IN3050_Assignment_3

April 23, 2021

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

Name: Anders Vestengen

Username: andergv

1.0.1 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/about/regulations/studies/studies-examinations/routines-cheating.html By submitting this assignment, you confirm that you are familiar with the rules and the consequences of breaking them.

1.0.2 Delivery

Deadline: Friday, April 23, 2021, 23:59

Your submission should be delivered in Devilry. You may redeliver in Devilry before the deadline, but include all files in the last delivery, as only the last delivery will be read. You are recommended to upload preliminary versions hours (or days) before the final deadline.

1.0.3 What to deliver?

You are recommended to solve the exercise in a Jupyter notebook, but you might solve it in a Python program if you prefer.

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 should make a pdf of your solution which shows the results of the runs.

If you prefer not to use notebooks, you should deliver the code, your run results, and a pdf-report where you answer all the questions and explain your work.

Your report/notebook should contain your name and username.

Deliver one single zipped 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.

1.0.4 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.

1.0.5 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.

1.0.6 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) Variance is the measure of how spread the points of a dataset is, in other words, how much does every point differ from the mean.

2) Covariance is the variance, but now generalized to two variables, so we can then look at how those two vary in comparison to eachother and the mean.

3) According to the Week 11 lecture slides, the covariance matrix is: C = (1/N * X.T * X), where N is the number of samples, and X is the input data.

4) It is the direction of which PCA assumes most of the underlying structure of information is found, and so the algorithm will find the eigenvectors along this path (through the size of their eigenvalues), and prioritize these when doing dimensional reduction.

5) This is the operating assumption of the PCA, it could therefore not work the same way without it. In the same vein, K-mean clustering assumes that points close to eachother hint at the underlying structure of information in the data. Without this assumption the clustering algorithm could not work the way it does.

6) The principle applies well to some problems and worse to others. I would say PCA works well within a defined problem space. I don't believe there normally is a place it absolutely does or does not apply, and since we don't have predetermined labels for our data there's always a gradient of success.

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*.

[25]:

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

    return A
```

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

[26]:

```
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)
print("yay we passed!") #If the assert fails this shouldn't execute
```

```
yay we passed!
```

1.2.3 Computing Covariance Matrix

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

[27]:

```
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*:

[90]:

```
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)
print("yay we passed!") #If the assert fails this shouldn't execute
```

```
yay we passed!
```

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!

[29]:

```
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)
    #indices = np.argsort(evals)
    #indices = indices[::-1]
    #evecs = evecs[:,indices]
    #evals = evals[indices]


    # 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*:

[91]:

```
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)
print("yay we passed!") #If the assert fails this shouldn't execute
```

```
yay we passed!
```

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]*.

[93]:

```
def sort_eigenvalue_eigenvectors(evals, evecs):
    # 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

    #based on Marslands algorithm p.137
    indices = np.argsort(evals)
    indices = indices[::-1]
    evecs = evecs[:,indices]
    evals = evals[indices]
    for i in range(np.shape(evecs)[1]):
        evecs[:,i] / np.linalg.norm(evecs[:,i] * np.sqrt(evals[i]))


    sorted_eigval = evals
    sorted_eigvec = evecs

    return sorted_eigval, sorted_eigvec
```

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

[94]:

```
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)
print("yay we passed!") #If the assert fails this shouldn't execute
```

```
yay we passed!
```

1.2.6 PCA Algorithm

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

[4]:

```
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)

    #pca_eigvec = None
    #data centering

    #Really nothing special. Again, based on Marsland p.137
    norm = np.mean(A, axis=0)
    A -= norm

    # Covariance matrix
    C = np.cov(np.transpose(A))

    #Computing eigenvalues and -vector

    evals, evecs = np.linalg.eig(C)
    evals = evals.real
    evecs = evecs.real

    indices = np.argsort(evals)
    indices = indices[::-1][:m] # dimensionality reduction here
    evecs = evecs[:,indices]
    evals = evals[indices]
    for i in range(np.shape(evecs)[1]):
        evecs[:,i] / np.linalg.norm(evecs[:,i]) * np.sqrt(evals[i])

    x = np.dot(np.transpose(evecs), np.transpose(A))

    y = np.transpose(np.dot(evecs, x))+norm
    P = x
    pca_eigvec = evecs
    return pca_eigvec, P.T
```

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

[305]:

```
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)
print("yay we passed!") #If the assert fails this shouldn't execute # this is␣
 ↪an important part
```

```
yay we passed!
```

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).

[82]:

