F.Y.M.C.A.
(TWO YEARS PATTERN)
SEMESTER - II (CBCS)

ARTIFICIAL INTELLIGENCE & MACHINE LEARNING LAB

SUBJECT CODE : MCAL21

Prof. Suhas Pednekar
Vice Chancellor
University of Mumbai, Mumbai.

Prof. Ravindra D. Kulkarni
Pro Vice-Chancellor,
University of Mumbai.

Prof. Prakash Mahanwar
Director
IDOL, University of Mumbai.

Programme Co-ordinator : **Prof. Mandar Bhanushe**
Head, Faculty of Science & Technology,
IDOL, University of Mumbai - 400 098.

Course Co-ordinator : **Ms. Shyam Mohan T**
Dep. of MCA IDOL,
University of Mumbai, Mumbai.

Course Writers : **Dr. Ghayathri J**
Associate Professor Kongu Arts and Science College (Autonomous)

: **Ms. Jayalalita**
Assistant Professor
NES Ratnam college of arts, science and commerce

: **Dr. R Jayakarthik M.Sc., M.Phil., Ph.D**
Associate Professor
Dept of Computer Science,Saveetha College of Liberal Arts & Sciences,
SIMATS Deemed to be University,
Saveetha Nagar,
Thandalam, Chennai - 602 105
Mobile : +91 7598209965

: **Mr. Sandeep Kamble**
Assistant Professor Cosmopolitan's Valia College

: **Ms. Hema Darne**
Assistant Professor Dr. Moonje Institute of Management
and Computer Studies, Nashik

: **Mr. Ather Iqbal**
Assistant Professor Vidyabharati Mahavidyalaya, Amravat

: **Dr. D.S.Rao**
Professor (CSE) & Associate Dean (Student Affairs)
Koneru Lakshmaiah Education Foundation. KL H (Deemed to be University)
Hyderabad Campus Hyderabad. Telangana

April 2022, Print I

Published by
Director
Institute of Distance and Open learning , University of Mumbai,Vidyanagari, Mumbai - 400 098.

DTP COMPOSED AND PRINTED BY
Mumbai University Press
Vidyanagari, Santacruz (E), Mumbai - 400098.

CONTENT

*****

SYLLABUS

Course Code	Course Name
MCAL21	Artificial Intelligence & Machine Learning Lab

Module	Detail Content	Hrs
1	**Logic programming with Prolog** To specify relationships among objects and properties of objects, problem solving. **Self Learning Topic:-** Define rules defining implicit relationships between objects	2
2	**Introduction to Python Programming:** Learn the different libraries - NumPy, Pandas, SciPy, Matplotlib, Scikit Learn **Self Learning Topic:-** Milk, Shogun	4
3	**Supervised Learning:** Linear Regression predicts a real-valued output based on an input value, Logistic regression- the notion of classification, the cost function for logistic regression, and the application of logistic regression, KNN- classification. **Self Learning Topic:-** Evaluation metrics like MSE, Accuracy, Confusion Matrix, Precision, Recall, ROC curve	4
4	**Dimensionality Reduction:** Features Extraction, Feature selection, Normalization, Transformation, Principal Components Analysis- visualizations of complex datasets. **Self Learning Topic:-** LDA (Linear Discriminant Analysis)	4
5	**Unsupervised Learning:** K-Means clustering algorithm, K-medoid clustering algorithm. **Self Learning Topic:-** Other Clustering Algorithms	2

6	**Classifying data using Support Vector Machines (SVMs):** SVM-RBF kernels. **Self Learning Topic**:-SVM-Kernels-Polynomial kernel	2
7	**Bagging Algorithm:**Decision Tree,different ensemble techniques like bagging, boosting, stacking and voting, Random Forest- bagging, Attribute bagging and voting for class selection. **Self Learning Topic:-** Extra Trees	4
8	**Boosting Algorithms:** AdaBoost, Stochastic Gradient Boosting, Voting Ensemble. **Self Learning Topic**:- AdaBoost as a Forward Stage wise Additive Model	2
9	**Deployment of Machine Learning Models:** simple Web API. **Self Learning Topic**:- Python Flask library	2

*****

UNIT I

1

ARTIFICIAL INTELLIGENCE LAB

Unit Structure

1.0 OBJECTIVES

After reading this chapter students will be able to:

- Explain the structure of PROLOG
- Describe the logic programming of PROLOG
- Have the knowledge about the objects and its working principles in PROLOG
- write the applications and problems of Artificial Intelligence programs using PROLOG

1.1 INTRODUCTION

PROLOG: Programming Logic language was designed in the 1970s by Alain Colmerauer and a team of researchers

It was possible to use logic to represent knowledge and to write programs.

It uses a subset of predicate logic and draws its structure from theoretical works of earlier logicians such as Herbrand (1930) and Robinson (1965) on the automation of theorem proving.

PROLOG supports:

- Natural Language Understanding

- Formal logic and associated forms of programming
- Reasoning modeling
- Database programming
- Expert System Development
- Real time AI programs

1.2 LOGIC PROGRAMMING WITH PROLOG

PROLOG programs are often described as *declarative*, although they unavoidably also have a procedural element. Programs are based on the techniques developed by logicians to form valid conclusions from available evidence. There are only two components to any program: facts and rules. The PROLOG system reads in the program and simply stores it. The user gives the queries which can be answered by the system using the facts and rules available to it. A simple example, is given below to illustrate the same.

```
dog (puppy).
dog (kutty).
dog (jimmy).
cat (valu).
cat (miaw).
cat (mouse).

    animal(Y):-dog(Y).
```

Output:

:- dog(puppy).

Yes

:- cat(kar).

No

PROLOG program, rules and facts, and also the use of queries that make PROLOG search through its facts and rules to work out the answer. Determining that puppy is an animal involves a very simple form of logical reasoning:

Given that any Y is an animal if it is a dog and Puppy is a dog Deduce Puppy must be an animal

1.3 RELATIONSHIPS AMONG OBJECTS AND PROPERTIES OF OBJECTS

The relationship between the objects and the particular relationship among the objects are explained through the following example.

Each family has three components: husband, wife and children are objects of the family. As the number of children varies from family to family the children are represented by a list that is capable of accommodating any number of items. Each person is, in turn, represented by a structure of four components: name or it specifies the working organization and salary. The family of can be stored in the database by the clause

family(

person(tom, fox, date(7,may,1950), works(bbc,15200)),

person(ann, fox, dat{9,may, 195 1), unemployed),

[person(pat, fox, date(5,may,1973), unemployed),

person(jim, fox, date(S,may,1973), unemployed)]).

This program shall be extended as adding the information on the gender of the people that occur in the parent relation. This can be done by simply adding the following facts to our program:

female(pam).

male(tom).

male(bob).

female(liz).

female(pat).

female(ann).

male(jim).

The relations introduced here are male and female. These relations are unary relations.

A binary relation like parent defines a relation between pairs of objects; on the other hand, unary relations can be used to declare simple yes/no properties of objects. The first unary clause above can be read: Pam is a female. The same information declared in the two unary relations with one binary relation, sex, instead. An alternative code snippet of program is :

gender(pam, feminine).

gender(tom, masculine).

gender(bob, masculine).

The offspring relation is as the inverse of the parent relation. We could define offspring in a similar way as the parent relation; that is, by simply providing a list of simple facts about the offspring relation, each fact mentioning one pair of people such that one is an offspring of the other. For example:

offspring(liz, tom).

However, the offspring relation can be defined much more elegantly by making use of the fact that it is the inverse of parent, and that parent has already been defined. This alternative way can be based on the following logical statement:

For all X and Y,

Y is an offspring of X if

X is a parent of Y.

This formulation is already close to the formalism of PROLOG. The corresponding PROLOG clause which has the same meaning is:

offspring(Y, X) :- parent(X, Y).

This clause can also be read as:

For all X and Y,

if X is a parent of Y then

Y is an offspring of X.

PROLOG clauses : Rules

offspring(Y, X) :- parent(X, Y).

Difference between facts and rules: A fact is something that is always, unconditionally, true. On the other hand, rules specify things that may be true if some condition is satisfied. Therefore we say that rules have:

A condition part and a conclusion part

The conclusion part is also called the head of a clause and the condition part the body of a clause. For example:

offspring(y, X) :- parent(X, y).

head body

If the condition parent(X, Y) is true then a logical consequence of this is offspring(Y, X).

How rules are actually used by PROLOG is illustrated as

:- offspring(liz, tom).

1.4 PROBLEM SOLVING

1.4.1 Water jug problem:

Problem Statement:

In the water jug problem in Artificial Intelligence, we are provided with two jugs: one having the capacity to hold 3 gallons of water and the other has the capacity to hold 4 gallons of water.

There is no other measuring equipment available and the jugs also do not have any kind of marking on them. So, the agent's task here is to fill the 4-gallon jug with 2 gallons of water by using only these two jugs and no other material. Initially, both our jugs are empty.

So, to solve this problem, following set of rules were proposed:

Production rules for solving the water jug problem

Here, let x denote the 4-gallon jug and y denote the 3-gallon jug.

S.No. Initial State Condition Final state Description of action taken

1. (x,y) If x<4 (4,y) Fill the 4 gallon jug completely
2. (x,y) if y<3 (x,3) Fill the 3 gallon jug completely
3. (x,y) If x>0 (x-d,y) Pour some part from the 4 gallon jug
4. (x,y) If y>0 (x,y-d) Pour some part from the 3 gallon jug
5. (x,y) If x>0 (0,y) Empty the 4 gallon jug
6. (x,y) If y>0 (x,0) Empty the 3 gallon jug
7. (x,y) If (x+y)<7 (4, y-[4-x]) Pour some water from the 3 gallon jug to fill the four gallon jug
8. (x,y) If (x+y)<7 (x-[3-y],y) Pour some water from the 4 gallon jug to fill the 3 gallon jug.
9. (x,y) If (x+y)<4 (x+y,0) Pour all water from 3 gallon jug to the 4 gallon jug
10. (x,y) if (x+y)<3 (0, x+y) Pour all water from the 4 gallon jug to the 3 gallon jug

To solve the water jug problem in a minimum number of moves, following set of rules in the given sequence should be performed:

Solution of water jug problem according to the production rules:

S.No.	4 gallon jug contents	3 gallon jug contents	Rule followed
1.	0 gallon	0 gallon	Initial state
2.	0 gallon	3 gallons	Rule no.2
3.	3 gallons	0 gallon	Rule no. 9
4.	3 gallons	3 gallons	Rule no. 2
5.	4 gallons	2 gallons	Rule no. 7
6.	0 gallon	2 gallons	Rule no. 5
7.	2 gallons	0 gallon	Rule no. 9

On reaching the 7th attempt, the goal state is reached.

Aim: Writing clauses in PROLOG to solve water jug problem

Software used: SWI-PROLOG

Program Listing:

```
database
   visited_state(integer,integer)
predicates
   state(integer,integer)
clauses
   state(2,0).
   state(X,Y):-
       X < 4,
       not(visited_state(4,Y)),
       assert(visited_state(X,Y)),
       write("Fill the 4-Gallon Jug: (",X,",",Y,") --> (", 4,",",Y,")\n"),
       state(4,Y).
   state(X,Y):- Y < 3,
         not(visited_state(X,3)),
       assert(visited_state(X,Y)),
       write("Fill the 3-Gallon Jug: (", X,",",Y,") --> (", X,",",3,")\n"),
       state(X,3).
   state(X,Y):- X > 0,
       not(visited_state(0,Y)),
```

```
    assert(visited_state(X,Y)),
    write("Empty the 4-Gallon jug on ground: (", X,",",Y,") -->
      (",0,",",Y,")\n"),
    state(0,Y).
state(X,Y):- Y > 0,
    not(visited_state(X,0)),
    assert(visited_state(X,0)),
    write("Empty the 3-Gallon jug on ground: (", X,",",Y,") -->
      (",X,",",0,")\n"),
    state(X,0).
state(X,Y):- X + Y >= 4,
    Y > 0,
    NEW_Y = Y - (4 - X),
    not(visited_state(4,NEW_Y)),
    assert(visited_state(X,Y)),
    write("Pour water from 3-Gallon jug to 4-gallon until it is full:
      (",X,",",Y,") --> (", 4,",",NEW_Y,")\n"),
    state(4,NEW_Y).
state(X,Y):- X + Y >=3,
    X > 0,
    NEW_X = X - (3 - Y),
    not(visited_state(X,3)),
    assert(visited_state(X,Y)),
    write("Pour water from 4-Gallon jug to 3-gallon until it is full:
      (",X,",",Y,") --> (", NEW_X,",",3,")\n"),
    state(NEW_X,3).
state(X,Y):- X + Y <=4,
      Y > 0,
      NEW_X = X + Y,
      not(visited_state(NEW_X,0)),
      assert(visited_state(X,Y)),
```

```
        write("Pour all the water from 3-Gallon jug to 4-gallon: (",X,",",Y,") --> (", NEW_X,",",0,")\n"),

        state(NEW_X,0).

state(X,Y):- X+Y<=3,

        X > 0,

        NEW_Y = X + Y,

        not(visited_state(0,NEW_Y)),

        assert(visited_state(X,Y)),

        write("Pour all the water from 4-Gallon jug to 3-gallon: (",X,",",Y,") --> (", 0,",",NEW_Y,")\n"),

        state(0,NEW_Y).

state(0,2):- not(visited_state(2,0)),

    assert(visited_state(0,2)),

    write("Pour 2 gallons from 3-Gallon jug to 4-gallon: (", 0,",",2,") --> (", 2,",",0,")\n"),

    state(2,0).

state(2,Y):- not(visited_state(0,Y)),

        assert(visited_state(2,Y)),

        write("Empty 2 gallons from 4-Gallon jug on the ground: (",2,",",Y,") --> (", 0,",",Y,")\n"),

        state(0,Y).
```

goal:-

```
makewindow(1,2,3,"4-3 Water Jug Problem",0,0,25,80),

state(0,0).
```

1.4.2 Tic-Tac.Toe Problem:

Aim: Tic-Tac-Toe using A* algorithm.

Theory: A board game (such as tic-tac-toe) is usually programmed as a state machine. Looking on the current-state and therefore the player's move, the game goes into the next-state.

tit-tat-toe (or Noughts and crosses, Xs and Os) could be a paper and pencil for 2 players, X and O, who take turns marking the areas in an exceedingly 3×3 grid.

The player who succeeds in putting 3 individual marks in an exceedingly horizontal, vertical or diagonal row wins the game. Players shortly discover that best play from each party ends up in a draw.

The game is generalized to an m,n,k-game during which 2 players alternate putting stones of their own colour on an m×n board, with the goal of obtaining k of their own colour in a row. Tit-Tat-Toe is the (3,3,3)-game.

```
/*A Tic-Tac-Toe program in PROLOG. */

/*Predicates that define the winning conditions:*/

win(Board, Player) :- rowwin(Board, Player).
win(Board, Player) :- colwin(Board, Player).
win(Board, Player) :- diagwin(Board, Player).

rowwin(Board, Player) :- Board = [Player,Player,Player,_,_,_,_,_,_].
rowwin(Board, Player) :- Board = [_,_,_,Player,Player,Player,_,_,_].
rowwin(Board, Player) :- Board = [_,_,_,_,_,_,Player,Player,Player].

colwin(Board, Player) :- Board = [Player,_,_,Player,_,_,Player,_,_].
colwin(Board, Player) :- Board = [_,Player,_,_,Player,_,_,Player,_].
colwin(Board, Player) :- Board = [_,_,Player,_,_,Player,_,_,Player].

diagwin(Board, Player) :- Board = [Player,_,_,_,Player,_,_,_,Player].
diagwin(Board, Player) :- Board = [_,_,Player,_,Player,_,Player,_,_].

/*Helping predicate for alternating play in a "self" game: */

other(x,o).
other(o,x).

game(Board, Player):- win(Board, Player), !, write([player, Player, wins]).

game(Board, Player):-
other(Player,Otherplayer),
move(Board,Player,Newboard),
!,
display(Newboard),
game(Newboard,Otherplayer).

move([b,B,C,D,E,F,G,H,I], Player, [Player,B,C,D,E,F,G,H,I]).
move([A,b,C,D,E,F,G,H,I], Player, [A,Player,C,D,E,F,G,H,I]).
move([A,B,b,D,E,F,G,H,I], Player, [A,B,Player,D,E,F,G,H,I]).
move([A,B,C,b,E,F,G,H,I], Player, [A,B,C,Player,E,F,G,H,I]).
move([A,B,C,D,b,F,G,H,I], Player, [A,B,C,D,Player,F,G,H,I]).
move([A,B,C,D,E,b,G,H,I], Player, [A,B,C,D,E,Player,G,H,I]).
```

```
move([A,B,C,D,E,F,b,H,I], Player, [A,B,C,D,E,F,Player,H,I]).
move([A,B,C,D,E,F,G,b,I], Player, [A,B,C,D,E,F,G,Player,I]).
move([A,B,C,D,E,F,G,H,b], Player, [A,B,C,D,E,F,G,H,Player]).

display([A,B,C,D,E,F,G,H,I]):-
 write([A,B,C]),nl,write([D,E,F]),nl,
 write([G,H,I]),nl,nl.

selfgame :- game([b,b,b,b,b,b,b,b,b],x).

/* Predicates to support playing a game with the user:*/

x_can_win_in_one(Board) :- move(Board, x, Newboard), win(Newboard,
x).

/*The predicate orespond generates the computer's (playing o) response from the
current Board. */

orespond(Board,Newboard):-
          move(Board, o, Newboard),
          win(Newboard, o),
          !.

orespond(Board,Newboard) :-
        move(Board, o, Newboard),
        not(x_can_win_in_one(Newboard)).

orespond(Board,Newboard) :-

        move(Board, o, Newboard).

orespond(Board,Newboard) :-
        not(member(b,Board)),
         !,
  write('Cats game!'), nl,
  Newboard = Board.

/* Translation  from an integer description  of x's move to a board
transformation.*/

xmove([b,B,C,D,E,F,G,H,I], 1, [x,B,C,D,E,F,G,H,I]).
xmove([A,b,C,D,E,F,G,H,I], 2, [A,x,C,D,E,F,G,H,I]).
xmove([A,B,b,D,E,F,G,H,I], 3, [A,B,x,D,E,F,G,H,I]).
xmove([A,B,C,b,E,F,G,H,I], 4, [A,B,C,x,E,F,G,H,I]).
xmove([A,B,C,D,b,F,G,H,I], 5, [A,B,C,D,x,F,G,H,I]).
xmove([A,B,C,D,E,b,G,H,I], 6, [A,B,C,D,E,x,G,H,I]).
```

```
xmove([A,B,C,D,E,F,b,H,I], 7, [A,B,C,D,E,F,x,H,I]).
xmove([A,B,C,D,E,F,G,b,I], 8, [A,B,C,D,E,F,G,x,I]).
xmove([A,B,C,D,E,F,G,H,b], 9, [A,B,C,D,E,F,G,H,x]).
xmove(Board, N, Board) :- write('Illegal move.'), nl.

% The 0-place predicate playo starts a game with the user.

playo :- explain, playfrom([b,b,b,b,b,b,b,b,b]).
```

explain :-

```
  write('You play X by entering integer positions followed by a period.'),
  nl,
  display([1,2,3,4,5,6,7,8,9]).

playfrom(Board) :- win(Board, x), write('You win!').
playfrom(Board) :- win(Board, o), write('I win!').
playfrom(Board) :- read(N),
      xmove(Board, N, Newboard),
      display(Newboard),
      orespond(Newboard, Newnewboard),
      display(Newnewboard),
      playfrom(Newnewboard).
```

1.4.3 8-Puzzle Problem:

/* This predicate initialises the problem states. The first argument of solve/3 is the initial state, the 2nd the goal state, and the third the plan that will be produced. */

test(Plan):-

```
   write('Initial state:'),nl,
   Init= [at(tile4,1), at(tile3,2), at(tile8,3), at(empty,4), at(tile2,5), at(tile6,6), at(tile5,7), at(tile1,8), at(tile7,9)],
   write_sol(Init),
   Goal= [at(tile1,1), at(tile2,2), at(tile3,3), at(tile4,4), at(empty,5), at(tile5,6), at(tile6,7), at(tile7,8), at(tile8,9)],
   nl,write('Goal state:'),nl,
   write(Goal),nl,nl,
   solve(Init,Goal,Plan).
```

```
solve(State, Goal, Plan):-
    solve(State, Goal, [], Plan).

/*Determines whether Current and Destination tiles are a valid move.  */

is_movable(X1,Y1) :- (1 is X1 - Y1) ; (-1 is X1 - Y1) ; (3 is X1 - Y1) ; (-3
is X1 - Y1).

/*This predicate produces the plan. Once the Goal list is a subset of the
current State the plan is complete and it is written to the screen using
write_sol */

solve(State, Goal, Plan, Plan):-
    is_subset(Goal, State), nl,
    write_sol(Plan).

solve(State, Goal, Sofar, Plan):-
    act(Action, Preconditions, Delete, Add),
    is_subset(Preconditions, State),
    \+ member(Action, Sofar),
    delete_list(Delete, State, Remainder),
    append(Add, Remainder, NewState),
    solve(NewState, Goal, [Action|Sofar], Plan).

act(move(X,Y,Z),
    [at(X,Y), at(empty,Z), is_movable(Y,Z)],
    [at(X,Y), at(empty,Z)],
    [at(X,Z), at(empty,Y)]).

/*Check is first list is a subset of the second */

is_subset([H|T], Set):-
    member(H, Set),
    is_subset(T, Set).
is_subset([], _).

/* Remove all elements of 1st list from second to create third. */

delete_list([H|T], Curstate, Newstate):-
    remove(H, Curstate, Remainder),
    delete_list(T, Remainder, Newstate).
delete_list([], Curstate, Curstate).

remove(X, [X|T], T).
```

```
remove(X, [H|T], [H|R]):-
   remove(X, T, R).

write_sol([]).
write_sol([H|T]):-
   write_sol(T),
   write(H), nl.

append([H|T], L1, [H|L2]):-
   append(T, L1, L2).
append([], L, L).

member(X, [X|_]).
member(X, [_|T]):-
   member(X, T).
```

1.5 SUMMARY

This chapter explains how prolog is used in the logical programs. Different applications like water jug problem, tic-tac-toe and decision making justification problems are described.

1.6 UNIT END EXERCISES

1. Write a PROLOG program to prove a person as a human
2. Explain the object and property relations
3. Write Towers of Hanoi program to apply PROLOG concept

1.7 REFERENCES

1. Logic Programming with Prolog, Max Bramer, Springer
2. Prolog Programming for Artificial Intelligence, E. Kardelj University . J. Stefan Institute

1.8 BIBLIOGRAPHY

1. https://www.cse.iitd.ac.in/~mcs052942/ai/print/13.txt
2. https://github.com/

*****

UNIT II

2

INTRODUCTION TO PYTHON PROGRAMMING: LEARN THE DIFFERENT LIBRARIES

Unit Structure

2.1 NUMPY

- Python library is nothing but a ready made moule.
- This library can be used whenever we want.
- If we are writing a code and if a particular requirement arises then instead of sitting and writing the whole code we can just use the ready made code available in the library.
- Thus by using the library our time is getting saved in a very wonderful manner.
- We can relate the Python library with the real world book library too. So if you imagine a book library it has a whole set of books with it. We can choose the book according to our requirements. Similarly in the python library we can choose a particular set of code which is needed.
- The extension of library files are ".dll"
- Full form of dll is Dynamic Load Libraries
- So whenever we add a library in our program during the execution phase it searches it and loads the particular module which is needed.
- Now in this module we are studying about numpy which is one of the libraries in python.
- NumPy stands for Numerical Python.
- It is one of the most widely used libray.

- As it contains the code related to numerical details it is most popular around data science and machine learning as both these fields need a lot of numerical logic getting applied in it.
- It is used whenever the situation in coding arises in working with an array.
- It does have methods that is made up for algebra related logics.
- This Numpy was made in the year 2005
- Example:

Lets try to insert array using numpy:

```
import numpy as ab
ar= ab.array(([1, 2, 3, 4, 5])
print(ar)
print(type(ar))
```

Output:

```
[1 2 3 4 5]
```

- In the above example in the first line we have imported the library by typing numpy.
- We have given our library a name called as ab, so now in the program whenever there is a requirement of numpy we just need to type ab.
- Then we created the variable called ar then we added array data inside the same
- Then we printed it.
- So output is printing the array data that has been inserted.

```
# The standard way to import NumPy:
import numpy as np
# Create a 2-D array, set every second element in
# some rows and find max per row:
x = np.arange(15, dtype=np.int64).reshape(3, 5)
x[1:, ::2] = -99
x
# array([[  0,   1,   2,   3,   4],
#        [-99,   6, -99,   8, -99],
```

```
#       [-99,  11, -99,  13, -99]])

x.max(axis=1)

# array([ 4,  8, 13])

# Generate normally distributed random numbers:

rng = np.random.default_rng()

samples = rng.normal(size=2500)

samples
```

Output

```
array([ 0.38054775, -0.06020411,  0.07380668, ...,  1.07546484,

    -0.20855135,  0.09773109])
```

2.2 PANDAS

- The main role of the pandas library is to analyze the data.
- It is open source in nature
- It is used in relational data
- On the top of Numpy library Pandas library is present.
- It is very quick in nature.
- It was made in the year 2008
- It is very efficient in datas.
- When it comes to pandas it is not necessary that the data should or should belong to a kind of category but instead it allows many.
- By using pandas you can reshape, analyze, and change your data very easily.
- Pandas supports two data structures:

1. Series:

- It is an array.
- It can hold any kind of data types like integer, float, character etc.
- It points to the column.
- Example1 : In the below example each column i.e name and roll no points to series. It is written in the following manner in the code:
- ab=se.Series(df [‘Name’])

- ab=se.Series(df [‘Roll no’])

Name	Roll no
Madhusri	01
Srivatsan	22
Anuradha	6
Balaguru	55

- Example 2:

```
import pandas as ab
import numpy as sj
 # Creating empty series
ser = sj.Series()
  print(ser)
# simple array
data = sj.array(['g', 'e', 'e', 'k', 's'])
ser = ab.Series(data)
print(ser)
```

- In the above example two libraries have been imported and are used namely numpy and pandas.
- The library Pandas is getting represented by ab and similarly the library numpy is getting represented by sj
- Then series are created by calling it, so an empty series is called and initialized.

- Then it is printed then the array is getting added using numpy
- Then finally they are printed
- The output comes out in the below fashion.
- **Output:**

```
Series([], dtype: float64)
0    g
1    e
2    e
3    k
4    s
dtype: object
```

2. Data Frame:

- It handles 3 parts, mainly data, columns and rows.
- Example:

```
import pandas as pd
  # Calling DataFrame constructor
df = pd.DataFrame()
print(df)
 # list of strings
lst = ['Madhu, 'For', 'Madhusri', 'is',
          'portal', 'for', 'students'']
  # Calling DataFrame constructor on list
df = pd.DataFrame(lst)
print(df)
```

Output:Empty DataFrame

Columns: []

Index: []

```
       0
0   Madhu
1     For
2   Madhusri
3      is
```

4 portal

5 for

6 students

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.to_string())
```

Output:

	Duration	Pulse	Maxpulse	Calories
0	60	110	130	409.1
1	60	117	145	479.0
2	60	103	135	340.0
3	45	109	175	282.4
4	45	117	148	406.0
5	60	102	127	300.5
6	60	110	136	374.0
7	45	104	134	253.3
8	30	109	133	195.1
9	60	98	124	269.0
10	60	103	147	329.3
11	60	100	120	250.7
12	60	106	128	345.3
13	60	104	132	379.3
14	60	98	123	275.0
15	60	98	120	215.2

.

;

;

;

And so on….

2.3 SCIPY

It falls under NumPy:

- It uses scientific and mathematical logic.
- It makes the python very effective as it allows user interaction too.
- It stands for “Scientific Python”
- It is open source
- Manipulating N-dimension array is done through SciPy

- Some sub packages of SciPy are as follows:
- **Scipy.clusetr:** K mean algorithm and such similar algorithms can be done using this library.
- **Scipy.io:** Inputs and outputs are handled here

```
import scipy

print(scipy.__version__)
```

Output:

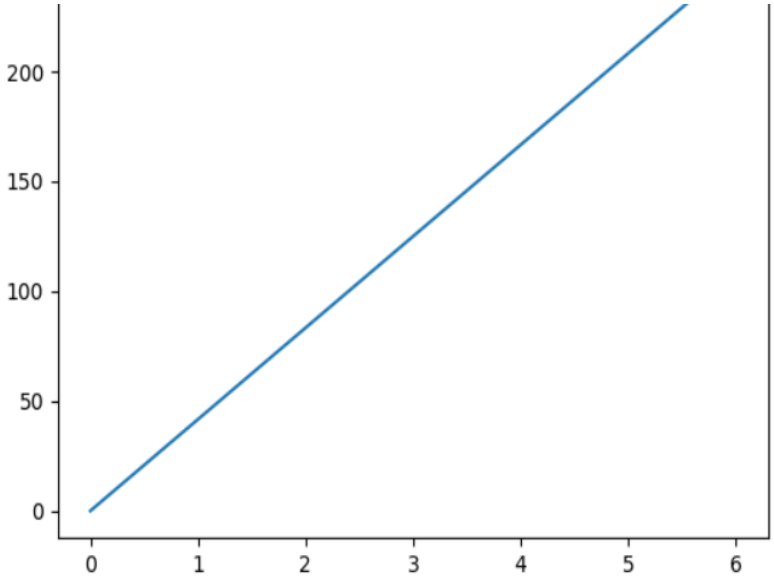
```
0.18.1
```

2.4 MATPLOTLIB

- It is used to plot graphs
- John D. Hunter created this
- It is open source
- In python you need to install matplotlib pip otherwise the code will not execute. To do this go to cmd and go the the folder where python is located any type the following command:

```
Pip install matplotlib
```

```
import matplotlib.pyplot as plt

import numpy as np

xpoints = np.array([0, 6])

ypoints = np.array([0, 250])

plt.plot(xpoints, ypoints)

plt.show()
```



2.5 SCIKIT LEARN

- It is mainly used in machine learning
- It has lot of statistics related tools
- It is open source.
- By using the Scikit library the efficiency will improve tremendously as it is quite accurate.
- It is very useful in algorithms which are very famous in machine learning like K-mean, K-nearest, clustering etc.
- It is available to everybody so any programmer if he or she feels like utilizing it then can use it.
- Scikit requires Numpy
- Installation of scikit is must to make the program run, this can be done in the following manner.

```
pip install -U scikit-learn
```

- Example:

```
from sklearn.datasets import load_iris

iris = load_iris()

A= iris.data

y = iris.target
```

```
feature_names = iris.feature_names
target_names = iris.target_names

print("Feature names:", feature_names)
print("Target names:", target_names)
print("\nFirst 10 rows of A:\n", A[:10])
```

Output:

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

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

First 10 rows of X:

```
=
[
  [5.1 3.5 1.4 0.2]
  [4.9 3. 1.4 0.2]
  [4.7 3.2 1.3 0.2]
  [4.6 3.1 1.5 0.2]
  [5. 3.6 1.4 0.2]
  [5.4 3.9 1.7 0.4]
  [4.6 3.4 1.4 0.3]
  [5. 3.4 1.5 0.2]
  [4.4 2.9 1.4 0.2]
  [4.9 3.1 1.5 0.1]
]
```

- Features of Scikit learn are as follows:
- Clustering: Scikit can be applied in clustering algorithm, in clustering the grouping is done on the basis of similarities like eg: age, color etc.
- Cross validation
- Feature selection
- Example:

```
# importing required libraries

import pandas as pd
```

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# read the train and test dataset
train_data = pd.read_csv('train.csv')
test_data = pd.read_csv('test.csv')

print(train_data.head())

# shape of the dataset
print('\nShape of training data :',train_data.shape)
print('\nShape of testing data :',test_data.shape)

# Now, we need to predict the missing target variable in the test data
# target variable - Item_Outlet_Sales

# seperate the independent and target variable on training data
train_x = train_data.drop(columns=['Item_Outlet_Sales'],axis=1)
train_y = train_data['Item_Outlet_Sales']

# seperate the independent and target variable on training data
test_x = test_data.drop(columns=['Item_Outlet_Sales'],axis=1)
test_y = test_data['Item_Outlet_Sales']

'''
Create the object of the Linear Regression model
You can also add other parameters and test your code here
Some parameters are : fit_intercept and normalize
Documentation of sklearn Linear Regression:
https://scikit-
learn.org/stable/modules/generated/sklearn.linear_model.LinearRegressio
n.html
 '''

model = LinearRegression()

# fit the model with the training data
model.fit(train_x,train_y)

# coefficeints of the trained model
```

```
print('\nCoefficient of model :', model.coef_)

# intercept of the model

print('\nIntercept of model',model.intercept_)

# predict the target on the test dataset

predict_train = model.predict(train_x)

print('\nItem_Outlet_Sales on training data',predict_train)

# Root Mean Squared Error on training dataset

rmse_train = mean_squared_error(train_y,predict_train)**(0.5)

print('\nRMSE on train dataset : ', rmse_train)

# predict the target on the testing dataset

predict_test = model.predict(test_x)

print('\nItem_Outlet_Sales on test data',predict_test)

# Root Mean Squared Error on testing dataset

rmse_test = mean_squared_error(test_y,predict_test)**(0.5)

print('\nRMSE on test dataset : ', rmse_test)
```

Output:

```
Item_Weight  ...  Outlet_Type_Supermarket Type3
0     6.800000  ...                          0
1    15.600000  ...                          0
2    12.911575  ...                          1
3    11.800000  ...                          0
4    17.850000  ...                          0
[5 rows x 36 columns]
Shape of training data : (1364, 36)
Shape of testing data : (341, 36)
```

Coefficient of model:

[-3.84197604e+00 9.83065945e+00 1.61711856e+01 6.09197622e+01
-8.64161561e+01 1.23593376e+02 2.34714039e+02 -2.44597425e+02
-2.72938329e+01 -8.09611456e+00 -3.01147840e+02 1.70727611e+02
-5.40194744e+01 7.34248834e+01 1.70313375e+00 -5.07701615e+01
1.63553657e+02 -5.85286125e+01 1.04913492e+02 -6.01944874e+01
1.98948206e+02 -1.40959023e+02 1.19426257e+02 2.66382669e+01

*****

UNIT III

3

SUPERVISED LEARNING

Unit Structure

3.0 OBJECTIVES

This Chapter would make you understand the following concepts:

- What is a Regression?
- Types of a Regression.
- What is the mean of Linear regression and the importance of Linear regression?
- Importance of cost function and gradient descent in a Linear regression.
- Impact of different values for learning rate.
- What is the mean of logistic regression and the importance of Linear regression?

- Importance of cost function and gradient descent in a logistic regression.

3.1 INTRODUCTION – REGRESSION

Regression is a supervised learning technique that supports finding the correlation among variables.

A regression problem is when the output variable is a real or continuous value.

3.1.1 What is a Regression:

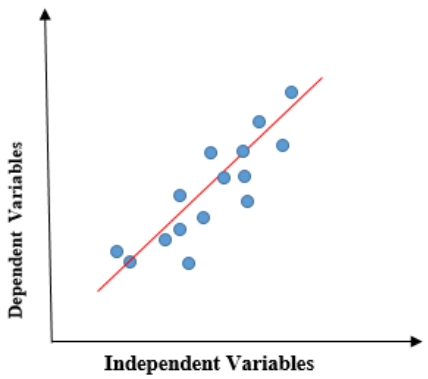
In Regression, we plot a graph between the variables which best fit the given data points. The machine learning model can deliver predictions regarding the data. In naïve words, **"Regression shows a line or curve that passes through all the data points on a target-predictor graph in such a way that the vertical distance between the data points and the regression line is minimum."** *It is used principally for prediction, forecasting, time series modeling, and determining the causal-effect relationship between variables.*

3.2 TYPES OF REGRESSION MODELS

1. Linear Regression
2. Polynomial Regression
3. Logistics Regression

3.2.1 Linear Regression:

Linear regression is a quiet and simple statistical regression method used for predictive analysis and shows the relationship between the continuous variables. Linear regression shows the linear relationship between the independent variable (X-axis) and the dependent variable (Y-axis), consequently called linear regression. If there is a single input variable (x), such linear regression is called **simple linear regression**. And if there is more than one input variable, such linear regression is called **multiple linear regression.** The linear regression model gives a sloped straight line describing the relationship within the variables.



The above graph presents the linear relationship between the dependent variable and independent variables. When the value of x (**independent variable**) increases, the value of y (**dependent variable**) is likewise increasing. The red line is referred to as the best fit straight line. Based on the given data points, we try to plot a line that models the points the best.

To calculate best-fit line linear regression uses a traditional slope-intercept form.

$$y = mx+b \quad \Longrightarrow \quad y = a_0+a_1x$$

y= Dependent Variable.

x= Independent Variable.

a0= intercept of the line.

a1 = Linear regression coefficient.

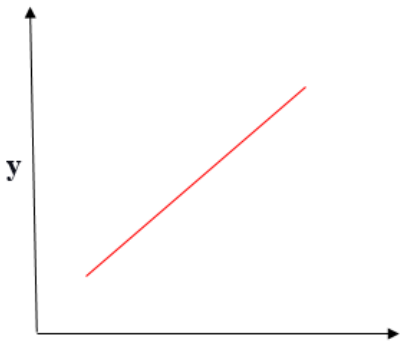
3.2.2 Need of a Linear regression:

Linear regression estimates the relationship between a dependent variable and an independent variable. Let's say we want to estimate the salary of an employee based on year of experience. You have the recent company data, which indicates that the relationship between experience and salary. Here year of experience is an independent variable, and the salary of an employee is a dependent variable, as the salary of an employee is dependent on the experience of an employee. Using this insight, we can predict the future salary of the employee based on current & past information.

A regression line can be a Positive Linear Relationship or a Negative Linear Relationship.

3.2.3 Positive Linear Relationship:

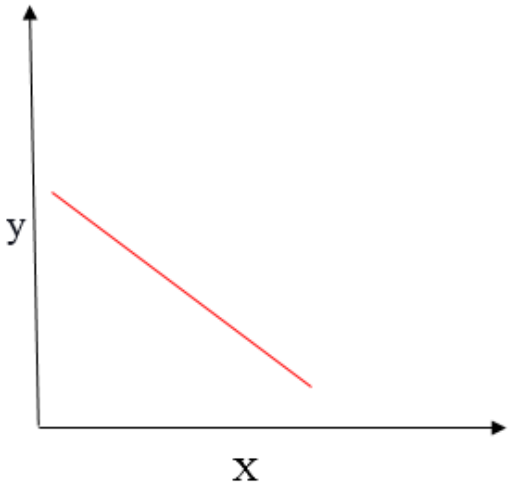
If the dependent variable expands on the Y-axis and the independent variable progress on X-axis, then such a relationship is termed a Positive linear relationship.



$y = a_0+a_1x$

3.2.4 Negative Linear Relationship:

If the dependent variable decreases on the Y-axis and the independent variable increases on the X-axis, such a relationship is called a negative linear relationship.



$$y = -a_0 + a_1 x$$

The goal of the linear regression algorithm is to get the best values for a0 and a1 to find the best fit line. The best fit line should have the least error means the error between predicted values and actual values should be minimized.

3.3 COST FUNCTION

The cost function helps to figure out the best possible values for a0 and a1, which provides the best fit line for the data points.

Cost function optimizes the regression coefficients or weights and measures how a linear regression model is performing. The cost function is used to find the accuracy of the **mapping function** that maps the input variable to the output variable. This mapping function is also known as **the Hypothesis function**.

In Linear Regression, **Mean Squared Error (MSE)** cost function is used, which is the average of squared error that occurred between the predicted values and actual values.

By simple linear equation y=mx+b we can calculate MSE as:

Let's y = actual values, y_i = predicted values

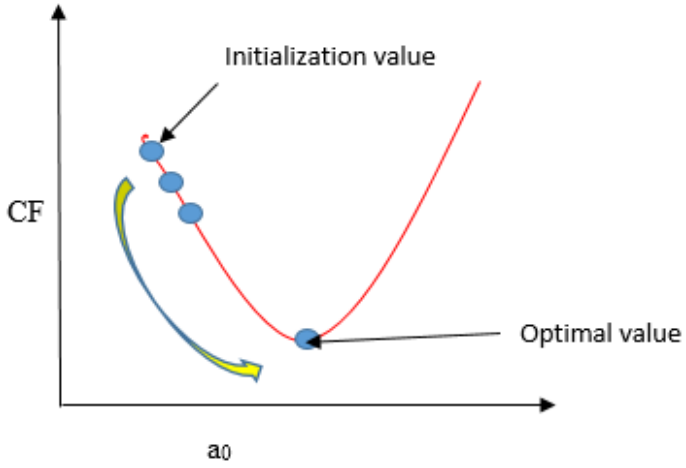
$$MSE = \frac{1}{N}\sum_{i=1}^{n}(y_i - (mx_i + b))^2$$

Using the MSE function, we will change the values of a0 and a1 such that the MSE value settles at the minima. Model parameters ***xi, b (a_0,a_1)*** can be manipulated to minimize the cost function. These parameters can be

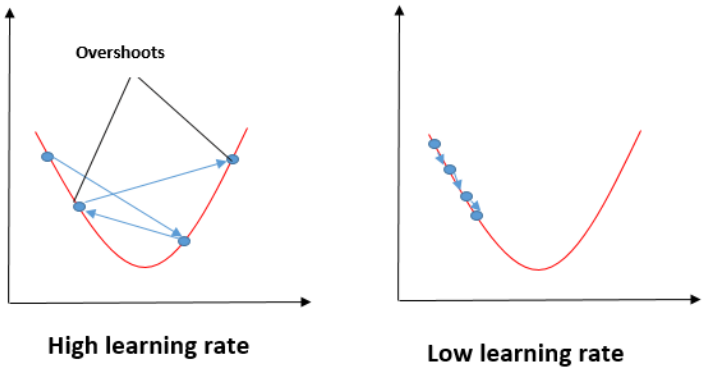
determined using the gradient descent method so that the cost function value is minimum.

3.3.1 Gradient descent:

Gradient descent is a method of updating a0 and a1 to minimize the cost function (MSE). A regression model uses gradient descent to update the coefficients of the line (a0, a1 => xi, b) by reducing the cost function by a random selection of coefficient values and then iteratively update the values to reach the minimum cost function.



Imagine a pit in the shape of U. You are standing at the topmost point in the pit, and your objective is to reach the bottom of the pit. There is a treasure, and you can only take a discrete number of steps to reach the bottom. If you decide to take one footstep at a time, you would eventually get to the bottom of the pit but, this would take a longer time. If you choose to take longer steps each time, you may get to sooner but, there is a chance that you could overshoot the bottom of the pit and not near the bottom. In the gradient descent algorithm, the number of steps you take is the learning rate, and this decides how fast the algorithm converges to the minima.



To update a_0 and a_1, we take gradients from the cost function. To find these gradients, we take partial derivatives for a_0 and a_1.

$$J = \frac{1}{n}\sum_{i=1}^{n}(a_0 + a_1 \cdot x_i - y_i)^2$$

$$\frac{\partial J}{\partial a_0} = \frac{2}{n}\sum_{i=1}^{n}(a_0 + a_1 \cdot x_i - y_i)$$

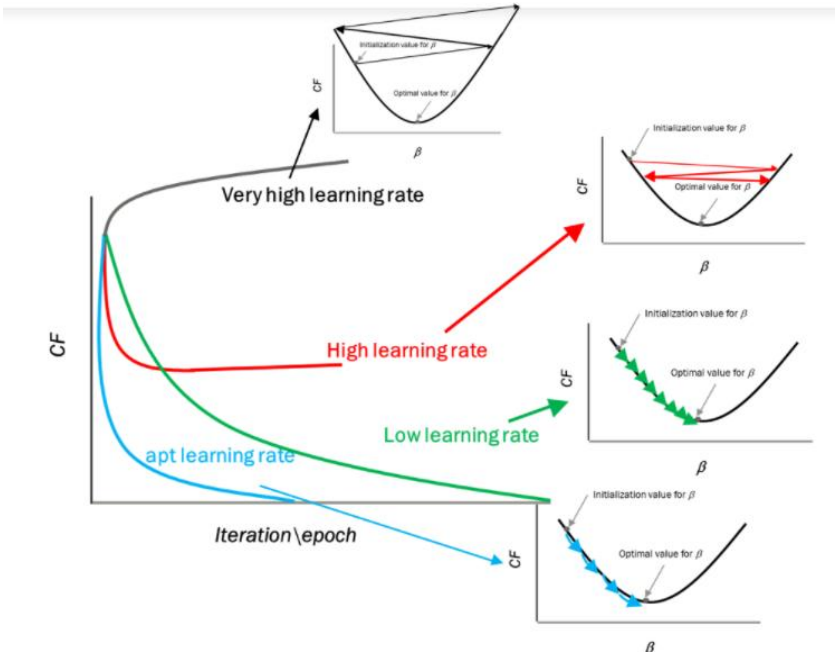
$$\frac{\partial J}{\partial a_1} = \frac{2}{n}\sum_{i=1}^{n}(a_0 + a_1 \cdot x_i - y_i) \cdot x_i$$

$$a_0 = a_0 - \alpha \cdot \frac{2}{n}\sum_{i=1}^{n}(pred_i - y_i)$$

$$a_1 = a_1 - \alpha \cdot \frac{2}{n}\sum_{i=1}^{n}(pred_i - y_i) \cdot x_i$$

The partial derivates are the gradients, and they are used to update the values of a_0 and a_1. Alpha is the learning rate.

3.3.2 Impact of different values for learning rate:



The blue line represents the optimal value of the learning rate, and the cost function value is minimized in a few iterations. The green line represents if the learning rate is lower than the optimal value, then the number of iterations required high to minimize the cost function. If the learning rate selected is very high, the cost function could continue to increase with iterations and saturate at a value higher than the minimum value, that represented by a red and black line.

3.3.3 Use case:

In this, I will take random numbers for the dependent variable (salary) and an independent variable (experience) and will predict the impact of a year of experience on salary.

