assignment03

May 12, 2021

1 IN3050/IN4050 Mandatory Assignment 3, 2021: Unsupervised Learning

1.0.1 Goals of the exercise

This exercise has three parts. The first part is focused on Principal Component Analysis (PCA). You will go through some basic theory, and implent PCA from scratch to do compression and visualization of data.

The second part focuses on clustering using K-means. You will use `scikit-learn` to run K-means clustering, and use PCA to visualize the results.

The last part ties supervised and unsupervised learning together in an effort to evaluate the output of K-means using a logistic regression for multi-class classification approach.

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

1.0.2 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 encurage the use of `scikit-learn` to iterate quickly on the problems.

1.0.3 Beware

This is a new assignment. There might occur typos or ambiguities. If anything is unclear, do not hesitate to ask. Also, if you think some assumptions are missing, make your own and explain them!

1.1 Principal Component Analysis (PCA)

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

1.1.1 Principle of Maximum Variance: what is PCA supposed to do?

First of all, let us recall the principle/assumption of PCA:

1. What is the variance?
2. What is the covariance?
3. How do we compute the covariance matrix?
4. What is the meaning of the principle of maximum variance?

5. Why do we need this principle?
6. Does the principle always apply?

Answers: Enter your answers here.

1. The variance is defined in mathematics as

$$\sigma^2 = E([X - E(X)^2])$$

where E() is the expectation value. In ther words, it is defined as the square of each observation distance from the mean divided by the total number of observations. It is used to measure how far a set of observations are spread out from their average value.

2. The covarinace is a measure of the linear dependence between two varying quantities. It is defined as

$$Cov(X, Y) = E[(X - E[X])(Y - E[Y])]$$

for two observations X and Y. It is used to assess whether there is a connection between two sets of variables or not.

3. To compute the covariance matrix we use the following equation:

$$C = \frac{1}{N} X^T X$$

where X is the data matrix, X^T is the transpose of the X and N is the total amount of observations.

4. The meaning of the principle of maximum variance is minimizing squared error. In the case of PCA, the algorithm allign the directions into where the data is the most spread out or the direction with the maximum variance.

5. We need this principle because it allows us to remove or ignore the variables where the variance is almost zeros or where there is no variation between data points.

6. This principle will be applied most of the times, but not always. It will struggle with a data set where the all variables have already a maximum variance.

1.2 Implementation: how is PCA implemented?

Here we implement the basic steps of PCA and we assemble them.

1.2.1 Importing libraries

We start importing the *numpy* library for performing matrix computations, the *pyplot* library for plotting data, and the *syntheticdata* module to import synthetic data.

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

import syntheticdata
```

1.2.2 Centering the Data

Implement a function with the following signature to center the data as explained in *Marsland*.

[3]:
```
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)

    center = np.mean(A,axis = 0)
    A = A - center
    return A
```

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

[4]:
```
testcase = np.array([[3.,11.,4.3],[4.,5.,4.3],[5.,17.,4.5],[4,13.,4.4]])
answer = np.array([[-1.,-0.5,-0.075],[0.,-6.5,-0.075],[1.,5.5,0.125],[0.,1.5,0.
↪025]])
np.testing.assert_array_almost_equal(center_data(testcase), answer)
```

1.2.3 Computing Covariance Matrix

Implement a function with the following signature to compute the covariance matrix as explained in *Marsland*.

[5]:
```
def compute_covariance_matrix(A):
    # INPUT:
    # A    [NxM] centered numpy data matrix (N samples, M features)
    #
    # OUTPUT:
    # C    [MxM] numpy covariance matrix (M features, M features)
    #
    # Do not apply centering here. We assume that A is centered before this␣
↪function is called.

    C = np.cov(np.transpose(A))

    return C
```

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

[6]:
```
testcase = center_data(np.array([[22.,11.,5.5],[10.,5.,2.5],[34.,17.,8.5],[28.
↪,14.,7]]))
answer = np.array([[580.,290.,145.],[290.,145.,72.5],[145.,72.5,36.25]])

# Depending on implementation the scale can be different:
to_test = compute_covariance_matrix(testcase)

answer = answer/answer[0, 0]
to_test = to_test/to_test[0, 0]
```

```
np.testing.assert_array_almost_equal(to_test, answer)
```

1.2.4 Computing eigenvalues and eigenvectors

Use the linear algebra package of `numpy` and its function `np.linalg.eig()` to compute eigenvalues and eigenvectors. Notice that we take the real part of the eigenvectors and eigenvalues. The covriance 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 lenght!

[7]:
```
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 that␣
 ↪here.
    eigval = eigval.real
    eigvec = eigvec.real

    return eigval, eigvec
```

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

[8]:
```
testcase = np.array([[2,0,0],[0,5,0],[0,0,3]])
answer1 = np.array([2.,5.,3.])
answer2 = np.array([[1.,0.,0.],[0.,1.,0.],[0.,0.,1.]])
x,y = compute_eigenvalue_eigenvectors(testcase)
np.testing.assert_array_almost_equal(x, answer1)
np.testing.assert_array_almost_equal(y, answer2)
```

1.2.5 Sorting eigenvalues and eigenvectors

Implement a function with the following signature to sort eigenvalues and eigenvectors as explained in *Marsland*.

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

[9]:
```
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

    indices = np.argsort(eigval) #finding indices of eigval sorted in ascending␣
 ↪order
    indices = indices[::-1] #reversing indices in descending order

    sorted_eigval = eigval[indices]
    sorted_eigvec = eigvec[:,indices]

    return sorted_eigval, sorted_eigvec
```

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

[10]:
```
testcase = np.array([[2,0,0],[0,5,0],[0,0,3]])
answer1 = np.array([5.,3.,2.])
answer2 = 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, answer1)
np.testing.assert_array_almost_equal(y, answer2)
```

1.2.6 PCA Algorithm

Implement a function with the following signature to compute PCA as explained in *Marsland* using the functions implemented above.

[11]:
```
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␣
 ↪dimensions, m eigenvectors)
    # P             [Nxm] numpy PCA data matrix (N samples, m features)


    A = center_data(A)
    C = compute_covariance_matrix(A)
    eigval,eigvec = compute_eigenvalue_eigenvectors(C)
```

```
    sorted_eigval, sorted_eigvec = sort_eigenvalue_eigenvectors(eigval, eigvec)

    if m > 0:
        pca_eigvec = sorted_eigvec[:,:m]

    P = np.dot(np.transpose(pca_eigvec),np.transpose(A))

    return pca_eigvec, P.T
```

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

[12]:
```
testcase = np.array([[22.,11.,5.5],[10.,5.,2.5],[34.,17.,8.5]])
x,y = pca(testcase,2)

import pickle
answer1_file = open('PCAanswer1.pkl','rb'); answer2_file = open('PCAanswer2.
↪pkl','rb')
answer1 = pickle.load(answer1_file); answer2 = pickle.load(answer2_file)


