DAKD 2024 EXERCISE 1: DATA UNDERSTANDING

This exercise relates to the *data understanding* and *data preparation* stages of the Crisp Data Mining (CRISP-DM) model presented on the course. The questions at this stage of a data-analysis project are for example:

- Is the data quality sufficient?
- How can we check the data for problems?
- How can we clean the data?
- How is the data best transformed for modeling?

It may be tempting to just run a model on data without checking it. However, not doing basic checks can ruin your whole analysis and make your results invalid as well as mislead you in further analyses. There is no excuse for not plotting and checking that the data is as we expect and clean. In this exercise we do just that, check the validity of data and familiarize ourselves with a dataset, also discussing preprocessing and multi-dimensional plotting.

*** FILL YOUR INFORMATION BELOW ***

Arttu Kuitunen
1500550
arasku@utu.fi
13.11.2024

General Guidance for Exercises

- **Complete all tasks**: Make sure to answer all questions, even if you cannot get your script to fully work.
- **Code clarity**: Write clear and readable code. Include comments to explain what your code does.
- **Effective visualizations**: Ensure all plots have labeled axes, legends, and captions. Your visualizations should clearly represent the underlying data.
- **Notebook organization**: You can add more code or markdown cells to improve the structure of your notebook as long as it maintains a logical flow.
- **Submission**: Submit both the `.ipynb` and `.html` or `.pdf` versions of your notebook. Before finalizing your notebook, use the "Restart & Run All" feature to ensure it runs correctly.

Grading Criteria

- The grading scale is **Fail/Pass/Pass with honors (+1)**.
  - To **pass**, you must complete the required parts [1-7].
  - To achieve **Pass with honors**, complete the bonus exercises.

Technical Issues

- **Initial troubleshooting**: If you encounter problems, start with an online search to find solutions, but do not simply copy and paste code. Understand any code you use and integrate it appropriately.
- **External sources**: Cite all external sources used, whether for code or explanations.
- **Help resources**: If problems persist, ask for help in the course discussion forum, at exercise sessions, or via email to the course assistants. .

Use of AI and Large Language Models

- We **do not encourage** the use of AI tools like ChatGPT. If you use them, critically evaluate their outputs.
  - **Documentation**: Describe how you used the AI tools in your work, including your input and how the output was beneficial.

Time Management

- **Avoid last-minute work**: Do not leave your work until the last moment. No feedback will be available during weekends.

Additional Notes

- You can find the specific deadlines and session times for each assignment on the Moodle course page.
- Ensure all your answers are **concise**—typically a few sentences per question.
- Your `.ipynb` notebook is expected to be **run to completion**, meaning it should execute without errors when all cells are run in sequence.

Packages needed for this exercise:

- The exercise can be done without importing any extra packages, but you can import new ones but bear in mind that if you are importing many new packages, you may be complicating your answer.

In [609…

```
# --- Libraries with a short description ---
import pandas as pd # for data manipulation
import matplotlib.pyplot as plt # for plotting
import numpy as np #for numeric calculations and making simulated data.
import seaborn as sns # for plotting, an extension on matplotlib

# - sklearn has many data analysis utility functions like scaling as well as a lar
```

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import minmax_scale
from sklearn.preprocessing import scale
from sklearn.manifold import TSNE
import math

# This forces plots to be shown inline in the notebook
%matplotlib inline
```

PLOTTING TUTORIAL

This small explanation of the matplotlib package aims to avoid confusion and help you avoid common mistakes and frustration. Matplotlib is an object-oriented plotting package with the benefit of giving the user a lot of control. The downside is that it can be confusing to new users. **If you are having problems with the plotting exercises, return to this tutorial as it explains the needed concepts to do the exercises!**

Figure and axes

All plots in matplotlib are structured with the **figure** and **axes** objects.

- The **figure** object is a container for all plotting elements (in other words, everything we see).
- A figure can have many **axes** . They are the objects you plot on to. The axes can be anywhere inside the figure and can even overlap. Position of axes is defined relative to the figure.

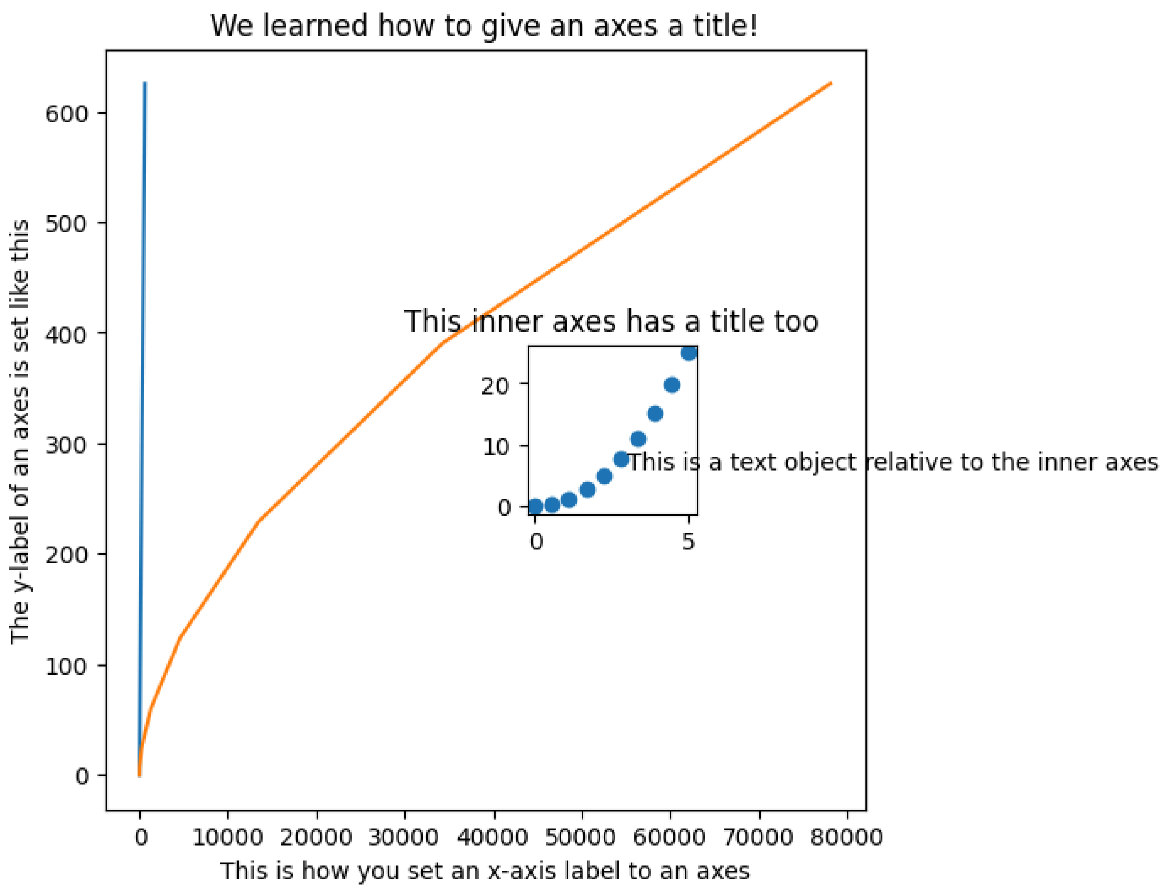
The **axes** objects have the methods you will use to define most of your plots. For example axes.hist() is used to draw a histogram and axes.set_title() to give one axes a title. The name of the object can be a bit confusing as it does not refer to the axes in the way "x-axis" does but to the container of a single plot.

- Below is an example that illustrates how **figures** and **axes** work together in matplotlib. The comments explain what is done in every row of code. You are encouraged to play around with it, but its not required in terms of the exercise . Below, we will create all figures and axes separately, but later on we will use a quicker way to do so.

This is not yet a part of the exercises themselves and you do not need to change anything !

In [610…
```
### --- Lets make some example data ---
x_example_data = np.linspace(0,5,10) # Generate 10 evenly spaced numbers between
y_example_data = x_example_data**2   # Square the x data to create the correspon

### ---- Create a Figure ----
example_figure = plt.figure(figsize=(5, 5))  # You give the size of the figure a

### ---- Create Outer Axes ----
'''
Create axes inside the figure. "e.g example_figure.add_axes(...)"  The list [0.1
  - The left side of the axes is 10% from the left of the figure
  - The bottom of the axes is 10% from the bottom of the figure
  - The axes take up 90% of the figure’s width and height
'''
example_axes_outer = example_figure.add_axes([0.1, 0.1, 0.9, 0.9])

### Set labels and titles for the outer axes ###
example_axes_outer.set_xlabel("This is how you set an x-axis label to an axes")
example_axes_outer.set_ylabel("The y-label of an axes is set like this")
example_axes_outer.set_title("We learned how to give an axes a title!")

### ---- Create Inner Axes ----
example_axes_inner = example_figure.add_axes([0.6, 0.45, 0.2, 0.2]) # Inner axes
example_axes_inner.set_title("This inner axes has a title too")

### ---- Plot Data and Customize ----
example_axes_inner.scatter(x_example_data, y_example_data)  # Scatter plot on in

# you can add multiple things such as (lines) can be plotted on same outer axis.
example_axes_outer.plot(x_example_data**4, y_example_data**2)
example_axes_outer.plot(x_example_data**7, y_example_data**2)