3.3.4 Steps to implement linear regression model:

a) Import some required libraries

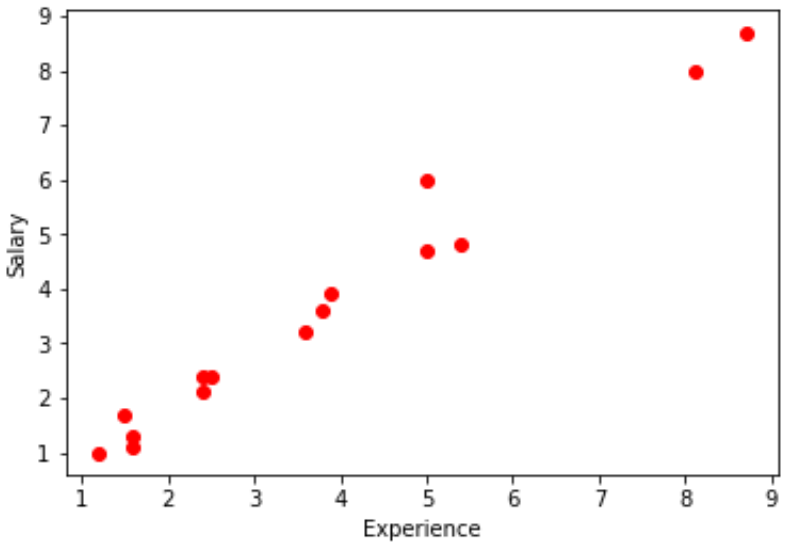
```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

b) Define the dataset

```
x= np.array([2.4,5.0,1.5,3.8,8.7,3.6,1.2,8.1,2.5,5,1.6,1.6,2.4,3.9,5.4])
y = np.array([2.1,4.7,1.7,3.6,8.7,3.2,1.0,8.0,2.4,6,1.1,1.3,2.4,3.9,4.8])
n = np.size(x)
```

c) Plot the data points

```
plt.scatter(experience,salary, color = 'red')
plt.xlabel("Experience")
plt.ylabel("Salary")
plt.show()
```



The main function to calculate values of coefficients:

1. Initialize the parameters.
2. Predict the value of a dependent variable by given an independent variable.
3. Calculate the error in prediction for all data points.
4. Calculate partial derivative w.r.t a0 and a1.
5. Calculate the cost for each number and add them.
6. Update the values of a0 and a1.

Initialize the parameters:

```
a0 = 0                  #intercept`
a1 = 0                  #Slop
lr = 0.0001             #Learning rate
iterations = 1000       # Number of iterations
error = []              # Error array to calculate cost for each iterations.
for itr in range(iterations):
    error_cost = 0
    cost_a0 = 0
    cost_a1 = 0
    for i in range(len(experience)):
        y_pred = a0+a1*experience[i]   # predict value for given x
        error_cost = error_cost +(salary[i]-y_pred)**2
        for j in range(len(experience)):
         partial_wrt_a0 = -2 *(salary[j] - (a0 + a1*experience[j]))
#partial derivative w.r.t a0
          partial_wrt_a1 = (-2*experience[j])*(salary[j]-(a0 +
a1*experience[j]))
           #partial derivative w.r.t a1
cost_a0 = cost_a0 + partial_wrt_a0      #calculate cost for each number
and add
```

cost_a1 = cost_a1 + partial_wrt_a1 **#calculate cost for each number and add**

a0 = a0 - lr * cost_a0 **#update a0**

a1 = a1 - lr * cost_a1 **#update a1**

print(itr,a0,a1) **#Check iteration and updated a0 and a1**

error.append(error_cost) **#Append the data in array**

```
51 -0.2100587036075669 1.0240594725158565
51 -0.210069416639929 1.0240604165874214
51 -0.2100827665464727 1.0240617279107738
51 -0.21009872814788516 1.024063481908892
51 -0.2101172734497008 1.0240657579772252
51 -0.21013837262662904 1.0240686345668573
51 -0.21016199500166557 1.0240721843016172
51 -0.21018810996167744 1.0240764694231033
51 -0.21021668775506228 1.0240815378378745
51 -0.21024770012426341 1.02408742000484
51 -0.21028112073594665 1.0240941268503343
51 -0.21031692538390484 1.0241016488365344
51 -0.21035509195351762 1.0241099562394875
52 -0.21035743358141284 1.024110693217287
52 -0.21036211816964787 1.0241121510338222
52 -0.2103691481960975 1.0241142983610119
52 -0.2103785269213727 1.024117090524692
52 -0.21039025787243382 1.024120472129565
52 -0.2104043442038269 1.0241243803082456
52 -0.21043078797530397 1.024128748398941
```

At approximate iteration 50- 60, we got the value of a0 and a1.

print(a0)

print(a1)

```
-0.21354150071690242
1.0247464287610857
```

Plotting the error for each iteration:

plt.figure(figsize=(10,5))

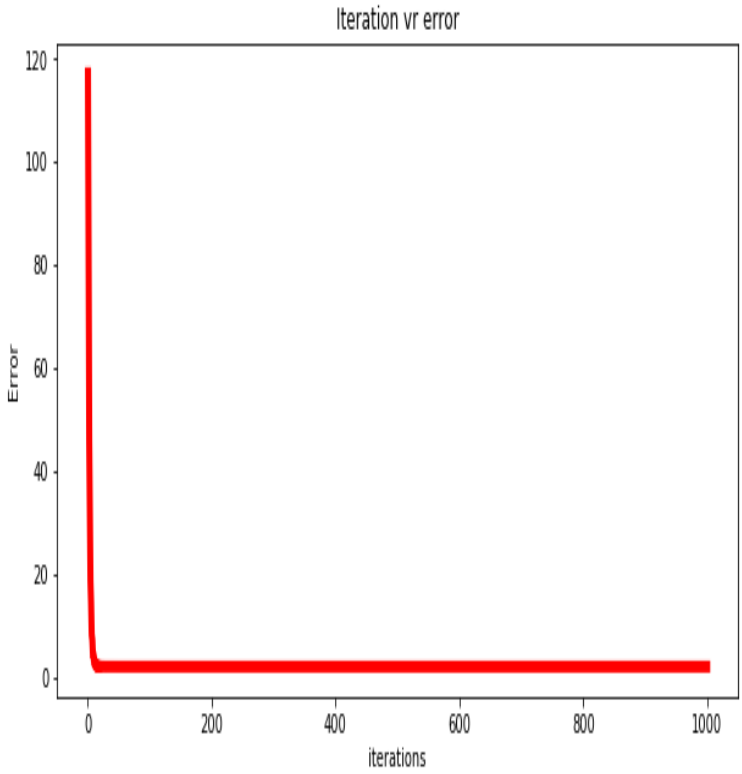
plt.plot(np.arange(1,len(error)+1),error,color='red',linewidth = 5)

plt.title("Iteration vr error")

plt.xlabel("iterations")

plt.ylabel("Error")

```
Text(0, 0.5, 'Error')
```



Predicting the values:

pred = a0+a1*experience

print(pred)

```
[2.24584993 4.91019064 1.32357814 3.68049493 8.70175243 3.47554564
 1.01615421 8.08690457 2.34832457 4.91019064 1.42605279 1.42605279
 2.24584993 3.78296957 5.32008921]
```

Plot the regression line:

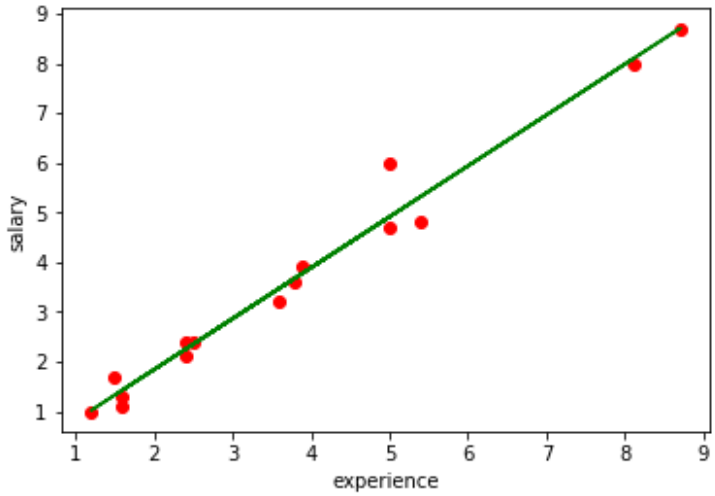
plt.scatter(experience,salary,color = 'red')

plt.plot(experience,pred, color = 'green')

plt.xlabel("experience")

plt.ylabel("salary")

```
Text(0, 0.5, 'salary')
```



Analyze the performance of the model by calculating the mean squared error.

error1 = salary - pred

se = np.sum(error1 ** 2)

mse = se/n

print("mean squared error is", mse)

```
mean squared error is 0.12785817711928918
```

Use the scikit library to confirm the above steps:

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error

experience = experience.reshape(-1,1)

model = LinearRegression()

model.fit(experience,salary)

salary_pred = model.predict(experience)

Mse = mean_squared_error(salary, salary_pred)

print('slop', model.coef_)

print("Intercept", model.intercept_)

print("MSE", Mse)

```
slop [1.02474643]
Intercept -0.2135415007169037
MSE 0.1278581771192891
```

3.4 WHAT IS LOGISTIC REGRESSION?

Logistic regression is a supervised learning algorithm that outputs values between zero and one.

3.4.1 Hypothesis:

The objective of a logistic regression is to learn a function that outputs the probability that the dependent variable is one for each training sample. To achieve that, a sigmoid / logistic function is required for the transformation.

3.4.2 A sigmoid function:

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

Visually, it looks like this:

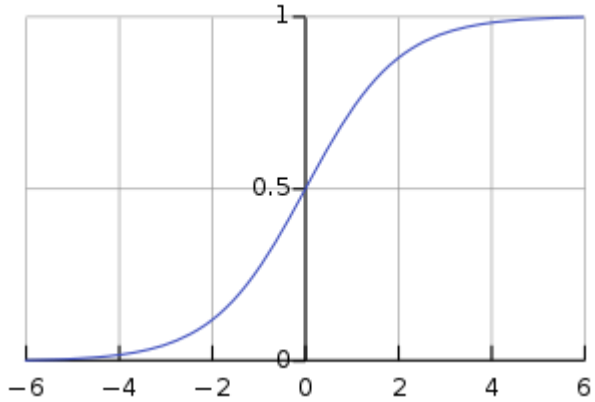


Fig. 1. Sigmoid Function

This hypothesis is typically represented by the following function:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

Where,

- θ is a vector of parameters that corresponds to each independent variable
- x is a vector of independent variables

3.5 COST FUNCTION

The cost function for logistic regression is derived from statistics using the principle of maximum likelyhood estimation, which allows efficient identification of parameters. In addition the covex property of the cost function allow gradient descent to work eff ectively.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^i) - y^i)^2$$

$$(h_\theta(x^i) - y^i)^2 = \begin{cases} -log(h_\theta(x^i)) \text{ if } y^i = 1 \\ -log(1 - h_\theta(x^i)) \text{ if } y^i = 0 \end{cases}$$

Where,

- i is one of the m^{th} training samples
- $h_\Theta(x^i)$ is the predicted value for the training sample
- y^i is the actual value for the training sample

To understand the cost function, we can look into each of the two components in isolation:

Suppose $y^i=1$:

if , $h_\Theta(x^i)$ =1 then the predicon error = 0

if , $h_\Theta(x^i)$ =0 then the predicon error approaches infinity

These two scenarios are represented by the blue line in Figure 2 below.

Suppose $y^i=0$:

if , $h_\Theta(x^i)$ =0 then the predicon error = 0

if , $h_\Theta(x^i)$ =1 then the predicon error approaches infinity

These two scenarios are represented by the blue line in Figure 2 below.

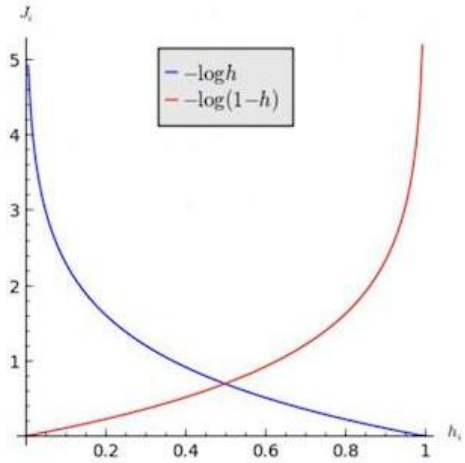


Fig. 2. Logistic Regression Cost Function

The logistic regression cost function can be further simplified into a one line equation:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^i) - y^i)^2$$

$$(h_\theta(x^i) - y^i)^2 = -y^i log(h_\theta(x^i)) - (1 - y^i)log(1 - h_\theta(x^i)$$

The overall objective is to minimise the cost function by iterating through diferent values of Θ.

$$\min_{\theta_0, \ldots, \theta_n} J(\theta_0, \ \ldots, \ \theta_n)$$

3.5.1 Gradient Descent

The gradient descent algorithm is as follows:

repeat until convergence

$$\left\{\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta_0, \ \ldots, \ \theta_n)\right\}$$

Where,

- values of j = 0,1, …, n
- α is the learning rate

Note: The gradient descent algorithm is identical to linear regression's

3.6 LETS SUM UP

- What is a Regression?
- Types of a Regression.
- What is the mean of Linear regression and the importance of Linear regression?
- Importance of cost function and gradient descent in a Linear regression.
- Impact of different values for learning rate.
- What is the mean of logistic regression and the importance of Linear regression?
- Importance of cost function and gradient descent in a logistic regression.

3.7 EXERCISES

- Differentiate the Linear regression and logistic regression with a real time example.

3.8 REFERENCES

- https://www.studytonight.com/post/linear-regression-and-predicting-values-based-on-a-training-dataset
- https://activewizards.com/blog/5-real-world-examples-of-logistic-regression-application
- https://www.marktechpost.com/2021/11/12/logistic-regression-with-a-real-world-example-in-python/
- https://www.statology.org/linear-regression-real-life-examples/#:~:text=For%20example%2C%20researchers%20might%20administer,pressure%20as%20the%20response%20variable.
- https://www.quora.com/What-are-applications-of-linear-and-logistic-regression
- https://www.statology.org/logistic-regression-real-life-examples/
- https://ncss-wpengine.netdna-ssl.com/wp-content/themes/ncss/pdf/Procedures/NCSS/Logistic_Regression.pdf
- https://www.analyticsvidhya.com/blog/2021/06/linear-regression-in-machine-learning/
- https://www.analyticsvidhya.com/blog/2022/01/an-introductory-note-on-linear-regression/
- http://home.iitk.ac.in/~shalab/regression/Chapter3-Regression-MultipleLinearRegressionModel.pdf
- https://www.princeton.edu/~otorres/Regression101.pdf
- https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761
- https://www.javatpoint.com/k-nearest-neighbor-algorithm-for-machine-learning
- https://www.analyticsvidhya.com/blog/2021/04/simple-understanding-and-implementation-of-knn-algorithm/

*****

4

SUPERVISED LEARNING

Unit Structure

4.0 OBJECTIVES

This Chapter would make you understand the following concepts:

- Advanced Optimization Algorithms
- Applications of Linear/Logistic regression.
- KNN- classification

4.1 ADVANCED OPTIMIZATION ALGORITHMS

However, gradient descent is not the only algorithm that can minimize the cost function. Other advanced optimization algorithms are:

- Conjugate gradient
- BFGS
- L-BFGS

While these advanced algorithms are more complex and difficult to understand, they have the advantages of converging faster and not needing to pick learning rate.

4.1.1 Multiclass Classification:

One-vs-rest is a method where you turn a n-class classification problem into a nth seperate binary classification problem.

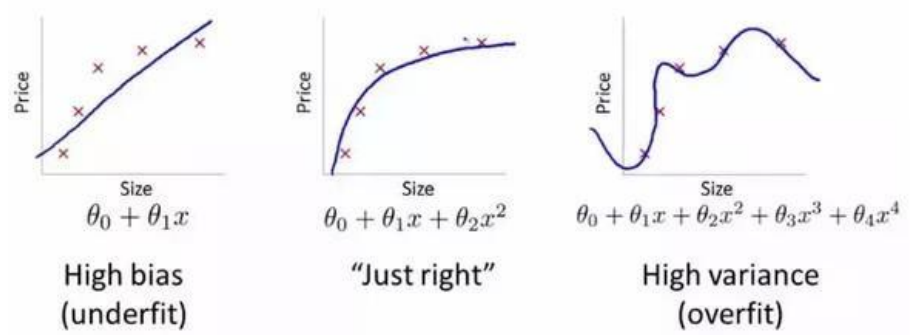
To deal with a multiclass problem, we then train a logistic regression binary classifier for each class to predict the probability that y = i. The prediction output for a given new input will be chosen based on the classifier that has the highest probability.

$$\min_i h_\theta^i(x)$$

where is $h_\theta^i(x)$ the binary classifier

4.1.2 Bias-Variance Tradeoff :

Overfing occurs when the algorithm tries too hard to fit the training data. This usually results in a learned hypothesis that is too complex, fails to generalize to new examples, and a cost funcon that is very close to zero on the training set. On the contrary, underfing occurs when the algorithm tries too lile to fit the training data. This usually results in a learned hypothesis

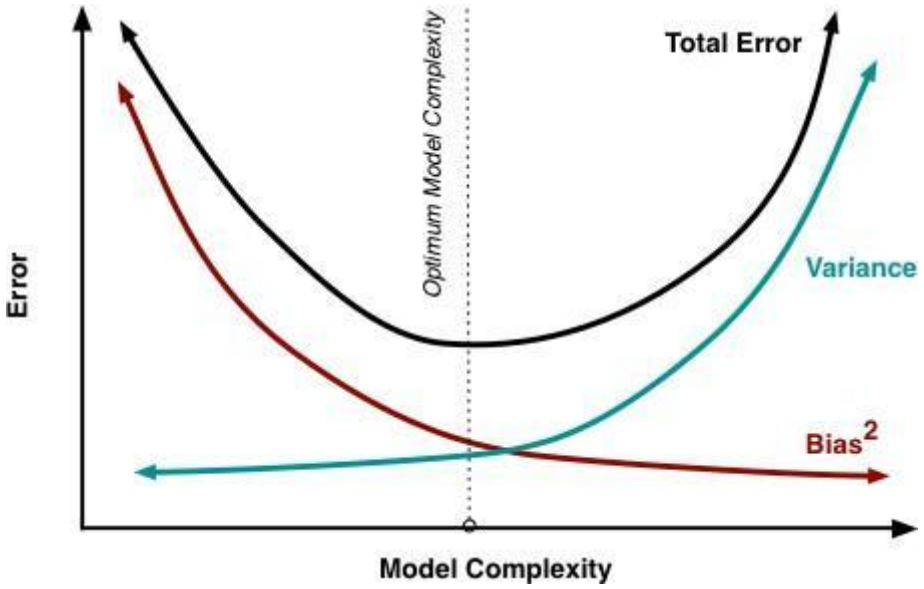


that is not complex enough, and fails to generalize to new examples.

Underfitting and Overfitting

Conceptually speaking, bias measures the difference between model predictions and the correct values. Variance refers to the variability of a model prediction for a given data point if you re-build the model multiple message.

As seen in Figure 4, the optimal level of model complexity is where prediction error on unseen data points is minimized. Below the optimal level of model complexity, bias will increase while variance will decrease due to a hypothesis that is too simplified. On the contrary, a very complex model will result in a low bias and high variance situation



Bias-Variance Tradeo

4.1.3 Regularization:

For a model to generalize well, regulariza**ti**on is usually introduced to reduce over fitting of the training data.

This is represented by a regularization term, that is added to the cost function that penalizes all parameters that are high in value. This leads to a simpler hypothesis that is less prone to fitting. The new cost func**ti**on then becomes:

$$J(\theta) = \frac{1}{m}\left[\sum_{i=1}^{m}(h_\theta(x^i) - y^i)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

Where,

- i is one of the training samples
- is the predicted value for the training sample *i*
- yi is the actual value for the training sample i
- λ is the regularizaon parameter that controls the tradeoff between fing the training dataset well and having the parameters θ small in values
- j is one of the parameter θ

Overall objecve remains the same: $\min_{\theta_0, \ldots, \theta_n} J(\theta_0, \ \ldots, \ \theta_n)$

Gradient descent remains the same as well:

$$\left\{\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \ \ldots, \ \theta_n)\right\}$$ repeat until convergence

4.2 APPLICATIONS OF LINEAR/LOGISTIC REGRESSION

Regression models are generally built on historical data which has some independent variables and a dependent variable. A dependent variable is a characteristic or quantity that you want to measure using the independent variables.

4.2.1 Two things you can do using regression are:

1. Find the impact of the dependent variables on the response based on the historical data.
2. Use this generalization to predict what can happen in the future using new cases.

Linear regression is used when the response is a continuous variable (CV). Some examples of CVs are height of a person, sales of a product, revenues of a company etc.

Logistic regression is used when the response you want to predict/measure is categorical with two or more levels. Some examples are gender of a person, outcome of a football match, etc.

For **example** let's take a scenario where you are analyzing the voting patterns of USA to predict who will win the next election.

In such case you would use:

1. **Linear Regression:** if you want to predict the number of people(continuous response) who will vote for democrats/republicans in each county/city/state etc.,
2. **Logistic Regression:** if you want to predict the probability that a certain person will vote for a democrat/republican or not.

Regressions can be used in real world applications such as:

1. Credit Scoring
2. Measuring the success rates of marketing campaigns
3. Predicting the revenues of a certain product
4. Is there going to be an earthquake on a particular day? etc.,

4.2.2 Application of logistic regression

Logistic Regression Real Life Example: 1

Medical researchers want to know how exercise and weight impact the probability of having a heart attack. To understand the relationship between the predictor variables and the probability of having a heart attack, researchers can perform logistic regression.

The response variable in the model will be heart attack and it has two potential outcomes:

- A heart attack occurs.
- A heart attack does not occur.

The results of the model will tell researchers exactly how changes in exercise and weight affect the probability that a given individual has a heart attack. The researchers can also use the fitted logistic regression model to predict the probability that a given individual has a heart attacked, based on their weight and their time spent exercising.

Logistic Regression Real Life Example: 2

Researchers want to know how GPA, ACT score, and number of AP classes taken impact the probability of getting accepted into a particular university. To understand the relationship between the predictor variables and the probability of getting accepted, researchers can perform logistic regression.

The response variable in the model will be "acceptance" and it has two potential outcomes:

- A student gets accepted.
- A student does not get accepted.

The results of the model will tell researchers exactly how changes in GPA, ACT score, and number of AP classes taken affect the probability that a given individual gets accepted into the university. The researchers can also use the fitted logistic regression model to predict the probability that a given individual gets accepted, based on their GPA, ACT score, and number of AP classes taken.

Logistic Regression Real Life Example :3

A business wants to know whether word count and country of origin impact the probability that an email is spam. To understand the relationship between these two predictor variables and the probability of an email being spam, researchers can perform logistic regression.

The response variable in the model will be "spam" and it has two potential outcomes:

- The email is spam.
- The email is not spam.

The results of the model will tell the business exactly how changes in word count and country of origin affect the probability of a given email being spam. The business can also use the fitted logistic regression model to predict the probability that a given email is spam, based on its word count and country of origin.

Logistic Regression Real Life Example :4

A credit card company wants to know whether transaction amount and credit score impact the probability of a given transaction being fraudulent. To understand the relationship between these two predictor variables and the probability of a transaction being fraudulent, the company can perform logistic regression.

The response variable in the model will be "fraudulent" and it has two potential outcomes:

- The transaction is fraudulent.
- The transaction is not fraudulent.

The results of the model will tell the company exactly how changes in transaction amount and credit score affect the probability of a given transaction being fraudulent. The company can also use the fitted logistic regression model to predict the probability that a given transaction is fraudulent, based on the transaction amount and the credit score of the individual who made the transaction.

4.3 K-NEAREST NEIGHBORS (KNN) CLASSIFICATION MODEL

1. Evaluation procedure 1 - Train and test on the entire dataset

1. Train the model on the **entire dataset**.
2. Test the model on the **same dataset**, and evaluate how well we did by comparing the **predicted** response values with the **true** response values.

In [1]:

 # read in the iris data

from sklearn.datasets import load_iris

iris = load_iris()

 # create X (features) and y (response)

X = iris**.**data

y = iris**.**target

1a. Logistic regression

In [2]:

 # import the class

from sklearn.linear_model import LogisticRegression

 # instantiate the model (using the default parameters)

logreg = LogisticRegression()

 # fit the model with data

logreg**.**fit(X, y)

 # predict the response values for the observations in X

logreg**.**predict(X)

Out[2]:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1,
       1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

In [3]:

 # store the predicted response values

y_pred = logreg**.**predict(X)

 # check how many predictions were generated

len(y_pred)

Out[3]:

150

Classification accuracy:

- **Proportion** of correct predictions
- Common **evaluation metric** for classification problems

In [4]:

 # compute classification accuracy for the logistic regression model

from sklearn import metrics

print(metrics**.**accuracy_score(y, y_pred))

0.96

- Known as **training accuracy** when you train and test the model on the same data
- 96% of our predictions are correct

1b. KNN (K=5)

In [5]:

from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=5)

knn**.**fit(X, y)

y_pred = knn**.**predict(X)

print(metrics**.**accuracy_score(y, y_pred))

0.966666666667

It seems, there is a higher accuracy here but there is a big issue of testing on your training data

1c. KNN (K=1)

In [6]:

knn = KNeighborsClassifier(n_neighbors=1)

knn**.**fit(X, y)

y_pred = knn**.**predict(X)

print(metrics**.**accuracy_score(y, y_pred))

1.0

- **KNN model:**

1. Pick a value for K.
2. Search for the K observations in the training data that are "nearest" to the measurements of the unknown iris
3. Use the most popular response value from the K nearest neighbors as the predicted response value for the unknown iris

- This would always have 100% accuracy, because we are testing on the exact same data, it would always make correct predictions
- KNN would search for one nearest observation and find that exact same observation
- KNN has memorized the training set
- Because we testing on the exact same data, it would always make the same prediction

1d. Problems with training and testing on the same data:

- Goal is to estimate likely performance of a model on **out-of-sample data**
- But, maximizing training accuracy rewards **overly complex models** that won't necessarily generalize
- Unnecessarily complex models **overfit** the training data

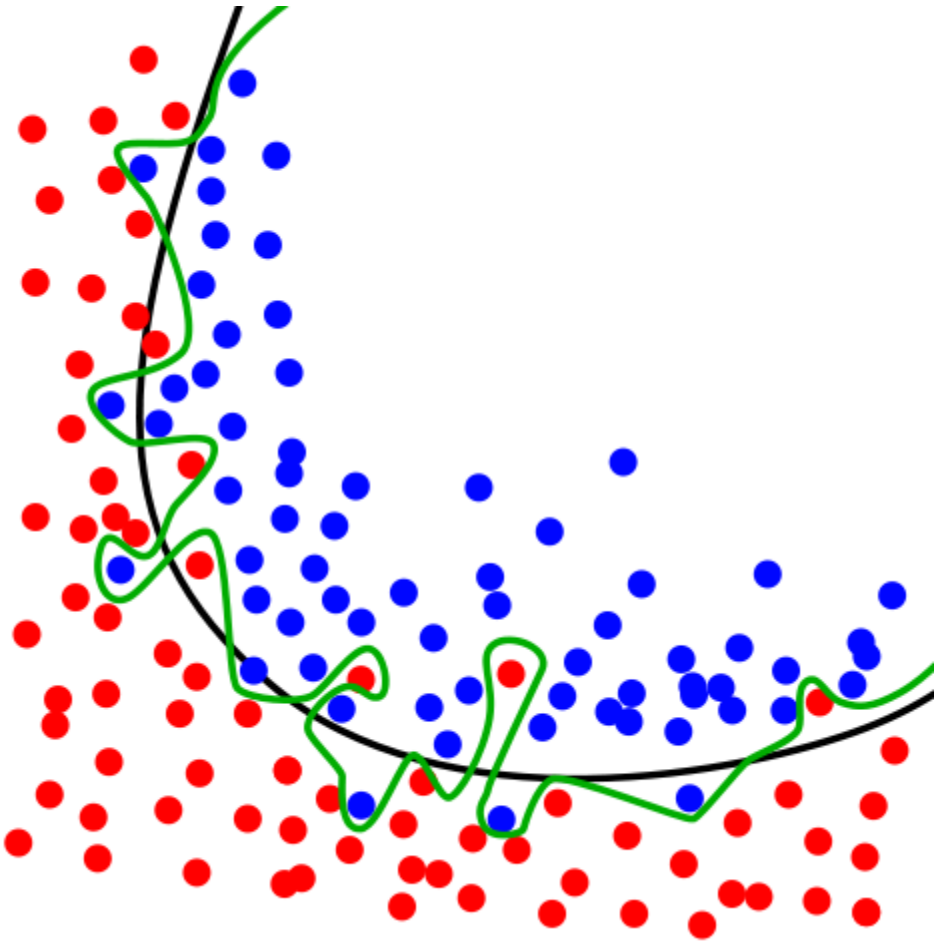


Image Credit: Overfitting by Chabacano. Licensed under GFDL via Wikimedia Commons.

- Green line (decision boundary): overfit

- Your accuracy would be high but may not generalize well for future observations
- Your accuracy is high because it is perfect in classifying your training data but not out-of-sample data
- Black line (decision boundary): just right
- Good for generalizing for future observations
- Hence we need to solve this issue using a **train/test split** that will be explained below

2. Evaluation procedure 2 - Train/test split

1. Split the dataset into two pieces: a **training set** and a **testing set**.
2. Train the model on the **training set**.
3. Test the model on the **testing set**, and evaluate how well we did.

In [7]:

 # print the shapes of X and y

 # X is our features matrix with 150 x 4 dimension

print(X**.**shape)

 # y is our response vector with 150 x 1 dimension

print(y**.**shape)

(150, 4)

(150,)

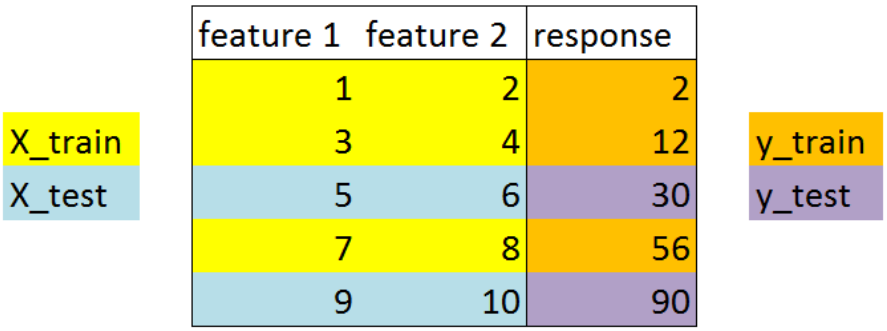
In [8]:

 # STEP 1: split X and y into training and testing sets

from sklearn.cross_validation import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=4)

- test_size=0.4
- 40% of observations to test set
- 60% of observations to training set
- data is randomly assigned unless you use random_state hyperparameter
- If you use random_state=4
- Your data will be split exactly the same way



What did this accomplish?

- Model can be trained and tested on **different data**
- Response values are known for the testing set, and thus **predictions can be evaluated**
- **Testing accuracy** is a better estimate than training accuracy of out-of-sample performance

In [9]:

 # print the shapes of the new X objects

print(X_train.shape)

print(X_test.shape)

(90, 4)

(60, 4)

In [10]:

 # print the shapes of the new y objects

print(y_train.shape)

print(y_test.shape)

(90,)

(60,)

In [11]:

 # STEP 2: train the model on the training set

logreg = LogisticRegression()

logreg.fit(X_train, y_train)

Out[11]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False,
fit_intercept=True,
      intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
      penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
      verbose=0, warm_start=False)
```

In [12]:

```
# STEP 3: make predictions on the testing set
y_pred = logreg.predict(X_test)
# compare actual response values (y_test) with predicted response values
(y_pred)
print(metrics.accuracy_score(y_test, y_pred))
```

0.95

Repeat for KNN with K=5:

In [13]:

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
print(metrics.accuracy_score(y_test, y_pred))
```

0.966666666667

Repeat for KNN with K=1:

In [14]:

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
print(metrics.accuracy_score(y_test, y_pred))
```

0.966666666667

Can we locate an even better value for K?

In [15]:

```
# try K=1 through K=25 and record testing accuracy
```

```
k_range = range(1, 26)

# We can create Python dictionary using [] or dict()

scores = []

# We use a loop through the range 1 to 26

# We append the scores in the dictionary

for k in k_range:

    knn = KNeighborsClassifier(n_neighbors=k)

    knn.fit(X_train, y_train)

    y_pred = knn.predict(X_test)

    scores.append(metrics.accuracy_score(y_test, y_pred))

print(scores)
```

```
[0.94999999999999996,  0.94999999999999996,  0.96666666666666667,
0.96666666666666667,  0.96666666666666667,  0.98333333333333328,
0.98333333333333328,  0.98333333333333328,  0.98333333333333328,
0.98333333333333328,  0.98333333333333328,  0.98333333333333328,
0.98333333333333328,  0.98333333333333328,  0.98333333333333328,
0.98333333333333328,  0.98333333333333328,  0.96666666666666667,
0.98333333333333328,  0.96666666666666667,  0.96666666666666667,
0.96666666666666667,  0.96666666666666667,  0.94999999999999996,
0.94999999999999996]
```

In [16]:

```
# import Matplotlib (scientific plotting library)

import matplotlib.pyplot as plt

# allow plots to appear within the notebook

%matplotlib inline

# plot the relationship between K and testing accuracy

# plt.plot(x_axis, y_axis)

plt.plot(k_range, scores)

plt.xlabel('Value of K for KNN')

plt.ylabel('Testing Accuracy')
```

Out[16]:

```
<matplotlib.text.Text at 0x111d43ba8>
```

- **Training accuracy** rises as model complexity increases
- **Testing accuracy** penalizes models that are too complex or not complex enough
- For KNN models, complexity is determined by the **value of K** (lower value = more complex)

3. Making predictions on out-of-sample data:

In [17]:

```
# instantiate the model with the best known parameters
knn = KNeighborsClassifier(n_neighbors=11)

# train the model with X and y (not X_train and y_train)
knn.fit(X, y)

# make a prediction for an out-of-sample observation
knn.predict([3, 5, 4, 2])
```

/Users/ritchieng/anaconda3/envs/py3k/lib/python3.5/site-packages/sklearn/utils/validation.py:386: DeprecationWarning:

Passing 1d arrays as data is deprecated in 0.17 and willraise ValueError in 0.19. Reshape your data either using X.reshape(-1, 1) if your data has a single feature or X.reshape(1, -1) if it contains a single sample.

DeprecationWarning)

Out[17]:

array([1])

4. Downsides of train/test split:

- Provides a **high-variance estimate** of out-of-sample accuracy
- **K-fold cross-validation** overcomes this limitation
- But, train/test split is still useful because of its **flexibility and speed**

4.4 LETS SUM UP

- Advanced Optimization Algorithms.
- Applications of Linear/Logistic regression.
- KNN- classification.

4.5 EXERCISES

- Appropriate the Linear regression and logistic regression with a real time example.
- Take a real time example and execute about KNN- classification

4.6 REFERENCES

- https://www.quora.com/What-are-applications-of-linear-and-logistic-regression
- https://www.statology.org/logistic-regression-real-life-examples/
- https://ncss-wpengine.netdna-ssl.com/wp-content/themes/ncss/pdf/Procedures/NCSS/Logistic_Regression.pdf
- https://www.analyticsvidhya.com/blog/2021/06/linear-regression-in-machine-learning/
- https://www.analyticsvidhya.com/blog/2022/01/an-introductory-note-on-linear-regression/
- https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761
- https://www.javatpoint.com/k-nearest-neighbor-algorithm-for-machine-learning
- https://www.analyticsvidhya.com/blog/2021/04/simple-understanding-and-implementation-of-knn-algorithm/

*****

UNIT IV

5

FEATURES AND EXTRACTION

Unit Structure

5.1 DIMENSIONALITY REDUCTION

Dimensionality reduction eliminates some features of the dataset and creates a restricted set of features that contains all of the information needed to predict the target variables more efficiently and accurately.

Reducing the number of features normally also reduces the output variability and complexity of the learning process. The covariance matrix is an important step in the dimensionality reduction process. It is a critical process to check the correlation between different features.

Correlation and its Measurement:

There is a concept of correlation in machine learning that is called multicollinearity. Multicollinearity exists when one or more independent variables highly correlate with each other. Multicollinearity makes variables highly correlated to one another, which makes the variables' coefficients highly unstable.

The coefficient is a significant part of regression, and if this is unstable, then there will be a poor outcome of the regression result. Multicollinearity is confirmed by using Variance Inflation Factors (VIF). Therefore, if multicollinearity is suspected, it can be checked using the variance inflation factor (VIF).

$$VIF = 1/(1-R^2)$$

Rules from VIF:

- A VIF of 1 would indicate complete independence from any other variable.
- A VIF between 5 and 10 indicates a very high level of collinearity [4].
- The closer we get to 1, the more ideal the scenario for predictive modeling.

- Each independent variable regresses against each independent variable, and we calculate the VIF.

Heatmap also plays a crucial role in understanding the correlation between variables.

The type of relationship between any two quantities varies over a period of time.

Correlation varies from **-1** to **+1**

To be precise,

- Values that are close to +1 indicate a positive correlation.
- Values close to -1 indicate a negative correlation.
- Values close to 0 indicate no correlation at all.

Below is the heatmap to show how we will correlate which features are highly dependent on the target feature and consider them.

The Covariance Matrix and Heatmap:

The covariance matrix is the first step in dimensionality reduction because it gives an idea of the number of features that strongly relate, and it is usually the first step in dimensionality reduction because it gives an idea of the number of strongly related features so that those features can be discarded.

It also gives the detail of all independent features. It provides an idea of the correlation between all the different pairs of features.

Identification of features in Iris dataset that are strongly correlated:

Import all the required packages:

```
import numpy as np
import pandas as pd
from sklearn import datasets
import matplotlib.pyplot as plt
```

Load Iris dataset:

```
iris = datasets.load_iris()
iris.data
```

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2],
       [5.4, 3.9, 1.7, 0.4],
       [4.6, 3.4, 1.4, 0.3],
       [5. , 3.4, 1.5, 0.2],
       [4.4, 2.9, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.1],
       [5.4, 3.7, 1.5, 0.2],
       [4.8, 3.4, 1.6, 0.2],
       [4.8, 3. , 1.4, 0.1],
       [4.3, 3. , 1.1, 0.1],
       [5.8, 4. , 1.2, 0.2],
       [5.7, 4.4, 1.5, 0.4],
       [5.4, 3.9, 1.3, 0.4],
       [5.1, 3.5, 1.4, 0.3],
```

Iris dataset.

List all features:

```
iris.feature_names
```

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

Features of the Iris dataset:

Create a covariance matrix:

```
cov_data = np.corrcoef(iris.data.T)cov_data
```

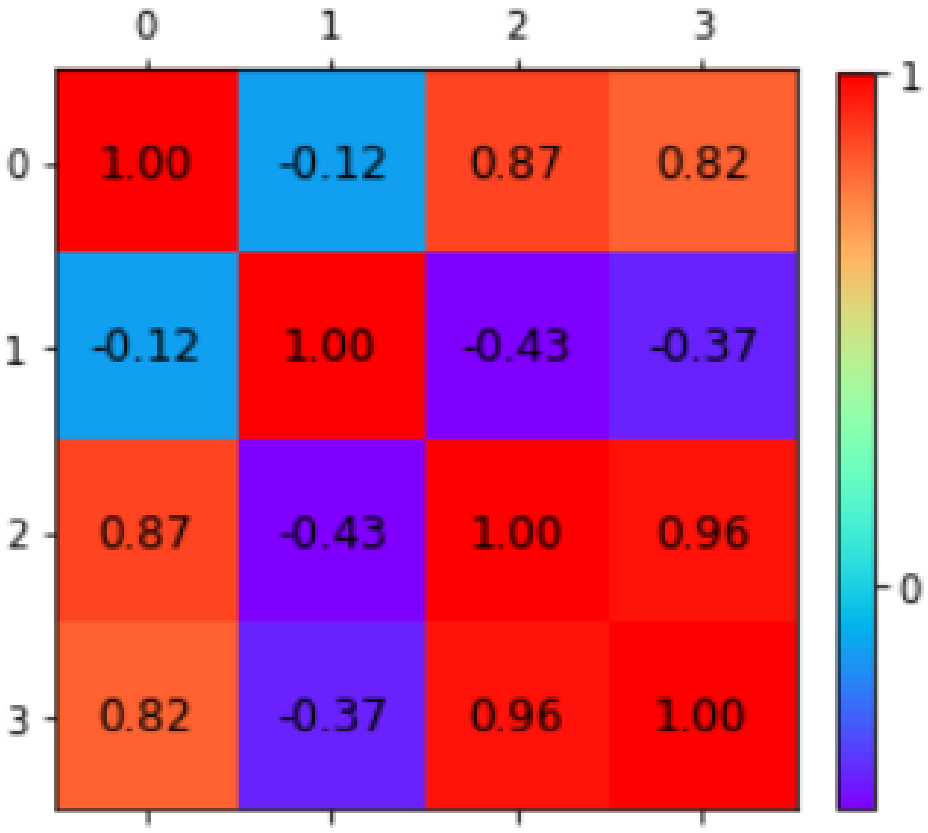
```
array([[ 1.        , -0.11756978,  0.87175378,  0.81794113],
       [-0.11756978,  1.        , -0.4284401 , -0.36612593],
       [ 0.87175378, -0.4284401 ,  1.        ,  0.96286543],
       [ 0.81794113, -0.36612593,  0.96286543,  1.        ]])
```

Covariance matrix of the Iris dataset.

Plot the covariance matrix to identify the correlation between features using a heatmap:

```
img = plt.matshow(cov_data, cmap=plt.cm.rainbow)
plt.colorbar(img, ticks = [-1, 0, 1], fraction=0.045)for x in
range(cov_data.shape[0]):
```

```
    for y in range(cov_data.shape[1]):
        plt.text(x, y, "%0.2f" % cov_data[x,y], size=12, color='black',
ha="center", va="center")

plt.show()
```



Heatmap of the correlation matrix.

A correlation from the representation of the heatmap:

- Among the **first** and the **third** features.
- Between the **first** and the **fourth** features.
- Between the **third** and the **fourth** features.

Independent features:

- The **second** feature is almost independent of the others.

Here the correlation matrix and its pictorial representation have given the idea about the potential number of features reduction. Therefore, two features can be kept, and other features can be reduced apart from those two features.

Feature Selection:

In feature selection, usually, a subset of original features is selected.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ . \\ . \\ x_N \end{bmatrix} \rightarrow \mathbf{y} = \begin{bmatrix} x_{i_1} \\ x_{i_2} \\ . \\ . \\ x_{i_K} \end{bmatrix} \quad K<<N$$

Feature selection

Feature Extraction:

In feature extraction, a set of new features are found. That is found through some mapping from the existing features. Moreover, mapping can be either linear or non-linear.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ . \\ . \\ x_N \end{bmatrix} \xrightarrow{f(\mathbf{x})} \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ y_K \end{bmatrix} \quad K<<N$$

Feature Extraction

Linear Feature Extraction:

Linear feature extraction is straightforward to compute and analytically traceable.

Widespread linear feature extraction methods:

- **Principal Component Analysis (PCA)**: It seeks a projection that preserves as much information as possible in the data.
- **Linear Discriminant Analysis (LDA)**:- It seeks a projection that best discriminates the data.

What is Principal Component Analysis?

Principal component analysis (PCA) is an unsupervised linear transformation technique which is primarily used for feature extraction and dimensionality reduction. It aims to **find the directions of maximum variance in high-dimensional data and projects the data onto a new subspace with equal or fewer dimensions than the original one.** In the diagram given below, note the directions of maximum variance of data. This is represented using PCA1 (first maximum variance) and PC2 (2nd maximum variance).

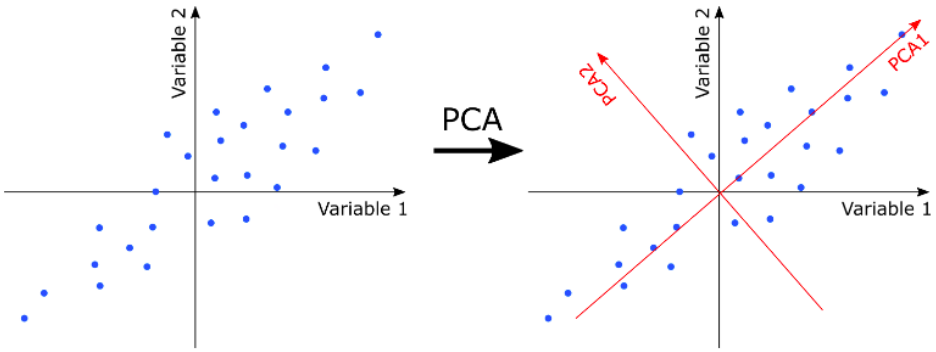


Fig 1. PCA – Directions of maximum variance

It is the direction of maximum variance of data that helps us identify an object. For example, in a movie, it is okay to identify objects by 2-dimensions as these dimensions represent direction of maximum variance. Take a look at a real-world example of understanding direction of maximum variance in the following picture representing Taj Mahal of Agra. The diagram below represents the side view of Taj Mahal. There are multiple dimensions consisting of information (maximum variance) which helps identify the picture as Taj Mahal.

Fig.2 Taj Mahal Side View

Take a look the following picture of Taj Mahal from top view. Note that there are only fewer dimensions in which information is varying and the variance is also not much. Hence, it is difficult to identify from top view whether the picture is of Taj Mahal. Thus, top view can be ignored easily.

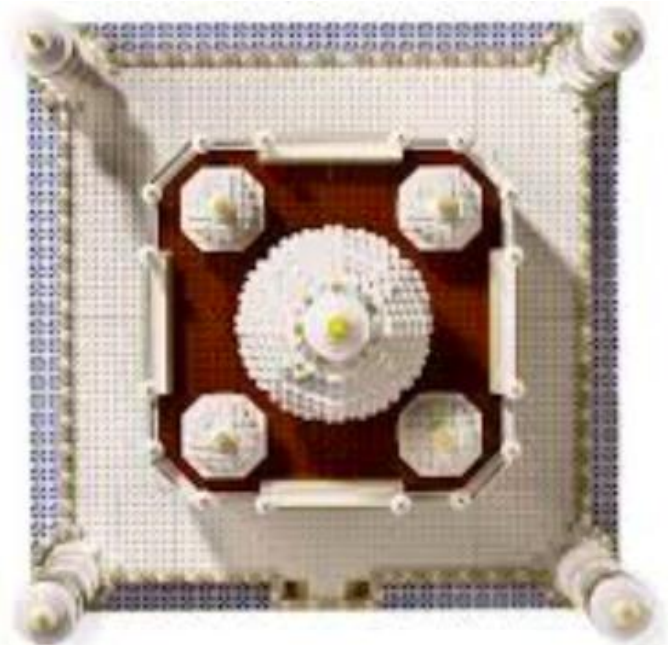


Fig3. Taj Mahal Top View

Thus, when training a model to classify whether a given structure is of Taj Mahal or not, one would want to ignore the dimensions / features related to top view as they don't provide much information (as a result of low variance).

How is PCA different than other feature selection techniques?

The way PCA is different from other feature selection techniques such as random forest, regularization techniques, forward/backward selection techniques etc is that **it does not require class labels to be present (thus called as unsupervised)**. More details along with Python code example will be shared in future posts.

Pca Algorithm for Feature Extraction:

The following represents 6 steps of principal component analysis (PCA) algorithm:

1. **Standardize the dataset:** Standardizing / normalizing the dataset is the first step one would need to take before performing PCA. The PCA calculates a new projection of the given data set representing one or more features. The new axes are based on the standard deviation of the value of these features. So, a feature / variable with a high standard deviation will have a higher weight for the calculation of axis than a variable / feature with a low standard deviation. If the data is normalized / standardized, the standard deviation of all fetaures / variables get measured on the same scale. Thus, all variables have the same weight and PCA calculates relevant axis appropriately. Note that the data is

standardized / normalized after creating training / test split. **Python's sklearn.preprocessing StandardScaler** class can be used for standardizing the dataset.

2. **Construct the covariance matrix:** Once the data is standardized, the next step is to create *n X n*-dimensional covariance matrix, where *n* is the number of dimensions in the dataset. The covariance matrix stores the pairwise covariances between the different features. Note that a positive covariance between two features indicates that the features increase or decrease together, whereas a negative covariance indicates that the features vary in opposite directions. Python' s **Numpy cov method can be used to create covariance matrix**.

3. **Perform Eigendecomposition of covariance matrix:** The next step is to decompose the covariance matrix into its eigenvectors and eigenvalues. The eigenvectors of the covariance matrix represent the principal components (the directions of maximum variance), whereas the corresponding eigenvalues will define their magnitude. **Numpy linalg.eig or linalg.eigh** can be used for decomposing covariance matrix into eigenvectors and eigenvalues.

4. **Selection of most important Eigenvectors/Eigenvalues:** Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors. Select *k* eigenvectors, which correspond to the *k* largest eigenvalues,

   where *k* is the dimensionality of the new feature subspace ( ). One can used the concepts of explained variables to select the *k* most important eigenvectors.

5. **Projection matrix creation of important eigenvectors:** Construct a projection matrix, *W*, from the top *k* eigenvectors.

6. **Training / test dataset transformation:** Finally, transform the *d*-dimensional input training and test dataset using the projection matrix to obtain the new *k*-dimensional feature subspace.

PCA Python Implementation Step-by-Step:

This section represents custom Python code for extracting the features using **PCA**.