```
X = syntheticdata.get_synthetic_data1()
```

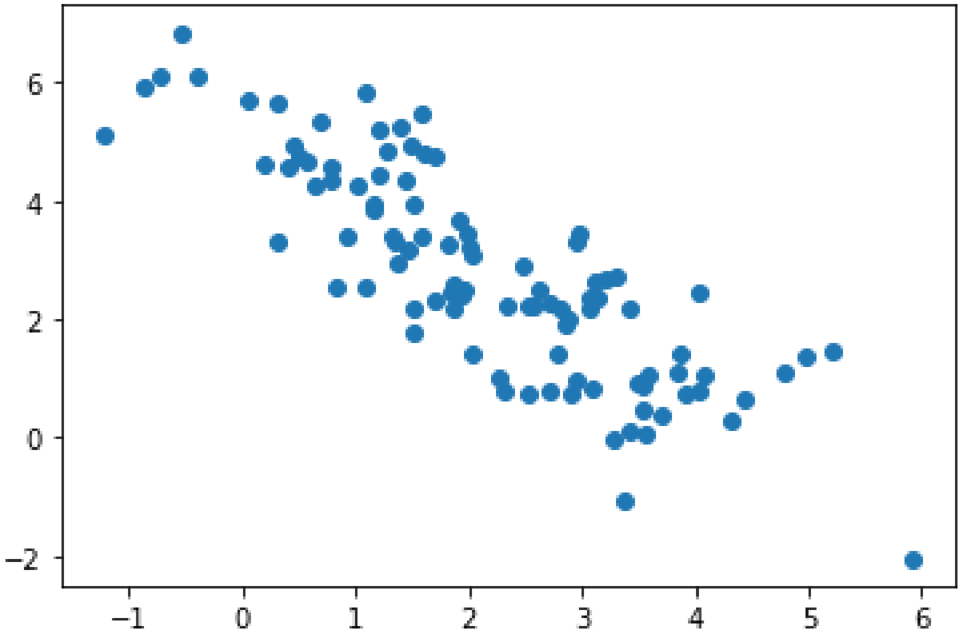
1.3.2 Visualizing the data

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

[83]:

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

[83]: <matplotlib.collections.PathCollection at 0x7f72bd665c50>

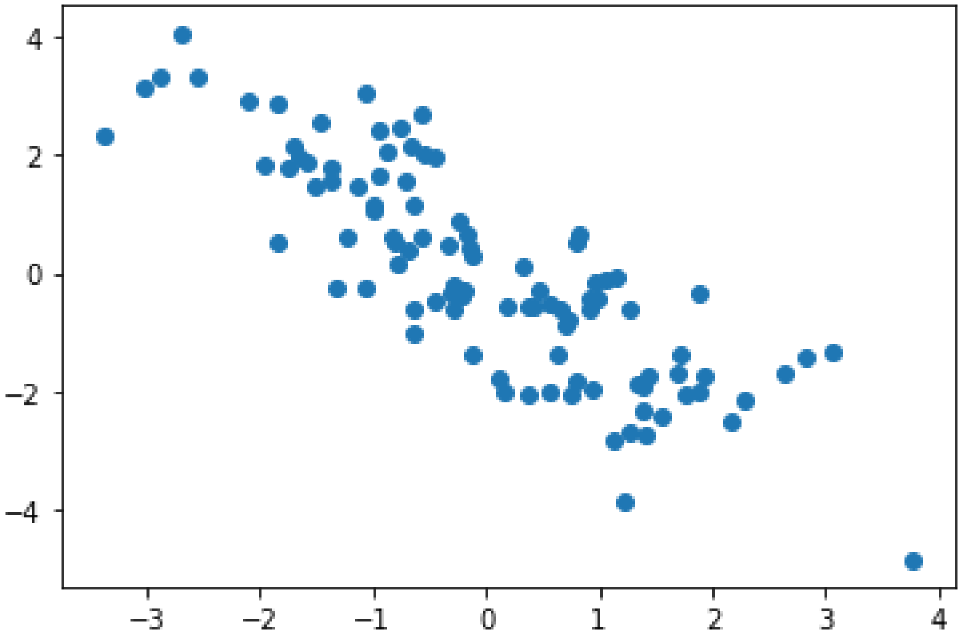


1.3.3 Visualize the centered data

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.

[84]:
```
norm = np.mean(X, axis=0)
X -= norm
plt.scatter(X[:,0],X[:,1])
```

[84]: <matplotlib.collections.PathCollection at 0x7f72bd498588>

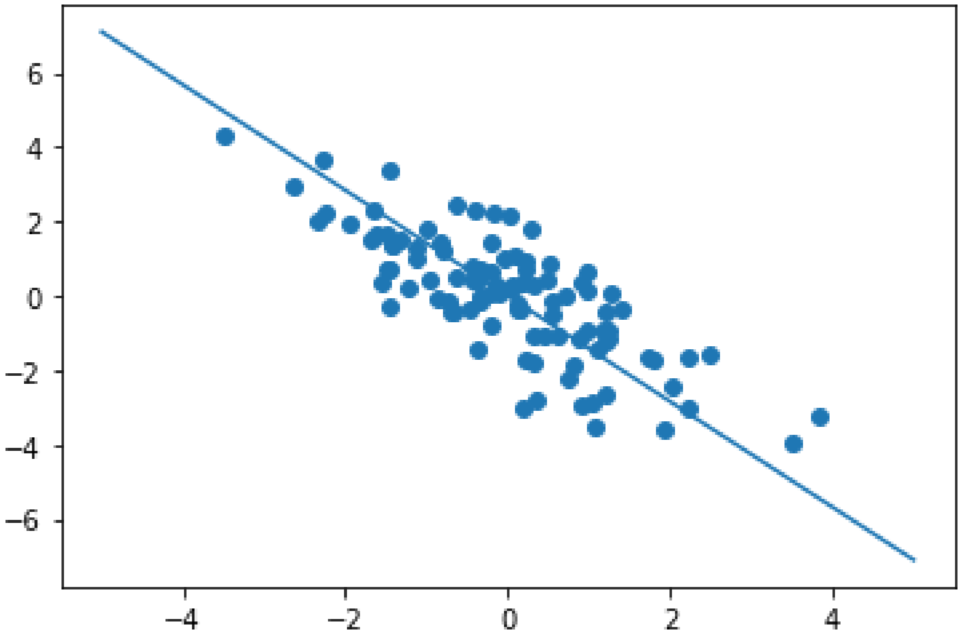


1.3.4 Visualize the first eigenvector

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.

[295]:
```
X = syntheticdata.get_synthetic_data1()
pca_eigvec, Y = 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)
```

[295]:
```
[<matplotlib.lines.Line2D at 0x7fbc1ddb1898>]
```

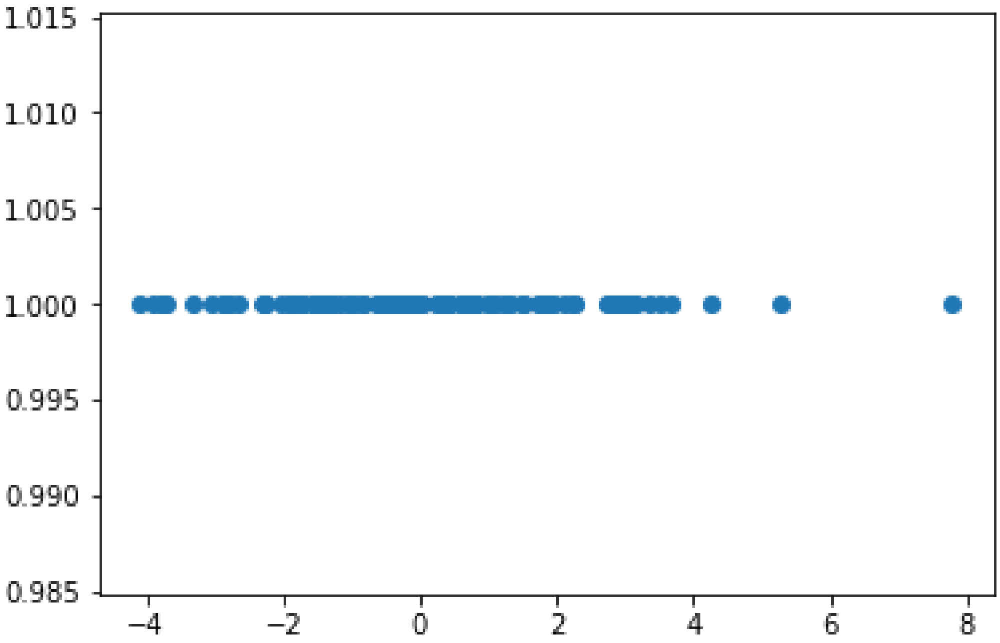


1.3.5 Visualize the PCA projection

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

```
[142]: X = syntheticdata.get_synthetic_data1()
       pca_eigvec, Y = pca(X,1)
       plt.scatter(Y,np.ones(Y.shape[0]))
```

```
[142]: <matplotlib.collections.PathCollection at 0x7f3595ae93c8>
```



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 preprocessing 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.

```
[179]: X,y = syntheticdata.get_synthetic_data_with_labels1()
       print(X.shape)
       print(y.shape)
```

```
(100, 2)
(100, 1)
```

1.4.2 Running PCA

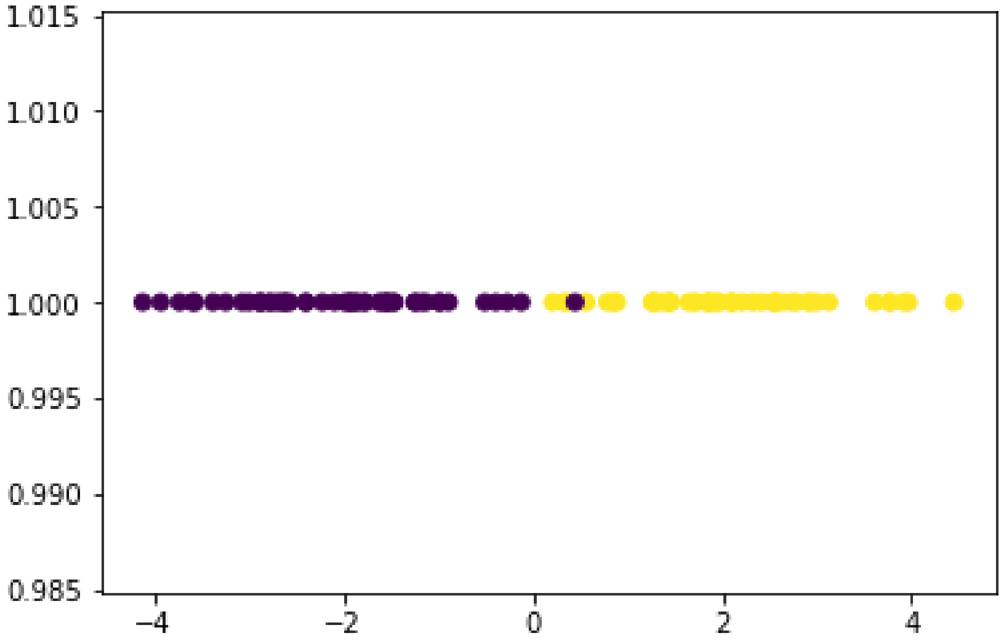
Process the data using the PCA algorithm and project it in one dimension. Plot the labeled data using *scatter()* before and after running PCA. Comment on the results.

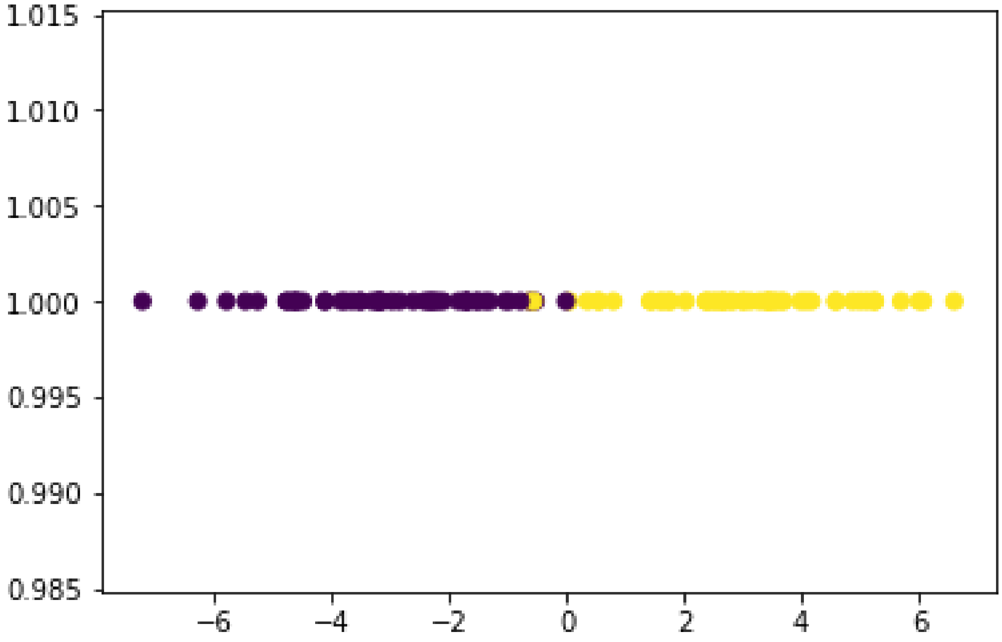
[184]:

```
pca_eigvec, Y = pca(X,1)
plt.scatter(X[:,0],np.ones(X.shape[0]),c=y[:,0])
plt.figure()
plt.scatter(Y,np.ones(Y.shape[0]),c=y[:,0])
```

[184]:

```
<matplotlib.collections.PathCollection at 0x7f35954e21d0>
```





Comment: Enter your comment here.

So, after running through the PCA the data is now bigger. Another observation is the way the data seems more pushed into the center, although it is hard to tell what is the effect of dimensionality reduction, and what is the effect of the scaling. I guess this is what they mean by X^, as its pretty much the input X, but there is also some difference, especially in the center

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.

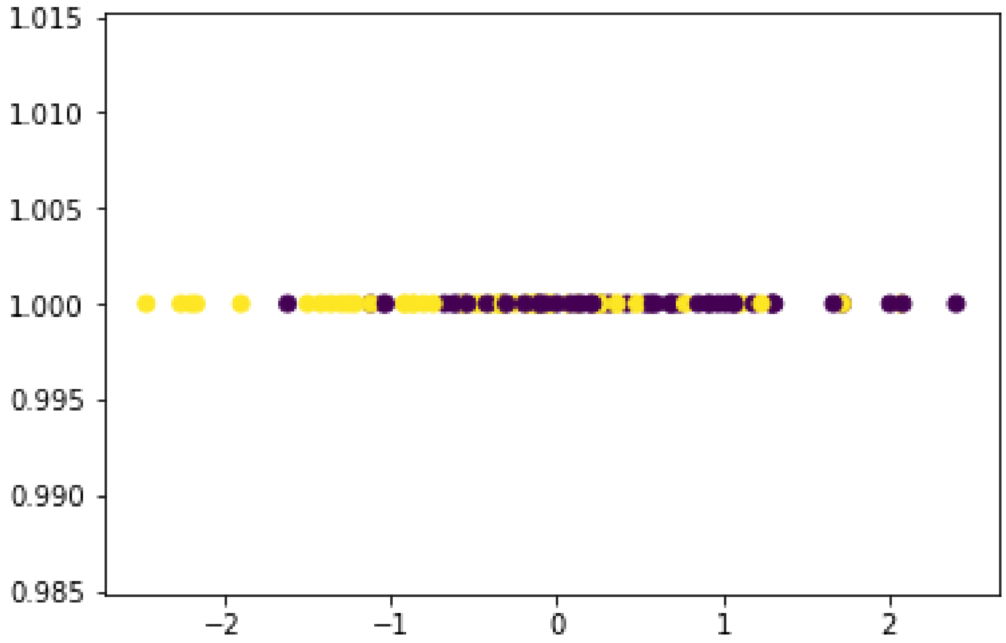
[292]:
```
X,y = syntheticdata.get_synthetic_data_with_labels2()
```

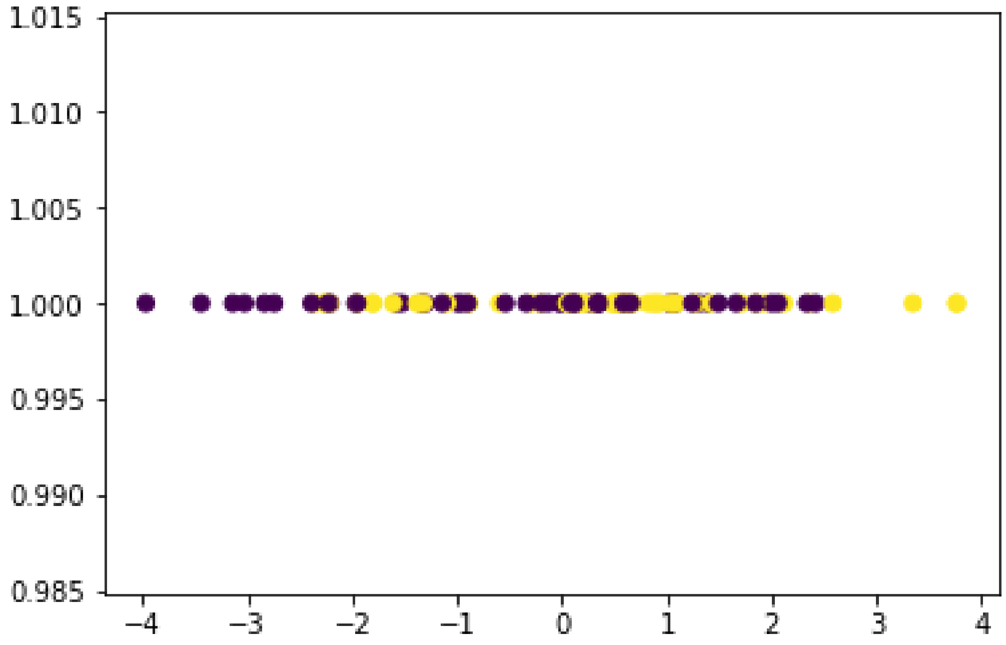
1.4.4 Running PCA

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.

[293]:
```
X,y = syntheticdata.get_synthetic_data_with_labels2()
pca_eigvec, Y = pca(X,1)
plt.scatter(X[:,0],np.ones(X.shape[0]),c=y[:,0])
plt.figure()
plt.scatter(Y,np.ones(Y.shape[0]),c=y[:,0])
```

[293]:
```
<matplotlib.collections.PathCollection at 0x7fbc1d725f60>
```





Comment: Enter your comment here.

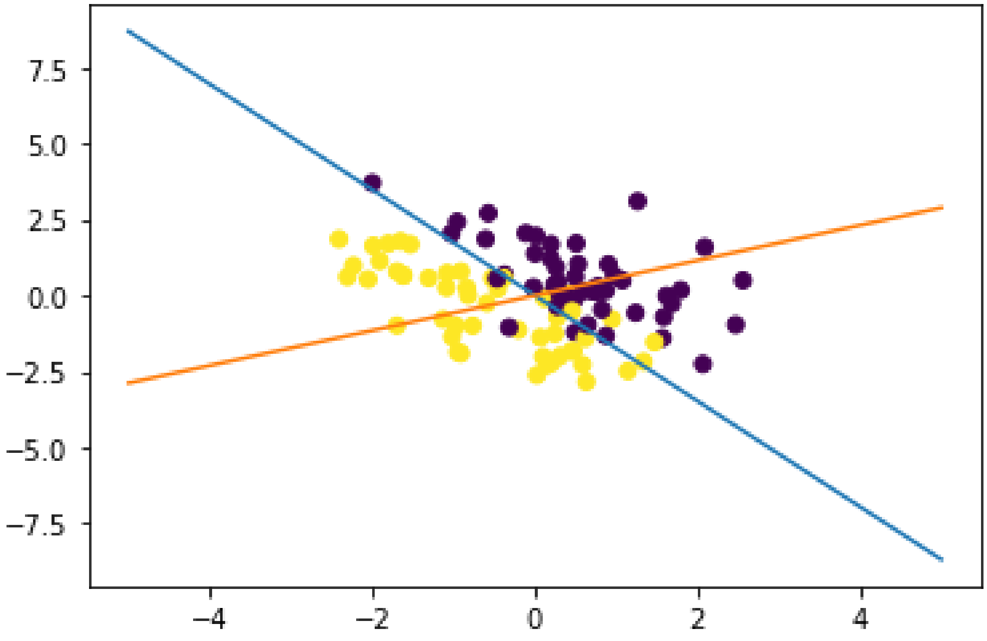
This is comparably worse than the prievious example. here the labels get mixed alot more. There is also some difference in the relative position, but then again the first example is not centered like it is after PCA.

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

[294]:

```
X,y = syntheticdata.get_synthetic_data_with_labels2()
pca_eigvec, Y = pca(X,2)
print(pca_eigvec)
plt.scatter(X[:,0],X[:,1],c=y[:,0])
x = np.linspace(-5, 5, 1000)
y = pca_eigvec[0,1]/pca_eigvec[0,0] * x
plt.plot(x,y)
y = pca_eigvec[1,1]/pca_eigvec[1,0] * x
plt.plot(x,y)
plt.show()
print("normal?:", 'Yes' if np.dot(pca_eigvec[0], pca_eigvec[1]) == 0 else 'No')
```

```
[[ 0.49913656 -0.86652334]
 [-0.86652334 -0.49913656]]
```



```
normal?: Yes
```

Answer: Well yes. Both eigenvectors will contain some information related to the space. This is very evident in the second set of data I plottet in one dimension above. In the first example the PCA handles the throughput just fine with only small differences between the input and output, while in the last example that has a second labeled set the PCA loses alot more information, and there is an immediate difference between the two examples.

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].

[303]:

```
X,y = syntheticdata.get_iris_data()
X.shape
```

[303]:

```
(150, 4)
```

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.

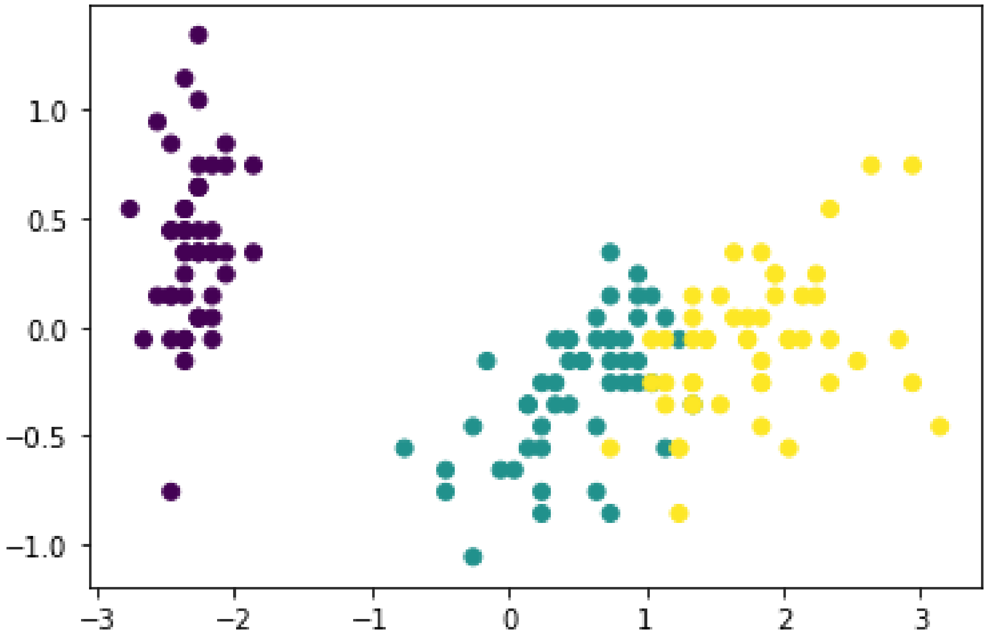
[24]:

```
eigvecs, output = pca(X,2)
#randoms = np.random.randint(4, size=2)
randoms = np.random.choice(range(4), 2, replace=False)
print(randoms)
plt.scatter(X[:,randoms[0]], X[:,randoms[1]],c=y)
```

```
[2 1]
```

[24]:

```
<matplotlib.collections.PathCollection at 0x7fbc2071d0f0>
```



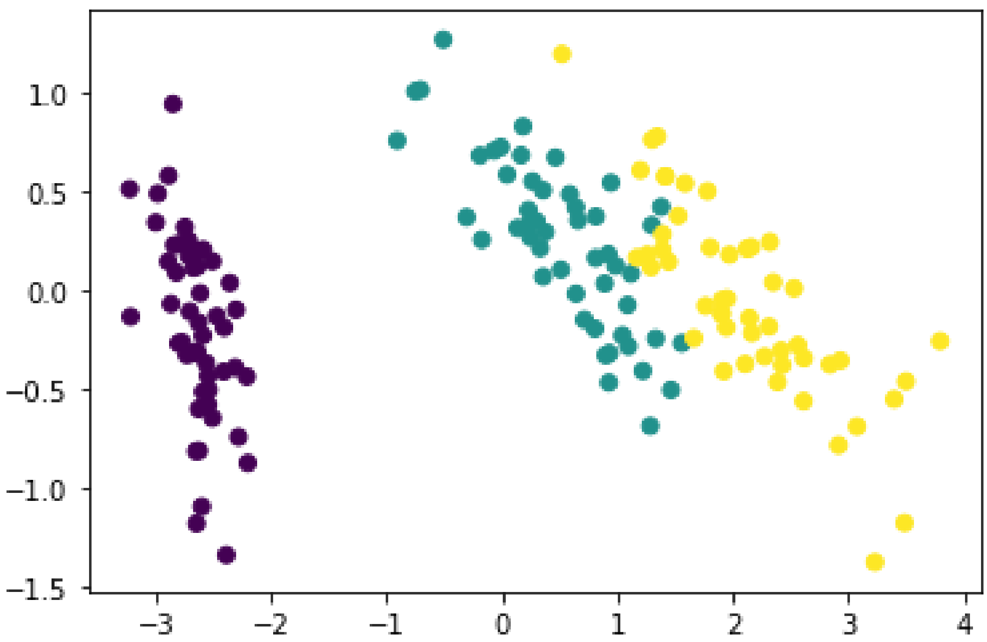
1.5.3 Visualizing the data by PCA

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

[304]:
```
eigvecs, output = pca(X,2)
print(y.shape)
plt.scatter(output[:,0],output[:,1],c=y)
```