### ---- Add Text Annotation ----
# If you want to add other objects, you add them to axes too, like text
# Now you specify the location relative to the parent axes
example_axes_inner.text(3, 6, "This is a text object relative to the inner axes"
```

Out[610…
```
Text(3, 6, 'This is a text object relative to the inner axes')
```



Subplots: Creating Multiple Axes in a Grid

A common way to start plotting in Matplotlib is to use the `plt.subplots()` function. This function automatically creates a figure and a specified number of axes (subplots) arranged in a grid, linking them to the figure. Even when creating just one subplot, `plt.subplots()` is often used as it offers flexibility and ease of management.

The most important arguments for `plt.subplots()` are:

- **nrows**: The number of rows of subplots in the grid.
- **ncols**: The number of columns of subplots in the grid.
- **figsize**: A tuple like `(6, 4)` that sets the figure size in inches. The first value is the width, and the second is the height.
- **sharex**: If `True`, all subplots share the same x-axis scale and ticks.
- **sharey**: If `True`, all subplots share the same y-axis scale and ticks.

Example:

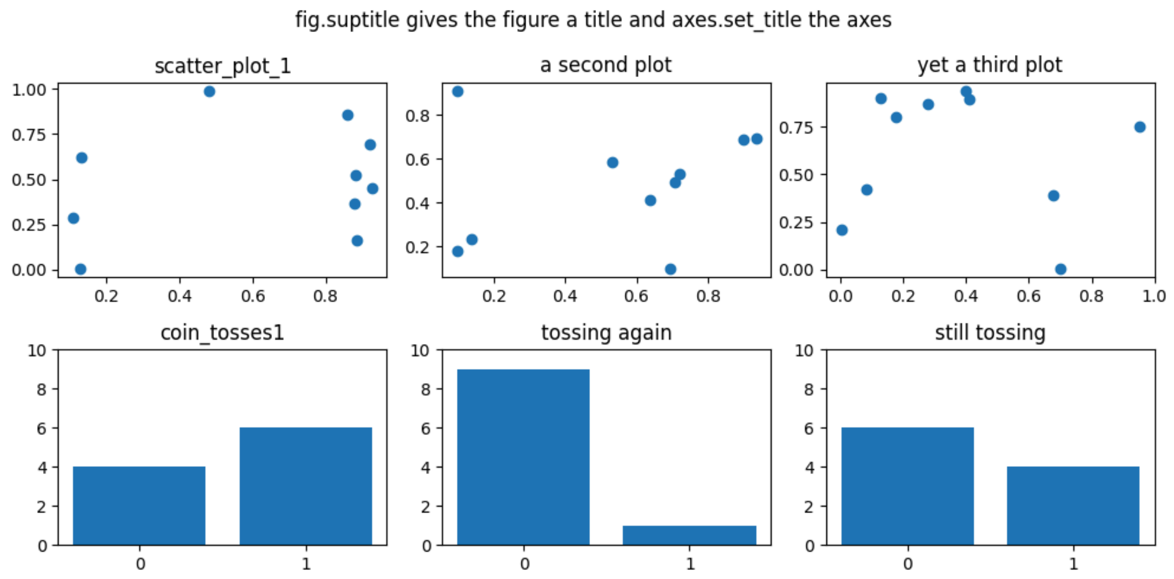
Below is an example of how to create subplots. It includes a loop to fill the subplots using the `enumerate()` function, which provides an index that can be used to access individual subplot axes. This is a useful pattern when plotting multiple subplots programmatically.

The function `plt.tight_layout()` is also handy for arranging subplots. It automatically adjusts the positions of the axes to ensure they don't overlap, making the figure look cleaner and more readable.It should be called after the plot is finished.

In [611…
```
# ----- Create some random data for the example -----
# We generate 3 sets of continuous numeric features and 3 sets of binary feature

# Generate random continuous data (3 arrays, each with 10 samples of 2 numeric f
numeric_datas = [np.random.rand(10, 2) for _ in range(3)]  # 3 arrays of 10x2 ra

# Generate binary data (3 arrays, each counting occurrences of 0s and 1s from ra
# np.random.randint creates random integers (0 or 1), np.unique counts them
binary_datas = [
    np.unique(np.random.randint(0, 2, size=10), return_counts=True)[1]
    for _ in range(3)
]  # 3 arrays of counts of 0s and 1s (binary features)
# numeric_datas
binary_datas
```

Out[611…
```
[array([4, 6], dtype=int64),
 array([9, 1], dtype=int64),
 array([6, 4], dtype=int64)]
```

In [612…
```
# Create figure with six axes in a 2*3 grid and set up titles -----------------
fig, axes = plt.subplots(2,3, figsize = (10,5)) # now axes have indexes like axe
numeric_plot_titles = ['scatter_plot_1', 'a second plot', 'yet a third plot' ]#s
binary_plot_titles = ['coin_tosses1', 'tossing again', 'still tossing' ]#some ti


# Enumerate the index into the axes, fill the first 3 columns of first row with
i = 0 # for indexing to the row of the axes [**i**, j]
for j, numeric_data in enumerate(numeric_datas): # j = [0,1, ... n_datasets] for
    axes[i, j].scatter(x = numeric_data[:, 0], y = numeric_data[:, 1]) #plots ar
    axes[i, j].set_title(numeric_plot_titles[j]) #set a title for each axes
plt.tight_layout()


# Plot the binary data ---------------------------------------------------------
i = 1 # second rowd
for j, binary_data in enumerate(binary_datas): # j = [0,1, ... n_datasets] for f
    axes[i, j].bar(x = ["0","1"], height = binary_data) #make a barplot
    axes[i, j].set_title(binary_plot_titles[j]) #set a title for each axes
    axes[i, j].set_ylim((0,10)) # set the yaxis limits, set_xlim works the same

fig.suptitle("fig.suptitle gives the figure a title and axes.set_title the axes"
plt.tight_layout()
```



Seaborn and Matplotlib (new)

Seaborn is a popular plotting library built on top of Matplotlib. It was designed to make creating statistical plots easier, more intuitive, and visually appealing with minimal effort. Seaborn is especially known for its default color palettes and built-in support for complex visualizations, making it faster to use compared to plain Matplotlib.

When working with Seaborn, it’s important to understand that there are two kinds of plotting functions:

- **Figure-level plots**: These functions manage the entire figure themselves (including creating the figure and subplots) and cannot be easily integrated into custom subplot grids. Examples include `sns.catplot()` and `sns.lmplot()`.
- **Axes-level plots**: These functions work on individual Matplotlib axes and can be combined with Matplotlib's `subplots()` to create complex, multi-plot figures. Examples include `sns.scatterplot()` and `sns.histplot()`.

For axes-level plots, you can pass a Matplotlib axes object to the Seaborn plotting function to specify where the plot should appear. This allows you to mix Matplotlib and Seaborn plots in the same figure. Below is an example of how to use Seaborn to plot on a specific set of axes.

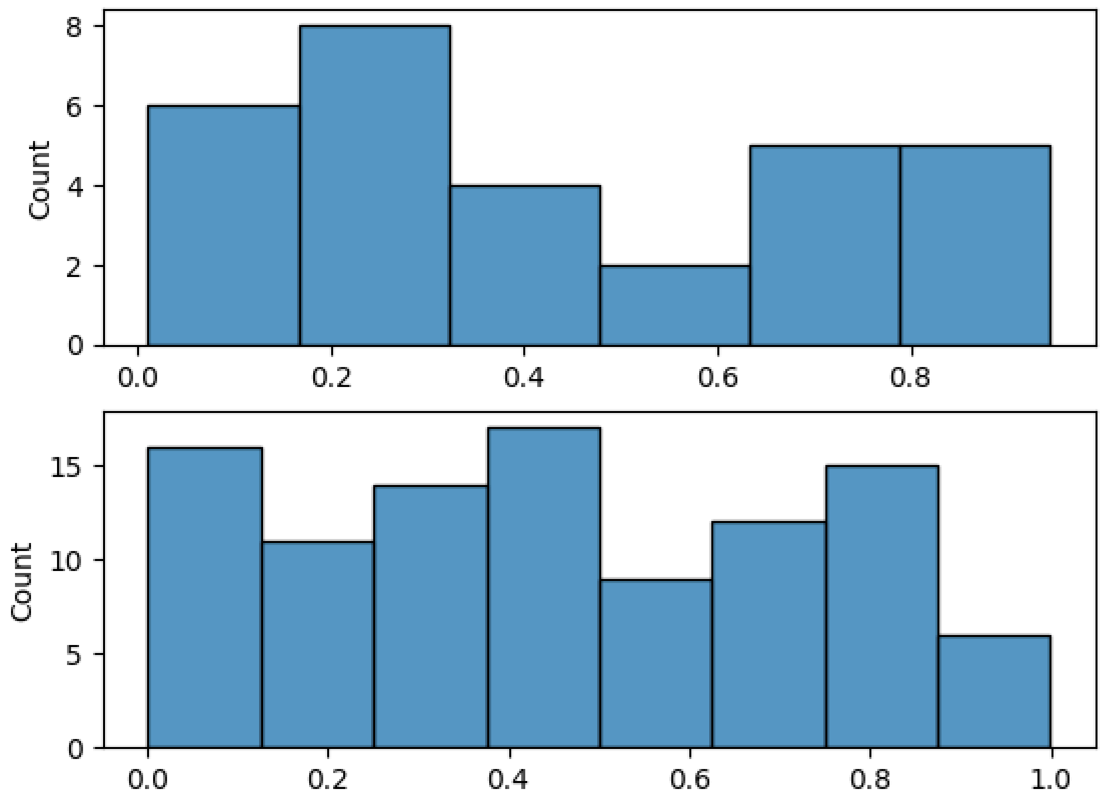
If you'd like to dive deeper into the different types of Seaborn functions and when to use them, you can find more information in the Seaborn function overview.

In [613…
```
fig, axes = plt.subplots(2)

# make some data
random_data_a = np.random.rand(30)
random_data_b = np.random.rand(100)

# print the data we are plotting
```

```
sns.histplot(data = random_data_a, ax = axes[0]) # we make a seaborn plot and pu
sns.histplot(data =  random_data_b, ax = axes[1]) # we make a seaborn plot and p
```

Out[613…

```
<Axes: ylabel='Count'>
```



START OF EXERCISES

1. Introduction to the dataset

The dataset in this exercice contains comprehensive health information from hospital patients with and without cardiovascular disease. The target variable "cardio," reflects the presence or absence of the disease, which is characterized by a buildup of fatty deposits inside the arteries (blood vessels) of the heart.

As is often the case with data analysis projects, the features/variables have been retrieved from different sources:

- doctors notes (texts)
- examination variables that have come from a database containing lab results or taken during a doctors examination
- self reported variables

The exercise data has the following columns/attributes:

| Feature | Type | Explanation | | :- | :- | :- | age | numeric | The age of the patient in days | gender | binary | Male/Female | body_mass | numeric | Patient's measured weight, in kilograms (kg). | height | numeric | Patient's measured height, in centimeters (cm). | blood_pressure_high | numeric | Measured Systolic blood pressure | blood_pressure_low | numeric | Measured Diastolic blood pressure | smoke | binary | A subjective feature based on asking the patient whether or not he/she smokes | active | binary | A subjective feature based on asking the patient whether or not he/she exercises regularly | serum_lipid_level | categorical | Serum lipid / Cholesterol associated risk information evaluated by a doctor |family_history| binary | Indicator for the presence of family history of cardiovascular disease based on medical records of patients | cardio | binary | Whether or not the patient has been diagnosed with cardiac disease.

Reading data

It is good practice to read the features in using their correct types instead of fixing them later. Below, there is ready-made code for you to read in the data, using the data types and column names listed in the above table. Don't change the name of the variable, *data*. It is important in later exercises (for example in ex. 5e) that this is the name of the variable. If you have the dataset in the same folder as this notebook, the path already given to you should work.

In [614…

