MASTER IN ENTREPRENEURSHIP
INNOVATION MANAGEMENT
IN COLLABORATION WITH MIT SLOAN

IN COLLABORATION WITH
MIT MANAGEMENT SLOAN SCHOOL

UNIVERSITÀ DEGLI STUDI DI NAPOLI
PARTHENOPE

MASTER MEIM 2021-2022

Digital AI – Unsupervised Learning Hands-on

Lesson given by prof. Alessio Ferone

Overview

- **Python tools for machine learning**
  - **First application**

- Unsupervised learning
  - K-Means

- Agglomerative Clustering and DBSCAN

- Principal Component Analysis

Python Tools

- Python combines the power of **general-purpose programming** languages with the ease of use of domain-specific scripting languages

- Python has **libraries** that provide data scientists with a large array of **general**- and **special-purpose functionality**

- Moreover Python allows to interact directly with the code using a **Jupyter Notebook**

Scikit-learn

- **scikit-learn** is an open source project that contains a number of state-of-the-art **machine learning algorithms**

- scikit-learn is the most prominent Python library for machine learning

- scikit-learn **works** well with a number of **other scientific Python tools**

Scikit-learn

- scikit-learn is built on top of the **NumPy** and **SciPy** scientific Python libraries

- In addition to NumPy and SciPy, **pandas** and **matplotlib** libraries will be also used

Jupyter Notebook

- The **Jupyter Notebook** is an **interactive environment** for running code in the browser

- It is a great tool for exploratory **data analysis** and is widely used by data scientists

- The Jupyter Notebook makes it easy to incorporate **code**, **text**, and **images**

