicate whether two variables relate to each other (direction of change), but does not show the strength of its relationship.



Covariance

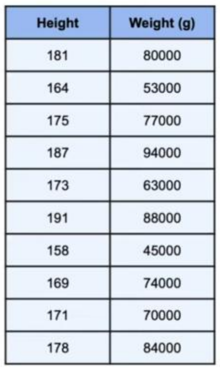
$$\mathrm{cov}(X,Y) = \mathrm{E}\left[(X-\mathrm{E}[X])(Y-\mathrm{E}[Y])\right]$$

$X-\mathrm{E}[X]$	$Y-\mathrm{E}[Y]$	$\mathrm{cov}(X,Y)$
+	+	+
+	-	-
-	+	-
-	-	+



Height	Weight (kg)
181	80
164	53
175	77
187	94
173	63
191	88
158	45
169	74
171	70
178	84

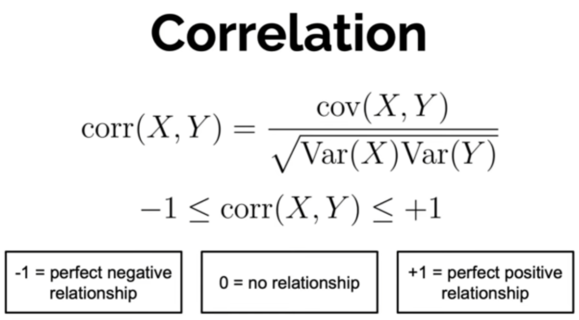
Covariance: 143



Height	Weight (g)
181	80000
164	53000
175	77000
187	94000
173	63000
191	88000
158	45000
169	74000
171	70000
178	84000

Covariance: 143155

Correlation can indicate the strength of the relationship, because it is standardized by division by its standard deviations!

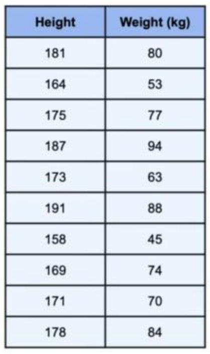


Correlation

$$\mathrm{corr}(X,Y) = \frac{\mathrm{cov}(X,Y)}{\sqrt{\mathrm{Var}(X)\mathrm{Var}(Y)}}$$

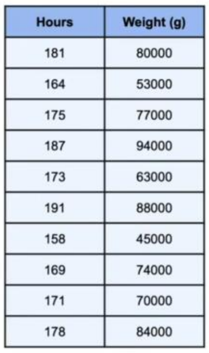
$$-1 \le \mathrm{corr}(X,Y) \le +1$$

-1 = perfect negative relationship
0 = no relationship
+1 = perfect positive relationship



Height	Weight (kg)
181	80
164	53
175	77
187	94
173	63
191	88
158	45
169	74
171	70
178	84

Correlation: 0.922



Hours	Weight (g)
181	80000
164	53000
175	77000
187	94000
173	63000
191	88000
158	45000
169	74000
171	70000
178	84000

Correlation: 0.922

No effect of different scale anymore...

Always remember that relationship does not always mean causality.

Slope of regression:

$$\beta_y = r_{x,y}\frac{s_x}{s_y}$$

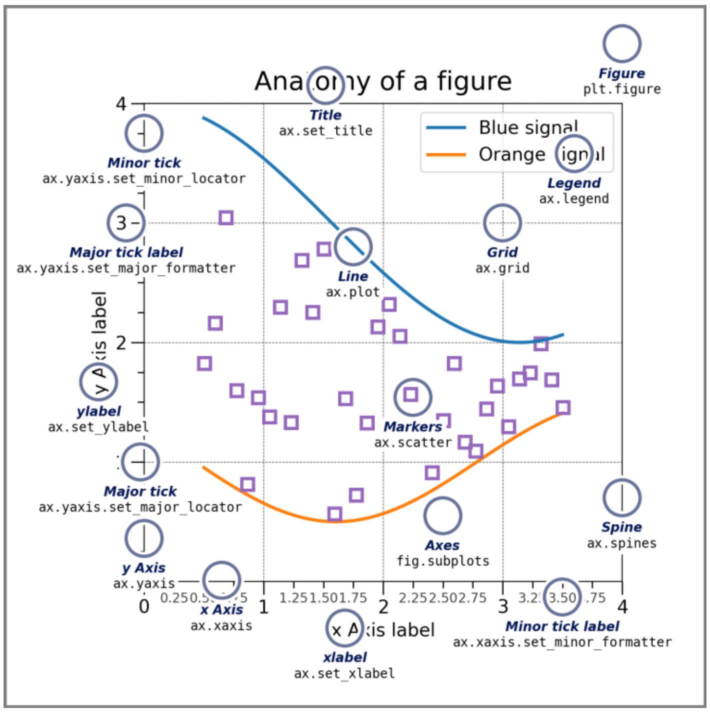
r = correlation, s = standard deviation

Plotting univariate data with Matplotlib

Use a semicolon after a line of code which plots data to get a clean output of the graph while suppressing other (unnecessary) output.

