



# Machine Learning

# Algorithms

## Scikit Learn

*Hichem Felouat*  
*hichemfel@gmail.com*



<https://www.linkedin.com/in/hichemfelouat>



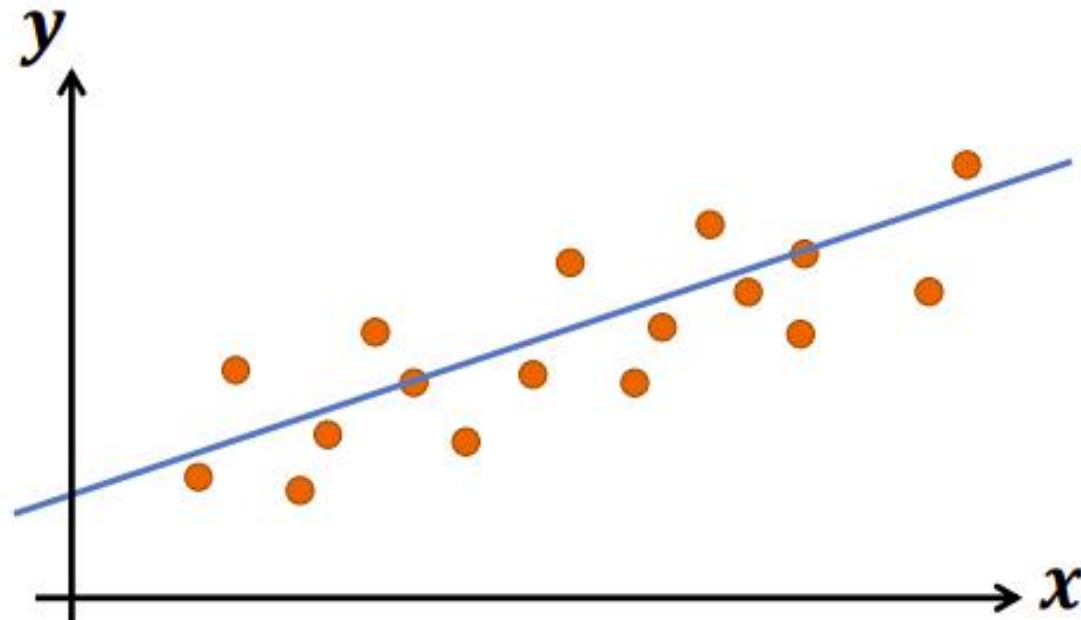
# Supervised learning

# Regression

# 1. Regression: Linear regression

**Linear regression** performs the task to predict a dependent variable value (**y**) based on a given independent variable (**x**). So, this regression technique **finds out a linear relationship** between **x** (input) and **y** (output).

$$Y = aX + b$$

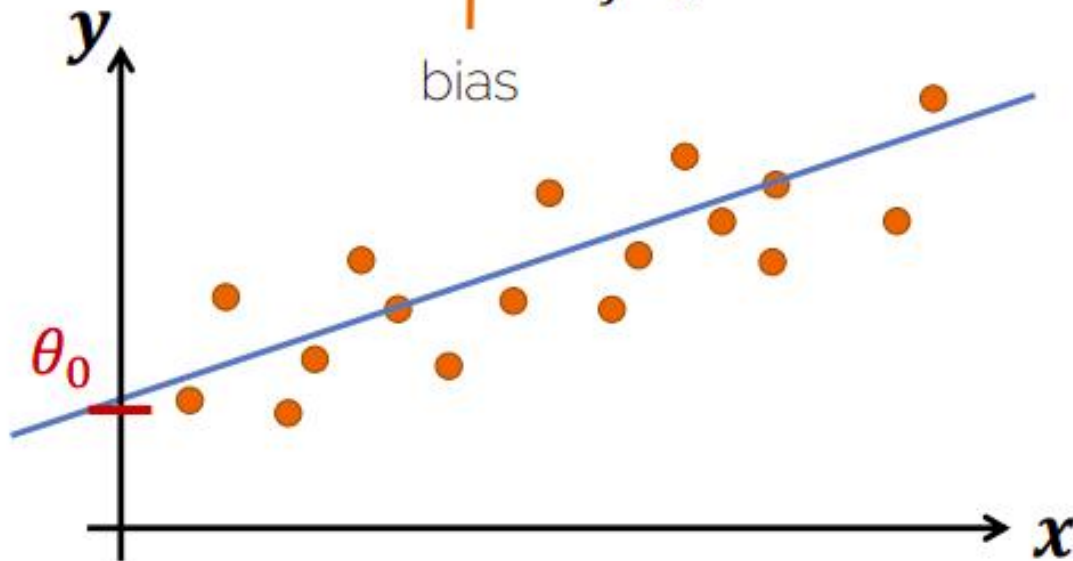


# 1. Regression: Linear regression


A linear model is expressed in the form :

$$\hat{y}_i = \boxed{\theta_0} + \sum_{j=1}^d x_{ij}\theta_j = \theta_0 + x_{i1}\theta_1 + x_{i2}\theta_2 + \dots + x_{id}\theta_d$$

↑  
bias



# 1. Regression: Linear regression

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix} = \theta_0 + \begin{bmatrix} x_{11} & \cdots & x_{1d} \\ x_{21} & \cdots & x_{2d} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nd} \end{bmatrix} \cdot \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix}$$


$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1d} \\ 1 & x_{21} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{nd} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{bmatrix} \Rightarrow \hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta}$$

# 1. Regression: Linear regression

$$\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta}$$

Prediction

Input features  
(one sample has  $d$  features)

Model parameters  
( $d$  weights and 1 bias)

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1d} \\ 1 & x_{21} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{nd} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{bmatrix}$$

# 1. Regression: Linear regression

How do we  
obtain the  
model?



Temperature  
of the building

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \end{bmatrix} = \begin{bmatrix} 1 & 25 & 50 & 2 & 50 \\ 1 & -10 & 50 & 0 & 10 \end{bmatrix} \cdot \begin{bmatrix} 0.2 \\ 0.64 \\ 0 \\ 1 \\ 0.14 \end{bmatrix}$$



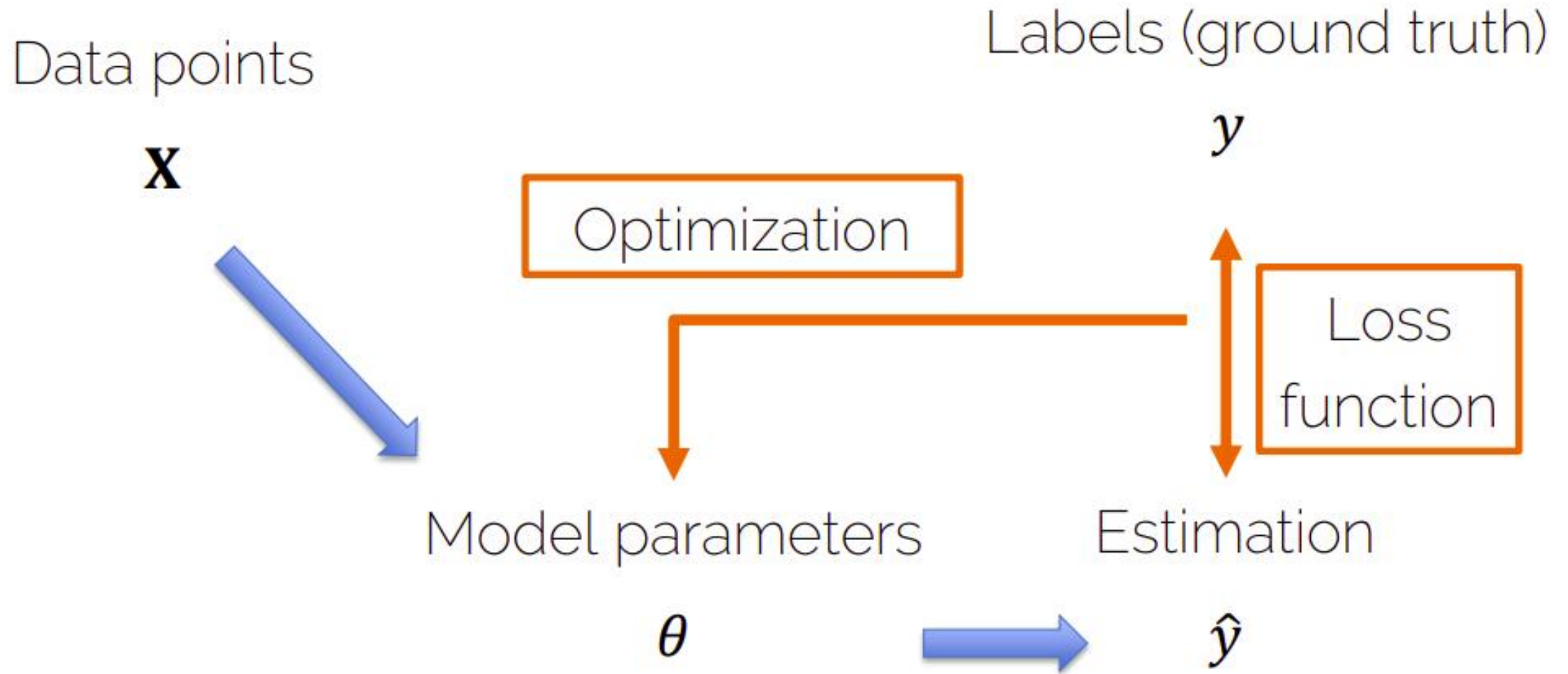
# 1. Regression: Linear regression

**Linear regression** refers to a model that assumes a linear relationship between **input** variables and the **target** variable.

With a **single input** variable, this relationship is a **line**, and with **higher dimensions**, this relationship can be thought of as a **hyperplane** that connects the input variables to the target variable. The **coefficients** of the model are found via an **optimization process** that seeks to minimize the sum squared error between the predictions (**yp**) and the expected target values (**y**).

$$loss = \sum_{i=0}^n (y_i - yp_i)^2$$

# 1. Regression: Linear regression



```

from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets, linear_model
from sklearn import metrics
import numpy as np

dat = datasets.load_boston()
X = dat.data
Y = dat.target
print("Examples = ", X.shape, ", Labels = ", Y.shape)

fig, ax = plt.subplots(figsize=(12,8))
ax.scatter(X[:,0], Y, edgecolors=(0, 0, 0))
ax.plot([Y.min(), Y.max()], [Y.min(), Y.max()], 'k--', lw=4)
ax.set_xlabel('F')
ax.set_ylabel('Y')
plt.show()

fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X[:, 0], X[:, 1], Y, c='b', marker='o', cmap=plt.cm.Set1, edgecolor='k', s=40)
ax.set_title("My Data")
ax.set_xlabel("F1")
ax.w_xaxis.set_ticklabels([])
ax.set_ylabel("F2")
ax.w_yaxis.set_ticklabels([])
ax.set_zlabel("Y")
ax.w_zaxis.set_ticklabels([])
plt.show()

X_train, X_test, Y_train, Y_test = train_test_split(X,
                                                    Y, test_size= 0.20, random_state=100)
print("X_train = ", X_train.shape, ", Y_test = ", Y_test.shape)
regressor = linear_model.LinearRegression()
regressor.fit(X_train, Y_train)
predicted = regressor.predict(X_test)

```

```

import pandas as pd
df = pd.DataFrame({'Actual': Y_test.flatten(), 'Predicted': predicted.flatten()})
print(df)

df1 = df.head(25)
df1.plot(kind='bar',figsize=(12,8))
plt.grid(which='major', linestyle='-', linewidth='0.5', color='green')
plt.grid(which='minor', linestyle=':', linewidth='0.5', color='black')
plt.show()

predicted_all = regressor.predict(X)
fig, ax = plt.subplots(figsize=(12,8))
ax.scatter(X[:,0], Y, edgecolors=(0, 0, 1))
ax.scatter(X[:,0], predicted_all, edgecolors=(1, 0, 0))
ax.set_xlabel('Measured')
ax.set_ylabel('Predicted')
plt.show()

fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X[:, 0], X[:, 1], Y, c='b', marker='o', cmap=plt.cm.Set1, edgecolor='k', s=40)
ax.scatter(X[:, 0], X[:, 1], predicted_all, c='r', marker='o', cmap=plt.cm.Set1, edgecolor='k',
s=40)
ax.set_title("My Data")
ax.set_xlabel("F1")
ax.w_xaxis.set_ticklabels([])
ax.set_ylabel("F2")
ax.w_yaxis.set_ticklabels([])
ax.set_zlabel("Y")
ax.w_zaxis.set_ticklabels([])
plt.show()

print('Mean Absolute Error   : ', metrics.mean_absolute_error(Y_test, predicted))
print('Mean Squared Error    : ', metrics.mean_squared_error(Y_test, predicted))
print('Root Mean Squared Error: ', np.sqrt(metrics.mean_squared_error(Y_test, predicted)))

```

# 1. Regression: Learning Curves

```
def plot_learning_curves(model, X, y):  
    from sklearn.metrics import mean_squared_error  
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)  
    train_errors, val_errors = [], []  
    for m in range(1, len(X_train)):  
        model.fit(X_train[:m], y_train[:m])  
        y_train_predict = model.predict(X_train[:m])  
        y_val_predict = model.predict(X_val)  
        train_errors.append(mean_squared_error(y_train_predict, y_train[:m]))  
        val_errors.append(mean_squared_error(y_val_predict, y_val))  
  
    fig, ax = plt.subplots(figsize=(12,8))  
    ax.plot(np.sqrt(train_errors), "r+", linewidth=2, label="train")  
    ax.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")  
    ax.legend(loc='upper right', bbox_to_anchor=(0.5, 1.1), ncol=1, fancybox=True, shadow=True)  
    ax.set_xlabel('Training set size')  
    ax.set_ylabel('RMSE')  
    plt.show()
```

# 1. Regression: Ridge regression

- A problem with linear regression is that estimated **coefficients** of the model can become **large**, making the model **sensitive** to inputs and possibly **unstable**. This is particularly true for problems with **few samples (n)** than input **predictors (p)** or variables (so-called **p >> n** problems).
- One approach to address the stability of regression models is to **change the loss function** to include **additional costs** for a model that has large coefficients. Linear regression models that use these modified loss functions during training are referred to collectively as **penalized linear regression**.
- One popular penalty is to penalize a model based on **the sum of the squared coefficient values**. This is called the **L2 penalty**.

$$loss = error(y, \hat{y}) + \gamma \sum_j \theta_j^2 \quad (\gamma \geq 0)$$

# 1. Regression: Ridge regression

Ridge regression addresses some of the problems of Ordinary Least Squares by imposing a penalty on the size of the coefficients.

```
from sklearn import datasets, linear_model
```

```
# Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization.
```

```
regressor = linear_model.Ridge(alpha=.5)  
regressor.fit(X_train, Y_train)  
predicted = regressor.predict(X_test)
```

```
# Bayesian Ridge Regression
```

```
regressor = linear_model.BayesianRidge()
```

```
# Lasso
```

```
regressor = linear_model.Lasso(alpha=0.1)
```

# 1. Regression: **Kernel Ridge regression**

In order to explore nonlinear relations of the regression problem

```
from sklearn.kernel_ridge import KernelRidge
```

```
# kernel = [linear, polynomial, rbf]
```

```
regressor = KernelRidge(kernel='rbf', alpha=1.0)
```

```
regressor.fit(X_train, Y_train)
```

```
predicted = regressor.predict(X_test)
```

# 1. Regression: Polynomial Regression

How to use a linear model to fit nonlinear data ?

A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called *Polynomial Regression*.



# 1. Regression: Polynomial Regression

```
# generate some nonlinear data
import numpy as np
m = 1000
X = 6 * np.random.rand(m, 1) - 3
Y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
print("Examples = ", X.shape, " Labels = ", Y.shape)

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig, ax = plt.subplots(figsize=(12,8))
ax.scatter(X[:,0], Y, edgecolors=(0, 0, 1))
ax.set_xlabel('F')
ax.set_ylabel('Y')
plt.show()

from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2,
include_bias=False)
X_poly = poly_features.fit_transform(X)
print("Examples = ", X_poly.shape, " Labels = ", Y.shape)

from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X_poly,
Y, test_size= 0.20, random_state=100)
```

```
from sklearn import linear_model
regressor = linear_model.LinearRegression()
regressor.fit(X_train, Y_train)
predicted = regressor.predict(X_poly)

fig, ax = plt.subplots(figsize=(12,8))
ax.scatter(X, Y, edgecolors=(0, 0, 1))
ax.scatter(X, predicted, edgecolors=(1, 0, 0))
ax.set_xlabel('F')
ax.set_ylabel('Y')
plt.show()

B = regressor.intercept_
A = regressor.coef_
print(A)
print(B)
print("The model estimates : Y = ", B[0], " + ", A[0,0], " X + ", A[0,1], " X^2")

from sklearn import metrics
predicted = regressor.predict(X_test)
print('Mean Absolute Error   : ', metrics.mean_absolute_error(Y_test, predicted))
print('Mean Squared Error    : ', metrics.mean_squared_error(Y_test, predicted))
print('Root Mean Squared Error: ', np.sqrt(metrics.mean_squared_error(Y_test,
predicted)))
```

# 1. Regression: **Support Vector Regression** **SVR**

The Support Vector Regression (SVR) uses the same principles as the SVM for classification, with only a few minor differences.

```
from sklearn.svm import SVR
```

```
svr_rbf = SVR(kernel='rbf', C=100, gamma=0.1)
```

```
svr_lin = SVR(kernel='linear', C=100, gamma=0.1)
```

```
svr_poly = SVR(kernel='poly', C=100, gamma=0.1, degree=3)
```

# 1. Regression: Support Vector Regression

## SVR

```
import numpy as np
from sklearn.svm import SVR
import matplotlib.pyplot as plt

# Generate sample data
X = np.sort(5 * np.random.rand(40, 1), axis=0)
Y = np.sin(X).ravel()
# Add noise to targets
Y[:,5] += 3 * (0.5 - np.random.rand(8))

print("Examples = ", X.shape, ", Y = ", Y.shape)
# Fit regression model
svr_rbf = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=.1)
svr_lin = SVR(kernel='linear', C=100, gamma='auto')
svr_poly = SVR(kernel='poly', C=100, gamma='auto', degree=3,
epsilon=.1, coef0=1)

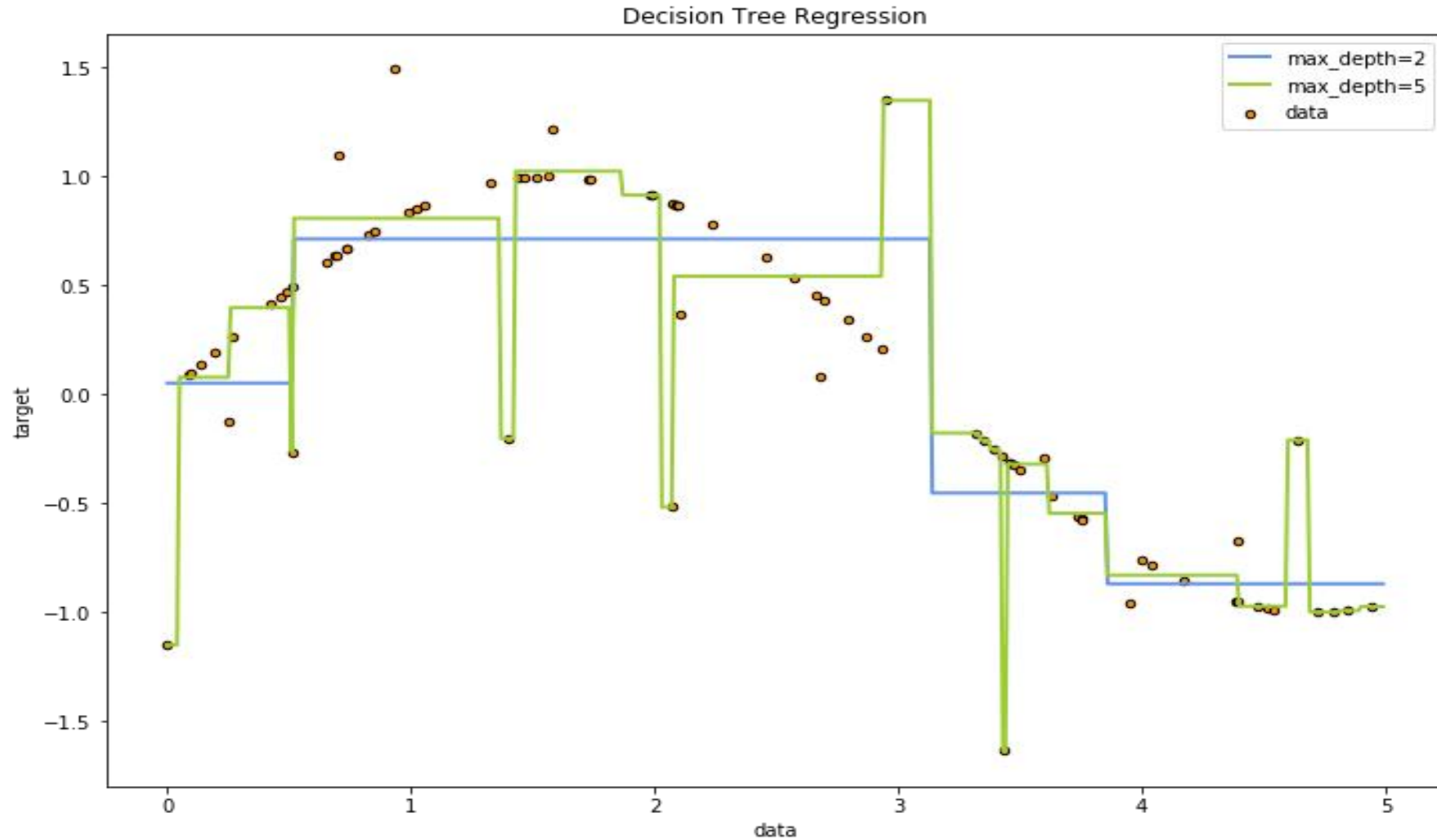
# Look at the results
lw = 2
svrs = [svr_rbf, svr_lin, svr_poly]
kernel_label = ['RBF', 'Linear', 'Polynomial']
model_color = ['m', 'c', 'g']
```

```
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(15, 10), sharey=True)
for ix, svr in enumerate(svrs):
    axes[ix].plot(X, svr.fit(X, Y).predict(X), color=model_color[ix], lw=lw,
label='{ } model'.format(kernel_label[ix]))

    axes[ix].scatter(X[svr.support_], Y[svr.support_], facecolor="none",
edgecolor=model_color[ix], s=50,
label='{ } support vectors'.format(kernel_label[ix]))
    axes[ix].scatter(X[np.setdiff1d(np.arange(len(X)), svr.support_)],
Y[np.setdiff1d(np.arange(len(X)), svr.support_)],
facecolor="none", edgecolor="k", s=50,
label='other training data')
    axes[ix].legend(loc='upper center', bbox_to_anchor=(0.5, 1.1),
ncol=1, fancybox=True, shadow=True)

fig.text(0.5, 0.04, 'data', ha='center', va='center')
fig.text(0.06, 0.5, 'target', ha='center', va='center', rotation='vertical')
fig.suptitle("Support Vector Regression", fontsize=14)
plt.show()
```

# 1. Regression: Decision Trees Regression



# 1. Regression: **Decision Trees Regression**

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.tree import DecisionTreeRegressor

rng = np.random.RandomState(1)
X = np.sort(5 * rng.rand(80, 1), axis=0)
Y = np.sin(X).ravel()
Y[::5] += 3 * (0.5 - rng.rand(16))
print("Examples = ", X.shape, " Labels = ", Y.shape)
X_test = np.arange(0.0, 5.0, 0.01)[:, np.newaxis]

regressor1 = DecisionTreeRegressor(max_depth=2)
regressor2 = DecisionTreeRegressor(max_depth=5)
```

```
regressor1.fit(X, Y)
regressor2.fit(X, Y)
predicted1 = regressor1.predict(X_test)
predicted2 = regressor2.predict(X_test)

plt.figure(figsize=(12,8))
plt.scatter(X, Y, s=20, edgecolor="black", c="darkorange",
            label="data")
plt.plot(X_test, predicted1,
         color="cornflowerblue", label="max_depth=2", linewidth=2)
plt.plot(X_test, predicted2, color="yellowgreen",
         label="max_depth=5", linewidth=2)
plt.xlabel("data")
plt.ylabel("target")
plt.title("Decision Tree Regression")
plt.legend()
plt.show()
```

# 1. Regression: **Random Forest Regressor**

```
from sklearn.ensemble import  
RandomForestRegressor
```

```
regressor = RandomForestRegressor(max_depth=5)  
regressor.fit(X, Y)  
predicted = regressor1.predict(X)
```

# 1. Regression: Multivariate Regression

- Multioutput regression is regression problems that involve predicting two or more numerical values given an input example.
- Not all regression algorithms support multioutput regression.

