# Learning Objective

Familiarize with the working of learning algorithms like Learning by Induction and Reinforcement Learning.

# a)Supervised Learning

**Tool**: Python

**Libraries Used**: numpy, matplotlib, pandas, sklearn

**Sample Problem**: Given a dataset with customer’s Gender, Age, Estimated Salary and whether the customer purchased a particular product or not, build a Decision Tree Classifier to predict if a customer will buy a particular product given his Gender, Age and Estimated salary.

**Input**: [Social\_Netwok\_ads.csv](https://drive.google.com/open?id=1mZ_VQTYbSHi0Ym9CORdXnIDWZPK-DTHJ)

Please click on the link to view the dataset.

**Classifier**: Decision Tree Classifier

**Implementation**

**# Importing the libraries**

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

**# Importing the dataset**

dataset = pd.read\_csv('../Dataset/Social\_Network\_Ads.csv')

X = dataset.iloc[:, [2, 3]].values

y = dataset.iloc[:, 4].values

**# Splitting the dataset into the Training set and Test set**

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.25, random\_state = 0) # Feature Scaling

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

**# Fitting Decision Tree Classification to the Training set**

from sklearn.tree import DecisionTreeClassifier

classifier = DecisionTreeClassifier(criterion = 'entropy', random\_state = 0)

classifier.fit(X\_train, y\_train)

**# Predicting the Test set results**

y\_pred = classifier.predict(X\_test)

**# Making the Confusion Matrix**

from sklearn.metrics import confusion\_matrix

cm = confusion\_matrix(y\_test, y\_pred)

cm

**# Visualising the Test set results**

from matplotlib.colors import ListedColormap

X\_set, y\_set = X\_test, y\_test

X1, X2 = np.meshgrid(np.arange(start = X\_set[:, 0].min() - 1, stop = X\_set[:, 0].max() + 1, step = 0.01),

np.arange(start = X\_set[:, 1].min() - 1, stop = X\_set[:, 1].max() + 1, step = 0.01))

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),

alpha = 0.75, cmap = ListedColormap(('red', 'green')))

plt.xlim(X1.min(), X1.max())

plt.ylim(X2.min(), X2.max())

for i, j in enumerate(np.unique(y\_set)):

plt.scatter(X\_set[y\_set == j, 0], X\_set[y\_set == j, 1],

c = ListedColormap(('red', 'green'))(i), label = j)

plt.title('Decision Tree Classification (Test set)')

plt.xlabel('Age')

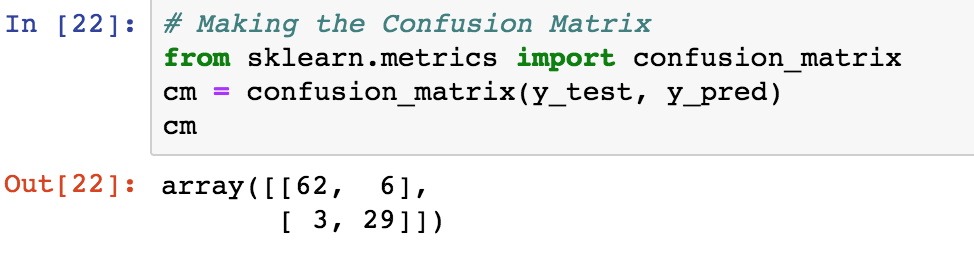
plt.ylabel('Estimated Salary')

plt.legend()

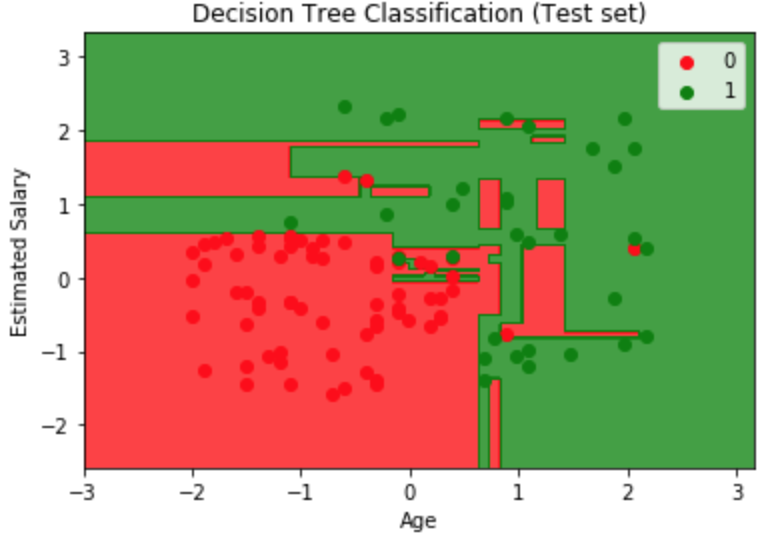
plt.show()

**Output**:

Confusion Matrix



**Screenshot**

****

## b) Reinforcement Learning

### **Tool**: Python

**Libraries Used**: numpy, gym, keras, rl

**Sample Problem**: Remember when we were kids, wewould pick a stick and try to balance it on one hand. Me and my friends used to have this competition where whoever balances it for more time would get a “reward”, a chocolate!

We would implement the same problem for a cartpole

### **Algorithm used:** Deep Q-Learning

**Implementation:**

import numpy as np

import gym

from keras.models import Sequential

from keras.layers import Dense, Activation, Flatten

from keras.optimizers import Adam

from rl.agents.dqn import DQNAgent

from rl.policy import EpsGreedyQPolicy

from rl.memory import SequentialMemory

# We first set the relevant variables

ENV\_NAME = 'CartPole-v0'

**# Get the environment and extract the number of actions available in the Cartpole problem**

env = gym.make(ENV\_NAME)

np.random.seed(123)

env.seed(123)

nb\_actions = env.action\_space.n

if \_\_name\_\_ == "\_\_main\_\_":

**#We build a very simple single hidden layer neural network model.**

model = Sequential()

model.add(Flatten(input\_shape=(1,) + env.observation\_space.shape))

model.add(Dense(16))

model.add(Activation('relu'))

model.add(Dense(nb\_actions))

model.add(Activation('linear'))

print(model.summary())

**# We configure and compile our agent. We set our policy as Epsilon Greedy and we also set our memory as Sequential Memory because we want to store the result of actions we performed and the rewards we get for each action**.

policy = EpsGreedyQPolicy()

memory = SequentialMemory(limit=50000, window\_length=1)

dqn = DQNAgent(model=model, nb\_actions=nb\_actions, memory=memory, nb\_steps\_warmup=10,

target\_model\_update=1e-2, policy=policy)

dqn.compile(Adam(lr=1e-3), metrics=['mae'])

**# Okay, now it's time to learn something! We visualize the training here for show, but this slows down training quite a lot.**

dqn.fit(env, nb\_steps=5000, visualize=True, verbose=2)

**#Now we test our reinforcement learning model**

dqn.test(env, nb\_episodes=5, visualize=True)

**Output**: Cartpole trying to balance itself

### **Screenshot**

