Ministry of Education, Culture, and Research of the Republic of Moldova

Technical University of Moldova

Department of Software Engineering and Automatics

Study Program: Software Engineering

Report

Data analysis and visualisation

Done by: Ion Dodon, IS211-M

Verified by: Grozavu Nistor

Chisinau, 2021

Laboratory work no. 1

Theoretical material

Feature scaling - Feature scaling is a method used to normalize the range of independent variables or features of data. In data processing, it is also known as data normalization and is generally performed during the data preprocessing step.

Since the range of values of raw data varies widely, in some machine learning algorithms, objective functions will not work properly without normalization. For example, many classifiers calculate the distance between two points by the Euclidean distance. If one of the features has a broad range of values, the distance will be governed by this particular feature. Therefore, the range of all features should be normalized so that each feature contributes approximately proportionately to the final distance.

Another reason why feature scaling is applied is that gradient descent converges much faster with feature scaling than without it.[1]

It's also important to apply feature scaling if regularization is used as part of the loss function (so that coefficients are penalized appropriately).

Partial component analysis - The principal components of a collection of points in a real coordinate space are a sequence of unit vectors, where the -th vector is the direction of a line that best fits the data while being orthogonal to the first vectors. Here, a best-fitting line is defined as one that minimizes the average squared distance from the points to the line. These directions constitute an orthonormal basis in which different individual dimensions of the data are linearly uncorrelated. Principal component analysis (PCA) is the process of computing the principal components and using them to perform a change of basis on the data, sometimes using only the first few principal components and ignoring the rest.

Linear discriminant analysis - Linear discriminant analysis (LDA), normal discriminant analysis (NDA), or discriminant function analysis is a generalization of Fisher's linear discriminant, a method used in statistics and other fields, to find a linear combination of features that characterizes or separates two or more classes of objects or events. The resulting combination may be used as a linear classifier, or, more commonly, for dimensionality reduction before later classification.

LDA is closely related to analysis of variance (ANOVA) and regression analysis, which also attempt to express one dependent variable as a linear combination of other features or measurements. However, ANOVA uses categorical independent variables and a continuous dependent variable, whereas discriminant analysis has continuous independent variables and a categorical dependent variable (*i.e.* the class label). Logistic regression and probit regression are more similar to LDA than ANOVA is, as they also explain a categorical variable by the values of continuous independent variables. These other methods are preferable

in applications where it is not reasonable to assume that the independent variables are normally distributed, which is a fundamental assumption of the LDA method.

Conclusions

Working on this laboratory work I used different dimensionality reduction techniques and analyzed the differences between them. I have understood how to print useful information about a dataset like feature names, the classes, the number of objects, etc., and how to plot the clusters on subplots.

I compared the correlations between all possible combinations of variables from the iris dataset and chose the best pair of features by analyzing the correlation given by *corrcoef* function and by analyzing the clusters on the plots. Also, I practiced how to display the number of images from the MNIST dataset.

Bibliography

- https://stats.stackexchange.com/questions/109071/standardizing-features-when-using-lda-as-a-pre-processing-step
- https://en.wikipedia.org/wiki/Feature_scaling#:~:text=Feature%20scaling%20is%20a%20method,during%20the%20data%20preprocessing%20step.
- https://en.wikipedia.org/wiki/Principal_component_analysis#:~:text=Principal%20component%20analysis%20(PCA)%20is,components%20and%20ignoring%20the%20rest.
- http://yann.lecun.com/exdb/mnist/

IonDodonLaboratoryWork1

December 23, 2021

0.0.1 A. An introduction to machine learning with scikit-learn

[ ]:
```
from sklearn import *
import numpy as np
import matplotlib.pyplot as plt
```

[ ]:
```
iris = datasets.load_iris()
```

[ ]:
```
# Print the number of data, names of variables and the name of classes (use␣
↪print).
print(iris.data.shape)
print(iris.feature_names)
print(iris.target_names)
```

```
(150, 4)
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width
(cm)']
['setosa' 'versicolor' 'virginica']
```

[ ]:
```
# for each object
for i in range(len(iris.target)):
    # print the name of the class
    print(iris.target_names[iris.target[i]], end=", ")
```

```
setosa, setosa, setosa, setosa, setosa, setosa, setosa, setosa, setosa, setosa,
setosa, setosa, setosa, setosa, setosa, setosa, setosa, setosa, setosa, setosa,
setosa, setosa, setosa, setosa, setosa, setosa, setosa, setosa, setosa, setosa,
setosa, setosa, setosa, setosa, setosa, setosa, setosa, setosa, setosa, setosa,
setosa, setosa, setosa, setosa, setosa, setosa, setosa, setosa, setosa, setosa,
versicolor, versicolor, versicolor, versicolor, versicolor, versicolor,
versicolor, versicolor, versicolor, versicolor, versicolor, versicolor,
versicolor, versicolor, versicolor, versicolor, versicolor, versicolor,
versicolor, versicolor, versicolor, versicolor, versicolor, versicolor,
versicolor, versicolor, versicolor, versicolor, versicolor, versicolor,
versicolor, versicolor, versicolor, versicolor, versicolor, versicolor,
versicolor, versicolor, versicolor, versicolor, versicolor, versicolor,
versicolor, versicolor, versicolor, versicolor, versicolor, versicolor,
versicolor, versicolor, virginica, virginica, virginica, virginica, virginica,
virginica, virginica, virginica, virginica, virginica, virginica, virginica,
```

```
virginica, virginica, virginica, virginica, virginica, virginica, virginica,
virginica, virginica, virginica, virginica, virginica, virginica, virginica,
virginica, virginica, virginica, virginica, virginica, virginica, virginica,
virginica, virginica, virginica, virginica, virginica, virginica, virginica,
virginica, virginica, virginica, virginica, virginica, virginica, virginica,
virginica, virginica, virginica,
```