```
In [26]: df.head()
```

Out[26]:

	gender	ssc_p	ssc_b	hsc_p	hsc_b	hsc_s	degree_p	degree_t	workex	etest_p	specialisation	mba_p	salary
0	M	67.00	Others	91.00	Others	Commerce	58.00	Sci&Tech	No	55.0	Mkt&HR	58.80	270000.0
1	M	79.33	Central	78.33	Others	Science	77.48	Sci&Tech	Yes	86.5	Mkt&Fin	66.28	200000.0
2	M	65.00	Central	68.00	Central	Arts	64.00	Comm&Mgmt	No	75.0	Mkt&Fin	57.80	250000.0
4	M	85.80	Central	73.60	Central	Commerce	73.30	Comm&Mgmt	No	96.8	Mkt&Fin	55.50	425000.0
7	M	82.00	Central	64.00	Central	Science	66.00	Sci&Tech	Yes	67.0	Mkt&Fin	62.14	252000.0

Dataset for PCA

Here are the steps followed for performing PCA:

- Perform one-hot encoding to transform categorical data set to numerical data set
- Perform training / test split of the dataset
- Standardize the training and test data set
- Construct covariance matrix of the training data set
- Construct eigendecomposition of the covariance matrix
- Select the most important features using explained variance
- Construct project matrix; In the code below, the projection matrix is created using the five eigenvectors that correspond to the top five eigenvalues (largest), to capture about 75% of the variance in this dataset
- Transform the training data set into new feature subspace

Here is the custom python code (**without using sklearn.decomposition PCA class**) to achieve the above **PCA algorithm steps** for **feature extraction:**

```
1  #
2  # Perform one-hot encoding
3  #
4  categorical_columns = df.columns[df.dtypes == object] # Find all categorical columns
5
6  df = pd.get_dummies(df, columns = categorical_columns, drop_first=True)
7  #
8  # Create training / test split
9  #
10 from sklearn.model_selection import train_test_split
11 X_train, X_test, y_train, y_test = X_train, X_test, y_train, y_test = train_test_split(df[df.columns[df.columns != 'salary']],
12 df['salary'], test_size=0.25, random_state=1)
13 #
14 # Standardize the dataset; This is very important before you apply PCA
15 #
16 from sklearn.preprocessing import StandardScaler
17 sc = StandardScaler()
18 sc.fit(X_train)
19 X_train_std = sc.transform(X_train)
20 X_test_std = sc.transform(X_test)
21 #
22 # Import eigh method for calculating eigenvalues and eigenvectirs
23 #
24 from numpy.linalg import eigh
25 #
```

```
# Determine covariance matrix
#
cov_matrix = np.cov(X_train_std, rowvar=False)
#
# Determine eigenvalues and eigenvectors
#
egnvalues, egnvectors = eigh(cov_matrix)
#
# Determine explained variance and select the most important eigenvectors based on explained variance
#
total_egnvalues = sum(egnvalues)
var_exp = [(i/total_egnvalues) for i in sorted(egnvalues, reverse=True)]
#
# Construct projection matrix using the five eigenvectors that correspond to the top five eigenvalues (largest), to capture about 75% of the variance in this dataset
#
egnpairs = [(np.abs(egnvalues[i]), egnvectors[:, i])
            for i in range(len(egnvalues))]
egnpairs.sort(key=lambda k: k[0], reverse=True)
projectionMatrix = np.hstack((egnpairs[0][1][:, np.newaxis],
                              egnpairs[1][1][:, np.newaxis],
                              egnpairs[2][1][:, np.newaxis],
                              egnpairs[3][1][:, np.newaxis],
                              egnpairs[4][1][:, np.newaxis]))
#
# Transform the training data set
#
X_train_pca = X_train_std.dot(projectionMatrix)
```

Python Sklearn Example:

This section represents Python code for extracting the features using **sklearn.decomposition** class PCA. Here is the screenshot of the data used. Salary is the label. The goal is to predict the salary.

```
In [26]: df.head()
```

Out[26]:

	gender	ssc_p	ssc_b	hsc_p	hsc_b	hsc_s	degree_p	degree_t	workex	etest_p	specialisation	mba_p	salary
0	M	67.00	Others	91.00	Others	Commerce	58.00	Sci&Tech	No	55.0	Mkt&HR	58.80	270000.0
1	M	79.33	Central	78.33	Others	Science	77.48	Sci&Tech	Yes	86.5	Mkt&Fin	66.28	200000.0
2	M	65.00	Central	68.00	Central	Arts	64.00	Comm&Mgmt	No	75.0	Mkt&Fin	57.80	250000.0
4	M	85.80	Central	73.60	Central	Commerce	73.30	Comm&Mgmt	No	96.8	Mkt&Fin	55.50	425000.0
7	M	82.00	Central	64.00	Central	Science	66.00	Sci&Tech	Yes	67.0	Mkt&Fin	62.14	252000.0

Here are the steps followed for performing PCA:

- Perform one-hot encoding to transform categorical data set to numerical data set
- Perform training / test split of the dataset
- Standardize the training and test data set
- Perform PCA by fitting and transforming the training data set to the new feature subspace and later transforming test data set.
- As a final step, the transformed dataset can be used for training/testing the model

Here is the python code to achieve the above **PCA algorithm steps** for **feature extraction**:

```
1  #
2  # Perform one-hot encoding
3  #
4  categorical_columns = df.columns[df.dtypes == object] # Find all categorical columns
5
6  df = pd.get_dummies(df, columns = categorical_columns, drop_first=True)
7  #
8  # Create training / test split
9  #
10 from sklearn.model_selection import train_test_split
11 X_train, X_test, y_train, y_test = X_train, X_test, y_train, y_test = train_test_split(df[df.columns[df.columns != 'salary']],
12                 df['salary'], test_size=0.25, random_state=1)
13 #
14 # Standardize the dataset; This is very important before you apply PCA
15 #
16 from sklearn.preprocessing import StandardScaler
17 sc = StandardScaler()
18 sc.fit(X_train)
19 X_train_std = sc.transform(X_train)
20 X_test_std = sc.transform(X_test)
21 #
22 # Perform PCA
23 #
24 from sklearn.decomposition import PCA
25 pca = PCA()
26 #
27 # Determine transformed features
28 #
29 X_train_pca = pca.fit_transform(X_train_std)
30 X_test_pca = pca.transform(X_test_std)
```

5.2 FEATURE SELECTION

Feature Selection is one of the core concepts in machine learning which hugely impacts the performance of your model. The data features that you use to train your machine learning models have a huge influence on the performance you can achieve. Irrelevant or partially relevant features can negatively impact model performance. Feature selection and Data cleaning should be the first and most important step of your model designing.

Feature Selection is the process where you automatically or manually select those features which contribute most to your prediction variable or output in which you are interested in.

Having irrelevant features in your data can decrease the accuracy of the models and make your model learn based on irrelevant features.

How to select features and what are Benefits of performing feature selection before modeling your data?

- **Reduces Overfitting**: Less redundant data means less opportunity to make decisions based on noise.
- **Improves Accuracy**: Less misleading data means modeling accuracy improves.
- **Reduces Training Time**: fewer data points reduce algorithm complexity and algorithms train faster.

I want to share my personal experience with this.

I prepared a model by selecting all the features and I got an accuracy of around 65% which is not pretty good for a predictive model and after doing some feature selection and feature engineering without doing any logical changes in my model code my accuracy jumped to 81% which is quite impressive

Now you know why I say feature selection should be the first and most important step of your model design.

Feature Selection Methods:

I will share 3 Feature selection techniques that are easy to use and also gives good results.

1. Univariate Selection
2. Feature Importance
3. Correlation Matrix with Heatmap

Let’s have a look at these techniques one by one with an example

Description of variables in the above file:

battery_power: Total energy a battery can store in one time measured in mAh

blue: Has Bluetooth or not

clock_speed: the speed at which microprocessor executes instructions

dual_sim: Has dual sim support or not

fc: Front Camera megapixels

four_g: Has 4G or not

int_memory: Internal Memory in Gigabytes

m_dep: Mobile Depth in cm

mobile_wt: Weight of mobile phone

n_cores: Number of cores of the processor

pc: Primary Camera megapixels

px_height

Pixel Resolution Height

px_width: Pixel Resolution Width

ram: Random Access Memory in MegaBytes

sc_h: Screen Height of mobile in cm

sc_w: Screen Width of mobile in cm

talk_time: The longest time that a single battery charge will last when you are

three_g: Has 3G or not

touch_screen: Has touch screen or not

wifi: Has wifi or not

price_range: This is the target variable with a value of 0(low cost), 1(medium cost), 2(high cost) and 3(very high cost).

1. Univariate Selection:

Statistical tests can be used to select those features that have the strongest relationship with the output variable.

The scikit-learn library provides the SelectKBest class that can be used with a suite of different statistical tests to select a specific number of features.

The example below uses the chi-squared (chi²) statistical test for non-negative features to select 10 of the best features from the Mobile Price Range Prediction Dataset.

```
import pandas as pd
import numpy as np
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2data =
pd.read_csv("D://Blogs//train.csv")
X = data.iloc[:,0:20]  #independent columns
y = data.iloc[:,-1]    #target column i.e price range#apply SelectKBest
class to extract top 10 best features
bestfeatures = SelectKBest(score_func=chi2, k=10)
fit = bestfeatures.fit(X,y)
dfscores = pd.DataFrame(fit.scores_)
dfcolumns = pd.DataFrame(X.columns)
#concat two dataframes for better visualization
featureScores = pd.concat([dfcolumns,dfscores],axis=1)
featureScores.columns = ['Specs','Score']  #naming the dataframe columns
print(featureScores.nlargest(10,'Score'))  #print 10 best features
```

	Specs	Score
13	ram	931267.519053
11	px_height	17363.569536
0	battery_power	14129.866576
12	px_width	9810.586750
8	mobile_wt	95.972863
6	int_memory	89.839124
15	sc_w	16.480319
16	talk_time	13.236400
4	fc	10.135166
14	sc_h	9.614878

Top 10 Best Features using SelectKBest class

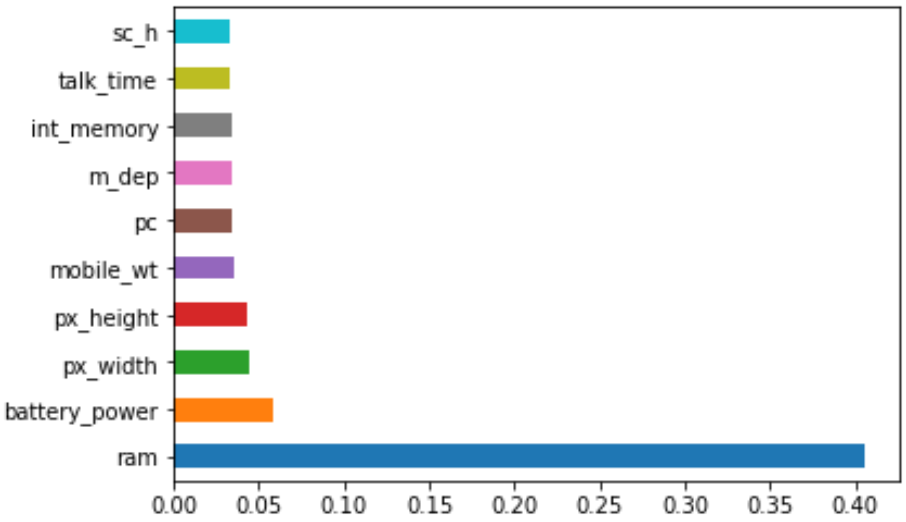
2. Feature Importance:

You can get the feature importance of each feature of your dataset by using the feature importance property of the model.

Feature importance gives you a score for each feature of your data, the higher the score more important or relevant is the feature towards your output variable.

Feature importance is an inbuilt class that comes with Tree Based Classifiers, we will be using Extra Tree Classifier for extracting the top 10 features for the dataset.

```
import pandas as pd
import numpy as np
data = pd.read_csv("D://Blogs//train.csv")
X = data.iloc[:,0:20]  #independent columns
y = data.iloc[:,-1]    #target column i.e price range
from sklearn.ensemble import ExtraTreesClassifier
import matplotlib.pyplot as plt
model = ExtraTreesClassifier()
model.fit(X,y)
print(model.feature_importances_) #use inbuilt class feature_importances
of tree based classifiers
#plot graph of feature importances for better visualization
feat_importances = pd.Series(model.feature_importances_,
index=X.columns)
feat_importances.nlargest(10).plot(kind='barh')
plt.show()
```



top 10 most important features in data

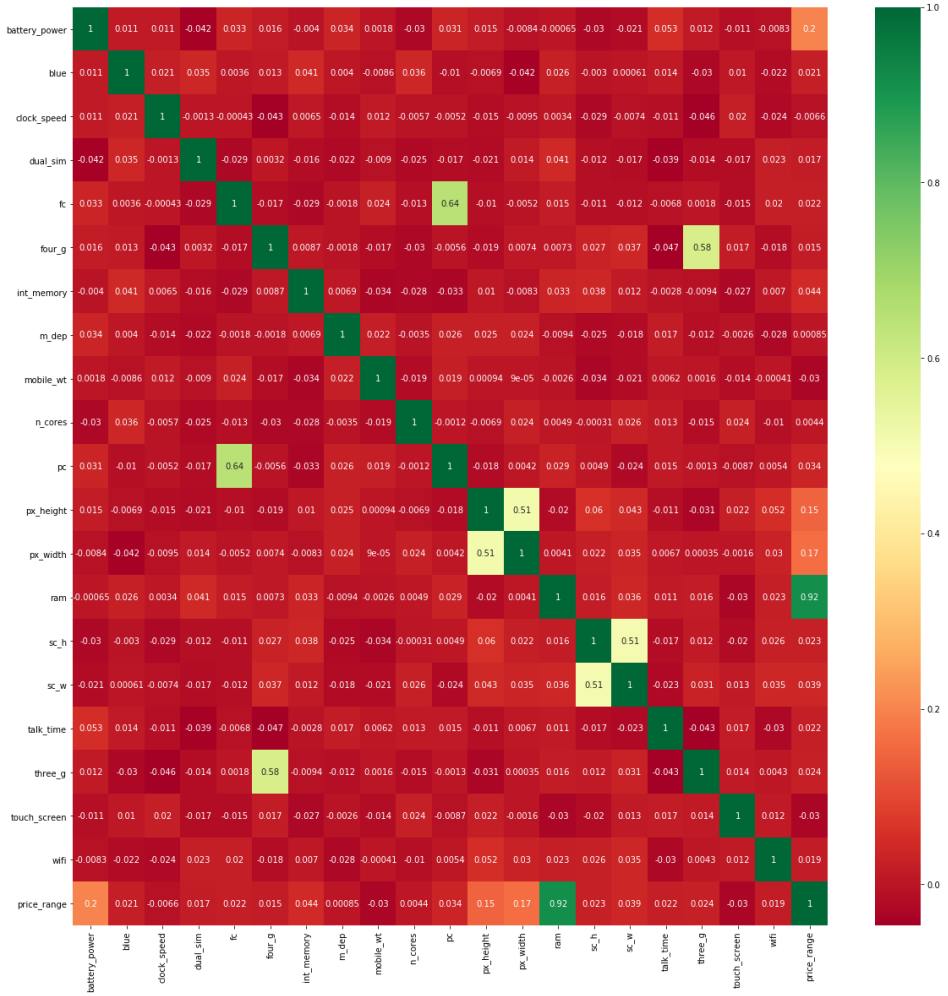
3. Correlation Matrix with Heatmap:

Correlation states how the features are related to each other or the target variable.

Correlation can be positive (increase in one value of feature increases the value of the target variable) or negative (increase in one value of feature decreases the value of the target variable)

Heatmap makes it easy to identify which features are most related to the target variable, we will plot heatmap of correlated features using the seaborn library.

```
import pandas as pd
import numpy as np
import seaborn as snsdata = pd.read_csv("D://Blogs//train.csv")
X = data.iloc[:,0:20]  #independent columns
y = data.iloc[:,-1]    #target column i.e price range
#get correlations of each features in dataset
corrmat = data.corr()
top_corr_features = corrmat.index
plt.figure(figsize=(20,20))
#plot heat map
g=sns.heatmap(data[top_corr_features].corr(),annot=True,cmap="RdYlGn
")
```



5.3 NORMALIZATION

Normalization is a technique often applied as part of data preparation for machine learning. The goal of normalization is to change the values of numeric columns in the dataset to use a common scale, without distorting

differences in the ranges of values or losing information. Normalization is also required for some algorithms to model the data correctly.

For example, assume your input dataset contains one column with values ranging from 0 to 1, and another column with values ranging from 10,000 to 100,000. The great difference in the *scale* of the numbers could cause problems when you attempt to combine the values as features during modelling.

Normalization avoids these problems by creating new values that maintain the general distribution and ratios in the source data, while keeping values within a scale applied across all numeric columns used in the model.

This component offers several options for transforming numeric data:

- You can change all values to a 0-1 scale, or transform the values by representing them as percentile ranks rather than absolute values.
- You can apply normalization to a single column, or to multiple columns in the same dataset.
- If you need to repeat the pipeline, or apply the same normalization steps to other data, you can save the steps as a normalization transform, and apply it to other datasets that have the same schema.

Normalization Techniques at a Glance:

Four common normalization techniques may be useful:

- scaling to a range
- clipping
- log scaling
- z-score

The following charts show the effect of each normalization technique on the distribution of the raw feature (price) on the left. The charts are based on the data set from 1985 Ward's Automotive Yearbook that is part of the UCI Machine Learning Repository under Automobile Data Set.

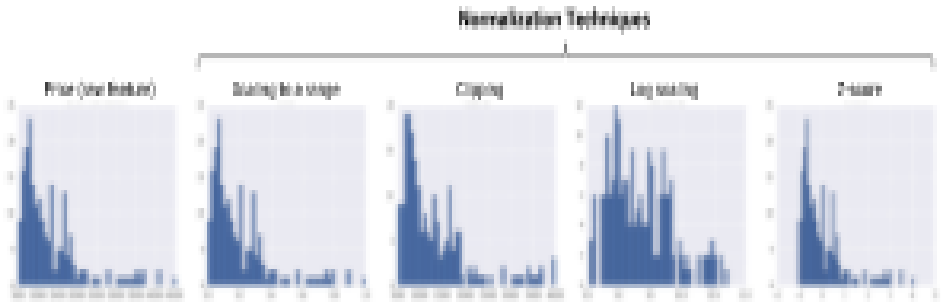


Figure 1. Summary of normalization techniques.

Scaling to a Range:

Recall from MLCC that **scaling** means converting floating-point feature values from their natural range (for example, 100 to 900) into a standard range—usually 0 and 1 (or sometimes -1 to +1). Use the following simple formula to scale to a range:

\[x' = (x - x_{min}) / (x_{max} - x_{min}) \]

Scaling to a range is a good choice when both of the following conditions are met:

- You know the approximate upper and lower bounds on your data with few or no outliers.
- Your data is approximately uniformly distributed across that range.

A good example is age. Most age values falls between 0 and 90, and every part of the range has a substantial number of people.

In contrast, you would *not* use scaling on income, because only a few people have very high incomes. The upper bound of the linear scale for income would be very high, and most people would be squeezed into a small part of the scale.

Feature Clipping:

If your data set contains extreme outliers, you might try feature clipping, which caps all feature values above (or below) a certain value to fixed value. For example, you could clip all temperature values above 40 to be exactly 40.

You may apply feature clipping before or after other normalizations.

Formula: Set min/max values to avoid outliers:

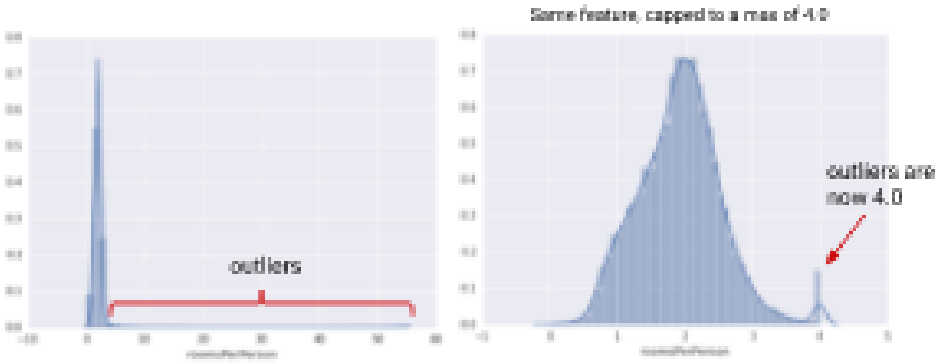


Figure 2. Comparing a raw distribution and its clipped version.

Another simple clipping strategy is to clip by z-score to +-Nσ (for example, limit to +-3σ). Note that σ is the standard deviation.

Log Scaling:

Log scaling computes the log of your values to compress a wide range to a narrow range.

$$x' = \log(x)$$

Log scaling is helpful when a handful of your values have many points, while most other values have few points. This data distribution is known as the *power law* distribution. Movie ratings are a good example. In the chart below, most movies have very few ratings (the data in the tail), while a few have lots of ratings (the data in the head). Log scaling changes the distribution, helping to improve linear model performance.

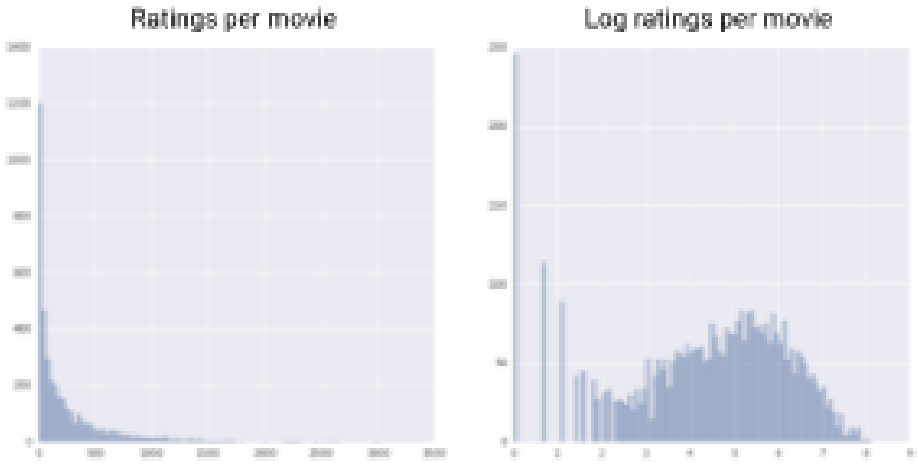


Figure 3. Comparing a raw distribution to its log.

Z-Score:

Z-score is a variation of scaling that represents the number of standard deviations away from the mean. You would use z-score to ensure your feature distributions have mean = 0 and std = 1. It's useful when there are a few outliers, but not so extreme that you need clipping.

The formula for calculating the z-score of a point, *x*, is as follows:

$$x' = (x - \mu) / \sigma$$

Note: μ is the mean and σ is the standard deviation.

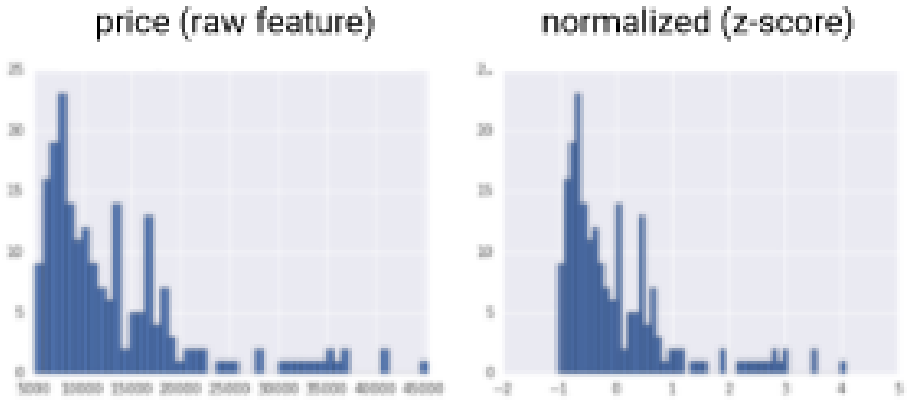


Figure 4. Comparing a raw distribution to its z-score distribution.

Notice that z-score squeezes raw values that have a range of ~40000 down into a range from roughly -1 to +4.

Suppose you're not sure whether the outliers truly are extreme. In this case, start with z-score unless you have feature values that you don't want the model to learn; for example, the values are the result of measurement error or a quirk.

Configure Normalize Data:

You can apply only one normalization method at a time using this component. Therefore, the same normalization method is applied to all columns that you select. To use different normalization methods, use a second instance of **Normalize Data**.

1. Add the **Normalize Data** component to your pipeline. You can find the component In Azure Machine Learning, under **Data Transformation**, in the **Scale and Reduce** category.

2. Connect a dataset that contains at least one column of all numbers.

3. Use the Column Selector to choose the numeric columns to normalize. If you don't choose individual columns, by default **all** numeric type columns in the input are included, and the same normalization process is applied to all selected columns.

This can lead to strange results if you include numeric columns that shouldn't be normalized! Always check the columns carefully.

If no numeric columns are detected, check the column metadata to verify that the data type of the column is a supported numeric type.

Tip:

To ensure that columns of a specific type are provided as input, try using the **Select Columns in Dataset** component before **Normalize Data**.

4. **Use 0 for constant columns when checked:** Select this option when any numeric column contains a single unchanging value. This ensures that such columns are not used in normalization operations.

5. From the **Transformation method** dropdown list, choose a single mathematical function to apply to all selected columns.

- **Zscore:** Converts all values to a z-score.

The values in the column are transformed using the following formula:

$$z = \frac{x - mean(x)}{stdev(x)}$$

Mean and standard deviation are computed for each column separately. Population standard deviation is used.

- **MinMax:** The min-max normalizer linearly rescales every feature to the [0,1] interval.

Rescaling to the [0,1] interval is done by shifting the values of each feature so that the minimal value is 0, and then dividing by the new maximal value (which is the difference between the original maximal and minimal values).

The values in the column are transformed using the following formula:

$$z = \frac{x - min(x)}{[\max(x) - min(x)]}$$

- **Logistic:** The values in the column are transformed using the following formula:

$$z = \frac{1}{1 + \exp(-x)}$$

- **LogNormal:** This option converts all values to a lognormal scale.

The values in the column are transformed using the following formula:

$$z = Lognormal.CDF(x;;\mu,\sigma)$$

Here μ and σ are the parameters of the distribution, computed empirically from the data as maximum likelihood estimates, for each column separately.

- **TanH:** All values are converted to a hyperbolic tangent.

The values in the column are transformed using the following formula:

$$p(k|x;\theta) = \frac{[E(Y|x)]^k e^{-E(Y|x)}}{k!}$$

6. Submit the pipeline, or double-click the **Normalize Data** component and select **Run Selected**.

Data Normalization with Pandas:

- **Pandas:** Pandas is an open-source library that's built on top of NumPy library. it is a Python package that provides various data structures and operations for manipulating numerical data and statistics. It's mainly popular for importing and analysing data much easier. Pandas is fast and it's high-performance & productive for users.

- **Data Normalization**: Data Normalization could also be a typical practice in machine learning which consists of transforming numeric columns to a standard scale. In machine learning, some feature values differ from others multiple times. The features with higher values will dominate the learning process.

Steps Needed:

Here, we will apply some techniques to normalize the data and discuss these with the help of examples. For this, let's understand the steps needed for data normalization with Pandas.

1. Import Library (Pandas)

2. Import / Load / Create data.

3. Use the technique to normalize the data.

Examples:

Here, we create data by some random values and apply some normalization techniques to it.

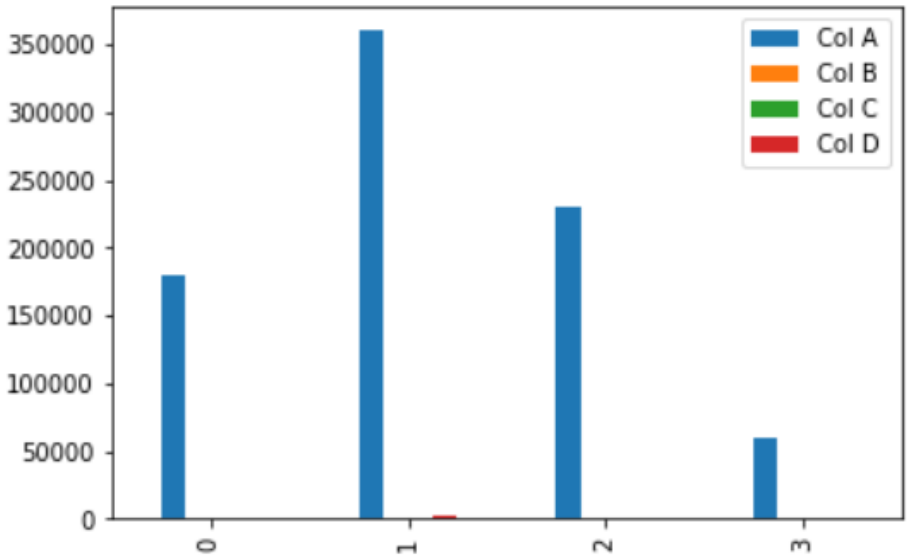
```
# importing packages
import pandas as pd

# create data
df = pd.DataFrame([
                [180000, 110, 18.9, 1400],
                [360000, 905, 23.4, 1800],
                [230000, 230, 14.0, 1300],
                [60000, 450, 13.5, 1500]],
                columns=['Col A', 'Col B',
                        'Col C', 'Col D'])
# view data
display(df)
```

Output:

	Col A	Col B	Col C	Col D
0	180000	110	18.9	1400
1	360000	905	23.4	1800
2	230000	230	14.0	1300
3	60000	450	13.5	1500

See the plot of this dataframe:

```
import matplotlib.pyplot as plt
df.plot(kind = 'bar')
```



Let's apply normalization techniques one by one.

Using The maximum absolute scaling:

The maximum absolute scaling rescales each feature between -1 and 1 by dividing every observation by its maximum absolute value. We can apply the maximum absolute scaling in Pandas using the .max() and .abs() methods, as shown below:

```
# copy the data
df_max_scaled = df.copy()

# apply normalization techniques
for column in df_max_scaled.columns:
    df_max_scaled[column] = df_max_scaled[column]  /
df_max_scaled[column].abs().max()

# view normalized data
display(df_max_scaled)
```

Output:

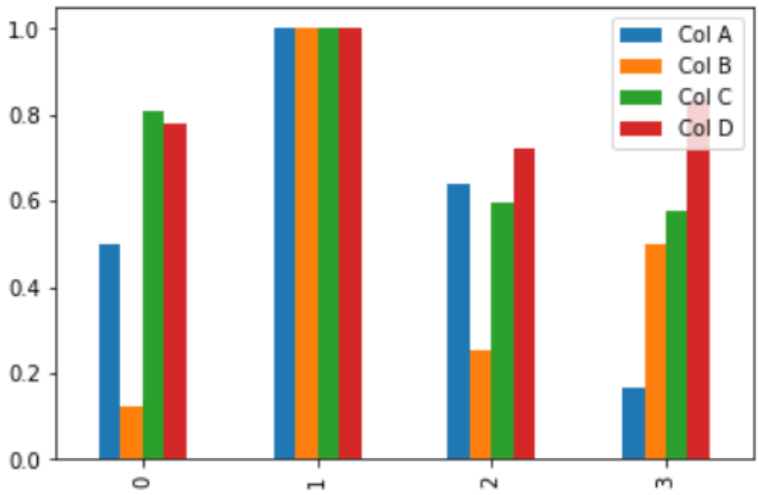
	Col A	Col B	Col C	Col D
0	0.500000	0.121547	0.807692	0.777778
1	1.000000	1.000000	1.000000	1.000000
2	0.638889	0.254144	0.598291	0.722222
3	0.166667	0.497238	0.576923	0.833333

See the plot of this dataframe:

```
import matplotlib.pyplot as plt
df_max_scaled.plot(kind = 'bar')
```


```
import matplotlib.pyplot as plt
df_max_scaled.plot(kind = 'bar')
```


Output:



Using The min-max feature scaling:

The min-max approach (often called normalization) rescales the feature to a hard and fast range of [0,1] by subtracting the minimum value of the feature then dividing by the range. We can apply the min-max scaling in Pandas using the .min() and .max() methods.

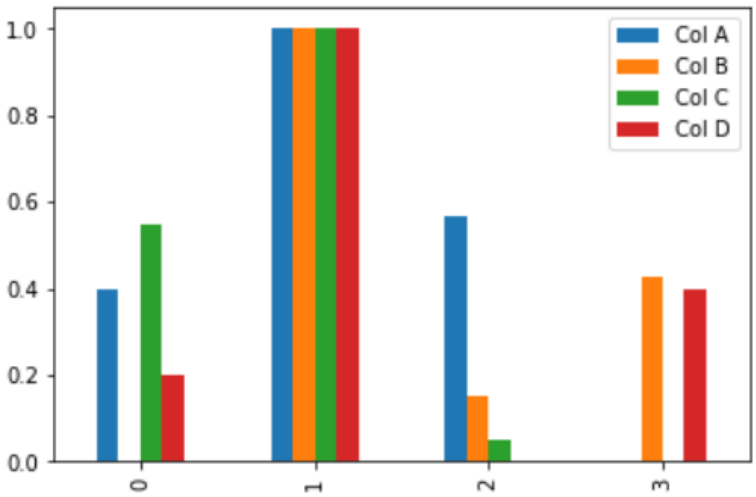
```
# copy the data
df_min_max_scaled = df.copy()
# apply normalization techniques
for column in df_min_max_scaled.columns:
df_min_max_scaled[column] = (df_min_max_scaled[column] -
df_min_max_scaled[column].min()) /
(df_min_max_scaled[column].max() -
df_min_max_scaled[column].min())
# view normalized data
print(df_min_max_scaled)
```

Output:

	Col A	Col B	Col C	Col D
0	0.400000	0.000000	0.545455	0.2
1	1.000000	1.000000	1.000000	1.0
2	0.566667	0.150943	0.050505	0.0
3	0.000000	0.427673	0.000000	0.4

Let's draw a plot with this dataframe:

```
import matplotlib.pyplot as plt

df_min_max_scaled.plot(kind = 'bar')
```



Using The z-score method:

The z-score method (often called standardization) transforms the info into distribution with a mean of 0 and a typical deviation of 1. Each standardized value is computed by subtracting the mean of the corresponding feature then dividing by the quality deviation.

```
# copy the data

df_z_scaled = df.copy()

# apply normalization techniques

for column in df_z_scaled.columns:

    df_z_scaled[column] = (df_z_scaled[column] -
```

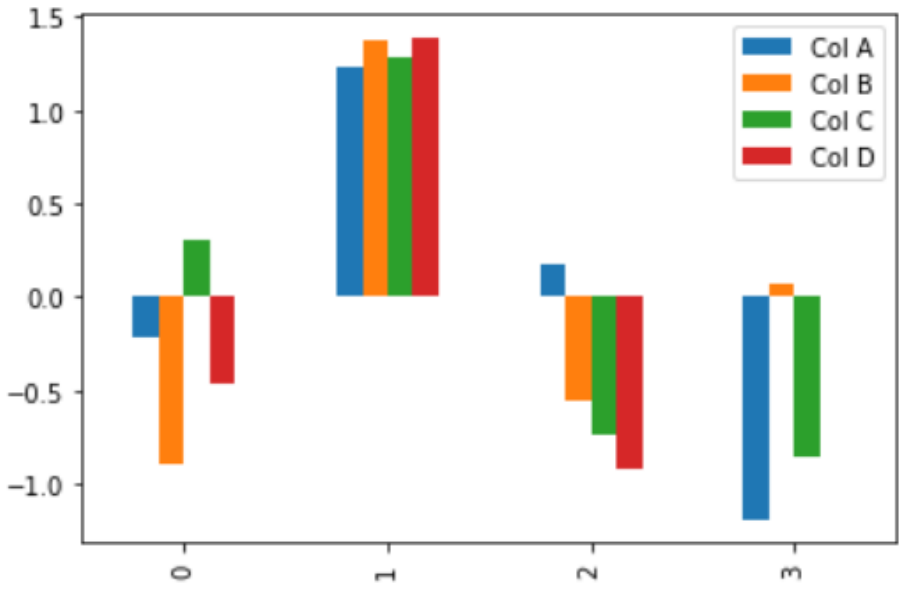
```
                    df_z_scaled[column].mean())                    /
df_z_scaled[column].std()

# view normalized data

display(df_z_scaled)
```

Output:

	Col A	Col B	Col C	Col D
0	-0.221422	-0.895492	0.311486	-0.46291
1	1.227884	1.373564	1.278167	1.38873
2	0.181163	-0.552993	-0.741122	-0.92582
3	-1.187625	0.074922	-0.848531	0.00000

Let's draw a plot with this dataframe:

```
import matplotlib.pyplot as plt

df_z_scaled.plot(kind='bar')
```



*****

6

TRANSFORMATION

Unit Structure

6.1 INTRODUCTION

What is AI Transformation?:

AI transformation is the next step after digital transformation. After a company adopts digital processes, the next step is to improve the intelligence of those processes. This would increase the level of automation as well as the effectiveness of those processes.

AI transformation touches all aspects of the modern enterprise including both commercial and operational activities. Tech giants are integrating AI into their processes and products. For example, Google is calling itself an "AI-first" organization. Besides tech giants, IDC estimates that at least 90% of new organizations will insert AI technology into their processes and products by 2025.

What are the steps to AI transformation?:

We have listed below a set of the top 6 steps for Fortune 500 firms. Smaller firms could skip having in-house teams and strive for less risky and less investment heavy approaches such as relying on consultants for targeted projects.

1. Outline your company's AI strategy:

An AI strategy should include initiatives which will be uncovered as a result of these exercises:

- Identify your company's most valuable unique data sources
- Identify the most important processes which can benefit from automation
- Identify internal resources to drive the AI transformation
- Set ambitious, time-bound business targets

2. Execute pilot projects to gain momentum:

First few projects should create measurable business value while being attainable. This is important for the transformation to gain trust across the organization with achieved projects and it creates momentum that will lead to AI projects with greater success.

These projects can rely on AI/ML powered tools in the marketplace or for more custom solutions, your company can run a data science competition and rely on the wisdom of hundreds of data scientists. These competitions use encrypted data and provide a low cost way to find high performing data science solutions.

Implementing process mining is one of those easy-to-achieve and impactful projects. With a process mining tool, your business can identify existing inefficiencies and automate or improve those processes to achieve savings or customer experience improvement. Thus, some process mining tools generate a digital twin of an organization (DTO) which provides an end-to-end overview of the processes in the company and offers simulation capabilities to compare actual and hypothetical scenarios.

Another easy-to-deploy and impactful project is automating document-based processes. While digital transformation projects in the 2000s just dealt with removing paper from processes, a modern AI/digital transformation project would reduce manual labour and automate data extraction and processing of document data.

3. Build an in-house AI transformation team:

Outsourcing the AI work eases the start of the AI transformation process but building an in-house AI transformation team can be more advantageous in the long run. If necessary, outsourced partners can help train your staff for upcoming projects.

4. Provide broad AI training:

Organizations should not expect adequate knowledge about AI technologies from their staff. In order to have a successful AI transformation, training each employee in accordance with their role can be beneficial to achieve objectives.

- Executives and seniors should have knowledge about what AI can do for the enterprise, how to develop an AI strategy and make proper resource allocation decisions.

- Leaders of AI project teams should learn how to set direction for AI projects, allocate resources, monitor and track progress.

- AI engineers should learn how to gather data, train AI models, and deliver specific AI projects.

5. Develop internal and external communications:

For the road to success in AI transformation, the organization should ensure alignment across the business by improving internal and external communication.

6. Update the company's AI strategy and continue with AI transformation:

When the team gains momentum from the initial AI projects and forms a deeper understanding of AI, the organization will have a better understanding of improvement areas where AI can create the most value. An updated strategy that considers the company's track record can set a better direction for the company.

Here are the four types of transformation in more detail:

Process Transformation:

A significant focus of corporate activity has been in business processes. Data, analytics, APIs, machine learning and other technologies offer corporations valuable new ways to reinvent processes throughout the corporation—with the goal of lowering costs, reducing cycle times, or increasing quality. We see process transformation on the shop floor where companies like Airbus have engaged heads-up display glasses to improve the quality of human inspection of airplanes. We also see process transformations in customer experience, where companies like Domino's Pizza have completely re-imagined the food ordering process; Dominos' AnyWare lets customers order from any device. This innovation increased customer convenience so much that it helped push the company to overtake Pizza Hut in sales. And we see companies implementing technologies like robotic process automation to streamline back office processes like accounting and legal, for example. Process transformation can create significant value and adopting technology in these areas is fast becoming table-stakes. Because these transformations tend to be focused efforts around specific areas of the business, they are often successfully led by a CIO or CDO.

Business Model Transformation:

Some companies are pursuing digital technologies to transform traditional business models. Whereas process transformation focuses on finite areas of the business, business model transformations are aimed at the fundamental building blocks of how value is delivered in the industry. Examples of this kind of innovation are well-known, from Netflix' reinvention of video distribution, to Apple's reinvention of music delivery (I-Tunes), to Uber's reinvention of the taxi industry. But this kind of transformation is occurring elsewhere. Insurance companies like Allstate and Metromile are using data and analytics to un-bundle insurance contracts and charge customers by-the-mile—a wholesale change to the auto insurance business model. And, though not yet a reality, there are

numerous efforts underway to transform the business of mining to a wholly robotic exercise, where no humans travel below the surface.

The complex and strategic nature of these opportunities require involvement and leadership by Strategy and/or Business Units and they are often launched as separate initiatives while continuing to operate the traditional business. By changing the fundamental building blocks of value, corporations that achieve business model transformation open significant new opportunities for growth. More companies should pursue this path.

Domain Transformation:

An area where we see surprisingly little focus—but enormous opportunity—is the area of domain transformation. New technologies are redefining products and services, blurring industry boundaries and creating entirely new sets of non-traditional competitors. What many executives don't appreciate is the very real opportunity for these new technologies to unlock wholly new businesses for their companies beyond currently served markets. And often, it is this type of transformation is that offers the greatest opportunities to create new value.

A clear example how domain transformation works may be the online retailer, Amazon. Amazon expanded into a new market domain with the launch of Amazon Web Services (AWS), now the largest cloud computing/infrastructure service, in a domain formerly owned by the IT giants like Microsoft and IBM. What made Amazon's entry into this domain possible was a combination of the strong digital capabilities it had built in storage, computing databases to support its core retail business coupled with an installed base of thousands of relationships with young, growing companies that increasingly needed computing services to grow. AWS is not a mere adjacency or business extension for Amazon, but a wholly different business in a fundamentally different market space. The AWS business now represents nearly 60% % of Amazon's annual profit.

It may be tempting for Executives of non-tech businesses to view the experience of Amazon or other digitally-native companies (such as Apple or Google that have also expanded into new domains) as special; their ability to acquire and leverage technology may be greater than other companies. But in today's digital world, technology gaps are no longer a barrier. Any company can access and acquire the new technologies needed to unlock new growth—and do so cheaply and efficiently. The building block technologies that are unlocking new business domains (artificial intelligence, machine learning, internet of things (IOT), augmented reality, etc.) can be sourced today not only from the traditional IT supply-base like Microsoft or IBM but also from a growing startup ecosystem, where we see the greatest innovation taking place. Corporations that know how to reach and leverage this innovation efficiently, particularly from new sources, are reaping the benefits of new growth.

We see (and have helped) numerous industrial companies that have undergone domain transformations. ThyssenKrupp, a diversified industrial engineering company, broadened its offerings to introduce a lucrative new digital business alongside its traditional business. The company leveraged a strong industrial market position and Internet of Things (IOT) capabilities to help clients manage the maintenance of elevators with asset health and predictive maintenance offerings—creating a significant new source of revenue beyond the core. In another example, a major equipment manufacturer is moving beyond its core machine offerings to introduce a digital platform of solutions for its client sites: job-site activity coordination, remote equipment tracking, situational awareness, and supply chain optimization. The company is moving to become no longer merely a heavy equipment provider, but also a digital solutions company.

The lesson is to recognize the new domain opportunities afforded by new technologies and understand they can be captured—even by traditional incumbents. Because these opportunities involve re-defining business boundaries, pursuing these opportunities often involves Strategy and the CEO.

Cultural/Organizational Transformation:

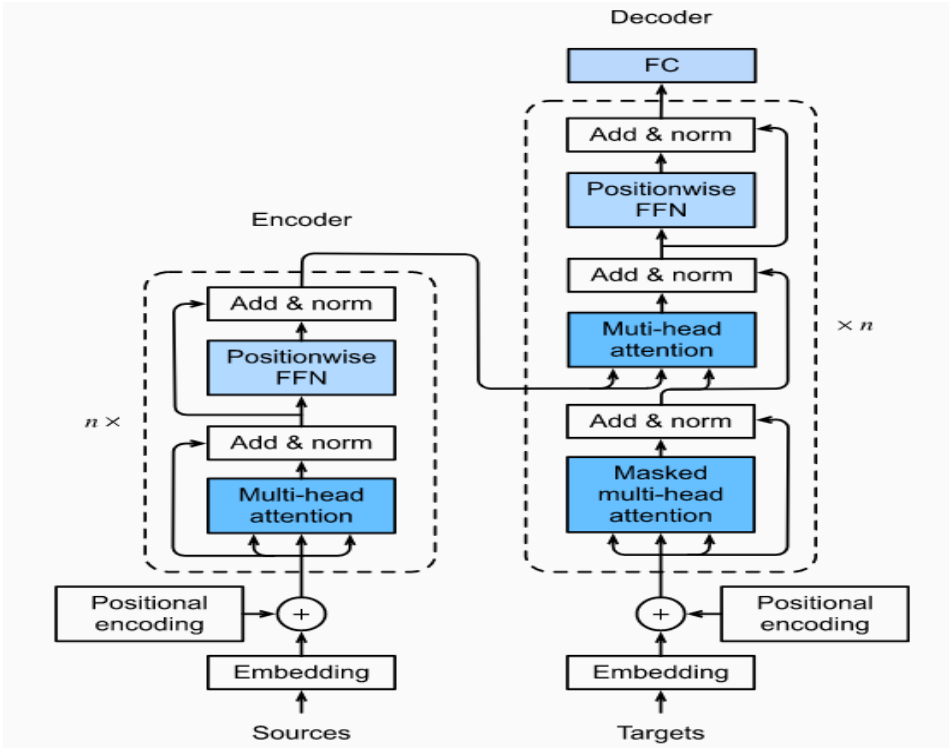
Full, long-term digital transformation requires redefining organizational mindsets, processes, and talent & capabilities for the digital world. Best-in-class corporations recognize digital requires agile workflows, a bias toward testing and learning, decentralized decision-making, and a greater reliance on business ecosystems. And they take active steps to bring change to their organizations. Experian, the consumer credit agency and one of the most successful digital transformations, changed its organization by embedding agile development and collaboration into its workflows and by driving a fundamental shift in employee focus from equipment to data, company-wide. Similarly, Pitney Bowes, the 100-year old postage equipment company, made the successful transition to become a "technology company" by promoting a "culture of innovation," according to its head of innovation, and by shifting company values to focus on customer-centricity.

But neither of these companies focused initially on organization and culture--being digital isn't the same as creating value from digital. Instead, these companies pulled innovation skills, digital mindsets and agility into the corporation on the back of concrete initiatives to drive growth. Experian recognized the importance of beginning with a lighthouse digital project to create internal APIs. It forced teams to adopt digital workflow practices but in doing so demonstrated the power of digital to change old organizational norms. Similarly, Pitney Bowes CEO Mark Lautenbach began its transformation with a primary focus on customer-facing offerings, developing new commerce cloud to allow customers to better manage and pay for shipments. "As you're thinking about transforming a company… try to realize those cores, those gems that you have that you can pivot off of to create that next chapter," he told Fortune. Progress on business initiatives dragged organizational change

like agile development and innovation along. Cultural/organizational change is a long-term requirement of success, but best in class companies regard the building of these capabilities as a product of, rather than a prerequisite for, business transformation initiatives.

As technology change increases, industries will continue to be forced to change. Corporations that regard and pursue digital transformation in a multi-dimensional way will find greater success than those that don't.

6.2 TRANSFORMERS



Transformers can be understood in terms of their three components:

1. An Encoder that encodes an input sequence into state representation vectors.
2. An Attention mechanism that enables our Transformer model to focus on the right aspects of the sequential input stream. This is used repeatedly within both the encoder and the decoder to help them contextualize the input data.
3. A Decoder that decodes the state representation vector to generate the target output sequence.

Understanding the Training Data:

Sample data Point: "write a function that adds two numbers":

Python Code:

```
def add_two_numbers (num1 ,num2 ):
   sum = num1 + num2
            return sum
```

Tokenizing the Data:

Our Input(SRC) and Output(TRG) sequence exist in the form of single strings that need to be further tokenized in order to be sent into the transformer model.

To tokenize the Input sequence we make use of spacy.