```
# --- READ IN DATA (no need to change) --------
data_path = "CardioCare_ex1.csv" #if you just give the name of the file it will
data = pd.read_csv(data_path, dtype = {'age': 'int64', 'height': 'int64', 'body
       'active':'boolean', 'cardio':'boolean', 'serum_lipid_level':'category',
```

Exercise 1 a)

1. First, print out the first five rows of the data.

2. Then, save the feature names to lists by their types:

   - Create three lists named **numeric_features**, **binary_features**, and **categorical_features**.
   - These lists should contain the **names** of the features based on their types:
     - Numeric features (e.g., `age`, `body_mass`, etc.)
     - Binary features (also known as boolean, e.g., `gender`, `smoke`, `cardio`, etc.)
     - Categorical features (e.g., `serum_lipid_level`)

Important Notes:

When working with DataFrames, it is often useful to organize column names into lists. This practice simplifies data manipulation and analysis. Once the feature names are organized, you can easily select, filter, or apply operations to specific groups of features. This also helps to avoid typing errors and reduces repetition.

For example, once you create your list of numeric features, you can select all columns containing numeric data with the following command:

```
data[numeric_features]
```

In [615…

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 210 entries, 0 to 209
Data columns (total 11 columns):
 #   Column               Non-Null Count  Dtype   
---  ------               --------------  -----   
 0   age                  210 non-null    int64   
 1   gender               210 non-null    boolean 
 2   height               210 non-null    int64   
 3   body_mass            210 non-null    int64   
 4   blood_pressure_high  210 non-null    int64   
 5   blood_pressure_low   210 non-null    int64   
 6   smoke                210 non-null    boolean 
 7   active               210 non-null    boolean 
 8   cardio               210 non-null    boolean 
 9   serum_lipid_level    210 non-null    category
 10  family_history       210 non-null    boolean 
dtypes: boolean(5), category(1), int64(5)
memory usage: 10.8 KB
```

In [616…

```
# --- Your code here for 1 a) ---
numeric_features = data.columns[data.dtypes == 'int64'].tolist()
binary_features = data.columns[data.dtypes == 'boolean'].tolist()
categorical_features = data.columns[data.dtypes == 'category'].tolist()

numeric_features, binary_features, categorical_features
```

Out[616…

```
(['age', 'height', 'body_mass', 'blood_pressure_high', 'blood_pressure_low'],
 ['gender', 'smoke', 'active', 'cardio', 'family_history'],
 ['serum_lipid_level'])
```

In many data analysis projects, the data is often not collected specifically for analysis purposes. Instead, it may come from various sources or be collected for entirely different reasons. As a result, the data might not be well-formatted and could contain errors or inconsistencies.

It might be tempting to immediately apply a model to the data "as is," but it is crucial to first **check the data for quality issues**. Ignoring potential data issues can lead to misleading conclusions, undermining the entire analysis.

Why Data Quality Checks Matter:

One standard routine to ensure data quality is:

1. **Calculate descriptive statistics** for each feature. This gives an overview of the distribution, range, and possible anomalies.
2. **Visualize the features** to check whether the values are realistic and within expected ranges.

This step helps identify outliers, incorrect data entries, or formatting issues, ensuring that your analysis is based on clean and reliable data.

Descriptive Statistics and Data Types

It's important to note that certain descriptive statistics might not be meaningful for specific types of features. For instance, calculating the "mean" for binary or categorical features may not offer valuable insight. In **pandas** (as in many other data analysis packages), some functions behave differently depending on the data type of the column.

In the following exercises, we will explore:

- **Descriptive statistics** for the dataset.
- How the results and behavior of descriptive functions can vary based on the data type (e.g., numeric vs. categorical features).

Exercise 2 a) Print out the data types of your dataset below.

*Perhaps the most common data types in pandas (see https://pandas.pydata.org/docs/user_guide/basics.html#basics-dtypes) are **float**, **int**, **bool** and **category**.*

In [617…

```
# --- 2 a) Print the feature types of your dataset --- #
data.dtypes
```

Out[617…

```
age                      int64
gender                 boolean
height                   int64
body_mass                int64
blood_pressure_high      int64
blood_pressure_low       int64
smoke                  boolean
active                 boolean
cardio                 boolean
serum_lipid_level     category
family_history         boolean
dtype: object
```

Exercise 2 b) Use the **DataFrame.describe() method** in the cell below on your data.

In [618…

```
# --- Your code for 2 b) --- #
data.describe()
```

Out[618…

	age	height	body_mass	blood_pressure_high	blood_pressure_low
count	210.000000	210.000000	210.000000	210.000000	210.000000
mean	19455.504762	164.180952	73.895238	127.857143	81.814286
std	2429.010199	7.534648	14.612326	17.508947	9.947652
min	14367.000000	142.000000	45.000000	90.000000	50.000000
25%	17635.750000	158.000000	64.000000	120.000000	80.000000
50%	19778.000000	164.000000	70.000000	120.000000	80.000000
75%	21230.500000	170.000000	81.000000	140.000000	90.000000
max	23565.000000	195.000000	125.000000	190.000000	120.000000

Exercise 2 c) Did you get all of the features statistics or not? What do you think happened?

DataFrame.describe() automatically ignores those that are not numerical, because most of the statistics provided are only applicable for numbers.

Exercise 2 d) Calculate descriptives for the binary (boolean) features and the categorical feature

tip: in python, same type data structures can in many cases be concatenated using the + operator. If youre using the lists of names you created to subset, you can concatenate the *two lists of feature names and use the resulting list to help you subset the dataframe*

In [619…

```
# 2 d) Your code here #
bin_cat_features = binary_features + categorical_features
data[bin_cat_features].describe()
```

Out[619…

	gender	smoke	active	cardio	family_history	serum_lipid_level
count	210	210	210	210	210	210
unique	2	2	2	2	2	4
top	False	False	True	False	False	normal
freq	129	186	162	105	128	153

Now, we will explore **what happens if the data is read using the default settings** (i.e., without specifying the data types for the features). In this case, we are **not providing information about the data types (dtypes)** to `pd.read_csv`, meaning no additional arguments are passed when loading the data.

Run the cell below (you don't need to modify the code) and observe the output of the data that has been incorrectly read due to missing dtype information. Then, compare this output with the data you loaded earlier using the correct dtypes, and check the descriptive statistics.

In [620…

```
# read in the dataset with no arguments
wrongly_read_data = pd.read_csv(data_path)

# calculate descriptives for the data that was wrongly read in.
wrongly_read_data.describe()
```

Out[620…

	age	gender	height	body_mass	blood_pressure_high	blood_
count	210.000000	210.000000	210.000000	210.000000	210.000000	
mean	19455.504762	0.385714	164.180952	73.895238	127.857143	
std	2429.010199	0.487927	7.534648	14.612326	17.508947	
min	14367.000000	0.000000	142.000000	45.000000	90.000000	
25%	17635.750000	0.000000	158.000000	64.000000	120.000000	
50%	19778.000000	0.000000	164.000000	70.000000	120.000000	
75%	21230.500000	1.000000	170.000000	81.000000	140.000000	
max	23565.000000	1.000000	195.000000	125.000000	190.000000	

Exercise 2 e) Based on the output above, can you identify what went wrong with the data presentation? Why was it important to correctly define the data types when loading the dataset?

Serum lipid level is missing (from the described features) because it is of type object. All boolean values are described as integer, therefore showing descriptive measures of the values between 0 and 1

---

3. Plotting numeric features

Descriptives don't really give a full or intuitive picture of the distribution of features. Next, we will make use of different plots to check the data quality.

Exercise 3 a) Plot histograms for the **numeric features** to visually inspect their distributions. (Refer to the tutorial if you need assistance with plotting.)

_tip: When using `plt.subplots()`, if you provide only one argument for the grid size (e.g., `plt.subplots(3)`), it will create a **one-dimensional grid**. You can then index this grid with a single index, making it easier to loop through and assign plots to each subplot.

In [621…

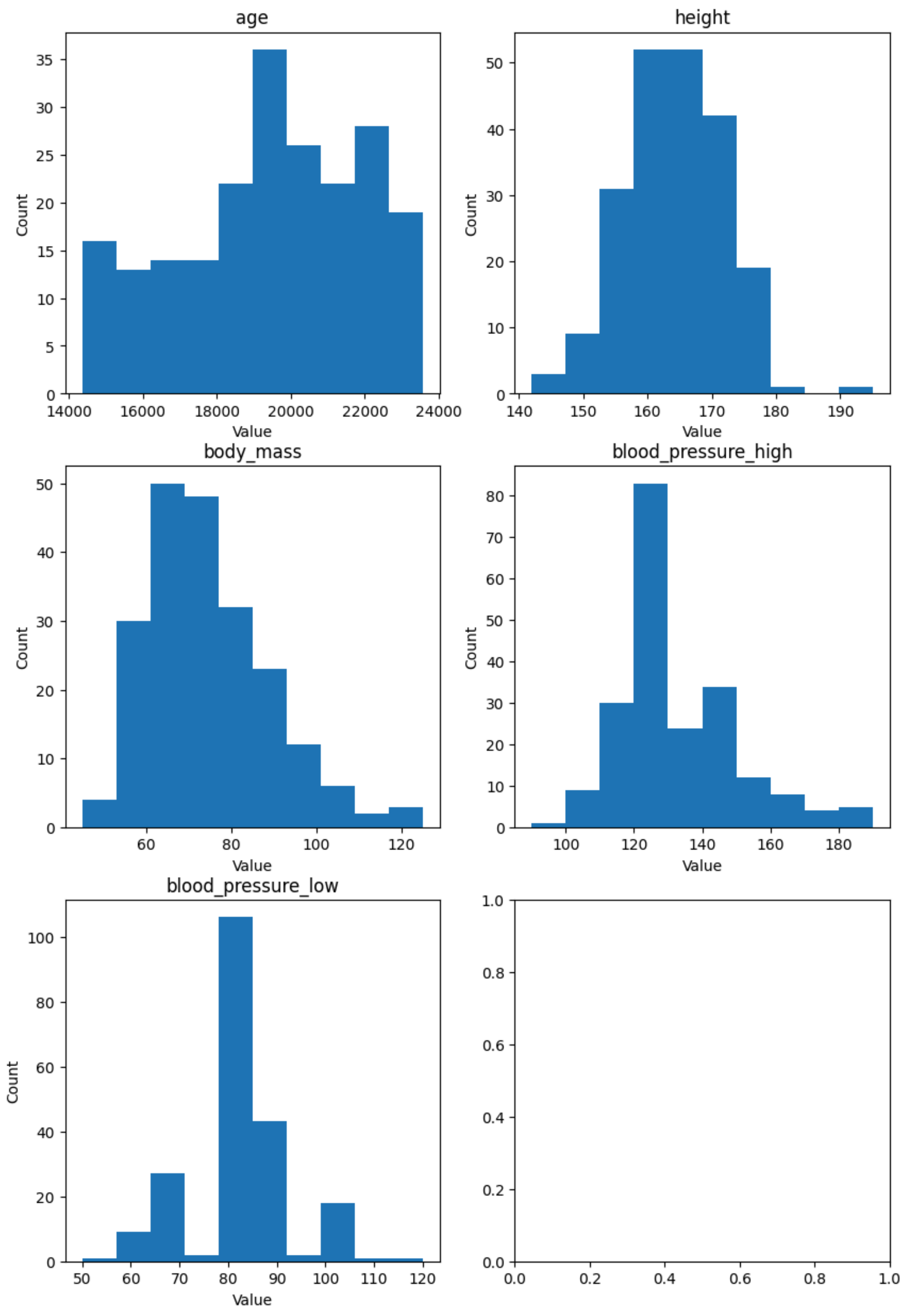
```
numeric_features
```

Out[621…

```
['age', 'height', 'body_mass', 'blood_pressure_high', 'blood_pressure_low']
```