0.0.2 B. Data normalization

[ ]:
```
# Import the packages numpy (scientific computation) and preprocessing (data preprocessing)
from sklearn import preprocessing
import numpy as np
```

[ ]:
```
# Crate the following matrice X :
# 1, -1, 2,
# 2, 0, 0,
# 0, 1, -1
X = np.array([[1, -1, 2], [2, 0, 0], [0, 1, -1]])
```

[ ]:
```
# Print X and compute the mean and the variance of X.
print(X)
print("Mean = {}".format(np.mean(X, axis=0)))
print("Variance = {}".format(np.var(X, axis=0)))
```

```
[[ 1 -1  2]
 [ 2  0  0]
 [ 0  1 -1]]
Mean = [1.         0.         0.33333333]
Variance = [0.66666667 0.66666667 1.55555556]
```

[ ]:
```
# Use the scale function to normalize X
X_scaled = preprocessing.scale(X, axis=0)
print(X_scaled)
```

```
[[ 0.         -1.22474487  1.33630621]
 [ 1.22474487  0.         -0.26726124]
 [-1.22474487  1.22474487 -1.06904497]]
```

[ ]:
```
# Compute the mean and the variance of the scaled X. What can you conclude?
print("Mean = {}".format(np.mean(X_scaled, axis=0)))
print("Variance = {}".format(np.var(X_scaled, axis=0)))
```

```
Mean = [0. 0. 0.]
Variance = [1. 1. 1.]
```

After scaling, all three features have the same mean and variance.

0.0.3 C. MinMax Normalization

An alternative standardization is scaling features to lie between a given minimum and maximum value, often between zero and one. This can be achieved using MinMaxScaler.

[ ]:
```
# Create the following matrix X2:
# 1, -1, 2,
# 2, 0, 0,
# 0, 1, -1
X2 = np.array([[1, -1, 2], [2, 0, 0], [0, 1, -1]])
```

[ ]:
```
# Print the matrix and compute the mean of the variables
print(X2)
print("Mean = {}".format(np.mean(X2, axis=0)))
print("Variance = {}".format(np.var(X2, axis=0)))
```

```
[[ 1 -1  2]
 [ 2  0  0]
 [ 0  1 -1]]
Mean = [1.         0.         0.33333333]
Variance = [0.66666667 0.66666667 1.55555556]
```

[ ]:
```
# Normalize the data using MinMaxScaler
from sklearn import preprocessing


def minmax_normalize(X):
    scaler = preprocessing.MinMaxScaler()
    scaler.fit(X)
    return scaler.transform(X)


X_minmax_scaled = minmax_normalize(X2)
print(X_minmax_scaled)
```

```
[[0.5        0.         1.        ]
 [1.         0.5        0.33333333]
 [0.         1.         0.        ]]
```

[ ]:
```
# compute the mean and the variance
print("Mean = {}".format(np.mean(X_minmax_scaled, axis=0)))
print("Variance = {}".format(np.var(X_minmax_scaled, axis=0)))
```

```
Mean = [0.5        0.5        0.44444444]
Variance = [0.16666667 0.16666667 0.17283951]
```

The mean and variance are almost the same on all features. With MinMaxScaler, the mean and variance are bigger, and this is prbably because the MinMaxScaler scales the features to lie between zero and one.

0.0.4 D. Data visualization

[ ]:
```
# Import the Iris dataset using : iris = datasets.load_iris()
from sklearn import datasets


iris = datasets.load_iris()
```

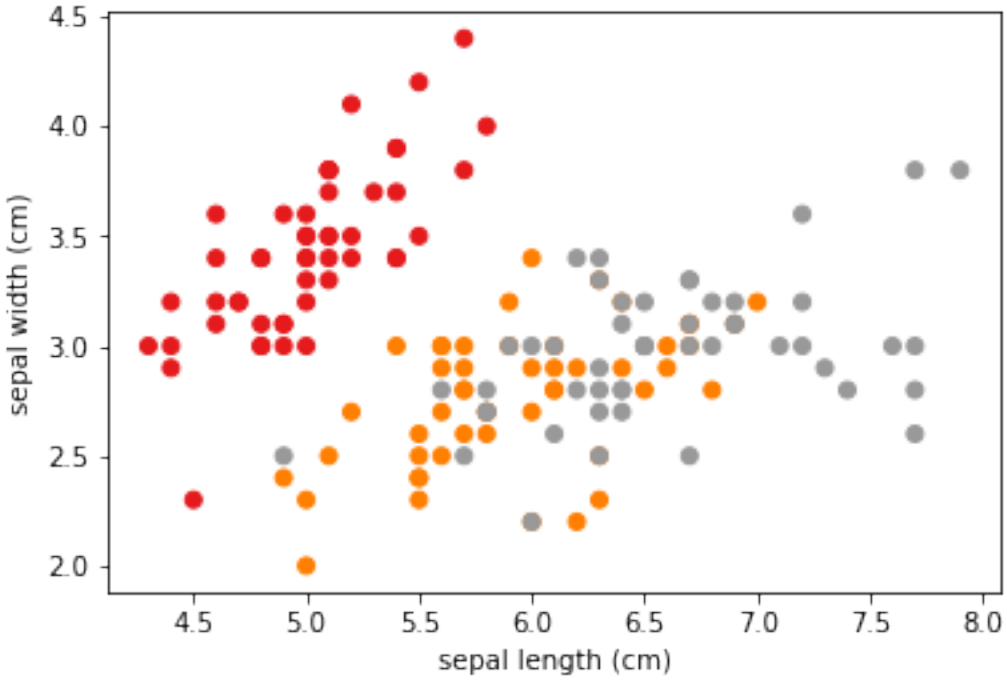
[ ]:
```
# The variable iris is an object in Phyton which contains the matrix of data
# (iris.data), the corresponding label (target), the names of the variables
# (feature_names) and the name of classes (target_names).
```