```
Input = data.Field(tokenize = 'spacy',
          init_token='<sos>',
          eos_token='<eos>',
          lower=True)
```

To tokenize our Output sequence we make use of our custom tokenizer built upon Python's source code tokenizer. Python's tokenizer returns several attributes for each token. We only extract the token type and the corresponding string attribute in form of a tuple(i.e., (token_type_int, token_string)) as the final token.

Tokenized Input:

```
SRC = [' ', 'write', 'a', 'python', 'function', 'to', 'add', 'two', 'user', 'provided',
 'numbers', 'and', 'return', 'the', 'sum']
```

Tokenized Output:

```
TRG = [(57, 'utf-8'), (1, 'def'), (1, 'add_two_numbers'), (53, '('), (1, 'num1'),
(53, ','), (1, 'num2'), (53, ')'), (53, ':'), (4, '\n'), (5, '    '), (1, 'sum'), (53, '='),
(1, 'num1'), (53, '+'), (1, 'num2'), (4, '\n'), (1, 'return'), (1, 'sum'), (4, ''), (6,
''), (0, '')]
```

Data Augmentations:

While tokenizing the python code, we mask the names of certain variables randomly(with 'var_1, 'var_2' etc) to ensure that the model that we train does not merely fixate on the way the variables are named and actually tries to understand the inherent logic and syntax of the python code.

For example, consider the following program.

```
def add_two_numbers (num1 ,num2 ):

   sum = num1 + num2

            return sum
```

We can replace some of the above variables to create new data points. The following are valid augmentations.

1.

```
def add_two_numbers (var_1 ,num2 ):
   sum = var_1 + num2
            return sum
```

2.

```
def add_two_numbers (num1 ,var_1 ):
  sum = num1 + var_1
           return sum
```

3.

```
def add_two_numbers (var_1 ,var_2 ):
   sum = var_1 + var_2
            return sum
```

In the above example, we have therefore expanded a single data point into 3 more data points using our random variable replacement technique.

We implement our augmentations at the time of generating our tokens.

While randomly picking variables to mask we avoid keyword literals(*keyword.kwlist*), control structures(as can be seen in below *skip_list*), and object properties. We add all such literals that need to be skipped into the *skip_list.*

We now apply our augmentations and tokenization using Pytorch's torchtext.data.Field**.**

```
Output = data.Field(tokenize = augment_tokenize_python_code,
                    init_token='<sos>',
                    eos_token='<eos>',
                    lower=False)
```

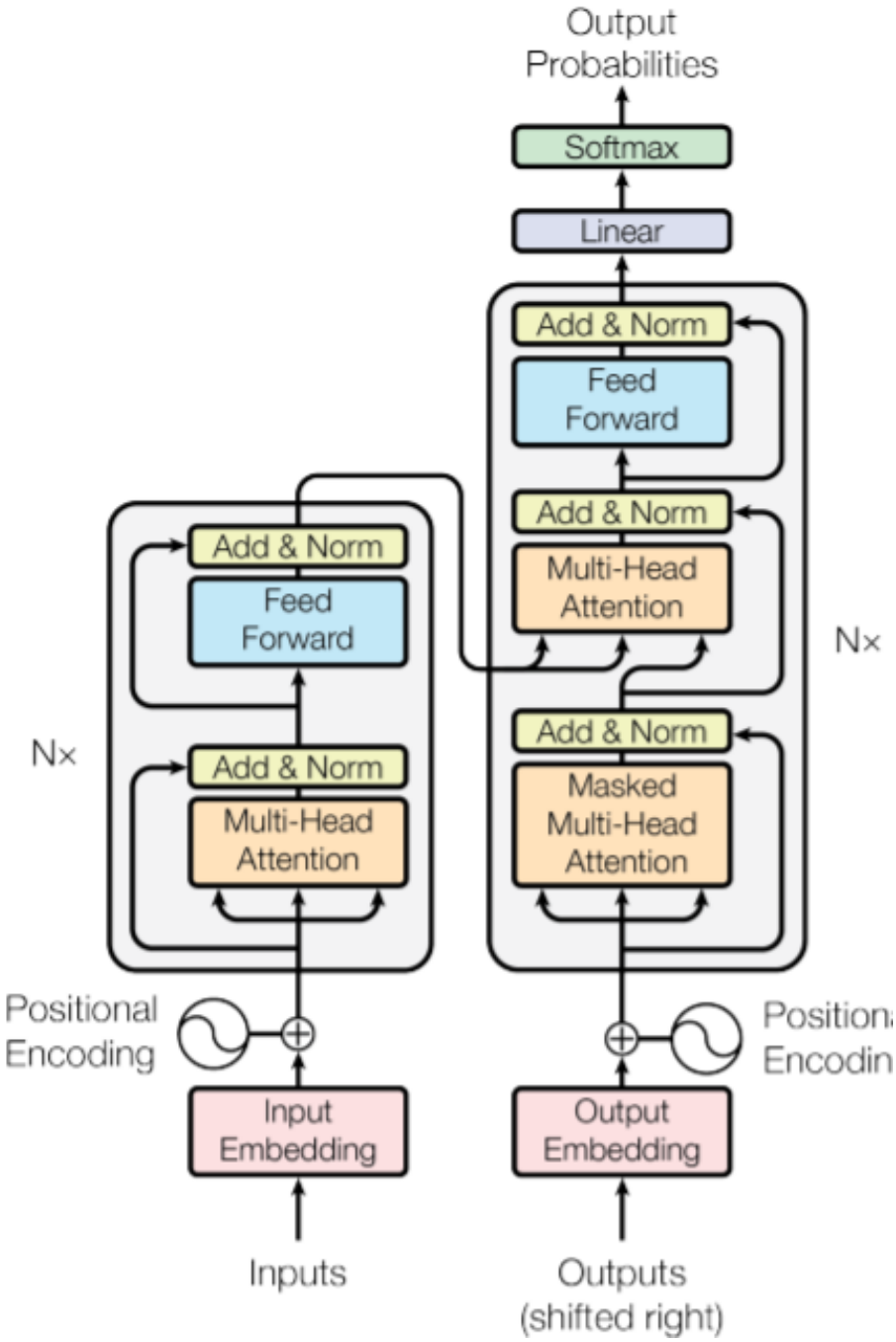
Our tokenized Output after applying tokenization:

```
TRG = [(57, 'utf-8'), (1, 'def'), (1, 'add_two_numbers'), (53, '('), (1, 'num1'), (53, ','), (1, 'var_1'), (53, ')'), (53, ':'), (4, '\n'), (5, '    '), (1, 'sum'), (53, '='),
```

(1, 'num1'), (53, '+'), (1, 'var_1'), (4, '\n'), (1, 'return'), (1, 'sum'), (4, ''), (6, ''), (0, '')]

Feeding Data:

To feed data into our model we first create batches. The tokenized predictions are then untokenized via the untokenize function of Python's source code tokenizer.



Loss Function:

We have used augmentations in our dataset to mask variable literals. This means that our model can predict a variety of values for a particular variable and all of them are correct as long as the predictions are

consistent through the code. This would mean that our training labels are not very certain and hence it would make more sense to treat them to be correct with probability *1- smooth_eps* and incorrect otherwise. This is what label smoothening does. By adding label smoothening to Cross-Entropy we ensure that the model does not become too confident in predicting some of our variables that can be replaced via augmentations.

> **Label Smoothing** is a regularization technique that introduces noise for the labels. This accounts for the fact that datasets may have mistakes in them, so maximizing the likelihood of $\log p(y \mid x)$ directly can be harmful. Assume for a small constant ϵ, the training set label y is correct with probability $1 - \epsilon$ and incorrect otherwise. Label Smoothing regularizes a model based on a softmax with k output values by replacing the hard 0 and 1 classification targets with targets of $\frac{\epsilon}{k-1}$ and $1 - \epsilon$ respectively.

Now with all our components set we can train our model using backpropagation. We split our dataset into training and validation data. Our model is trained until our validation loss does not improve any further.

It is important to note that label smoothening leads to much higher loss values as compared to models that do not make use of label smoothening. But this is as expected as we do not intend to be certain with our label predictions. This is particularly the case with variables as there can be multiple correct options as long as the predictions are consistent through the target code sequence.

Sample Results:

Input: "program to sort a list of dictionaries by key"
Output:

```
var_1 ={'Nikhil':{'roll':24 ,'marks':17 },
'Akshat':{'roll':54 ,'marks':12 },
'Akash':{'roll':15 },'marks':15 }}
sort_key ='marks'
res ='marks'
res =var_2 (test_dict .items (),key =lambda x :x [1 ][sort_key ])
print ("The sorted dictionary by marks is : "+str (res ))
```

Input: "function to sum odd elements of list"

***Output*:**

```
def sum_odd_elements (l :list ):
   return sum ([i for i in l if i %2 = =1 ])
```

Input: "program to reverse a string"

***Output*:**

```
var_1 = 'Today is bad day'
var_1 [::-1 ]
```

6.3 PRINCIPLE COMPONENTS ANALYSIS (PCA):

Principal Component Analysis is an unsupervised learning algorithm that is used for the dimensionality reduction in machine learning. It is a statistical process that converts the observations of correlated features into a set of linearly uncorrelated features with the help of orthogonal transformation. These new transformed features are called the **Principal Components**. It is one of the popular tools that is used for exploratory data analysis and predictive modelling. It is a technique to draw strong patterns from the given dataset by reducing the variances.

PCA generally tries to find the lower-dimensional surface to project the high-dimensional data.

PCA works by considering the variance of each attribute because the high attribute shows the good split between the classes, and hence it reduces the dimensionality. Some real-world applications of PCA are ***image processing, movie recommendation system, optimizing the power allocation in various communication channels.*** It is a feature extraction technique, so it contains the important variables and drops the least important variable.

The PCA algorithm is based on some mathematical concepts such as:

- Variance and Covariance
- Eigenvalues and Eigen factors

Some common terms used in PCA algorithm:

- **Dimensionality:** It is the number of features or variables present in the given dataset. More easily, it is the number of columns present in the dataset.
- **Correlation:** It signifies that how strongly two variables are related to each other. Such as if one changes, the other variable also gets changed. The correlation value ranges from -1 to +1. Here, -1 occurs if variables are inversely proportional to each other, and +1 indicates that variables are directly proportional to each other.
- **Orthogonal:** It defines that variables are not correlated to each other, and hence the correlation between the pair of variables is zero.
- **Eigenvectors:** If there is a square matrix M, and a non-zero vector v is given. Then v will be eigenvector if Av is the scalar multiple of v.
- **Covariance Matrix:** A matrix containing the covariance between the pair of variables is called the Covariance Matrix.

Principal Components in PCA:

As described above, the transformed new features or the output of PCA are the Principal Components. The number of these PCs are either equal to or less than the original features present in the dataset. Some properties of these principal components are given below:

- The principal component must be the linear combination of the original features.
- These components are orthogonal, i.e., the correlation between a pair of variables is zero.
- The importance of each component decreases when going to 1 to n, it means the 1 PC has the most importance, and n PC will have the least importance.

Steps for PCA Algorithm:

1. **Getting the dataset:** Firstly, we need to take the input dataset and divide it into two subparts X and Y, where X is the training set, and Y is the validation set.
2. **Representing data into a structure:** Now we will represent our dataset into a structure. Such as we will represent the two-dimensional matrix of independent variable X. Here each row corresponds to the data items, and the column corresponds to the Features. The number of columns is the dimensions of the dataset.
3. **Standardizing the data:** In this step, we will standardize our dataset. Such as in a particular column, the features with high variance are more important compared to the features with lower variance. If the importance of features is independent of the variance of the feature, then we will divide each data item in a column with the standard deviation of the column. Here we will name the matrix as Z.
4. **Calculating the Covariance of Z:** To calculate the covariance of Z, we will take the matrix Z, and will transpose it. After transpose, we will multiply it by Z. The output matrix will be the Covariance matrix of Z.
5. **Calculating the Eigen Values and Eigen Vectors:** Now we need to calculate the eigenvalues and eigenvectors for the resultant covariance matrix Z. Eigenvectors or the covariance matrix are the directions of the axes with high information. And the coefficients of these eigenvectors are defined as the eigenvalues.
6. **Sorting the Eigen Vectors:** In this step, we will take all the eigenvalues and will sort them in decreasing order, which means from largest to smallest. And simultaneously sort the eigenvectors accordingly in matrix P of eigenvalues. The resultant matrix will be named as P*.

7. **Calculating the new features Or Principal Components:** Here we will calculate the new features. To do this, we will multiply the P* matrix to the Z. In the resultant matrix Z*, each observation is the linear combination of original features. Each column of the Z* matrix is independent of each other.

8. **Remove less or unimportant features from the new dataset:** The new feature set has occurred, so we will decide here what to keep and what to remove. It means, we will only keep the relevant or important features in the new dataset, and unimportant features will be removed out.

Applications of Principal Component Analysis:

- PCA is mainly used as the dimensionality reduction technique in various AI applications such **as computer vision, image compression, etc.**
- It can also be used for finding hidden patterns if data has high dimensions. Some fields where PCA is used are Finance, data mining, Psychology, etc.

We can use principal component analysis (PCA) for the following purposes:

- To reduce the number of dimensions in the dataset.
- To find patterns in the high-dimensional dataset
- To visualize the data of high dimensionality
- To ignore noise
- To improve classification
- To gets a compact description
- To captures as much of the original variance in the data as possible

In summary, we can define **principal component analysis (PCA)** as the transformation of any high number of variables into a smaller number of uncorrelated variables called principal components (PCs), developed to capture as much of the data's variance as possible.

PCA was invented in 1901 by Karl Pearson and Harold Hotelling as an analog of the Principal axis theorem [1] [2] [3].

Mathematically the main objective of PCA is to:

- Find an orthonormal basis for the data.
- Sort dimensions in the order of importance.
- Discard the low significance dimensions.

- Focus on uncorrelated and Gaussian components.

Steps involved in PCA:

- Standardize the PCA.
- Calculate the covariance matrix.
- Find the eigenvalues and eigenvectors for the covariance matrix.
- Plot the vectors on the scaled data.

Example of a problem where PCA is required:

There are 100 students in a class with *m* different features like grade, age, height, weight, hair color, and others.

Most of the features may not be relevant that describe the student. Therefore, it is vital to find the critical features that characterize a student.

Some analysis based on the observation of different features of a student:

- Every student has a vector of data that defines him the length of m. e.g. (height, weight, hair_color, grade,….) or (181, 68, black, 99, ….).
- Each column is one student vector. So, n = 100.
- It creates an *m*n* matrix.
- Each student lies in an m-dimensional vector space.

Features to Ignore:

- Collinear features or linearly dependent features. e.g., leg size and height.
- Noisy features that are constant. e.g., the thickness of hair
- Constant features. e.g., Number of teeth.

Features to Keep:

- Non-collinear features or low covariance.
- Features that change a lot, high variance. e.g., grade.

Math Behind PCA:

It is essential to understand the mathematics involved before kickstarting PCA. Eigenvalues and eigenvector play important roles in PCA.

Eigenvectors and eigenvalues:

The eigenvectors and eigenvalues of a covariance matrix (or correlation) describe the source of the PCA. Eigenvectors (main components)

determine the direction of the new attribute space, and eigenvalues determine its magnitude.

The PCA's main objective is to reduce the data's dimensionality by projecting it into a smaller subspace, where the eigenvectors form the axes. However, the eigenvectors define only the new axes' directions because they all have a size of **1**. Consequently, to decide which eigenvector(s), we can discard without losing much information in the subspace construction and checking the corresponding eigenvalues. The eigenvectors with the highest values are the ones that include more information about the distribution of our data.

Covariance Matrix:

The classic PCA approach calculates the covariance matrix, where each element represents the covariance between two attributes. The covariance between two attributes is calculated as shown below:

$$\sigma_{jk} = 1/n\text{-}1\sum_{i=1}^{N}(x_{ij} - \bar{x}_j)(x_{ik} - \bar{x}_k)$$

Figure 10: The equation to calculate the covariance between two attributes.

Create a matrix:

```
import pandas as pd
import numpy as npmatrix = np.array([[0, 3, 4], [1, 2, 4], [3, 4, 5]])
matrix
```

```
array([[0, 3, 4],
       [1, 2, 4],
       [3, 4, 5]])
```

Figure 11: Matrix.

Convert matrix to covariance matrix:

```
np.cov(matrix)
```

```
array([[4.33333333, 2.83333333, 2.        ],
       [2.83333333, 2.33333333, 1.5       ],
       [2.        , 1.5       , 1.        ]])
```

Figure 12: Covariance matrix.

An exciting feature of the covariance matrix is that the sum of the matrix's main diagonal is equal to the eigenvalues' sum.

Correlation Matrix:

Another way to calculate eigenvalues and eigenvectors is by using the correlation matrix. Although the matrices are different, they will result in the same eigenvalues and eigenvectors (shown later) since the covariance matrix's normalization gives the correlation matrix.

$$corr(x, y) \quad cov(x, y)/\sigma_x \sigma_y$$

Figure 13: Equation of the correlation matrix.

Create a matrix:

```
matrix_a = np.array([[0.1, .32, .2,  0.4, 0.8],
          [.23, .18, .56, .61, .12],
          [.9,   .3,  .6,  .5,  .3],
          [.34, .75, .91, .19, .21]])
```

Convert to correlation matrix:

```
np.corrcoef(matrix_a.T)
```

```
array([[ 1.        , -0.03783885,  0.34905716,  0.14648975, -0.34945863],
       [-0.03783885,  1.        ,  0.67888519, -0.96102583, -0.12757741],
       [ 0.34905716,  0.67888519,  1.        , -0.45104803, -0.80429469],
       [ 0.14648975, -0.96102583, -0.45104803,  1.        , -0.15132323],
       [-0.34945863, -0.12757741, -0.80429469, -0.15132323,  1.        ]])
```

Figure 14: Correlation matrix:

How does PCA work?:

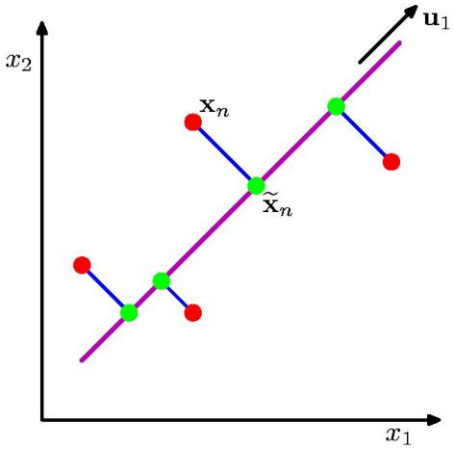


Figure 15: Working with PCA [5].

The orthogonal projection of data from high dimensions to lower dimensions such that (from figure 15):

- Maximizes the variance of the projected line (purple)
- Minimizes the MSE between the data points and projections (blue)

Applications of PCA:

These are the typical applications of PCA:

- Data Visualization.
- Data Compression.

- Noise Reduction.
- Data Classification.
- Image Compression.
- Face Recognition.

Implementation of PCA With Python:

Implementation of principal component analysis (PCA) on the Iris dataset with Python:

Load Iris dataset:

```
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaleriris = load_iris()
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)df['class']
= iris.target
df
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	class
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	2
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2
149	5.9	3.0	5.1	1.8	2

Figure 16: Iris dataset.

Get the value of x and y:

```
x = df.drop(labels='class', axis=1).values
y = df['class'].values
```

Implementation of PCA with a covariance Matrix:

```
class convers_pca():
    def __init__(self, no_of_components):
        self.no_of_components = no_of_components
        self.eigen_values = None
        self.eigen_vectors = None

    def transform(self, x):
```

```
        return np.dot(x - self.mean, self.projection_matrix.T)

    def inverse_transform(self, x):
        return np.dot(x, self.projection_matrix) + self.mean

    def fit(self, x):
        self.no_of_components = x.shape[1] if self.no_of_components is
None else self.no_of_components

        self.mean = np.mean(x, axis=0)

        cov_matrix = np.cov(x - self.mean, rowvar=False)

        self.eigen_values, self.eigen_vectors = np.linalg.eig(cov_matrix)
        self.eigen_vectors = self.eigen_vectors.T

        self.sorted_components = np.argsort(self.eigen_values)[::-1]

        self.projection_matrix =
self.eigen_vectors[self.sorted_components[:self.no_of_components]]self.e
xplained_variance = self.eigen_values[self.sorted_components]

        self.explained_variance_ratio = self.explained_variance /
self.eigen_values.sum()
```

Standardization of x:

```
std = StandardScaler()
transformed = StandardScaler().fit_transform(x)
```

PCA with two components:

```
pca = convers_pca(no_of_components=2)
pca.fit(transformed)
```

Check eigenvectors:

```
cov_pca.eigen_vectors
```

Check eigenvalues:

```
cov_pca.eigen_values
```

Check sorted component:

```
cov_pca.sorted_components
```

Plot PCA with several components = 2:

```
x_std = pca.transform(transformed)plt.figure()
plt.scatter(x_std[:, 0], x_std[:, 1], c=y)
```

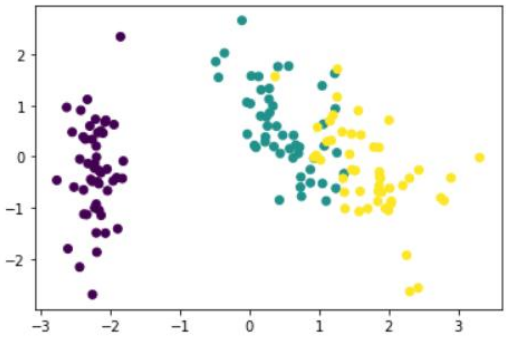


Figure 17: PCA visualization.

*****

UNIT V

7

UNSUPERVISED LEARNING K-MEANS CLUSTERING ALGORITHM

Unit Structure

7.0 OBJECTIVES

This Chapter would make you understand the following concepts:

- What is K-Means clustering algorithm
- Definition of K-Means clustering algorithm
- Basics of K-Means clustering
- Practical advantages of K-Means clustering algorithm
- Stages of K-Means clustering algorithm
- Pseudo code of K-Means clustering algorithm
- The K-Means Algorithm Fits within the Framework of Cover's Theorem

- Partitioning Clustering Approach
- The *K-means* algorithm: a heuristic method
- How K-means partitions?
- K-means Demo
- Application of K-Means algorithm
- Relevant issues of K-Means algorithm

7.1 INTRODUCTION – K-MEANS CLUSTERING ALGORITHM

K-Means Clustering is an unsupervised learning algorithm that is used to solve the clustering problems in machine learning or data science.

7.2 DEFINITION: K-MEANS CLUSTERING ALGORITHM

A prototypical unsupervised learning algorithm is K-means, which is clustering algorithm. Given X = $\{x_1,...,x_m\}$ the goal of K-means is to partition it into k clusters such that each point in a cluster is similar to points from its own cluster than with points from some other cluster

7.3 BASIC ALGORITHMS

Towards this end, define prototype vectors $\mu_1,...,\mu_k$ and an indicator vector r_{ij} which is 1 if, and only if, x_i is assigned to cluster j. To cluster our dataset we will minimize the following distortion measure, which minimizes the distance of each point from the prototype vector:

$$J(r,\mu) := \frac{1}{2}\sum_{i=1}^{m} \sum_{j=1}^{k} r_{ij} ||x_i - \mu_j||^2$$

where r = $\{r_{ij}\}$, μ = $\{\mu_j\}$, and $||\,.\,||^2$ denotes the usual Euclidean square norm.

7.3.1 K-Means clustering:

The computation is to be performed in an unsupervised manner. In this section, we describe a solution to this problem that is rooted in clustering, by which we mean the following:

Clustering is a form of unsupervised learning whereby a set of observations (i.e., data points) is partitioned into natural groupings or clusters of patterns in such a way that the measure of similarity between any pair of observations assigned to each cluster minimizes a specified cost function.

We have chosen to focus on the so-called *K-means algorithm*, because it is simple to implement, yet effective in performance, two features that have made it highly popular.

Let $\{X_i\}_{i=1}^{N}$ denote a set of multidimensional observations that is to be partitioned into a proposed set of K clusters, where K is smaller than the number of observations, N. Let the relationship.

$$\mathbf{j = C(i),\ \ i = 1, 2, ..., N}$$

denote a many-to-one mapper, called the encoder, which assigns the i_{th} observation x_i to the j_{th} cluster according to a rule yet to be defined. To do this encoding, we need a measure of similarity between every pair of vectors x_i and $x_{i'}$ which is denoted by $d(x_i, x_{i'})$.When the measure d(xi, $x_{i'}$) is small enough, both x_i and $x_{i'}$ are assigned to the same cluster; otherwise, they are assigned to different clusters.

To optimize the clustering process, we introduce the following cost function (Hastie et al.,2001):

$$J(C) = \frac{1}{2}\sum_{j=1}^{K}\sum_{C(i)=j}\sum_{C(i')=j} d(\mathbf{x}_i, \mathbf{x}_{i'})$$

For a prescribed K, the requirement is to find the encoder C(i)=jfor which the cost function J(C) is minimized. At this point in the discussion, we note that the encoder C is unknown—hence the functional dependence of the cost function J on C.

In K-means clustering, the squared Euclidean norm is used to define the measure of similarity between the observations x_i and $x_{i'}$ as shown by

$$d(\mathbf{x}_i, \mathbf{x}_{i'}) = \|\mathbf{x}_i - \mathbf{x}_{i'}\|^2$$

Hence,

$$J(C) = \frac{1}{2}\sum_{j=1}^{K}\sum_{C(i)=j}\sum_{C(i')=j}\|\mathbf{x}_i - \mathbf{x}_{i'}\|^2$$

We now make two points:

1. The squared Euclidean distance between the observations x_i and $x_{i'}$ is symmetric; that is,

$$\|\mathbf{x}_i - \mathbf{x}_{i'}\|^2 = \|\mathbf{x}_{i'} - \mathbf{x}_{i'}\|^2$$

2. The inner summation reads as follows: For a given , the encoder C assigns to cluster j all the observations that are closest to xi. Except for a scaling factor, the sum of the observations so assigned is an estimate of the mean vector pertaining to cluster j; the scaling factor in question is 1/Nj, where Nj is the number of data points within

cluster j. On account of these two points, we may therefore reduce to the simplified form

$$J(C) = \sum_{j=1}^{K} \sum_{C(i)=j} \|\mathbf{x}_i - \hat{\boldsymbol{\mu}}_j\|^2$$

where denotes the "estimated" mean vector associated with cluster j[4] .In effect, the mean may be viewed as the center of cluster j. In light of we may now restate the clustering problem as follows:

Given a set of N observations, find the encoder C that assigns these observations to the K clusters in such a way that, within each cluster, the average measure of dissimilarity of the assigned observations from the cluster mean is minimized.

Indeed, it is because of the essence of this statement that the clustering technique described herein is commonly known as the K-means algorithm.

For an interpretation of the cost function J(C) we may say that, except for a scaling factor 1/Nj, the inner summation in this equation is an estimate of the variance of the observations associated with cluster j for a given encoder C, as shown by

$$\hat{\sigma}_j^2 = \sum_{C(i)=j} \|\mathbf{x}_i - \hat{\boldsymbol{\mu}}_j\|^2$$

Accordingly, we may view the cost function J(C) as a measure of the total cluster variance resulting from the assignments of all the N observations to the K clusters that are made by encoder C.

With encoder C being unknown, how do we minimize the cost function J(C) To address this key question, we use an iterative descent algorithm, each iteration of which involves a two-step optimization. The first step uses the nearest neighbor rule to minimize the cost function J(C) of with respect to the mean vector for a given encoder C. The second step minimizes the inner summation with respect to the encoder C for a given mean vector .This two-step iterative procedure is continued until convergence is attained.

Thus, in mathematical terms, the K-means algorithm proceeds in two steps:

Step 1: For a given encoder C, the total cluster variance is minimized with respect to the assigned set of cluster means ; that is, we perform, the following minimization:

$$\min_{\{\hat{\boldsymbol{\mu}}\}_{j=1}^{K}} \sum_{j=1}^{K} \sum_{C(i)=j} \|\mathbf{x}_i - \hat{\boldsymbol{\mu}}_j\|^2$$ for a given C

Step 2: Having computed the optimized cluster means in step 1,we next optimize the encoder as follows

$$C(i) = \arg\min_{1 \le j \le K} \|\mathbf{x}(i) - \hat{\boldsymbol{\mu}}_j\|^2$$

Starting from some initial choice of the encoder C, the algorithm goes back and forth between these two steps until there is no further change in the cluster assignments.

Each of these two steps is designed to reduce the cost function J(C) in its own way; hence, convergence of the algorithm is assured. However, because the algorithm lacks a global optimality criterion, the result may converge to a local minimum, resulting in a suboptimal solution to the clustering assignment.

7.3.2 Practical advantages:

Nevertheless, the algorithm has |Practical advantages:

1. The K-means algorithm is computationally efficient, in that its complexity is linear in the number of clusters.

2. When the clusters are compactly distributed in data space, they are faithfully recovered by the algorithm.

One last comment is in order: To initialize the K-means algorithm, the recommended procedure is to start the algorithm with many different random choices for the means for the proposed size K and then choose the particular set for which the double summation in assumes the smallest value

7.4 STAGES OF K-MEANS CLUSTERING ALGORITHM

Our goal is to find r and μ, but since it is not easy to jointly minimize J with respect to both r and μ, we will adapt a two stage strategy:

Stage 1:

Keep the μ fixed and determine r.

In this case, it is easy to see that the minimization decomposes into m independent problems. The solution for the i-th data point xi can be found by setting:

$$r_{ij} = 1 \quad if \quad j = ||x_i - \mu_{j'}||^2,$$

and 0 otherwise.

Stage 2:

Keep the r fixed and determine μ. Since the r's are fixed, J is an quadratic function of μ. It can be minimized by setting the derivative with respect to μj to be 0.

$$\sum_{i=1}^{m} r_{ij}\,(x_i - \mu_j) = 0 \; for \; all \; j$$

$$\mu_j = \frac{\sum_i \; r_{ij} x_i}{\sum_i \; r_{ij}}$$

Rearranging obtains

Since $\sum_i \; r_{ij}$ counts the number of points assigned to cluster j, we are essentially setting μj to be the sample mean of the points assigned to cluster j.

7.5 PSEUDO-CODE

Detailed pseudo-code can be found in K-Means Algorithms:

Cluster(**X**) {Cluster dataset **X**}

Initialize cluster centers μ_j for j = 1,...,k randomly

Repeat

for i = 1 **to** m **do**

Compute j' = arg $\min_{j=1,...,k}$ d(xi,μj)

Set $r_{ij'}$ = 1 and r_{ij} = 0 for all j'≠j

end for

for j = 1 **to** k **do**

Compute μj = $\frac{\sum_i \; r_{ij} x_i}{\sum_i \; r_{ij}}$

end for

until Cluster assignments r_{ij} are unchanged

return {μ_1,...,μ_k} and r_{ij}

The algorithm stops when the cluster assignments do not change significantly.

7.6 THE K-MEANS ALGORITHM FITS WITHIN THE FRAMEWORK OF COVER'S THEOREM

The K-means algorithm applies a nonlinear transformation to the input signal x. We say so because the measure of dissimilarity—namely, the squared Euclidean distance ,on which it is based—is a nonlinear function of the input signal x for a given cluster center xj. Furthermore, with each cluster discovered by the K-means algorithm defining a particular computational unit in the hidden layer, it follows that if the number of

clusters, K, is large enough, the K-means algorithm will satisfy the other requirement of Cover's theorem—that is, that the dimensionality of the hidden layer is high enough. We therefore conclude that the K-means algorithm is indeed computationally powerful enough to transform a set of nonlinearly separable patterns into separable ones in accordance with this theorem. Now that this objective has been satisfied, we are ready to consider designing the linear output layer of the RBF network.

7.7 PARTITIONING CLUSTERING APPROACH

- a typical clustering analysis approach via iteratively partitioning training data set to learn a partition of the given data space
- learning a partition on a data set to produce several non-empty clusters (usually, the number of clusters given in advance)
- in principle, optimal partition achieved via ptimizeg the sum of

$$E = \sum_{k}^{K} \sum_{\mathbf{x} \in C} d^2(\mathbf{x}, \mathbf{m}_k)$$

squared distance to its "representative object" in each cluster

e.g., Euclidean distance $d^2(\mathbf{x},\mathbf{m}) = \sum^{N}(x_n - m_{kn})^2$

n=1

- **Given a *K*, find a partition of *K clusters* to ptimize the chosen partitioning criterion (cost function)**
- global optimum: exhaustively search all partitions

7.8 THE *K-MEANS* ALGORITHM: A HEURISTIC METHOD

- K-means algorithm (MacQueen'67): each cluster is represented by the centre of the cluster and the algorithm converges to stable centriods of clusters.
- K-means algorithm is the simplest partitioning method for clustering analysis and widely used in data mining applications.

Given the cluster number *K*, the *K-means* algorithm is carried out in three steps after initialisation:

Initialisation: set seed points (randomly)

1. Assign each object to the cluster of the nearest seed point measured with a specific distance metric

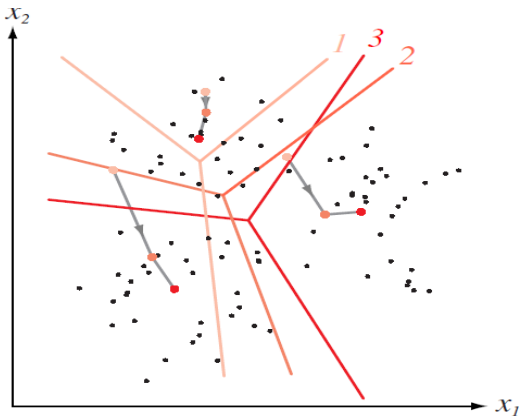
2. Compute new seed points as the centroids of the clusters of the current partition (the centroid is the centre, i.e., *mean point*, of the cluster)

3. Go back to Step 1), stop when no more new assignment (i.e., membership in each cluster no longer changes)

7.8.1 How K-means partitions?:

When K centroids are set/fixed, they partition the whole data space into K mutually exclusive subspaces to form a partition.

A partition amounts to a Voronoi Diagram -Changing positions of centroids leads to a new partitioning.

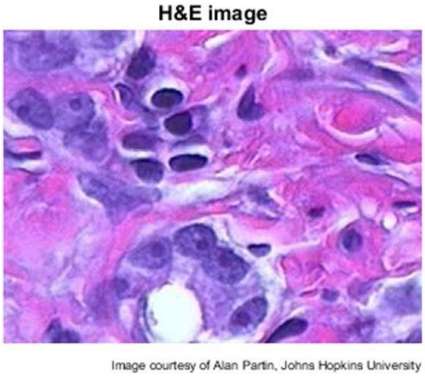


7.8.2 K-means Demo:

7.8.3 Application:

Colour-Based Image Segmentation Using *K*-means

Step 1: Loading a colour image of tissue stained with hemotoxylin and eosin **(H&E)**



Colour-Based Image Segmentation Using *K*-means

Step 2: Convert the image from RGB colour space to L*a*b* colour space

- Unlike the RGB colour model, L*a*b* colour is designed to approximate human vision.
- There is a complicated transformation between RGB and L*a*b*.

(L*, a*, b*) = T(R, G, B).

(R, G, B) = T'(L*, a*, b*).

Colour-Based Image Segmentation Using *K*-means:

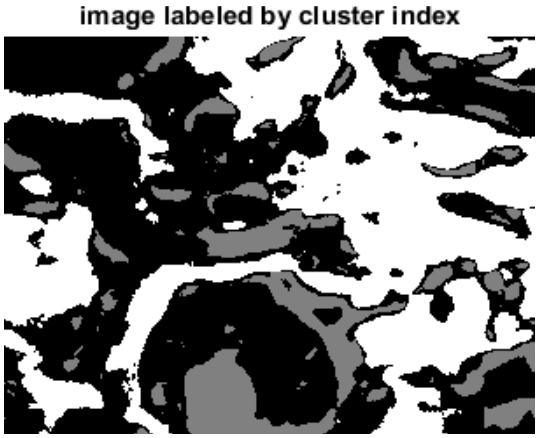
Step 3: Undertake clustering analysis in the (a*, b*) colour space with the *K*-means algorithm

- In the L*a*b* colour space, each pixel has a properties or feature vector:(L*, a*, b*).
- Like feature selection, L* feature is discarded. As a result, each pixel has a feature vector (a*, b*).
- Applying the *K*-means algorithm to the image in the a*b* feature space where $K = 3$ by applying the domain knowledge.

Colour-Based Image Segmentation Using *K*-means:

Step 4: Label every pixel in the image using the results from

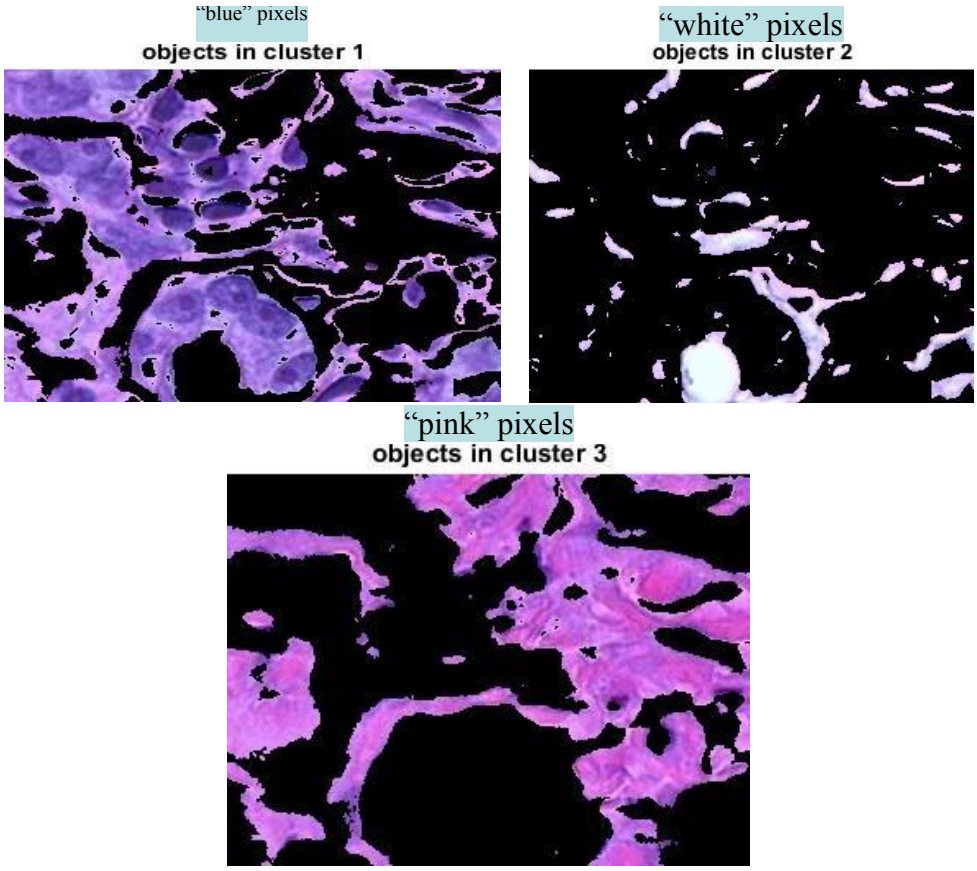
K-means clustering (indicated by three different grey levels)



Colour-Based Image Segmentation Using *K*-means:

Step 5: Create Images that Segment the H&E Image by Colour

- Apply the label and the colour information of each pixel to achieve separate colour images corresponding to three clusters.

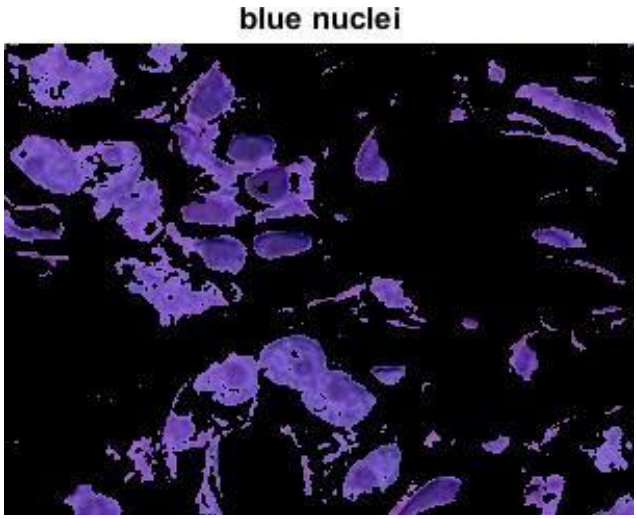


Colour-Based Image Segmentation Using *K*-means:

Step 6: Segment the nuclei into a separate image with the L* feature

- In cluster 1, there are dark and light blue objects (pixels). The dark blue objects (pixels) correspond to nuclei (with the domain knowledge).