```
(150,)
```

[304]:
```
<matplotlib.collections.PathCollection at 0x7fbc1d287898>
```



Comment:

After trying a few different pairs of features I think the PCA comes very close to replicating some of them, but there seems to be something about the placement that's lost throught PCA. In the original all samples across labels are placed in a sort of barcode-fashion. I don't know if its relevant, because it might be rightfully noise, but its interesting to see it dissapear. Otherwise PCA seems to do a fairly good job in the x-axis, but there's a stark change in the relative y-axis compared to the original.

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 :)

[318]:
```
X,y,h,w = syntheticdata.get_lfw_data()
```

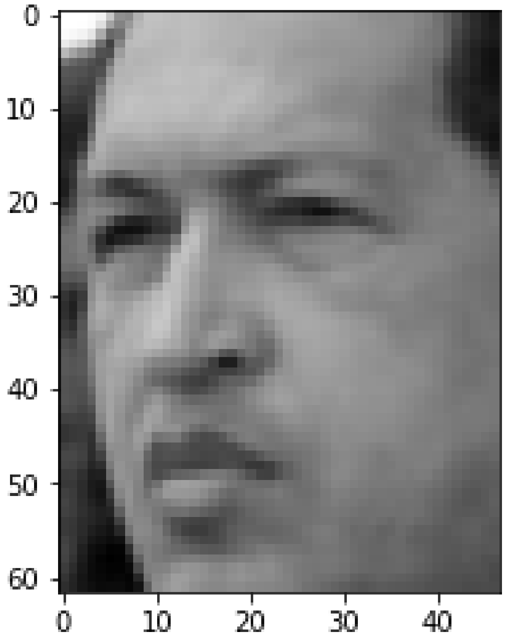
1.6.2 Inspecting the data

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.

[309]:
```
plt.imshow(X[0,:].reshape((h, w)), cmap=plt.cm.gray)
```

[309]:
```
<matplotlib.image.AxesImage at 0x7fbc1d5d65f8>
```



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.

[314]:
```
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:
    # pca_eigvec     [Mxm] numpy matrix containing the eigenvectors (M␣
↪dimensions, m eigenvectors)
    # P              [Nxm] numpy PCA data matrix (N samples, m features)

    #pca_eigvec = None
    #data centering
    norm = np.mean(A, axis=0)
    A -= norm

    # Covariance matrix
    C = np.cov(np.transpose(A))

    #Computing eigenvalues and -vector

    evals, evecs = np.linalg.eig(C)
    evals = evals.real
    evecs = evecs.real

    indices = np.argsort(evals)
    indices = indices[::-1][:m]
    evecs = evecs[:,indices]
    evals = evals[indices]
    for i in range(np.shape(evecs)[1]):
        evecs[:,i] / np.linalg.norm(evecs[:,i]) * np.sqrt(evals[i])

    x = np.dot(np.transpose(evecs), np.transpose(A))

    y = np.transpose(np.dot(evecs, x))+norm # Added this from Marsland p.137

    P = y
    pca_eigvec = evecs
    return P
```

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).

[319]:

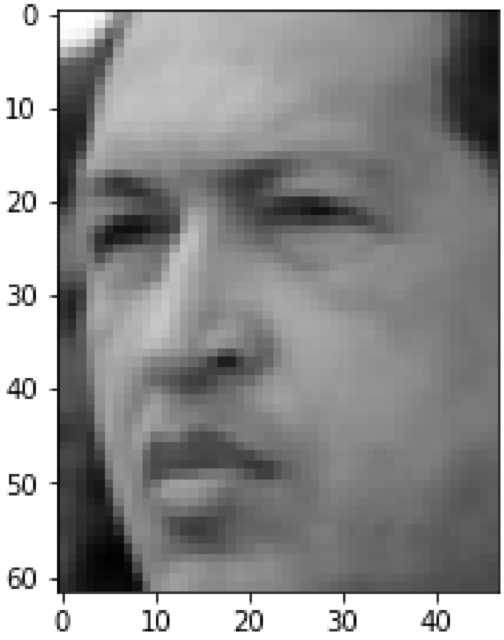
```
Xhat = encode_decode_pca(X,200)
```

1.6.5 Inspecting the reconstructed data

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

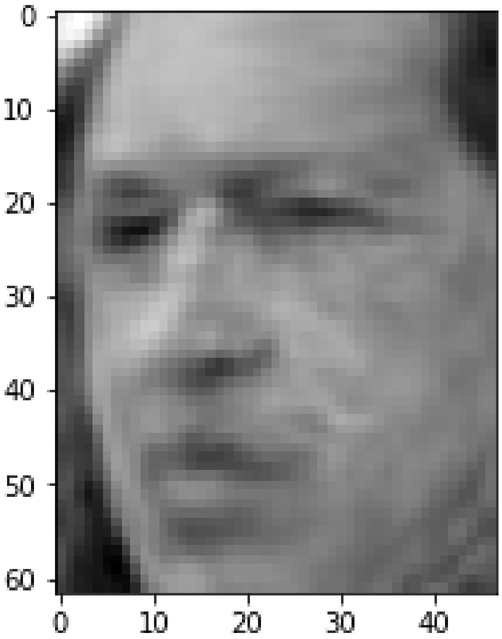
```
[58]: plt.imshow(X[0,:].reshape((h, w)), cmap=plt.cm.gray)
```

```
[58]: <matplotlib.image.AxesImage at 0x7fbc1fa46080>
```



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

```
[320]: <matplotlib.image.AxesImage at 0x7fbc1e88cac8>
```



Comment:

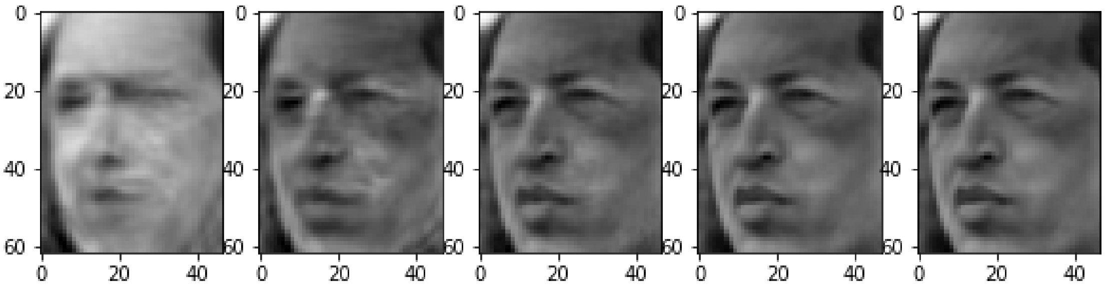
So, like most of the results with PCA. It's pretty good overall. Some detail is lost, but you can still recognize the same picture. One good thing here compared to the iris_data is the fact that the structure remains the same after the output. If this had been like the iris example the nose, eyes or mouth might have moved compared to the rest of the face, which would onviously be a bad thing. Even in the Iris example the output might still have preserved structure, but it's hard to say without testing the iris output on something which relies on those structures.

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.

[324]:
```
X,y,h,w = syntheticdata.get_lfw_data()
#plt.imshow(X[0,:].reshape((h, w)), cmap=plt.cm.gray)

import time
fig, axs = plt.subplots(1, 5, figsize=(10, 5))
imrange = [100, 200, 500, 1000]
im = 0
pre = time.time()
for i in imrange:
    start = time.time()
    Xhat = encode_decode_pca(X,i)
    axs[im].imshow(Xhat[0,:].reshape((h, w)), cmap=plt.cm.gray)
    print("Finished PCA for reduction to:", i, "pixels")
    print("processing time:", time.time() - start, "seconds")
    im += 1
print("Total processing time:", time.time() - pre, "seconds")
axs[4].imshow(X[0,:].reshape((h, w)), cmap=plt.cm.gray)
print("Original on the right!!!")
plt.show()
```

```
Finished PCA for reduction to: 100 pixels
processing time: 15.501089811325073 seconds
Finished PCA for reduction to: 200 pixels
processing time: 16.565566062927246 seconds
Finished PCA for reduction to: 500 pixels
processing time: 16.632429838180542 seconds
Finished PCA for reduction to: 1000 pixels
processing time: 14.9996018409729 seconds
Total processing time: 63.69947695732117 seconds
Original on the right!!!
```



Comment:

For 100 pixels PCA misplaces an eye, which is unfortunate for our candidate. In the 200, 500, and 1000 pixel representations, the algorithm preserves the candidates facial features which must be the underlying structure of information we're hoping PCA can capture, while reducing the size of the image. This isn't a profile picture I would want to post on Facebook, but I would call it a success in terms of how much the processed image retains the persons likenes, even while reducing down to 200 pixels.

1.7 Master Students: PCA Tuning

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*:

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

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.

[ ]:

```
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 features)
    # m     integer reporting the number of dimensions selected

    m = None
    Ahat = None

    return Ahat,m
```

Import the *lfw* dataset using the *get_lfw_data()* in *syntheticdata*. Use the implemented function to encode and decode the data by projecting on a lower dimensional space such that POV=0.9. Use the function *imshow* to plot and compare original and reconstructed pictures. Comment on the results.

[ ]:

```
X,y,h,w = syntheticdata.get_lfw_data()
```

[ ]:

```
Xhat,m = encode_decode_pca_with_pov(X,None)
```

[ ]:

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

Comment: Enter your comment here.

2 K-Means Clustering (Bachelor and master students)

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*.

[85]:
```
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].