Jupyter Notebook

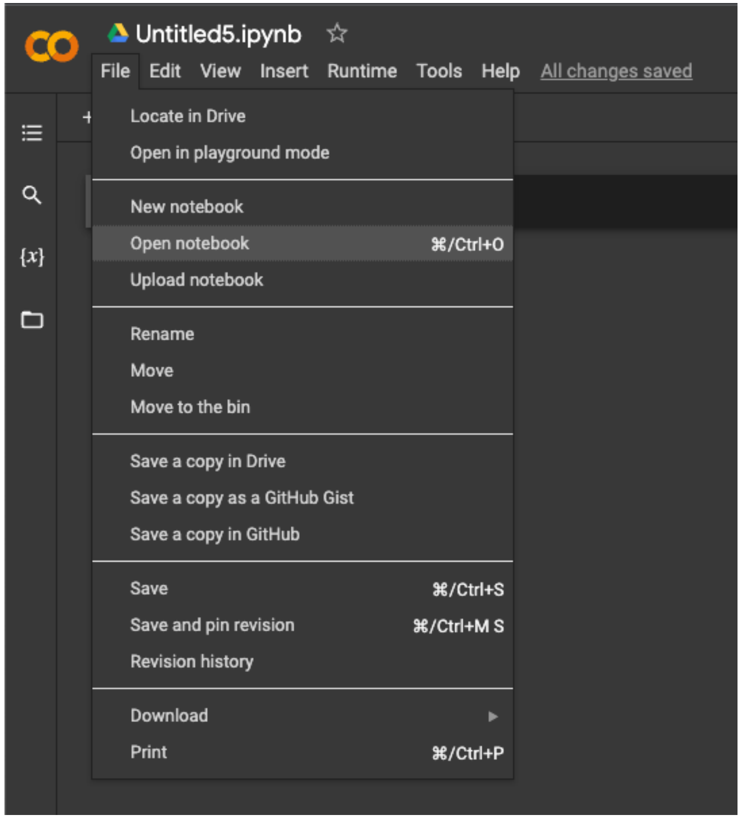
- https://colab.research.google.com

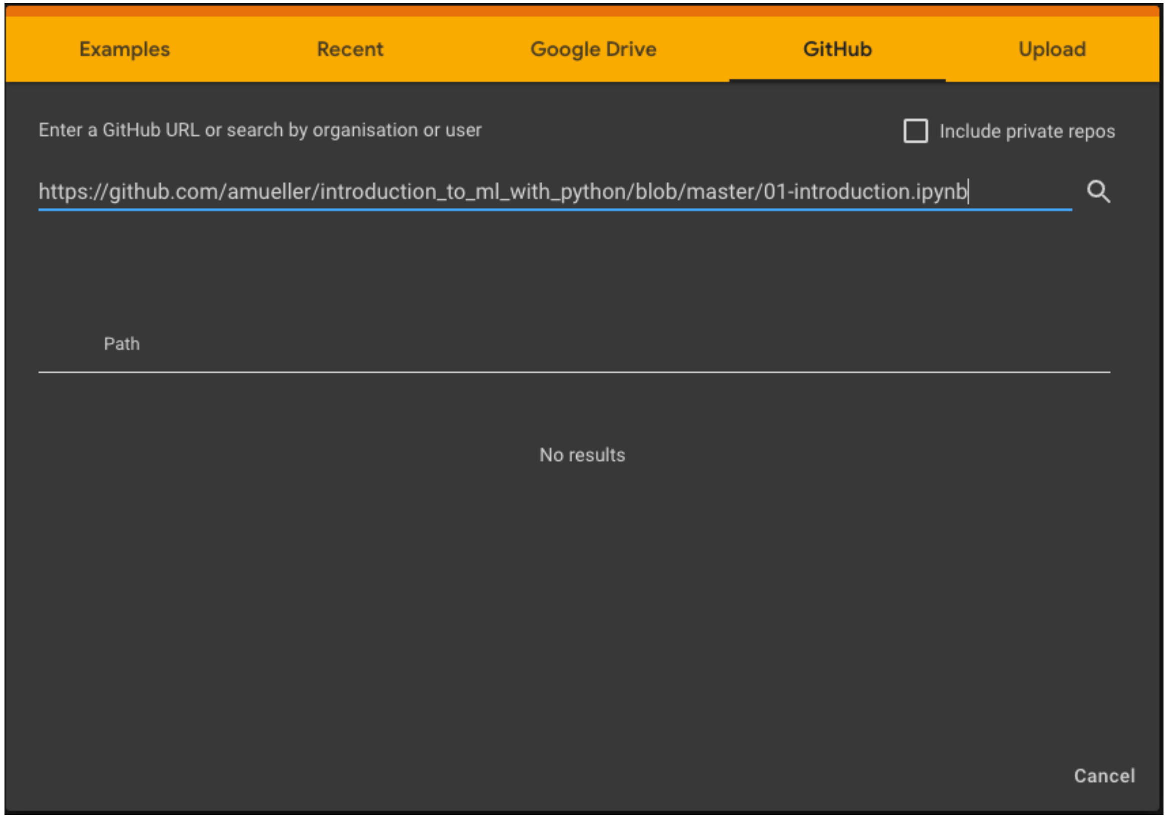
- Create an account or log in

- Open the first notebook (from local o Github)

- https://github.com/amueller/introduction_to_ml_with_python

Jupyter Notebook





NumPy

- **NumPy** is one of the fundamental packages for scientific computing in Python (**multidimensional arrays**, high-level **mathematical functions**, etc.)

- In scikit-learn, the **NumPy array** is the fundamental data structure since it takes in data in the form of NumPy arrays

- The core functionality of **NumPy** is the **ndarray class**, a **multidimensional (n-dimensional) array** of elements of the same type

- Notebook…

NumPy

NumPy

```python
import numpy as np

x = np.array([[1, 2, 3], [4, 5, 6]])
print("x:\n{}".format(x))
```

```
x:
[[1 2 3]
 [4 5 6]]
```

SciPy

- **SciPy** is a collection of **functions** for **scientific computing** in Python (advanced **linear algebra** routines, mathematical **function optimization**, **signal processing**, special mathematical functions, etc)

- **scikit-learn** for implementing its algorithms uses **scipy.sparse** (sparse matrices, that contains mostly zeros)

- Notebook...

SciPy

SciPy

```
from scipy import sparse

# Create a 2D NumPy array with a diagonal of ones, and zeros everywhere else
eye = np.eye(4)
print("NumPy array:\n", eye)
```

```
NumPy array:
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

```
# Convert the NumPy array to a SciPy sparse matrix in CSR format
# Only the nonzero entries are stored
sparse_matrix = sparse.csr_matrix(eye)
print("\nSciPy sparse CSR matrix:\n", sparse_matrix)
```

```
SciPy sparse CSR matrix:
   (0, 0)	1.0
  (1, 1)	1.0
  (2, 2)	1.0
  (3, 3)	1.0
```

```
data = np.ones(4)
row_indices = np.arange(4)
col_indices = np.arange(4)
eye_coo = sparse.coo_matrix((data, (row_indices, col_indices)))
print("COO representation:\n", eye_coo)
```

```
COO representation:
   (0, 0)	1.0
  (1, 1)	1.0
  (2, 2)	1.0
  (3, 3)	1.0
```

Matplotlib

- **matplotlib** is the primary scientific **plotting library** in Python for making **visualizations** such as line charts, histograms, scatter plots

- When working inside the Jupyter Notebook, it is possible to **show figures** directly in the **browser**

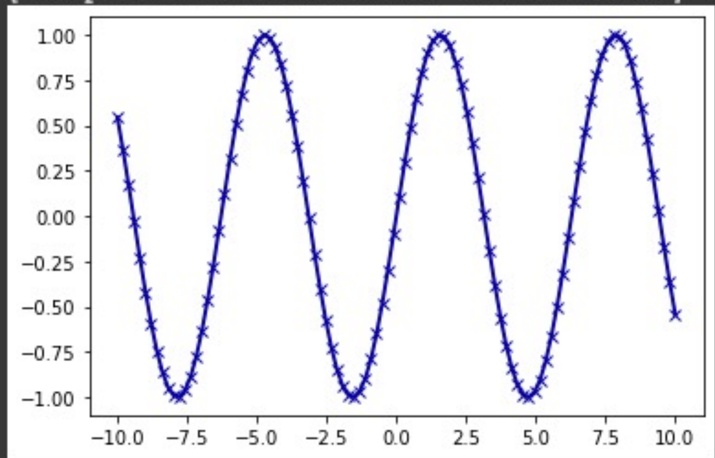
- Notebook…

Matplotlib

matplotlib

```python
%matplotlib inline
import matplotlib.pyplot as plt

# Generate a sequence of numbers from -10 to 10 with 100 steps in between
x = np.linspace(-10, 10, 100)
# Create a second array using sine
y = np.sin(x)
# The plot function makes a line chart of one array against another
plt.plot(x, y, marker="x")
```

```
[<matplotlib.lines.Line2D at 0x1be867b9748>]
```



Pandas

- **Pandas** is a Python library for data analysis built around a **data structure** called the **DataFrame**
- A pandas **DataFrame is a table**, similar to an **Excel spreadsheet**
- Pandas allows **each column** to have a **separate type** (integers, dates, floating-point numbers, and strings)
- Notebook...

Pandas

pandas

```
import pandas as pd

# create a simple dataset of people
data = {'Name': ["John", "Anna", "Peter", "Linda"],
        'Location' : ["New York", "Paris", "Berlin", "London"],
        'Age' : [24, 13, 53, 33]
       }

data_pandas = pd.DataFrame(data)
# IPython.display allows "pretty printing" of dataframes
# in the Jupyter notebook
display(data_pandas)
```

	Name	Location	Age
0	John	New York	24
1	Anna	Paris	13
2	Peter	Berlin	53
3	Linda	London	33