- L* feature specifies the brightness values of each colour.
- With a threshold for L*, we achieve an image containing the nuclei only.



7.8.4 Relevant issues of K-Means algorithm

Computational complexity

- O(*tKn*), where *n* is number of objects, *K*is number of clusters, and *t*is number of iterations. Normally, *K*, *t* << *n*.

Local optimum

- sensitive to initial seed points
- converge to a local optimum: maybe an unwanted solution

Other problems

- Need to specify *K*, the *number* of clusters, in advance
- Unable to handle noisy data and outliers (*K-Medoids* algorithm)
- Not suitable for discovering clusters with non-convex shapes

  the *K*-mean performance?

Two issues with K-Means are worth noting.

First, it is sensitive to the choice of the initial cluster centers μ. A number of practical heuristics have been developed. For instance, one could randomly choose k points from the given dataset as cluster centers. Other methods try to pick k points from X which are farthest away from each other.

Second, it makes a hard assignment of every point to a cluster center. Variants which we will encounter later in the book will relax this. Instead

of letting $r_{ij} \in \{0,1\}$ these soft variants will replace it with the probability that a given xi belongs to cluster j.

The K-Means algorithm concludes our discussion of a set of basic machine learning methods for classification and regression. They provide a useful starting point for an aspiring machine learning researcher.

7.9 LET'S SUM UP

We will have a clear idea about Definition , Basic Algorithms, Stages and Pseudo code of K-Means clustering algorithm.

7.10 UNIT END EXERCISES

- Take a Data set available and execute on different inputs of K-Means clustering algorithm.

7.11 REFERENCES

- https://www.javatpoint.com/k-means-clustering-algorithm-in-machine-learning
- https://towardsdatascience.com/k-means-clustering-algorithm-applications-evaluation-methods-and-drawbacks-aa03e644b48a
- https://www.analyticsvidhya.com/blog/2021/11/understanding-k-means-clustering-in-machine-learningwith-examples/
- https://www.geeksforgeeks.org/k-means-clustering-introduction/
- https://www.simplilearn.com/tutorials/machine-learning-tutorial/k-means-clustering-algorithm

*****

8

UNSUPERVISED LEARNING
K- MEDOID CLUSTERING ALGORITHM

Unit Structure

8.0 OBJECTIVES

This Chapter would make you understand the following concepts:

- What is K-Medoid clustering algorithm
- Definition of K-Medoid clustering algorithm
- Comparison of K-Medoid clustering algorithm
- K-Medoid Basic algorithm
- K-Medoid PAM algorithm
- Clara – Clustering Large Applications
- Working and Practical Implementation

8.1 DEFINITION – K-MEDOID CLUSTERING ALGORITHM

K-Medoids is a clustering algorithm resembling the K-Means clustering technique. It falls under the category of unsupervised machine learning.

8.2 INTRODUCTION - K-MEDOID CLUSTERING ALGORITHM

It majorly differs from the K-Means algorithm in terms of the way it selects the clusters' centres. The former selects the average of a cluster's points as its centre (which may or may not be one of the data points) while the latter always picks the actual data points from the clusters as their centres (also known as '**exemplars**' or '**medoids**'). K-Medoids also differs in this respect from the K-Medians algorithm which is the same as K-means, except that it chooses the medians (instead of means) of the clusters as centres.

The mean in k-means clustering is sensitive to outliers. Since an object with an extremely high value may substantially distort the distribution of data. Hence we move to k-medoids. Instead of taking mean of cluster we take the most centrally located point in cluster as it's center. These are called medoids.

8.3 K-MEANS & K-MEDOIDS CLUSTERING-OUTLIERS COMPARISON

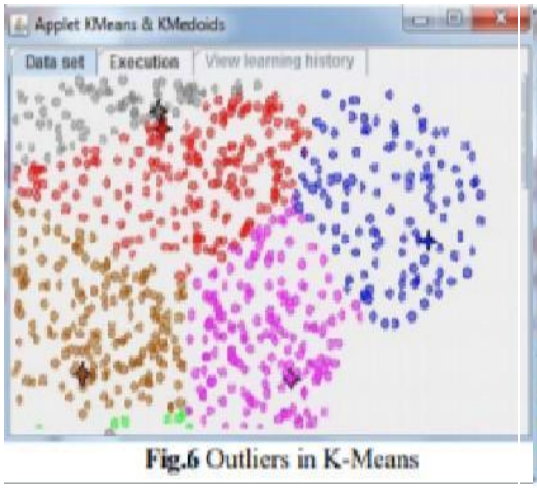


Fig.6 Outliers in K-Means

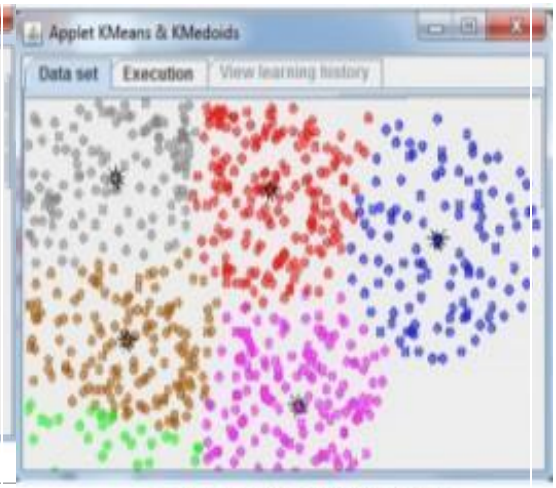


Fig.7 Outliers in K-Medoids

8.4 K-MEDOIDS - BASIC ALGORITHM

Input: Number of K (the clusters to form)

Initialize: Select K points as the initial representative objects i.e initial K-medoids of our K clusters.

Repeat: Assign each point to the cluster with the closest medoid m.

Randomly select a non-representative object o_i

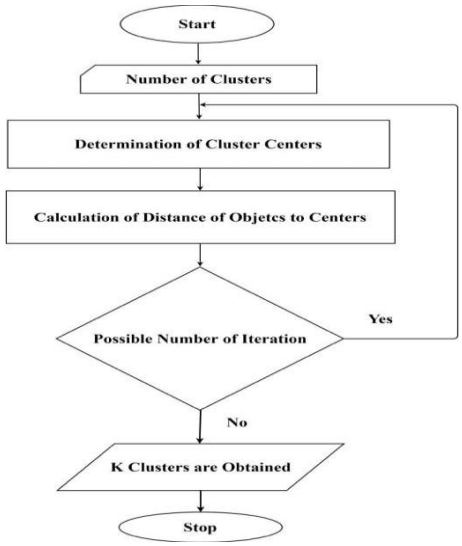
Compute the total cost of **swapping** S, the medoid m with o_i

If S < 0:

Swap m with o_i to form new set of medoids.

Stop when convergence criteria is meet.

8.5 K-MEDOIDS - PAM ALGORITHM



PAM stands for **Partitioning Around Medoids**.

GOAL: To find Clusters that have minimum average dissimilarity between objects that belong to same cluster.

Algorithm:

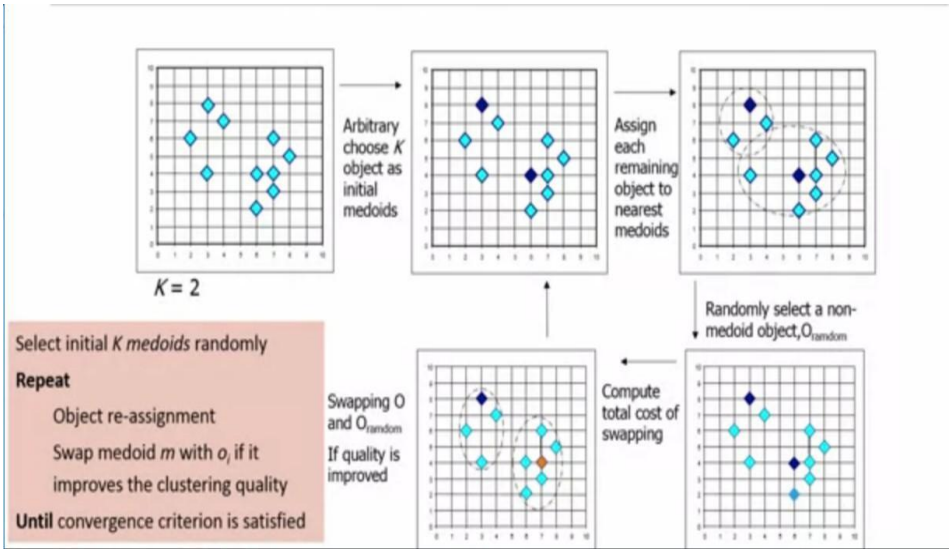
1. Start with initial set of medoids.
2. Iteratively replace one of the medoids with a non-medoid if it reduces total sum of SSE of resulting cluster.

SSE is calculated as below:

$$SSE(X) = \sum_{i=1}^{K} \sum_{x \in C_i} dist^2(m_i, x)$$

Where k is number of clusters and x is a data point in cluster C_i and M_i is medo id of C_i

8.5.1 Typical Pam Example:



Data Objects

	A_1	A_2
O_1	2	6
O_2	3	4
O_3	3	8
O_4	4	7
O_5	6	2
O_6	6	4
O_7	7	3
O_8	7	4
O_9	8	5
O_{10}	7	6

K-Medoids (Pam) Example:

For K = 2

Randomly Select m1 = (3,4) and m2 =(7,4)

Using Manhattan as similarity metric we get,

C1 = (o1, o2, o3, o4)

C2 = (o5, o6, o7, o8, o9, o10)

Compute absolute error as follows:

E = (o1-o2) + (o3-o2) + (o4-o2) + (o5-o8) +(o6-o8)+(o7-o8) +(o9-o8) + (o10-o8)

E = (3+4+4) + (3+1+1+2+2)

Therefore, **E = 20**

Swapping o8 with o7

Compute absolute error as follows:

E = (o1-o2) + (o3-o2) + (o4-o2) + (o5-o7) +(o6-o7)+(o8-o7) +(o9-o7) + (o10-o7)

E = (3+4+4) + (2+2+1+3+3)

Therefore, **E = 22**

Let's now calculate cost function S for this swap, S = E for (o2,07) - E for (o2, o8)

S = 22- 20

Therefore S > 0,

This swap is undesirable

8.6 ADVANTAGES and DISADVANTAGES of PAM:

Advantages:

- PAM is more flexible as it can use any similarity measure.
- PAM is more robust than k-means as it handles noise better.

Disadvantages:

PAM algorithm for K-medoid clustering works well for dataset but cannot scale well for large data set due to high computational overhead.

Pam Complexity : $O(k(n-k)^2)$ this is because we compute distance of n-k points with each k point, to decide in which cluster it will fall and after this we try to replace each of the medoid with a non medoid and find it's distance with n-k points.

To overcome this we make use of CLARA

8.7 CLARA – CLUSTERING LARGE APPLICATIONS

- Improvement over PAM
- Finds medoids in a sample from the dataset
- [Idea]: If the samples are sufficiently random, the medoids of the sample approximate the medoids of the dataset
- [Heuristics]: 5 samples of size 40+2k gives satisfactory results
- Works well for large datasets (n=1000, k=10)

8.7.1 Clara Algorithm:

1. Split randomly the data sets in multiple subsets with fixed size (sampsize)
2. Compute PAM algorithm on each subset and choose the corresponding k representative objects (medoids). Assign each observation of the entire data set to the closest medoid.
3. Calculate the mean (or the sum) of the dissimilarities of the observations to their closest medoid. This is used as a measure of the goodness of the clustering.
4. Retain the sub-dataset for which the mean (or sum) is minimal. A further analysis is carried out on the final partition.

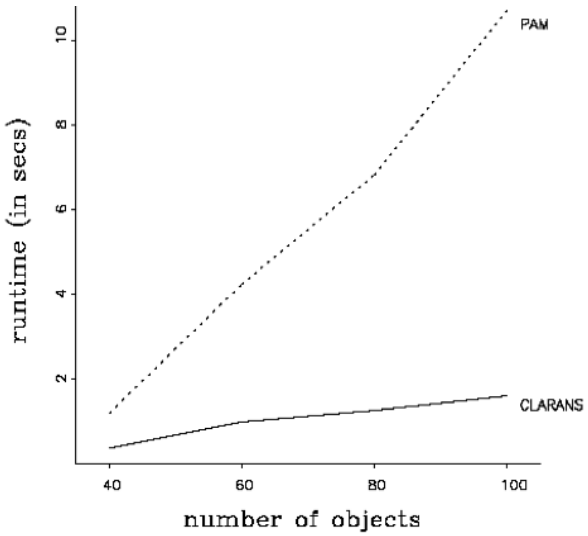
8.8 COMPARISON CLARA vs PAM

Strength:

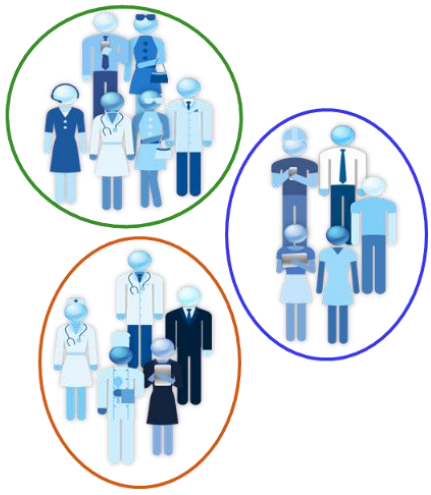
- deals with larger data sets than *PAM*
- CLARA Outperforms PAM in terms of running time and quality of clustering

Weakness:

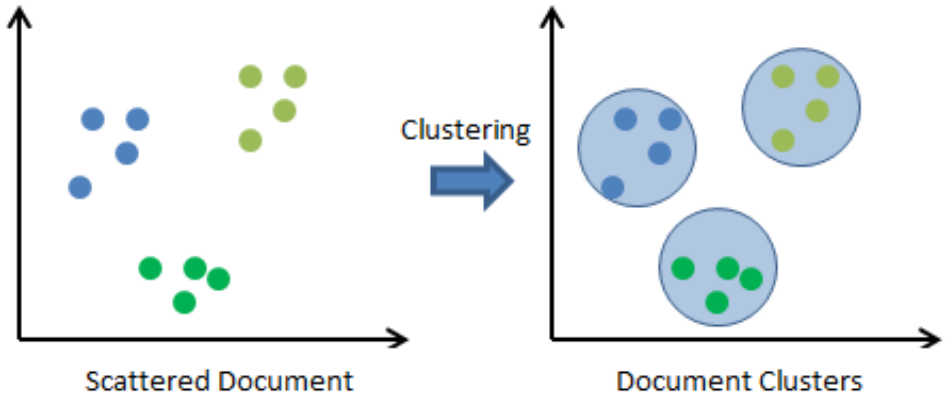
- Efficiency depends on the sample size
- A good clustering based on samples will not necessarily represent a good clustering of the whole



8.9 APPLICATIONS



Social Network:



Document Clustering

8.10 GENERAL APPLICATIONS OF CLUSTERING

1. Recognition

2. Spatial Data Analysis

a. create thematic maps in GIS by clustering feature spaces

b. detect spatial clusters and explain them in spatial data mining

1. Image Processing

2. Economic Science (especially market research)

3. WWW

a. Document classification

b. Cluster Weblog data to discover groups of similar access patterns

8.11 WORKING OF THE K-MEDOIDS APPROACH

The steps followed by the K-Medoids algorithm for clustering are as follows:

1. Randomly choose 'k' points from the input data ('k' is the number of clusters to be formed). The correctness of the choice of k's value can be assessed using methods such as silhouette method.

2. Each data point gets assigned to the cluster to which its nearest medoid belongs.

3. For each data point of cluster i, its distance from all other data points is computed and added. The point of i^{th} cluster for which the computed sum of distances from other points is minimal is assigned as the medoid for that cluster.

4. Steps (2) and (3) are repeated until convergence is reached i.e. the medoids stop moving.

8.11.1 Complexity of K-Medoids algorithm:

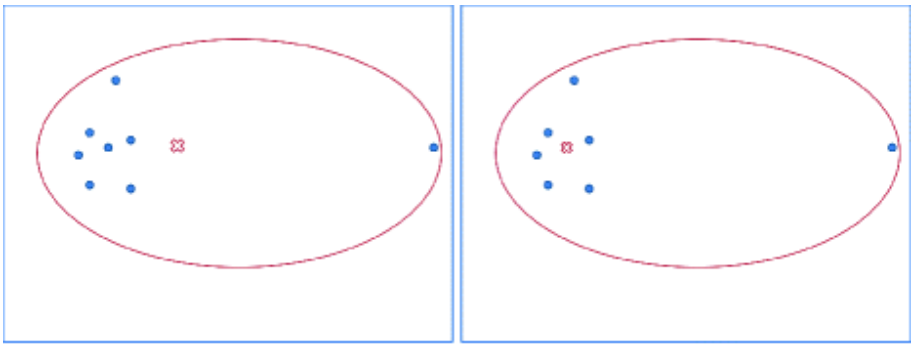
The complexity of the K-Medoids algorithm comes to O(N2CT) where N, C and T denote the number of data points, number of clusters and number of iterations respectively. With similar notations, the complexity K-Means algorithm can be given as O(NCT).

8.11.2 Advantages of the technique:

Mean of the data points is a measure that gets highly affected by the extreme points. So in K-Means algorithm, the centroid may get shifted to a wrong position and hence result in incorrect clustering if the data has outliers because then other points will move away from. On the contrary, a medoid in the K-Medoids algorithm is the most central element of the

cluster, such that its distance from other points is minimum. Since medoids do not get influenced by extremities, the K-Medoids algorithm is more robust to outliers and noise than K-Means algorithm.

The following figure explains how mean's and medoid's positions can vary in the presence of an outlier.



(a) Mean (b) Medoid

Besides, K-Medoids algorithm can be used with arbitrarily chosen dissimilarity measure (e.g. cosine similarity) or any distance metric, unlike K-Means which usually needs Euclidean distance metric to arrive at efficient solutions.

K-Medoids algorithm is found useful for practical applications such as face recognition. The medoid can correspond to the typical photo of the individual whose face is to be recognized. But if K-Means algorithm is used instead, some blurred image may get assigned as the centroid, which has mixed features from several photos of the individual and hence makes the face recognition task difficult.

8.12 PRACTICAL IMPLEMENTATION

Here's a demonstration of implementing K-Medoids algorithm on a dataset containing 8*8 dimensional images of handwritten digits. The task is to divide the data points into 10 clusters (for classes 0-9) using K-Medoids. The dataset used is a copy of the test set of the original dataset available on UCI ML Repository. The code here has been implemented in Google colab using **Python 3.7.10** and **scikit-learn-extra 0.1.0b2** versions.

Step-Wise Explanation of The Code Is As Follows:

1. Install:

scikit-learn-extra Python module, an extension of scikit-learn designed for implementing more advanced algorithms that cannot be used by mere inclusion of scikit-learn in the code.

!pip install scikit-learn-extra

2. Import required libraries and modules:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn_extra.cluster import KMedoids
#Import the digits' dataset available in sklearn.datasets package
from sklearn.datasets import load_digits
“””
```

Instead of using all 64 attributes of the dataset, we use Principal Component Analysis (PCA) to reduce the dimensions of features set such that most of the useful information is covered.

```
“””
from sklearn.decomposition import PCA
“””
```

Import module for standardizing the dataset i.e. rescaling the data such that its has mean of 0 and standard deviation of 1

```
“””
from sklearn.preprocessing import scale
```

3. Prepare the input data:

```
#Load the digits dataset
dataset = load_digits()
#Standardize the data
digit_data = scale(dataset.data)
“””
```

Compute number of output classes i.e. number of digits for which we have the data (here 10 (0-9))

```
“””
num_digits = len(np.unique(dataset.target))
```

4. Reduce the dimensions of the data using PCA:

```
red_data = PCA(n_components=2).fit_transform(digit_data)
“””
```

PCA constructs new components by linear combinations of original features. **‘n_components’** parameter denotes the number of newly formed

components to be considered. **fit_transform() method** fits the PCA models and performs dimensionality reduction on digit_data.

‘‘’’’

5. Plot the decision boundaries for each cluster. Assign a different color to each for differentiation:

```
h = 0.02 #step size of the mesh

#Minimum and maximum x-coordinates

xmin, xmax = red_data[:, 0].min() - 1, red_data[:, 0].max() + 1

#Minimum and maximum y-coordinates

ymin, ymax = red_data[:, 1].min() - 1, red_data[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(xmin, xmax, h), np.arange(ymin, ymax, h))
```

6. Define an array of K-Medoids variants to be used:

We have used three different distance metrics (**Manhattan distance, Euclidean distance and Cosine dissimilarity/distance**) for computing the distance of each data point from every other data point while selecting the medoid.

Visit this page to know about the distance metrics used in detail.

The parameters we have **specified in the KMedoids() method** have the following significance:

- **metric** – distance metric to be used (default: ‘euclidean’)
- **n_clusters** – number of clusters to be formed and hence the number of medoids (one per cluster) (default value: 8)
- **init** – ‘heuristic’ method used for medoid initialization

For each data point, itd distance from all other points is computed and the distances are summed up. N_clusters number of points for which such a sum of distances are minimum, are chosen as medoids.

- **max_iter** – maximum number of the algorithm’s iterations to be performed when fitting the data

The KMedoids() method of scikit-learn-extra by default used the PAM (Partition Around Medoids) algorithm for finding the medoids.

```
models = [
    (
        KMedoids(metric="manhattan", n_clusters=num_digits,
```

```
        init="heuristic", max_iter=2),"Manhattan metric",
    ),
    (
        KMedoids(metric="euclidean", n_clusters=num_digits,
        init="heuristic", max_iter=2),"Euclidean metric",
    ),
    (KMedoids(metric="cosine", n_clusters=num_digits, init="heuristic",
     max_iter=2), "Cosine metric", ),
]
```

7. Initialize the number of rows and columns of the plot for plotting subplots of each of the three metrics' results:

```
#number of rows = integer(ceiling(number of model variants/2))
num_rows = int(np.ceil(len(models) / 2.0))
#number of columns
num_cols = 2
```

8. Fit each of the model variants to the data and plot the resultant clustering:

```
#Clear the current figure first (if any)
plt.clf()
#Initialize dimensions of the plot
plt.figure(figsize=(15,10))
‘‘’’’
```

The 'models' array defined in step (6) contains three tuples, each having a model variant's parameters and its descriptive text. We iterate through each of the tuples, fit the data to the model and plot the results.

```
‘‘’’’
for i, (model, description) in enumerate(models):
    # Fit each point in the mesh to the model
    model.fit(red_data)
   #Predict the labels for points in the mesh
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    # Put the result  into a color plot
    Z = Z.reshape(xx.shape)
```

```
#Subplot for the ith model variant
 plt.subplot(num_cols, num_rows, i + 1)
#Display the subplot
 plt.imshow(
    Z,    #data to be plotted
    interpolation="nearest",
#bounding box coordinates (left,right,bottom,top)
    extent=(xx.min(), xx.max(), yy.min(), yy.max()),
    cmap=plt.cm.Paired,  #colormap
    aspect="auto", #aspect ratio of the axes
    origin="lower",  #set origin as lower left corner of the axes
 )
 plt.plot(
    red_data[:, 0], red_data[:, 1], "k.", markersize=2, alpha=0.3
 )
 # Plot the centroids as white cross marks
 centroids = model.cluster_centers_
 plt.scatter(
    centroids[:, 0],
    centroids[:, 1],
    marker="x",
    s=169,  #marker's size (points^2)
    linewidths=3, #width of boundary lines
    color="w",  #white color for centroids markings
    zorder=10,  #drawing order of axes     )
 #describing text of the tuple will be title of the subplot
 plt.title(description)
 plt.xlim(xmin, xmax)  #limits of x-coordinates
 plt.ylim(ymin, ymax)  #limits of y-coordinates
```

```
    plt.xticks(())
    plt.yticks(())
#Upper title of the whole plot
plt.suptitle(
  #Text to be displayed
   "K-Medoids algorithm implemented with different metrics\n\n",
   fontsize=20,  #size of the fonts
)
plt.show()
```

8.13 LET'S SUM UP

We will have a clear idea about:

- What is K-Medoid clustering algorithm
- Definition of K-Medoid clustering algorithm
- Comparison of K-Medoid clustering algorithm
- K-Medoid Basic algorithm
- K-Medoid PAM algorithm
- Clara – Clustering Large Applications
- Working and Practical Implementation

8.14 UNIT END EXERCISES

- Take a Data set available and execute on different inputs of K-Medoid clustering algorithm.

8.15 REFERENCES

- **http://www.math.le.ac.uk/people/ag153/homepage/KmeansKmedoids/Kmeans_Kmedoids.html**
- **https://www.datanovia.com/en/lessons/k-medoids-in-r-algorithm-and-practical-examples/**
- **https://towardsdatascience.com/understanding-k-means-k-means-and-k-medoids-clustering-algorithms-ad9c9fbf47ca**
- **https://iq.opengenus.org/k-medoids-clustering/**
- **https://www.datanovia.com/en/lessons/k-medoids-in-r-algorithm-and-practical-examples/**

*****

UNIT VI

9

CLASSIFYING DATA USING SUPPORT VECTOR MACHINES (SVMS): SVM-RBF KERNELS

Unit Structure

9.0 INTRODUCTION TO SVMS

Support vector machines (SVMs, also known as support vector networks) are supervised learning models with related learning algorithms for classification and regression analysis in machine learning. A Support Vector Machine (SVM) is a discriminative classifier with a separating hyperplane as its formal definition. In other words, the algorithm produces an ideal hyperplane that categorizes fresh samples given labeled training data (supervised learning).

9.1 WHAT IS A SUPPORT VECTOR MACHINE, AND HOW DOES IT WORK?

An SVM model is a representation of the examples as points in space, mapped so that the examples of the different categories are separated by as wide a gap as possible. SVMs may do non-linear classification, implicitly translating their inputs into high-dimensional feature spaces, in addition to linear classification.

9.2 WHAT IS THE PURPOSE OF SVM?

An SVM training algorithm creates a model that assigns new examples to one of two categories, making it a non-probabilistic binary linear classifier, given a series of training examples that are individually designated as belonging to one of two categories.

Before you go any further, make sure you have a basic knowledge of this topic. In this article, I'll show you how to use machine learning techniques like scikit-learn to classify cancer UCI datasets using SVM.

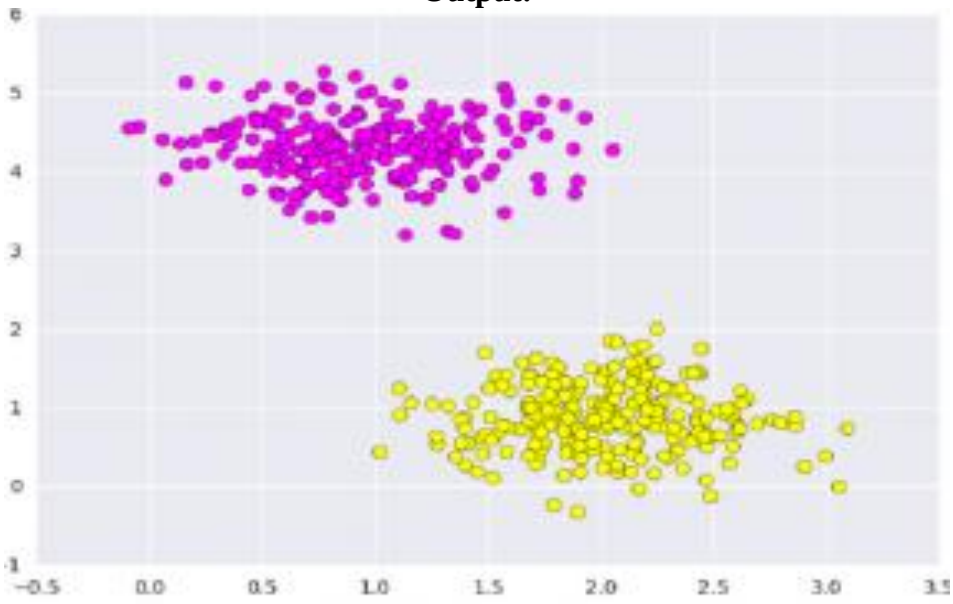
Numpy, Pandas, matplot-lib, and scikit-learn are required.

Let's look at a simple support vector categorization example. To begin, we must first generate a dataset:

Implemention in python

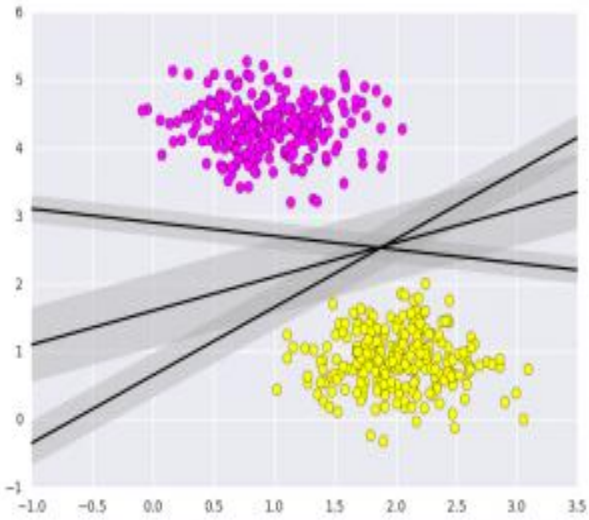
```
# importing scikit learn with make_blobs
from sklearn.datasets.samples_generator import make_blobs

# creating datasets X containing n_samples
# Y containing two classes
X, Y = make_blobs(n_samples=500, centers=2,
                                    random_state=0, cluster_std=0.40)
import matplotlib.pyplot as plt
# plotting scatters
plt.scatter(X[:, 0], X[:, 1], c=Y, s=50, cmap='spring');
plt.show()
```

Output:



Support vector machines consider a region around the line of a particular width in addition to drawing a line between two classes. Here's an example of how it may appear:

```
# creating line space between -1 to 3.5
xfit = np.linspace(-1, 3.5)
# plotting scatter
plt.scatter(X[:, 0], X[:, 1], c=Y, s=50, cmap='spring')
# plot a line between the different sets of data
for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:
    yfit = m * xfit + b
    plt.plot(xfit, yfit, '-k')
    plt.fill_between(xfit, yfit - d, yfit + d, edgecolor='none',
    color='#AAAAAA', alpha=0.4)
plt.xlim(-1, 3.5);
plt.show()
```



9.3 IMPORTING DATASETS

Support vector machines, which optimize a linear discriminant model reflecting the perpendicular distance between datasets, have this understanding. Let's now use our training data to train the classifier. We must first import cancer datasets as a CSV file, from which we will train two features out of all the features.

```
# importing required libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
 # reading csv file and extracting class column to y.
x = pd.read_csv("C:\...\cancer.csv")
a = np.array(x)
y  = a[:,30] # classes having 0 and 1

 # extracting two features
x = np.column_stack((x.malignant,x.benign))

 # 569 samples and 2 features
x.shape
print (x),(y)
```

```
[[  122.8   1001.  ]
 [  132.9   1326.  ]
 [  130.    1203.  ]
 ...,
 [  108.3    858.1 ]
 [  140.1   1265.  ]
 [   47.92   181.  ]]

array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  1.,  1.,  1.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  1.,  1.,  1.,  1.,
        1.,  0.,  0.,  1.,  0.,  0.,  1.,  1.,  1.,  1.,  0.,  1., ....,
        1.])
```

9.4 THE ESTABLISHMENT OF A SUPPORT VECTOR MACHINE

These locations will now be fitted with a Support Vector Machine Classifier. While the mathematical specifics of the likelihood model are fascinating, we'll save those for another time. Instead, we'll approach the scikit-learn algorithm as a black box that performs the aforementioned work.

```
# import support vector classifier
# "Support Vector Classifier"
from sklearn.svm import SVC
clf = SVC(kernel='linear')
```

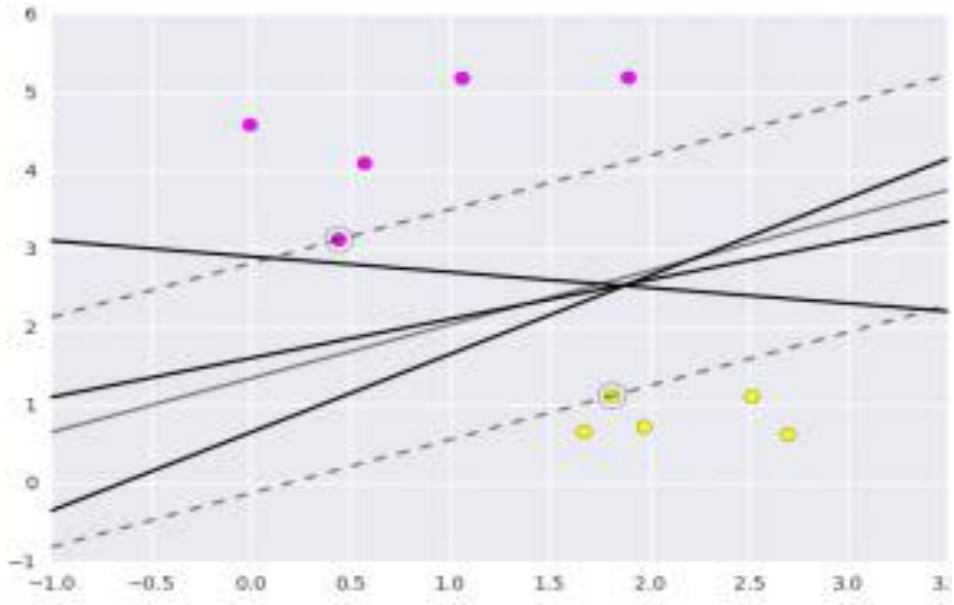
```
# fitting x samples and y classes
clf.fit(x, y)
```

The model can then be used to forecast new values after it has been fitted:

```
clf.predict([[120, 990]])
clf.predict([[85, 550]])
```

```
array([ 0.])
array([ 1.])
```

Let's have a look at the graph to see what this means.



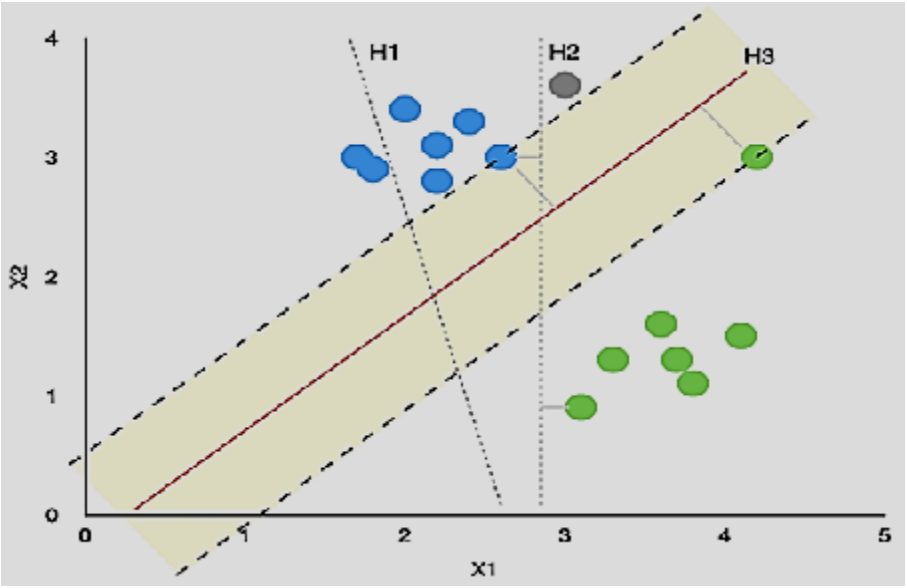
9.5 A SIMPLE DESCRIPTION OF THE SVM CLASSIFICATION ALGORITHM

Assume we have a set of points that are divided into two classes. We want to split those two classes so that we can accurately assign any new points to one or the other in the future.

The SVM algorithm seeks out a hyperplane that separates these two classes by the greatest margin possible. A hard margin can be utilized if classes are entirely linearly separable. Otherwise, a soft margin is required.

Note that support vectors are the points that end up on the margins.

Hard-margin:



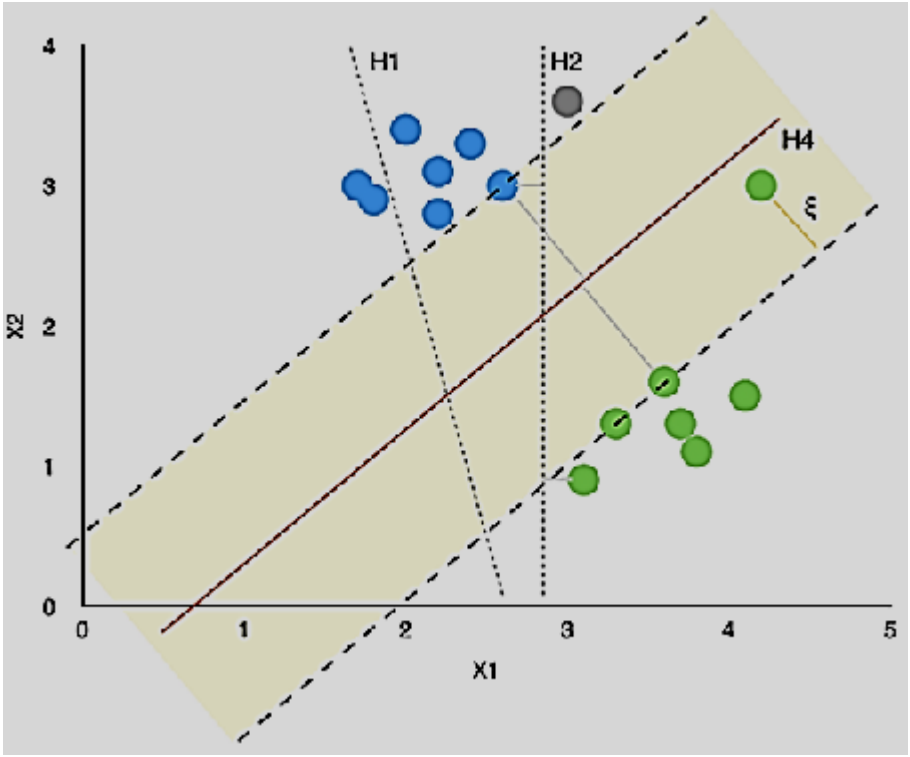
The SVM method is used to separate the two classes of points. Scenario with a tight margin.

- The "H1" hyperplane is incapable of accurately separating the two classes; hence it is not a suitable solution to our problem.
- The "H2" hyperplane accurately splits classes. The distance between the hyperplane and the nearest blue and green points, on the other hand, is extremely small. As a result, there's a good risk that any future new points may be classified erroneously. The algorithm, for example, would allocate the new grey point (x1=3, x2=3.6) to the green class when it is evident that it should belong to the blue class instead.
- Finally, the "H3" hyperplane appropriately and with the greatest possible margin divides the two classes (yellow shaded area). A solution has been discovered!

It's worth noting that determining the maximum feasible margin allows for a more accurate classification of additional data, resulting in a far more robust model. When utilizing the "H3" hyperplane, you can see that the new grey point is correctly allocated to the blue class.

Soft-Margin:

It may not always be possible to completely separate the two classes. In such cases, a soft-margin is employed, with some points permitted to be misclassified or to fall within the margin (yellow shaded area). This is where the "slack" value, represented by ξ (xi).



The SVM method is used to separate the two classes of points. Scenario with a soft margin.

The green point inside the margin is treated as an outlier by the "H4" hyperplane in this case. As a result, the support vectors are the two green spots closest to the main group. This increases the model's resilience by allowing for a bigger margin.

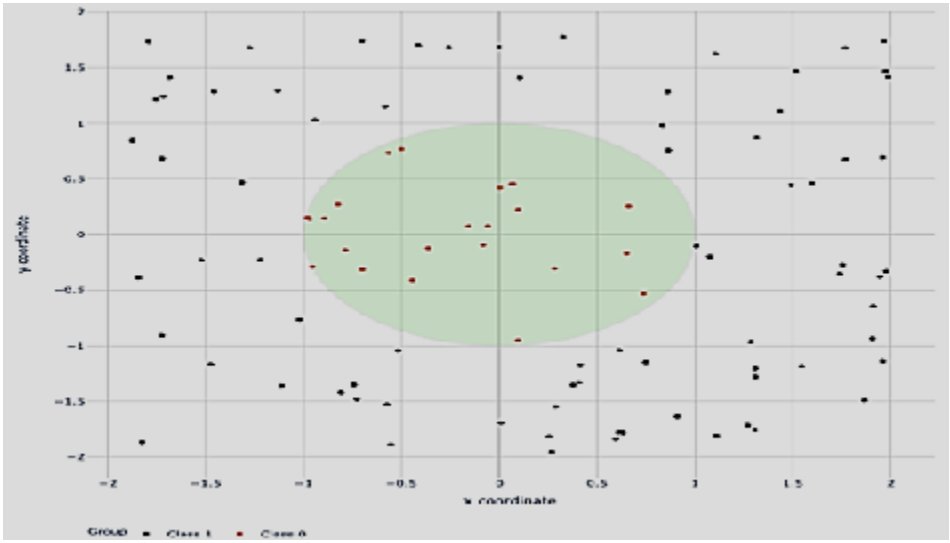
Note that you may tweak the hyperparameter C to decide how much you care about misclassifications (and points inside the margin) in the algorithm. C is essentially a weight that has been assigned to. A low C wants to categorize all training instances correctly, producing a closer match to the training data but making it less robust, whereas a high C strives to classify all training examples correctly, producing a closer fit to the training data but making it less robust.

While a high C value will likely result in higher model performance on the training data, there is a substantial risk of over fitting the model, which will result in poor test data outcomes.

Kernel Trick:

SVM was previously explained in the context of linearly separable blue and green classes. What if we wanted to use SVMs to solve non-linear problems? How would we go about doing that? The kernel technique comes into play at this point. A kernel is a function that takes a nonlinear problem and converts it to a linear problem in a higher-dimensional space. Let's look at an example to demonstrate this method.

Assume you have two classes, red and black, as indicated in the diagram below:



Data in its original two-dimensional form.

As you can see, red and black points are not linearly separable because there is no way to construct a line that separates these two classes. We can, however, distinguish them by drawing a circle with all of the red dots inside and the black points outside.

9.6 WHAT IS THE BEST WAY TO TRANSFORM THIS PROBLEM INTO A LINEAR ONE?

Make a third dimension out of the sum of squared x and y values:

$$z = x^2 + y^2$$

We can now design a hyperplane (flat 2D surface) to separate red and black points using this three-dimensional space with x, y, and z values. As a result, the SVM classification algorithm is now available.

9.7 KERNEL FOR THE RADIAL BASIS FUNCTION (RBF) AND PYTHON EXAMPLES

The default kernel in sklearn's SVM classification algorithm is RBF, which can be defined using the formula:

$$K(x, x') = e^{-\gamma||x-x'||^2}$$

Where gamma can be adjusted manually and must be greater than zero. In sklearn's SVM classification method, the default value for gamma is:

$$\gamma = \frac{1}{n\,features * \sigma^2}$$

Briefly:

$||x - x'||^2$ Between two feature vectors, 2 is the squared Euclidean distance (2 points). Gamma is a scalar that expresses how powerful a single training sample (point) can be.

As a result of the above design, we can control the influence of specific points on the overall algorithm. The bigger the gamma, the closer the other points must be to have an impact on the model. In the Python examples below, we'll see how adjusting gamma affects the results.

Setup:

The following data and libraries will be used:

- Kaggle chess games data
- Scikit-learn library for separating the data into train-test samples, creating SVM classification models, and model evaluation
- Data manipulation with Pandas and Numpy

Let's import all the libraries:

make optimal hyperplanes using matplotlib function.

```
import pandas as pd # for data manipulation
import numpy as np # for data manipulation

from sklearn.model_selection import train_test_split # for splitting the data into train and test samples
from sklearn.metrics import classification_report # for model evaluation metrics
from sklearn.svm import SVC # for Support Vector Classification model

import plotly.express as px  # for data visualization
import plotly.graph_objects as go # for data visualization
```

After you've saved the data to your machine, use the code below to ingest it. We also get a few new variables that we can use in the modeling.

```
# Read in the csv
df=pd.read_csv('games.csv', encoding='utf-8')

# Difference between white rating and black rating - independent variable
df['rating_difference']=df['white_rating']-df['black_rating']

# White wins flag (1=win vs. 0=not-win) - dependent (target) variable
df['white_win']=df['winner'].apply(lambda x: 1 if x=='white' else 0)
```

Print a snapshot of a few columns

df.iloc[:,[0,1,5,6,8,9,10,11,13,16,17]]

	id	rated	victory_status	winner	white_id	white_rating	black_id	black_rating	opening_eco	rating_difference	white_win
0	TZJHLljE	False	outoftime	white	bourgris	1500	a-00	1191	D10	309	1
1	l1NXvwaE	True	resign	black	a-00	1322	skinnerua	1261	B00	61	0
2	mIICvQHh	True	mate	white	ischia	1496	a-00	1500	C20	-4	1
3	kWKvrqYL	True	mate	white	daniamurashov	1439	adivanov2009	1454	D02	-15	1
4	9tXo1AUZ	True	mate	white	nik221107	1523	adivanov2009	1469	C41	54	1
...	...	...	...	...	...	...	...	...	...	...	...
20053	EfqH7WWH	True	resign	white	belcolt	1691	jamboger	1220	A80	471	1
20054	WSJDhbPI	True	mate	black	jamboger	1233	farrukhasomiddinov	1196	A41	37	0
20055	yrAas0Kj	True	mate	white	jamboger	1219	schaaksmurf3	1286	D00	-67	1
20056	b0v4tRyF	True	resign	white	marcodisogno	1360	jamboger	1227	B07	133	1
20057	N8G2JHGG	True	mate	black	jamboger	1235	ffbob	1339	D00	-104	0

20058 rows x 11 columns

Let's now write a few functions that we may use to generate different models and plot the results.

This function divides the data into train and test samples, fits the model, predicts the outcome on a test set, and calculates model performance metrics.