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)
```

1.3 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.

1.3.1 Loading the data

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

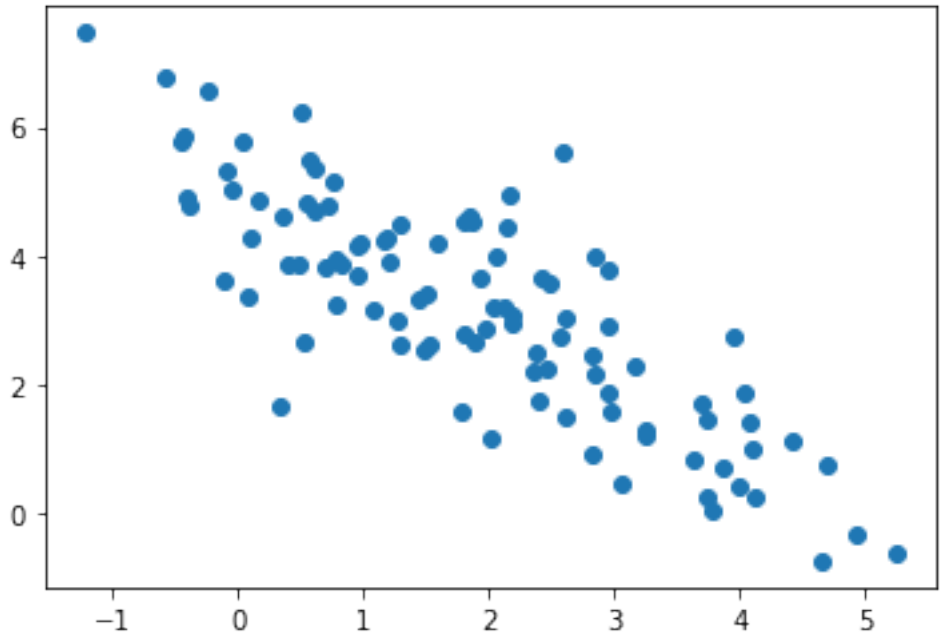
[13]:
```
X = syntheticdata.get_synthetic_data1()
```

1.3.2 Visualizing the data

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

[14]:
```
plt.scatter(X[:,0],X[:,1])
```

[14]:
```
<matplotlib.collections.PathCollection at 0x7fb0f3998ad0>
```

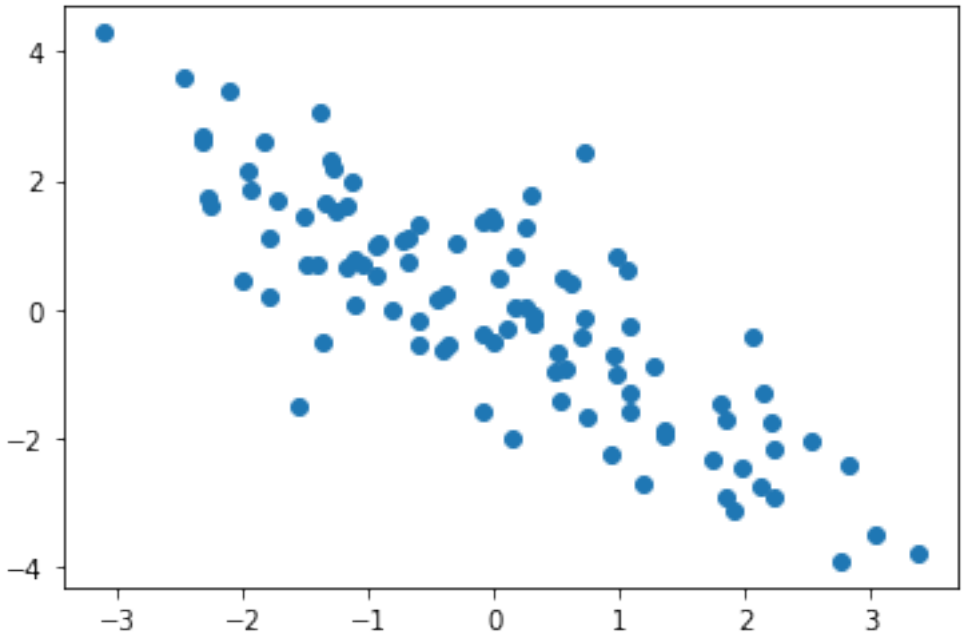


1.3.3 Visualize the centered data

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.

[15]:
```
X = center_data(X)
plt.scatter(X[:,0], X[:,1])
```

[15]:
```
<matplotlib.collections.PathCollection at 0x7fb0f3abe290>
```

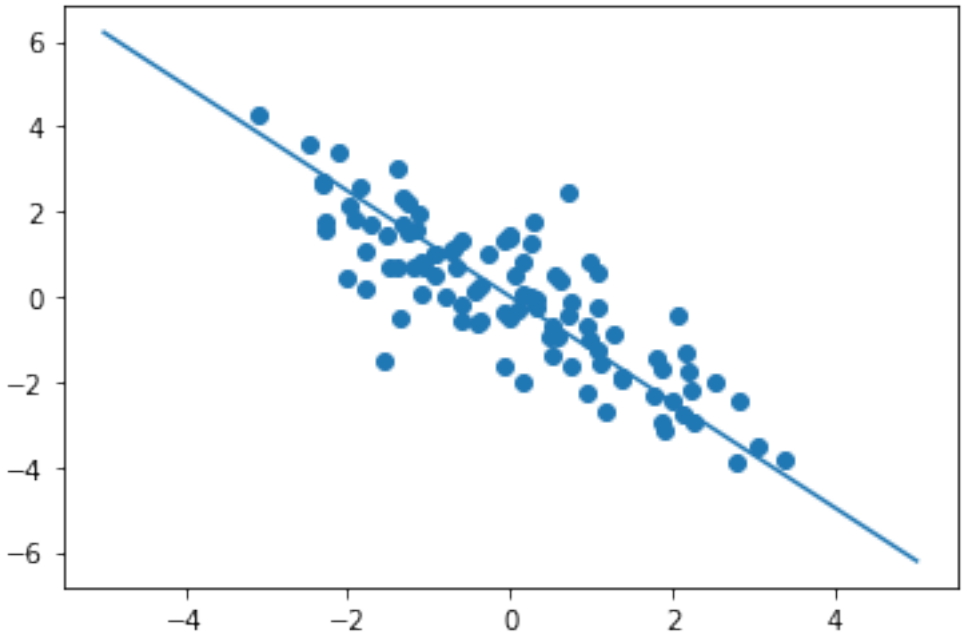


1.3.4 Visualize the first eigenvector

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.

[16]:
```
pca_eigvec, _ = pca(X,2)
first_eigvec = pca_eigvec[0]

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

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

[16]:
```
[<matplotlib.lines.Line2D at 0x7fb0f3b66cd0>]
```

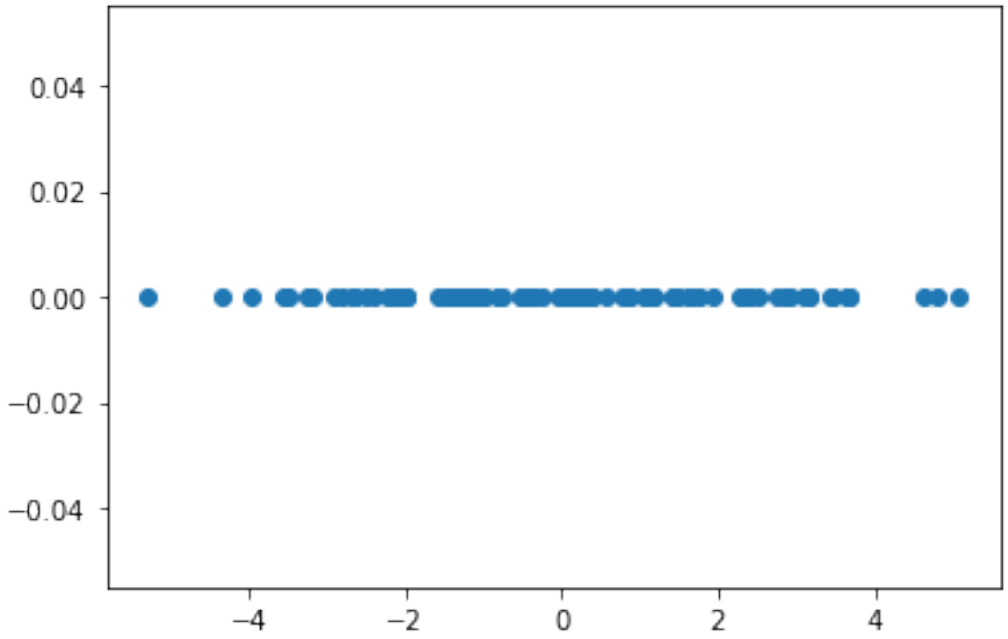


1.3.5 Visualize the PCA projection

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

[17]:
```
_,P = pca(X,2)

#projecting from (x,y) -> (x,0)
plt.scatter(P[:,0],np.zeros(P.shape[0]))
```

[17]:
```
<matplotlib.collections.PathCollection at 0x7fb0f3dfc8d0>
```



1.4 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.

1.4.1 Loading the first set of labels

The function *get_synthetic_data_with_labels1()* from the module *syntethicdata* provides a first labeled dataset.