```
from sklearn.multioutput import MultiOutputRegressor
from sklearn.svm import LinearSVR
```

```
# define base model
model = LinearSVR()
# define the multioutput wrapper model
model_MOR = MultiOutputRegressor(model)
# fit model
model_MOR.fit(X_train, y_train)
```

```
predicted = model_MOR.predict(X_test)
```

```
from sklearn.multioutput import RegressorChain
from sklearn.svm import LinearSVR
```

```
# define base model
model = LinearSVR()
# define the chained multioutput wrapper model
model_RC = RegressorChain(model, order=[0, 1])
# fit model
model_RC.fit(X_train, y_train)
```

# Classification

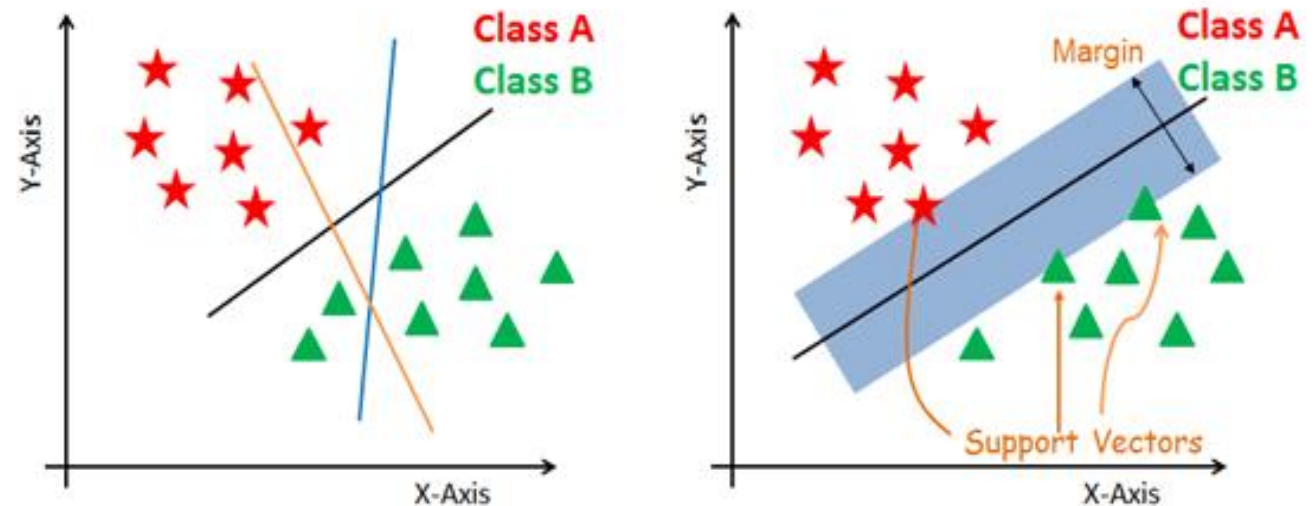


## 2. Classification: Support Vector Machines

The objective is to **select a hyperplane** with the **maximum possible margin** between support vectors in the given dataset. **SVM** searches for the maximum marginal hyperplane in the following steps:

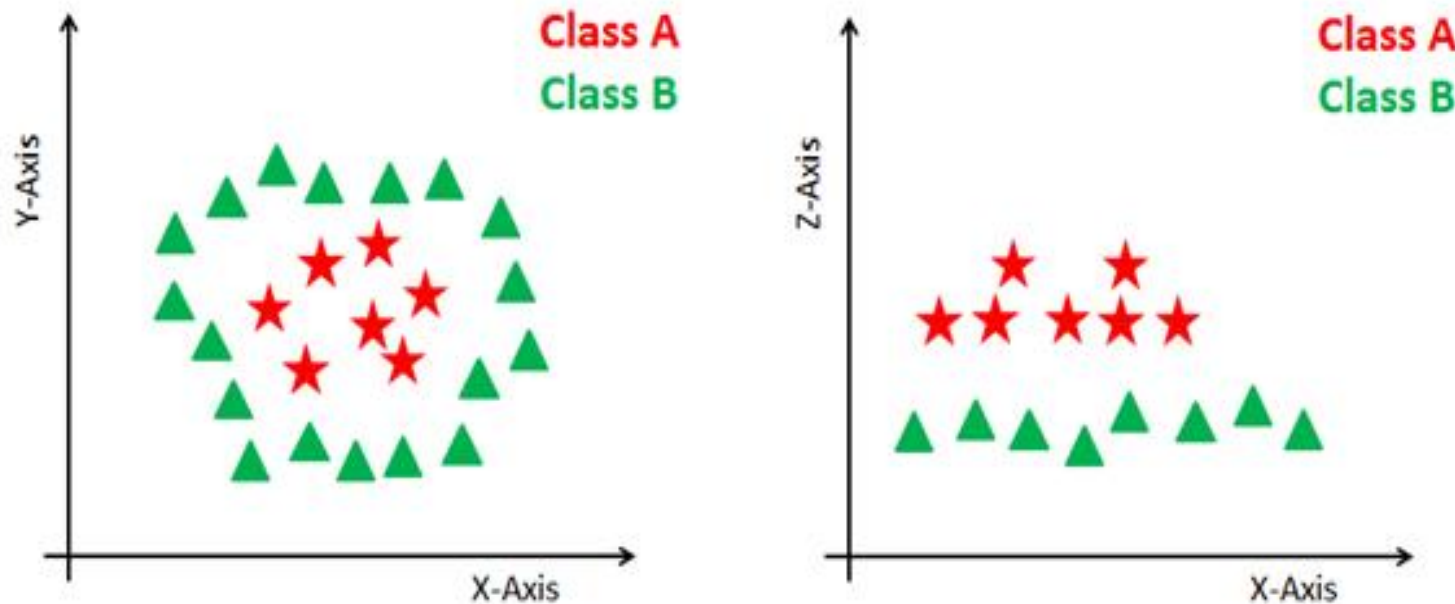
1) **Generate hyperplanes** that separate the classes in the best way. Left-hand side figure showing three hyperplanes: black, blue and orange. Here, the blue and orange have higher classification errors, but **the black** is separating the two classes correctly.

2) **Select the right hyperplane** with the maximum separation from either nearest data points as shown in the right-hand side figure.



## 2. Classification: Support Vector Machines - Kernel

- Some problems **can not be solved using linear hyperplane**, as shown in the figure below (left-hand side).
- In such situation, **SVM uses a kernel** to transform the input space to a higher dimensional space as shown on the right (  $z=x^2 + y^2$  ).



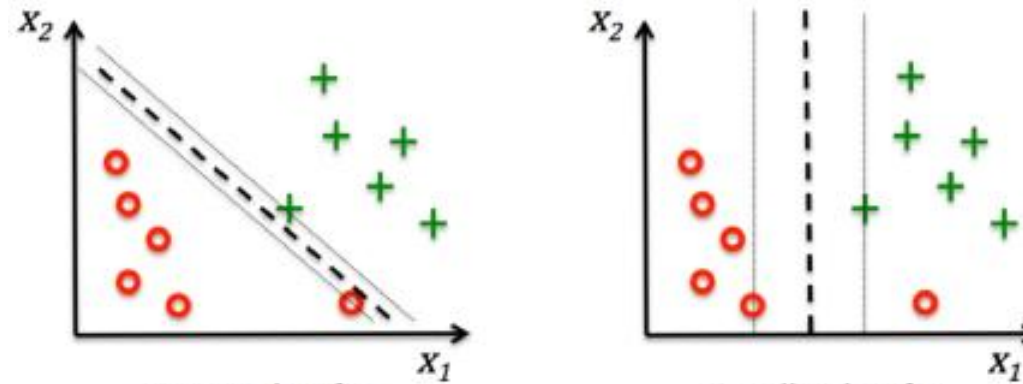
## 2. Classification: Support Vector Machines - Kernel + Tuning Hyperparameters

**Regularization:** the **C parameter** controls how much you want to **punish your model** for each misclassified point for a given curve:

Large values of C (Large effect of noisy points. A plane with very few misclassifications will be given precedence). Small Values of C (Low effect of noisy points. Planes that separate the points well will be found, even if there are some misclassifications).

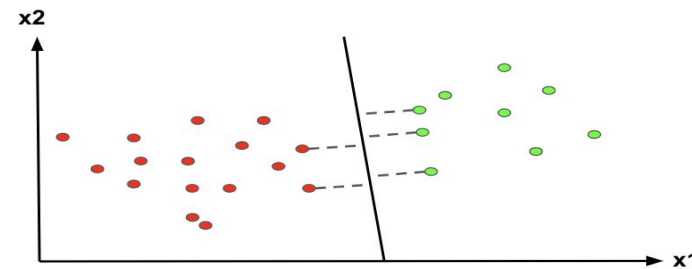
**Gamma (Kernel coefficient for rbf, poly and sigmoid):** the **high value** of gamma considers only **nearby points** in calculating the separation line, while a **low value** of gamma considers **all the data points** in the calculation of the separation line.

## 2. Classification: Support Vector Machines - Kernel + Tuning Hyperparameters



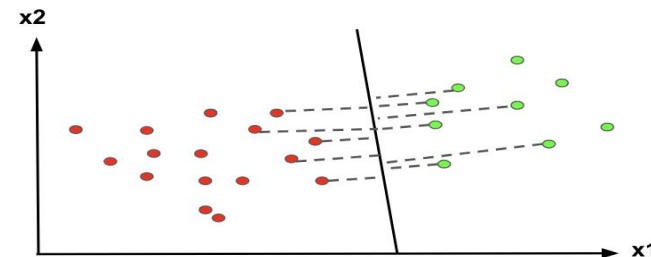
Large value for  
parameter  $C$

Small value for  
parameter  $C$



**High Gamma**

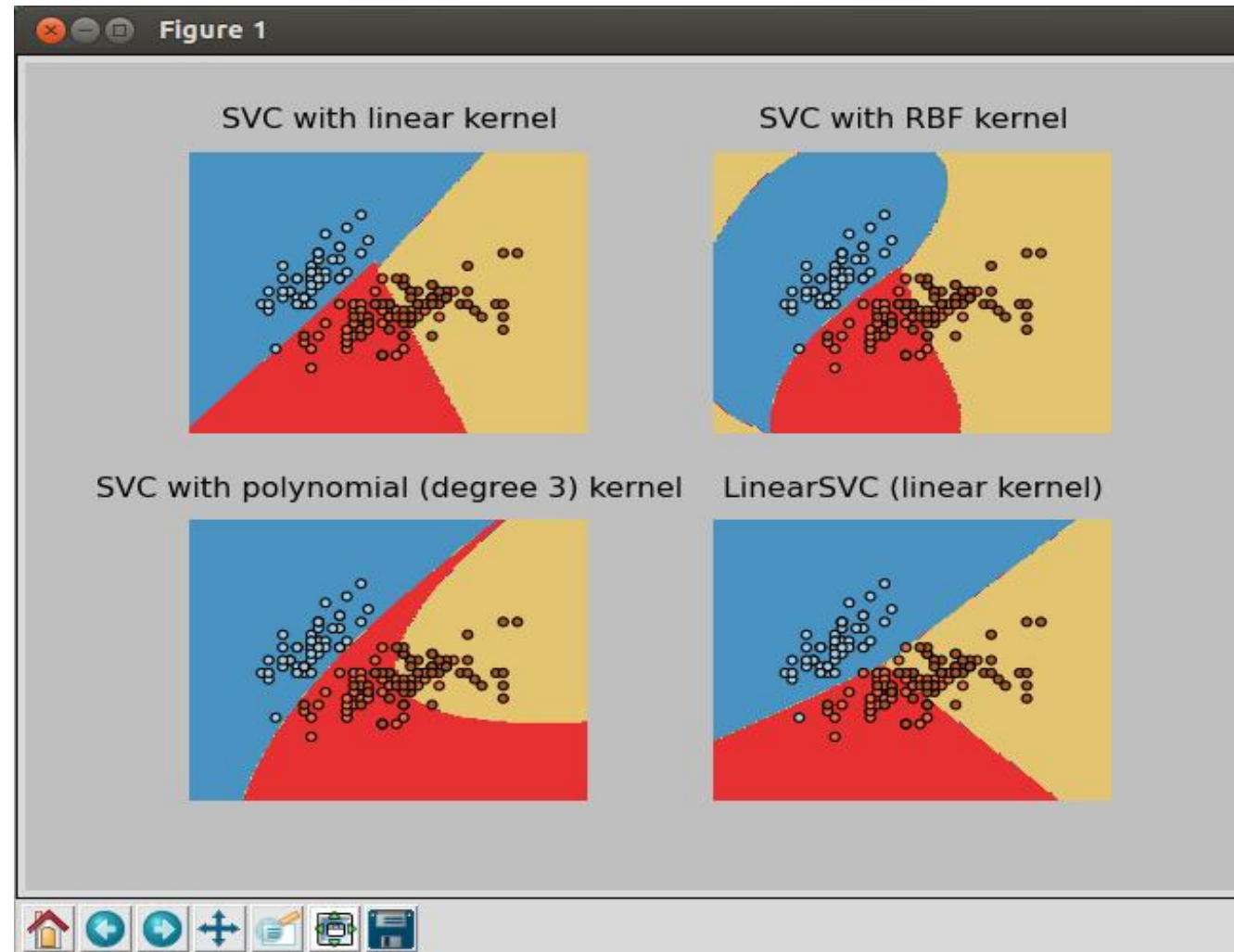
- only near points are considered.



**Low Gamma**

- far away points are also considered

## 2. Classification: Support Vector Machines - Kernel + Tuning Hyperparameters



## 2. Classification: Support Vector Machines -Kernel + Tuning Hyperparameters

```
from sklearn.svm import SVC
# linear kernel
svc_cls = SVC(kernel='linear')
# Polynomial Kernel
svc_cls = SVC(kernel='poly', degree=5, C=10, gamma=0.1)
# Gaussian Kernel
svc_cls = SVC(kernel='rbf', C=10, gamma=0.1)
# Sigmoid Kernel
svc_cls = SVC(kernel='sigmoid', C=10, gamma=0.1)

svc_cls.fit(X_train, y_train)
```

## 2. Classification: Support Vector Machines -Kernel + Tuning Hyperparameters

```
import matplotlib.pyplot as plt
from sklearn import datasets, svm, metrics
from sklearn.model_selection import train_test_split
```

*# Load the dataset*

```
digits = datasets.load_digits()
print("digits shape : ", digits.images.shape)
```

```
fig = plt.figure()
for i in range(9):
    plt.subplot(3,3,i+1)
    plt.tight_layout()
    plt.imshow(digits.images[i], cmap="gray",
interpolation="none")
    plt.title("Digit: {}".format(digits.target[i]))
    plt.xticks([])
    plt.yticks([])
fig
```

*# To apply a classifier on this data, we need to flatten the image,  
# to turn the data in a (samples, feature) matrix:*

```
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))
print("data shape : ", data.shape)
```

*# Split data into train and test subsets*

```
X_train, X_test, y_train, y_test = train_test_split(
    data, digits.target, test_size=0.2, shuffle=False)
print("X_train : ", X_train.shape, " y_train : ", y_train.shape)
print("X_test : ", X_test.shape, " y_test : ", y_test.shape)
```

```
print("-----")
```

*# Create a classifier: a support vector classifier*

```
classifier = svm.SVC(kernel="linear")
# classifier = svm.SVC(kernel="poly", degree=5, C=10, gamma=0.01)
# classifier = svm.SVC(kernel="sigmoid", C=10, gamma=0.001)
# classifier = svm.SVC(kernel="rbf", C=10, gamma=0.01)
```

```
classifier.fit(X_train, y_train)
```

*# Now predict the value of the digit on the second half:*

```
predicted = classifier.predict(X_test)
```

```
print("Classification report : \n", classifier, "\n",
metrics.classification_report(y_test, predicted))
disp = metrics.plot_confusion_matrix(classifier, X_test, y_test)
disp.figure_.suptitle("Confusion Matrix")
```

```
print("Confusion matrix: \n", disp.confusion_matrix)
```



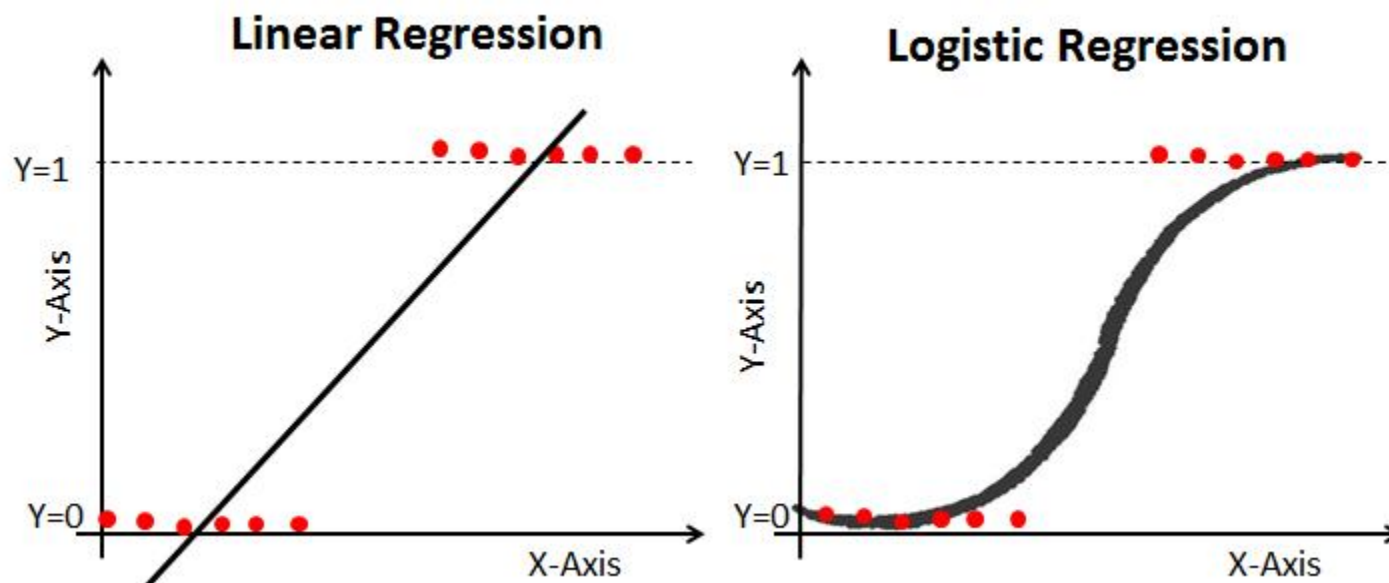
## 2. Classification: Logistic Regression Classifier

**Logistic regression classifier:** is a fundamental classification technique. It belongs to the group of linear classifiers and is somewhat similar to **polynomial and linear regression**. Logistic regression is **fast and relatively uncomplicated**, and it's **convenient** for you to interpret the results. Although it's essentially a method for **binary classification**, it can also be applied to **multiclass problems**.

Your goal is to find the logistic regression function  $p(x)$  such that the predicted responses  $p(x_i)$  are as close as possible to the actual response  $y_i$  for each observation  $i = 1, \dots, n$ .

**Sigmoid Function :**

$$f(x) = \frac{1}{1 + e^{-(x)}}$$





## 2. Classification: Logistic Regression Classifier

```
from sklearn import datasets, metrics
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

dat = datasets.load_breast_cancer()
print("Examples = ", dat.data.shape, " Labels = ", dat.target.shape)
X_train, X_test, Y_train, y_test = train_test_split(dat.data, dat.target, test_size= 0.20, random_state=100)
# default = 'auto'
logreg = LogisticRegression()
logreg.fit(X_train, Y_train)
predicted = logreg.predict(X_test)

print("Classification report for classifier : \n", metrics.classification_report(y_test, predicted))

disp = metrics.plot_confusion_matrix(logreg, X_test, y_test)
disp.figure_.suptitle("Confusion Matrix")
print("Confusion matrix: \n", disp.confusion_matrix)
plt.show()
```

## 2. Classification: Logistic Regression Classifier

### Advantages:

Because of its efficient and straightforward nature, **doesn't require high computation power**, **easy** to implement, **easily interpretable**, used widely by data analyst and scientist. Also, it doesn't require scaling of features. Logistic regression provides a probability score for observations.

### Disadvantages:

Logistic regression **is not able to handle a large number** of categorical features/variables. It is vulnerable to **overfitting**. Also, **can't solve the non-linear** problem with the logistic regression that is why it requires a transformation of non-linear features. Logistic regression will not perform well with **independent variables** that **are not correlated to the target variable** and are very similar or correlated to each other.

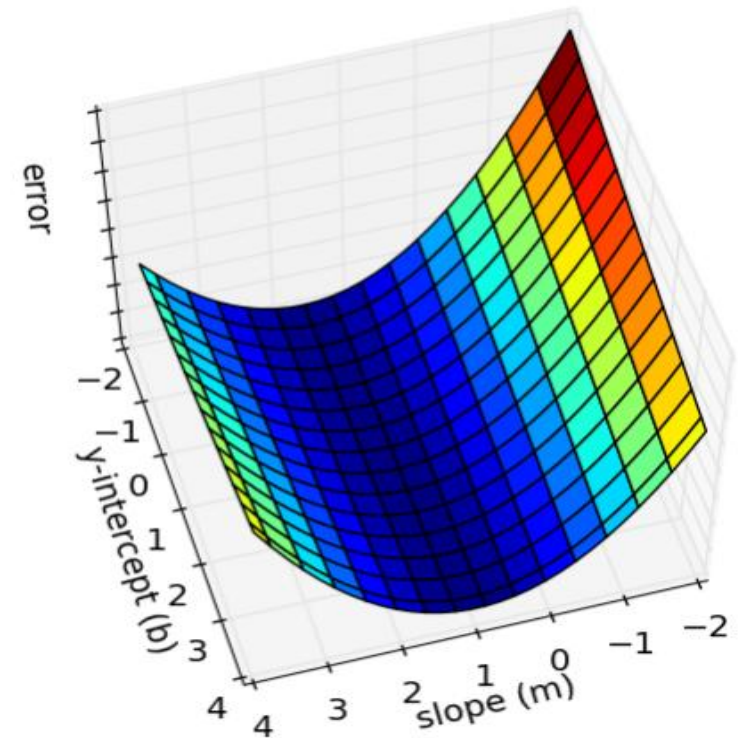
## 2. Classification: Stochastic Gradient Descent

**Gradient Descent** : at a theoretical level, gradient descent is an algorithm that minimizes functions. Given a function defined by a set of parameters, gradient descent starts with an initial set of parameter values and iteratively moves toward a set of parameter values that minimize the function. This iterative minimization is achieved using calculus, taking steps in the negative direction of the function gradient.

**Example:**  $y = mx + b$  line equation where  $m$  is the line's slope and  $b$  is the line's y-intercept. To find the best line for our data, we need to find the best set of slope  $m$  and y-intercept  $b$  values.

**$f(m, b)$ :** error function (also called a cost function) that measures how “good” a given line is.

*Each point in this two-dimensional space represents a line. The height of the function at each point is the error value for that line.*



## 2. Classification: Stochastic Gradient Descent

**Stochastic gradient descent (SGD):** also known as **incremental gradient descent**, is an iterative method for optimizing a differentiable objective function, a stochastic approximation of gradient descent optimization.

**SGD** has been successfully applied to **large-scale** and **sparse machine learning problems** often encountered in **text classification** and **natural language processing**. Given that the data is **sparse**, the classifiers in this module **easily scale** to problems with more than  **$10^5$  training examples** and more than  **$10^5$  features**.

## 2. Classification: Stochastic Gradient Descent

```
from sklearn.linear_model import SGDClassifier
```

```
X = [[0., 0.], [1., 1.]]
```

```
y = [0, 1]
```

```
# loss : hinge, log, modified_huber, squared_hinge, perceptron
```

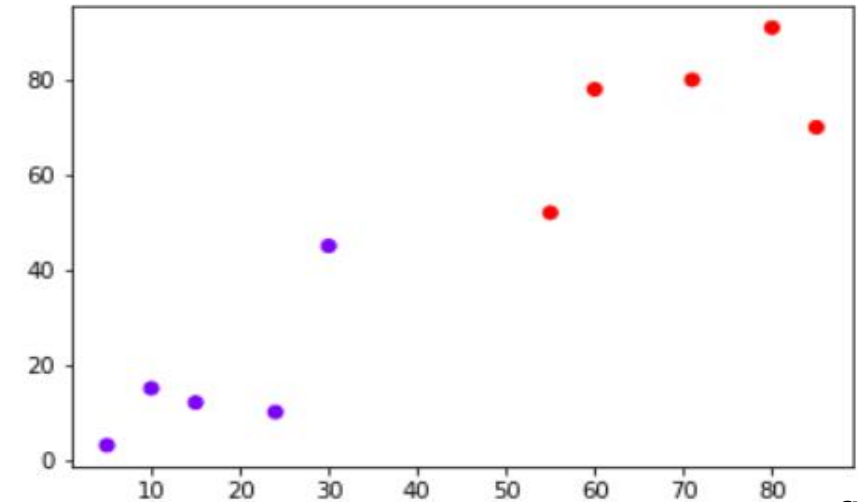
```
clf = SGDClassifier(loss="hinge", penalty="l2", max_iter=5)
```

```
clf.fit(X, y)
```

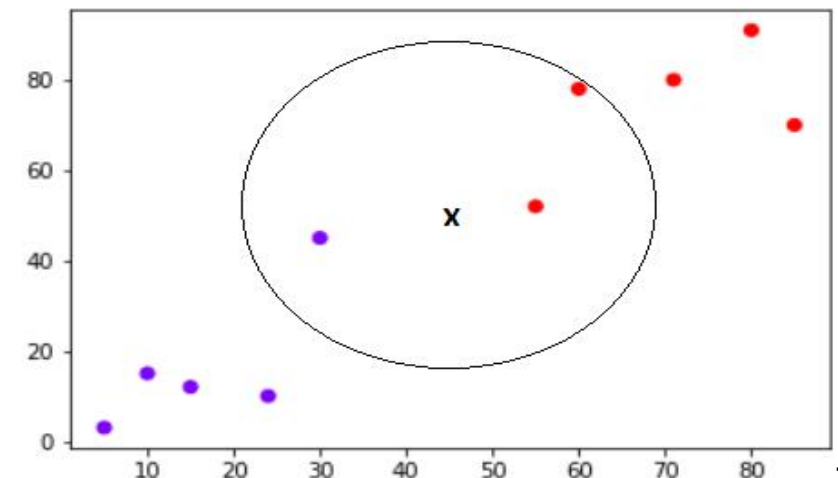
```
predicted = clf.predict(X_test)
```

## 2. Classification: K-Nearest Neighbors KNN

- 1) Let's see this algorithm in action with the help of a simple example. Suppose you have a dataset with two variables, which when plotted, looks like the one in the following figure (fig<sub>1</sub>).
- 2) Your task is to classify a new data point with 'X' into "Blue" class or "Red" class. The coordinate values of the data point are  $x=45$  and  $y=50$ . Suppose the value of  $K$  is 3. The KNN algorithm starts by calculating *the distance of point X from all the points*. It then finds *the 3 nearest points* with least distance to point X. This is shown in the figure below. The three nearest points have been **encircled**.
- 3) The final step of the KNN algorithm is to assign new point to the class to which *majority of the three nearest points belong*. From the fig<sub>2</sub> we can see that the two of the three nearest points belong to the class "Red" while one belongs to the class "Blue". Therefore the new data point will be classified as "Red".



fig<sub>1</sub>



fig<sub>2</sub>

## 2. Classification: **K-Nearest Neighbors KNN**

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.metrics import classification_report,
confusion_matrix
import numpy as np
import matplotlib.pyplot as plt

# Loading data
irisData = load_iris()
# Create feature and target arrays
X = irisData.data
y = irisData.target
# Split into training and test set
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size = 0.2, random_state=42)

knn = KNeighborsClassifier(n_neighbors = 7)
knn.fit(X_train, y_train)
predicted = knn.predict(X_test)
print(confusion_matrix(y_test, predicted))
print(classification_report(y_test, predicted))
```

```
neighbors = np.arange(1, 25)
train_accuracy = np.empty(len(neighbors))
test_accuracy = np.empty(len(neighbors))
# Loop over K values
for i, k in enumerate(neighbors):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)

# Compute training and test data accuracy
train_accuracy[i] = knn.score(X_train, y_train)
test_accuracy[i] = knn.score(X_test, y_test)

# Generate plot
plt.plot(neighbors, test_accuracy, label = 'Testing dataset
Accuracy')
plt.plot(neighbors, train_accuracy, label = 'Training
dataset Accuracy')
plt.legend()
plt.xlabel('n_neighbors')
plt.ylabel('Accuracy')
plt.show()
```

## 2. Classification: K-Nearest Neighbors KNN

### Advantages:

- 1) It is extremely **easy** to implement
- 2) Requires **no training prior** to making **real time** predictions. This makes the KNN algorithm **much faster** than other algorithms that require training e.g SVM, linear regression, etc.
- 3) New data can be added **seamlessly**.
- 4) There are only two parameters required to implement KNN i.e. **the value of K** and **the distance function** (e.g. Euclidean or Manhattan etc.)