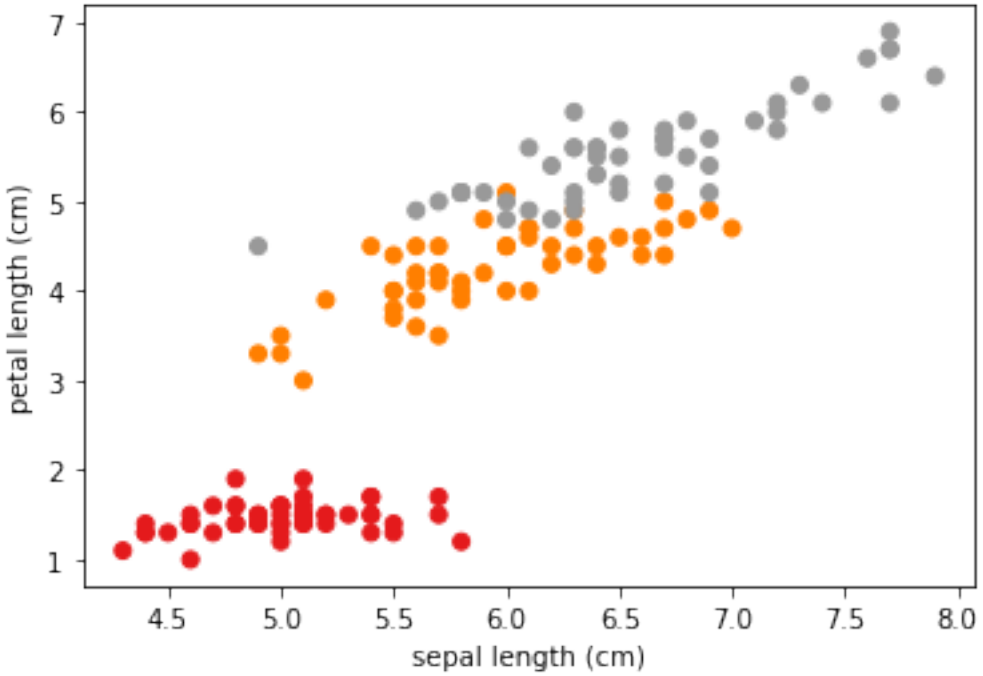
[ ]:
```
iris.feature_names
```

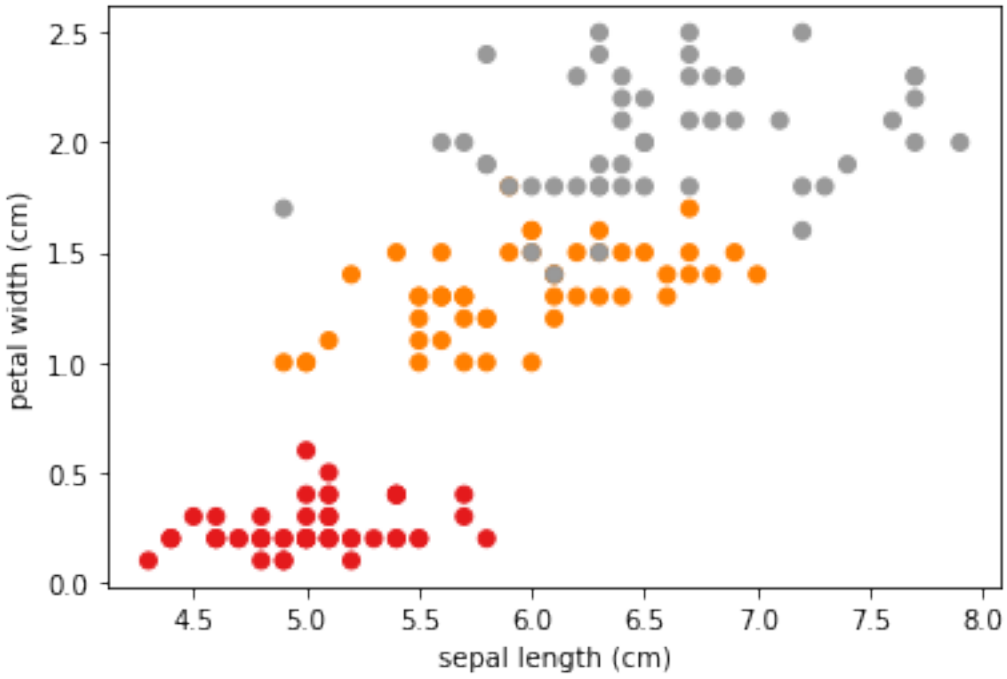
[ ]:
```
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

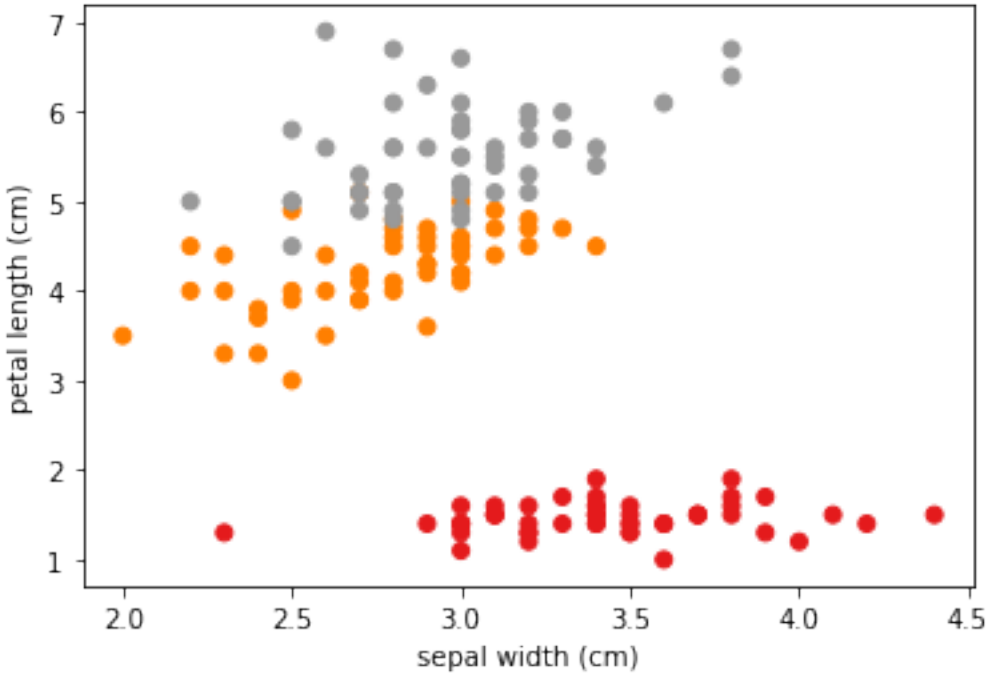
[ ]:
```
# Plot the data points into 2D dimension with all the possible combination␣
↪between variables and use the label for the color points
plt.scatter(iris.data[:, 0], iris.data[:, 1], c=iris.target, cmap=plt.cm.Set1)
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.show()