```
# Select all rows that have an age column greater than 30
display(data_pandas[data_pandas.Age > 30])
```

	Name	Location	Age
2	Peter	Berlin	53
3	Linda	London	33

mglearn

- **mglearn** is a library of **utility functions** for plotting and data loading
- **pip install mglearn**
- **import mglearn**
- Notebook…

First application

- A **simple machine learning application** for distinguishing the **species** of some **iris flowers**
- For each iris some **measurement** have been collected: the **length** and **width** of the **petals** and the **length** and **width** of the **sepals**
- Some irises have been **previously classified** by an expert botanist as belonging to the species **setosa**, **versicolor**, or **virginica**
- The goal is to build a **machine learning model** that can **learn** from the measurements whose species is known, in order **to predict** the **species** for a new **iris**

First application

- Because **we have measurements** for which we know the correct species of iris, this is a **supervised learning** problem (**classification**)
- **Every iris** in the dataset (data point) belongs to **one of three classes**
- For a particular **data point**, the **species it belongs** to is called its **label**

Data

- The data we will use for this example is the **Iris dataset**
- It is included in **scikit-learn** in the **datasets** module
- It is possible to **load** it by calling the **load_iris function**
- Notebook...

Data

Meet the Data

```
[ ] from sklearn.datasets import load_iris
    iris_dataset = load_iris()
```

```
[ ] print("Keys of iris_dataset:\n", iris_dataset.keys())
```

```
Keys of iris_dataset:
 dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names', 'filename'])
```

```
print(iris_dataset['DESCR'][:193] + "\n...")
```

```
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, pre
...
```

Data

```
[ ] print("Target names:", iris_dataset['target_names'])

    Target names: ['setosa' 'versicolor' 'virginica']

[ ] print("Feature names:\n", iris_dataset['feature_names'])

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

[ ] print("Type of data:", type(iris_dataset['data']))

    Type of data: <class 'numpy.ndarray'>

[ ] print("Shape of data:", iris_dataset['data'].shape)

    Shape of data: (150, 4)
```

Data

```
[ ] print("First five rows of data:\n", iris_dataset['data'][:5])

First five rows of data:
 [[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]
```

```
[ ] print("Type of target:", type(iris_dataset['target']))

Type of target: <class 'numpy.ndarray'>
```

```
[ ] print("Shape of target:", iris_dataset['target'].shape)

Shape of target: (150,)
```

```
[ ] print("Target:\n", iris_dataset['target'])

Target:
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

Training and Test

- The goal is to build a **machine learning model** from this data that can **predict** the **species** of **iris** for a new set of **measurements**

- To assess the model's **performance**, **new data** for which we have labels is **presented** to the **model**

- To this aim, the **dataset** is **splitted** in two parts: **one part** to **train** the model (**training set**) and the **other part** to assess its **performance** (**test set**)

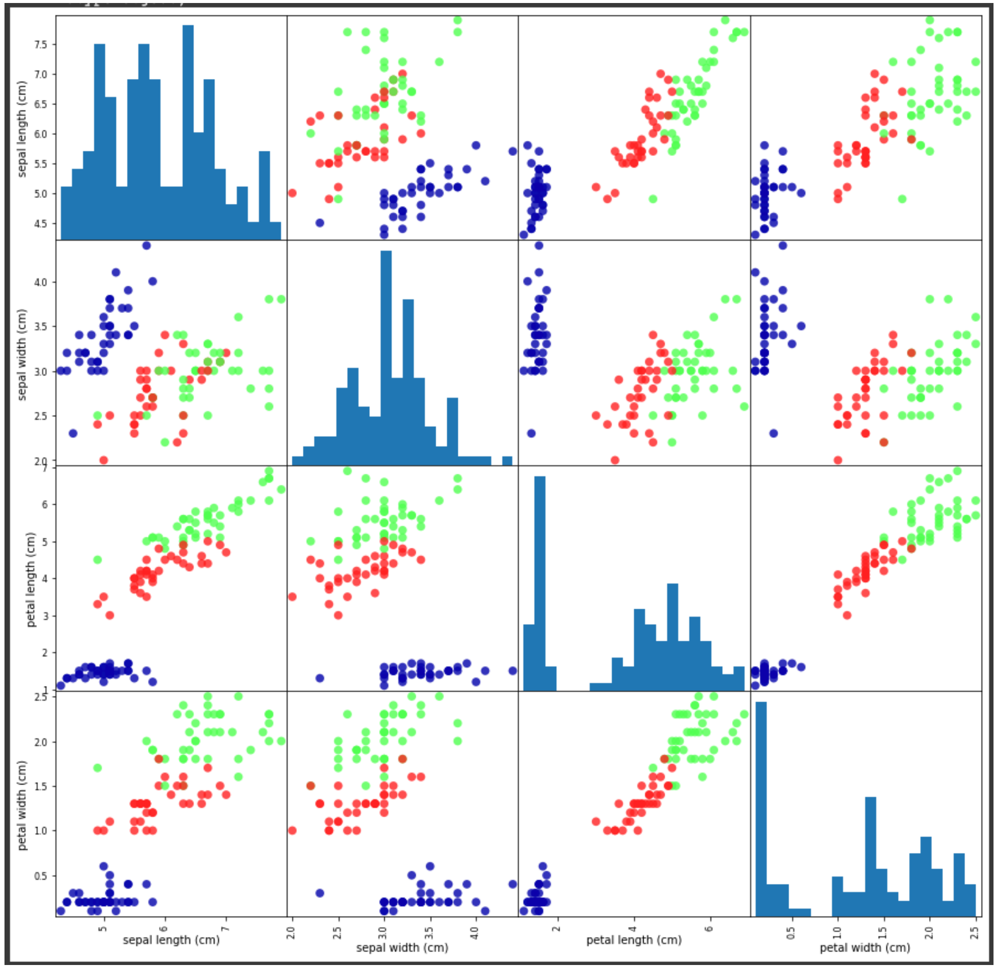
Training and Test

- **scikit-learn** contains a **function** that **shuffles** the dataset and **splits** it
- The **train_test_split** function extracts **75%** of the rows in the data as the **training set** and the remaining **25%** of the data as the **test set**
- Notebook...

