IN3050 Assignment 3

Ludvik Bjørklund

April 2022

Contents

1 Principal Component Analysis

1.1 What is PCA supposed to do?

1. Variance is a measure of how 'spread out' the data set is. The higher the variance, the more spread out. The lower (closer to 0), the more concentrated the data set is spread out.
2. Covariance is a measure of how two data sets are related to eachother. For example if more of x means less of y or vice versa or that they are totally unrelated. Such information is encoded in the covariance.
3. Given N datapoints on the form $\mathbf{x}_i = (x_{1i}, \cdots, x_{Mi})$ row vectors, define the matrix

$$\mathbf{X} = \begin{pmatrix} x_{11} & \cdots & x_{1M} \\ \vdots & \ddots & \vdots \\ x_{N1} & \cdots & x_{NM} \end{pmatrix}$$

consisting of all datapoints. Generate the centred matrix $\mathbf{B}$ where each column of $\mathbf{X}$ is replaced with $\mathbf{X} - m_i$ where m_i is the mean of the i-th column. The covariance matrix is then

$$\mathbf{C} = \frac{1}{N}\mathbf{B}^T\mathbf{B}$$

4. The principle of maximal variance is to pick the components of the data set that has most variance. This will mean that we pick the components that have more variability, meaning more "spread out", loosely speaking. The principle of maximal variance, is then to have as much variation in the lower dimension data set as possible, in order to discern different types of data from each other.
5. We need this principle to more easily produce categories of data. If the data was very concentrated (low variance), then we would have a harder time to pick out what separates one data point from another. An analogy that might help: *Imagine you were to give an estimate on how many leaves lies on a lawn. If the leaves are all clustered together it is hard to tell how many there are, however if they are scattered it becomes much easier.*
6. If the data is incredibly dispersed in the sample space, it is easier to discern the data. However, to pick which data points belong to which category would also become increasingly harder, as there would be less similarities between the data. Thus, you'd have to normalise the data to create less dispersity

and make the variance actually give better information.
Conversely, if the data is concentrated, then the variability starts to matter much more, making it easier to discern data points from eachother. Which means maximal variance is probably desirable.

1.2 How is PCA implemented?

1.2.1 Importing libraries

Importing the libraries is rather trivial, and is at the top of the `PCA.py` code. It is posted below:

```python
import numpy as np
import matplotlib.pyplot as plt
import syntheticdata
```

1.2.2 Centering data

Below, I've posted my code:

```python
def center_data(A):
    N, M = A.shape
    B = np.zeros(shape=(N, M))
    for i in range(M):
        col = A[:, i]
        mean = 1/len(col) * np.sum(col)
        B[:, i] = col - mean

    return B
```

and I've tested it below with the return signature as 4*B instead to trigger an AssertionError:

```python
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)
```

below is the output with return signature in `center_data` as 4*B

```
AssertionError:
Arrays are not almost equal to 6 decimals

Mismatched elements: 10 / 12 (83.3%)
Max absolute difference: 19.5
Max relative difference: 3.
 x: array([[ -4. ,  -2. ,  -0.3],
       [  0. , -26. ,  -0.3],
       [  4. ,  22. ,   0.5],
       [  0. ,   6. ,   0.1]])
 y: array([[-1.   , -0.5  , -0.075],
       [ 0.   , -6.5  , -0.075],
       [ 1.   ,  5.5  ,  0.125],
       [ 0.   ,  1.5  ,  0.025]])
```

and otherwise with return signature as above, the `test_case` testing code resulted in no output, meaning the code passed the test.

1.2.3 Computing Covariance Matrix

The code

```python
def compute_covariance_matrix(A): # assumes A is centred
    N = A.shape[0]
    C = 1/N * np.transpose(A) @ A
    return C
```

passed the test given in the assignment sheet, as no AssertionError was raised. The code used for the test is below, for convenience's sake:

```python
testcase = center_data(np.array([[22.,11.,5.5],[10.,5.,2.5],[34.,17.,8.5],[28.,14.
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

For computing the eigenvalues and eigenvectors, consort the program below:

```python
def compute_eigenvalue_eigenvectors(A):
    eigval, eigvec = np.linalg.eig(A)
    eigval = eigval.real
    eigvec = eigvec.real

    return eigval, eigvec
```

with the test code to run below, shall there be any doubt of my code's performance. On my laptop, it successfully raised no AssertionErrors, hence passing the test.