IQR = Q3 – Q1

Outliers lay outside of Q1 – 1.5*IQR or Q3 + 1.5*IQR.



We will plot graphs by defining a Figure on which Axes are plotted.

fig, ax = plt.subplots()
ax.plot()

OR

multiple axes:

fig, axes = plt.subplots(rows=1,columns=2)

axes[0].plot()

axes[1].plot()

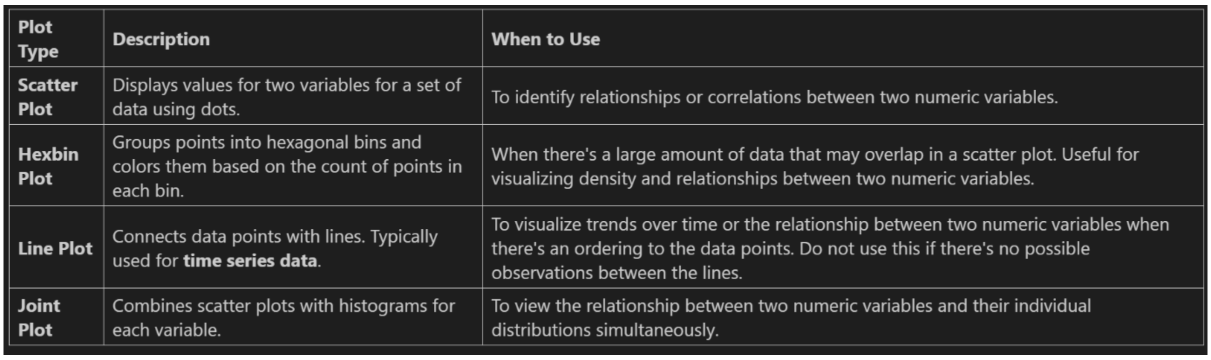
Two ways of plotting with example:

1. ax.barplot(dataset)
   ax.set_title(“Title”)
   ax.set_ylabel(“Y-label”)

2. dataset.plot(kind=’bar’, ax=ax, title=”Title”, ylabel=”Y-label”)

Plotting bivariate data with Seaborn

NUMERIC



Plot Type	Description	When to Use
Scatter Plot	Displays values for two variables for a set of data using dots.	To identify relationships or correlations between two numeric variables.
Hexbin Plot	Groups points into hexagonal bins and colors them based on the count of points in each bin.	When there's a large amount of data that may overlap in a scatter plot. Useful for visualizing density and relationships between two numeric variables.
Line Plot	Connects data points with lines. Typically used for **time series data**.	To visualize trends over time or the relationship between two numeric variables when there's an ordering to the data points. Do not use this if there's no possible observations between the lines.
Joint Plot	Combines scatter plots with histograms for each variable.	To view the relationship between two numeric variables and their individual distributions simultaneously.

fig, ax = plt.subplots()

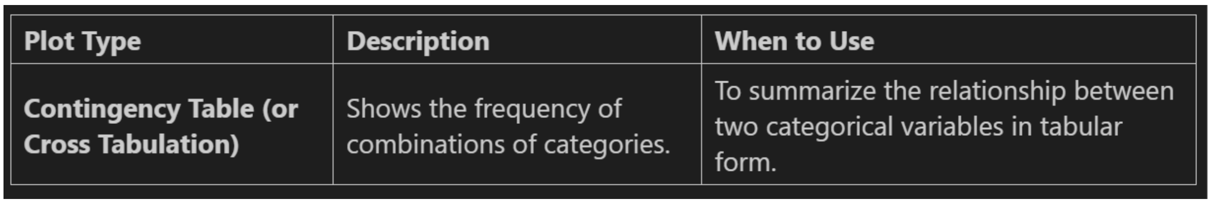
sns.plot(dataset, x=’variable1’, y=’variable2’, ax=ax)

OR

g = sns.jointplot(dataset, x=’variable1’, y=’variable2’, ax=ax, kind=’’)

g.ax_joint.yaxis.set_major_locator(ticker.MultipleLocator(5000))

CATEGORICAL



Plot Type	Description	When to Use
Contingency Table (or Cross Tabulation)	Shows the frequency of combinations of categories.	To summarize the relationship between two categorical variables in tabular form.