Training and Test

```
[ ] from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test = train_test_split(
        iris_dataset['data'], iris_dataset['target'], random_state=0)

[ ] print("X_train shape:", X_train.shape)
    print("y_train shape:", y_train.shape)

    X_train shape: (112, 4)
    y_train shape: (112,)

[ ] print("X_test shape:", X_test.shape)
    print("y_test shape:", y_test.shape)

    X_test shape: (38, 4)
    y_test shape: (38,)
```

Inspect the data
Scatter plot



First ML model

- All **machine learning models** in scikit-learn are implemented in their own classes, which are called **Estimator classes**

- The **knn** object encapsulates the **algorithm** that **will be used** to build the model from the **training** data and the algorithm to **make predictions** on new data points

First ML model

- To build the model on the **training** set, we call the **fit method** of the knn object, which takes as arguments
  - NumPy array **X_train** containing the **training data**
  - NumPy array **y_train** of the corresponding **training labels**

- Notebook...

First ML model

```
[ ]  from sklearn.neighbors import KNeighborsClassifier
     knn = KNeighborsClassifier(n_neighbors=1)
```

```
[ ]  knn.fit(X_train, y_train)

     KNeighborsClassifier(n_neighbors=1)
```

First ML model

- We can now **make predictions** using this model **on new data** for which we might not know the correct labels
- Imagine we **found an iris** in the wild with a **sepal length of 5 cm**, a **sepal width of 2.9 cm**, a **petal length of 1 cm**, and a **petal width of 0.2 cm**
- **What species** of iris **would this be**? We can **put** this **data** into a **NumPy array** and call the **predict method** of the knn object

- Notebook...

First ML model

```
[12] X_new = np.array([[5, 2.9, 1, 0.2]])
     print("X_new.shape:", X_new.shape)

     X_new.shape: (1, 4)
```

```
prediction = knn.predict(X_new)
print("Prediction:", prediction)
print("Predicted target name:",
       iris_dataset['target_names'][prediction])

Prediction: [0]
Predicted target name: ['setosa']
```

First ML model

- The **test set**, created earlier, was **not used** to **build** the **model**, but we do **know** what the **correct species** is **for each iris** in the **test set**
- Therefore, we can **make a prediction for each iris** in the test data and **compare** it against **its label** (the known species)
- We can **measure** how well the **model** works by computing the **accuracy**, which is the **fraction** of flowers for which the **right species** was **predicted**

- Notebook...

First ML model

```
[14] y_pred = knn.predict(X_test)
     print("Test set predictions:\n", y_pred)

     Test set predictions:
      [2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0
      2]

[15] print("Test set score: {:.2f}".format(np.mean(y_pred == y_test)))

     Test set score: 0.97

     print("Test set score: {:.2f}".format(knn.score(X_test, y_test)))

     Test set score: 0.97
```

Summary

- We formulated the task of **predicting** which **species** of iris a particular flower belongs to by **using physical measurements** of the **flower**
- We used a **dataset** of **measurements** that was **annotated** by an **expert** with the **correct species** to build our model, making this a **supervised learning task**
- The possible **species** are called **classes** in the classification problem, and the **species** of a **single iris** is called its **label**

Summary

- The **Iris dataset** consists of **two NumPy arrays**: one containing the **data**, which is referred to as **X** in scikit-learn, and one containing the **correct** or desired **outputs**, which is called **y**
- We **split** our **dataset** into a **training set**, to **build** our **model**, and a **test set**, to **evaluate** how well our **model** will generalize to new, previously unseen data
- We **built the model** by calling the **fit method**, passing the **training data** (X_train) and **training outputs** (y_train) as parameters
- We **evaluated** the **model** using the **score method**, which computes the **accuracy** of the model

Overview

- Python tools for machine learning
  - First application

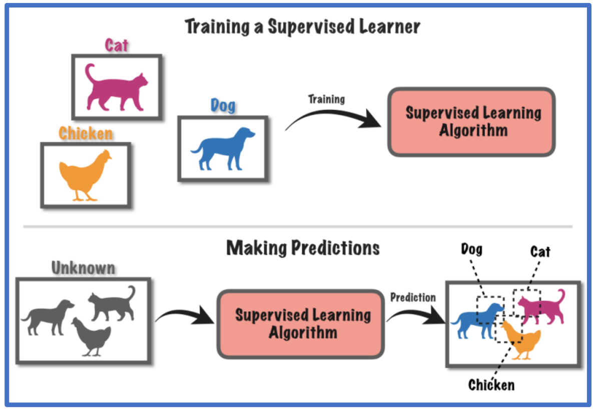
- **Unsupervised learning**
  - **K-Means**

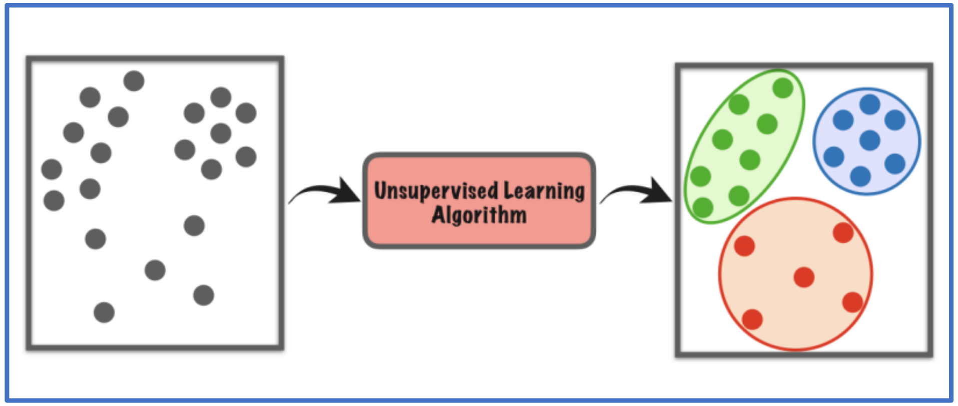
- Agglomerative Clsutering and DBSCAN

- Principal Component Analysis

Unsupervised learning

- **Unsupervised learning** embraces all kinds of machine learning where there is **no known output**, no teacher to instruct the learning algorithm
- In unsupervised learning, the **learning algorithm** is just shown the **input data** and asked to **extract knowledge** from this data