```python
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

We sort the eigenvalues and eigenvectors with the method below, as is done in Marsland's book.

```python
def sort_eigenvalue_eigenvectors(eigval, eigvec):
    indices = np.argsort(eigval)
    indices = indices[::-1] #reverse order from argsort

    sorted_eigval = eigval[indices]
    sorted_eigvec = eigvec[:, indices]
    # sort eigenvectors by the sorting done for eigenvalues,
    # so that each eigen-pairing is correct

    return sorted_eigval, sorted_eigvec
```

which succesfully passed the testing code posted below:

```python
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

Below is the PCA algorithm implemented.

```python
def pca(A, m):
    N, M = A.shape
    A = center_data(A)
    cov_A = compute_covariance_matrix(A)
    eigval, eigvec = compute_eigenvalue_eigenvectors(cov_A)
    pca_eigval, pca_eigvec = sort_eigenvalue_eigenvectors(eigval, eigvec)

    mean = np.mean(A, axis=0)
    pca_eigvec = pca_eigvec[:,:m]

    P = np.transpose(pca_eigvec) @ np.transpose(A)

    return pca_eigvec, P.T
```

as before, this code passed the testing with no errors raised. The code used for testing is shown below:

```python
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)
```

We now turn to §1.3.

1.3 Understanding: how does PCA work?

I will include all the code from **1.3.1** to **1.3.5** below:

```python
# 1.3 how does PCA work?

# UNCOMMENT THE PLOT COMMANDS TO SEE THE PLOTS

# 1.3.1
X = syntheticdata.get_synthetic_data1()

# 1.3.2
#plt.scatter(X[:,0],X[:,1])
```

```python
#plt.show()


# 1.3.3
X = center_data(X)
# plt.scatter(X[:,0],X[:,1], color="maroon")
# plt.show()

# 1.3.4
pca_eigvec, _ = pca(X, m=2)
first_eigvec = pca_eigvec[0]
#plt.scatter(X[:,0],X[:,1], color="navy")
x = np.linspace(-5, 5, 1000)
y = first_eigvec[1]/first_eigvec[0] * x
#plt.plot(x,y, color="red")
#plt.show()

