**=====================1 – Simple==========================**

**1(A) - Rules and Facts – weather**

weather(pheonix, summer, hot).

weather(la, summer, warm).

weather(Pheonix, winter, warm).

**Prolog:**

weather(pheonix, summer, hot).

Weather(city, \_\_\_, warm).

**1(B) - Relationship -Weather**

weather(pheonix, hot, summer).

weather(la, warm, summer).

warmer\_than(C1, C2).

weather(C1, hot, summer).

weather(C2, warm, summer).

**Prolog:**

write(C1), write( ' is warme than '), write(C2).

weather(C1, hot,summer), weather(C2, warm, summer), write(C1), write( ' is warme than '), write(C2).

**1(C) - Relationship with User defined**

%section A

result(rahim, 3.6).

result(ajay, 3.7).

result(priya, 2.8).

result(rahul, 3.9).

result(kim, 3.10).

%section B

result(sam, 4.0).

result(vickey, 3.9).

result(priyanka, 3.8).

result(ram, 3.6).

result(kunal, 3.0).

getresult :-

write("enter section A student name : "),nl,

read(X),nl,

result(X,Y),nl,

write("enter section A student result : "),nl,

write(Y),nl,

write("enter section B student name : "),nl,

read(P),nl,

result(P,Q),nl,

write("enter section B student result : "),nl,

write(Q),nl,

compare(Y,Q).

compare(Y,Q):-

Y>Q, nl,

write("section A student is best ");

Y<Q, nl,

write("section B student is best ");

Y=:=Q, nl,

write("all student are same").

**Prolog:**

getresult().

priyanka.

ram.

getresult().

ajay.

adi.

**======================2 – Water Jug======================**

start(2,0):-write(' 4lit Jug: 2 | 3lit Jug: 0|\n'),

write('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n'),

write('Goal Reached! Congrats!!\n'),

write('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n').

start(X,Y):-write(' 4lit Jug: '),write(X),write('| 3lit Jug: '),

write(Y),write('|\n'),

write(' Enter the move::'),

read(N),

contains(X,Y,N).

contains(\_,Y,1):-start(4,Y).

contains(X,\_,2):-start(X,3).

contains(\_,Y,3):-start(0,Y).

contains(X,\_,4):-start(X,0).

contains(X,Y,5):-N is Y-4+X, start(4,N).

contains(X,Y,6):-N is X-3+Y, start(N,3).

contains(X,Y,7):-N is X+Y, start(N,0).

contains(X,Y,8):-N is X+Y, start(0,N).

main():-write(' Water Jug Game \n'),

write('Intial State: 4lit Jug- 0lit\n'),

write(' 3lit Jug- 0lit\n'),

write('Final State: 4lit Jug- 2lit\n'),

write(' 3lit Jug- 0lit\n'),

write('Follow the Rules: \n'),

write('Rule 1: Fill 4lit Jug\n'),

write('Rule 2: Fill 3lit Jug\n'),

write('Rule 3: Empty 4lit Jug\n'),

write('Rule 4: Empty 3lit Jug\n'),

write('Rule 5: Pour water from 3lit Jug to fill 4lit Jug\n'),

write('Rule 6: Pour water from 4lit Jug to fill 3lit Jug\n'),

write('Rule 7: Pour all of water from 3lit Jug to 4lit Jug\n'),

write('Rule 8: Pour all of water from 4lit Jug to 3lit Jug\n'),

write(' 4lit Jug: 0 | 3lit Jug: 0'),nl,

write(' Enter the move::'),

read(N),nl,

contains(0,0,N).

**PROLOG**

main().

**=====================3 – Tic tac toe Using BFS========================**

play :- my\_turn([]).

my\_turn(Game) :-

valid\_moves(ValidMoves, Game, x),

any\_valid\_moves(ValidMoves, Game).

any\_valid\_moves([], \_) :-

write('It is a tie'), nl.

any\_valid\_moves([\_|\_], Game) :-

findall(NextMove, game\_analysis(x, Game, NextMove), MyMoves),

do\_a\_decision(MyMoves, Game).

% This can only fail in the beginning.

do\_a\_decision(MyMoves, Game) :-

not(MyMoves = []),

length(MyMoves, MaxMove),

random(0, MaxMove, ChosenMove),

nth0(ChosenMove, MyMoves, X),

NextGame = [X | Game],

print\_game(NextGame),

(victory\_condition(x, NextGame) ->

(write('I won. You lose.'), nl);

your\_turn(NextGame), !).

your\_turn(Game) :-

valid\_moves(ValidMoves, Game, o),

(ValidMoves = [] -> (write('It is a tie'), nl);

(write('Available moves:'), write(ValidMoves), nl,

ask\_move(Y, ValidMoves),

NextGame = [Y | Game],

(victory\_condition(o, NextGame) ->

(write('I lose. You win.'), nl);

my\_turn(NextGame), !))).

ask\_move(Move, ValidMoves) :-

write('Give your move:'), nl,

read(Move), member(Move, ValidMoves), !.

ask\_move(Y, ValidMoves) :-

write('not a move'), nl,

ask\_move(Y, ValidMoves).

movement\_prompt(X, Y, ValidMoves) :-

write('Give your X:'), nl, read(X), member(move(o, X, Y), ValidMoves), !,

write('Give your Y:'), nl, read(Y), member(move(o, X, Y), ValidMoves).

% A routine for printing games.. Well you can use it.

print\_game(Game) :-

plot\_row(0, Game), plot\_row(1, Game), plot\_row(2, Game).

plot\_row(Y, Game) :-

plot(Game, 0, Y), plot(Game, 1, Y), plot(Game, 2, Y), nl.

plot(Game, X, Y) :-

(member(move(P, X, Y), Game), ground(P)) -> write(P) ; write('.').

% This system determines whether there's a perfect play available.

game\_analysis(\_, Game, \_) :-

victory\_condition(Winner, Game),

Winner = x. % We do not want to lose.

% Winner = o. % We do not want to win. (egostroking mode).

% true. % If you remove this constraint entirely, it may let you win.

game\_analysis(Turn, Game, NextMove) :-

not(victory\_condition(\_, Game)),

game\_analysis\_continue(Turn, Game, NextMove).

game\_analysis\_continue(Turn, Game, NextMove) :-

valid\_moves(Moves, Game, Turn),

game\_analysis\_search(Moves, Turn, Game, NextMove).

% Comment these away and the system refuses to play,

% because there are no ways to play this without a possibility of tie.

game\_analysis\_search([], o, \_, \_). % Tie on opponent's turn.

game\_analysis\_search([], x, \_, \_). % Tie on our turn.

game\_analysis\_search([X|Z], o, Game, NextMove) :- % Whatever opponent does,

NextGame = [X | Game], % we desire not to lose.

game\_analysis\_search(Z, o, Game, NextMove),

game\_analysis(x, NextGame, \_), !.

game\_analysis\_search(Moves, x, Game, NextMove) :-

game\_analysis\_search\_x(Moves, Game, NextMove).

game\_analysis\_search\_x([X|\_], Game, X) :-

NextGame = [X | Game],

game\_analysis(o, NextGame, \_).

game\_analysis\_search\_x([\_|Z], Game, NextMove) :-

game\_analysis\_search\_x(Z, Game, NextMove).

% This thing describes all kinds of valid games.

valid\_game(Turn, Game, LastGame, Result) :-

victory\_condition(Winner, Game) ->

(Game = LastGame, Result = win(Winner)) ;

valid\_continuing\_game(Turn, Game, LastGame, Result).

valid\_continuing\_game(Turn, Game, LastGame, Result) :-

valid\_moves(Moves, Game, Turn),

tie\_or\_next\_game(Moves, Turn, Game, LastGame, Result).

tie\_or\_next\_game([], \_, Game, Game, tie).

tie\_or\_next\_game(Moves, Turn, Game, LastGame, Result) :-

valid\_gameplay\_move(Moves, NextGame, Game),

opponent(Turn, NextTurn),

valid\_game(NextTurn, NextGame, LastGame, Result).

% Victory conditions for tic tac toe.

victory(P, Game, Begin) :-

valid\_gameplay(Game, Begin),

victory\_condition(P, Game).

victory\_condition(P, Game) :-

(X = 0; X = 1; X = 2),

member(move(P, X, 0), Game),

member(move(P, X, 1), Game),

member(move(P, X, 2), Game).

victory\_condition(P, Game) :-

(Y = 0; Y = 1; Y = 2),

member(move(P, 0, Y), Game),

member(move(P, 1, Y), Game),

member(move(P, 2, Y), Game).

victory\_condition(P, Game) :-

member(move(P, 0, 2), Game),

member(move(P, 1, 1), Game),

member(move(P, 2, 0), Game).

victory\_condition(P, Game) :-

member(move(P, 0, 0), Game),

member(move(P, 1, 1), Game),

member(move(P, 2, 2), Game).

% This describes a valid form of gameplay.

% Which player did the move is disregarded.

valid\_gameplay(Start, Start).

valid\_gameplay(Game, Start) :-

valid\_gameplay(PreviousGame, Start),

valid\_moves(Moves, PreviousGame, \_),

valid\_gameplay\_move(Moves, Game, PreviousGame).

valid\_gameplay\_move([X|\_], [X|PreviousGame], PreviousGame).

valid\_gameplay\_move([\_|Z], Game, PreviousGame) :-

valid\_gameplay\_move(Z, Game, PreviousGame).

% The set of valid moves must not be affected by the decision making

% of the prolog interpreter.

% Therefore we have to retrieve them like this.

% This is equivalent to the (∀x∈0..2)(∀y∈0..2)(....

% uh wait.. There's no way to represent this using those quantifiers.

valid\_moves(Moves, Game, Turn) :-

valid\_moves\_column(0, M1, [], Game, Turn),

valid\_moves\_column(1, M2, M1, Game, Turn),

valid\_moves\_column(2, Moves, M2, Game, Turn).

valid\_moves\_column(X, M3, M0, Game, Turn) :-

valid\_moves\_cell(X, 0, M1, M0, Game, Turn),

valid\_moves\_cell(X, 1, M2, M1, Game, Turn),

valid\_moves\_cell(X, 2, M3, M2, Game, Turn).

valid\_moves\_cell(X, Y, M1, M0, Game, Turn) :-

member(move(\_, X, Y), Game) -> M0 = M1 ; M1 = [move(Turn,X,Y) | M0].

% valid\_move(X, Y, Game) :-

% (X = 0; X = 1; X = 2),

% (Y = 0; Y = 1; Y = 2),

% not(member(move(\_, X, Y), Game)).

opponent(x, o).

opponent(o, x).

**PROLOg:**

**play().**

**move(o,1,2).**

**==================4 - 8 puzzele using hill climbing========================**

initial([1,2,3,

0,4,5,

6,7,8]).

goal([1,2,3,

4,0,5,

6,7,8]).

move([X1,0,X3, X4,X5,X6, X7,X8,X9],

[0,X1,X3, X4,X5,X6, X7,X8,X9]).

move([X1,X2,0, X4,X5,X6, X7,X8,X9],

[X1,0,X2, X4,X5,X6, X7,X8,X9]).

%% move left in the middle row

move([X1,X2,X3, X4,0,X6,X7,X8,X9],

[X1,X2,X3, 0,X4,X6,X7,X8,X9]).

move([X1,X2,X3, X4,X5,0,X7,X8,X9],

[X1,X2,X3, X4,0,X5,X7,X8,X9]).

%% move left in the bottom row

move([X1,X2,X3, X4,X5,X6, X7,0,X9],

[X1,X2,X3, X4,X5,X6, 0,X7,X9]).

move([X1,X2,X3, X4,X5,X6, X7,X8,0],

[X1,X2,X3, X4,X5,X6, X7,0,X8]).

%% move right in the top row

move([0,X2,X3, X4,X5,X6, X7,X8,X9],

[X2,0,X3, X4,X5,X6, X7,X8,X9]).

move([X1,0,X3, X4,X5,X6, X7,X8,X9],

[X1,X3,0, X4,X5,X6, X7,X8,X9]).

%% move right in the middle row

move([X1,X2,X3, 0,X5,X6, X7,X8,X9],

[X1,X2,X3, X5,0,X6, X7,X8,X9]).

move([X1,X2,X3, X4,0,X6, X7,X8,X9],

[X1,X2,X3, X4,X6,0, X7,X8,X9]).

%% move right in the bottom row

move([X1,X2,X3, X4,X5,X6,0,X8,X9],

[X1,X2,X3, X4,X5,X6,X8,0,X9]).

move([X1,X2,X3, X4,X5,X6,X7,0,X9],

[X1,X2,X3, X4,X5,X6,X7,X9,0]).

%% move up from the middle row

move([X1,X2,X3, 0,X5,X6, X7,X8,X9],

[0,X2,X3, X1,X5,X6, X7,X8,X9]).

move([X1,X2,X3, X4,0,X6, X7,X8,X9],

[X1,0,X3, X4,X2,X6, X7,X8,X9]).

move([X1,X2,X3, X4,X5,0, X7,X8,X9],

[X1,X2,0, X4,X5,X3, X7,X8,X9]).

%% move up from the bottom row

move([X1,X2,X3, X4,X5,X6, X7,0,X9],

[X1,X2,X3, X4,0,X6, X7,X5,X9]).

move([X1,X2,X3, X4,X5,X6, X7,X8,0],

[X1,X2,X3, X4,X5,0, X7,X8,X6]).

move([X1,X2,X3, X4,X5,X6, 0,X8,X9],

[X1,X2,X3, 0,X5,X6, X4,X8,X9]).

%% move up from the top row

move([0,X2,X3, X4,X5,X6, X7,X8,X9],

[X4,X2,X3, 0,X5,X6, X7,X8,X9]).

move([X1,0,X3, X4,X5,X6, X7,X8,X9],

[X1,X5,X3, X4,0,X6, X7,X8,X9]).

move([X1,X2,0, X4,X5,X6, X7,X8,X9],

[X1,X2,X6, X4,X5,0, X7,X8,X9]).

%% move down from the middle row

move([X1,X2,X3, 0,X5,X6, X7,X8,X9],

[X1,X2,X3, X7,X5,X6, 0,X8,X9]).

move([X1,X2,X3, X4,0,X6, X7,X8,X9],

[X1,X2,X3, X4,X8,X6, X7,0,X9]).

move([X1,X2,X3, X4,X5,0, X7,X8,X9],

[X1,X2,X3, X4,X5,X9, X7,X8,0]).

puzzle(S, [S]) :- goal(S).

puzzle(S, [S|Rest]) :- move(S, S2), puzzle(S2, Rest).