plt.scatter(iris.data[:, 0], iris.data[:, 2], c=iris.target, cmap=plt.cm.Set1)
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[2])
plt.show()

plt.scatter(iris.data[:, 0], iris.data[:, 3], c=iris.target, cmap=plt.cm.Set1)
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[3])
plt.show()

plt.scatter(iris.data[:, 1], iris.data[:, 2], c=iris.target, cmap=plt.cm.Set1)
plt.xlabel(iris.feature_names[1])
plt.ylabel(iris.feature_names[2])
plt.show()

plt.scatter(iris.data[:, 1], iris.data[:, 3], c=iris.target, cmap=plt.cm.Set1)
plt.xlabel(iris.feature_names[1])
plt.ylabel(iris.feature_names[3])
plt.show()

plt.scatter(iris.data[:, 2], iris.data[:, 3], c=iris.target, cmap=plt.cm.Set1)
plt.xlabel(iris.feature_names[1])
```

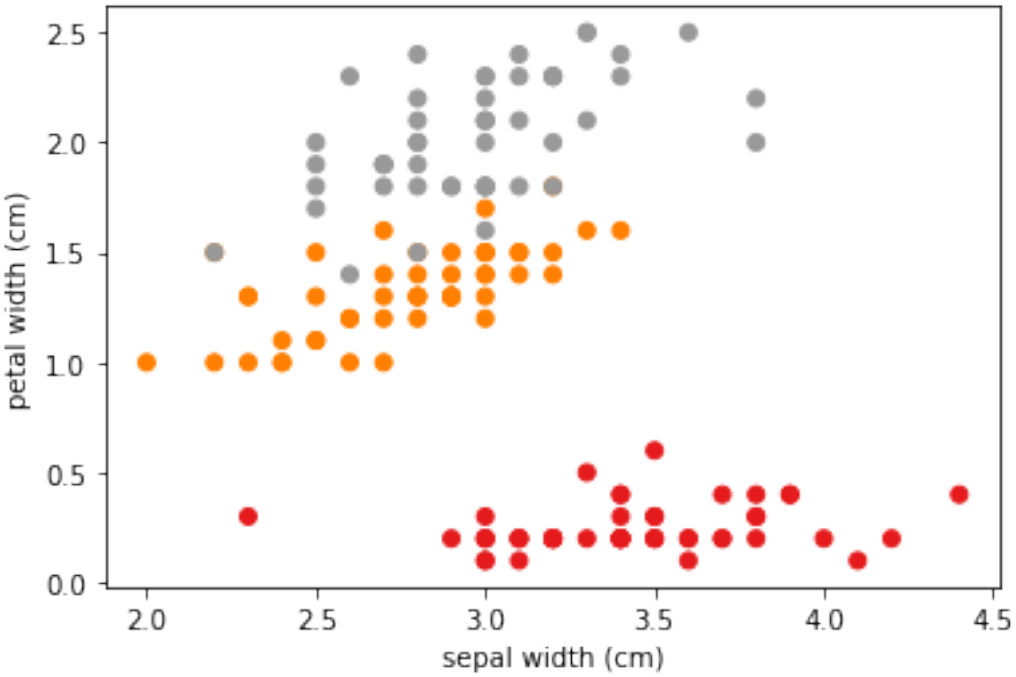
```
plt.ylabel(iris.feature_names[3])
plt.show()
```

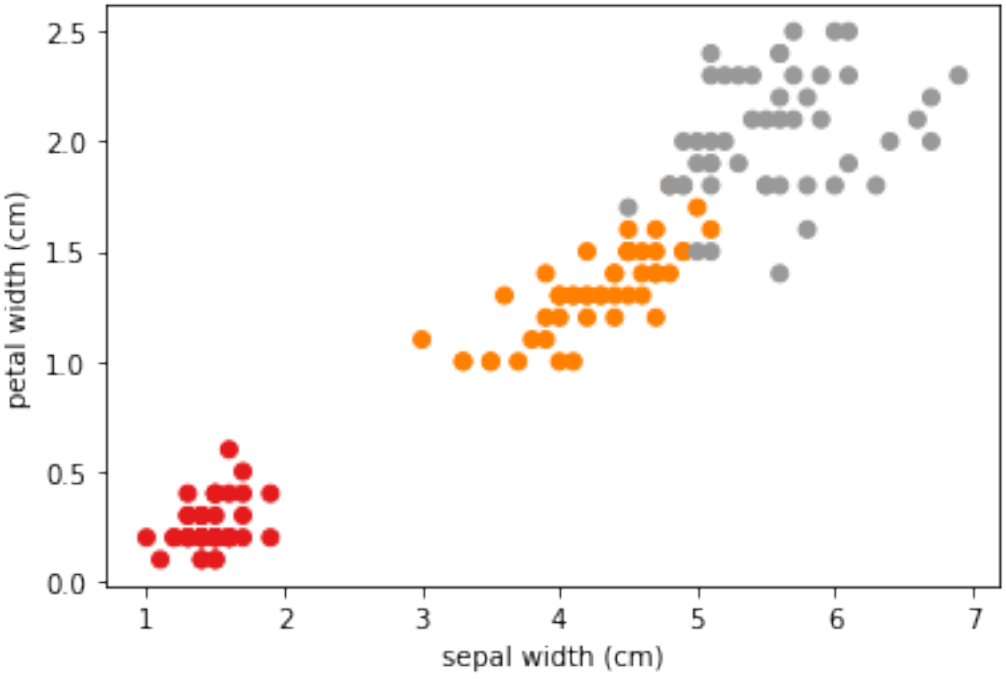





petal width (cm)
sepal length (cm)
0.0
0.5
1.0
1.5
2.0
2.5
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0


petal length (cm)
sepal width (cm)
1
2
3
4
5
6
7
2.0
2.5
3.0
3.5
4.0
4.5





