Data Handling

Content from Data Frames - Part 1

Last updated on 2024-05-24 | Edit this page

Download Chapter notebook (ipynb)

Mandatory Lesson Feedback Survey

OVERVIEW

Questions

- How is data read into a dataframe?
- What are different ways to manipulate data in dataframes?
- What makes data visualisation simple in Python?

Objectives

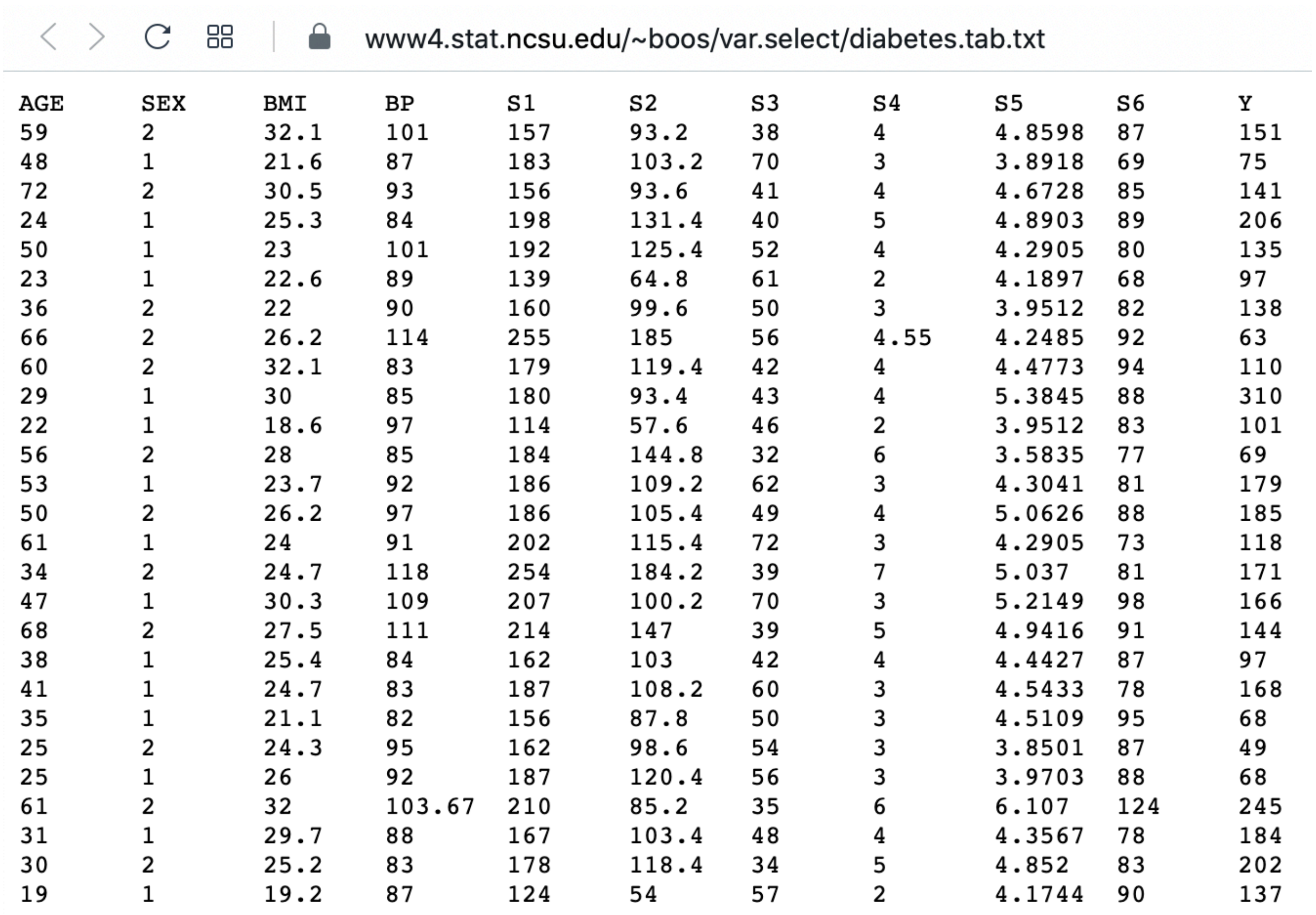
- Import data set as Pandas dataframe
- Inspect data frame and access data
- Produce an overview of data features
- Create data plots using Matplotlib

PREREQUISITES

- Indexing of Arrays
- For Loop through Array
- Basic Statistics (distributions, mean, median, standard deviation)

Challenge: The diabetes data set

Here is a screenshot of the so-called diabetes data set. It is taken from this webpage and it is one of the example data sets used to illustrate machine learning functionality in scikit-learn (Part III and Part IV of the course).


www4.stat.ncsu.edu/~boos/var.select/diabetes.tab.txt

AGE	SEX	BMI	BP	S1	S2	S3	S4	S5	S6	Y
59	2	32.1	101	157	93.2	38	4	4.8598	87	151
48	1	21.6	87	183	103.2	70	3	3.8918	69	75
72	2	30.5	93	156	93.6	41	4	4.6728	85	141
24	1	25.3	84	198	131.4	40	5	4.8903	89	206
50	1	23	101	192	125.4	52	4	4.2905	80	135
23	1	22.6	89	139	64.8	61	2	4.1897	68	97
36	2	22	90	160	99.6	50	3	3.9512	82	138
66	2	26.2	114	255	185	56	4.55	4.2485	92	63
60	2	32.1	83	179	119.4	42	4	4.4773	94	110
29	1	30	85	180	93.4	43	4	5.3845	88	310
22	1	18.6	97	114	57.6	46	2	3.9512	83	101
56	2	28	85	184	144.8	32	6	3.5835	77	69
53	1	23.7	92	186	109.2	62	3	4.3041	81	179
50	2	26.2	97	186	105.4	49	4	5.0626	88	185
61	1	24	91	202	115.4	72	3	4.2905	73	118
34	2	24.7	118	254	184.2	39	7	5.037	81	171
47	1	30.3	109	207	100.2	70	3	5.2149	98	166
68	2	27.5	111	214	147	39	5	4.9416	91	144
38	1	25.4	84	162	103	42	4	4.4427	87	97
41	1	24.7	83	187	108.2	60	3	4.5433	78	168
35	1	21.1	82	156	87.8	50	3	4.5109	95	68
25	2	24.3	95	162	98.6	54	3	3.8501	87	49
25	1	26	92	187	120.4	56	3	3.9703	88	68
61	2	32	103.67	210	85.2	35	6	6.107	124	245
31	1	29.7	88	167	103.4	48	4	4.3567	78	184
30	2	25.2	83	178	118.4	34	5	4.852	83	202
19	1	19.2	87	124	54	57	2	4.1744	90	137

This figure captures only the top part of the data. On the webpage you need to scroll down considerably to view the whole content. Thus, to get an **overview** of the dataset is the first main task in Data Science.

THE LESSON

- introduces code to read and inspect the data
- works with a specific data frame and extracts some techniques to get an overview
- discusses the concept 'distribution' as a way of summarising data in a single figure

TO GET TO KNOW A DATASET YOU NEED TO

- access the data
- check the content
- produce a summary of basic properties

In this lesson we will only look at univariate features where each data column is studied independently of the others. Further properties and bivariate features will be the topic of the next lesson.

Work Through Example

READING DATA INTO A PANDAS DATAFRAME

The small practice data file for this section is called 'everleys_data.csv' and can be downloaded using the link given above in "Materials for this Lesson". To start, please create a subfolder called 'data' in the current directory and put the data file in it. It can now be accessed using the relative path `data/everleys_data.csv` or `data\everleys_data.csv`, respectively.

The file `everleys_data.csv` contains blood concentrations of calcium and sodium ions from 17 patients with Everley's syndrome. The data are taken from a BMJ statistics tutorial. The data are stored as comma-separated values (csv), two values for each patient.

To get to know a dataset, we will use the Pandas package and the Matplotlib plotting library. The Pandas package for data science is included in the Anaconda distribution of Python. Check this link for installation instructions to get started.

If you are not using the Anaconda distribution, please refer to these guidelines.

To use the functions contained in Pandas they need to be imported. Our dataset is in '.csv' format, and we therefore need to read it from a csv file. For this, we import the function `read_csv`. This function will create a *Pandas dataframe*.

PYTHON

```python
from pandas import read_csv
```

Executing this code does not lead to any output on the screen. However, the function is now ready to be used. To use it, we type its name and provide the required arguments. The following code should import the Everley's data into your JupyterLab notebook (or other Python environment):

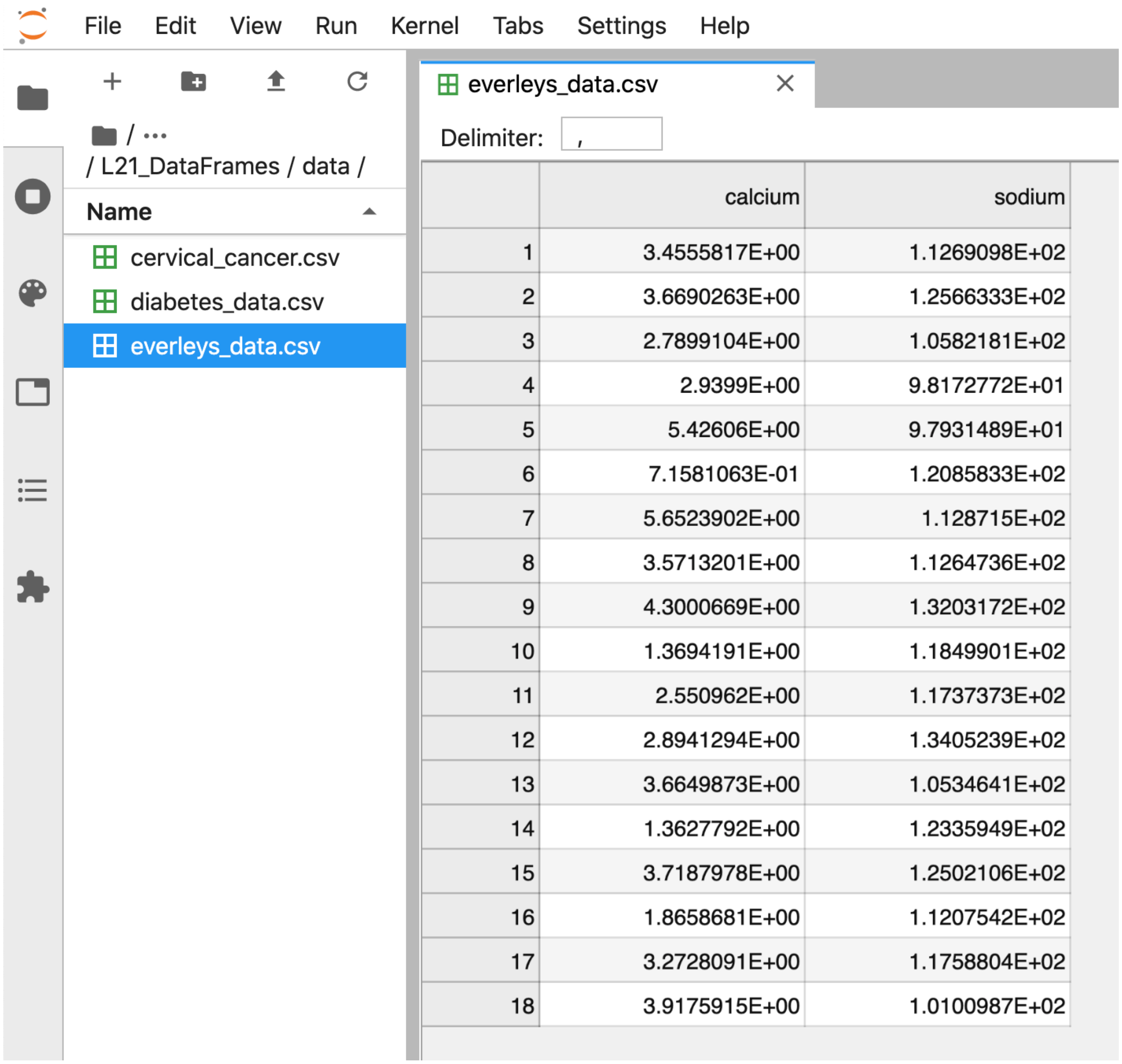
PYTHON

```python
# for Mac OSX and Linux
# (please go to the next cell if using Windows)

df = read_csv("data/everleys_data.csv")
```

PYTHON

```python
# Please uncomment for Windows
# (please go to previous cell if using Mac OSX or Linux)

# df = read_csv("data\everleys_data.csv")
```

This code uses the `read_csv` function from Pandas to read data from a data file, in this case a file with extension '.csv'. Note that the location of the data file is specified within quotes by the relative path to the subfolder 'data' followed by the file name. Use the JupyterLab file browser to check that subfolder exists and has the file in it.



	calcium	sodium
1	3.4555817E+00	1.1269098E+02
2	3.6690263E+00	1.2566333E+02
3	2.7899104E+00	1.0582181E+02
4	2.9399E+00	9.8172772E+01
5	5.42606E+00	9.7931489E+01
6	7.1581063E-01	1.2085833E+02
7	5.6523902E+00	1.128715E+02
8	3.5713201E+00	1.1264736E+02
9	4.3000669E+00	1.3203172E+02
10	1.3694191E+00	1.1849901E+02
11	2.550962E+00	1.1737373E+02
12	2.8941294E+00	1.3405239E+02
13	3.6649873E+00	1.0534641E+02
14	1.3627792E+00	1.2335949E+02
15	3.7187978E+00	1.2502106E+02
16	1.8658681E+00	1.1207542E+02
17	3.2728091E+00	1.1758804E+02
18	3.9175915E+00	1.0100987E+02

After execution of the code, the data are contained in a variable called `df`. This is a structure referred to as a Pandas *DataFrame*.

> A Pandas dataframe is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it as a spreadsheet.

To see the contents of `df`, simply use:

PYTHON

```python
df
```

OUTPUT

```
     calcium      sodium
0   3.455582  112.690980
1   3.669026  125.663330
2   2.789910  105.821810
3   2.939900   98.172772
4   5.426060   97.931489
5   0.715811  120.858330
6   5.652390  112.871500
7   3.571320  112.647360
8   4.300067  132.031720
9   1.369419  118.499010
10  2.550962  117.373730
11  2.894129  134.052390
12  3.664987  105.346410
13  1.362779  123.359490
14  3.718798  125.021060
15  1.865868  112.075420
16  3.272809  117.588040
17  3.917591  101.009870
```

(Compare with the result of `print(df)` which displays the contents in a different format.)

The output shows in the first column an index, integers from 0 to 17; and the calcium and sodium concentrations in columns 2 and 3, respectively. The default indexing starts from zero (Python is a 'zero-based' programming language).

In a dataframe, the first column is referred to as *Indices*, the first row is referred to as *Labels*. Note that the row with the labels is excluded from the row count. Similarly, the row with the indices is excluded from the column count.

For large data sets, the function `head` is a convenient way to get a feel of the dataset.

PYTHON

```python
df.head()
```

OUTPUT

```
    calcium      sodium
0  3.455582  112.690980
1  3.669026  125.663330
2  2.789910  105.821810
3  2.939900   98.172772
4  5.426060   97.931489
```

Without any input argument, this displays the first five data lines of the dataframe. You can specify alter the number of rows displayed by including a single integer as argument, e.g. `head(10)`.

If you feel there are too many decimal places in the default view, you can restrict their number by using the `round` function:

PYTHON

```python
df.head().round(2)
```

OUTPUT

```
   calcium  sodium
0     3.46  112.69
1     3.67  125.66
2     2.79  105.82
3     2.94   98.17
4     5.43   97.93
```

While we can see how many rows there are in a dataframe when we display the whole data frame and look at the last index, there is a convenient way to obtain the number directly:

PYTHON

```python
no_rows = len(df)

print('Data frame has', no_rows, 'rows')
```

OUTPUT

```
Data frame has 18 rows
```

You could see above, that the columns of the dataframe have labels. To see all labels:

PYTHON

```python
column_labels = df.columns

print(column_labels)
```

OUTPUT

```
Index(['calcium', 'sodium'], dtype='object')
```

Now we can count the labels to obtain the number of columns:

PYTHON

```python
no_columns = len(column_labels)

print('Data frame has', no_columns, 'columns')
```

OUTPUT

```
Data frame has 2 columns
```

And if you want to have both the number of the rows and the columns together, use `shape`. Shape returns a tuple of two numbers, first the number of rows, then the number of columns.

PYTHON

```
df_shape = df.shape

print('Data frame has', df_shape[0], 'rows and',df_shape[1],  'columns')
```

OUTPUT

```
Data frame has 18 rows and 2 columns
```

Notice that `shape` (like `columns`) is not followed by round parenthesis. It is not a function that can take arguments. Technically, `shape` is a 'property' of the dataframe.

To find out what data type is contained in each of the columns, us `dtypes`, another 'property':

PYTHON

```
df.dtypes
```

OUTPUT

```
calcium    float64
sodium     float64
dtype: object
```

In this case, both columns contain floating point (decimal) numbers.

DIY1: READ DATA INTO A DATAFRAME

Download the data file 'loan_data.csv' using the link given above in "Materials for this Lesson". It contains data that can be used for the assessment of loan applications. Read the data into a DataFrame. It is best to assign it a name other than 'df' (to avoid overwriting the Evereley data set).

Display the first ten rows of the Loan data set to see its contents. It is taken from a tutorial on Data Handling in Python which you might find useful for further practice.

From this exercise we can see that a dataframe can contain different types of data: real numbers (e.g. LoanAmount), integers (ApplicantIncome), categorical data (Gender), and strings (Loan_ID).

Solution

PYTHON

```python
from pandas import read_csv
# dataframe from .csv file
df_loan = read_csv("data/loan_data.csv")
# display contents
df_loan.head(10)
```

OUTPUT

```
    Loan_ID  Gender Married  ... Loan_Amount_Term Credit_History Property_Area
0  LP001015    Male     Yes  ...            360.0            1.0         Urban
1  LP001022    Male     Yes  ...            360.0            1.0         Urban
2  LP001031    Male     Yes  ...            360.0            1.0         Urban
3  LP001035    Male     Yes  ...            360.0            NaN         Urban
4  LP001051    Male      No  ...            360.0            1.0         Urban
5  LP001054    Male     Yes  ...            360.0            1.0         Urban
6  LP001055  Female      No  ...            360.0            1.0     Semiurban
7  LP001056    Male     Yes  ...            360.0            0.0         Rural
8  LP001059    Male     Yes  ...            240.0            1.0         Urban
9  LP001067    Male      No  ...            360.0            1.0     Semiurban

[10 rows x 12 columns]
```

Accessing data in a DataFrame

If a datafile is large and you only want to check the format of data in a specific column, you can limit the display to that column. To access data contained in a specific column of a dataframe, we can use a similar convention as in a Python dictionary, treating the column names as 'keys'. E.g. to show all rows in column 'Calcium', use:

PYTHON

```python
df['calcium']
```

OUTPUT

```
0     3.455582
1     3.669026
2     2.789910
3     2.939900
4     5.426060
5     0.715811
6     5.652390
7     3.571320
8     4.300067
9     1.369419
10    2.550962
11    2.894129
12    3.664987
13    1.362779
14    3.718798
15    1.865868
16    3.272809
17    3.917591
Name: calcium, dtype: float64
```

To access individual rows of a column we use two pairs of square brackets:

PYTHON

```python
df['calcium'][0:3]
```

OUTPUT

```
0    3.455582
1    3.669026
2    2.789910
Name: calcium, dtype: float64
```

Here all rules for slicing can be applied. As for lists and tuples, the indexing of rows is semi-inclusive, lower boundary included, upper boundary excluded. Note that the first pair of square brackets refers to a column and the second pair refers to the rows. This is different from e.g. accessing items in a nested list.

Accessing items in a Pandas dataframe is analogous to accessing the values in a Python dictionary by referring to its keys.

To access non-contiguous elements, we use an additional pair of square brackets (as if for a list within a list):

PYTHON

```python
df['calcium'][[1, 3, 7]]
```

OUTPUT

```
1    3.669026
3    2.939900
7    3.571320
Name: calcium, dtype: float64
```

Another possibility to index and slice a dataframe is the use of the 'index location' or `iloc` property. It refers first to rows and then to columns by index, all within a single pair of brackets. For example, to get all rows of the first column (index `0`), you use:

PYTHON

```
df.iloc[:, 0]
```

OUTPUT

```
0     3.455582
1     3.669026
2     2.789910
3     2.939900
4     5.426060
5     0.715811
6     5.652390
7     3.571320
8     4.300067
9     1.369419
10    2.550962
11    2.894129
12    3.664987
13    1.362779
14    3.718798
15    1.865868
16    3.272809
17    3.917591
Name: calcium, dtype: float64
```

To display only the first three calcium concentrations, you use slicing, remembering that the upper bound is excluded):

PYTHON

```
df.iloc[0:3, 0]
```

OUTPUT

```
0    3.455582
1    3.669026
2    2.789910
Name: calcium, dtype: float64
```

To access non-consecutive values, we can use a pair of square brackets within the pair of square brackets:

PYTHON

```
df.iloc[[2, 4, 7], 0]
```

OUTPUT

```
2    2.78991
4    5.42606
7    3.57132
Name: calcium, dtype: float64
```

Similarly, we can access the values from multiple columns:

PYTHON

```
df.iloc[[2, 4, 7], :]
```

OUTPUT

```
   calcium      sodium
2  2.78991  105.821810
4  5.42606   97.931489
7  3.57132  112.647360
```

To pick only the even rows from the two columns, check this colon notation:

PYTHON

```
df.iloc[:18:2, :]
```

OUTPUT

```
     calcium      sodium
0   3.455582  112.690980
2   2.789910  105.821810
4   5.426060   97.931489
6   5.652390  112.871500
8   4.300067  132.031720
10  2.550962  117.373730
12  3.664987  105.346410
14  3.718798  125.021060
16  3.272809  117.588040
```

The number after the second colon indicates the stepsize.

DIY2: SELECT DATA FROM DATAFRAME

Display the calcium and sodium concentrations of all patients except the first. Check the model solution at the bottom for options.

Solution

PYTHON

```python
df[['calcium', 'sodium']][1:]
```

OUTPUT

```
     calcium      sodium
1   3.669026  125.663330
2   2.789910  105.821810
3   2.939900   98.172772
4   5.426060   97.931489
5   0.715811  120.858330
6   5.652390  112.871500
7   3.571320  112.647360
8   4.300067  132.031720
9   1.369419  118.499010
10  2.550962  117.373730
11  2.894129  134.052390
12  3.664987  105.346410
13  1.362779  123.359490
14  3.718798  125.021060
15  1.865868  112.075420
16  3.272809  117.588040
17  3.917591  101.009870
```

Mixing the ways to access specific data in a dataframe can be confusing and needs practice.

Search for missing values

Some tables contain missing entries. You can check a dataframe for such missing entries. If no missing entry is found, the function `isnull` will return `False`.

PYTHON

```python
df.isnull().any()
```

OUTPUT

```
calcium    False
sodium     False
dtype: bool
```

This shows that there are no missing entries in our dataframe.

DIY3: FIND NAN IN DATAFRAME

In the Loan data set, check the entry ‘Self-employed’ for ID LP001059. It shows how a missing value is represented as ‘NaN’ (not a number).

Verify that the output of `isnull` in this case is `True`

Solution

PYTHON

```
df_loan['Self_Employed'][8]
```

OUTPUT

```
nan
```

PYTHON

```
df_loan['Self_Employed'][8:9].isnull()
```

OUTPUT

```
8    True
Name: Self_Employed, dtype: bool
```

Basic data features

Summary Statistics

To get a summary of basic data features use the function `describe`:

PYTHON

```
description = df.describe()

description
```

OUTPUT

```
          calcium      sodium
count  18.000000   18.000000
mean    3.174301  115.167484
std     1.306652   10.756852
min     0.715811   97.931489
25%     2.610699  107.385212
50%     3.364195  115.122615
75%     3.706355  122.734200
max     5.652390  134.052390
```

The `describe` function produces a new dataframe (here called 'description') that contains the number of samples, the mean, the standard deviation, minimum, 25th, 50th, 75th percentile, and the maximum value for each column of the data. Note that the indices of the rows have now been replaced by strings. To access rows, it is possible to refer to those names using the `loc` property. E.g. to access the mean of the calcium concentrations from the description, each of the following is valid:

PYTHON

```python
# Option 1
description.loc['mean']['calcium']

# Option 2
description.loc['mean'][0]

# Option 3
description['calcium']['mean']

# Option 4
description['calcium'][1]
```

OUTPUT

```
3.1743005405555555
3.1743005405555555
3.1743005405555555
3.1743005405555555
```

DIY4: PRACTICE

Use your own .csv data set to practice. (If you don't have a data set at hand, any excel table can be exported as .csv.) Read it into a dataframe, check its header, access individual values or sets of values. Create a statistics using `describe` and check for missing values using `.isnull`.

Solution

[ad libitum]

Iterating through the columns

Now we know how to access all data in a dataframe and how to get a summary statistics over each column.

Here is code to iterate through the columns and access the first two concentrations:

PYTHON

```
for col in df:

    print(df[col][0:2])
```

OUTPUT

```
0    3.455582
1    3.669026
Name: calcium, dtype: float64
0    112.69098
1    125.66333
Name: sodium, dtype: float64
```

As a slightly more complex example, we access the median ('50%') of each column in the description and add it to a list:

PYTHON

```
conc_medians = list()

for col in df:

    conc_medians.append(df[col].describe()['50%'])

print('The columns are: ', list(df.columns))
print('The medians are: ', conc_medians)
```

OUTPUT

```
The columns are:  ['calcium', 'sodium']
The medians are:  [3.3641954, 115.122615]
```

This approach is useful for data frames with a large number of columns. For instance, it is possible to then create a boxplot or histogram for the means, medians etc. of the dataframe and thus to get an overview of all (comparable) columns.

Selecting a subset based on a template

An analysis of a data set may need to be done on part of the data. This can often be formulated by using a logical condition which specifies the required subset.

For this we will assume that some of the data are labelled '0' and some are labelled '1'. Let us therefore see how to add a new column to our Evereleys data frame which contains the (in this case arbitrary) labels.

First we randomly create as many labels as we have rows in the data frame. We can use the `randint` function which we import from 'numpy.random'. `randint` in its simple form takes two arguments. First the upper bound of the integer needed, where by default it starts from zero. As Python is exclusive on the upper bound, providing '2' will thus yield either '0' or '1' only.

PYTHON ❮ ❯

```
from numpy.random import randint

no_rows = len(df)

randomLabel = randint(2, size=no_rows)

print('Number of rows:  ', no_rows)
print('Number of Labels:', len(randomLabel))
print('Labels:          ', randomLabel)
```

OUTPUT ❮ ❯

```
Number of rows:   18
Number of Labels: 18
Labels:           [0 0 1 0 1 0 1 1 0 0 1 0 0 1 1 1 1 0]
```

Note how we obtain the number of rows (18) using `len` and do not put it directly into the code.

Now we create a new data column in our `df` dataframe which contains the labels. To create a new column, you can simply refer to a column name that does not yet exist and assign values to it. Let us call it 'gender', assuming that '0' represents male and '1' represents female.

As gender specification can include more than two labels, try to create a column with more than two randomly assigned labels e.g. (0, 1, 2).

PYTHON ❮ ❯

```
df['gender'] = randomLabel

df.head()
```

OUTPUT ❮ ❯

```
    calcium      sodium  gender
0  3.455582  112.690980       0
1  3.669026  125.663330       0
2  2.789910  105.821810       1
3  2.939900   98.172772       0
4  5.426060   97.931489       1
```

Now we can use the information contained in 'gender' to filter the data by gender. To achieve this, we use a conditional statement. Let us check which of the rows are labelled as '1':

PYTHON ❮ ❯

```
df['gender'] == 1
```

OUTPUT

```
0     False
1     False
2      True
3     False
4      True
5     False
6      True
7      True
8     False
9     False
10     True
11    False
12    False
13     True
14     True
15     True
16     True
17    False
Name: gender, dtype: bool
```

If we assign the result of the conditional statement (Boolean True or False) to a variable, then this variable can act as a template to filter the data. If we call the data frame with that variable, we will only get the rows where the condition was found to be True:

PYTHON

```python
df_female = df['gender'] == 1

df[df_female]
```

OUTPUT

```
     calcium      sodium  gender
2   2.789910  105.821810       1
4   5.426060   97.931489       1
6   5.652390  112.871500       1
7   3.571320  112.647360       1
10  2.550962  117.373730       1
13  1.362779  123.359490       1
14  3.718798  125.021060       1
15  1.865868  112.075420       1
16  3.272809  117.588040       1
```

Using the Boolean, we only pick the rows that are labelled ‘1’ and thus get a subset of the data according to the label.

DIY5: USING A TEMPLATE

Modify the code to calculate the number of samples labelled 0 and check the number of rows of that subset.

Solution

PYTHON

```python
from numpy.random import randint
no_rows = len(df)
randomLabel = randint(2, size=no_rows)
df['gender'] = randomLabel
df_male = df['gender'] == 0
no_males = len(df[df_male])
print(no_males, 'samples are labelled "male".')
```

OUTPUT

```
11 samples are labelled "male".
```

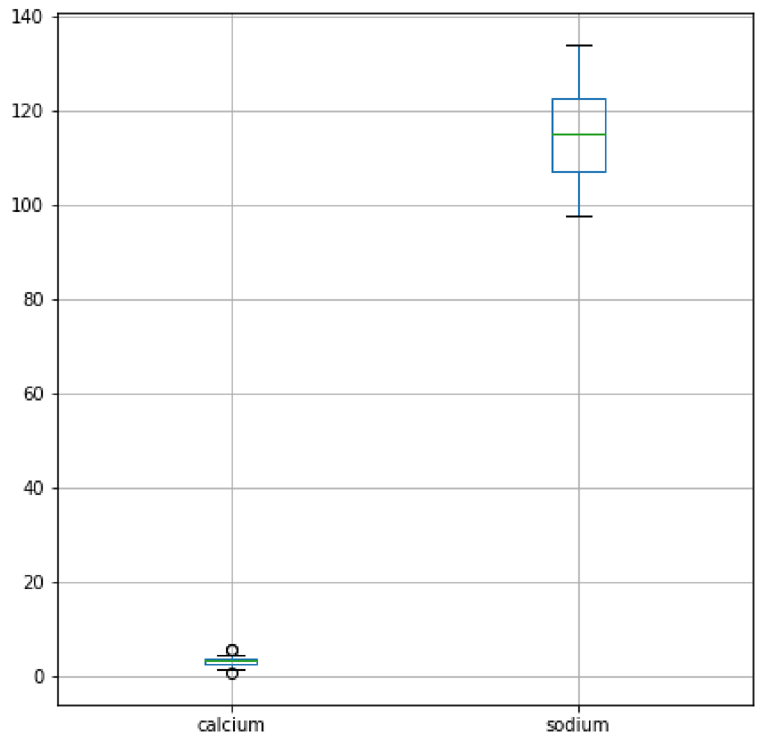
Visualisation of data

It is easy to see from the numbers that the concentrations of sodium are much higher than that of calcium. However, to also compare the medians, percentiles and the spread of the data it is better to use visualisation.

The simplest way of visualisation is to use Pandas functionality which offers direct ways of plotting. Here is an example where a boxplot is created for each column:

PYTHON

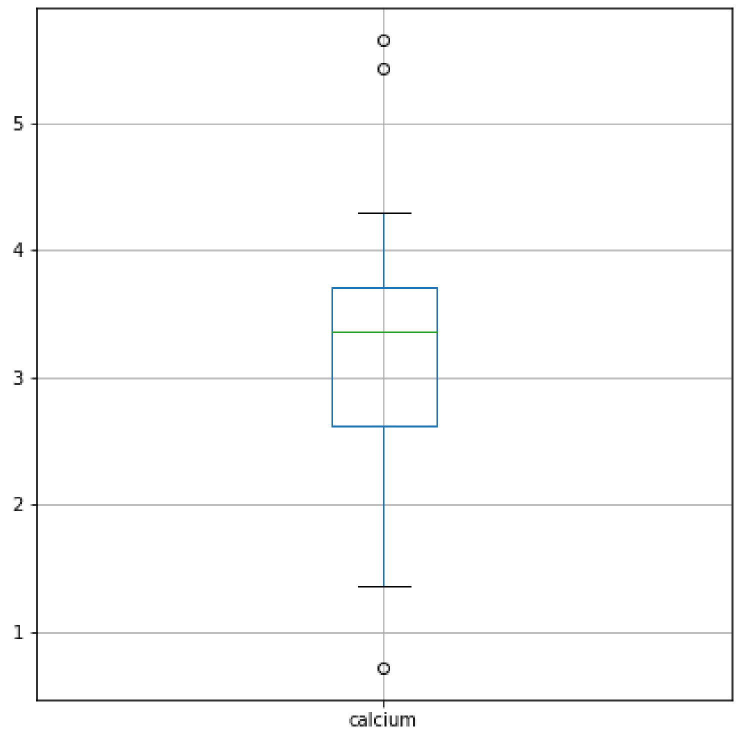
```python
df = read_csv("data/everleys_data.csv")
df.boxplot()
```



By default, boxplots are shown for all columns if no further argument is given to the function (empty round parenthesis). As the calcium plot is rather squeezed we may wish to see it individually. This can be done by specifying the calcium column as an argument:

PYTHON

```python
# Boxplot of calcium results
df.boxplot(column='calcium')
```



Plots using Matplotlib

> Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

The above is an easy way to create boxplots directly on the dataframe. It is based on the library Matplotlib and specifically uses the **pyplot library**. For simplicity, the code is put in a convenient Pandas function.

However, we are going to use **Matplotlib** extensively later on in the course, and we therefore now introduce the direct, generic way of using it.

