**HOW TO USE UNSUPERVISED LEARNING WITH PYTHON TO FIND PATTERNS IN DATA**

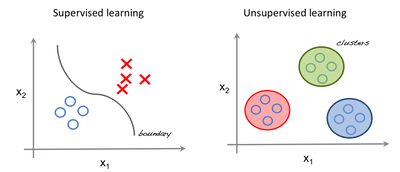
In unsupervised learning, the algorithms are left to themselves to discover interesting structures in the data.

Unsupervised learning is a class of machine learning (ML) techniques used to find patterns in data. The data given to unsupervised algorithms is not labelled, which means only the input variables (x) are given with no corresponding output variables.

## SUPERVISED VS UNSUPERVISED LEARNING

In supervised learning, the system tries to learn from the previous examples given. In unsupervised learning, the system attempts to find the patterns directly from the example given. So, if the dataset is labeled it is a supervised problem, and if the dataset is unlabelled then it is an unsupervised problem.

Below is a simple pictorial representation of how supervised and unsupervised learning can be viewed.



**Supervised vs. Unsupervised Learning**

The left image an example of supervised learning (we use regression techniques to find the best fit line between the features). In unsupervised learning the inputs are segregated based on features and the prediction is based on which cluster it belonged to.

**Important Terminology**

* **Feature**: An input variable used in making predictions.
* **Predictions:**A model’s output when provided with an input example.
* **Example**: One row of a dataset. An example contains one or more features and possibly a label.
* **Label:**Result of the feature.

## PREPARING DATA FOR UNSUPERVISED LEARNING

For our example, we'll use the Iris dataset to make predictions. The dataset contains a set of 150 records under four attributes  —  petal length, petal width, sepal length, sepal width, and three iris classes: setosa, virginica and versicolor. We'll feed the four features of our flower to the unsupervised algorithm and it will predict which class the iris belongs to.

We use the scikit-learn library in Python to load the Iris dataset and matplotlib for data visualization. Below is the code snippet for exploring the dataset.

Example.py

# Importing Modules

from sklearn import datasets

import matplotlib.pyplot as plt

# Loading dataset

iris\_df = datasets.load\_iris()

# Available methods on dataset

print('dataset:',dir(iris\_df))

# Features

print('features name:',iris\_df.feature\_names)

# Targets

print('target :',iris\_df.target)

# Target Names

print('target name :',iris\_df.target\_names)

label = {0: 'red', 1: 'blue', 2: 'green'}

print(iris\_df)

# Dataset Slicing

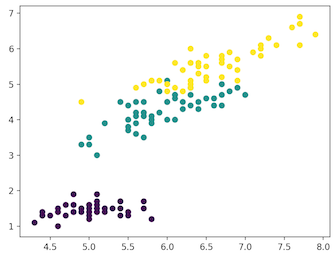
x\_axis = iris\_df.data[:, 0] # Sepal Length

y\_axis = iris\_df.data[:, 2] # Sepal Width

# Plotting

plt.scatter(x\_axis, y\_axis, c=iris\_df.target)

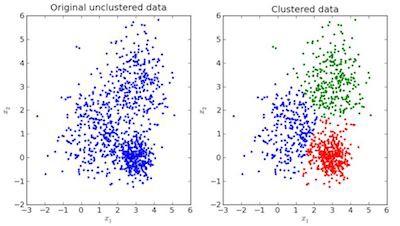
plt.show()



VIOLET: SETOSA, GREEN: VERSICOLOR, YELLOW: VIRGINICA

## CLUSTERING

In clustering, the data is divided into several groups with similar traits.



In the image above, the left is raw data without classification, while the right is clustered based on its features. When an input is given which is to be predicted then it checks in the cluster it belongs to based on its features, and the prediction is made.

### K-MEANS CLUSTERING IN PYTHON

K-means clustering is an iterative clustering algorithm that aims to find local maxima in each iteration. Initially, desired number of clusters are chosen. In our example, we know there are three classes involved, so we program the algorithm to group the data into three classes by passing the parameter “n\_clusters” into our k-means model. Randomly, three points (inputs) are assigned into three clusters. Based on the centroid distance between each point, the next given inputs are segregated into respected clusters and the centroids are re-computed for all the clusters.

Each centroid of a cluster is a collection of feature values which define the resulting groups. Examining the centroid feature weights can be used to qualitatively interpret what kind of group each cluster represent.

We import the k-means model from scikit-learn library, fit out features and predict.

K-means implementation in Python : clustering\_iris.py

# Importing Modules

from sklearn import datasets

from sklearn.cluster import KMeans

# Loading dataset

iris\_df = datasets.load\_iris()

# Declaring Model

model = KMeans(n\_clusters=3)

# Fitting Model

model.fit(iris\_df.data)

# Predicitng a single input

predicted\_label = model.predict([[7.2, 3.5, 0.8, 1.6]])

# Prediction on the entire data

all\_predictions = model.predict(iris\_df.data)

# Printing Predictions

print(predicted\_label)

print(all\_predictions)

### HIERARCHICAL CLUSTERING

As its name implies, hierarchical clustering is an algorithm that builds a hierarchy of clusters. This algorithm begins with all the data assigned to a cluster, then the two closest clusters are joined into the same cluster. The algorithm ends when only a single cluster is left.

The completion of hierarchical clustering can be shown using dendrogram. Now let’s look at an example of hierarchical clustering using grain data. The dataset can be found here.

Hierarchical clustering implementation in Python: hierchical-clustering.py

# Importing Modules

from scipy.cluster.hierarchy import linkage, dendrogram

import matplotlib.pyplot as plt

import pandas as pd

# Reading the DataFrame

seeds\_df = pd.read\_csv(

"https://raw.githubusercontent.com/vihar/unsupervised-learning-with-python/master/seeds-less-rows.csv")

# Remove the grain species from the DataFrame, save for later

varieties = list(seeds\_df.pop('grain\_variety'))

# Extract the measurements as a NumPy array

samples = seeds\_df.values

"""

Perform hierarchical clustering on samples using the

linkage() function with the method='complete' keyword argument.

Assign the result to mergings.

"""

mergings = linkage(samples, method='complete')

"""

Plot a dendrogram using the dendrogram() function on mergings,

specifying the keyword arguments labels=varieties, leaf\_rotation=90,

and leaf\_font\_size=6.

"""

dendrogram(mergings,

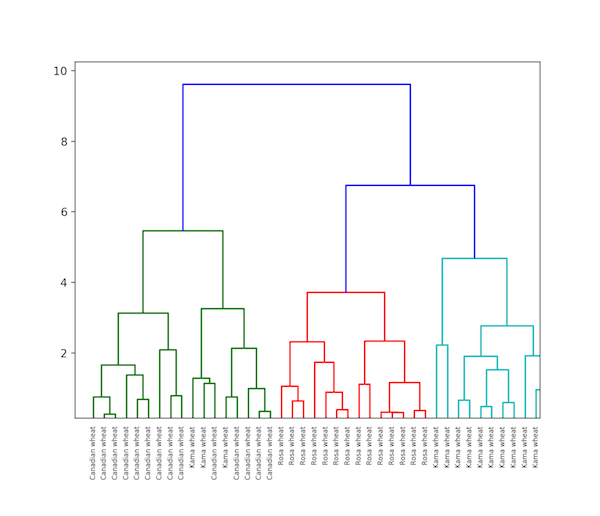
labels=varieties,

leaf\_rotation=90,

leaf\_font\_size=6,

)

plt.show()



### DIFFERENCE BETWEEN K-MEANS AND HIERARCHICAL CLUSTERING

* Hierarchical clustering can’t handle big data very well but k-means clustering can. This is because the time complexity of k-means is linear i.e. O(n) while that of hierarchical clustering is quadratic i.e. O(n2).
* K-means clustering starts with an arbitrary choice of clusters, and the results generated by running the algorithm multiple times might differ. Results are reproducible in hierarchical clustering.
* K-means is found to work well when the shape of the clusters is hyperspherical (like a circle in 2D or a sphere in 3D).
* K-means doesn't allow noisy data, while hierarchical clustering can directly use the noisy dataset for clustering.

### T-SNE CLUSTERING

One of the unsupervised learning methods for visualization is **t-distributed stochastic neighbor embedding, or t-SNE.**It maps high-dimensional space into a two or three-dimensional space which can then be visualized. Specifically, it models each high-dimensional object by a two- or three-dimensional point in such a way that similar objects are modeled by nearby points and dissimilar objects are modeled by distant points with high probability.

t-SNE Implementation in Python on Iris dataset: t\_sne\_clustering.py

# Importing Modules

from sklearn import datasets

from sklearn.manifold import TSNE

import matplotlib.pyplot as plt

# Loading dataset

iris\_df = datasets.load\_iris()

# Defining Model

model = TSNE(learning\_rate=100)

# Fitting Model

transformed = model.fit\_transform(iris\_df.data)

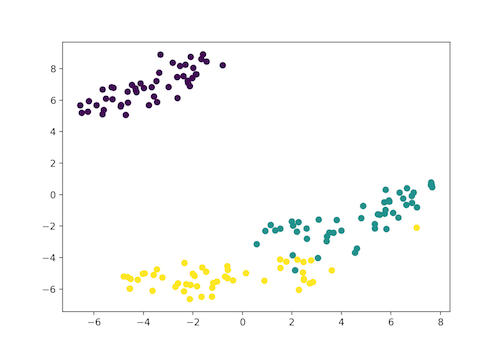
# Plotting 2d t-Sne

x\_axis = transformed[:, 0]

y\_axis = transformed[:, 1]

plt.scatter(x\_axis, y\_axis, c=iris\_df.target)

plt.show()

Violet: Setosa, Green: Versicolor, Yellow: Virginica

Here, the Iris dataset has four features (4d) and is transformed and represented in the two-dimensional figure. Similarly, t-SNE model can be applied to a dataset which has n-features.

### DBSCAN CLUSTERING

Density-based spatial clustering of applications with noise, or DBSCAN, is a popular clustering algorithm used as a replacement for k-means in predictive analytics. To run it doesn’t require an input for the number of clusters but it does need to tune two other parameters.

The scikit-learn implementation provides a default for the eps and min\_samples parameters, but you’re generally expected to tune those. The eps parameter is the maximum distance between two data points to be considered in the same neighborhood. The min\_samples parameter is the minimum amount of data points in a neighborhood to be considered a cluster.

DBSCAN clustering in Python: dbscan.py

# Importing Modules

from sklearn.datasets import load\_iris

import matplotlib.pyplot as plt

from sklearn.cluster import DBSCAN

from sklearn.decomposition import PCA

# Load Dataset

iris = load\_iris()

# Declaring Model

dbscan = DBSCAN()

# Fitting

dbscan.fit(iris.data)

# Transoring Using PCA

pca = PCA(n\_components=2).fit(iris.data)

pca\_2d = pca.transform(iris.data)

# Plot based on Class

for i in range(0, pca\_2d.shape[0]):

if dbscan.labels\_[i] == 0:

c1 = plt.scatter(pca\_2d[i, 0], pca\_2d[i, 1], c='r', marker='+')

elif dbscan.labels\_[i] == 1:

c2 = plt.scatter(pca\_2d[i, 0], pca\_2d[i, 1], c='g', marker='o')

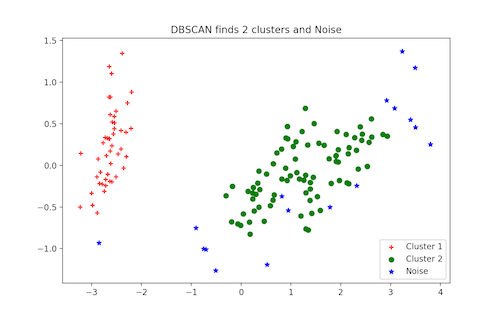
elif dbscan.labels\_[i] == -1:

c3 = plt.scatter(pca\_2d[i, 0], pca\_2d[i, 1], c='b', marker='\*')

plt.legend([c1, c2, c3], ['Cluster 1', 'Cluster 2', 'Noise'])

plt.title('DBSCAN finds 2 clusters and Noise')

plt.show()

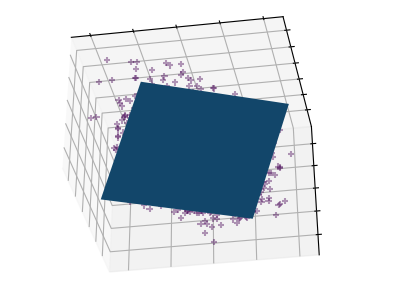
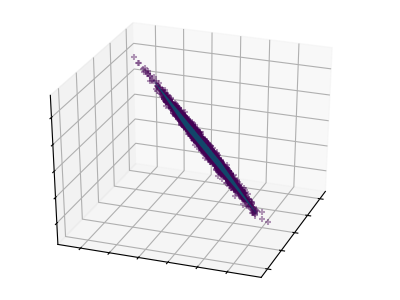
DBSCAN Clustering

## MORE UNSUPERVISED LEARNING TECHNIQUES

* Principal component analysis (PCA)
* Anomaly detection
* Autoencoders
* Deep belief nets
* Hebbian learning
* Generative adversarial networks (GANs)
* Self-organizing maps

### Principal component analysis: PCA

[Principal component analysis (PCA)](https://scikit-learn.org/stable/modules/decomposition.html#pca) selects the successive components that explain the maximum variance in the signal.

[](https://scikit-learn.org/stable/auto_examples/decomposition/plot_pca_3d.html) [](https://scikit-learn.org/stable/auto_examples/decomposition/plot_pca_3d.html)

The point cloud spanned by the observations above is very flat in one direction: one of the three univariate features can almost be exactly computed using the other two. PCA finds the directions in which the data is not flat

When used to transform data, PCA can reduce the dimensionality of the data by projecting on a principal subspace.

**>>>** *# Create a signal with only 2 useful dimensions*

**>>>** x1 = np.random.normal(size=100)

**>>>** x2 = np.random.normal(size=100)

**>>>** x3 = x1 + x2

**>>>** X = np.c\_[x1, x2, x3]

**>>> from** **sklearn** **import** decomposition

**>>>** pca = decomposition.PCA()

**>>>** pca.fit(X)

PCA()

**>>>** print(pca.explained\_variance\_)

[ 2.18565811e+00 1.19346747e+00 8.43026679e-32]

**>>>** *# As we can see, only the 2 first components are useful*

**>>>** pca.n\_components = 2

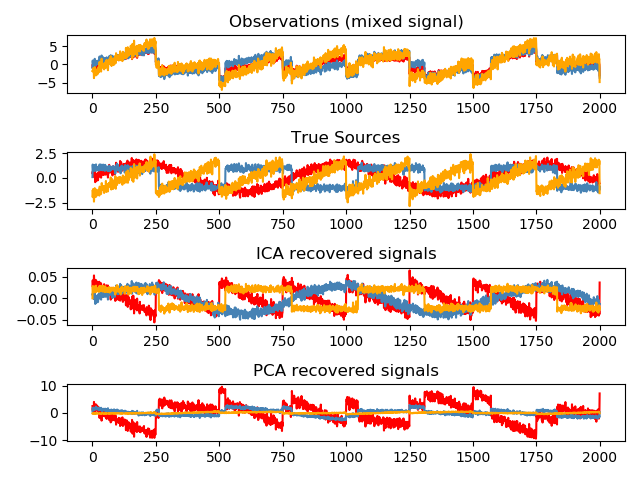
**>>>** X\_reduced = pca.fit\_transform(X)

**>>>** X\_reduced.shape

(100, 2)

### Independent Component Analysis: ICA

[Independent component analysis (ICA)](https://scikit-learn.org/stable/modules/decomposition.html#ica) selects components so that the distribution of their loadings carries a maximum amount of independent information. It is able to recover **non-Gaussian** independent signals:

[](https://scikit-learn.org/stable/auto_examples/decomposition/plot_ica_blind_source_separation.html)

**>>>** *# Generate sample data*

**>>> import** **numpy** **as** **np**

**>>> from** **scipy** **import** signal

**>>>** time = np.linspace(0, 10, 2000)

**>>>** s1 = np.sin(2 \* time) *# Signal 1 : sinusoidal signal*

**>>>** s2 = np.sign(np.sin(3 \* time)) *# Signal 2 : square signal*

**>>>** s3 = signal.sawtooth(2 \* np.pi \* time) *# Signal 3: saw tooth signal*

**>>>** S = np.c\_[s1, s2, s3]

**>>>** S += 0.2 \* np.random.normal(size=S.shape) *# Add noise*

**>>>** S /= S.std(axis=0) *# Standardize data*

**>>>** *# Mix data*

**>>>** A = np.array([[1, 1, 1], [0.5, 2, 1], [1.5, 1, 2]]) *# Mixing matrix*

**>>>** X = np.dot(S, A.T) *# Generate observations*

**>>>** *# Compute ICA*

**>>>** ica = decomposition.FastICA()

**>>>** S\_ = ica.fit\_transform(X) *# Get the estimated sources*

**>>>** A\_ = ica.mixing\_.T

**>>>** np.allclose(X, np.dot(S\_, A\_) + ica.mean\_)

True