The best combination of features for clustering is petal width and sepal length, because the foramted clusters are well-separated and the data in a cluser is more homogeneous and condensed.

[ ]:
```
# correlation between iris.feature_names[0] and iris.feature_names[1]
print(np.corrcoef(iris.data[:, 0], iris.data[:, 1]))
print(np.corrcoef(iris.data[:, 0], iris.data[:, 2]))
print(np.corrcoef(iris.data[:, 0], iris.data[:, 3]))
print(np.corrcoef(iris.data[:, 1], iris.data[:, 2]))
print(np.corrcoef(iris.data[:, 1], iris.data[:, 3]))
print(np.corrcoef(iris.data[:, 2], iris.data[:, 3]))
```

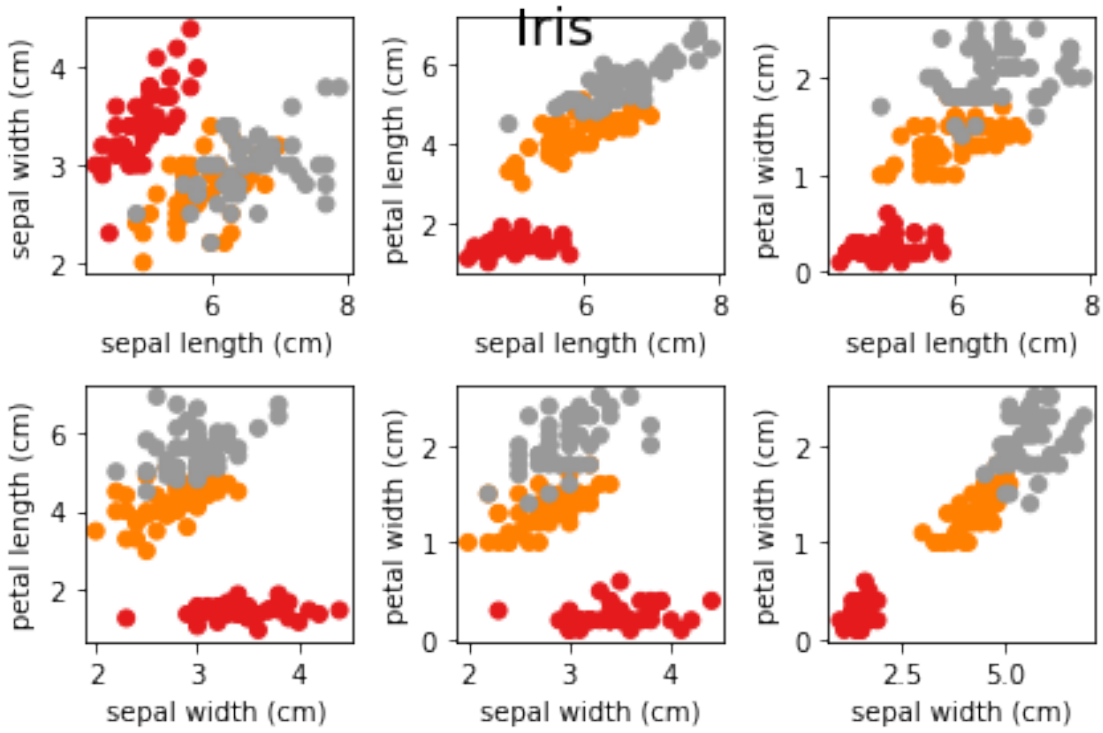
```
[[ 1.         -0.11756978]
 [-0.11756978  1.        ]]
[[1.         0.87175378]
 [0.87175378 1.        ]]
[[1.         0.81794113]
 [0.81794113 1.        ]]
[[ 1.        -0.4284401]
 [-0.4284401  1.       ]]
[[ 1.         -0.36612593]
 [-0.36612593  1.        ]]
[[1.         0.96286543]
 [0.96286543 1.        ]]
```

The correlaction between petal width and sepal length is 0.96%, and yes indeed this pair of features is the highest correlated which means tey are best to use for clusterization.

[ ]:
```
# Subplots in matplotlib
import matplotlib.pyplot as plt

