IN3050/IN4050 Mandatory Assignment 3: Unsupervised Learning

**Name: The Dat Le

**Username: Thedl

Rules

Before you begin the exercise, review the rules at this website: https://www.uio.no/english/studies/examinations/compulsory-activities/mn-ifi-mandatory.html , in particular the paragraph on cooperation. This is an individual assignment. You are not allowed to deliver together or copy/share source-code/answers with others. Read also the "Routines for handling suspicion of cheating and attempted cheating at the University of Oslo": https://www.uio.no/english/studies/examinations/cheating/index.html We do not entirely prohibit the use of generative language models ("smart assistants" like ChatGPT, Llama, Claude or Copilot), but you must clearly acknowledge this at all times, following the UiO guidelines: https://www.uio.no/english/studies/resources/ai_student.html Note also that you must fully understand *all* the parts of you submissions, even if you got some help from a generative model. This will be tested during your peer review sessions (https://www.uio.no/studier/emner/matnat/ifi/IN3050/v25/Peer%20review/). By submitting this assignment, you confirm that you are familiar with the rules and the consequences of breaking them.

Delivery

Deadline: Tuesday, April 22, 2025, 23:59 CEST

Your submission should be delivered in Devilry. You may make several submissions in Devilry before the deadline, but only the last delivery will be read, so make sure to include all files in the last delivery. It is recommended to upload preliminary versions hours (or days) before the final deadline.

What to deliver?

We recommend you to solve all exercises in this Jupyter notebook (alternative 1), but it is also possible to work with a regular Python script (alternative 2).

Alternative 1: If you choose Jupyter, you should deliver the notebook. You should answer all questions and explain what you are doing in Markdown. Still, the code should be properly commented. The notebook should contain results of your runs. In addition, you also have to submit a PDF of your solution which shows the results of the runs.

Alternative 2: If you prefer not to use notebooks, you should deliver the code as Python scripts together with your run results and a PDF report where you answer all the questions and explain your work.

Here is a list of *absolutely necessary* (but not sufficient) conditions to get the assignment marked as passed:

- You must deliver your code (Python script or Jupyter notebook) you used to solve the assignment.
- The code used for making the output and plots must be included in the assignment.
- You must include example runs that clearly shows how to run all implemented functions and methods.
- All the code (in notebook cells or python main-blocks) must be runnable. If you have unfinished code that crashes, please comment it out and document what you think causes it to crash.
- You must also deliver a PDF of the code, outputs, comments and plots as explained above.

Make sure to include your name and username in the submitted files. Deliver one single compressed folder (.zip, .tgz or .tar.gz) which contains your complete solution.

Important: if you weren't able to finish the assignment, use the PDF report/Markdown to elaborate on what you've tried and what problems you encountered. Students who have made an effort and attempted all parts of the assignment will get a second chance even if they fail initially. This exercise will be graded PASS/FAIL.

Goals of the assignment

This assignment has two parts:

1. You will go through the basic theory of **Principal Component Analysis (PCA)** and implement PCA from scratch to compress and visualize data.
2. You will run **K-means clustering** using the `scikit-learn` toolkit and use PCA to visualize the results. You will also evaluate the output of K-means using a multi-class logistic regression classifier.

IN4050 students will have to do one extra part about tuning PCA to balance compression with information lost.

Tools

You may freely use code from the weekly exercises and the published solutions. In the first part about PCA you may **NOT** use ML libraries like `scikit-learn`. In the K-means part and beyond we encourage the use of `scikit-learn` to iterate quickly on the problems.

We will use the *numpy* library for performing matrix computations and the *pyplot* library for plotting data, as well as *scikit-learn* for K-means clustering. This assignment also comes with a module called *data_assignment3* that you will use to import different (synthetic and real) datasets. Let's start by making sure that everything is installed.

In [96]:
```
%pip install numpy matplotlib scikit-learn

import data_assignment3
```

```
Defaulting to user installation because normal site-packages is not writea
ble
Requirement already satisfied: numpy in /Users/cc/Library/Python/3.10/lib/
python/site-packages (1.26.4)
Requirement already satisfied: matplotlib in /Users/cc/Library/Python/3.1
0/lib/python/site-packages (3.7.2)
Requirement already satisfied: scikit-learn in /Users/cc/Library/Python/3.
10/lib/python/site-packages (1.6.1)
Requirement already satisfied: contourpy>=1.0.1 in /Users/cc/Library/Pytho
n/3.10/lib/python/site-packages (from matplotlib) (1.1.0)
Requirement already satisfied: cycler>=0.10 in /Users/cc/Library/Python/3.
10/lib/python/site-packages (from matplotlib) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in /Users/cc/Library/Pyth
on/3.10/lib/python/site-packages (from matplotlib) (4.42.1)
Requirement already satisfied: kiwisolver>=1.0.1 in /Users/cc/Library/Pyth
on/3.10/lib/python/site-packages (from matplotlib) (1.4.5)
Requirement already satisfied: packaging>=20.0 in /Users/cc/Library/Pytho
n/3.10/lib/python/site-packages (from matplotlib) (23.1)
Requirement already satisfied: pillow>=6.2.0 in /Users/cc/Library/Python/
3.10/lib/python/site-packages (from matplotlib) (10.0.0)
Requirement already satisfied: pyparsing<3.1,>=2.3.1 in /Users/cc/Library/
Python/3.10/lib/python/site-packages (from matplotlib) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in /Users/cc/Library/P
ython/3.10/lib/python/site-packages (from matplotlib) (2.8.2)
Requirement already satisfied: scipy>=1.6.0 in /Users/cc/Library/Python/3.
10/lib/python/site-packages (from scikit-learn) (1.15.2)
Requirement already satisfied: joblib>=1.2.0 in /Users/cc/Library/Python/
3.10/lib/python/site-packages (from scikit-learn) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /Users/cc/Library/P
ython/3.10/lib/python/site-packages (from scikit-learn) (3.5.0)
Requirement already satisfied: six>=1.5 in /Users/cc/Library/Python/3.10/l
ib/python/site-packages (from python-dateutil>=2.7->matplotlib) (1.16.0)

[notice] A new release of pip is available: 23.2.1 -> 25.0.1
[notice] To update, run: pip3 install --upgrade pip
Note: you may need to restart the kernel to use updated packages.
```

Part 1: Principal Component Analysis (PCA)

In this section, you will work with the PCA algorithm in order to understand its definition and explore its uses.

Part 1.1: Implementation

Here we implement the basic steps of PCA and we assemble them. We will only need functions from *numpy* for this part.

In [97]:
```
import numpy as np
```

Centering the Data

Implement a function with the following signature to center the data. Remember that every *feature* should be centered.

In [98]:
```
def center_data(A):
    # INPUT:
    # A    [NxM] numpy data matrix (N samples, M features)
    #
    # OUTPUT:
    # X    [NxM] numpy centered data matrix (N samples, M features)
    mean = np.mean(A,axis=0)
    X = A-mean


    return X
```

Test your function checking the following assertion on *testcase* (absence of output means that the assertion holds):

In [99]:
```
testcase = np.array([[3., 11., 4.3], [4., 5., 4.3], [5., 17., 4.5], [4, 1
answer = np.array([[-1., -0.5, -0.075], [0., -6.5, -0.075], [1., 5.5, 0.1
np.testing.assert_array_almost_equal(center_data(testcase), answer)
```

Computing Covariance Matrix

Implement a function with the following signature to compute the covariance matrix. In order to get this at the correct scale, divide by $N-1$, not N.

Note: Numpy provides a function `np.cov()` that does exactly this, but in this exercise we ask you to implement the code from scratch without using this function.

In [100…
```
def compute_covariance_matrix(A):
    # INPUT:
    # A    [NxM] numpy data matrix (N samples, M features)
    #
    # OUTPUT:
    # C    [MxM] numpy covariance matrix (M features, M features)

    A_centered = A-np.mean(A,axis= 0)
    C = (A_centered.T @ A_centered) / (A.shape[0]-1)

    return C
```

Test your function by comparing its output to the output of `np.cov()` :

In [101…

```
test_array = np.array([[22.,11.,5.5],[10.,5.,2.5],[34.,17.,8.5],[28.,14.,
answer = np.cov(np.transpose(test_array))
to_test = compute_covariance_matrix(test_array)
np.testing.assert_array_almost_equal(to_test, answer)
```

Computing eigenvalues and eigenvectors

Use the linear algebra package of `numpy` and its function `np.linalg.eig()` to compute eigenvalues and eigenvectors. Note that we only take the real part of the eigenvectors and eigenvalues. The covariance matrix *should* be a symmetric matrix, but the actual implementation in `compute_covariance_matrix()` can lead to small round off errors that lead to tiny imaginary additions to the eigenvalues and eigenvectors. These are purely numerical artifacts that we can safely remove.

Note: If you decide to NOT use `np.linalg.eig()` you must make sure that the eigenvalues you compute are of unit length!

In [102…

```
def compute_eigenvalue_eigenvectors(A):
    # INPUT:
    # A     [DxD] numpy matrix
    #
    # OUTPUT:
    # eigval     [D] numpy vector of eigenvalues
    # eigvec     [DxD] numpy array of eigenvectors

    eigval, eigvec = np.linalg.eig(A)

    # Numerical roundoff can lead to (tiny) imaginary parts. We correct t
    eigval = eigval.real
    eigvec = eigvec.real

    return eigval, eigvec
```

Test your function checking the following assertion on *testcase*:

In [103…

```
testcase = np.array([[2, 0, 0], [0, 5, 0], [0, 0, 4]])
answer_eigval = np.array([2., 5., 4.])
answer_eigvec = np.array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]])
x, y = compute_eigenvalue_eigenvectors(testcase)
np.testing.assert_array_almost_equal(x, answer_eigval)
np.testing.assert_array_almost_equal(y, answer_eigvec)
```

Sorting eigenvalues and eigenvectors

Implement a function with the following signature to sort eigenvalues and eigenvectors in descending order.