**PROLOG**

**initial(S).**

**puzzle([1,2,3,0,4,5,6,7,8],Path).**

**goal(S).**

**========================5 – 1 – NumPy (JupterLab)========================**

Import numpy

import numpy as np

**# create an array**

digits = np.array([

[1, 2, 3],

[4, 5, 6],

[6, 7, 9],

])

digits

**# addition of two integers**

a=2

b=4

c=a+b

c

**# Study shape and axes of an array.**

temperatures = np.array([

29.3, 42.1, 18.8, 16.1, 38.0, 12.5,

12.6, 49.9, 38.6, 31.3, 9.2, 22.2

]).reshape(2, 2, 3)

In [3]: temperatures.shape

In [4]: temperatures

In [5]: np.swapaxes(temperatures, 1, 2)

table = np.array([

...: [5, 3, 7, 1],

...: [2, 6, 7 ,9],

...: [1, 1, 1, 1],

...: [4, 3, 2, 0],

...: ])

table.max()

table.max(axis=0)

table.max(axis=1)

**#Study of Broadcasting with an array.**

A= np.arange(32).reshape(4, 1, 8)

A

B = np.arange(48).reshape(1, 6, 8)

B

**# Addition of two Arrays.**

A+B

**#Find the Square of an array.**

square = np.array([

[16, 3, 2, 13],

[5, 10, 11, 8],

[9, 6, 7, 12],

[4, 15, 14, 1]

])

for i in range(4):

...: assert square[:, i].sum() == 34

...: assert square[i, :].sum() == 34

...:

assert square[:2, :2].sum() == 34

assert square[2:, :2].sum() == 34

assert square[:2, 2:].sum() == 34

assert square[2:, 2:].sum() == 34

**#Study of masking and filtering.**

numbers = np.linspace(5, 50, 24, dtype=int).reshape(4, -1)

numbers

mask = numbers % 4 == 0

mask

numbers[mask]

by\_four = numbers[numbers % 4 == 0]

by\_four

from numpy.random import default\_rng

rng = default\_rng()

values = rng.standard\_normal(10000)

values[:5]

std = values.std()

std

filtered = values[(values > -2 \* std) & (values < 2 \* std)]

filtered.size

values.size

filtered.size / values.size

a = np.array([

[1, 2],

[3, 4],

[5, 6],

])

**#Transposing, Sorting, and Concatenating of arrays.**

a.T

a.transpose()

data = np.array([

...: [7, 1, 4],

...: [8, 6, 5],

...: [1, 2, 3]

...: ])

np.sort(data)

np.sort(data, axis=None)

np.sort(data, axis=0)

a = np.array([

...: [4, 8],

...: [6, 1]

...: ])

b = np.array([

...: [3, 5],

...: [7, 2],

...: ])

np.hstack((a, b))

np.vstack((b, a))

np.concatenate((a, b))

np.concatenate((a, b), axis=None)

**#Implementation of Maclaurin Series.**

from math import e, factorial

fac = np.vectorize(factorial)

def e\_x(x, terms=10):

"""Approximates e^x using a given number of terms of

the Maclaurin series

"""

n = np.arange(terms)

return np.sum((x \*\* n) / fac(n))

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

print("Actual:", e \*\* 3) # Using e from the standard library

print("N (terms)\tMaclaurin\tError")

for n in range(1, 14):

maclaurin = e\_x(3, terms=n)

print(f"{n}\t\t{maclaurin:.03f}\t\t{e\*\*3 - maclaurin:.03f}")

**#Study of different Datatypes(numerical,String)**

a = np.array([1, 3, 5.5, 7.7, 9.2], dtype=np.single)

a

b = np.array([1, 3, 5.5, 7.7, 9.2], dtype=np.uint8)

b

names = np.array(["bob", "amy", "han"], dtype=str)

names

names.itemsize

names = np.array(["bob", "amy", "han"])

names

more\_names = np.array(["bobo", "jehosephat"])

np.concatenate((names, more\_names))

names[2] = "jasica"

names

**#Study of structured array**

data = np.array([

("joe", 32, 6),

("mary", 15, 20),

("felipe", 80, 100),

("beyonce", 38, 9001),

], dtype=[("name", str, 10), ("age", int), ("power", int)])

data[0]

data["name"]

data[data["power"] > 9000]["name"]

**#numpy version**

print("Numpy Version:",np.version.version)

**======================5 – 2 – Panda=============================**

**# import Pandas Print version**

import pandas as pd

print(pd.\_\_version\_\_)

**#create a dataframe**

data = {

'apples': [3, 2, 0, 1],

'oranges': [0, 3, 7, 2]

}

purchases = pd.DataFrame(data)

purchases

purchases = pd.DataFrame(data, index=['June', 'Robert', 'Lily', 'David'])

purchases

purchases.loc['June']

**#read csv file**

df = pd.read\_csv('f.csv')

df

**#read csv without index**

df = pd.read\_csv('f.csv', index\_col=0)

df

**#Create Dataframe:**

import pandas as pd

df = pd.DataFrame({'X':[78,85,96,80,86], 'Y':[84,94,89,83,86],'Z':[86,97,96,72,83]});

print(df)

**#create series**

s = pd.Series([2, 4, 6, 8, 10])

print(s)

**#load the data and make sure to change the path for your localdirectory**

data = pd.read\_csv('project\_data.csv')

**#first 5 rows**

data.head()

**#last 5 rows**

data.tail()

**#check the basic information of the data**

data.info()

**#extract the shape of the data**

data.shape

data['marital\_status'].unique()

**#count educational level**

round(data['educational\_level'].value\_counts(normalize=True),2)

**#missing values or duplicate values**

data.isnull()

data.duplicated().sum()

data['educational\_level'].isnull().sum()

**#specifying Education as a variable where we should look for the sum of missing values**

**#Select and filter data: loc and iloc**

subset\_data = data[['year\_of\_birth ', 'educational\_level', 'annual\_income']]

subset\_data

**#select a unique category of the education by specifying that only “Master” should be returned from the data frame.**

data[data["educational\_level"] == "Master"]

**#specify the rows and columns as labels**

data.loc[:6, ['educational\_level', 'recency']]

**#speciy rows and columns as integer based values**

data.iloc[:6, [2,6]]

**#choosing the customers with an income higher than 75,000 and with a master’s degree.**

data.iloc[list((data.annual\_income > 75000) & (data.educational\_level == 'Master')), :,]

**#Apply data operations: index, new variables, data types**

**#set the index as customer\_id**

data.set\_index("customer\_id")

**#sort the data by year\_of\_birth, ascending is default;**

data.sort\_values(by = ['year\_of\_birth '], ascending = True)

**# if we want it in descending we should set ascending = False**

**#create a new variable which is the sum of all purchases performed by customers**

data['sum\_purchases'] = data.online\_purchases + data.store\_purchases

data['sum\_purchases']

**#create an income category (low, meduim, high) based on the income variable**

income\_categories = ['Low', 'Meduim', 'High'] #set the categories

bins = [0,75000,120000,600000] #set the income boundaries

cats= pd.cut(data['annual\_income'],bins, labels=income\_categories) #apply the pd.cut method

data['Income\_Category'] = cats #assign the categories based on income

data[['annual\_income', 'Income\_Category']]

**#apply groupby to find the mean of income, recency, number of web and store purchases by educational group**

aggregate\_view = pd.DataFrame(data.groupby(by='educational\_level')[['annual\_income', 'recency', 'store\_purchases', 'online\_purchases']].mean()).reset\_index()

aggregate\_view

**#apply pivot table to find the aggregated sum of purchases and mean of recency per education and marital status group**

import numpy as np

pivot\_table = pd.DataFrame(pd.pivot\_table(data, values=['sum\_purchases', 'recency'], index=['marital\_status'],

columns=['educational\_level'], aggfunc={'recency': np.mean, 'sum\_purchases': np.sum}, fill\_value=0)).reset\_index()

pivot\_table

**========================5 -3 Scipy==================================**

**#Data Analysis with SciPy**

**# import numpy library**

import numpy as np

A = np.array([[1,2,3],[4,5,6],[7,8,8]])

**#Linear Algebra**

**#Determinant of a Matrix**

**# importing linalg function from scipy**

from scipy import linalg

**# Compute the determinant of a matrix**

linalg.det(A)

**#pivoted LU decomposition of a matrix**

P, L, U = linalg.lu(A)

print(P)

print(L)

print(U)

**# print LU decomposition**

print(np.dot(L,U))

**#Eigen values and eigen vectors of above matrix**

eigen\_values, eigen\_vectors = linalg.eig(A)

print(eigen\_values)

print(eigen\_vectors)

**#linear equations**

v = np.array([[2],[3],[5]])

print(v)

s = linalg.solve(A,v)

print(s)

**#Sparse Linear Algebra**

from scipy import sparse

**# Row-based linked list sparse matrix**

A = sparse.lil\_matrix((1000, 1000))

print(A)

A[0,:100] = np.random.rand(100)

A[1,100:200] = A[0,:100]

A.setdiag(np.random.rand(1000))

print(A)

**#Integration**

import scipy.integrate

f= lambda x:np.exp(-x\*\*2)

**# print results**

i = scipy.integrate.quad(f, 0, 1)

print(i)

**#Double Integrals**

from scipy import integrate

f = lambda y, x: x\*y\*\*2

i = integrate.dblquad(f, 0, 2, lambda x: 0, lambda x: 1)

**# print the results**

print(i)

**==================5 – 4 Matplotbit===============================**

**# importing matplotlib module**

from matplotlib import pyplot as plt

**# x-axis values**

x = [5, 2, 9, 4, 7]

**# Y-axis values**

y = [10, 5, 8, 4, 2]

**# Function to plot**

plt.plot(x, y)

**# function to show the plot**

plt.show()

**#Histogram**

from matplotlib import pyplot as plt

**# Y-axis values**

y = [10, 5, 8, 4, 2]

**# Function to plot histogram**

plt.hist(y)

**# Function to show the plot**

plt.show()

**#Scatter Plot**

**# x-axis values**

x = [5, 2, 9, 4, 7]

**# Y-axis values**

y = [10, 5, 8, 4, 2]

**# Function to plot scatter**

plt.scatter(x, y)

**# function to show the plot**

plt.show()

**#Adding title and Labeling the Axes in the graph**

**# x-axis values**

x = [5, 2, 9, 4, 7]

**# Y-axis values**

y = [10, 5, 8, 4, 2]

**# Function to plot**

plt.scatter(x, y)

**# Adding Title**

plt.title("GeeksFoeGeeks")

**# Labeling the axes**

plt.xlabel("Time (hr)")

plt.ylabel("Position (Km)")

**# function to show the plot**

plt.show()

**#Multiple Graphs**

x = [1, 2, 3, 4, 5]

y = [1, 4, 9, 16, 25]

plt.scatter(x, y)

**# function to show the plot**

plt.show()

plt.plot(x, y)

**# function to show the plot**

plt.show()

**=========================6 – 1 Linear Regression** **=======================**

import matplotlib

import matplotlib.pyplot as plt

import numpy as np

from sklearn import datasets, linear\_model

import pandas as pd

# Load Csv and Colums

df = pd.read\_csv("Housing.csv")

Y = df['price']

X = df['lotsize']

X = X.values.reshape(len(X), 1)

Y = Y.values.reshape(len(Y), 1)

# Split the targets into training/testing sets

X\_train = X[:-250]

X\_test = X[-250:]

# Split the targets into training/testing sets

Y\_train = Y[:-250]

Y\_test = Y[-250:]

#Plot outputs

plt.scatter(X\_test, Y\_test, color='black')

plt.title('Test Data')

plt.xlabel('Size')

plt.ylabel('Price')

plt.xticks(())

plt.yticks(())

plt.show()

**# Create Line Regression Obkject**

regr = linear\_model.LinearRegression()

#Train the model using the training sets

regr.fit(X\_train, Y\_train)

# Plot outputs

plt.plot(X\_test, regr.predict(X\_test), color='red', linewidth=3)

**=========================6 – 2 Linear Regression** **=======================**

import numpy as np

import pandas as pd

from sklearn import preprocessing

import matplotlib.pyplot as plt

plt.rc("font", size=14)

import seaborn as sns

sns.set(style="white") #white background style for seaborn plots

sns.set(style="whitegrid", color\_codes=True)

import warnings

warnings.simplefilter(action='ignore')

**# Read CSV train data file into DataFrame**

train\_df = pd.read\_csv("titanic\_train.csv")

**# Read CSV test data file into DataFrame**

test\_df = pd.read\_csv("titanic\_test.csv")

**# preview train data**

train\_df.head()

print('The number of samples into the train data is {}.'.format(train\_df.shape[0]))

test\_df.head()

print('The number of samples into the test data is {}.'.format(test\_df.shape[0]))

**# check missing values in train data**

train\_df.isnull().sum()

**# percent of missing "Age"**

print('Percent of missing "Age" records is %.2f%%' %((train\_df['Age'].isnull().sum()/train\_df.shape[0])\*100))

ax = train\_df["Age"].hist(bins=15, density=True, stacked=True, color='teal', alpha=0.6)

train\_df["Age"].plot(kind='density', color='teal')

ax.set(xlabel='Age')

plt.xlim(-10,85)

plt.show()

**# mean age**

print('The mean of "Age" is %.2f' %(train\_df["Age"].mean(skipna=True)))

**# median age**

print('The median of "Age" is %.2f' %(train\_df["Age"].median(skipna=True)))

**# percent of missing "Cabin"**

print('Percent of missing "Cabin" records is %.2f%%' %((train\_df['Cabin'].isnull().sum()/train\_df.shape[0])\*100))

**#percent of missing "Embarked"**

print('Percent of missing "Embarked" records is %.2f%%' %((train\_df['Embarked'].isnull().sum()/train\_df.shape[0])\*100))

print('Boarded passengers grouped by port of embarkation (C = Cherbourg, Q = Queenstown, S = Southampton):')

print(train\_df['Embarked'].value\_counts())

sns.countplot(x='Embarked', data=train\_df, palette='Set2')

plt.show()

print('The most common boarding port of embarkation is %s.' %train\_df['Embarked'].value\_counts().idxmax())

train\_data = train\_df.copy()

train\_data["Age"].fillna(train\_df["Age"].median(skipna=True), inplace=True)

train\_data["Embarked"].fillna(train\_df['Embarked'].value\_counts().idxmax(), inplace=True)

train\_data.drop('Cabin', axis=1, inplace=True)

**# check missing values in adjusted train data**

train\_data.isnull().sum()

**# preview adjusted train data**

train\_data.head()

plt.figure(figsize=(15,8))

ax = train\_df["Age"].hist(bins=15, density=True, stacked=True, color='teal', alpha=0.6)

train\_df["Age"].plot(kind='density', color='teal')

ax = train\_data["Age"].hist(bins=15, density=True, stacked=True, color='orange', alpha=0.5)

train\_data["Age"].plot(kind='density', color='orange')

ax.legend(['Raw Age', 'Adjusted Age'])

ax.set(xlabel='Age')

plt.xlim(-10,85)

plt.show()

**# Create categorical variable for traveling alone**

train\_data['TravelAlone']=np.where((train\_data["SibSp"]+train\_data["Parch"])>0, 0, 1)

train\_data.drop('SibSp', axis=1, inplace=True)

train\_data.drop('Parch', axis=1, inplace=True)

**#create categorical variables and drop some variables**

training=pd.get\_dummies(train\_data, columns=["Pclass","Embarked","Sex"])

training.drop('Sex\_female', axis=1, inplace=True)

training.drop('PassengerId', axis=1, inplace=True)

training.drop('Name', axis=1, inplace=True)

training.drop('Ticket', axis=1, inplace=True)

final\_train = training

final\_train.head()

Now, apply the same changes to the test data.