fig = plt.figure()

ax1 = fig.add_subplot(231)
ax1.scatter(iris.data[:, 0], iris.data[:, 1], c=iris.target, cmap=plt.cm.Set1)
ax1.set_xlabel(iris.feature_names[0])
ax1.set_ylabel(iris.feature_names[1])

ax2 = fig.add_subplot(232)
ax2.scatter(iris.data[:, 0], iris.data[:, 2], c=iris.target, cmap=plt.cm.Set1)
ax2.set_xlabel(iris.feature_names[0])
ax2.set_ylabel(iris.feature_names[2])

ax3 = fig.add_subplot(233)
ax3.scatter(iris.data[:, 0], iris.data[:, 3], c=iris.target, cmap=plt.cm.Set1)
ax3.set_xlabel(iris.feature_names[0])
ax3.set_ylabel(iris.feature_names[3])

ax4 = fig.add_subplot(234)
ax4.scatter(iris.data[:, 1], iris.data[:, 2], c=iris.target, cmap=plt.cm.Set1)
ax4.set_xlabel(iris.feature_names[1])
ax4.set_ylabel(iris.feature_names[2])
```

```
ax4 = fig.add_subplot(235)
ax4.scatter(iris.data[:, 1], iris.data[:, 3], c=iris.target, cmap=plt.cm.Set1)
ax4.set_xlabel(iris.feature_names[1])
ax4.set_ylabel(iris.feature_names[3])

ax4 = fig.add_subplot(236)
ax4.scatter(iris.data[:, 2], iris.data[:, 3], c=iris.target, cmap=plt.cm.Set1)
ax4.set_xlabel(iris.feature_names[1])
ax4.set_ylabel(iris.feature_names[3])

plt.tight_layout()
fig = plt.gcf()

# set title
fig.suptitle('Iris', fontsize=20)
```

[ ]: Text(0.5, 0.98, 'Iris')



0.0.5 E. Data reduction and visualization

Use the correlations information's found in D.3 and reduce the dataset to 3 variables then to 2 variables.

[ ]:
```
iris.feature_names
```

[ ]:
```
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

[ ]:
```
# The variables that form the highest correlations are: sepal length, petal length, petal width, sepal width
# reduce the dataset to the variables that form the highest correlations

iris.feature_names = ['sepal length', 'sepal width', 'petal width']

# reduce the dataset to the variables that form the highest correlations
iris.data = iris.data[:, [0, 2, 3]]
```

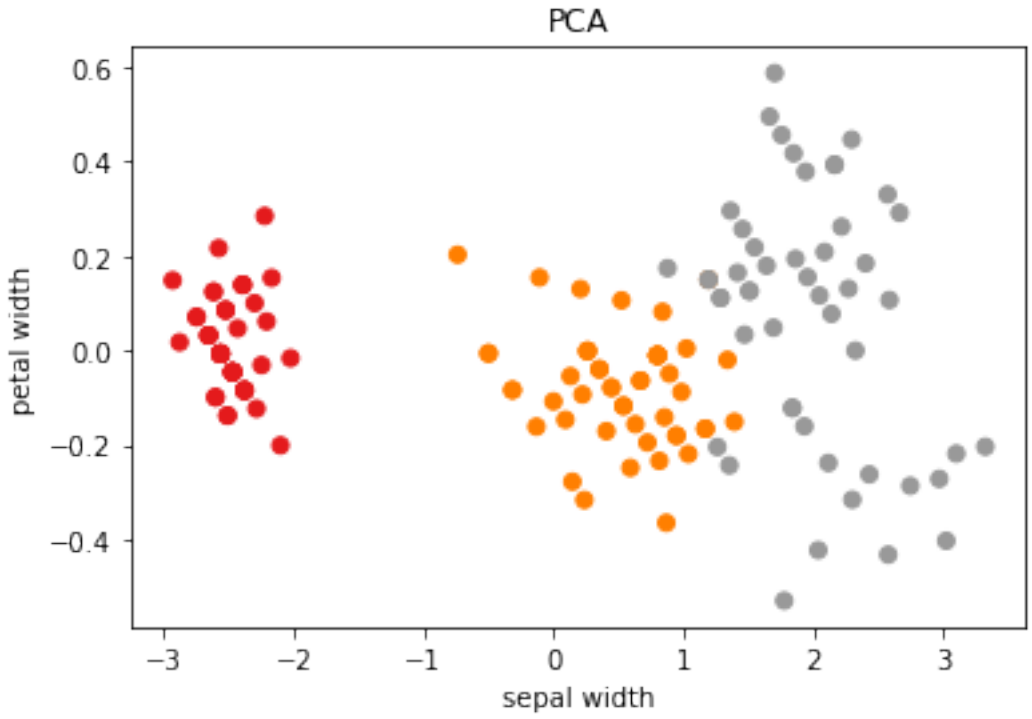
[ ]:
```
print(np.corrcoef(iris.data[:, 0], iris.data[:, 1]))
print(np.corrcoef(iris.data[:, 0], iris.data[:, 2]))
print(np.corrcoef(iris.data[:, 1], iris.data[:, 2]))
```