Remember that eigenvalue `eigval[i]` corresponds to eigenvector `eigvec[:,` i] .

In [104…
```
def sort_eigenvalue_eigenvectors(eigval, eigvec):
    # INPUT:
    # eigval    [D] numpy vector of eigenvalues
    # eigvec    [DxD] numpy array of eigenvectors
    #
    # OUTPUT:
    # sorted_eigval    [D] numpy vector of eigenvalues
    # sorted_eigvec    [DxD] numpy array of eigenvectors

    idx = np.argsort(eigval)[::-1]

    sorted_eigval, sorted_eigvec = eigval[idx], eigvec[:,idx]
    return sorted_eigval, sorted_eigvec
```

Test your function checking the following assertion on *testcase*:

In [105…
```
testcase = np.array([[2, 0, 0], [0, 5, 0], [0, 0, 4]])
answer_eigval = np.array([5., 4., 2.])
answer_eigvec = np.array([[0., 0., 1.], [1., 0., 0.], [0., 1., 0.]])
x, y = compute_eigenvalue_eigenvectors(testcase)
x, y = sort_eigenvalue_eigenvectors(x, y)
np.testing.assert_array_almost_equal(x, answer_eigval)
np.testing.assert_array_almost_equal(y, answer_eigvec)
```

PCA Algorithm

Implement a function with the following signature to compute PCA using the functions implemented above.

In [106…
```
def pca(A, m):
    # INPUT:
    # A    [NxM] numpy data matrix (N samples, M features)
    # m    integer number denoting the number of learned features (m <= M
    #
    # OUTPUT:
    # pca_eigvec    [Mxm] numpy matrix containing the eigenvectors (M dim
    # P             [Nxm] numpy PCA data matrix (N samples, m features)

    A_centered = center_data(A)
    cov_matrix = compute_covariance_matrix(A_centered)
    eigval,eigvec = compute_eigenvalue_eigenvectors(cov_matrix)
    sorted_eigval,sorted_eigvac = sort_eigenvalue_eigenvectors(eigval,eig

    pca_eigvec = sorted_eigvac[:, :m]
    P = A_centered @ pca_eigvec
    return pca_eigvec, P
```

Test your function checking the following assertion on *testcase*:

In [107…
```
import pickle
testcase = np.array([[22., 11., 5.5], [10., 5., 2.5], [34., 17., 8.5]])
x, y = pca(testcase, 2)

answer1 = pickle.load(open('PCAanswer1.pkl', 'rb'))
answer2 = pickle.load(open('PCAanswer2.pkl', 'rb'))

test_arr_x = np.sum(np.abs(np.abs(x) - np.abs(answer1)), axis=0)
np.testing.assert_array_almost_equal(test_arr_x, np.zeros(2))

test_arr_y = np.sum(np.abs(np.abs(y) - np.abs(answer2)))
np.testing.assert_almost_equal(test_arr_y, 0)
```

Part 1.2: Understanding - How does PCA work?

We now use the PCA algorithm you implemented on a toy data set in order to understand its inner workings.

Loading the data

The module *data_assignment3* provides a small synthetic dataset of dimension [100x2] (100 samples, 2 features):

In [108…
```
import data_assignment3

X = data_assignment3.get_synthetic_data()
```

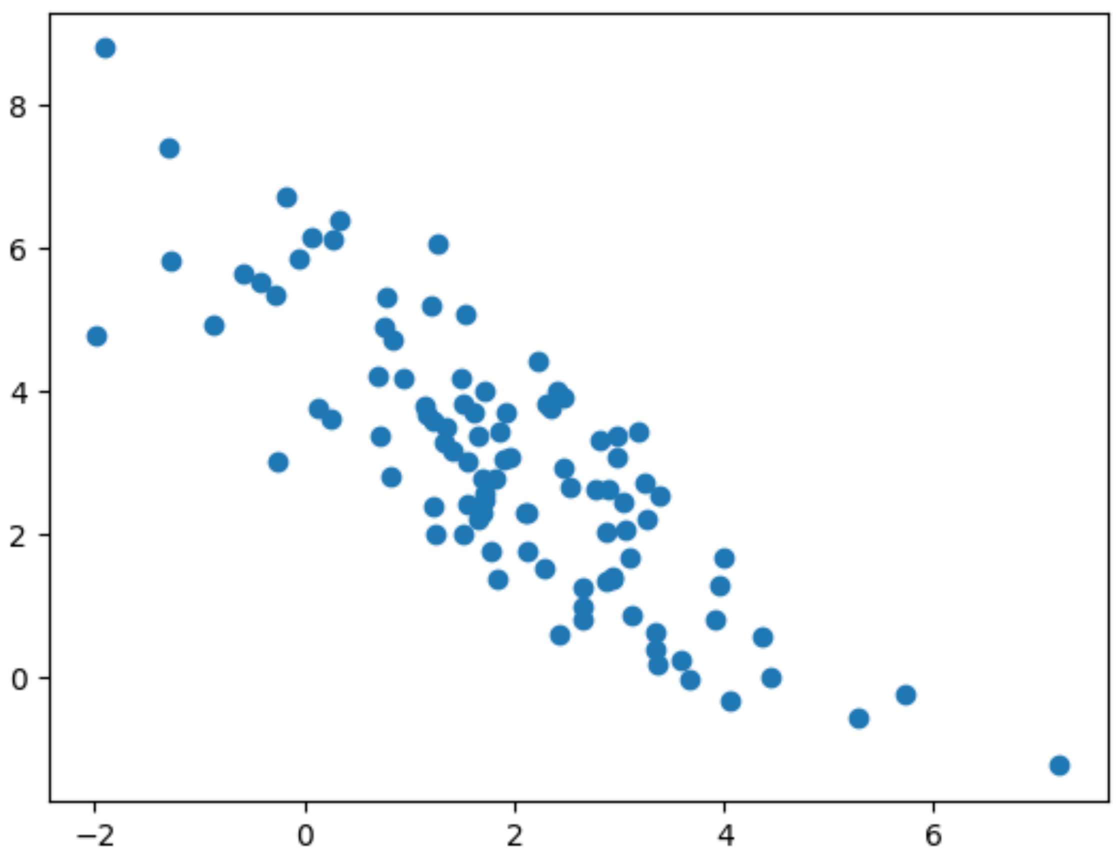
Visualizing the data

Visualize the synthetic data using the function *scatter()* from the *matplotlib* library.

In [109…
```
import matplotlib.pyplot as plt

plt.scatter(X[:, 0], X[:, 1])
```

Out[109…
```
<matplotlib.collections.PathCollection at 0x148af6170>
```

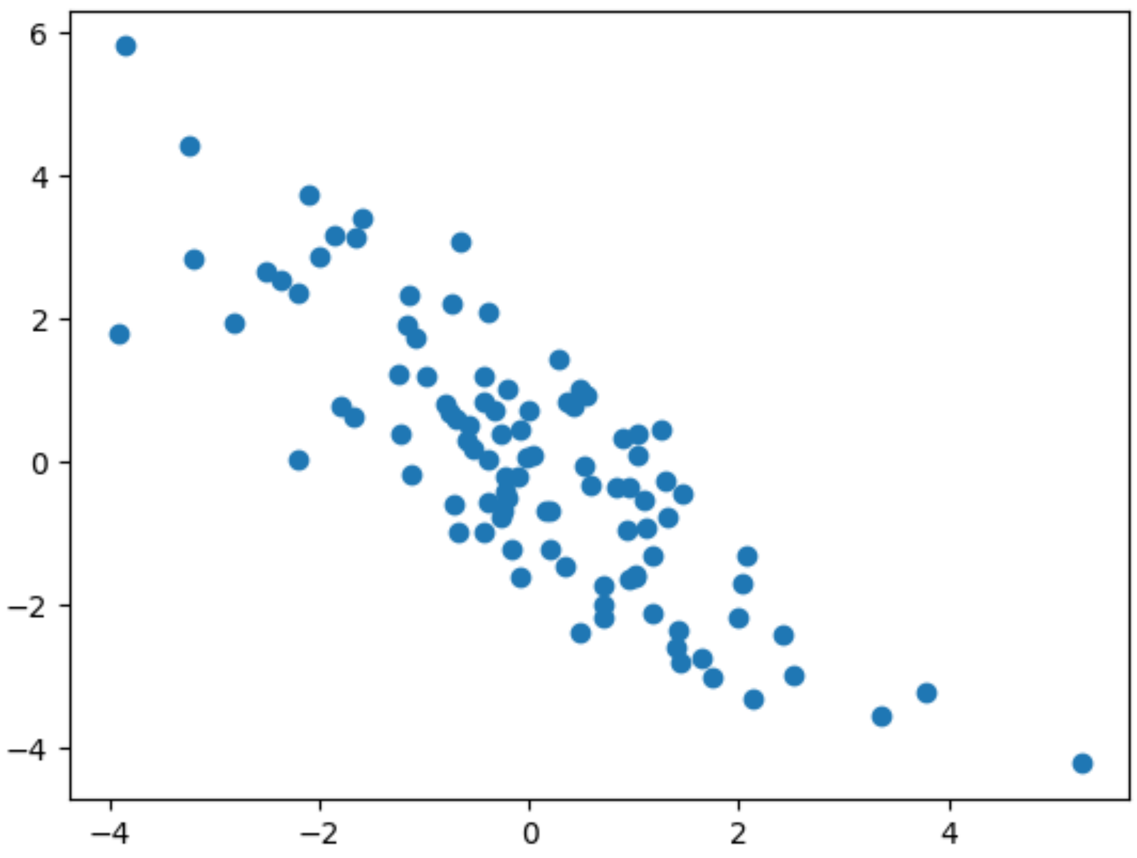


Visualize the centered data

Notice that the data visualized above is not centered on the origin (0,0). Use the function defined above to center the data, and the replot it.