[24]:
```
X,y = syntheticdata.get_synthetic_data_with_labels1()
```

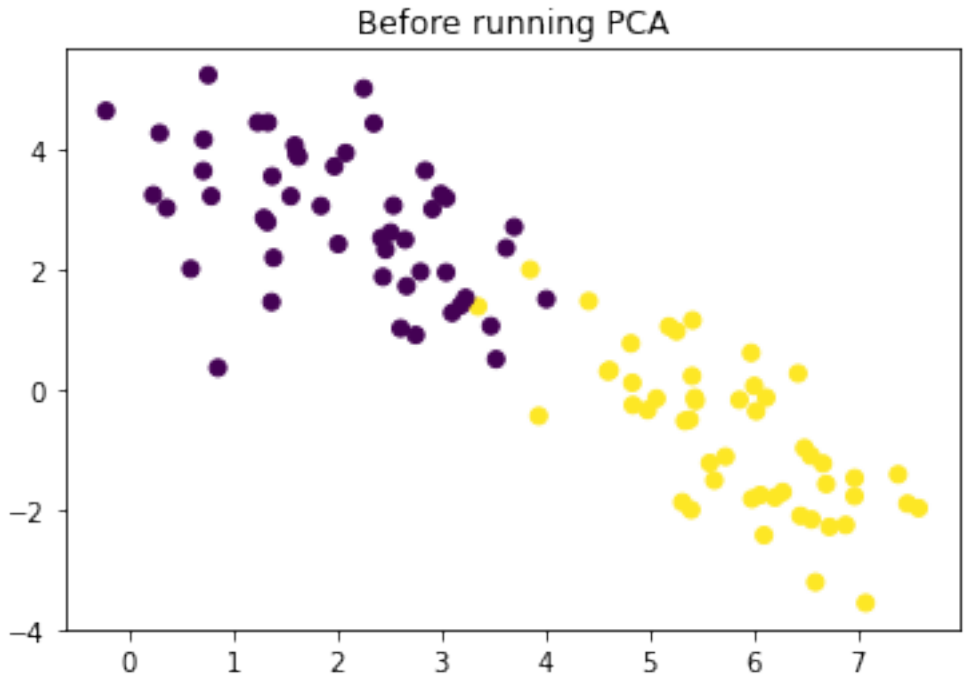
1.4.2 Running PCA

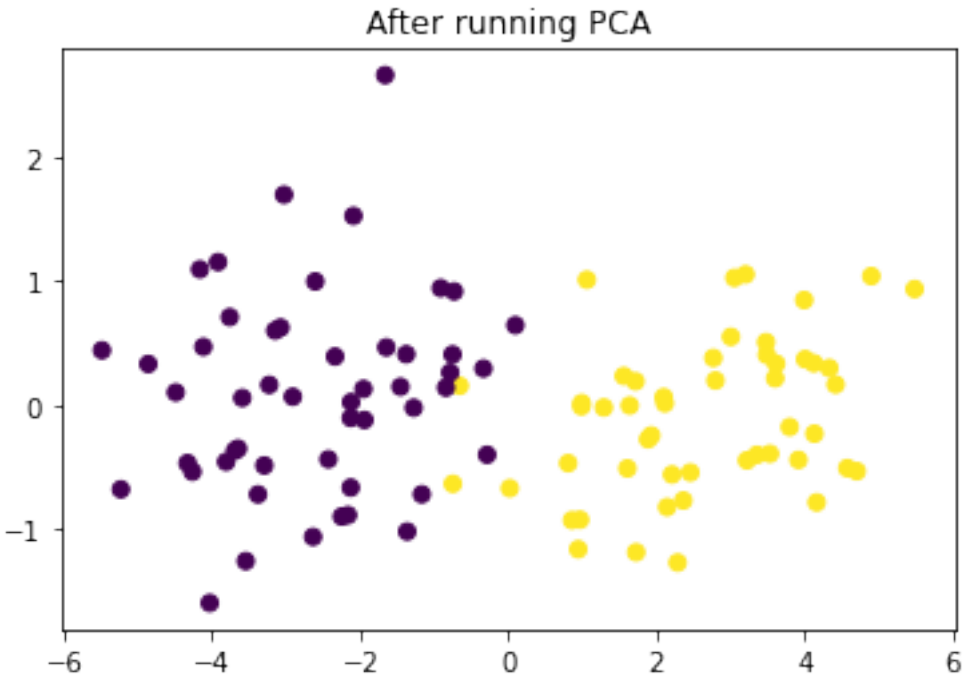
Process the data using the PCA algorithm and project it in one dimension. Plot the labeled data using *scatter()* before and after running PCA. Comment on the results.

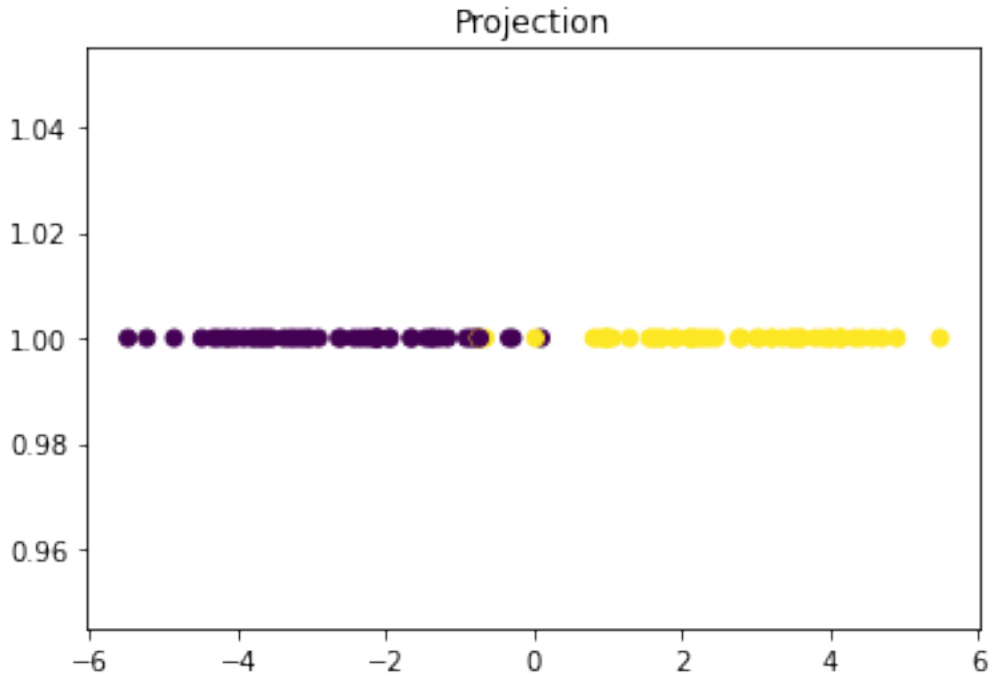
[26]:
```
plt.scatter(X[:,0],X[:,1],c=y[:,0])
plt.title("Before running PCA")

plt.figure()
_,P = pca(X,2)
plt.scatter(P[:,0],P[:,1],c=y[:,0])
plt.title("After running PCA")
plt.figure()
plt.scatter(P[:,0],np.ones(P.shape[0]),c=y[:,0])
```

```
plt.title("Projection")
```

[26]: Text(0.5, 1.0, 'Projection')







Comment: Each data point has been tranformed from two dimensions into one dimension. We can see that the projection is quite similiar to the actual plot and we can clearly see project of each point.

1.4.3 Loading the second set of labels

The function *get_synthetic_data_with_labels2()* from the module *syntethicdata* provides a second labeled dataset.

[27]:
```
X,y = syntheticdata.get_synthetic_data_with_labels2()
```

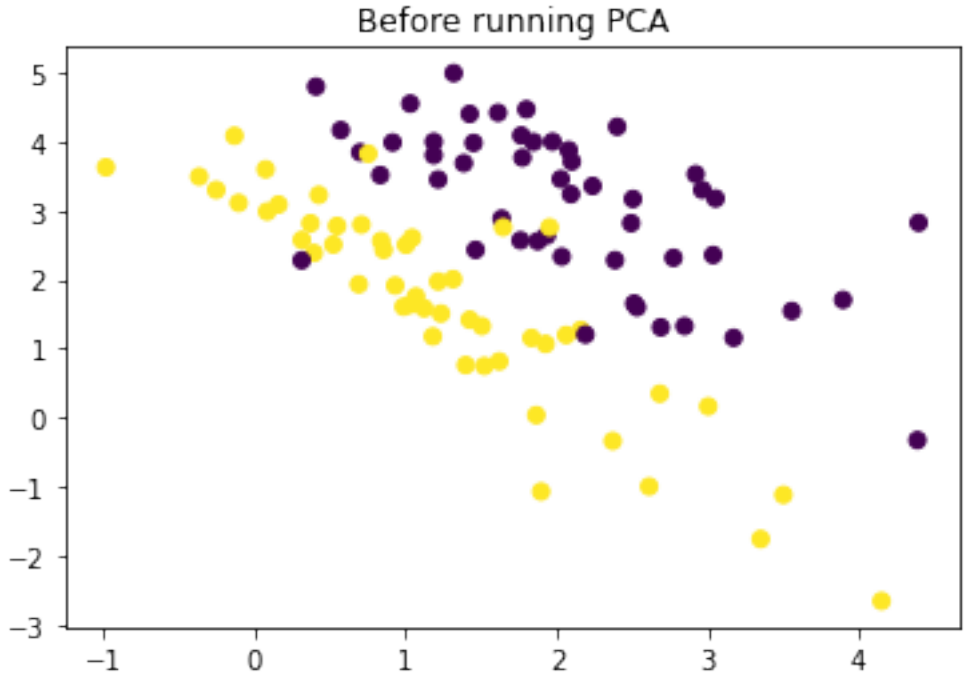
1.4.4 Running PCA

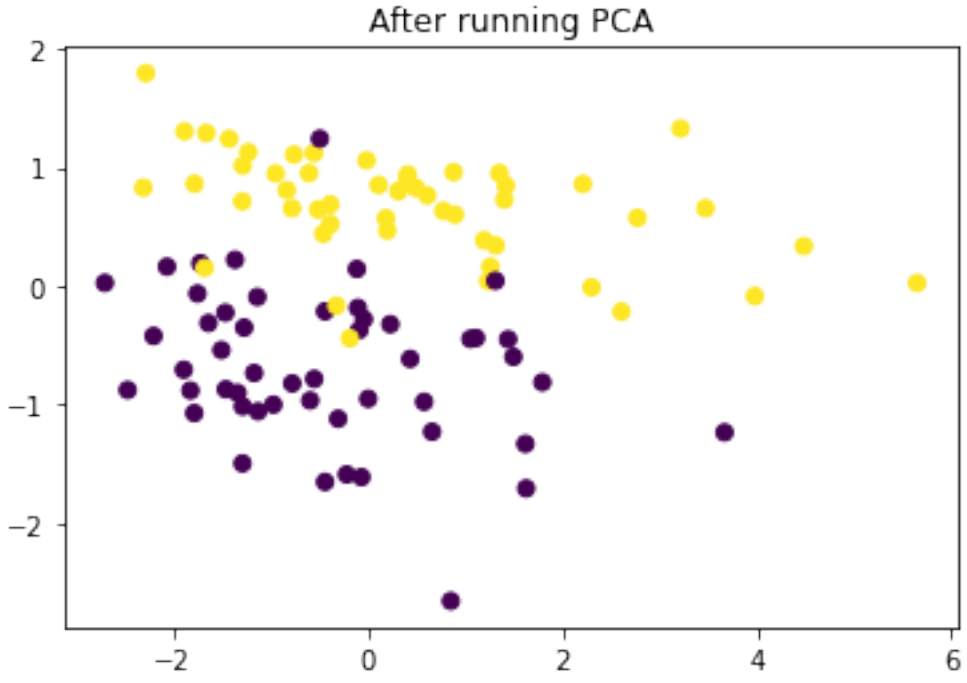
As before, process the data using the PCA algorithm and project it in one dimension. Plot the labeled data using *scatter()* before and after running PCA. Comment on the results.