```
def fitting(X, y, C, gamma):
    # Create training and testing samples
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
    # Fit the model
    # Note, available kernels: {‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’}, default=’rbf’
    model = SVC(kernel='rbf', probability=True, C=C, gamma=gamma)
    clf = model.fit(X_train, y_train)

    # Predict class labels on training data
    pred_labels_tr = model.predict(X_train)
    # Predict class labels on a test data
    pred_labels_te = model.predict(X_test)

    # Use score method to get accuracy of the model
    print('----- Evaluation on Test Data -----')
    score_te = model.score(X_test, y_test)
```

```
    print('Accuracy Score: ', score_te)
    # Look at classification report to evaluate the model
    print(classification_report(y_test, pred_labels_te))
    print('------------------------------------------------------')
   print('----- Evaluation on Training Data -----')
    score_tr = model.score(X_train, y_train)
    print('Accuracy Score: ', score_tr)
    # Look at classification report to evaluate the model
    print(classification_report(y_train, pred_labels_tr))
    print('------------------------------------------------------')

    # Return relevant data for chart plotting
    return X_train, X_test, y_train, y_test, clf
```

With the test data and model prediction surface, the following function will create a Plotly 3D scatter graph.

```
def Plot_3D(X, X_test, y_test, clf):

    # Specify a size of the mesh to be used
    mesh_size = 5
    margin = 1

    # Create a mesh grid on which we will run our model
    x_min, x_max = X.iloc[:, 0].fillna(X.mean()).min() - margin, X.iloc[:,
0].fillna(X.mean()).max() + margin
    y_min, y_max = X.iloc[:, 1].fillna(X.mean()).min() - margin, X.iloc[:,
1].fillna(X.mean()).max() + margin
    xrange = np.arange(x_min, x_max, mesh_size)
    yrange = np.arange(y_min, y_max, mesh_size)
    xx, yy = np.meshgrid(xrange, yrange)

    # Calculate predictions on grid
    Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
    Z = Z.reshape(xx.shape)
```

```
# Create a 3D scatter plot with predictions
fig = px.scatter_3d(x=X_test['rating_difference'], y=X_test['turns'], z=y_test,
                    opacity=0.8, color_discrete_sequence=['black'])


# Set figure title and colors
fig.update_layout(#title_text="Scatter 3D Plot with SVM Prediction Surface",
                  paper_bgcolor = 'white',
                  scene = dict(xaxis=dict(backgroundcolor='white',
                                          color='black',
                                          gridcolor='#f0f0f0'),
                               yaxis=dict(backgroundcolor='white',
                                          color='black',
                                          gridcolor='#f0f0f0'
                                          ),
                               zaxis=dict(backgroundcolor='lightgrey',
                                          color='black',
                                          gridcolor='#f0f0f0',
                                          )))
# Update marker size
fig.update_traces(marker=dict(size=1))


# Add prediction plane
fig.add_traces(go.Surface(x=xrange, y=yrange, z=Z, name='SVM Prediction',
                          colorscale='RdBu', showscale=False,
                          contours = {"z": {"show": True, "start": 0.2, "end": 0.8, "size": 0.05}}))
fig.show()
```

9.8 BUILD A MODEL WITH DEFAULT VALUES FOR C AND GAMMA

Let's create our first SVM model with the 'rating difference' and 'turns' fields as independent variables (attributes/predictors) and the 'white win' flag as the target.

Note that we're cheating a little because the final number of moves won't be known until after the match. As a result, if we were to make model predictions before the match, we wouldn't be able to use 'turns.' However, this is merely for demonstration purposes, therefore we'll use it in the examples below.

The code is brief because we're using our previously defined 'fitting' function.

```
# Select data for modeling
X=df[['rating_difference', 'turns']]
y=df['white_win'].values

# Fit the model and display results
X_train, X_test, y_train, y_test, clf = fitting(X, y, 1, 'scale')
```

The function prints the following model evaluation metrics:

```
      Evaluation on Test Data
Accuracy Score:  0.6530408773678963
              precision    recall  f1-score   support

           0       0.64      0.70      0.67      2024
           1       0.66      0.60      0.63      1988

    accuracy                           0.65      4012
   macro avg       0.65      0.65      0.65      4012
weighted avg       0.65      0.65      0.65      4012
```

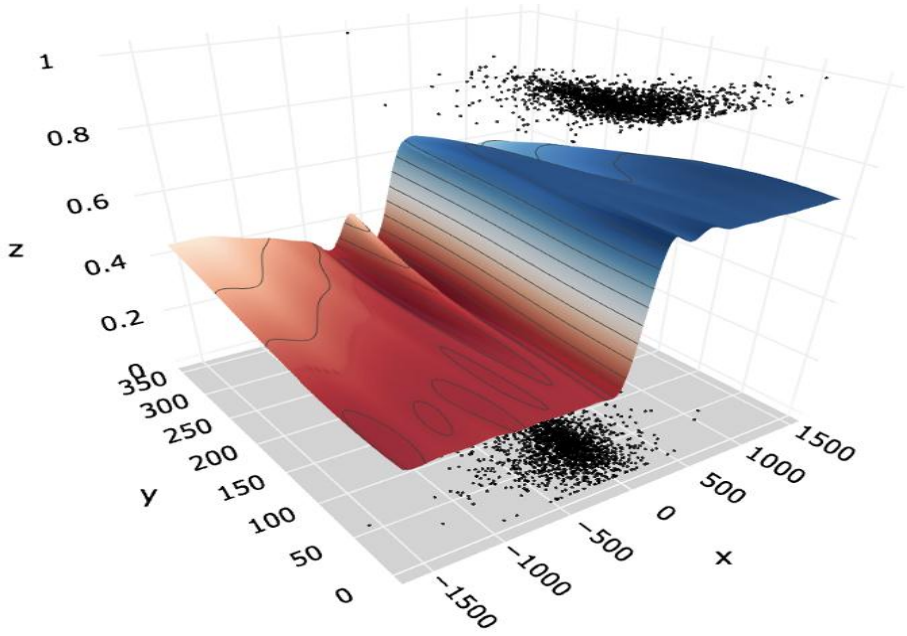
```
      Evaluation on Training Data
Accuracy Score:  0.6468901907017325
              precision    recall  f1-score   support

           0       0.64      0.68      0.66      8033
           1       0.66      0.62      0.64      8013

    accuracy                           0.65     16046
   macro avg       0.65      0.65      0.65     16046
weighted avg       0.65      0.65      0.65     16046
```

SVM model performance metrics.

We can see that the model's performance on test data is similar to that on training data, indicating that the default hyperparameters allow the model to generalize well.

Now we'll use the Plot 3D function to see the prediction:

```
Plot_3D(X, X_test, y_test, clf)
```



SVM classification model prediction plane using default hyperparameters.

Note that the top black spots are actual class=1 (white won), whereas the bottom black points are actual class=0 (white did not win). Meanwhile, the surface represents the model's chance of white wine.

While the probability varies locally, the decision boundary is about x=0 (i.e., rating difference=0) because this is where the probability crosses the p=0.5 line.

SVM MODEL 2 — GAMMA = 0.1

Let's examine what happens if we set gamma to a relatively high value.

Evaluation on Test Data

Accuracy Score: 0.603938185443669

	precision	recall	f1-score	support
0	0.60	0.64	0.62	2024
1	0.61	0.57	0.59	1988
accuracy			0.60	4012
macro avg	0.60	0.60	0.60	4012
weighted avg	0.60	0.60	0.60	4012

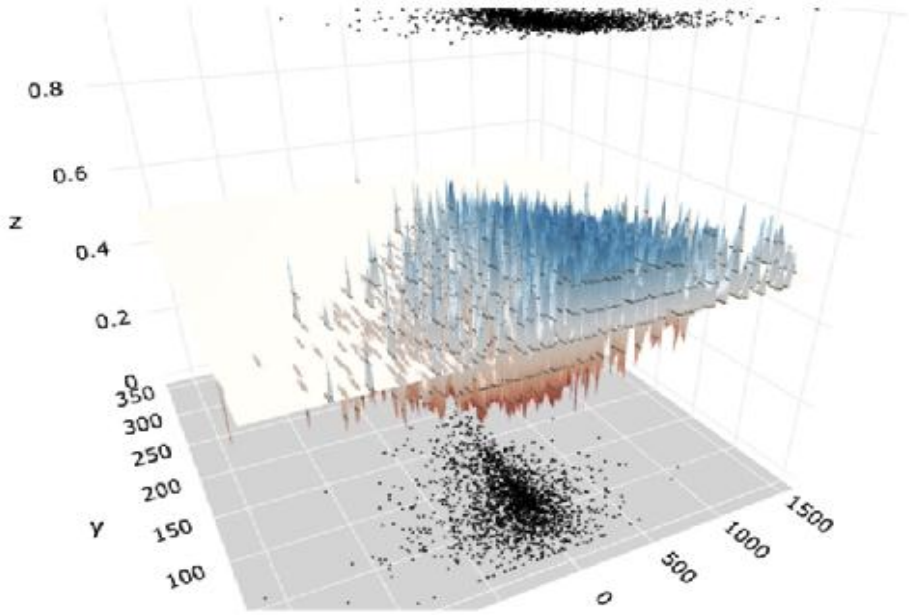
Evaluation on Training Data

Accuracy Score: 0.8003240683036271

	precision	recall	f1-score	support
0	0.80	0.81	0.80	8033
1	0.80	0.80	0.80	8013
accuracy			0.80	16046
macro avg	0.80	0.80	0.80	16046
weighted avg	0.80	0.80	0.80	16046

SVM model performance metrics with Gamma=0.1.

As can be shown, raising gamma improves model performance on training data but degrades model performance on test data. The graph below explains why this is the case.



Prediction plane for a gamma=0.1 SVM classification model. Colorscale='Aggrnyl' was used in the featured image.

Rather than a smooth prediction surface, we now have one that is highly "spiky." We need to look into the kernel function a little more to see why this happens.

When we use a high gamma value, we are telling the function that the close points are significantly more crucial for the prediction than the far points. As a result, we see these "spikes" since the prediction is based on individual points in the training instances rather than the environment.

Reducing gamma, on the other hand, tells the function that when generating a forecast, it's not only the specific point that matters, but also the points around it. Let's look at another case with a low gamma value to see if this is correct.

SVM MODEL 3— GAMMA = 0.000001

Let’s rerun the functions:

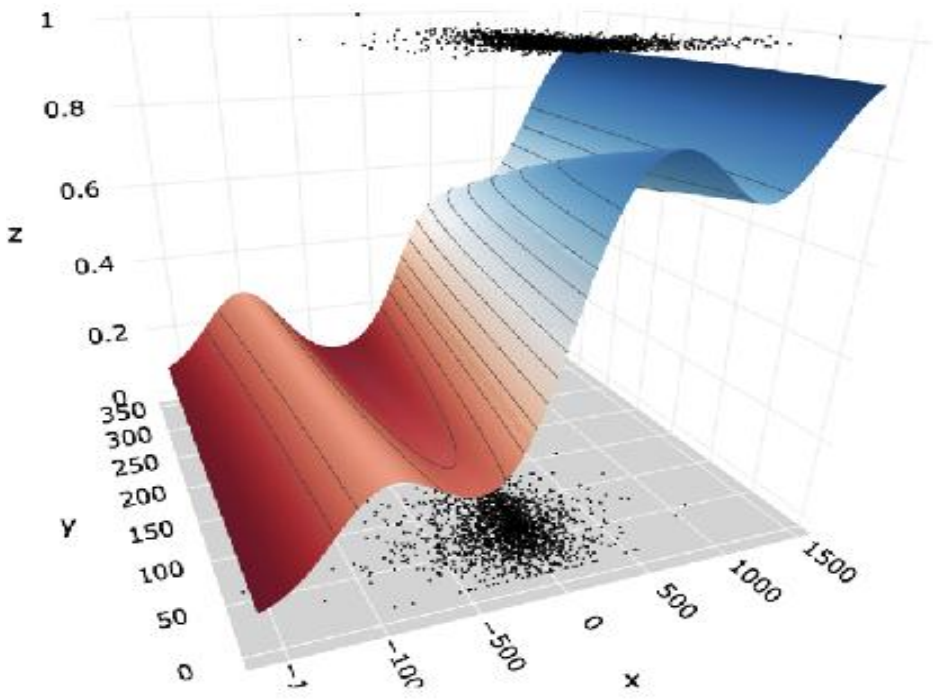
```
      Evaluation on Test Data
Accuracy Score:  0.6602691924227319
              precision    recall  f1-score   support

           0       0.65      0.70      0.68      2024
           1       0.67      0.62      0.64      1988

    accuracy                           0.66      4012
   macro avg       0.66      0.66      0.66      4012
weighted avg       0.66      0.66      0.66      4012
```

```
      Evaluation on Training Data
Accuracy Score:  0.6463916240807678
              precision    recall  f1-score   support

           0       0.64      0.67      0.65      8033
           1       0.65      0.62      0.64      8013

    accuracy                           0.65     16046
   macro avg       0.65      0.65      0.65     16046
weighted avg       0.65      0.65      0.65     16046
```

```
--------------------------------------------------------
```

SVM model performance metrics with Gamma=0.000001.

Reducing gamma improved the model's robustness, as expected, with an increase in model performance on the test data (accuracy = 0.66). The graph below shows how much smoother the prediction surface has gotten after giving the spots further away more influence.



Prediction plane for SVM classification model with gamma=0.000001..

C. Hyperparameter Adjustment:

I chose not to add examples in this tale using various C values because it impacts the smoothness of the prediction plane similarly to gamma, but for different reasons. You may observe for yourself by using the "fitting" function with a value of C=100. some points permitted to be misclassified or to fall within the margin (yellow shaded area) this increases the model's resilience by allowing for a bigger margin.

9.9 RADIAL BASIS FUNCTION (RBF) KERNEL: THE GO-TO KERNEL

We're working on a non-linear dataset with a Machine Learning technique like Support Vector Machines, but you can't seem to figure out the correct feature transform or kernel to employ. Fear not, because the Radial Basis Function (RBF) Kernel is here to save the day.

Due to its resemblance to the Gaussian distribution, RBF kernels are the most generic form of kernelization and one of the most extensively used kernels. For two points X1 and X2, the RBF kernel function computes their similarity, or how near they are to one other. This kernel can be expressed mathematically as follows:

$$K(X_1, X_2) = exp(-\frac{||X_1 - X_2||^2}{2\sigma^2})$$

Where,

1. ‘σ’ is the variance and our hyper parameter

2. $||X_1 - X_2||$ is the Euclidean (L_2-norm) Distance between two points X_1 and X_2

Let d_{12} be the distance between the two points X_1 and X_2, we can now represent d_{12} as follows:

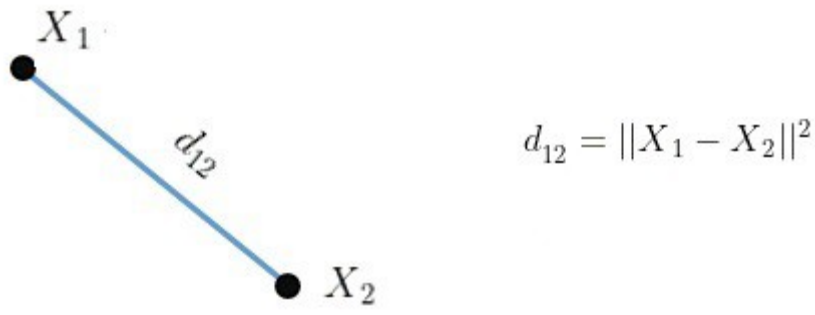


Fig 2: In space, the distance between two points is called the distance between two points in space.

The following is a rewrite of the kernel equation:

$$K(X_1, X_2) = exp(-\frac{d_{12}}{2\sigma^2})$$

The RBF kernel can have a maximum value of 1 when d_{12} is 0, which means that the points are equal, i.e. $X_1 = X_2$.

1. There is no distance between the points when they are the same, therefore they are incredibly comparable.
2. The kernel value is less than 1 and close to 0 when the points are separated by a wide distance, indicating that the points are dissimilar.

Because we can see that as the distance between the point's increases, they become less similar, distance can be regarded of as an analogue to dissimilarity.

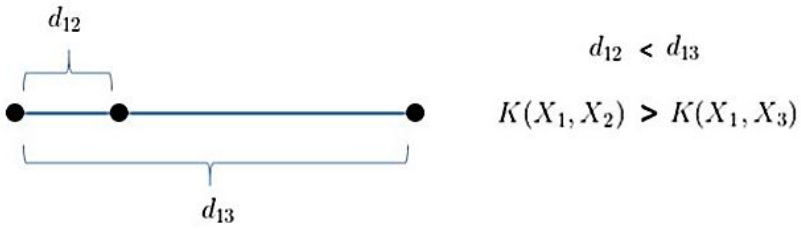


Fig 3: As distance grows, similarity reduces.

Finding the proper value of "to determine which points should be regarded comparable is critical, and this can be proved on a case-by-case basis..

a] σ = 1

When σ = 1, σ² = 1 and the RBF kernel's mathematical equation will be as follows:

$$K(X_1, X_2) = exp(-\frac{||X_1 - X_2||^2}{2})$$

The curve for this equation is shown below, and we can see that the RBF Kernel reduces exponentially as the distance rises, and is 0 for distances larger than 4.

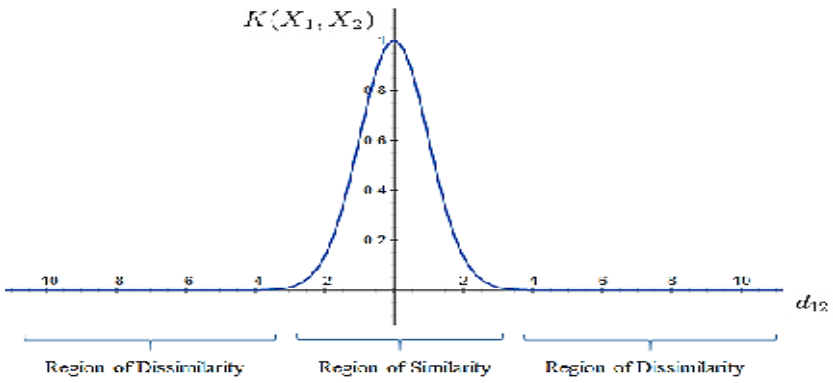


Fig 4: RBF Kernel for σ = 1 [Image by Author]

1. We can see that when d_{12} = 0, the similarity is 1, and when d_{12} exceeds 4 units, the similarity is 0.

2. We can see from the graph that if the distance between the points is less than 4, the points are similar, and if the distance is larger than 4, the points are dissimilar.

b] σ = 0.1

When σ = 0.1, σ² = 0.01 and the RBF kernel's mathematical equation will be as follows:

$$K(X_1, X_2) = exp(-\frac{||X_1 - X_2||^2}{0.01})$$

For σ = 0.1, the width of the Region of Similarity is the smallest, therefore only extremely close points are considered comparable.

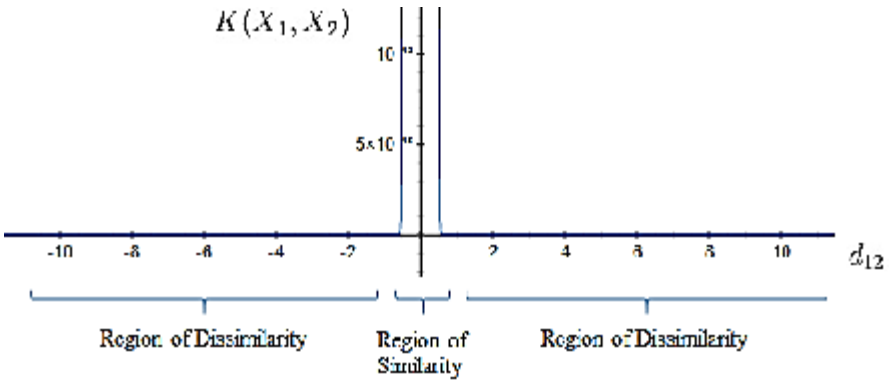


Fig 4a: RBF Kernel for σ = 0.1

1. The curve is severely peaked, with a value of 0 for distances larger than 0.2.

2. Only if the distance between the points is less than or equal to 0.2 is the pair considered comparable.

b] σ = 10

When σ = 10, σ² = 100 and the RBF kernel's mathematical equation will be as follows:

$$K(X_1, X_2) = exp(-\frac{||X_1 - X_2||^2}{100})$$

For σ = 100, the width of the Region of Similarity is enormous, allowing for the comparison of points that are far apart.

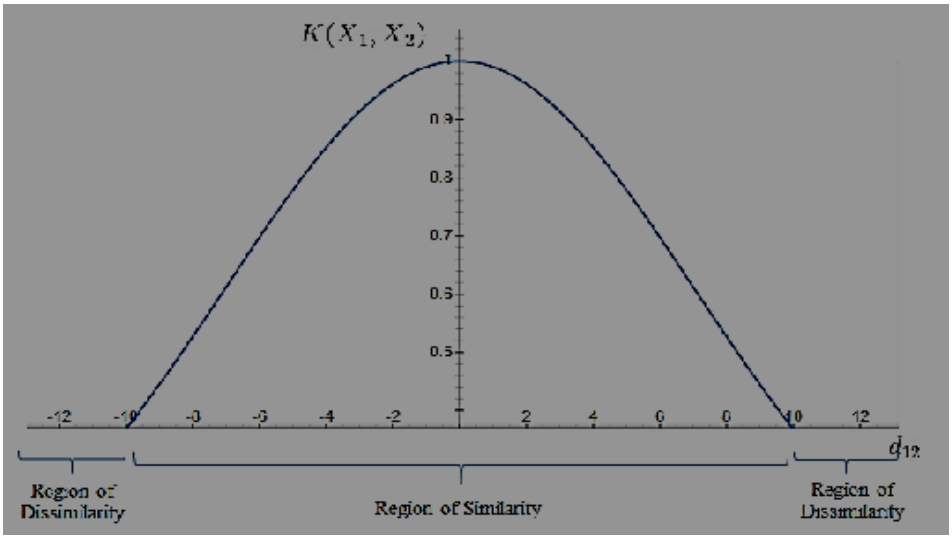


Fig 5: RBF Kernel for σ = 10

1. The curve has a great width.

2. For distances up to 10 units, the points are deemed comparable; but, for distances greater than 10 units, they are considered distinct.

The width of the Region of Similarity changes as changes, as shown in the examples above.

Using hyperparameter tuning approaches such as Grid Search Cross-Validation and Random Search Cross-Validation, you may find the appropriate for a particular dataset.

The RBF Kernel is well-known due to its resemblance to the K-Nearest Neighbor Algorithm. Because RBF Kernel Support Vector Machines only need to store the support vectors during training and not the complete dataset, it has the advantages of K-NN and avoids the space complexity problem.

The RBF Kernel Support Vector Machines are included in the scikit-learn toolkit and have two hyperparameters: 'C' for SVM and "for the RBF Kernel. In this case, is inversely proportional to.

$$\gamma \ \alpha \ \frac{1}{\sigma}$$

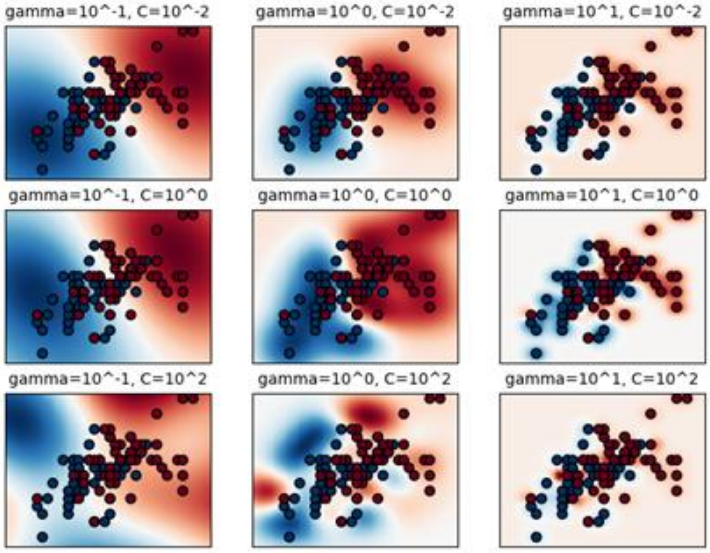


Fig 6: RBF Kernel SVM for Iris Dataset

The RBF Kernel Support Vector Machines are included in the scikit-learn toolkit and have two hyper parameters: 'C' for SVM and " for the RBF Kernel. In this case, is inversely proportional to.

9.10 CONCLUSION

A Support Vector Machine (SVM) is a discriminative classifier with a separating hyperplane as its formal definition. An SVM training algorithm creates a model that assigns new examples to one of two categories, making it a non-probabilistic binary linear classifier. To train the classifier, we must first import the cancer datasets as a CSV file. We then extract two features out of all the samples and train them on top of each other. The SVM algorithm seeks out a hyperplane that separates these two classes by the greatest margin possible.

A hard margin can be utilized if classes are entirely linearly separable. Otherwise, a soft margin is required. Let's have a look at the graph to see what this means. The SVM method is used to separate the two classes of points. In such cases, a soft margin is employed, with some points permitted to be misclassified or to fall within the margin (yellow shaded area) This increases the model's resilience by allowing for a bigger margin.

9.11 REFERENCES

- https://www.geeksforgeeks.org/classifying-data-using-support-vector-machinessvms-in-python/
- https://towardsdatascience.com/svm-classifier-and-rbf-kernel-how-to-make-better-models-in-python-73bb4914af5b
- https://towardsdatascience.com/radial-basis-function-rbf-kernel-the-go-to-kernel-acf0d22c798a

*****

UNIT VII

10

DECISION TREE

Unit structure

10.0 OBJECTIVES

This chapter will enable students to:

- Make use of Data sets in implementing the machine learning algorithms
- Implement the machine learning concepts and algorithms in any suitable language of choice.

Data sets can be taken from standard repositories or constructed by the students.

10.1 DECISION TREE

Objectives: This chapter will enable students to:

- Make use of Data sets in implementing the machine learning algorithms
- Implement the machine learning concepts and algorithms in any suitable language of choice.

Data sets can be taken from standard repositories or constructed by the students.

Introduction:

Decision-tree algorithm falls under the category of supervised learning algorithms. It works for both continuous as well as categorical output variables. Makes use of the Tree representation. Can be used for

classification. Given a decision tree, how do we predict an outcome for a class label? We start from the root of the tree. CART stands for Classification and Regression Trees.

For example, consider a dataset of cats and dogs, with their features. The label here is accordingly "cat", or "dog", and the goal is to identify the animal based on its features, using a decision tree. Say, if at a particular node in the tree, the input to a node contains only a single type of label, say cats, we can infer that it is perfectly grouped, or "unmixed". On the other hand, if the input contains a mix of cats and dogs, we would have to ask another question about the features in the dataset that can help us narrow down, and divide the mix further to try and "unmix" them completely.

```
# Program to implement decision tree in Python

# Importing the required packages

import numpy as np

import pandas as pd

from sklearn.metrics import confusion_matrix

from sklearn.cross_validation import train_test_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy_score

from sklearn.metrics import classification_report

# Function importing Dataset

def importdata():

balance_data = pd.read_csv('decisiontree.data, sep= ',', header = None)

    # Printing the dataswet shape

    print ("Dataset Length: ", len(balance_data))

    print ("Dataset Shape: ", balance_data.shape)

    # Printing the dataset obseravtions

    print ("Dataset: ",balance_data.head())

    return balance_data


# Function to split the dataset

def splitdataset(balance_data):

    # Separating the target variable
```

```
    X = balance_data.values[:, 1:5]
    Y = balance_data.values[:, 0]

    # Splitting the dataset into train and test
    X_train, X_test, y_train, y_test = train_test_split
    (X, Y, test_size = 0.3, random_state = 100)

        return X, Y, X_train, X_test, y_train, y_test

    # Function to perform training with giniIndex.
def train_using_gini(X_train, X_test, y_train):
    # Creating the classifier object
    clf_gini = DecisionTreeClassifier(criterion = "gini",
            random_state = 100,max_depth=3, min_samples_leaf=5)
    # Performing training
    clf_gini.fit(X_train, y_train)
    return clf_gini
    # Function to perform training with entropy.
def tarin_using_entropy(X_train, X_test, y_train):
    # Decision tree with entropy
    clf_entropy = DecisionTreeClassifier(
            criterion = "entropy", random_state = 100,
            max_depth = 3, min_samples_leaf = 5)
    # Performing training
    clf_entropy.fit(X_train, y_train)
    return clf_entropy
# Function to make predictions
def prediction(X_test, clf_object):
    # Predicton on test with giniIndex
    y_pred = clf_object.predict(X_test)
    print("Predicted values:")
    print(y_pred)
```

```
    return y_pred
    # Function to calculate accuracy
def cal_accuracy(y_test, y_pred):
        print("Confusion Matrix: ",
        confusion_matrix(y_test, y_pred))
    print ("Accuracy : ",
    accuracy_score(y_test,y_pred)*100)
        print("Report : ",
    classification_report(y_test, y_pred))
# Driver code
def main():
        # Building Phase
    data = importdata()
    X, Y, X_train, X_test, y_train, y_test = splitdataset(data)
    clf_gini = train_using_gini(X_train, X_test, y_train)
    clf_entropy = tarin_using_entropy(X_train, X_test, y_train)
    # Operational Phase
    print("Results Using Gini Index:")
    # Prediction using gini
    y_pred_gini = prediction(X_test, clf_gini)
    cal_accuracy(y_test, y_pred_gini)
    print("Results Using Entropy:")
    # Prediction using entropy
    y_pred_entropy = prediction(X_test, clf_entropy)
    cal_accuracy(y_test, y_pred_entropy)
    # Calling main function
if __name__=="__main__":
    main()
```

A supervised learning algorithm. Makes use of the Tree representation. Can be used for classification.

10.2 ENSEMBLE TECHNIQUES – BAGGING

```
# importing utility modules

# download the train data set from "https://www.kaggle.com/hesh97/titanicdataset-traincsv"

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.metrics import mean_squared_error

# importing machine learning models for prediction

import xgboost as xgb

# importing bagging module

from sklearn.ensemble import BaggingRegressor

# loading train data set in dataframe from train_data.csv file

df = pd.read_csv("train_data.csv")

# getting target data from the dataframe

target = df["target"]

# getting train data from the dataframe

train = df.drop("target")

# Splitting between train data into training and validation dataset

X_train, X_test, y_train, y_test = train_test_split

(train, target, test_size=0.20)

# initializing the bagging model using XGboost as base model with default parameters

model = BaggingRegressor(base_estimator=xgb.XGBRegressor())

# training model

model.fit(X_train, y_train)

# predicting the output on the test dataset

pred = model.predict(X_test)
```

```
# printing the root mean squared error between real value and predicted value

print(mean_squared_error(y_test, pred_final))
```

10.3 ENSEMBLE TECHNIQUES – BOOSTING

```
# importing utility modules

import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# importing machine learning models for prediction

from sklearn.ensemble import GradientBoostingRegressor

# loading train data set in dataframe from train_data.csv file

df = pd.read_csv("train_data.csv")

# getting target data from the dataframe

target = df["target"]

# getting train data from the dataframe

train = df.drop("target")

# Splitting between train data into training and validation dataset

X_train, X_test, y_train, y_test = train_test_split

(train, target, test_size=0.20)

# initializing the boosting module with default parameters

model = GradientBoostingRegressor()

# training the model on the train dataset

model.fit(X_train, y_train)

# predicting the output on the test dataset

pred_final = model.predict(X_test)

# printing the root mean squared error between real value and predicted value

print(mean_squared_error(y_test, pred_final))
```

10.4 ENSEMBLE TECHNIQUES – STACKING

```
# importing utility modules
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
# importing machine learning models for prediction
from sklearn.ensemble import RandomForestRegressor
import xgboost as xgb
from sklearn.linear_model import LinearRegression
# importing stacking lib
from vecstack import stacking
# loading train data set in dataframe from train_data.csv file
df = pd.read_csv("train_data.csv")
# getting target data from the dataframe
target = df["target"]
# getting train data from the dataframe
train = df.drop("target")
# Splitting between train data into training and validation dataset
X_train, X_test, y_train, y_test = train_test_split
        (train, target, test_size=0.20)
# initializing all the base model objects with default parameters
model_1 = LinearRegression()
model_2 = xgb.XGBRegressor()
model_3 = RandomForestRegressor()
# putting all base model objects in one list
all_models = [model_1, model_2, model_3]
# computing the stack features
s_train, s_test = stacking(all_models, X_train, X_test, y_train,
regression=True, n_folds=4)
```

```
# initializing the second-level model

final_model = model_1

# fitting the second level model with stack features

final_model = final_model.fit(s_train, y_train)

# predicting the final output using stacking

pred_final = final_model.predict(X_test)

# printing the root mean squared error between real value and predicted value

print(mean_squared_error(y_test, pred_final))
```

10.5 ENSEMBLE TECHNIQUES – VOTING

```
# importing utility modules

import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import log_loss

# importing machine learning models for prediction
from xgboost import XGBClassifier
from sklearn.linear_model import LogisticRegression

# importing voting classifier

from sklearn.ensemble import VotingClassifier

# loading train data set in dataframe from train_data.csv file

df = pd.read_csv("train_data.csv")

# getting target data from the dataframe

target = df["Weekday"]

# getting train data from the dataframe

train = df.drop("Weekday")

# Splitting between train data into training and validation dataset

X_train, X_test, y_train, y_test = train_test_split(train, target, test_size=0.20)

# initializing all the model objects with default parameters
model_1 = LogisticRegression()
```

```
model_2 = XGBClassifier()

# Making the final model using voting classifier

final_model = VotingClassifier(

    estimators=[('lr', model_1), ('xgb', model_2), ('rf', model_3)], voting='hard')

# training all the model on the train dataset

final_model.fit(X_train, y_train)

# predicting the output on the test dataset

pred_final = final_model.predict(X_test)

# printing log loss between actual and predicted value

print(log_loss(y_test, pred_final))
```

10.6 RANDOM FOREST- BAGGING ATTRIBUTE BAGGING AND VOTING FOR CLASS SELECTION

Random forest is like bootstrapping algorithm with Decision tree (CART) model. Suppose we have 1000 observations in the complete population with 10 variables. Random forest will try to build multiple CART along with different samples and different initial variables. It will take a random sample of 100 observations and then chose 5 initial variables randomly to build a CART model. It will go on repeating the process say about 10 times and then make a final prediction on each of the observations. Final prediction is a function of each prediction. This final prediction can simply be the mean of each prediction.

Random Forest- bagging Attribute bagging and voting for class selection

```
# importing utility modules

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.metrics import log_loss

# importing machine learning models for prediction

from sklearn.ensemble import RandomForestClassifier

# loading train data set in dataframe from train_data.csv file

df = pd.read_csv("train_data.csv")

# getting target data from the dataframe

target = df["Weekday"]
```

```
# getting train data from the dataframe

train = df.drop("Weekday")

# Splitting between train data into training and validation dataset

X_train, X_test, y_train, y_test = train_test_split(train, target, test_size=0.20)

# initializing all the model objects with default parameters

model_3 = RandomForestClassifier()

# training all the model on the train dataset

final_model.fit(X_train, y_train)

# predicting the output on the test dataset

pred_final = final_model.predict(X_test)

# printing log loss between actual and predicted value

print(log_loss(y_test, pred_final))
```

example 2:

```
import pandas as pd

import numpy as np

dataset = pd.read_csv('/content/petrol_consumption.csv')

dataset.head()

X = dataset.iloc[:, 0:4].values

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

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

# Feature Scaling

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

X_train = sc.fit_transform(X_train)

X_test = sc.transform(X_test)

from sklearn.ensemble import Random Forest Regressor

regressor = Random Forest Regressor(n_estimators=20,random_state=0)
```

```
regressor.fit(X_train, y_train)

y_pred = regressor.predict(X_test)

from sklearn import metrics

print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))

print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))

print('Root Mean Squared Error:',

np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
```

Using Random Forest for Classification:

```
import pandas as pd

import numpy as np

dataset = pd.read_csv('/content/bill_authentication.csv')

dataset.head()

X = dataset.iloc[:, 0:4].values

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

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

# Feature Scaling

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

X_train = sc.fit_transform(X_train)

X_test = sc.transform(X_test)

from sklearn.ensemble import Random Forest Classifier

classifier = RandomForestClassifier(n_estimators=20, random_state=0)

classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)

from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

print(confusion_matrix(y_test,y_pred))

print(classification_report(y_test,y_pred))
```

print(accuracy_score(y_test, y_pred))

from sklearn.ensemble import Random Forest Classifier

classifier = Random Forest Classifier(n_estimators=200, random_state=0)

classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)

10.7 SUMMARY

Ensemble means a group of elements viewed as a whole rather than individually. An Ensemble method creates multiple models and combines them to solve it. Ensemble methods help to improve the robustness/generalizability of the model. In this chapter, we had discussed some methods with their implementation in Python.

10.8 REFERENCES

1 Aurelian Géron, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition.

2 Paul J. Deitel, Python Fundamentals.

3 Stuart Russell, Peter Norvig ,Artificial Intelligence – A Modern Approach, , Pearson Education / Prentice Hall of India, 3rd Edition, 2009.

4 EthemAlpaydın, Introduction to Machine Learning, PHI, Third Edition, ISBN No. 978-81-203- 5078-6.

5 Peter Harrington, Machine Learning in Action. Manning Publications, April 2012ISBN 9781617290183.

6 Introduction to Computer Programming using Python, John V Guttag

7 Core Python Programming, R. Nageswara Rao

8 https://talentsprint.com/pages/artificial-intelligence-machine-learning-iiit-hprogram/program-details.pdf

9 https://learning.oreilly.com/library/view/learning-robotics using/9781783287536/cover.html

10 http://www.qboticslabs.com

11 https://subscription.packtpub.com/book/big_data_and_business_intelligence

12 https://scikit-learn.org/0.16/modules/generated/sklearn.lda.LDA.html

13 https://machinelearningmastery.com/ensemble-machine-learning-algorithmspython-scikit-learn/

14 https://www.coursera.org/learn/machine-learning#syllabus

15 https://data-flair.training/blogs/python-ml-data-preprocessing/

*****

UNIT VIII

11

BOOSTING ALGORITHMS

Unit Structure

11.0 BOOSTING ALGORITHM

Boosting algorithms are the exceptional algorithms that are utilized to enhance the existing result of the data model and assist to fix the errors. [1,4,7] They utilize the concept of the weak learner and strong learner discussion through the weighted average values and higher votes values for prediction. They use decision stamp, margin maximizing classification for processing purpose. Machine learning algorithms like AdaBoost or Adaptive boosting Algorithm, Gradient, XG Boosting algorithm and Voting Ensemble are used to follow the process of training for predicting and fine-tuning of the result. [1,4,7]

Example:

Let's understand this with an example of the email, which recognize whether the email, is a spam or not? It can be recognized it by the following conditions:

Spam:

- If an email contains lots of source like that means it is spam.

- If an email contains only one file image, then it is spam.
- If an email contains the message of "You Own a lottery of $xxxxx", that means it is spam.

Not Spam:

- If an email contains some known source, then it is not spam.
- If it contains the official domain like educba.com, etc., that means it is not spam.

The above-mentioned rules are not that powerful to recognize the spam or not; hence these rules are called as weak learners.

To convert weak learner to strong learner, combine the prediction of the weak learner using the following methods.

- Using weighted average.
- Consider prediction has a higher vote.

Consider the above 5 rules; there are 3 votes for spam and 2 votes for not spam. As there is high vote for spam, we consider it as spam.

11.1 HOW IT WORKS?

To choose the right distributions follow the steps as specified:

Step 1: The base Learning algorithm combines each distribution and applies equal weight to each distribution.

Step 2: If any prediction occurs during the first base learning algorithm, then we pay high attention to that prediction error.

Step 3: Repeat step 2 until the limit of the Base Learning algorithm has been reached or high accuracy.

Step 4: Combines the entire weak learner to create one strong prediction rule.

11.2 TYPES OF BOOSTING ALGORITHM

1. AdaBoost (Adaptive Boosting) algorithm
2. Gradient Boosting algorithm
3. XG Boost algorithm
4. Voting Ensemble

11.3 INTRODUCTION TO ADABOOST ALGORITHM

An adaBoost calculation can be utilized to boost the execution of any machine learning calculation. Machine Learning has gotten to be a capable tool which can make predictions based on a huge sum of data. It has ended up so well known in later times that the application of machine learning can be found in our day-to-day exercises [1,4,7]. A common illustration of it is getting proposals for items whereas shopping online based on the past things bought by the client. Machine Learning, frequently alluded to as predictive analysis, can be characterized as the capability of computers to memorize without being programmed unequivocally. As a substitute, it utilizes the algorithms to analyze input data to foresee output inside an specified range [1,4,7].

11.3.1 What is AdaBoost Algorithm?:

Boosting originated from the question of whether a set of weak classifiers could be converted to a strong classifier or not? A weak learner is a learner who is better than random guessing. AdaBoost transforms weak learners or predictors to strong predictors in order to solve problems of classification [1,4,7].

For classification, the final equation can be put as below:

$$F(x) = sign(\sum_{m=1}^{M} \theta_m f_m(x)),$$

Here f_m designates the m^{th} weak classifier, and Θ_m represents its corresponding weight.

11.3.2 How it works?:

AdaBoost can be used to improve the performance of machine learning algorithms. It is used best with weak learners, to achieve high accuracy [1,4,7]. Consider a data set containing n number of points:

$$x_i \in \mathbb{R}^d, y_i \in \{-1, 1\}.$$

-1 represents negative class, and 1 indicate positive. It is initialized as below, the weight for each data point as:

$$w(x_i, y_i) = \frac{1}{n}, i = 1, ..., n.$$

If we consider iteration from 1 to M for m, we will get the below expression:

First, we have to select the weak classifier with the lowest weighted classification error by fitting the weak classifiers to the data set.

$$\epsilon_m = E_{w_m}[1_{y \neq f(x)}]$$

Then calculating the weight for the mth weak classifier as below:

$$\theta_m = \frac{1}{2} ln(\frac{1 - \epsilon_m}{\epsilon_m}).$$

The weight is positive for any classifier with an accuracy > 50%, becomes larger if the classifier is more accurate, and negative if the classifier has an accuracy < 50%. The prediction can be combined by inverting the sign. By inverting the sign of the prediction, a classifier with a 40% accuracy can be converted into a 60% accuracy [1,4,7].

Updating the weight for each data point as below:

$$w_{m+1}(x_i, y_i) = \frac{w_m(x_i, y_i) exp[-\theta_m y_i f_m(x_i)]}{Z_m},$$

Z_m is here the normalization factor. It makes sure that the sum total of all instance weights becomes equal to 1.

11.3.3 What is AdaBoost Algorithm Used for?:

AdaBoost can be used for face detection as it appears to be the standard algorithm for face detection in images. It employs a rejection cascade comprising of numerous layers of classifiers. As the detection window is not recognized at any layer as a face, it gets rejected. The first classifier in the window discards the negative window keeping the computational cost to the least. Even if AdaBoost combines the weak classifiers, the principles of AdaBoost are utilized to find the best features to utilize in each layer of the cascade [1,4,7].

11.3.4 Pros and Cons:

Pros:

AdaBoost Algorithm is it is fast, simple and easy to program. It has the flexibility to be combined with any machine learning algorithm, and doesn't need to tune the parameters except for T. It has been extended to learning problems beyond binary classification, and it is versatile as it can be used with text or numeric data [1,4,7].

Cons:

Weak classifiers being too weak can lead to low margins and overfitting [1,4,7].

11.3.5 Pseudocode of AdaBoost [2,3,6]:

1. Initially set uniform example weights.

2. for Each base learner do:

   Train base learner with a weighted sample.

   Test base learner on all data.

   Set learner weight with a weighted error.

   Set example weights based on ensemble predictions.

3. end for

Implementation of AdaBoost Using Python:

Step 1: Importing the Modules:

Import the required packages and modules.

In Python we have the AdaBoostClassifier and AdaBoostRegressor classes from the scikit-learn library. As we deal we would import AdaBoostClassifier. The train_test_split method is used to split our dataset into training and test sets. We also import datasets, from which we will use the the Iris Dataset [2,3,6].

```
from sklearn.ensemble import AdaBoostClassifier

from sklearn import datasets

from sklearn.model_selection import train_test_split

from sklearn import metrics
```

Step 2: Exploring the data:

This dataset contains four features about different types of Iris flowers (sepal length, sepal width, petal length, petal width). The target is to predict the type of flower from three possibilities: Setosa, Versicolour, and Virginica. The dataset is available in the scikit-learn library, or you can also download it from the UCI Machine Learning Library [2,3,6].

Next, we make our data ready by loading it from the datasets package using the load_iris() method. We assign the data to the iris variable [2,3,6].

Further, we split our dataset into input variable X, which contains the features sepal length, sepal width, petal length, and petal width.

Y is our target variable, or the class that we have to predict: either Iris Setosa, Iris Versicolour, or Iris Virginica. Below is an example of what our data looks like.

```
iris = datasets.load_iris()
X = iris.data
y = iris.target
print(X)
print(Y)

Output:

[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.8 4.  1.2 0.2]
 [5.7 4.4 1.5 0.4]
. . . .
. . . .
]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2]
```

Step 3: Splitting the data:

Splitting the dataset into training and testing datasets is a good idea to see if our model is classifying the data points correctly on unseen data [2,3,6].

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

Split the dataset into 70% training and 30% test.

Step 4: Fitting the Model:

Building the AdaBoost Model. AdaBoost takes Decision Tree as its learner model by default. We make an AdaBoostClassifier object and name it **abc** [2,3,6]. Few important parameters of AdaBoost are :

- base_estimator: It is a weak learner used to train the model.
- n_estimators: Number of weak learners to train in each iteration.
- learning_rate: It contributes to the weights of weak learners. It uses 1 as a default value.