# 1.3.5
_, P = pca(X, 1)
#plt.scatter(P, np.zeros(shape=P.shape), color="purple")
#plt.show()
```

The plots produced by the scatter plots in **1.3.2** and **1.3.3** are posted below:

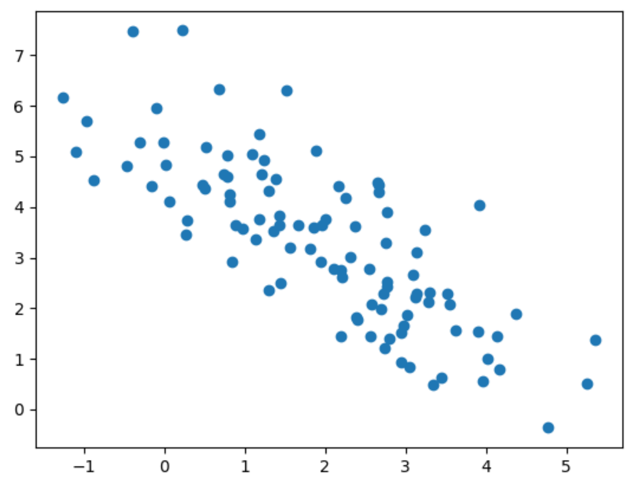


Figure 1: Uncentered data

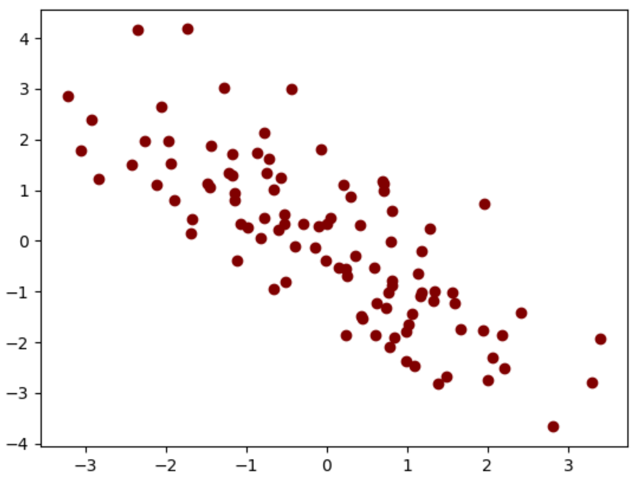


Figure 2: Centered data

The above data sets look identical in how they are distributed, but they are placed differently in 2-space. Particularily, the centered data is scattered around the origin. Below is the plot for **1.3.4** and **1.3.5**

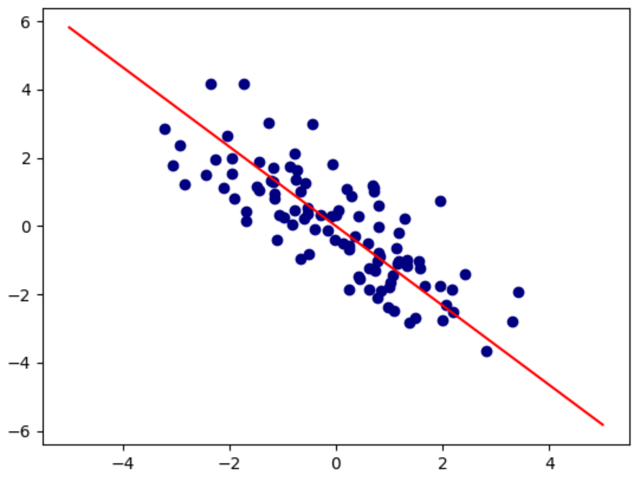


Figure 3: 1st eigenvector plot

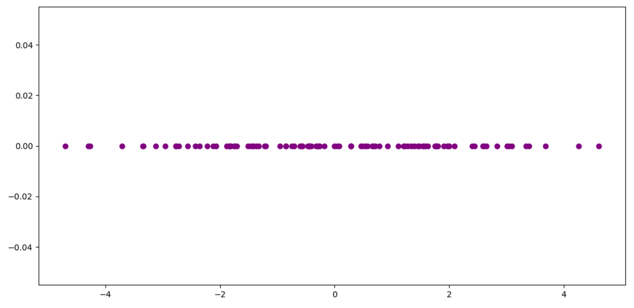


Figure 4: Projection to 1 dimension

1.4 Evalutation: when are the results of PCA sensible?

1.4.1 Loading first set of labels

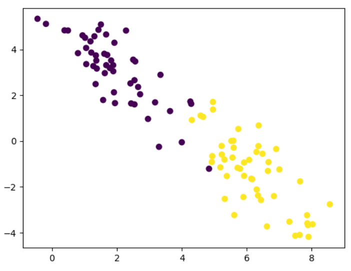
```
# 1.4.1
X, y = syntheticdata.get_synthetic_data_with_labels1()
```

1.4.2 Running PCA

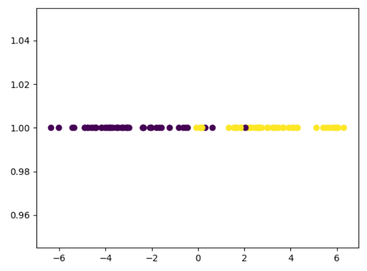
See the below code for how the data set is plotted before vs. after.