### Disadvantages:

- 1) The KNN algorithm doesn't work well with **high dimensional data** because with large number of dimensions, it becomes difficult for the algorithm to calculate distance in each dimension.
- 2) The KNN algorithm **has a high prediction cost** for large datasets. This is because in large datasets the cost of calculating distance between new point and each existing point becomes higher.
- 3) Finally, the KNN algorithm **doesn't work well with categorical features** since it is difficult to find the distance between dimensions with **categorical features**.



## 2. Classification: Naive Bayes Classification

**Naive Bayes:** is a probabilistic machine learning algorithm that can be used in a wide variety of classification tasks. Typical applications include **filtering spam**, **classifying documents**, **sentiment prediction**, etc. It is based on the **Bayes theory**.

**Bayes Rule**

$$P(Y | X) = \frac{P(X | Y) * P(Y)}{P(X)}$$

## 2. Classification: Naive Bayes Classification

The Bayes Rule provides the formula for *the probability of Y given X*. But, in real-world problems, you typically have multiple **X variables**.

When the features are **independent**, we can extend the Bayes Rule to what is called **Naive Bayes**.

When there are multiple X variables, we simplify it by assuming the X's are independent, so the **Bayes** rule

$$P(Y=k | X) = \frac{P(X | Y=k) * P(Y=k)}{P(X)}$$

where, k is a class of Y

becomes, Naive **Bayes**

$$P(Y=k | X_1..X_n) = \frac{P(X_1 | Y=k) * P(X_2 | Y=k) ... * P(X_n | Y=k) * P(Y=k)}{P(X_1) * P(X_2) ... * P(X_n)}$$

## 2. Classification: Naive Bayes Classification

Since  $P(x_1, x_2, \dots, x_n)$  is constant given the input, we can use the following classification rule:

$$P(y|x_1, \dots, x_n) \propto P(y) \prod_{i=1}^n P(x_i|y)$$



$$y = \operatorname{argmax}_y P(y) \prod_{i=1}^n P(x_i|y)$$

### Laplace Correction :

when you have a model with many features if the entire probability will become zero because one of the **feature's value was zero**. To avoid this, we increase the count of the variable with zero to a **small value (usually 1)** in the numerator, so that the overall probability doesn't become zero.

## 2. Classification: Naive Bayes Classification

	weather	Temperature	Play
1	Sunny	Hot	No
2	Sunny	Hot	No
3	Overcast	Hot	Yes
4	Rainy	Mild	Yes
5	Rainy	Cool	Yes
6	Rainy	Cool	No
7	Overcast	Cool	Yes
8	Sunny	Mild	No
9	Sunny	Cool	Yes
10	Rainy	Mild	Yes
11	Sunny	Mild	Yes
12	Overcast	Mild	Yes
13	Overcast	Hot	No
14	Rainy	Mild	No

weather	No	Yes	P(No)	P(Yes)
Sunny	3	2	3/6	2/8
Overcast	1	3	1/6	3/8
Rainy	2	3	2/6	3/8
Total	6	8	6/14	8/14

Temperature	No	Yes	P(No)	P(Yes)
Hot	3	1	3/6	1/8
Mild	2	4	2/6	4/8
cool	1	3	1/6	3/8
Total	6	8	6/14	8/14

today = (Overcast, Mild)

$$P(\text{Yes} / \text{today}) = P(\text{Overcast} / \text{Yes}) \times P(\text{Mild} / \text{Yes}) \times P(\text{Yes})$$

$$P(\text{Yes} / \text{today}) = \frac{3}{8} \times \frac{4}{8} \times \frac{8}{14} = 0,107$$

$$P(\text{No} / \text{today}) = P(\text{Overcast} / \text{No}) \times P(\text{Mild} / \text{No}) \times P(\text{No})$$

$$P(\text{No} / \text{today}) = \frac{1}{6} \times \frac{2}{6} \times \frac{6}{14} = 0,023$$

$$P(\text{Yes} / \text{today}) = \frac{0,107}{0,107 + 0,023} = 0,82 \quad P(\text{No} / \text{today}) = \frac{0,023}{0,107 + 0,023} = 0,18$$

## 2. Classification: Gaussian Naive Bayes Classification

```
# Assigning features and label variables
weather=['Sunny','Sunny','Overcast','Rainy','Rainy','Rainy','Overcast','Sunny','Sunny','Rainy','Sunny','Overcast','Overcast','Rainy']
temp=['Hot','Hot','Hot','Mild','Cool','Cool','Cool','Mild','Cool','Mild','Mild','Mild','Hot','Mild']
play=['No','No','Yes','Yes','Yes','No','Yes','No','Yes','Yes','Yes','Yes','No','No']
```

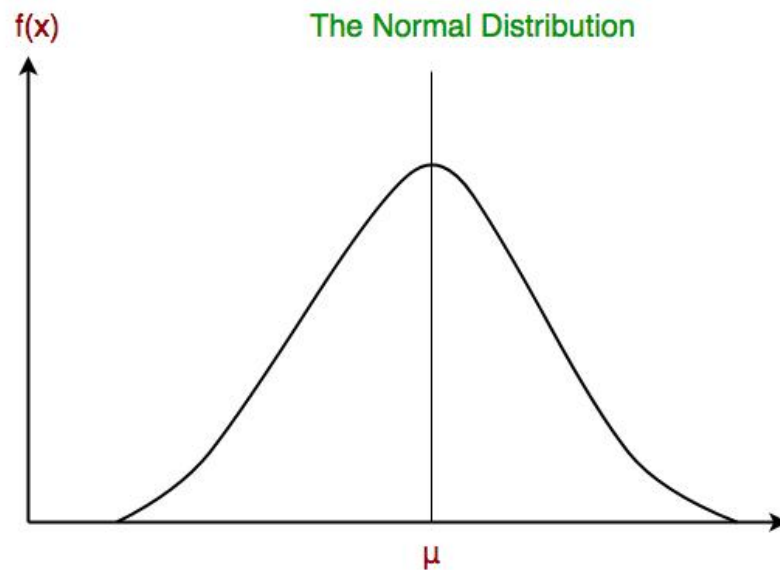
```
# Import LabelEncoder
from sklearn import preprocessing
# creating labelEncoder
le = preprocessing.LabelEncoder()
# Converting string labels into numbers.
weather_encoded=le.fit_transform(weather)
print("weather:",weather_encoded)
# Converting string labels into numbers
temp_encoded=le.fit_transform(temp)
label=le.fit_transform(play)
print("Temp: ",temp_encoded)
print("Play: ",label)
```

```
# Combinig weather and temp into single listof tuples
features = zip(weather_encoded,temp_encoded)
import numpy as np
features = np.asarray(list(features))
print("features : ",features)
```

```
#Import Gaussian Naive Bayes model
from sklearn.naive_bayes import GaussianNB
#Create a Gaussian Classifier
model = GaussianNB()
# Train the model using the training sets
model.fit(features,label)
# Predict Output
predicted= model.predict([[0,2]]) # 0:Overcast, 2:Mild
print ("Predicted Value (No = 0, Yes = 1):", predicted)
```

## 2. Classification: Gaussian Naive Bayes Classification

In Gaussian Naive Bayes, **continuous values** associated with each feature are **assumed** to be distributed according to a **Gaussian distribution**. A Gaussian distribution is also called Normal distribution.



The likelihood of the features is assumed to be Gaussian, hence, conditional probability is given by:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

## 2. Classification: Gaussian Naive Bayes Classification

```
# load the iris dataset
from sklearn.datasets import load_iris
iris = load_iris()
# store the feature matrix (X) and response vector (y)
X = iris.data
y = iris.target
# splitting X and y into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=100)
# training the model on training set
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
gnb.fit(X_train, y_train)
# making predictions on the testing set
y_pred = gnb.predict(X_test)
# comparing actual response values (y_test) with predicted response values (y_pred)
from sklearn import metrics
print("Gaussian Naive Bayes model accuracy(in %):", metrics.accuracy_score(y_test, y_pred)*100)
print("Number of mislabeled points out of a total %d points : %d" % (X_test.shape[0], (y_test !=
y_pred).sum()))
```

## 2. Classification: Multinomial Naive Bayes Classification

The **multinomial Naive Bayes classifier** is suitable for classification with **discrete features** (e.g., word counts for text classification). The multinomial distribution normally requires **integer feature counts**. However, in practice, fractional counts such as tf-idf may also work.

### Advantages:

- 1) They are extremely **fast** for both training and prediction.
- 2) They provide a **straightforward probabilistic** prediction.
- 3) They are often **very easily** interpretable.
- 4) They have very **few (if any) tunable parameters**.



## 2. Classification: Multinomial Naive Bayes

### Classification

```
from sklearn.datasets import fetch_20newsgroups
data = fetch_20newsgroups()
print(data.target_names)
# For simplicity here, we will select just a few of these categories
categories = ['talk.religion.misc', 'soc.religion.christian',
              'sci.space', 'comp.graphics']
sub_data = fetch_20newsgroups(subset='train',
                              categories=categories)
X, Y = sub_data.data, sub_data.target
print("Examples = ", len(X), " Labels = ", len(Y))
# Here is a representative entry from the data:
print(X[5])
# In order to use this data for machine learning, we need to be
# able to convert the content of each string into a vector of numbers.
# For this we will use the TF-IDF vectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
vec = TfidfVectorizer()
vec.fit(X)
texts = vec.transform(X).toarray()
print(" texts shape = ", texts.shape)
```

```
# split the dataset into training data and test data
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(texts, Y,
                                                    test_size= 0.20, random_state=100, stratify=Y)
# training the model on training set
from sklearn.naive_bayes import MultinomialNB
clf = MultinomialNB()
clf.fit(X_train, Y_train)
# making predictions on the testing set
predicted = clf.predict(X_test)

from sklearn import metrics
print("Classification report : \n", clf, "\n",
      metrics.classification_report(Y_test, predicted))
disp = metrics.plot_confusion_matrix(clf, X_test, Y_test)
disp.figure_.suptitle("Confusion Matrix")
print("Confusion matrix: \n", disp.confusion_matrix)
```

## 2. Classification: Decision Trees

**Decision trees are supervised learning algorithms used for both, classification and regression tasks.**

The main idea of decision trees is to **find the best descriptive features** which contain the most information regarding the target feature and **then split the dataset along the values of these features** such that the target feature values for the resulting **sub\_datasets** are as pure as possible.

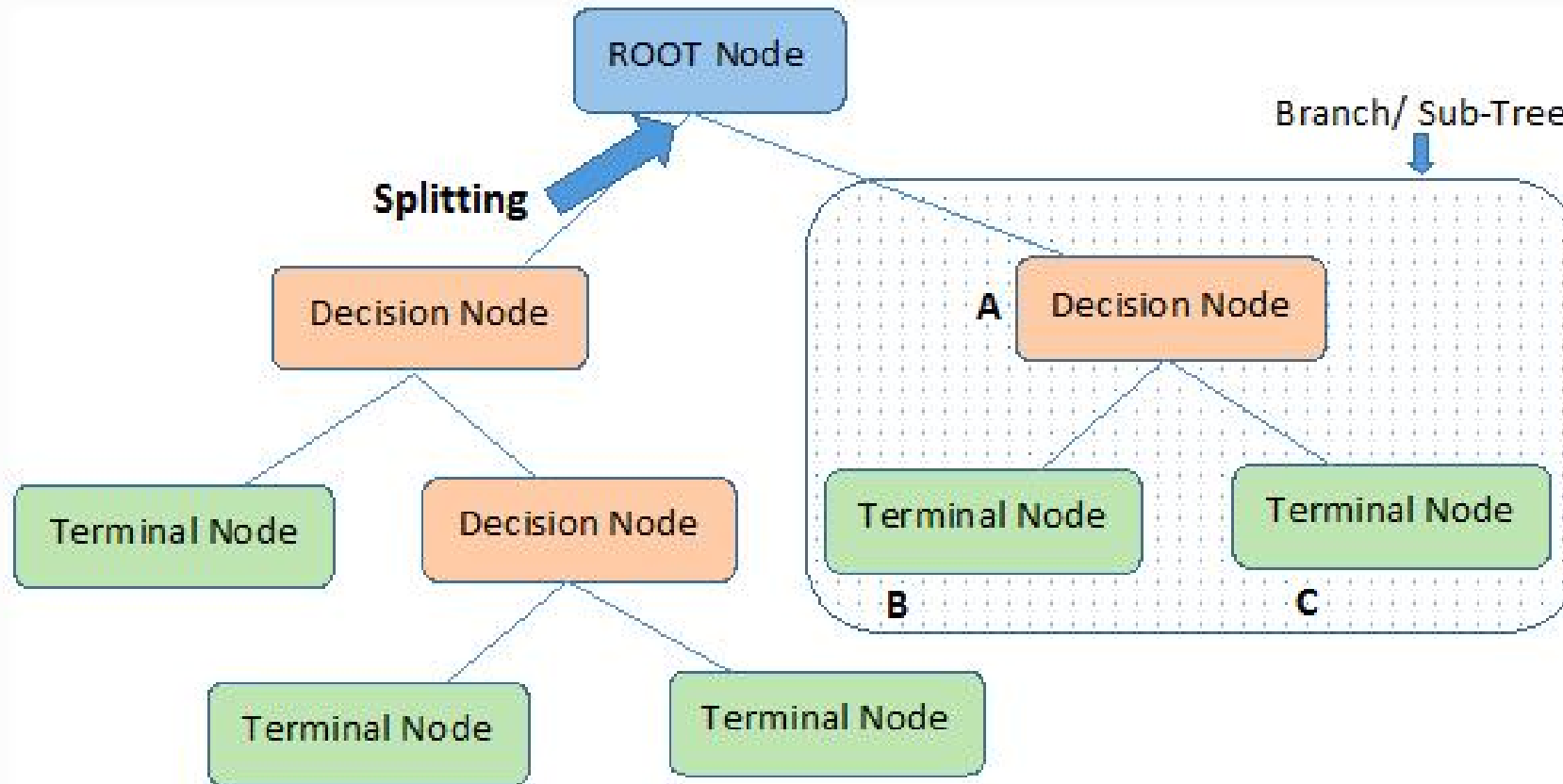
The descriptive feature which leaves the target feature most purely is said to be the most informative one. **This process of finding the most informative feature is done until we accomplish a stopping criterion** where we then finally end up in so-called **leaf nodes**. The leaf nodes contain the predictions we will make for new query instances presented to our trained model. This is possible since the model has kind of learned the underlying structure of the training data and hence can, given some assumptions, make predictions about the target feature value (class) of unseen query instances.

**Decision trees can handle both categorical and numerical data.**

## 2. Classification: Decision Trees

- 1) **Root Node:** This attribute is used for dividing the data into two or more sets. The feature attribute in this node is selected based on Attribute Selection Techniques.
- 2) **Branch or Sub-Tree:** A part of the entire decision tree is called branch or sub-tree.
- 3) **Splitting:** Dividing a node into two or more sub-nodes based on **if-else** conditions.
- 4) **Decision Node:** After splitting the sub-nodes into further sub-nodes, then it is called as the decision node.
- 5) **Leaf or Terminal Node:** This is the end of the decision tree where it cannot be split into further sub-nodes.

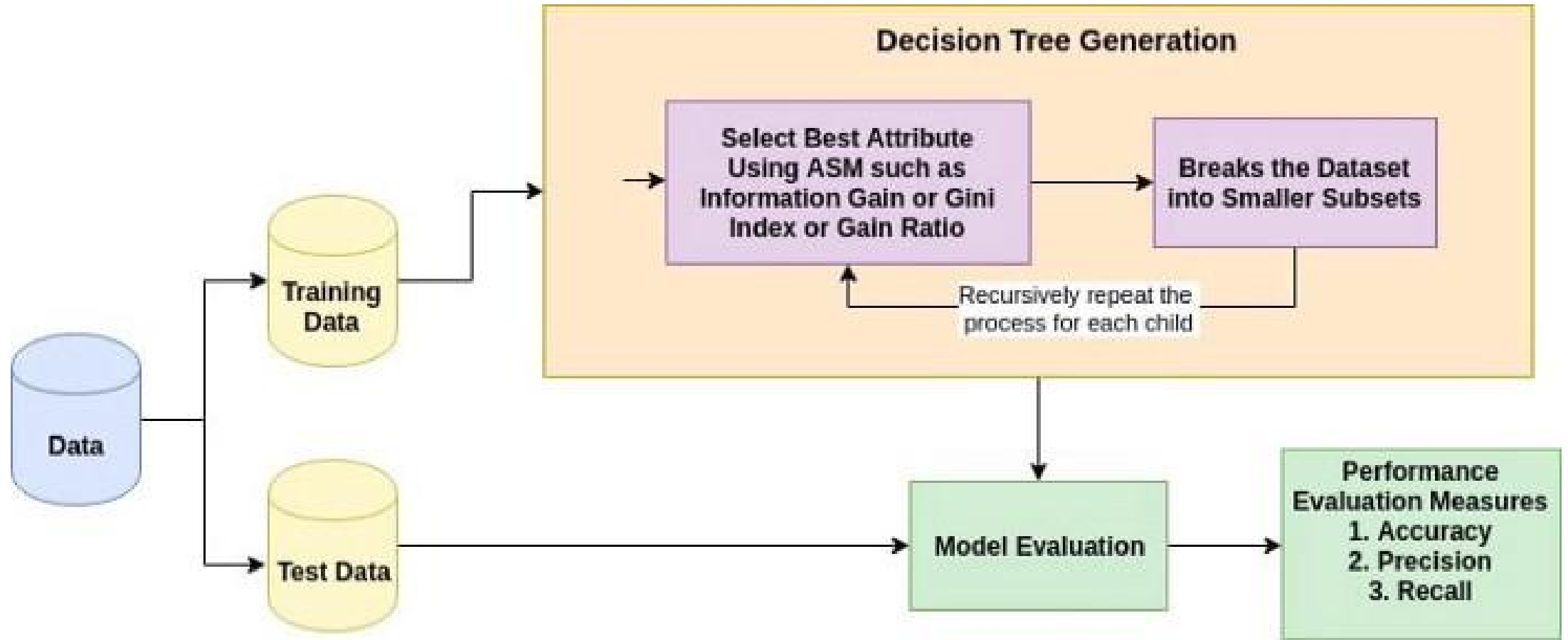
## 2. Classification: **Decision Trees**



**Note:-** A is parent node of B and C.

## 2. Classification: Decision Trees

How does the Decision Tree algorithm work?

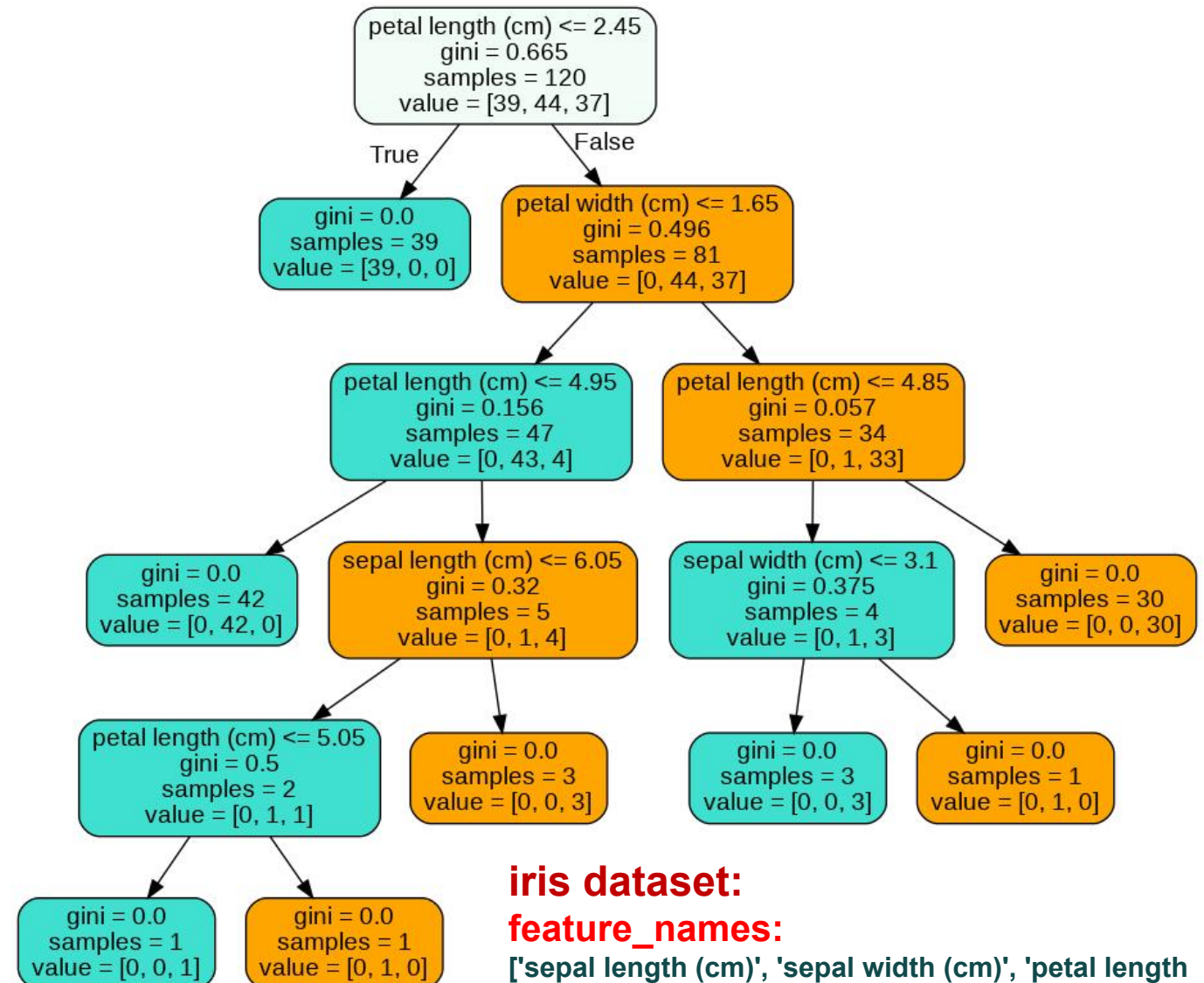


## 2. Classification: Decision Trees

- **The root node feature** is selected based on the results from the Attribute Selection Measure(ASM).
- **Attribute selection measure** (ASM) is a **heuristic** for selecting the splitting criterion that partition the data into the best possible manner.
- The ASM is repeated until there is a leaf node or a terminal node where it cannot be split into sub-nodes.
- The two main ASM techniques are: **Gini impurity** and **Entropy**