In [110…
```python
## Your code here
X_centered = center_data(X)
plt.scatter(X_centered[:, 0],X_centered[:, 1])
```

Out[110…
```
<matplotlib.collections.PathCollection at 0x1495bac50>
```



Visualize the first eigenvector

Visualize the vector defined by the first eigenvector. To do this you need:

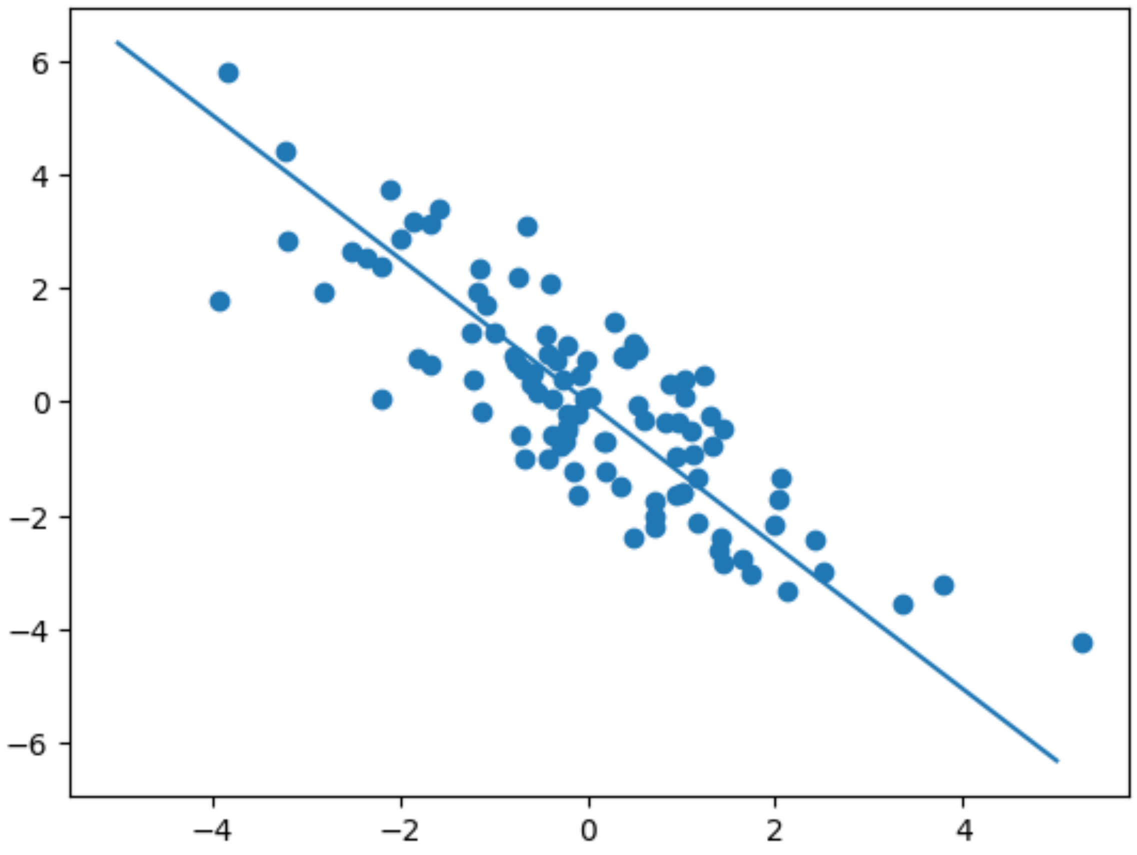
- Use the *PCA()* function to recover the eigenvectors
- Plot the centered data as done above
- The first eigenvector is a 2D vector (x0,y0). This defines a vector with origin in (0,0) and head in (x0,y0). Use the function *plot()* from matplotlib to plot a line over the first eigenvector.

In [111…
```
pca_eigvec = pca(X_centered,1)[0]
first_eigvec = pca_eigvec[:,0]

plt.scatter(X_centered[:,0],X_centered[:,1])

x = np.linspace(-5, 5, 1000)
y = first_eigvec[1]/first_eigvec[0] * x
plt.plot(x, y)
```

Out[111…
```
[<matplotlib.lines.Line2D at 0x1493d7f40>]
```



Visualize the PCA projection

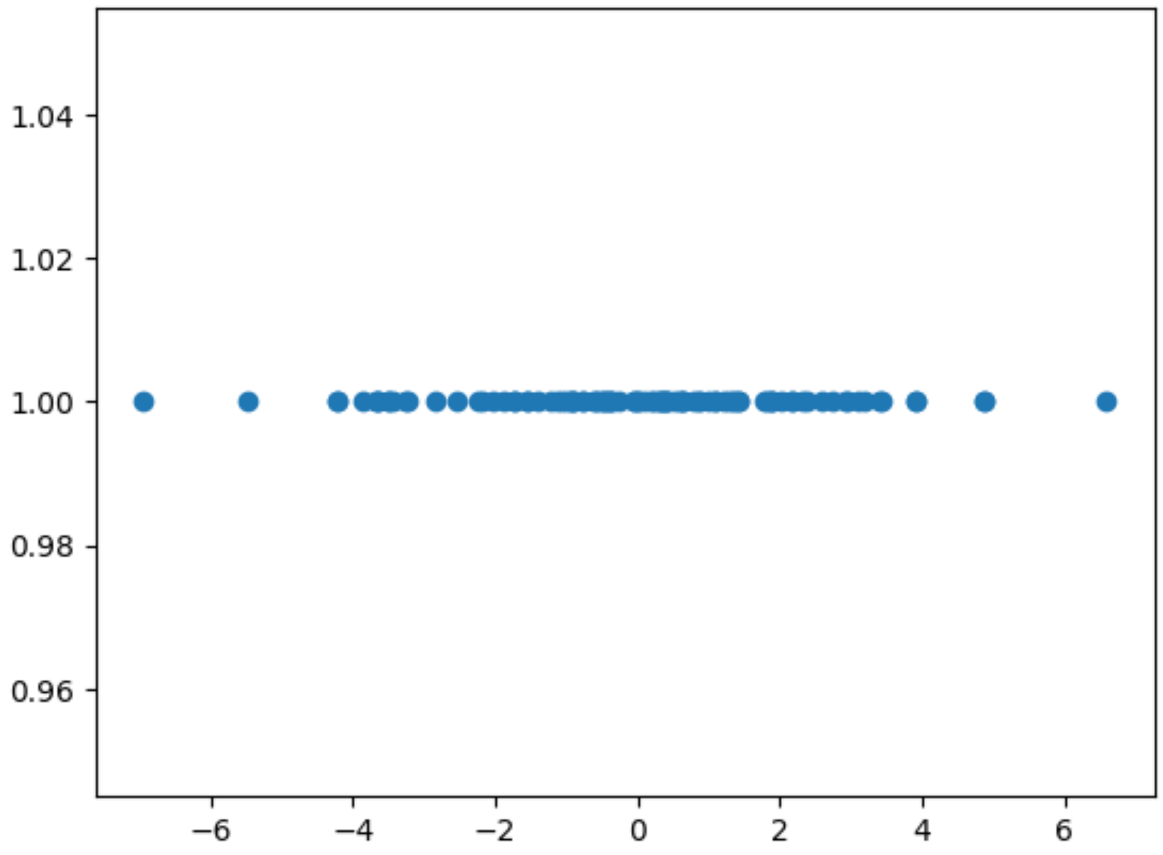
Finally, use the *PCA()* algorithm to project on a single dimension and visualize the result using again the *scatter()* function.

In [112…

```
## Your code here
P = pca(X,1)[1]
plt.scatter(P[:,0],np.ones_like(P[:,0]))
```

Out[112…

```
<matplotlib.collections.PathCollection at 0x1494c39d0>
```



Part 1.3: Evaluation - When are the results of PCA sensible?

So far we have used PCA on synthetic data. Let us now imagine we are using PCA as a pre-processing step before a classification task. This is a common setup with high-dimensional data. We explore when the use of PCA is sensible.

Loading the first set of labels

The function *get_synthetic_labeled_data_1()* from the module *data_assignment3* provides a first labeled dataset.