```
# 1.4.2
plt.scatter(X[:,0],X[:,1],c=y[:,0])
plt.figure()
_, P = pca(X, 1)
plt.scatter(P, np.ones(P.shape[0]),c=y[:,0])
plt.show()
```

Here are the plots:



(a) Before



(b) After

Comment: Both plots show us the data are overall rather separate. The "after" plot shows better (in my opinion) for which first coordinate values the variation starts to matter with respect to the targets. The "before" plot is in that regard perhaps harder to decipher, in the sense that even data points of the same class have "in-class" variation. Whereas the "after" plot does not share that problem to the same degree.

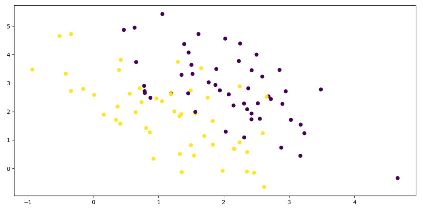
1.4.3 Loading second set of labels

Very similar to the previous loading, shown below:

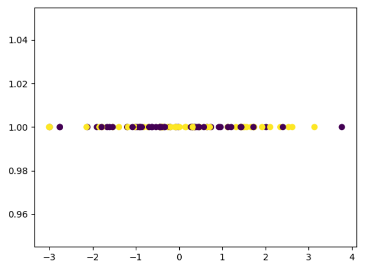
```
# 1.4.3
X, y = syntheticdata.get_synthetic_data_with_labels2()
```

1.4.4 Running PCA (again)

```
# 1.4.4
plt.scatter(X[:,0],X[:,1],c=y[:,0])
plt.figure()
_, P = pca(X, 1)
plt.scatter(P, np.ones(P.shape[0]),c=y[:,0])
plt.show()
```



(a) Before



(b) After

Comments: In this case, the PCA gets very hard to decipher in comparison to the Before plot. The data classes has tremendous variation from another, but also quite the overlap "in the middle". Thus, the PCA loses the variation but has more overlap. The "after" plot keeps this information, but is harder to look at where the variability occurs (in my opinion).
To plot the directions of the eigenvectors, (as they are in 1D, they are floats) see below:

```
X = center_data(X)
# 1.4.4
```

```
plt.scatter(X[:,0],X[:,1],c=y[:,0])
#plt.figure()
pca_eigvecs, P = pca(X, 1)
# plt.scatter(P, np.ones(P.shape[0]),c=y[:,0])
# plt.show()
x = np.linspace(-5, 5, 1000)
eig0 = pca_eigvecs[0]
y0 = eig0 * x
eig1 = pca_eigvecs[1]
y1 = eig1 * x

plt.plot(x, y0, color="blue")
plt.plot(x, y1, color="red")
plt.show()
```

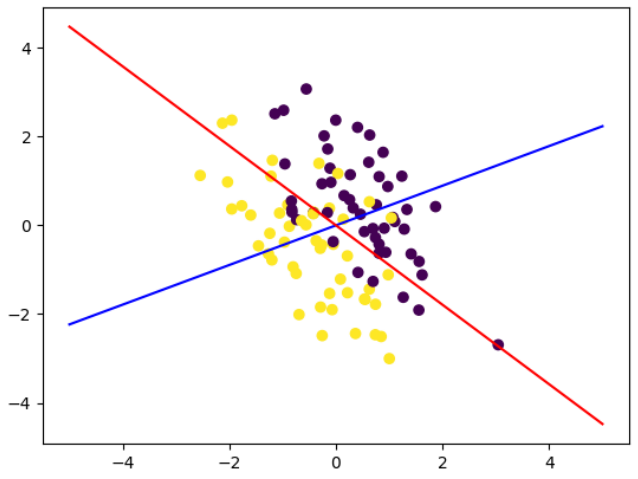


Figure 7: Eigenvectors, blue is the first, red the second

Seeing this plot, we can find some important distinctions about the choice of eigenvectors and how the data set looks like. Choosing the first (blue), we have maximal variability between the classes with less overlap. Choosing the second (red), we find tremendous overlap.

1.5 Case Study 1: PCA for visualization

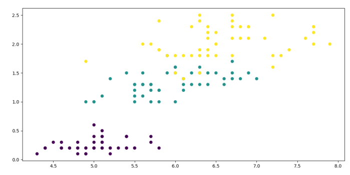
1.5.1 Loading the data

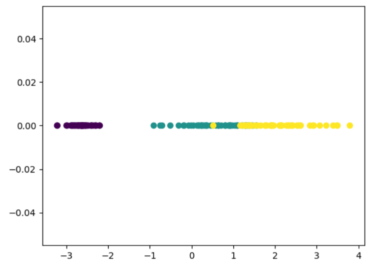
```
# 1.5.1
X,y = syntheticdata.get_iris_data()
```

1.5.2 Visualizing the data via. selecting features

1.5.3 Visualizing the data via. PCA