I will apply to same imputation for "Age" in the Test data as I did for my Training data (if missing, Age = 28).

I'll also remove the "Cabin" variable from the test data, as I've decided not to include it in my analysis.

There were no missing values in the "Embarked" port variable.

I'll add the dummy variables to finalize the test set.

Finally, I'll impute the 1 missing value for "Fare" with the median, 14.45.

test\_df.isnull().sum()

test\_data = test\_df.copy()

test\_data["Age"].fillna(train\_df["Age"].median(skipna=True), inplace=True)

test\_data["Fare"].fillna(train\_df["Fare"].median(skipna=True), inplace=True)

test\_data.drop('Cabin', axis=1, inplace=True)

test\_data['TravelAlone']=np.where((test\_data["SibSp"]+test\_data["Parch"])>0, 0, 1)

test\_data.drop('SibSp', axis=1, inplace=True)

test\_data.drop('Parch', axis=1, inplace=True)

testing = pd.get\_dummies(test\_data, columns=["Pclass","Embarked","Sex"])

testing.drop('Sex\_female', axis=1, inplace=True)

testing.drop('PassengerId', axis=1, inplace=True)

testing.drop('Name', axis=1, inplace=True)

testing.drop('Ticket', axis=1, inplace=True)

final\_test = testing

final\_test.head()

plt.figure(figsize=(15,8))

ax = sns.kdeplot(final\_train["Age"][final\_train.Survived == 1], color="darkturquoise", shade=True)

sns.kdeplot(final\_train["Age"][final\_train.Survived == 0], color="lightcoral", shade=True)

plt.legend(['Survived', 'Died'])

plt.title('Density Plot of Age for Surviving Population and Deceased Population')

ax.set(xlabel='Age')

plt.xlim(-10,85)

plt.show()

plt.figure(figsize=(20,8))

avg\_survival\_byage = final\_train[["Age", "Survived"]].groupby(['Age'], as\_index=False).mean()

g = sns.barplot(x='Age', y='Survived', data=avg\_survival\_byage, color="LightSeaGreen")

plt.show()

final\_train['IsMinor']=np.where(final\_train['Age']<=16, 1, 0)

final\_test['IsMinor']=np.where(final\_test['Age']<=16, 1, 0)

**#Exploration of Fare**

plt.figure(figsize=(15,8))

ax = sns.kdeplot(final\_train["Fare"][final\_train.Survived == 1], color="darkturquoise", shade=True)

sns.kdeplot(final\_train["Fare"][final\_train.Survived == 0], color="lightcoral", shade=True)

plt.legend(['Survived', 'Died'])

plt.title('Density Plot of Fare for Surviving Population and Deceased Population')

ax.set(xlabel='Fare')

plt.xlim(-20,200)

plt.show()

**#Exploration of Passenger Class**

sns.barplot('Pclass', 'Survived', data=train\_df, color="darkturquoise")

plt.show()

#Exploration of Embarked Port

sns.barplot('Embarked', 'Survived', data=train\_df, color="teal")

plt.show()

**#Exploration of Traveling Alone vs. With Family**

sns.barplot('TravelAlone', 'Survived', data=final\_train, color="mediumturquoise")

plt.show()

#Exploration of Gender Variable

sns.barplot('Sex', 'Survived', data=train\_df, color="aquamarine")

plt.show()

**#Logistic Regression and Results**

from sklearn.linear\_model import LogisticRegression

from sklearn.feature\_selection import RFE

cols = ["Age","Fare","TravelAlone","Pclass\_1","Pclass\_2","Embarked\_C","Embarked\_S","Sex\_male","IsMinor"]

X = final\_train[cols]

y = final\_train['Survived']

**# Build a logreg and compute the feature importances**

model = LogisticRegression()

**# create the RFE model and select 8 attributes**

# rfe = RFE(model, 8)

rfe = RFE(estimator=model, n\_features\_to\_select=8)

rfe = rfe.fit(X, y)

**# summarize the selection of the attributes**

print('Selected features: %s' % list(X.columns[rfe.support\_]))

from sklearn.linear\_model import LogisticRegression

from sklearn.feature\_selection import RFECV

**# Create the RFE object and compute a cross-validated score.**

**# The "accuracy" scoring is proportional to the number of correct classifications**

rfecv = RFECV(estimator=LogisticRegression(), step=1, cv=10, scoring='accuracy')

rfecv.fit(X, y)

print("Optimal number of features: %d" % rfecv.n\_features\_)

print('Selected features: %s' % list(X.columns[rfecv.support\_]))

**# Plot number of features VS. cross-validation scores**

plt.figure(figsize=(10,6))

plt.xlabel("Number of features selected")

plt.ylabel("Cross validation score (nb of correct classifications)")

plt.plot(range(1, len(rfecv.grid\_scores\_) + 1), rfecv.grid\_scores\_)

plt.show()

Selected\_features = ['Age', 'TravelAlone', 'Pclass\_1', 'Pclass\_2', 'Embarked\_C',

'Embarked\_S', 'Sex\_male', 'IsMinor']

X = final\_train[Selected\_features]

plt.subplots(figsize=(8, 5))

sns.heatmap(X.corr(), annot=True, cmap="RdYlGn")

plt.show()

**#Review of model evaluation procedures**

from sklearn.model\_selection import train\_test\_split, cross\_val\_score

from sklearn.metrics import accuracy\_score, classification\_report, precision\_score, recall\_score

from sklearn.metrics import confusion\_matrix, precision\_recall\_curve, roc\_curve, auc, log\_loss

**# create X (features) and y (response)**

X = final\_train[Selected\_features]

y = final\_train['Survived']

**# use train/test split with different random\_state values**

**# we can change the random\_state values that changes the accuracy scores**

**# the scores change a lot, this is why testing scores is a high-variance estimate**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=2)

**# check classification scores of logistic regression**

logreg = LogisticRegression()

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

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

y\_pred\_proba = logreg.predict\_proba(X\_test)[:, 1]

[fpr, tpr, thr] = roc\_curve(y\_test, y\_pred\_proba)

print('Train/Test split results:')

print(logreg.\_\_class\_\_.\_\_name\_\_+" accuracy is %2.3f" % accuracy\_score(y\_test, y\_pred))

print(logreg.\_\_class\_\_.\_\_name\_\_+" log\_loss is %2.3f" % log\_loss(y\_test, y\_pred\_proba))

print(logreg.\_\_class\_\_.\_\_name\_\_+" auc is %2.3f" % auc(fpr, tpr))

idx = np.min(np.where(tpr > 0.95)) # index of the first threshold for which the sensibility > 0.95

plt.figure()

plt.plot(fpr, tpr, color='coral', label='ROC curve (area = %0.3f)' % auc(fpr, tpr))

plt.plot([0, 1], [0, 1], 'k--')

plt.plot([0,fpr[idx]], [tpr[idx],tpr[idx]], 'k--', color='blue')

plt.plot([fpr[idx],fpr[idx]], [0,tpr[idx]], 'k--', color='blue')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate (1 - specificity)', fontsize=14)

plt.ylabel('True Positive Rate (recall)', fontsize=14)

plt.title('Receiver operating characteristic (ROC) curve')

plt.legend(loc="lower right")

plt.show()

print("Using a threshold of %.3f " % thr[idx] + "guarantees a sensitivity of %.3f " % tpr[idx] +

"and a specificity of %.3f" % (1-fpr[idx]) +

", i.e. a false positive rate of %.2f%%." % (np.array(fpr[idx])\*100))

**# 10-fold cross-validation logistic regression**

logreg = LogisticRegression()

**# Use cross\_val\_score function**

**# We are passing the entirety of X and y, not X\_train or y\_train, it takes care of splitting the data**

**# cv=10 for 10 folds**

**# scoring = {'accuracy', 'neg\_log\_loss', 'roc\_auc'} for evaluation metric - althought they are many**

scores\_accuracy = cross\_val\_score(logreg, X, y, cv=10, scoring='accuracy')

scores\_log\_loss = cross\_val\_score(logreg, X, y, cv=10, scoring='neg\_log\_loss')

scores\_auc = cross\_val\_score(logreg, X, y, cv=10, scoring='roc\_auc')

print('K-fold cross-validation results:')

print(logreg.\_\_class\_\_.\_\_name\_\_+" average accuracy is %2.3f" % scores\_accuracy.mean())

print(logreg.\_\_class\_\_.\_\_name\_\_+" average log\_loss is %2.3f" % -scores\_log\_loss.mean())

print(logreg.\_\_class\_\_.\_\_name\_\_+" average auc is %2.3f" % scores\_auc.mean())

from sklearn.model\_selection import cross\_validate

scoring = {'accuracy': 'accuracy', 'log\_loss': 'neg\_log\_loss', 'auc': 'roc\_auc'}

modelCV = LogisticRegression()

results = cross\_validate(modelCV, X, y, cv=10, scoring=list(scoring.values()),

return\_train\_score=False)

print('K-fold cross-validation results:')

for sc in range(len(scoring)):

print(modelCV.\_\_class\_\_.\_\_name\_\_+" average %s: %.3f (+/-%.3f)" % (list(scoring.keys())[sc], -results['test\_%s' % list(scoring.values())[sc]].mean()

if list(scoring.values())[sc]=='neg\_log\_loss'

else results['test\_%s' % list(scoring.values())[sc]].mean(),

results['test\_%s' % list(scoring.values())[sc]].std()))

**#What happens when we add the feature "Fare"?**

cols = ["Age","Fare","TravelAlone","Pclass\_1","Pclass\_2","Embarked\_C","Embarked\_S","Sex\_male","IsMinor"]

X = final\_train[cols]

scoring = {'accuracy': 'accuracy', 'log\_loss': 'neg\_log\_loss', 'auc': 'roc\_auc'}

modelCV = LogisticRegression()

results = cross\_validate(modelCV, final\_train[cols], y, cv=10, scoring=list(scoring.values()),

return\_train\_score=False)

print('K-fold cross-validation results:')

for sc in range(len(scoring)):

print(modelCV.\_\_class\_\_.\_\_name\_\_+" average %s: %.3f (+/-%.3f)" % (list(scoring.keys())[sc], -results['test\_%s' % list(scoring.values())[sc]].mean()

if list(scoring.values())[sc]=='neg\_log\_loss'

else results['test\_%s' % list(scoring.values())[sc]].mean(),

results['test\_%s' % list(scoring.values())[sc]].std()))

from sklearn.model\_selection import GridSearchCV

X = final\_train[Selected\_features]

param\_grid = {'C': np.arange(1e-05, 3, 0.1)}

scoring = {'Accuracy': 'accuracy', 'AUC': 'roc\_auc', 'Log\_loss': 'neg\_log\_loss'}

gs = GridSearchCV(LogisticRegression(), return\_train\_score=True,

param\_grid=param\_grid, scoring=scoring, cv=10, refit='Accuracy')

gs.fit(X, y)

results = gs.cv\_results\_

print('='\*20)

print("best params: " + str(gs.best\_estimator\_))

print("best params: " + str(gs.best\_params\_))

print('best score:', gs.best\_score\_)

print('='\*20)

plt.figure(figsize=(10, 10))

plt.title("GridSearchCV evaluating using multiple scorers simultaneously",fontsize=16)

plt.xlabel("Inverse of regularization strength: C")

plt.ylabel("Score")

plt.grid()

ax = plt.axes()

ax.set\_xlim(0, param\_grid['C'].max())

ax.set\_ylim(0.35, 0.95)

**# Get the regular numpy array from the MaskedArray**

X\_axis = np.array(results['param\_C'].data, dtype=float)

for scorer, color in zip(list(scoring.keys()), ['g', 'k', 'b']):

for sample, style in (('train', '--'), ('test', '-')):

sample\_score\_mean = -results['mean\_%s\_%s' % (sample, scorer)] if scoring[scorer]=='neg\_log\_loss' else results['mean\_%s\_%s' % (sample, scorer)]

sample\_score\_std = results['std\_%s\_%s' % (sample, scorer)]

ax.fill\_between(X\_axis, sample\_score\_mean - sample\_score\_std,

sample\_score\_mean + sample\_score\_std,

alpha=0.1 if sample == 'test' else 0, color=color)

ax.plot(X\_axis, sample\_score\_mean, style, color=color,

alpha=1 if sample == 'test' else 0.7,

label="%s (%s)" % (scorer, sample))

best\_index = np.nonzero(results['rank\_test\_%s' % scorer] == 1)[0][0]

best\_score = -results['mean\_test\_%s' % scorer][best\_index] if scoring[scorer]=='neg\_log\_loss' else results['mean\_test\_%s' % scorer][best\_index]

**# Plot a dotted vertical line at the best score for that scorer marked by x**

ax.plot([X\_axis[best\_index], ] \* 2, [0, best\_score],

linestyle='-.', color=color, marker='x', markeredgewidth=3, ms=8)

**# Annotate the best score for that scorer**

ax.annotate("%0.2f" % best\_score,

(X\_axis[best\_index], best\_score + 0.005))

plt.legend(loc="best")

plt.grid('off')

plt.show()

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import RepeatedStratifiedKFold

from sklearn.pipeline import Pipeline

**#Define simple model**

###############################################################################

C = np.arange(1e-05, 5.5, 0.1)

scoring = {'Accuracy': 'accuracy', 'AUC': 'roc\_auc', 'Log\_loss': 'neg\_log\_loss'}

log\_reg = LogisticRegression()