In [622…

```
# --- Your code for 3 a) here --- #
# Using ceil() to calulate figure rows dynamically based on feature count
fig, axes = plt.subplots(math.ceil(len(numeric_features)/2), 2, figsize = (10,

# Looping feature plots to 2 columns. Could still hide the empty plot.
for j, feature in enumerate(numeric_features):
    row = j // 2
    col = j % 2
    axes[row, col].hist(data[feature], bins = 10)
    axes[row, col].set_title(feature, fontsize = 12)
    axes[row, col].set_xlabel("Value", fontsize = 10)
    axes[row, col].set_ylabel("Count", fontsize = 10)

# if len(numeric_features) % 2 != 0:
#     fig.delaxes(axes[-1, -1])
```



4. Plotting binary and categorical features

Exercise 4 a) Plot **barplots** for each of the **non-numeric features** in the dataset. Make sure to **use fractions** instead of the actual frequencies of the categories.

Tips:

- To create the barplots, refer to the documentation for `axes.bar`.
- To obtain the fractions of each category, use the `value_counts()` function with the `normalize` argument set to `True`. This will return the relative frequencies of each category (proportion of each category relative to the total).

Note:

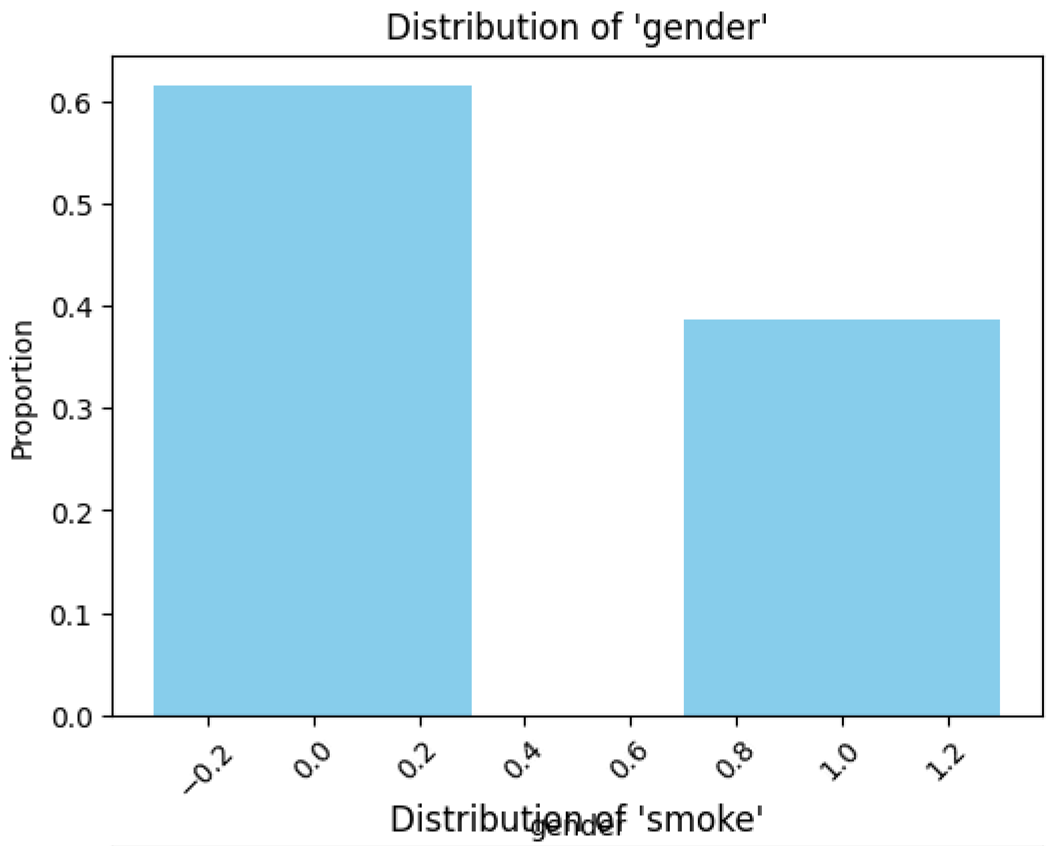
If you imported boolean features as `pandas` dtype `boolean`, you may find it easier to work with plotting libraries like `matplotlib` when these values are represented as numbers (`0` and `1`) instead of `True` and `False`.

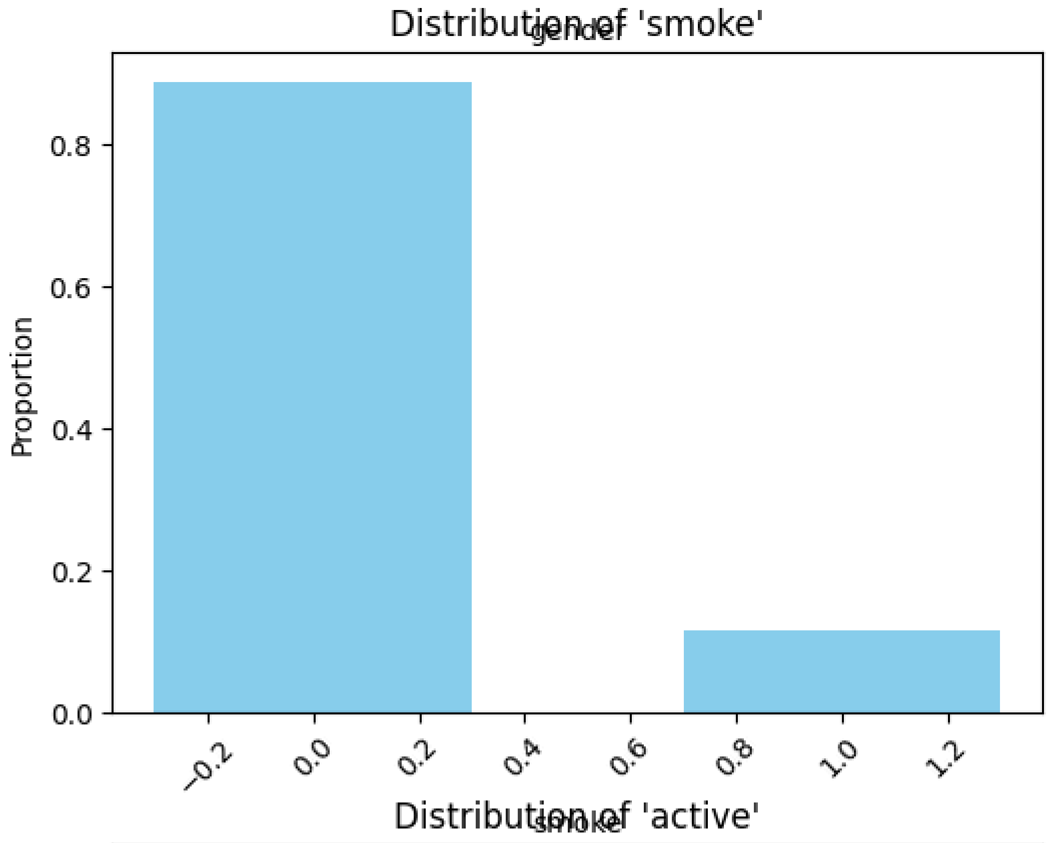
If you encounter any errors while plotting, you can temporarily convert these boolean values to integers or floats using the `.astype()` method:

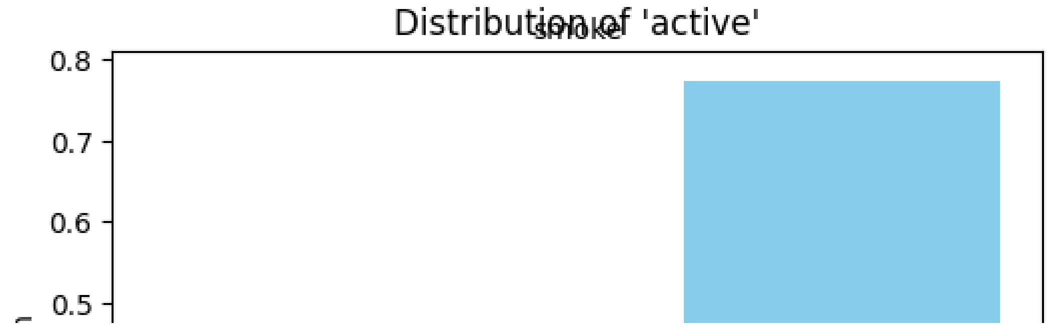
```
# Example of converting boolean to int:
data['..'] = data['..'].astype(int)
```