fig, ax = plt.subplots()
sns.countplot(dataset, x=”category”, ax=ax)

Small multiples

Different graph for each category, but same scale: allows to compare certain variable across categories.

g = sns.displot(dataset, x=”variable”, col=”category_type”)

OR

fig, ax = plt.subplots()
sns.boxplot(dataset, x=”category_type”, y=”variable”, ax=ax)

Colour-coding

All data on one graph. In contrary, small multiples use different graphs and thus no different colours are needed.

fig, ax = plt.subplots()
sns.histplot(dataset, x=”variable”, hue=”category”, ax=ax)

- ➔ the y axis will automatically be the count per category cfr. each histogram (univariate analysis)
- ➔ hue = ‘shade of colour’

Tip: you can use a heatmap to visualize a crosstab!

Multivariate analysis

Combination of small multiples and colour coding!

- ➔ Multiple data dimensions at once

fig, ax = plt.subplots()
sns.scatterplot(dataset, x=’variable1’, y=’variable2’, hue=’category’, ax=ax)

g = sns.relplot(dataset, x=’variable1’, y=’variable2’, col=’category1’, hue=’category2’, col_wrap=4);

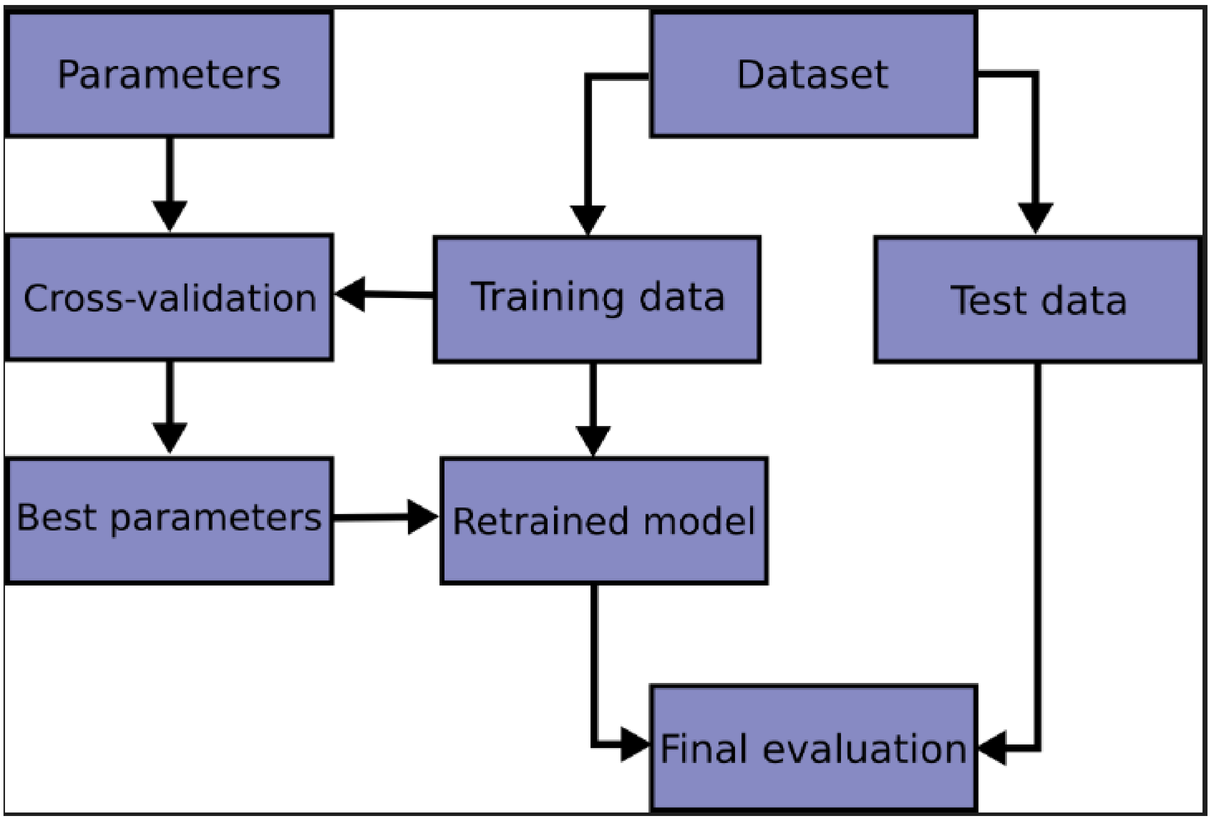
Tip: use col_wrap to define max amount of graphs next to each other.