```
from random import randint
# 1.5.2 & 1.5.3
i, j = randint(0, 3), randint(0, 3)
while (i==j):
    i = randint(0, 3)
plt.scatter(X[:,i],X[:,j],c=y)
plt.figure()
_,P = pca(X, 1)
plt.scatter(P, np.zeros(P.shape[0]),c=y)
plt.show()
```


(a) Before



(b) After

Comment: Both plots are pretty good at describing the data set. The PCA plot shows the overlaps between the greenblue and yellow class somewhat clearer though.
The 'before' plot however shows where in 2D space each class lies, with purple-ish lying "lower", and yellow lying "higher".

1.6 Case Study 2: PCA for compression

1.6.1 Loading data

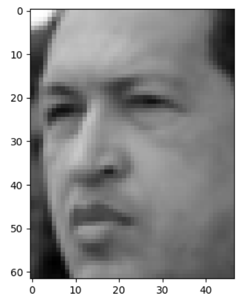
1.6.2 Inspecting the data

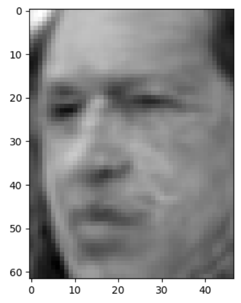
1.6.3 Implementing compression-decompression

1.6.4 Compressing and decompressing data

1.6.5 Inspecting reconstructed data

```python
# 1.6

# 1.6.1
X, y, h, w = syntheticdata.get_lfw_data()

# 1.6.2
plt.imshow(X[0, :].reshape((h, w)), cmap=plt.cm.gray)
plt.show()

# 1.6.3
def encode_decode_pca(A, m):
    mean = np.mean(A, axis=0)
    eigvecs, A = pca(A, m)
    A = A.T

    A_hat = (eigvecs @ A).T + mean

    return A_hat

# 1.6.4
X_hat = encode_decode_pca(X, 200)

# 1.6.5
plt.imshow(X_hat[0, :].reshape((h, w)), cmap=plt.cm.gray)
plt.show()
```

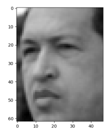

(a) Before

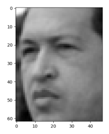

(b) After

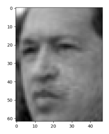
The difference before compression/decompression and after is quite significant. The main features such as eyes, nose, mouth, eyebrows and such are kept quite similar. However, there are noticable distortions where the grayscale does not vary as heavily. My guess is this is the 'weakness' of the PCA algorithm; that the data with less variability can be prone to distortion as we tend to consider the dimension(s) of maximal variability.

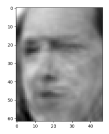
1.6.6 Evaluating different compressions

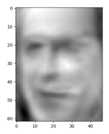
```python
# 1.6.6
for i in [1000, 800, 300, 100, 40]:
    X_hat = encode_decode_pca(X, i)
    plt.figure()
    plt.imshow(X_hat[0, :].reshape((h, w)), cmap=plt.cm.gray)
    plt.savefig(f"1.6.6_{i}.png")

plt.show()
```


(a) m = 1000


(b) m = 800


(c) m = 300


(d) m = 100


(e) m = 40

Comments: Although the images are quite small, they do exhibit the behaviour of choosing sufficiently small m rather well. Trying to reduce the dimensions too much, results with heavier distortions. It is as if the computer has extreme short term memory when $m \leq 100$ and only "remembers" few discerning features. One could ask the question: Is this how human memory works? But that is beyond the scope of this assignment.

2 K-Means Clustering

2.0.1 Importing scikit-learn

2.0.2 Loading data

2.0.3 Projecting data using PCA