Unsupervised learning: types

- **Two kinds** of unsupervised learning:
  - **transformations of the dataset**
  - **clustering**

- **Unsupervised transformations** of a dataset are algorithms that create a **new representation** of the data which might be **easier** for **humans** or other machine learning **algorithms** to **understand**

- **Clustering algorithms**, on the other hand, **partition data** into distinct **groups** of **similar items**

Clustering: example

- Consider the example of **uploading photos** to a social media site
- In order to **organize your pictures**, the site might want to **group** together **pictures** that show the **same person**
- The **site** doesn't know which pictures show whom, and it **doesn't know** how many **different people** appear in your photo collection
- A possible **approach** would be to **extract all the faces** and **divide** them into **groups** of faces that look **similar**

Clustering: challenges

- A **major challenge** in unsupervised learning is **evaluating** whether the algorithm learned something useful
- **Unsupervised learning** algorithms are usually applied to **data** that does **not** contain any **label** information
- We **don't know** what the **right output** should be
- **Unsupervised algorithms** are used often in an **exploratory setting**, when a data scientist wants **to understand the data** better

Clustering

- **clustering** is the task of **partitioning** the dataset into **groups**, called **clusters**
- The **goal** is to **split** up the **data** in such a way that **points within a single cluster** are very **similar** and **points** in **different clusters** are **different**
- Similarly to classification algorithms, **clustering algorithms assign** (or predict) a **number** to each **data point**, indicating which **cluster** a particular **point belongs to**

K-Means

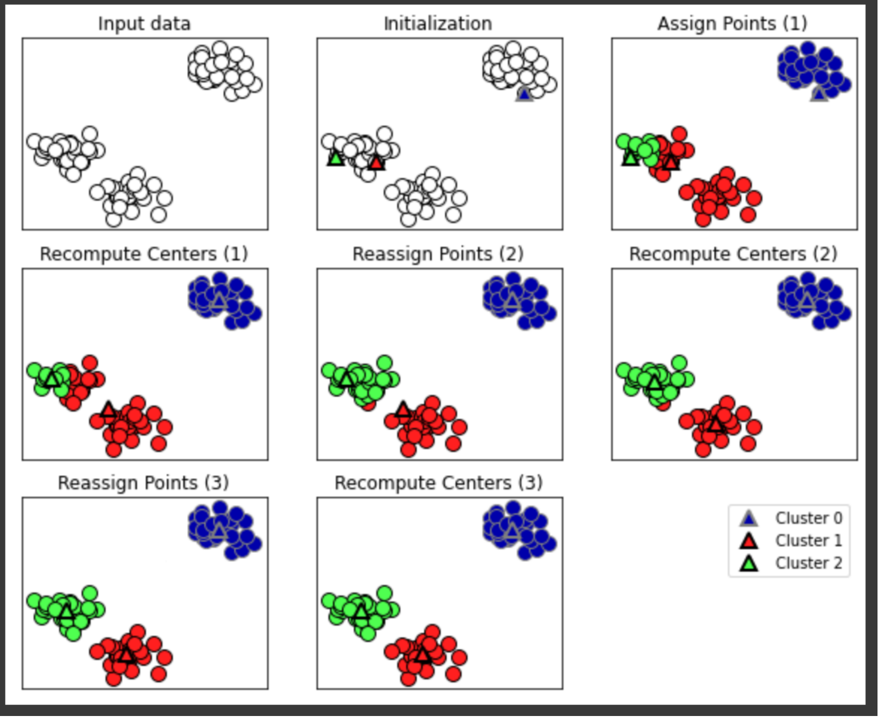
- **k-means clustering** is one of the **simplest** and most **commonly used** clustering algorithms
- It tries to **find cluster centers** that are **representative** of certain **group** of the data
- The algorithm alternates between **two steps**: **assigning** each data **point** to the **closest cluster center**, and then setting each **cluster center** as the **mean** of the data points that are assigned to it
- The algorithm is **finished** when the **assignment** of instances to clusters **no** longer **changes**

K-Means

```
[4] pip install mglearn
```

```
[13] import mglearn
     import matplotlib.pyplot as plt
```

```
mglearn.plots.plot_kmeans_algorithm()
```