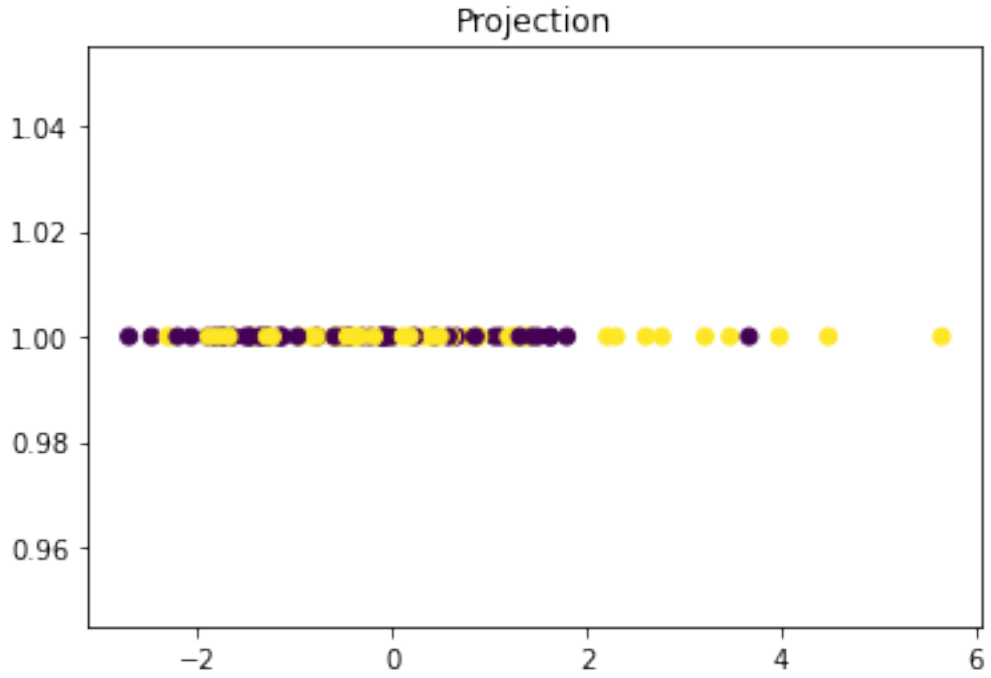
[29]:
```
plt.scatter(X[:,0],X[:,1],c=y[:,0])
plt.title("Before running PCA")

plt.figure()
_,P = pca(X,2)
plt.scatter(P[:,0],P[:,1],c=y[:,0])
plt.title("After running PCA")
plt.figure()
```

```
plt.scatter(P[:,0],np.ones(P.shape[0]),c=y[:,0])
plt.title("Projection")
```

[29]: Text(0.5, 1.0, 'Projection')







Comment: We can see that the final result differs from above since the points are spread out and have higher variance. The projection of the data looks more mixed and we do not have a clear separation point between the two variables.

How would the result change if you were to consider the second eigenvector? Or if you were to consider both eigenvectors?

Answer: If we had considered the second eigenvector we would have a less spread out data points, lower variance and potensially more errors by having the data points more closer to each other on the axis.

1.5 Case study 1: PCA for visualization

We now consider the *iris* dataset, a simple collection of data (N=150) describing iris flowers with four (M=4) features. The features are: Sepal Length, Sepal Width, Petal Length and Petal Width. Each sample has a label, identifying each flower as one of 3 possible types of iris: Setosa, Versicolour, and Virginica.

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

1.5.1 Loading the data

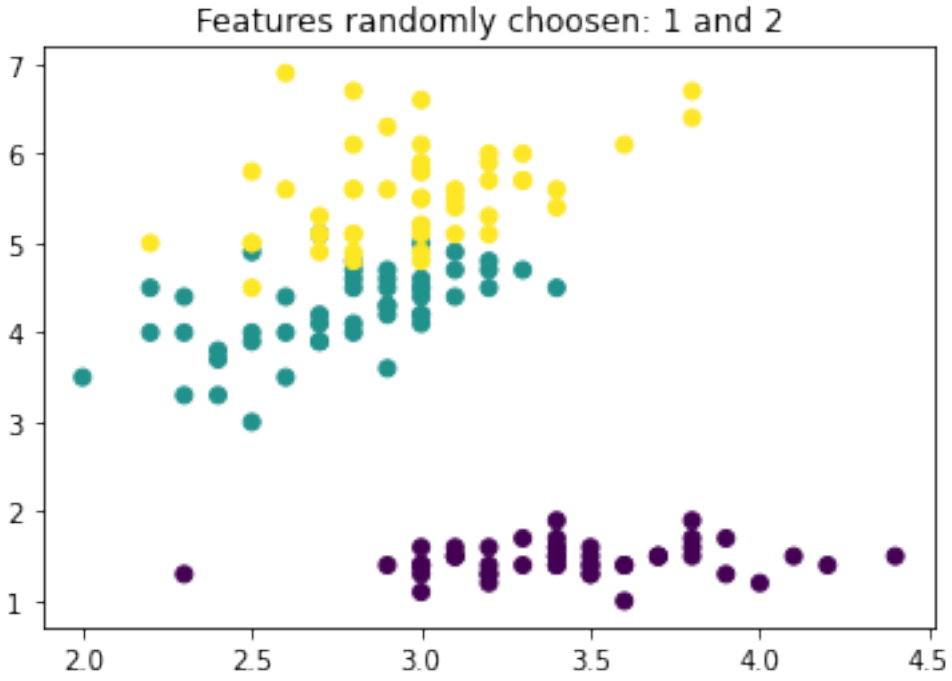
The function *get_iris_data()* from the module *syntethicdata* returns the *iris* dataset. It returns a data matrix of dimension [150x4] and a label vector of dimension [150].

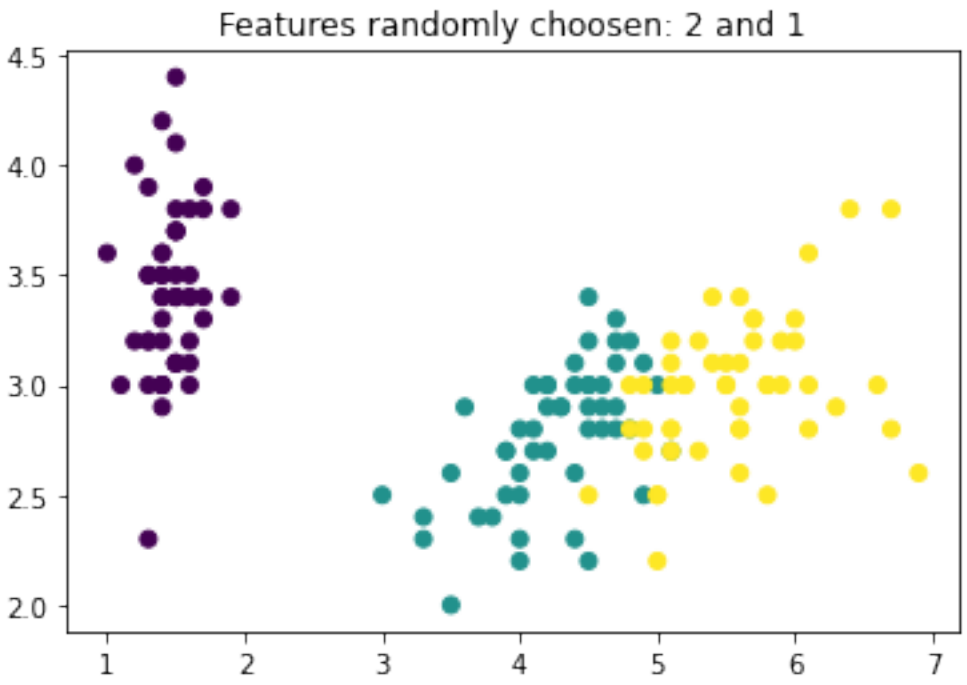
[30]:
```
X,y = syntheticdata.get_iris_data()
```