## 2. Classification: Decision Trees - Example

```
--- petal length (cm) <= 2.45
|--- class: 0
--- petal length (cm) > 2.45
|--- petal width (cm) <= 1.65
|   |--- petal length (cm) <= 4.95
|   |   |--- class: 1
|   |--- petal length (cm) > 4.95
|   |   |--- sepal length (cm) <= 6.05
|   |   |   |--- petal length (cm) <= 5.05
|   |   |   |   |--- class: 2
|   |   |   |   |--- petal length (cm) > 5.05
|   |   |   |   |   |--- class: 1
|   |   |--- sepal length (cm) > 6.05
|   |   |   |--- class: 2
|--- petal width (cm) > 1.65
|   |--- petal length (cm) <= 4.85
|   |   |--- sepal width (cm) <= 3.10
|   |   |   |--- class: 2
|   |   |--- sepal width (cm) > 3.10
|   |   |   |--- class: 1
|   |--- petal length (cm) > 4.85
|   |   |--- class: 2
```



**iris dataset:**

**feature\_names:**

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

**target\_names:** ['setosa' 'versicolor' 'virginica']



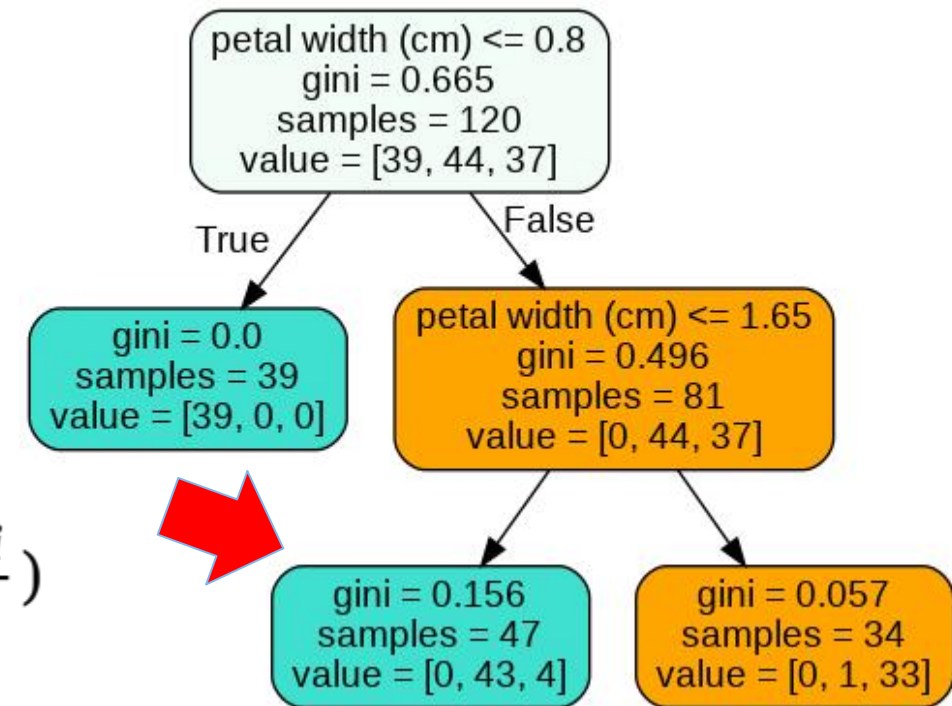
## 2. Classification: Decision Trees

**Gini impurity:**

$$G_i = 1 - \sum_{k=1}^n \frac{P_i^2}{k}$$
$$GI = 1 - ((\frac{0}{47})^2 + (\frac{43}{47})^2 + (\frac{4}{47})^2) = 0,156$$

**Entropy:**

$$H_i = - \sum_{k=1}^n \frac{P_i}{k} \times \log_2(\frac{P_i}{k})$$
$$H_i = - \frac{43}{47} \times \log_2(\frac{43}{47}) - \frac{4}{47} \times \log_2(\frac{4}{47})$$





## 2. Classification: Decision Trees

### Estimating Class Probabilities:

A Decision Tree can also estimate the probability that an instance belongs to a particular class  $k$ : first it traverses the tree to find the leaf node for this instance, and then it returns the ratio of training instances of class  $k$  in this node.

```
print("predict_proba : ",tree_clf.predict_proba([[1, 1.5, 3.2, 0.5]]))
```

```
predict_proba : [[0.          0.91489362 0.08510638]]
```

**Scikit-Learn** uses the **CART algorithm**, which produces only **binary trees**: nonleaf nodes always have two children (i.e., questions only have **yes/no** answers). However, other algorithms such as **ID3** can produce Decision Trees with nodes that **have more than two children**.

## 2. Classification: Decision Trees

### Advantages of CART

- Simple to understand, interpret, visualize.
- Decision trees implicitly perform variable screening or **feature selection**.
- Can handle both **numerical and categorical data**. Can also handle multi-output problems.
- Decision trees require relatively **little effort from users** for data preparation.
- **Nonlinear relationships** between parameters do not affect tree performance.

### Disadvantages of CART

- Decision-tree learners can create over-complex trees that do not generalize the data well. This is called **overfitting**.
- Decision trees **can be unstable because** small variations in the data might result in a completely different tree being generated. This is called variance, which needs to be lowered by methods like bagging and boosting.
- **Greedy algorithms** cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees, where the features and samples are randomly sampled with replacement.
- Decision tree learners **create biased trees if some classes dominate**. It is therefore recommended to **balance the dataset** prior to fitting with the decision tree.

## 2. Classification: Decision Trees

Optimizing the performance of the trees :

- 1) **max\_depth**: The maximum depth of the tree.
- 2) **criterion**: This parameter takes the criterion method as the value. The default value is **Gini**.
- 3) **splitter**: This parameter allows us to choose the split strategy. **Best** and **random** are available types of the split. The default value is best.

How to avoid Over-Fitting?

Use Random Forest trees.

## 2. Classification: Decision Trees - Example

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
# load the iris dataset
dat = load_iris()
# store the feature matrix (X) and response vector (y)
X = dat.data
y = dat.target
# splitting X and y into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=100)
# training the model on training set
# criterion='gini' splitter='best' max_depth=2
tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X_train, y_train)
# making predictions on the testing set
predicted = tree_clf.predict(X_test)
# comparing actual response values (y_test) with predicted response values (predicted)
from sklearn import metrics
print("Classification report : \n", tree_clf, "\n", metrics.classification_report(y_test, predicted))
disp = metrics.plot_confusion_matrix(tree_clf, X_test, y_test)
disp.figure_.suptitle("Confusion Matrix")
print("Confusion matrix: \n", disp.confusion_matrix)
# plot the tree with the plot_tree function:
from sklearn import tree
tree.plot_tree(tree_clf.fit(X_train, y_train))
```

## 2. Classification: Decision Trees - Example

# Visualize data

```
import collections
```

```
import pydotplus
```

```
dot_data = tree.export_graphviz(tree_clf, feature_names=dat.feature_names, out_file=None, filled=True, rounded=True)
```

```
graph = pydotplus.graph_from_dot_data(dot_data)
```

```
colors = ('turquoise', 'orange')
```

```
edges = collections.defaultdict(list)
```

```
for edge in graph.get_edge_list():
```

```
    edges[edge.get_source()].append(int(edge.get_destination()))
```

```
for edge in edges:
```

```
    edges[edge].sort()
```

```
    for i in range(2):
```

```
        dest = graph.get_node(str(edges[edge][i]))[0]
```

```
        dest.set_fillcolor(colors[i])
```

```
graph.write_png('tree1.png')
```

#pdf

```
import graphviz
```

```
dot_data = tree.export_graphviz(tree_clf, out_file=None)
```

```
graph = graphviz.Source(dot_data)
```

```
graph.render("iris")
```

```
dot_data = tree.export_graphviz(tree_clf, out_file=None, feature_names=dat.feature_names, class_names=dat.target_names, filled=True, rounded=True, special_characters=True)
```

```
graph = graphviz.Source(dot_data)
```

```
graph.view('tree2.pdf')
```

# The tree can also be exported in textual format

```
from sklearn.tree import export_text
```

```
r = export_text(tree_clf, feature_names=dat.feature_names)
```

```
print(r)
```

Hichem Felouat - hichemfel@gmail.com

## 2. Classification: **Multioutputclassifier**

Scikit-learn API provides a `MultiOutputClassifier` class that helps to classify multi-output data.

```
from sklearn.multioutput import MultiOutputClassifier
from sklearn import svm, metrics
```

```
# Create a classifier
```

```
classifier = svm.SVC(kernel="linear")
```

```
# define the MultiOutputClassifier wrapper model
```

```
model = MultiOutputClassifier(estimator=classifier)
```

```
# Train the classifier
```

```
model.fit(X_train, y_train)
```

```
predicted = model.predict(X_test)
```

```
[ 3. 12.  9.  8.  0.  5.  6.  2.  2.  5.] => [0 1 0]
[ 3.  2.  6.  0.  5. 10.  9.  0.  5.  3.] => [1 0 1]
[ 9.  9.  5.  5.  3.  8.  9.  1.  3.  5.] => [1 1 1]
[ 4. 11.  6.  4.  3.  7.  5.  3.  2.  8.] => [0 1 1]
[ 6.  8.  9.  6.  0.  4.  4.  4.  7.  2.] => [1 1 0]
[ 1.  3.  4.  7.  0.  6.  4.  3.  2.  7.] => [0 1 0]
[ 7.  6.  8.  5.  3.  5.  4.  1.  2.  6.] => [1 1 1]
[ 3.  5.  8.  6.  7.  7.  6.  0.  7.  5.] => [1 0 1]
[ 6.  7.  2.  5.  2. 10.  5.  1.  5.  8.] => [1 1 1]
[ 5.  6.  4. 10.  0.  8.  6.  5.  4.  6.] => [1 1 0]
```

# Ensemble Methods

# 3. Ensemble Methods

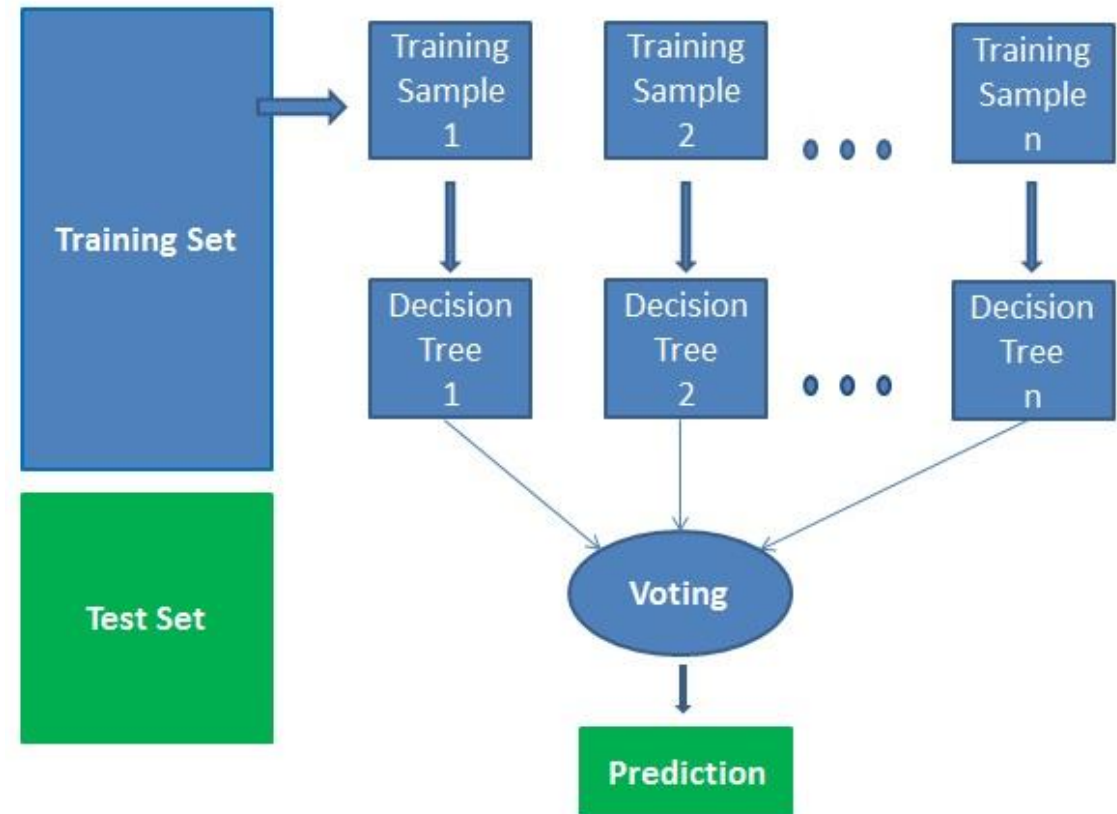
- The goal of ensemble methods is to **combine the predictions of several base estimators** built with a given learning algorithm in order to improve **generalizability / robustness** over a **single estimator**.
- Two families of ensemble methods are usually distinguished:
  - **In averaging methods**, the driving principle is to build several estimators independently and then to **average their predictions**. On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced.  
**Examples: Bagging methods, Forests of randomized trees, ...**
  - By contrast, **in boosting methods**, base estimators are built sequentially and one tries to reduce the bias of the combined estimator. The motivation is to **combine several weak models** to produce a powerful ensemble.  
**Examples: AdaBoost, Gradient Tree Boosting, ...**



### 3. Ensemble Methods: Random Forest Classifier

**The random forest classifier** is a supervised learning algorithm and composed of multiple decision trees. By averaging out the impact of several decision trees, random forests tend to improve prediction.

- 1) Select random samples from a given dataset.
- 2) Construct a decision tree for each sample and get a prediction result from each decision tree.
- 3) Perform a vote for each predicted result.
- 4) Select the prediction result with the most votes as the final prediction,



### 3. Ensemble Methods: Random Forest Classifier

#### Advantages:

- 1) Random forests is considered as a **highly accurate and robust method** because of the number of decision trees participating in the process.
- 2) It **does not suffer from the overfitting problem**. The main reason is that it takes the average of all the predictions, which cancels out the biases.
- 3) The algorithm can be used in both **classification** and **regression** problems.

#### Disadvantages:

- 1) Random forests is **slow** in generating predictions because it has multiple decision trees. Whenever it makes a prediction, all the trees in the forest have to make a prediction for the same given input and then perform voting on it. This whole process is time-consuming.
- 2) The model is **difficult to interpret** compared to a decision tree, where you can easily make a decision by following the path in the tree.

### 3. Ensemble Methods: Random Forest Classifier

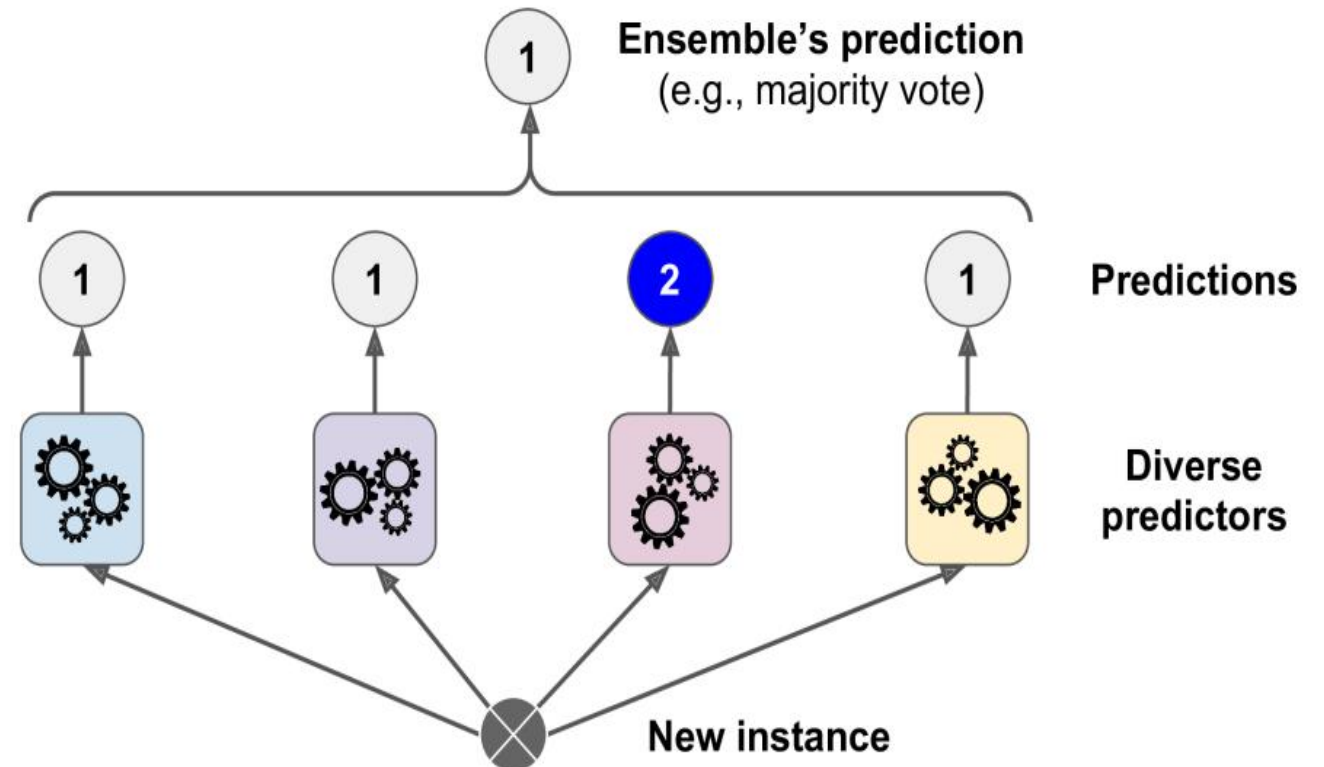
```
# Import scikit-learn dataset library
from sklearn import datasets
# Load dataset
dat = datasets.load_iris()
X = dat.data
y = dat.target
# Split dataset into training set and test set 70% training and 30% test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=100)
# training the model on training set
from sklearn.ensemble import RandomForestClassifier
# n_estimators : The number of trees in the forest.
# max_depth : The maximum depth of the tree.
# n_jobsint : The number of jobs to run in parallel
rf_clf = RandomForestClassifier(n_estimators=10)
rf_clf.fit(X_train, y_train)
# making predictions on the testing set
predicted = rf_clf.predict(X_test)
# comparing actual response values (y_test) with predicted response values (predicted)
from sklearn import metrics
print("Classification report : \n", rf_clf, "\n", metrics.classification_report(y_test, predicted))
disp = metrics.plot_confusion_matrix(rf_clf, X_test, y_test)
disp.figure_.suptitle("Confusion Matrix")
print("Confusion matrix: \n", disp.confusion_matrix)
```

### 3. Ensemble Methods: Voting Classifier

The idea behind the VotingClassifier is to combine conceptually different machine learning classifiers and use a **majority vote (Hard Voting)** or **the average predicted probabilities (soft vote)** to predict the class labels. Such a classifier can be useful for a set of equally well performing model in order to balance out their individual weaknesses.

*Ensemble methods work best when the predictors are as **independent from one another as possible**.*

*One way to get diverse classifiers is to **train them using very different algorithms**. This increases the chance that they will make very different types of errors, improving the ensemble's accuracy.*



### 3. Ensemble Methods: Voting Classifier

#### Majority/Hard Voting

E.g., if the prediction for a given sample is

classifier 1 -> class 1

classifier 2 -> class 1

classifier 3 -> class 2

the VotingClassifier (with voting='hard') would classify the sample as “**class 1**” based on the majority class label.

### 3. Ensemble Methods: Voting Classifier - (Soft Voting)

**Soft voting** returns the class label as argmax of the sum of **predicted probabilities**. Specific weights can be assigned to each classifier via the weights parameter. When weights are provided, the predicted class probabilities for each classifier are collected, multiplied by the classifier weight, and averaged. The final class label is then derived from the class label with the highest average probability.

#### Example:

$w_1=1, w_2=1, w_3=1$ .

Here, the predicted class label is **2**, since it has the highest average probability.

classifier	class 1	class 2	class 3
classifier 1	$w_1 * 0.2$	$w_1 * 0.5$	$w_1 * 0.3$
classifier 2	$w_2 * 0.6$	$w_2 * 0.3$	$w_2 * 0.1$
classifier 3	$w_3 * 0.3$	$w_3 * 0.4$	$w_3 * 0.3$
weighted average	0.37	0.4	0.23

### 3. Ensemble Methods: Voting Classifier

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
from sklearn import metrics

dat = datasets.load_breast_cancer()
print("Examples = ", dat.data.shape, " Labels = ", dat.target.shape)
X_train, X_test, Y_train, Y_test = train_test_split(dat.data,
                                                    dat.target, test_size=0.20, random_state=100)

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()
voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)], voting='hard')
voting_clf.fit(X_train, Y_train)

for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    scores = cross_val_score(clf, X, y, scoring='accuracy', cv=5)
    print("-----")
    print("cls :", clf.__class__.__name__, " Accuracy: ", scores.mean(), " std : ", scores.std())
    print("-----")
```

### 3. Ensemble Methods: Voting Classifier - (Soft Voting)

```
from sklearn import datasets
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from itertools import product
from sklearn.ensemble import VotingClassifier
from sklearn.model_selection import train_test_split

# Loading some example data
dat = datasets.load_iris()
X_train, X_test, Y_train, Y_test = train_test_split(dat.data, dat.target, test_size= 0.20, random_state=100)
# Training classifiers
clf1 = DecisionTreeClassifier(max_depth=4)
clf2 = KNeighborsClassifier(n_neighbors=7)
clf3 = SVC(kernel='rbf', probability=True)

voting_clf_soft = VotingClassifier(estimators=[('dt', clf1), ('knn', clf2), ('svc', clf3)],
                                   voting='soft', weights=[2, 1, 3])

for clf in (clf1, clf2, clf3, voting_clf_soft):
    clf.fit(X_train, Y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(Y_test, y_pred))
```



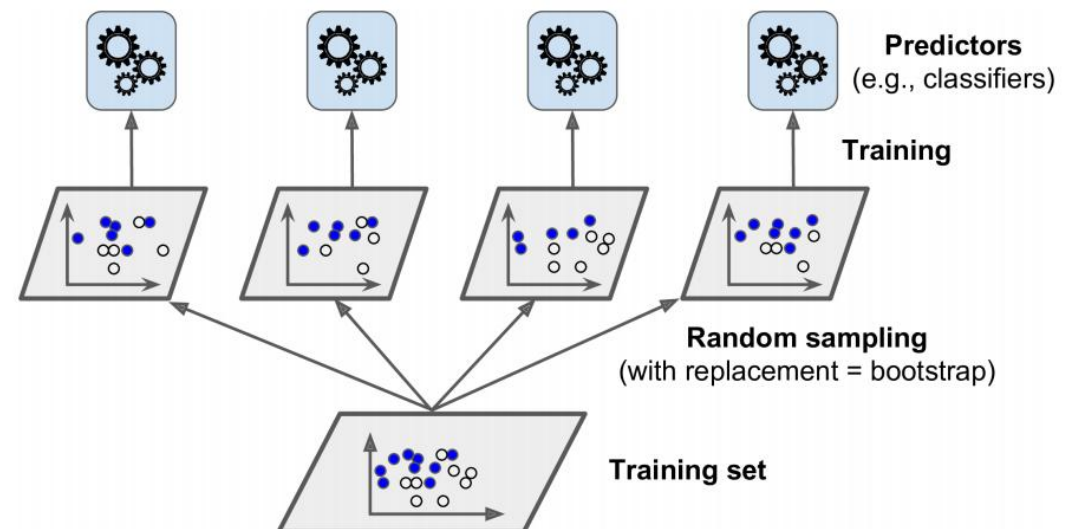
# 3. Ensemble Methods: Voting Regressor

```
from sklearn.datasets import load_boston
from sklearn.ensemble import GradientBoostingRegressor, RandomForestRegressor, VotingRegressor
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn import metrics
import numpy as np
# Loading some example data
dat = load_boston()
X_train, X_test, Y_train, Y_test = train_test_split(dat.data,
                                                    dat.target, test_size= 0.20, random_state=100)
# Training classifiers
reg1 = GradientBoostingRegressor(random_state=1, n_estimators=10)
reg2 = RandomForestRegressor(random_state=1, n_estimators=10)
reg3 = LinearRegression()
voting_reg = VotingRegressor(estimators=[('gb', reg1), ('rf', reg2), ('lr', reg3)], weights=[1, 3, 2])
for clf in (reg1, reg2, reg3, voting_reg):
    clf.fit(X_train, Y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, " : *****")
    print('Mean Absolute Error   : ', metrics.mean_absolute_error(Y_test, y_pred))
    print('Mean Squared Error    : ', metrics.mean_squared_error(Y_test, y_pred))
    print('Root Mean Squared Error: ', np.sqrt(metrics.mean_squared_error(Y_test, y_pred)))
```

### 3. Ensemble Methods: Bagging and Pasting

In this approach, **we use the same training algorithm** for every predictor, but to train them on **different random subsets** of the training set and then aggregate their individual predictions (either by voting or by averaging) to form a final prediction. When sampling is performed with replacement, this method is called **bagging** (short for bootstrap aggregating). When sampling is performed without replacement, it is called **pasting**.

This classifier can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.



### 3. Ensemble Methods: Bagging and Pasting

How bagging works on an imaginary training dataset is shown below. Since **Bagging resamples the original training dataset** with **replacement**, some instance(or data) may be **present multiple times** while others are **left out**.