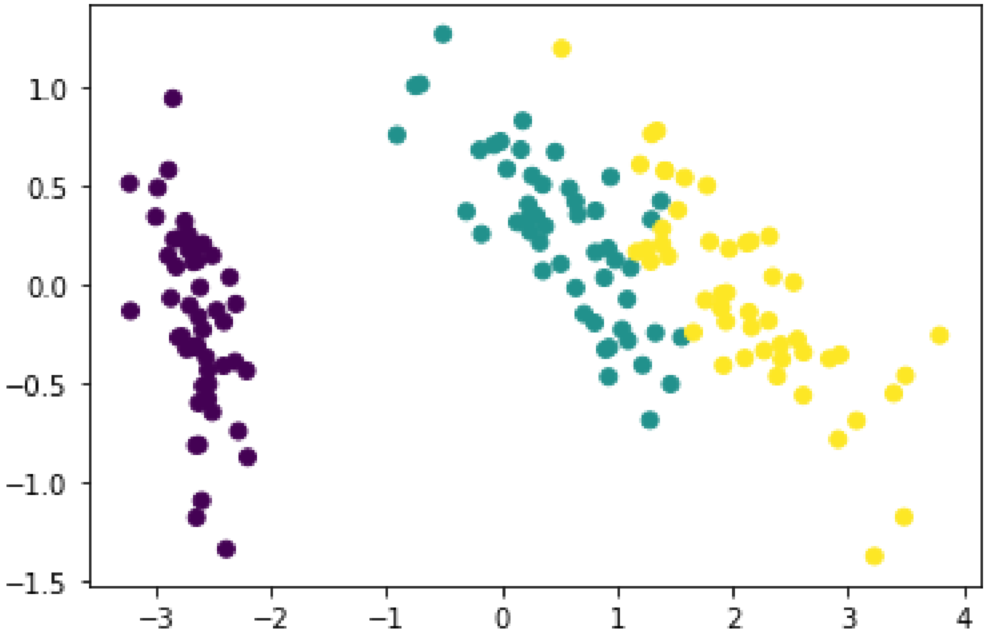
[91]:
```
X,y = syntheticdata.get_iris_data()
```

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.

[87]:
```
eigvecs, output = pca(X,2)
plt.scatter(output[:,0],output[:,1],c=y)
```