```
abc = AdaBoostClassifier(n_estimators=50,
                         learning_rate=1)
```

We then go ahead and fit our object abc to our training dataset. We call it a model.

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

Step 5: Making the Predictions:

Our next step would be to see how good or bad our model is to predict our target values.

y_pred = model.predict(X_test)

Step 6: Evaluating the model:

The Model accuracy will tell us how many times our model predicts the correct classes.

print("Accuracy:", metrics.accuracy_score(y_test, y_pred))

Output:

Accuracy:0.8666666666666667

An accuracy of 86.66% is achieved.

11.4 GRADIENT BOOSTING ALGORITHM

Gradient boosting algorithm is a machine learning technique used to define loss function and reduce it [4,7,8]. It is also used to solve problems of classification using various prediction models involving the following steps:

1. Loss Function:

The use of the loss function depends on the type of problem. The advantage of gradient boosting is that there is no need for a new boosting algorithm for each loss function [4,7,8].

2. Weak Learner:

In gradient boosting, decision trees are used as a weak learner. A regression tree is used to give true values, which can be combined together to create correct predictions. Like in the AdaBoost algorithm, small trees with a single split are used, i.e. decision stump. Larger trees are used for large levels I,e 4-8 levels [4,7,8].

3. Additive Model:

In this model, trees are added one at a time. existing trees remains the same. During the addition of trees, gradient descent is used to minimize the loss function.

The Gradient Boosting Machine is a powerful ensemble machine learning algorithm that uses decision trees.

Gradient boosting is a generalization of AdaBoosting, improving the performance of the approach and introducing ideas from bootstrap aggregation to further improve the models, such as randomly sampling the samples and features when fitting ensemble members.

Gradient boosting performs well, if not the best, on a wide range of tabular datasets, and versions of the algorithm like XGBoost and LightBoost often play an important role in winning machine learning competitions [4,7,8].

Gradient Boosting ensemble is an ensemble created from decision trees added sequentially to the model.

11.4 GRADIENT BOOSTING MACHINES ALGORITHM

Gradient boosting refers to a class of ensemble machine learning algorithms that can be used for classification or regression predictive modeling problems.

Gradient boosting is also known as gradient tree boosting, stochastic gradient boosting, and gradient boosting machines. Models are fit using any arbitrary differentiable loss function and gradient descent optimization algorithm. This gives the technique its name, "gradient boosting," as the loss gradient is minimized as the model is fit, much like a neural network [4,7,8].

Gradient boosting works by building weak prediction models sequentially where each model tries to predict the error left over by the previous model. Because of this, the algorithm tends to over-fit rather quick.

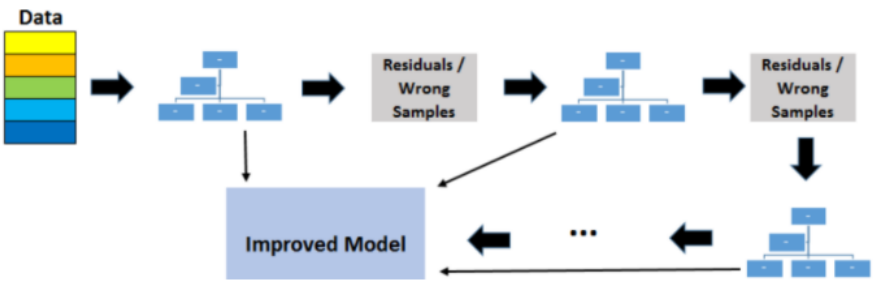
Implementations of the algorithm:

1. Gradient Boosting from scratch

2. Using the scikit-learn in-built function.

In gradient boosting decision trees, we combine many weak learners to come up with one strong learner. The weak learners here are the individual decision trees. All the trees are connected in series and each tree tries to minimise the error of the previous tree. Sequential boosting algorithms are slow to learn, but highly accurate [1,4,7].

Image('residual.png')



The weak learners are fit in such a way that each new learner fits into the residuals of the previous step so as the model improves. The final model aggregates the result of each step and thus a strong learner is achieved. A loss function is used to detect the residuals. Mean squared error (MSE) is

used for a regression task and logarithmic loss (log loss) is used for classification tasks [1,4,7].

Learning rate and n_estimators (Hyperparameters):

Hyperparemetes are key parts of learning algorithms which influence the performance and accuracy of a model. Learning rate and n_estimators are two basic hyperparameters for gradient boosting decision trees. Learning rate, signified as α, basically implies how quick the show learns. Each tree added modifies the overall model. The size of the modification is controlled by learning rate. Learning rate is proportional to model learns. The advantage of slower learning rate is that the model becomes more robust and efficient [1,4,7].

Note:

Problem in gradient boosting decision trees is overfitting due to addition of too many trees whereas in random forests, addition of too many tress won't cause overfitting.

Algorithm:

Let's say the output model y when fit to only 1 decision tree, is given by $$A_1 + B_1x +e_1

where $e1$ is there sidual from this decision tree. In gradient boosting, we fit the consecutive decision trees on there sidual from the last one [1,4,7]. So when gradient boosting is applied to this model, the consecutive decision trees will be mathematically represented as:

e_1 = A_2 + B_2x + e_2

e_2 = A_3 + B_3x + e_3

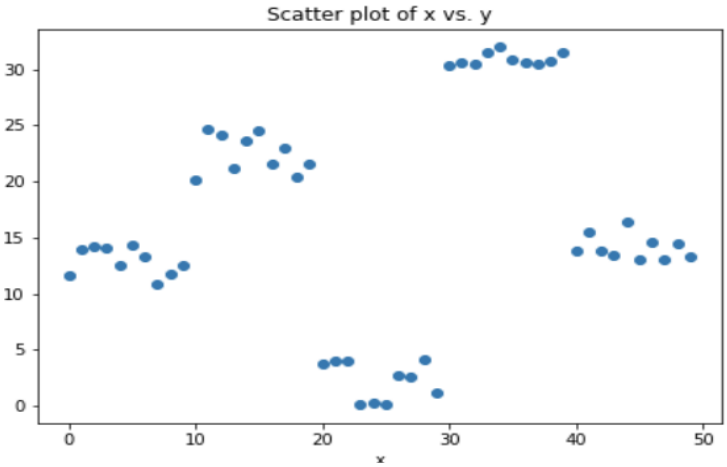
Note that here we stop at 3 decision trees, but in an actual gradient boosting model, the number of learners or decision trees is much more [1,4,7]. The final model of the decision tree will be given by:

y = A_1 + A_2 + A_3 + B_1x + B_2x + B_3x + e_3 $$

11.4.1 Implementation:

Implementation from Scratch

Consider simulated data as shown in scatter plot below with 1 input (x) and 1 output (y) variables.




```
# The above plot has been generated using the following code
x = np.arange(0, 50)
x = pd.DataFrame({'x': x})
# just random uniform distributions in differnt range
y1 = np.random.uniform(10, 15, 10)
y2 = np.random.uniform(20, 25, 10)
y3 = np.random.uniform(0, 5, 10)
y4 = np.random.uniform(30, 32, 10)
y5 = np.random.uniform(13, 17, 10)
y = np.concatenate((y1, y2, y3, y4, y5))
y = y[:, None]
Fit a decision tree on data. [call x as input and y as output]
xi = x  # initialization of input
yi = y  # initialization of target
# x,y --> use where no need to change original y
ei = 0  # initialization of error
n = len(yi)  # number of rows
predf = 0  # initial prediction 0
for i in range(30):  # loop will make 30 trees (n_estimators).
    # DecisionTree scratch code can be found on www.kaggle.com/grroverpr/gradient-boosting-
simplified
    tree = DecisionTree(xi, yi)
    # It just create a single decision tree with provided min. sample leaf
    # For selected input variable, this splits (<n and >n) data so that std. deviation of
    tree.find_better_split(0)
    # target variable in both splits is minimum as compared to all other splits
    # finds index where this best split occurs
    r = np.where(xi == tree.split)[0][0]
    left_idx = np.where(xi <= tree.split)[0]  # index lhs of split
    right_idx = np.where(xi > tree.split)[0]  # index rhs of split
```

Calculate error residuals. Actual target value, minus predicted target value [e1= y – y_predicted1]

Fit a new model on error residuals as target variable with same input variables [call it e1_predicted]

Add the predicted residuals to the previous predictions [y_predicted2 = y_predicted1 + e1_predicted]

Fit another model on residuals that is still left. i.e. [e2 = y – y_predicted2] and repeat steps 2 to 5 until it starts overfitting or the sum of residuals become constant. Overfitting can be controlled by consistently checking accuracy on validation data.

```
# predictions by ith decisision tree
predi = np.zeros(n)
# replace left side mean y
np.put(predi, left_idx, np.repeat(np.mean(yi[left_idx]), r))
np.put(predi, right_idx, np.repeat(
   np.mean(yi[right_idx]), n-r))  # right side mean y
predi = predi[:, None]  # make long vector (nx1) in compatible with y
# final prediction will be previous prediction value + new prediction of residual
predf = predf + predi
ei = y - predf  # needed originl y here as residual always from original y
yi = ei  # update yi as residual to reloop
```

The code above is a very basic implementation of gradient boosting trees. The actual libraries have a lot of hyperparameters that can be tuned for better results. This can be better understood by using the gradient boosting algorithm on a real dataset.

11.4.2 Implementation using Scikit-learn:

Using the PIMA Indians Diabetes dataset, which has information about a an individual's health parameters and an output of 0 or 1, depending on whether or not he has diabates. The task here is classify a individual as diabetic, when given the required inputs about his health.

```
# Relevant libraries
from sklearn.ensemble import GradientBoostingClassifier
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn import preprocessing
import warnings
warnings.filterwarnings("ignore")
```

Now load the dataset and look at the columns to understand the given information better.

 # Load the dataset [1,4,7]

```
pima = pd.read_csv('diabetes.csv')
pima.head()
```

Diabetes Table For training and testing our model, the data has to be divided into train and test data. We will also scale the data to lie between 0 and 1.

 # Split dataset into test and train data

```
X_train, X_test, y_train, y_test = train_test_split(pima.drop('Outcome', axis=1),
                                                    pima['Outcome'], test_size=0.2)
```

 # Scale the data [1,4,7]

```
scaler = preprocessing.StandardScaler().fit(X_train)
X_train_transformed = scaler.transform(X_train)
X_test_transformed = scaler.transform(X_test)
```

Now that the data has been sufficiently preprocessed, let's go ahead with defining the Gradient Boosting Classifier along with it's hyperparameters. Next, we will fit this model on the training data.

 # Define Gradient Boosting Classifier with hyperparameters [1,4,7]

```
gbc=GradientBoostingClassifier(n_estimators=500,learning_rate=0.05,random_state=100,max_features=5 )
```

 # Fit train data to GBC

```
gbc.fit(X_train_transformed, y_train)
GradientBoostingClassifier(criterion='friedman_mse', init=None,
                           learning_rate=0.05, loss='deviance', max_depth=3,
                           max_features=5, max_leaf_nodes=None,
                           min_impurity_decrease=0.0, min_impurity_split=None,
                           min_samples_leaf=1, min_samples_split=2,
                           min_weight_fraction_leaf=0.0, n_estimators=500,
                           n_iter_no_change=None, presort='auto',
                           random_state=100, subsample=1.0, tol=0.0001,
                           validation_fraction=0.1, verbose=0,
                           warm_start=False)
```

 # Confusion matrix will give number of correct and incorrect classifications

```
print(confusion_matrix(y_test, gbc.predict(X_test_transformed)))
[[82 17]
 [25 30]]
```

The number of misclassifications by the Gradient Boosting Classifier are 42, comapared to 112 correct classifications. The model has performed decently.

 # Accuracy of model

```
print("GBC accuracy is %2.2f" % accuracy_score(y_test, gbc.predict(X_test_transformed)))
```

GBC accuracy is 0.73

The accuracy is 73%, which is average. This can be improved by tuning the hyperparameters or processing the data to remove outliers.

Improving perfomance of gradient boosted decision trees [1,4,7]:

Gradient boosting algorithms are prone to overfitting and consequently poor perfomance on test dataset. There are some pointers you can keep in mind to improve the perfomance of gradient boosting algorithm.

11.4.3 Stochastic Gradient Boosting:

Stochastic gradient boosting involves sub sampling the training dataset and training individual learners on random samples created by this sub sampling. This reduces the correlation between results from individual learners and combining results with low correlation provides us with a better overall result.

11.4.4 Shrinkage:

The predictions of each tree are added together sequentially. Instead, the contribution of each tree to this sum can be weighted to slow down the learning by the algorithm. This weighting is called a shrinkage or a learning rate. Using a low learning rate can dramatically improve the perfomance of your gradient boosting model. Usually a learning rate in the range of 0.1 to 0.3 gives the best results [1,4,7].

11.4.5 Regularization:

L1 and L2 regularization penalties can be implemented on leaf weight values to slow down learning and prevent over-fitting. Gradient tree boosting implementations often also use regularization by limiting the minimum number of observations in trees' terminal nodes.

11.4.6 Tree Constraints:

There are a number of ways in which a tree can be constrained to improve performance.

- Number of trees
- Tree depth
- Minimum improvement in loss
- Number of observations per split

*****

12

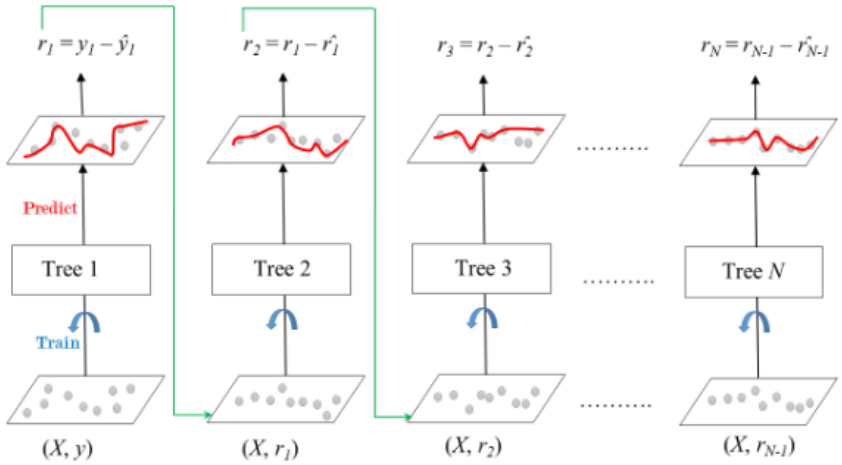
EXAMPLES

Unit Structure

12.1 EXAMPLE 1

Gradient Boosting is a popular boosting algorithm. In gradient boosting, each predictor corrects its predecessor's error. There is a technique called the Gradient Boosted Trees whose base learner is CART (Classification and Regression Trees) [5].

The below diagram explains how gradient boosted trees are trained for regression problems.



Gradient Boosted Trees for Regression:

The ensemble consists of N trees. Tree1 is trained using the feature matrix X and the labels y. The predictions labelled y1(hat) are used to determine the training set residual errors r1. Tree2 is then trained using the feature matrix X and the residual errors r1 of Tree1 as labels. The predicted

results r1(hat) are then used to determine the residual r2. The process is repeated until all the N trees forming the ensemble are trained [5].

There is an important parameter used in this technique known as **Shrinkage**.

Shrinkage refers to the fact that the prediction of each tree in the ensemble is shrunk after it is multiplied by the learning rate (eta) which ranges between 0 to 1. There is a trade-off between eta and number of estimators, decreasing learning rate needs to be compensated with increasing estimators in order to reach certain model performance. Since all trees are trained now, predictions can be made [5].

Each tree predicts a label and final prediction is given by the formula,

y(pred) = y1 + (eta * r1) + (eta * r2) + ... + (eta * rN)

The class of the gradient boosting regression in scikit-learn is **GradientBoostingRegressor.** A similar algorithm is used for classification known as **GradientBoostingClassifier**.

```
# Import models and utility functions
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error as MSE
from sklearn import datasets
# Setting SEED for reproducibility [5]
SEED = 1
# Importing the dataset
bike = datasets.load_bike()
X, y = bike.data, bike.target
# Splitting dataset
train_X, test_X, train_y, test_y = train_test_split(X, y, test_size = 0.3, random_state = SEED)
# Instantiate Gradient Boosting Regressor
gbr = GradientBoostingRegressor(n_estimators = 200, max_depth = 1, random_state = SEED)
# Fit to training set
gbr.fit(train_X, train_y)
# Predict on test set
pred_y = gbr.predict(test_X)
# test set RMSE
test_rmse = MSE(test_y, pred_y) ** (1 / 2)
# Print rmse
print('RMSE test set: {:.2f}'.format(test_rmse))
```

Output:

```
RMSE test set: 4.01
```

12.2 EXAMPLE 2

Gradient Boosting Scikit-Learn API:

Using a modern version of the library by running the following script:

```
# check scikit-learn version
import sklearn
print(sklearn.__version__)
```

Running the script will print your version of scikit-learn.

Running the script will print your version of scikit-learn.

Gradient boosting is provided via the Gradient Boosting Regressor and Gradient Boosting Classifier classes.

Both models operate the same way and take the same arguments that influence how the decision trees are created and added to the ensemble.

Randomness is used in the construction of the model. This means that each time the algorithm is run on the same data, it will produce a slightly different model.

When using machine learning algorithms that have a stochastic learning algorithm, it is good practice to evaluate them by averaging their performance across multiple runs or repeats of cross-validation. When fitting a final model, it may be desirable to either increase the number of trees until the variance of the model is reduced across repeated evaluations, or to fit multiple final models and average their predictions [3,9].

Let's take a look at how to develop a Gradient Boosting ensemble for both classification and regression.

12.3 GRADIENT BOOSTING FOR CLASSIFICATION [1, 4, 7]

In this section, we will look at using Gradient Boosting for a classification problem.

First, we can use the make_classification() function to create a synthetic binary classification problem with 1,000 examples and 20 input features.

The complete example is listed below.

```
# 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=7)
# summarize the dataset
print(X.shape, y.shape)
```

Running the example creates the dataset and summarizes the shape of the input and output components.

1. (1000, 20) (1000,)

Next, we can evaluate a Gradient Boosting algorithm on this dataset [3,9]..

We will evaluate the model using repeated stratified k-fold cross-validation, with three repeats and 10 folds. We will report the mean and standard deviation of the accuracy of the model across all repeats and folds [1].

```
# evaluate gradient boosting algorithm for classification
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.ensemble import GradientBoostingClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
random_state=7)
# define the model
model = GradientBoostingClassifier()
# define the evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model on the dataset
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Running the example reports the mean and standard deviation accuracy of the model.

Gradient Boosting ensemble with default hyperparameters achieves a classification accuracy of about 89.9 percent on this test dataset.

Mean Accuracy: 0.899 (0.030)

First, the Gradient Boosting ensemble is fit on all available data, then the predict() function can be called to make predictions on new data.

The example below demonstrates this on our binary classification dataset.

```
from sklearn.datasets import make_classification
from sklearn.ensemble import GradientBoostingClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
random_state=7)
# define the model
model = GradientBoostingClassifier()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [0.2929949, -4.21223056, -1.288332, -2.17849815, -0.64527665, 2.58097719,
0.28422388, -7.1827928, -1.91211104, 2.73729512, 0.81395695, 3.96973717, -2.66939799,
3.34692332, 4.19791821, 0.99990998, -0.30201875, -4.43170633, -2.82646737, 0.44916808]
yhat = model.predict([row])
# summarize prediction
print('Predicted Class: %d' % yhat[0])
```

Running the example fits the Gradient Boosting ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

Predicted Class: 1

Now that we are familiar with using Gradient Boosting for classification, let's look at the API for regression.

12.4 GRADIENT BOOSTING FOR REGRESSION

Using make_regression() function to create a synthetic regression problem with 1,000 examples and 20 input features.

The complete example is listed below.

```
# 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=7)
# summarize the dataset
print(X.shape, y.shape)
```

Running the example creates the dataset and summarizes the shape of the input and output components.

1. (1000, 20) (1000,)

Next, we can evaluate a Gradient Boosting algorithm on this dataset.

As we did with the last section, we will evaluate the model using repeated k-fold cross-validation, with three repeats and 10 folds. We will report the mean absolute error (MAE) of the model across all repeats and folds. The scikit-learn library makes the MAE negative so that it is maximized instead of minimized. This means that larger negative MAE are better and a perfect model has a MAE of 0.

The complete example is listed below [1].

```
# evaluate gradient boosting ensemble 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.ensemble import GradientBoostingRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
random_state=7)
# define the model
model = GradientBoostingRegressor()
# define the evaluation procedure
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model
n_scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv,
n_jobs=-1)
# report performance
print('MAE: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Running the example reports the mean and standard deviation accuracy of the model.

In this case, we can see the Gradient Boosting ensemble with default hyperparameters achieves a MAE of about 62.

1. MAE: -62.475 (3.254)

We can also use the Gradient Boosting model as a final model and make predictions for regression.

First, the Gradient Boosting ensemble is fit on all available data, then the predict() function can be called to make predictions on new data.

The example below demonstrates this on our regression dataset [1].

```
# gradient boosting ensemble for making predictions for regression
from sklearn.datasets import make_regression
from sklearn.ensemble import GradientBoostingRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
random_state=7)
# define the model
model = GradientBoostingRegressor()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [0.20543991, -0.97049844, -0.81403429, -0.23842689, -0.60704084, -0.48541492,
0.53113006, 2.01834338, -0.90745243, -1.85859731, -1.02334791, -0.6877744, 0.60984819, -
0.70630121, -1.29161497, 1.32385441, 1.42150747, 1.26567231, 2.56569098, -0.11154792]
yhat = model.predict([row])
# summarize prediction
print('Prediction: %d' % yhat[0])
```

Running the example fits the Gradient Boosting ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

Prediction: 37

Now that we are familiar with using the scikit-learn API to evaluate and use Gradient Boosting ensembles, let's look at configuring the model [1].

12.5 GRADIENT BOOSTING HYPERPARAMETERS

The number of trees can be set via the "n_estimators" argument and defaults to 100.

The example below explores the effect of the number of trees with values between 10 to 5,000.

```
# explore gradient boosting number of trees effect on performance
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.ensemble import GradientBoostingClassifier
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=7)
        return X, y
```

```
# get a list of models to evaluate
def get_models():
	models = dict()
	# define number of trees to consider
	n_trees = [10, 50, 100, 500, 1000, 5000]
	for n in n_trees:
		models[str(n)] = GradientBoostingClassifier(n_estimators=n)
	return models
# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
	# define the evaluation procedure
	cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
	# evaluate the model and collect the results
	scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
	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():
	# evaluate the model
	scores = evaluate_model(model, X, y)
	# store the results
	results.append(scores)
	names.append(name)
	# summarize the performance along the way
	print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

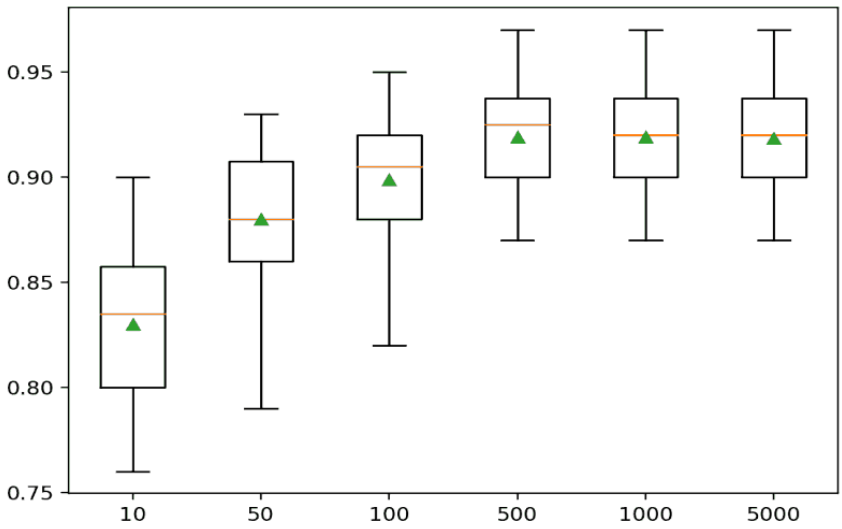
Running the example first reports the mean accuracy for each configured number of decision trees.

In this case, we can see that that performance improves on this dataset until about 500 trees, after which performance appears to level off. Unlike AdaBoost, Gradient Boosting appears to not overfit as the number of trees is increased in this case [1].

```
1 >10 0.830 (0.037)

2 >50 0.880 (0.033)

3 >100 0.899 (0.030)

4 >500 0.919 (0.025)

5 >1000 0.919 (0.025)

6 >5000 0.918 (0.026)
```

A box and whisker plot is created for the distribution of accuracy scores for each configured number of trees.

We can see the general trend of increasing model performance and ensemble size.



Box Plot of Gradient Boosting Ensemble Size vs. Classification Accuracy

12.6 EXPLORE NUMBER OF SAMPLES

The number of samples used to fit each tree can be varied. This means that each tree is fit on a randomly selected subset of the training dataset [1, 4, 7].

Using fewer samples introduces more variance for each tree, although it can improve the overall performance of the model.

The number of samples used to fit each tree is specified by the "subsample" argument and can be set to a fraction of the training dataset size. By default, it is set to 1.0 to use the entire training dataset.

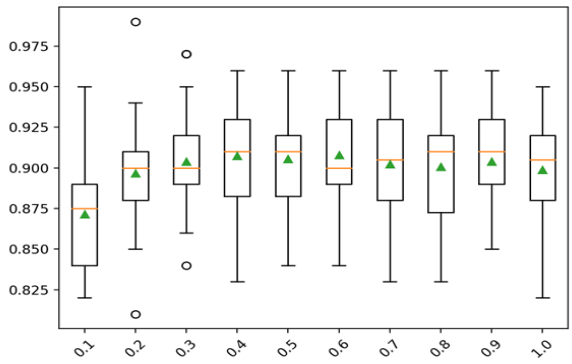
The example below demonstrates the effect of the sample size on model performance [1, 4, 7].

```
# explore gradient boosting ensemble number of samples effect on performance
from numpy import mean
from numpy import std
from numpy import arange
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import GradientBoostingClassifier
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=7)
        return X, y
# get a list of models to evaluate
def get_models():
        models = dict()
```

```
        # explore sample ratio from 10% to 100% in 10% increments
        for i in arange(0.1, 1.1, 0.1):
                key = '%.1f' % i
                models[key] = GradientBoostingClassifier(subsample=i)
        return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
        # define the evaluation procedure
        cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
        # evaluate the model and collect the results
        scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)

        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():
        # evaluate the model
        scores = evaluate_model(model, X, y)
        # store the results
        results.append(scores)
        names.append(name)
        # summarize the performance along the way
        print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

In this case, we can see that mean performance is probably best for a sample size that is about half the size of the training dataset, such as 0.4 or higher [1, 4, 7].

```
>0.1 0.872 (0.033)
>0.2 0.897 (0.032)
>0.3 0.904 (0.029)
>0.4 0.907 (0.032)
>0.5 0.906 (0.027)
>0.6 0.908 (0.030)
>0.7 0.902 (0.032)
>0.8 0.901 (0.031)
>0.9 0.904 (0.031)
>1.0 0.899 (0.030)
```



Box Plot of Gradient Boosting Ensemble Sample Size vs. Classification Accuracy

12.7 EXPLORE NUMBER OF FEATURES [1, 4, 7]

The number of features used to fit each decision tree can be varied.

Like changing the number of samples, changing the number of features introduces additional variance into the model, which may improve performance, although it might require an increase in the number of trees.

The number of features used by each tree is taken as a random sample and is specified by the "max_features" argument and defaults to all features in the training dataset.

The example below explores the effect of the number of features on model performance for the test dataset between 1 and 20.

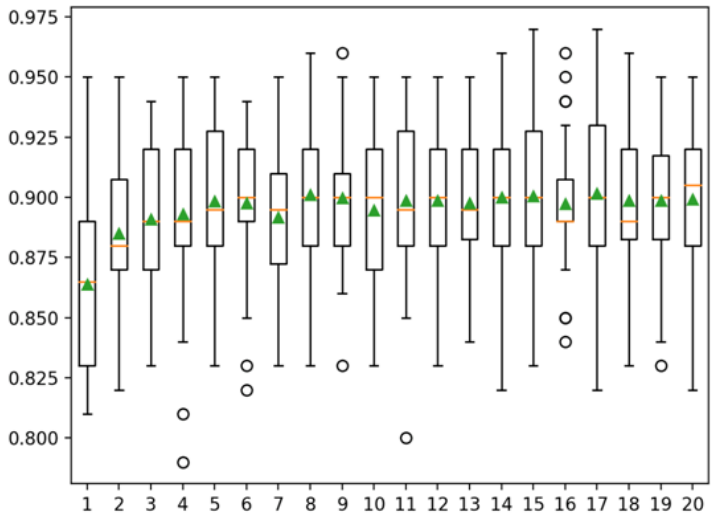
```
# explore gradient boosting number of features on performance
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import GradientBoostingClassifier
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=7)
	return X, y
# get a list of models to evaluate
def get_models():

	models = dict()
	# explore number of features from 1 to 20
	for i in range(1,21):
		models[str(i)] = GradientBoostingClassifier(max_features=i)
	return models
# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
	# define the evaluation procedure
	cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
	# evaluate the model and collect the results
	scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
	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():
        # evaluate the model
        scores = evaluate_model(model, X, y)
        # store the results
        results.append(scores)
        names.append(name)
        # summarize the performance along the way
        print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

```
1  >1 0.864 (0.036)
2  >2 0.885 (0.032)
3  >3 0.891 (0.031)
4  >4 0.893 (0.036)
5  >5 0.898 (0.030)
6  >6 0.898 (0.032)
7  >7 0.892 (0.032)
8  >8 0.901 (0.032)
9  >9 0.900 (0.029)
10 >10 0.895 (0.034)
11 >11 0.899 (0.032)
12 >12 0.899 (0.030)
13 >13 0.898 (0.029)
14 >14 0.900 (0.033)
15 >15 0.901 (0.032)
16 >16 0.897 (0.028)
17 >17 0.902 (0.034)
18 >18 0.899 (0.032)
19 >19 0.899 (0.032)
20 >20 0.899 (0.030)
```

A box and whisker plot is created for the distribution of accuracy scores for each configured number of trees [1, 4, 7].

We can see the general trend of increasing model performance perhaps peaking around eight or nine features and staying somewhat level.



Box Plot of Gradient Boosting Ensemble Number of Features vs. Classification Accuracy

12.8 EXPLORE LEARNING RATE [1, 4, 7]

Learning rate controls the amount of contribution that each model has on the ensemble prediction. Smaller rates may require more decision trees in the ensemble, whereas larger rates may require an ensemble with fewer trees. It is common to explore learning rate values on a log scale, such as between a very small value like 0.0001 and 1.0. The learning rate can be controlled via the "learning_rate" argument and defaults to 0.1.

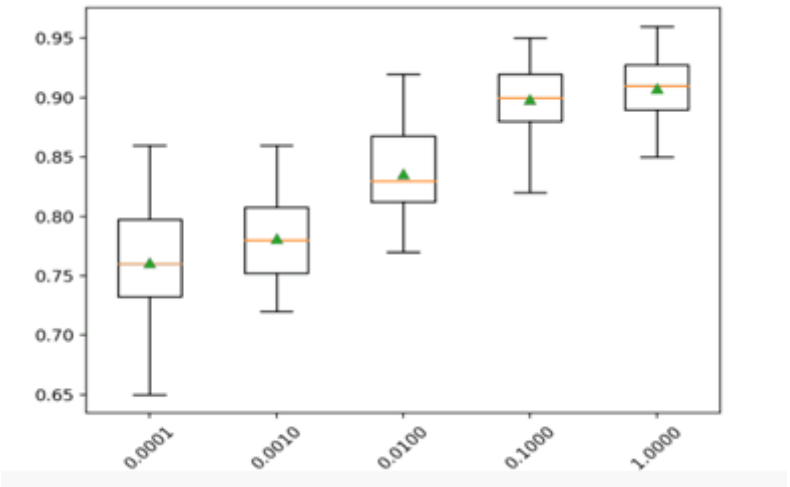
The example below explores the learning rate and compares the effect of values between 0.0001 and 1.0.

```
# explore gradient boosting ensemble learning rate effect on performance
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.ensemble import GradientBoostingClassifier
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=7)
        return X, y
# get a list of models to evaluate
def get_models():
        models = dict()
        # define learning rates to explore
        for i in [0.0001, 0.001, 0.01, 0.1, 1.0]:
                key = '%.4f' % i
                models[key] = GradientBoostingClassifier(learning_rate=i)
        return models
# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
        # define the evaluation procedure
        cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
        # evaluate the model and collect the results
        scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
        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():
        # evaluate the model
        scores = evaluate_model(model, X, y)
        # store the results
        results.append(scores)
        names.append(name)
        # summarize the performance along the way
        print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

This highlights the trade-off between the number of trees (speed of training) and learning rate, e.g. we can fit a model faster by using fewer trees and a larger learning rate.

```
1 >0.0001 0.761 (0.043)
2 >0.0010 0.781 (0.034)
3 >0.0100 0.836 (0.034)
4 >0.1000 0.899 (0.030)
5 >1.0000 0.908 (0.025)
```

A box and whisker plot is created for the distribution of accuracy scores for each configured number of trees.



Box Plot of Gradient Boosting Ensemble Learning Rate vs. Classification Accuracy

12.9 EXPLORE TREE DEPTH [1, 4, 7]

Like varying the number of samples and features used to fit each decision tree, varying the depth of each tree is another important hyperparameter for gradient boosting.

The tree depth controls how specialized each tree is to the training dataset: how general or overfit it might be. Trees are preferred that are not too shallow and general and not too deep and specialized.

Gradient boosting performs well with trees that have a modest depth finding a balance between skill and generality [1, 4, 7].

Tree depth is controlled via the “max_depth” argument and defaults to 3.

The example below explores tree depths between 1 and 10 and the effect on model performance.

```
# explore gradient boosting tree depth effect on performance
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.ensemble import GradientBoostingClassifier
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=7)
        return X, y
# get a list of models to evaluate
def get_models():
        models = dict()
        # define max tree depths to explore between 1 and 10
        for i in range(1,11):
                models[str(i)] = GradientBoostingClassifier(max_depth=i)
        return models
# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
        # define the evaluation procedure
        cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
        # evaluate the model and collect the results
        scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
        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():
        # evaluate the model
        scores = evaluate_model(model, X, y)
        # store the results
        results.append(scores)
        names.append(name)
        # summarize the performance along the way
        print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

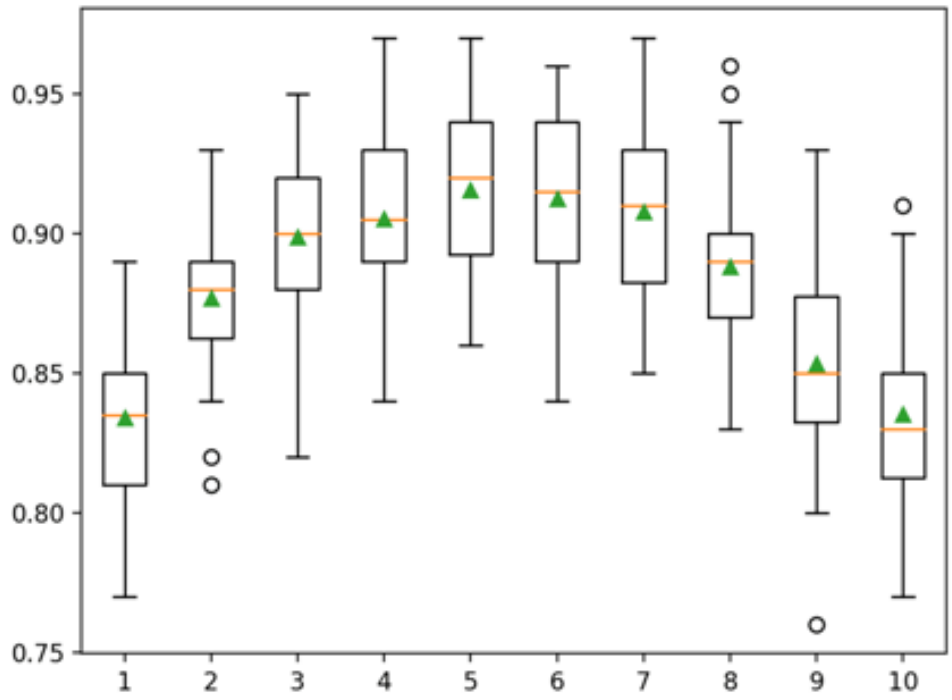
Running the example first reports the mean accuracy for each configured tree depth.

Performance improves with tree depth, perhaps peaking around a depth of 3 to 6, after which the deeper, more specialized trees result in worse performance.

```
1  >1 0.834 (0.031)
2  >2 0.877 (0.029)
3  >3 0.899 (0.030)
4  >4 0.905 (0.032)
5  >5 0.916 (0.030)
6  >6 0.912 (0.031)
7  >7 0.908 (0.033)
8  >8 0.888 (0.031)
9  >9 0.853 (0.036)
10 >10 0.835 (0.034)
```

A box and whisker plot is created for the distribution of accuracy scores for each configured tree depth.

We can see the general trend of increasing model performance with the tree depth to a point, after which performance begins to degrade rapidly with the over-specialized trees.



Box Plot of Gradient Boosting Ensemble Tree Depth vs. Classification Accuracy

12.10 GRID SEARCH HYPERPARAMETERS [1,4,7]

Gradient boosting can be challenging to configure as the algorithm as many key hyperparameters that influence the behavior of the model on training data and the hyperparameters interact with each other.

As such, it is a good practice to use a search process to discover a configuration of the model hyperparameters that works well or best for a given predictive modeling problem. Popular search processes include a random search and a grid search.

In this section we will look at grid searching common ranges for the key hyperparameters for the gradient boosting algorithm that you can use as starting point for your own projects. This can be achieving using the GridSearchCV class and specifying a dictionary that maps model hyperparameter names to the values to search.

In this case, we will grid search four key hyperparameters for gradient boosting: the number of trees used in the ensemble, the learning rate, subsample size used to train each tree, and the maximum depth of each tree. We will use a range of popular well performing values for each hyperparameter.

Each configuration combination will be evaluated using repeated k-fold cross-validation and configurations will be compared using the mean score, in this case, classification accuracy.

The complete example of grid searching the key hyperparameters of the gradient boosting algorithm on our synthetic classification dataset is listed below.

```
# example of grid searching key hyperparameters for gradient boosting on a classification dataset
from sklearn.datasets import make_classification
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import GradientBoostingClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5, random_state=7)
# define the model with default hyperparameters
model = GradientBoostingClassifier()
# define the grid of values to search
grid = dict()
grid['n_estimators'] = [10, 50, 100, 500]
grid['learning_rate'] = [0.0001, 0.001, 0.01, 0.1, 1.0]
grid['subsample'] = [0.5, 0.7, 1.0]
grid['max_depth'] = [3, 7, 9]
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define the grid search procedure
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='accuracy')
# execute the grid search
grid_result = grid_search.fit(X, y)
# summarize the best score and configuration
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
# summarize all scores that were evaluated
```

```
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Running the example many take a while depending on your hardware. At the end of the run, the configuration that achieved the best score is reported first, followed by the scores for all other configurations that were considered.

A configuration with a learning rate of 0.1, max depth of 7 levels, 500 trees and a subsample of 70% performed the best with a classification accuracy of about 94.6 percent.

The model may perform even better with more trees such as 1,000 or 5,000 although these configurations were not tested in this case to ensure that the grid search completed in a reasonable time.

```
Best: 0.946667 using {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 500, 'subsample': 0.7}

0.529667 (0.089012) with: {'learning_rate': 0.0001, 'max_depth': 3, 'n_estimators': 10,
'subsample': 0.5}

0.525667 (0.077875) with: {'learning_rate': 0.0001, 'max_depth': 3, 'n_estimators': 10,
'subsample': 0.7}

0.524000 (0.072874) with: {'learning_rate': 0.0001, 'max_depth': 3, 'n_estimators': 10,
'subsample': 1.0}

0.772667 (0.037500) with: {'learning_rate': 0.0001, 'max_depth': 3, 'n_estimators': 50,
'subsample': 0.5}

0.767000 (0.037696) with: {'learning_rate': 0.0001, 'max_depth': 3, 'n_estimators': 50,
'subsample': 0.7}
```

*****

UNIT IX

13

XG BOOST

Unit Structure

13.1 XG BOOST

Extreme Gradient Boosting (XG Boost) is an upgraded implementation of the Gradient Boosting Algorithm, which is developed for high computational speed, scalability, and better performance **[1-4,7]**.

XG Boost has various features, which are as follows:

1. Parallel Processing
2. Cross-Validation
3. Cache Optimization
4. Distributed Computing

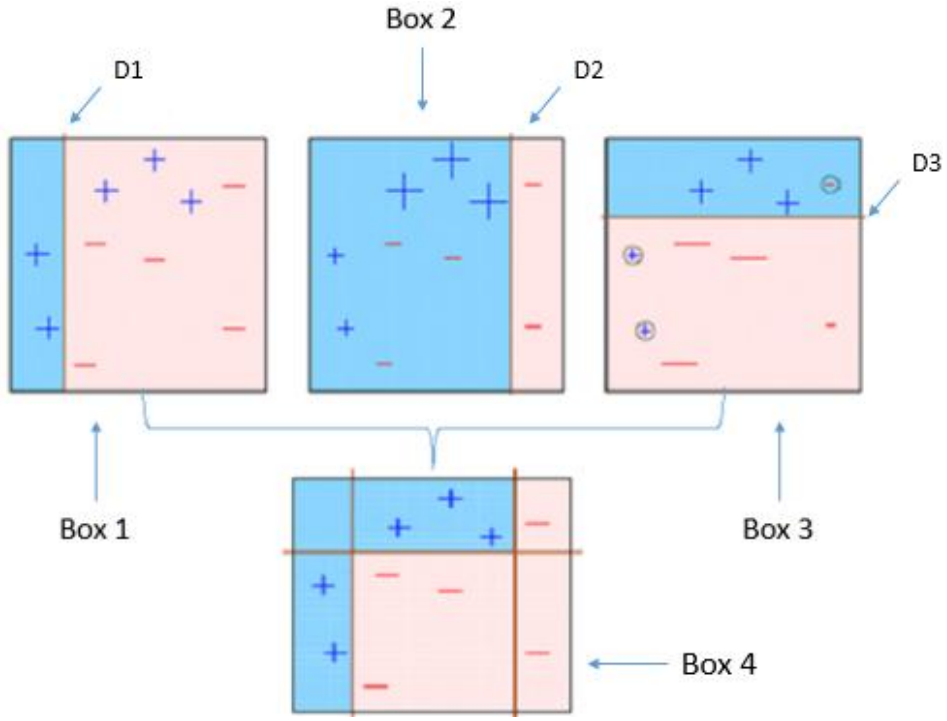
XGBoost is becoming popular:

- Speed and performance
- Core algorithm is parallelizable
- Consistently outperforms other algorithm methods
- Wide variety of tuning parameters

XGBoost (Extreme Gradient Boosting) belongs to a family of boosting algorithms and uses the gradient boosting (GBM) framework at its core. It is an optimized distributed gradient boosting library. But wait, what is boosting? Well, keep on reading.

13.1.0 Boosting [1-4,7]:

Boosting is a sequential technique which works on the principle of an ensemble. It combines a set of weak learners and delivers improved prediction accuracy. At any instant t, the model outcomes are weighed based on the outcomes of previous instant t-1. The outcomes predicted correctly are given a lower weight and the ones miss-classified are weighted higher. Let's understand boosting in general with a simple illustration.



Four classifiers (in 4 boxes), shown above, are trying to classify + and - classes as homogeneously as possible.

1. Box 1: The first classifier (usually a decision stump) creates a vertical line (split) at D1. It says anything to the left of D1 is + and anything to the right of D1 is -. However, this classifier misclassifies three + points.

Note: a Decision Stump is a Decision Tree model that only splits off at one level, therefore the final prediction is based on only one feature.

2. Box 2: The second classifier gives more weight to the three + misclassified points (see the bigger size of +) and creates a vertical line at D2. Again it says, anything to the right of D2 is - and left is +. Still, it makes mistakes by incorrectly classifying three - points.

3. Box 3: Again, the third classifier gives more weight to the three - misclassified points and creates a horizontal line at D3. Still, this classifier fails to classify the points (in the circles) correctly.

4. Box 4: This is a weighted combination of the weak classifiers (Box 1,2 and 3). As you can see, it does a good job at classifying all the points correctly.

That's the basic idea behind boosting algorithms is building a weak model, making conclusions about the various feature importance and parameters, and then using those conclusions to build a new, stronger model and capitalize on the misclassification error of the previous model and try to reduce it. Now, let's come to XGBoost. To begin with, you should know about the default base learners of XGBoost: **tree ensembles**. The tree ensemble model is a set of classification and regression trees (CART). Trees are grown one after another ,and attempts to reduce the misclassification rate are made in subsequent iterations. Here's a simple example of a CART that classifies whether someone will like computer games straight from the XGBoost's documentation.

If you check the image in Tree Ensemble section, you will notice each tree gives a different prediction score depending on the data it sees and the scores of each individual tree are summed up to get the final score.

13.1.1 Using XGBoost in Python [1-4,7]:

import the Boston Housing dataset and store it in a variable called boston.

```
from sklearn.datasets import load_boston

boston = load_boston()
```

The boston variable itself is a dictionary, so you can check for its keys using the .keys() method.

```
print(boston.keys())

dict_keys(['data', 'target', 'feature_names', 'DESCR'])
```

You can easily check for its shape by using the boston.data.shape attribute, which will return the size of the dataset.

```
print(boston.data.shape)

(506, 13)
```

As you can see it returned (506, 13), that means there are 506 rows of data with 13 columns. Now, if you want to know what the 13 columns are, you can simply use the .feature_names attribute and it will return the feature names.