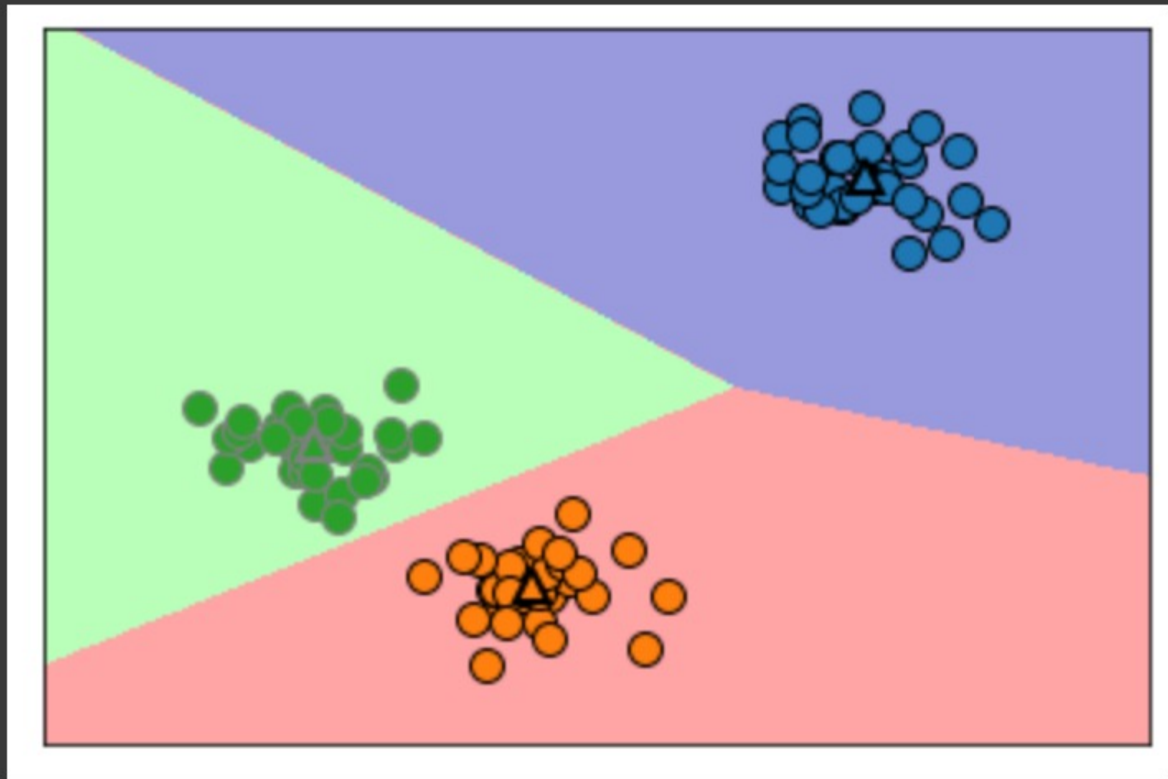


K-Means

- Cluster **centers** are shown as **triangles**
- Data **points** are shown as **circles**
- **Colors** indicate **cluster membership**
- We are looking for **three clusters**, so the algorithm was **initialized** by declaring **three data points** randomly **as cluster centers**
- First, each **data point** is **assigned** to the **closest cluster center**
- Next, the **cluster centers** are **updated** to be the **mean** of the assigned points (the process is repeated two more times)
- After the third iteration, the **assignment** of points to cluster centers remained **unchanged**, so the **algorithm stops**

K-Means

```
mglearn.plots.plot_kmeans_boundaries()
```



K-Means with scikit-learn

```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# generate synthetic two-dimensional data
X, y = make_blobs(random_state=1)

# build the clustering model
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
```

```
KMeans(n_clusters=3)
```

K-Means with scikit-learn

```
print("Cluster memberships:\n{}".format(kmeans.labels_))
```

```
Cluster memberships:
[0 2 2 2 1 1 1 2 0 0 2 2 1 0 1 1 1 0 2 2 1 2 1 0 2 1 1 0 0 1 0 0 1 0 2 1 2
 2 2 1 1 2 0 2 2 1 0 0 0 0 2 1 1 1 0 1 2 2 0 0 2 1 1 2 2 1 0 1 0 2 2 2 1 0
 0 2 1 1 0 2 0 2 2 1 0 0 0 0 2 0 1 0 0 2 2 1 1 0 1 0]
```

K-Means with scikit-learn

- You can also **assign cluster labels** to **new points**, using the **predict** method
- Each **new point** is **assigned** to the **closest cluster center** when predicting, **but** the existing **model** is **not changed**
- Running **predict** on the **training set returns** the same result as **labels_**
- **Clustering** is somewhat **similar to classification**, in that **each item gets a label**
- However, there is **no ground truth**, and consequently the **labels** themselves **have no a priori meaning**

K-Means with scikit-learn

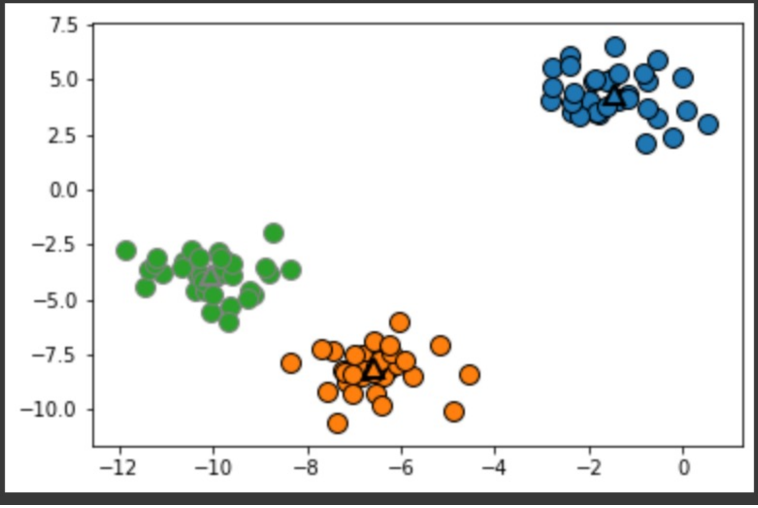
```
print(kmeans.predict(X))
```

```
[0 2 2 2 1 1 1 2 0 0 2 2 1 0 1 1 1 0 2 2 1 2 1 0 2 1 1 0 0 1 0 0 1 0 2 1 2
 2 2 1 1 2 0 2 2 1 0 0 0 0 2 1 1 1 0 1 2 2 0 0 2 1 1 2 2 1 0 1 0 2 2 2 1 0
 0 2 1 1 0 2 0 2 2 1 0 0 0 0 2 0 1 0 0 2 2 1 1 0 1 0]
```

K-Means with scikit-learn

```
mglearn.discrete_scatter(X[:, 0], X[:, 1], kmeans.labels_, markers='o')
mglearn.discrete_scatter(
    kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], [0, 1, 2],
    markers='^', markeredgewidth=2)
```

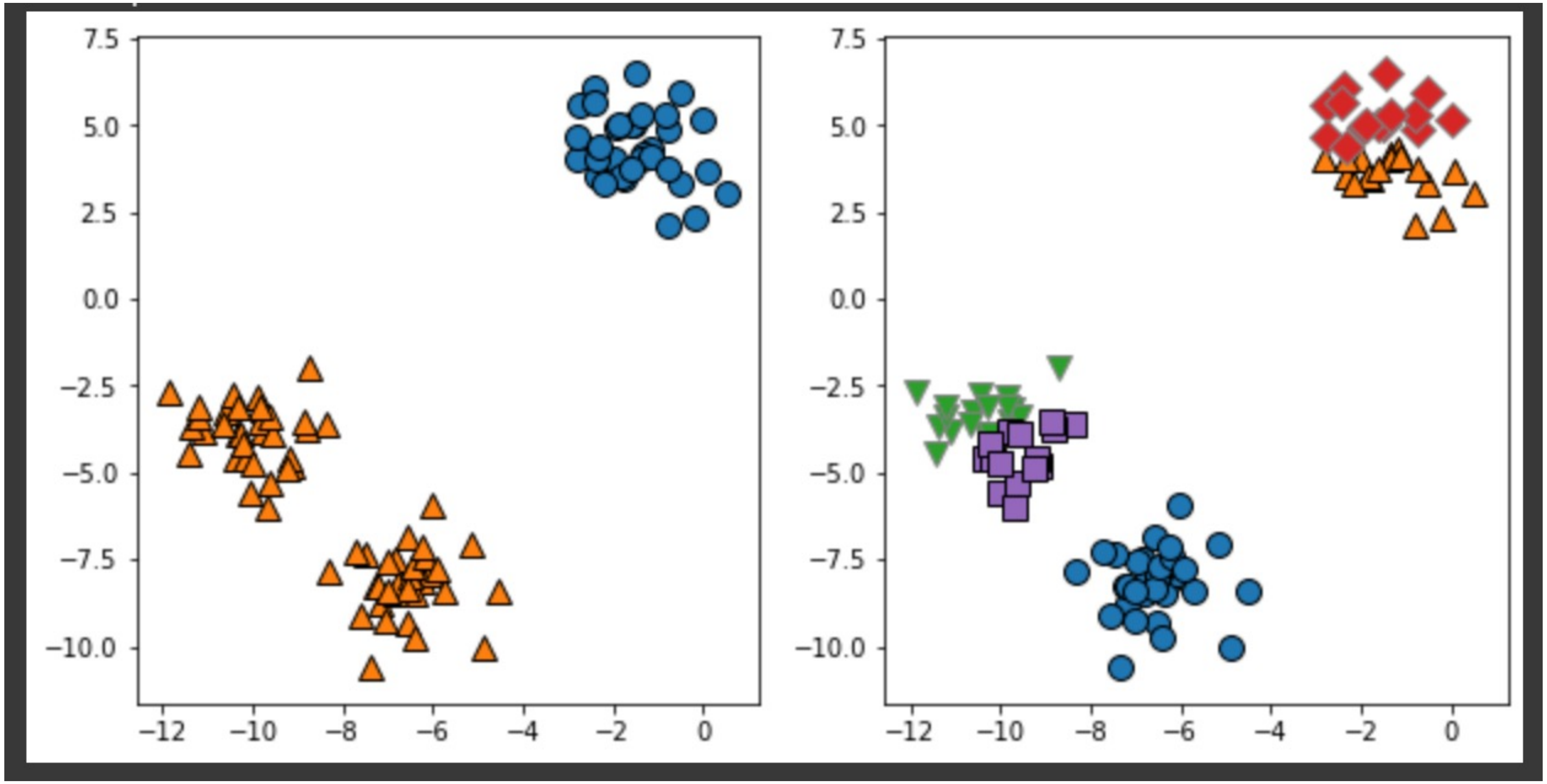
```
[<matplotlib.lines.Line2D at 0x7f83fb958050>,
 <matplotlib.lines.Line2D at 0x7f83fb9585d0>,
 <matplotlib.lines.Line2D at 0x7f83fb958b50>]
```



K-Means with scikit-learn

```python
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

# using two cluster centers:
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
assignments = kmeans.labels_

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignments, ax=axes[0])

# using five cluster centers:
kmeans = KMeans(n_clusters=5)
kmeans.fit(X)
assignments = kmeans.labels_

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignments, ax=axes[1])
```

K-Means with scikit-learn



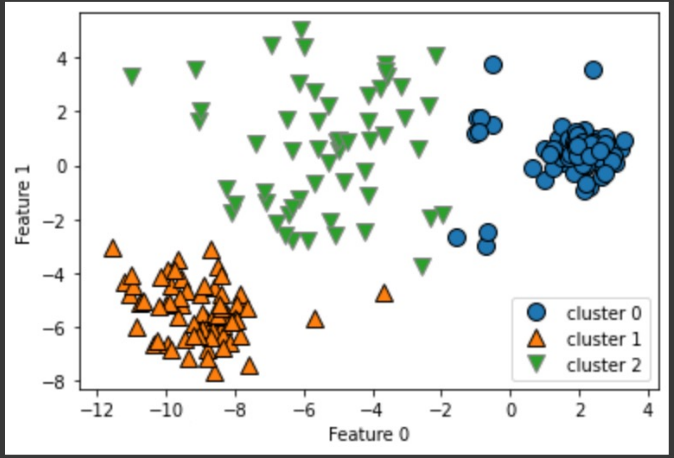
K-Means failure cases

- Even if you know the "**right**" **number of clusters** for a given dataset, **k-means** might **not** always be **able to recover** them
- **Each cluster** is defined solely by its **center**, which means that each cluster is a **convex shape**
- **k-means** also assumes that **all clusters** have the **same** "**diameter**" in some sense: it always draws the **boundary between clusters** to be exactly in the **middle between** the **cluster centers**