```python
# 2.0.1 - 2.0.3
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt
from PCA import encode_decode_pca, pca
import syntheticdata


X, y = syntheticdata.get_iris_data()

_, P = pca(X,2)

plt.scatter(P, np.zeros(shape=P.shape), color="maroon")
plt.show()
```

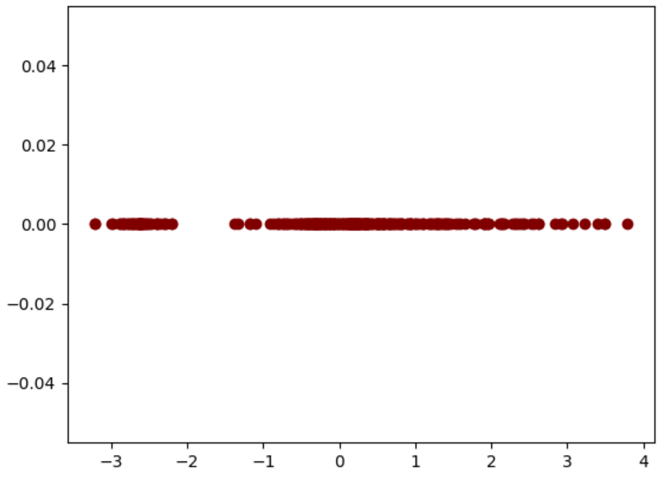


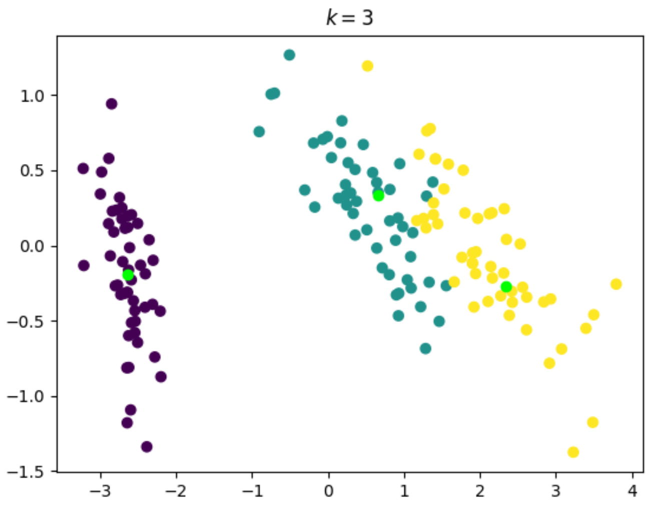
Figure 11: iris_data projected in 2-space with PCA

2.0.4 Running k-means

```
# 2.0.4
KM2 = KMeans(2)
yhat2 = KM2.fit_predict(P)

KM3 = KMeans(3)
yhat3 = KM3.fit_predict(P)

KM4 = KMeans(4)
yhat4 = KM4.fit_predict(P)

KM5 = KMeans(5)
yhat5 = KM5.fit_predict(P)
```

2.0.5 Qualitative assessment

```python
# 2.0.5
for i in [2,3,4,5]:
    KM = KMeans(i)
    centres = KM.cluster_centers_
    plt.figure()
    plt.scatter(P[:,0], P[:,1], c=y)
    plt.scatter(centres[:,0], centres[:,1], marker="o", color="lime")
    plt.title(f"$k={i}$")
    plt.savefig(f"Kmeans_{i}.png")
```



Note: I believe I might have misused the c=y argument in the code. Since the k that is the same as the number of labels (3 in this case), the one shown above (more of them will be stored if you run `K_means.py`. You may have to uncomment some code, but it is the same segment as the part above).

Comments: We can see from the $k = 3$ plot that the centres are on the purple class rather decently placed, as well as for the green/blue class. However, the yellow class seems to have a center that is too far shifted down, and hence we might need to increase k. (Please keep the 'savefig' line and have access to the plots readily, as I will discuss them now). $k = 2$ gives only a rough sketch of where the data is situated. $k = 4$ shows a decent tendency of the data set and where the data is situated. $k = 5$ does somewhat the same as $k = 4$. The problem though, is that the purple class is sort of "left out".

3 Quantitative Assessment of K-means

This code is found in the `quant_assessment.py` file.