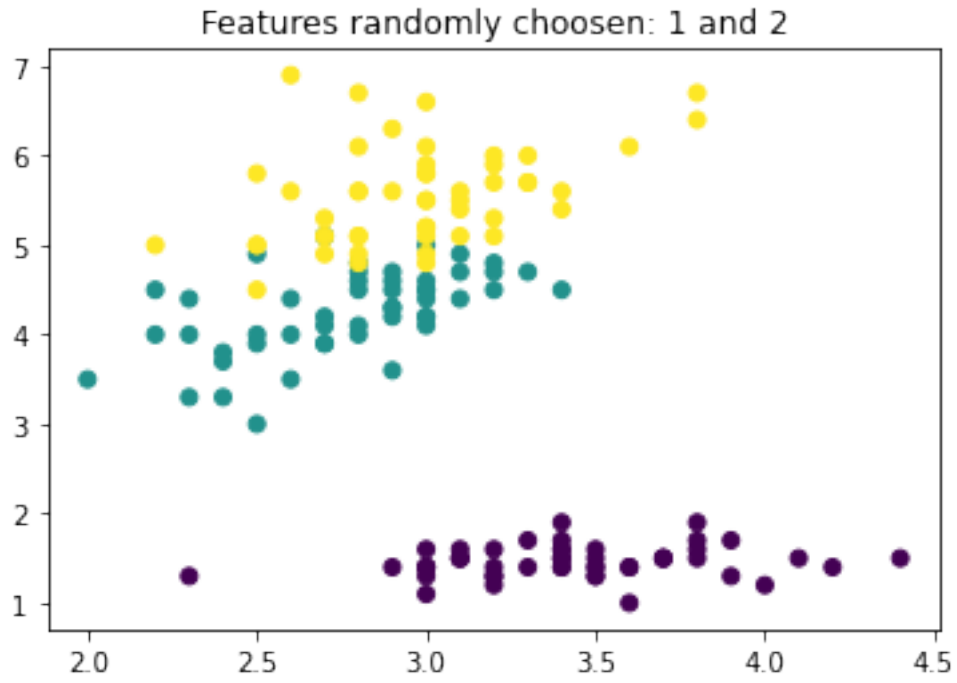
1.5.2 Visualizing the data by selecting features

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.

[34]:
```
import random as rd

for _ in range(3):
    #using random.sample to generate a list of 2 random numbers
    rand_num = rd.sample(range(0, 3), 2)
    rand_num1 = rand_num[0]
    rand_num2 = rand_num[1]

    plt.title(f'Features randomly choosen: {rand_num1} and {rand_num2}')
    plt.scatter(X[:,rand_num1],X[:,rand_num2],c=y)
    plt.figure()

#The plots are showing visualization of three pairs of randomly choosen␣
↪features:
```







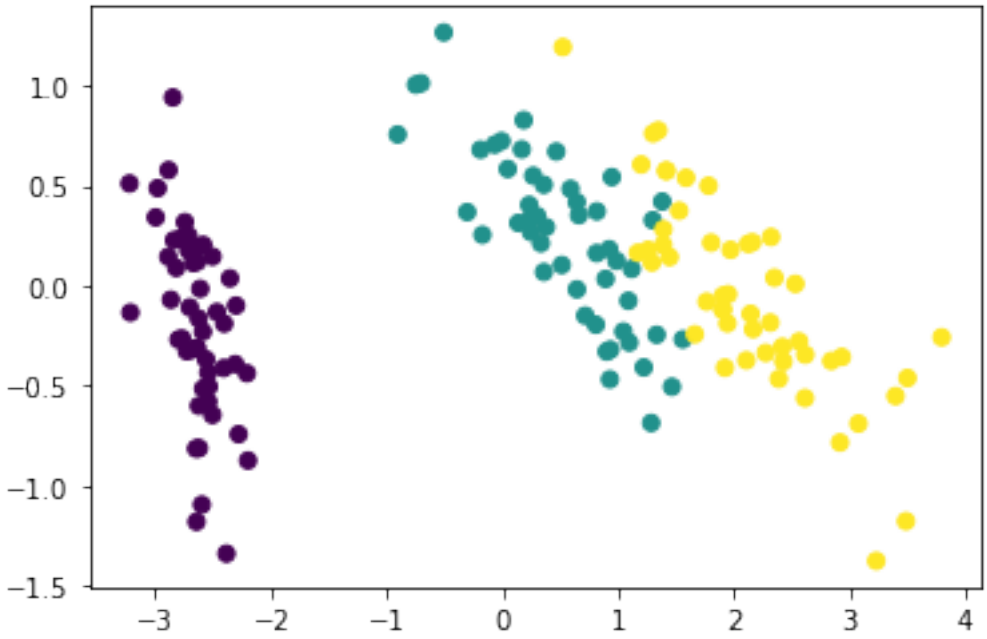

```
<Figure size 432x288 with 0 Axes>
```

1.5.3 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.

[35]:
```
pca_eigvec, P = pca(X,2)
plt.scatter(P[:,0],P[:,1],c=y)
```

[35]:
```
<matplotlib.collections.PathCollection at 0x7fb0f5468a90>
```



Comment: We can see that there is a clear separation layer between the three variables after running the PCA. The data has now got a higher variance and it is easier to seperate the variables.

1.6 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).

1.6.1 Loading the data

The function *get_lfw_data()* from the module *syntethicdata* 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 :)

[36]:
```
X,y,h,w = syntheticdata.get_lfw_data()
```

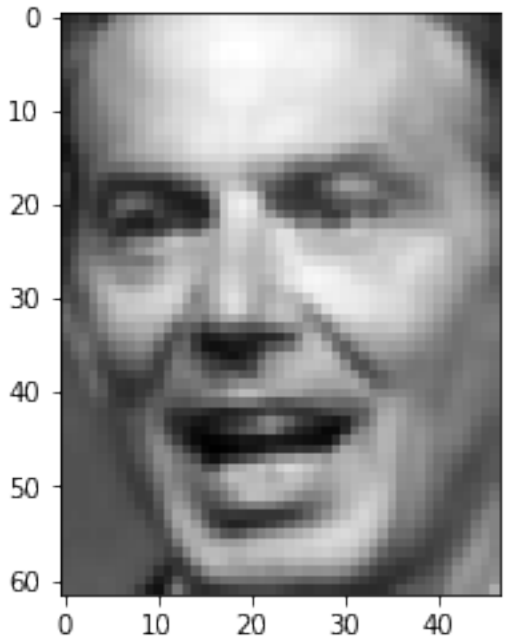
1.6.2 Inspecting the data

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 *imshow* receives as a first argument an image to be plot; 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.

[37]:
```
plt.imshow(X[1,:].reshape((h, w)), cmap=plt.cm.gray)
```

[37]:
```
<matplotlib.image.AxesImage at 0x7fb0f54c0310>
```



1.6.3 Implementing a compression-decompression function

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

Hint: Most of the code is the same as the previous PCA() function you implemented. You may want to refer to *Marsland* to check out how reconstruction is performed.

[38]:
```
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 features)


    #using the pca-function to get eigvec and P
    pca_eigvec, P = pca(A,m)
    Ahat = np.dot(P,np.transpose(pca_eigvec))

    return Ahat
```

1.6.4 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).

[39]:
```
Xhat = encode_decode_pca(X,200)
```

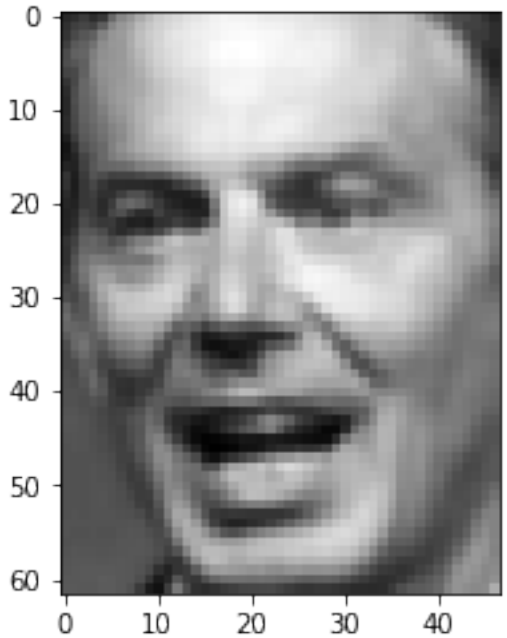
1.6.5 Inspecting the reconstructed data

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

[40]:
```
print("Original:")
plt.imshow(X[1,:].reshape((h, w)), cmap=plt.cm.gray)
```

```
Original:
```