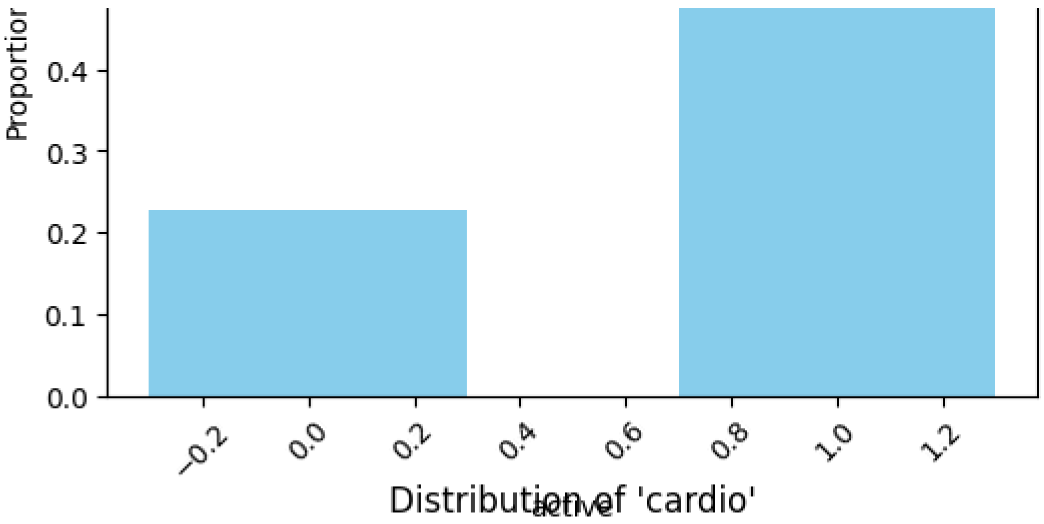
In [623…

```
### Your code for 4 a) here ###
non_numeric = binary_features + categorical_features

# Set up subplots: one subplot per feature
fig, axes = plt.subplots(len(non_numeric), 1, figsize=(6, len(non_numeric) * 5)

# Loop over each feature and create a bar plot in its respective subplot
for i, feature in enumerate(non_numeric):
    # Get the counts for each category in the feature
    counts = data[feature].value_counts(normalize=True)

    # Plot on the specific subplot
    axes[i].bar(counts.index, counts.values, color='skyblue', width=0.6)
    axes[i].set_title(f"Distribution of '{feature}'", fontsize=12)
    axes[i].set_ylabel("Proportion", fontsize=10)
    axes[i].set_xlabel(feature, fontsize=10)

    # Rotate x-tick labels if they are too long
    axes[i].tick_params(axis='x', rotation=45)



# --------------------------

# axes = plt.gca()
# # Plotting all variables to one bar plot.
# for feature in non_numeric:

#     axes.bar(feature, data[feature].value_counts(normalize=True), width=0.7)
#     axes.set_title("Non numeric variables", fontsize = 15)
#     axes.set_ylabel("Count", fontsize = 12)

# # Make figure and variable labels more readable
# # axes.legend(title='Variables', labels=non_numeric, loc='upper right')
# axes.set_xticks(non_numeric) # Avoiding a warning raised by set_xticklabels()
# axes.set_xticklabels(non_numeric, rotation=45, ha="right", fontsize=10)
```

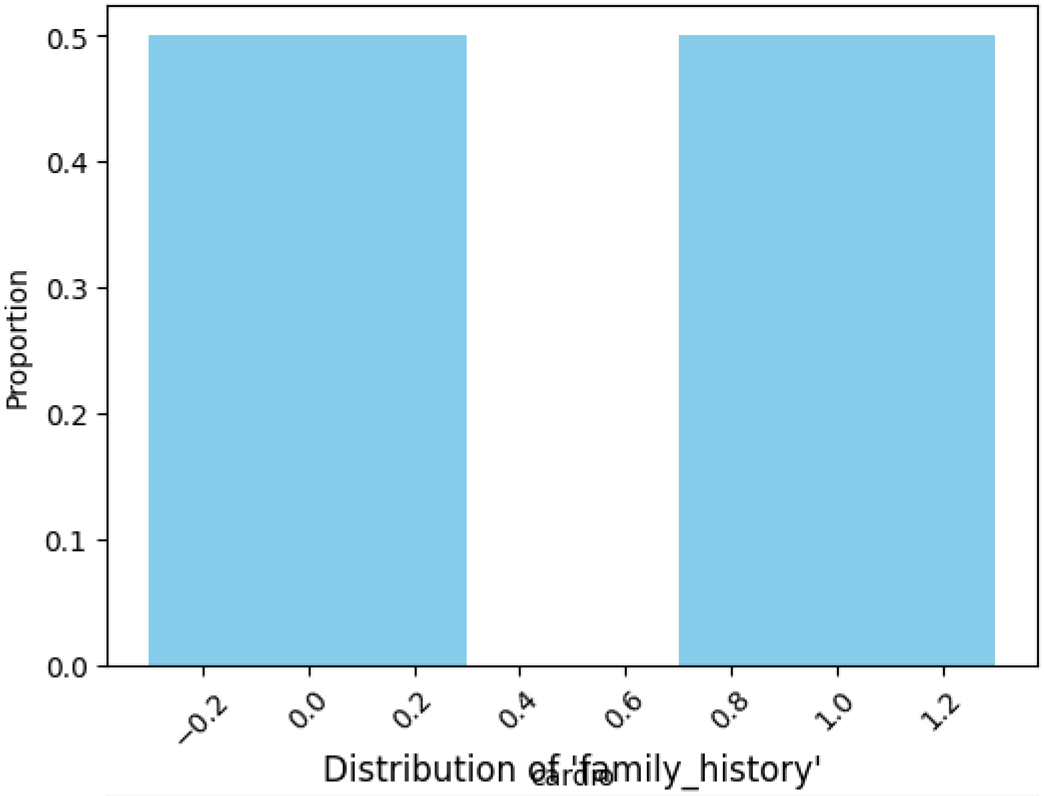

Distribution of 'gender'
Proportion
0.0
0.1
0.2
0.3
0.4
0.5
0.6
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2
gender


Distribution of 'smoke'
Proportion
0.0
0.2
0.4
0.6
0.8
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2
smoke

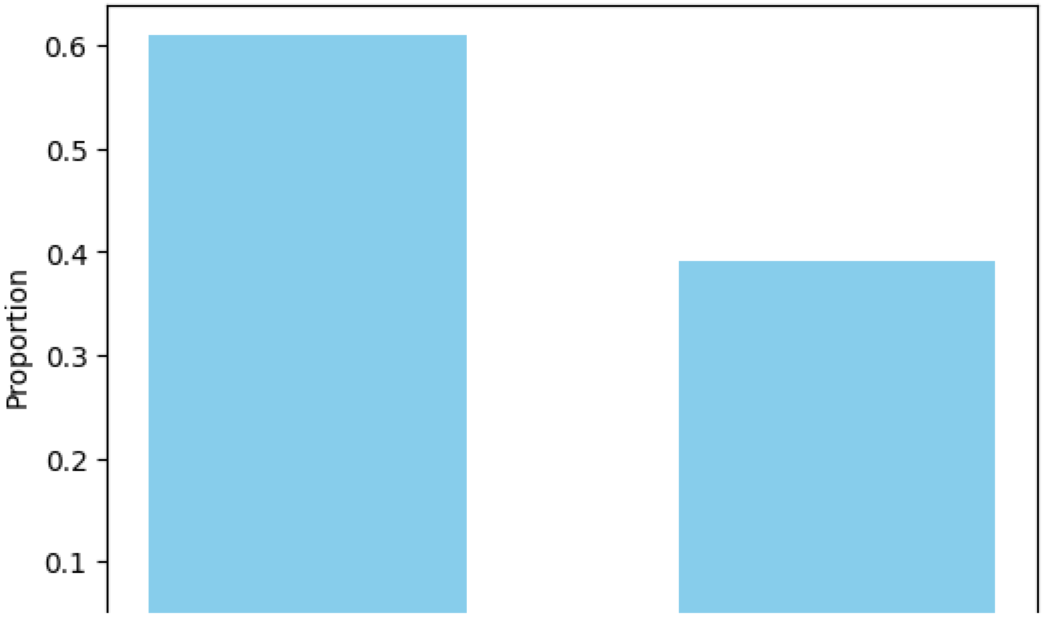

Distribution of 'active'
0.5
0.6
0.7
0.8

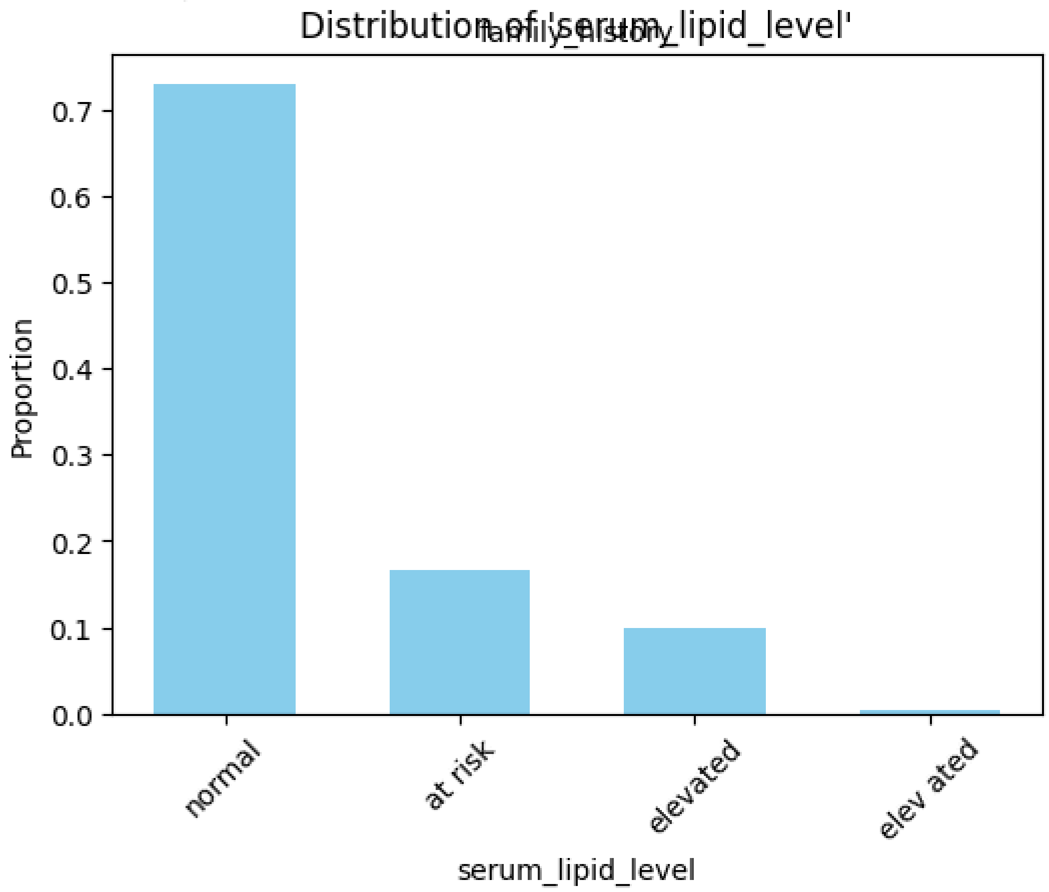


Distribution of 'cardio'



Distribution of 'family_history'





Exercise 4 b) After reviewing the barplots above, Do you notice anything (unusual/irrelevant) with one of the features? If so, Let's try fix it.