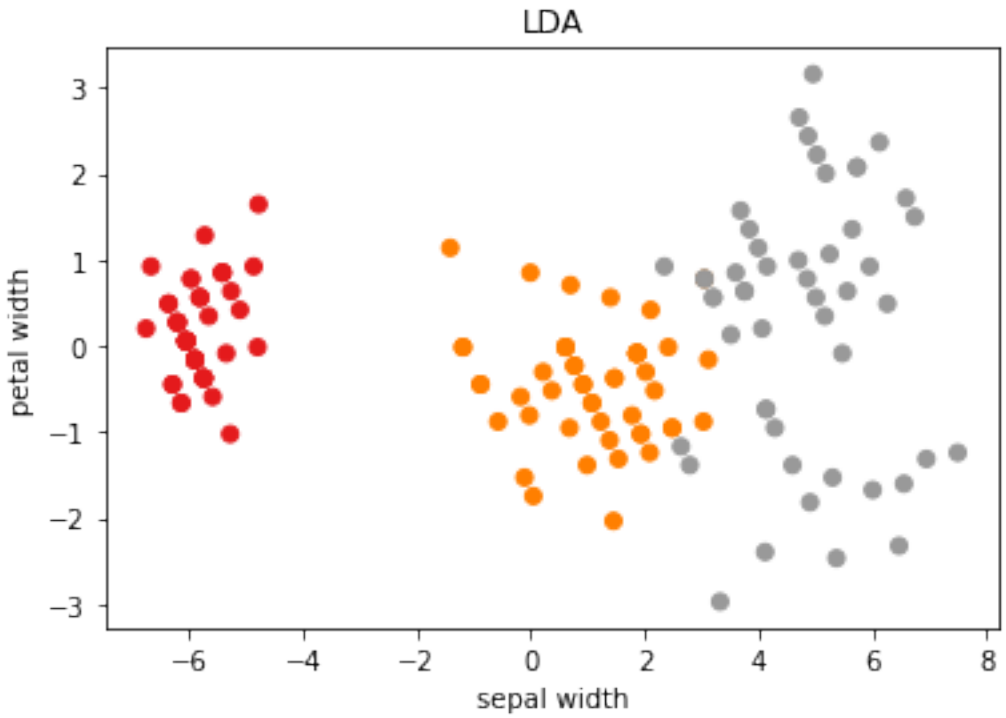
```
[[1.         0.87175378]
 [0.87175378 1.        ]]
[[1.         0.81794113]
 [0.81794113 1.        ]]
[[1.         0.96286543]
 [0.96286543 1.        ]]
```

[ ]:
```
# reduce the dataset to 2 variables
iris.feature_names = ['sepal width', 'petal width']

iris.data = iris.data[:, [1, 2]]
```

[ ]:
```
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

pca = PCA(n_components=2)
lda = LDA(n_components=2)

# PCA
pca.fit(iris.data)
IrisPCA = pca.transform(iris.data)


# LDA
lda.fit(iris.data, iris.target)
IrisLDA = lda.transform(iris.data)
```

```
# PCA
plt.scatter(IrisPCA[:, 0], IrisPCA[:, 1], c=iris.target, cmap=plt.cm.Set1)
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.title('PCA')
plt.show()


# LDA
plt.scatter(IrisLDA[:, 0], IrisLDA[:, 1], c=iris.target, cmap=plt.cm.Set1)
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.title('LDA')
plt.show()
```

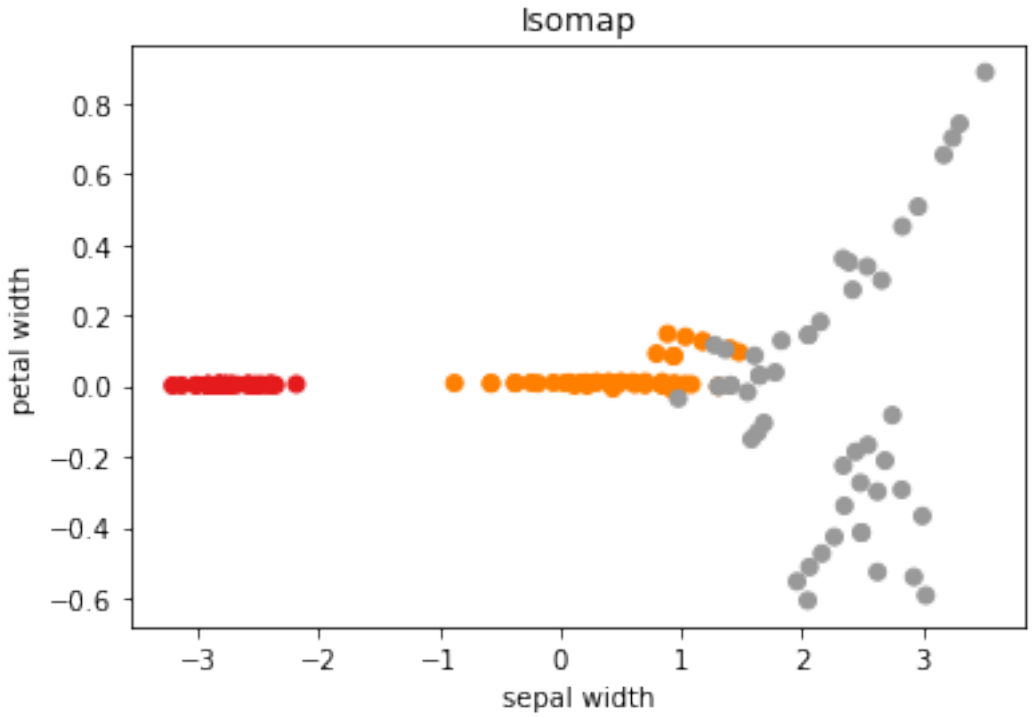




The distributions have the same shape but the ranges are different. On the plot with PCA, the variables are less separated and the data is more condensed. On the plot where LDA was used, the data is more separated and the OX acis take more space. On the Oy axis, the the points is separated similarly. With the LDA, the points are more towards smaller values of `petal width` and bigger values of `sepal width`. This is exppained by the fact that PCA as a technique finds the directions of maximal variance, whereas LDA attempts to find a feature subspace that maximizes class separability.