In [113…
```
X, y = data_assignment3.get_synthetic_labeled_data_1()
```

Running PCA

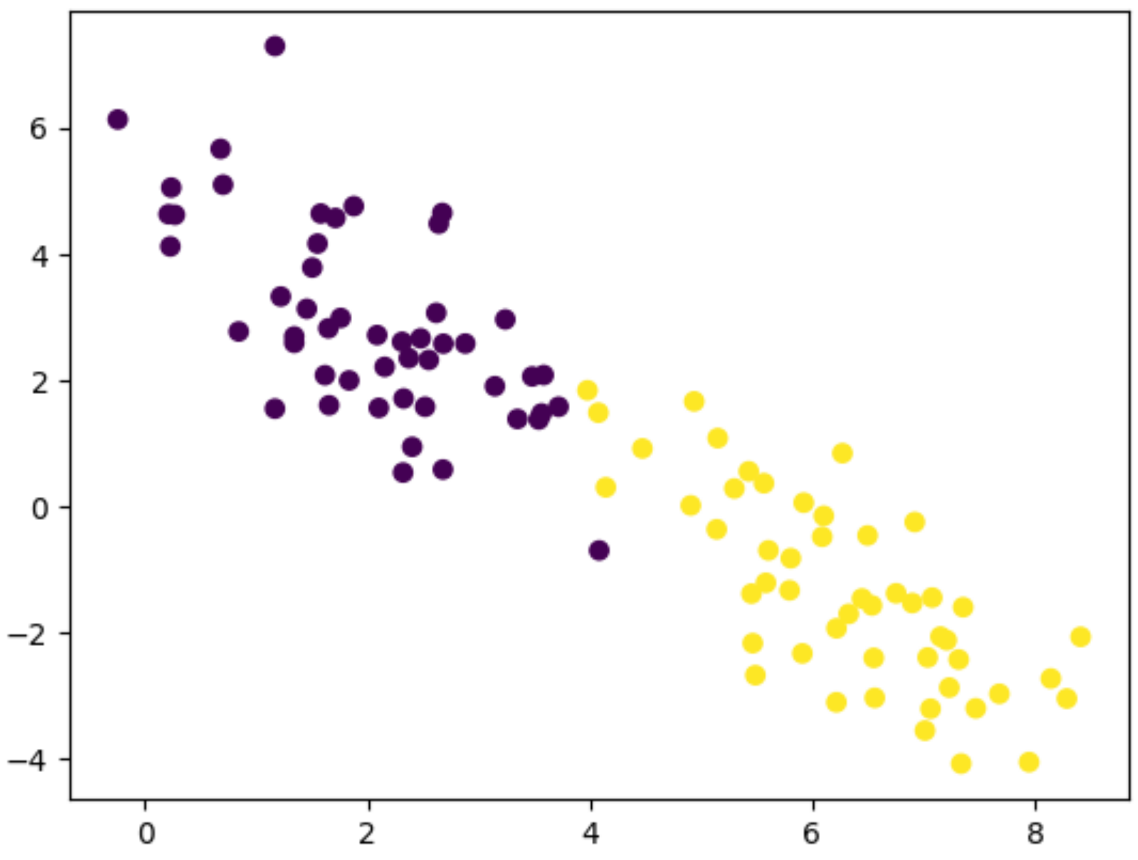
Process the data using the PCA algorithm and project it in one dimension. Plot the labeled data using *scatter()* before and after running PCA. Comment on the results.

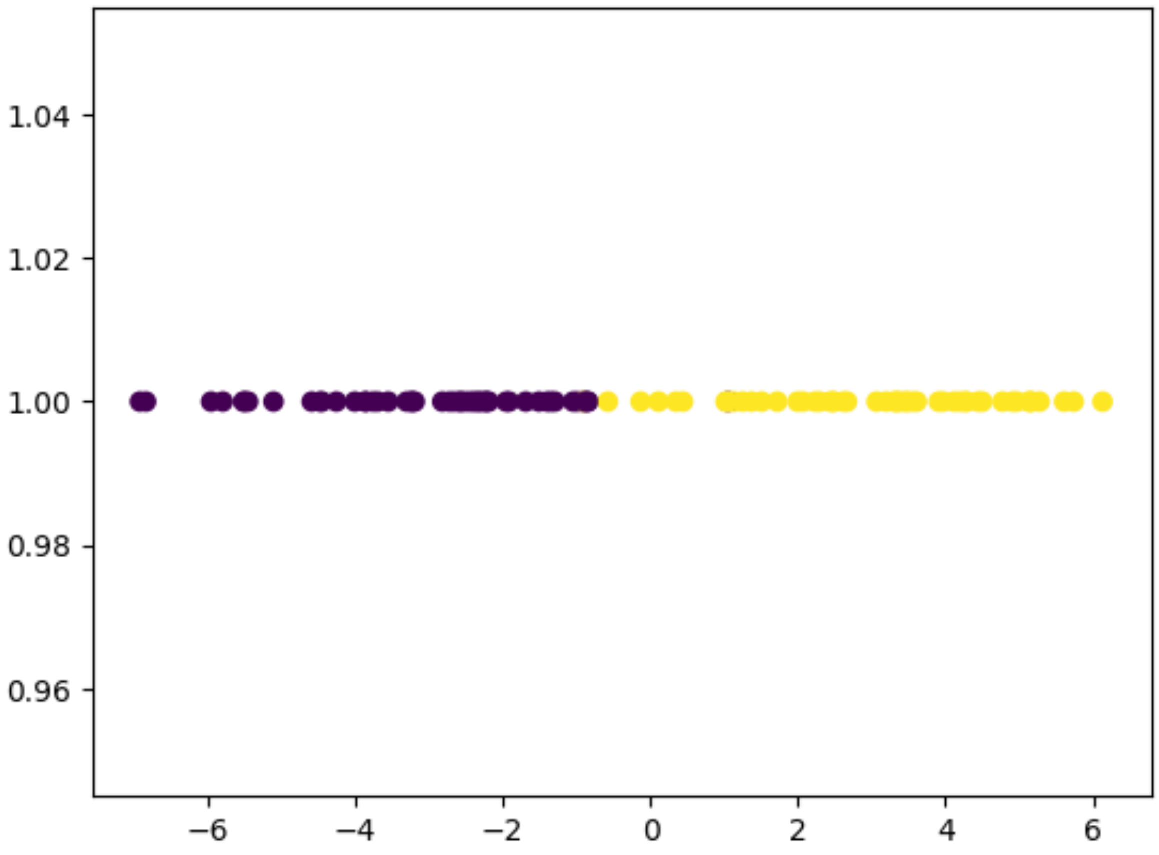
In [114…
```
# before PCA
plt.scatter(X[:, 0], X[:, 1], c=y[:, 0])
```

```
# after PCA
plt.figure()
P = pca(X,1)[1]
plt.scatter(P[:,0],np.ones_like([P[:,0]]), c=y[:,0])
# ...
```

Out[114…

```
<matplotlib.collections.PathCollection at 0x12c0ae530>
```





Comment: before pca the data is scattered across more dimensions, after the pca it is one dimensional, we can also see that the it is separated with no overlap.

Loading the second set of labels

The function *get_synthetic_labeled_data_2()* from the module *data_assignment3* provides a second labeled dataset.

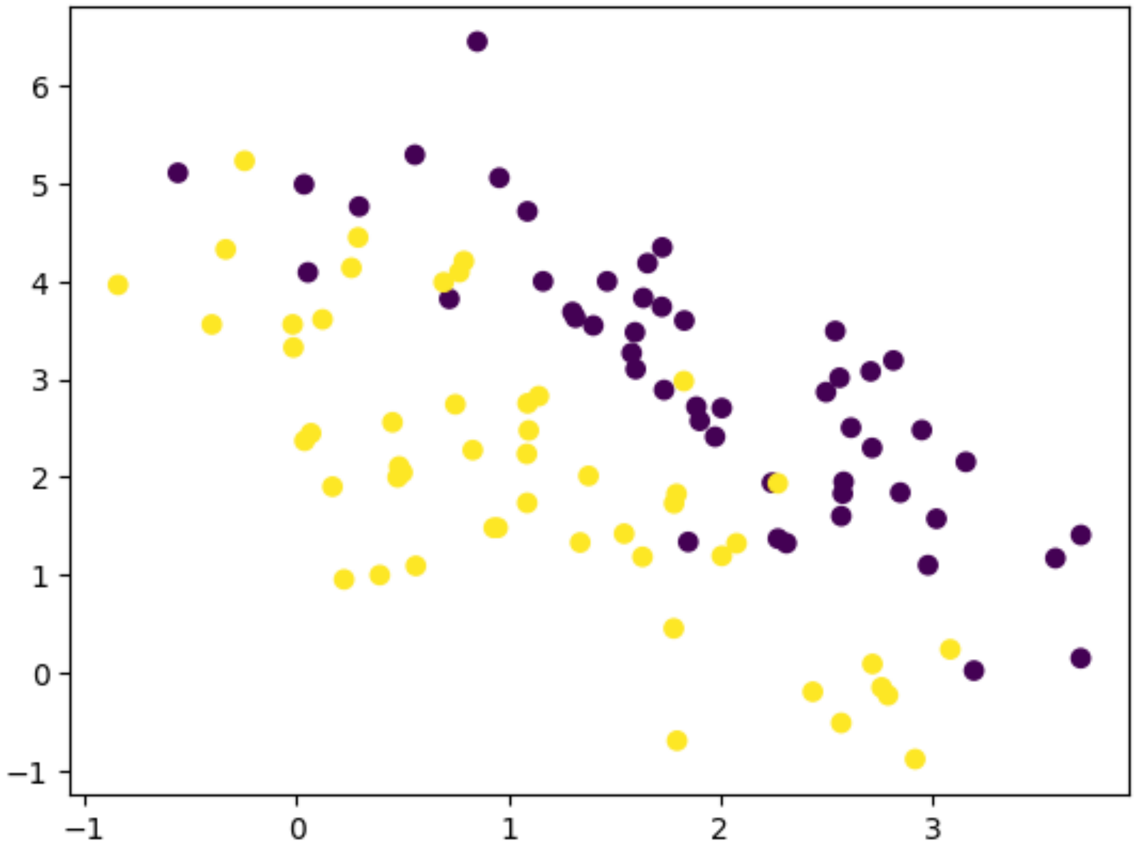
In [115…
```
X, y = data_assignment3.get_synthetic_labeled_data_2()
```