If you have read the dtype of a categorical feature as `pandas` dtype `categorical`, you must also use the `remove_categories()` function to remove any unnecessary category levels.

To remove a specific category level, you can use the following example syntax:

```
data['feature_name'] =
data['feature_name'].cat.remove_categories("category name to delete")
```

Serum lipid level has 4 categories, where 1 is wrongly spelled and a duplicate of another categorie. This should be deleted or changed.

In [624…

```
### Your code for 4 b) here ###

# Two options. One to delete the categories, the other to replace them with the
# cat.remoce_categories() will leave a NaN value inplace.
data['serum_lipid_level'] = data['serum_lipid_level'].cat.remove_categories('el
# data['serum_lipid_level'] = data['serum_lipid_level'].replace('elev ated', 'e
```

In [625…
```
for feature in non_numeric:
    print(f"Unique values in '{feature}':")
    print(data[feature].unique())
```

```
Unique values in 'gender':
<BooleanArray>
[False, True]
Length: 2, dtype: boolean
Unique values in 'smoke':
<BooleanArray>
[False, True]
Length: 2, dtype: boolean
Unique values in 'active':
<BooleanArray>
[True, False]
Length: 2, dtype: boolean
Unique values in 'cardio':
<BooleanArray>
[False, True]
Length: 2, dtype: boolean
Unique values in 'family_history':
<BooleanArray>
[False, True]
Length: 2, dtype: boolean
Unique values in 'serum_lipid_level':
['elevated', 'normal', 'at risk', NaN]
Categories (3, object): ['at risk', 'elevated', 'normal']
```

---

5. Feature generation and exploration

Feature Engineering is a crucial step in the process of preparing data for most data analysis projects. It involves creating new features or modifying existing ones to improve the performance of predictive models. Feature engineering is a combination of domain knowledge, creativity, and data analysis, and it can have a significant impact on the success of a data analysis project.

BMI, or **Body Mass Index**, is a simple numerical measure that is commonly used to assess an individual's body weight in relation to their height. In our use case, BMI can be a useful indicator in the prediction of cardiovascular problems, as it could provide a well-established link between obesity and an increased risk of developing the disease.

$$\mathrm{BMI} = \frac{\text{Body mass (kg)}}{(\text{height (m)})^2}$$

Exercise 5 a) Generate a new feature called **BMI** using the provided formula that incorporates the **height** and **body_mass** features.

_tip: In this dataset, the **height** is recorded in centimeters. Before applying the formula, ensure that you convert the height from centimeters to meters by dividing by 100.

In [626…
```
### Your code for 5 a) here ###
def BMI(height, body_mass):
    height = height / 100
    return body_mass / (height**2)

# Added this when I noticed in the Excercise 6 that I need a BMI column.
data['BMI'] = BMI(data['height'], data['body_mass'])
```

Exercise 5 b) Using the previously calculated feature **BMI** generate a new feature named **BMI_category** that categorizes the values into groups, according to the standard BMI categories :

- Underweight: BMI less than 18.5
- Normal Weight: BMI between 18.5 and 24.9
- Overweight: BMI between 25 and 29.9
- Obese: BMI of 30 or greater

In [627…
```
### Your code for 5 b) here ###
# data['BMI_category'] =
# Using cut to make categorical values out of BMI values, where bins are the va

data['BMI_category'] = pd.cut(data[['height', 'body_mass']].apply(lambda x: BMI
       bins = [0, 18.5, 25, 30, float('inf')],
       labels = ['Underweight', 'Normal', 'Overweight', 'Obese'],
       right = False)

data['BMI_category']
```

Out[627…
```
0          Normal
1          Normal
2          Normal
3           Obese
4           Obese
          ...
205        Normal
206        Normal
207         Obese
208        Normal
209    Overweight
Name: BMI_category, Length: 210, dtype: category
Categories (4, object): ['Underweight' < 'Normal' < 'Overweight' < 'Obese']
```

Now that we have our BMI values, it's a good practice to see if we can spot a hidden trend in our data.

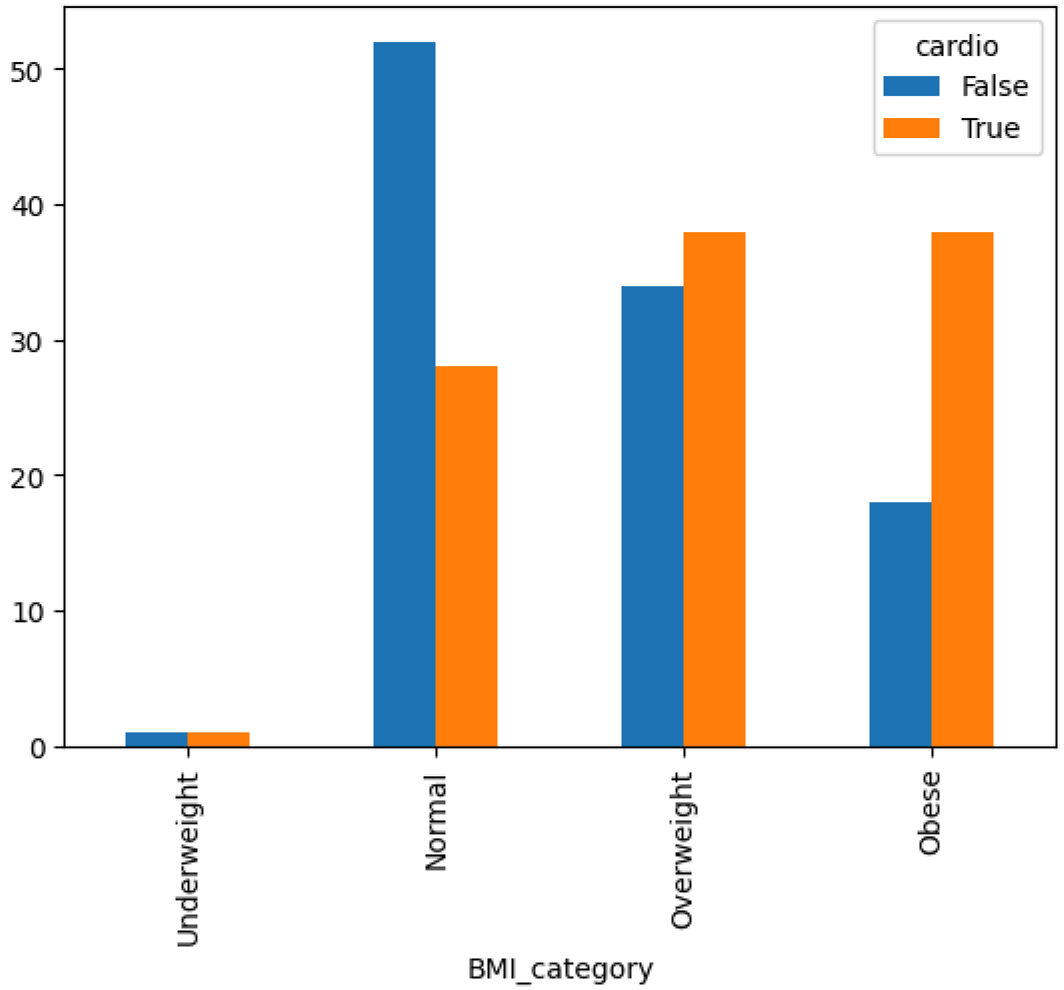
Exercise 5 c) Create a countplot to visualize the distribution of cardio (target variable) across different BMI categories. Here, countplot refers to a type of bar plot that displays the frequency (count) of observations in each category of a categorical variable, visualizing the distribution of data by showing how many instances fall into each category.

In [628…

```
### Your code for 5 c) here ###
bmi_data = data.groupby('BMI_category', observed = True)['cardio'].value_counts

# bmi_data['BMI_category'].plot.bar()
bmi_data.plot.bar()
```

Out[628…

```
<Axes: xlabel='BMI_category'>
```



5 d) Can you notice any relationship or visible trend?

According to the data, those who are Normal weight do markably less cardio, which seems a bit odd in my opinion.

Below, there is ready-made code for you to appropriatly add the newly created features to the right column type list. You don't need to change anything about the code, just make sure that the names of the added features are as specified earlier (**BMI** and **BMI_category**)

In [629…

```python
# ---- Add features to column type list (no need to change) --------
numeric_features.append("BMI")
data['BMI_category'] = data['BMI_category'].astype('category')
categorical_features.append("BMI_category")
```

In [630…

```python
data
```

Out[630…

	age	gender	height	body_mass	blood_pressure_high	blood_pressure_low	sm
0	19797	False	161	55	102	68	F
1	22571	True	178	68	120	70	F
2	16621	True	169	69	120	80	F
3	16688	False	156	77	120	80	F
4	19498	True	170	98	130	80	-
...	...	...	...	...	...	...	
205	16630	False	158	55	120	80	-
206	16742	False	170	68	110	70	F
207	23117	False	157	78	100	60	F
208	15236	False	153	55	120	80	F
209	18043	True	172	78	140	90	F

210 rows × 13 columns

In [631…

```python
numeric_features
```

Out[631…

```
['age',
 'height',
 'body_mass',
 'blood_pressure_high',
 'blood_pressure_low',
 'BMI']
```

6. Preprocessing numeric features

Scaling the data is a crucial step in the preprocessing phase of machine learning, as it can significantly improve algorithm performance. In many cases, if scaling is not applied, it may lead to poor performance. This is particularly true for distance-based algorithms covered in the course, such as PCA, t-SNE, KNN and Kmeans where features with larger values can dominate the distance calculations.

Common Scaling Techniques:

In this exercise, we will explore two commonly used methods for scaling data:

1. **Min-Max Scaling to [0, 1]:**
    - This technique rescales the feature values to a range between 0 and 1. It is particularly useful when you want to maintain the relationships between the values while fitting the data into a specific range. This method is often used in training neural networks, where matching the input range to the range of activation functions is important.
2. **Standardization :**
    - standardizing the features to 0 mean and unit variance. Standardizing values is very common in statistics.