Lab 5

Machine learning with Sci-kit Learn

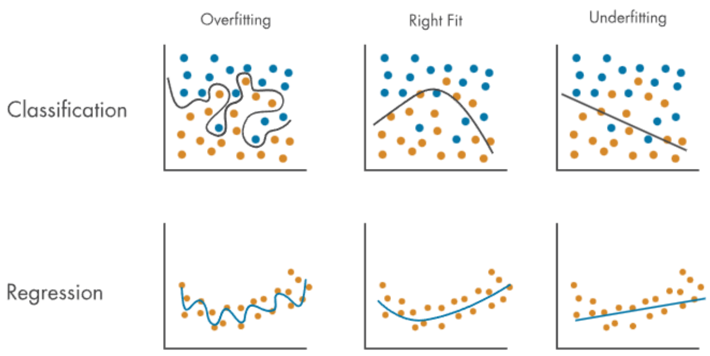
Sci-kit Learn uses Pandas and NumPy.

Generating ML models follows a certain flowchart:



Split the data

Be sure to test your model on the test data group to evaluate a model that generalizes well without missing the mark (underfitting) or is to focussed on noise (overfitting).



Define X containing all independent variables while y contains the dependent variable.

from sklearn.model_selection import train_test_split

X = dataset.drop(columns='dependent_var')
y = dataset['dependent_var']

X_train, X_test, y_train, y_test = train_test_split(X, y, trainsize=0.8, random_state=42)

Feature selection

Visualisation of features can be interactively done with Plotly Express.

px.imshow → heatmap
px.histogram → aggregated bar chart if multiple categories, histogram if Series
px.scatter → scatterplot

PARAMETERS

data_frame=dataset

x='variable1'
y='variable2'

labels={'x':'Variable 1', 'y':'Variable 2'}

title='Title'

color='category1'
symbol='category2'

log_x OR log_y=True/False to have log scales
marginal_x OR marginal_y='histogram' to add histogram in margins

Preprocessing: standard scaling and one-hot-encoding

In sci-kit learn you need to fit models and objects such as a one hot encoder and a standard scaler. Then they're ready to be used.

Transforming is applying a preprocessing transformation (typically after fitting it first).

Logically, when applying one hot encoder; you will have a lot of columns extra!

One hot encoders preprocesses categorical features, standard scalers do the same for numeric features.

from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import StandardScaler

```
ohe = OneHotEncoder()
ohe.fit(X_train[cat_columns])
cat_cols_train = X_train.transform(X_train[cat_columns])


scaler = StandardScaler()
scaler.fit(X_train[num_columns])
num_cols_train = X_train.transform(X_train[num_columns])


X_train_preprocessed = np.hstack((cat_cols_train, num_cols_train))
```

Remember to ONLY use the training data to fit the one hot encoder and the standard scaler. DO NOT fit these objects with the test set…

Machine learning models

```
from sklearn.linear_model import LinearRegression
```

```
lin_reg = LinearRegression()
lin_reg.fit(X_train_preprocessed, y_train)
predictions_lr = lin_reg.predict(X_train_preprocessed)
```

Lab 6

Pipeline

This avoids making errors by using a standardized approach to generate machine learning models. E.g. wrongfully fit model with test data.

```
from sklearn.compose import make_column_transformer
from sklearn.pipeline import make_pipeline
```

```
numeric_columns = [‘var1’, ‘var2’, …]
cat_columns = [‘var3’, ‘var4’, …]
```

- ➔ Define the features by data type

```
preprocessing = make_column_transformer(
        (StandardScaler(), numeric_columns),
        (OneHotEncoder(), cat_columns),
        remainder="drop"
```

)
→ choose transformation object for each data type

preprocessing.fit_transform(X_train)
→ transformation of data

lin_reg_pipe = make_pipeline(preprocessing, LinearRegression())
→ pipeline selects preprocessing object and type of model

lin_reg_pipe.fit(X_train, y_train)
predictions_lin_reg_train = lin_reg_pipe.predict(X_train)
predictions_lin_reg_test = lin_reg_pipe.predict(X_test)
→ fit pipeline will preprocess data and fit model in one go

How to train 3 models in one loop:

```
model_name_pair = [("random_forest", RandomForestRegressor()),
("gradient boosting", HistGradientBoostingRegressor()), ("decision
tree", DecisionTreeRegressor())]
results = []
for pair in model_name_pair:
    name, model = pair
    pipe = make_pipeline(preprocessing, model)
    pipe.fit(X_train, y_train)
    predictions_train = pipe.predict(X_train)
    predictions_test = pipe.predict(X_test)
    result.append([(name, predictions_train, predictions_test)])
```

Model evaluation

errors = prediction_test – y_test

MAE (mean absolute error) vs. MSE (mean of the squared errors)

np.mean(np.abs(errors))

np.mean(np.square(errors))

→ large errors amplified
→ penalises large errors

RMSE (root of MSE)

np.sqrt(np.mean(np.square(errors)))
→ same scale as original dependent variable

Compute both MAE and RMSE.