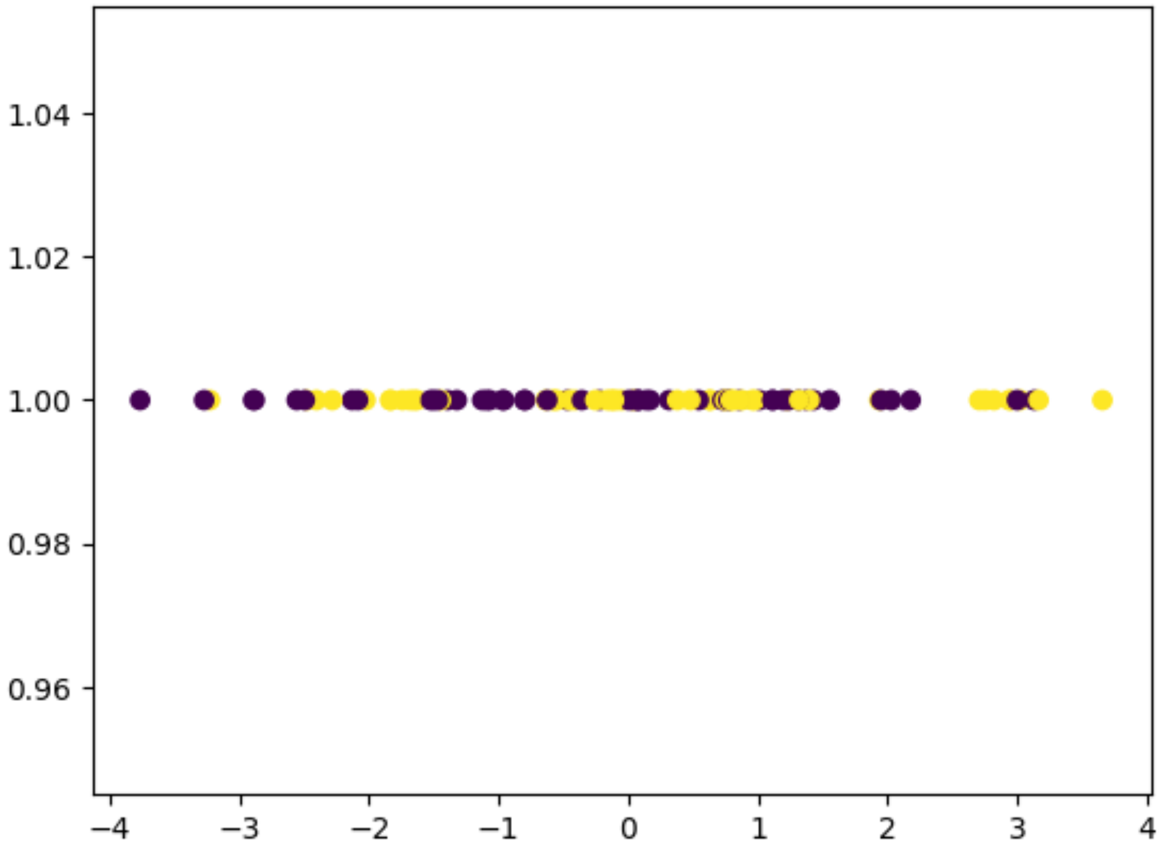
Running PCA

As before, process the data using the PCA algorithm and project it in one dimension. Plot the labeled data using *scatter()* before and after running PCA. Comment on the results.

In [116…
```
# Your code here
#before
plt.scatter(X[:, 0], X[:, 1], c=y[:, 0])
plt.figure()
#after
P = pca(X,1)[1]
plt.scatter(P[:,0],np.ones_like([P[:,0]]), c=y[:,0])
```

```
Out[116… <matplotlib.collections.PathCollection at 0x1485c5180>
```





Comment: Same as the previous dataset with before pca the data is mulit dimensional while after pca it is single dimensional. On this dataset it is not seperated at all even after pca.

How would the result change if you were to consider only the second eigenvector? What about if you were to consider both eigenvectors?

Answer: With the second eigenvector we might lose important data about data variability which means less accuracy. If we consider both eigenvectors we would get more stucture and variablility of the data which in turn means better accuracy on the data.

Part 1.4: Case study 1 - PCA for visualization

The *iris* flower dataset is one of the oldest and best known data collections used for machine learning. It consists of 50 samples from each of three species of iris flowers (*Iris setosa, Iris virginica and Iris versicolor*). Four features were measured from each sample: sepal length, sepal width, petal length and petal width, all in centimeters.

Visualizing a 4-dimensional dataset is impossible; therefore we will use PCA to project our data in 2 dimensions and visualize it.

Loading the data

The function *get_iris_data()* from the module *data_assignment3* returns the *iris* dataset. It returns a data matrix of dimension [150x4] and a label vector of dimension [150].

In [117…
```python
X, y = data_assignment3.get_iris_data()
```

Visualizing the data by selecting features

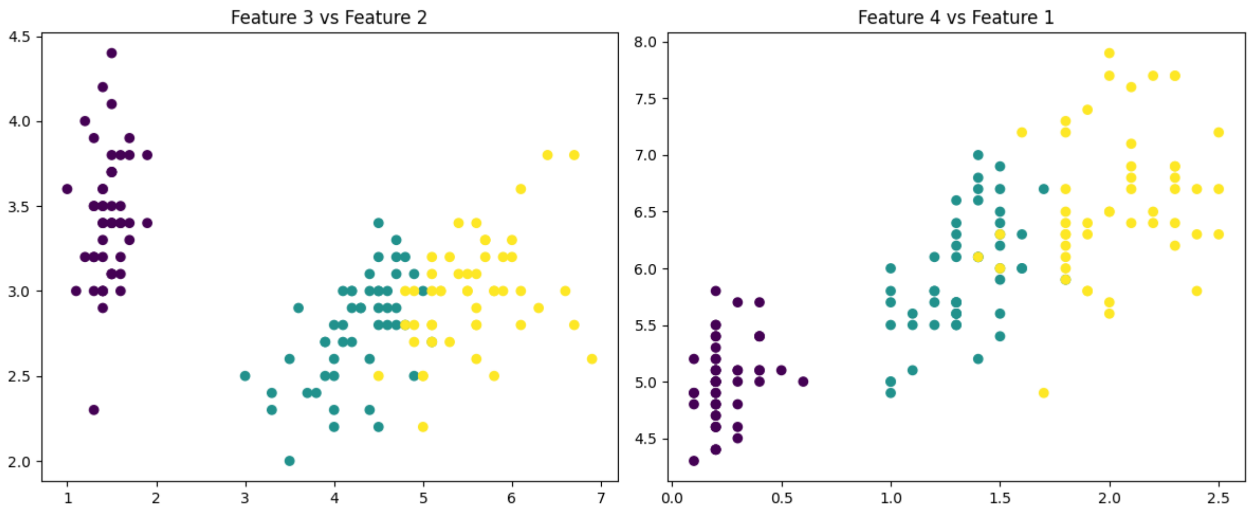
Try to visualize the data (using label information) by randomly selecting two out of the four features of the data. You may try different pairs of features.

In [134…
```python
# Your code here
pairs = np.random.choice(range(4),size=(2,2),replace=False)

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.scatter(X[:, pairs[0][0]], X[:, pairs[0][1]], c=y)
plt.title(f"Feature {pairs[0][0]+1} vs Feature {pairs[0][1]+1}")

plt.subplot(1, 2, 2)
plt.scatter(X[:, pairs[1][0]], X[:, pairs[1][1]], c=y)
plt.title(f"Feature {pairs[1][0]+1} vs Feature {pairs[1][1]+1}")
plt.tight_layout()
plt.show()
```

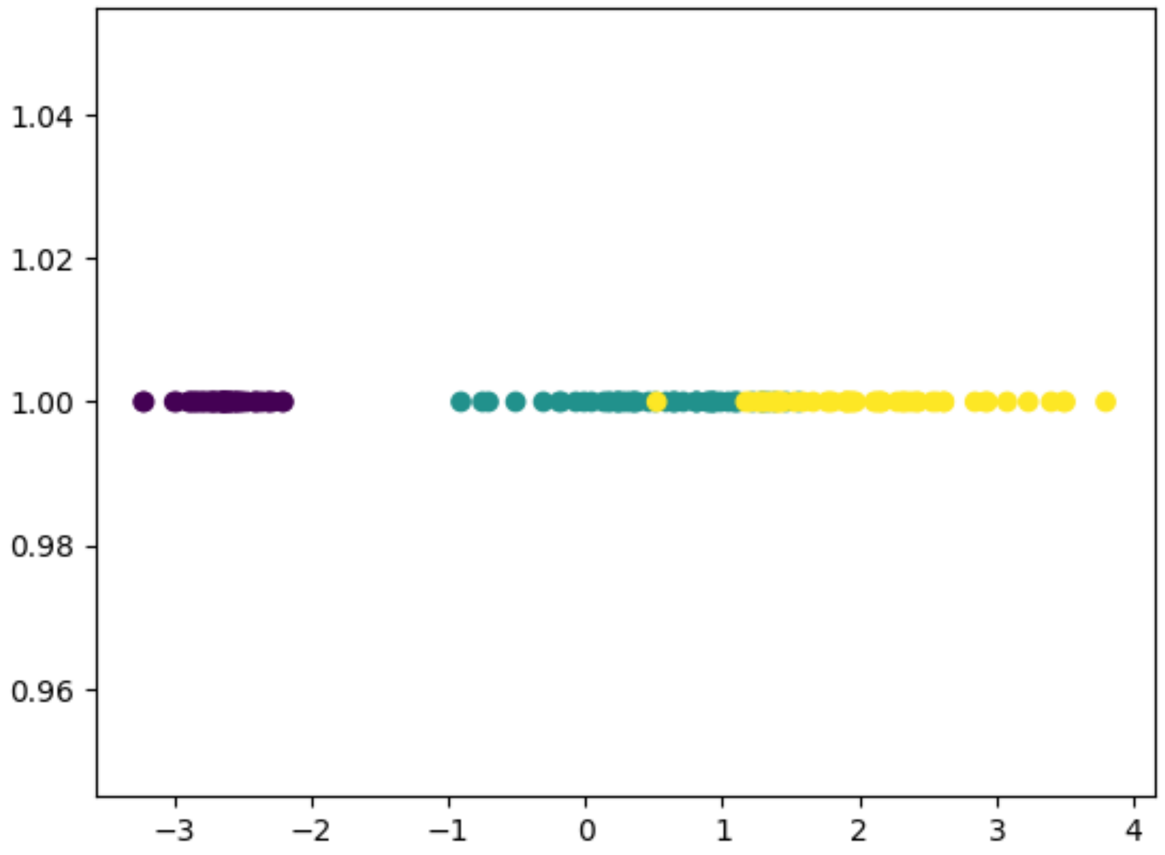


Visualizing the data by PCA

Process the data using PCA and visualize it (using label information). Compare with the previous visualization and comment on the results.