[87]:
```
<matplotlib.collections.PathCollection at 0x7f72bc906c50>
```

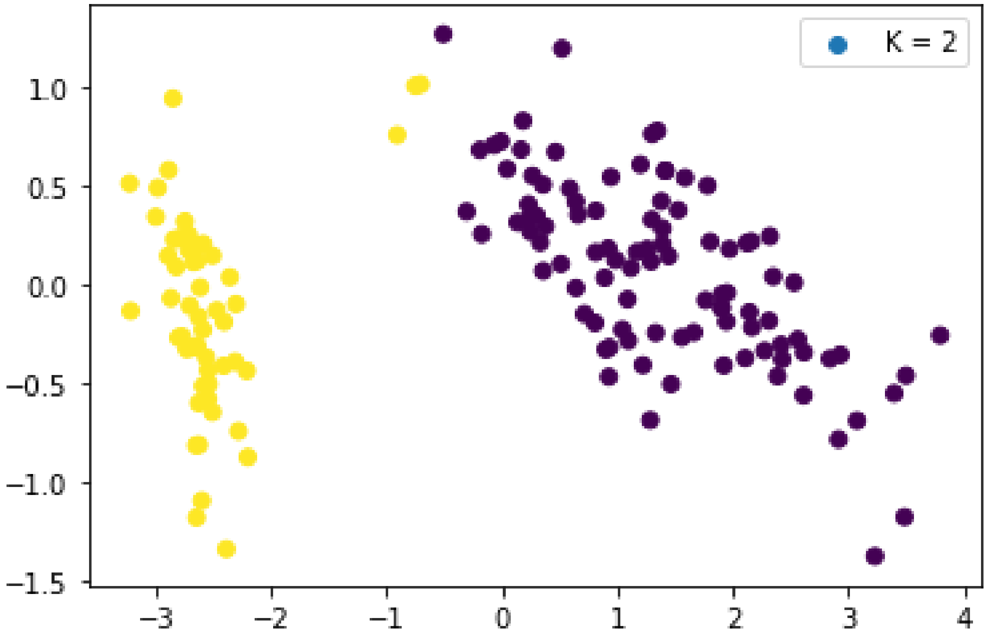


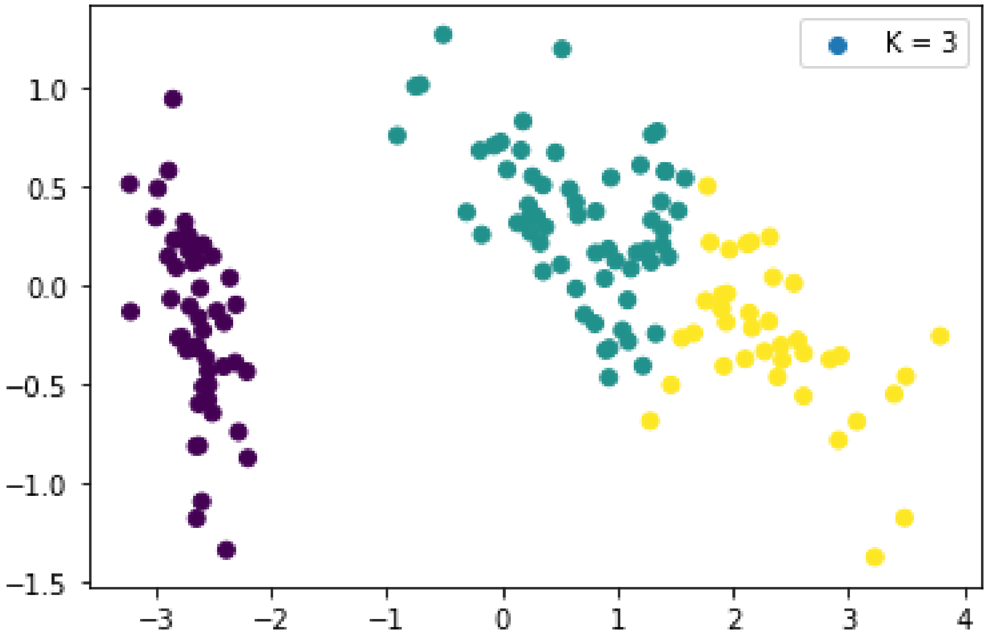
2.0.4 Running k-means

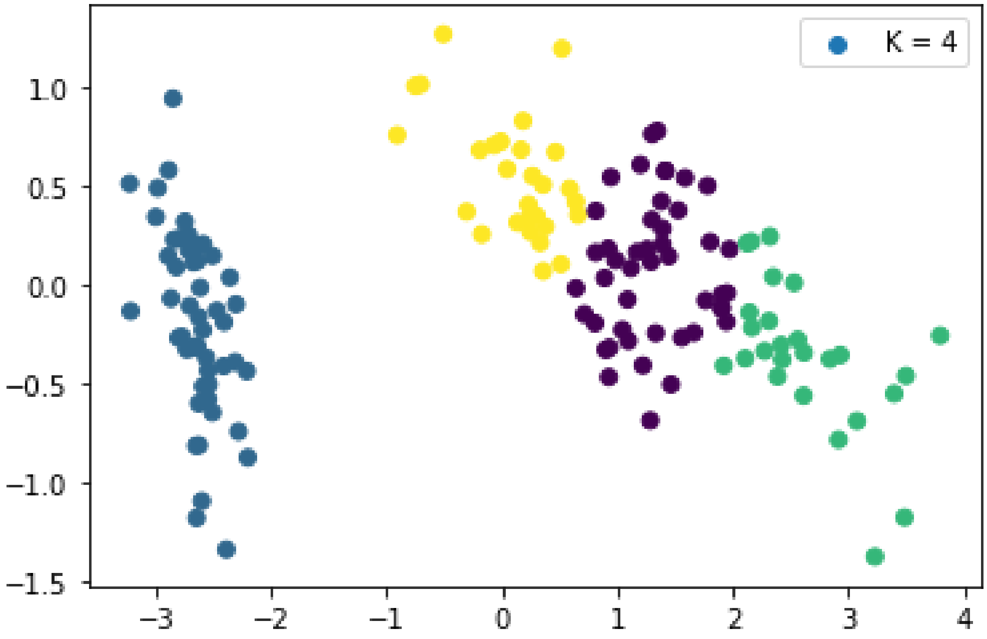
We will now consider the *iris* data set as an unlabeled set, and perform clustering to this unlabeled set. We can compare the results of the clustering to the lableled calsses.

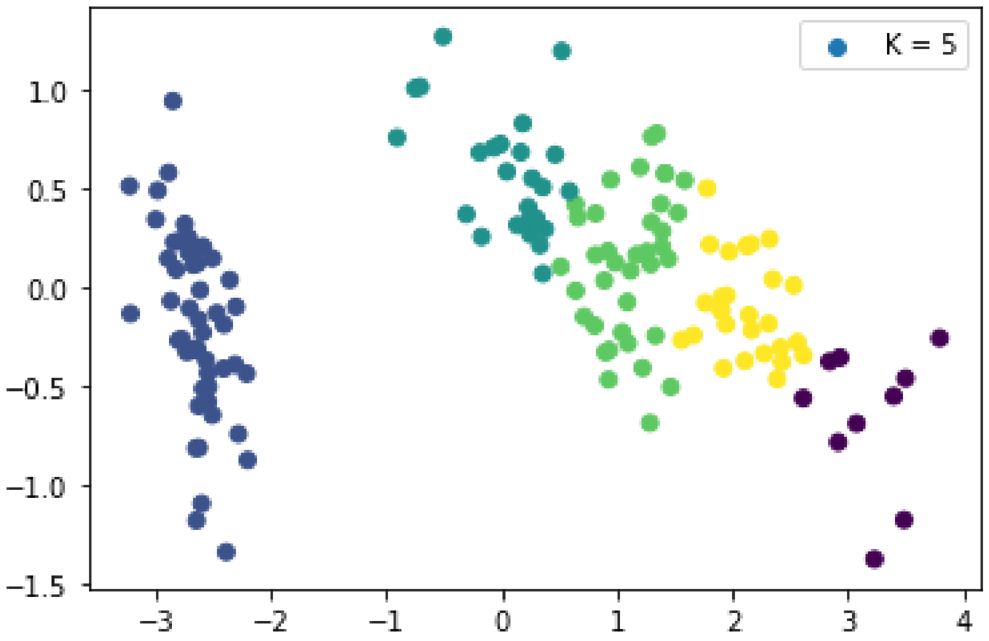
Use the class *KMeans* to fit and predict the output of the *k-means* algorithm on the projected data. Run the algorithm using the following values of $k = \{2, 3, 4, 5\}$.

[88]:
```
k_vals = [2, 3, 4, 5]
data = []
for i in k_vals:
    KM = KMeans(n_clusters=i)
    title = "K = " + str(i)
    yhat2 = KM.fit_predict(output)
    plt.scatter(output[:,0],output[:,1], c=yhat2, label=title) #add legends
    plt.legend()
    plt.show()
    data.append(yhat2)
data = np.asarray(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



2.0.5 Qualitative assessment

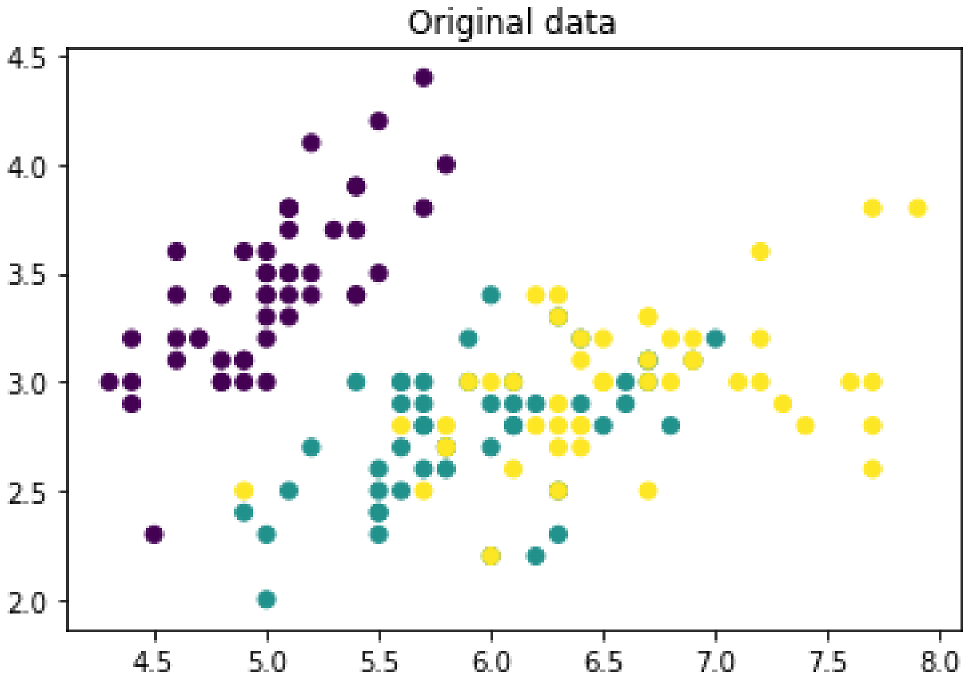
Plot the results of running the k-means algorithm, compare with the true labels, and comment.

[92]:

```
# my plots based on the PCA is above
plt.figure()
plt.scatter(X[:,0],X[:,1],c=y)
plt.title('Original data')
```

[92]:

```
Text(0.5, 1.0, 'Original data')
```



Comment:

Ok, so from the various k-means clusterings, we see that we're never actually super close to the original partition. From the outset we may have expected the k=3 cluster to be the closest given the fact that the original PCA has 3 partitions, but because of the way K-means calculates the clusters we end up with something malformed compared to the original. What we instead see is that k=3 is probably the most inaccurate of the k-partitions. Finally, k=4 and k=5 starts to partition the errounous region into smaller regions, and this will probably increase the overall accuracy for the final quantitative assessment.

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.

[80]:

```
# making several different types of dimensions for the different k-means␣
↪implementations.
X,y = syntheticdata.get_iris_data()

def define_k_dims(arr, k_vals): # Way too proud of this
    """
    A scalable dimensionality inducer, which is dependent on the k_vals␣
↪containing the amount of trainig labels per set,
    and the arr, which contains all the trained k_cluster data for those k_vals␣
↪dimensions.

    the loop creates a fresh array which consists of the number of features and␣
↪the length of the samples
    it then runs a second loop disseminating the 'one-hot' encoded information␣
↪into their nested places.

    the result will be a top array with all the different nested arrays␣
↪containing the labels dimensioned with 'one-hot.