The RMSE is best overall as it is more sensitive to large errors. However, if there are error outliers then the difference between the MAE and the RMSE is likely going to be large.

In the case of model prone to outliers, it is typically more interesting to look at the MAE.

Model evaluation with sci-kit learn

mean_absolute_error(y_train, predictions_train)
mean_absolute_error(y_test, predictions_test)

root_mean_squared_error(y_train, predictions_train)
root_mean_squared_error(y_test, predictions_test)

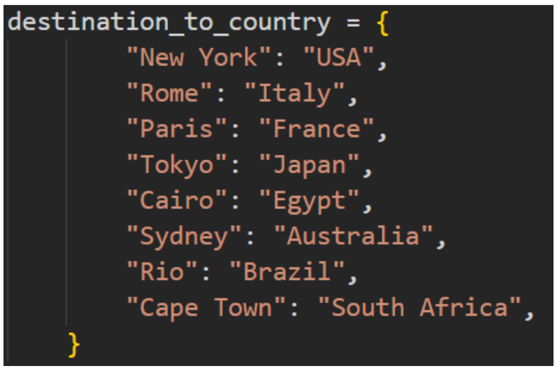
Overfitting when errors on training set << errors on test set.

Underfitting when errors on training set and test set equally large.

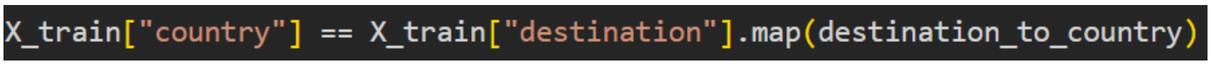
Improve model with feature engineering

INTERACTION

Combine categorical columns after mapping them:



```
destination_to_country = {
        "New York": "USA",
        "Rome": "Italy",
        "Paris": "France",
        "Tokyo": "Japan",
        "Cairo": "Egypt",
        "Sydney": "Australia",
        "Rio": "Brazil",
        "Cape Town": "South Africa",
    }
```

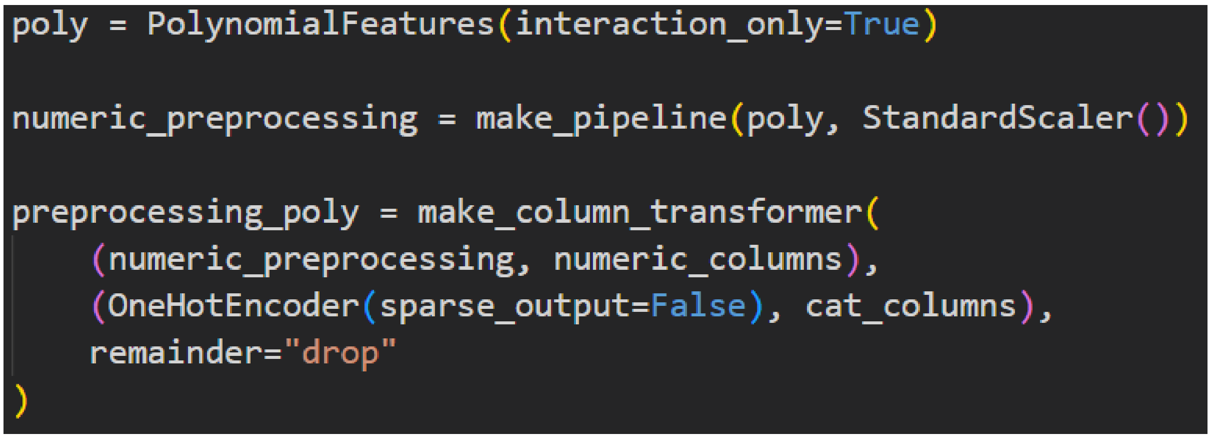


```
X_train["country"] == X_train["destination"].map(destination_to_country)
```

Boolean mask returns True or False for every row.