In [139…
```
# Your code here
P = pca(X,X.shape[1])[1]
plt.scatter(P[:, 0], np.ones(P.shape[0]), c=y)
```

Out[139…
```
<matplotlib.collections.PathCollection at 0x148fb40d0>
```



Comment: One is seperated by a lot from the others

Part 1.5: Case study 2 - PCA for compression

We now consider the *faces in the wild (lfw)* dataset, a collection of pictures (N=1280) of people. Each pixel in the image is a feature (M=2914).

Loading the data

The function *get_lfw_data()* from the module *data_assignment3* returns the *lfw* dataset. It returns a data matrix of dimension [1280x2914] and a label vector of dimension [1280]. It also returns two parameters, h and w, reporting the height and the width of the images (these parameters are necessary to plot the data samples as images). Beware, it might take some time to download the data. Be patient :)

In [120…

```
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
# This part over is only because this part of the code wont run locally o
X, y, h, w = data_assignment3.get_lfw_data()
```

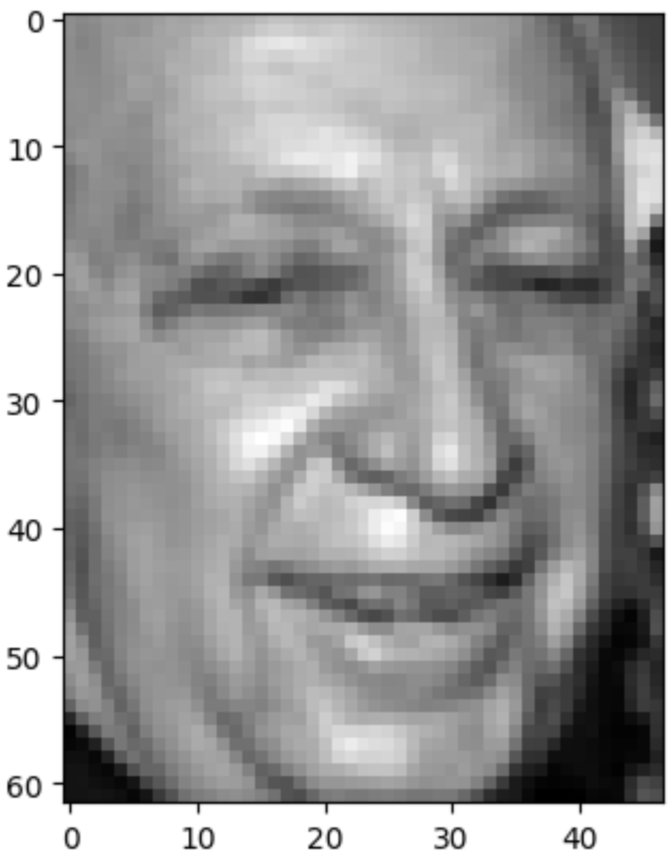
Inspecting the data

Choose one datapoint to visualize (first coordinate of the matrix X) and use the function imshow() to plot and inspect some of the pictures.

Notice that the first argument of *imshow* is the image to be plotted; the image must be provided as a rectangular matrix, therefore we reshape a sample from the matrix X to have height h and width w. The parameter *cmap* specifies the color coding; in our case we will visualize the image in black-and-white with different gradations of grey.

In [121…
```python
image_number = 4    # display the 4th image of the collection
plt.imshow(X[image_number, :].reshape((h, w)), cmap=plt.cm.gray)
```

Out[121…
```
<matplotlib.image.AxesImage at 0x148a41390>
```



Implementing a compression-decompression function

Implement a function that first uses PCA to project samples in low-dimensions, and then reconstruct the original image.

Hint: Most of the code is the same as the previous PCA() function you implemented.

In [122…
```python
def encode_decode_pca(A, m):
    # INPUT:
    # A    [NxM] numpy data matrix (N samples, M features)
    # m    integer number denoting the number of learned features (m <= M
    #
    # OUTPUT:
    # Ahat [NxM] numpy PCA reconstructed data matrix (N samples, M featur

    pca_eigvec,P = pca(A,m)
    Ahat = np.dot(P,pca_eigvec.T)



    return Ahat
```

Compressing and decompressing the data

Use the implemented function to encode and decode the data by projecting on a lower dimensional space of dimension 200 (m=200).

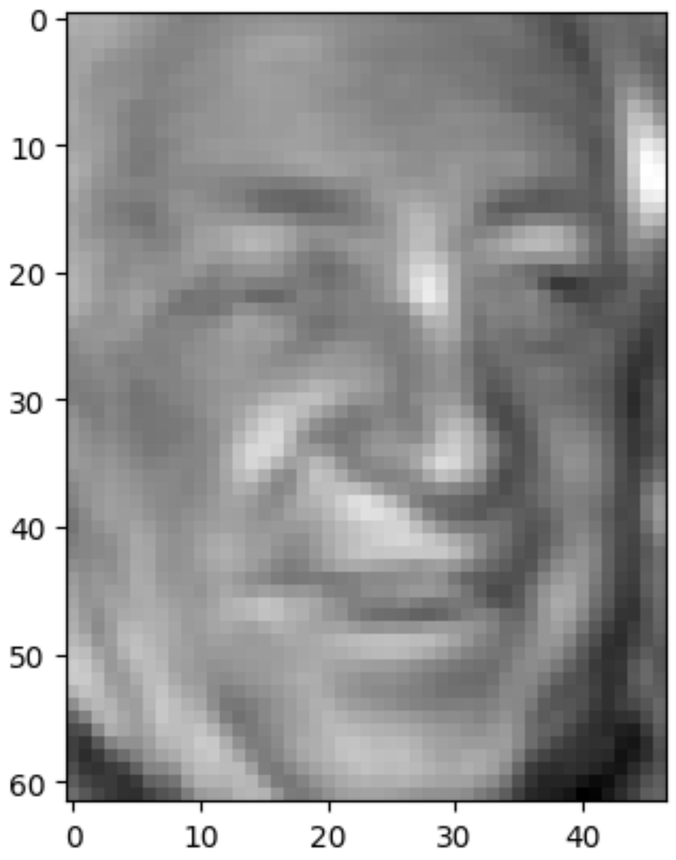
In [123…
```
Xhat = encode_decode_pca(X,200)
```

Inspecting the reconstructed data

Use the function *imshow* to plot and compare original and reconstructed pictures. Comment on the results.

In [124…
```
# Your code here
plt.imshow(Xhat[image_number, :].reshape((h, w)),cmap=plt.cm.gray)
```

Out[124…
```
<matplotlib.image.AxesImage at 0x149186050>
```



Comment: The reconstruced picture is less detailed than the pre reconstructed picture

Evaluating different compressions

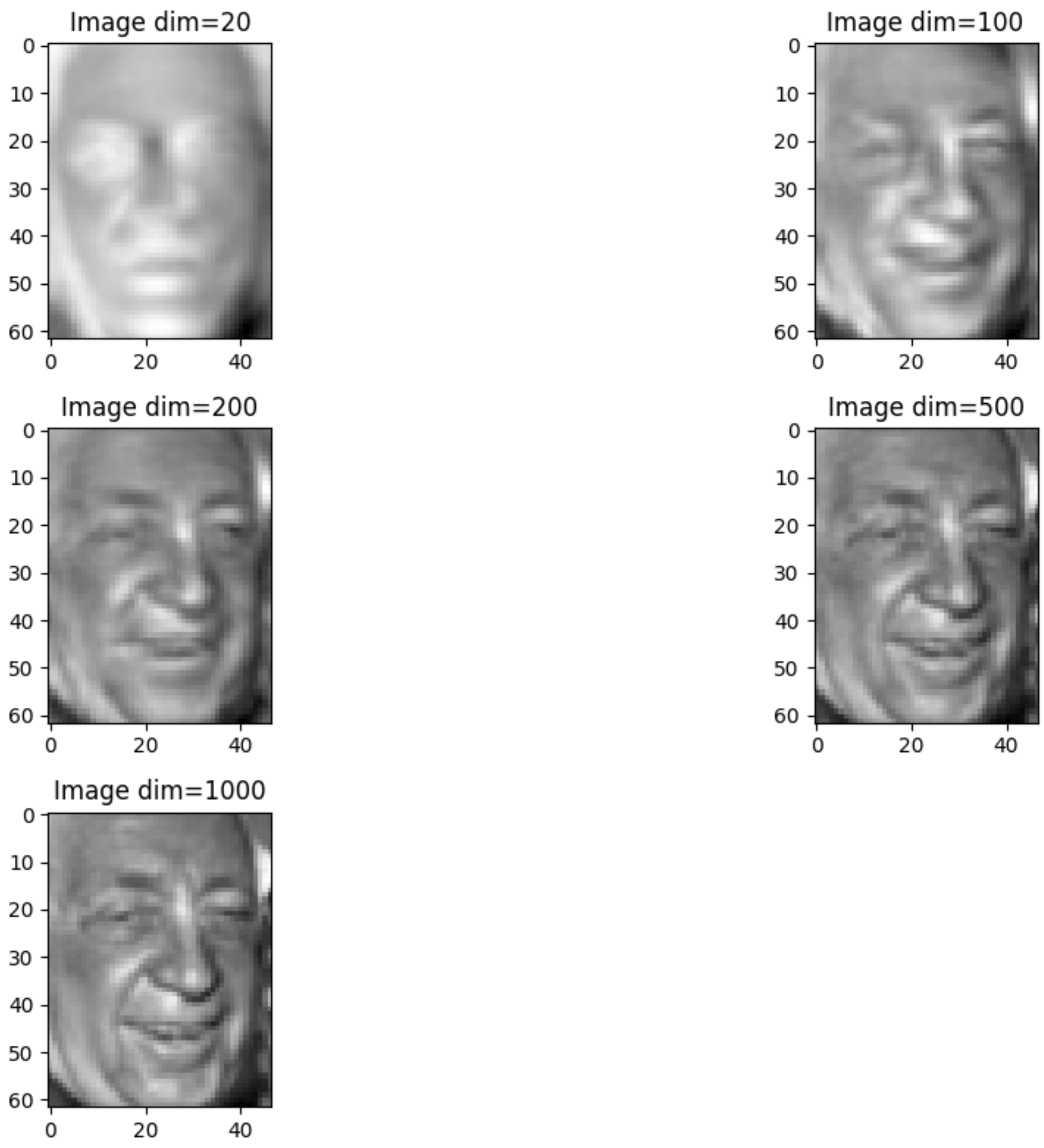
Use the previous setup to generate compressed images using different values of low dimensions in the PCA algorithm (e.g.: 20, 100, 200, 500, 1000). Plot and comment on the results. You can use `plt.subplot(n_rows, n_cols, position)` and `plt.title(titlestring)` to get a nice plot.

```
In [ ]: plt.figure(figsize=(12, 8))

        dims = [20, 100,200,500,1000]  # Example dimensions


        n_images = len(dims)
        n_cols = 2
        n_rows = (n_images + n_cols - 1) // n_cols

        for i, dim in enumerate(dims):
            position = i + 1
            plt.subplot(n_rows, n_cols, position)
            Xhat = encode_decode_pca(X, dim)
            plt.imshow(Xhat[image_number, :].reshape((h, w)), cmap=plt.cm.gray)
            plt.title(f"Image dim={dim}")
        plt.tight_layout()
        plt.show()
```