    In this way you can directly slice the output array into a classifier, like␣
↪I do in the next cell.

    """
    k_boss = []
    for i in range(arr.shape[0]):
        k_d = np.zeros((i+2, arr[1].shape[0]))
        for j in range(k_vals[i]):
            #print(i, j)
            k_d[j] = (arr[i] == j)
        k_boss.append(k_d)


    return np.asarray(k_boss)

k_fits = np.zeros((4, 150))

k_vals = [2, 3, 4, 5]
```

```
for i in range(4):
    eigs, Data = pca(X,2)
    KM = KMeans(n_clusters=k_vals[i])
    yhat2 = KM.fit_predict(Data)
    k_fits[i,:] = yhat2

K_trains = define_k_dims(k_fits, k_vals)
```

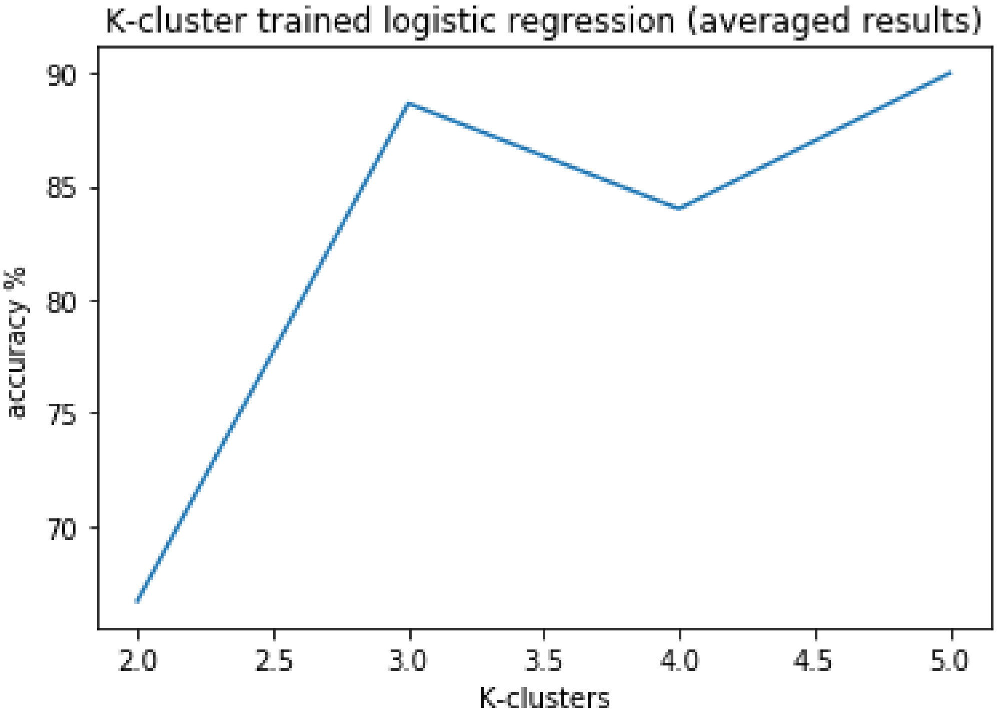
[78]:

```
X,y = syntheticdata.get_iris_data()
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
import time
k_vals = [2, 3, 4, 5]
N = 100 # iterations to average
accuracies = np.zeros((4, N))
AA = np.zeros(4)
x_dim = np.arange(4)
start = time.time()
for j in range(N):
    for i in range(4):
        K_dud = K_trains[i].T
        log_reg_cl = LogisticRegression(solver='lbfgs', multi_class='auto')
        log_reg_cl.fit(K_dud, y)
        accuracies[i,j] = log_reg_cl.score(K_dud, y)*100
        """
        eigs, Data = pca(X,2)
        KM = KMeans(n_clusters=k_vals[i])
        yhat2 = KM.fit_predict(Data)
        k_fits[i,:] = yhat2
        K_trains = define_k_dims(k_fits)

        """
print("processing time:", time.time() - start, "seconds")

for i in range(4):
    print("K-",i+2," cluster Scored (averaged){:.2f} %".format( (np.
 ↪sum(accuracies[i,:])/N )))
    #print("K-",i+2," cluster Score (max)): {:.2f} %".format(np.
 ↪amax(accuracies[i,:], axis=0)))
    AA[i] = np.sum(accuracies[i,:])/N
```

```
processing time: 2.4892385005950928 seconds
K- 2  cluster Scored (averaged)66.67 %
K- 3  cluster Scored (averaged)88.67 %
K- 4  cluster Scored (averaged)84.00 %
K- 5  cluster Scored (averaged)90.67 %
```

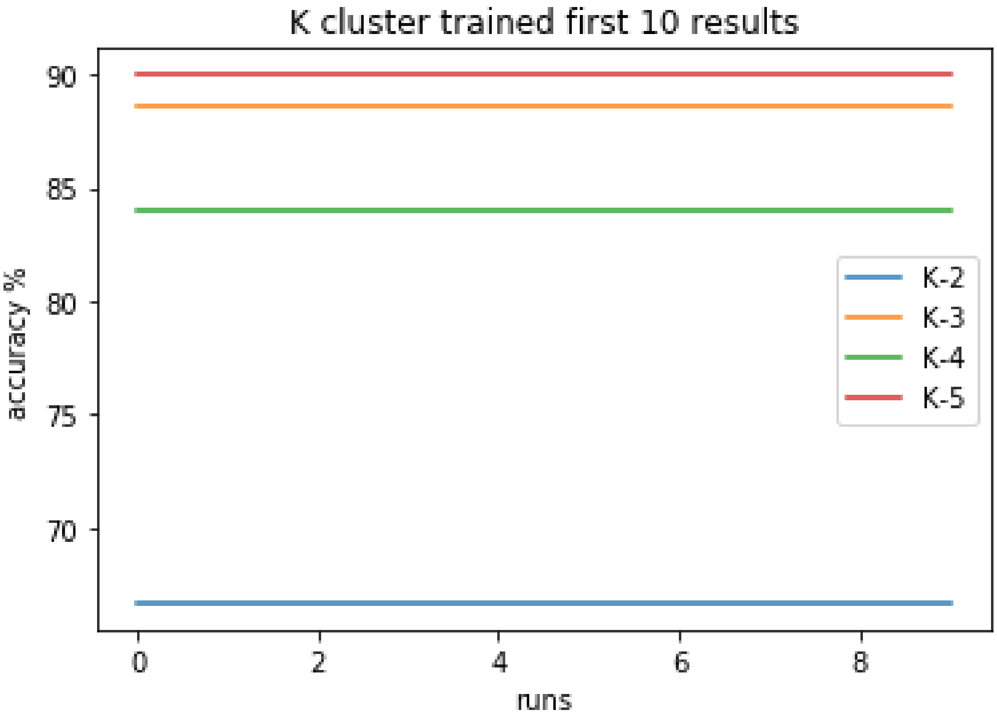
[69]:

```
x_dim = np.arange(4)+2
plt.plot(x_dim, AA)
plt.title('K-cluster trained logistic regression (averaged results)')
plt.xlabel('K-clusters')
plt.ylabel('accuracy %')
plt.show()
```



[70]:

```
#Logistic training issues below
x_dim2 = np.arange(10)
for i in range(4):
    title = "K-" + str(i+2)
    plt.plot(x_dim2, accuracies[i,:10], label=title)
plt.title('K cluster trained first 10 results')
plt.xlabel('runs')
plt.ylabel('accuracy %')
plt.legend()
plt.show()
```



Comment:

After some tinkering the algorithm now reaches the expected results i mentioned during the qualitative assesment. I did not expect the k=3 to be so high, i figured that k=3 would be worse because of the fact that the most indefinite area is usually contained in one cluster, while k=4 and k=5 has this are better defined into more areas, which I thought would make it easier for the classifier to handle. i also thought k=2 would be better than k=3 because it left the entire right area up to the classifier, instead of having the k-means clustering making assumptions before the classifier even began.

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.