[ ]:
```
# Use another dimensional reduction technique from scikit-learn
from sklearn.manifold import Isomap


# Isomap
iso = Isomap(n_components=2)
iso.fit(iris.data)
IrisIsomap = iso.transform(iris.data)


# Isomap
plt.scatter(IrisIsomap[:, 0], IrisIsomap[:, 1], c=iris.target, cmap=plt.cm.Set1)
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.title('Isomap')
plt.show()
```

```
/home/ion/.local/lib/python3.9/site-packages/sklearn/manifold/_isomap.py:302:
UserWarning: The number of connected components of the neighbors graph is 2 > 1.
Completing the graph to fit Isomap might be slow. Increase the number of
neighbors to avoid this issue.
  self._fit_transform(X)
/home/ion/.local/lib/python3.9/site-packages/scipy/sparse/_index.py:82:
SparseEfficiencyWarning: Changing the sparsity structure of a csr_matrix is
expensive. lil_matrix is more efficient.
  self._set_intXint(row, col, x.flat[0])
```



With Isomap the data looks a lot more different. The clusters were formed along the OX axis, at least two of them.

0.0.6 F. MNIST dataset

[ ]:
```
# It is possible to load a datset directly from mldata.org which contains a lot␣
↪of available datasets using the function datasets.fetch_mldata
from sklearn.datasets import fetch_openml


# Import the dataset 'MNIST original'
mnist = fetch_openml('mnist_784')
```

[ ]:
```
# Plot the dataset matrix
mnist.data
```

[ ]:
```
       pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  pixel8  pixel9  \
0         0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
1         0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
2         0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
3         0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
4         0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
…         …       …       …       …       …       …       …       …       …
69995     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
69996     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
69997     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
69998     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
69999     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0

       pixel10  …  pixel775  pixel776  pixel777  pixel778  pixel779  \
0          0.0  …       0.0       0.0       0.0       0.0       0.0
1          0.0  …       0.0       0.0       0.0       0.0       0.0
2          0.0  …       0.0       0.0       0.0       0.0       0.0
3          0.0  …       0.0       0.0       0.0       0.0       0.0
4          0.0  …       0.0       0.0       0.0       0.0       0.0
…          …    …       …         …         …         …         …
69995      0.0  …       0.0       0.0       0.0       0.0       0.0
69996      0.0  …       0.0       0.0       0.0       0.0       0.0
69997      0.0  …       0.0       0.0       0.0       0.0       0.0
69998      0.0  …       0.0       0.0       0.0       0.0       0.0
69999      0.0  …       0.0       0.0       0.0       0.0       0.0

       pixel780  pixel781  pixel782  pixel783  pixel784
0           0.0       0.0       0.0       0.0       0.0
1           0.0       0.0       0.0       0.0       0.0
2           0.0       0.0       0.0       0.0       0.0
3           0.0       0.0       0.0       0.0       0.0
4           0.0       0.0       0.0       0.0       0.0
…           …         …         …         …         …
69995       0.0       0.0       0.0       0.0       0.0
69996       0.0       0.0       0.0       0.0       0.0
69997       0.0       0.0       0.0       0.0       0.0
69998       0.0       0.0       0.0       0.0       0.0
69999       0.0       0.0       0.0       0.0       0.0

[70000 rows x 784 columns]
```

[ ]:
```
# show the number of data
mnist.data.shape
```

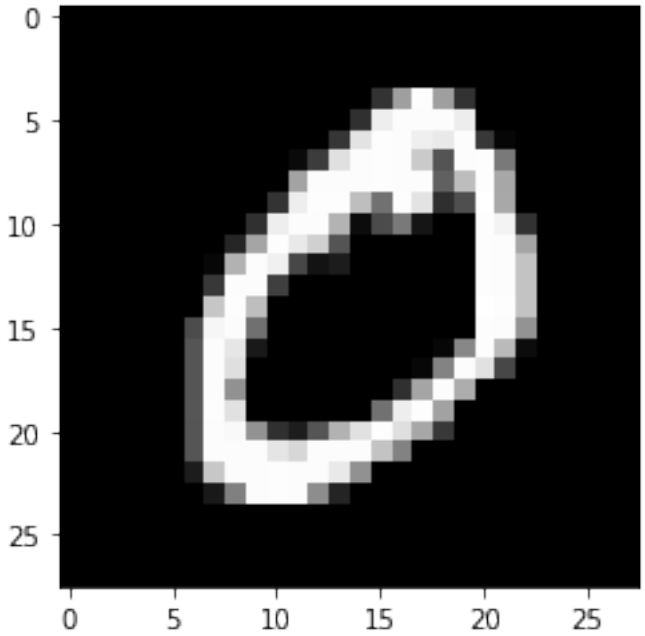
[ ]: (70000, 784)

Show the number of variables The feature names are `pixel{1-784}`

[ ]:
```
# show the number of classes
mnist.target_names
```

[ ]: ['class']

[ ]:
```
from tensorflow.keras.datasets import mnist
%matplotlib inline

(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

# pick a sample to plot
sample = 1
image = X_train[sample]# plot the sample
fig = plt.figure
plt.imshow(image, cmap='gray')
plt.show()
```



[ ]:
```
num = 10
images = X_train[:num]
labels = Y_train[:num]
```

[ ]:
```
num_row = 2
num_col = 5# plot images
fig, axes = plt.subplots(num_row, num_col, figsize=(1.5*num_col,2*num_row))
for i in range(num):
    ax = axes[i//num_col, i%num_col]
    ax.imshow(images[i], cmap='gray')
    ax.set_title('Label: {}'.format(labels[i]))
plt.tight_layout()
plt.show()
```