Original training dataset: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Resampled training set 1: 2, 3, 3, 5, 6, 1, 8, 10, 9, 1

Resampled training set 2: 1, 1, 5, 6, 3, 8, 9, 10, 2, 7

Resampled training set 3: 1, 5, 8, 9, 2, 10, 9, 7, 5, 4

### 3. Ensemble Methods: Bagging and Pasting

This is an example of bagging, but if you want to use pasting instead, just set `bootstrap=False`

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
```

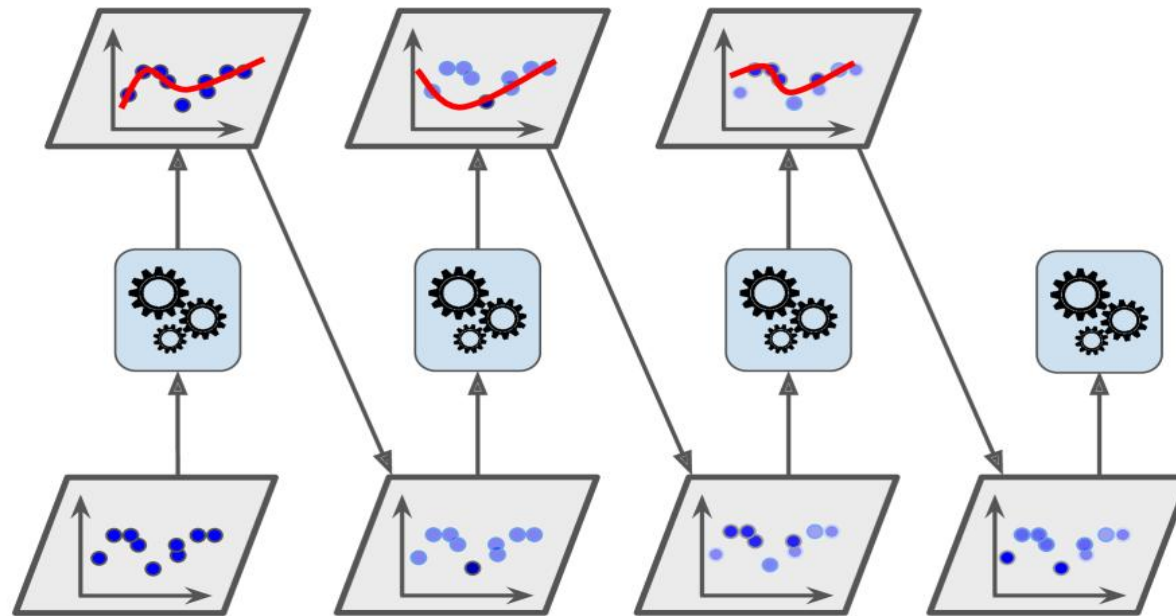
```
bag_clf = BaggingClassifier(  
DecisionTreeClassifier(), n_estimators=500, max_samples=100,  
bootstrap=True, n_jobs=-1)
```

```
bag_clf.fit(X_train, y_train)  
y_pred = bag_clf.predict(X_test)
```

# 3. Ensemble Methods: AdaBoost

The technique used by AdaBoost is the new predictor correcting its predecessor.

An AdaBoost classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where **the weights of incorrectly classified instances are adjusted** such that subsequent classifiers **focus more on difficult cases**.



### 3. Ensemble Methods: AdaBoost

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y, test_size= 0.20, random_state=100)

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=100,
    algorithm="SAMME.R", learning_rate=0.5)
ada_clf.fit(X_train, y_train)

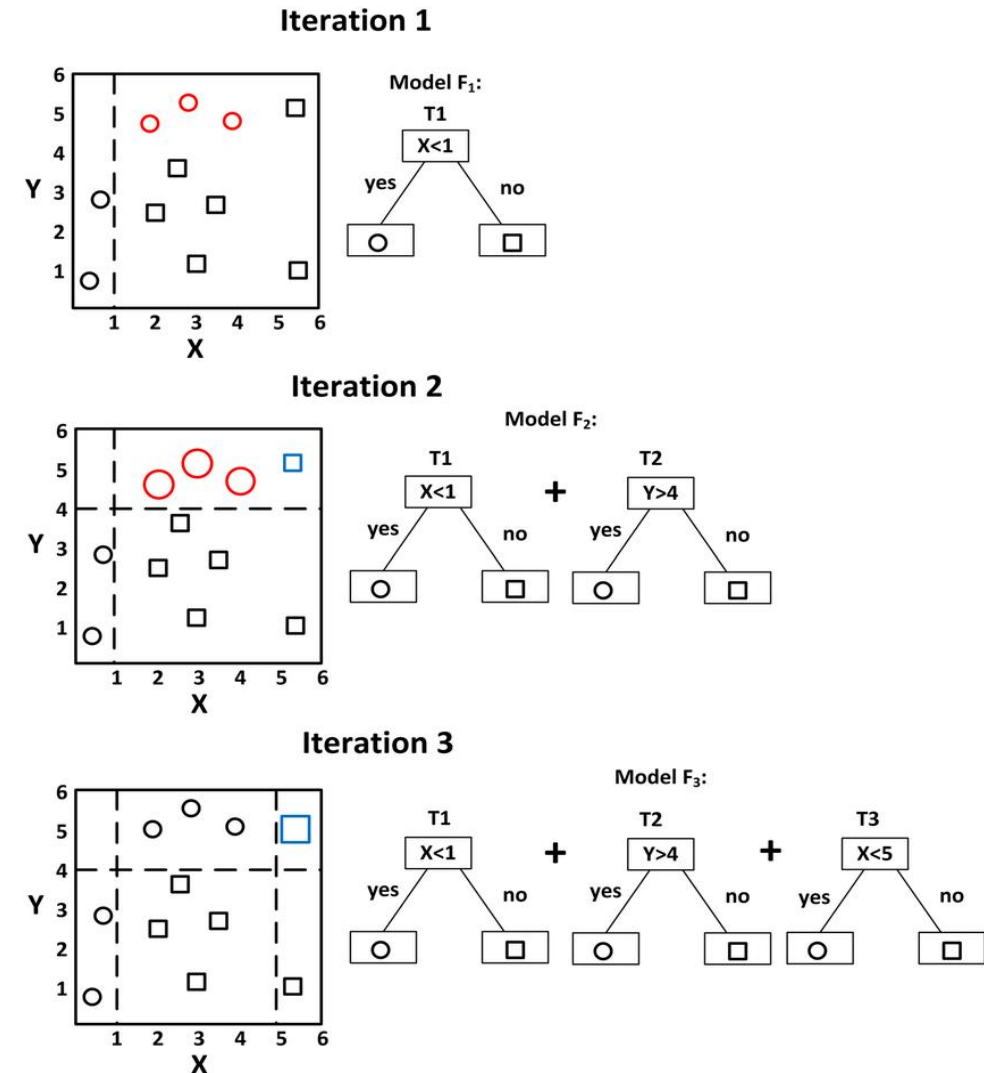
# making predictions on the testing set
predicted = ada_clf.predict(X_test)
# comparing actual response values (y_test) with predicted response values (predicted)
from sklearn import metrics
print("Classification report : \n", ada_clf, "\n", metrics.classification_report(y_test, predicted))
disp = metrics.plot_confusion_matrix(ada_clf, X_test, y_test)
disp.figure_.suptitle("Confusion Matrix")
print("Confusion matrix: \n", disp.confusion_matrix)
```

### 3. Ensemble Methods: Gradient Boosting

- Gradient boosting classifiers are a **group of machine learning algorithms** that combine many weak learning models together **to create a strong** predictive model.
- **Decision trees** are usually used when doing gradient boosting.
- Gradient boosting models are becoming **popular** because of their **effectiveness** at classifying **complex datasets**, and have recently been used to win many Kaggle data science competitions.

# 3. Ensemble Methods: Gradient Boosting

Instead of adjusting the instance weights at every iteration as AdaBoost does, this method **tries to fit the new predictor to the residual errors made by the previous predictor.**





### 3. Ensemble Methods: Gradient Boosting - Classification

```
from sklearn import datasets
# Load dataset
dat = datasets.load_iris()
X = dat.data
y = dat.target
# Split dataset into training set and test set 70% training and 30% test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=100)

# training the model on training set
from sklearn.ensemble import GradientBoostingClassifier
clf = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1,
    max_depth=5, random_state=0)
clf.fit(X_train, y_train)

# making predictions on the testing set
predicted = clf.predict(X_test)
# comparing actual response values (y_test) with predicted response values (predicted)
from sklearn import metrics
print("Classification report : \n", clf, "\n", metrics.classification_report(y_test, predicted))
disp = metrics.plot_confusion_matrix(clf, X_test, y_test)
disp.figure_.suptitle("Confusion Matrix")
print("Confusion matrix: \n", disp.confusion_matrix)
```

### 3. Ensemble Methods: Gradient Boosting - Regression

```
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn import metrics
import numpy as np
# Loading some example data
dat = load_boston()
X_train, X_test, y_train, y_test = train_test_split(dat.data,
                                                    dat.target, test_size= 0.20, random_state=100)

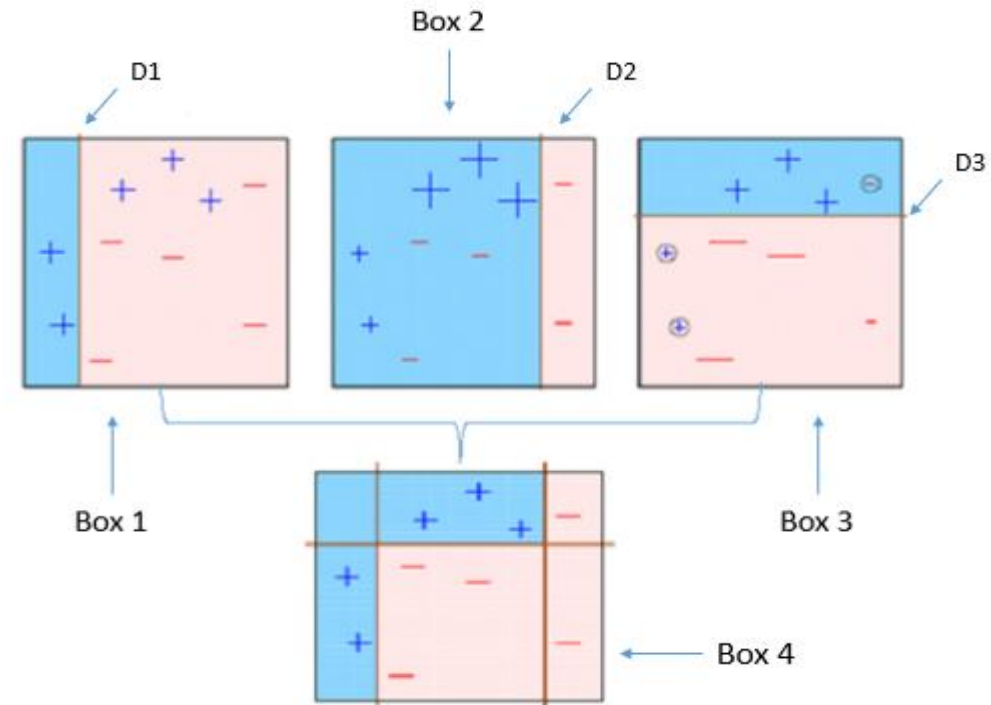
# Training classifiers
from sklearn.ensemble import GradientBoostingRegressor
Reg = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1,
                                max_depth=5, random_state=0)
Reg.fit(X_train, y_train)

y_pred = clf.predict(X_test)
print('Mean Absolute Error   : ', metrics.mean_absolute_error(Y_test, y_pred))
print('Mean Squared Error    : ', metrics.mean_squared_error(Y_test, y_pred))
print('Root Mean Squared Error: ', np.sqrt(metrics.mean_squared_error(Y_test, y_pred)))
```

### 3. Ensemble Methods: **XGBoost**

**XGBoost** is a specific implementation of the **Gradient Boosting** method which uses **more accurate approximations** to find the best tree model. It employs a number of nifty tricks that make it exceptionally successful, particularly with structured data. **XGBoost is often an important component of the winning entries in ML competitions.**

- 1) Speed and performance
- 2) Core algorithm is parallelizable
- 3) Consistently outperforms other algorithm methods
- 4) Wide variety of tuning parameters



### 3. Ensemble Methods: XGBoost - Classification

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X,
    y, test_size= 0.20, random_state=100)
# In order for XGBoost to be able to use our data,
# we'll need to transform it into a specific format that XGBoost can handle.
D_train = xgb.DMatrix(X_train, label=y_train)
D_test = xgb.DMatrix(X_test, label=y_test)
# Training classifier
import xgboost as xgb
param = {'eta': 0.3, 'max_depth': 3, 'objective':
'multi:softprob',
    'num_class': 3}
steps = 20 # The number of training iterations
xg_clf = xgb.train(param, D_train, steps)
# making predictions on the testing set
predicted = xg_clf.predict(D_test)
predicted = predicted.tolist()
predicted = [round( value.index(max(value)) ) for value in predicted]

# comparing actual response values (y_test) with predicted response values (predicted)
from sklearn import metrics
print("Classification report : \n", xg_clf, "\n", metrics.classification_report(y_test, predicted))
disp.figure_.suptitle("Confusion Matrix")
print("Confusion matrix: \n", disp.confusion_matrix)
```

```
# Visualize Boosting Trees and
# Feature Importance
import matplotlib.pyplot as plt
xgb.plot_tree(xg_clf,num_trees=0)
plt.rcParams['figure.figsize'] = [10,
10]
plt.show()

xgb.plot_importance(xg_clf)
plt.rcParams['figure.figsize'] = [5, 5]
plt.show()
```

### 3. Ensemble Methods: XGBoost - Regression

```
import xgboost as xgb
xg_reg = xgb.XGBRegressor(objective="reg:squarederror", colsample_bytree = 0.3,
                           learning_rate = 0.1, max_depth = 5,
                           alpha = 10, n_estimators = 10)

xg_reg.fit(xtrain,ytrain)

score = xg_reg.score(xtrain, ytrain)
print("Training score: ", score)

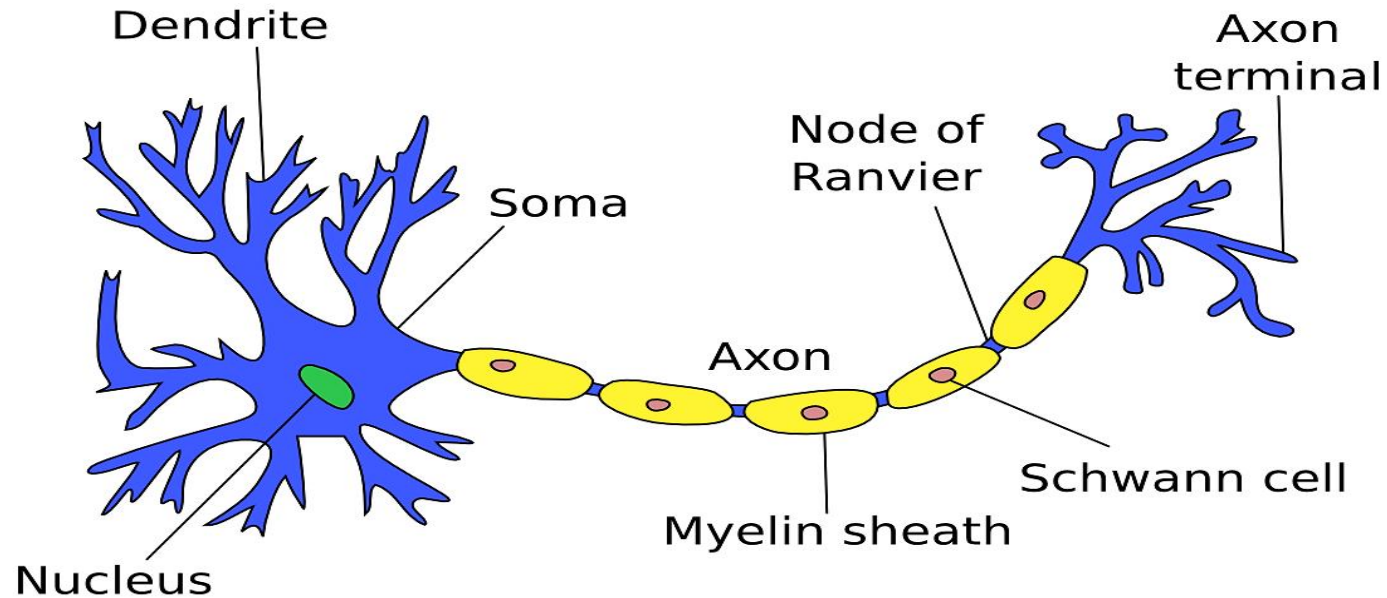
ypred = xg_reg.predict(xtest)
mse = mean_squared_error(ytest, ypred)
print("MSE: %.2f" % mse)
print("RMSE: %.2f" % (mse**(1/2.0)))