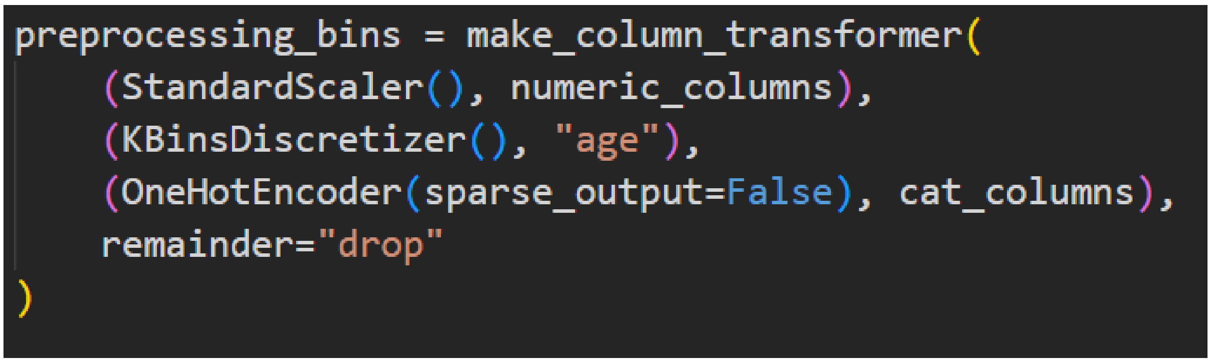
Part behind the == will convert every destination to the key-value pair defined in the dictionary used in the .map function.

Combine numeric columns with PolynomialFeatures. Add this into the numeric processing:



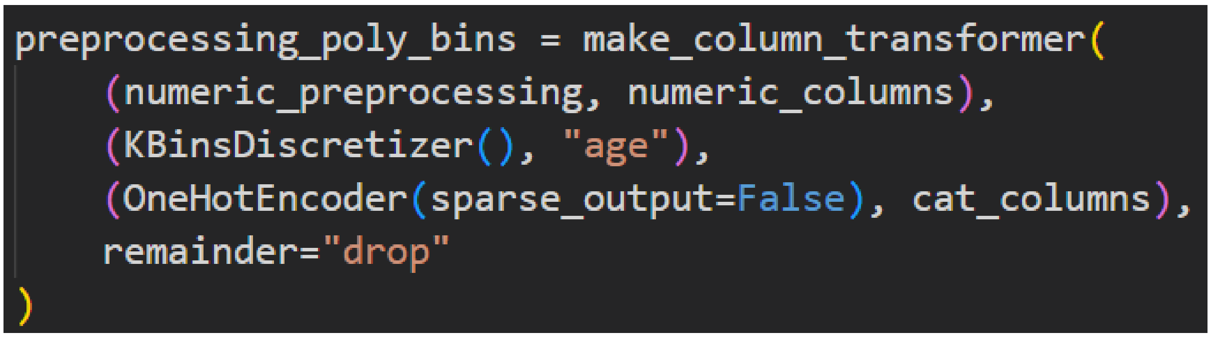
```
poly = PolynomialFeatures(interaction_only=True)

numeric_preprocessing = make_pipeline(poly, StandardScaler())

preprocessing_poly = make_column_transformer(
    (numeric_preprocessing, numeric_columns),
    (OneHotEncoder(sparse_output=False), cat_columns),
    remainder="drop"
)
```

BINNING



```
preprocessing_bins = make_column_transformer(
    (StandardScaler(), numeric_columns),
    (KBinsDiscretizer(), "age"),
    (OneHotEncoder(sparse_output=False), cat_columns),
    remainder="drop"
)
```

Or even combine it together with polynomial features:



```
preprocessing_poly_bins = make_column_transformer(
    (numeric_preprocessing, numeric_columns),
    (KBinsDiscretizer(), "age"),
    (OneHotEncoder(sparse_output=False), cat_columns),
    remainder="drop"
)
```

Find the best model with cross validation

k-fold cross validation!
→ k is number of chunks created and thus k times splitting the dataset **on the training data only**

→ model gets k shots at proving its worth
→ mean performance is observed to select a constant performing model

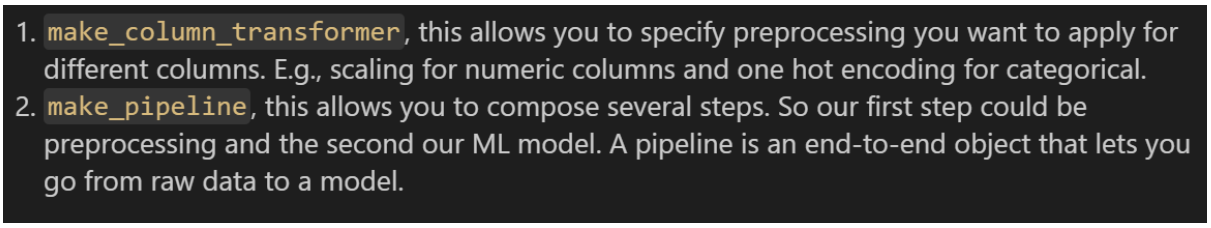
cross_val_score(model_pipe, X_train, y_train, cv=N, scoring=”neg_root_mean_squared_error”)

N = number of k folds

CAVEAT computationally expensive!