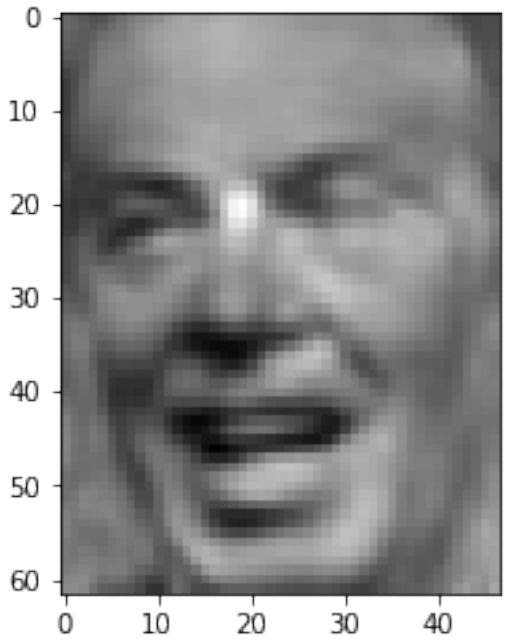
[40]:
```
<matplotlib.image.AxesImage at 0x7fb0f4b12d90>
```



[41]:
```
print("Reconstructed: ")
plt.imshow(Xhat[1,:].reshape((h, w)), cmap=plt.cm.gray)
```

```
Reconstructed:
```

[41]:
```
<matplotlib.image.AxesImage at 0x7fb0f4b0de10>
```



Comment: We can see that the reconstructed pictures is not exactly the same as the original, but it is very close. By looking at it, we can see that this image originates from the first image.

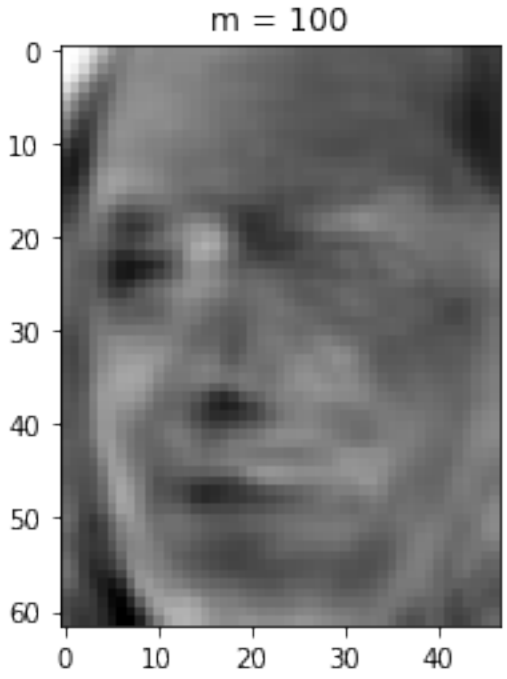
1.6.6 Evaluating different compressions

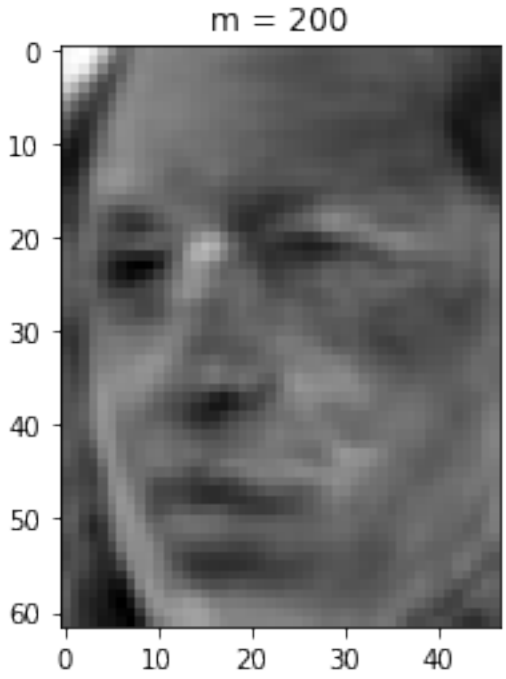
Use the previous setup to generate compressed images using different values of low dimensions in the PCA algorithm (e.g.: 100, 200, 500, 1000). Plot and comment on the results.

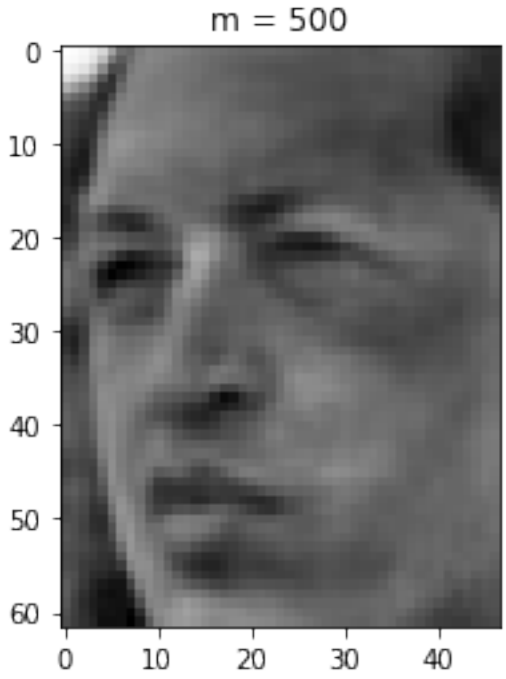
[44]:
```
plt.imshow(X[0,:].reshape((h, w)), cmap=plt.cm.gray)
plt.title("Original")

print("Plotting the reconstructed images: ")
m_val = [100,200,500,1000,1500]
for m in m_val:
    plt.figure()
    Xhat = encode_decode_pca(X,m)
    plt.title(f'm = {m}')
```

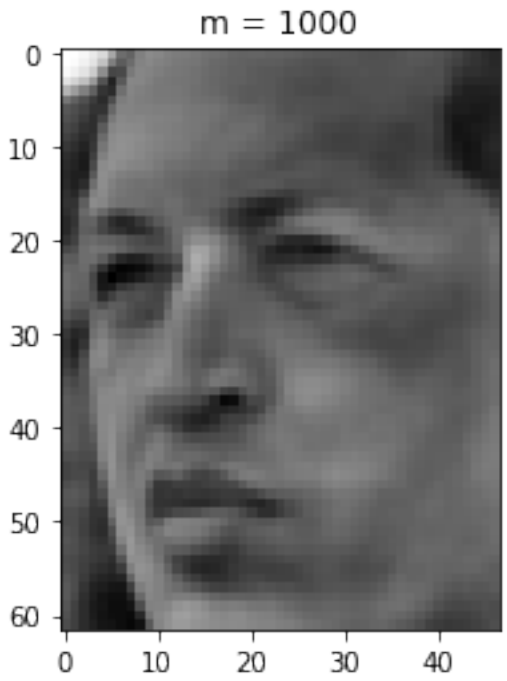
```
    plt.imshow(Xhat[0,:].reshape((h, w)), cmap=plt.cm.gray)
```

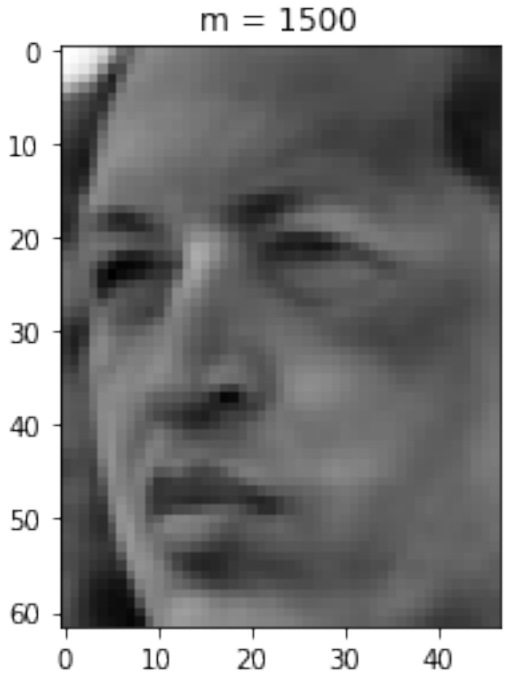
Plotting the reconstructed images:




m = 200
0
10
20
30
40
50
60
0
10
20
30
40


m = 500
0
10
20
30
40
50
60
0
10
20
30
40





Comment: The plots above show the reconstructed for different values of low dimensions in the PCA algorithm. We can see that the pictures improves as increasing the m-value until m=1000.

For m = 1000 and m =1500, it comes close to the original pictures but not exactly the same. I tried to run the the algorithm for bigger m values but did not get a better result.

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 assesment of the results.

2.0.1 Importing scikit-learn library

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

[45]:
```
from sklearn.cluster import KMeans
```

2.0.2 Loading the data

We will use once again the *iris* data set. The function *get_iris_data()* from the module *syntethicdata* returns the *iris* dataset. It returns a data matrix of dimension [150x4] and a label vector of dimension [150].

[46]:
```
X,y = syntheticdata.get_iris_data()
```

2.0.3 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. However, we use PCA, as this will allow for an easy visualization.

[47]:
```
_,P = pca(X,2)
```

2.0.4 Running k-means

We will now consider the *iris* data set as an unlabeled set, and perform clustering to this unlabeled set. We can compare the results of the clustering to the lableled calsses.