Comment: From this we can see tgat as we increase the dimension, the image becomes clearer and more detailed as the original one. Also the jump from 20 to 100 is a lot more significant than the jump from 500 to 1000.

Part 1.6: PCA tuning (compulsory for Master students only)

If we use PCA for compression or decompression, it may be not trivial to decide how many dimensions to keep. In this section we review a principled way to decide how many dimensions to keep.

The number of dimensions to keep is the only *hyper-parameter* of PCA. A method designed to decide how many dimensions/eigenvectors is the *proportion of variance*:

$$\mathrm{POV} = \frac{\sum_{i=1}^{m} \lambda_i}{\sum_{j=1}^{M} \lambda_j},$$

where λ are eigenvalues, M is the dimensionality of the original data, and m is the chosen lower dimensionality.

Using the POV formula we may select a number m of dimensions/eigenvalues so that the proportion of variance is, for instance, equal to 95%.

Implement a new PCA for encoding and decoding that receives in input not the number of dimensions for projection, but the amount of proportion of variance to be preserved.

In [126…
```python
def encode_decode_pca_with_pov(A, p):
    # INPUT:
    # A    [NxM] numpy data matrix (N samples, M features)
    # p    float number between 0 and 1 denoting the POV to be preserved
    #
    # OUTPUT:
    # Ahat [NxM] numpy PCA reconstructed data matrix (N samples, M featur
    # m    integer reporting the number of dimensions selected

    Ahat = None
    m = None
    return Ahat, m
```

Import the `lfw` dataset again. Use the implemented function to encode and decode the data by projecting on a lower dimensional space such that `POV=0.95`. Use the function `imshow` to plot and compare original and reconstructed pictures. Comment on the results.

In [127…
```python
X, y, h, w = data_assignment3.get_lfw_data()

Xhat, m = None, None
print("Selected dimensions:", m)

# Plot the images here
```

```
Selected dimensions: None
```

Comment: Enter your comment here.

Part 2: K-Means Clustering

In this section you will use the *k-means clustering* algorithm to perform unsupervised clustering. Then you will perform a qualitative assessment of the results.

Part 2.1: Applying K-Means and Qualitative Assessment

Importing scikit-learn library

We start importing the module `sklearn.cluster.KMeans` from the standard machine learning library `scikit-learn`.

In [128…

```
from sklearn.cluster import KMeans
```

Loading the data

We will use once again the *iris* data set. Start by loading the dataset again.

In [129…

```
X, y = data_assignment3.get_iris_data()
```

Projecting the data using PCA

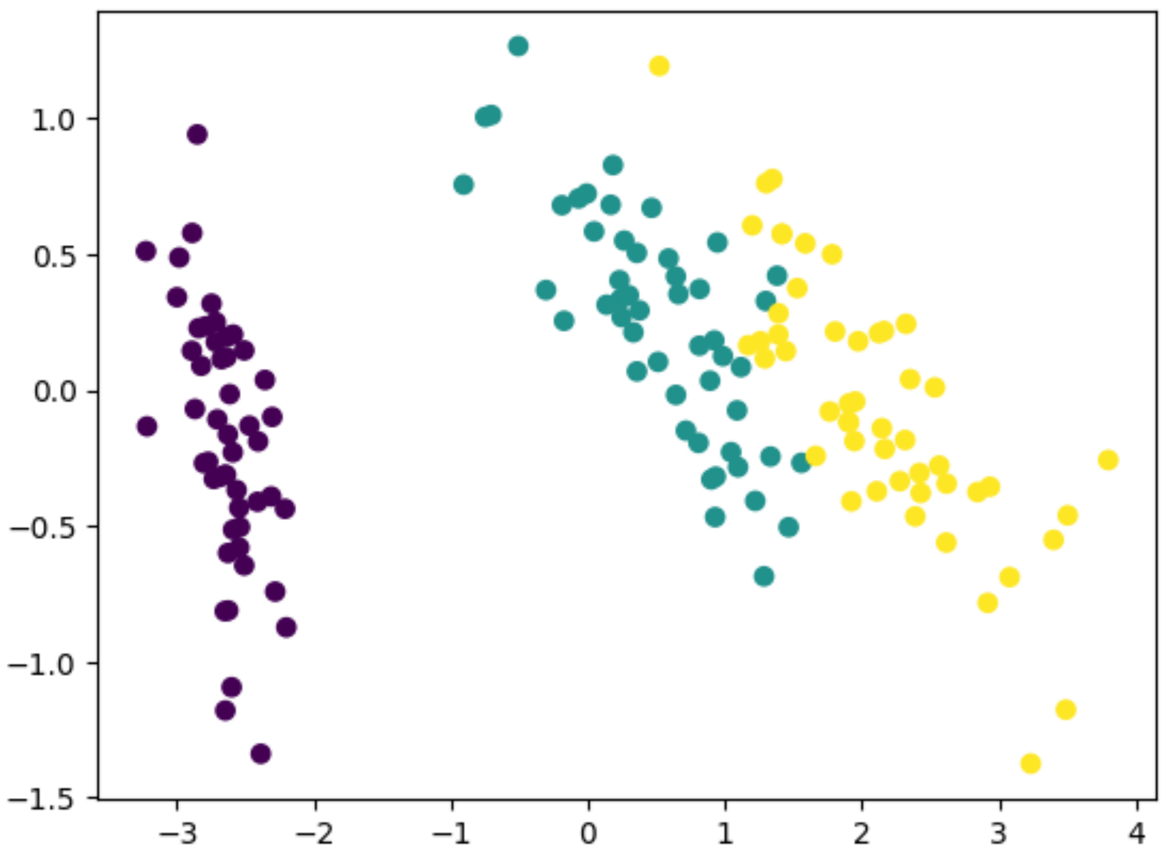
To allow for visualization, we project our data in two dimensions as we did previously. This step is not necessary, and we may want to try to use *k-means* later without the PCA pre-processing. But to start, we use PCA, as this will allow for an easy visualization.

In [130…

```
# Your code here
X_pca = pca(X,2)[1]
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y)
```

Out[130…

```
<matplotlib.collections.PathCollection at 0x1493a1ba0>
```



Running k-means

Use the class *KMeans* to fit and predict the output of the *k-means* algorithm on the projected data (note that we don't use the true labels `y` here). Run the algorithm using the following values of $k = \{2, 3, 4, 5\}$.