You can judge a model without using the test set!

Lab 7 (part 1)



1. `make_column_transformer`, this allows you to specify preprocessing you want to apply for different columns. E.g., scaling for numeric columns and one hot encoding for categorical.
2. `make_pipeline`, this allows you to compose several steps. So our first step could be preprocessing and the second our ML model. A pipeline is an end-to-end object that lets you go from raw data to a model.

make_column_transformer: Also option to use KBinsDiscritizer on specific numeric column(s).

Parameter and hyperparameters (THEORY ONLY)

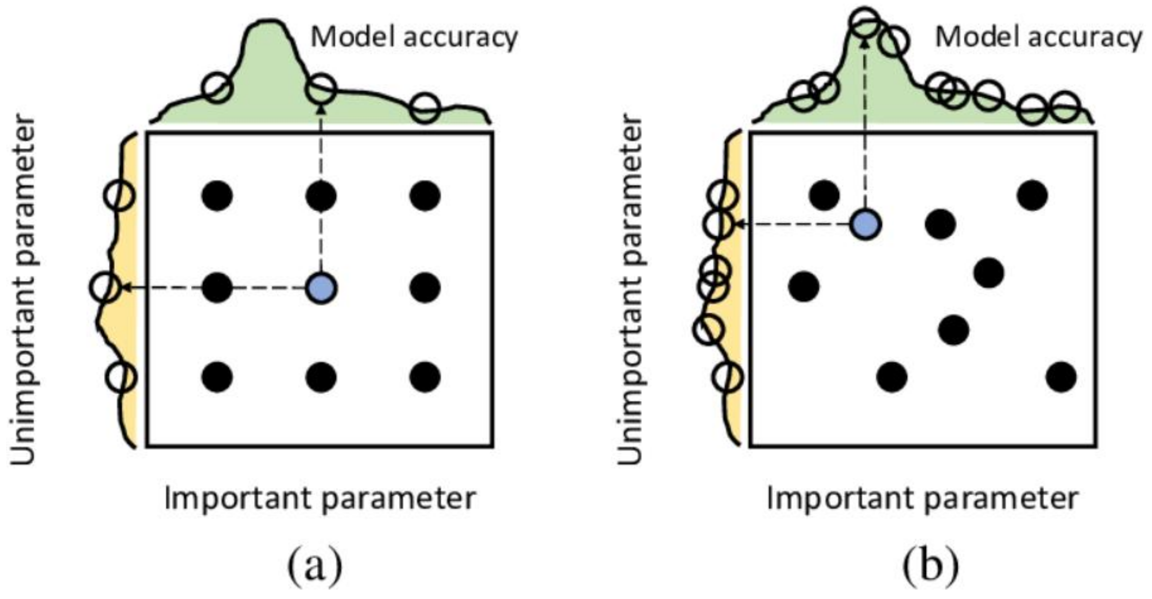
Hyperparameters are the settings of the ML model.

Parameters are what the model uses to make predictions e.g. coefficients of linear regression or the splits in a decision tree.

Examples of hyperparameters:

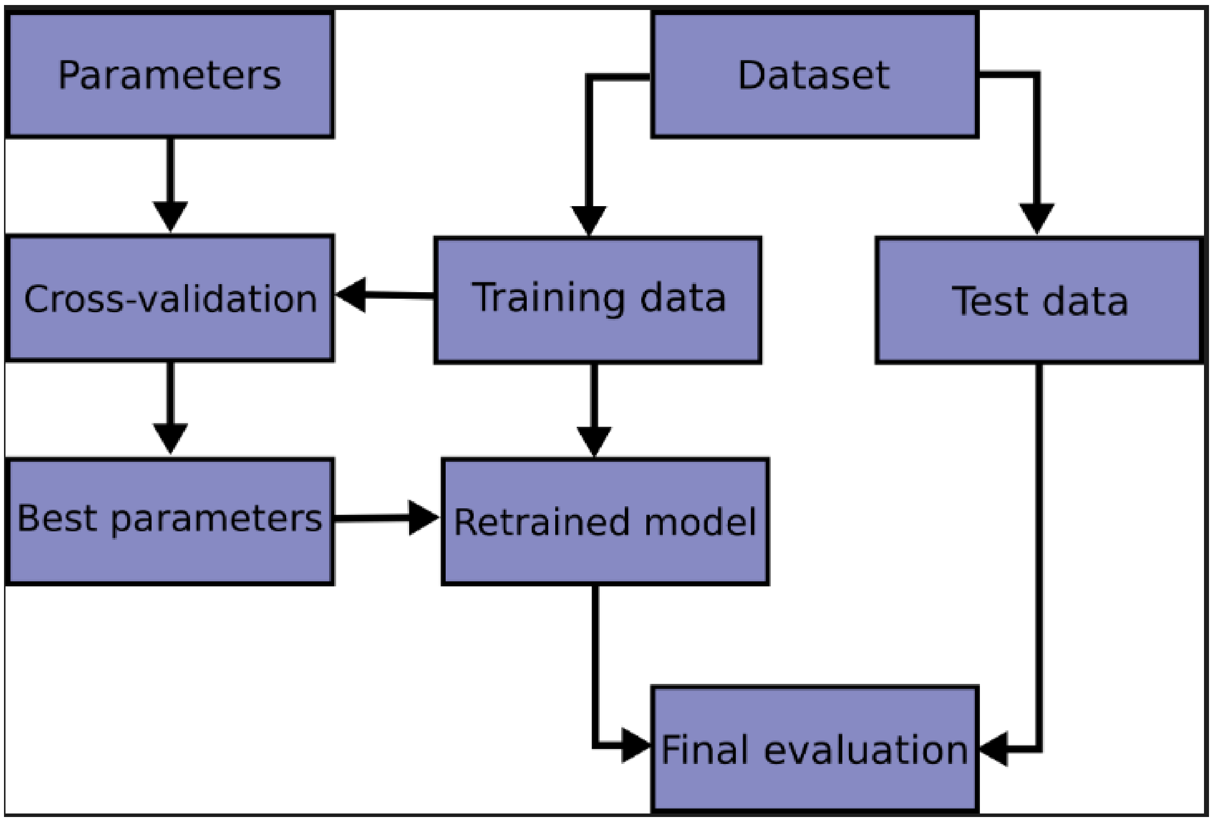
- Max amount of splits in decision tree
- Amount of trees in random forest and gradient boosting
- Regularization constant in linear regression

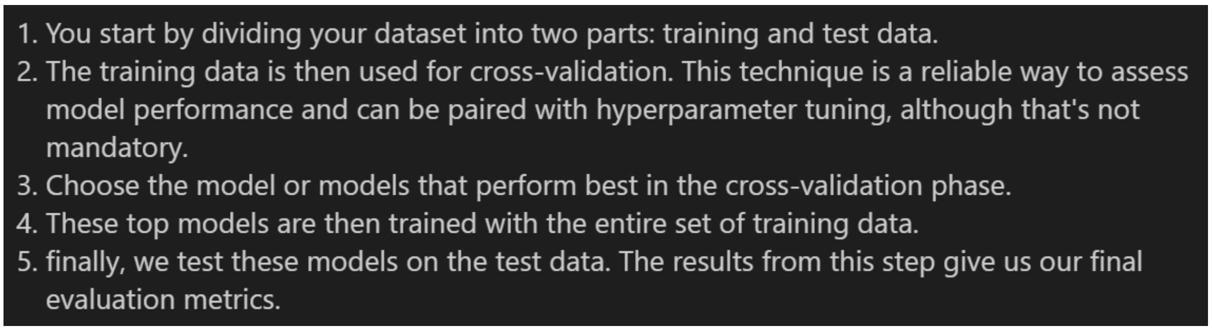
➔ These help to combat overfitting!



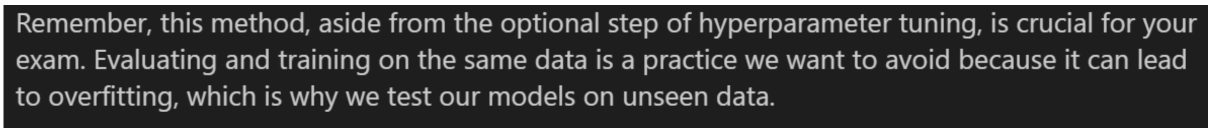
Look for best hyperparameter via grid search (A: thorough but computationally very expensive) or via random search (B: quicker but less focused).

THIS SHOULD NOW BE CLEAR:





1. You start by dividing your dataset into two parts: training and test data.
2. The training data is then used for cross-validation. This technique is a reliable way to assess model performance and can be paired with hyperparameter tuning, although that's not mandatory.
3. Choose the model or models that perform best in the cross-validation phase.
4. These top models are then trained with the entire set of training data.
5. finally, we test these models on the test data. The results from this step give us our final evaluation metrics.



Remember, this method, aside from the optional step of hyperparameter tuning, is crucial for your exam. Evaluating and training on the same data is a practice we want to avoid because it can lead to overfitting, which is why we test our models on unseen data.

Lab 7 (part 2)