Available Functions:

To assist you in applying these scaling techniques, the following functions from the `sklearn` library have been imported for your use:

- `sklearn.preprocessing.minmax_scale` : For Min-Max Scaling.
- `sklearn.preprocessing.scale` : For Standardization.

6 a) Min-max numeric attributes to [0,1] and **store the results in a new dataframe called data_min_maxed**. You might have to wrap the data to a dataframe again using pd.DataFrame()

In [632…

```
# --- Your code for 6 a) here --- #

data_min_maxed = pd.DataFrame(minmax_scale(data[numeric_features]),
                              columns = numeric_features)
data_min_maxed
```

Out[632…

	age	height	body_mass	blood_pressure_high	blood_pressure_low	BM
0	0.590346	0.358491	0.1250	0.12	0.257143	0.09418
1	0.891933	0.679245	0.2875	0.30	0.285714	0.10182
2	0.245053	0.509434	0.3000	0.30	0.428571	0.18649
3	0.252337	0.264151	0.4000	0.30	0.428571	0.42136
4	0.557839	0.528302	0.6625	0.40	0.428571	0.49261
...	...	...	...	...	...	.
205	0.246032	0.301887	0.1250	0.30	0.428571	0.11971
206	0.258208	0.528302	0.2875	0.20	0.285714	0.16673
207	0.951294	0.283019	0.4125	0.10	0.142857	0.42148
208	0.094477	0.207547	0.1250	0.30	0.428571	0.16566
209	0.399652	0.566038	0.4125	0.50	0.571429	0.25577

210 rows × 6 columns

Exercise 6 b) Standardize the numeric attributes of the dataset to have a mean of 0 and a standard deviation of 1. Store the standardized results in a new DataFrame called `data_standardized`.

In [633…

```
# Your code for 6 b here --- #
data_standardized = pd.DataFrame(scale(data_min_maxed, with_mean = True, with_s
                                 columns = numeric_features)
data_standardized
```

Out[633...

	age	height	body_mass	blood_pressure_high	blood_pressure_low	l
0	0.140926	-0.423185	-1.296193	-1.480325	-1.392016	-1.156
1	1.285684	1.838449	-0.404407	-0.449822	-1.190484	-1.111
2	-1.169727	0.641113	-0.335808	-0.449822	-0.182819	-0.610
3	-1.142077	-1.088372	0.212983	-0.449822	-0.182819	0.778
4	0.017537	0.774151	1.653560	0.122679	-0.182819	1.199
...	...	...	...	...	...	
205	-1.166013	-0.822297	-1.296193	-0.449822	-0.182819	-1.005
206	-1.119793	0.774151	-0.404407	-1.022324	-1.190484	-0.727
207	1.511004	-0.955335	0.281582	-1.594825	-2.198148	0.778
208	-1.741280	-1.487484	-1.296193	-0.449822	-0.182819	-0.733
209	-0.582904	1.040225	0.281582	0.695180	0.824845	-0.200

210 rows × 6 columns

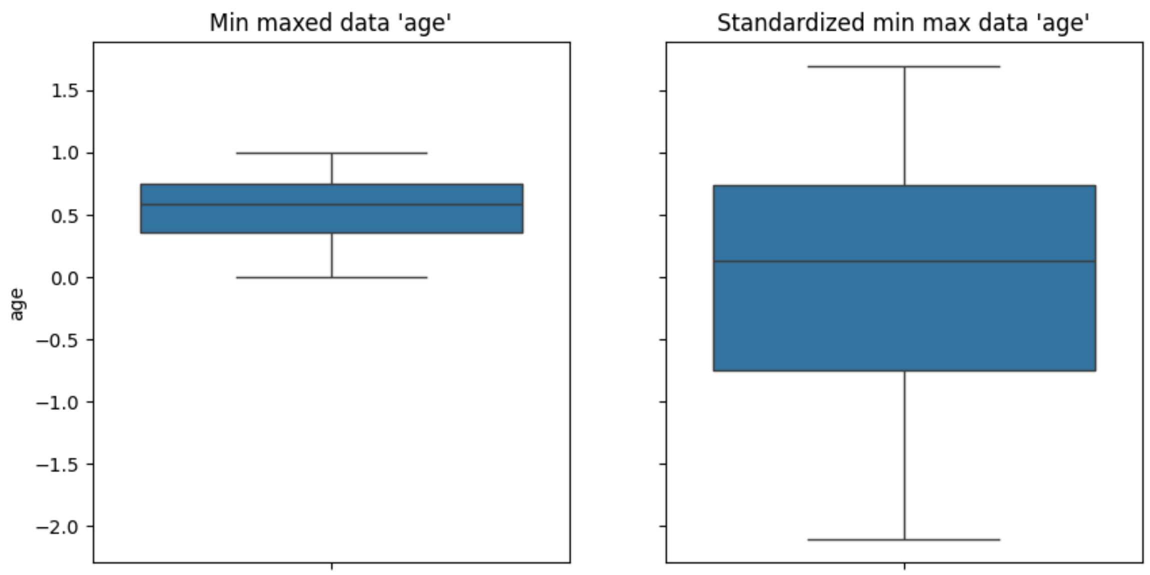
Exercise 6 c) Create two boxplots for the 'age' feature: one using the `data_min_maxed` DataFrame and the other using the `data_standardized` DataFrame. Display the plots side-by-side and provide titles for each plot. See the tutorial in the beginning for help.

In [634...

```python
# Your code for 6 c) here --- #
fig, axes = plt.subplots(1, 2, figsize=(10, 5), sharey=True)

# Boxplot the original data
sns.boxplot(data=data_min_maxed['age'], ax=axes[0])
axes[0].set_title("Min maxed data 'age'")

# Boxplot the standardized data
sns.boxplot(data=data_standardized['age'], ax=axes[1])
axes[1].set_title("Standardized min max data 'age'")
```

Out[634...

```
Text(0.5, 1.0, "Standardized min max data 'age'")
```



Execise 6 d) Describe what you would expect to see in these two boxplots. How would the characteristics of the boxplots differ for min-max scaled data and standardized data?

tip: Consider factors like the location of the mean, and the range of values presented.

Min maxed data should proportionalize the given datapoints giving a generalized view of the population. Vertical line in the box part shows median which is slightly skewed towards higher top 50% of the sample. Maximum values are shoen with vertical lines in top and bottom. They show smallest as 0 and highest as 1 as the min-max produces on default. When standardised the data is distributed on a larger scale visualizing the results better and centering it in the middle of the figure around 0.

Let's compare the effects of these preprocessing methods on a dataset with an outlier. We'll replace the last data point with an outlier (a value significantly different from the rest) and then apply min-max scaling and standardization. Finally, we'll visualize the results to observe how each method handles the outlier. The code to add the value is given for you and you shouldn't change it.

Exercise 6 e) Do the following:

1. **Use the Provided Data:**
    - Start with the given data for the 'age' feature, which includes an outlier. This variable is referred to as `age_w_outlier`. The value of `age_w_outlier` is already set for you, so you don't need to modify it.
2. **Create Min-Max Scaled Variable:**

- Use the `sklearn.preprocessing.minmax_scale` function to apply Min-Max scaling to `age_w_outlier`. Store the scaled values in a new variable named `age_w_outlier_minmaxed`.

3. **Create Standardized Variable:**

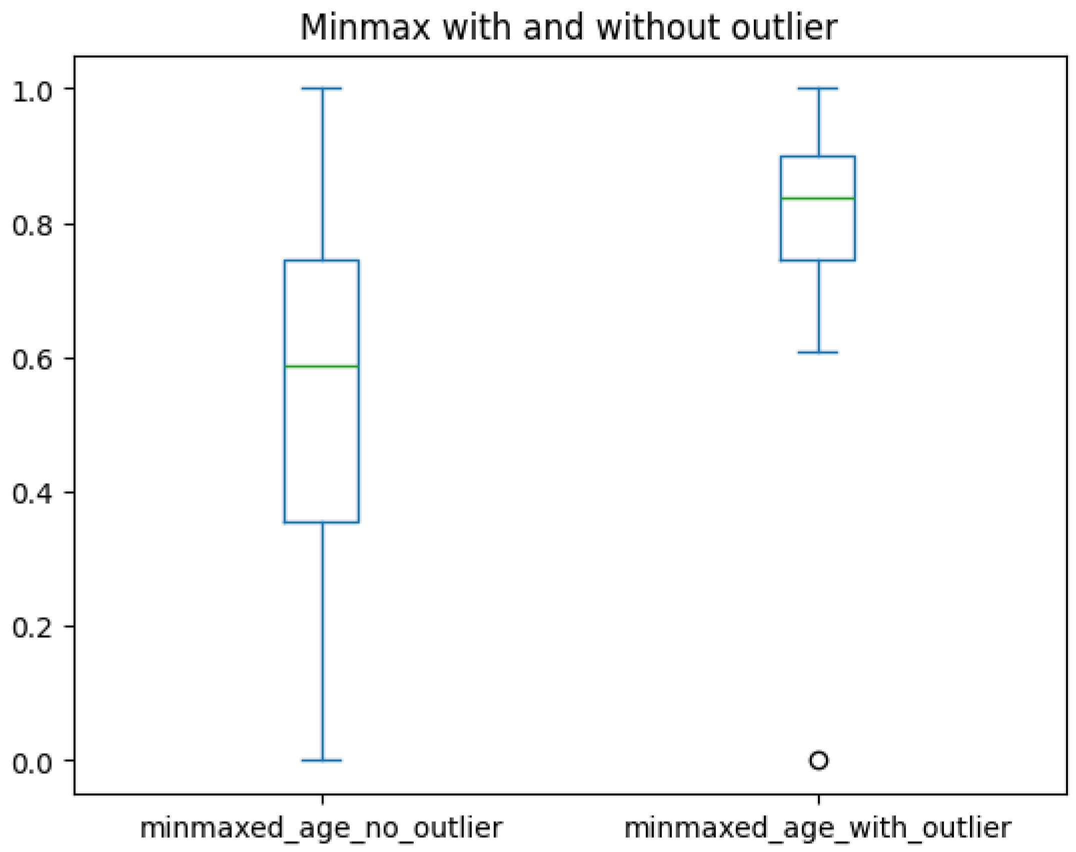
- Use the `sklearn.preprocessing.scale` function to standardize the values of `age_w_outlier`. Store the standardized values in a new variable named `age_w_outlier_standardized`.

