lab1_studentfin

January 16, 2024

1 Lab 1: Getting started

The goals of this lab are: 1. Probability warmup 2. Familiarize ourselves with Jupyter notebooks 3. Use Python in the Jupyter Notebook 4. Perform mathmatical computations using the NumPy library 5. Import and manipulate data using the Pandas library 6. Plot data with Matplotlib and Seaborn 7. Practice exploring new datasets and visualizing them

1.1 Probability Practice (Show all work)

1.1.1 Question 1 (2 pts)

My neighbor has two children. Assuming that the gender of a child is like a coin flip, it is most likely, a priori, that my neighbor has one boy and one girl, with probability 1/2. The other possibilities—two boys or two girls—have probabilities 1/4 and 1/4.

a. Suppose I ask him whether he has any boys, and he says yes. What is the probability that one child is a girl?

b. Suppose instead that I happen to see one of his children run by, and it is a boy. What is the probability that the other child is a girl?

Your solution:

Part a

- **Given:** At least one child is a boy.
- **Possible combinations:** {Boy, Boy}, {Boy, Girl}, {Girl, Boy}.
- **Probability of one girl:** 2 out of 3 combinations have a girl, so

$$P(\text{one girl}) = \frac{2}{3}$$

.

Part b

- **Given:** One seen child is a boy.
- **Possible combinations:** {Boy, Boy}, {Boy, Girl}.
- **Probability of the other child being a girl:** 1 out of 2 combinations has a girl, so

- $$P(\text{other child is a girl}) = \frac{1}{2}$$
  .

1.1.2 Question 2 (2 pts)

In 1995, they introduced blue M&M's. Before then, the color mix in a bag of plain M&M's was 30% Brown, 20% Yellow, 20% Red, 10% Green, 10% Orange, 10% Tan. Afterward it was 24% Blue, 20% Green, 16% Orange, 14% Yellow, 13% Red, 13% Brown.

Suppose a friend of mine has two bags of M&M's, and he tells me that one is from 1994 and one from 1996. He won't tell me which is which, but he gives me one M&M from each bag. One is yellow and one is green. What is the probability that the yellow one came from the 1994 bag?

Your solution:

20% chance to draw a yellow from the 1994 bag. 14% chance to draw a yellow from the 1996 bag.

10% chance to draw a green from the 1994 bag. 20% chance to draw a green from the 1996 bag.

- Let (A) be the event that the yellow M&M comes from the 1994 bag.
- Let (B) be the event of getting a yellow M&M from one bag and a green M&M from the other.

Find (P(A|B)), the probability that the yellow M&M came from the 1994 bag given that we have one yellow and one green M&M.

Bayes' theorem states that:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$

1. **(P(B|A))**: This is the probability of getting a yellow M&M from the 1994 bag and a green M&M from the 1996 bag. Since the bags are independent, this is the product of the probabilities of getting a yellow M&M from the 1994 bag and a green M&M from the 1996 bag. From the given data, these probabilities are (20%) for a yellow M&M from the 1994 bag and (20%) for a green M&M from the 1996 bag.

$$P(B|A) = 0.20 * 0.20 = 0.04$$

2. **(P(A))**: This is the prior probability of the yellow M&M coming from the 1994 bag, which is (50%) since naively there is an equal chance of the yellow M&M being from either the 1994 or 1996 bag.

$$P(A) = 0.50$$

3. **(P(B))**: This is the probability of the observed evidence, i.e., getting one yellow and one green M&M. This can happen in two ways: a yellow M&M from the 1994 bag and a green M&M from the 1996 bag, or a yellow M&M from the 1996 bag and a green M&M from the 1994 bag. We add these probabilities:
   - Yellow from 1994 and green from 1996: (20% ×20%).

- Yellow from 1996 and green from 1994: (14% ×10%).

$$P(B) = 0.5 * 0.20 * 0.20 + 0.5 * 0.10 * 0.14 = 0.027$$

Calculate (P(A|B)) using these values.

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)} = \frac{(0.04 * 0.50)}{0.027} = 0.74$$

The probability that the yellow M&M came from the 1994 bag, given that one is yellow and one is green, is approximately **(75%)**.

1.2 Jupyter notebooks

Jupyter notebooks are useful for tinkering because they're very readable, but research projects that require more abstraction (more lines of code, or cyclomatic complexity, or Halstead complexity, or etc.) should be coded in Python scripts using an object-oriented programming paradigm in an integrated development environment with a more full suite of features. For this course, the Lab*n*_student.ipynb notebooks provided are sufficient.

Via a JupyterHub hosted by UCLA's IDRE, your notebooks will be in your own Docker container, where all the libraries you will need for the assignments should be installed. We recommend Chrome for accessing JupyterHub.

Tips: - Save your notebook frequently (CMD+S on Mac.) - Be careful about the order in which you run cells.

1.2.1 Resources:

- Assignment notebooks will contain some useful links.
- Stackoverflow is a great resource for troubleshooting.
- This book contains many good introductions to topics we'll discuss in the class, and it includes Python code written in Jupyter Notebooks.
- Here is a Jupyter Notebook cheatsheet.

1.2.2 Two Useful Commands

The code in the cell below imports a package. You can run the cell by clicking the button at the top of this webpage, or using the shortcuts CTRL/CMD+ENTER or SHIFT+ENTER to run the cell and keep the cell selected or to run the cell then select the next cell, respectively. Also, here is a link to the NumPy documentation.

[2]:
```python
from numpy.testing import assert_array_almost_equal, assert_almost_equal,␣
  ↪assert_array_equal
import numpy as np
```

Using an exclamation point, one can run shell commands inside of a Jupyter Notebook cell. For example, the following cell issues the `pwd` command, which prints the working directory.

1.2.3 Question 3 (1 pt)

Run the `%magic` command, then try out one of the commands listed in the resulting display. Make sure to delete the output cell before submitting the lab due to the significant length.

Your solution:

```
[ ]: %magic
```

1.2.4 Question 4 (1 pt)

It's useful to know where python is located on your computer and what version of python is installed; try out the following terminal commands: `! which python` and `! python --version`

Your solution:

```
[ ]: ! which python
```

```
[5]: ! python --version
```

```
Python 3.10.6
```

1.2.5 Question 5 (1 pt)

1.3 Python Review

1.3.1 Question 6 (2 pts)

First, define a function called `mult` that accepts two variables, multiplies the variables, and returns the product.

Your solution:

```
[6]: def mult(a,b):
         return a*b
```

Next, call `mult` and pass it 5 and 3. Assign the output to a variable named `product`.

Your solution:

```
[7]: product = mult(5,3)
     print(product)
```

```
15
```

Modify `mult` so that, instead of returning the product of two integer or float inputs, it returns the dot product between the vectors defined by two lists or NumPy arrays (*hint:* look at the documentation for the linear algebra methods in NumPy.) With your new function, assign the output of the product between arrays `[1,2,3]` and `[4,5,6]` to a variable valled `dotprod`, then print `dotprod`.

Your solution:

```
[8]: def mult(a,b):
       return np.dot(a,b)
```

```
[9]: a = [1,2,3]
     b = [4,5,6]
     dotprod = mult(a,b)
     print(dotprod)
```

```
32
```

1.3.2 Lists and Arrays

Array algebra is the core of lots of machine learning, and Numpy is the package we'll use to store and operate on arrays. *Note* that one important challenge is minimizing the computational cost of array algebra.

1.3.3 Question 7 (2 pts)

In the cell below, assign a list containing integers spanning the interval between 0 and 9 to a variable named `l`. Use a NumPy method to generate the list; don't type in each integer. *Note* that, if the method you use generates an array, you'll need to convert the array to a list.

Your solution:

```
[10]: l = np.array(range(0,10)).tolist()
      print(l)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Run the cell below to verify that `l` is the desired list.

```
[11]: assert l == [0,1,2,3,4,5,6,7,8,9]
```

Using list comprehension, generate the same list and assign it to a variable called `l2`.

Your solution:

```
[12]: l2 = [x+1 for x in range(-1,9)]
      l2
```

```
[12]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Run the cell below to verify that `l2` is the desired list.

```
[13]: assert l2 == [0,1,2,3,4,5,6,7,8,9]
```

Using the `np.array()` method, assign an array with the same elements as the list `l2` to a variable named `arr` (generate the array using `l2`.)

Your solution:

```
[14]: arr = np.array(12)
      arr
```

```
[14]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Run the cell below to verify.

```
[15]: assert isinstance(arr, np.ndarray)
```

1.3.4 Question 8 (7 pts)

Unlike lists, arrays can have multiple dimensions of data. Non-vector tensors are the bulk of the arrays which appear in machine learning problems. Using a NumPy method, create a 3×5 array called `arr2` with 1 for every element.

Your solution:

```
[16]: arr2 = np.ones((3,5))
      arr2
```

```
[16]: array([[1., 1., 1., 1., 1.],
             [1., 1., 1., 1., 1.],
             [1., 1., 1., 1., 1.]])
```

Verify your solution:

```
[17]: assert_array_equal(arr2,np.ones((3,5)))
```

Another common task in ML is to create arrays filled with random numbers. There are lots of different NumPy methods for generating random numbers! Create a 3×3 array of random integers normally distributed on the interval `[0,10]`, and assign it to a variable called `arr3`.

Your solution:

```
[18]: arr3 = np.random.randint(0,10,(3,3))
      arr3
```

```
[18]: array([[3, 6, 2],
             [9, 8, 9],
             [8, 7, 6]])
```

Verify the shape of your array:

```
[19]: assert arr3.shape == (3,3)
```

Assign the `shape` attribute of `arr3` to a variable called `s`.

Your solution:

```
[20]: s = arr3.shape
      s
```

```
[20]: (3, 3)
```

Verify the type of `s`:

```
[21]: assert isinstance(s, tuple)
```

Assign the number of array dimensions of `arr3` to a variable called `d`. You can find the right method in this guide.

Your solution:

```
[22]: d = arr3.ndim
```

Using array indexing, print the element of `arr3` in the first row and first column and assign it to a variable called `x`.

Your solution:

```
[23]: x = arr3[0,0]
```

```
[24]: assert x == arr3[0,0]
```

Create a variable `e` with the value of the element in the row with the largest index and column with the largest index in `arr3`. *Note* that there is a way to index the last column or row that doesn't depend on the size of the array.

Your solution:

```
[25]: e = arr3[-1,-1] # negative one indexes at last element
```

```
[26]: assert e == arr3[-1,-1]
```

Since datasets are often large in ML contexts, we need to slice arrays and only use a portion of a dataset. One way to slice an array is using the command `array[start_index:stop_index]`. We can also specify a step size, `array[start_index:stop_index:step_size]`. Create a new variable called `y` which contains just the first two rows of `arr3`.

Your solution:

```
[27]: y = arr3[:2]
      y
```

```
[27]: array([[3, 6, 2],
             [9, 8, 9]])
```

```
[28]: assert y.all() == arr3[:2].all()
```

Using NumPy's `arange` and `reshape` methods, create an array called `arr55` so that `print(arr55)` returns this:

```
[[1,2,3,4,5],   [6,7,8,9,10],   [11,12,13,14,15],   [16,17,18,19,20],
[21,22,23,24,25]]
```

Your solution:

[29]:
```python
arr55 = np.arange(1,26).reshape(5,5)
print(arr55)
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]
 [21 22 23 24 25]]
```

[30]:
```python
assert arr55.all() == np.arange(1,26).reshape(5,5).all()
```

1.4 Broadcasting:

Broadcasting, defined by VanderPlas: a set of rules for applying binary ufuncs (e.g., addition, multiplication, etc.) on arrays of different sizes. For arrays of the same size, binary operations are performed on an element-by-element basis. Broadcasting allows these operations to be performed on arrays of different sizes, for example adding a scalar to an array. Here's a brief demonstration of how it works:

Adding two arrays with different shapes:

First, define an array with shape (4,1).

`a = np.array(          [[1.],           [2.],           [3.],           [4.]])`

Note that the periods after the numbers tell Python that the elements should have *float* DataTypes rather than *integer*. Now define another array with shape (3,).

`b = np.array([0., 1., 2.])`

Now we operate on the arrays:

`c = a + b`

And find that

`c = np.array([[1., 2., 3.],        [2., 3., 4.],        [3., 4., 5.],        [4., 5., 6.]])`

Here's what happened inside the NumPy machinery:

NumPy aligned the shapes of a and b on the last axis and prepended 1s to the shape with fewer axes: a: 4 x 1 —> a: 4 x 1 b: 3 —> b: 1 x 3 Then checked that the sizes of the axes matched or were equal to 1:
a: 4 x 1
b: 1 x 3
`a` and `b` satisfied this criterion.
NumPy stretched both arrays on their 1-valued axes so that their shapes matched, then added them together. `a` was replicated 3 times in the second axis, while `b` was replicated 4 times in the first axis. This meant that the addition in the final step was

```
[[1., 1., 1.],            [[0., 1., 2.],    [2., 2., 2.],      +      [0., 1.,
2.],    [3., 3., 3.],              [0., 1., 2.],    [4., 4., 4.]]            [0.,
1., 2.]]
```

Addition was then carried out element-by-element, as you can verify by referring back to the output of the code cell above. This resulted in an output with shape 4 x 3.

1.5 Numpy's broadcasting rule

Broadcasting rules describe how values should be transmitted when the inputs to an operation do not match. In numpy, the broadcasting rule is very simple:

Prepend 1s to the smaller shape, check that the axes of both arrays have sizes that are equal or 1, then stretch the arrays in their size-1 axes.

A crucial aspect of this rule is that it does not require the input arrays have the same number of axes. Another consequence of it is that a broadcasting output will have the largest size of its inputs in each axis. Take the following multiplication as an example:

```
a: 3 x 7 x 1
b:     1 x 5
```

a * b: 3 x 7 x 5 You can see that the output shape is the maximum of the sizes in each axis.

NumPy's broadcasting rule also does not require that one of the arrays has to be bigger in all axes.

Broadcasting behavior also points to an efficient way to compute an outer product in NumPy:

[31]:
```
a = np.array([-1., 0., 1.])
b = np.array([0., 1., 2., 3.])

a[:, np.newaxis] * b  # outer product ab^T, where a and b are column vectors
```

[31]:
```
array([[-0., -1., -2., -3.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  1.,  2.,  3.]])
```

The idea of numpy stretching the arrays in their size-1 axes is useful and is functionally correct. But this is not what numpy literally does behind the scenes, since that would be an inefficient use of memory. Instead, numpy carries out the operation by looping over singleton (size-1) dimensions.

1.5.1 Question 9 (2 pts)

To give you some practise with broadcasting, try predicting the output shapes for the following operations:

[32]:
```
# Run this cell, which defines three arrays with different shapes
a = np.array([[1.], [2.], [3.]])
b = np.zeros(shape=[10, 1, 1])
c = np.ones(shape=[4])
d = np.array([1,2,3,4])
```

What are the shapes of `a`, `b`, `c`, and `d`? Define `d1` as the shape of `d`.

[33]:
```
print(a.shape)
print(b.shape)
print(c.shape)
d1 = d.shape
print(d1)
```

```
(3, 1)
(10, 1, 1)
(4,)
(4,)
```

[34]:
```
assert d1==(4,)
```

Multiply two arrays with differen shapes:

[35]:
```
a = np.array([[[0.01], [0.1]],
              [[1.00], [10.]]])  # what shape is this? it's (2, 2, 1)
b = np.array([[[2., 2.]],
              [[3., 3.]]])       # what shape is this? it's (2, 1, 2)

c = a * b
```

What is the shape of `c`? Type the answer below and assign to `v`.

[36]:
```
v = (2,2,2)
```

[37]:
```
assert v == (2,2,2)
```

1.6 Pandas:

Pandas is a Python library built on Numpy. It defines Series and DataFrame objects, which are multi-dimensional arrays with row and column labels attached naturally. They are good with various data types and missing data. This is a link to the Pandas documentation.

[38]:
```
import pandas as pd
```

A Pandas Series object is a one dimensional array of indexed data. Such an object can be created from a list in a number of ways, and so can DataFrames. We will try to introduce the various ways; they'll be confusing for now, but it gets easier with more exposure; you will start to see the different, equivalent ways of performing ML and data science tasks.

1.6.1 Question 10 (1 pt)

[39]:
```
data = pd.Series([0.30, 0.5, 0.78, 1.0])
```

In the cells below, try data.values and data.index, and try to index the Series using `data[2]`.

[40]:
```
data.values
```

[40]:
```
array([0.3 , 0.5 , 0.78, 1.  ])
```

[41]:
```
data.index
```

[41]:
```
RangeIndex(start=0, stop=4, step=1)
```

[42]:
```
data[2]
```

[42]:
```
0.78
```

1.6.2 DataFrames

1.6.3 Question 11 (8 pts)

You can think of a Pandas DataFrame object as a sequence of aligned Series objects (aligned in the sense that they share the same index.) These are some ways you can construct DataFrames: from a single Series object, from a list of dictionaries, from a dictionary of Series objects, from a 2-d numpy array, from a Numpy Structured array, or by importing some data (for example, a CSV file). If you know SQL, DataFrames are designed to allow for different joins of datasets, and lots of other functionalities.

Next, we are going to load a dataset and perform Pandas operations on the dataset.

We'll learn how to calculate statstics on datasets and how to renaming columns, add rows, drop columns, combine columns, create new columns out of existing ones, locate information with queries, filter, etc.

[43]:
```
# (1 pt.)

import pandas as pd
import numpy as np
from sklearn.utils import Bunch
data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
target = raw_df.values[1::2, 2]
feature_names = np.array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE',␣
 ↪'DIS', 'RAD',
        'TAX', 'PTRATIO', 'B', 'LSTAT'])
boston = Bunch(data=data, target=target, feature_names=feature_names)

# in the space below, print the datatype of boston.
### BEGIN SOLUTION
print(type(boston))
### END SOLUTION
```

```
<class 'sklearn.utils._bunch.Bunch'>
```

Print the datatype of `boston` in the cell below.

[44]:
```python
print(boston) # just takin a peek :)
```

```
{'data': array([[6.3200e-03, 1.8000e+01, 2.3100e+00, …, 1.5300e+01,
3.9690e+02,
         4.9800e+00],
        [2.7310e-02, 0.0000e+00, 7.0700e+00, …, 1.7800e+01, 3.9690e+02,
         9.1400e+00],
        [2.7290e-02, 0.0000e+00, 7.0700e+00, …, 1.7800e+01, 3.9283e+02,
         4.0300e+00],
        …,
        [6.0760e-02, 0.0000e+00, 1.1930e+01, …, 2.1000e+01, 3.9690e+02,
         5.6400e+00],
        [1.0959e-01, 0.0000e+00, 1.1930e+01, …, 2.1000e+01, 3.9345e+02,
         6.4800e+00],
        [4.7410e-02, 0.0000e+00, 1.1930e+01, …, 2.1000e+01, 3.9690e+02,
         7.8800e+00]]), 'target': array([24. , 21.6, 34.7, 33.4, 36.2, 28.7,
22.9, 27.1, 16.5, 18.9, 15. ,
       18.9, 21.7, 20.4, 18.2, 19.9, 23.1, 17.5, 20.2, 18.2, 13.6, 19.6,
       15.2, 14.5, 15.6, 13.9, 16.6, 14.8, 18.4, 21. , 12.7, 14.5, 13.2,
       13.1, 13.5, 18.9, 20. , 21. , 24.7, 30.8, 34.9, 26.6, 25.3, 24.7,
       21.2, 19.3, 20. , 16.6, 14.4, 19.4, 19.7, 20.5, 25. , 23.4, 18.9,
       35.4, 24.7, 31.6, 23.3, 19.6, 18.7, 16. , 22.2, 25. , 33. , 23.5,
       19.4, 22. , 17.4, 20.9, 24.2, 21.7, 22.8, 23.4, 24.1, 21.4, 20. ,
       20.8, 21.2, 20.3, 28. , 23.9, 24.8, 22.9, 23.9, 26.6, 22.5, 22.2,
       23.6, 28.7, 22.6, 22. , 22.9, 25. , 20.6, 28.4, 21.4, 38.7, 43.8,
       33.2, 27.5, 26.5, 18.6, 19.3, 20.1, 19.5, 19.5, 20.4, 19.8, 19.4,
       21.7, 22.8, 18.8, 18.7, 18.5, 18.3, 21.2, 19.2, 20.4, 19.3, 22. ,
       20.3, 20.5, 17.3, 18.8, 21.4, 15.7, 16.2, 18. , 14.3, 19.2, 19.6,
       23. , 18.4, 15.6, 18.1, 17.4, 17.1, 13.3, 17.8, 14. , 14.4, 13.4,
       15.6, 11.8, 13.8, 15.6, 14.6, 17.8, 15.4, 21.5, 19.6, 15.3, 19.4,
       17. , 15.6, 13.1, 41.3, 24.3, 23.3, 27. , 50. , 50. , 50. , 22.7,
       25. , 50. , 23.8, 23.8, 22.3, 17.4, 19.1, 23.1, 23.6, 22.6, 29.4,
       23.2, 24.6, 29.9, 37.2, 39.8, 36.2, 37.9, 32.5, 26.4, 29.6, 50. ,
       32. , 29.8, 34.9, 37. , 30.5, 36.4, 31.1, 29.1, 50. , 33.3, 30.3,
       34.6, 34.9, 32.9, 24.1, 42.3, 48.5, 50. , 22.6, 24.4, 22.5, 24.4,
       20. , 21.7, 19.3, 22.4, 28.1, 23.7, 25. , 23.3, 28.7, 21.5, 23. ,
       26.7, 21.7, 27.5, 30.1, 44.8, 50. , 37.6, 31.6, 46.7, 31.5, 24.3,
       31.7, 41.7, 48.3, 29. , 24. , 25.1, 31.5, 23.7, 23.3, 22. , 20.1,
       22.2, 23.7, 17.6, 18.5, 24.3, 20.5, 24.5, 26.2, 24.4, 24.8, 29.6,
       42.8, 21.9, 20.9, 44. , 50. , 36. , 30.1, 33.8, 43.1, 48.8, 31. ,
       36.5, 22.8, 30.7, 50. , 43.5, 20.7, 21.1, 25.2, 24.4, 35.2, 32.4,
       32. , 33.2, 33.1, 29.1, 35.1, 45.4, 35.4, 46. , 50. , 32.2, 22. ,
       20.1, 23.2, 22.3, 24.8, 28.5, 37.3, 27.9, 23.9, 21.7, 28.6, 27.1,
       20.3, 22.5, 29. , 24.8, 22. , 26.4, 33.1, 36.1, 28.4, 33.4, 28.2,
       22.8, 20.3, 16.1, 22.1, 19.4, 21.6, 23.8, 16.2, 17.8, 19.8, 23.1,
       21. , 23.8, 23.1, 20.4, 18.5, 25. , 24.6, 23. , 22.2, 19.3, 22.6,
       19.8, 17.1, 19.4, 22.2, 20.7, 21.1, 19.5, 18.5, 20.6, 19. , 18.7,
       32.7, 16.5, 23.9, 31.2, 17.5, 17.2, 23.1, 24.5, 26.6, 22.9, 24.1,
       18.6, 30.1, 18.2, 20.6, 17.8, 21.7, 22.7, 22.6, 25. , 19.9, 20.8,
       16.8, 21.9, 27.5, 21.9, 23.1, 50. , 50. , 50. , 50. , 50. , 13.8,
       13.8, 15. , 13.9, 13.3, 13.1, 10.2, 10.4, 10.9, 11.3, 12.3,  8.8,
        7.2, 10.5,  7.4, 10.2, 11.5, 15.1, 23.2,  9.7, 13.8, 12.7, 13.1,
       12.5,  8.5,  5. ,  6.3,  5.6,  7.2, 12.1,  8.3,  8.5,  5. , 11.9,
       27.9, 17.2, 27.5, 15. , 17.2, 17.9, 16.3,  7. ,  7.2,  7.5, 10.4,
        8.8,  8.4, 16.7, 14.2, 20.8, 13.4, 11.7,  8.3, 10.2, 10.9, 11. ,
        9.5, 14.5, 14.1, 16.1, 14.3, 11.7, 13.4,  9.6,  8.7,  8.4, 12.8,
       10.5, 17.1, 18.4, 15.4, 10.8, 11.8, 14.9, 12.6, 14.1, 13. , 13.4,
       15.2, 16.1, 17.8, 14.9, 14.1, 12.7, 13.5, 14.9, 20. , 16.4, 17.7,
       19.5, 20.2, 21.4, 19.9, 19. , 19.1, 19.1, 20.1, 19.9, 19.6, 23.2,
       29.8, 13.8, 13.3, 16.7, 12. , 14.6, 21.4, 23. , 23.7, 25. , 21.8,
       20.6, 21.2, 19.1, 20.6, 15.2,  7. ,  8.1, 13.6, 20.1, 21.8, 24.5,
       23.1, 19.7, 18.3, 21.2, 17.5, 16.8, 22.4, 20.6, 23.9, 22. , 11.9]),
'feature_names': array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE',
'DIS', 'RAD',
       'TAX', 'PTRATIO', 'B', 'LSTAT'], dtype='<U7')}
```

[45]:
```python
import sklearn as sk
assert isinstance(boston, sk.utils.Bunch) #bunch file has dicts of labels,␣
 ↪separate target, etc.
```

(1pt.) What is a bunch object? What is it made up of? Answer in the next cell.

a bunch object is like a container which functions similarly to a dictionary but also allows you to access its values as if they were attributes of an object. This makes it a flexible and user-friendly way to store and access data.

A bunch object is made up of, data (features/independent variables), targets (dependent variables), feature names (labels for features), other attributes like metadata.

(1pt.) Transform just boston.data into a Pandas DataFrame (not the other items in the bunch) and name the dataframe `Xbos`. *Hint 1*: you want to assign the columns to be boston.feature_names. *Hint 2*: don't worry about the y value (the target,) we won't be using that for this exercise.

[46]:
```python
Xbos = pd.DataFrame(boston.data, columns=boston.feature_names)
```

[47]:
```python
from pandas.testing import assert_frame_equal
assert_frame_equal (Xbos, pd.DataFrame(boston.data, columns = boston.
 ↪feature_names))
```

(1pt.) Check out your dataframe! in the space below, try `Xbos.head()`, `Xbos['insert_col_name'].value_counts()`, `Xbos.values`, `Xbos.std()`, and `Xbos.mean()`. Refer back to the documentation in sklearn to understand the column values.

[48]:
```python
Xbos.head()
```

```
[48]:       CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
      0  0.00632  18.0   2.31   0.0  0.538  6.575  65.2  4.0900  1.0  296.0
      1  0.02731   0.0   7.07   0.0  0.469  6.421  78.9  4.9671  2.0  242.0
      2  0.02729   0.0   7.07   0.0  0.469  7.185  61.1  4.9671  2.0  242.0
      3  0.03237   0.0   2.18   0.0  0.458  6.998  45.8  6.0622  3.0  222.0
      4  0.06905   0.0   2.18   0.0  0.458  7.147  54.2  6.0622  3.0  222.0

         PTRATIO       B  LSTAT
      0     15.3  396.90   4.98
      1     17.8  396.90   9.14
      2     17.8  392.83   4.03
      3     18.7  394.63   2.94
      4     18.7  396.90   5.33
```

```
[49]: Xbos['CRIM'].value_counts()
```

```
[49]: CRIM
      0.01501     2
      14.33370    2
      0.03466     1
      0.03113     1
      0.03049     1
                 ..
      1.51902     1
      1.83377     1
      1.46336     1
      1.27346     1
      0.04741     1
      Name: count, Length: 504, dtype: int64
```

```
[50]: Xbos.values
```

```
[50]: array([[6.3200e-03, 1.8000e+01, 2.3100e+00, …, 1.5300e+01, 3.9690e+02,
              4.9800e+00],
             [2.7310e-02, 0.0000e+00, 7.0700e+00, …, 1.7800e+01, 3.9690e+02,
              9.1400e+00],
             [2.7290e-02, 0.0000e+00, 7.0700e+00, …, 1.7800e+01, 3.9283e+02,
              4.0300e+00],
             …,
             [6.0760e-02, 0.0000e+00, 1.1930e+01, …, 2.1000e+01, 3.9690e+02,
              5.6400e+00],
             [1.0959e-01, 0.0000e+00, 1.1930e+01, …, 2.1000e+01, 3.9345e+02,
              6.4800e+00],
             [4.7410e-02, 0.0000e+00, 1.1930e+01, …, 2.1000e+01, 3.9690e+02,
              7.8800e+00]])
```

```
[51]: Xbos.std()
```

```
[51]: CRIM         8.601545
      ZN          23.322453
      INDUS        6.860353
      CHAS         0.253994
      NOX          0.115878
      RM           0.702617
      AGE         28.148861
      DIS          2.105710
      RAD          8.707259
      TAX        168.537116
      PTRATIO      2.164946
      B           91.294864
      LSTAT        7.141062
      dtype: float64
```

```
[52]: Xbos.mean()
```

```
[52]: CRIM         3.613524
      ZN          11.363636
      INDUS       11.136779
      CHAS         0.069170
      NOX          0.554695
      RM           6.284634
      AGE         68.574901
      DIS          3.795043
      RAD          9.549407
      TAX        408.237154
      PTRATIO     18.455534
      B          356.674032
      LSTAT       12.653063
      dtype: float64
```

```
[53]: type(Xbos)
```

```
[53]: pandas.core.frame.DataFrame
```

Now we are going to manipulate our dataframe Xbos. Keep in mind when querying and changing– are you copying the dataframe or altering the actual dataframe, or getting a 'view' of a dataframe without actually altering it. When reading the Pandas documentation- think about ways to be careful here!

Let's start with selecting data with loc and iloc. (This is a common interview question for data science jobs/internships, to explain the difference)

This is a pretty decent set of examples to show the difference between loc and iloc: https://thispointer.com/select-rows-columns-by-name-or-index-in-dataframe-using-loc-iloc-python-pandas/

loc: loc is label-based, which means that we have to specify the name of the rows and columns that we need to filter out.

iloc: iloc is integer index-based. So here, we have to specify rows and columns by their integer index.

Starting with loc, try: Xbos.loc where CRIM >= 45. assign this to the variable cri. Answer in the cell below.

[54]:
```
cri = Xbos.loc[Xbos['CRIM'] >= 45] #filters for rows where value of CRIM␣
↪collumn is >= 45

cri.info
```

[54]:
```
<bound method DataFrame.info of          CRIM   ZN  INDUS  CHAS    NOX     RM
AGE     DIS   RAD    TAX
380  88.9762  0.0   18.1   0.0  0.671  6.968   91.9  1.4165  24.0  666.0  \
405  67.9208  0.0   18.1   0.0  0.693  5.683  100.0  1.4254  24.0  666.0
410  51.1358  0.0   18.1   0.0  0.597  5.757  100.0  1.4130  24.0  666.0
414  45.7461  0.0   18.1   0.0  0.693  4.519  100.0  1.6582  24.0  666.0
418  73.5341  0.0   18.1   0.0  0.679  5.957  100.0  1.8026  24.0  666.0

     PTRATIO       B  LSTAT
380     20.2  396.90  17.21
405     20.2  384.97  22.98
410     20.2    2.60  10.11
414     20.2   88.27  36.98
418     20.2   16.45  20.62  >
```

[55]:
```
assert_frame_equal (cri , Xbos.loc[Xbos.CRIM >= 45])
```

(1 pt.) Now try one with multiple conditions, CRIM >= 45 and AGE < 100 and assign to b

[56]:
```
b = Xbos.loc[(Xbos.CRIM >= 45) & (Xbos.AGE < 100)]

b.info
```

[56]:
```
<bound method DataFrame.info of          CRIM   ZN  INDUS  CHAS    NOX     RM
AGE     DIS   RAD    TAX
380  88.9762  0.0   18.1   0.0  0.671  6.968  91.9  1.4165  24.0  666.0  \

     PTRATIO      B  LSTAT
380     20.2  396.9  17.21  >
```

[57]:
```
assert_frame_equal (b, Xbos.loc[(Xbos.CRIM >= 45) & (Xbos.AGE < 100)] )
```

You can also use loc to update the values of a particular column(s) on selected row(s), for example you have a dataframe called data and you have rows of information about individuals. You want to select all people over the age of 18, and then update the column 'age_level' with an 'A' (for adult). you would do this by data.loc[(data.age >= 18), [‘age_level’]] = ‘A’ (Hopefully you were wondering, why are the dot method and bracket method used here? Read on for a little explanation here: https://stackoverflow.com/questions/41130255/what-is-the-difference-between-using-squared-brackets-or-dot-to-access-a-column)

(1 pt.) Take your Xbos, find all instances where AGE >80, and update the TAX column to 245.0. Check to see it worked.

[58]:
```
Xbos.loc[Xbos['AGE'] > 80, 'TAX'] = 245.0
```

Show your update worked by printing out your previous query below.

[59]:
```
Xbos.loc[(Xbos.CRIM >= 45) & (Xbos.AGE < 100)]
```

[59]:
```
        CRIM   ZN  INDUS  CHAS    NOX     RM   AGE     DIS   RAD    TAX  \
380  88.9762  0.0   18.1   0.0  0.671  6.968  91.9  1.4165  24.0  245.0

     PTRATIO      B  LSTAT
380     20.2  396.9  17.21
```

Other necessary pandas tasks are to drop columns, copy columns to make new dataframes, and append columns into a table. To copy a column from Xbos into its own dataframe, you can do this: newbos = Xbos.TAX.copy() Then you can drop the TAX column from Xbos: Xbos.drop([‘TAX’], axis =1) and add it back again. Xbos.append(newbos) You will need to do this particularly if your datasets have the target cols included in the dataframe, or separate, just be aware of this! On this, you can merge series and dataframes, concatenate, do types of joins, and so on.

Pandas is great for dealing with missing data. You can remove rows or cols with missing data, or impute them to other values such as the mean or mode.

(1pt.) Run a query that finds NANs in the entire dataset Xbos, both rows and columns. Assign to the variable r and print results.

[60]:
```
r = Xbos.isna().any()
print(r)
```

```
CRIM       False
ZN         False
INDUS      False
CHAS       False
NOX        False
RM         False
AGE        False
DIS        False
RAD        False
TAX        False
PTRATIO    False
B          False
LSTAT      False
dtype: bool
```

Let's add a row to Xbos. There are many ways to do this. Create a Pandas series and call it new_row.

[61]:
```
import pandas as pd

new_row = pd.Series({name: 0.0 for name in Xbos.columns})
new_row
```

[61]:
```
CRIM       0.0
ZN         0.0
INDUS      0.0
CHAS       0.0
NOX        0.0
RM         0.0
AGE        0.0
DIS        0.0
RAD        0.0
TAX        0.0
PTRATIO    0.0
B          0.0
LSTAT      0.0
dtype: float64
```

Append row to the dataframe

[62]:
```
new_row_df = pd.DataFrame([new_row])  # Convert the Series to a DataFrame
Xbos = pd.concat([Xbos, new_row_df], ignore_index=True)  # Concatenate along␣
 ↪rows
```

[63]:
```
Xbos = Xbos[:-1]
```

[64]:
```
Xbos.tail()
```

[64]:
```
        CRIM   ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
501  0.06263  0.0  11.93   0.0  0.573  6.593  69.1  2.4786  1.0  273.0
502  0.04527  0.0  11.93   0.0  0.573  6.120  76.7  2.2875  1.0  273.0
503  0.06076  0.0  11.93   0.0  0.573  6.976  91.0  2.1675  1.0  245.0
504  0.10959  0.0  11.93   0.0  0.573  6.794  89.3  2.3889  1.0  245.0
505  0.04741  0.0  11.93   0.0  0.573  6.030  80.8  2.5050  1.0  245.0

     PTRATIO       B  LSTAT
501     21.0  391.99   9.67
502     21.0  396.90   9.08
503     21.0  396.90   5.64
504     21.0  393.45   6.48
505     21.0  396.90   7.88
```

Let's add a column of data to Xbos, and give it a default value of 0.

```
[65]: Xbos['Address'] = 0
      Xbos.tail()
```

```
[65]:        CRIM   ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
      501  0.06263  0.0  11.93   0.0  0.573  6.593  69.1  2.4786  1.0  273.0
      502  0.04527  0.0  11.93   0.0  0.573  6.120  76.7  2.2875  1.0  273.0
      503  0.06076  0.0  11.93   0.0  0.573  6.976  91.0  2.1675  1.0  245.0
      504  0.10959  0.0  11.93   0.0  0.573  6.794  89.3  2.3889  1.0  245.0
      505  0.04741  0.0  11.93   0.0  0.573  6.030  80.8  2.5050  1.0  245.0

           PTRATIO       B  LSTAT  Address
      501     21.0  391.99   9.67        0
      502     21.0  396.90   9.08        0
      503     21.0  396.90   5.64        0
      504     21.0  393.45   6.48        0
      505     21.0  396.90   7.88        0
```

1.7 Visualizing and Examining data: the Iris Dataset

In this section, you will start using the scikit-learn Python package and examine a classic dataset used for machine learning. This section will give you practice in thinking about data, which is the often the most important aspect of machine learning.

Now we are going to import the Iris dataset right from sklearn. There are other versions available as well on the web– keep that in mind, some vary slightly or are subsets of a larger set! We are going to load the dataset, and examine its properties (you may have to search these pandas commands). We will then look at some preprocessing and visualizations.

```
[66]: from sklearn import datasets # check these datasets out for built ins
      # We are going to load the famous iris dataset.
```

```
[67]: import matplotlib.pyplot as plt
      from mpl_toolkits.mplot3d import Axes3D
      from sklearn.decomposition import PCA
      import numpy as np
      import pandas as pd
      # import some data to play with
      iris = datasets.load_iris()
      #type(iris)

      data1 = pd.DataFrame(data= np.c_[iris['data'], iris['target']],
                           columns= iris['feature_names'] + ['target'])
      # Notice  np.c_  What is this doing? type np.c_? for help.
```

1.7.1 Question 12 (6 pts)

In the cells below, look at data1. Answer each of these questions in a separate cell, using numpy and pandas functions.

Use markdown cells to answer the written questions.

1. General written question: Describe in a few sentences what this data is. You can get it from the actual set, or you can look it up online, either way is fine.
2. What is the data types?
3. What are the dimensions?
4. Is there any missing data?
5. What is the statistical overview? (mean, std etc.)
6. General written question: What is the target column? What can it be used for?

Use the code cells below to get your answers and type in your answers in the markdown cell after that.

[68]:
```
data1.head()
```

[68]:
```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

   target
0     0.0
1     0.0
2     0.0
3     0.0
4     0.0
```

1. The classic Iris dataset is a toy dataset for learning. It contains data about species of Iris flowers. Each species is represented by 50 samples, making a total of 150 data points. For each sample, the dataset includes four features (seen in the above preview): sepal length, sepal width, petal length, and petal width, all measured in centimeters. The dataset looks to be for classification task practice.

[69]:
```
data1_dtypes = data1.dtypes
print(data1_dtypes)
```

```
sepal length (cm)    float64
sepal width (cm)     float64
petal length (cm)    float64
petal width (cm)     float64
target               float64
dtype: object
```

2. The data in each collumn is of dtype, float64.

[70]:
```
data1.shape
```

[70]:
```
(150, 5)
```

3. The dataset consists of 150 rows of 5 columns.

[71]:
```
missing_data = data1.isna().any()
print(missing_data)
```

```
sepal length (cm)    False
sepal width (cm)     False
petal length (cm)    False
petal width (cm)     False
target               False
dtype: bool
```

4. There are no missing data values in any of the 150 samples.

[72]:
```
data1.describe()
```

[72]:
```
       sepal length (cm)  sepal width (cm)  petal length (cm)  \
count         150.000000        150.000000         150.000000
mean            5.843333          3.057333           3.758000
std             0.828066          0.435866           1.765298
min             4.300000          2.000000           1.000000
25%             5.100000          2.800000           1.600000
50%             5.800000          3.000000           4.350000
75%             6.400000          3.300000           5.100000
max             7.900000          4.400000           6.900000

       petal width (cm)      target
count        150.000000  150.000000
mean           1.199333    1.000000
std            0.762238    0.819232
min            0.100000    0.000000
25%            0.300000    0.000000
50%            1.300000    1.000000
75%            1.800000    2.000000
max            2.500000    2.000000
```

5. The mean and standard deviations for each feature are summarized in the table above along with the min, max, and quartiles.

[73]:
```
data1['target'].unique()
```

[73]:
```
array([0., 1., 2.])
```

6. There are three target values, 0, 1, and 2. The targets can be used as labels for a classifier and in this context they represent the species of Iris that the indexed belongs to.

You *could* rename colums as data1.columns = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width'] What might be wrong with this way of renaming the columns?

If the list you provide doesn't exactly match the number of columns in the DataFrame, it will result in an error. In this case, the DataFrame data1 has five columns, but the list has four names. The mismatch will cause an error.

1.7.2 Correlation between different features

There's lots to think about when preprocessing a dataset. We'll be going over methods in each of the labs. The first thing you might want to do is look at the correlation between two features. The Pearson correlation coefficient (named for Karl Pearson) can be used to summarize the strength of the linear relationship between two data samples.

The Pearson's correlation coefficient is calculated as the covariance of the two variables divided by the product of the standard deviation of each data sample. It is the normalization of the covariance between the two variables to give an interpretable score. In statistics, the Pearson correlation coefficient, also referred to as Pearson's r, the Pearson product-moment correlation coefficient (PPMCC), or the bivariate correlation, is a statistic that measures linear correlation between two variables X and Y. It has a value between +1 and −1. A value of +1 is total positive linear correlation, 0 is no linear correlation, and −1 is total negative linear correlation.

In our case, X and Y are two columns of our Iris dataset. While numpy, scipy, and sklearn have methods to calculate correlations of features, we are going to do this "by hand". Use only numpy! Do not use any corr() built in functions!

Bear with me, this may seem pointless or tedious but it is not! The reason is you will be calculating a formula with vectors– this is definitely something you will see in machine learning and will get you familiar with using numpy.

to calculate r, this is the formula you will translate into numpy:

$$r = \frac{n\sum(XY) - (\sum(X)(\sum(Y)}{\sqrt{[n\sum X^2 - (\sum X)^2][n\sum Y^2 - (\sum Y)^2]}}$$

Where X and Y are feature columns (i.e. sepal width and sepal length) and n is the number of elements in each column.

1.8 Question 13 (3 pts)

Calculate the Pearson correlations for your Iris dataset in the cell below. (Many ways to do this, if you want to get fancy for practice go for it). For this correlation question, feel free to use more than one cell, just make it clear.

[74]:
```python
def PPMCC(X, Y):
    # Ensure X and Y are numpy arrays
    X = np.array(X)
    Y = np.array(Y)

    n = len(X)

    # Calculate correlation coefficient
    numerator = n * np.sum(X * Y) - (np.sum(X) * np.sum(Y))
    denominator = np.sqrt((n * np.sum(X**2) - np.sum(X)**2) * (n * np.sum(Y**2)␣
 ↪- np.sum(Y)**2))
    r = numerator / denominator

    return r
```

[78]:
```
# Exclude 'target' column
features = data1.columns.drop('target')

# Calculate and print the Pearson correlation coefficient for each pair of␣
 ↪features
for i in range(len(features)):
    for j in range(i+1, len(features)):
        r = PPMCC(data1[features[i]], data1[features[j]])
        print(f"Pearson correlation between {features[i]} and {features[j]}: {r:
 ↪.2f}")
```

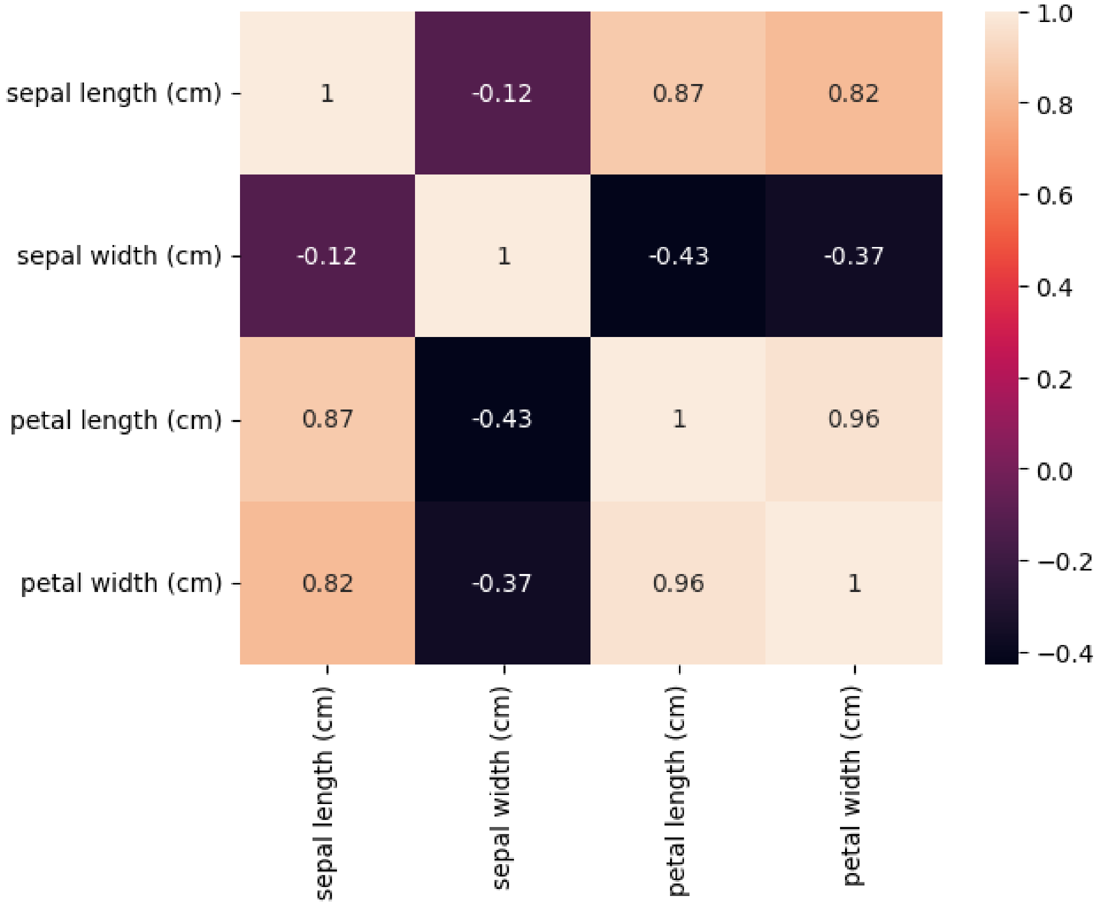
```
Pearson correlation between sepal length (cm) and sepal width (cm): -0.12
Pearson correlation between sepal length (cm) and petal length (cm): 0.87
Pearson correlation between sepal length (cm) and petal width (cm): 0.82
Pearson correlation between sepal width (cm) and petal length (cm): -0.43
Pearson correlation between sepal width (cm) and petal width (cm): -0.37
Pearson correlation between petal length (cm) and petal width (cm): 0.96
```

1.8.1 Question 14 (1 pt)

What are the two most correlated features?

To help answer this we can calculate the correlation matrix and create a map of the correlations between features.

[76]:
```
# Here, we show you a seaborn heatmap. It is a matrix of the correlations␣
 ↪between features. It uses
# color to illustrate intensity of correlations. This will become very useful␣
 ↪when looking at what features
# are imporant for training your models.
import seaborn as sns
data2 = data1.drop('target', axis=1)
sns.heatmap(data2.corr(), annot = True);
#annot = True adds the numbers onto the squares
```



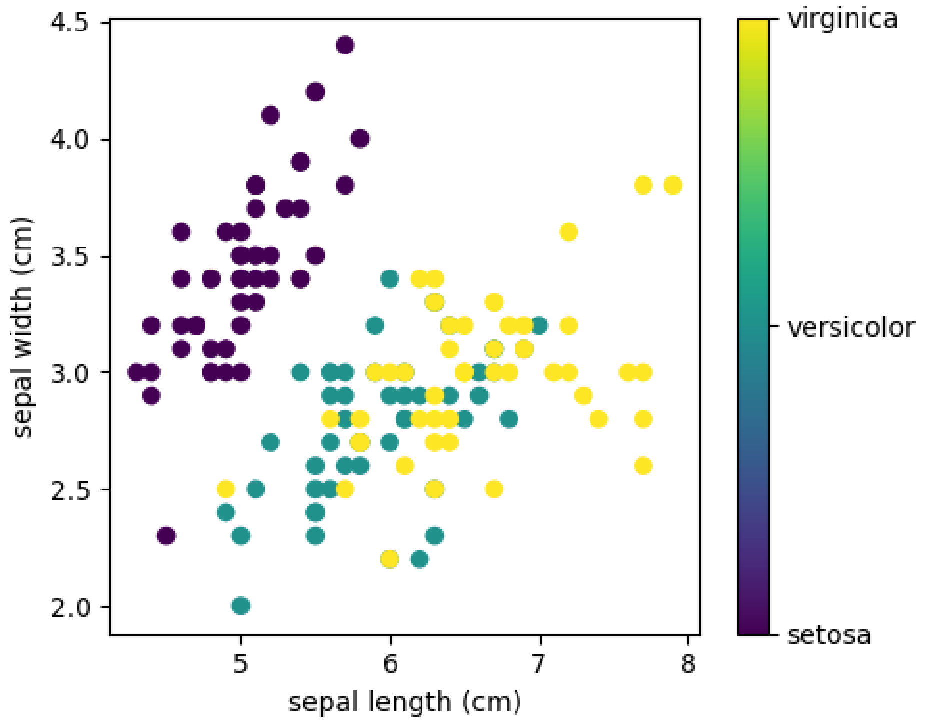
Based on your plot, put your answer here

We consider the absolute value of the coefficient when determining the strength of correlation. The strongest correlations are **petal length vs petal width** at 0.96, and **sepal length with petal length** at 0.87.

1.8.2 Question 15 (2 pts)

[79]:
```
# Let's make a scatter plot of the data

# The indices of the features that we are plotting
x_index = 0
y_index = 1
# this formatter will label the colorbar with the correct target names
# notice that the target has numerical values, the target_names exist in the␣
 ↪iris data
# we just plot right from the dataset, not from our dataframe.
formatter = plt.FuncFormatter(lambda i, *args: iris.target_names[int(i)])
```

```
# In the cell below the plot, can you explain what this lambda function is␣
↪doing?
# Do you understand how the data is gathered and shown?

plt.figure(figsize=(5, 4))
plt.scatter(iris.data[:, x_index], iris.data[:, y_index], c=iris.target)
plt.colorbar(ticks=[0, 1, 2], format=formatter)
plt.xlabel(iris.feature_names[x_index])
plt.ylabel(iris.feature_names[y_index])
plt.tight_layout()
plt.show()
```



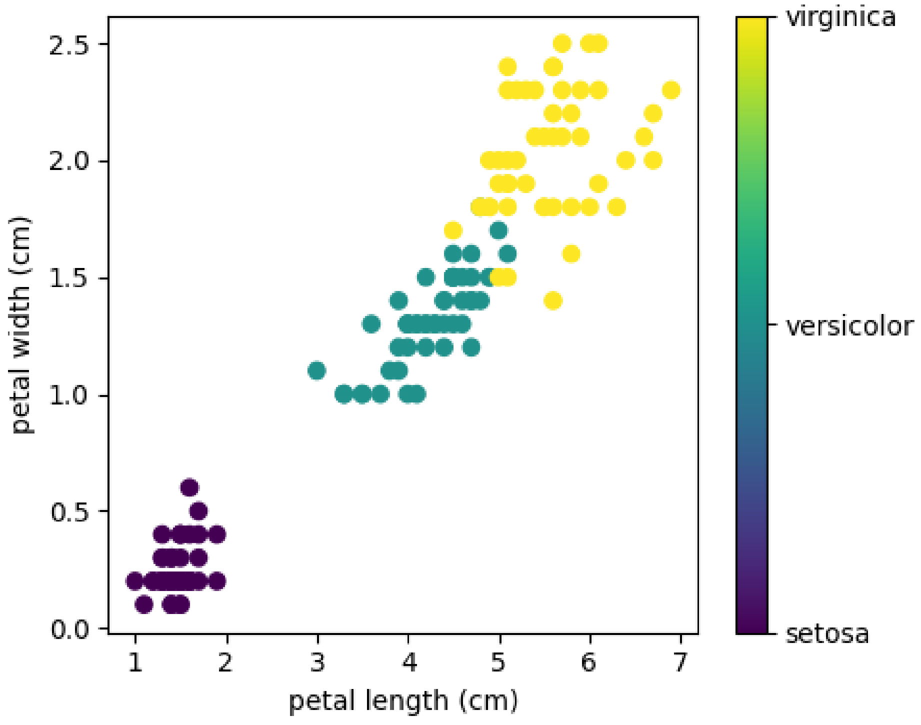
(1pt.) What is the lambda function/funcformatter doing? Answer in the next cell.

- `lambda` is a keyword in Python used to create an anonymous (unnamed) function.
- `i, *args` are the parameters for this lambda function. Here, `i` is the primary parameter, and `*args` allows for any additional arguments (which are not used in this context).
- `iris.target_names[int(i)]` is the function's return value. It accesses the `iris.target_names` array and returns the name corresponding to the index `i`. The `int(i)` conversion ensures that the index is an integer, as it may be passed as a float.

When this lambda function is applied to the color bar in your plot, it essentially maps each numeric target value (0, 1, 2) to its corresponding target name (like 'setosa') from the `iris.target_names` array. This way, the color bar will show these names instead of just the numeric values.

(1pt.) Copying and pasting code, now plot petal length vs petal width by changing the x_index and y_index values.

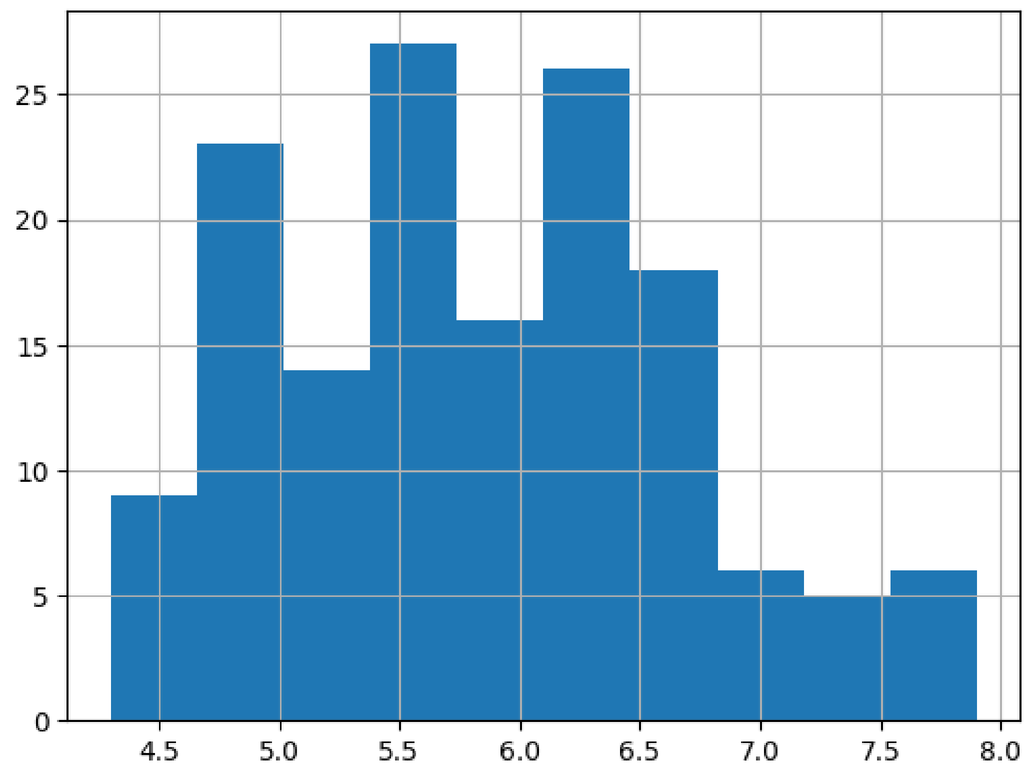
```
[81]: # The indices of the features that we are plotting
x_index = 2
y_index = 3
# this formatter will label the colorbar with the correct target names
# notice that the target has numerical values, the target_names exist in the␣
↪iris data
# we just plot right from the dataset, not from our dataframe.
formatter = plt.FuncFormatter(lambda i, *args: iris.target_names[int(i)])

# In the cell below the plot, can you explain what this lambda function is␣
↪doing?
# Do you understand how the data is gathered and shown?

plt.figure(figsize=(5, 4))
plt.scatter(iris.data[:, x_index], iris.data[:, y_index], c=iris.target)
plt.colorbar(ticks=[0, 1, 2], format=formatter)
plt.xlabel(iris.feature_names[x_index])
plt.ylabel(iris.feature_names[y_index])
plt.tight_layout()
plt.show()
```



Pandas built-in Viz Pandas has built in data visualization, based on matplotlib, so we show a few examples. (Feel free to use any plotting package you like for this course, but these are pretty convenient if you have a pandas dataframe ready to go.

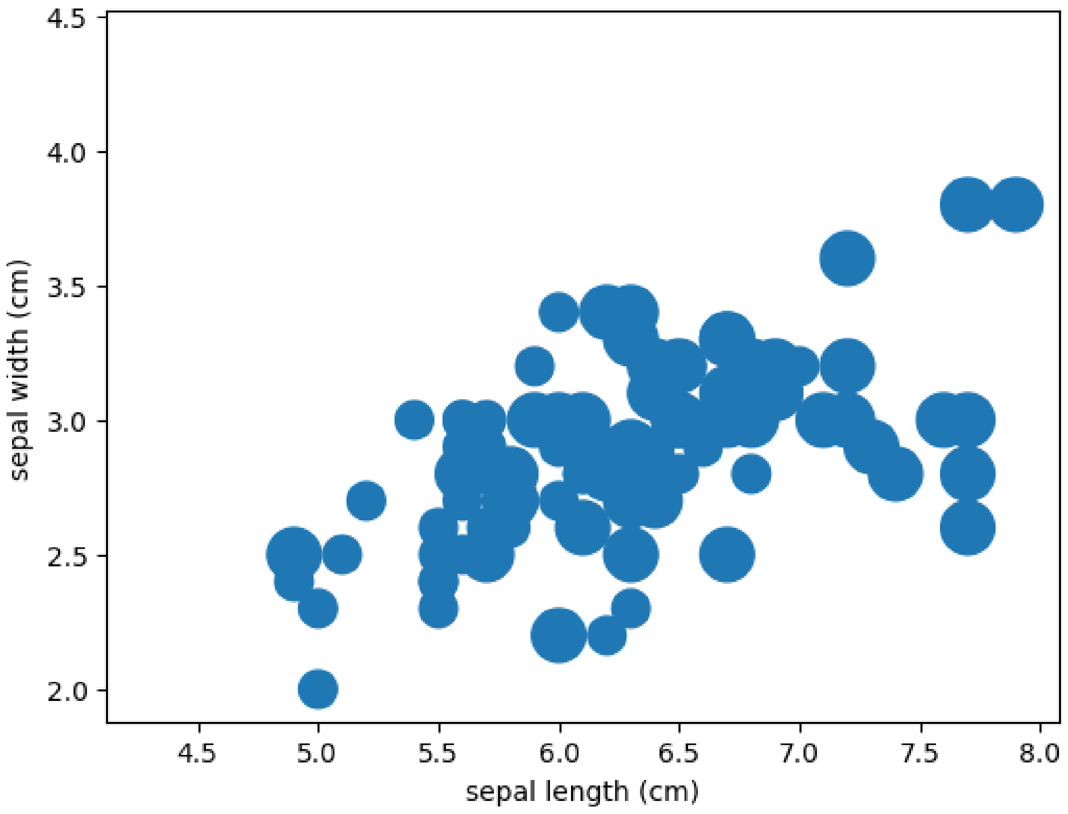
```
[82]: data1['sepal length (cm)'].hist()
```

```
[82]: <Axes: >
```



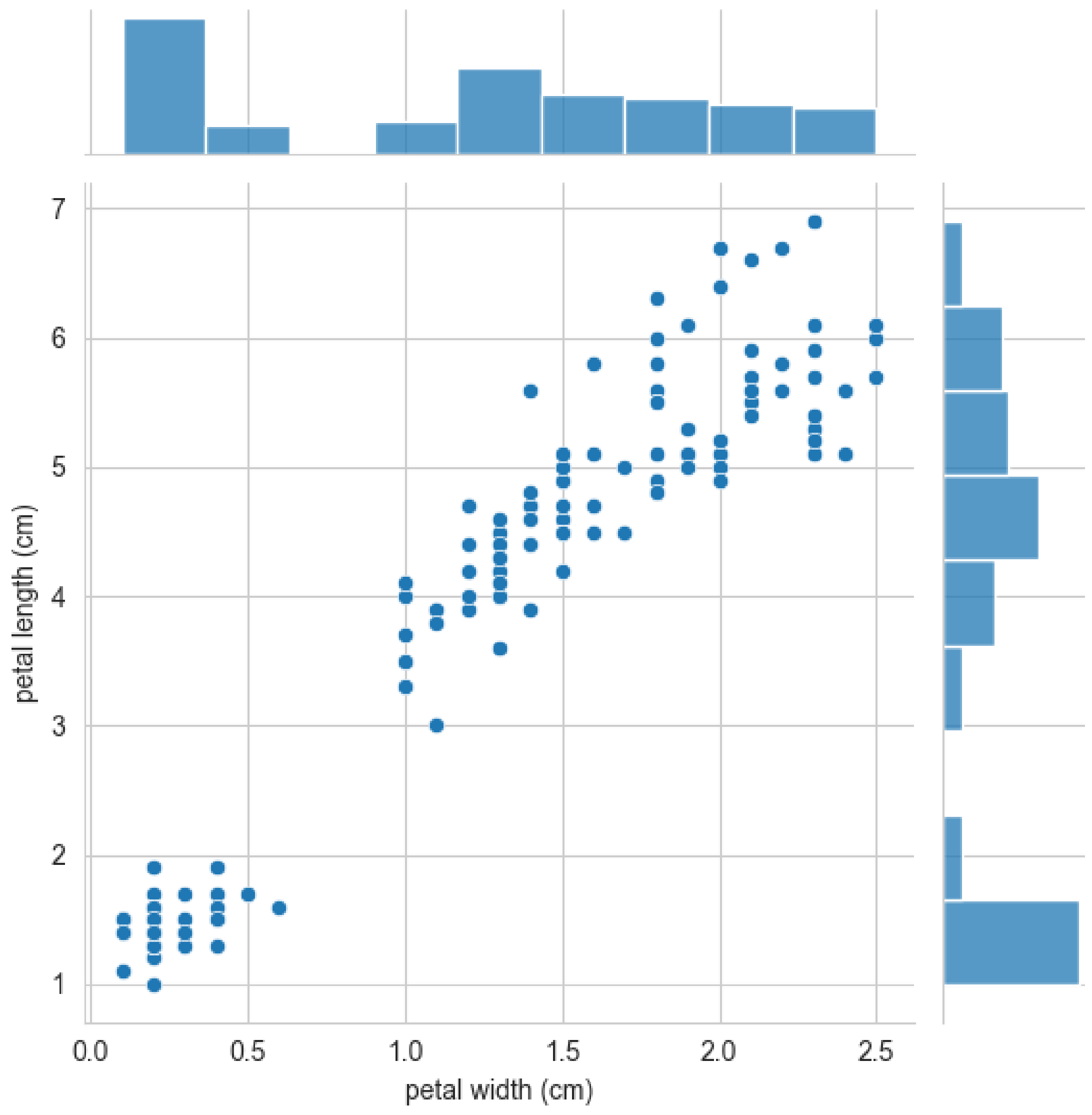
```
[83]: data1.plot.scatter(x='sepal length (cm)', y= 'sepal width (cm)',␣
        ↪s=data1['target']*200)
```

```
[83]: <Axes: xlabel='sepal length (cm)', ylabel='sepal width (cm)'>
```



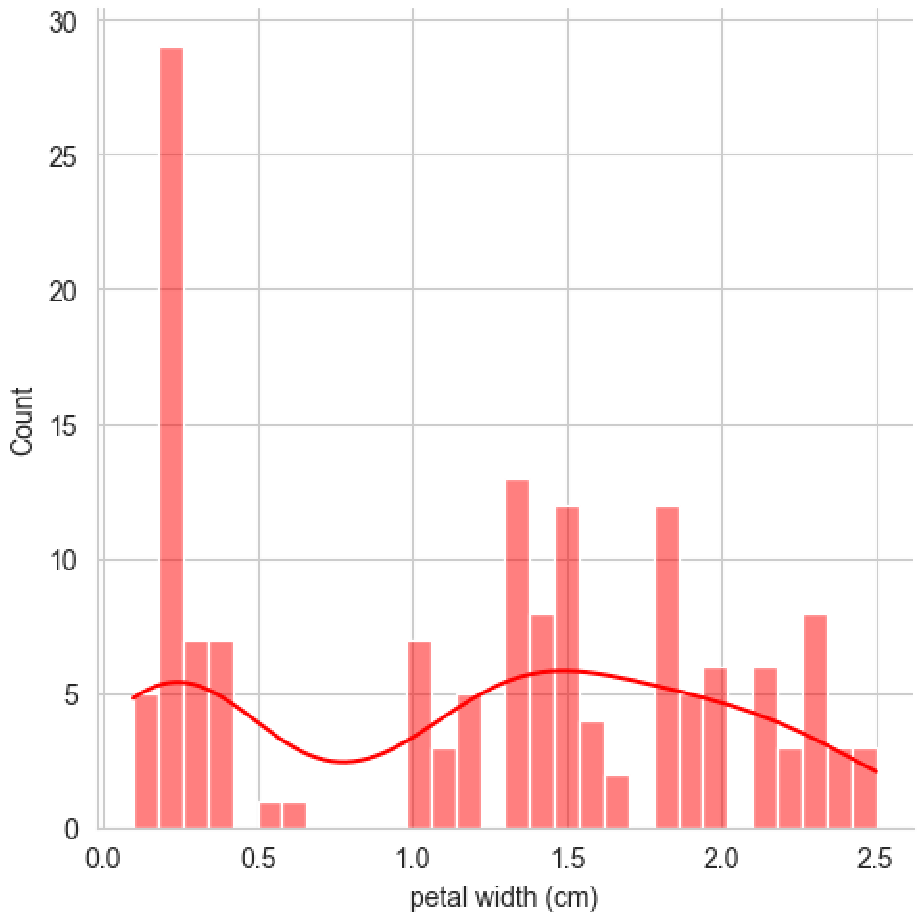
```
[84]: # Let's do some plotting with seaborn.
      # We start with a jointplot
      sns.set_style('whitegrid')
      sns.jointplot(x= 'petal width (cm)', y = 'petal length (cm)', data=data1)
```

```
[84]: <seaborn.axisgrid.JointGrid at 0x2b943f627a0>
```



```
[85]: # (1pt.) Using data1 and petal width, create a seaborn displot with 30 bins,␣
       ↪kde=True, color='red'

      ### BEGIN SOLUTION
      sns.displot(data1['petal width (cm)'], bins=30, kde=True, color='red')
      plt.show
      ### END SOLUTION
```

```
[85]: <function matplotlib.pyplot.show(close=None, block=None)>
```



1.9 Plotting Data

For this final section of the lab, we are going to be plotting using seaborn and the Titanic dataset. Your tasks will be to plot various directives. (There are also lots of Jupyter notebooks on Kaggle.com from challenges using the Titanic dataset, FYI)

[86]:
```
titanic = sns.load_dataset('titanic') #notice where we are getting Titanic from!
titanic.head()
```

[86]:
```
   survived  pclass     sex   age  sibsp  parch     fare embarked  class  \
0         0       3    male  22.0      1      0   7.2500        S  Third
1         1       1  female  38.0      1      0  71.2833        C  First
2         1       3  female  26.0      0      0   7.9250        S  Third
3         1       1  female  35.0      1      0  53.1000        S  First
4         0       3    male  35.0      0      0   8.0500        S  Third

     who  adult_male deck  embark_town alive  alone
0    man        True  NaN  Southampton    no  False
```

```
1  woman        False    C    Cherbourg   yes  False
2  woman        False  NaN  Southampton   yes   True
3  woman        False    C  Southampton   yes  False
4    man         True  NaN  Southampton    no   True
```

1.9.1 Question 16 (16 pts)

For this question you are going to make a series of plots with seaborn in the cells below using the Titanic dataset. **Remember to label your axes!** Include units if you have them.

Part 1: In a cell below, create a correlation matrix of features (you can use any method you like)

Part 2: Using seaborn and Titanic, in a cell below create a joint plot with the x axis as fare, the y axis as age. feel free to experiment with size and/or colors if you like.

Part 3: In a cell below, create a histogram of the fare, kde= False, and choose an appropriate number of bins

Part 4: Create a box plot with x axis being class type, and y axis age. Try palette ='rainbow'

Part 5: Create a swarm plot with x axis class, and y axis age, try the palette 'Set2'. You might get a warning– can you fix it? (it is okay to leave the warning, however)

Part 6: Create a count plot with x axis using sex.

Part 7: Create a heatmap of the correlations. Title it "Titanic"

Part 8: Create a facet grid using sex as the columns, then map the histogram on age.

1.9.2 1. Correlations Matrix

In order to create the correlation matrix for all the features we need to convert those features of incompatible data types inot compatible ones. I see objects, categories, and bools in the data so each of these will need to be converted into number values and we will create a dictionary of the mapping for reference.

[95]:
```python
# Function to create a mapping dictionary for a column
def create_mapping(column):
    unique_values = column.unique()
    return {value: idx for idx, value in enumerate(unique_values)}

# Identify columns with string or Boolean values
columns_to_map = titanic.select_dtypes(include=['object', 'category', 'bool']).
 ↪columns

# Dictionary to hold mappings for each column
mappings = {}

# New DataFrame for the mapped data
mapped_titanic = titanic.copy()

# Process each column
```

```
for col in columns_to_map:
    mapping = create_mapping(titanic[col])
    mappings[col] = mapping
    mapped_titanic[col] = titanic[col].map(mapping)

# Display the new DataFrame and the mappings
print("\nMappings:")
for col, mapping in mappings.items():
    print(f"{col}: {mapping}")

print("\nMapped DataFrame:")
mapped_titanic.head()
```

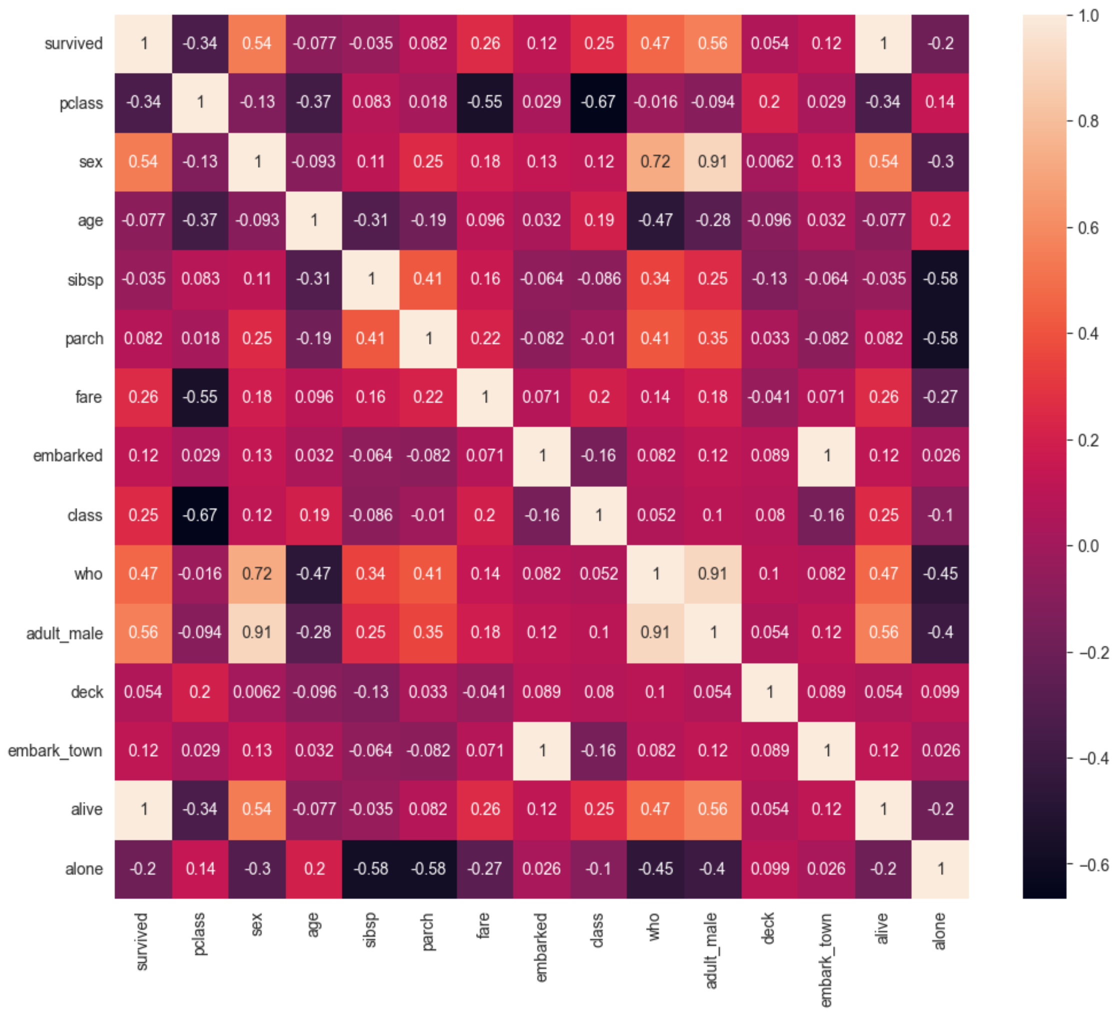
```
Mappings:
sex: {'male': 0, 'female': 1}
embarked: {'S': 0, 'C': 1, 'Q': 2, nan: 3}
class: {'Third': 0, 'First': 1, 'Second': 2}
who: {'man': 0, 'woman': 1, 'child': 2}
adult_male: {True: 0, False: 1}
deck: {nan: 0, 'C': 1, 'E': 2, 'G': 3, 'D': 4, 'A': 5, 'B': 6, 'F': 7}
embark_town: {'Southampton': 0, 'Cherbourg': 1, 'Queenstown': 2, nan: 3}
alive: {'no': 0, 'yes': 1}
alone: {False: 0, True: 1}

Mapped DataFrame:
```

[95]:

```
   survived  pclass  sex   age  sibsp  parch     fare  embarked  class  who  \
0         0       3    0  22.0      1      0   7.2500         0      0    0
1         1       1    1  38.0      1      0  71.2833         1      1    1
2         1       3    1  26.0      0      0   7.9250         0      0    1
3         1       1    1  35.0      1      0  53.1000         0      1    1
4         0       3    0  35.0      0      0   8.0500         0      0    0

   adult_male deck  embark_town  alive  alone
0           0  NaN            0      0      0
1           1    1            1      1      0
2           1  NaN            0      1      1
3           1    1            0      1      0
4           0  NaN            0      0      1
```

[101]:

```
#We need a large figure since there are many features
plt.figure(figsize=(12, 10))
sns.heatmap(mapped_titanic.corr(), annot = True);
plt.show()
```

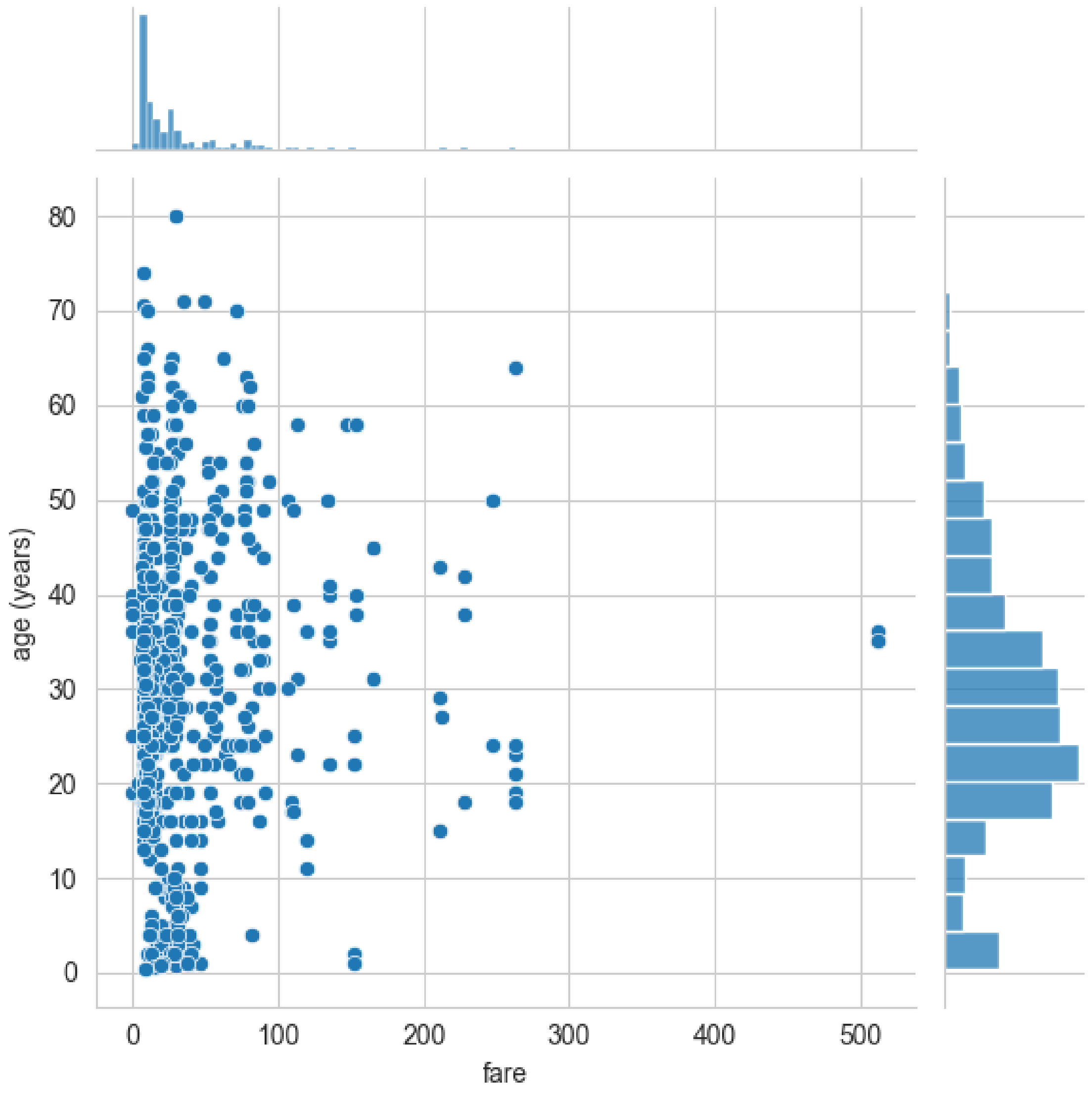


The plot is busy but we can see 1's on the main diagonal as should be the case. There is perfect corelation between 'embarked' and 'embark_town' the data appears to be in duplicate here. There is also 1 to 1 correlation between alive and survived so these appear to be representing the same data also. At a glance the matrix appears reasonable.

1.9.3 2. Fare VS Age, Jointplot

```
[114]: sns.set_style('whitegrid')
       joint_plot = sns.jointplot(x= 'fare', y = 'age', data=titanic)
       joint_plot.set_axis_labels('fare', 'age (years)')
       plt.show
       # Fare is probably in pounds but I'm not sure.
```

```
[114]: <function matplotlib.pyplot.show(close=None, block=None)>
```

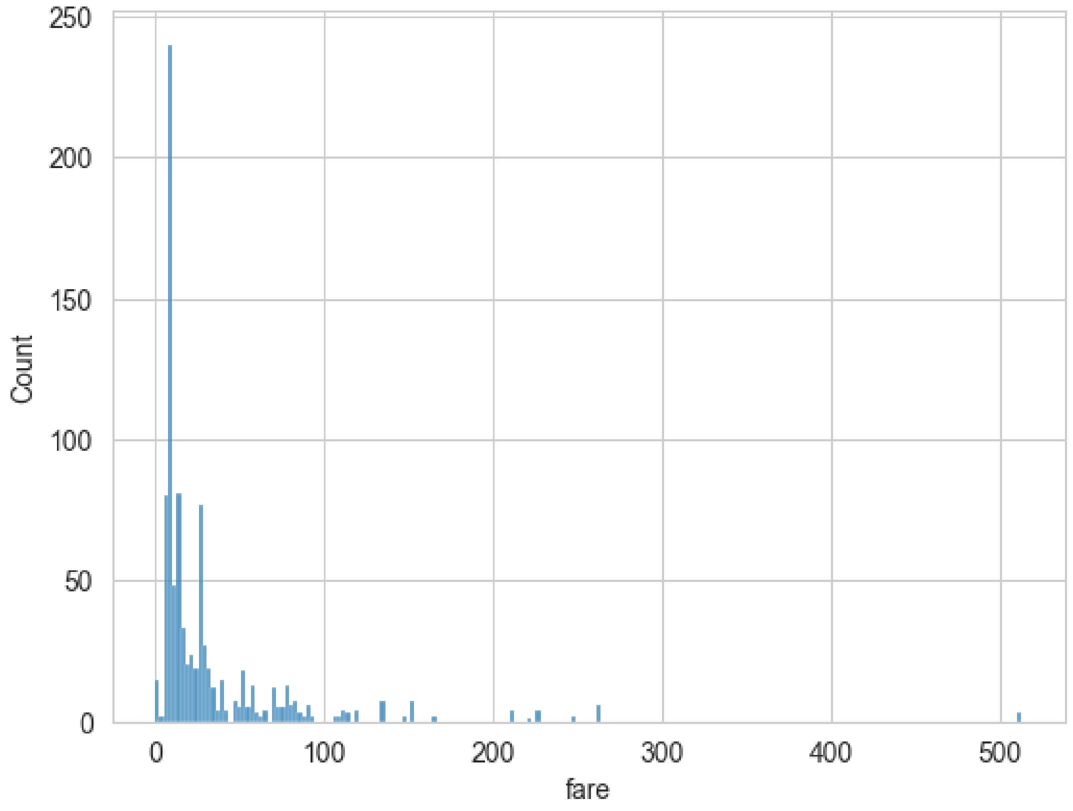


1.9.4 3. Fare Histogram

Here I increase the number of bins to distinguish this plot from the above joint plot.

```
[115]: sns.histplot(titanic['fare'], kde=False, bins=200)
```

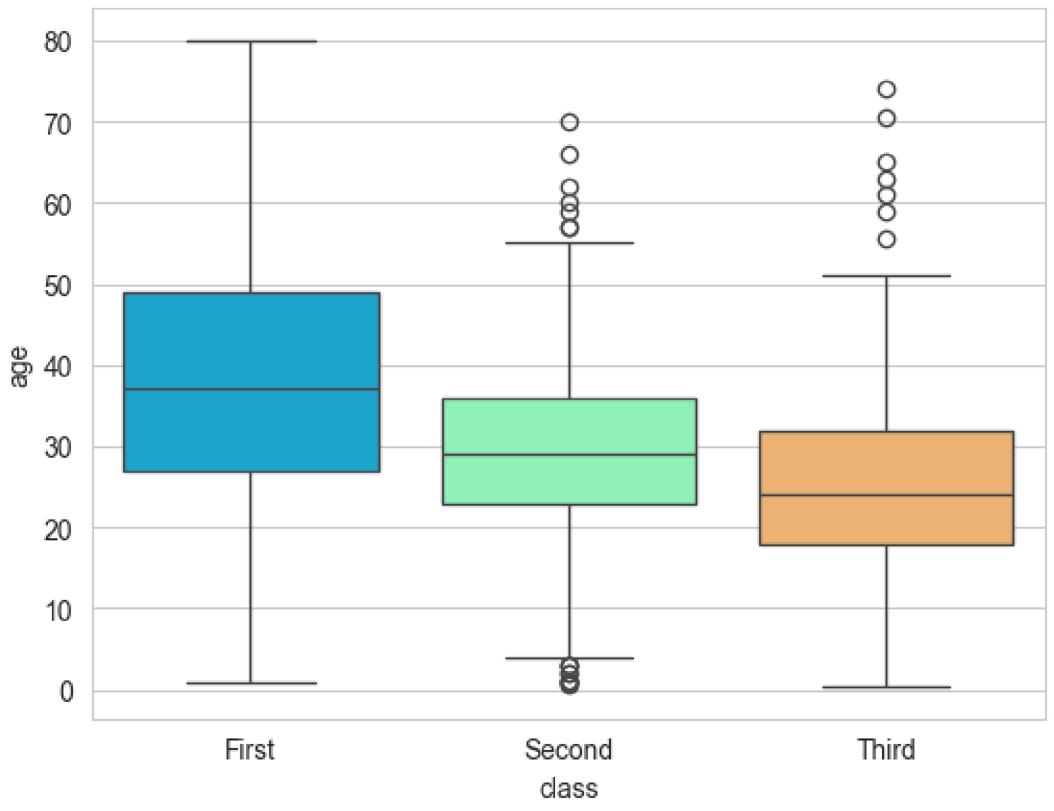
```
[115]: <Axes: xlabel='fare', ylabel='Count'>
```



1.9.5 4. Class vs Age, Box Plot

```
[118]: sns.boxplot(x='class', y='age', hue='class', data=titanic, palette='rainbow')
       plt.show
```

```
[118]: <function matplotlib.pyplot.show(close=None, block=None)>
```



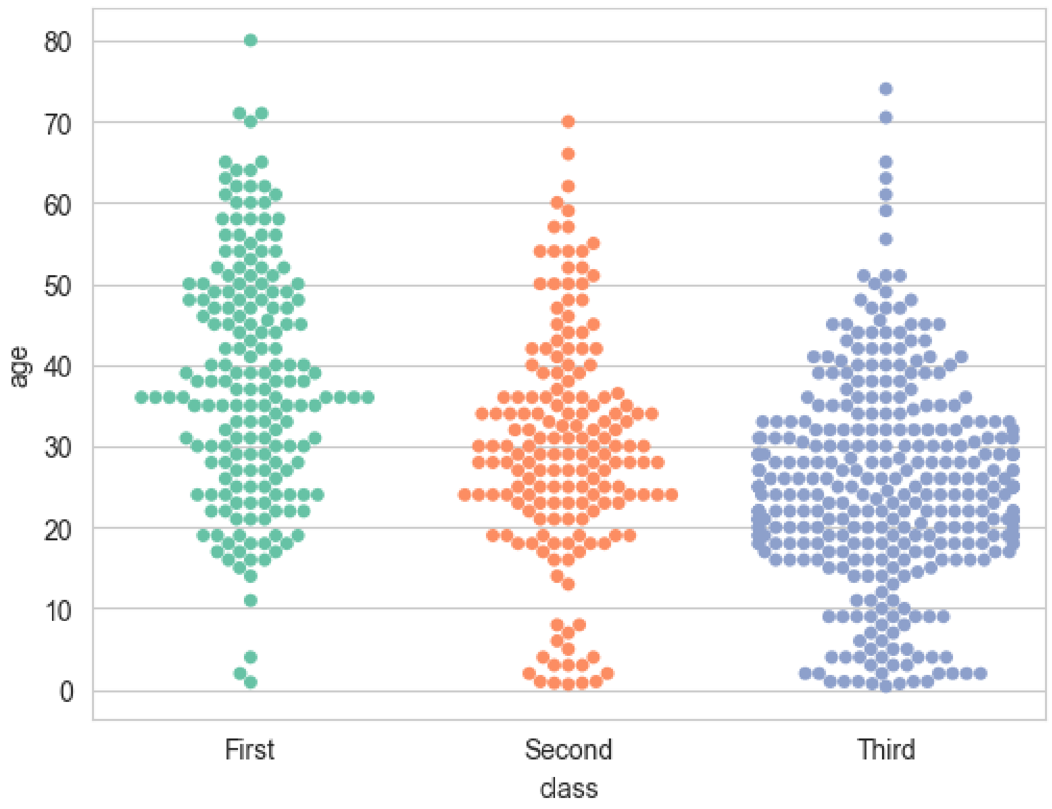
1.9.6 5. Class vs Age, Swarm Plot

[120]:
```
sns.swarmplot(x='class', y='age', data=titanic, palette='Set2')
```

```
C:\Users\Captain\AppData\Local\Temp\ipykernel_17448\2191637981.py:1:
FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same
effect.

  sns.swarmplot(x='class', y='age', data=titanic, palette='Set2')
```

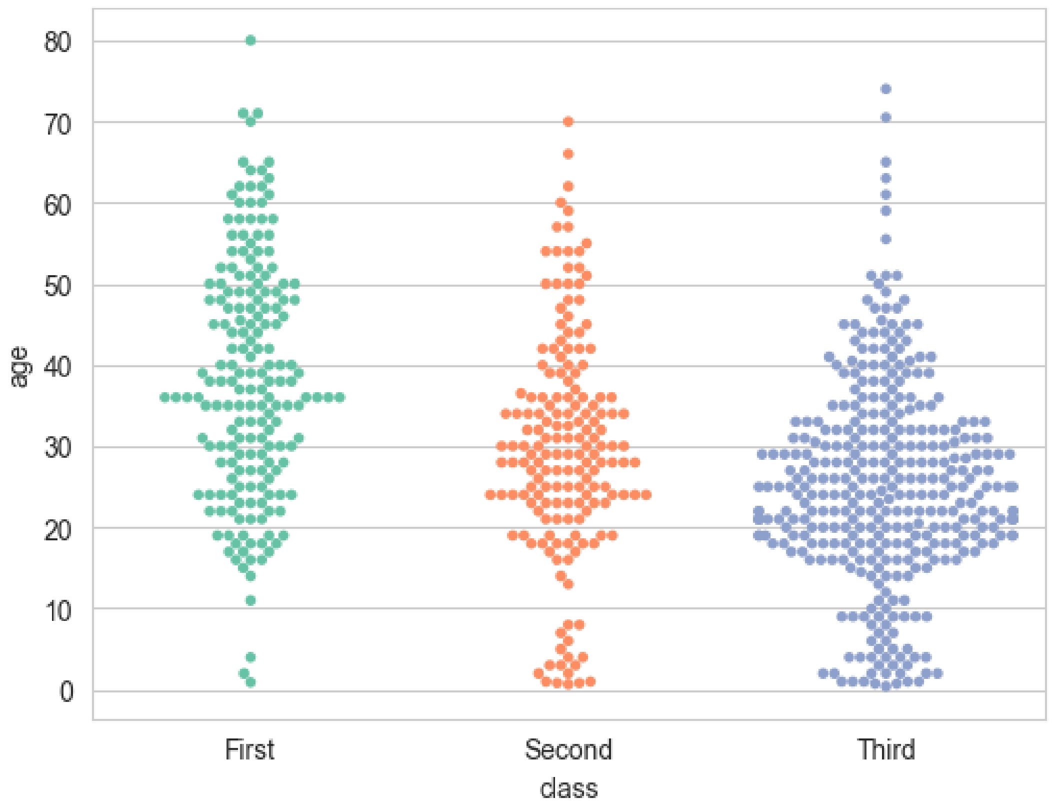
[120]:
```
<Axes: xlabel='class', ylabel='age'>
```

```
C:\Users\Captain\AppData\Local\Programs\Python\Python310\lib\site-
packages\seaborn\categorical.py:3398: UserWarning: 15.2% of the points cannot be
placed; you may want to decrease the size of the markers or use stripplot.
  warnings.warn(msg, UserWarning)
```



[127]:
```
# Fix warnings by adding 'hue' param and shrinkiing swarm point size
sns.swarmplot(x='class', y='age', hue='class', data=titanic, palette='Set2',␣
    ↪size=4)
```

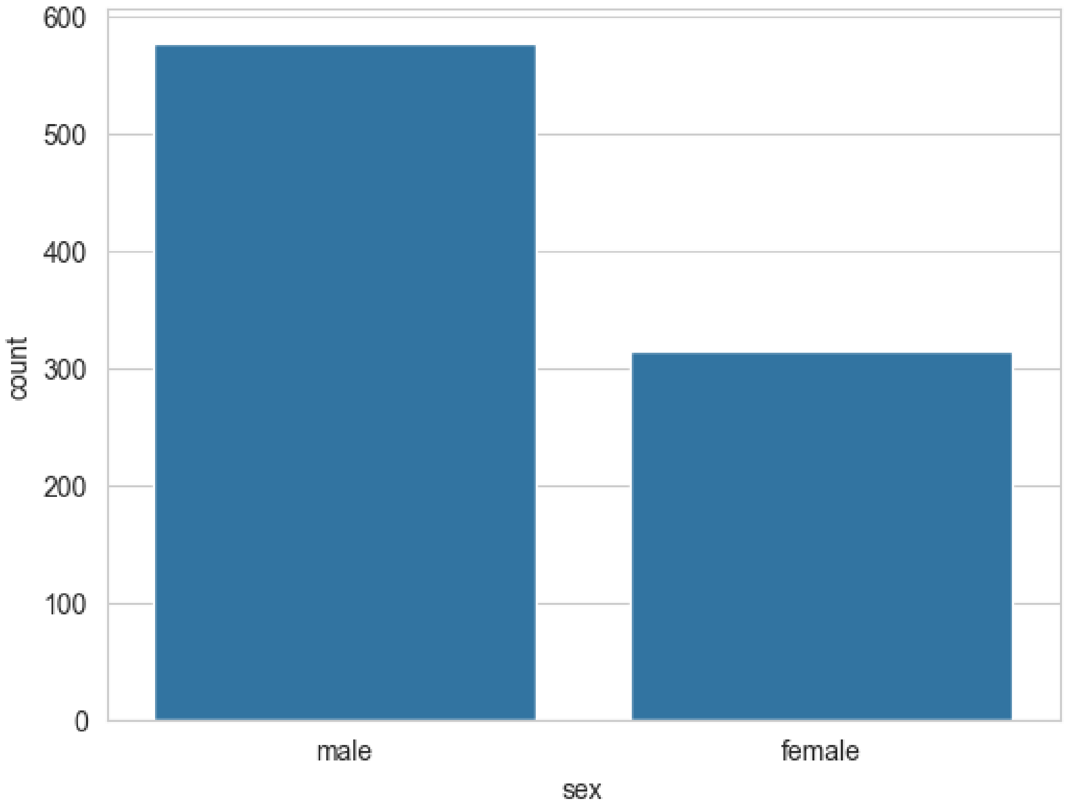
[127]:
```
<Axes: xlabel='class', ylabel='age'>
```



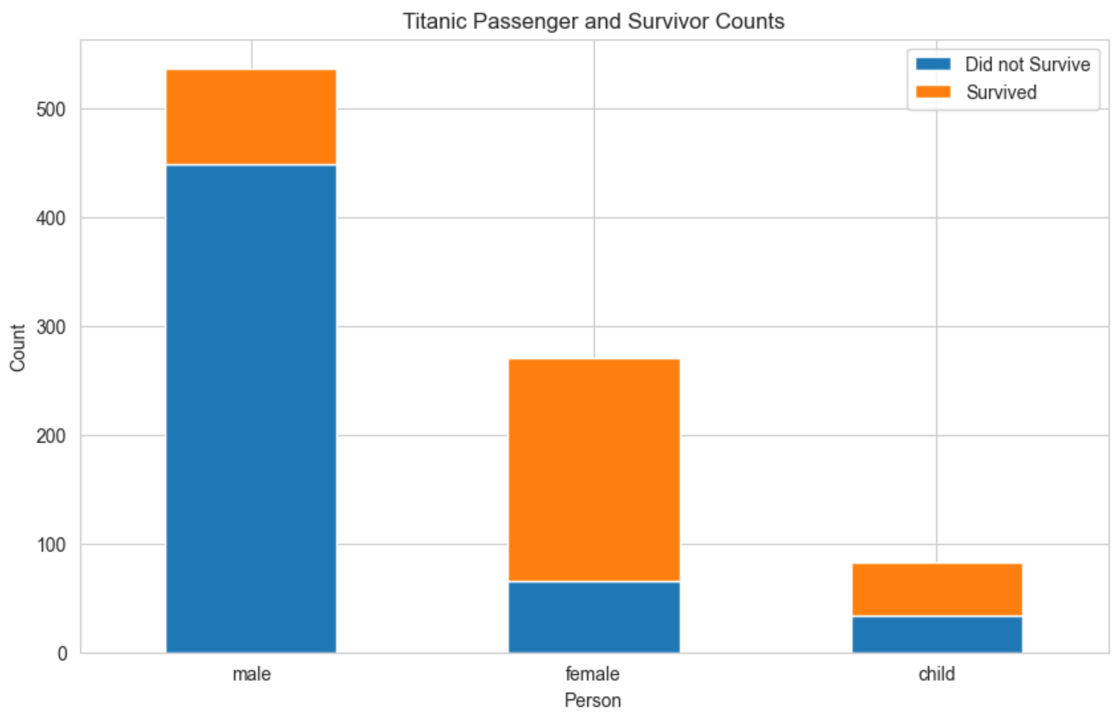
1.9.7 6. Count Plot by Sex

```
[140]: sns.countplot(x='sex', data=titanic)
```

```
[140]: <Axes: xlabel='sex', ylabel='count'>
```

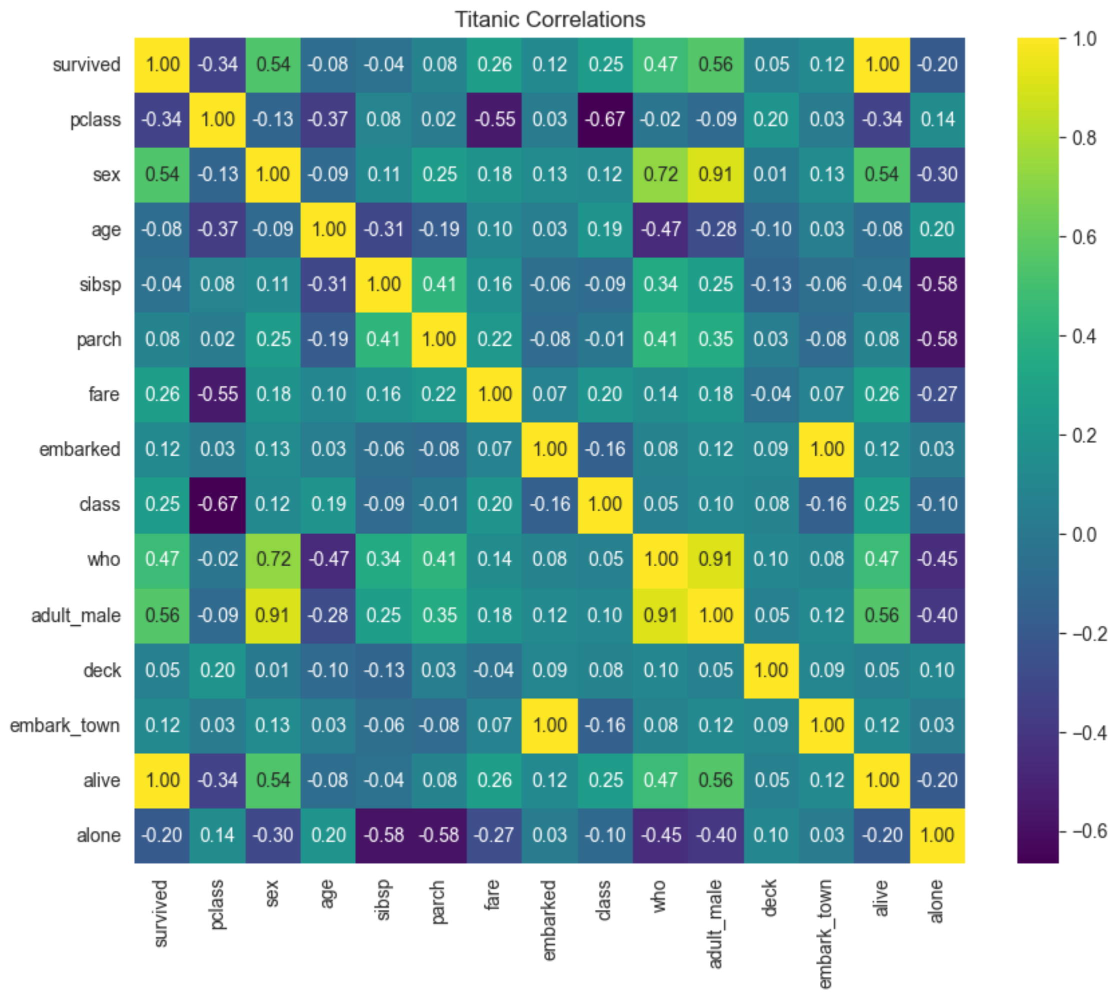


[147]:
```
# Compare passenger counts to survivor counts for men, women, and children
# Very sad to look at but I wanted to know

# Create the 'person' category
titanic['person'] = titanic.apply(lambda row: 'child' if row['age'] < 16 else␣
↪row['sex'], axis=1)

# Create a pivot table for the stacked bar chart
pivot = titanic.pivot_table(index='person', columns='survived', aggfunc='size',␣
↪fill_value=0)

# Order the rows of the pivot table
pivot = pivot.reindex(['male', 'female', 'child'])

# Plotting the stacked bar chart
pivot.plot(kind='bar', stacked=True, figsize=(10, 6))

# Setting the labels and title
plt.xlabel('Person')
plt.ylabel('Count')
plt.title('Titanic Passenger and Survivor Counts')
plt.xticks(rotation=0)
plt.legend(['Did not Survive', 'Survived'])

# Display the plot
plt.show()
```



1.9.8 7. Correlation Heatmap

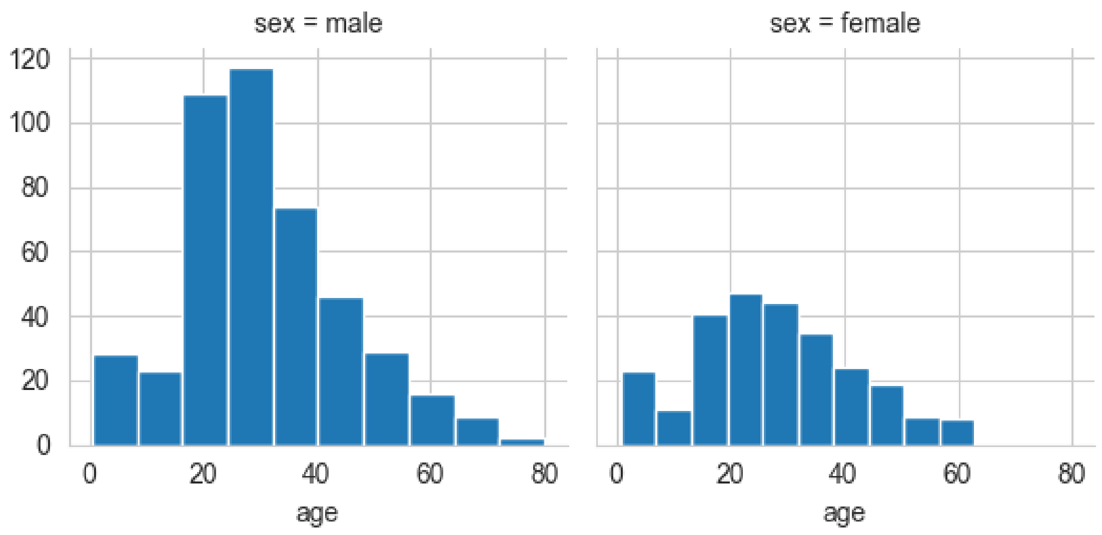
[150]:
```
# I included this above in part 1 also so I'll just change it up a bit

# Calculate the correlation matrix
corr = mapped_titanic.corr()

# Create the heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(corr, annot=True, fmt=".2f", cmap='viridis')
plt.title('Titanic Correlations')
# Display the plot
plt.show()
```



1.9.9 8. Sex and Age, Facet Grid

[151]:
```python
g = sns.FacetGrid(data=titanic, col='sex')
g.map(plt.hist, 'age')
plt.show
```

[151]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```



```
[ ]: # answer people are missing
     # g = sns.FacetGrid(data=titanic,col='sex')
     # g.map(plt.hist,'age')
```

2 Good Job!

Rerun your cells and submit your assignment.