For this, we import the function `subplots` from the pyplot library:

PYTHON

```python
from matplotlib.pyplot import subplots, show
```

The way to use `subplots` is to first set up a figure environment (below it is called 'fig') and an empty coordinate system (below called 'ax'). The plot is then done using one of the many methods available in Matplotlib. We apply it to the coordinate system 'ax'.

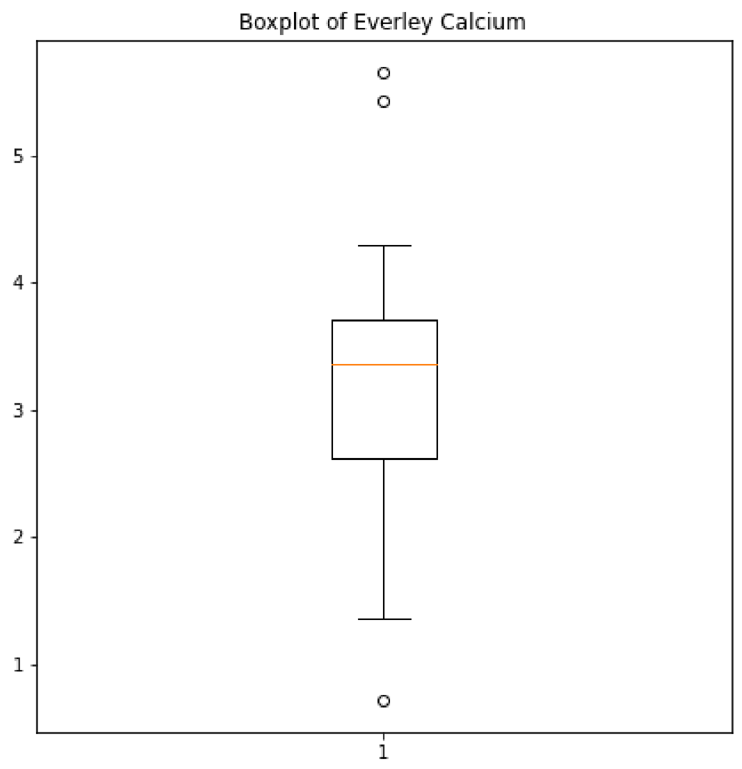
As an example, let us create a boxplot of the calcium variable. As an argument of the function we need to specify the data. We can use the values of the 'calcium' concentrations from the column with that name:

PYTHON

```
fig, ax = subplots()

ax.boxplot(df['calcium'])
```

PYTHON

```
ax.set_title('Boxplot of Everley Calcium')

show()
```



Note how following the actual plot we define the title of the plot by referring to the same coordinate system `ax`.

The value of `subplots` becomes apparent when we try to create more than one plot in a single figure.

Here is an example to create two boxplots next to each other. The keyword arguments to use is 'ncols' which is the number of figures per row. 'ncols=2' indicates that you want to have two plots next to each other.

PYTHON

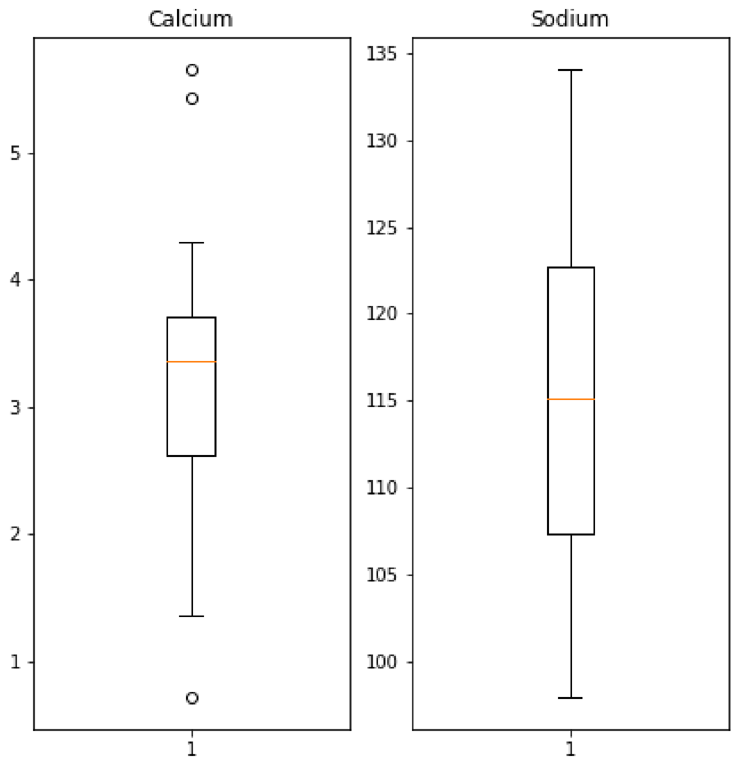
```
fig, ax = subplots(ncols=2)

ax[0].boxplot(df['calcium'])
```

PYTHON

```
ax[0].set_title('Calcium')

ax[1].boxplot(df['sodium'])
```

PYTHON

```
ax[1].set_title('Sodium');

show()
```



Note that you now have to refer to each of the subplots by indexing the coordinate system ‘ax’. This figure gives a good overview of the Everley’s data.

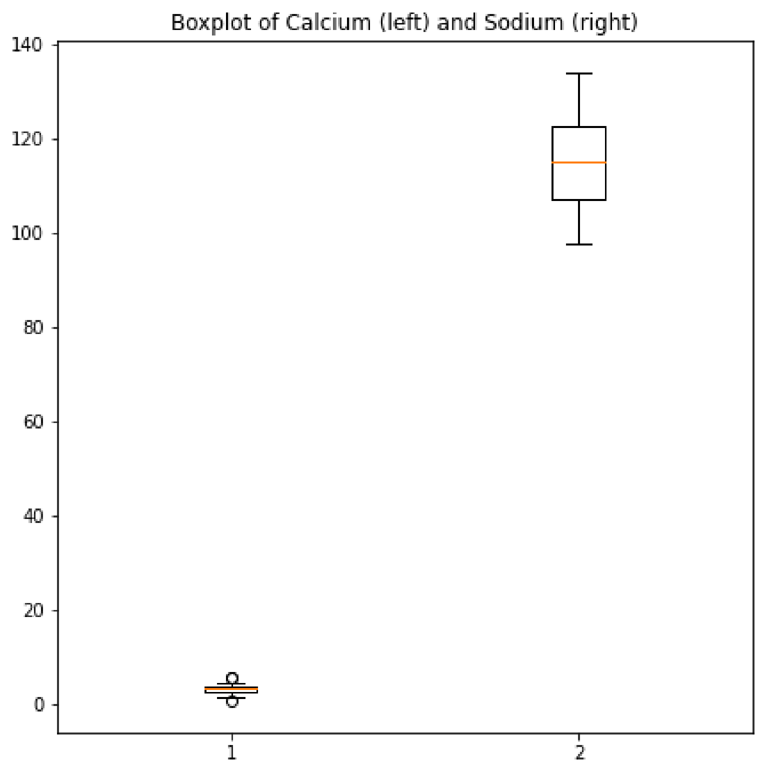
If you prefer to have the boxplots of both columns in a single figure, that can also be done:

PYTHON

```
fig, ax = subplots(ncols=1, nrows=1)

ax.boxplot([df['calcium'], df['sodium']], positions=[1, 2])
```

PYTHON

```python
ax.set_title('Boxplot of Calcium (left) and Sodium (right)')

show()
```



DIY6: BOXPLOT FROM LOAN DATA

Plot the boxplots of the 'ApplicantIncome' and the 'CoapplicantIncome' in the Loan data using the above code.

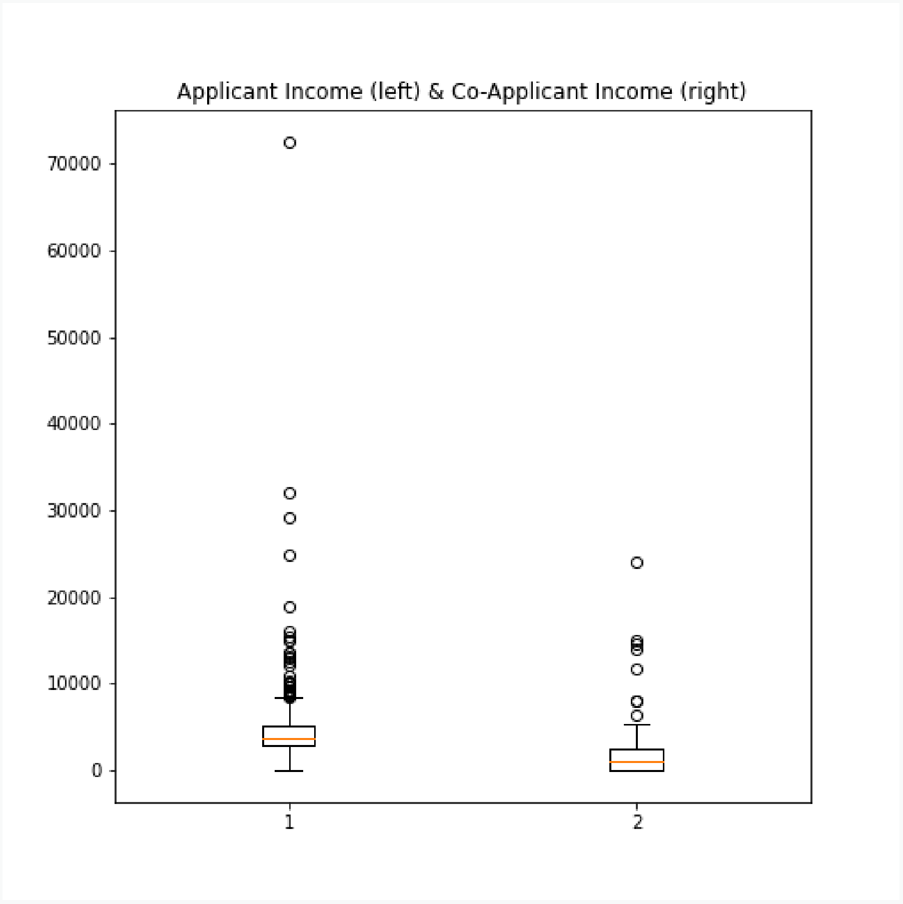
Solution

PYTHON

```python
fig, ax = subplots(ncols=1, nrows=1)
ax.boxplot([df_loan['ApplicantIncome'], df_loan['CoapplicantIncome']], positions=[1, 2])
```

PYTHON

```python
ax.set_title('Applicant Income (left) & Co-Applicant Income (right)');

show()
```

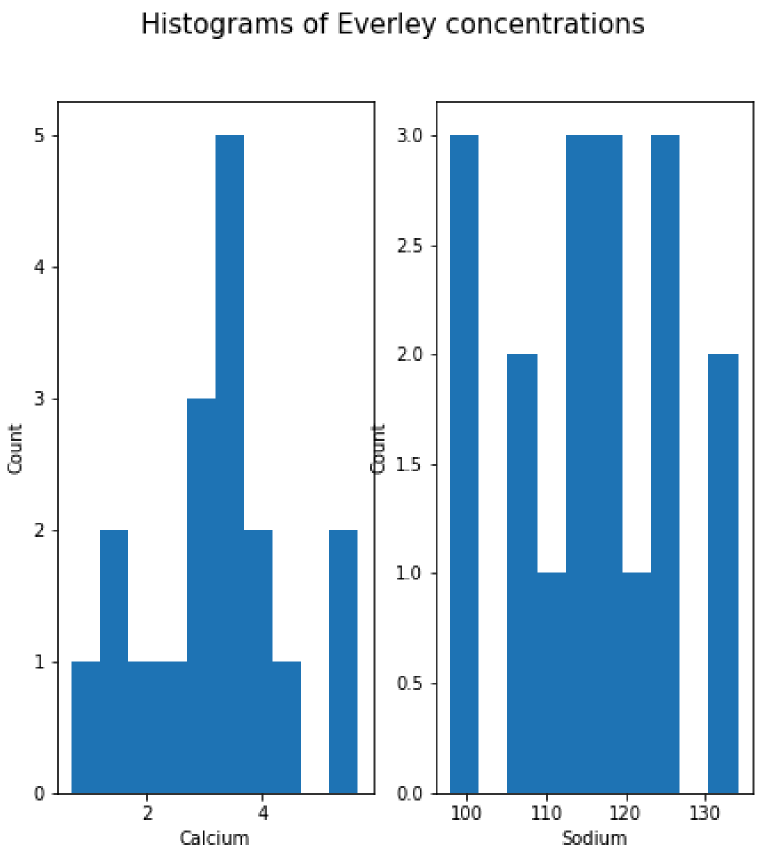


Histogram

Another good overview is the histogram: Containers or 'bins' are created over the range of values found within a column and the count of the values for each bin is plotted on the vertical axis.

PYTHON

```python
fig, ax = subplots(ncols=2, nrows=1)

ax[0].hist(df['calcium'])
ax[0].set(xlabel='Calcium', ylabel='Count');

ax[1].hist(df['sodium'])
ax[1].set(xlabel='Sodium', ylabel='Count');

fig.suptitle('Histograms of Everley concentrations', fontsize=15);

show()
```

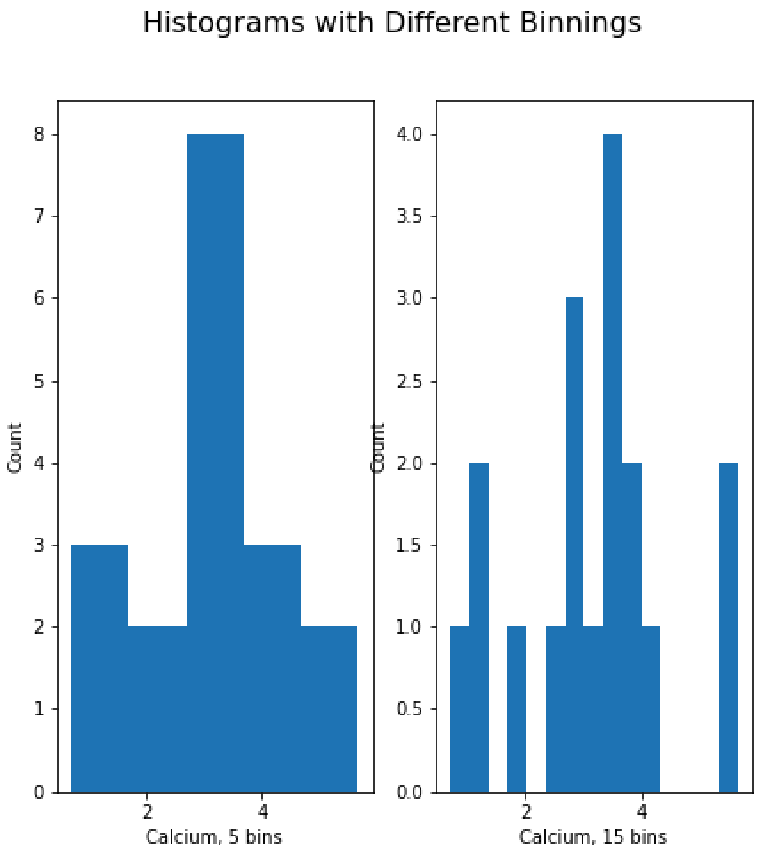


This also shows how to add labels to the axes and a title to the overall figure.

This uses the default value for the generation of the bins. It is set to 10 bins over the range of which values are found. The number of bins in the histogram can be changed using the keyword argument ‘bins’:

PYTHON

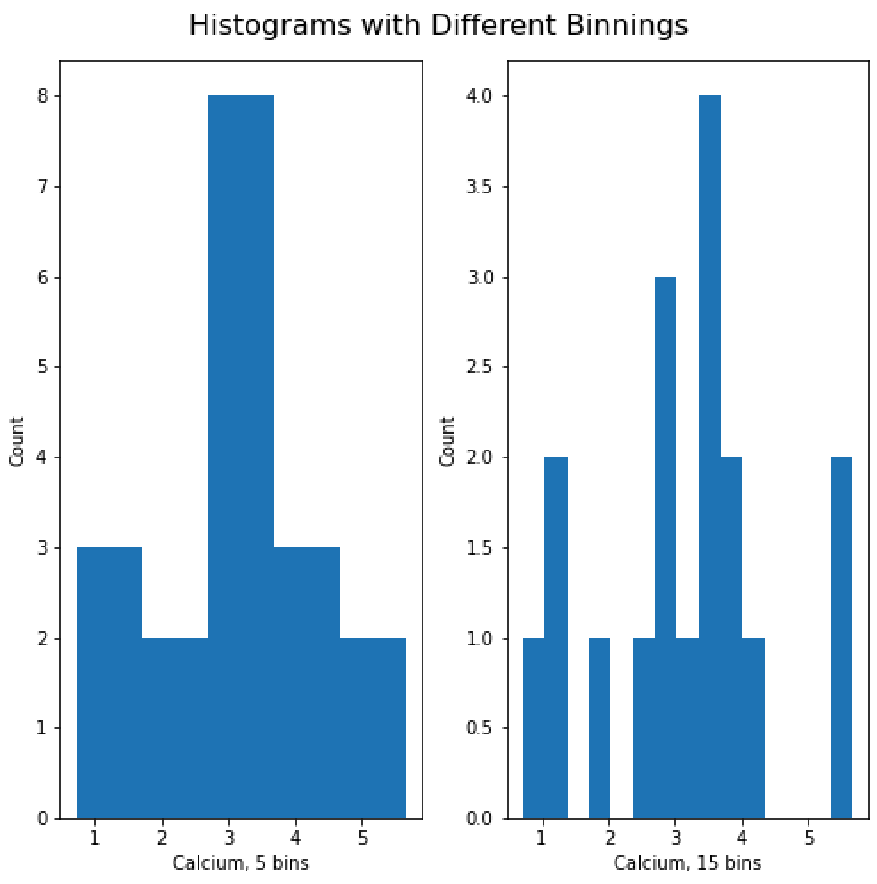
```python
fig, ax = subplots(ncols=2, nrows=1)

ax[0].hist(df['calcium'], bins=5)
ax[0].set(xlabel='Calcium, 5 bins', ylabel='Count');

ax[1].hist(df['calcium'], bins=15)
ax[1].set(xlabel='Calcium, 15 bins', ylabel='Count');
fig.suptitle('Histograms with Different Binnings', fontsize=16);

show()
```



Note how the y-label of the right figure is not placed well. To correct for the placement of the labels and the title, you can use `tight_layout` on the figure:

PYTHON

```python
fig, ax = subplots(ncols=2, nrows=1)

ax[0].hist(df['calcium'], bins=5)
ax[0].set(xlabel='Calcium, 5 bins', ylabel='Count');

ax[1].hist(df['calcium'], bins=15)
ax[1].set(xlabel='Calcium, 15 bins', ylabel='Count');
fig.suptitle('Histograms with Different Binnings', fontsize=16);
fig.tight_layout()

show()
```



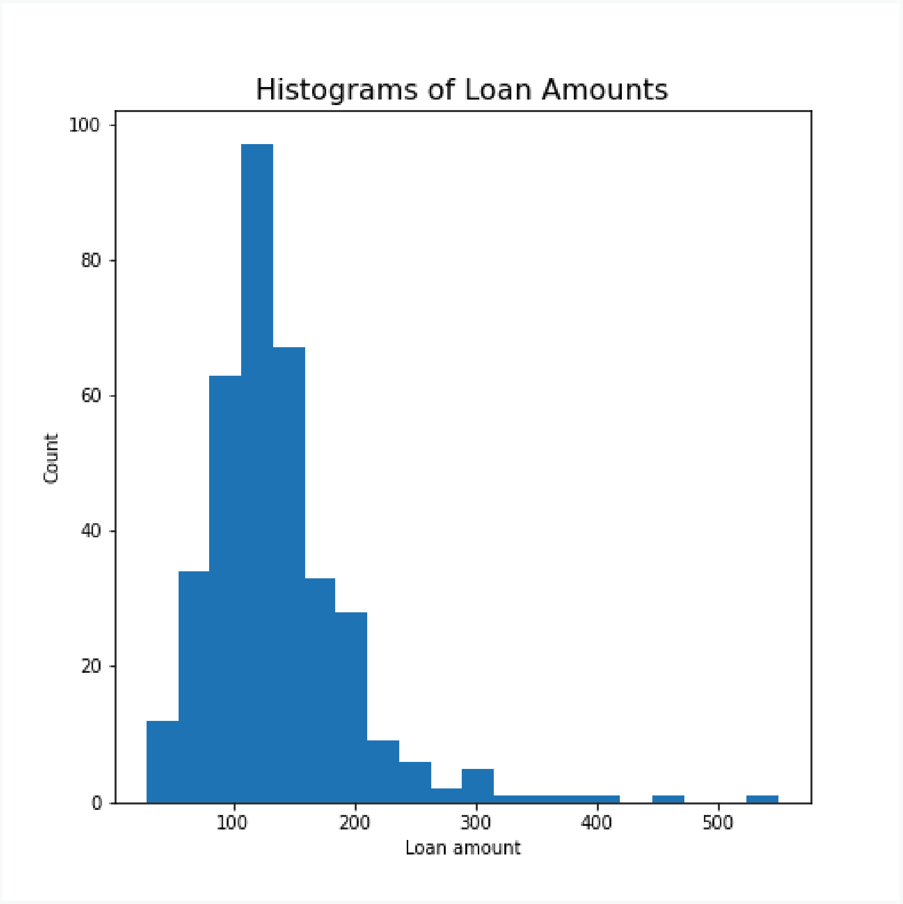
DIY7: CREATE THE HISTOGRAM OF A COLUMN

Take the loan data and display the histogram of the loan amount that people asked for. (Loan amounts are divided by 1000, i.e. in k£ on the horizontal axis). Use e.g. 20 bins.

Solution

PYTHON

```python
# Histogram of loan amounts in k£
fig, ax = subplots()
ax.hist(df_loan['LoanAmount'], bins=20)
ax.set(xlabel='Loan amount', ylabel='Count');
ax.set_title('Histograms of Loan Amounts', fontsize=16);

show()
```



Handling the Diabetes Data Set

We now return to the data set that started our enquiry into the handling of data in a dataframe.

We will now:

- Import the diabetes data from 'sklearn'
- Check the shape of the dataframe and search for NANs
- Get a summary plot of one of its statistical quantities (e.g. mean) for all columns

First we import the data set and check its head. Wait until the numbers show below the code, it might take a while.

PYTHON

```python
from sklearn import datasets

diabetes = datasets.load_diabetes()

X = diabetes.data

from pandas import DataFrame

df_diabetes = DataFrame(data=X)

df_diabetes.head()
```

OUTPUT

```
          0         1         2  ...         7         8         9
0  0.038076  0.050680  0.061696  ... -0.002592  0.019907 -0.017646
1 -0.001882 -0.044642 -0.051474  ... -0.039493 -0.068332 -0.092204
2  0.085299  0.050680  0.044451  ... -0.002592  0.002861 -0.025930
3 -0.089063 -0.044642 -0.011595  ...  0.034309  0.022688 -0.009362
4  0.005383 -0.044642 -0.036385  ... -0.002592 -0.031988 -0.046641

[5 rows x 10 columns]
```

If you don't see all columns, use the cursor to scroll to the right. Now let's check the number of columns and rows.

PYTHON

```python
no_rows = len(df_diabetes)
no_cols = len(df_diabetes.columns)

print('Rows:', no_rows, 'Columns:', no_cols)
```

OUTPUT

```
Rows: 442 Columns: 10
```

There are 442 rows organised in 10 columns.

To get an overview, let us extract the mean of each column using 'describe' and plot all means as a bar chart. The Matplotlib function to plot a bar chart is `bar`:

PYTHON

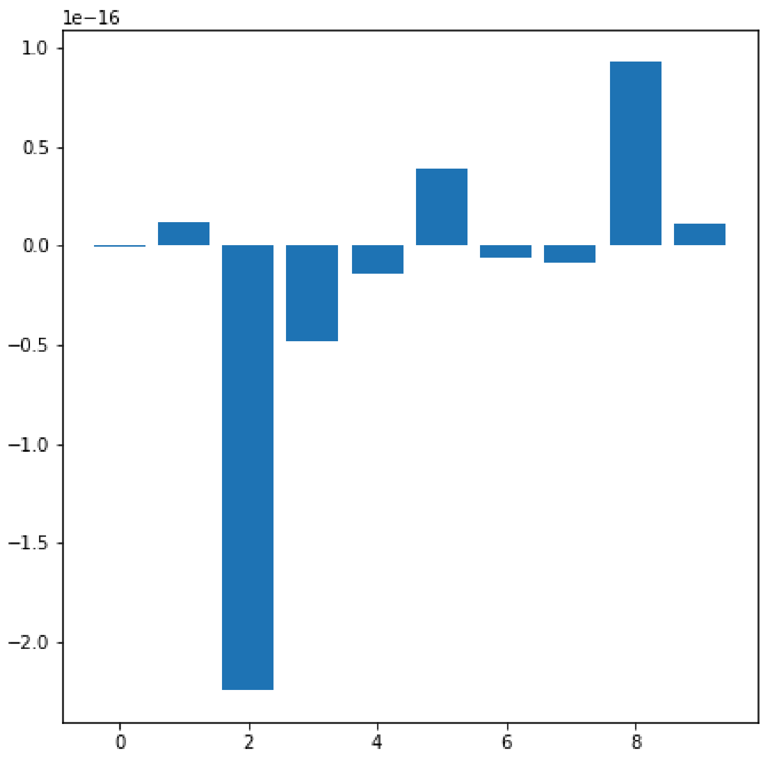
```python
conc_means = list()

for col in df_diabetes:
    conc_means.append(df_diabetes[col].describe()['mean'])

print('The columns are: ', list(df_diabetes.columns))
print('The medians are: ', conc_means, 2)
```

OUTPUT

```
The columns are:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
The medians are:  [-2.511816797794472e-19, 1.2307902309192911e-17, -2.2455642172282577e-16, -4.79757008378
```

PYTHON

```
fig, ax = subplots()

bins = range(10)

ax.bar(bins, conc_means);

show()
```



The bars in this plot go up and down. Note, however, that the vertical axis has values ranging from $-10^{(-16)}$ to $+10^{(-16)}$. This means that for all practical purposes all means are zero. This is not a coincidence. The original values have been normalised to zero mean for the purpose of applying some machine learning algorithm to them.

In this example, we see how important it is to check the data before working with them.

Exercises

END OF CHAPTER EXERCISES

Download the cervical cancer data set provided, import it using `read_csv`.

1. How many rows and columns are there?
2. How many columns contain floating point numbers (float64)?
3. How many of the subjects are smokers?
4. Calculate the percentage of smokers
5. Plot the age distribution (with e.g. 50 bins)
6. Get the mean and standard distribution of age of first sexual intercourse

Solution

Q1

PYTHON

```
df_cervix = read_csv("data/cervical_cancer.csv")

df_cervix.head(10)


cervix_rows, cervix_cols = len(df_cervix), len(df_cervix.columns)

print('Number of rows:', cervix_rows)
print('Number of columns:', cervix_cols)
```

OUTPUT

```
   Age  Number of sexual partners  ...  Citology  Biopsy
0   18                        4.0  ...         0       0
1   15                        1.0  ...         0       0
2   52                        5.0  ...         0       0
3   46                        3.0  ...         0       0
4   42                        3.0  ...         0       0
5   51                        3.0  ...         0       1
6   26                        1.0  ...         0       0
7   45                        1.0  ...         0       0
8   44                        3.0  ...         0       0
9   27                        1.0  ...         0       0

[10 rows x 34 columns]
Number of rows: 668
Number of columns: 34
```

Q2

PYTHON

```
df_types = df_cervix.dtypes == 'float64'

print('There are', df_types.sum(), 'columns with floating point numbers')
```

OUTPUT

```
There are 24 columns with floating point numbers
```

PYTHON

```
df_types
```

OUTPUT

```
Age                                   False
Number of sexual partners              True
First sexual intercourse               True
Num of pregnancies                     True
Smokes                                 True
Smokes (years)                         True
Smokes (packs/year)                    True
Hormonal Contraceptives                True
Hormonal Contraceptives (years)        True
IUD                                    True
IUD (years)                            True
STDs                                   True
STDs (number)                          True
STDs:condylomatosis                    True
STDs:cervical condylomatosis           True
STDs:vaginal condylomatosis            True
STDs:vulvo-perineal condylomatosis     True
STDs:syphilis                          True
STDs:pelvic inflammatory disease       True
STDs:genital herpes                    True
STDs:molluscum contagiosum             True
STDs:AIDS                              True
STDs:HIV                               True
STDs:Hepatitis B                       True
STDs:HPV                               True
STDs: Number of diagnosis             False
Dx:Cancer                             False
Dx:CIN                                False
Dx:HPV                                False
Dx                                    False
Hinselmann                            False
Schiller                              False
Citology                              False
Biopsy                                False
dtype: bool
```

Q3

PYTHON

```python
for col in df_cervix:

    print(type(df_cervix[col][0]))


cervix_smoker = df_cervix['Smokes'] == 1.0
```

OUTPUT

```
<class 'numpy.int64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
```

Q4

PYTHON

```python
print('There are', cervix_smoker.sum(), 'smokers.')
print('This is', round(100*cervix_smoker.sum() / cervix_rows, 1), '% of the total.')
```

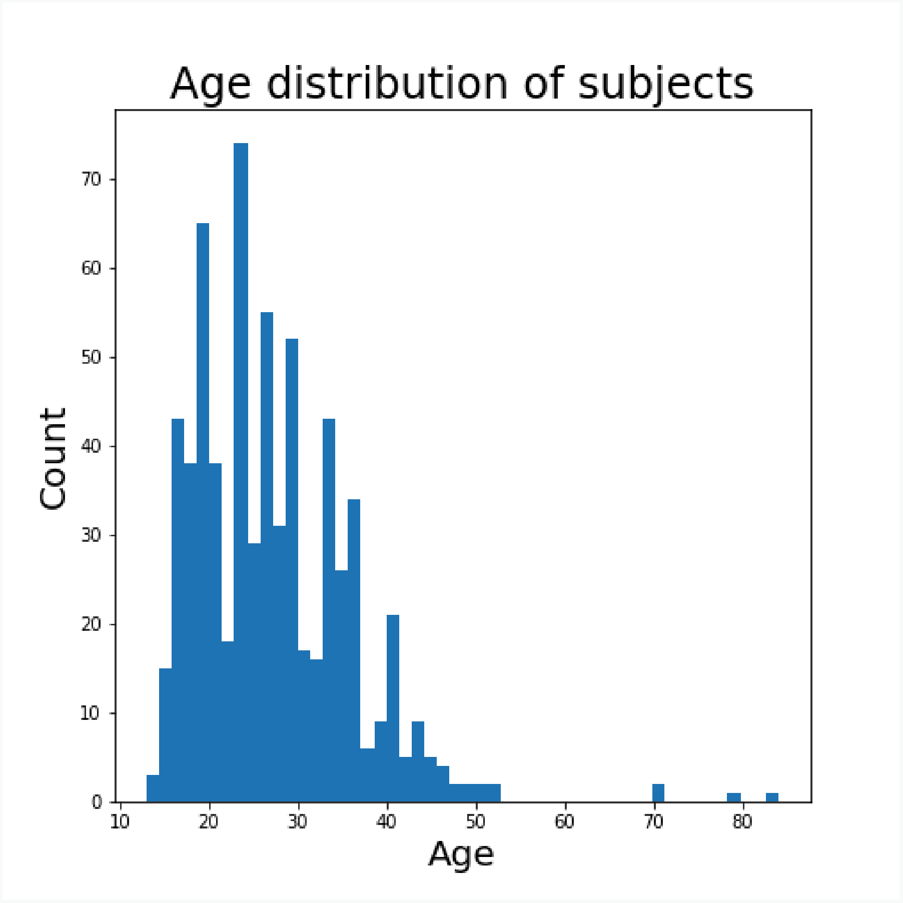
OUTPUT

```
There are 96 smokers.
This is 14.4 % of the total.
```

Q5

PYTHON

```python
fig, ax = subplots()

ax.hist(df_cervix['Age'], bins=50)
ax.set_xlabel('Age', fontsize=20)
ax.set_ylabel('Count', fontsize=20)
ax.set_title('Age distribution of subjects',  fontsize=24);
show()
```



Q6

PYTHON

```python
int_mean = df_cervix['First sexual intercourse'].mean()

int_std = df_cervix['First sexual intercourse'].std()

print('Mean of age of first sexual intercourse: ', round(int_mean, 1))
print('Standard distribution of age of first sexual intercourse: ', round(int_std, 1))
```

OUTPUT

```
Mean of age of first sexual intercourse:  17.1
Standard distribution of age of first sexual intercourse:  2.9
```

KEY POINTS

- `Pandas` package contains useful functions to work with dataframes.
- `iloc` property is used to index and slice a dataframe.
- `describe` function is used to get a summary of basic data features.
- The simplest way of visualisation is to use Pandas functionality.
- `Matplotlib` is a comprehensive library for creating static, animated, and interactive visualizations in Python.

Content from Data Frames - Part 2

Last updated on 2024-05-24 | Edit this page

Download Chapter PDF

Download Chapter notebook (ipynb)

Mandatory Lesson Feedback Survey

OVERVIEW

Questions

- What is bivariate or multivariate analysis?
- How are bivariate properties of data interpreted?
- How can a bivariate quantity be explained?
- When to use the correlation matrix?
- What are the ways to study relationships in data?

Objectives

- Practise working with Pandas dataframes and Numpy arrays.
- Bivariate analysis of Pandas dataframe / Numpy array.
- The Pearson correlation coefficient (PCC).
- Correlation Matrix as an example of bivariate summary statistics.

PREREQUISITES

- Python Arrays
- Basic Statistics, in particular the correlation coefficient
- Pandas dataframes: import and handling

The following cell contains functions that need to be imported, please execute it before continuing with the Introduction.

PYTHON

```python
# To import data from a csv file into a Pandas dataframe
from pandas import read_csv

# To import a dataset from scikit-learn
from sklearn import datasets

# To create figure environments and plots
from matplotlib.pyplot import subplots, show

# Specific numpy functions, description in the main body
from numpy import corrcoef, fill_diagonal, triu_indices, arange
```

NOTE

In many online tutorials you can find the following convention when importing functions:

PYTHON

```python
import numpy as np
import pandas as pd
```

(or similar). In this case, the whole library is imported and any function in that library is then available using e.g. `pd.read_csv(my_file)`

We don't recommend this as the import of the whole library uses a lot of working memory (e.g. on the order of 100 MB for Numpy).

Introduction

In the previous lesson we have obtained some basic data quantifications using `describe`. Each of these quantities was calculated for individual columns (where each column contained a different measured variable). However, in data analysis in general, and especially in machine learning, the main point of analysis is to also try and exploit the presence of information that lies in relationships *between* variables, i.e. columns in our data.

Quantities that are based on data from two variables are referred to as **bivariate** measures. Analysis that makes use of bivariate (and potentially higher order) quantities is referred to as bivariate or (in general) **multivariate data analysis**.

When we combine uni- and multivariate analysis we can get an excellent overview of basic properties of a dataset.

EXAMPLE: THE DIABETES DATA SET

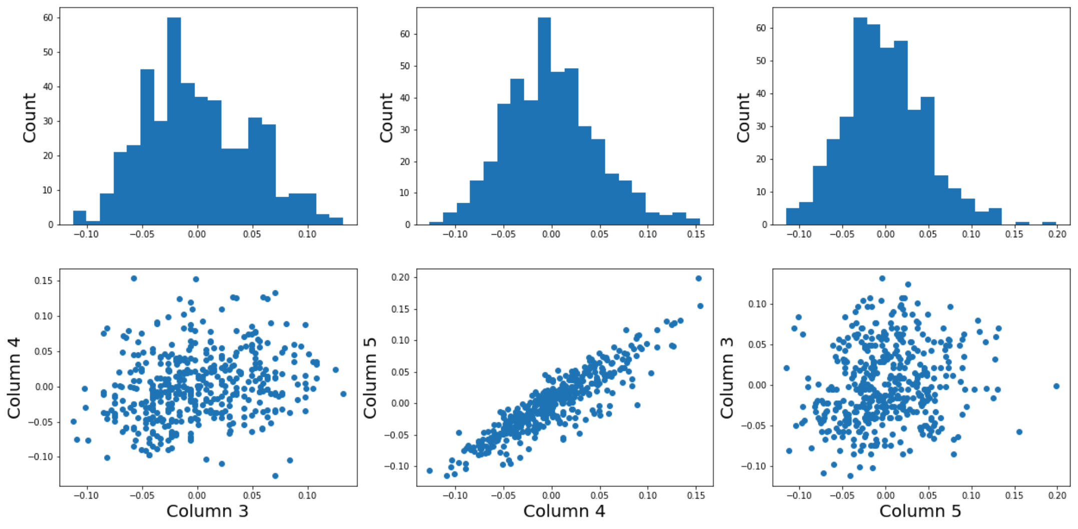
Using the diabetes dataset (introduced in the previous lesson), let us look at the data from three of its columns: The upper row of the below figure shows three histograms. A histogram is a summary plot of the recordings of a single variable. The histograms of columns with indices 3, 4, and 5 have similar means and variances but that is due to prior normalisation. The shapes differ but this does not tell us anything about a relationship between the measurements.

One thing that we want to know before we start to apply any machine learning is whether or not there is evidence of relationships between the individual variables in a dataframe. One of the potential relationships is that the variables are 'similar'. One way to check for the similarity between variables in a dataset is to create a scatter plot. The bottom row of the figure below contains the three scatter plots between variables used to create the histograms in the top row.

(Please execute the code to create the figures. We will describe the scatter plot and its features later on.)

PYTHON

```python
# Figure Code

diabetes = datasets.load_diabetes()

diabetes_data = diabetes.data

fig, ax = subplots(figsize=(21, 10), ncols=3, nrows=2)

# Histograms
ax[0,0].hist(diabetes_data[:,3], bins=20)
ax[0,0].set_ylabel('Count', fontsize=20)

ax[0,1].hist(diabetes_data[:,4], bins=20)
ax[0,1].set_ylabel('Count', fontsize=20)

ax[0,2].hist(diabetes_data[:,5], bins=20)
ax[0,2].set_ylabel('Count', fontsize=20)

# Scatter plots
ax[1,0].scatter(diabetes_data[:,3], diabetes_data[:,4]);
ax[1,0].set_xlabel('Column 3', fontsize=20)
ax[1,0].set_ylabel('Column 4', fontsize=20)

ax[1,1].scatter(diabetes_data[:,4], diabetes_data[:,5]);
ax[1,1].set_xlabel('Column 4', fontsize=20)
ax[1,1].set_ylabel('Column 5', fontsize=20)

ax[1,2].scatter(diabetes_data[:,5], diabetes_data[:,3]);
ax[1,2].set_xlabel('Column 5', fontsize=20)
ax[1,2].set_ylabel('Column 3', fontsize=20);

show()
```



When plotting the data against each other in pairs (lower row), data column 3 versus column 4 (left) and column 5 versus 3 (right) both show a fairly uniform circular distribution of points. This is what would be expected if the data in the two columns were independent of each other.

In contrast, column 4 versus 5 (centre) shows an elliptic, pointed shape along the main diagonal. This shows that something particular goes on between these data sets. Specifically, it indicates that the two variables recorded in these columns (indices 4 and 5) are not independent of each other. They are more similar than would be expected for independent variables.

In this lesson we aim to get an overview of the similarities in a data set. We first introduce bivariate visualisation using Matplotlib. Then we use Numpy functions to calculate correlation coefficients and the correlation matrix as an introduction to multivariate analysis. Combined with the basic statistics obtained in the previous lesson we can get a good overview of a high-dimensional data set before applying machine learning algorithms.

Work Through: Properties of a Data Set

Univariate properties

For recordings of variables that are contained e.g. in the columns of a dataframe, we often assume the independence of samples: the measurement in one row does not depend on the recording in another row. Therefore all results of the features obtained e.g. under `describe` will not depend on the order of the rows. Also, while the numbers obtained from different rows can be similar (or even the same) by chance, there is no way to *predict* the values in one row from the values of another row.

When comparing different variables arranged in columns, in contrast, this is not necessarily so. (We assume here that they are consistent, e.g. all values in a single row obtained from the same subject.) The values in one column can be related to the numbers in another column and specifically they can show degrees of similarity. If, for instance, we have a number of subjects investigated some of who have an inflammatory disease and some of who are healthy controls, an inflammatory marker might be increased in the diseased subjects. If several markers are recorded from each subject (i.e. more than one column in the data frame), the values of several inflammatory markers may be elevated simultaneously in the diseased subjects. Thus, the profiles of these markers across the whole group will show a certain similarity.

The goal of multivariate data analysis is to find out whether similarities (or, in general, any relationships) between recorded variables exist.

Let us first import a demo data set and check its basic statistics.
For a work through example, let us work with the ‘patients’ data set. We import the data from the .csv file using `read_csv` from Pandas into a dataframe. We then check the number of columns and rows using the `len` function. We also check the data type of each column to find out which columns can be used for quantitative analysis.

PYTHON

```
# Please adjust path according to operating system & personal path to file
df = read_csv('data/patients.csv')

df.head()
print('Number of columns: ', len(df.columns))
print('Number of rows: ', len(df))
df.head()
```

OUTPUT

```
   Age  Height  Weight  Systolic  Diastolic  Smoker  Gender
0   38      71   176.0     124.0       93.0       1    Male
1   43      69   163.0     109.0       77.0       0    Male
2   38      64   131.0     125.0       83.0       0  Female
3   40      67   133.0     117.0       75.0       0  Female
4   49      64   119.0     122.0       80.0       0  Female
Number of columns:  7
Number of rows:  100
   Age  Height  Weight  Systolic  Diastolic  Smoker  Gender
0   38      71   176.0     124.0       93.0       1    Male
1   43      69   163.0     109.0       77.0       0    Male
2   38      64   131.0     125.0       83.0       0  Female
3   40      67   133.0     117.0       75.0       0  Female
4   49      64   119.0     122.0       80.0       0  Female
```

PYTHON

```
print('The columns are of the following data types:')
df.dtypes
```

OUTPUT

```
The columns are of the following data types:
Age            int64
Height         int64
Weight       float64
Systolic     float64
Diastolic    float64
Smoker         int64
Gender        object
dtype: object
```

Out of the seven columns, three containing integers, three containing floating point (decimal) numbers, and the last one containing gender specification as ‘female’ or ‘male’. We note that the sixth column contains a binary classification. Numerical analysis can thus be restricted to columns with indices 0 to 4.

DIY1: UNIVARIATE PROPERTIES OF THE PATIENTS DATA

1. Get the basic statistical properties of first five columns from the ‘describe’ function.
2. Create a barchart of the means of each column. To access a row by its name you can use the convention ‘df_describe.loc[’name’]’.
3. **Optional:** In the bar chart of the means, try to add the standard deviation as an errorbar, using the keyword argument `yerr` in the form ‘yerr = df_describe.loc[’std’]’.

Solution

PYTHON

```python
df = read_csv('data/patients.csv')
df_describe = df.iloc[:, :5].describe()
df_describe.round(2)
```

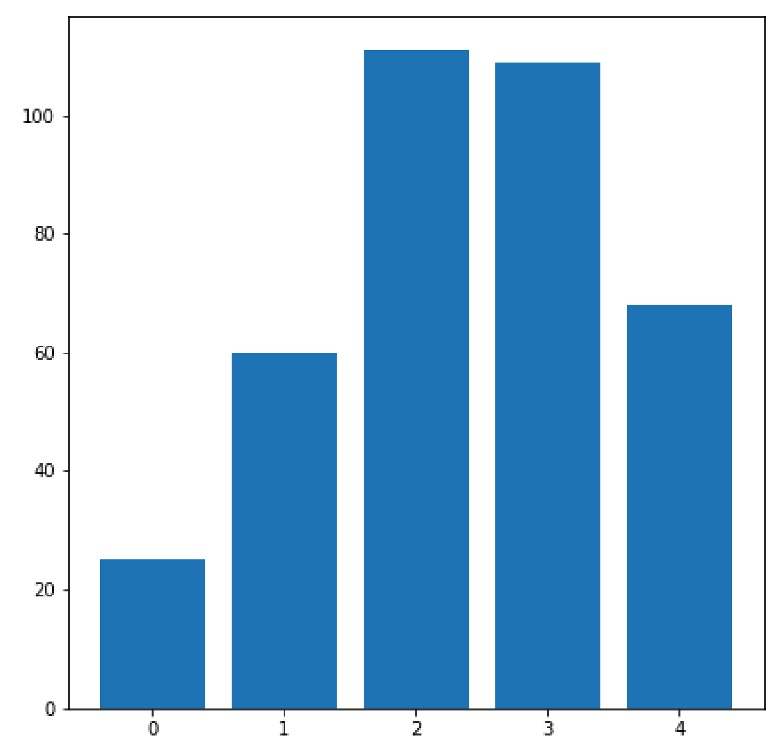
OUTPUT

```
         Age  Height  Weight  Systolic  Diastolic
count  100.00  100.00  100.00    100.00     100.00
mean    38.28   67.07  154.00    122.78      82.96
std      7.22    2.84   26.57      6.71       6.93
min     25.00   60.00  111.00    109.00      68.00
25%     32.00   65.00  130.75    117.75      77.75
50%     39.00   67.00  142.50    122.00      81.50
75%     44.00   69.25  180.25    127.25      89.00
max     50.00   72.00  202.00    138.00      99.00
```

Solution

PYTHON

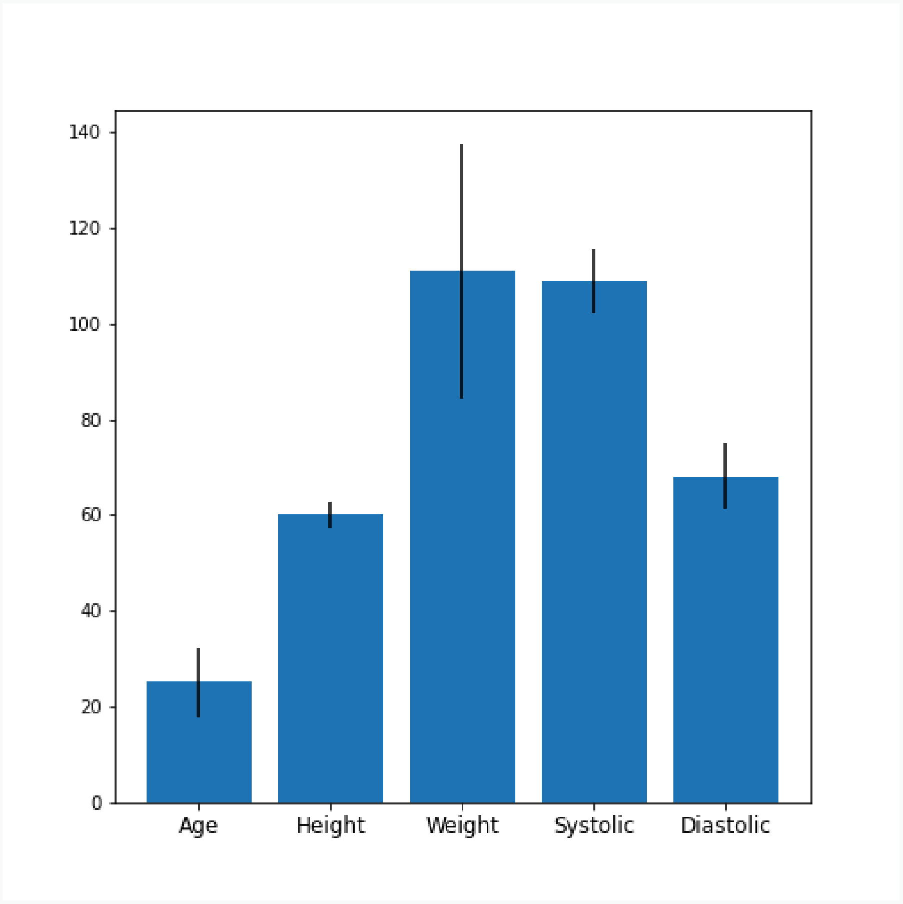
```
fig, ax = subplots()
bins = arange(5)
ax.bar(bins, df_describe.loc['min'])
show()
```



Solution

PYTHON

```
fig, ax = subplots()
bins = arange(5)
ax.bar(bins, df_describe.loc['min'], yerr=df_describe.loc['std'])
ax.set_xticks(bins)
ax.set_xticklabels(df.columns[:5], fontsize=12);
show()
```

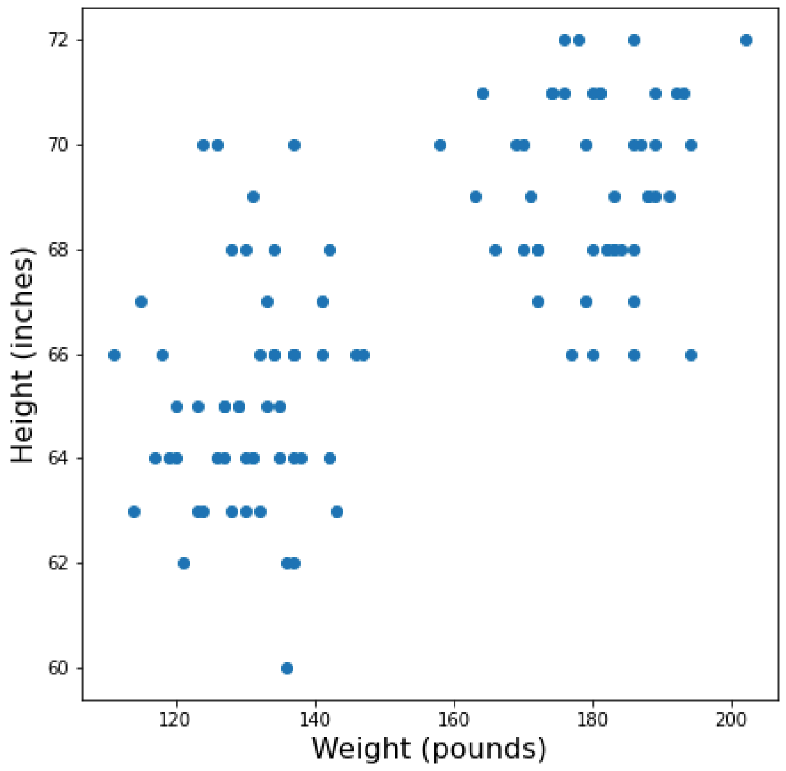


Visual Search for Similarity: the Scatter Plot

In Matplotlib, the function `scatter` allows plotting of one variable against the other. This is a common way to visually check for relationships between individual columns in a dataframe.

PYTHON

```
# Scatter plot
fig, ax = subplots();

ax.scatter(df['Weight'], df['Height']);
ax.set_xlabel('Weight (pounds)', fontsize=16)
ax.set_ylabel('Height (inches)', fontsize=16)

show()
```



The data points appear to be grouped into two clouds. We will not deal with this qualitative aspect further at present. Grouping will be discussed as Unsupervised Machine Learning or Clustering later in the Course.

However, from the plot one might also suspect that there is a trend of heavier people being taller. For instance, we note that there are no points in the lower right corner of the plot (weight >160 pounds and height < 65 inches).

DIY2: SCATTER PLOT FROM THE PATIENTS DATA

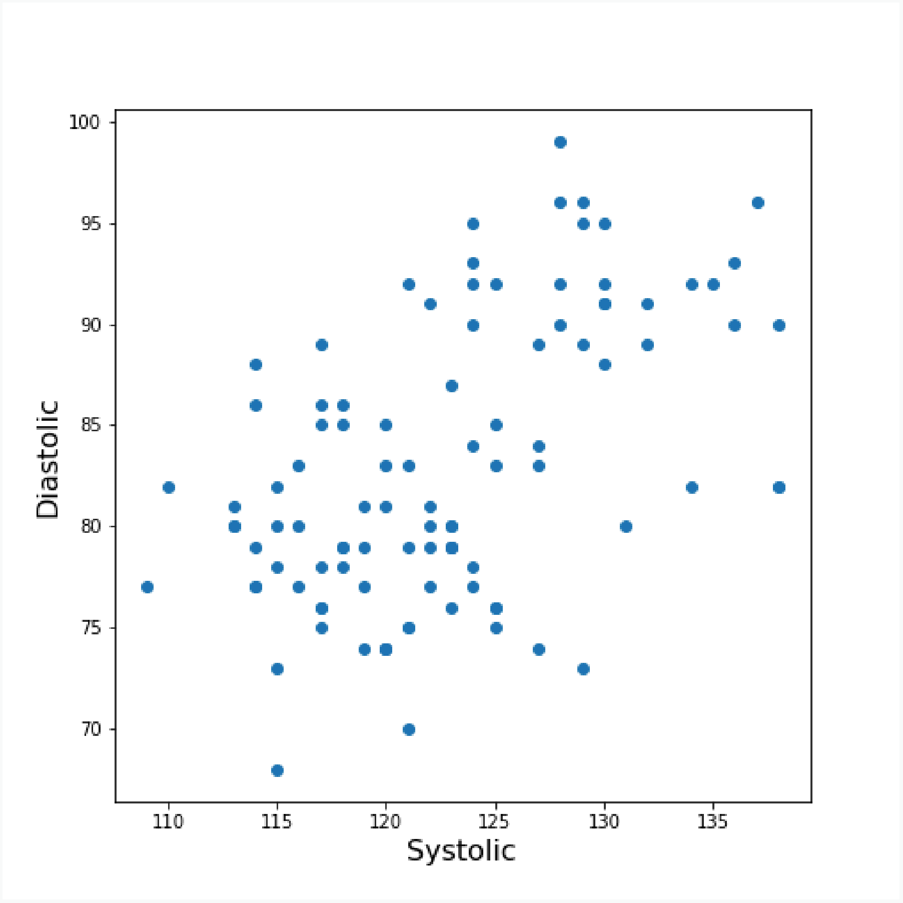
Create a scatter plot of the systolic via the diastolic blood pressure. Do the two variables appear to be independent or related?

Scatter plots are useful for the inspection of select pairs of data. However, they are only qualitative and thus, in general, it is preferred to have a numerical quantity.

Solution

PYTHON

```python
fig, ax = subplots();
ax.scatter(df['Systolic'], df['Diastolic']);
ax.set_xlabel('Systolic', fontsize=16)
ax.set_ylabel('Diastolic', fontsize=16)

show()
```



From the plot one might suspect that a larger systolic value is connected with a larger diastolic value. However, the plot in itself is not conclusive in that respect.

The Correlation Coefficient

Bivariate measures are quantities that are calculated from two variables of data. Bivariate features are the most widely used subset of multivariate features - all of which require more than one variable in order to be calculated.

The concept behind many bivariate measures is to quantify “similarity” between two data sets. If any similarity is discovered it is assumed that there is some connection or relationship between the sets. For similar variables, knowledge of one leads to some expectation about the other.

Here we are going to look at a specific bivariate quantity: the Pearson correlation coefficient PCC.

The formula for the Pearson PCC is set up such that two identical data sets yield a PCC of 1. (Technically this is done by normalising all variances to be equal to 1). This implies that all data points in a scatter plot of a variable against itself are aligned along the main diagonal (with positive slope).

Two perfectly antisymmetrical data sets (one variable can be obtained by multiplying the other by -1) yield a value -1. This implies that all data points in a scatter plot are aligned along the negative or anti diagonal (with negative slope). All other situations lie in between. A value of 0 refers to exactly balanced positive and negative contributions to the measure. (Note that strictly speaking, the latter does not necessarily mean that there is no relationship between the variables).

The PCC is an **undirected** measure in the sense that its value for the comparison between data set 1 and data set 2 is the same as the PCC between data set 2 and data set 1.

A direct way to calculate the PCC of two data-sets is to use the function `corr` applied to a dataframe. For instance, we can apply it to the Everleys data:

PYTHON

```python
df_everley = read_csv('data/everleys_data.csv')
df_everley.corr()
```

OUTPUT

```
          calcium    sodium
calcium  1.000000 -0.258001
sodium  -0.258001  1.000000
```

The result as a matrix of two-by-two numbers. Along the diagonal (top left and bottom right) are the values for the comparison of a column to itself. As any dataset is identical with itself, the values are one by definition.

The non-diagonal elements show that the $CC \approx -0.26$ for the two data sets. Both $CC(12)$ and $CC(21)$ are given in the matrix but because of the symmetry we would only need to report one of the two.

NOTE

Note that in this lesson we introduce how to calculate the PCC but do not discuss its significance. E.g. the interpretation of the value above needs to be checked against the fact that we only have 18 data points. Specifically, we refrain from concluding that because the PCC is negative, a high value of the calcium concentration is associated with a small value of the sodium concentration (relative to their respective means).

One quantitative way to assess whether or not a given value of the PCC is meaningful or not is to use surrogate data. In our case, we could e.g. create random numbers in an array with shape (18, 2) such that the two means and standard deviations are the same as in the Everley data but the two columns are independent of each other. Creating many realisations, we can check what distribution of PCC values is expected from the randomly generated data and compare this with the values from the Everleys data.

A lot of what we are going to do in the machine learning sessions will involve Numpy arrays. Let us therefore convert the Everleys data from a Pandas dataframe into a Numpy array.

PYTHON

```python
everley_numpy = df_everley.to_numpy()
everley_numpy
```

OUTPUT

```
array([[  3.4555817 , 112.69098   ],
       [  3.6690263 , 125.66333   ],
       [  2.7899104 , 105.82181   ],
       [  2.9399    ,  98.172772  ],
       [  5.42606   ,  97.931489  ],
       [  0.71581063, 120.85833   ],
       [  5.6523902 , 112.8715    ],
       [  3.5713201 , 112.64736   ],
       [  4.3000669 , 132.03172   ],
       [  1.3694191 , 118.49901   ],
       [  2.550962  , 117.37373   ],
       [  2.8941294 , 134.05239   ],
       [  3.6649873 , 105.34641   ],
       [  1.3627792 , 123.35949   ],
       [  3.7187978 , 125.02106   ],
       [  1.8658681 , 112.07542   ],
       [  3.2728091 , 117.58804   ],
       [  3.9175915 , 101.00987   ]])
```

We can see that the numbers remain the same but the format changed. E.g. we have lost the names of the columns. Similar to the Pandas dataframe, we can use ‘shape’ to see the dimensions of the data array.

PYTHON

```python
everley_numpy.shape
```

OUTPUT

```
(18, 2)
```

We can now use the Numpy function `corrcoef` to calculate the Pearson correlation:

PYTHON

```python
from numpy import corrcoef

corr_matrix = corrcoef(everley_numpy, rowvar=False)

print(corr_matrix)
```

OUTPUT

```
[[ 1.         -0.25800058]
 [-0.25800058  1.        ]]
```

The function `corrcoef` takes a two-dimensional array as input. The keyword argument `rowvar` is True by default which means the correlation will be calculated along the rows. As we have the data features in the columns, it needs to be set to False. (You can check what happens if you set it to ‘True’. Instead of a 2x2 matrix for two columns you will get a 18x18 matrix for eighteen pair comparisons.)

We mentioned that the values of the PCC are calculated such that they must lie between -1 and 1. This is achieved by normalisation with the variance. If for some reason we don’t want the similarity calculated using this normalisation, what we get is the so-called **covariance**. In contrast to the PCC its values will depend on the absolute size of the numbers in the data array. From Numpy, we can use the function `cov` to calculate the covariance:

PYTHON

```python
from numpy import cov

cov_matrix = cov(everley_numpy, rowvar=False)

print(cov_matrix)
```

OUTPUT

```
[[  1.70733842  -3.62631625]
 [ -3.62631625 115.70986192]]
```

The result shows how the covariance is strongly dependent on the actual numerical values in a data column. The two values along the diagonal are identical with the variances obtained by squaring the standard deviation (calculated for example using the `describe` function).

DIY3: CORRELATIONS FROM THE PATIENTS DATA

Calculate the Pearson PCC between the systolic and the diastolic blood pressure from the patients data using

i. the Pandas dataframe and

ii. the data as Numpy array.

Solution

PYTHON

```python
df = read_csv('data/patients.csv')

df[['Systolic', 'Diastolic']].corr()
```

OUTPUT

```
           Systolic  Diastolic
Systolic   1.000000   0.511843
Diastolic  0.511843   1.000000
```

Solution

PYTHON

```python
df_SysDia_numpy = df[['Systolic', 'Diastolic']].to_numpy()

df_SysDia_corr = corrcoef(df_SysDia_numpy, rowvar=False)

print('Correlation coefficient between Systole and Diastole:', round(df_SysDia_corr[0, 1], 2))
```

OUTPUT

```
Correlation coefficient between Systole and Diastole: 0.51
```

It is worth noting that it is equally possible to calculate the correlation between rows of a two-dimension array (i.e. rowvar=True) but the interpretation will differ. Imagine a dataset where for two subjects a large number, call it N, of metabolites were determined quantitatively (a Metabolomics dataset). If that dataset is of shape (2, N) then one can calculate the correlation between the two rows. This would be done to determine the correlation of the metabolite profiles between the two subjects.

The Correlation Matrix

If we have more than two columns of data, we can obtain a Pearson correlation coefficient for each pair. In general, for N columns, we get N^2 pairwise values. We omit the correlations of each column with itself, of which there are N, which means we are left with $N * (N - 1)$ pairs. Because each value appears twice due to symmetry of the calculation, we can ignore half of them and we are left with $N * (N - 1)/2$ coefficients for N columns.

Here is an example for the 'patients' data:

PYTHON

```python
df = read_csv('data/patients.csv')

df.corr()
```

OUTPUT

```
ValueError: could not convert string to float: 'Male'
```

If we do the calculation with the Pandas dataframe, the 'Gender' is automatically ignored and by default we get $6 * 5/2 = 15$ coefficients for the six remaining columns. Note that the values that involves the 'Smoker' column are meaningless.

Let us now convert the dataframe to a Numpy array and check its shape:

PYTHON

```python
patients_numpy = df.to_numpy()
patients_numpy.shape
```

OUTPUT

```
(100, 7)
```

Now we can try to calculate the correlation matrix for the first five columns of this data array. If we do it directly to the array, we get an AttributeError: 'float' object has no attribute 'shape'.

This is mended by converting the array to floating point before using the `corrcoef` function. For this we use `astype(float)`:

PYTHON

```
cols = 5

patients_numpy_float = patients_numpy[:, :cols].astype(float)

patients_corr = corrcoef(patients_numpy_float, rowvar=False)

patients_corr
```

OUTPUT

```
array([[1.        , 0.11600246, 0.09135615, 0.13412699, 0.08059714],
       [0.11600246, 1.        , 0.6959697 , 0.21407555, 0.15681869],
       [0.09135615, 0.6959697 , 1.        , 0.15578811, 0.22268743],
       [0.13412699, 0.21407555, 0.15578811, 1.        , 0.51184337],
       [0.08059714, 0.15681869, 0.22268743, 0.51184337, 1.        ]])
```

The result is called the **correlation matrix**. It contains all the bivariate comparisons possible for the five columns chosen.

In the calculation above we used the PCC to calculate the matrix. In general, any bivariate measure can be used to obtain a matrix of same shape.

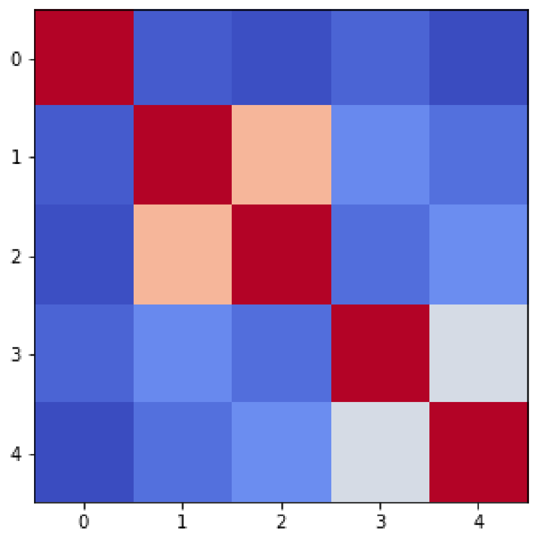
Heat map in Matplotlib

To get an illustration of the correlation pattern in a dataset we can plot the correlation matrix as a heatmap.

Here is some code using Matplotlib to plot a heatmap of the correlation matrix from the patients dataset. We use the function `imshow`:

PYTHON

```
fig, ax = subplots(figsize=(5,5))

im = ax.imshow(patients_corr, cmap='coolwarm');

show()
```



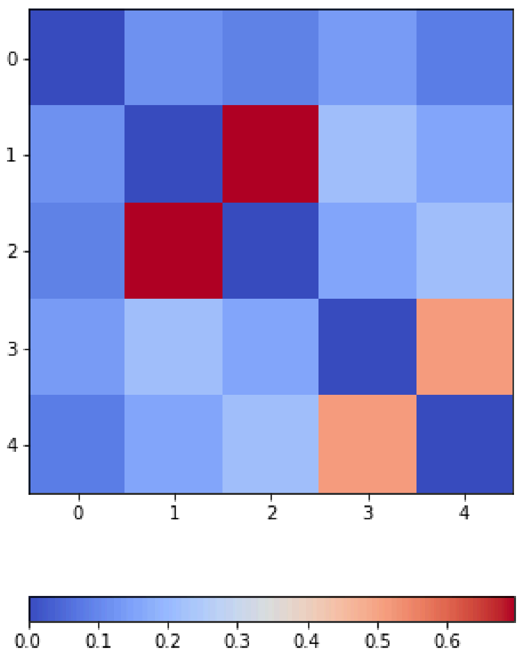
Note that we have specified the colour map ‘coolwarm’. For a list of Matplotlib colour maps, please refer to the gallery in the documentation. The names to use in the code are on the left hand side of the colour bar.

Let us add two more features to improve the figure.
First, to have true correlations stand out (rather than the trivial self correlations along the diagonal which are always one) we we can deliberately set the diagonal equal to zero. To achieve this, we use the Numpy function `fill_diagonal`.

Second, `imshow` scales the colours by default to the minimum and maximum value of the array. As such we don’t know what red or blue means. To see the colour bar, it can be added to the figure environment ‘fig’ using `colorbar`.

PYTHON

```
from numpy import fill_diagonal

fill_diagonal(patients_corr, 0)

fig, ax = subplots(figsize=(7,7))

im = ax.imshow(patients_corr, cmap='coolwarm');

fig.colorbar(im, orientation='horizontal', shrink=0.7);

show()
```



The result is that the correlation between columns ‘Height’ and ‘Weight’ is the strongest and presumably higher than could be expected if these two measures were independent. We can confirm this by plotting a scatter plot for these two columns and compare to the scatter plot for (original) columns 2 (Height) and 5 (Diastolic blood pressure):

DIY4: SPEARMAN CORRELATIONS FROM THE PATIENTS DATA

Calculate and plot the correlation matrix of the first five columns as above based on the Spearman rank correlation coefficient. It is based on the ranking of values instead of their numerical values as for the Pearson coefficient. Spearman therefore tests for tests for monotonic relationships whereas Pearson tests for linear relationships.

To import the function use:

```
from scipy.stats import spearmanr
```

You can then apply it in the form:

```
data_spearman_corr = spearmanr(data).correlation
```

Solution

PYTHON

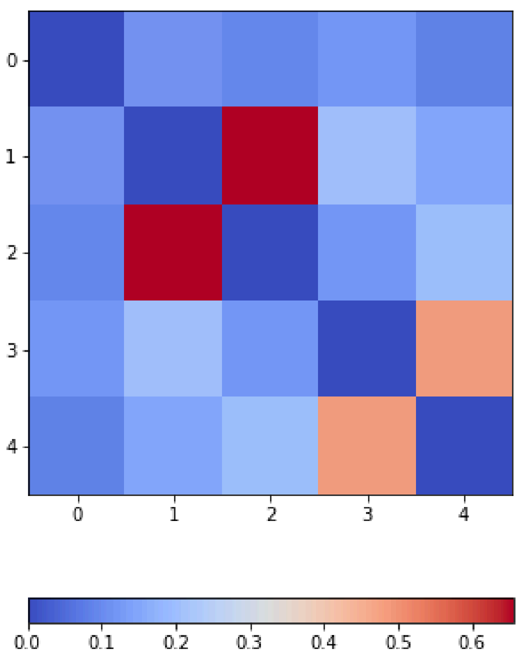
```
from scipy.stats import spearmanr
patients_numpy = df.to_numpy()
cols = 5

patients_numpy_float = patients_numpy[:, :cols].astype(float)
patients_spearman = spearmanr(patients_numpy_float).correlation

patients_spearman
```

OUTPUT

```
array([[1.        , 0.11636668, 0.09327152, 0.12105741, 0.08703685],
       [0.11636668, 1.        , 0.65614849, 0.20036338, 0.14976559],
       [0.09327152, 0.65614849, 1.        , 0.12185782, 0.19738765],
       [0.12105741, 0.20036338, 0.12185782, 1.        , 0.48666928],
       [0.08703685, 0.14976559, 0.19738765, 0.48666928, 1.        ]])
```

PYTHON

```
from numpy import fill_diagonal
fill_diagonal(patients_spearman, 0)

fig, ax = subplots(figsize=(7,7))

im = ax.imshow(patients_spearman, cmap='coolwarm');
fig.colorbar(im, orientation='horizontal', shrink=0.7);

show()
```



Analysis of the Correlation matrix

The Correlation Coefficients

To analyse the correlations in a data set, we are only interested in the $N*(N-1)/2$ unduplicated correlation coefficients. Here is a way to extract them and assign them to a variable.

We import the function `triu_indices`. It provides the indices of a matrix with specified size. The size we need is obtained from our correlation matrix, using `len`. It is identical to the number of columns for which we calculated the CCs.

We also need to specify that we do not want the diagonal to be included. For this, there is an offset parameter 'k' which will collect the indices excluding the diagonal if it is set to 1. (To include the indices of the diagonal it would have to be 0).

PYTHON

```
from numpy import triu_indices

# Get the number of rows of the correlation matrix
no_cols = len(patients_corr)

# Get the indices of the 10 correlation coefficients for 5 data columns
corr_coeff_indices = triu_indices(no_cols, k=1)

# Get the 10 correlation coefficients
corr_coeffs = patients_corr[corr_coeff_indices]

print(corr_coeffs)
```

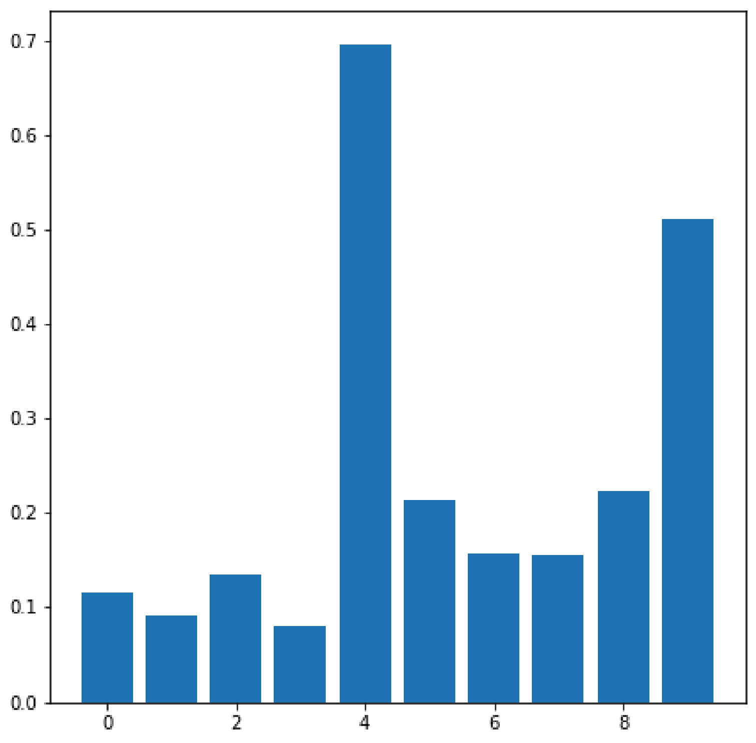
OUTPUT

```
[0.11600246 0.09135615 0.13412699 0.08059714 0.6959697  0.21407555
 0.15681869 0.15578811 0.22268743 0.51184337]
```

Now we plot these correlation coefficients as a bar chart to see them one next to each other.

PYTHON

```
fig, ax = subplots()

bins = arange(len(corr_coeffs))

ax.bar(bins, corr_coeffs);

show()
```



If there is a large number of coefficients, we can also display their histogram or a boxplot as a summary statistics.

The Average Correlation per Column

On a higher level, we can calculate the overall or average correlation per data column. We achieve this by averaging over either the rows or the columns of the correlation matrix. Because our similarity measure is undirected, both ways of summing yield the same result.

However, we need to consider the sign. The correlation coefficients can be positive or negative. As such, adding for instance +1 ans -1 would yield an average of zero even though both indicate perfect correlation and anti-correlation, respectively. We address this by using the absolute value `abs`, ignoring the sign.

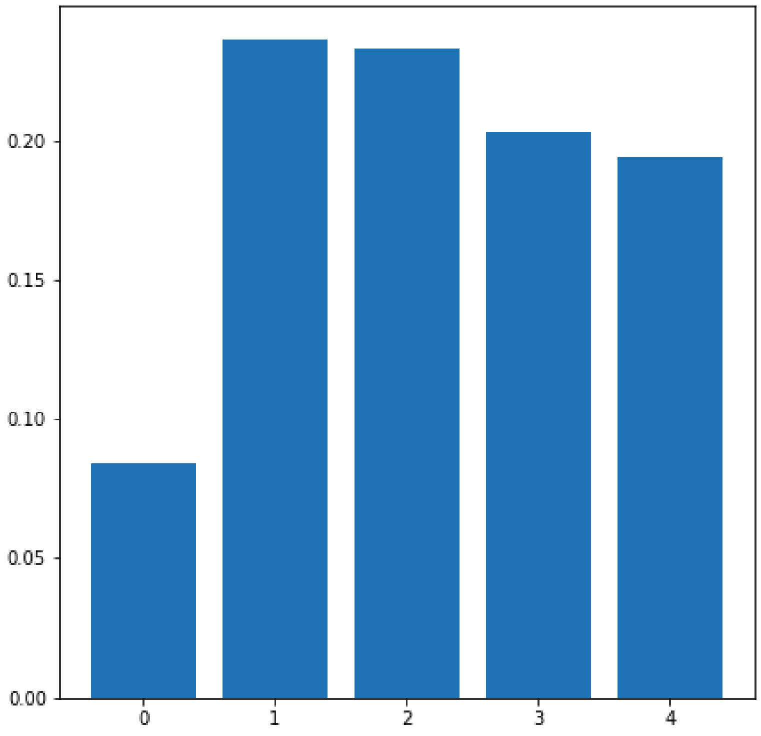
To average, we use function `mean`. This function by default averages over all values of the matrix. To obtain the five values by averaging over the columns, we specify the ‘axis’ keyword argument as 0.

PYTHON

```python
from numpy import abs, mean

# Absolute values of correlation matrix
corr_matrix_abs = abs(patients_corr)

# Average of the correlation strengths
corr_column_average = mean(corr_matrix_abs, axis=0)

fig, ax = subplots()

bins = arange(corr_column_average.shape[0])

ax.bar(bins, corr_column_average );

print(corr_column_average)

show()
```

OUTPUT

```
[0.08441655 0.23657328 0.23316028 0.20316681 0.19438933]
```



The result is that the average column correlation is on the order of 0.2 for the columns with indices 1 to 4 and less than 0.1 for the column with index 0, which is the age.

The Average Data Set Correlation

The sum over rows or columns has given us a reduced set of values to look at. We can now take the final step and average over all correlation coefficients. This will yield the average correlation of the data set. It condenses the full bivariate analysis into a single number and can be a starting point when comparing e.g. different data sets of the same type.

PYTHON

```python
# Average of the correlation strengths
corr_matrix_average = mean(corr_matrix_abs)

print('Average correlation strength: ', round(corr_matrix_average, 3))
```

OUTPUT

```
Average correlation strength:  0.19
```

Application: The Diabetes Data Set

We now return to the data set that started our enquiry into dataframes in the previous lesson. Let us apply the above and do a summary analysis of its bivariate features.

First we import the data. it is one of the example datasets of scikit-learn, the Python library for Machine Learning. As such it is already included in the Anaconda package and you can import it directly.

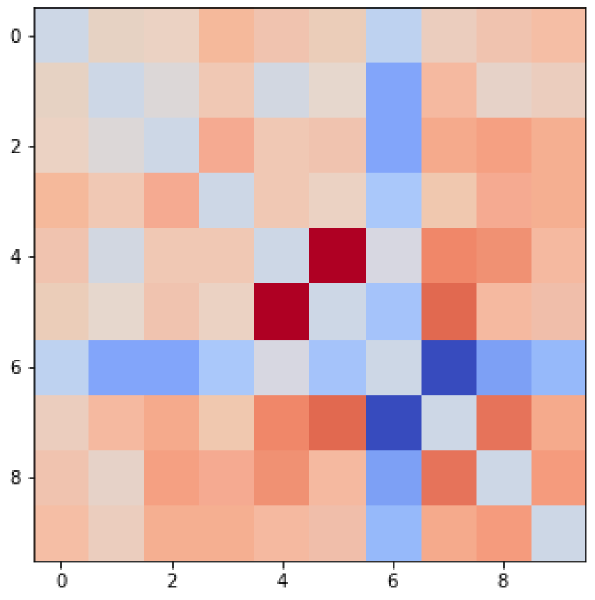
PYTHON

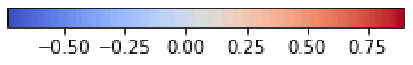
```python
from sklearn import datasets

diabetes = datasets.load_diabetes()

data_diabetes = diabetes.data
```

For the bivariate features, let us get the correlation matrix and plot it as a heatmap. We use code introduced above.

PYTHON

```python
from numpy import fill_diagonal

data_corr_matrix = corrcoef(data_diabetes, rowvar=False)

fill_diagonal(data_corr_matrix, 0)

fig, ax = subplots(figsize=(8, 8))

im = ax.imshow(data_corr_matrix, cmap='coolwarm');

fig.colorbar(im, orientation='horizontal', shrink=0.5);

show()
```



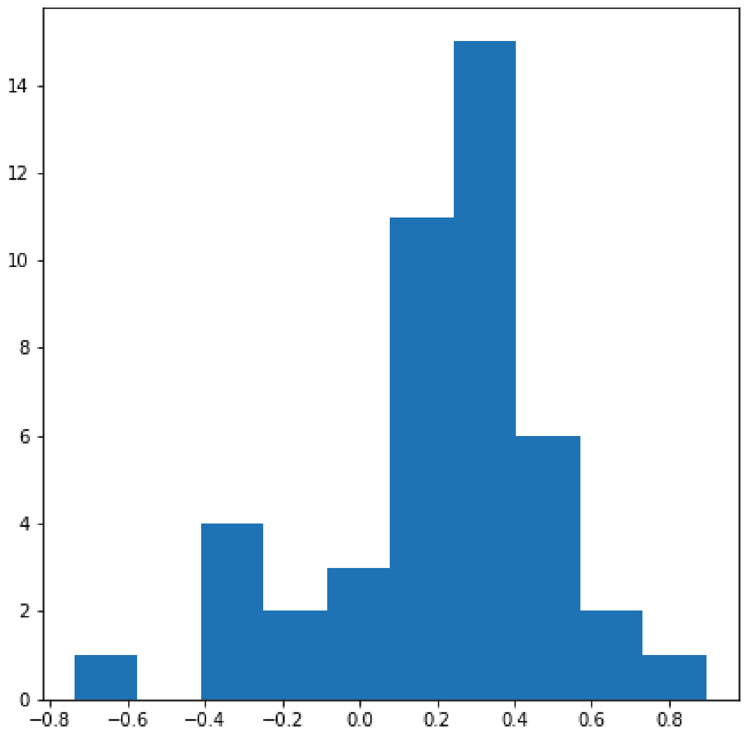


There is one strongly correlated pair (column indices 4 and 5) and one strongly anti-correlated pair (column indices 6 and 7).

Now we calculate the $10 * 9/2 = 45$ correlation coefficients and plot them as a histogram:

PYTHON

```
from numpy import triu_indices

data_corr_coeffs = data_corr_matrix[triu_indices(data_corr_matrix.shape[0], k=1)]

fig, ax = subplots()

ax.hist(data_corr_coeffs, bins=10);

show()
```

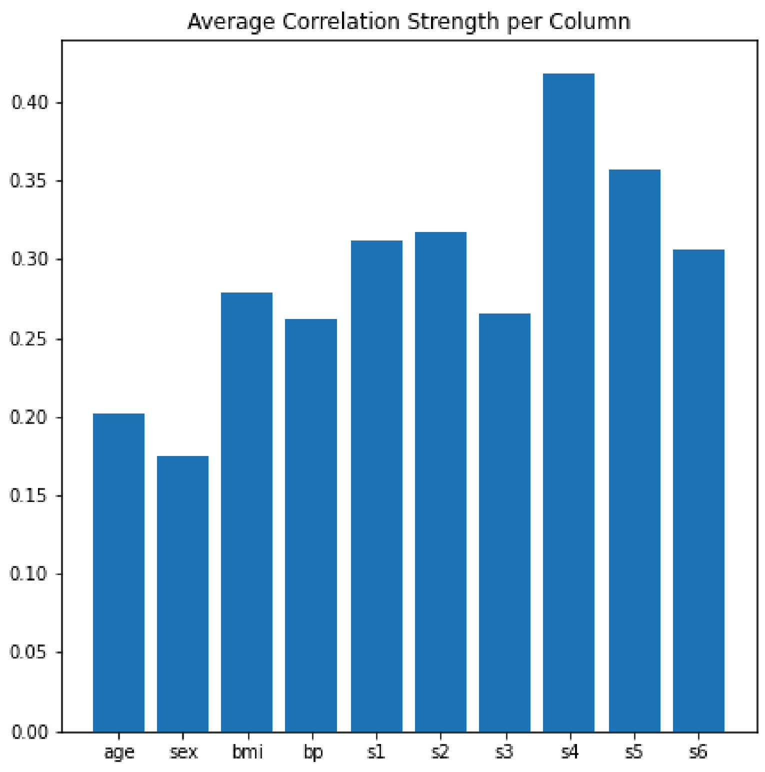


This histogram shows that the data have a distribution that is shifted towards positive correlations. However, only four values are (absolutely) larger than 0.5 (three positive, one negative).

Next we can get the average (absolute) correlation per column.

PYTHON

```python
data_column_average = mean(abs(data_corr_matrix), axis=0)

fig, ax = subplots()

bins = arange(len(data_column_average))

ax.bar(bins, data_column_average);
ax.set_title('Average Correlation Strength per Column')
ax.set_xticks(arange(len(diabetes.feature_names)))
ax.set_xticklabels(diabetes.feature_names);

show()
```



In the plot, note how the column names were extracted from the 'diabetes' data using `diabetes.feature_names`.

Finally, we can obtain the average correlation of the whole data set.

PYTHON

```
# Average of the correlation strengths
data_corr_matrix_average = mean(abs(data_corr_matrix))

print('Average Correlation Strength: ', round(data_corr_matrix_average, 3))
```

OUTPUT

```
Average Correlation Strength:  0.29
```

Exercises

END OF CHAPTER EXERCISES

Assignment: The Breast Cancer Data

Import the breast cancer data set using `read_csv`. Based on the code of this lesson, try to do the following:

1. Get the summary (univariate) statistics of columns 2-10 (accessing indices 1:10) using `describe`

2. Plot the means of each column as a bar chart with standard deviations as error bars. Why are some bars invisible?

3. Extract the values as Numpy array using `to_numpy`. The shape of the array should be (569, 31).

4. Calculate the correlation matrix using `corrcoef` from Numpy and plot it as a heatmap. The shape of the matrix should be (31, 31). Use `fill_diagonal` to set the diagonal elements to 0.

5. Calculate the average column correlation and plot it as a bar chart.

6. Calculate the average correlation strength of the data set.

In case of doubt, try to get help from the respective documentations for Pandas dataframes, Numpy and Matplotlib.

Solution

Q1

PYTHON

```
# To import data from a csv file into a Pandas dataframe
from pandas import read_csv

# To import a dataset from scikit-learn
from sklearn import datasets

# To create figure environments and plots
from matplotlib.pyplot import subplots, show

# Specific numpy functions, description in the main body
from numpy import corrcoef, fill_diagonal, triu_indices, arange
from numpy import mean

df_bc = read_csv("data/breast_cancer.csv")

df_bc_describe = df_bc.iloc[:, 1:10].describe()

df_bc_describe.round(2)
```

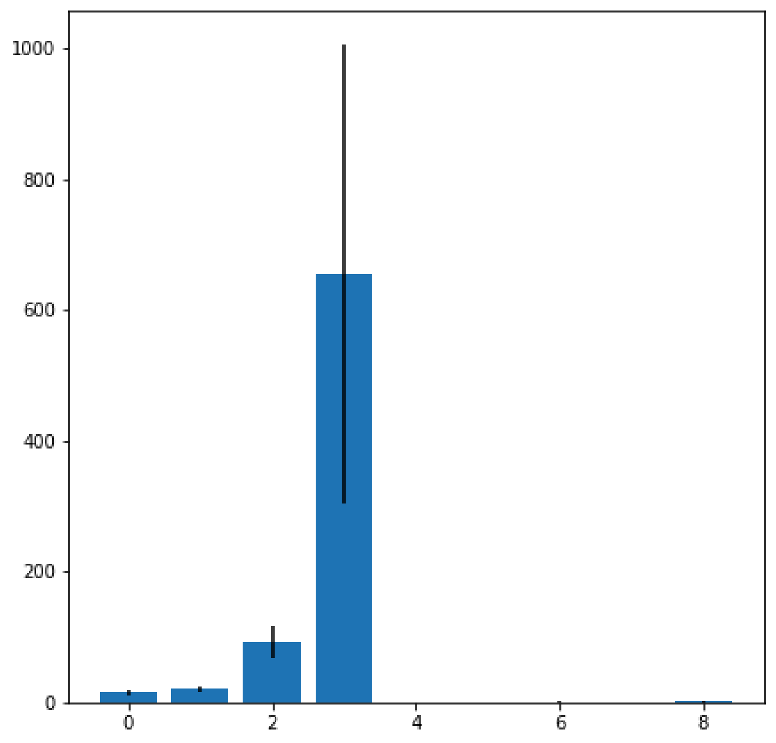
OUTPUT

```
       radius_mean  texture_mean  ...  concave points_mean  symmetry_mean
count       569.00        569.00  ...               569.00         569.00
mean         14.13         19.29  ...                 0.05           0.18
std           3.52          4.30  ...                 0.04           0.03
min           6.98          9.71  ...                 0.00           0.11
25%          11.70         16.17  ...                 0.02           0.16
50%          13.37         18.84  ...                 0.03           0.18
75%          15.78         21.80  ...                 0.07           0.20
max          28.11         39.28  ...                 0.20           0.30

[8 rows x 9 columns]
```

Q2

PYTHON

```
fig, ax = subplots()

bins = arange(df_bc_describe.shape[1])

ax.bar(bins, df_bc_describe.loc['mean'], yerr=df_bc_describe.loc['std'])

show()
```



Q3

PYTHON

```
bc_numpy = df_bc.to_numpy()

bc_numpy.shape
```

OUTPUT

```
(569, 31)
```

Q4

PYTHON

```
bc_corr = corrcoef(bc_numpy, rowvar=False)

bc_corr.shape
```

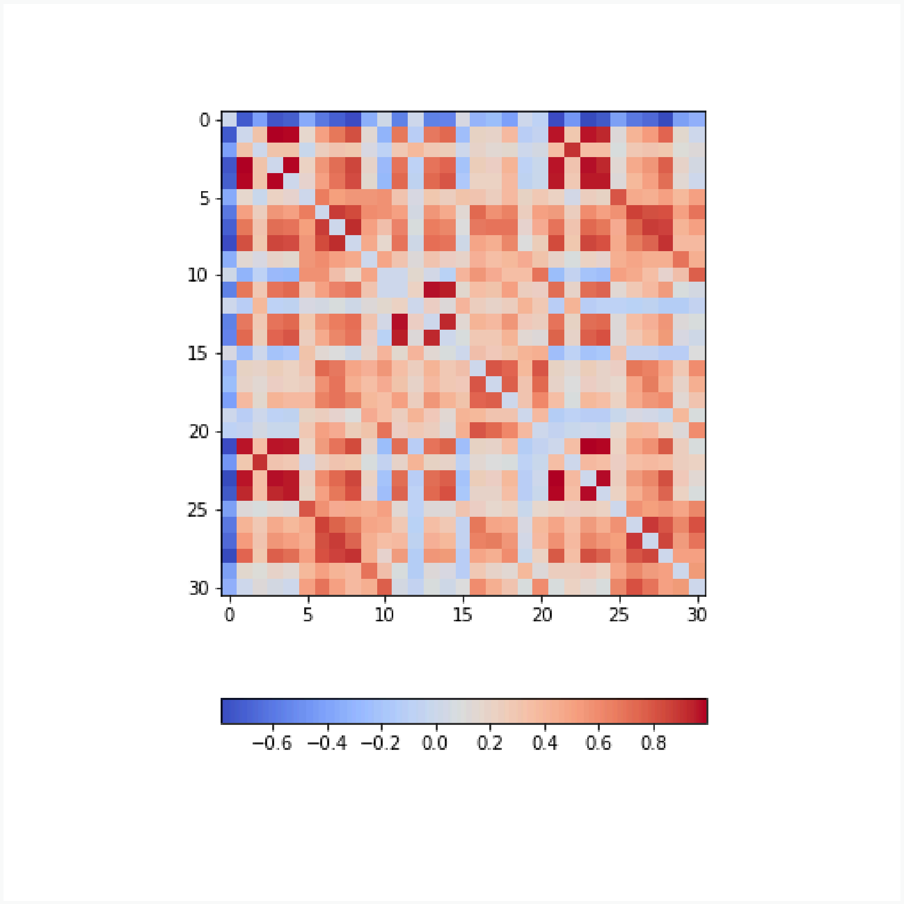
OUTPUT

```
(31, 31)
```

Q5

PYTHON

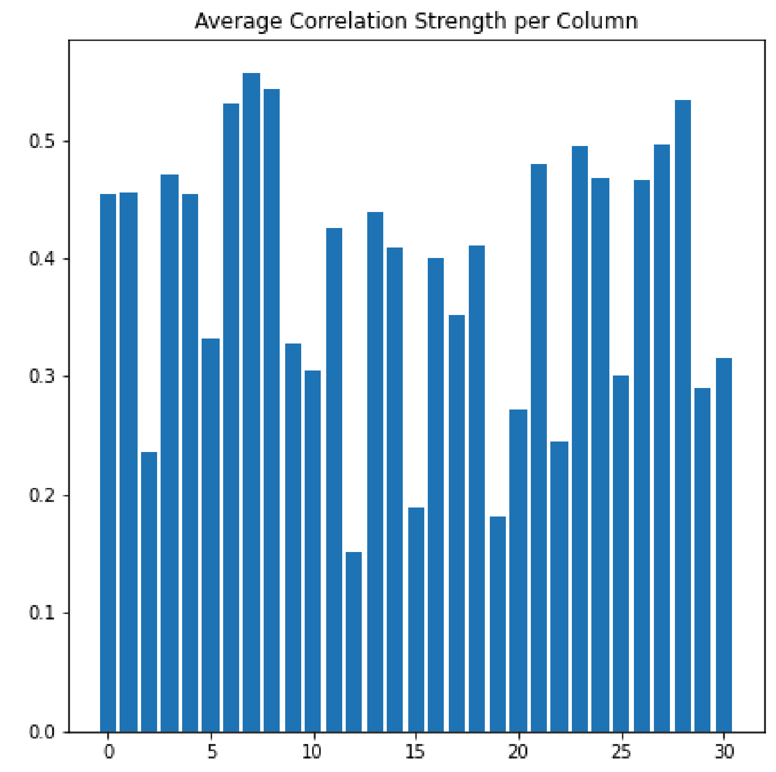
```python
from numpy import fill_diagonal

fill_diagonal(bc_corr, 0)

fig, ax = subplots(figsize=(7,7))

im = ax.imshow(bc_corr, cmap='coolwarm');

fig.colorbar(im, orientation='horizontal', shrink=0.7);

show()
```



Q6

PYTHON

```python
bc_column_average = mean(abs(bc_corr), axis=0)

fig, ax = subplots()

bins = arange(len(bc_column_average))

ax.bar(bins, bc_column_average);
ax.set_title('Average Correlation Strength per Column');

show()
```



PYTHON

```
# Average of the correlation strengths
bc_corr_matrix_average = mean(abs(bc_corr))

print('Average Correlation Strength: ', round(bc_corr_matrix_average, 3))
```

OUTPUT

```
Average Correlation Strength:  0.387
```

KEY POINTS

- Quantities based on data from two variables are referred as bivariate measures.
- Bivariate properties can be studied using `matplotlib` and `numpy`.
- Multivariate data analysis helps to find out relationships between recorded variables.
- Functions `corr` and `corrcoef` are used to calculate the PCC.
- A correlation matrix is visualised as a heatmap.

Content from Image Handling

Last updated on 2024-05-24 | Edit this page

Download Chapter PDF

Download Chapter notebook (ipynb)

Mandatory Lesson Feedback Survey

OVERVIEW

Questions

- How to read and process images in Python?
- How is an image mask created?
- What are colour channels in images?
- How to deal with big images?

Objectives

- Understanding 2-dimensional greyscale images.
- Learning image masking.
- 2-dimensional colour images, colour channels
- Decreasing memory load

PREREQUISITES

- Numpy arrays
- Plots and subplots with matplotlib

EXERCISES

This lesson has no explicit exercises. At each step, use images of your own choice to practice. There are many image file formats, different colour schemes etc for which you can try to find similar or analogous solutions.

Challenge

Reading and Processing Images

In biology, we often deal with images, for example from microscopy and different medical imaging modalities. In many cases, we wish to extract some quantitative information from these images. The focus of this lesson is to read and process images in Python. This includes:

- Working with 2-dimensional greyscale images
- Creating and applying binary image masks
- Working with 2-dimensional colour images, and interpreting colour channels
- Decreasing the memory for further processing by reducing resolution or patching
- Working with 3-dimensional images

Image Example

The example in Figure 1 is an image from the cell image library with the following description:

> “Midsaggital section of rat cerebellum, captured using confocal imaging. Section shows inositol trisphosphate receptor (IP3R) labelled in green, DNA in blue, and synaptophysin in magenta. Honorable Mention, 2010 Olympus BioScapes Digital Imaging Competition®.”

We might want to, for example, determine the relative amounts of IP3R, DNA and synaptophysin in this image. This tutorial will guide you through some of the steps to get you started with processing images of all sorts using Python. At the end, you will have the opportunity to come back to this image example and perform some analysis of your own.

Figure 1: Example image, rat cerebellum

Figure 1: Example image, rat cerebellum

Work Through Example

Reading and Plotting a 2-dimensional Image

First, we want to read in an image. For this part of the lesson, we use a histological slice through an axon bundle as an example. We use Matplotlib’s image module, from which we import `imread` to store the image in a variable called img. The function `imread` can interpret many different image formats, including jpg, png and tif images.

PYTHON

```python
from matplotlib.image import imread

img = imread('fig/axon_slice.jpg')
```

OUTPUT

```
PIL.UnidentifiedImageError: cannot identify image file 'fig/axon_slice.jpg'
```

We can check what type of variable this is:

PYTHON

```python
print(type(img))
```

OUTPUT

```
NameError: name 'img' is not defined
```

This tells us that the image is stored in a Numpy array. We can check some other properties of this array, for example, what the image dimensions are.

PYTHON

```python
print(img.shape)
```

OUTPUT

```
NameError: name 'img' is not defined
```

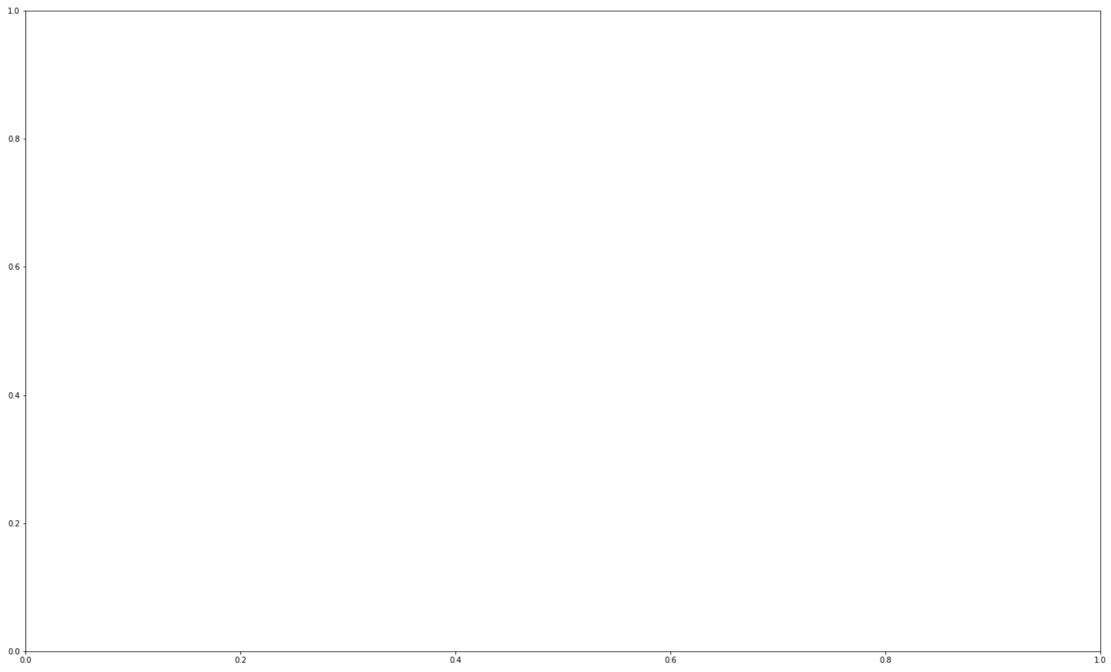
This tells us that our image is composed of 2300 by 3040 data units, or *pixels* as we are dealing with an image. It is equivalent to the image resolution. The array has two dimensions, and so we can expect our image to be two-dimensional as well. Let us now use matplotlib.pyplot's `imshow` function to plot the image to see what it looks like. We set the colour map to `gray` to overwrite the default colour map.

PYTHON

```python
from matplotlib.pyplot import subplots, show

fig, ax = subplots(figsize=(25, 15))

ax.imshow(img, cmap='gray');
```

PYTHON

```python
show()
```



`imshow` has allowed us to plot the Numpy array of our image data as a picture. The figure is divided up into a number of pixels, and each of those pixels is assigned an intensity value stored in the Numpy array. Let's have a closer look by selecting a smaller region of our image and plotting that.

PYTHON

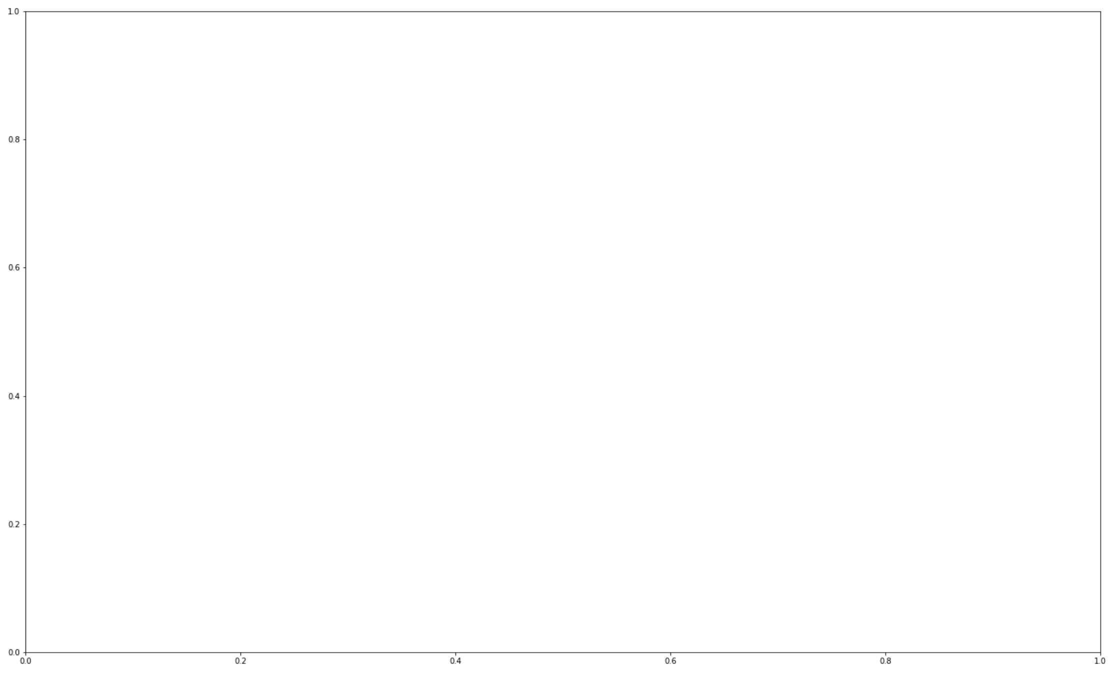
```python
from matplotlib.pyplot import subplots, show

fig, ax = subplots(figsize=(25, 15))

ax.imshow(img[:50, :70], cmap='gray');
```

OUTPUT

```
NameError: name 'img' is not defined
```

PYTHON

```python
show()
```



With `img[:50, :70]` we select the first 50 values from the first dimension, and the first 70 values from the second dimension. Thus, the image above shows a very small part of the upper left corner of our original image. As we are now zoomed in quite close to that corner, we can easily see the individual pixels here. Let's take a quick look at an even smaller section.

PYTHON

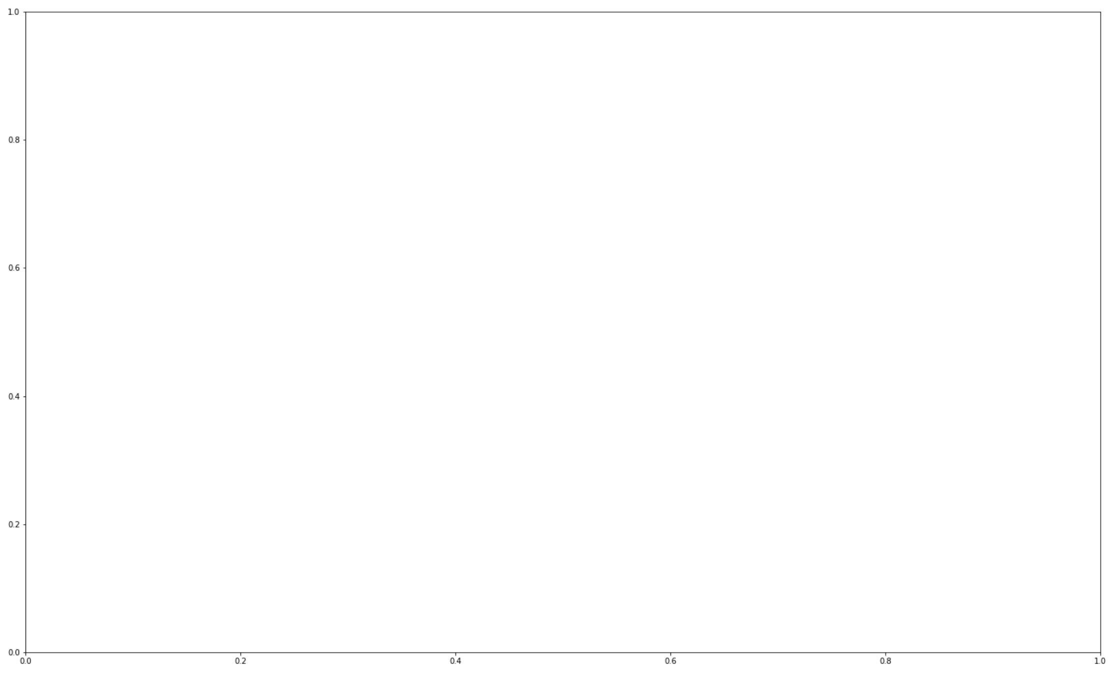
```
fig, ax = subplots(figsize=(25, 15))

ax.imshow(img[:20, :15], cmap='gray');
```

OUTPUT

```
NameError: name 'img' is not defined
```

PYTHON

```
show()
```



This is a small section from that same upper left corner. Each square is a pixel and it has one grey value. So how exactly are the pixel values assigned? By the numbers stored in the Numpy array, `img`. Let us have a look at those values by picking a slice from the array.

PYTHON

```
print(img[:20, :15])
```

OUTPUT

```
NameError: name 'img' is not defined
```

Each of these numbers corresponds to an intensity in the specified colourmap. These numbers range from 0 to 255, implying 256 shades of grey.

We chose `cmap = gray`, which assigns darker grey colours to smaller numbers, and lighter grey colours to higher numbers. However, we can also pick a colourmap to plot our image, and we can even show a colourbar to keep track of the intensity values. Matplotlib has a large number of very nice colourmaps that you can look through here. We show an example of the colourmaps called `viridis` and `magma`:

PYTHON

```
fig, ax = subplots(nrows=1, ncols=2, figsize=(25, 15))

p1 = ax[0].imshow(img[:20, :15], cmap='viridis')
```

PYTHON

```
p2 = ax[1].imshow(img[:20, :15], cmap='magma')
```

PYTHON

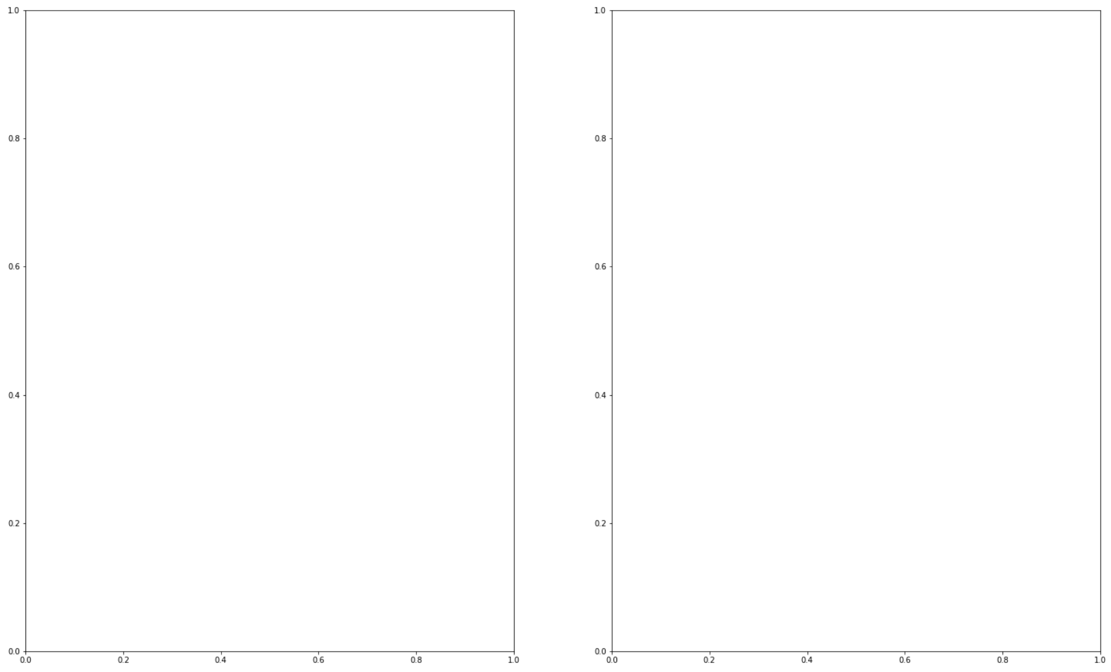
```
fig.colorbar(p1, ax=ax[0], shrink = 0.8)
```

PYTHON

```
fig.colorbar(p2, ax=ax[1], shrink = 0.8);
```

PYTHON

```
show()
```



Note, that even though we can plot our greyscale image with colourful colourschemes, it still does not qualify as a colour image. This is because colour images will require **three sets** of intensities for each pixel, not just one as in this example. In the case above, the number in the array represented a grey value and the colour was assigned to that grey value by Matplotlib. These represent 'false' colours.

Creating an Image Mask

Now that we know that the images are composed of a set of intensities that are just numbers in a Numpy array, we can start using these numbers to process our image.

As a first approach, we can plot a histogram of the original image intensities. We use the `.flatten()` method to turn the original 2300 x 3040 array into a one-dimensional array with 6,992,000 values. This rearrangement allows the inclusion of an image as a single column in a matrix or dataframe!

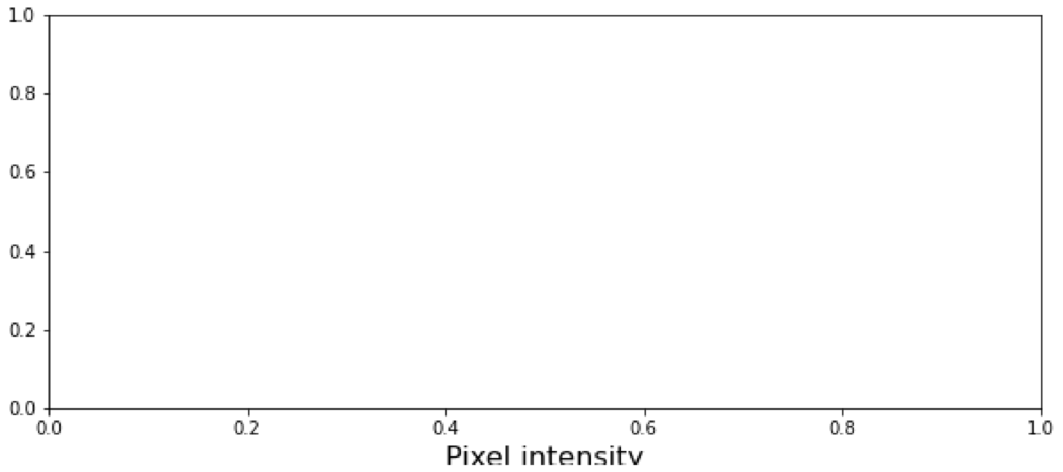
The histogram plot shows how many of each of the intensities are found in this image:

PYTHON

```
fig, ax = subplots(figsize=(10, 4))

ax.hist(img.flatten(), bins = 50)
```

PYTHON

```
ax.set_xlabel("Pixel intensity", fontsize=16);

show()
```



The histogram is a distribution with intensity values mostly between about 50 and 250.
The image shows a cut through an axon bundle. Say we are now interested in the myelin sheath surrounding the axons (the dark rings). We can create a **mask** that isolates pixels whose intensity value is below a certain threshold (because darker pixels have lower intensity values). Everything below this threshold can be assigned to e.g. 1 (representing True), and everything above will be assigned to 0 (representing False). This is called a binary or Boolean mask.

Based on the histogram above, we might try to adjust that threshold somewhere between 100 and 200. Let's see what we get with a threshold set to 125. We first use a conditional statement to create the mask. Then we apply the mask to the image. As a result we plot both the mask and the masked image.

PYTHON

```
threshold = 125

mask = img < threshold
```

PYTHON

```
img_masked = img*mask
```

PYTHON

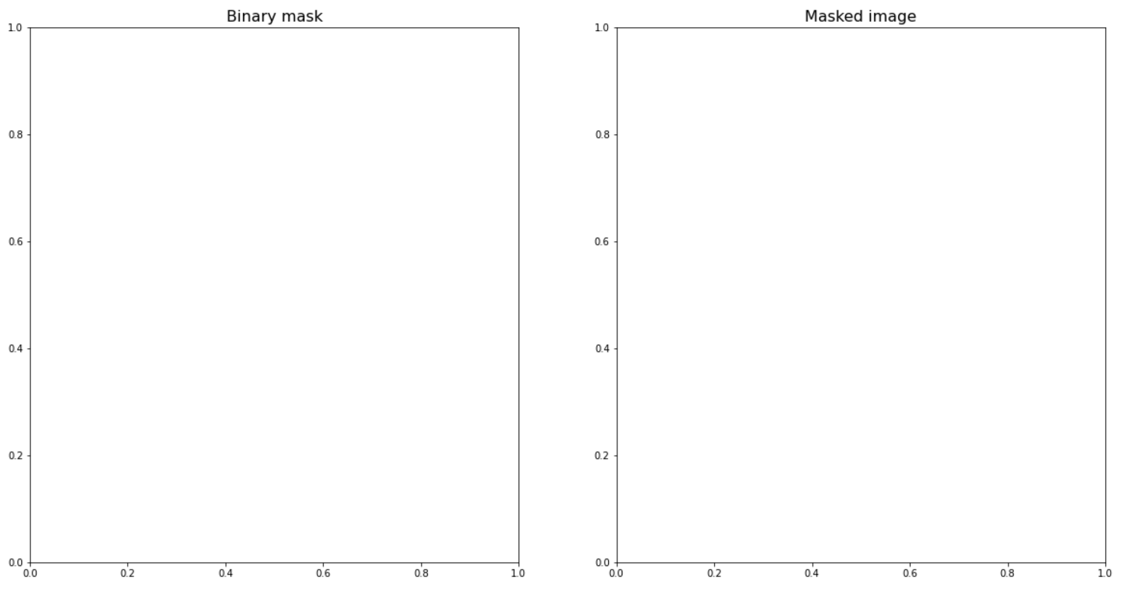
```python
fig, ax = subplots(nrows=1, ncols=2, figsize=(20, 10))

ax[0].imshow(mask, cmap='gray')
```

PYTHON

```python
ax[0].set_title('Binary mask', fontsize=16)
ax[1].imshow(img_masked, cmap='gray')
```

PYTHON

```python
ax[1].set_title('Masked image', fontsize=16)

show()
```



The left subplot shows the binary mask itself. White represents values where our condition is true, and black where our condition is false. The right image shows the original image after we have applied the binary mask, i.e. the original pixel intensities in regions where the mask value is true.

Note that “applying the mask” means that the intensities where the condition is true are left unchanged and the intensities where the condition is false are multiplied with zero ans therefore set to zero.

Let’s have a look at the resulting image histograms.

PYTHON

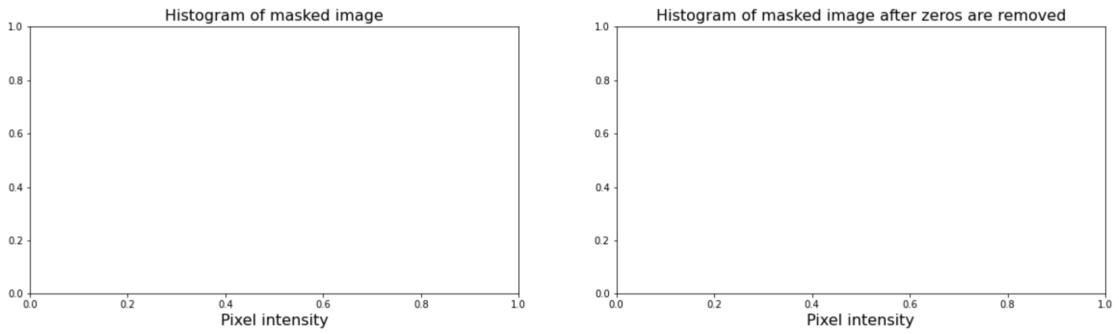
```python
fig, ax = subplots(nrows=1, ncols=2, figsize=(20, 5))

ax[0].hist(img_masked.flatten(), bins=50)
```

PYTHON

```
ax[0].set_title('Histogram of masked image', fontsize=16)
ax[0].set_xlabel("Pixel intensity", fontsize=16)

ax[1].hist(img_masked[img_masked != 0].flatten(), bins=25)
```

PYTHON

```
ax[1].set_title('Histogram of masked image after zeros are removed', fontsize=16)
ax[1].set_xlabel("Pixel intensity", fontsize=16)

show()
```



On the left we show all the values for the masked image. There is a large peak at zero, as a large part of the image is masked. On the right, we show only the non-zero pixel intensities. We can see that our mask worked as expected, only values up to 125 are found. This is because our threshold causes a sharp cut-off at a pixel intensity of 125.

Colour Images

Often we want to work with colour images. So far, our image had a single intensity value for each pixel. In colour images, we will have three so-called channels corresponding to red, green and blue intensities. Any colour will be a composite of the intensity value for each of these colours. We now show an example with a colour image of the rat cerebellar cortex. Let us import it and check its shape.

PYTHON

```
img_col = imread('fig/rat_brain_low_res.jpg')
```

OUTPUT

```
PIL.UnidentifiedImageError: cannot identify image file 'fig/rat_brain_low_res.jpg'
```

PYTHON

```
img_col.shape
```

OUTPUT

```
NameError: name 'img_col' is not defined
```

Our image array now contains three dimensions. The first two are the spatial dimensions corresponding to the pixel positions. The last one contains the three colour channels. So we have three layers of intensity values on top of each other.

First, let us plot the whole image.

PYTHON

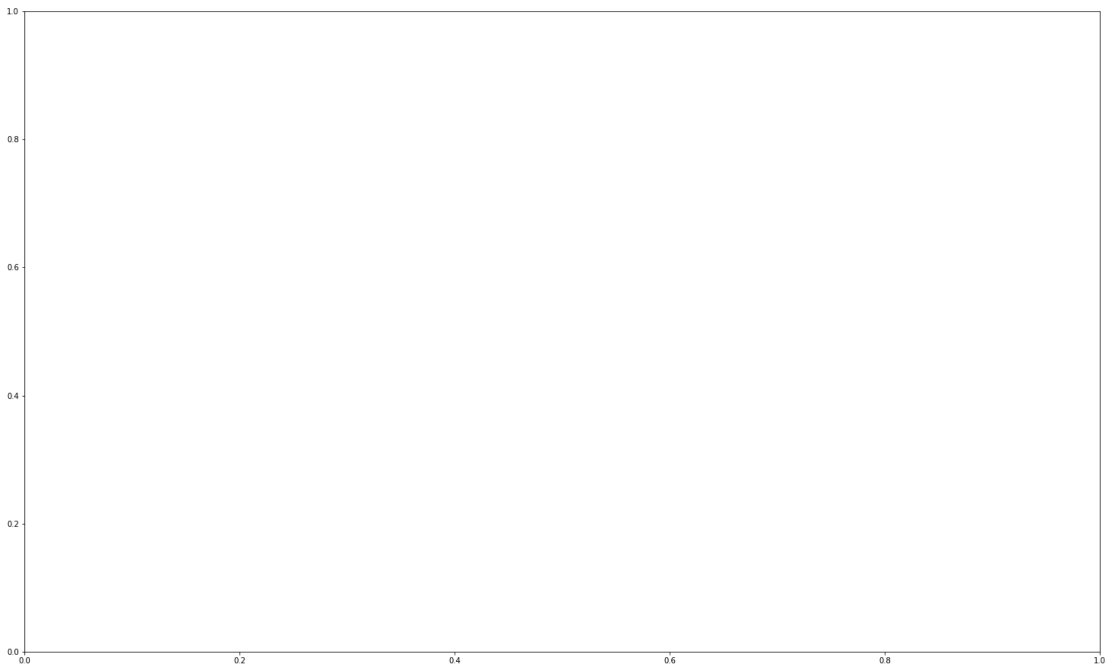
```
fig, ax = subplots(figsize=(25, 15))

ax.imshow(img_col);
```

OUTPUT

```
NameError: name 'img_col' is not defined
```

PYTHON

```
show()
```



The sample is labeled for Hoechst stain (blue), the Inositol trisphosphate (IP3) receptor (green) and Glial fibrillary acidic protein (GFAP) (red). Now we can visualise the three colour channels individually by slicing the Numpy array. The stack with index 0 corresponds to 'red', index 1 corresponds to 'green' and index 2 corresponds to 'blue':

PYTHON

```
red_channel   = img_col[:, :, 0]
```

OUTPUT

```
NameError: name 'img_col' is not defined
```

PYTHON

```
green_channel = img_col[:, :, 1]
```

OUTPUT

```
NameError: name 'img_col' is not defined
```

PYTHON

```
blue_channel  = img_col[:, :, 2]
```

OUTPUT

```
NameError: name 'img_col' is not defined
```

PYTHON

```
fig, ax = subplots(nrows=1, ncols=3, figsize=(20, 10))

imgplot_red = ax[0].imshow(red_channel, cmap="Reds")
```

PYTHON

```
imgplot_green = ax[1].imshow(green_channel, cmap="Greens")
```

PYTHON

```
imgplot_blue = ax[2].imshow(blue_channel, cmap="Blues")
```

PYTHON

```
fig.colorbar(imgplot_red, ax=ax[0], shrink=0.4)
```

PYTHON

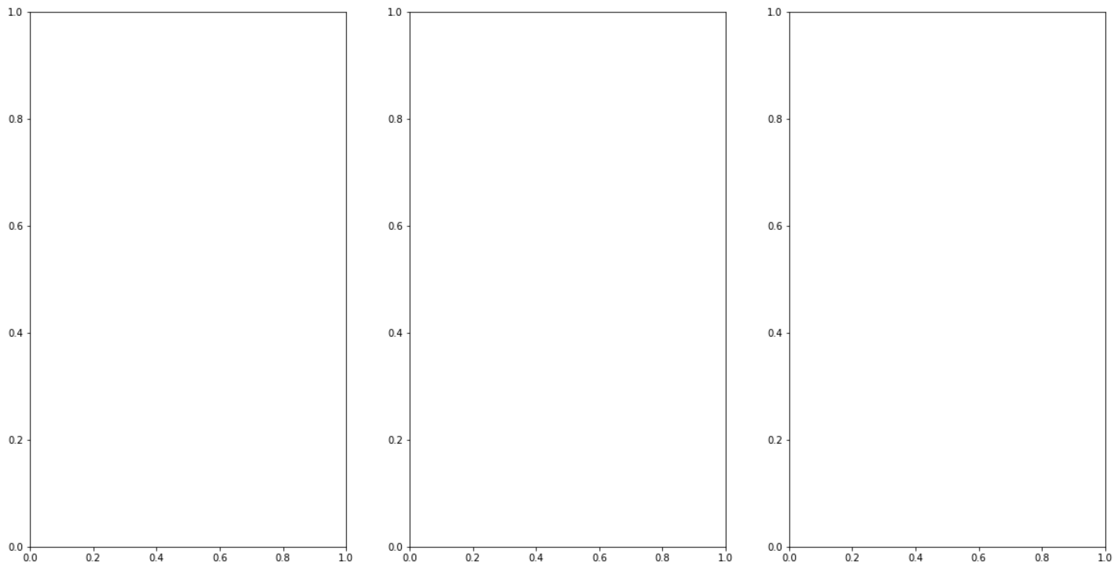
```
fig.colorbar(imgplot_green, ax=ax[1], shrink=0.4)
```

PYTHON

```
fig.colorbar(imgplot_blue, ax=ax[2], shrink=0.4);
```

PYTHON

```python
show()
```



This shows what colour combinations each of the pixels is made up of. Notice that the intensities go up to 255. This is because RGB (red, green and blue) colours are defined within the range 0-255. This gives a total of 16,777,216 possible colour combinations!

We can plot histograms of each of the colour channels.

PYTHON

```python
fig, ax = subplots(nrows=1, ncols=3, figsize=(20, 5))

ax[0].hist(red_channel.flatten(), bins=50)
```

PYTHON

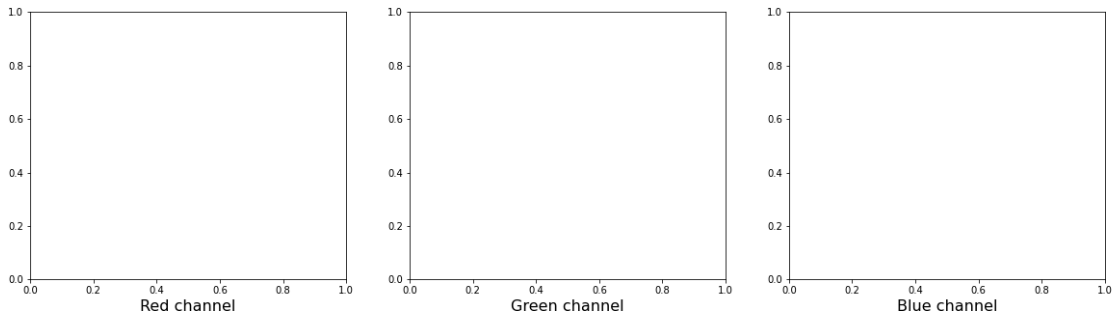
```python
ax[0].set_xlabel("Pixel intensity", fontsize=16)
ax[0].set_xlabel("Red channel")
ax[1].hist(green_channel.flatten(), bins=50)
```

PYTHON

```python
ax[1].set_xlabel("Pixel intensity", fontsize=16)
ax[1].set_xlabel("Green channel")
ax[2].hist(blue_channel.flatten(), bins=50)
```

PYTHON

```
ax[2].set_xlabel("Pixel intensity", fontsize=16)
ax[2].set_xlabel("Blue channel")

show()
```



Dealing with Large Images

Sometimes (or quite often, depending on the field of research), we have to deal with very large images that are composed of many pixels. It can be quite difficult to process these images, as they can require a lot of computer memory when they are processed. We will look at two different strategies for dealing with this problem: decreasing resolution and using patches from the original image. We will use the full-resolution version of the rat brain in the above example.

PYTHON

```
img_hr = imread('fig/rat_brain.jpg')
img_hr.shape
```

OUTPUT

```
PIL.UnidentifiedImageError: cannot identify image file 'fig/rat_brain.jpg'
NameError: name 'img_hr' is not defined
```

In fact, we can even get an warning from python that say something like “Image size (324649360 pixels) exceeds limit of 244158474 pixels, could be decompression bomb DOS attack.” This refers to malicious files which are designed to crash or cause disruption by using up a lot of memory.

We can get around this by changing the maximum pixel limit as follows.

To do this, we import Image from the Python Image Library PIL:

PYTHON

```
from PIL import Image

Image.MAX_IMAGE_PIXELS = 1000000000
```

Let’s try again. Be patient, it might take a moment.

PYTHON

```
img_hr = imread('fig/rat_brain.jpg')
```

OUTPUT

```
PIL.UnidentifiedImageError: cannot identify image file 'fig/rat_brain.jpg'
```

PYTHON

```python
img_hr.shape
```

OUTPUT

```
NameError: name 'img_hr' is not defined
```

Now we can plot the full high-resolution image:

PYTHON

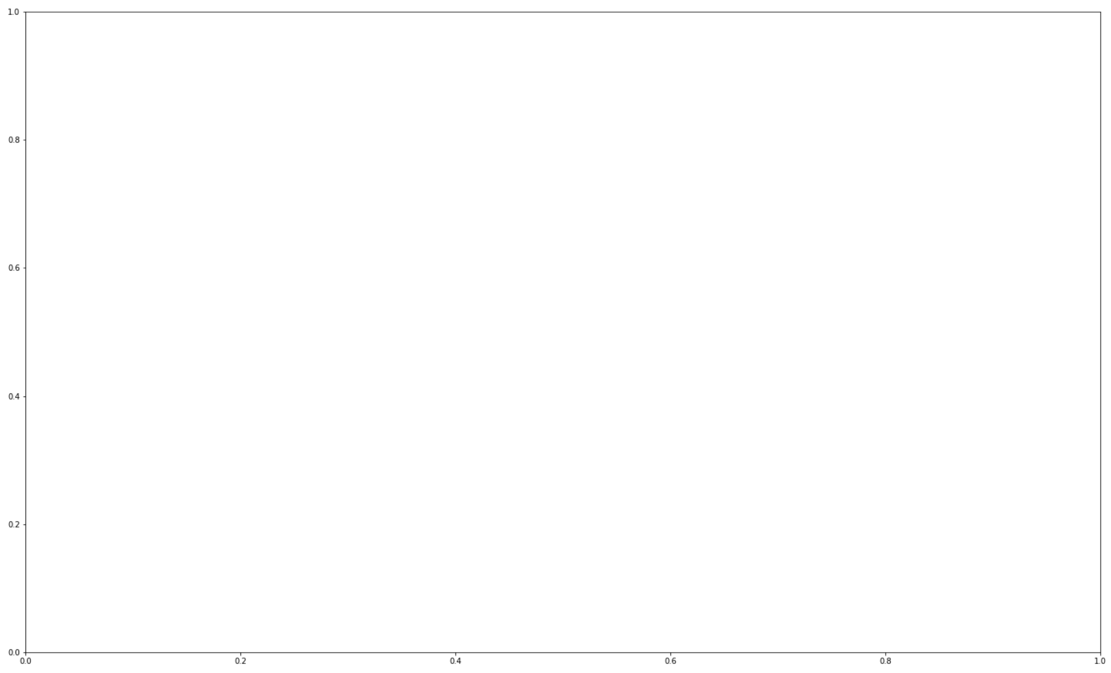
```python
fig, ax = subplots(figsize=(25, 15))

ax.imshow(img_hr, cmap='gray');
```

OUTPUT

```
NameError: name 'img_hr' is not defined
```

PYTHON

```python
show()
```



Although now we can plot this image, it consists of over 300 million pixels, and we could run into memory problems when trying to process it. One approach is simply to reduce the resolution. One way to do this is to import the image using Image from the PIL library that we imported above. This library gives us more tools to process images, including decreasing the resolution. It is a rich library with lots of useful tools. As always, having a look at the documentation and playing around is recommended!

We use `resize` to downsample the image:

PYTHON

```python
img_pil = Image.open('fig/rat_brain.jpg')
img_small = img_pil.resize((174, 187))

print(type(img_small))
```

OUTPUT

```
PIL.UnidentifiedImageError: cannot identify image file 'fig/rat_brain.jpg'
NameError: name 'img_pil' is not defined
NameError: name 'img_small' is not defined
```

Plotting should now be quicker.

PYTHON

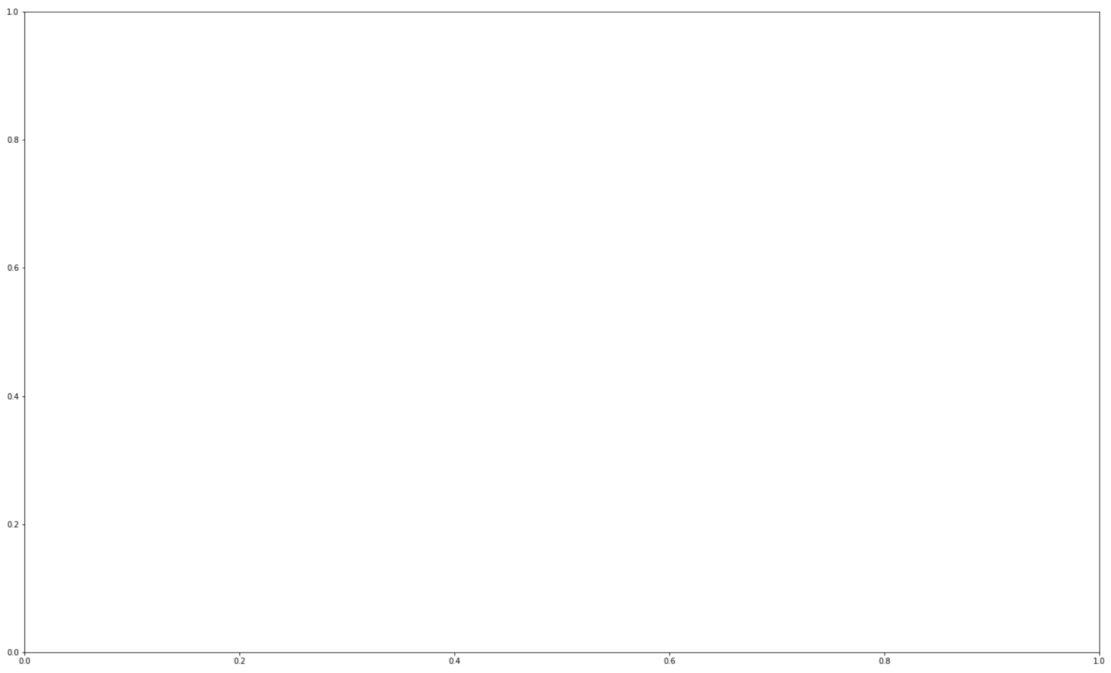
```python
fig, ax = subplots(figsize=(25, 15))

ax.imshow(img_small, cmap='gray');
```

OUTPUT

```
NameError: name 'img_small' is not defined
```

PYTHON

```python
show()
```



With this code, we have resized the image to 174 by 187 pixels. We should be aware though, that our image is no longer in a Numpy array form but rather it now has type 'PIL.Image.Image'. We can, however, easily convert it back into a Numpy array using `array`, if we wish.

PYTHON

```python
from numpy import array

img_numpy = array(img_small)
```

OUTPUT

```
NameError: name 'img_small' is not defined
```

PYTHON

```python
print(type(img_numpy))
```

OUTPUT

```
NameError: name 'img_numpy' is not defined
```

Often, we like to have full resolution images, as resizing causes a loss of information. An alternative approach to downsampling that is commonly used is to *patch* the images, i.e. divide the picture up into smaller chunks, or patches.

For this, we can use functionality from the Scikit-Learn library.

PYTHON

```python
from sklearn.feature_extraction.image import extract_patches_2d
```

'extract_patches_2d' is used to extract parts of the image. The shape of each patch as well as maxiaml number of patches can be specified.

PYTHON

```python
patches = extract_patches_2d(img_hr, (174, 187), max_patches=100)
```

OUTPUT

```
NameError: name 'img_hr' is not defined
```

PYTHON

```python
patches.shape
```

OUTPUT

```
NameError: name 'patches' is not defined
```

Note that patching itself can be a memory-intensive task. Extracting lots and lots of patches might take a long time. To look at the patches we can use a for loop:

PYTHON

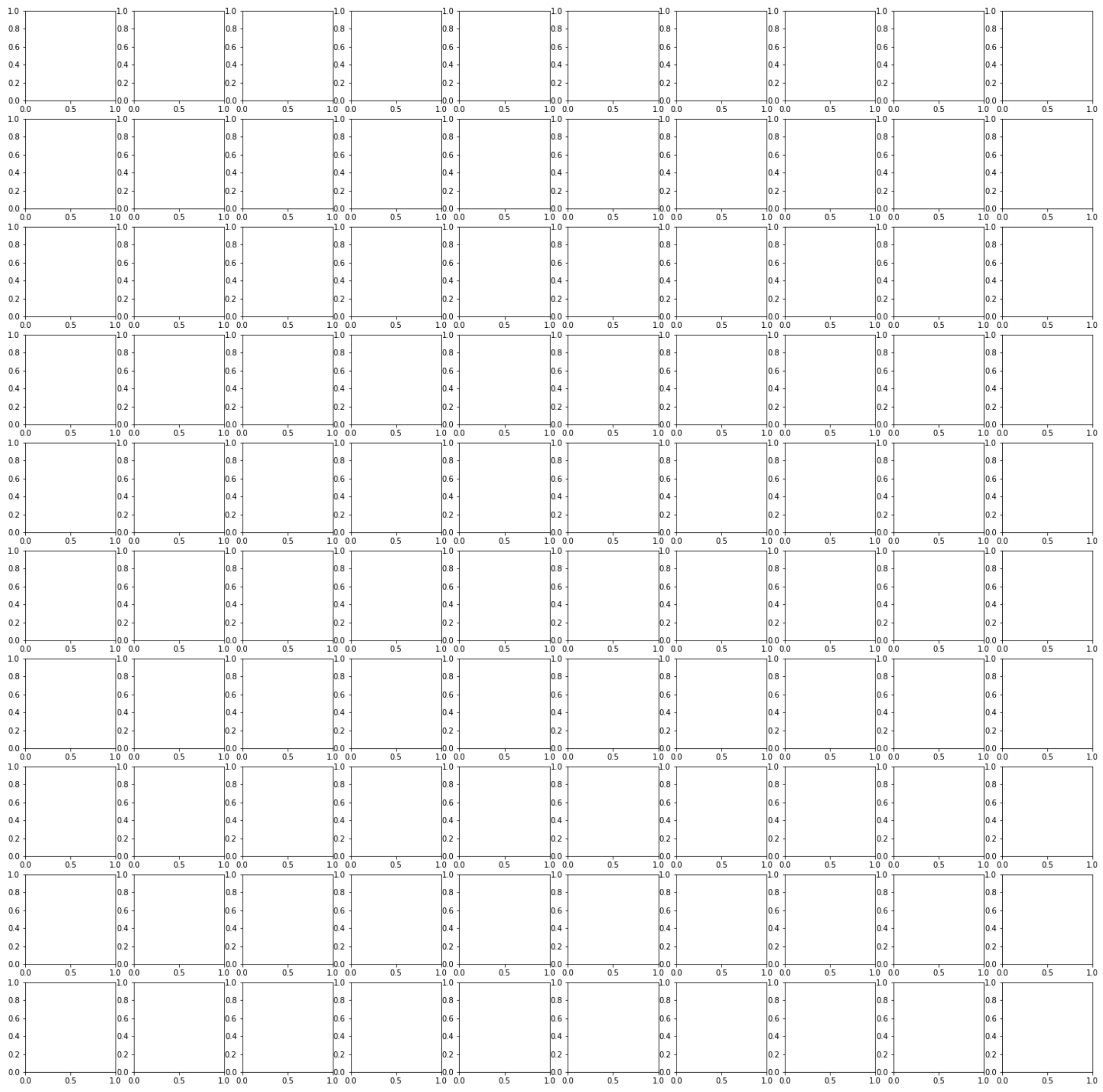
```python
fig, ax = subplots(nrows=10, ncols=10, figsize=(25, 25))

ax = ax.flatten()

for index in range(patches.shape[0]):
    ax[index].imshow(patches[index, :, :, :])
```

OUTPUT

```
NameError: name 'patches' is not defined
```

PYTHON

```
show()
```



Now, working with these smaller, individual patches will be much more manageable!

3D Images

Sometimes we might want to work with 3D images. A good example for this are MRI scans. These don't come as 'csv' format but in specialised image formats. One example is `nii`, the *Neuroimaging Informatics Technology Initiative (NIfTI)* open file format. For these types of images we will need special software. In particular, we will be using the open source library called **nibabel**. Documentation for this package is available at https://nipy.org/nibabel/.

As it is not contained in your Python installation by default, it needs to be installed first.

To install it, please run:

```
conda install -c conda-forge nibabel
```

in your command line or terminal if you have an **Anaconda distribution** of Python.

Alternatively, you can install it using:

```
pip install nibabel
```

in your command line or terminal.

PYTHON

```python
import nibabel as nib
```

The package is now available for use. If a function comes from that package, we call it by referring to the package using `nib`, followed by a period and the name of the function:

PYTHON

```python
img_3d = nib.load('fig/brain.nii')

img_data = img_3d.get_fdata()

print(type(img_data))
```

OUTPUT

```
<class 'numpy.memmap'>
```

PYTHON

```python
print(img_data.shape)
```

OUTPUT

```
(256, 256, 124)
```

We can see that this image has three dimensions, and a total of 256 x 256 x 124 volume pixels (or voxels). To visualise our image, we can plot one slice at a time. Below, we show three different slices, in the transverse direction (from chin to the top of the head. To access an image from the transverse direction, you pick a single value from the third dimension of the image:

PYTHON

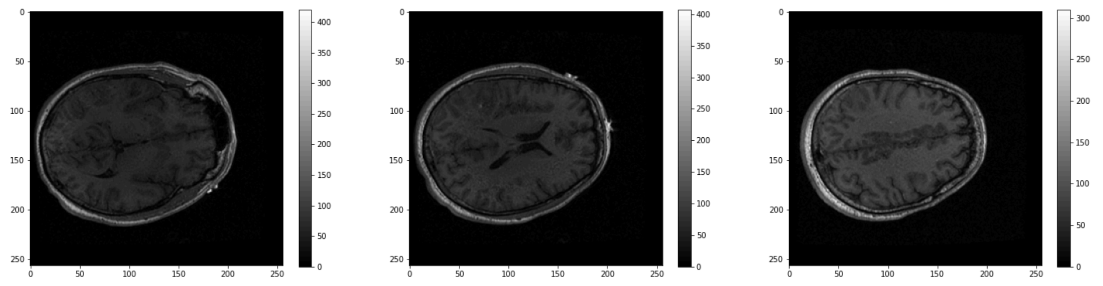
```
fig, ax = subplots(ncols=3, figsize=(25, 15))

p1 = ax[0].imshow(img_data[:, :, 60], cmap='gray')
p2 = ax[1].imshow(img_data[:, :, 75], cmap='gray')
p3 = ax[2].imshow(img_data[:, :, 90], cmap='gray')

fig.colorbar(p1, ax=ax[0], shrink=0.4)
```

PYTHON

```
fig.colorbar(p2, ax=ax[1], shrink=0.4)
```

PYTHON

```
fig.colorbar(p3, ax=ax[2], shrink=0.4);

show()
```



These look fairly dark. We can improve the contrast, by adjusting the intensity range. This requires setting of the keyword arguments `vmin` and `vmax`.

`vmin` and `vmax` define the data range that the colormap (in our case the ‘grey’ map) covers. By default, the colormap covers the complete value range of the supplied data. For an image that will be somewhere between 0 and 255. If we want to brighten up the darker shades of grey, we can reduce the value of `vmax`

Expanding the above code:

PYTHON

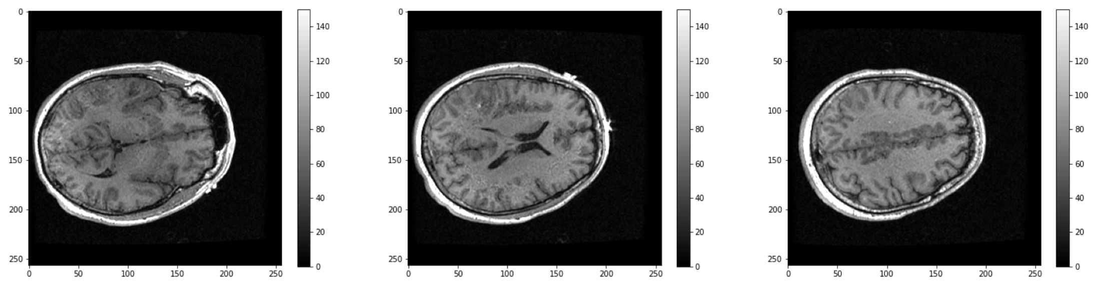
```python
fig, ax = subplots(ncols=3, figsize=(25, 15))

p1 = ax[0].imshow(img_data[:, :, 60], cmap='gray', vmin=0, vmax=150)
p2 = ax[1].imshow(img_data[:, :, 75], cmap='gray', vmin=0, vmax=150)
p3 = ax[2].imshow(img_data[:, :, 90], cmap='gray', vmin=0, vmax=150)

fig.colorbar(p1, ax=ax[0], shrink=0.4)
```

PYTHON

```python
fig.colorbar(p2, ax=ax[1], shrink=0.4)
```

PYTHON

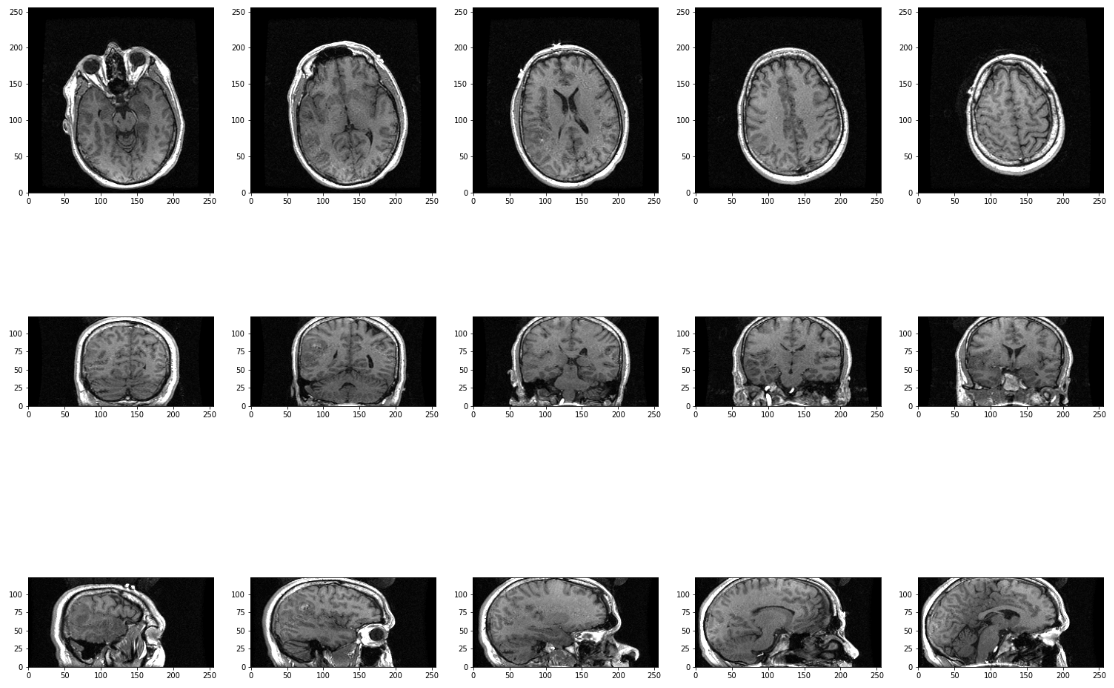
```python
fig.colorbar(p3, ax=ax[2], shrink=0.4);

show()
```



What about the other dimensions? We can also plot coronal and sagittal slices but note that the respective slices have different pixel resolution.

PYTHON

```python
fig, ax = subplots(nrows=3, ncols=5, figsize=(26, 18))

t1 = ax[0, 0].imshow(img_data[:, :, 45].T, cmap='gray', vmin=0, vmax=150, origin='lower')
t2 = ax[0, 1].imshow(img_data[:, :, 60].T, cmap='gray', vmin=0, vmax=150, origin='lower')
t3 = ax[0, 2].imshow(img_data[:, :, 75].T, cmap='gray', vmin=0, vmax=150, origin='lower')
t4 = ax[0, 3].imshow(img_data[:, :, 90].T, cmap='gray', vmin=0, vmax=150, origin='lower')
t5 = ax[0, 4].imshow(img_data[:, :, 105].T, cmap='gray', vmin=0, vmax=150, origin='lower')

c1 = ax[1, 0].imshow(img_data[:, 50, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')
c2 = ax[1, 1].imshow(img_data[:, 75, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')
c3 = ax[1, 2].imshow(img_data[:, 90, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')
c4 = ax[1, 3].imshow(img_data[:, 105, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')
c5 = ax[1, 4].imshow(img_data[:, 120, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')

s1 = ax[2, 0].imshow(img_data[75, :, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')
s2 = ax[2, 1].imshow(img_data[90, :, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')
s3 = ax[2, 2].imshow(img_data[105, :, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')
s4 = ax[2, 3].imshow(img_data[120, :, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')
s5 = ax[2, 4].imshow(img_data[135, :, :].T, cmap='gray', vmin=0, vmax=150, origin='lower');

show()
```



Now, we can see all three viewing planes for this 3-dimensional brain scan!

Exercises

END OF CHAPTER EXERCISES

Assignment

Using the image from the beginning of this lesson, “rat_cerebellum.jpg”, do the following tasks:

1. Import the image and display it.
2. Show histograms of each of the colour channels and plot the contributions of each of the RGB colours separately.
3. Create three different binary masks using manually determined thresholds: one for mostly red pixels, one for mostly green pixels, and one for mostly blue pixels. Note that you can apply conditions that are either greater than or smaller than a threshold of your choice.
4. Plot the three masks and the corresponding masked images.
5. Using your masks, approximate the relative amounts of synaptophysin, IP3R, and DNA in the image. To do this, you can assume that the number of red pixels represents synaptophysin, green pixels represents IP3R and blue pixels represent DNA. The results will vary depending on the setting of the thresholds. How do different threshold values change your results?
6. Change the resolution of your image to different values. How does the resolution affect your results?

Solution

Q1

PYTHON

```
## Import the image
from matplotlib.image import imread

img_task = imread('fig/rat_cerebellum.jpg')
```

OUTPUT

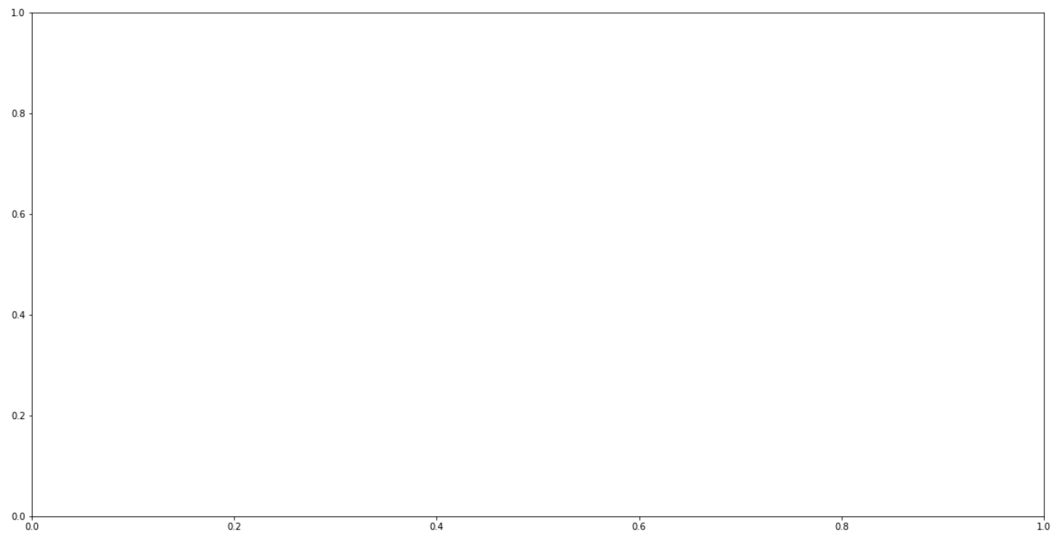
```
PIL.UnidentifiedImageError: cannot identify image file 'fig/rat_cerebellum.jpg'
```

PYTHON

```
## Display the image

from matplotlib.pyplot import subplots, show

fig, ax = subplots(figsize=(20, 10))

ax.imshow(img_task, cmap='gray');
```

PYTHON

```
show()
```



Q2

PYTHON

```python
red_channel   = img_task[:, :, 0]
```

PYTHON

```python
green_channel = img_task[:, :, 1]
```

PYTHON

```python
blue_channel  = img_task[:, :, 2]
```

PYTHON

```python
fig, ax = subplots(ncols=3, figsize=(20, 5))

ax[0].hist(red_channel.flatten(), bins=50)
```

PYTHON

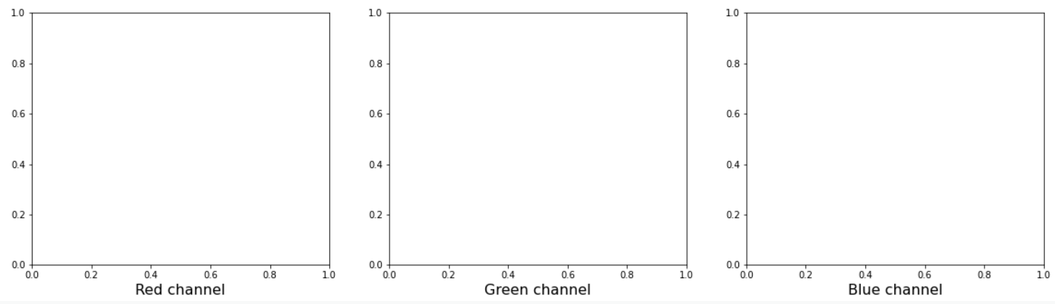
```python
ax[0].set_xlabel("Pixel intensity", fontsize=16)
ax[0].set_xlabel("Red channel")
ax[1].hist(green_channel.flatten(), bins=50)
```

PYTHON

```python
ax[1].set_xlabel("Pixel intensity", fontsize=16)
ax[1].set_xlabel("Green channel")
ax[2].hist(blue_channel.flatten(), bins=50)
```

PYTHON

```python
ax[2].set_xlabel("Pixel intensity", fontsize=16)
ax[2].set_xlabel("Blue channel");

show()
```



PYTHON

```
fig, ax = subplots(ncols=3, figsize=(20, 10))

imgplot_red   = ax[0].imshow(red_channel, cmap="Reds")
```

PYTHON

```
imgplot_green = ax[1].imshow(green_channel, cmap="Greens")
```

PYTHON

```
imgplot_blue  = ax[2].imshow(blue_channel, cmap="Blues")
```

PYTHON

```
fig.colorbar(imgplot_red,   ax=ax[0], shrink=0.5)
```

PYTHON

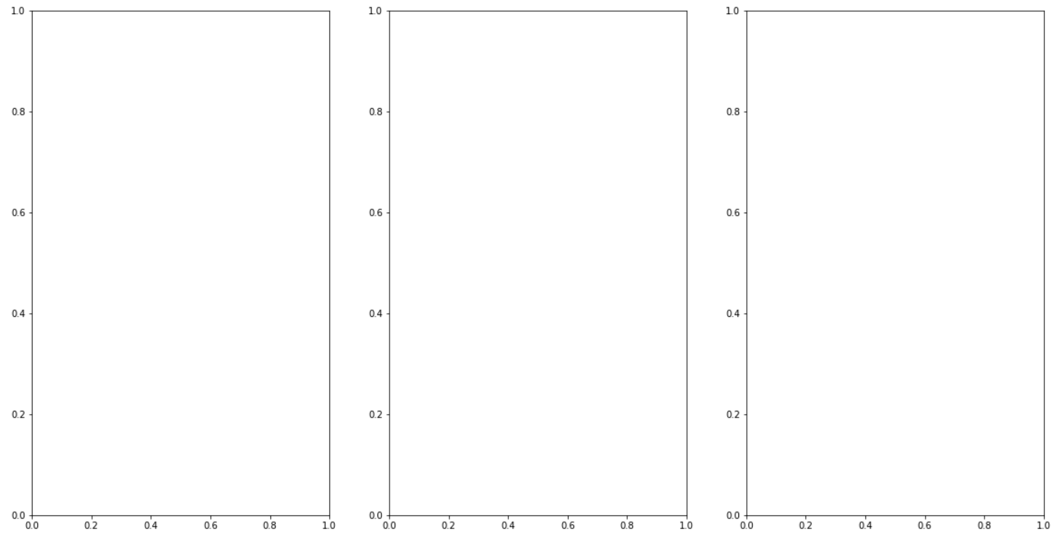
```
fig.colorbar(imgplot_green, ax=ax[1], shrink=0.5)
```

PYTHON

```
fig.colorbar(imgplot_blue,  ax=ax[2], shrink=0.5);
```

PYTHON

```
show()
```



Q3-4

PYTHON

```python
red_mask   = red_channel   > 120
```

PYTHON

```python
green_mask = green_channel > 100
```

PYTHON

```python
blue_mask  = blue_channel  > 100
```

PYTHON

```python
red_masked   = red_channel*red_mask
```

PYTHON

```python
green_masked = green_channel*green_mask
```

PYTHON

```python
blue_masked  = blue_channel*blue_mask
```

PYTHON

```python
fig, ax = subplots(nrows=3, ncols=2, figsize=(18, 20))

ax[0, 0].imshow(red_mask, cmap='gray')
```

PYTHON

```python
ax[0, 0].set_title('Red binary mask', fontsize=16)
ax[0, 1].imshow(red_masked, cmap='Reds')
```

PYTHON

```python
ax[0, 1].set_title('Masked image', fontsize=16)
ax[1, 0].imshow(green_mask, cmap='gray')
```

PYTHON

```python
ax[1, 0].set_title('Green binary mask', fontsize=16)
ax[1, 1].imshow(green_masked, cmap='Greens')
```

PYTHON

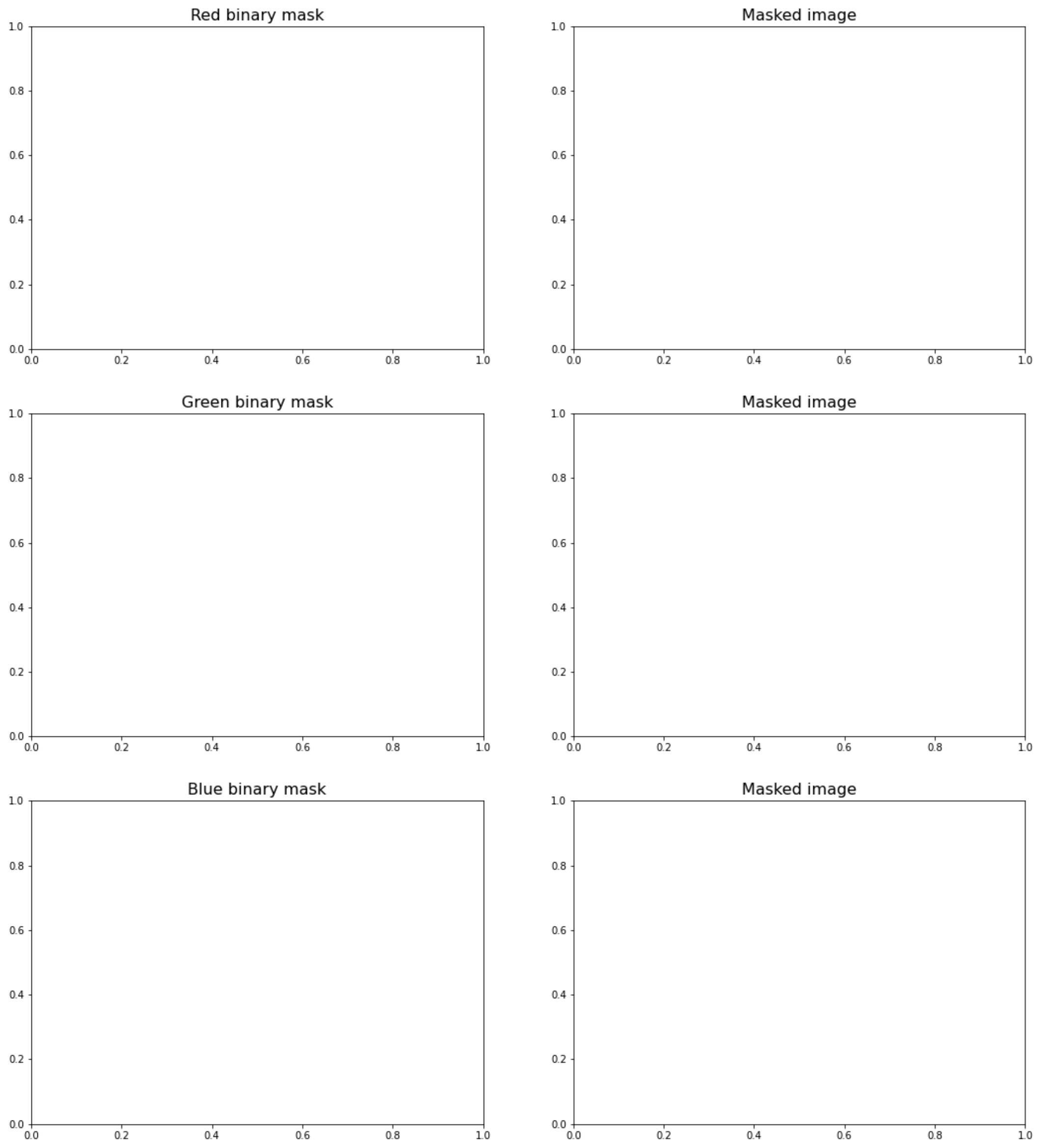
```python
ax[1, 1].set_title('Masked image', fontsize=16)
ax[2, 0].imshow(blue_mask, cmap='gray')
```

PYTHON

```python
ax[2, 0].set_title('Blue binary mask', fontsize=16)
ax[2, 1].imshow(blue_masked, cmap='Blues')
```

PYTHON

```python
ax[2, 1].set_title('Masked image', fontsize=16);

show()
```



Q5

PYTHON

```
from numpy import sum as numpy_sum
total_pixels = img_task.shape[0]*img_task.shape[1]

red_counts   = numpy_sum(red_mask)
green_counts = numpy_sum(green_mask)
blue_counts  = numpy_sum(blue_mask)
print("Approximately %d"%(red_counts/total_pixels*100)+"% of the image is synaptophysin")
print("Approximately %d"%(green_counts/total_pixels*100)+"% of the image is IP3R")
print("Approximately %d"%(blue_counts/total_pixels*100)+"% of the image is DNA")
```

OUTPUT

```
NameError: name 'img_task' is not defined
NameError: name 'red_mask' is not defined
NameError: name 'green_mask' is not defined
NameError: name 'blue_mask' is not defined
NameError: name 'red_counts' is not defined
NameError: name 'green_counts' is not defined
NameError: name 'blue_counts' is not defined
```

Q6

```
[ad libitum]
```

KEY POINTS

- `imread` function can interpret many different image formats.
- Masking isolates pixels whose intensity value is below a certain threshold.
- The colour images are comprised of three channels (corresponding to red, green and blue intensities).
- Python Image Library (PIL) helps to set high pixel limit for larger images.

Content from Time Series

Last updated on 2024-05-24 | Edit this page

Download Chapter PDF

Download Chapter notebook (ipynb)

Mandatory Lesson Feedback Survey

OVERVIEW

Questions

- How is timeseries data visualised?
- Why do we need to tidy up/filter the data?
- How to study correlation among timeseries data points?

Objectives

- Learning ways to display multiple time series.

- Understanding why filtering is needed.
- Explaining fourier spectrum of time series.
- Knowledge of correlation matrix of time series.

PREREQUISITES

- Dataframes 1 and 2
- Image Handling
- Basics of Numpy Arrays

PYTHON

```python
from pandas import read_csv

from numpy import arange, zeros, linspace, sin, pi, c_, mean, var, array
from numpy import correlate, corrcoef, fill_diagonal, amin, amax, asarray
from numpy import around
from numpy.ma import masked_less, masked_greater

from matplotlib.pyplot import subplots, yticks, legend, axis, figure, show
```

PYTHON FUNCTION

Please execute the following function definition before proceeding. The function code takes data and creates a plot of all columns as time series, one above the other. When you execute the function code nothing happens. Similar to the import, running a function code will only activate it and make it available for later use.

PYTHON

```python
def plot_series(data, sr):
    '''
    Time series plot of multiple time series
    Data are normalised to mean=0 and var=1

    data: nxm numpy array. Rows are time points, columns are recordings
    sr: sampling rate, same time units as period
    '''

    samples = data.shape[0]
    sensors = data.shape[1]

    period = samples // sr

    time = linspace(0, period, period*sr)

    offset = 5 # for mean=0 and var=1 normalised data

    # Calculate means and standard deviations of all columns
    means = data.mean(axis=0)
    stds = data.std(axis=0)

    # Plot each series with an offset
    fig, ax = subplots(figsize=(7, 8))

    ax.plot(time, (data - means)/stds + offset*arange(sensors-1,-1,-1));

    ax.plot(time, zeros((samples, sensors)) + offset*arange(sensors-1,-1,-1),'--',color='gray')

    yticks([]);

    names = [str(x) for x in range(sensors)]
    legend(names)

    ax.set(xlabel='Time')

    axis('tight');

    return fig, ax
```

Example: Normal and Pathological EEG

As an example, let us import two sets of time series data and convert them to Numpy arrays, here called *data_back* and *data_epil*. They represent human electroencephalogram (EEG) as recorded during normal *background* activity and during an epileptic seizure called *absence* seizure.

PYTHON

```python
df_back = read_csv("data/EEG_background.txt", delim_whitespace=True)
df_epil = read_csv("data/EEG_absence.txt", delim_whitespace=True)

sr = 256     # 1 / seconds
period = 6   # seconds
channels = 10

d1 = df_back.to_numpy()
d2 = df_epil.to_numpy()

data_back = d1[:period*sr:, :channels]
data_epil = d2[:period*sr:, :channels]
```

The `read_csv` function is used with the keyword argument `delim_whitespace`. When set to True, this allows to import data that are space-separated (rather than comma-separated). If you check the data .txt files, you will see that the numbers (which represent voltages) are indeed separated by space, not comma.

Next, three constants are assigned: The sampling rate, sr, which is given in number of samples recorded per seconds; the duration of the recording, period, which is given in seconds; and the number of columns, referred to as channels, to be extracted from the recording. We use the first 10 columns in this lesson.

The data are then converted from a data frame to Numpy arrays.

To see the names of the channels (or recording sensors) we can use `head`.

PYTHON

```python
df_back.head()
```

OUTPUT

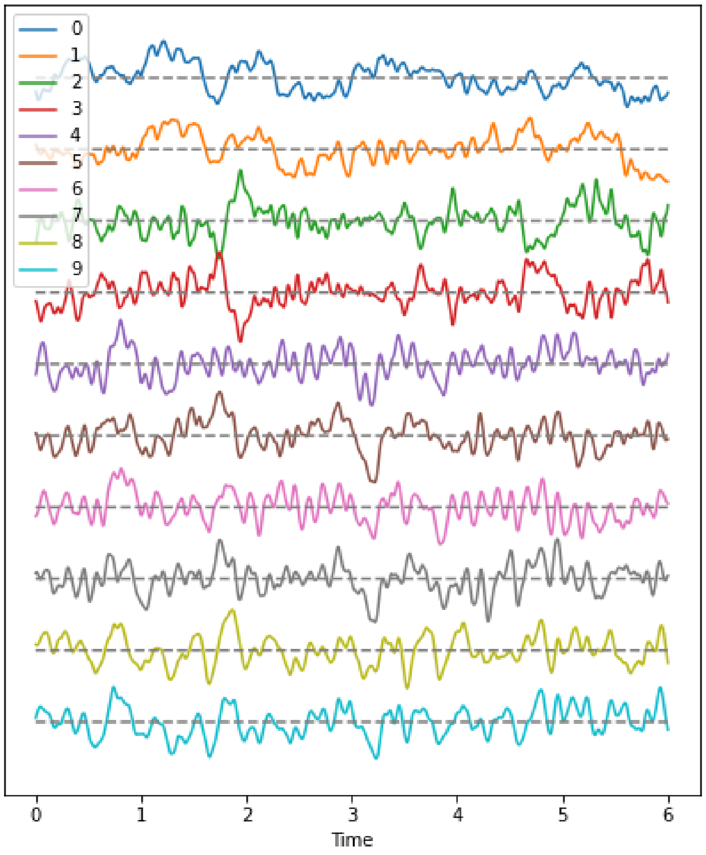
```
       FP1       FP2        F3       F4  ...      EO2      EM1      EM2      PHO
0  -7.4546   22.8428   6.28159  15.6212  ...  13.7021  12.9109  13.7034  9.37573
1 -11.1060   21.4828   6.89088  15.0562  ...  13.7942  13.0194  13.7628  9.44731
2 -14.4000   20.0907   7.94856  14.1624  ...  13.8982  13.1116  13.8239  9.51796
3 -17.2380   18.7206   9.36857  13.0093  ...  14.0155  13.1927  13.8914  9.58770
4 -19.5540   17.4084  11.06040  11.6674  ...  14.1399  13.2692  13.9652  9.65654

[5 rows x 28 columns]
```

The row indices and column names for the seizure data look the same. The names of the recording channels are from the commonly used 10-20 system to record voltages of brain activity from the scalp in humans. E.g. 'F' stands for the frontal lobe.

We can now use the plot function from above to plot the data. Note from the definition (first line) that two input arguments are required: the data set and the sampling rate.

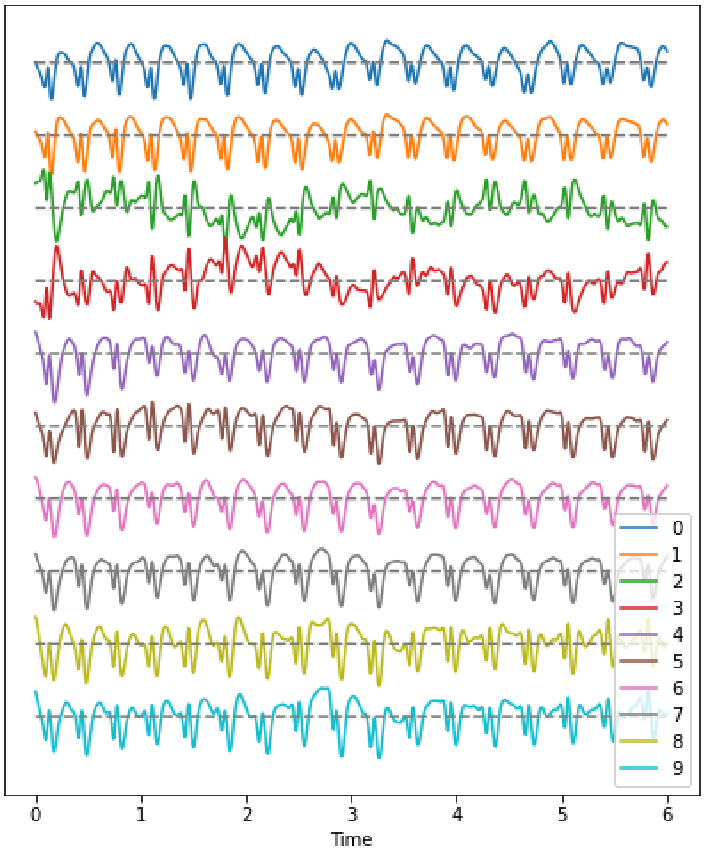
PYTHON

```python
plot_series(data_back, sr)
show()
```



PYTHON

```
plot_series(data_epil, sr);
show()
```



Observations

1. Background:

- There are irregular oscillations of all recorded brain potentials.
- Oscillations recorded at different locations above the brain differ.
- Oscillations are not stable but modulated over time.
- There are different frequency components in each trace.

2. Epileptic Seizure:

- There are regular oscillations.
- Oscillations recorded at different locations are not identical but similar or at least related in shape.
- Despite some modulation, oscillations are fairly stable over time.
- There are repetitive motifs composed of two major components throughout the recording, a sharp *spike* and a slow *wave*.

Task

Quantify features of these time series data to get an overview. As a univariate feature we can use the frequency content. This takes into account the fact that the rows (or samplles) are not independent of each other but are organised along the time axis. In consequence, there are correlations between data points along the rows of each column and the **Fourier spectrum** can be used to identify these.

The Fourier spectrum assumes that the data are stationary and can be thought of as a superposition of regular sine waves with different frequencies. Its output will show which of the frequencies are present in the data and also their respective amplitudes.

As a bivariate feature we can use the cross-correlation matrix.

Work Through Example

Check the Numpy array containing the background and seizure data.

PYTHON

```python
print(data_back.shape, data_epil.shape)
```

OUTPUT

```
(1536, 10) (1536, 10)
```

There are 1536 rows and 10 columns.

Display data with offset

Take a look at the code of the function `plot_series` that creates the time series plot. It requires the input of a data file where the row index is interpreted as time. In addition, the sampling rate sr is required to be able to extract the time scale. The sampling rate specifies the number of samples recorded per unit time.

The sensors or recording channels are assumed to be in the columns.

PYTHON

```python
def plot_series(data, sr):
    '''
    Time series plot of multiple time series
    Data are normalised to mean=0 and var=1

    data: nxm numpy array. Rows are time points, columns are channels
    sr: sampling rate, same time units as period
    '''

    samples = data.shape[0]
    sensors = data.shape[1]

    period = samples // sr

    time = linspace(0, period, period*sr)

    offset = 5 # for mean=0 and var=1 normalised data

    # Calculate means and standard deviations of all columns
    means = data.mean(axis=0)
    stds = data.std(axis=0)

    # Plot each series with an offset
    fig, ax = subplots(figsize=(7, 8))

    ax.plot(time, (data - means)/stds + offset*arange(sensors-1,-1,-1));

    ax.plot(time, zeros((samples, sensors)) + offset*arange(sensors-1,-1,-1),'--',color='gray');

    yticks([]);

    names = [str(x) for x in range(sensors)]
    legend(names)

    ax.set(xlabel='Time')

    axis('tight');

    return fig, ax
```

The declaration syntax `def` is followed by the function name and, in parenthesis, the input arguments. It is completed with a colon.

Following the declaration is the documentation of the function.

Next comes the function code, all indented.

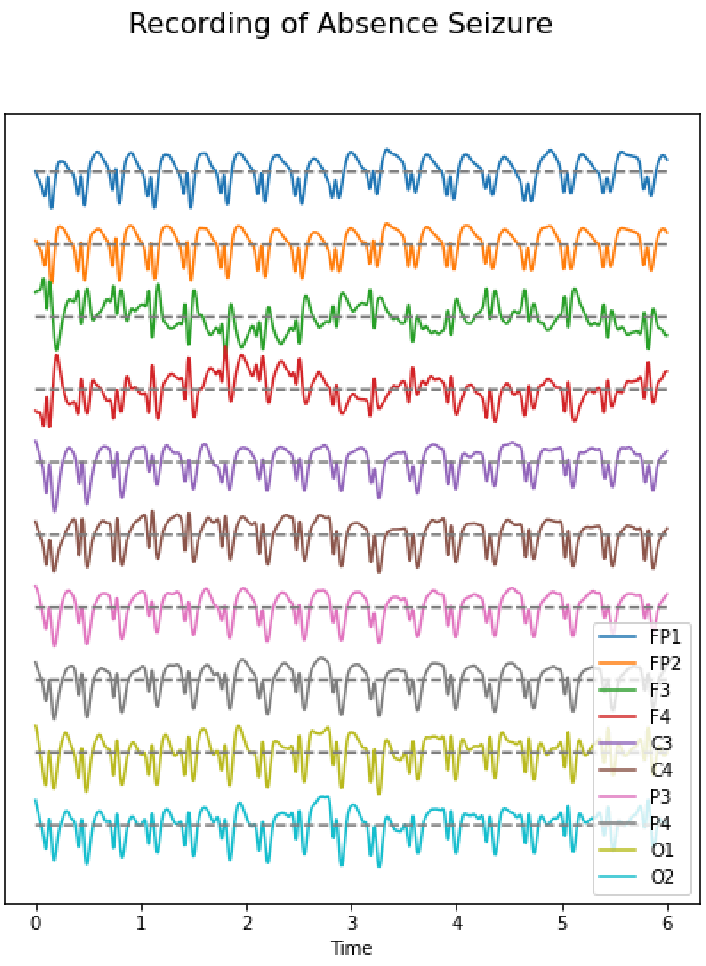
The function closes with the optional output syntax `return` and any number of returned variables, anything that might be used as a product of running the function.

In our example, the figure environment and the coordinate system are ‘returned’, and can in principle be used to further modify the plot.

Here is how to call the function and then add a title and the sensor names to the display.

PYTHON

```python
(fig, ax) = plot_series(data_epil, sr)

names = df_back.columns[:channels]

fig.suptitle('Recording of Absence Seizure', fontsize=16);

legend(names);

show()
```



A function usually (but not necessarily) takes in one or several variables or values, processes them, and produces a specific result. The variable(s) given to a function and those produced by it are referred to as input arguments, and outputs respectively.

There are different ways to create functions in Python. In this course, we will be using `def` to implement our own functions. This is the easiest, and by far the most common method for declaring functions. The structure of a typical function defined using `def` can be see in the `plot_series` example:

There are several points to remember in relation to functions:

- The name of a function follows same principles as that of any other variable. It must be in lower-case characters.
- The input arguments of a function, e.g. *data* and *sr* in the example, are essentially variables whose scope is the function. That is, they are only accessible within the function itself, and not from outside the function.
- Variables defined inside of a function, should not use the same name as variables defined outside. Otherwise they may override each other.

It is important for a function to only perform one specific task. As such it can be used independent of the current context. Try to avoid incorporating separable tasks into a single function.

Once you start creating functions for different purposes you can start to build your own library of ready-to-use functions to address different needs. This is the primary principle of a popular programming paradigm known as functional programming.

Filtering

Data sets with complex waveforms contain many different components which may or may not be relevant for a specific question.
Data filtering is applied to take out specific components. Component in this context refers to ‘frequency’, i.e. the number of cycles per unit of time. Thus a small number refers to low frequencies with long periods (cycles) and a large number to high frequencies with short periods.

Let us see a simple example how both low- and high-frequency components can be filtered (suppressed) in the example time series.

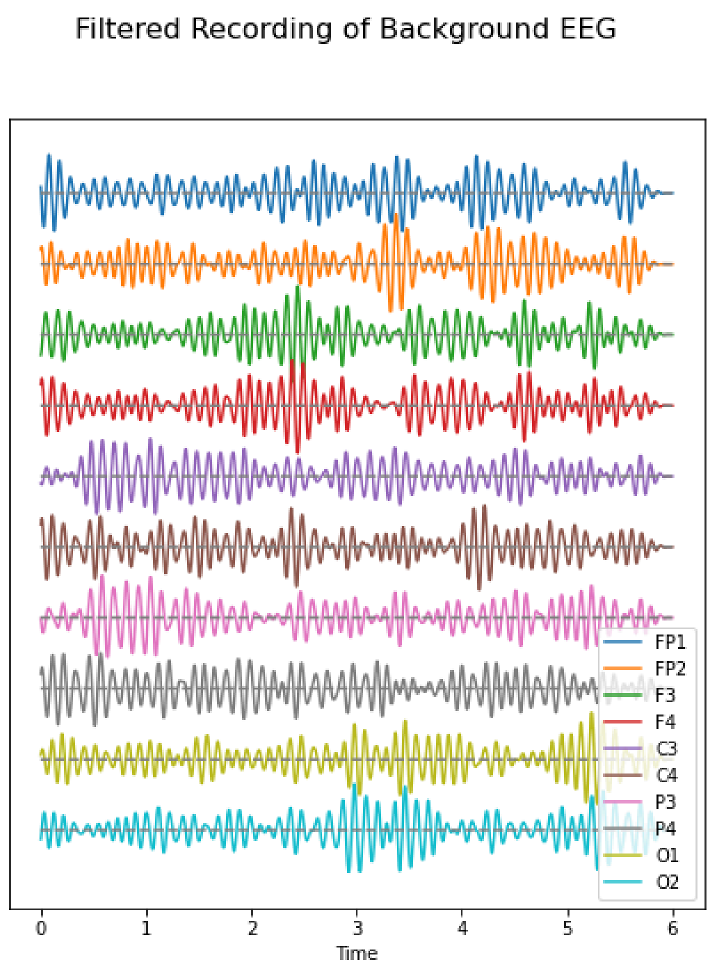
Here is a simple function which takes two additional input arguments, the low and the high cut-off.

PYTHON

```python
def data_filter(data, sr, low, high):
    """
    Filtering of multiple time series.

    data: nxm numpy array. Rows are time points, columns are recordings
    sr: sampling rate, same time units as period

    low:  Low cut-off frequency (high-pass filter)
    high: High cut-off frequency (low-pass filter)

    return: filtered data
    """

    from scipy.signal import butter, sosfilt

    order = 5

    filter_settings = [low, high, order]

    sos = butter(order, (low,high), btype='bandpass', fs=sr, output='sos')

    data_filtered = zeros((data.shape[0], data.shape[1]))

    for index, column in enumerate(data.transpose()):
        forward = sosfilt(sos, column)
        backwards = sosfilt(sos, forward[-1::-1])
        data_filtered[:, index] = backwards[-1::-1]

    return data_filtered
```

PYTHON

```python
data_back_filt = data_filter(data_back, sr, 8, 13)

(fig, ax) = plot_series(data_back_filt, sr)

fig.suptitle('Filtered Recording of Background EEG', fontsize=16);

legend(names);

show()
```



The frequency range from 8 to 13 Hz is referred to as alpha band in the electroencephalogram. It is thought that this represents a kind of idling rhythm of the brain, i.e. an activity where the brain is not actively processing sensory input.

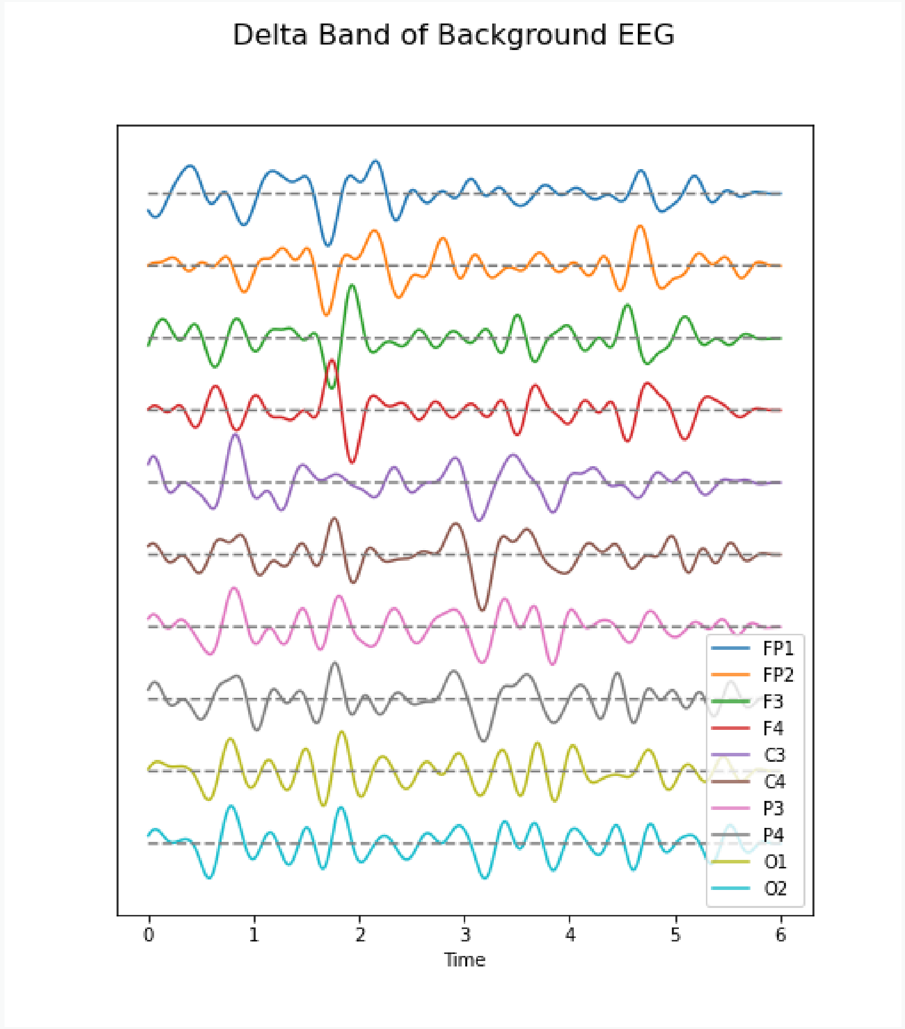
DIY1: BAND-PASS FILTERED DATA

Create figures of the delta (1-4 Hz) band for both the background and the seizure EEG. Note the differences.

Solution

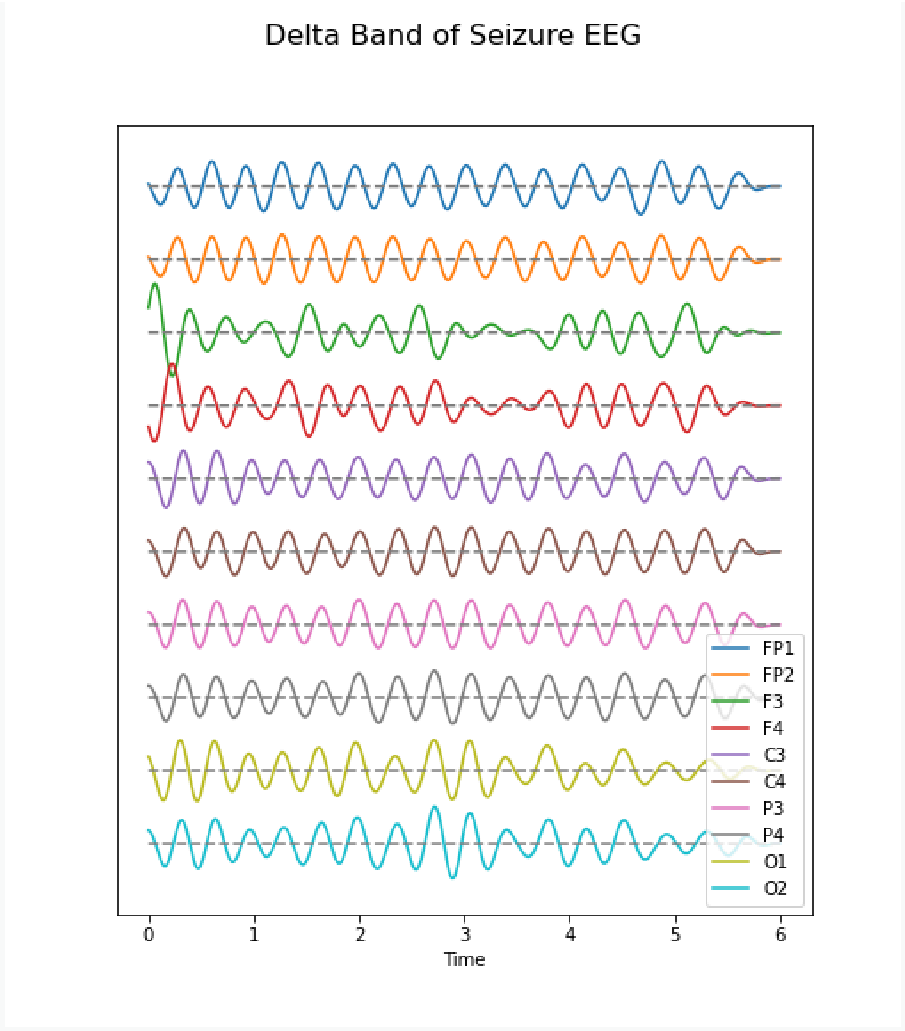
PYTHON

```python
data_back_filt = data_filter(data_back, sr, 1, 4)

(fig, ax) = plot_series(data_back_filt, sr)

fig.suptitle('Delta Band of Background EEG', fontsize=16);

legend(names);

show()
```



PYTHON

```python
data_epil_filt = data_filter(data_epil, sr, 1, 4)

(fig, ax) = plot_series(data_epil_filt, sr)

fig.suptitle('Delta Band of Seizure EEG', fontsize=16);

legend(names);

show()
```



Fourier Spectrum

The Fourier spectrum decomposes the time series into a sum of sine waves. The spectrum gives the coefficients of each of the sine wave components. The coefficients are directly related to the amplitudes needed to optimally fit the sum of all sine waves to recreate the original data.

However, the assumption behind the Fourier transform is that the data are provided as in infinitely long stationary time series. These assumptions are not fulfilled as the data are finite and stationarity of a biological system can typically not be guaranteed. Thus, interpretation needs to be cautious.

Fourier Transform of EEG data

We import the Fourier transform function `fft` from **scipy.fftpack** and can use it to transform all columns at the same time.

PYTHON

```
from scipy.fftpack import fft

data_back_fft = fft(data_back, axis=0)
```

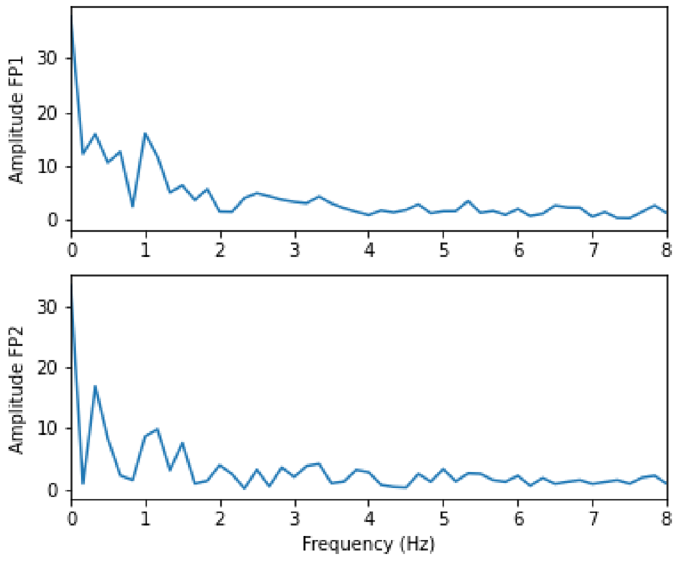
To plot the results, a couple of steps are required.
First, we obtain a Fourier spectrum for every data column, so we need to define how many plots we want to have. If we take only columns, we can display them all in one go.

Second, the Fourier transform results in twice the number of complex coefficients (positive and negative) of which we only need the first half.

Third, the Fourier transform outputs complex numbers. To display the ‘amplitude’ of a frequency (the coefficient corresponding to the amplitude of the sine wave with that frequency) we take the absolute value of the complex numbers.

PYTHON

```python
no_win = 2

rows = data_back.shape[0]

freqs = (sr/2)*linspace(0, 1, int(rows/2))

amplitudes_back = (2.0 / rows) * abs(data_back_fft[:rows//2, :2])


fig, axes = subplots(figsize=(6, 5), ncols=1, nrows=no_win, sharex=False)

names = df_back.columns[:2]

for index, ax in enumerate(axes.flat):
    axes[index].plot(freqs, amplitudes_back[:, index])
    axes[index].set_xlim(0, 8)
    axes[index].set(ylabel=f'Amplitude {names[index]}')

axes[index].set(xlabel='Frequency (Hz)');

show()
```



We can see that in these two channels, the main amplitude contributions lie in the low frequencies, below 2 Hz.

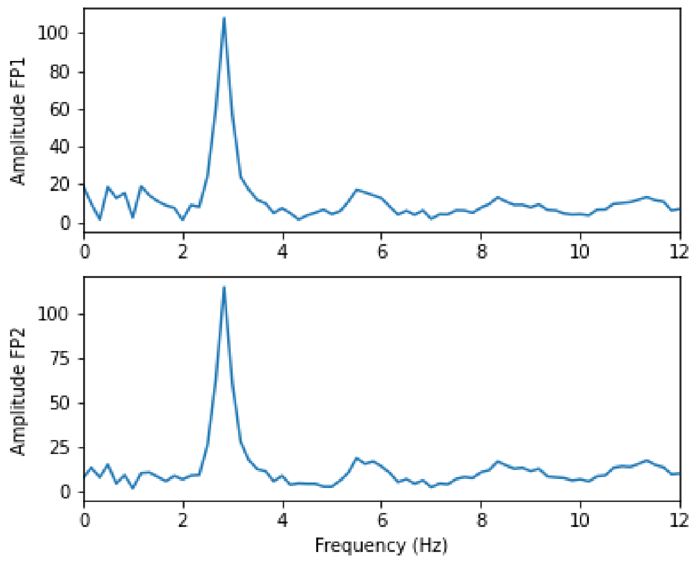
Let us compare the corresponding figure for the case of seizure activity:

PYTHON

```python
data_epil_fft = fft(data_epil, axis=0)
```

PYTHON

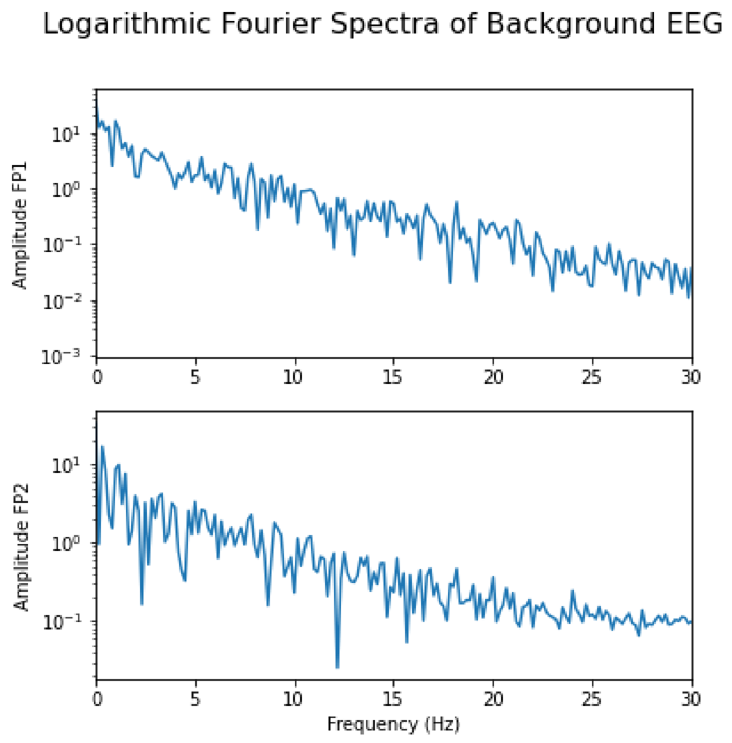
```python
fig, axes = subplots(figsize=(6, 5), ncols=1, nrows=no_win, sharex=False)

names = df_epil.columns[:2]

amplitudes_epil = (2.0 / rows) * abs(data_epil_fft[:rows//2, :2])

for index, ax in enumerate(axes.flat):
    axes[index].plot(freqs, amplitudes_epil[:, index])
    axes[index].set_xlim(0, 12)
    axes[index].set(ylabel=f'Amplitude {names[index]}')

axes[index].set(xlabel='Frequency (Hz)');

show()
```



The main frequency of the epileptic rhythm is between 2 and 3 Hz.
As we can see above in the Fourier spectra above, the amplitudes are high for low frequencies and tend to decrease with increasing frequency. Sometimes it is useful to see the high frequencies enhanced. This can be achieved with a logarithmic plot of the powers.

PYTHON

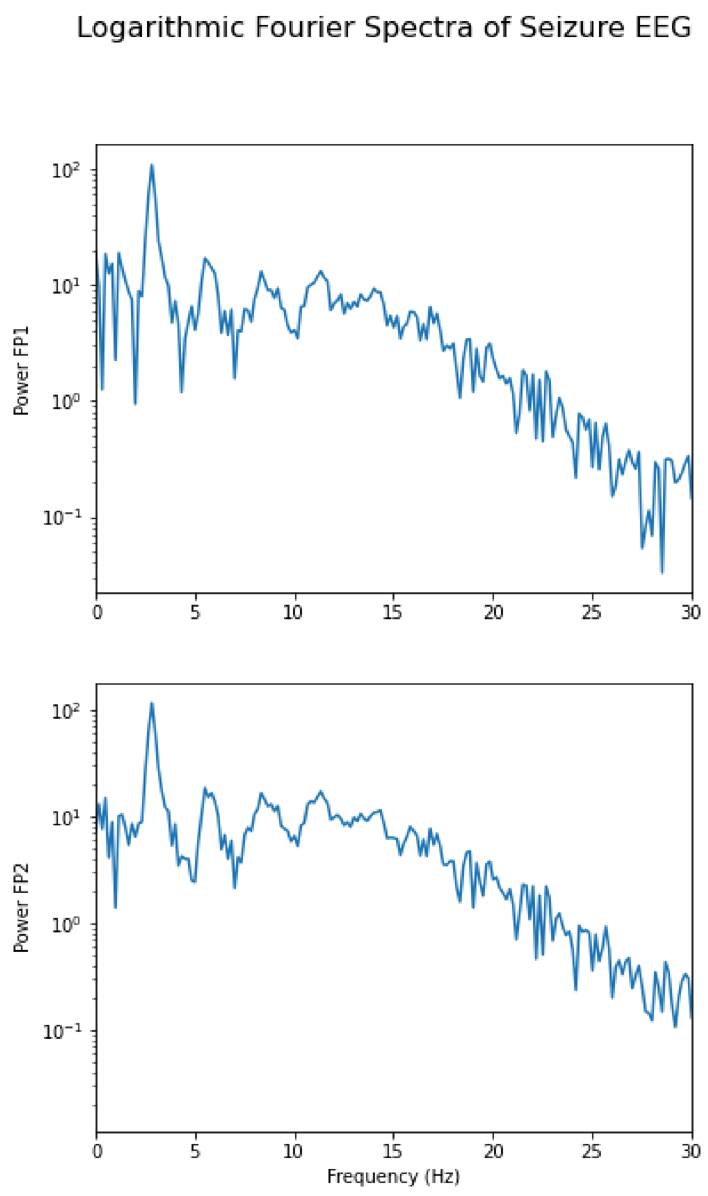
```python
fig, axes = subplots(figsize=(6, 6), ncols=1, nrows=no_win, sharex=False)

for index, ax in enumerate(axes.flat):

    axes[index].plot(freqs, amplitudes_back[:, index])
    axes[index].set_xlim(0, 30)
    axes[index].set(ylabel=f'Amplitude {names[index]}')
    axes[index].set_yscale('log')

axes[no_win-1].set(xlabel='Frequency (Hz)');
fig.suptitle('Logarithmic Fourier Spectra of Background EEG', fontsize=16);

show()
```



And for the seizure data:

PYTHON

```python
fig, axes = subplots(figsize=(6, 10), ncols=1, nrows=no_win, sharex=False)

for index, ax in enumerate(axes.flat):

    axes[index].plot(freqs, amplitudes_epil[:, index])
    axes[index].set_xlim(0, 30)
    axes[index].set(ylabel=f'Power {names[index]}')
    axes[index].set_yscale('log')

axes[no_win-1].set(xlabel='Frequency (Hz)');
fig.suptitle('Logarithmic Fourier Spectra of Seizure EEG', fontsize=16);

show()
```



In the spectrum of the absence data, it is now more obvious that there are further maxima at 6, 9, 12, and perhaps 15Hz. These are integer multiples or 'harmonics' of the basic frequency at around 3Hz, also referred to as the fundamental frequency.

A feature that can be used as a summary statistic, is to caclulate the **band power** for each channel. The band power can be obtained as the sum of all powers within a specified range of frequencies, also called the 'band'. The band power thus represents a single number.

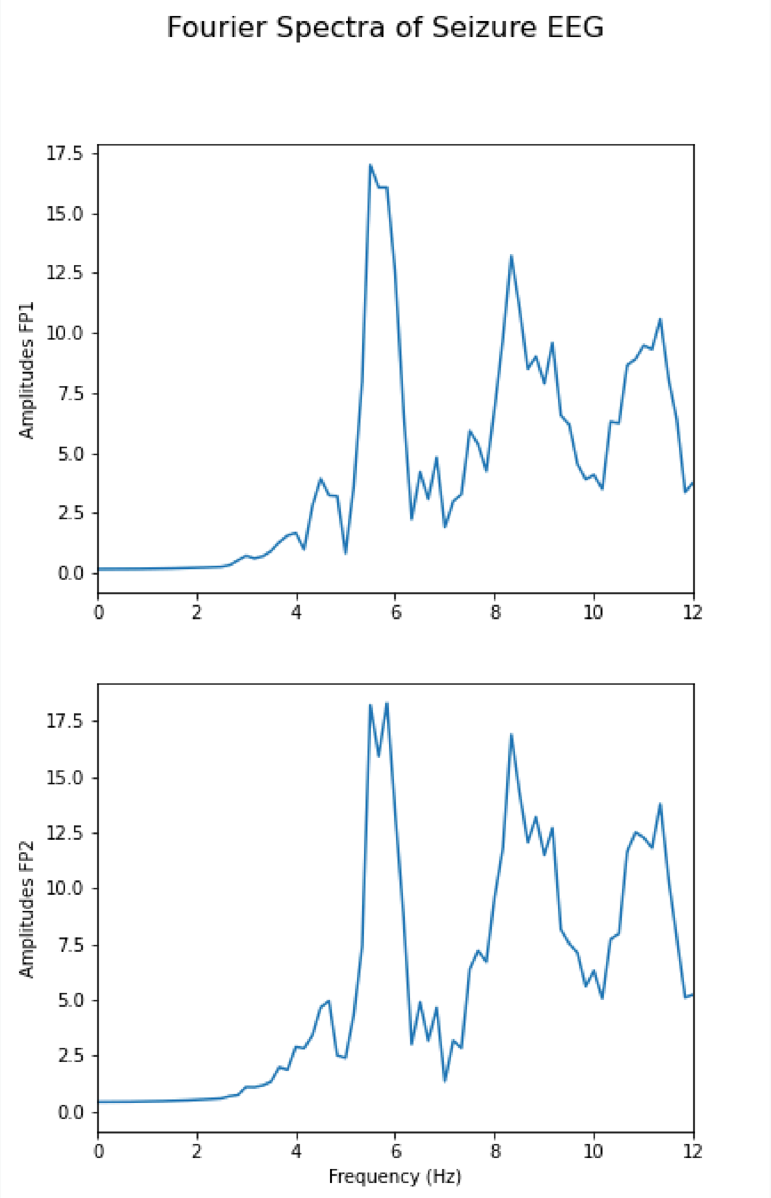
DIY2: FOURIER SPECTRA OF FILTERED DATA

Calculate and display the Fourier spectra of the first two channels filtered between 4 and 12 Hz for the absence seizure data. Can you find harmonics?

Solution

PYTHON

```python
data_epil_filt = data_filter(data_epil, sr, 4, 12)

data_epil_fft = fft(data_epil_filt, axis=0)

rows = data_epil.shape[0]

freqs = (sr/2)*linspace(0, 1, int(rows/2))

amplitudes_epil = (2.0 / rows) * abs(data_epil_fft[:rows//2, :no_win])

fig, axes = subplots(figsize=(6, 10), ncols=1, nrows=no_win, sharex=False)

for index, ax in enumerate(axes.flat):
    axes[index].plot(freqs, amplitudes_epil[:, index])
    axes[index].set_xlim(0, 12)
    axes[index].set(ylabel=f'Amplitudes {names[index]}')
axes[no_win-1].set(xlabel='Frequency (Hz)');

fig.suptitle('Fourier Spectra of Seizure EEG', fontsize=16);

show()
```



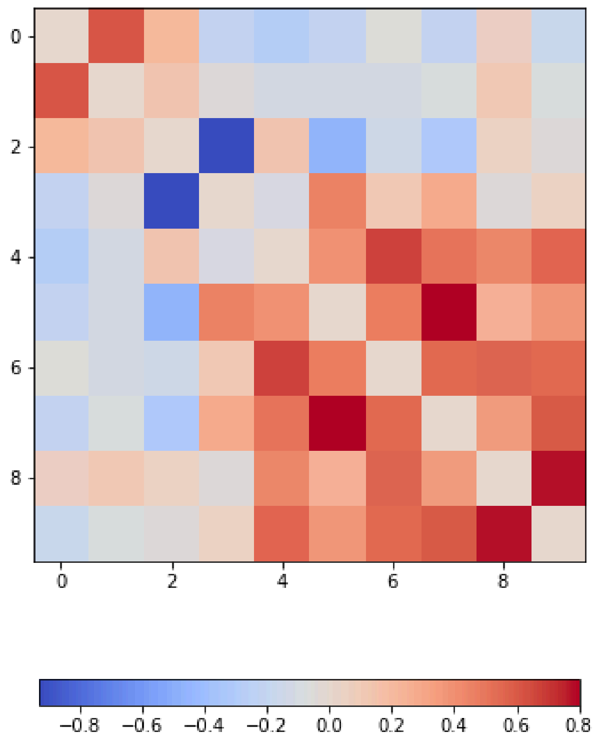
Cross-Correlation Matrix

As one example of a multivariate analysis of time series data, we can calculate the cross-correlation matrix.

Here it is done for the background:

PYTHON

```python
corr_matrix_back = corrcoef(data_back, rowvar=False)

fill_diagonal(corr_matrix_back, 0)

fig, ax = subplots(figsize = (8,8))

im = ax.imshow(corr_matrix_back, cmap='coolwarm');

fig.colorbar(im, orientation='horizontal', shrink=0.68);

show()
```



The diagonal is set to zero. This is done to improve the visual display. If left to one, the diagonal tends to dominate the visual impression even though it is trivial and nothing can be learned from it.

Looking at the non-diagonal elements, we find:

- two strongly correlated series (indices 5 and 7)
- two strongly anti-correlated series (indices 3 and 4)
- a block of pronounced correlations between series with indices 4 through 9)

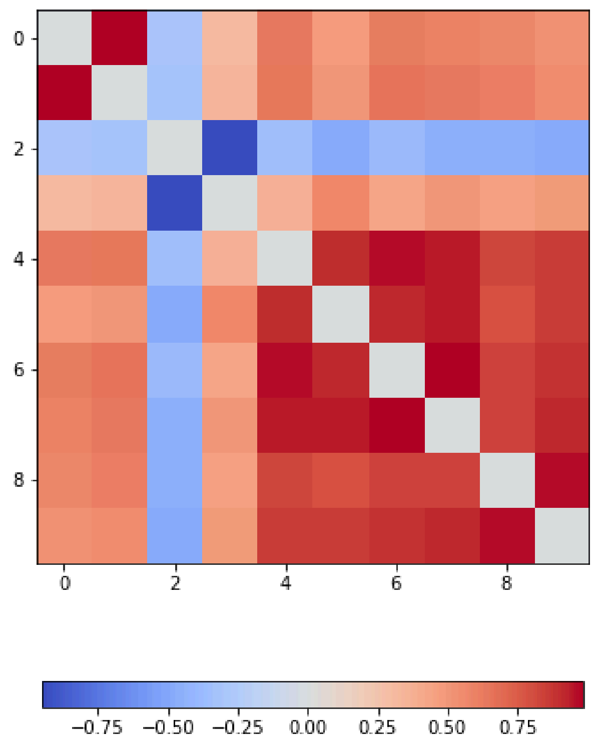
DIY3: DISPLAY THE CORRELATION MATRIX FOR THE SEIZURE DATA

Calculate the correlation matrix for the seizure data and compare the correlation pattern to the one from the background data.

Solution

PYTHON

```
corr_matrix_epil = corrcoef(data_epil, rowvar=False)

fill_diagonal(corr_matrix_epil, 0)

fig, ax = subplots(figsize = (8,8))

im = ax.imshow(corr_matrix_epil, cmap='coolwarm');

fig.colorbar(im, orientation='horizontal', shrink=0.68);

show()
```



We find - a number of pairs of strongly correlated series - two strongly anti-correlated series (in- dices 3 and 4) - a block of pronounced correlations between series with indices 4 through 9).

So interestingly, while the time series changes dramatically in shape, the correlation pattern still shows some qualitative resemblance.

All results shown so far, represent the recording of the segment of 6 seconds chosen at the beginning. The human brain produces time-dependent voltage changes 24 hours a day and as seeing only a few seconds is only a partial view. The next step to investigate is to show how the features found for one segment vary over time.

Exercises

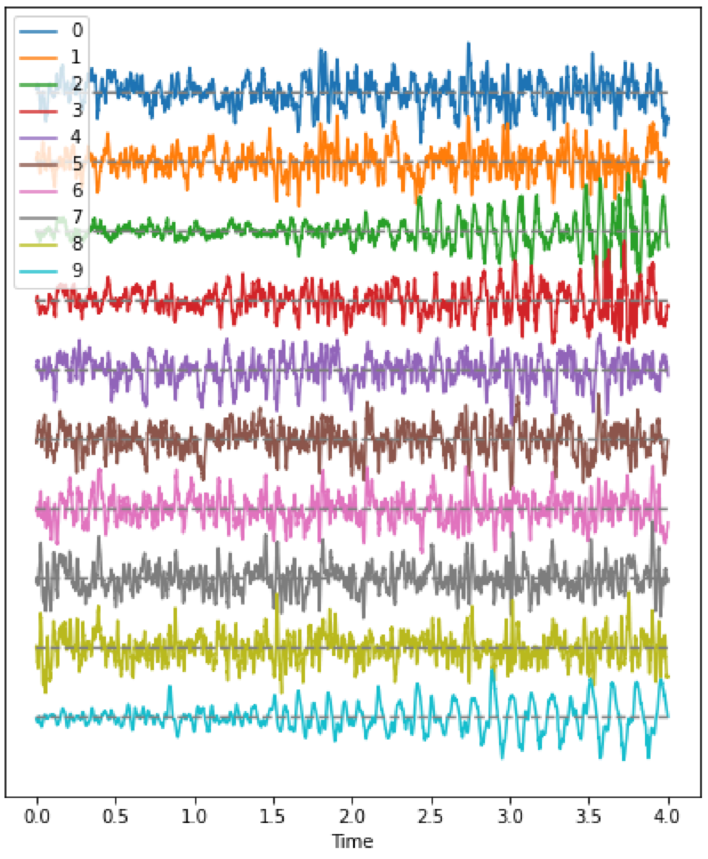
END OF CHAPTER EXERCISES

Pathological Human Brain Rhythms

Look at the image of brain activity from a child at the start of an epileptic seizure. It shows 4 seconds of evolution of the first 10 channels of a seizure rhythm at sampling rate sr=1024.

PYTHON

```
path = 'data/P1_Seizure1.csv'

data = read_csv(path, delimiter=r"\s+")

data_P1 = data.to_numpy()

sr = 1024

period = 4

channels = 10

plot_series(data_P1[:sr*period, :channels], sr);

show()
```



Using the code from this lesson, import the data from the file `P1_Seizure1.csv` and generate an overview of uni- and multivariate features in the following form:

1. Pick the first two seconds of the recording as background, the last two second as epileptic seizure rhythm. Use the first ten channels in both cases. Data should have the shape (2048, 10).

2. Filter the data to get rid of frequencies below 1 Hz and frequencies faster than 20 Hz.

3. Plot time series of both.

4. Fourier transform both filtered data sets and display the Fourier spectra of the first 4 channels. What are the strongest frequencies in the two sets?

5. Plot the correlation matrices of both data sets. Which channels show the strongest change in correlations?

Solution

PYTHON

```
from pandas import read_csv

from numpy import arange, zeros, linspace, sin, pi, c_, mean, var, array

from numpy import correlate, corrcoef, fill_diagonal, amin, amax, asarray

from numpy import around, triu_indices

from numpy.ma import masked_less, masked_greater

from scipy.fftpack import fft

from matplotlib.pyplot import subplots, yticks, legend, axis, figure, show
```

Q1

Extract data for first and last two seconds

PYTHON

```
sr = 1024

duration = 2 # seconds
data_n = data.to_numpy()

# dat_back = data.iloc[:duration*sr, :10]
dat_back = data_n[:duration*sr, :10]
dat_epil = data_n[data_n.shape[0] - duration*sr:, :10]

time = linspace(0, 1, duration*sr)

print(dat_back.shape, dat_epil.shape, time.shape)
```

OUTPUT

```
(2048, 10) (2048, 10) (2048,)
```

Q2 and Q3

Filter and display data

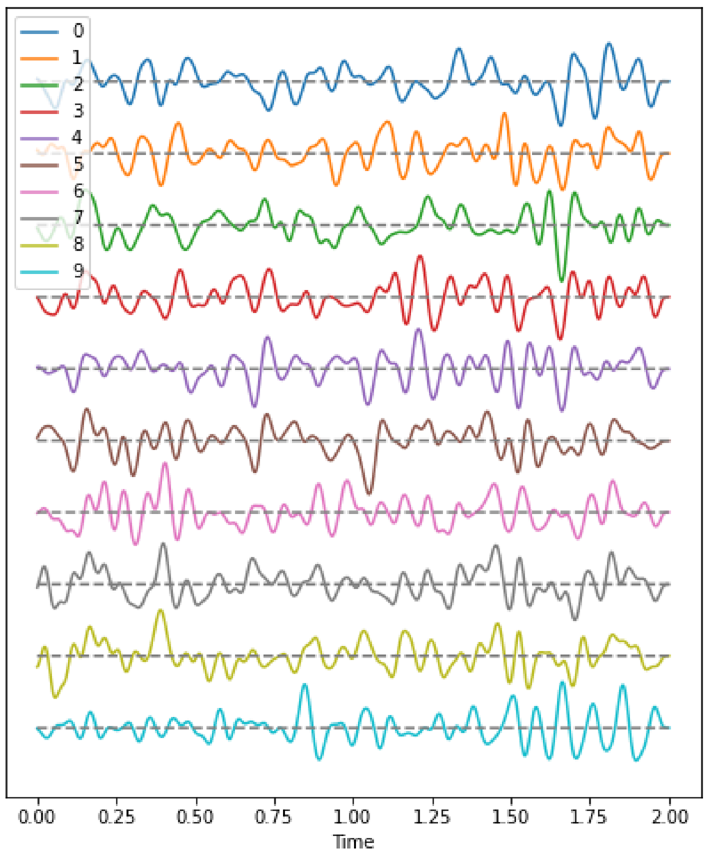
PYTHON

```python
def data_filter(data, sr, low, high):
    """
    Filtering of multiple time series.

    data: nxm numpy array. Rows are time points, columns are recordings
    sr: sampling rate, same time units as period

    low:  Low cut-off frequency (high-pass filter)
    high: High cut-off frequency (low-pass filter)

    return: filtered data
    """

    from scipy.signal import butter, sosfilt

    order = 5

    filter_settings = [low, high, order]

    sos = butter(order, (low,high), btype='bandpass', fs=sr, output='sos')

    data_filtered = zeros((data.shape[0], data.shape[1]))

    for index, column in enumerate(data.transpose()):
        forward = sosfilt(sos, column)
        backwards = sosfilt(sos, forward[-1::-1])
        data_filtered[:, index] = backwards[-1::-1]

    return data_filtered
```

PYTHON

```python
dat_back_filt = data_filter(dat_back, sr, 1, 20)

(fig, ax) = plot_series(dat_back_filt, sr)

fig.suptitle('First two seconds: Background EEG', fontsize=16);

show()
```

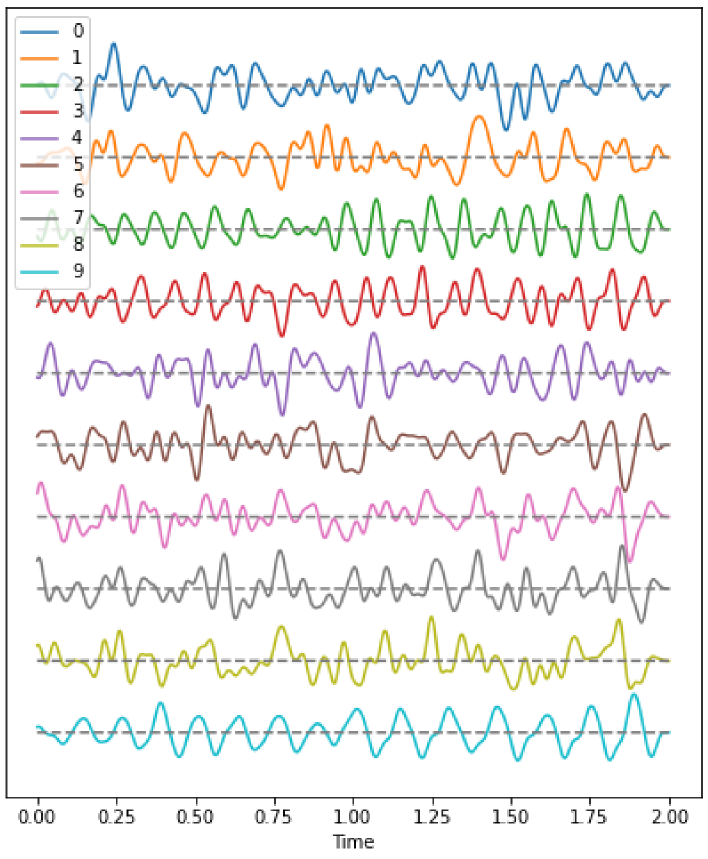
First two seconds: Background EEG



PYTHON

```python
dat_epil_filt = data_filter(dat_epil, sr, 1, 20)

(fig, ax) = plot_series(dat_epil_filt, sr)

fig.suptitle('Last 2 seconds: Seizure EEG', fontsize=16);

show()
```

Last 2 seconds: Seizure EEG



Q4

Fourier Transform

PYTHON

```python
rows = dat_back.shape[0]

dat_back_fft = fft(dat_back_filt, axis=0)

powers_back2 = (2.0 / rows) * abs(dat_back_fft[:rows//2, :])

dat_epil_fft = fft(dat_epil_filt, axis=0)

powers_epil2 = (2.0 / rows) * abs(dat_epil_fft[:rows//2, :])
```

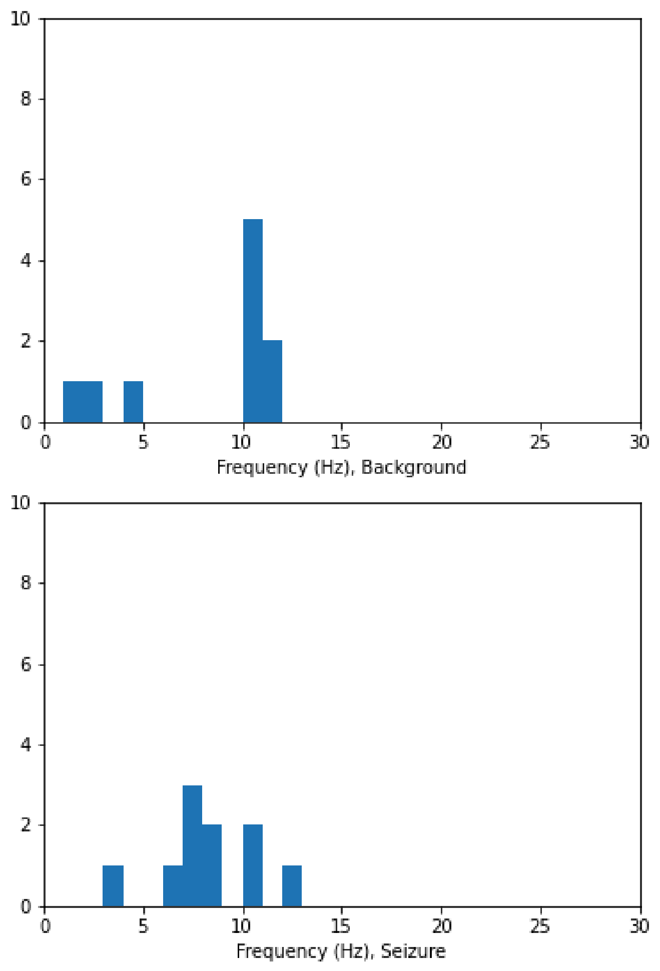
You can see the frequencies with largest power visually from a display of the Fourier spectrum. Here is an example code how to extract those values for each channel using Numpy functions.

PYTHON

```python
# Get the maxima of powers for each channel

powermax_back2 = amax(powers_back2, axis=0)
powermax_epil2 = amax(powers_epil2, axis=0)


# Get the frequency index of that maximum

powermax_back2_index = zeros(powers_back2.shape[1])

for ind, pow in enumerate(list(powers_back2.transpose())):
    pow_ind = list(pow).index(powermax_back2[ind])
    powermax_back2_index[ind] = pow_ind

powermax_epil2_index = zeros(powers_epil2.shape[1])

for ind, pow in enumerate(list(powers_epil2.transpose())):
    pow_ind = list(pow).index(powermax_epil2[ind])
    powermax_epil2_index[ind] = pow_ind



powermax_back2_freq = asarray(powermax_back2_index)*(sr / 2 / powers_back2.shape[0])
powermax_epil2_freq = asarray(powermax_epil2_index)*(sr / 2 / powers_epil2.shape[0])

print('Frequencies of max power in background (Hz): ', '\n', around(powermax_back2_freq, decimals=1))
print('')
print('Frequencies of max power in seizure (Hz): ', '\n', around(powermax_epil2_freq, decimals=1))

#print(powermax_back2_freq, '\n',  powermax_epil2_freq)

fig, ax = subplots(nrows=2, figsize=(6,9))

binwidth = 1

(counts1, bins1, bars1) = ax[0].hist(powermax_back2_freq, bins=arange(0, 50 + binwidth, binwidth))
ax[0].set_xlim(0, 30)
ax[0].set_ylim(0, 10)
ax[0].set(xlabel='Frequency (Hz), Background')

(counts2, bins2, bars2) = ax[1].hist(powermax_epil2_freq, bins=arange(0, 50 + binwidth, binwidth))
ax[1].set_xlim(0, 30)
ax[1].set_ylim(0, 10)
ax[1].set(xlabel='Frequency (Hz), Seizure');

show()
```

OUTPUT

```
Frequencies of max power in background (Hz):
 [ 2.5 11.   4.5 10.5 10.5  1.5 10.5 10.5 10.5 11. ]

Frequencies of max power in seizure (Hz):
 [ 6.   7.5 10.  10.  12.   8.   3.5  7.   7.   8. ]
(0.0, 30.0)
(0.0, 10.0)
(0.0, 30.0)
(0.0, 10.0)
```



Fourier spectrum with max frequency

PYTHON

```python
print('Maximum count: background: ', amax(counts1))

print('Maximum count: seizure:    ', amax(counts2))

print('Frequency with maximum count: background: ', '10-11 Hz')

print('Frequency with maximum count: seizure:    ', ' 7- 8 Hz')
```

OUTPUT

```
Maximum count: background:  5.0
Maximum count: seizure:     3.0
Frequency with maximum count: background:  10-11 Hz
Frequency with maximum count: seizure:      7- 8 Hz
```

PYTHON

```python
freqs = (sr/2)*linspace(0, 1, int(rows/2))

fig, axes = subplots(figsize=(6, 14), ncols=1, nrows=4, sharex=False)

axes[0].plot(freqs, powers_back2[:, 3])
axes[0].set_xlim(0, 20)
```

PYTHON

```python
axes[0].set_ylim(0, 12)
```

PYTHON

```python
axes[0].set(ylabel=f'Power channel 3')
axes[0].set(xlabel='Frequency (Hz)');

axes[1].plot(time, dat_back_filt[:, 3], c='r')
axes[1].set(xlabel='Time (s)');
axes[1].set_ylim(-50, 60)
```

PYTHON

```python
axes[2].plot(freqs, powers_epil2[:, 3])
axes[2].set_xlim(0, 20)
```

PYTHON

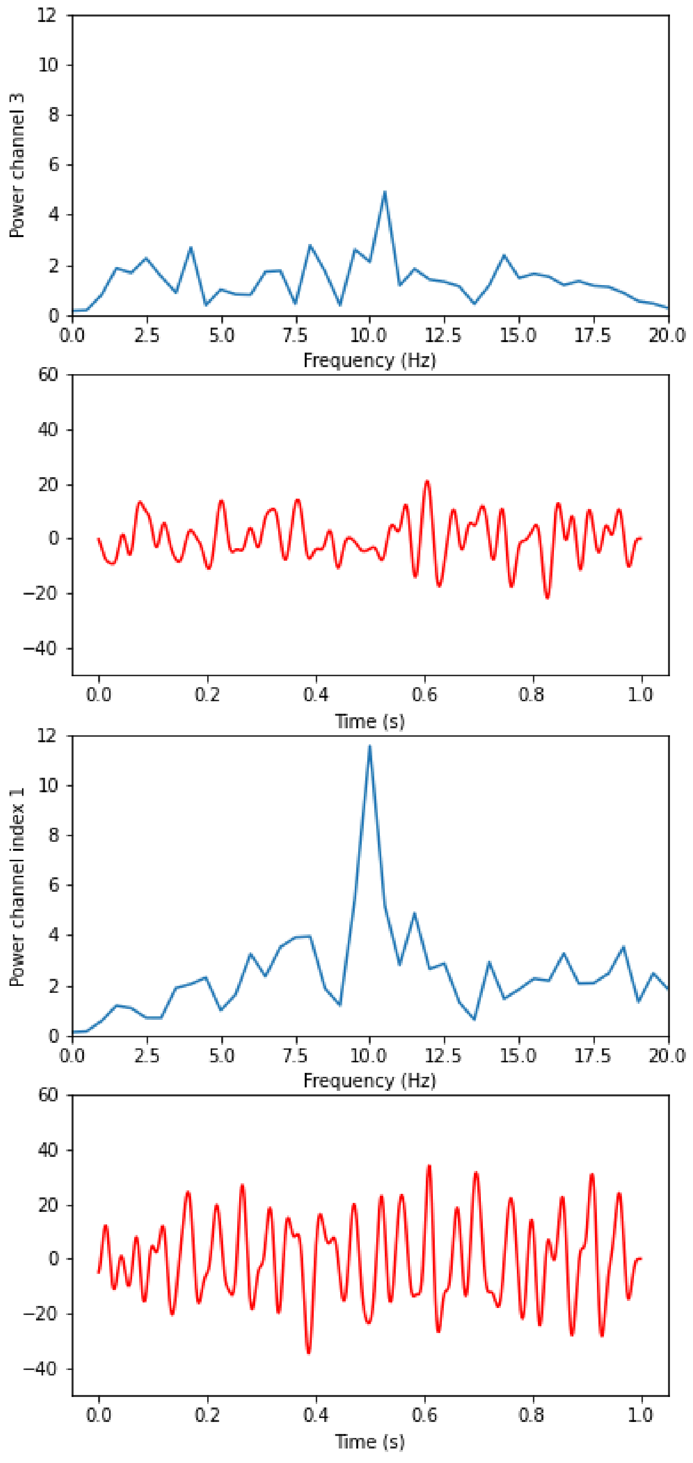
```python
axes[2].set_ylim(0, 12)
```

PYTHON

```python
axes[2].set(ylabel='Power channel index 1')
axes[2].set(xlabel='Frequency (Hz)');

axes[3].plot(time, dat_epil_filt[:, 3], c='r')
axes[3].set(xlabel='Time (s)');
axes[3].set_ylim(-50, 60)
```

PYTHON

```python
show()
```



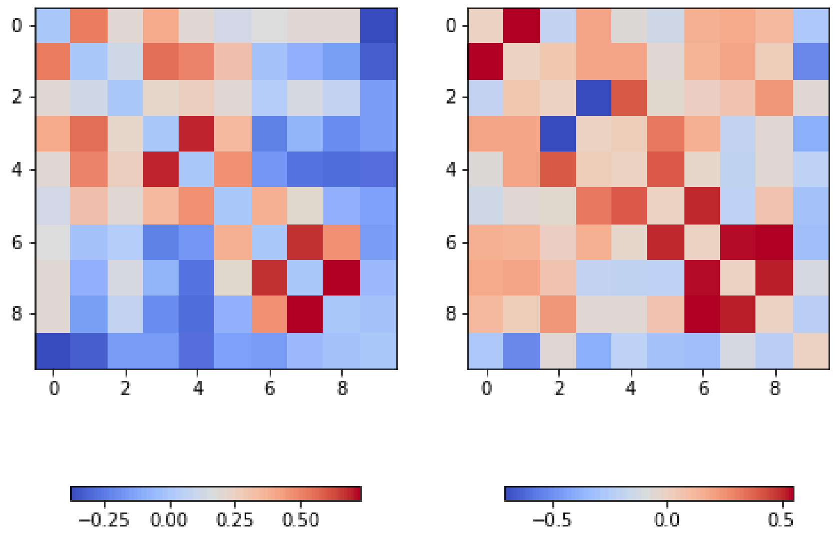
Q5 ### Correlation Matrix

PYTHON

```
corr_matrix_back2 = corrcoef(dat_back_filt, rowvar=False)

fill_diagonal(corr_matrix_back2, 0)

corr_matrix_epil2 = corrcoef(dat_epil_filt, rowvar=False)

fill_diagonal(corr_matrix_epil2, 0)

fig, ax = subplots(figsize = (8,8), ncols=2)

im1 = ax[0].imshow(corr_matrix_back2, cmap='coolwarm');
fig.colorbar(im1, ax=ax[0], orientation='horizontal', shrink=0.8)
```

PYTHON

```
im2 = ax[1].imshow(corr_matrix_epil2, cmap='coolwarm');
fig.colorbar(im2, ax=ax[1], orientation='horizontal', shrink=0.8);

show()
```



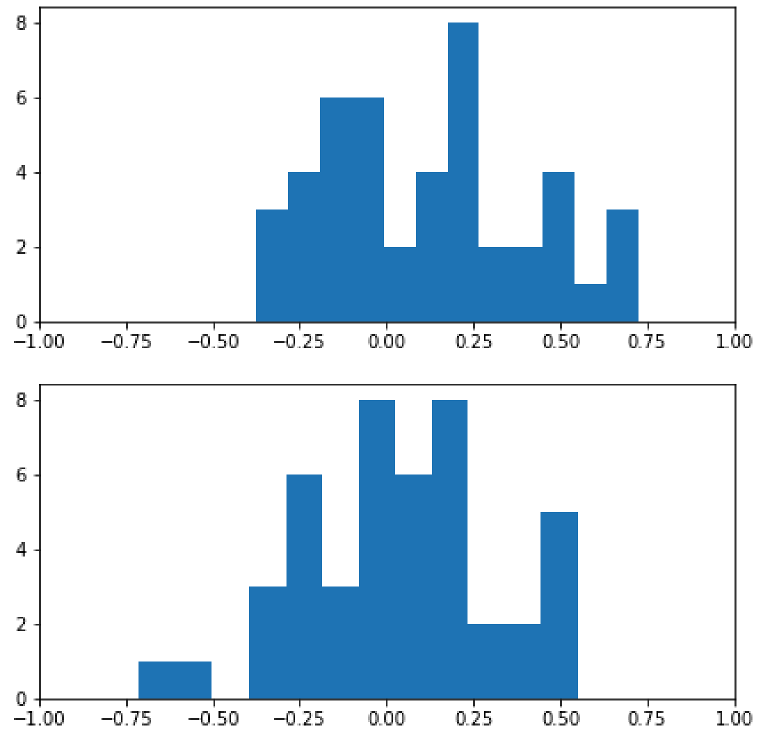
Hist of Correlation Coefficients

PYTHON

```
channels = dat_back.shape[1]

corr_coeffs_back2 = corr_matrix_back2[triu_indices(channels, k=1)]
corr_coeffs_epil2 = corr_matrix_epil2[triu_indices(channels, k=1)]

fig, ax = subplots(nrows=2)

ax[0].hist(corr_coeffs_back2, bins = 12);
ax[0].set_xlim(-1, 1)
```

PYTHON

```
ax[1].hist(corr_coeffs_epil2, bins = 12);
ax[1].set_xlim(-1, 1);

show()
```



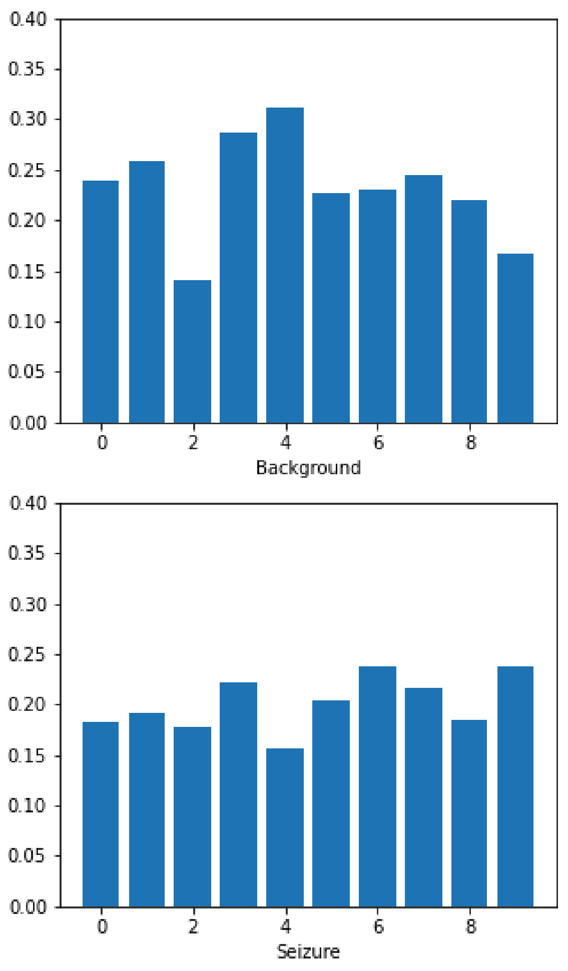
Channel Correlations

PYTHON

```python
corr_coeffs_back2_mean = mean(abs(corr_matrix_back2), axis=0)
corr_coeffs_epil2_mean = mean(abs(corr_matrix_epil2), axis=0)

fig, ax = subplots(nrows=2, figsize=(5,9))

bins = arange(corr_coeffs_back2_mean.shape[0])

ax[0].bar(bins, corr_coeffs_back2_mean);
ax[0].set_xlabel('Background')
ax[0].set_ylim(0, 0.4)
```

PYTHON

```python
ax[1].bar(bins, corr_coeffs_epil2_mean);
ax[1].set_xlabel('Seizure')
ax[1].set_ylim(0, 0.4);

show()
```



PYTHON

```python
corr_back2_mean = mean(corr_coeffs_back2_mean)
corr_epil2_mean = mean(corr_coeffs_epil2_mean)

print('Average correlation background: ', around(corr_back2_mean, decimals=2))
print('Average correlation seizure:    ', around(corr_epil2_mean, decimals=2))
```

OUTPUT

```
Average correlation background:  0.23
Average correlation seizure:     0.2
```

Channels with strongest correlations during the seizure

PYTHON

```
threshold = 0.2

corr_coeffs_epil2_large = corr_coeffs_epil2_mean > threshold

corr_coeffs_epil2_large
```

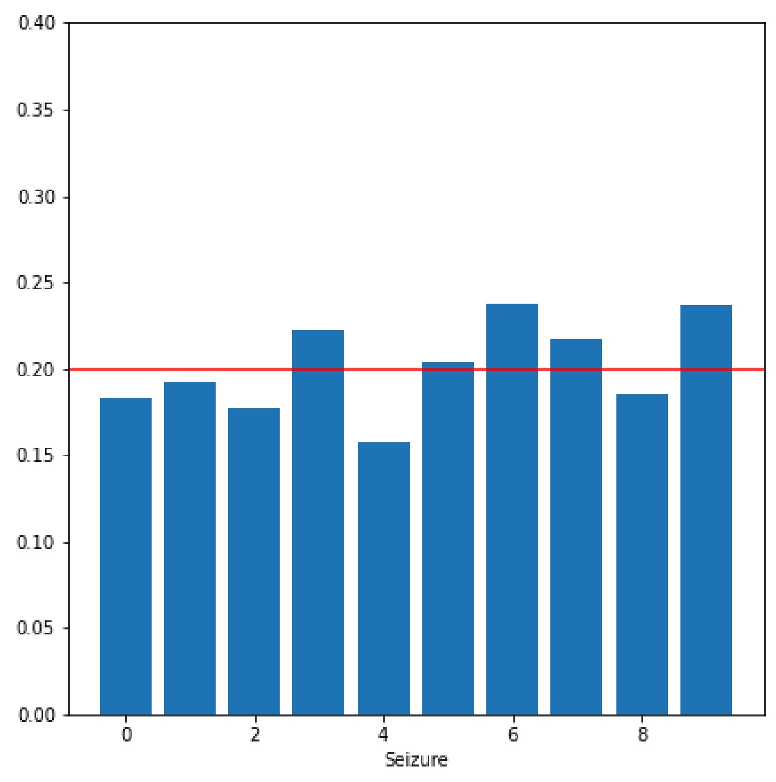
OUTPUT

```
array([False, False, False,  True, False,  True,  True,  True, False,
        True])
```

A horizontal line can be used to indicate the threshold:

PYTHON

```
fig, ax = subplots()

bins = arange(corr_coeffs_back2_mean.shape[0])

ax.bar(bins, corr_coeffs_epil2_mean);
ax.set_xlabel('Seizure')
ax.set_ylim(0, 0.4);
ax.axhline(y=threshold, c='r');

show()
```



To find the indices of the channels that are larger than the threshold (i.e. those displaying True), we can use Numpy function `nonzero`:

PYTHON

```python
from numpy import nonzero

nonzero(corr_coeffs_epil2_large)
```

OUTPUT

```
(array([3, 5, 6, 7, 9]),)
```

KEY POINTS

- `plot_series` is a Python function created to display multiple timeseries plots.
- Data filtering is applied to take out specific and relevant components.
- The Fourier spectrum decomposes the time series into a sum of sine waves.
- Cross-correlation matrix is used for multivariate analysis.