Use the class *KMeans* to fit and predict the output of the *k-means* algorithm on the projected data. Run the algorithm using the following values of $k = \{2, 3, 4, 5\}$.

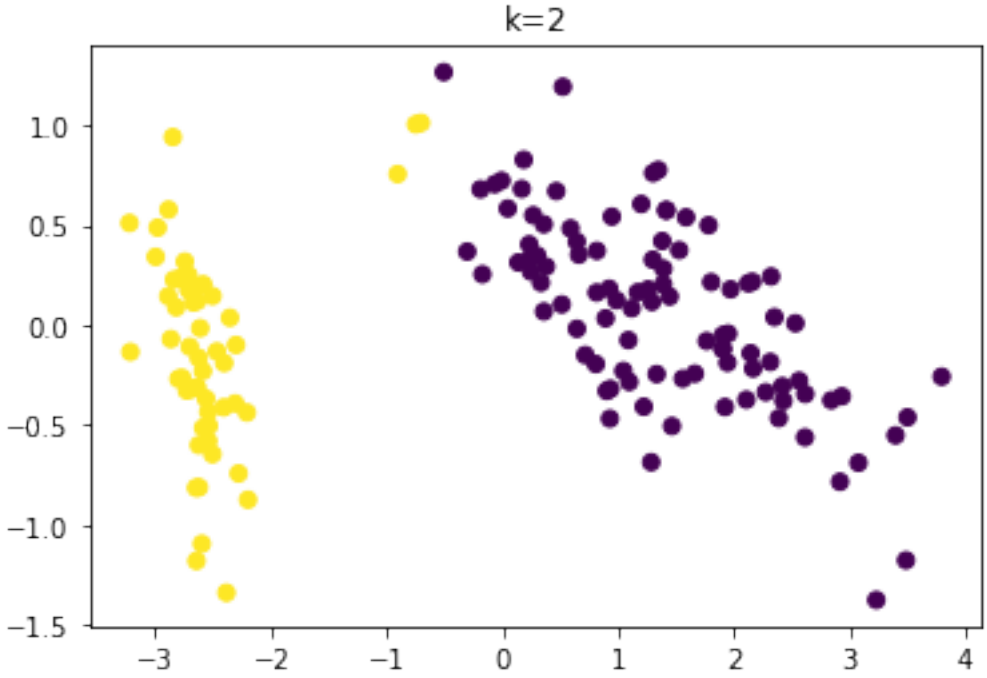
[48]:
```
yhat2 = []
for k in range(2,6): #running k-means algorithm for k = {2,3,4,5}
    KM = KMeans(k)
    yhat2.append(KM.fit_predict(P))
```

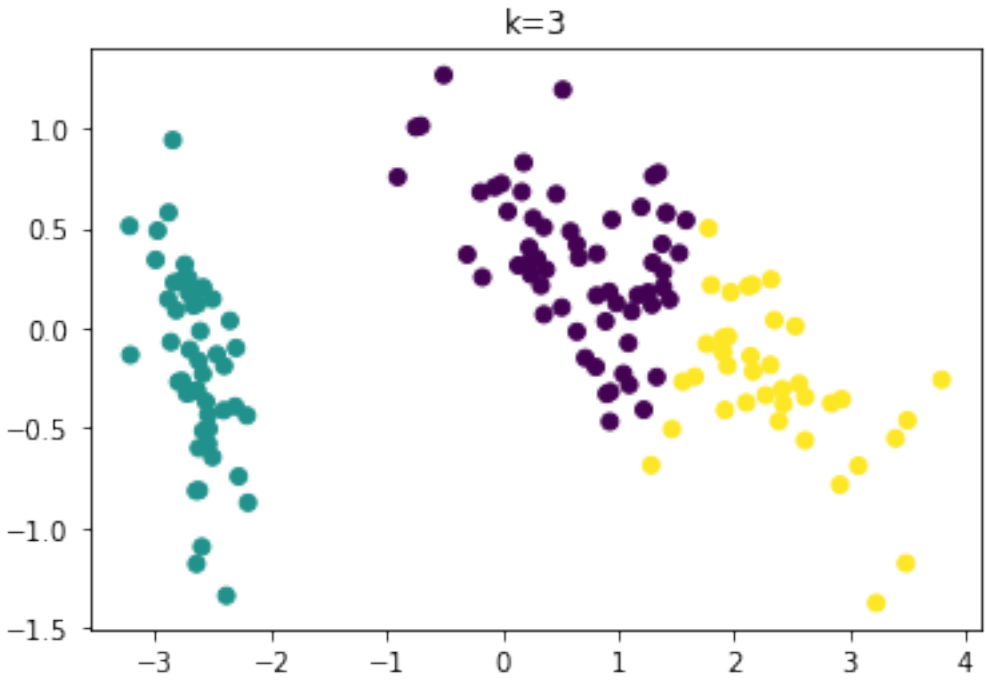
2.0.5 Qualitative assessment

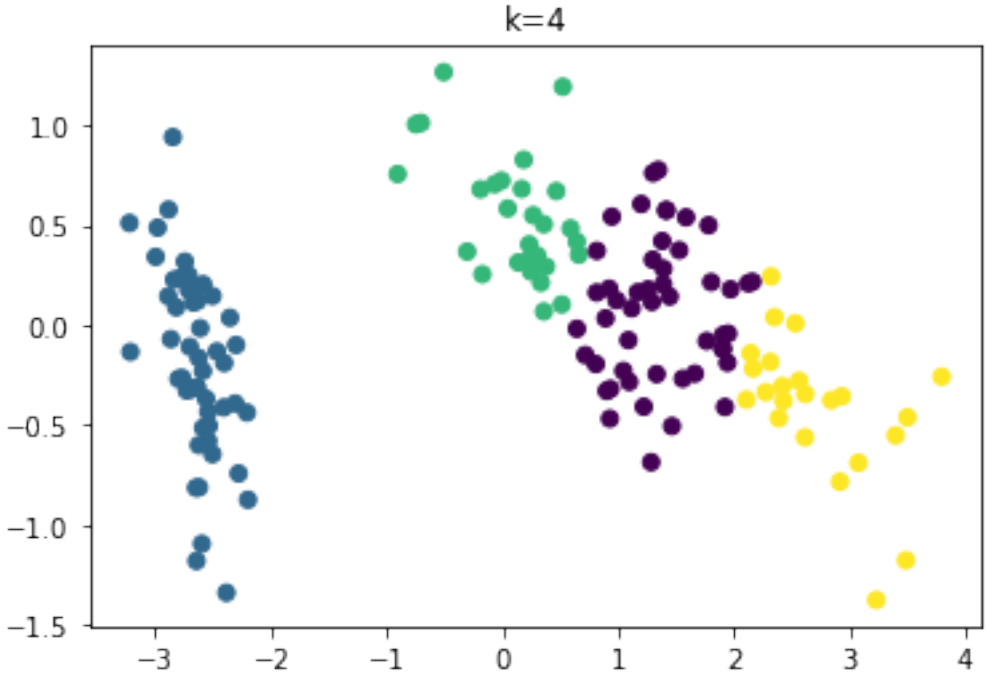
Plot the results of running the k-means algorithm, compare with the true labels, and comment.

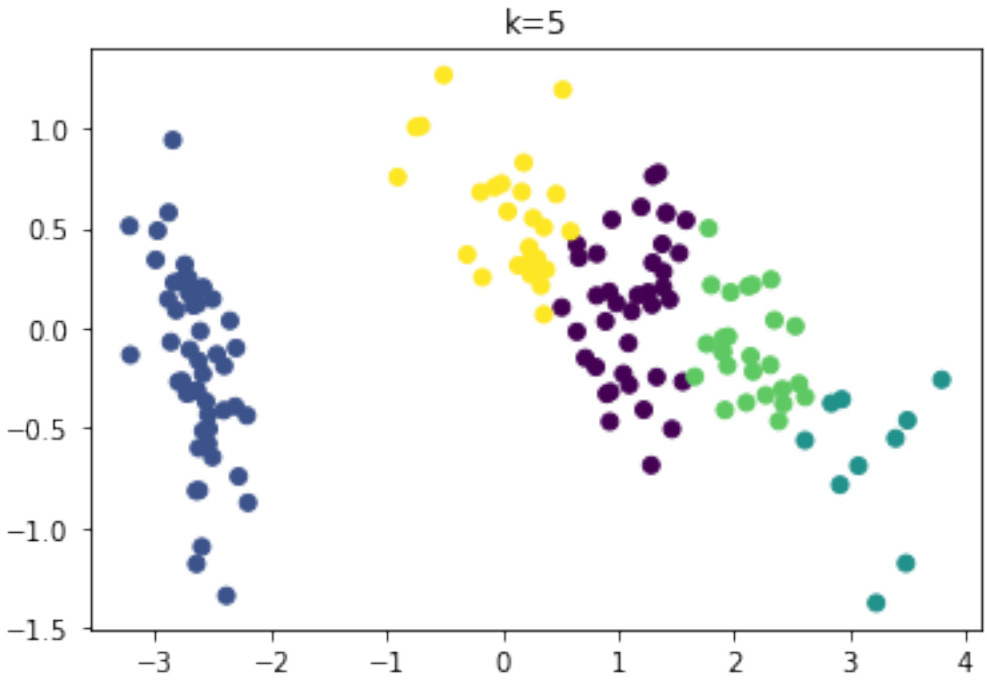
[49]:
```
for i in range(len(yhat2)):
    plt.scatter(P[:,0],P[:,1],c=yhat2[i])
    plt.title(f'k={i +2}')
    plt.figure()

plt.scatter(P[:,0],P[:,1],c=y)
plt.title('Original data')
```