In [131…

```
k_values = [2, 3, 4, 5]
y_hats = []

for k in k_values:
    # Initialize KMeans object with correct value for k
    # Fit the algorithm to the training data and store the predictions in
    kmeans = KMeans(n_clusters=k,random_state=42)
    y_hat = kmeans.fit_predict(X_pca)
    y_hats.append(y_hat)
```

Qualitative assessment

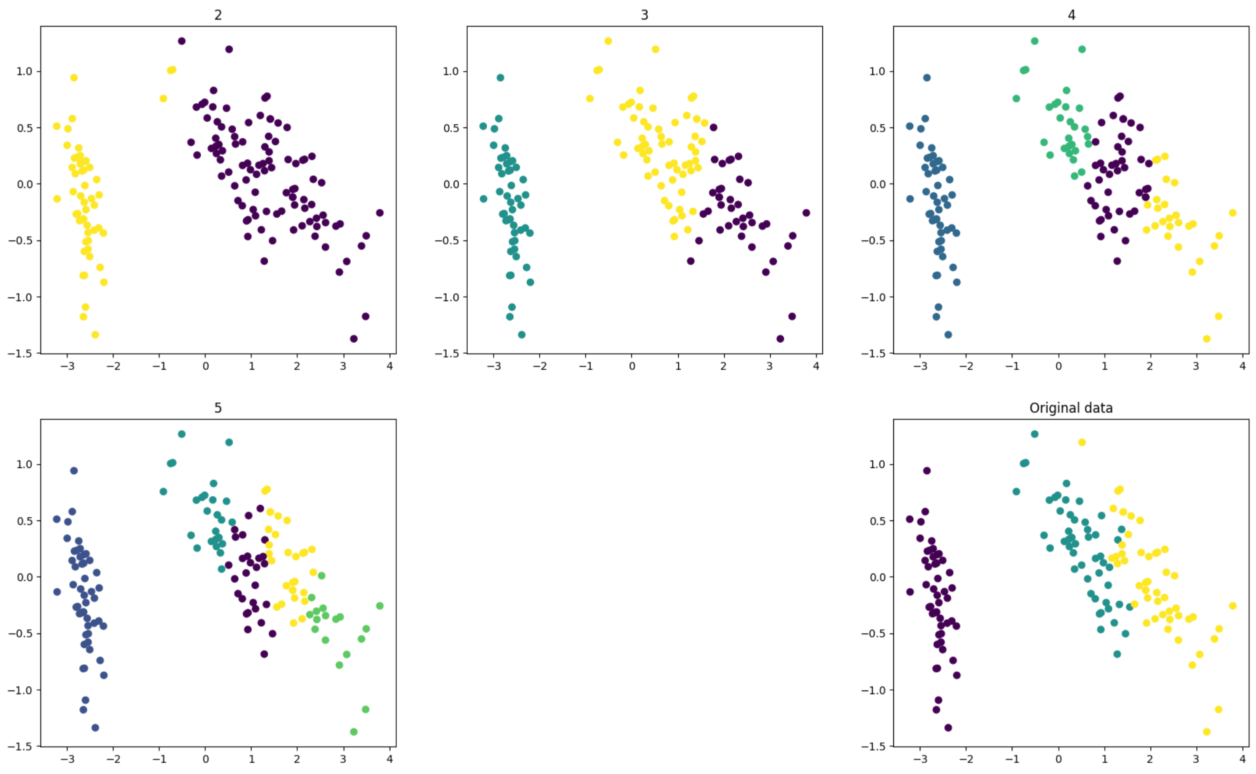
Plot the results of running the k-means algorithm, compare with the true labels, and comment.

In [132…

```
plt.figure(figsize=(20, 12))
for i, y_hat in enumerate(y_hats):
    plt.subplot(2, 3, i+1)
    plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y_hat)
    plt.title(i+2)


plt.subplot(2, 3, 6)
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y)
plt.title('Original data')


plt.show()
```



Comment: k = 3 is most equal to the true labels

Part 2.2: Quantitative Assessment of K-Means

Above, we used k-means for clustering and assessed the results qualitatively by visualizing them. However, we often want to measure in a quantitative way how good a clustering is. To do this, we will use a classification task to **evaluate numerically how good the learned clusters are for all the different values of k you used above (2 to 5)**.

Informally, our evaluation will work as follows: For each of our clusterings ($k = 2$ to $k = 5$), we will try to learn a mapping from the identified clusters to the correct labels of our datapoints. The reason this can be a sensible evaluation, is that to learn a good mapping, we have to have identified clusters that correspond to the actual classes in our data. We will in other words train 4 different classification models (one for each k value), where the input to our classifier is the cluster each datapoint belongs to, and the target is the correct class for this datapoint. In other words, **we aim to learn to**

classify datapoints as well as possible with the only information available to the classifier being the cluster that datapoint belongs to. For some values of k, we will get poorly performing classifiers, indicating that this clustering has not revealed the correct class division in our data.

In practice, you will do the following: Reload the *iris* dataset. Import a standard `LogisticRegression` classifier from the module `sklearn.linear_model`. Use the k-means representations learned previously (`yhats[2],...,yhats[5]`) and the true label to train the classifier. Evaluate your model on the training data (we do not have a test set, so this procedure will assess the model fit instead of generalization) using the `accuracy_score()` function from the `sklearn.metrics` module. Plot a graph showing how the accuracy score varies when changing the value of k. Comment on the results.

- Train a Logistic regression model using the first two dimensions of the PCA of the iris data set as input, and the true classes as targets.
- Report the model fit/accuracy on the training set.
- For each value of K:
  - One-Hot-Encode the classes outputed by the K-means algorithm.
  - Train a Logistic regression model on the K-means classes as input vs the real classes as targets.
  - Calculate model fit/accuracy vs. value of K.
- Plot your results in a graph and comment on the K-means fit.

In [146…

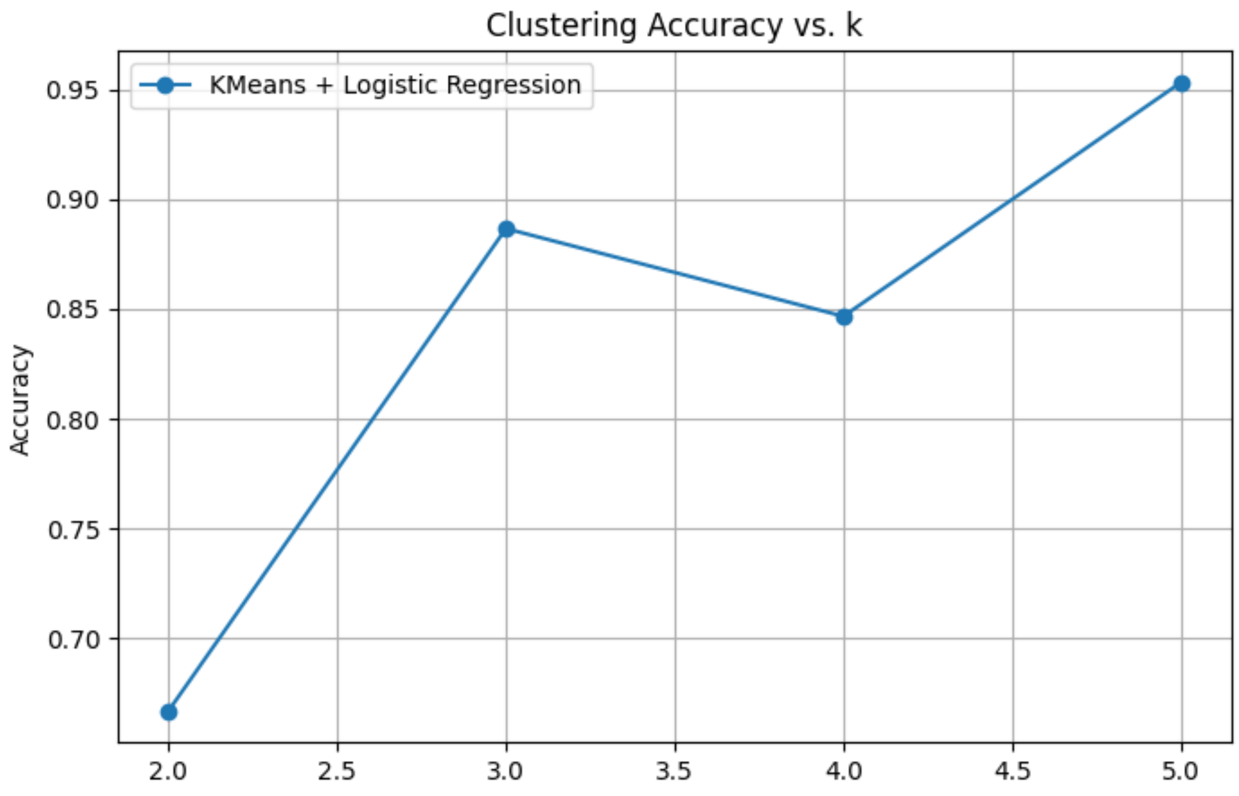
```python
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
from sklearn.preprocessing import OneHotEncoder

logreg = LogisticRegression(max_iter=200)
logreg.fit(X_pca, y)
baseline_acc = metrics.accuracy_score(y, logreg.predict(X_pca))
k_values = [2, 3, 4, 5]
kmeans_accuracies = []

for i, k in enumerate(k_values):
    y_hat = y_hats[i].reshape(-1, 1)  # reshape for OneHotEncoder
    encoder = OneHotEncoder(sparse_output=False)
    X_kmeans_encoded = encoder.fit_transform(y_hat)

    logreg = LogisticRegression(max_iter=200)
    logreg.fit(X_kmeans_encoded, y)
    preds = logreg.predict(X_kmeans_encoded)
    acc = metrics.accuracy_score(y, preds)
    kmeans_accuracies.append(acc)

plt.figure(figsize=(8, 5))
plt.plot(k_values, kmeans_accuracies, marker='o', label='KMeans + Logisti
plt.ylabel('Accuracy')
plt.title('Clustering Accuracy vs. k')
plt.legend()
plt.grid(True)
plt.show()
```



Comment: With more clusters the accuracy becomes better with the exception og k=4. The highest accuracy shown by the graph is 0.96 at k = 5

Conclusions

In this notebook we studied **unsupervised learning** considering two important and representative algorithms: **PCA** and **k-means**.

First, we implemented the PCA algorithm step by step; we then ran the algorithm on synthetic data in order to see its working and evaluated when it makes sense to use it and when it doesn't. We then considered two typical uses of PCA: for **visualization** on the *iris* dataset, and for **compression-decompression** on the *lfw* dataset. We also looked at an additional question that arises when using PCA: the problem of **selection of hyper-parameters**, that is, how to select the optimal hyper-parameter of our algorithm for a particular task.

We then moved to consider the k-means algorithm. In this case we used the implementation provided by *scikit-learn* and applied it to another prototypical unsupervised learning problem: **clustering**. We processed the *iris* dataset with *k-means* and evaluated the results visually. We also considered the problem of **quantitative evaluation** of the results, that is, how to measure the performance or usefulness of k-means clustering on a downstream task (classifying the *iris* samples into their species).