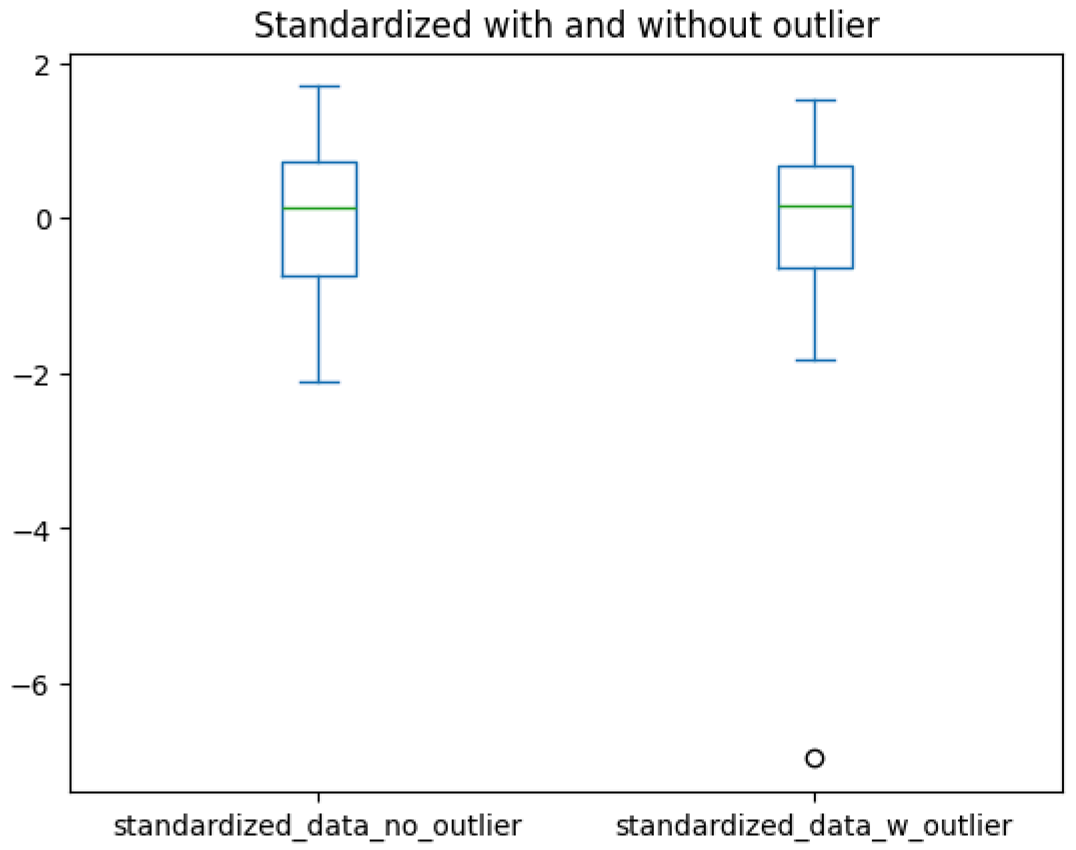
In [635…
```
### Add an outlier, DONT CHANGE THIS CELL CODE, JUST RUN IT ###
data_w_outlier = data.copy() #data should be the name of the variable where you
data_w_outlier.loc[data.shape[0] -1 , 'age'] = 150 #change the last value of ag
age_w_outlier = data_w_outlier.age
```

In [636…
```
# --- Your code for 6 e) ---
# Min max the data with the outlier
age_w_outlier_minmaxed = pd.DataFrame(minmax_scale(data_w_outlier[numeric_featu
                                       columns = numeric_features)
age_w_outlier_standardized = pd.DataFrame(scale(age_w_outlier_minmaxed),
                                   columns = numeric_features)
```

Below there is pre-written code for you to plot the different cases. Run it. The code should run if you have named your features appropriately. Run the code.

In [637…
```
# Wrap in a dataframe that will have two features - the age feature without the
minmaxed_datas = pd.DataFrame({"minmaxed_age_no_outlier" : data_min_maxed.age,
              "minmaxed_age_with_outlier": age_w_outlier_minmaxed.age })

# Wrap in a dataframe that will have two features - the age feature without the
standardized_datas = pd.DataFrame({"standardized_data_no_outlier" : data_standa
              "standardized_data_w_outlier": age_w_outlier_standardized.age })

axes_minmaxed = minmaxed_datas[['minmaxed_age_no_outlier', 'minmaxed_age_with_o
axes_std = standardized_datas[['standardized_data_no_outlier', 'standardized_da
```





Exercise 6 f) Look at the output of the above cell and answer the following:

1. Can you notice a difference between the two cases (min-maxed and standardized)?
2. Can you say something about the difference of the effect of min-maxing and standardization?

6 f)

- Min max includes the outlier into the scale as a min or max value. And because it is an outlier it heavily disrupts the result. It squeezes the valid data to the other extereme end of the distribution.
- Results are quite similiar even with the outlier in the data. This way results are at least good for guidance even before perfect data preparation. Overall distribution is effected and therefore a bit narrower in the figure but still resembles the properly treated data moderately.

---

7. Preprocessing categorical features

We can roughly divide categorical variables/features to two types: ***nominal categorical*** and ***ordinal categorical*** variables/features. Some cases are clear in terms of which of the two a feature falls into. For example nationality is not an ordered feature, but which grade in school someone is has a natural ordering. **One-hot encoding** was presented in the lectures and will be used in the following exercises with different learning methods.

Nominal categorical features need to be encoded, because not encoding them implies that they have an order. For example, consider a dataset where you would have rows by different countries, encoded randomly with numbers, for ex. Finland = 1, Norway = 2 and so on. For some analyses and methods this would imply that Norway is somehow "greater" in value than Finland. For some algorithms, the implication would also be, that some of the countries would be "closer" to each other.

Ordinal categorical features do not necessarily need to be encoded, but there are cases where it can be wise. One case is that the categories are not even distance from each other, which is the case with the 'serum_lipid_level' feature with the levels 'normal', 'elevated' and 'at risk'. Its not clear that these are equal in distance from each other. When unsure, it may also be better to one-hot encode, and a lot of packages do it for you behind the scenes. Here we decide to one-hot encode.

Exercise 7 a) Apply One-hot-encode to the `serum_lipid_level` feature and add the resulting one-hot encoded features back to the DataFrame. Give the new features meaningful names. Print the first rows of the resulting dataframe.

tip: pandas has a function for this, google!

In [638…

```
# --- Your code for 7 a) here ---

# Creating one-hot-encoding for sll
sll_encoded = pd.get_dummies(data['serum_lipid_level'])

# Saving each column, could be done in oneliner
sll_at_risk = sll_encoded['at risk']
sll_elevated = sll_encoded['elevated']
sll_normal = sll_encoded['normal']

# Adding new columns to the data
data['sll_normal'] = sll_normal
data['sll_at_risk'] = sll_at_risk
data['sll_elevated'] = sll_elevated

# First 5 rows
data.head(5)
```

Out[638…

	age	gender	height	body_mass	blood_pressure_high	blood_pressure_low	smok
0	19797	False	161	55	102	68	Fals
1	22571	True	178	68	120	70	Fals
2	16621	True	169	69	120	80	Fals
3	16688	False	156	77	120	80	Fals
4	19498	True	170	98	130	80	Tru

BONUS EXERCISES

- Below are the bonus exercises. You can stop here, and get the "pass" grade.
- By doing both of the bonus exercises below, you can get a "pass with honors", which means you will get one point bonus for the exam.

The following exercises are more challenging and not as straight-forward and may require some research of your own. However, perfect written answers are not required, but answers that show that you have tried to understand the problems and explain them with your own words.

---

8. BONUS: Dimensionality reduction and plotting with PCA

In the lectures, PCA was introduced as a dimensionality reduction technique. Here we will use it to reduce the dimensionality of the numeric features of this dataset and use the resulting compressed view of the dataset to plot it. This means you have to, run PCA and then project the data you used to fit the PCA to the new space, where the principal components are the axes.

Exercise 8 a) Do PCA with two components with and without z-score standardization **for the numeric features in the data**.

In [639…

```
# --- Your for 8 a) code here --- #
data_numeric = data[numeric_features]
# Create a PCA object
pca = PCA(n_components=2)
pca_components = pca.fit_transform(data_numeric)

data_zscore = (data_numeric - data_numeric.mean())/data_numeric.std()
pca_standardized = pca.fit_transform(data_zscore)
```

Exercise 8 b) Plot the data, projected on to the PCA space as a scatterplot, the x-axis being one component and y the other. **Add the total explained variance to your plot as an annotation. See the documentation of the pca method on how to get the explained variance.

- *Tip: It may be easier to try the seaborn scatterplot for this one. For help see documentation on how to do annotation (see tutorial). The total explained variance is the sum of both the components explained variance.*
- *Tip2*: Depending on how you approach annotating the plot, you might have to cast the feature name to be a string. One nice way to format values in python is the f - formatting string, which allows you to insert expressions inside strings (see example below):

---

name = Valtteri
print(f"hello_{name}")

You can also set the number of wanted decimals for floats
For example f'{float_variable:.2f}' would result in 2 decimals making it to the string created

---

In [640…
```
# --- Your code for 8 b) --- you can make more cells if you like ---
```

Exercise 8 c) Gather information for the next part of the exercise and print out the following things:

- First, the standard deviation of the original data features (not standardized, and with the numeric features only).
- Second, the standard deviation of the standardized numeric features

In [641…
```
# --- Your code for 8 c) here --- #
```

Exercise 8 d) Look at the output above and the explained variance information you added as annotations to the plots. Try to think about the following questions and give a short answer of what you think has happened:

1. Where do you think the difference between the amounts of explained variance might come from?

2. Can you say something about why it is important to scale the features for PCA by looking at the evidence youve gathered?

Answer in your own words, here it is not important to get the perfect answer but to try to think and figure out what has happened

---

Your answer for 8 d)

---

9. Bonus: t-SNE and high dimensional data

Another method that can be used to plot high-dimensional data introduced in the lectures was t-distributed Stochastic Neighbor Embedding (t-SNE).

Exercise 9 a) Run t-SNE for both standardized and non standardized data (as you did with PCA).

In [642…
```
# --- Your code for 9 a) here --- #
```

Exercise 9 b) Plot t-sne, similarly to PCA making the color of the points correspond to the levels of the cardio feature, but having only numerical features as a basis of the T-SNE.

In [643…
```
# --- Code for 9 b) --- #
```

Exercise 9 c)

- What do you think might have happened between the two runs of t-SNE on unstandardized and standardized data? Why is it important to standardize before using the algorithm?

Here the aim is to think about this and learn, not come up with a perfect explanation. Googling is encouraged. Think about whether t-sne is a distance based algorithm or not?

Your answer for 9 c)

In [ ]: