practical_exercise_6, Methods 3, 2021, autumn semester

[FILL IN YOUR NAME]

[FILL IN THE DATE]

Exercises and objectives

1) Get acquainted with *Python*, and learn some of the differences between it and *R*

2) Estimate bias and variance based on a true underlying function

REMEMBER: In your report, make sure to include code that can reproduce the answers requested in the exercises below (**MAKE A KNITTED VERSION**)
REMEMBER: All exercises should be done in *Python*

EXERCISE 1 Get acquainted with *Python* learn some of the differences between it and *R*

To make sure that *Python* runs within *R Markdown*, make sure you have the *reticulate* package installed `install.packages('reticulate')`
Also create a text file that is called *.Renviron* (remember the dot) placed in the folder where your *RProj* file is. It should have a single line: `RETICULATE_PYTHON=PATH` where `PATH` is the path to your *methods3* conda environment. Use the commands below to find the paths:

```
library(reticulate)
print(conda_list())
```

```
##          name                                          python
## 1   miniconda3                   /home/lau/miniconda3/bin/python
## 2     methods3     /home/lau/miniconda3/envs/methods3/bin/python
## 3          mne          /home/lau/miniconda3/envs/mne/bin/python
## 4     mne_0.17     /home/lau/miniconda3/envs/mne_0.17/bin/python
## 5 mne_func_sig  /home/lau/miniconda3/envs/mne_func_sig/bin/python
## 6       mnedev       /home/lau/miniconda3/envs/mnedev/bin/python
## 7     psychopy     /home/lau/miniconda3/envs/psychopy/bin/python
## 8    fslpython               /usr/local/fsl/fslpython/bin/python
## 9    fslpython /usr/local/fsl/fslpython/envs/fslpython/bin/python
```

To update your environment based on the updated `methods3_environment.yml` file, go to your *week_06* folder and run the following from a *bash* interpreter (e.g. *terminal*):

```
conda env create --force -f methods3_environment.yml
```

The `--force` flag allows for overwriting

Good to know about *Python* (in no particular order)

```
## assignment is done and only done with "=" (no arrows)
a = 2
# a <- 2 # results in a syntax error
## already assigned variables can be reassigned with basic arithmetic operations
a += 2
print(a)
```

```
## 4
```

```
a -= 1
print(a)
```

```
## 3
```

```
a *= 4
print(a)
```

```
## 12
```

```
a //= 2 # integer division
print(a)
```

```
## 6
```

```
a /= 2 # float  (numeric from R) division
print(a)
```

```
## 3.0
```

```
a **= 3 # exponentiation
print(a)
```

```
## 27.0
```

```
a_list = [1, 2] # initiate a list (the square brackets) with the integers 1 and 2
b = a_list ## b now points to a_list, not to a new list with the integers 1 and 2

a_list.append(3) # add a new value to the end of the list
print(a_list)
```

```
## [1, 2, 3]
```

```
print(b) # make sure you understand this
```

```
## [1, 2, 3]
```

```
print(a_list[0]) # zero-indexing
```

```
## 1
```

```
print(a_list[1])
```

```
## 2
```

```
new_list = [0, 1, 2, 3, 4, 5]
print(new_list[0:3])  # slicing
```

```
## [0, 1, 2]
```

```
for index in range(0, 5): # indentation (use tabulation) controls scope of control variables
    #(no brackets necessary),
    if index == 0: # remember the colon
        value = 0
    else:
        value += index
    print(value)
```

```
## 0
## 1
## 3
## 6
## 10
```

```
this_is_true = True # logical values
this_is_false = False

# define functions using def
def fix_my_p_value(is_it_supposed_to_be_significant):
    if is_it_supposed_to_be_significant:
        p = 0.01
    else:
        p = 0.35
    return(p)

print(fix_my_p_value(True))
```

remember indentation

```
## 0.01
```

```
import numpy # methods of numpy can now be accessed as below
# importing packages (similar to library)

print(numpy.arange(1, 10)) # see the dot
```

need to specify the package
Just like dplyr::ggplot

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

```
print(numpy.abs(-3))
```

```
## 3
```

```
import numpy as np # you can import them with another name than its default
print(np.cos(np.pi))
```

```
## -1.0
```

```
from numpy import pi, arange # or you can import specific methods
print(arange(1, 7))
```

```
## [1 2 3 4 5 6]
```

```
print(pi)
```

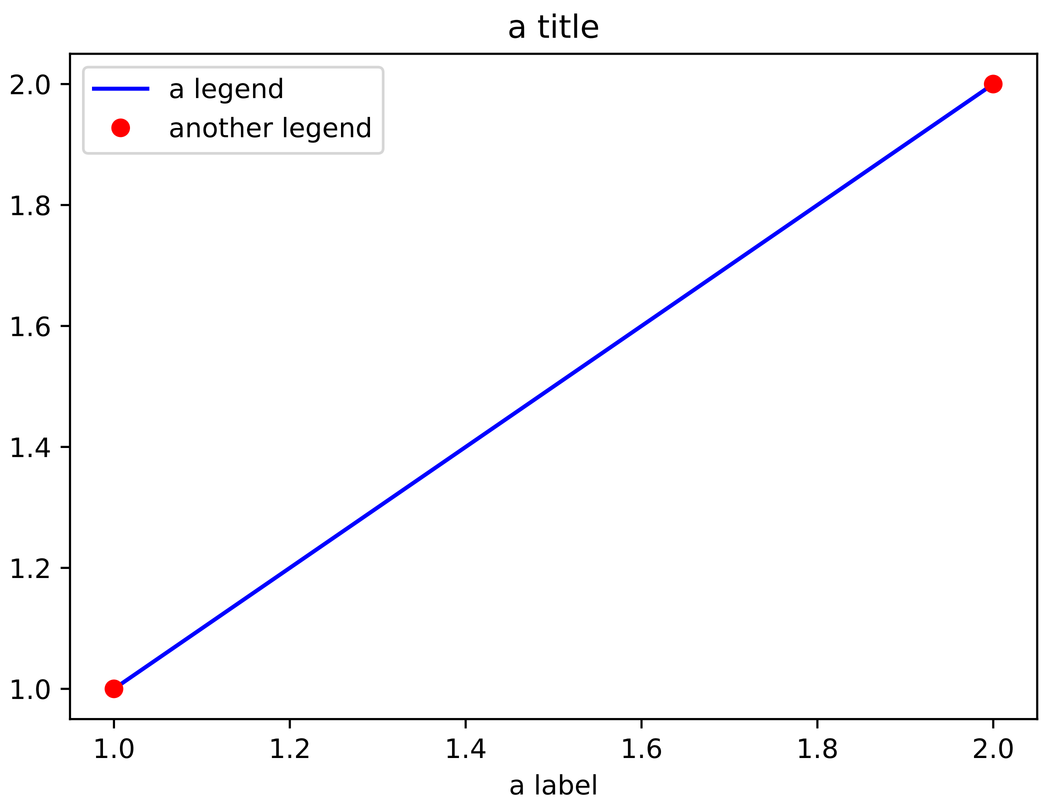
```
## 3.141592653589793
```

```
matrix = np.ones(shape=(5, 5)) # create a matrix of ones
identity = np.identity(5) # create an identity matrix (5x5)
identity[:, 2] = 5 # exchange everything in the second column with 5's
```

Text

```
## no dots in names - dots indicate applying a method like the dollar sign $ in R

import matplotlib.pyplot as plt
plt.figure() # create new figure
plt.plot([1, 2], [1, 2], 'b-') # plot a blue line
# plt.show() # show figure
plt.plot([2, 1], [2, 1], 'ro') # scatter plot (red)
# plt.show()
plt.xlabel('a label')
plt.title('a title')
plt.legend(['a legend', 'another legend'])
# plt.show()
```



1) Do a linear regression based on x, X and y below (y as the dependent variable) (Exercise 1.1)
   i. find $\hat{\beta}$ and $\hat{y}$ (@ is matrix multiplication)
   ii. plot a scatter plot of x, y and add a line based on $\hat{y}$ (use `plt.plot` after running `import matplotlib.pyplot as plt`)

2) Create a model matrix, X that estimates, $\hat{\beta}$ the means of the three sets of observations below, y_1, y_2, y_3 (Exercise 1.2)
   i. find $\hat{\beta}$ based on this X

   ii. Then create an X where the resulting $\hat{\beta}$ indicates: 1) the difference between the mean of y_1 and the mean of y_2; 2) the mean of y_2; 3) the difference between the mean of y_3 and the mean of y_1

3) Finally, find the F-value for this model (from exercise 1.2.ii) and its degrees of freedom. What is the p-value associated with it? (You can import the inverse of the cumulative probability density function `ppf` for F using `from scipy.stats import f` and then run `1 - f.ppf`)
    i. plot the probability density function `f.pdf` for the correct F-distribution and highlight the F-value that you found

    ii. how great a percentage of the area of the curve is to right of the highlighted point

```
# Exercise 1.1
import numpy as np
np.random.seed(7) # for reproducibility

x = np.arange(10)
y = 2 * x
y = y.astype(float)
n_samples = len(y)
y += np.random.normal(loc=0, scale=1, size=n_samples)

X = np.zeros(shape=(n_samples, 2))
X[:, 0] = x ** 0
X[:, 1] = x ** 1
```

```
# Exercise 1.2
y1 = np.array([3, 2, 7, 6, 9])
y2 = np.array([10, 4, 2, 1, -3])
y3 = np.array([15, -2, 0, 0, 3])
y = np.concatenate((y1, y2, y3))
```

EXERCISE 2 - Estimate bias and variance based on a true underlying function

We can express regression as $y = f(x) + \epsilon$ with $E[\epsilon] = 0$ and $var(\epsilon) = \sigma^2$ (E means expected value)

For a given point: x_0, we can decompose the expected prediction error , $E[(y_0 - \hat{f}(x_0))^2]$ into three parts - **bias**, **variance** and **irreducible error** (the first two together are the **reducible error**):

The expected prediction error is, which we also call the **Mean Squared Error**:
$E[(y_0 - \hat{f}(x_0))^2] = bias(\hat{f}(x_0))^2 + var(\hat{f}(x_0)) + \sigma^2$

where **bias** is;

$bias(\hat{f}(x_0)) = E[\hat{f}(x_0)] - f(x_0)$

1) Create a function, $f(x)$ that squares its input. This is our **true** function
    i. generate data, y_{true}, based on an input range of [0, 6] with a spacing of 0.1. Call this x
    ii. add normally distributed noise to y_{true} with $\sigma = 5$ (set a seed to 7 `np.random.seed(7)`) and call it y_{noise}
    iii. plot the true function and the generated points

2) Fit a linear regression using `LinearRegression` from `sklearn.linear_model` based on y_{noise} and x (see code below)
    i. plot the fitted line (see the `.intercept_` and `.coef_` attributes of the `regressor` object) on top of the plot (from 2.1.iii)
    ii. now run the code associated with Exercise 2.2.ii - what does X_quadratic amount to?
    iii. do a quadratic and a fifth order fit as well and plot them (on top of the plot from 2.2.i)

3) Simulate 100 samples, each with sample size `len(x)` with $\sigma = 5$ normally distributed noise added on top of the true function
   i. do linear, quadratic and fifth-order fits for each of the 100 samples
      ii create a **new** figure, `plt.figure`, and plot the linear and the quadratic fits (colour them appropriately); highlight the true value for $x_0 = 3$. From the graphics alone, judge which fit has the highest bias and which has the highest variance

   ii. create a **new** figure, `plt.figure`, and plot the quadratic and the fifth-order fits (colour them appropriately); highlight the true value for $x_0 = 3$. From the graphics alone, judge which fit has the highest bias and which has the highest variance

   iii. estimate the **bias** and **variance** at x_0 for the linear, the quadratic and the fifth-order fits (the expected value $E[\hat{f}(x_0)]$ is found by taking the mean of all the simulated, $\hat{f}(x_0)$, differences)

   iv. show how the **squared bias** and the **variance** are related to the complexity of the fitted models

   v. simulate **epsilon**: `epsilon = np.random.normal(scale=5, size=100)`. Based on your simulated values of **bias, variance and epsilon**, what is the **Mean Squared Error** for each of the three fits? Which fit is better according to this measure?

```
# Exercise 2.2
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit() ## what goes in here?
```

```
# Exercise 2.2.ii
from sklearn.preprocessing import PolynomialFeatures
quadratic = PolynomialFeatures(degree=2)
X_quadratic = quadratic.fit_transform(x.reshape(-1, 1))
regressor = LinearRegression()
regressor.fit() # what goes in here?
```