**#Simple pre-processing estimators**

###############################################################################

std\_scale = StandardScaler(with\_mean=False, with\_std=False)

#std\_scale = StandardScaler()

**#Defining the CV method: Using the Repeated Stratified K Fold**

###############################################################################

n\_folds=5

n\_repeats=5

rskfold = RepeatedStratifiedKFold(n\_splits=n\_folds, n\_repeats=n\_repeats, random\_state=2)

**#Creating simple pipeline and defining the gridsearch**

###############################################################################

log\_clf\_pipe = Pipeline(steps=[('scale',std\_scale), ('clf',log\_reg)])

log\_clf = GridSearchCV(estimator=log\_clf\_pipe, cv=rskfold,

scoring=scoring, return\_train\_score=True,

param\_grid=dict(clf\_\_C=C), refit='Accuracy')

log\_clf.fit(X, y)

results = log\_clf.cv\_results\_

print('='\*20)

print("best params: " + str(log\_clf.best\_estimator\_))

print("best params: " + str(log\_clf.best\_params\_))

print('best score:', log\_clf.best\_score\_)

print('='\*20)

plt.figure(figsize=(10, 10))

plt.title("GridSearchCV evaluating using multiple scorers simultaneously",fontsize=16)

plt.xlabel("Inverse of regularization strength: C")

plt.ylabel("Score")

plt.grid()

ax = plt.axes()

ax.set\_xlim(0, C.max())

ax.set\_ylim(0.35, 0.95)

**# Get the regular numpy array from the MaskedArray**

X\_axis = np.array(results['param\_clf\_\_C'].data, dtype=float)

for scorer, color in zip(list(scoring.keys()), ['g', 'k', 'b']):

for sample, style in (('train', '--'), ('test', '-')):

sample\_score\_mean = -results['mean\_%s\_%s' % (sample, scorer)] if scoring[scorer]=='neg\_log\_loss' else results['mean\_%s\_%s' % (sample, scorer)]

sample\_score\_std = results['std\_%s\_%s' % (sample, scorer)]

ax.fill\_between(X\_axis, sample\_score\_mean - sample\_score\_std,

sample\_score\_mean + sample\_score\_std,

alpha=0.1 if sample == 'test' else 0, color=color)

ax.plot(X\_axis, sample\_score\_mean, style, color=color,

alpha=1 if sample == 'test' else 0.7,

label="%s (%s)" % (scorer, sample))

best\_index = np.nonzero(results['rank\_test\_%s' % scorer] == 1)[0][0]

best\_score = -results['mean\_test\_%s' % scorer][best\_index] if scoring[scorer]=='neg\_log\_loss' else results['mean\_test\_%s' % scorer][best\_index]

**# Plot a dotted vertical line at the best score for that scorer marked by x**

ax.plot([X\_axis[best\_index], ] \* 2, [0, best\_score],

linestyle='-.', color=color, marker='x', markeredgewidth=3, ms=8)

**# Annotate the best score for that scorer**

ax.annotate("%0.2f" % best\_score,

(X\_axis[best\_index], best\_score + 0.005))

plt.legend(loc="best")

plt.grid('off')

plt.show()

final\_test['Survived'] = log\_clf.predict(final\_test[Selected\_features])

final\_test['PassengerId'] = test\_df['PassengerId']

submission = final\_test[['PassengerId','Survived']]

submission.to\_csv("submission.csv", index=False)

submission.tail()

**===============================6 – 3 KNN Classification=======================**

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

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import classification\_report, confusion\_matrix

from sklearn import datasets

iris=datasets.load\_iris()

x = iris.data

y = iris.target

print ('sepal-length', 'sepal-width', 'petal-length', 'petal-width')

print(x)

print('class: 0-Iris-Setosa, 1- Iris-Versicolour, 2- Iris-Virginica')

print(y)

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x,y,test\_size=0.3)

**#To Training the model and Nearest nighbors K=5**

classifier = KNeighborsClassifier(n\_neighbors=5)

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

**#To make predictions on our test data**

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

print('Confusion Matrix')

print(confusion\_matrix(y\_test,y\_pred))

print('Accuracy Metrics')

print(classification\_report(y\_test,y\_pred))

**=====================7 – 1 – Feature Selection PCA ===================**

import numpy as np

#np.random.seed(23423478238423978) # random seed for consistency

# A reader pointed out that Python 2.7 would raise a

# "ValueError: object of too small depth for desired array".

# This can be avoided by choosing a smaller random seed, e.g. 1

# or by completely omitting this line, since I just used the random seed for

# consistency.

mu\_vec1 = np.array([0,0,0])

cov\_mat1 = np.array([[1,0,0],[0,1,0],[0,0,1]])

class1\_sample = np.random.multivariate\_normal(mu\_vec1, cov\_mat1, 20).T

assert class1\_sample.shape == (3,20), "The matrix has not the dimensions 3x20"

mu\_vec2 = np.array([1,1,1])

cov\_mat2 = np.array([[1,0,0],[0,1,0],[0,0,1]])

class2\_sample = np.random.multivariate\_normal(mu\_vec2, cov\_mat2, 20).T

assert class2\_sample.shape == (3,20), "The matrix has not the dimensions 3x20"

%pylab inline

from matplotlib import pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

from mpl\_toolkits.mplot3d import proj3d

fig = plt.figure(figsize=(8,8))

ax = fig.add\_subplot(111, projection='3d')

plt.rcParams['legend.fontsize'] = 10

ax.plot(class1\_sample[0,:], class1\_sample[1,:], class1\_sample[2,:], 'o', markersize=8, color='blue', alpha=0.5, label='class1')

ax.plot(class2\_sample[0,:], class2\_sample[1,:], class2\_sample[2,:], '^', markersize=8, alpha=0.5, color='red', label='class2')

plt.title('Samples for class 1 and class 2')

ax.legend(loc='upper right')

plt.show()

all\_samples = np.concatenate((class1\_sample, class2\_sample), axis=1)

assert all\_samples.shape == (3,40), "The matrix has not the dimensions 3x40"

mean\_x = np.mean(all\_samples[0,:])

mean\_y = np.mean(all\_samples[1,:])

mean\_z = np.mean(all\_samples[2,:])

mean\_vector = np.array([[mean\_x],[mean\_y],[mean\_z]])

scatter\_matrix = np.zeros((3,3))

for i in range(all\_samples.shape[1]):

scatter\_matrix += (all\_samples[:,i].reshape(3,1) - mean\_vector).dot((all\_samples[:,i].reshape(3,1) - mean\_vector).T)

print('Scatter Matrix:\n', scatter\_matrix)

cov\_mat = np.cov([all\_samples[0,:],all\_samples[1,:],all\_samples[2,:]])

print('Covariance Matrix:\n', cov\_mat)

# eigenvectors and eigenvalues for the from the scatter matrix

eig\_val\_sc, eig\_vec\_sc = np.linalg.eig(scatter\_matrix)

# eigenvectors and eigenvalues for the from the covariance matrix

eig\_val\_cov, eig\_vec\_cov = np.linalg.eig(cov\_mat)

for i in range(len(eig\_val\_sc)):

eigvec\_sc = eig\_vec\_sc[:,i].reshape(1,3).T

eigvec\_cov = eig\_vec\_cov[:,i].reshape(1,3).T

assert eigvec\_sc.all() == eigvec\_cov.all(), 'Eigenvectors are not identical'

print('Eigenvector {}: \n{}'.format(i+1, eigvec\_sc))

print('Eigenvalue {} from scatter matrix: {}'.format(i+1, eig\_val\_sc[i]))

print('Eigenvalue {} from covariance matrix: {}'.format(i+1, eig\_val\_cov[i]))

print('Scaling factor: ', eig\_val\_sc[i]/eig\_val\_cov[i])

print(40 \* '-')

for i in range(len(eig\_val\_sc)):

eigv = eig\_vec\_sc[:,i].reshape(1,3).T

np.testing.assert\_array\_almost\_equal(scatter\_matrix.dot(eigv), eig\_val\_sc[i] \* eigv,

decimal=6, err\_msg='', verbose=True)

# =========================ma'am==================

# %matplotlib inline

# from matplotlib import pyplot as plt

# from mpl\_toolkits.mplot3d import Axes3D

# from mpl\_toolkits.mplot3d import proj3d

# from matplotlib.patches import FancyArrowPatch

# class Arrow3D(FancyArrowPatch):

# def \_\_init\_\_(self, xs, ys, zs, \*args, \*\*kwargs):

# FancyArrowPatch.\_\_init\_\_(self, (0,0), (0,0), \*args, \*\*kwargs)

# self.\_verts3d = xs, ys, zs

# def draw(self, renderer):

# xs3d, ys3d, zs3d = self.\_verts3d

# xs, ys, zs = proj3d.proj\_transform(xs3d, ys3d, zs3d, renderer.M)

# self.set\_positions((xs[0],ys[0]),(xs[1],ys[1]))

# FancyArrowPatch.draw(self, renderer)

# =========================MY==================

import numpy as np

from matplotlib import pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

from matplotlib.patches import FancyArrowPatch

from mpl\_toolkits.mplot3d import proj3d

class Arrow3D(FancyArrowPatch):

def \_\_init\_\_(self, xs, ys, zs, \*args, \*\*kwargs):

super().\_\_init\_\_((0,0), (0,0), \*args, \*\*kwargs)

self.\_verts3d = xs, ys, zs

def do\_3d\_projection(self, renderer=None):

xs3d, ys3d, zs3d = self.\_verts3d

xs, ys, zs = proj3d.proj\_transform(xs3d, ys3d, zs3d, self.axes.M)

self.set\_positions((xs[0],ys[0]),(xs[1],ys[1]))

return np.min(zs)

fig = plt.figure(figsize=(7,7))

ax = fig.add\_subplot(111, projection='3d')

ax.plot(all\_samples[0,:], all\_samples[1,:], all\_samples[2,:], 'o', markersize=8, color='green', alpha=0.2)

ax.plot([mean\_x], [mean\_y], [mean\_z], 'o', markersize=10, color='red', alpha=0.5)

for v in eig\_vec\_sc.T:

a = Arrow3D([mean\_x, v[0]], [mean\_y, v[1]], [mean\_z, v[2]], mutation\_scale=20, lw=3, arrowstyle="-|>", color="r")

ax.add\_artist(a)

ax.set\_xlabel('x\_values')

ax.set\_ylabel('y\_values')

ax.set\_zlabel('z\_values')

plt.title('Eigenvectors')

plt.show()

for ev in eig\_vec\_sc:

numpy.testing.assert\_array\_almost\_equal(1.0, np.linalg.norm(ev))

# instead of 'assert' because of rounding errors

# Make a list of (eigenvalue, eigenvector) tuples

eig\_pairs = [(np.abs(eig\_val\_sc[i]), eig\_vec\_sc[:,i]) for i in range(len(eig\_val\_sc))]

# Sort the (eigenvalue, eigenvector) tuples from high to low

eig\_pairs.sort(key=lambda x: x[0], reverse=True)

# Visually confirm that the list is correctly sorted by decreasing eigenvalues

for i in eig\_pairs:

print(i[0])

matrix\_w = np.hstack((eig\_pairs[0][1].reshape(3,1), eig\_pairs[1][1].reshape(3,1)))

print('Matrix W:\n', matrix\_w)

transformed = matrix\_w.T.dot(all\_samples)

assert transformed.shape == (2,40), "The matrix is not 2x40 dimensional."

plt.plot(transformed[0,0:20], transformed[1,0:20], 'o', markersize=7, color='blue', alpha=0.5, label='class1')

plt.plot(transformed[0,20:40], transformed[1,20:40], '^', markersize=7, color='red', alpha=0.5, label='class2')

plt.xlim([-4,4])

plt.ylim([-4,4])

plt.xlabel('x\_values')

plt.ylabel('y\_values')

plt.legend()

plt.title('Transformed samples with class labels')

plt.show()

**=====================7 – 1 – Feature EXCTRACTION PCA ===================**

# Import packages

import numpy as np

from sklearn import decomposition, datasets

from sklearn.preprocessing import StandardScaler

# Load the breast cancer dataset

dataset = datasets.load\_breast\_cancer()

# Load the features

X = dataset.data

# View the shape of the dataset

X.shape

# View the data

X

# Create a scaler object

sc = StandardScaler()

# Fit the scaler to the features and transform

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

# Create a pca object with the 2 components as a parameter

pca = decomposition.PCA(n\_components=2)

# Fit the PCA and transform the data

X\_std\_pca = pca.fit\_transform(X\_std)

# View the new feature data's shape

X\_std\_pca.shape

# View the new feature data

X\_std\_pca

**=====================8 – 1 – KMEAN CLUSTRING ===================**

#import libraries

import pandas as pd

import numpy as np

import random as rd

import matplotlib.pyplot as plt

data = pd.read\_csv('clustering.csv')

data.head()

X = data[["LoanAmount","ApplicantIncome"]]

#Visualise data points

plt.scatter(X["ApplicantIncome"],X["LoanAmount"],c='black')

plt.xlabel('AnnualIncome')

plt.ylabel('Loan Amount (In Thousands)')

plt.show()

Steps 1 and 2 of K-Means were about choosing the number of clusters (k) and selecting random centroids for each cluster. We will pick 3 clusters and then select random observations from the data as the centroids:

K=3

# Select random observation as centroids

Centroids = (X.sample(n=K))

plt.scatter(X["ApplicantIncome"],X["LoanAmount"],c='black')

plt.scatter(Centroids["ApplicantIncome"],Centroids["LoanAmount"],c='red')

plt.xlabel('AnnualIncome')

plt.ylabel('Loan Amount (In Thousands)')

plt.show()

# Step 3 - Assign all the points to the closest cluster centroid

# Step 4 - Recompute centroids of newly formed clusters

# Step 5 - Repeat step 3 and 4

pd.options.mode.chained\_assignment = None # default='warn'

diff = 1

j=0

while(diff!=0):

XD=X

i=1

for index1,row\_c in Centroids.iterrows():

ED=[]

for index2,row\_d in XD.iterrows():

d1=(row\_c["ApplicantIncome"]-row\_d["ApplicantIncome"])\*\*2

d2=(row\_c["LoanAmount"]-row\_d["LoanAmount"])\*\*2

d=np.sqrt(d1+d2)

ED.append(d)

X[i]=ED

i=i+1

C=[]

for index,row in X.iterrows():

min\_dist=row[1]

pos=1

for i in range(K):

if row[i+1] < min\_dist:

min\_dist = row[i+1]

pos=i+1

C.append(pos)