```
from sklearn.cluster import KMeans
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import OneHotEncoder
from PCA import encode_decode_pca, pca
import syntheticdata
from K_means import KM2, KM3, KM4, KM5, yhat2, yhat3, yhat4, yhat5

def one_hot_encoding(t):
    t_one_hot = []
    t_max = max(t) + 1
    for i in t:
        vec = np.zeros(t_max)
        vec[i] = 1
        t_one_hot.append(vec)
    t_one_hot = np.array(t_one_hot)
    return t_one_hot

# I
X, y = syntheticdata.get_iris_data()
_ , P = pca(X, 2)

logreg = LogisticRegression()
logreg.fit(P, y)
# print(accuracy_score(y, yhat2))
```

```python

# II
yhat2_hot = one_hot_encoding(yhat2)
yhat3_hot = one_hot_encoding(yhat3)
yhat4_hot = one_hot_encoding(yhat4)
yhat5_hot = one_hot_encoding(yhat5)

logreg2 = LogisticRegression()
logreg3 = LogisticRegression()
logreg4 = LogisticRegression()
logreg5 = LogisticRegression()

logreg2.fit(yhat2_hot, y)
logreg3.fit(yhat3_hot, y)
logreg4.fit(yhat4_hot, y)
logreg5.fit(yhat5_hot, y)


print("k \t||\t Accuracy")
print(2, "\t||\t", logreg2.score(yhat2_hot, y))
print(3, "\t||\t", logreg3.score(yhat3_hot, y))
print(4, "\t||\t", logreg4.score(yhat4_hot, y))
print(5, "\t||\t", logreg5.score(yhat5_hot, y))

k_arr = [2,3,4,5]
accs = [logreg2.score(yhat2_hot, y), logreg3.score(yhat3_hot, y),
        logreg4.score(yhat4_hot, y), logreg5.score(yhat5_hot, y)]

plt.plot(k_arr, accs, "o--")
plt.show()
```

The above code produces following outputs:

```
k       ||      Accuracy
2       ||      0.6666666666666666
3       ||      0.8866666666666667
4       ||      0.88
5       ||      0.8933333333333333
```

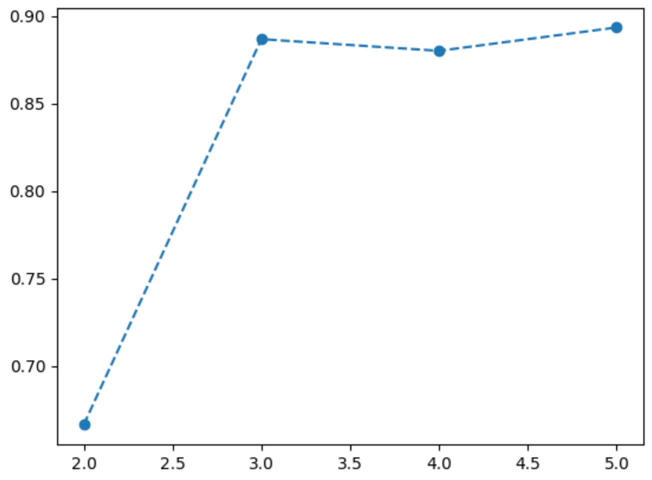


Figure 12: Accuracy over time

Comment: It seems that $k = 5$ gives decent accuracy, but it might be that I've used the wrong method. The reason I used the score method in the `LogisticRegression` class, is because I couldn't relate the logistic regression models trained above to the call-signature to `accuracy_score`.

⋆ ⋆ ⋆