x_ax = range(len(ytest))
plt.plot(x_ax, ytest, label="original")
plt.plot(x_ax, ypred, label="predicted")
plt.title("Boston test and predicted data")
plt.legend()
plt.show
```

# Neural Networks

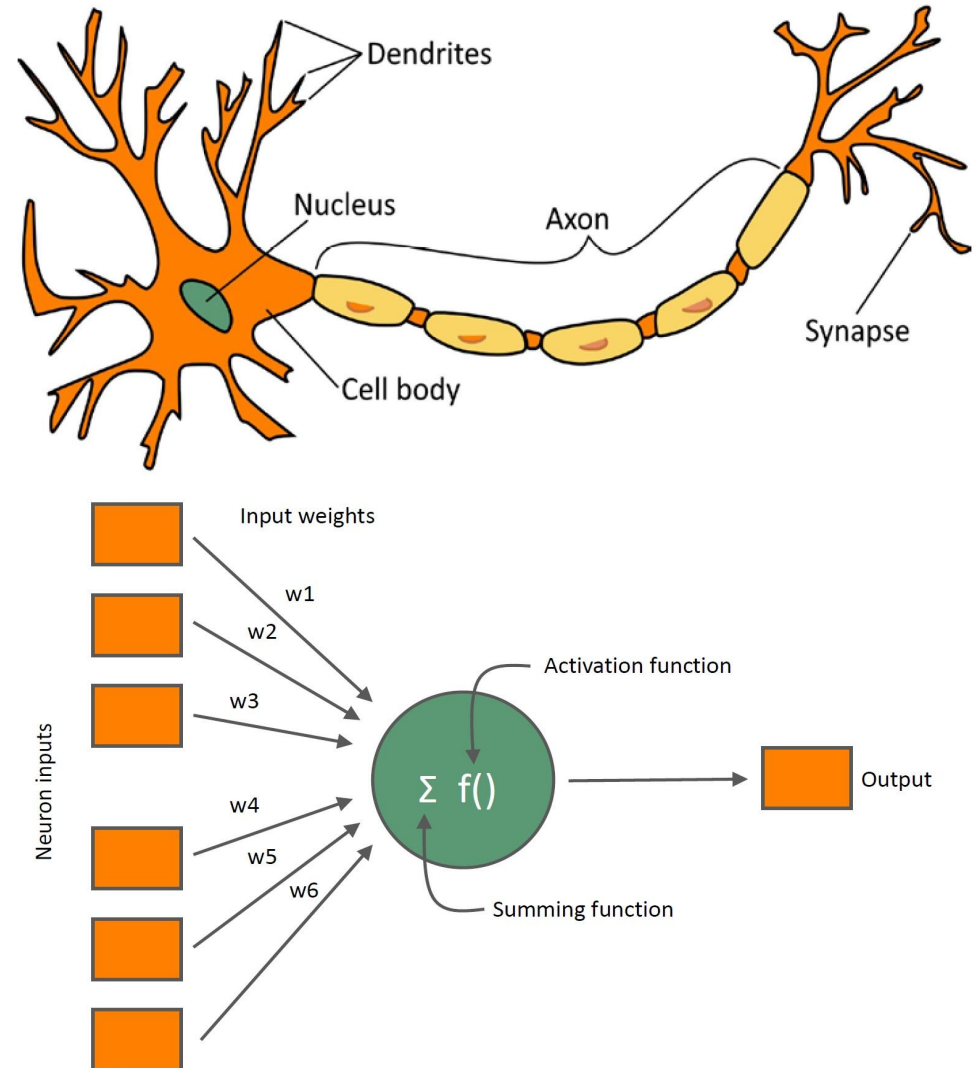
# Human Brain Network

- The brain is composed of **neurons** linked together by **synapses**.
- Human brain has  **$\sim 10^{11}$  neurons**  $\approx$  number of trees in the amazon forest.
- The number of synapses  $\approx$  number of **tree leaves** in the amazon forest.



# From Biological to Artificial Neurons

- Biological neurons are **fired** based on the intensity of the entering signals.
- Can be simulated by **activation functions**.
- **Artificial neurons** are arranged in layers and linked by **weights** in a similar way to **synapses** linking biological neurons.





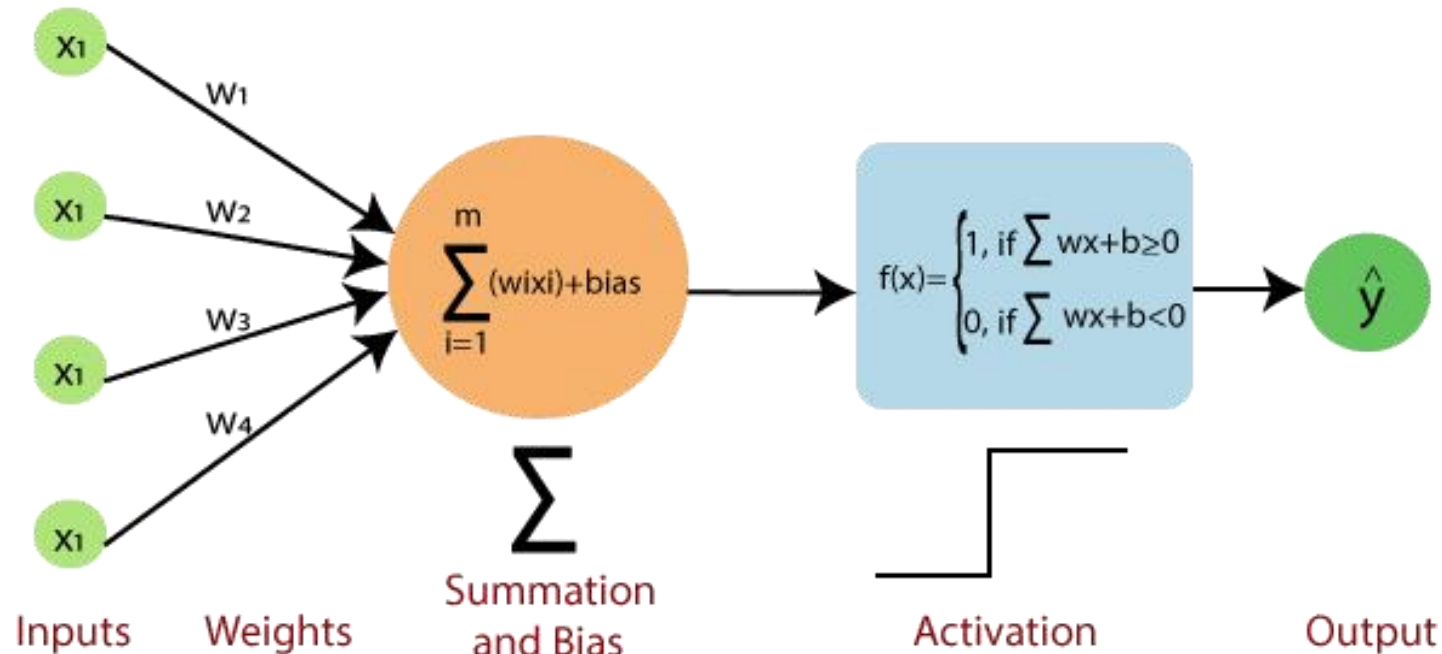
# Perceptron

- Has been invented in **1957** par Frank Rosenblatt at the Aerospatiale labs at Cornell University.
- Used **analog hardware** to ensure **connections** between neurons.



# Perceptron

- The inputs and output are numbers and each input connection is associated with a weight.
- Compute the **weighted sum of its inputs** then applies an **activation function** to that sum and **outputs the result**.



# Algorithm for Training a Perceptron

Input :  $(\mathcal{D}, \mathbf{w}^{(0)})$ .

Output :  $\mathbf{w}$ .

for each data point  $\mathbf{x}^{(j)}, j = 1, \dots, n$ , do

if  $(y^{(j)} = +1 \text{ and } f(\mathbf{x}^{(j)}) \leq 0)$  then

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \alpha \mathbf{x}^{(j)}$$



else if  $(y^{(j)} = -1 \text{ and } f(\mathbf{x}^{(j)}) \geq 0)$  then

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha \mathbf{x}^{(j)}$$



else

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)}.$$



end

end

end

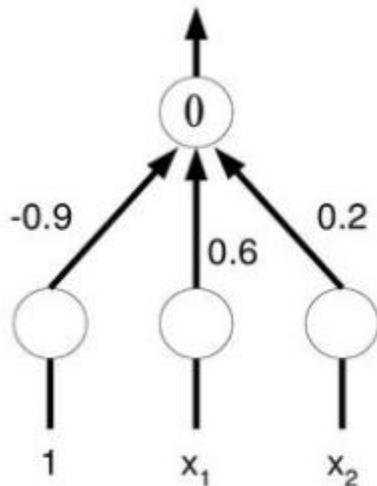
# Algorithm for Training a Perceptron

## Our Dataset

X1	X2	R
1	1	1
1	-1	-1
-1	1	-1
-1	-1	-1

training set:

$x_1 = 1, x_2 = 1 \rightarrow 1$   
 $x_1 = 1, x_2 = -1 \rightarrow -1$   
 $x_1 = -1, x_2 = 1 \rightarrow -1$   
 $x_1 = -1, x_2 = -1 \rightarrow -1$



randomly, let:  $w_0 = -0.9$ ,  $w_1 = 0.6$ ,  $w_2 = 0.2$

using these weights:

$x_1 = 1, x_2 = 1$ :  $-0.9 \cdot 1 + 0.6 \cdot 1 + 0.2 \cdot 1 = -0.1 \rightarrow -1$   
 $x_1 = 1, x_2 = -1$ :  $-0.9 \cdot 1 + 0.6 \cdot 1 + 0.2 \cdot -1 = -0.5 \rightarrow -1$   
 $x_1 = -1, x_2 = 1$ :  $-0.9 \cdot 1 + 0.6 \cdot -1 + 0.2 \cdot 1 = -1.3 \rightarrow -1$   
 $x_1 = -1, x_2 = -1$ :  $-0.9 \cdot 1 + 0.6 \cdot -1 + 0.2 \cdot -1 = -1.7 \rightarrow -1$

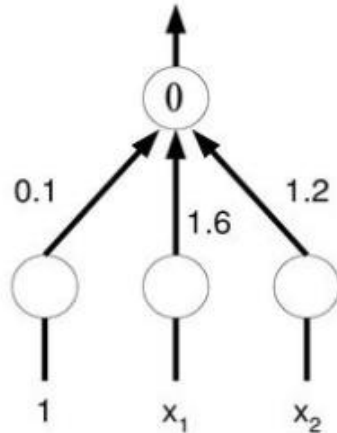
WRONG

OK  
OK  
OK

new weights:  $w_0 = -0.9 + 1 = 0.1$   
 $w_1 = 0.6 + 1 = 1.6$   
 $w_2 = 0.2 + 1 = 1.2$

$\alpha = 1$

# Algorithm for Training a Perceptron



using these updated weights:

$$x_1 = 1, x_2 = 1: 0.1*1 + 1.6*1 + 1.2*1 = 2.9 \rightarrow 1 \quad \text{OK}$$

$$x_1 = 1, x_2 = -1: 0.1*1 + 1.6*1 + 1.2*(-1) = 0.5 \rightarrow 1 \quad \text{WRONG}$$

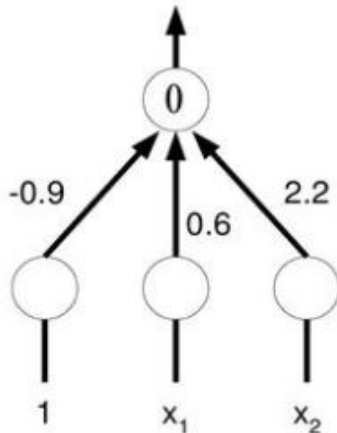
$$x_1 = -1, x_2 = 1: 0.1*1 + 1.6*(-1) + 1.2*1 = -0.3 \rightarrow -1 \quad \text{OK}$$

$$x_1 = -1, x_2 = -1: 0.1*1 + 1.6*(-1) + 1.2*(-1) = -2.7 \rightarrow -1 \quad \text{OK}$$

new weights:  $w_0 = 0.1 - 1 = -0.9$

$$w_1 = 1.6 - 1 = 0.6$$

$$w_2 = 1.2 + 1 = 2.2$$



using these updated weights:

$$x_1 = 1, x_2 = 1: -0.9*1 + 0.6*1 + 2.2*1 = 1.9 \rightarrow 1 \quad \text{OK}$$

$$x_1 = 1, x_2 = -1: -0.9*1 + 0.6*1 + 2.2*(-1) = -2.5 \rightarrow -1 \quad \text{OK}$$

$$x_1 = -1, x_2 = 1: -0.9*1 + 0.6*(-1) + 2.2*1 = 0.7 \rightarrow 1 \quad \text{WRONG}$$

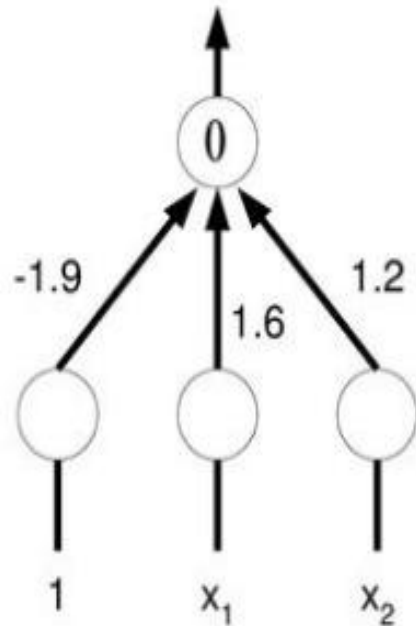
$$x_1 = -1, x_2 = -1: -0.9*1 + 0.6*(-1) + 2.2*(-1) = -3.7 \rightarrow -1 \quad \text{OK}$$

new weights:  $w_0 = -0.9 - 1 = -1.9$

$$w_1 = 0.6 + 1 = 1.6$$

$$w_2 = 2.2 - 1 = 1.2$$

# Algorithm for Training a Perceptron



using these updated weights:

$$x_1 = 1, x_2 = 1: -1.9*1 + 1.6*1 + 1.2*1 = 0.9 \rightarrow 1 \quad \text{OK}$$

$$x_1 = 1, x_2 = -1: -1.9*1 + 1.6*1 + 1.2*-1 = -1.5 \rightarrow -1 \quad \text{OK}$$

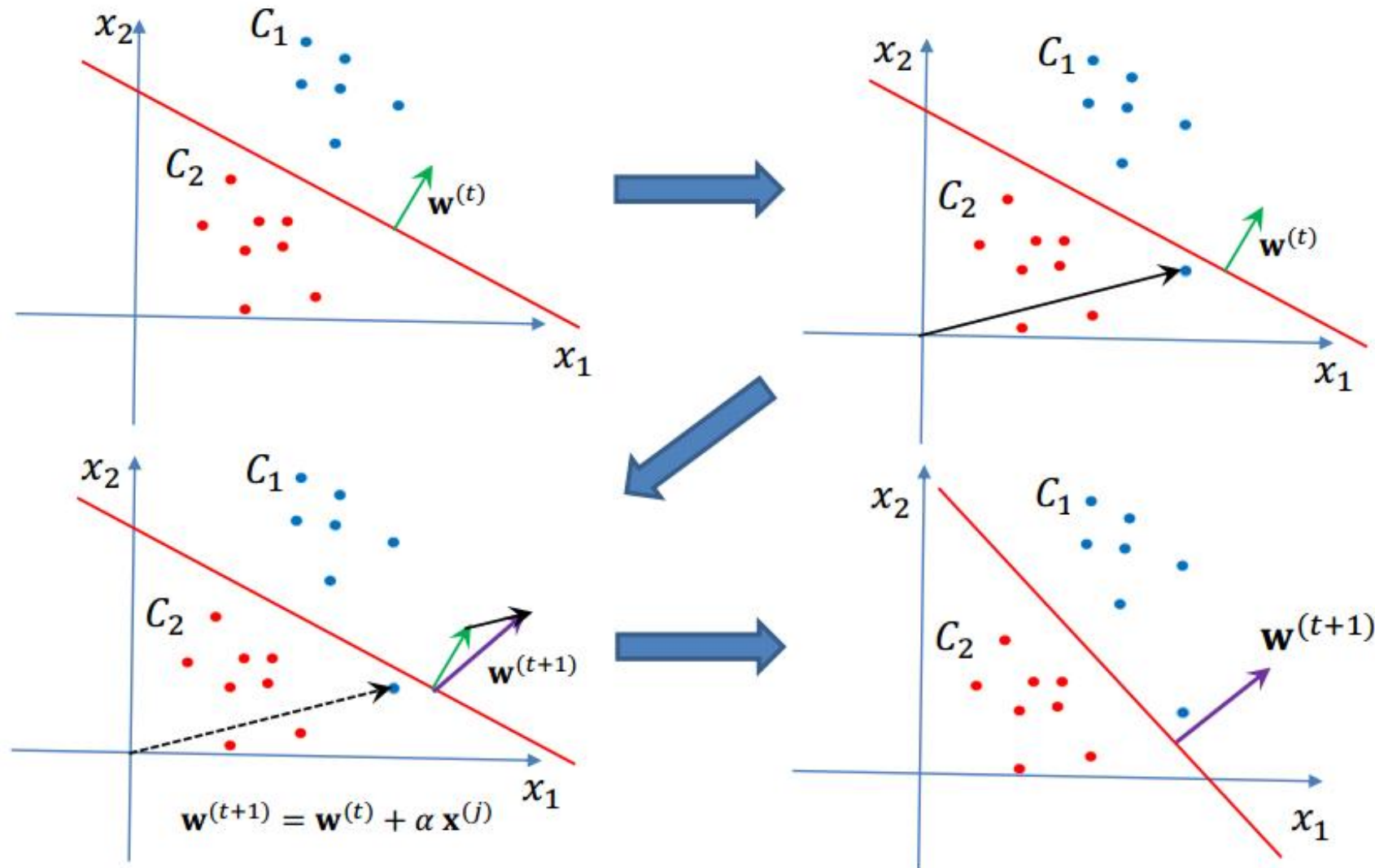
$$x_1 = -1, x_2 = 1: -1.9*1 + 1.6*-1 + 1.2*1 = -2.3 \rightarrow -1 \quad \text{OK}$$

$$x_1 = -1, x_2 = -1: -1.9*1 + 1.6*-1 + 1.2*-1 = -4.7 \rightarrow -1 \quad \text{OK}$$

**DONE!**



# Algorithm for Training a Perceptron



# Limitations of Perceptron

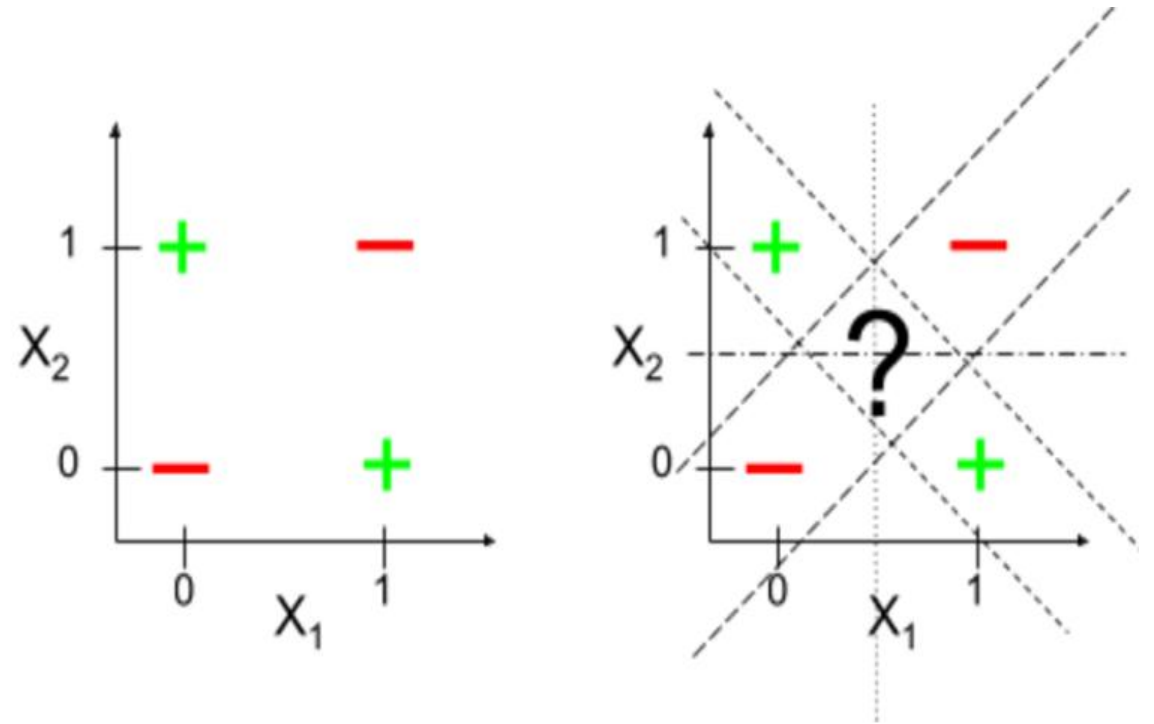
A perceptron can only separate **linearly separable classes**, but it is unable to separate **non-linear class boundaries**.

**Example:** Let the following problem of binary classification (problem of the **XOR**).

**Clearly, no line can separate the two classes!**

**Solution :**

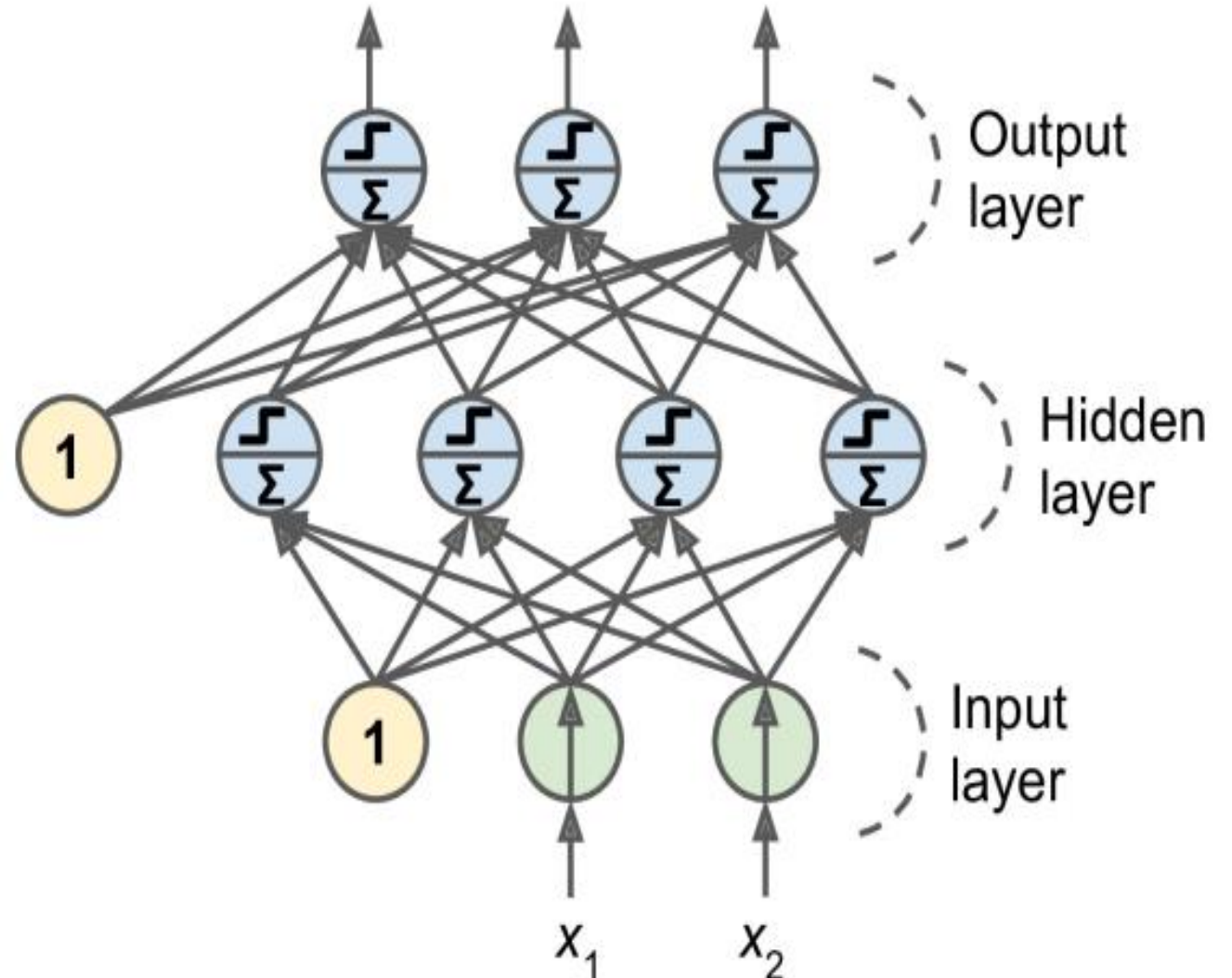
- Use two lines instead of one!
- Use an intermediary layer of neurons in the NN.





# The Multilayer Perceptron MLP

- The signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a **feedforward neural network (FNN)**.
- When an ANN contains a deep stack of hidden layers, it is called a **deep neural network (DNN)**.

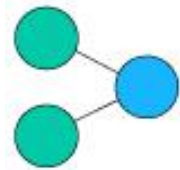


# The Multilayer Perceptron MLP

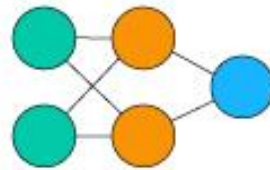
- In 1986, the **backpropagation** training algorithm was introduced, which is still used today.
- **The backpropagation** consists of only two passes through the network (**one forward, one backward**), the backpropagation algorithm is able to compute the gradient of the **network's error** with regard to every single model parameter. In other words, it can find out how each **connection weight** and each **bias** term should be tweaked in order to reduce the error.

David Rumelhart et al. "Learning Internal Representations by Error Propagation," (Defense Technical Information Center technical report, September 1985).

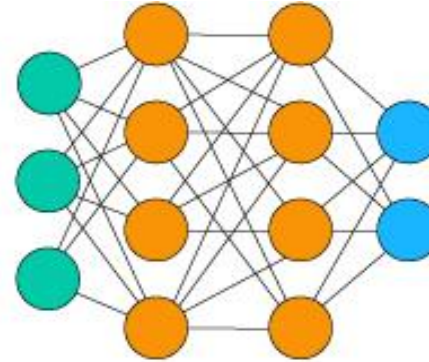
# Popular NN Architecture



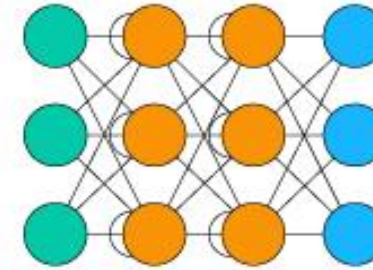
Single Layer Perceptron



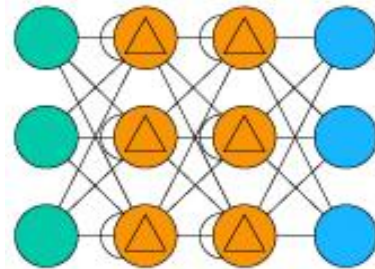
Radial Basis Network (RBN)



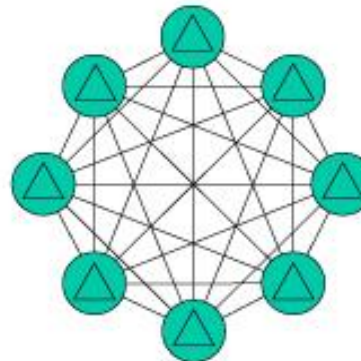
Multi Layer Perceptron



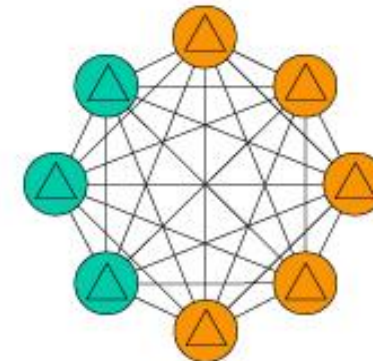
Recurrent Neural Network



LSTM Recurrent Neural Network



Hopfield Network



Boltzmann Machine

● Input Unit

● Hidden Unit

△ Backfed Input Unit

● Output Unit

△ Feedback with Memory Unit

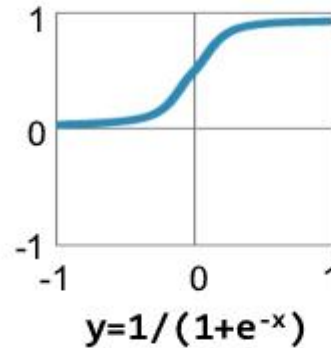
△ Probabilistic Hidden Unit

Hichem Felouat - hichemfel@gmail.com

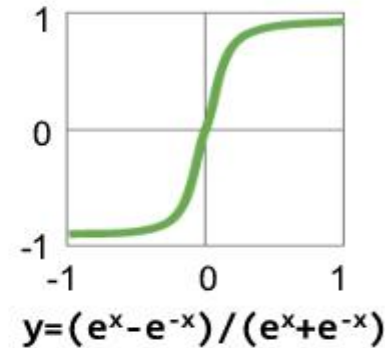
# Popular Activation functions for MLP

Traditional  
Non-Linear  
Activation  
Functions

Sigmoid

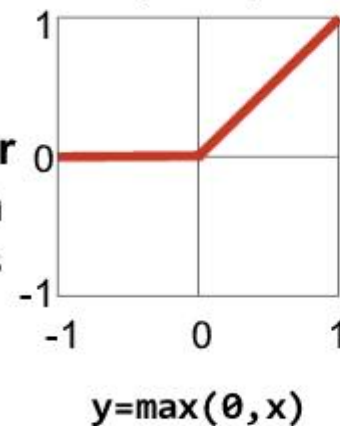


Hyperbolic Tangent

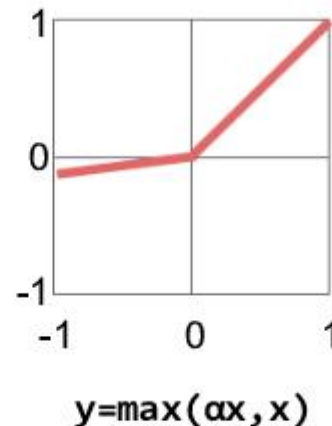


Modern  
Non-Linear  
Activation  
Functions

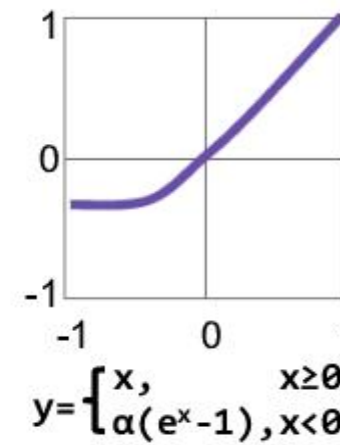
Rectified Linear Unit  
(ReLU)



Leaky ReLU



Exponential LU



$\alpha = \text{small const. (e.g. 0.1)}$

# **Neural Network vocabulary**

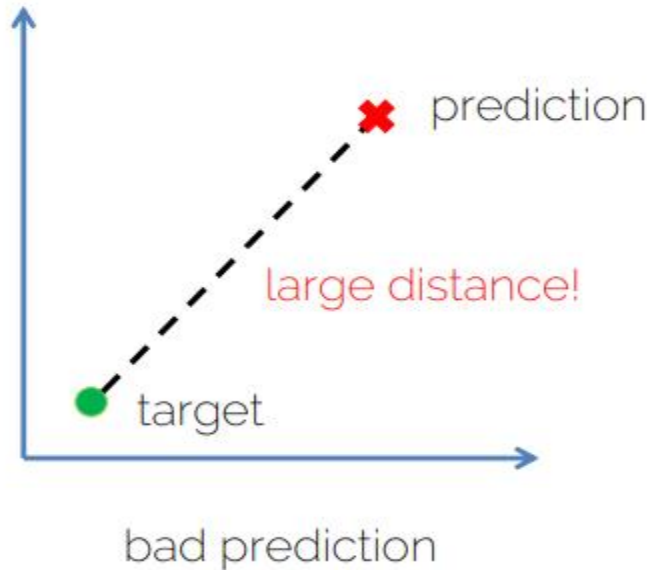
- 1) Cost Function**
- 2) Gradient Descent**
- 3) Learning Rate**
- 4) Backpropagation**
- 5) Batches**
- 6) Epochs**

# Neural Network vocabulary

**Cost Function:** When we build a network, the network tries to predict the output as close as possible to the actual value. We measure this accuracy of the network using the **cost/loss function**.

**Idea:** calculate a “**distance**” between prediction and target!

*The choice of the loss function depends on the concrete problem or the distribution of the target variable.*

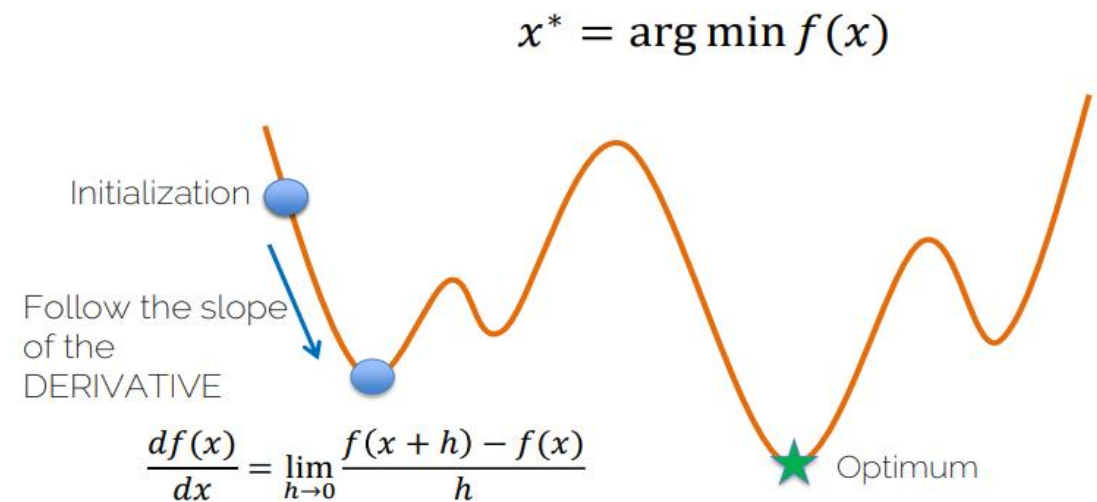
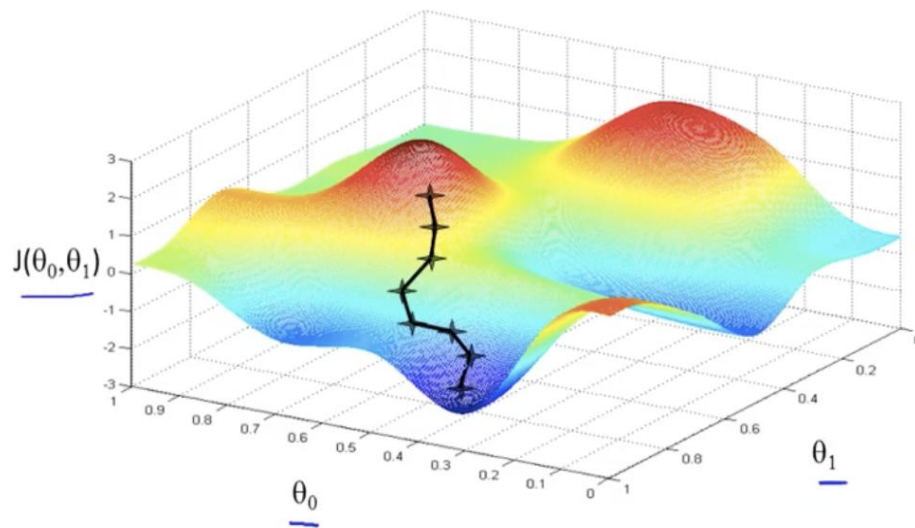




# Neural Network vocabulary

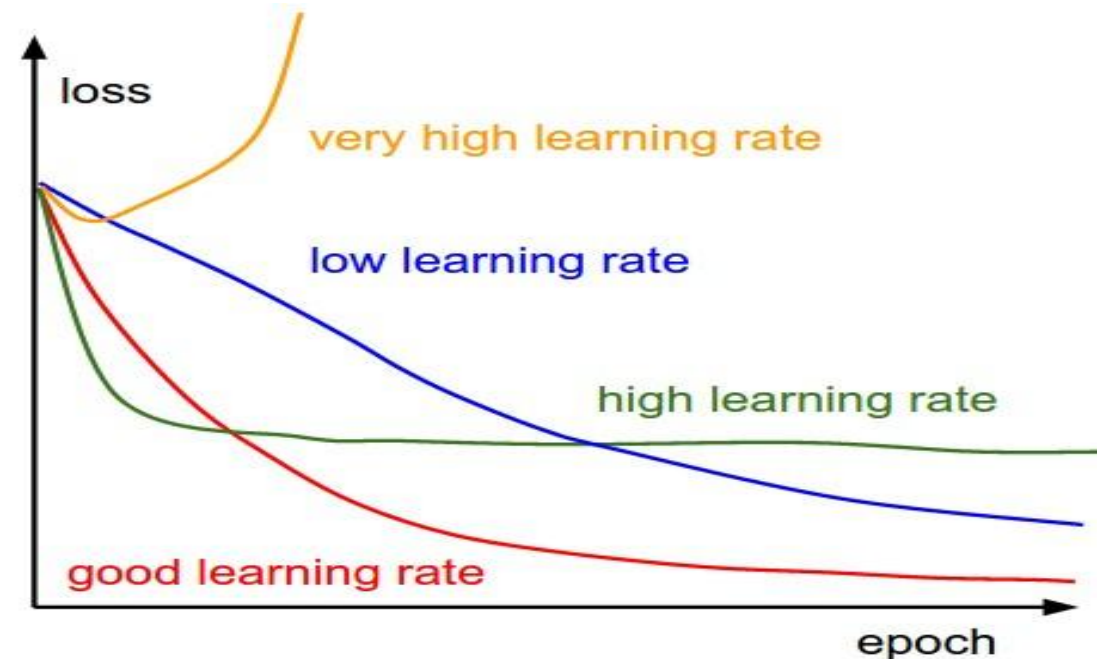
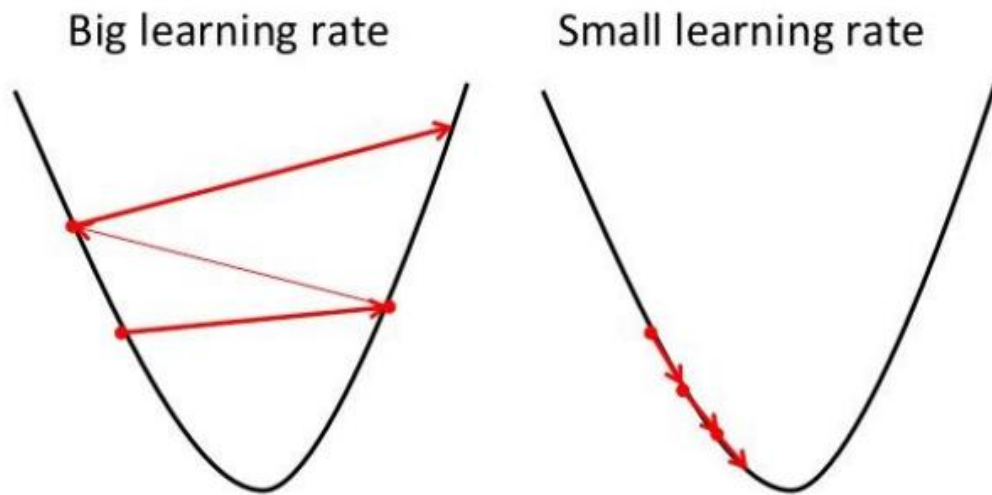
**Gradient descent:** is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In machine learning, we use gradient descent to **update the parameters of our model**. Parameters refer to **weights** in neural networks.

To put it simply, we use gradient descent to minimize the cost function,  **$J(w)$** .



# Neural Network vocabulary

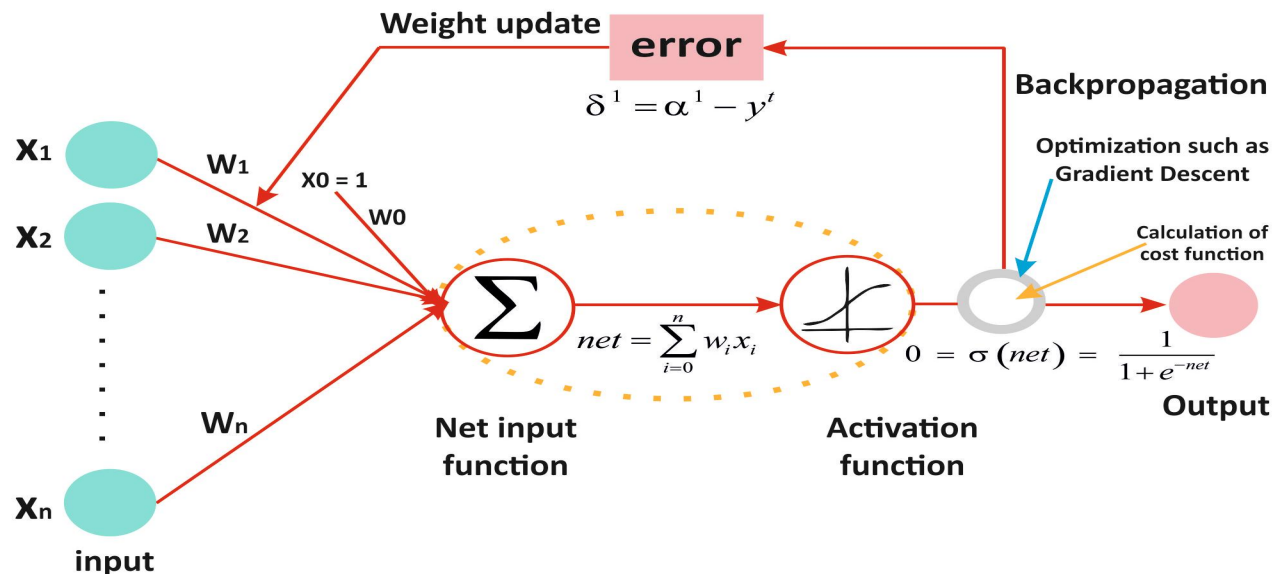
**Learning Rate:** is defined as the **amount of minimization in the cost function** in each iteration. In simple terms, the rate at which we descend towards the minima of the cost function is the learning rate. **We should choose the learning rate very carefully** since it should neither be very large that the optimal solution is missed and nor should be very low that it takes forever for the network to converge.





# Neural Network vocabulary

**Backpropagation:** after we calculated the error of the network, this error is then fed back to the network along with the gradient of the cost function to update the weights of the network. These weights are then updated so that the errors in the subsequent iterations is reduced. **This updating of weights using the gradient of the cost function is known as backpropagation.**



# Neural Network vocabulary

**Batches:** while training a neural network, instead of sending the entire input in one go, **we divide in input into several chunks of equal size randomly**. Training the data on batches makes the model **more generalized** as compared to the model built when the entire data set is fed to the network in one go.

- **It requires less memory.**
- **Typically networks train faster with mini-batches.**
- **Batch Gradient Descent** : Batch Size = Size of Training Set
- **Stochastic Gradient Descent:** Batch Size = 1
- **Mini-Batch Gradient Descent:**  $1 < \text{Batch Size} < \text{Size of Training Set}$

# Neural Network vocabulary

**Epochs:** an epoch is defined as **a single training iteration** of all batches in both **forward and backpropagation**. This means **1 epoch** is a single forward and backward pass of the entire input data.

The number of epochs you would use to train your network can be **chosen by you**. It is highly likely that **more number of epochs would show higher accuracy** of the network, however, it would also take longer for the network to converge. Also, you must take care that if the number of epochs are too high, the network might be **over-fit**.

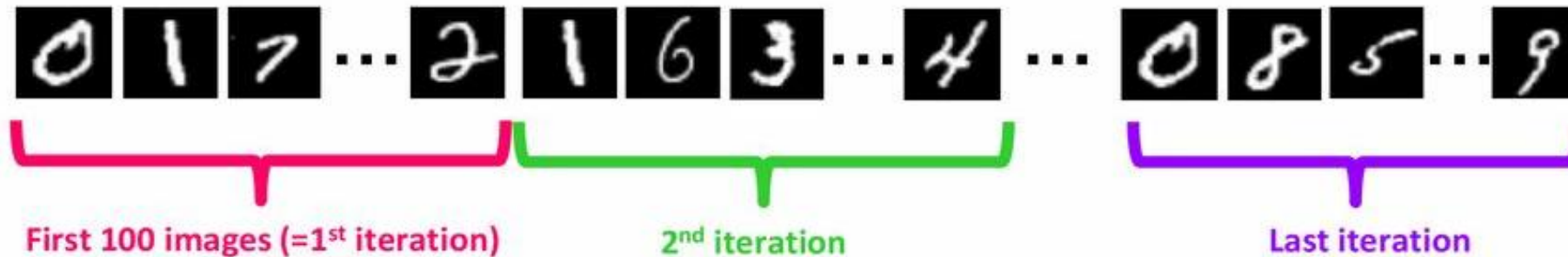
# Neural Network vocabulary

## Epoch / Iteration

---

### Example: MNIST data

- number of training data: **N=55,000**
- Let's take batch size of **B=100**



- How many iteration in each epoch?  $55000/100 = 550$

**1 epoch = 550 iteration**

# Neural Network vocabulary

**Updating weights** - In a neural network, weights are updated as follows:

- **Step 1:** Take a batch of training data.
- **Step 2:** Perform forward propagation to obtain the corresponding loss.
- **Step 3:** Backpropagate the loss to get the gradients.
- **Step 4:** Use the gradients to update the weights of the network.

# 4. Neural Networks - MLP

## # Classification

## # Feature Scaling

```
from sklearn.preprocessing import StandardScaler  
scaler = StandardScaler()  
scaler.fit(X_trainNN)
```

```
X_trainNN = scaler.transform(X_trainNN)  
X_testNN = scaler.transform(X_testNN)
```

## # Training and Predictions

```
from sklearn.neural_network import MLPClassifier  
  
mlp = MLPClassifier(hidden_layer_sizes=(128, 64,), batch_size=64,  
                    solver="adam", activation="relu", max_iter=1000)  
  
mlp.fit(X_trainNN, y_trainNN)  
  
predicted = mlp.predict(X_testNN)
```

# 4. Neural Networks - MLP

## # Regression

## # Feature Scaling

```
from sklearn import preprocessing  
scaler = preprocessing.StandardScaler()  
scaler.fit(X_trainNN)
```

```
X_trainNN = scaler.transform(X_trainNN)  
X_testNN = scaler.transform(X_testNN)
```

## # Training and Predictions

```
from sklearn.neural_network import MLPRegressor  
mlpR = MLPRegressor(hidden_layer_sizes=(128, 64,), batch_size=64,  
                      solver="adam", activation="relu", max_iter=1000)  
  
mlpR.fit(X_trainNN, y_trainNN)  
  
predicted = mlpR.predict(X_testNN)
```

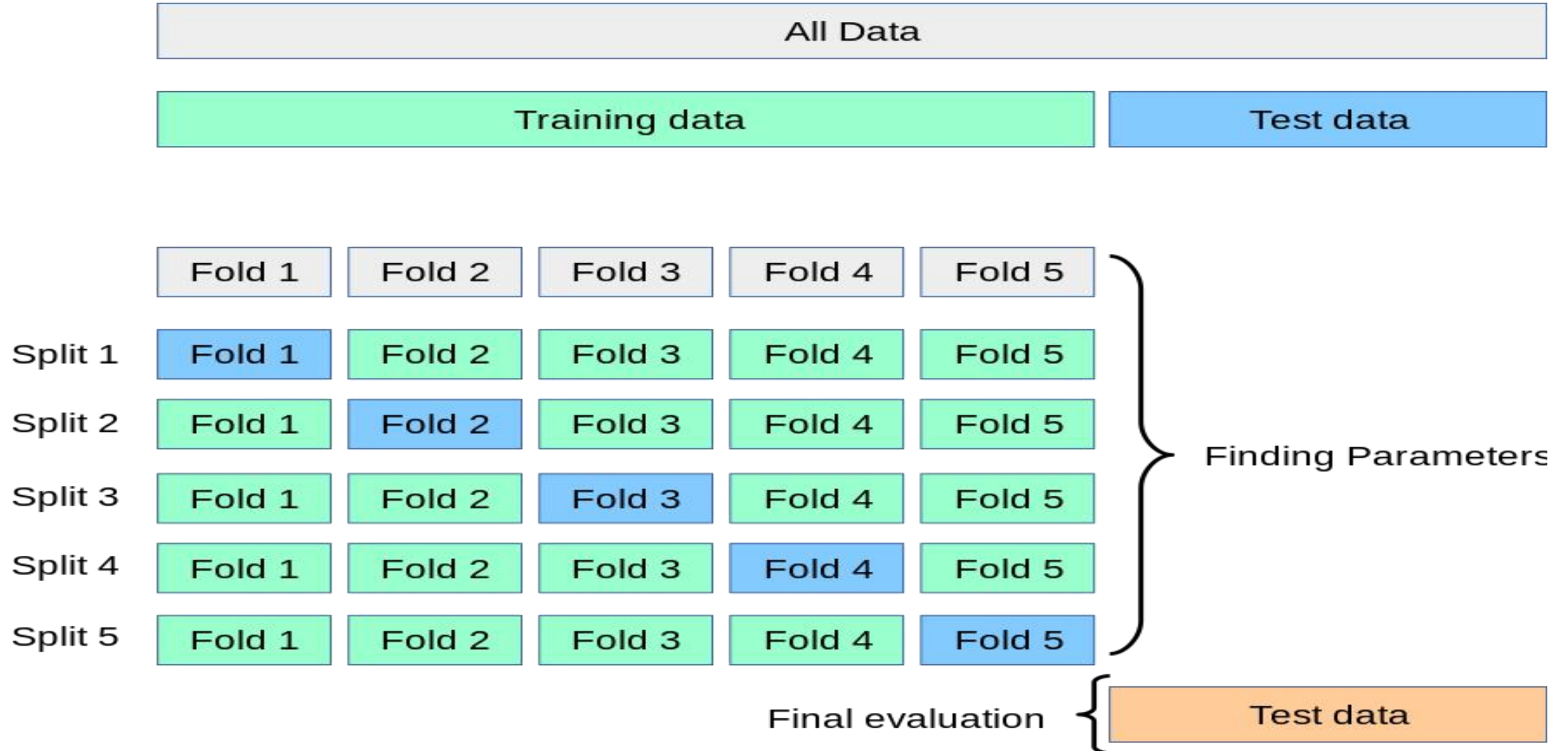
# **Model selection and evaluation**



## 5. Cross-Validation

**Cross-Validation (CV)** : the training set is **split** into **complementary subsets**, and each model is trained against a different combination of these subsets and validated against the remaining parts. Once the model type and hyperparameters have been selected, a final model is trained using these hyperparameters on the full training set, and the generalized error is measured on the test set.

## 5. Cross-Validation



# 5. Cross-Validation

**k-fold cross-validation:** the data is divided into **k folds**. The model is trained on **k-1** folds with **one fold held back for testing**. This process gets repeated **to ensure each fold of the dataset gets the chance to be the held backset**. Once the process is completed, we can summarize the evaluation metric using the **mean** or/and **the standard deviation**.

**Stratified K-Fold approach** is a variation of k-fold cross-validation that returns stratified folds, i.e., **each set containing approximately the same ratio of target labels as the complete data**.

## 5. Cross-Validation

**Leave One Out Cross-Validation (LOOCV):** is the cross-validation technique in which the size of the fold is “1” with “k” being set to the number of observations in the data. This variation is **useful when the training data is of limited size** and the number of parameters to be tested is not high.

**Repeated Random Test-Train Splits:** is a hybrid of traditional train-test splitting and the k-fold cross-validation method. In this technique, **we create random splits of the data in the training-test set** manner and then repeat the process of splitting and evaluating the algorithm multiple times, just like the cross-validation method.

# 5. Cross-Validation

```
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn import svm
from sklearn.model_selection import cross_val_score
from sklearn import model_selection
from sklearn import metrics

dat = datasets.load_breast_cancer()
print("Examples = ",dat.data.shape," Labels = ", dat.target.shape)
print("Example 0 = ",dat.data[0])
print("Label 0 =",dat.target[0])
print(dat.target)
X = dat.data
Y = dat.target
# Make a train/test split using 20% test size
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size= 0.20,
                                                    random_state=100)
print("X_test = ",X_test.shape)

print("Without Validation : *****")
model_1 = svm.SVC(kernel='linear', C=10.0, gamma= 0.1)
model_1.fit(X_train, Y_train)
y_pred1 = model_1.predict(X_test)
print("Accuracy 1 :",metrics.accuracy_score(Y_test, y_pred1))

print("K-fold Cross-Validation : *****")
from sklearn.model_selection import KFold
kfold = KFold(n_splits=10, random_state=100)
model_2 = svm.SVC(kernel='linear', C=10.0, gamma= 0.1)
results_model_2 = cross_val_score(model_2, X, Y, cv=kfold)
accuracy2 = results_model_2.mean()
print("Accuracy 2 :", accuracy2)
```

```
print("Stratified K-fold Cross-Validation : *****")
from sklearn.model_selection import StratifiedKFold
skfold = StratifiedKFold(n_splits=3, random_state=100)
model_3 = svm.SVC(kernel='linear', C=10.0, gamma= 0.1)
results_model_3 = cross_val_score(model_3, X, Y, cv=skfold)
accuracy3 = results_model_3.mean()
print("Accuracy 3 :", accuracy3)

print("Leave One Out Cross-Validation : *****")
from sklearn.model_selection import LeaveOneOut
loocv = model_selection.LeaveOneOut()
model_4 = svm.SVC(kernel='linear', C=10.0, gamma= 0.1)
results_model_4 = cross_val_score(model_4, X, Y, cv=loocv)
accuracy4 = results_model_4.mean()
print("Accuracy 4 :", accuracy4)

print("Repeated Random Test-Train Splits : *****")
from sklearn.model_selection import ShuffleSplit
kfold2 = model_selection.ShuffleSplit(n_splits=10, test_size=0.30,
                                      random_state=100)
model_5 = svm.SVC(kernel='linear', C=10.0, gamma= 0.1)
results_model_5 = cross_val_score(model_5, X, Y, cv=kfold2)
accuracy5 = results_model_5.mean()
print("Accuracy 5 :", accuracy5)
```

# 5. Hyperparameter Tuning

**Hyper-parameters are parameters that are not directly learnt within estimators.** In scikit-learn they are passed as arguments to the constructor of the estimator classes. Typical examples include C, kernel and gamma for Support Vector Classifier, ect.

**The Exhaustive Grid Search** provided by [GridSearchCV](#) exhaustively generates candidates from a grid of parameter values specified with the param\_grid parameter. For instance, the following **param\_grid** (SVM):

```
param_grid = [  
    {'C': [1, 10, 100, 1000], 'kernel': ['linear']},  
    {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001], 'kernel': ['rbf']},  
]
```

specifies that two grids should be explored: one with a linear kernel and C values in [1, 10, 100, 1000], and the second one with an RBF kernel, and the cross-product of C values ranging in [1, 10, 100, 1000] and gamma values in [0.001, 0.0001].

# 5. Hyperparameter Tuning

## **Randomized Parameter Optimization:** RandomizedSearchCV

In contrast to GridSearchCV, not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions. The number of parameter settings that are tried is given by **n\_iter**.

## **Parallelism:**

**GridSearchCV** and **RandomizedSearchCV** evaluate each parameter setting independently. Computations can be run in parallel if your OS supports it, by using the keyword **n\_jobs=-1**.

## 5. Hyperparameter Tuning

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
from sklearn.svm import SVC

dat = datasets.load_breast_cancer()
print("Examples = ",dat.data.shape , " Labels = ",
      dat.target.shape)
X_train, X_test, Y_train, Y_test = train_test_split(dat.data,
                                                    dat.target, test_size= 0.20, random_state=100)

param_grid = [
    {'C': [1, 10, 100, 1000], 'kernel': ['linear']},
    {'C': [2, 20, 200, 2000], 'gamma': [0.001, 0.0001], 'kernel': ['rbf']},
]
grid = GridSearchCV(SVC(), param_grid, refit = True, verbose =
3)
grid.fit(X_train, Y_train)
print('The best parameter after tuning :',grid.best_params_)
print('our model looks after hyper-parameter
tuning',grid.best_estimator_)
grid_predictions = grid.predict(X_test)
print(classification_report(Y_test, grid_predictions))
```

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import expon

param_rdsearch = {
    'C': expon(scale=100), 'gamma': expon(scale=.1),
    'kernel': ['rbf'], 'class_weight':['balanced', None]
}
clf_rds = RandomizedSearchCV(SVC(), param_rdsearch,
n_iter=100)

clf_rds.fit(X_train, Y_train)

print("Best: %f using %s" % (clf_rds.best_score_,
                             clf_rds.best_params_))
rds_predictions = clf_rds.predict(X_test)
print(classification_report(Y_test, rds_predictions))
```



# 6. Pipeline: chaining estimators

**Pipeline** can be used to **chain multiple estimators into one**. This is useful as there is often a fixed sequence of steps in processing the data, for example feature selection, normalization and classification. Pipeline serves multiple purposes here:

**Convenience and encapsulation:**

You only have to call fit and predict once on your data to fit a whole sequence of estimators.

**Joint parameter selection:**

You can grid search over parameters of all estimators in the pipeline at once.

**Safety:**

Pipelines help avoid leaking statistics from your test data into the trained model in cross-validation, by ensuring that the same samples are used to train the transformers and predictors.

# 6. Pipeline: chaining estimators

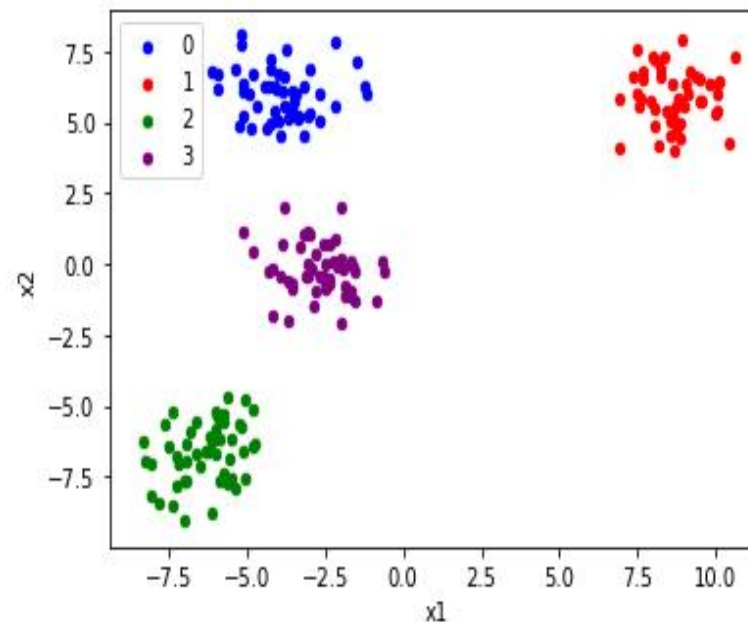
```
from sklearn.datasets import load_breast_cancer
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.svm import SVC
from sklearn.decomposition import PCA
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn import metrics
dat = load_breast_cancer()
X = dat.data
Y = dat.target
print("Examples = ",X.shape , " Labels = ", Y.shape)
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size = 0.3, random_state=100)
model_pipeline = Pipeline(steps=[
    ("feature_union", FeatureUnion([
        ('missing_values',SimpleImputer(missing_values=np.nan, strategy='mean')),
        ('scale', StandardScaler()),
        ("reduce_dim", PCA(n_components=10)),
    ])),
    ('clf', SVC(kernel='rbf', gamma= 0.001, C=5))
])

model_pipeline.fit(X_train, y_train)
predictions = model_pipeline.predict(X_test)
print(" Accuracy :",metrics.accuracy_score(y_test, predictions))
```

# Unsupervised learning

# 7. Clustering

**Clustering** is the most popular technique in unsupervised learning where **data is grouped based on the similarity of the data-points**. Clustering has many real-life applications where it can be used in a variety of situations.



# 7. Clustering

Method name	Parameters	Scalability	Usecase	Geometry (metric used)
K-Means	number of clusters	Very large <code>n_samples</code> , medium <code>n_clusters</code> with <a href="#">MiniBatch code</a>	General-purpose, even cluster size, flat geometry, not too many clusters	Distances between points
Affinity propagation	damping, sample preference	Not scalable with <code>n_samples</code>	Many clusters, uneven cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Mean-shift	bandwidth	Not scalable with <code>n_samples</code>	Many clusters, uneven cluster size, non-flat geometry	Distances between points
Spectral clustering	number of clusters	Medium <code>n_samples</code> , small <code>n_clusters</code>	Few clusters, even cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Ward hierarchical clustering	number of clusters or distance threshold	Large <code>n_samples</code> and <code>n_clusters</code>	Many clusters, possibly connectivity constraints	Distances between points
Agglomerative clustering	number of clusters or distance threshold, linkage type, distance	Large <code>n_samples</code> and <code>n_clusters</code>	Many clusters, possibly connectivity constraints, non Euclidean distances	Any pairwise distance
DBSCAN	neighborhood size	Very large <code>n_samples</code> , medium <code>n_clusters</code>	Non-flat geometry, uneven cluster sizes	Distances between nearest points
OPTICS	minimum cluster membership	Very large <code>n_samples</code> , large <code>n_clusters</code>	Non-flat geometry, uneven cluster sizes, variable cluster density	Distances between points
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation	Mahalanobis distances to centers
Birch	branching factor, threshold, optional global clusterer.	Large <code>n_clusters</code> and <code>n_samples</code>	Large dataset, outlier removal, data reduction.	Euclidean distance between points

# 7. Clustering - K-means

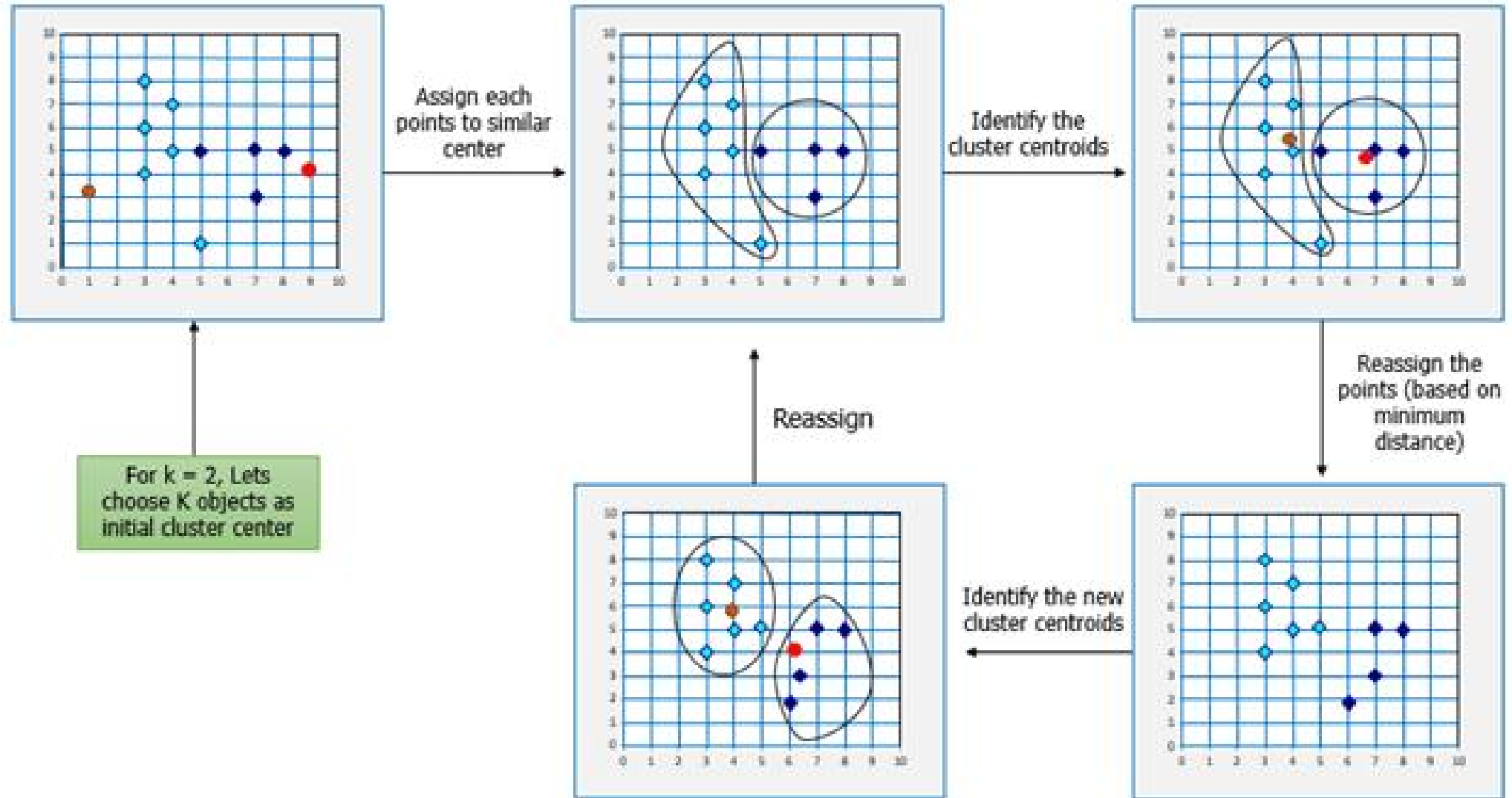
**The KMeans algorithm** clusters data by trying to separate samples in **n groups** of equal variance, minimizing a criterion known as the **inertia or within-cluster sum-of-squares**. This algorithm **requires the number of clusters to be specified**. It scales well to large number of samples and has been used across a large range of application areas in many different fields.

The k-means algorithm divides a set of **N** samples **X** into **K** disjoint clusters **C**, each described by the mean  **$\mu_j$**  of the samples in the cluster. The means are commonly called the cluster **“centroids”**; note that they are not, in general, points from **X**, although they live in the same space.

The K-means algorithm aims to choose centroids that minimise **the inertia, or within-cluster sum-of-squares criterion**:

$$\sum_{i=0}^n \min_{\mu_j \in C} (||x_i - \mu_j||^2)$$

# 7. Clustering - K-means





# 7. Clustering - K-means

## Clustering performance evaluation :

Clustering	
'adjusted_mutual_info_score'	<code>metrics.adjusted_mutual_info_score</code>
'adjusted_rand_score'	<code>metrics.adjusted_rand_score</code>
'completeness_score'	<code>metrics.completeness_score</code>
'fowlkes_mallows_score'	<code>metrics.fowlkes_mallows_score</code>
'homogeneity_score'	<code>metrics.homogeneity_score</code>
'mutual_info_score'	<code>metrics.mutual_info_score</code>
'normalized_mutual_info_score'	<code>metrics.normalized_mutual_info_score</code>
'v_measure_score'	<code>metrics.v_measure_score</code>



## 7. Clustering - K-means

### Clustering performance evaluation :

```
from sklearn import metrics
```

```
labels_true = [0, 0, 0, 1, 1, 1]
```

```
labels_pred = [0, 0, 1, 1, 2, 2]
```

```
metrics.adjusted_rand_score(labels_true, labels_pred)
```

### Centroid initialization methods :

```
good_init = np.array([-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2])
```

```
kmeans = KMeans(n_clusters=5, init=good_init)
```

# 7. Clustering - K-means

```
import matplotlib.pyplot as plt
import seaborn as sns; sns.set() # for plot styling
import numpy as np
from sklearn.datasets.samples_generator import make_blobs

# n_featuresint, optional (default=2)
X, y_true = make_blobs(n_samples=300, centers=4, cluster_std=1.5, random_state=100)
print("Examples = ", X.shape)
# Visualize the Data
fig, ax = plt.subplots(figsize=(8,5))
ax.scatter(X[:, 0], X[:, 1], s=50)
ax.set_title("Visualize the Data")
ax.set_xlabel("X")
ax.set_ylabel("Y")
plt.show()

# Create Clusters
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=4)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)
print("the 4 centroids that the algorithm found:
\n", kmeans.cluster_centers_)

# Visualize the results
fig, ax = plt.subplots(figsize=(8,5))
ax.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')
centers = kmeans.cluster_centers_
ax.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5);
ax.set_title("Visualize the results")
ax.set_xlabel("X")
ax.set_ylabel("Y")
plt.show()
```

```
# Assign new instances to the cluster whose centroid is closest:
X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
y_kmeans_new = kmeans.predict(X_new)
# The transform() method measures the distance from each instance
to every centroid:
print("The distance from each instance to every centroid:
\n", kmeans.transform(X_new))
# Visualize the new results
fig, ax = plt.subplots(figsize=(8,5))
ax.scatter(X_new[:, 0], X_new[:, 1], c=y_kmeans_new, s=50,
cmap='viridis')
centers = kmeans.cluster_centers_
ax.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5);
ax.set_title("Visualize the new results")
ax.set_xlabel("X")
ax.set_ylabel("Y")
plt.show()

print("inertia : ", kmeans.inertia_)
print("score : ", kmeans.score(X))
```

## 7. Clustering - Density-based spatial clustering of applications with noise (DBSCAN)

The **DBSCAN** algorithm should be used to find **associations** and **structures** in data that are hard to find manually but that can be relevant and useful to find **patterns** and **predict trends**. It defines clusters as continuous regions of high density.

The **DBSCAN** algorithm basically **requires 2 parameters**:

**eps**: specifies how close points should be to each other to be considered a part of a cluster. It means that **if the distance between two points is lower or equal to this value (eps), these points are considered neighbors**.

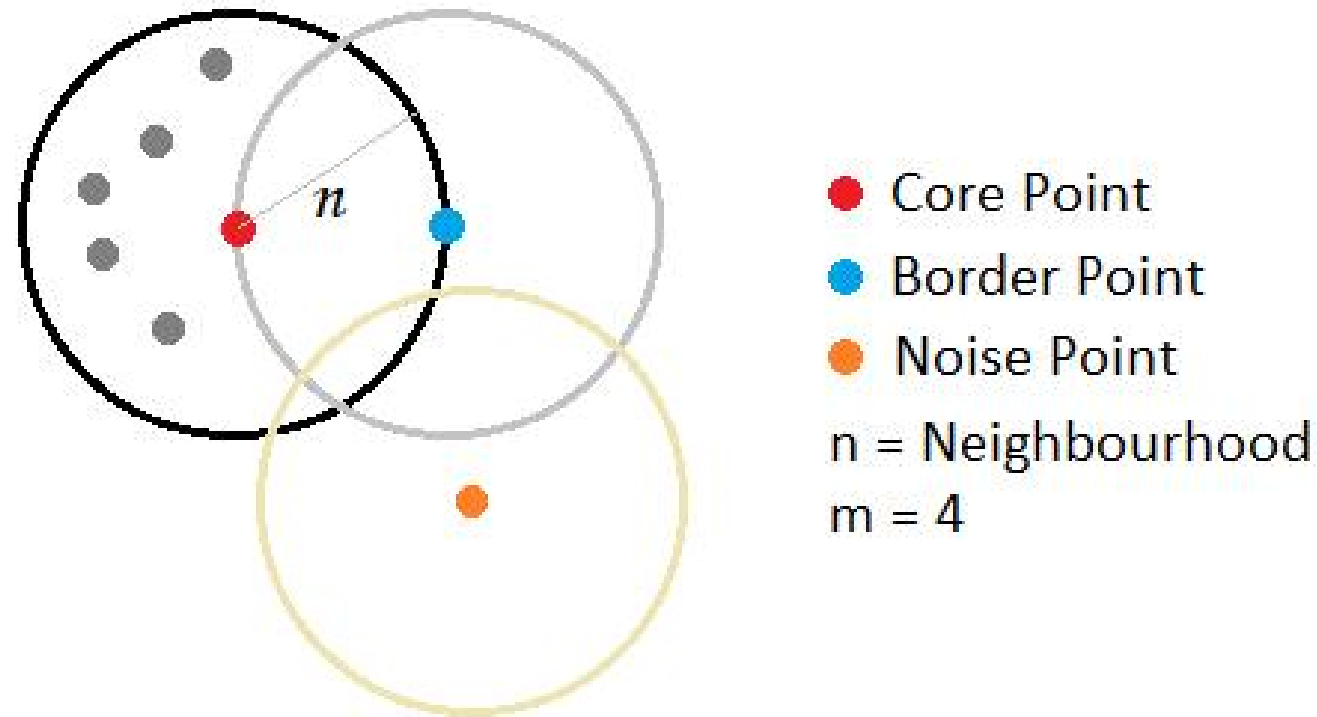
**min\_samples**: **the minimum number of points to form a dense region**. For example, if we set the min\_samples parameter as 5, then we need at least 5 points to form a dense region,

## 7. Clustering - Density-based spatial clustering of applications with noise (DBSCAN)

Here is how it works:

- 1) For each instance, the algorithm counts how many instances are located within a small distance  $\epsilon$  (epsilon) from it. This region is called the instance's  $\epsilon$ -neighborhood.
- 2) If an instance has at least `min_samples` instances in its  $\epsilon$ -neighborhood (including itself), then it is considered a **core instance**. In other words, **core instances** are those that are located in dense regions.
- 3) All instances in the neighborhood of a **core instance** belong to the same **cluster**. This neighborhood may include other core instances; therefore, a long sequence of neighboring core instances forms a single cluster.
- 4) Any instance that is not a core instance and does not have one in its neighborhood is considered an **anomaly**.

## 7. Clustering - Density-based spatial clustering of applications with noise (DBSCAN)

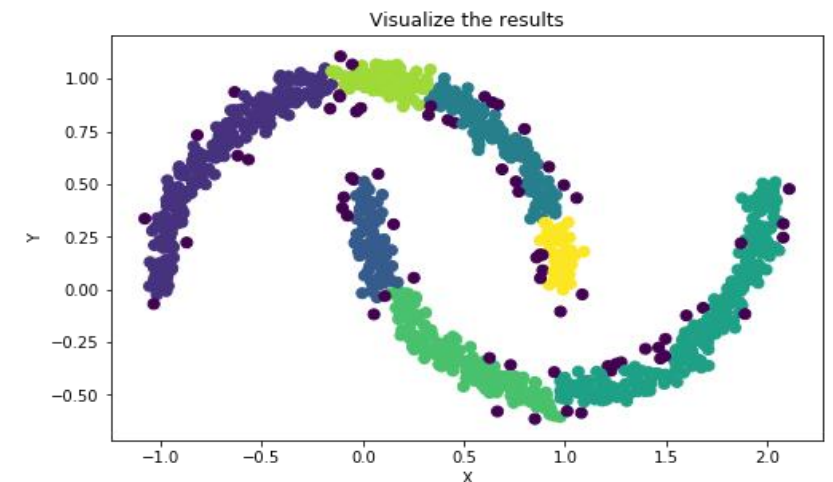


DBSCAN CLUSTERING

*Abhijit Annaldas*

## 7. Clustering - Density-based spatial clustering of applications with noise (DBSCAN)

```
from sklearn.cluster import DBSCAN
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=1000, noise=0.05)
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
# The labels of all the instances. Notice that some instances have a cluster index equal to -1, which means that
# they are considered as anomalies by the algorithm.
print("The labels of all the instances : \n",dbscan.labels_)
# The indices of the core instances
print("The indices of the core instances : \n Len = ",len(dbscan.core_sample_indices_), "\n",
dbscan.core_sample_indices_)
# The core instances
print("the core instances : \n", dbscan.components_)
# Visualize the results
fig, ax = plt.subplots(figsize=(8,5))
ax.scatter(X[:, 0], X[:, 1], c= dbscan.labels_, s=50, cmap='viridis')
ax.set_title("Visualize the results")
ax.set_xlabel("X")
ax.set_ylabel("Y")
plt.show()
```



## 7. Clustering - Hierarchical clustering

**Hierarchical clustering** is a general family of clustering algorithms that build **nested clusters** by **merging** or **splitting** them successively. This hierarchy of clusters is represented as a **tree** (or **dendrogram**). **The root of the tree is the unique cluster** that gathers all the samples, the leaves being the clusters with only one sample.

The **AgglomerativeClustering** object performs a hierarchical clustering using a bottom up approach: each observation starts in its own cluster, and clusters are successively merged together.

# 7. Clustering - Hierarchical clustering

Steps to Perform Hierarchical Clustering (agglomerative clustering):

- 1) **At the start, treat each data point as one cluster. Therefore, the number of clusters at the start will be  $K$ , while  $K$  is an integer representing the number of data points.**
- 2) **Form a cluster by joining the two closest data points resulting in  $K-1$  clusters.**
- 3) **Form more clusters by joining the two closest clusters resulting in  $K-2$  clusters.**
- 4) **Repeat the above three steps until one big cluster is formed.**
- 5) **Once single cluster is formed, dendrograms are used to divide into multiple clusters depending upon the problem.**

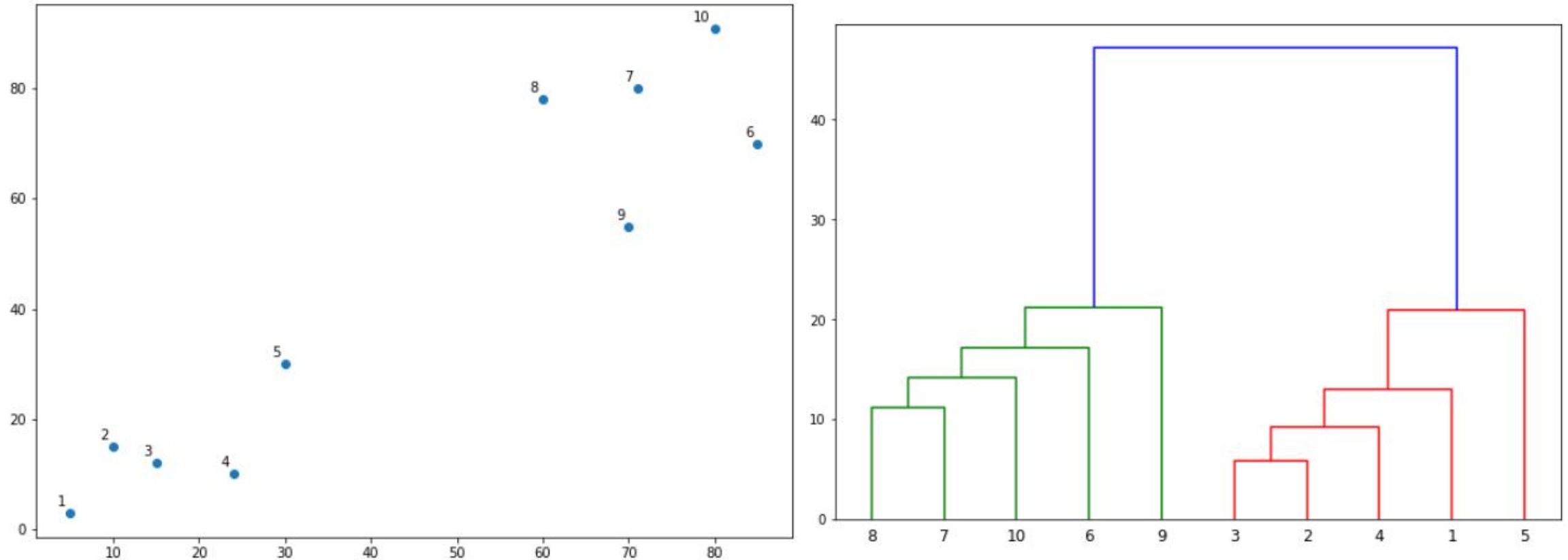


# 7. Clustering - Hierarchical clustering

There are different ways to find distance between the clusters. The distance itself can be **Euclidean** or **Manhattan** distance. Following are some of the options to measure distance between two clusters:

- 1) Measure the distance between the closes points of two clusters.
- 2) Measure the distance between the farthest points of two clusters.
- 3) Measure the distance between the centroids of two clusters.
- 4) Measure the distance between all possible combination of points between the two clusters and take the mean.

# 7. Clustering - Hierarchical clustering



# 7. Clustering - Hierarchical clustering

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.cluster.hierarchy import dendrogram
from sklearn.cluster import AgglomerativeClustering
```

```
def plot_dendrogram(model, **kwargs):
    # Create linkage matrix and then plot the dendrogram
    # create the counts of samples under each node
    counts = np.zeros(model.children_.shape[0])
    n_samples = len(model.labels_)
    for i, merge in enumerate(model.children_):
        current_count = 0
        for child_idx in merge:
            if child_idx < n_samples:
                current_count += 1 # leaf node
            else:
                current_count += counts[child_idx - n_samples]
        counts[i] = current_count
    linkage_matrix = np.column_stack([model.children_,
                                     counts]).astype(float)

    # Plot the corresponding dendrogram
    dendrogram(linkage_matrix, **kwargs)
```

```
X = np.array([[5,3],[10,15],[15,12],[24,10],[30,30],
               [85,70],[71,80],[60,78],[70,55],[80,91],])
```

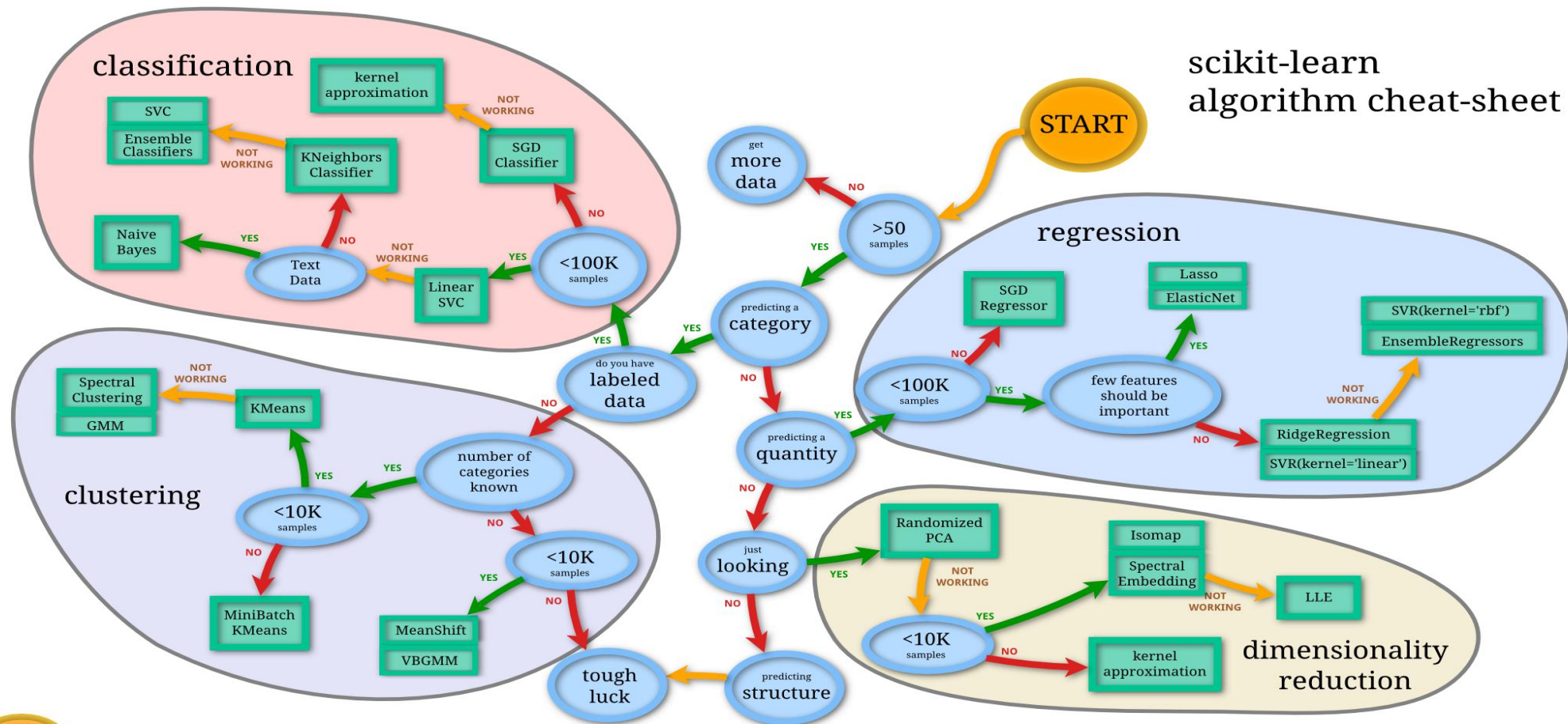
```
model = AgglomerativeClustering(distance_threshold=0,
n_clusters=None)
```

```
model.fit(X)
```

```
plt.title("Hierarchical Clustering Dendrogram")
# plot the top three levels of the dendrogram
plot_dendrogram(model, truncate_mode="level", p=3)
plt.xlabel("Number of points in node (or index of point if no
parenthesis).")
plt.show()
```

```
"""
import scipy.cluster.hierarchy as shc
plt.figure(figsize=(10, 7))
plt.title("Customer Dendograms")
dend = shc.dendrogram(shc.linkage(X, method='ward'))
"""
```

## 8. Choosing the right estimator



## 9. Save and Load Machine Learning Models

# Save the model

```
from pickle import dump  
dump(model, open("model.pkl", "wb"))
```

# Load the model

```
from pickle import load  
my_model = load(open("model.pkl", "rb"))
```

# Use the loaded model to make predictions

```
pred = my_model.predict(X_test)
```

*Thank you for  
your attention*

*Hichem Felouat ...*