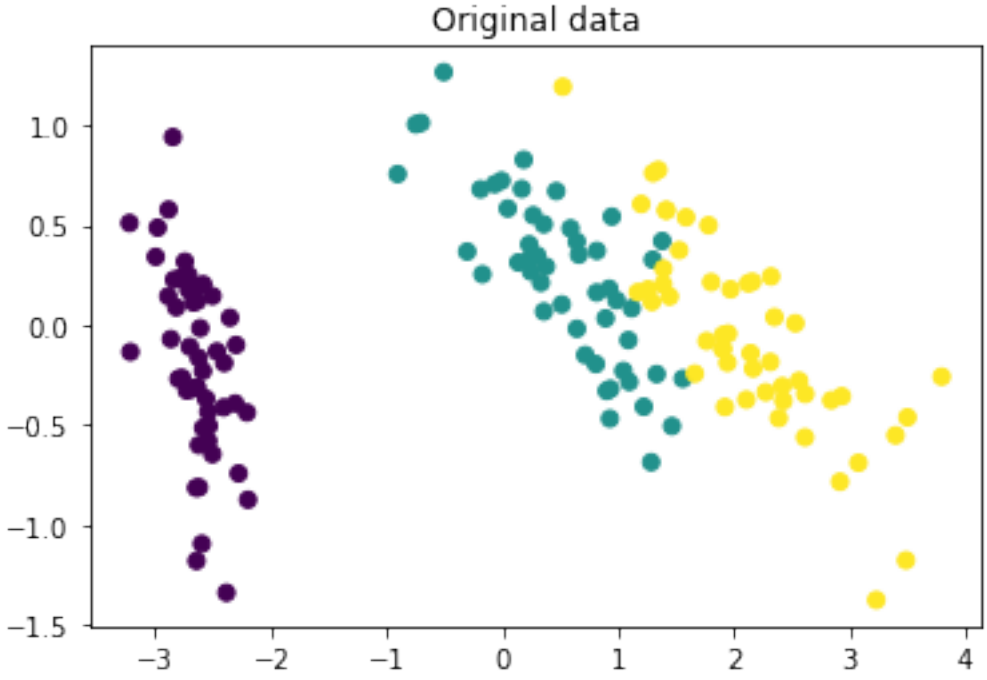
[49]: Text(0.5, 1.0, 'Original data')




k=3
1.0
0.5
0.0
−0.5
−1.0
−1.5
−3
−2
−1
0
1
2
3
4


k=4
1.0
0.5
0.0
−0.5
−1.0
−1.5
−3
−2
−1
0
1
2
3
4





Comment: With different k-values the k-means algorithm divid the data into k groups. In this case, we can see that the plot with the k = 3 comes very close to the original data. However it is

difficult to distinguish between the labels in the plot for k=3 and the original data.

3 Quantitative Assessment of K-Means (Bachelor and master students)

We used k-means for clustering and we assessed the results qualitatively by visualizing them. However, we often want to be able to measure in a quantitative way how good the clustering was. To do this, we will use a classification task to evaluate numerically the goodness of the representation learned via k-means.

Reload the *iris* dataset. Import a standard `LogisticRegression` classifier from the module `sklearn.linear_model`. Use the k-means representations learned previously (`yhat2,...,yhat5`) 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.

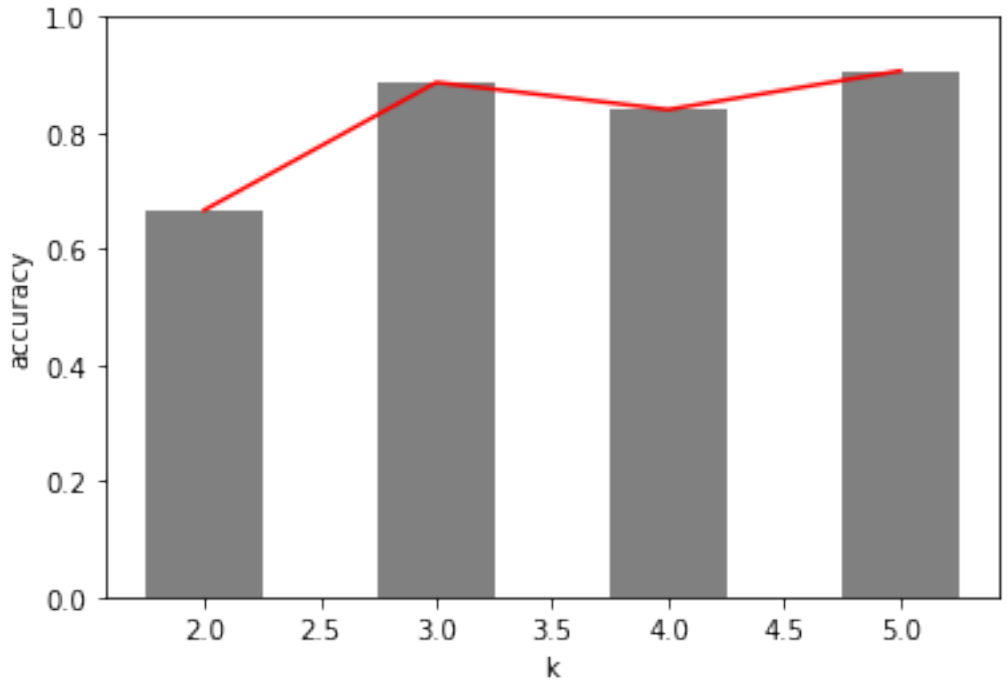
- Train a Logistic regression model using the first two dimensions of the PCA of the iris data set as input, and the true classes as targets.
- Report the model fit/accuracy on the training set.
- For each value of K:
  - One-Hot-Encode the classes outputed by the K-means algorithm.
  - Train a Logistic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

[64]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import OneHotEncoder
from sklearn import metrics

#reloading the iris dataset:
X,y = syntheticdata.get_iris_data()

#getting the first two dimensions
_,P = pca(X,2)

#training and predicting the score():
logReg = LogisticRegression().fit(P,y)
accuracy = logReg.score(P,y)
print(f'Accuracy = {accuracy:4.4f}')

accuracies = []
k = []
for i in range(2,6):
    #predicting yhat and reshaping it to a nested list
    yhat = KMeans(i).fit_predict(P).reshape(-1,1)
    #using OneHotEncoder() to transform from [0],[1] ... -> [1 0],[0 1]
```

```
    new_yhat = OneHotEncoder().fit_transform(yhat).toarray()

    #training with logistic
    logReg = LogisticRegression().fit(new_yhat,y)
    predict = logReg.predict(new_yhat)
    accuracy = metrics.accuracy_score(y,predict)
    print(f'k = {i} | accuracy = {accuracy:4.4f} ')

    accuracies.append(accuracy)
    k.append(i)


plt.bar(k,accuracies,width=0.5,color = 'grey')
plt.plot(k,accuracies,color= 'red')
plt.ylim([0,1])
plt.xlabel("k")
plt.ylabel("accuracy")
```

```
Accuracy = 0.9667
k = 2 | accuracy = 0.6667
k = 3 | accuracy = 0.8867
k = 4 | accuracy = 0.8400
k = 5 | accuracy = 0.9067
```

[64]: Text(0, 0.5, 'accuracy')



Comment: After runing the algorithm for different k-values, we can see that the very best accuracy was founded to be at k = 5 and the second best was at k = 3. Generally, one could expect that the accuracy would increase with increased k. But in this case it does not hold up for k = 4. Other than that, the algorithm gives good result.

4 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 run the algorithm on synthetic data in order to see its working and evaluate when it may make sense to use it and when not. We then considered two typical uses of PCA: for **visualization** on the *iris* dataset, and for **compression-decompression** on the *lfw* dateset.

We then moved to consider the k-means algorithm. In this case we used the implementation provided by *scikit-learn* and we applied it to another prototypical unsupervised learning problem: **clustering**; we used *k-means* to process the *iris* dataset and we evaluated the results visually.

In the final part, we considered two additional questions that may arise when using the above algorithms. For PCA, we considered the problem of **selection of hyper-parameters**, that is, how we can select the hyper-parameter of ou algorithm in a reasonable fashion. For k-means, we considered the problem of the **quantitative evaluation** of our results, that is, how can we measure the performance or usefulness of our algorithms.