K-Means failure cases

```
X_varied, y_varied = make_blobs(n_samples=200,
                                cluster_std=[1.0, 2.5, 0.5],
                                random_state=170)
y_pred = KMeans(n_clusters=3, random_state=0).fit_predict(X_varied)

mglearn.discrete_scatter(X_varied[:, 0], X_varied[:, 1], y_pred)
plt.legend(["cluster 0", "cluster 1", "cluster 2"], loc='best')
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

```
Text(0, 0.5, 'Feature 1')
```



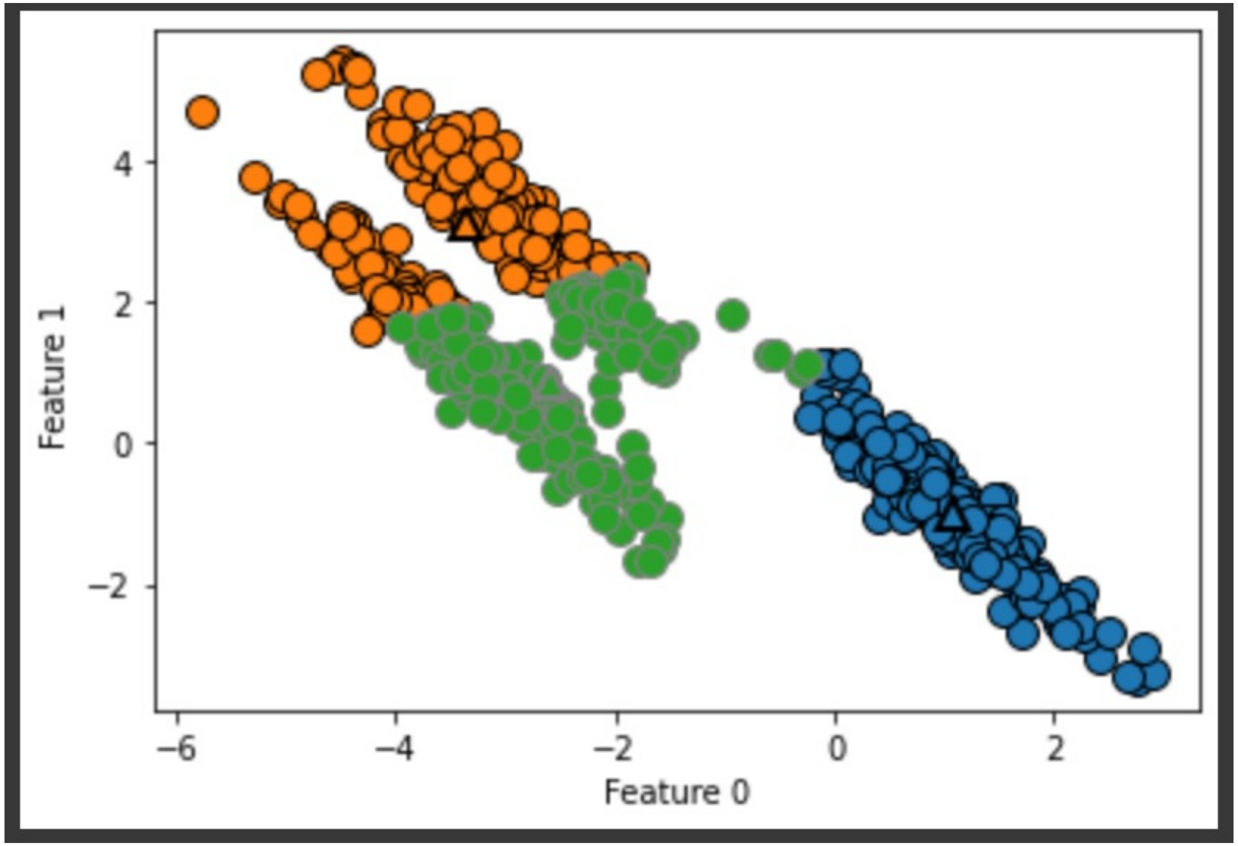
K-Means failure cases

- **k-means** also assumes that **all directions** are **equally important** for **each cluster**
- **k-means** only considers the **distance** to the **nearest cluster center**, it **can't handle groups** that are **stretched** toward the **diagonal**
- **k-means** also **performs poorly** if the **clusters** have more **complex shapes**

K-Means failure cases

```python
# generate some random cluster data
X, y = make_blobs(random_state=170, n_samples=600)
rng = np.random.RandomState(74)

# transform the data to be stretched
transformation = rng.normal(size=(2, 2))
X = np.dot(X, transformation)

# cluster the data into three clusters
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
y_pred = kmeans.predict(X)

# plot the cluster assignments and cluster centers
mglearn.discrete_scatter(X[:, 0], X[:, 1], kmeans.labels_, markers='o')
mglearn.discrete_scatter(
    kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], [0, 1, 2],
    markers='^', markeredgewidth=2)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

K-Means failure cases



K-Means failure cases

```python
# generate synthetic two_moons data (with less noise this time)
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

# cluster the data into two clusters
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
y_pred = kmeans.predict(X)

# plot the cluster assignments and cluster centers
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=mglearn.cm2, s=60, edgecolor='k')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            marker='^', c=[mglearn.cm2(0), mglearn.cm2(1)], s=100, linewidth=2,
            edgecolor='k')
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

K-Means failure cases