X["Cluster"]=C

Centroids\_new = X.groupby(["Cluster"]).mean()[["LoanAmount","ApplicantIncome"]]

if j == 0:

diff=1

j=j+1

else:

diff = (Centroids\_new['LoanAmount'] - Centroids['LoanAmount']).sum() + (Centroids\_new['ApplicantIncome'] - Centroids['ApplicantIncome']).sum()

print(diff.sum())

Centroids = X.groupby(["Cluster"]).mean()[["LoanAmount","ApplicantIncome"]]

When this difference is 0, we are stopping the training. Let’s now visualize the clusters we have got:

color=['blue','green','cyan']

for k in range(K):

data=X[X["Cluster"]==k+1]

plt.scatter(data["ApplicantIncome"],data["LoanAmount"],c=color[k])

plt.scatter(Centroids["ApplicantIncome"],Centroids["LoanAmount"],c='red')

plt.xlabel('Income')

plt.ylabel('Loan Amount (In Thousands)')

plt.show()

**=====================8 – 2 – KMEDOIDALGO =======================**

import numpy as np

import matplotlib.pyplot as plt

import torch

import sys

import random

mean1 = [-2, 0]

cov1 = [[2, 0.9], [0.9, 1]]

data1 = np.random.multivariate\_normal(mean1, cov1, 300)

mean2 = [3, 4]

cov2 = [[3, 0.1], [0.1, 3]]

data2 = np.random.multivariate\_normal(mean2, cov2, 300)

mean3 = [6, -6]

cov3 = [[2, -1], [-1, 2]]

data3 = np.random.multivariate\_normal(mean3, cov3, 300)

data = np.concatenate((data1, data2, data3), axis=0)

centers = np.asarray([mean1, mean2, mean3])

# Plot

fig1 = plt.scatter(data1[:, 0], data1[:, 1])

fig2 = plt.scatter(data2[:, 0], data2[:, 1])

fig3 = plt.scatter(data3[:, 0], data3[:, 1])

fig4 = plt.scatter(centers[:, 0], centers[:, 1], marker="+", s=100)

# plt.scatter(data[:, 0], data[:, 1])

plt.title('Gaussian-Mixture Model')

plt.xlabel('x')

plt.ylabel('y')

plt.legend((fig1, fig2, fig3, fig4),

('Gaussian1', 'Guassian2', 'Guassian3', 'Means'),

scatterpoints=1,

loc='lower left',

ncol=2,

fontsize=10)

plt.savefig('before\_k\_medoids.png')

plt.show()

# construct the similarity matrix

num = len(data)

similarity\_matrix = np.zeros((num, num))

for i in range(0, num):

for j in range(i+1, num):

diff = data[i] - data[j]

dist\_tmp = np.linalg.norm(diff)

similarity\_matrix[i][j] = dist\_tmp

similarity\_matrix[j][i] = dist\_tmp

similarity\_matrix = torch.from\_numpy(similarity\_matrix)

def k\_medoids(similarity\_matrix, k):

# Step 1: Select initial medoids

num = len(similarity\_matrix)

row\_sums = torch.sum(similarity\_matrix, dim=1)

normalized\_sim = similarity\_matrix.T / row\_sums

normalized\_sim = normalized\_sim.T

priority\_scores = -torch.sum(normalized\_sim, dim=0)

values, indices = priority\_scores.topk(k)

tmp = -similarity\_matrix[:, indices]

tmp\_values, tmp\_indices = tmp.topk(1, dim=1)

min\_distance = -torch.sum(tmp\_values)

cluster\_assignment = tmp\_indices.resize\_(num)

print(min\_distance)

# Step 2: Update medoids

for i in range(k):

sub\_indices = (cluster\_assignment == i).nonzero()

sub\_num = len(sub\_indices)

sub\_indices = sub\_indices.resize\_(sub\_num)

sub\_similarity\_matrix = torch.index\_select(similarity\_matrix, 0, sub\_indices)

sub\_similarity\_matrix = torch.index\_select(sub\_similarity\_matrix, 1, sub\_indices)

sub\_row\_sums = torch.sum(sub\_similarity\_matrix, dim=1)

sub\_medoid\_index = torch.argmin(sub\_row\_sums)

# update the cluster medoid index

indices[i] = sub\_indices[sub\_medoid\_index]

# Step 3: Assign objects to medoids

tmp = -similarity\_matrix[:, indices]

tmp\_values, tmp\_indices = tmp.topk(1, dim=1)

total\_distance = -torch.sum(tmp\_values)

cluster\_assignment = tmp\_indices.resize\_(num)

print(total\_distance)

while (total\_distance < min\_distance):

min\_distance = total\_distance

# Step 2: Update medoids

for i in range(k):

sub\_indices = (cluster\_assignment == i).nonzero()

sub\_num = len(sub\_indices)

sub\_indices = sub\_indices.resize\_(sub\_num)

sub\_similarity\_matrix = torch.index\_select(similarity\_matrix, 0, sub\_indices)

sub\_similarity\_matrix = torch.index\_select(sub\_similarity\_matrix, 1, sub\_indices)

sub\_row\_sums = torch.sum(sub\_similarity\_matrix, dim=1)

sub\_medoid\_index = torch.argmin(sub\_row\_sums)

# update the cluster medoid index

indices[i] = sub\_indices[sub\_medoid\_index]

# Step 3: Assign objects to medoids

tmp = -similarity\_matrix[:, indices]

tmp\_values, tmp\_indices = tmp.topk(1, dim=1)

total\_distance = -torch.sum(tmp\_values)

cluster\_assignment = tmp\_indices.resize\_(num)

print(total\_distance)

return indices

indices = k\_medoids(similarity\_matrix, k=3)

medoids = []

for i in range(3):

medoids.append(data[indices[i]])

medoids = np.asarray(medoids)

fig1 = plt.scatter(data1[:, 0], data1[:, 1])

fig2 = plt.scatter(data2[:, 0], data2[:, 1])

fig3 = plt.scatter(data3[:, 0], data3[:, 1])

fig4 = plt.scatter(centers[:, 0], centers[:, 1], marker="+", s=200)

fig5 = plt.scatter(medoids[:, 0], medoids[:, 1], marker="\*", s=200)

plt.title('Gaussian-Mixture Model')

plt.xlabel('x')

plt.ylabel('y')

plt.legend((fig1, fig2, fig3, fig4, fig5),

('Gaussian1', 'Guassian2', 'Guassian3', 'Means', 'Medoids'),

scatterpoints=1,

loc='lower left',

ncol=2,

fontsize=10)

plt.savefig('after\_k\_medoids.png')

plt.show()

**=====================9 – 1 – SVM\_linear ===================**

%matplotlib inline

import matplotlib

import matplotlib.pyplot as plt

def plot\_svc\_decision\_boundary(svm\_clf, xmin, xmax):

w = svm\_clf.coef\_[0]

b = svm\_clf.intercept\_[0]

# At the decision boundary, w0\*x0 + w1\*x1 + b = 0

# => x1 = -w0/w1 \* x0 - b/w1

x0 = np.linspace(xmin, xmax, 200)

decision\_boundary = -w[0]/w[1] \* x0 - b/w[1]

margin = 1/w[1]

gutter\_up = decision\_boundary + margin

gutter\_down = decision\_boundary - margin

svs = svm\_clf.support\_vectors\_

plt.scatter(svs[:, 0], svs[:, 1], s=180, facecolors='#FFAAAA')

plt.plot(x0, decision\_boundary, "k-", linewidth=2)

plt.plot(x0, gutter\_up, "k--", linewidth=2)

plt.plot(x0, gutter\_down, "k--", linewidth=2)

from sklearn.svm import SVC

from sklearn import datasets

iris = datasets.load\_iris()

#print(iris)

X = iris["data"][:, (2, 3)] # petal length, petal width

#print(X)

y = iris["target"]

setosa\_or\_versicolor = (y == 0) | (y == 1)

X = X[setosa\_or\_versicolor]

y = y[setosa\_or\_versicolor]

# SVM Classifier model

#the hyperparameter control the margin violations

#smaller C leads to more margin violations but wider street

#C can be inferred

svm\_clf = SVC(kernel="linear", C=float("inf"))

svm\_clf.fit(X, y)

svm\_clf.predict([[2.4, 3.1]])

#SVM classifiers do not output a probability like logistic regression classifiers

#plot the decision boundaries

import numpy as np

plt.figure(figsize=(12,3.2))

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

svm\_clf.fit(X\_scaled, y)

plt.plot(X\_scaled[:, 0][y==1], X\_scaled[:, 1][y==1], "bo")

plt.plot(X\_scaled[:, 0][y==0], X\_scaled[:, 1][y==0], "ms")

plot\_svc\_decision\_boundary(svm\_clf, -2, 2)

plt.xlabel("Petal Width normalized", fontsize=12)

plt.ylabel("Petal Length normalized", fontsize=12)

plt.title("Scaled", fontsize=16)

plt.axis([-2, 2, -2, 2])

**=====================9 – 2 – SVM\_NON linear ===================**

from sklearn.datasets import make\_moons

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import PolynomialFeatures

from sklearn.preprocessing import StandardScaler

from sklearn.svm import SVC

import numpy as np

%matplotlib inline

import matplotlib

import matplotlib.pyplot as plt

from sklearn.datasets import make\_moons

X, y = make\_moons(n\_samples=100, noise=0.15, random\_state=42)

#define a function to plot the dataset

def plot\_dataset(X, y, axes):

plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")

plt.plot(X[:, 0][y==1], X[:, 1][y==1], "ms")

plt.axis(axes)

plt.grid(True, which='both')

plt.xlabel(r"$x\_1$", fontsize=20)

plt.ylabel(r"$x\_2$", fontsize=20, rotation=0)

#Let's have a look at the data we have generated

plot\_dataset(X, y, [-1.5, 2.5, -1, 1.5])

plt.show()

#define a function plot the decision boundaries

def plot\_predictions(clf, axes):

#create data in continous linear space

x0s = np.linspace(axes[0], axes[1], 100)

x1s = np.linspace(axes[2], axes[3], 100)

x0, x1 = np.meshgrid(x0s, x1s)

X = np.c\_[x0.ravel(), x1.ravel()]

y\_pred = clf.predict(X).reshape(x0.shape)

y\_decision = clf.decision\_function(X).reshape(x0.shape)

plt.contourf(x0, x1, y\_pred, cmap=plt.cm.brg, alpha=0.2)

plt.contourf(x0, x1, y\_decision, cmap=plt.cm.brg, alpha=0.1)

#C controls the width of the street

#Degree of data

#create a pipeline to create features, scale data and fit the model

polynomial\_svm\_clf = Pipeline((

("poly\_features", PolynomialFeatures(degree=3)),

("scalar", StandardScaler()),

("svm\_clf", SVC(kernel="poly", degree=10, coef0=1, C=5))

))

#call the pipeline

polynomial\_svm\_clf.fit(X,y)

#plot the decision boundaries

plt.figure(figsize=(11, 4))

#plot the decision boundaries

plot\_predictions(polynomial\_svm\_clf, [-1.5, 2.5, -1, 1.5])

#plot the dataset

plot\_dataset(X, y, [-1.5, 2.5, -1, 1.5])

plt.title(r"$d=3, coef0=1, C=5$", fontsize=18)

plt.show()

**=====================10 – 1 – DECISION TREE ===================**

from matplotlib import pyplot as plt

from sklearn import datasets

from sklearn.tree import DecisionTreeClassifier

from sklearn import tree

# Prepare the data data

iris = datasets.load\_iris()

X = iris.data

y = iris.target

# Fit the classifier with default hyper-parameters

clf = DecisionTreeClassifier(random\_state=1234)

model = clf.fit(X, y)

text\_representation = tree.export\_text(clf)

print(text\_representation)

with open("decistion\_tree.log", "w") as fout:

fout.write(text\_representation)

fig = plt.figure(figsize=(25,20))

\_ = tree.plot\_tree(clf,

feature\_names=iris.feature\_names,

class\_names=iris.target\_names,

filled=True)

**=====================10 – 2 – Random Forest Classfication ===================**

Random Forest for Classifying Digits

from sklearn.datasets import load\_digits

import matplotlib.pyplot as plt

digits = load\_digits()

digits.keys()

# set up the figure

fig = plt.figure(figsize=(6, 6)) # figure size in inches

fig.subplots\_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)

# plot the digits: each image is 8x8 pixels

for i in range(64):

ax = fig.add\_subplot(8, 8, i + 1, xticks=[], yticks=[])

ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')

# label the image with the target value

ax.text(0, 7, str(digits.target[i]))

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

from sklearn.ensemble import RandomForestClassifier

Xtrain, Xtest, ytrain, ytest = train\_test\_split(digits.data, digits.target,

random\_state=0)

model = RandomForestClassifier(n\_estimators=1000)

model.fit(Xtrain, ytrain)

ypred = model.predict(Xtest)

from sklearn import metrics

print(metrics.classification\_report(ypred, ytest))

from sklearn.metrics import confusion\_matrix

import seaborn as sns

mat = confusion\_matrix(ytest, ypred)

sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False)

plt.xlabel('true label')

plt.ylabel('predicted label');

**=====================10 – 2 – Random Forest Regrations ===================**

from sklearn.datasets import load\_digits

import matplotlib.pyplot as plt

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

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import confusion\_matrix

import seaborn as sns

import numpy as np

rng = np.random.RandomState(42)

x = 10 \* rng.rand(200)

def model(x, sigma=0.3):

fast\_oscillation = np.sin(5 \* x)

slow\_oscillation = np.sin(0.5 \* x)

noise = sigma \* rng.randn(len(x))

return slow\_oscillation + fast\_oscillation + noise

y = model(x)

plt.errorbar(x, y, 0.3, fmt='o');

from sklearn.ensemble import RandomForestRegressor

forest = RandomForestRegressor(200)

forest.fit(x[:, None], y)

xfit = np.linspace(0, 10, 1000)

yfit = forest.predict(xfit[:, None])

ytrue = model(xfit, sigma=0)

plt.errorbar(x, y, 0.3, fmt='o', alpha=0.5)

plt.plot(xfit, yfit, '-r');

plt.plot(xfit, ytrue, '-k', alpha=0.5);

**=====================11 – 1 – ADABOOST ===================**

from typing import Optional

import numpy as np

import matplotlib.pyplot as plt

import matplotlib as mpl

def plot\_adaboost(X: np.ndarray,

y: np.ndarray,

clf=None,

sample\_weights: Optional[np.ndarray] = None,

annotate: bool = False,

ax: Optional[mpl.axes.Axes] = None) -> None:

""" Plot ± samples in 2D, optionally with decision boundary """

assert set(y) == {-1, 1}, 'Expecting response labels to be ±1'

if not ax:

fig, ax = plt.subplots(figsize=(5, 5), dpi=100)

fig.set\_facecolor('white')

pad = 1

x\_min, x\_max = X[:, 0].min() - pad, X[:, 0].max() + pad

y\_min, y\_max = X[:, 1].min() - pad, X[:, 1].max() + pad

if sample\_weights is not None:

sizes = np.array(sample\_weights) \* X.shape[0] \* 100

else:

sizes = np.ones(shape=X.shape[0]) \* 100

X\_pos = X[y == 1]

sizes\_pos = sizes[y == 1]

ax.scatter(\*X\_pos.T, s=sizes\_pos, marker='+', color='red')

X\_neg = X[y == -1]

sizes\_neg = sizes[y == -1]

ax.scatter(\*X\_neg.T, s=sizes\_neg, marker='.', c='blue')

if clf:

plot\_step = 0.01

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, plot\_step),

np.arange(y\_min, y\_max, plot\_step))

Z = clf.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

# If all predictions are positive class, adjust color map acordingly

if list(np.unique(Z)) == [1]:

fill\_colors = ['r']

else:

fill\_colors = ['b', 'r']

ax.contourf(xx, yy, Z, colors=fill\_colors, alpha=0.2)

if annotate:

for i, (x, y) in enumerate(X):

offset = 0.05

ax.annotate(f'$x\_{i + 1}$', (x + offset, y - offset))

ax.set\_xlim(x\_min+0.5, x\_max-0.5)

ax.set\_ylim(y\_min+0.5, y\_max-0.5)

ax.set\_xlabel('$x\_1$')

ax.set\_ylabel('$x\_2$')

from sklearn.datasets import make\_gaussian\_quantiles

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

def make\_toy\_dataset(n: int = 100, random\_seed: int = None):

""" Generate a toy dataset for evaluating AdaBoost classifiers """

n\_per\_class = int(n/2)

if random\_seed:

np.random.seed(random\_seed)

X, y = make\_gaussian\_quantiles(n\_samples=n, n\_features=2, n\_classes=2)

return X, y\*2-1

X, y = make\_toy\_dataset(n=10, random\_seed=10)

plot\_adaboost(X, y)

from sklearn.ensemble import AdaBoostClassifier

bench = AdaBoostClassifier(n\_estimators=10, algorithm='SAMME').fit(X, y)

plot\_adaboost(X, y, bench)

train\_err = (bench.predict(X) != y).mean()

print(f'Train error: {train\_err:.1%}')

class AdaBoost:

def \_\_init\_\_(self):

self.stumps = None

self.stump\_weights = None

self.errors = None

self.sample\_weights = None

def \_check\_X\_y(self, X, y):

""" Validate assumptions about format of input data"""

assert set(y) == {-1, 1}, 'Response variable must be ±1'

return X, y

from sklearn.tree import DecisionTreeClassifier

def fit(self, X: np.ndarray, y: np.ndarray, iters: int):

""" Fit the model using training data """

X, y = self.\_check\_X\_y(X, y)

n = X.shape[0]

# init numpy arrays

self.sample\_weights = np.zeros(shape=(iters, n))

self.stumps = np.zeros(shape=iters, dtype=object)

self.stump\_weights = np.zeros(shape=iters)

self.errors = np.zeros(shape=iters)

# initialize weights uniformly

self.sample\_weights[0] = np.ones(shape=n) / n

for t in range(iters):

# fit weak learner

curr\_sample\_weights = self.sample\_weights[t]

stump = DecisionTreeClassifier(max\_depth=1, max\_leaf\_nodes=2)

stump = stump.fit(X, y, sample\_weight=curr\_sample\_weights)

# calculate error and stump weight from weak learner prediction

stump\_pred = stump.predict(X)

err = curr\_sample\_weights[(stump\_pred != y)].sum()# / n

stump\_weight = np.log((1 - err) / err) / 2

# update sample weights

new\_sample\_weights = (

curr\_sample\_weights \* np.exp(-stump\_weight \* y \* stump\_pred)

)

new\_sample\_weights /= new\_sample\_weights.sum()

# If not final iteration, update sample weights for t+1

if t+1 < iters:

self.sample\_weights[t+1] = new\_sample\_weights

# save results of iteration

self.stumps[t] = stump

self.stump\_weights[t] = stump\_weight

self.errors[t] = err

return self

#Making predictions

#We make a final prediction by taking a “weighted majority vote”, calculated as the sign (±) of the linear combination of each stump’s prediction and its corresponding stump weight.

#$$ H\_t(x) = \text{sign} \Big( \sum\_{t=1}^T a\_t h\_t(x) \Big) $$

def predict(self, X):

""" Make predictions using already fitted model """

stump\_preds = np.array([stump.predict(X) for stump in self.stumps])

return np.sign(np.dot(self.stump\_weights, stump\_preds))

# assign our individually defined functions as methods of our classifier

AdaBoost.fit = fit

AdaBoost.predict = predict

clf = AdaBoost().fit(X, y, iters=10)

plot\_adaboost(X, y, clf)

train\_err = (clf.predict(X) != y).mean()

print(f'Train error: {train\_err:.1%}')

def truncate\_adaboost(clf, t: int):

""" Truncate a fitted AdaBoost up to (and including) a particular iteration """

assert t > 0, 't must be a positive integer'

from copy import deepcopy

new\_clf = deepcopy(clf)

new\_clf.stumps = clf.stumps[:t]

new\_clf.stump\_weights = clf.stump\_weights[:t]

return new\_clf

def plot\_staged\_adaboost(X, y, clf, iters=10):

""" Plot weak learner and cumulaive strong learner at each iteration. """

# larger grid

fig, axes = plt.subplots(figsize=(8, iters\*3),

nrows=iters,

ncols=2,

sharex=True,

dpi=100)

fig.set\_facecolor('white')

\_ = fig.suptitle('Decision boundaries by iteration')

for i in range(iters):

ax1, ax2 = axes[i]

# Plot weak learner

\_ = ax1.set\_title(f'Weak learner at t={i + 1}')

plot\_adaboost(X, y, clf.stumps[i],

sample\_weights=clf.sample\_weights[i],

annotate=False, ax=ax1)

# Plot strong learner

trunc\_clf = truncate\_adaboost(clf, t=i + 1)

\_ = ax2.set\_title(f'Strong learner at t={i + 1}')

plot\_adaboost(X, y, trunc\_clf,

sample\_weights=clf.sample\_weights[i],

annotate=False, ax=ax2)

plt.tight\_layout()

plt.subplots\_adjust(top=0.95)

plt.show()

clf = AdaBoost().fit(X, y, iters=10)

plot\_staged\_adaboost(X, y, clf)

**=====================11 – 2 – ADABOOST decision tree ===================**

#importing libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from random import sample

import random

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import confusion\_matrix

from sklearn import tree

from math import log,exp

pd.set\_option('display.max\_rows', 500)

pd.set\_option('display.max\_columns', 500)

#importing file

iris = pd.read\_csv("iris.csv")

iris = iris.drop('Unnamed: 0', axis=1)

iris.head(1)

#considering only two classes

example = iris[(iris['Species'] == 'versicolor') | (iris['Species'] == 'virginica')]

example.head(2)

#replacing the two classes with +1 and -1

example['Label'] = example['Species'].replace(to\_replace = ['versicolor','virginica'], value=[1,-1])

example = example.drop('Species', axis = 1)

#Initially assign same weights to each records in the dataset

example['probR1'] = 1/(example.shape[0])

example.head(5)

#simple random sample with replacement

random.seed(10)

example1 = example.sample(len(example), replace = True, weights = example['probR1'])

example1

#X\_train and Y\_train split

X\_train = example1.iloc[0:len(iris),0:4]

y\_train = example1.iloc[0:len(iris),4]

#fitting the DT model with depth one

clf\_gini = DecisionTreeClassifier(criterion = "gini", random\_state = 100, max\_depth=1)

clf = clf\_gini.fit(X\_train, y\_train)

#plotting tree for round 1 boosting

tree.plot\_tree(clf)

#prediction

y\_pred = clf\_gini.predict(example.iloc[0:len(iris),0:4])

y\_pred

#adding a column pred1 after the first round of boosting

example['pred1'] = y\_pred

example

#misclassified = 0 if the label and prediction are same

example.loc[example.Label != example.pred1, 'misclassified'] = 1

example.loc[example.Label == example.pred1, 'misclassified'] = 0

#error calculation

e1 = sum(example['misclassified'] \* example['probR1'])

e1

#calculation of alpha (performance)

alpha1 = 0.5\*log((1-e1)/e1)

#update weight

new\_weight = example['probR1']\*np.exp(-1\*alpha1\*example['Label']\*example['pred1'])

#normalized weight

z = sum(new\_weight)

normalized\_weight = new\_weight/sum(new\_weight)

example['prob2'] = round(normalized\_weight,4)

example

#round 2

random.seed(20)

example2 = example.sample(len(example), replace = True, weights = example['prob2'])

example2 = example2.iloc[:,0:5]

X\_train = example2.iloc[0:len(iris),0:4]

y\_train = example2.iloc[0:len(iris),4]

clf\_gini = DecisionTreeClassifier(criterion = "gini", random\_state = 100, max\_depth=1)

clf = clf\_gini.fit(X\_train, y\_train)

y\_pred = clf\_gini.predict(example.iloc[0:len(iris),0:4])

#adding a column pred2 after the second round of boosting

example['pred2'] = y\_pred

#plotting tree for round 2 boosting

tree.plot\_tree(clf)

example

#adding a field misclassified2

example.loc[example.Label != example.pred2, 'misclassified2'] = 1

example.loc[example.Label == example.pred2, 'misclassified2'] = 0

# calculation of error

e2 = sum(example['misclassified2'] \* example['prob2'])

e2

#calculation of alpha

alpha2 = 0.5\*log((1-e2)/e2)

alpha2

#update weight

new\_weight = example['prob2']\*np.exp(-1\*alpha2\*example['Label']\*example['pred2'])

z = sum(new\_weight)

normalized\_weight = new\_weight/sum(new\_weight)

example['prob3'] = round(normalized\_weight,4)

example

#round 3

random.seed(30)

example3 = example.sample(len(example), replace = True, weights = example['prob3'])

example3 = example3.iloc[:,0:5]

X\_train = example3.iloc[0:len(iris),0:4]

y\_train = example3.iloc[0:len(iris),4]

clf\_gini = DecisionTreeClassifier(criterion = "gini", random\_state = 100, max\_depth=1)

clf = clf\_gini.fit(X\_train, y\_train)

#adding a column pred3 after the third round of boosting

y\_pred = clf\_gini.predict(example.iloc[0:len(iris),0:4])

example['pred3'] = y\_pred

#plotting tree for round 3 boosting

tree.plot\_tree(clf)

#adding a field misclassified3

example.loc[example.Label != example.pred3, 'misclassified3'] = 1

example.loc[example.Label == example.pred3, 'misclassified3'] = 0

#weighted error calculation

e3 = sum(example['misclassified3'] \* example['prob3']) #/len(example)

e3

#calculation of performance(alpha)

alpha3 = 0.5\*log((1-e3)/e3)

#update weight

new\_weight = example['prob3']\*np.exp(-1\*alpha3\*example['Label']\*example['pred3'])

z = sum(new\_weight)

normalized\_weight = new\_weight/sum(new\_weight)

example['prob4'] = round(normalized\_weight,4)

example

#Round 4

random.seed(40)

example4 = example.sample(len(example), replace = True, weights = example['prob4'])

example4 = example4.iloc[:,0:5]

X\_train = example4.iloc[0:len(iris),0:4]

y\_train = example4.iloc[0:len(iris),4]

clf\_gini = DecisionTreeClassifier(criterion = "gini", random\_state = 100, max\_depth=1)

clf = clf\_gini.fit(X\_train, y\_train)

#adding a column pred4 after the fourth round of boosting

y\_pred = clf\_gini.predict(example.iloc[0:len(iris),0:4])

example['pred4'] = y\_pred

#plotting tree for round 4 boosting

tree.plot\_tree(clf)

#adding a field misclassified4

example.loc[example.Label != example.pred4, 'misclassified4'] = 1

example.loc[example.Label == example.pred4, 'misclassified4'] = 0

#error calculation

e4 = sum(example['misclassified4'] \* example['prob4'])

e4

# calculation of performance (alpha)

alpha4 = 0.5\*log((1-e4)/e4)

#printing the alpha value which is used in each round of boosting

print(alpha1)

print(alpha2)

print(alpha3)

print(alpha4)

#final prediction

t = alpha1 \* example['pred1'] + alpha2 \* example['pred2'] + alpha3 \* example['pred3'] + alpha4 \* example['pred4']

#sign of the final prediction

np.sign(list(t))

example['final\_pred'] = np.sign(list(t))

example

#Confusion matrix

c=confusion\_matrix(example['Label'], example['final\_pred'])

c

#Overall Accuracy

(c[0,0]+c[1,1])/np.sum(c)\*100

#Fitting the model using the adaboost classifier library

from sklearn.ensemble import AdaBoostClassifier

iris = pd.read\_csv("iris.csv")

iris = iris.drop('Unnamed: 0', axis=1)

iris = iris[(iris['Species'] == 'versicolor') | (iris['Species'] == 'virginica')]

#X\_train and Y\_train split

X\_train = iris.iloc[0:len(iris),0:4]

y\_train = iris.iloc[0:len(iris),4]

clf = AdaBoostClassifier(n\_estimators=4, random\_state=0)

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

clf.predict([[5.5, 2.5, 4.0, 1.3]])

clf.score(X\_train, y\_train)

**=====================11 – 3 – HARD\_VOTING ===================**

# test classification dataset

from sklearn.datasets import make\_classification

# define dataset

X, y = make\_classification(n\_samples=1000, n\_features=20, n\_informative=15, n\_redundant=5, random\_state=2)

# summarize the dataset

print(X.shape, y.shape)

Voting Ensemble for Classification

Hard Voting Ensemble for Classification

# get a voting ensemble of models

def get\_voting():

# define the base models

models = list()

models.append(('knn1', KNeighborsClassifier(n\_neighbors=1)))

models.append(('knn3', KNeighborsClassifier(n\_neighbors=3)))

models.append(('knn5', KNeighborsClassifier(n\_neighbors=5)))

models.append(('knn7', KNeighborsClassifier(n\_neighbors=7)))

models.append(('knn9', KNeighborsClassifier(n\_neighbors=9)))

# define the voting ensemble

ensemble = VotingClassifier(estimators=models, voting='hard')

return ensemble

# get a list of models to evaluate

def get\_models():

models = dict()

models['knn1'] = KNeighborsClassifier(n\_neighbors=1)

models['knn3'] = KNeighborsClassifier(n\_neighbors=3)

models['knn5'] = KNeighborsClassifier(n\_neighbors=5)

models['knn7'] = KNeighborsClassifier(n\_neighbors=7)

models['knn9'] = KNeighborsClassifier(n\_neighbors=9)

models['hard\_voting'] = get\_voting()

return models

# evaluate a give model using cross-validation

def evaluate\_model(model, X, y):

cv = RepeatedStratifiedKFold(n\_splits=10, n\_repeats=3, random\_state=1)

scores = cross\_val\_score(model, X, y, scoring='accuracy', cv=cv, n\_jobs=-1, error\_score='raise')

return scores

# compare hard voting to standalone classifiers

from numpy import mean

from numpy import std

from sklearn.datasets import make\_classification

from sklearn.model\_selection import cross\_val\_score

from sklearn.model\_selection import RepeatedStratifiedKFold

from sklearn.neighbors import KNeighborsClassifier

from sklearn.ensemble import VotingClassifier

from matplotlib import pyplot

# get the dataset

def get\_dataset():

X, y = make\_classification(n\_samples=1000, n\_features=20, n\_informative=15, n\_redundant=5, random\_state=2)

return X, y

# get a voting ensemble of models

def get\_voting():

# define the base models

models = list()

models.append(('knn1', KNeighborsClassifier(n\_neighbors=1)))

models.append(('knn3', KNeighborsClassifier(n\_neighbors=3)))

models.append(('knn5', KNeighborsClassifier(n\_neighbors=5)))

models.append(('knn7', KNeighborsClassifier(n\_neighbors=7)))

models.append(('knn9', KNeighborsClassifier(n\_neighbors=9)))

# define the voting ensemble

ensemble = VotingClassifier(estimators=models, voting='hard')

return ensemble

# get a list of models to evaluate

def get\_models():

models = dict()

models['knn1'] = KNeighborsClassifier(n\_neighbors=1)

models['knn3'] = KNeighborsClassifier(n\_neighbors=3)

models['knn5'] = KNeighborsClassifier(n\_neighbors=5)

models['knn7'] = KNeighborsClassifier(n\_neighbors=7)

models['knn9'] = KNeighborsClassifier(n\_neighbors=9)

models['hard\_voting'] = get\_voting()

return models

# evaluate a give model using cross-validation

def evaluate\_model(model, X, y):

cv = RepeatedStratifiedKFold(n\_splits=10, n\_repeats=3, random\_state=1)

scores = cross\_val\_score(model, X, y, scoring='accuracy', cv=cv, n\_jobs=-1, error\_score='raise')

return scores

# define dataset

X, y = get\_dataset()

# get the models to evaluate

models = get\_models()

# evaluate the models and store results

results, names = list(), list()

for name, model in models.items():

scores = evaluate\_model(model, X, y)

results.append(scores)

names.append(name)

print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))

# plot model performance for comparison

pyplot.boxplot(results, labels=names, showmeans=True)

pyplot.show()

# make a prediction with a hard voting ensemble

from sklearn.datasets import make\_classification

from sklearn.ensemble import VotingClassifier

from sklearn.neighbors import KNeighborsClassifier

# define dataset

X, y = make\_classification(n\_samples=1000, n\_features=20, n\_informative=15, n\_redundant=5, random\_state=2)

# define the base models

models = list()

models.append(('knn1', KNeighborsClassifier(n\_neighbors=1)))

models.append(('knn3', KNeighborsClassifier(n\_neighbors=3)))

models.append(('knn5', KNeighborsClassifier(n\_neighbors=5)))

models.append(('knn7', KNeighborsClassifier(n\_neighbors=7)))

models.append(('knn9', KNeighborsClassifier(n\_neighbors=9)))

# define the hard voting ensemble

ensemble = VotingClassifier(estimators=models, voting='hard')

# fit the model on all available data

ensemble.fit(X, y)

# make a prediction for one example

data = [[5.88891819,2.64867662,-0.42728226,-1.24988856,-0.00822,-3.57895574,2.87938412,-1.55614691,-0.38168784,7.50285659,-1.16710354,-5.02492712,-0.46196105,-0.64539455,-1.71297469,0.25987852,-0.193401,-5.52022952,0.0364453,-1.960039]]

yhat = ensemble.predict(data)

print('Predicted Class: %d' % (yhat))

**=====================11 – 4 – SOFT\_VOTING ===================**

# get a voting ensemble of models

def get\_voting():

# define the base models

models = list()

models.append(('svm1', SVC(probability=True, kernel='poly', degree=1)))

models.append(('svm2', SVC(probability=True, kernel='poly', degree=2)))

models.append(('svm3', SVC(probability=True, kernel='poly', degree=3)))

models.append(('svm4', SVC(probability=True, kernel='poly', degree=4)))

models.append(('svm5', SVC(probability=True, kernel='poly', degree=5)))

# define the voting ensemble

ensemble = VotingClassifier(estimators=models, voting='soft')

return ensemble

# get a list of models to evaluate

def get\_models():

models = dict()

models['svm1'] = SVC(probability=True, kernel='poly', degree=1)

models['svm2'] = SVC(probability=True, kernel='poly', degree=2)

models['svm3'] = SVC(probability=True, kernel='poly', degree=3)

models['svm4'] = SVC(probability=True, kernel='poly', degree=4)

models['svm5'] = SVC(probability=True, kernel='poly', degree=5)

models['soft\_voting'] = get\_voting()

return models

# compare soft voting ensemble to standalone classifiers

from numpy import mean

from numpy import std

from sklearn.datasets import make\_classification

from sklearn.model\_selection import cross\_val\_score

from sklearn.model\_selection import RepeatedStratifiedKFold

from sklearn.svm import SVC

from sklearn.ensemble import VotingClassifier

from matplotlib import pyplot

# get the dataset

def get\_dataset():

X, y = make\_classification(n\_samples=1000, n\_features=20, n\_informative=15, n\_redundant=5, random\_state=2)

return X, y

# get a voting ensemble of models

def get\_voting():

# define the base models

models = list()

models.append(('svm1', SVC(probability=True, kernel='poly', degree=1)))

models.append(('svm2', SVC(probability=True, kernel='poly', degree=2)))

models.append(('svm3', SVC(probability=True, kernel='poly', degree=3)))

models.append(('svm4', SVC(probability=True, kernel='poly', degree=4)))

models.append(('svm5', SVC(probability=True, kernel='poly', degree=5)))

# define the voting ensemble

ensemble = VotingClassifier(estimators=models, voting='soft')

return ensemble

# get a list of models to evaluate

def get\_models():

models = dict()

models['svm1'] = SVC(probability=True, kernel='poly', degree=1)

models['svm2'] = SVC(probability=True, kernel='poly', degree=2)

models['svm3'] = SVC(probability=True, kernel='poly', degree=3)

models['svm4'] = SVC(probability=True, kernel='poly', degree=4)

models['svm5'] = SVC(probability=True, kernel='poly', degree=5)

models['soft\_voting'] = get\_voting()

return models

# evaluate a give model using cross-validation

def evaluate\_model(model, X, y):

cv = RepeatedStratifiedKFold(n\_splits=10, n\_repeats=3, random\_state=1)

scores = cross\_val\_score(model, X, y, scoring='accuracy', cv=cv, n\_jobs=-1, error\_score='raise')

return scores

# define dataset

X, y = get\_dataset()

# get the models to evaluate

models = get\_models()

# evaluate the models and store results

results, names = list(), list()

for name, model in models.items():

scores = evaluate\_model(model, X, y)

results.append(scores)

names.append(name)

print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))

# plot model performance for comparison

pyplot.boxplot(results, labels=names, showmeans=True)

pyplot.show()

# make a prediction with a soft voting ensemble

from sklearn.datasets import make\_classification

from sklearn.ensemble import VotingClassifier

from sklearn.svm import SVC

# define dataset

X, y = make\_classification(n\_samples=1000, n\_features=20, n\_informative=15, n\_redundant=5, random\_state=2)

# define the base models

models = list()

models.append(('svm1', SVC(probability=True, kernel='poly', degree=1)))

models.append(('svm2', SVC(probability=True, kernel='poly', degree=2)))

models.append(('svm3', SVC(probability=True, kernel='poly', degree=3)))

models.append(('svm4', SVC(probability=True, kernel='poly', degree=4)))

models.append(('svm5', SVC(probability=True, kernel='poly', degree=5)))

# define the soft voting ensemble

ensemble = VotingClassifier(estimators=models, voting='soft')

# fit the model on all available data

ensemble.fit(X, y)

# make a prediction for one example

data = [[5.88891819,2.64867662,-0.42728226,-1.24988856,-0.00822,-3.57895574,2.87938412,-1.55614691,-0.38168784,7.50285659,-1.16710354,-5.02492712,-0.46196105,-0.64539455,-1.71297469,0.25987852,-0.193401,-5.52022952,0.0364453,-1.960039]]

yhat = ensemble.predict(data)

print('Predicted Class: %d' % (yhat))

**=====================11 – 5 – VOTING\_REGRESSION ===================**

# test regression dataset

from sklearn.datasets import make\_regression

# define dataset

X, y = make\_regression(n\_samples=1000, n\_features=20, n\_informative=15, noise=0.1, random\_state=1)

# summarize the dataset

print(X.shape, y.shape)

# get a voting ensemble of models

def get\_voting():

# define the base models

models = list()

models.append(('cart1', DecisionTreeRegressor(max\_depth=1)))

models.append(('cart2', DecisionTreeRegressor(max\_depth=2)))

models.append(('cart3', DecisionTreeRegressor(max\_depth=3)))

models.append(('cart4', DecisionTreeRegressor(max\_depth=4)))

models.append(('cart5', DecisionTreeRegressor(max\_depth=5)))

# define the voting ensemble

ensemble = VotingRegressor(estimators=models)

return ensemble

# get a list of models to evaluate

def get\_models():

models = dict()

models['cart1'] = DecisionTreeRegressor(max\_depth=1)

models['cart2'] = DecisionTreeRegressor(max\_depth=2)

models['cart3'] = DecisionTreeRegressor(max\_depth=3)

models['cart4'] = DecisionTreeRegressor(max\_depth=4)

models['cart5'] = DecisionTreeRegressor(max\_depth=5)

models['voting'] = get\_voting()

return models

# compare voting ensemble to each standalone models for regression

from numpy import mean

from numpy import std

from sklearn.datasets import make\_regression

from sklearn.model\_selection import cross\_val\_score

from sklearn.model\_selection import RepeatedKFold

from sklearn.tree import DecisionTreeRegressor

from sklearn.ensemble import VotingRegressor

from matplotlib import pyplot

# get the dataset

def get\_dataset():

X, y = make\_regression(n\_samples=1000, n\_features=20, n\_informative=15, noise=0.1, random\_state=1)

return X, y

# get a voting ensemble of models

def get\_voting():

# define the base models

models = list()

models.append(('cart1', DecisionTreeRegressor(max\_depth=1)))

models.append(('cart2', DecisionTreeRegressor(max\_depth=2)))

models.append(('cart3', DecisionTreeRegressor(max\_depth=3)))

models.append(('cart4', DecisionTreeRegressor(max\_depth=4)))

models.append(('cart5', DecisionTreeRegressor(max\_depth=5)))

# define the voting ensemble

ensemble = VotingRegressor(estimators=models)

return ensemble

# get a list of models to evaluate

def get\_models():

models = dict()

models['cart1'] = DecisionTreeRegressor(max\_depth=1)

models['cart2'] = DecisionTreeRegressor(max\_depth=2)

models['cart3'] = DecisionTreeRegressor(max\_depth=3)

models['cart4'] = DecisionTreeRegressor(max\_depth=4)

models['cart5'] = DecisionTreeRegressor(max\_depth=5)

models['voting'] = get\_voting()

return models

# evaluate a give model using cross-validation

def evaluate\_model(model, X, y):

cv = RepeatedKFold(n\_splits=10, n\_repeats=3, random\_state=1)

scores = cross\_val\_score(model, X, y, scoring='neg\_mean\_absolute\_error', cv=cv, n\_jobs=-1, error\_score='raise')

return scores

# define dataset

X, y = get\_dataset()

# get the models to evaluate

models = get\_models()

# evaluate the models and store results

results, names = list(), list()

for name, model in models.items():

scores = evaluate\_model(model, X, y)

results.append(scores)

names.append(name)

print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))

# plot model performance for comparison

pyplot.boxplot(results, labels=names, showmeans=True)

pyplot.show()

# make a prediction with a voting ensemble

from sklearn.datasets import make\_regression

from sklearn.tree import DecisionTreeRegressor

from sklearn.ensemble import VotingRegressor

# define dataset

X, y = make\_regression(n\_samples=1000, n\_features=20, n\_informative=15, noise=0.1, random\_state=1)

# define the base models

models = list()

models.append(('cart1', DecisionTreeRegressor(max\_depth=1)))

models.append(('cart2', DecisionTreeRegressor(max\_depth=2)))

models.append(('cart3', DecisionTreeRegressor(max\_depth=3)))

models.append(('cart4', DecisionTreeRegressor(max\_depth=4)))

models.append(('cart5', DecisionTreeRegressor(max\_depth=5)))

# define the voting ensemble

ensemble = VotingRegressor(estimators=models)

# fit the model on all available data

ensemble.fit(X, y)

# make a prediction for one example

data = [[0.59332206,-0.56637507,1.34808718,-0.57054047,-0.72480487,1.05648449,0.77744852,0.07361796,0.88398267,2.02843157,1.01902732,0.11227799,0.94218853,0.26741783,0.91458143,-0.72759572,1.08842814,-0.61450942,-0.69387293,1.69169009]]

yhat = ensemble.predict(data)

print('Predicted Value: %.3f' % (yhat))

**=====================11 – 6 – GRADIENT\_BOOSTING ===================**

def gradient\_descent(gradient, start, learn\_rate, n\_iter):

vector = start

for \_ in range(n\_iter):

diff = -learn\_rate \* gradient(vector)

vector += diff

return vector

import numpy as np

def gradient\_descent(

gradient, start, learn\_rate, n\_iter=50, tolerance=1e-06

):

vector = start

for \_ in range(n\_iter):

diff = -learn\_rate \* gradient(vector)

if np.all(np.abs(diff) <= tolerance):

break

vector += diff

return vector

gradient\_descent(

... gradient=lambda v: 2 \* v, start=10.0, learn\_rate=0.2

... )

gradient\_descent(

... gradient=lambda v: 2 \* v, start=10.0, learn\_rate=0.8

... )

gradient\_descent(

... gradient=lambda v: 2 \* v, start=10.0, learn\_rate=0.005

... )

gradient\_descent(

... gradient=lambda v: 2 \* v, start=10.0, learn\_rate=0.005,

... n\_iter=100

... )

3.660323412732294

>>> gradient\_descent(

... gradient=lambda v: 2 \* v, start=10.0, learn\_rate=0.005,

... n\_iter=1000

... )

0.0004317124741065828

>>> gradient\_descent(

... gradient=lambda v: 2 \* v, start=10.0, learn\_rate=0.005,

... n\_iter=2000

... )

gradient\_descent(

... gradient=lambda v: 4 \* v\*\*3 - 10 \* v - 3, start=0,

... learn\_rate=0.2

... )

gradient\_descent(

... gradient=lambda v: 4 \* v\*\*3 - 10 \* v - 3, start=0,

... learn\_rate=0.1

... )

**===================12 -1 – Spyeder – Deployer – Salary guess ========**

**APP.py**

import numpy as np

from flask import Flask, request, jsonify, render\_template

import pickle

app = Flask(\_\_name\_\_)

model = pickle.load(open('model.pkl', 'rb'))

@app.route('/')

def home():

    return render\_template('index.html')

@app.route('/predict',methods=['POST'])

def predict():

    '''

    For rendering results on HTML GUI

    '''

    int\_features = [int(x) for x in request.form.values()]

    final\_features = [np.array(int\_features)]

    prediction = model.predict(final\_features)

    output = round(prediction[0], 2)

    return render\_template('index.html', prediction\_text='Employee Salary should be $ {}'.format(output))

@app.route('/predict\_api',methods=['POST'])

def predict\_api():

    '''

    For direct API calls trought request

    '''

    data = request.get\_json(force=True)

    prediction = model.predict([np.array(list(data.values()))])

    output = prediction[0]

    return jsonify(output)

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

    app.run(debug=True)

**model.py**

# Importing the libraries

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

import pickle

dataset = pd.read\_csv('hiring.csv')

dataset['experience'].fillna(0, inplace=True)

dataset['test\_score'].fillna(dataset['test\_score'].mean(), inplace=True)

X = dataset.iloc[:, :3]

#Converting words to integer values

def convert\_to\_int(word):

    word\_dict = {'one':1, 'two':2, 'three':3, 'four':4, 'five':5, 'six':6, 'seven':7, 'eight':8,

                'nine':9, 'ten':10, 'eleven':11, 'twelve':12, 'zero':0, 0: 0}

    return word\_dict[word]

X['experience'] = X['experience'].apply(lambda x : convert\_to\_int(x))

y = dataset.iloc[:, -1]

#Splitting Training and Test Set

#Since we have a very small dataset, we will train our model with all availabe data.

from sklearn.linear\_model import LinearRegression

regressor = LinearRegression()

#Fitting model with trainig data

regressor.fit(X, y)

# Saving model to disk

pickle.dump(regressor, open('model.pkl','wb'))

# Loading model to compare the results

model = pickle.load(open('model.pkl','rb'))

print(model.predict([[2, 9, 6]]))

**Request.py**

import requests

url = 'http://localhost:5000/predict\_api'

r = requests.post(url,json={'experience':2, 'test\_score':9, 'interview\_score':6})

print(r.json())

**INDEX.html**

<!DOCTYPE html>

<html >

<!--From https://codepen.io/frytyler/pen/EGdtg-->

<head>

  <meta charset="UTF-8">

  <title>ML API</title>

  <link href='https://fonts.googleapis.com/css?family=Pacifico' rel='stylesheet' type='text/css'>

<link href='https://fonts.googleapis.com/css?family=Arimo' rel='stylesheet' type='text/css'>

<link href='https://fonts.googleapis.com/css?family=Hind:300' rel='stylesheet' type='text/css'>

<link href='https://fonts.googleapis.com/css?family=Open+Sans+Condensed:300' rel='stylesheet' type='text/css'>

<link rel="stylesheet" href="{{ url\_for('static', filename='css/style.css') }}">

</head>

<body>

 <div class="login">

  <h1>Predict Salary Analysis</h1>

     <!-- Main Input For Receiving Query to our ML -->

    <form action="{{ url\_for('predict')}}"method="post">

      <input type="text" name="experience" placeholder="Experience" required="required" />

        <input type="text" name="test\_score" placeholder="Test Score" required="required" />

    <input type="text" name="interview\_score" placeholder="Interview Score" required="required" />

        <button type="submit" class="btn btn-primary btn-block btn-large">Predict</button>

    </form>

   <br>

   <br>

   {{ prediction\_text }}

 </div>

</body>

</html>

**Style.css**

@import url(https://fonts.googleapis.com/css?family=Open+Sans);

.btn { display: inline-block; \*display: inline; \*zoom: 1; padding: 4px 10px 4px; margin-bottom: 0; font-size: 13px; line-height: 18px; color: #333333; text-align: center;text-shadow: 0 1px 1px rgba(255, 255, 255, 0.75); vertical-align: middle; background-color: #f5f5f5; background-image: -moz-linear-gradient(top, #ffffff, #e6e6e6); background-image: -ms-linear-gradient(top, #ffffff, #e6e6e6); background-image: -webkit-gradient(linear, 0 0, 0 100%, from(#ffffff), to(#e6e6e6)); background-image: -webkit-linear-gradient(top, #ffffff, #e6e6e6); background-image: -o-linear-gradient(top, #ffffff, #e6e6e6); background-image: linear-gradient(top, #ffffff, #e6e6e6); background-repeat: repeat-x; filter: progid:dximagetransform.microsoft.gradient(startColorstr=#ffffff, endColorstr=#e6e6e6, GradientType=0); border-color: #e6e6e6 #e6e6e6 #e6e6e6; border-color: rgba(0, 0, 0, 0.1) rgba(0, 0, 0, 0.1) rgba(0, 0, 0, 0.25); border: 1px solid #e6e6e6; -webkit-border-radius: 4px; -moz-border-radius: 4px; border-radius: 4px; -webkit-box-shadow: inset 0 1px 0 rgba(255, 255, 255, 0.2), 0 1px 2px rgba(0, 0, 0, 0.05); -moz-box-shadow: inset 0 1px 0 rgba(255, 255, 255, 0.2), 0 1px 2px rgba(0, 0, 0, 0.05); box-shadow: inset 0 1px 0 rgba(255, 255, 255, 0.2), 0 1px 2px rgba(0, 0, 0, 0.05); cursor: pointer; \*margin-left: .3em; }

.btn:hover, .btn:active, .btn.active, .btn.disabled, .btn[disabled] { background-color: #e6e6e6; }

.btn-large { padding: 9px 14px; font-size: 15px; line-height: normal; -webkit-border-radius: 5px; -moz-border-radius: 5px; border-radius: 5px; }

.btn:hover { color: #333333; text-decoration: none; background-color: #e6e6e6; background-position: 0 -15px; -webkit-transition: background-position 0.1s linear; -moz-transition: background-position 0.1s linear; -ms-transition: background-position 0.1s linear; -o-transition: background-position 0.1s linear; transition: background-position 0.1s linear; }

.btn-primary, .btn-primary:hover { text-shadow: 0 -1px 0 rgba(0, 0, 0, 0.25); color: #ffffff; }

.btn-primary.active { color: rgba(255, 255, 255, 0.75); }

.btn-primary { background-color: #4a77d4; background-image: -moz-linear-gradient(top, #6eb6de, #4a77d4); background-image: -ms-linear-gradient(top, #6eb6de, #4a77d4); background-image: -webkit-gradient(linear, 0 0, 0 100%, from(#6eb6de), to(#4a77d4)); background-image: -webkit-linear-gradient(top, #6eb6de, #4a77d4); background-image: -o-linear-gradient(top, #6eb6de, #4a77d4); background-image: linear-gradient(top, #6eb6de, #4a77d4); background-repeat: repeat-x; filter: progid:dximagetransform.microsoft.gradient(startColorstr=#6eb6de, endColorstr=#4a77d4, GradientType=0);  border: 1px solid #3762bc; text-shadow: 1px 1px 1px rgba(0,0,0,0.4); box-shadow: inset 0 1px 0 rgba(255, 255, 255, 0.2), 0 1px 2px rgba(0, 0, 0, 0.5); }

.btn-primary:hover, .btn-primary:active, .btn-primary.active, .btn-primary.disabled, .btn-primary[disabled] { filter: none; background-color: #4a77d4; }

.btn-block { width: 100%; display:block; }

\* { -webkit-box-sizing:border-box; -moz-box-sizing:border-box; -ms-box-sizing:border-box; -o-box-sizing:border-box; box-sizing:border-box; }

html { width: 100%; height:100%; overflow:hidden; }

body {

    width: 100%;

    height:100%;

    font-family: 'Open Sans', sans-serif;

    background: #092756;

    color: #fff;

    font-size: 18px;

    text-align:center;

    letter-spacing:1.2px;

    background: -moz-radial-gradient(0% 100%, ellipse cover, rgba(104,128,138,.4) 10%,rgba(138,114,76,0) 40%),-moz-linear-gradient(top,  rgba(57,173,219,.25) 0%, rgba(42,60,87,.4) 100%), -moz-linear-gradient(-45deg,  #670d10 0%, #092756 100%);

    background: -webkit-radial-gradient(0% 100%, ellipse cover, rgba(104,128,138,.4) 10%,rgba(138,114,76,0) 40%), -webkit-linear-gradient(top,  rgba(57,173,219,.25) 0%,rgba(42,60,87,.4) 100%), -webkit-linear-gradient(-45deg,  #670d10 0%,#092756 100%);

    background: -o-radial-gradient(0% 100%, ellipse cover, rgba(104,128,138,.4) 10%,rgba(138,114,76,0) 40%), -o-linear-gradient(top,  rgba(57,173,219,.25) 0%,rgba(42,60,87,.4) 100%), -o-linear-gradient(-45deg,  #670d10 0%,#092756 100%);

    background: -ms-radial-gradient(0% 100%, ellipse cover, rgba(104,128,138,.4) 10%,rgba(138,114,76,0) 40%), -ms-linear-gradient(top,  rgba(57,173,219,.25) 0%,rgba(42,60,87,.4) 100%), -ms-linear-gradient(-45deg,  #670d10 0%,#092756 100%);

    background: -webkit-radial-gradient(0% 100%, ellipse cover, rgba(104,128,138,.4) 10%,rgba(138,114,76,0) 40%), linear-gradient(to bottom,  rgba(57,173,219,.25) 0%,rgba(42,60,87,.4) 100%), linear-gradient(135deg,  #670d10 0%,#092756 100%);

    filter: progid:DXImageTransform.Microsoft.gradient( startColorstr='#3E1D6D', endColorstr='#092756',GradientType=1 );

}

.login {

    position: absolute;

    top: 40%;

    left: 50%;

    margin: -150px 0 0 -150px;

    width:400px;

    height:400px;

}

.login h1 { color: #fff; text-shadow: 0 0 10px rgba(0,0,0,0.3); letter-spacing:1px; text-align:center; }

input {

    width: 100%;

    margin-bottom: 10px;

    background: rgba(0,0,0,0.3);

    border: none;

    outline: none;

    padding: 10px;

    font-size: 13px;

    color: #fff;

    text-shadow: 1px 1px 1px rgba(0,0,0,0.3);

    border: 1px solid rgba(0,0,0,0.3);

    border-radius: 4px;

    box-shadow: inset 0 -5px 45px rgba(100,100,100,0.2), 0 1px 1px rgba(255,255,255,0.2);

    -webkit-transition: box-shadow .5s ease;

    -moz-transition: box-shadow .5s ease;

    -o-transition: box-shadow .5s ease;

    -ms-transition: box-shadow .5s ease;

    transition: box-shadow .5s ease;

}

input:focus { box-shadow: inset 0 -5px 45px rgba(100,100,100,0.4), 0 1px 1px rgba(255,255,255,0.2); }

**================================12 - 2 – spyder IR project ====================**

**APP.PY**

import numpy as np

from flask import Flask, request, jsonify, render\_template

import pickle

model = pickle.load(open('model.pkl', 'rb'))

app = Flask(\_\_name\_\_)

@app.route('/')

def home():

      return render\_template('index.html')

@app.route('/predict',methods=['POST'])

def predict():

    '''

    For rendering results on HTML GUI

    '''

    int\_features = [float(x) for x in request.form.values()]

    final\_features = [np.array(int\_features)]

    prediction = model.predict(final\_features)

    output =prediction[0]

    return render\_template('index.html', prediction\_text='The Flower is {}'.format(output))

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

    app.run(debug=True)

**MODEl.PY**

import pandas as pd

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import classification\_report

from sklearn.metrics import accuracy\_score

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

import pickle

data=pd.read\_csv('iris.csv')

# X = feature values, all the columns except the last column

X = data.iloc[:, :-1]

# y = target values, last column of the data frame

y = data.iloc[:, -1]

#Split the data into 80% training and 20% testing

x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

#Train the model

model = LogisticRegression()

model.fit(x\_train, y\_train) #Training the model

#Test the model

predictions = model.predict(x\_test)

print( classification\_report(y\_test, predictions) )

print( accuracy\_score(y\_test, predictions))

pickle.dump(model,open('model.pkl','wb'))

p=model.predict([[5.1,3.5,1.4,0.2]])

print(p[0])

**index.html**

<!DOCTYPE html>

<html >

<head>

<meta charset="UTF-8">

<title>ML API</title>

</head>

<body>

<div class="login">

<h1>Predict type of flower</h1>

<!-- Main Input For Receiving Query to our ML -->

<form action="{{ url\_for('predict')}}"method="post">

<input type="text" name="SepalLength" placeholder="SepalLength" required="required" />

<input type="text" name="SepalWidth" placeholder="SepalWidth" required="required" />

<input type="text" name="PetalLength" placeholder="PetalLength" required="required" />

<input type="text" name="PetalWidth" placeholder="PetalWidth" required="required" />

<button type="submit" class="btn btn-primary btn-block btn-large">Predict</button>

</form>

<br>

<br>

{{ prediction\_text }}

</div>

</body>

</html>