```
print(boston.feature_names)

['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX'
'PTRATIO'

 'B' 'LSTAT']
```

The description of the dataset is available in the dataset itself. You can take a look at it using .DESCR.

print(boston.DESCR)

Boston House Prices dataset

==========

Notes:

Data Set Characteristics:

: Number of Instances: 506

: Number of Attributes: 13 numeric/categorical predictive

: Median Value (attribute 14) is usually the target

: Attribute Information (in order):

- CRIM per capita crime rate by town
- ZN proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS proportion of non-retail business acres per town
- CHAS Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
- NOX nitric oxides concentration (parts per 10 million)
- RM average number of rooms per dwelling
- AGE proportion of owner-occupied units built prior to 1940
- DIS weighted distances to five Boston employment centres
- RAD index of accessibility to radial highways
- TAX full-value property-tax rate per $10,000
- PTRATIO pupil-teacher ratio by town
- B 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
- LSTAT % lower status of the population
- MEDV Median value of owner-occupied homes in $1000's

: Missing Attribute Values: None

Now let’s convert it into a pandas DataFrame! For that you need to import the pandas library and call the DataFrame() function passing the argument

boston.data. To label the names of the columns, use the .columnns attribute of the pandas DataFrame and assign it to boston.feature_names.

import pandas as pd

data = pd.DataFrame(boston.data)

data.columns = boston.feature_names

Explore the top 5 rows of the dataset by using head() method on your pandas DataFrame.

data.head()

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33

You'll notice that there is no column called PRICE in the DataFrame. This is because the target column is available in another attribute called boston.target. Append boston.target to your pandas DataFrame.

data['PRICE'] = boston.target

Run the .info() method on your DataFrame to get useful information about the data.

data.info()

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 506 entries, 0 to 505

Data columns (total 14 columns):

CRIM 506 non-null float64

ZN 506 non-null float64

INDUS 506 non-null float64

CHAS 506 non-null float64

NOX 506 non-null float64

RM 506 non-null float64

AGE 506 non-null float64

DIS 506 non-null float64

RAD 506 non-null float64

TAX 506 non-null float64

PTRATIO 506 non-null float64

B 506 non-null float64

LSTAT 506 non-null float64

PRICE 506 non-null float64

dtypes: float64(14)

memory usage: 55.4 KB

Turns out that this dataset has 14 columns (including the target variable PRICE) and 506 rows. Notice that the columns are of float data-type indicating the presence of only continuous features with no missing values in any of the columns. To get more summary statistics of the different features in the dataset you will use the describe() method on your DataFrame.

Note that describe() only gives summary statistics of columns which are continuous in nature and not categorical.

data.describe()

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	PRICE
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.593761	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.237154	18.455534	356.674032	12.653063	22.532806
std	8.596783	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.537116	2.164946	91.294864	7.141062	9.197104
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000	0.320000	1.730000	5.000000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000	17.400000	375.377500	6.950000	17.025000
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.000000	19.050000	391.440000	11.360000	21.200000
75%	3.647423	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.000000	20.200000	396.225000	16.955000	25.000000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000	396.900000	37.970000	50.000000

If you plan to use XGBoost on a dataset which has categorical features you may want to consider applying some encoding (like one-hot encoding) to such features before training the model.

Without delving into more exploratory analysis and feature engineering, you will now focus on applying the algorithm to train the model on this data.

Install python libraries like xgboost on your system using pip install xgboost on cmd.

```
import xgboost as xgb
from sklearn.metrics import mean_squared_error
import pandas as pd
import numpy as np
Separate the target variable and rest of the variables using .iloc to subset the data.
X, y = data.iloc[:,:-1],data.iloc[:,-1]
data_dmatrix = xgb.DMatrix(data=X,label=y)
```

XGBoost's hyperparameters:

At this point, before building the model, you should be aware of the tuning parameters that XGBoost provides. Well, there are a plethora of tuning parameters for tree-based learners in XGBoost and you can read all about them here. But the most common ones that you should know are:

learning_rate: step size shrinkage used to prevent overfitting. Range is [0,1]

max_depth: determines how deeply each tree is allowed to grow during any boosting round.

subsample: percentage of samples used per tree. Low value can lead to underfitting.

colsample_bytree: percentage of features used per tree. High value can lead to overfitting.

n_estimators: number of trees you want to build.

objective: determines the loss function to be used like reg:linear for regression problems, reg:logistic for classification problems with only decision, binary:logistic for classification problems with probability.

XGBoost also supports regularization parameters to penalize models as they become more complex and reduce them to simple (parsimonious) models [1-4,7].

gamma: controls whether a given node will split based on the expected reduction in loss after the split. A higher value leads to fewer splits. Supported only for tree-based learners.

alpha: L1 regularization on leaf weights. A large value leads to more regularization.

lambda: L2 regularization on leaf weights and is smoother than L1 regularization.

It's also worth mentioning that though you are using trees as your base learners, you can also use XGBoost's relatively less popular linear base learners and one other tree learner known as dart. All you have to do is set the booster parameter to either gbtree (default),gblinear or dart.

Now, you will create the train and test set for cross-validation of the results using the train_test_split function from sklearn's model_selection module with test_size size equal to 20% of the data. Also, to maintain reproducibility of the results, a random_state is also assigned.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=123)
```

The next step is to instantiate an XGBoost regressor object by calling the XGBRegressor() class from the XGBoost library with the hyper-parameters passed as arguments. For classification problems, you would have used the XGBClassifier() class.

```
xg_reg = xgb.XGBRegressor(objective ='reg:linear', colsample_bytree = 0.3, learning_rate = 0.1,

            max_depth = 5, alpha = 10, n_estimators = 10)
```

Fit the regressor to the training set and make predictions on the test set using the familiar .fit() and .predict() methods.

```
xg_reg.fit(X_train,y_train)

preds = xg_reg.predict(X_test)
```

Compute the rmse by invoking the mean_squared_error function from sklearn's metrics module.

```
rmse = np.sqrt(mean_squared_error(y_test, preds))

print("RMSE: %f" % (rmse))

RMSE: 10.569356
```

Well, you can see that your RMSE for the price prediction came out to be around 10.8 per 1000$.

13.1.2 k-fold Cross Validation using XGBoost [1-4,7]:

In order to build more robust models, it is common to do a k-fold cross validation where all the entries in the original training dataset are used for both training as well as validation. Also, each entry is used for validation just once. XGBoost supports k-fold cross validation via the cv() method. All you have to do is specify the nfolds parameter, which is the number of cross validation sets you want to build. Also, it supports many other parameters (check out this link) like:

num_boost_round: denotes the number of trees you build (analogous to n_estimators)

metrics: tells the evaluation metrics to be watched during CV

as_pandas: to return the results in a pandas DataFrame.

early_stopping_rounds: finishes training of the model early if the hold-out metric ("rmse" in our case) does not improve for a given number of rounds.

seed: for reproducibility of results.

This time you will create a hyper-parameter dictionary params which holds all the hyper-parameters and their values as key-value pairs but will

exclude the n_estimators from the hyper-parameter dictionary because you will use num_boost_rounds instead.

You will use these parameters to build a 3-fold cross validation model by invoking XGBoost's cv() method and store the results in a cv_results DataFrame. Note that here you are using the Dmatrix object you created before.

params = {"objective":"reg:linear",'colsample_bytree': 0.3,'learning_rate': 0.1,

'max_depth': 5, 'alpha': 10}

cv_results = xgb.cv(dtrain=data_dmatrix, params=params, nfold=3,

num_boost_round=50,early_stopping_rounds=10,metrics="rmse", as_pandas=True, seed=123)

cv_results contains train and test RMSE metrics for each boosting round.

cv_results.head()

	test-rmse-mean	test-rmse-std	train-rmse-mean	train-rmse-std
0	21.746693	0.019311	21.749371	0.033853
1	19.891096	0.053295	19.859423	0.029633
2	18.168509	0.014465	18.072169	0.018803
3	16.687861	0.037342	16.570206	0.018556
4	15.365013	0.059400	15.206344	0.015451

Extract and print the final boosting round metric.

print((cv_results["test-rmse-mean"]).tail(1))

49 4.031162

Name: test-rmse-mean, dtype: float64

You can see that your RMSE for the price prediction has reduced as compared to last time and came out to be around 4.03 per 1000$. You can reach an even lower RMSE for a different set of hyper-parameters. You may consider applying techniques like Grid Search, Random Search and Bayesian Optimization to reach the optimal set of hyper-parameters.

Visualize Boosting Trees and Feature Importance [1-4,7]:

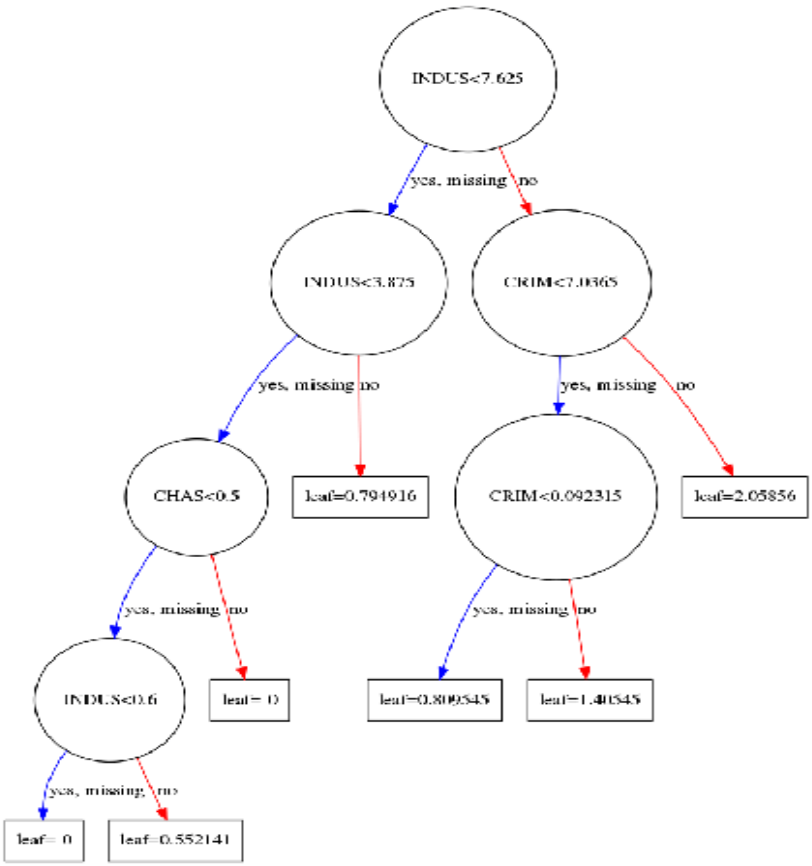
You can also visualize individual trees from the fully boosted model that XGBoost creates using the entire housing dataset. XGBoost has a plot_tree() function that makes this type of visualization easy. Once you train a model using the XGBoost learning API, you can pass it to the

plot_tree() function along with the number of trees you want to plot using the num_trees argument.

xg_reg = xgb.train(params=params, dtrain=data_dmatrix, num_boost_round=10)

Plotting the first tree with the matplotlib library:

```
import matplotlib.pyplot as plt
xgb.plot_tree(xg_reg,num_trees=0)
plt.rcParams['figure.figsize'] = [50, 10]
plt.show()
```

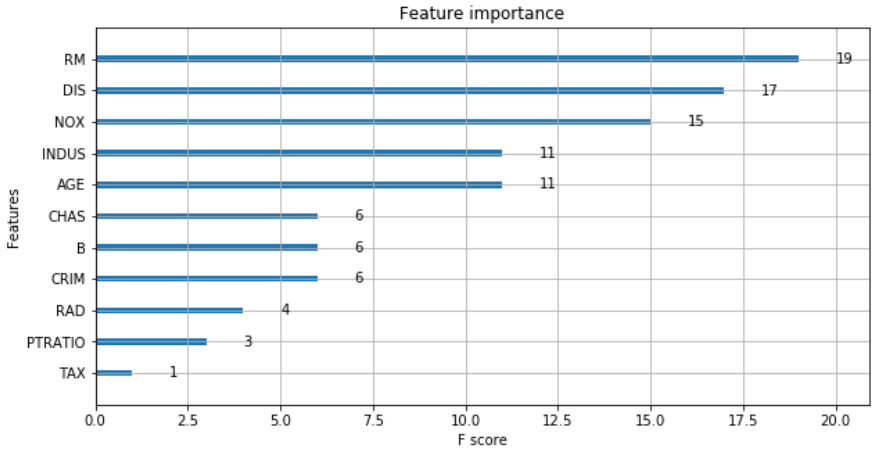


These plots provide insight into how the model arrived at its final decisions and what splits it made to arrive at those decisions.

Another way to visualize your XGBoost models is to examine the importance of each feature column in the original dataset within the model.

One simple way of doing this involves counting the number of times each feature is split on across all boosting rounds (trees) in the model, and then visualizing the result as a bar graph, with the features ordered according to how many times they appear. XGBoost has a plot_importance() function that allows you to do exactly this.

```
xgb.plot_importance(xg_reg)

plt.rcParams['figure.figsize'] = [5, 5]

plt.show()
```



As you can see the feature RM has been given the highest importance score among all the features.

Example 2:

XGBoost Regression API [1-4,7]

XGBoost can be installed as a standalone library and an XGBoost model can be developed using the scikit-learn API.

Install the XGBoost library.

sudo pip install xgboost

You can then confirm that the XGBoost library was installed correctly and can be used by running the following script.

check xgboost version

import xgboost

print(xgboost.__version__)

Running the script will print your version of the XGBoost library you have installed.

Your version should be the same or higher. If not, you must upgrade your version of the XGBoost library.

If you do have errors when trying to run the above script, I recommend downgrading to version 1.0.1 (or lower). This can be achieved by specifying the version to install to the pip command, as follows:

sudo pip install xgboost==1.0.1

If you require specific instructions for your development environment, see the tutorial:

13.1.3 XGBoost Installation Guide [1-4,7]:

The XGBoost library has its own custom API, although we will use the method via the scikit-learn wrapper classes: XGBRegressor and XGBClassifier. This will allow us to use the full suite of tools from the scikit-learn machine learning library to prepare data and evaluate models.

An XGBoost regression model can be defined by creating an instance of the XGBRegressor class; for example:

...

create an xgboost regression model

model = XGBRegressor()

You can specify hyperparameter values to the class constructor to configure the model.

Perhaps the most commonly configured hyperparameters are the following:

n_estimators: The number of trees in the ensemble, often increased until no further improvements are seen.

max_depth: The maximum depth of each tree, often values are between 1 and 10.

eta: The learning rate used to weight each model, often set to small values such as 0.3, 0.1, 0.01, or smaller.

subsample: The number of samples (rows) used in each tree, set to a value between 0 and 1, often 1.0 to use all samples.

colsample_bytree: Number of features (columns) used in each tree, set to a value between 0 and 1, often 1.0 to use all features.

For example:

...

create an xgboost regression model

model = XGBRegressor(n_estimators=1000, max_depth=7, eta=0.1, subsample=0.7, colsample_bytree=0.8)

Good hyperparameter values can be found by trial and error for a given dataset, or systematic experimentation such as using a grid search across a range of values.

Randomness is used in the construction of the model. This means that each time the algorithm is run on the same data, it may produce a slightly different model.

When using machine learning algorithms that have a stochastic learning algorithm, it is good practice to evaluate them by averaging their performance across multiple runs or repeats of cross-validation. When fitting a final model, it may be desirable to either increase the number of trees until the variance of the model is reduced across repeated evaluations, or to fit multiple final models and average their predictions.

Let's take a look at how to develop an XGBoost ensemble for regression.

XGBoost Regression Example [1-4,7]:

In this section, we will look at how we might develop an XGBoost model for a standard regression predictive modeling dataset.

First, let's introduce a standard regression dataset.

We will use the housing dataset.

The housing dataset is a standard machine learning dataset comprising 506 rows of data with 13 numerical input variables and a numerical target variable.

Using a test harness of repeated stratified 10-fold cross-validation with three repeats, a naive model can achieve a mean absolute error (MAE) of about 6.6. A top-performing model can achieve a MAE on this same test harness of about 1.9. This provides the bounds of expected performance on this dataset.

The dataset involves predicting the house price given details of the house's suburb in the American city of Boston.

Housing Dataset (housing.csv) [1-4,7]:

Housing Description (housing.names)

No need to download the dataset; we will download it automatically as part of our worked examples.

The example below downloads and loads the dataset as a Pandas DataFrame and summarizes the shape of the dataset and the first five rows of data.

```
# load and summarize the housing dataset

from pandas import read_csv

from matplotlib import pyplot

# load dataset
```

```
url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/housing.csv'

dataframe = read_csv(url, header=None)

# summarize shape

print(dataframe.shape)

# summarize first few lines

print(dataframe.head())
```

Running the example confirms the 506 rows of data and 13 input variables and a single numeric target variable (14 in total). We can also see that all input variables are numeric.

(506, 14)

	0	1	2	3	4	5	...	8	9	10	11	12	13
0	0.00632	18.0	2.31	0	0.538	6.575	...	1	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	...	2	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	...	2	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	...	3	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	...	3	222.0	18.7	396.90	5.33	36.2

[5 rows x 14 columns]

Next, let's evaluate a regression XGBoost model with default hyperparameters on the problem.

First, we can split the loaded dataset into input and output columns for training and evaluating a predictive model.

```
...

# split data into input and output columns

X, y = data[:, :-1], data[:, -1]
```

Next, we can create an instance of the model with a default configuration.

```
...

# define model

model = XGBRegressor()
```

We will evaluate the model using the best practice of repeated k-fold cross-validation with 3 repeats and 10 folds.

This can be achieved by using the RepeatedKFold class to configure the evaluation procedure and calling the cross_val_score() to evaluate the model using the procedure and collect the scores.

Model performance will be evaluated using mean squared error (MAE). Note, MAE is made negative in the scikit-learn library so that it can be

maximized. As such, we can ignore the sign and assume all errors are positive.

...

define model evaluation method

cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)

evaluate model

scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)

Once evaluated, we can report the estimated performance of the model when used to make predictions on new data for this problem.

In this case, because the scores were made negative, we can use the absolute() NumPy function to make the scores positive.

We then report a statistical summary of the performance using the mean and standard deviation of the distribution of scores, another good practice.

...

force scores to be positive

scores = absolute(scores)

print('Mean MAE: %.3f (%.3f)' % (scores.mean(), scores.std()))

Tying this together, the complete example of evaluating an XGBoost model on the housing regression predictive modeling problem is listed below.

```
# evaluate an xgboost regression model on the housing dataset

from numpy import absolute

from pandas import read_csv

from sklearn.model_selection import cross_val_score

from sklearn.model_selection import RepeatedKFold

from xgboost import XGBRegressor

# load the dataset

url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/housing.csv'

dataframe = read_csv(url, header=None)
```

```
# split data into input and output columns

X, y = data[:, :-1], data[:, -1]

# define model

model = XGBRegressor()

# define model evaluation method

cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)

# evaluate model
```

```
scores = cross_val_score(model, X, y,
scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)

# force scores to be positive

scores = absolute(scores)

print('Mean MAE: %.3f (%.3f)' % (scores.mean(), scores.std()) )
```

Running the example evaluates the XGBoost Regression algorithm on the housing dataset and reports the average MAE across the three repeats of 10-fold cross-validation.

In this case, we can see that the model achieved a MAE of about 2.1.

This is a good score, better than the baseline, meaning the model has skill and close to the best score of 1.9.

Mean MAE: 2.109 (0.320)

We may decide to use the XGBoost Regression model as our final model and make predictions on new data.

This can be achieved by fitting the model on all available data and calling the predict() function, passing in a new row of data.

For example:

```
...

# make a prediction
yhat = model.predict(new_data)
```

We can demonstrate this with a complete example, listed below.

```
# fit a final xgboost model on the housing dataset and make a prediction
from numpy import asarray
from pandas import read_csv
from xgboost import XGBRegressor
# load the dataset
```

```
url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/housing.csv'
dataframe = read_csv(url, header=None)
data = dataframe.values
# split dataset into input and output columns
X, y = data[:, :-1], data[:, -1]
# define model
model = XGBRegressor()
# fit model
model.fit(X, y)
# define new data
row = [0.00632,18.00,2.310,0,0.5380,6.5750,65.20,4.0900,1,296.0,15.30,396.90,
new_data = asarray([row])
# make a prediction
yhat = model.predict(new_data)
# summarize prediction
print('Predicted: %.3f' % yhat)
```

Running the example fits the model and makes a prediction for the new rows of d
Note: Your results may vary given the stochastic nature of the algorithm or evalu
or differences in numerical precision. Consider running the example a few tim
the average outcome.
In this case, we can see that the model predicted a value of about 24.

```
Predicted: 24.019
```

13.2 VOTING ENSEMBLES [1-4,7]

A voting ensemble (or a "majority voting ensemble") is an ensemble machine learning model that combines the predictions from multiple other models.

It is a technique that may be used to improve model performance, ideally achieving better performance than any single model used in the ensemble.

A voting ensemble works by combining the predictions from multiple models. It can be used for classification or regression. In the case of regression, this involves calculating the average of the predictions from the models. In the case of classification, the predictions for each label are summed and the label with the majority vote is predicted.

Regression Voting Ensemble: Predictions are the average of contributing models.

Classification Voting Ensemble: Predictions are the majority vote of contributing models.

There are two approaches to the majority vote prediction for classification; they are hard voting and soft voting.

Hard voting involves summing the predictions for each class label and predicting the class label with the most votes. Soft voting involves summing the predicted probabilities (or probability-like scores) for each class label and predicting the class label with the largest probability.

Hard Voting: Predict the class with the largest sum of votes from models

Soft Voting: Predict the class with the largest summed probability from models.

A voting ensemble may be considered a meta-model, a model of models.

As a meta-model, it could be used with any collection of existing trained machine learning models and the existing models do not need to be aware that they are being used in the ensemble. This means you could explore using a voting ensemble on any set or subset of fit models for your predictive modeling task.

A voting ensemble is appropriate when you have two or more models that perform well on a predictive modeling task. The models used in the ensemble must mostly agree with their predictions.

Use voting ensembles when:

- All models in the ensemble have generally the same good performance.
- All models in the ensemble mostly already agree.

Hard voting is appropriate when the models used in the voting ensemble predict crisp class labels. Soft voting is appropriate when the models used in the voting ensemble predict the probability of class membership. Soft voting can be used for models that do not natively predict a class membership probability, although may require calibration of their probability-like scores prior to being used in the ensemble (e.g. support vector machine, k-nearest neighbors, and decision trees).

Hard voting is for models that predict class labels.

Soft voting is for models that predict class membership probabilities.

The voting ensemble is not guaranteed to provide better performance than any single model used in the ensemble. If any given model used in the ensemble performs better than the voting ensemble, that model should probably be used instead of the voting ensemble.

This is not always the case. A voting ensemble can offer lower variance in the predictions made over individual models. This can be seen in a lower variance in prediction error for regression tasks. This can also be seen in a lower variance in accuracy for classification tasks. This lower variance may result in a lower mean performance of the ensemble, which might be desirable given the higher stability or confidence of the model.

Use a voting ensemble if:

- It results in better performance than any model used in the ensemble.
- It results in a lower variance than any model used in the ensemble.

A voting ensemble is particularly useful for machine learning models that use a stochastic learning algorithm and result in a different final model

each time it is trained on the same dataset. One example is neural networks that are fit using stochastic gradient descent.

Another particularly useful case for voting ensembles is when combining multiple fits of the same machine learning algorithm with slightly different hyperparameters.

Voting ensembles are most effective when:

- Combining multiple fits of a model trained using stochastic learning algorithms.
- Combining multiple fits of a model with different hyperparameters.

A limitation of the voting ensemble is that it treats all models the same, meaning all models contribute equally to the prediction. This is a problem if some models are good in some situations and poor in others.

An extension to the voting ensemble to address this problem is to use a weighted average or weighted voting of the contributing models. This is sometimes called blending. A further extension is to use a machine learning model to learn when and how much to trust each model when making predictions. This is referred to as stacked generalization, or stacking for short.

Extensions to voting ensembles:

- Weighted Average Ensemble (blending).
- Stacked Generalization (stacking).

Voting Ensemble Scikit-Learn API [1-4,7]:

Voting ensembles can be implemented from scratch, although it can be challenging for beginners.

The scikit-learn Python machine learning library provides an implementation of voting for machine learning.

It is available in version 0.22 of the library and higher.

First, confirm that you are using a modern version of the library by running the following script:

```
# check scikit-learn version

import sklearn

print(sklearn.__version__)
```

Running the script will print your version of scikit-learn.

Your version should be the same or higher. If not, you must upgrade your version of the scikit-learn library.

Voting is provided via the VotingRegressor and VotingClassifier classes.

Both models operate the same way and take the same arguments. Using the model requires that you specify a list of estimators that make predictions and are combined in the voting ensemble.

A list of base models is provided via the "estimators" argument. This is a Python list where each element in the list is a tuple with the name of the model and the configured model instance. Each model in the list must have a unique name.

For example, below defines two base models:

```
...
models = [('lr',LogisticRegression()),('svm',SVC())]
ensemble = VotingClassifier(estimators=models)
```

Each model in the list may also be a Pipeline, including any data preparation required by the model prior to fitting the model on the training dataset.

For example:

```
...
models = [('lr',LogisticRegression()),('svm',make_pipeline(StandardScaler(),SVC()))]
ensemble = VotingClassifier(estimators=models)
```

When using a voting ensemble for classification, the type of voting, such as hard voting or soft voting, can be specified via the "voting" argument and set to the string 'hard' (the default) or 'soft'.

For example:

```
...
models = [('lr',LogisticRegression()),('svm',SVC())]
ensemble = VotingClassifier(estimators=models, voting='soft')
```

Now that we are familiar with the voting ensemble API in scikit-learn, let's look at some worked examples.

13.2.1 Voting Ensemble for Classification [1-4,7]:

First, we can use the make_classification() function to create a synthetic binary classification problem with 1,000 examples and 20 input features.

The complete example is listed below.

```
# 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)
```

Running the example creates the dataset and summarizes the shape of the input and output components.

```
(1000, 20) (1000,)
```

Next, we will demonstrate hard voting and soft voting for this dataset.

13.2.2 Hard Voting Ensemble for Classification [1-4,7]:

We can demonstrate hard voting with a k-nearest neighbor algorithm.

We can fit five different versions of the KNN algorithm, each with a different number of neighbors used when making predictions. We will use 1, 3, 5, 7, and 9 neighbors (odd numbers in an attempt to avoid ties).

Our expectation is that by combining the predicted class labels predicted by each different KNN model that the hard voting ensemble will achieve a better predictive performance than any standalone model used in the ensemble, on average.

First, we can create a function named get_voting() that creates each KNN model and combines the models into a hard voting ensemble.

```
# 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
```

We can then create a list of models to evaluate, including each standalone version of the KNN model configurations and the hard voting ensemble.

This will help us directly compare each standalone configuration of the KNN model with the ensemble in terms of the distribution of classification accuracy scores. The get_models() function below creates the list of models for us to evaluate.

```
# 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
```

Each model will be evaluated using repeated k-fold cross-validation.

The evaluate_model() function below takes a model instance and returns as a list of scores from three repeats of stratified 10-fold cross-validation.

```
# 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
```

We can then report the mean performance of each algorithm, and also create a box and whisker plot to compare the distribution of accuracy scores for each algorithm.

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()
```

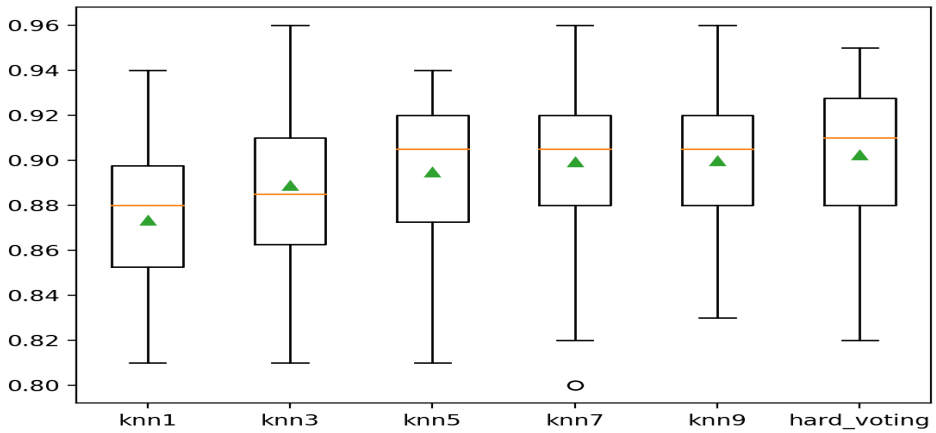
Running the example first reports the mean and standard deviation accuracy for each model.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

We can see the hard voting ensemble achieves a better classification accuracy of about 90.2% compared to all standalone versions of the model.

```
>knn1 0.873 (0.030)
>knn3 0.889 (0.038)
>knn5 0.895 (0.031)
>knn7 0.899 (0.035)
>knn9 0.900 (0.033)
>hard_voting 0.902 (0.034)
```

A box-and-whisker plot is then created comparing the distribution accuracy scores for each model, allowing us to clearly see that hard voting ensemble performing better than all standalone models on average.



First, the hard voting ensemble is fit on all available data, then the predict() function can be called to make predictions on new data.

The example below demonstrates this on our binary classification dataset [1-4,7]

```
# 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))
```

Running the example fits the hard voting ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

Predicted Class: 1

Soft Voting Ensemble for Classification

We can demonstrate soft voting with the support vector machine (SVM) algorithm.

The SVM algorithm does not natively predict probabilities, although it can be configured to predict probability-like scores by setting the “probability” argument to “True” in the SVC class.

We can fit five different versions of the SVM algorithm with a polynomial kernel, each with a different polynomial degree, set via the “degree” argument. We will use degrees 1-5.

Our expectation is that by combining the predicted class membership probability scores predicted by each different SVM model that the soft voting ensemble will achieve a better predictive performance than any standalone model used in the ensemble, on average.

First, we can create a function named get_voting() that creates the SVM models and combines them into a soft voting ensemble.

```
# 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
```

We can then create a list of models to evaluate, including each standalone version of the SVM model configurations and the soft voting ensemble.

This will help us directly compare each standalone configuration of the SVM model with the ensemble in terms of the distribution of classification accuracy scores. The get_models() function below creates the list of models for us to evaluate.

```
# 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

    return models
```

We can evaluate and report model performance using repeated k-fold cross-validation as we did in the previous section.

Tying this together, the complete example is listed below.

```
# 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()
```

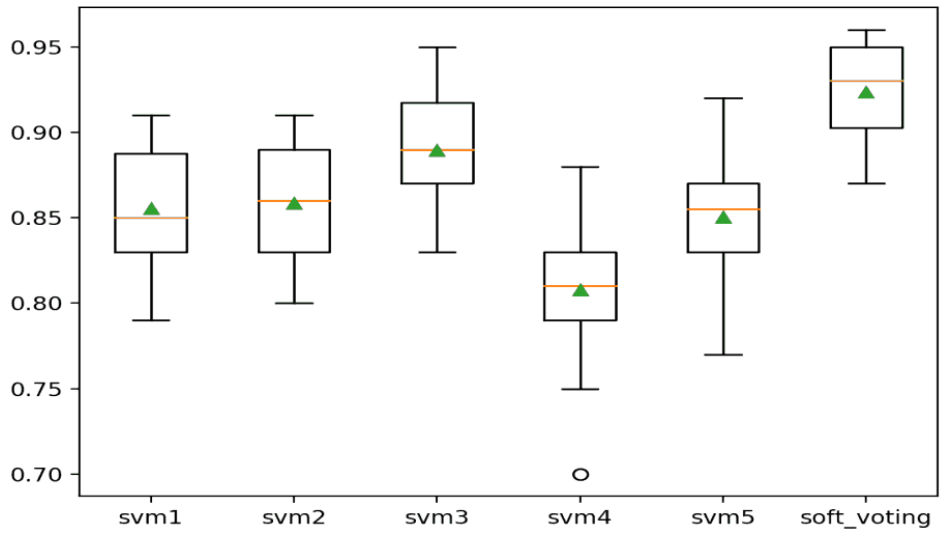
Running the example first reports the mean and standard deviation accuracy for each model.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

We can see the soft voting ensemble achieves a better classification accuracy of about 92.4% compared to all standalone versions of the model.

```
>svm1 0.855 (0.035)
>svm2 0.859 (0.034)
>svm3 0.890 (0.035)
>svm4 0.808 (0.037)
>svm5 0.850 (0.037)
>soft_voting 0.924 (0.028)
```

A box-and-whisker plot is then created comparing the distribution accuracy scores for each model, allowing us to clearly see that soft voting ensemble performing better than all standalone models on average.



If we choose a soft voting ensemble as our final model, we can fit and use it to make predictions on new data just like any other model.

First, the soft voting ensemble is fit on all available data, then the predict() function can be called to make predictions on new data.

The example below demonstrates this on our binary classification dataset.

```
# 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))
```

Running the example fits the soft voting ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

Predicted Class: 1

Voting Ensemble for Regression

We will look at using voting for a regression problem.

First, we can use the make_regression() function to create a synthetic regression problem with 1,000 examples and 20 input features.

The complete example is listed below.

```
# 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)
```

Running the example creates the dataset and summarizes the shape of the input and output components.

(1000, 20) (1000,)

We can demonstrate ensemble voting for regression with a decision tree algorithm, sometimes referred to as a classification and regression tree (CART) algorithm.

We can fit five different versions of the CART algorithm, each with a different maximum depth of the decision tree, set via the "max_depth" argument. We will use depths of 1-5.

Our expectation is that by combining the values predicted by each different CART model that the voting ensemble will achieve a better predictive performance than any standalone model used in the ensemble, on average.

First, we can create a function named get_voting() that creates each CART model and combines the models into a voting ensemble.

```
# 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
```

We can then create a list of models to evaluate, including each standalone version of the CART model configurations and the soft voting ensemble.

This will help us directly compare each standalone configuration of the CART model with the ensemble in terms of the distribution of error scores. The get_models() function below creates the list of models for us to evaluate.

```
# 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
```

We can evaluate and report model performance using repeated k-fold cross-validation as we did in the previous section.

Models are evaluated using mean absolute error (MAE). The scikit-learn makes the score negative so that it can be maximized. This means that the reported MAE scores are negative, larger values are better, and 0 represents no error.

Tying this together, the complete example is listed below.

```
# 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()
```

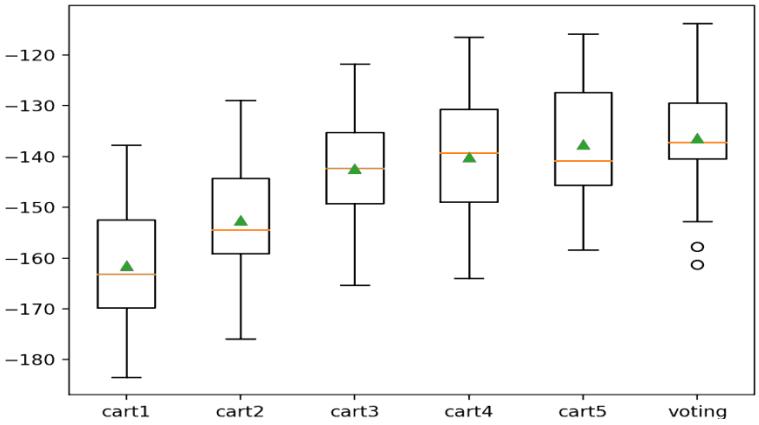
Running the example first reports the mean and standard deviation accuracy for each model.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

We can see the voting ensemble achieves a better mean squared error of about -136.338, which is larger (better) compared to all standalone versions of the model.

```
>cart1 -161.519 (11.414)

>cart2 -152.596 (11.271)

>cart3 -142.378 (10.900)

>cart4 -140.086 (12.469)

>cart5 -137.641 (12.240)

>voting -136.338 (11.242)
```

A box-and-whisker plot is then created comparing the distribution negative MAE scores for each model, allowing us to clearly see that voting ensemble performing better than all standalone models on average.



If we choose a voting ensemble as our final model, we can fit and use it to make predictions on new data just like any other model.

First, the voting ensemble is fit on all available data, then the predict() function can be called to make predictions on new data.

The example below demonstrates this on our binary classification dataset.

```
# 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))
```

Running the example fits the voting ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

Predicted Value: 141.319

*****

14

DEPLOYMENT OF MACHINE LEARNING ALGORITHMS

Unit Structure

14.1 DEPLOY YOUR MACHINE LEARNING MODELS [12]

Machine learning deployment is the process of deploying a machine learning model in a live environment. The model can be deployed across a range of different environments and will often be integrated with apps through an API. Deployment is a key step in an organisation gaining operational value from machine learning.

Machine learning models will usually be developed in an offline or local environment, so will need to be deployed to be used with live data. A data scientist may create many different models, some of which never make it to the deployment stage. Developing these models can be very resource intensive. Deployment is the final step for an organisation to start generating a return on investment for the organisation.

However, deployment from a local environment to a real-world application can be complex. Models may need specific infrastructure and will need to be closely monitored to ensure ongoing effectiveness. For this reason, machine learning deployment must be properly managed so it's efficient and streamlined.

This guide explores the basic steps required for machine learning deployment in a containerised environment, the challenges organisations may face, and the tools available to streamline the process.

14.1.0 How to deploy machine learning models [12]:

Machine learning deployment can be a complex task and will differ depending on the system environment and type of machine learning model. Each organisation will likely have existing DevOps processes that may need to be adapted for machine learning deployment. However, the general deployment process for machine learning models deployed to a containerised environment will consist of four broad steps.

- The four steps to machine learning deployment include:
- Develop and create a model in a training environment.
- Test and clean the code ready for deployment.
- Prepare for container deployment.
- Plan for continuous monitoring and maintenance after machine learning deployment.

Create the machine learning model in a training environment

Data scientists will often create and develop many different machine learning models, of which only a few will make it into the deployment phase. Models will usually be built in a local or offline environment, fed by training data. There are different types of machine learning processes for developing different models. These will differ depending on the task the algorithm is being trained to complete. Examples include supervised machine learning in which a model is trained on labelled datasets or unsupervised machine learning where the algorithm identifies patterns and trends in data.

Organisations may use machine learning models for a range of reasons. Examples include streamlining monotonous administrative tasks, fine-tuning marketing campaigns, driving system efficiency, or completing the initial stages of research and development. A popular use is the categorisation and segmentation of raw data into defined groups. Once the model is trained and performing to a given accuracy on training data, it is ready to be prepared for deployment.

14.1.1 Test and clean code ready for deployment [12]:

The next step is to check if the code is of sufficient quality to be deployed. This is to ensure the model functions in a new live environment, but also so other members of the organisation can understand the model's creation process. The model is likely to have been developed in an offline environment by a data scientist. So, for deployment in a live setting the code will need to be scrutinised and streamline where possible.

Accurately explaining the results of a model is a key part of the machine learning oversight process. Clarity around development is needed for the results and predictions to be accepted in a business setting. For this reason, a clear explanatory document or 'read me' file should be produced.

There are three simple steps to prepare for deployment at this stage:

- Create a 'read me' file to explain the model in detail ready for deployment by the development team.
- Clean and scrutinise the code and functions and ensure clear naming conventions using a style guide.
- Test the code to check if the model functions as expected.

14.1.2 Prepare the model for container deployment [12]:

Containerisation is a powerful tool in machine learning deployment. Containers are the perfect environment for machine learning deployment and can be described as a kind of operating system visualisation. It's a popular environment for machine learning deployment and development because containers make scaling easy. Containerised code also makes updating or deploying distinct areas of the model straightforward. This lowers the risk of downtime for the whole model and makes maintenance more efficient.

The containers contain all elements needed for the machine learning code to function, ensuring a consistent environment. Numerous containers will often make up machine learning model architecture. Yet, as each container is deployed in isolation from the wider operating system and infrastructure, it can draw resources from a range of settings including local and cloud systems. Container orchestration platforms like Kubernetes help with the automation of container management such as monitoring, scheduling, and scaling.

14.1.3 Beyond machine learning deployment [12]:

Successful machine learning deployment is more than just ensuring the model is initially functioning in a live setting. Ongoing governance is needed to ensure the model is on track and working effectively and efficiently. Beyond the development of machine learning models, establishing the processes to monitor and deploy the model can be a challenge. However, it's a vital part of the ongoing success of machine learning deployment, and models can be kept optimised to avoid data drift or outliers.

Once the processes are planned and in place to monitor the machine learning model, data drift and emerging inefficiencies can be detected and resolved. Some models can also be regularly retrained with new data to avoid the model drifting too far from the live data. Considering the model

after deployment means machine learning will be effective in an organisation for the long term.

14.1.4 Challenges for machine learning deployment [12]:

The training and development of machine learning models is usually resource-intensive and will often be the focus of an organisation. The process of machine learning deployment is also a complex task and requires a high degree of planning to be effective. Taking a model developed in an offline environment and deploying it in a live environment will always bring unique risks and challenges. A major challenge is bridging the gap between data scientists who developed the model and the developers that will deploy the model. Skillsets and expertise may not overlap in these distinct areas, so efficient workflow management is vital.

Machine learning deployment can be a challenge for many organisations, especially if infrastructure must be built for deployment. Considerations around scaling the model to meet capacity add another layer of complexity. The effectiveness of the model itself is also a key challenge. Ensuring results are accurate with no bias can be difficult. After machine learning deployment, the model should be continuously tested and monitored to drive improvements and continuous optimisation.

The main challenges for machine learning deployment include [12]:

- A lack of communication between the development team and data scientists causing inefficiencies in the deployment process.
- Ensuring the right infrastructure and environment is in place for machine learning deployment.
- The ongoing monitoring of model accuracy and efficiency in a real-world setting can be difficult but is vital to achieving optimisation.
- Scaling machine learning models from training environment to real-world data, especially when capacity needs to be elastic.
- Explaining predictions and results from a model so that the algorithm is trusted within the organisation.
- Products for streamlining machine learning deployment

Planning and executing machine learning deployment can often be a complex task. Models need to be managed and monitored to ensure ongoing functionality, and initial deployment must be expertly planned for peak efficiency. Products like Seldon Deploy provide all the elements for a successful machine learning deployment, as well as the insight tools needed for ongoing maintenance.

The platform is language-agnostic, so it is prepared for any model developed by a development team. It can easily integrate deployed machine learning models with other apps through API connections. It’s a

platform for collaboration between data scientists and the development team, helping to simplify the deployment process.

Seldon Deploy features for machine learning deployment include [12]:

- Workflow management tools to test and deploy models and make planning more straightforward.
- Integration with Seldon Core, a platform for containerised machine learning deployment using Kubernetes. It converts machine learning models in a range of languages ready for containerised deployment.
- Accessible analytics dashboards to monitor and visualise the ongoing health of the model including monitoring data drift and detecting anomalies
- Innate scalability to help organisations expand to meet varying levels of capacity, avoiding the risk of downtime.
- The ability to be installed across different local or cloud systems to fit the organisation's current system architecture.

14.2 WAYS TO DEPLOY MACHINE LEARNING MODELS IN PRODUCTION

Deploy ML models and make them available to users or other components of your project[12]

Deploying machine learning models as web services [12]:

The simplest way to deploy a machine learning model is to create a web service for prediction. In this example, we use the Flask web framework to wrap a simple random forest classifier built with scikit-learn.

14.2.1 To create a machine learning web service, you need at least three steps [12]:

The first step is to create a machine learning model, train it and validate its performance. The following script will train a random forest classifier.

Model testing and validation are not included here to keep it simple. But do remember those are an integral part of any machine learning project.

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
df = pd.read_csv('titanic.csv')
x = df[df.columns.difference(['Survived'])
y = df['Survived']
classifier = RandomForestClassifier()
classifier.fit(x, y)
```

In the next step, we need to persist the model. The environment where we deploy the application is often different from where we train them. Training usually requires a different set of resources. Thus this separation helps organizations optimize their budget and efforts.

Scikit-learn offers python specific serialization that makes model persistence and restoration effortless. The following is an example of how we can store the trained model in a pickle file.

from sklearn.externals import joblib

joblib.dump(classifier, 'classifier.pkl')

Finally, we can serve the persisted model using a web framework. The following code creates a REST API using Flask. This file is hosted in a different environment, often in a cloud server.

```
from flask import Flask

app = Flask(__name__)

@app.route('/predict', methods=['POST'])

def predict():

    json_ = request.json

    query_df = pd.DataFrame(json_)

    query = pd.get_dummies(query_df)

        classifier = joblib.load('classifier.pkl')

    prediction = classifier.predict(query)

    return jsonify({'prediction': list(prediction)})

if __name__ == '__main__':

    app.run(port=8080)
```

The above code takes input in a POST request through https://localhost:8080/predict and returns the prediction in a JSON response.

14.2.2 Deploying machine learning models for batch prediction [12]:

While online models can serve prediction, on-demand batch predictions are sometimes preferable.

Offline models can be optimized to handle a high volume of job instances and run more complex models. In batch production mode, you don't need to worry about scaling or managing servers either.

Batch prediction can be as simple as calling the predict function with a data set of input variables. The following command does it.

prediction = classifier.predict(UNSEEN_DATASET)

Sometimes you will have to schedule the training or prediction in the batch processing method. There are several ways to do this. My favorite is to use either Airflow or Prefect to automate the task.

```
import requests
from datetime import timedelta, datetime
import pandas as pd
from prefect import task, Flow
from prefect.schedules import IntervalSchedule
@task(max_retries=3, retry_delay=timedelta(5))
def predict(input_data_path:str):
    """
    This task load the saved model, input data and returns prediction.
    If failed this task will retry 3 times at 5 min interval and fail permenantly.
    """
        classifier = joblib.load('classifier.pkl')
        df = pd.read_csv(input_data_path)
        prediction = classifier.predict(df)
        return jsonify({'prediction': list(prediction)})
@task(max_retries=3, retry_delay=timedelta(5))
def save_prediction(data, output_data_path:str):
    """
    This task will save the prediction to an output file.
    If failed, this task will retry for 3 times and fail permenantly.
    """
```

```
    with open(output_data_path, 'w') as f:
      f.write(data)


# Create a schedule object.
# This object starts 5 seconds from the time of script execution and repeat once a week.
schedule = IntervalSchedule(
    start_date=datetime.utcnow() + timedelta(seconds=5),
    interval=timedelta(weeks=1),
)
# Attach the schedule object and orchastrate the workflow.
with Flow("predictions", schedule=schedule) as flow:
    prediction = predict("./input_data.csv")
    save_prediction(prediction. "./output_data.csv")
```

The above script schedules prediction on a weekly basis starting from 5 seconds after the script execution. Prefect will retry the tasks 3 times if they fail.

However, building the model may require multiple stages in the batch processing framework. You need to decide what features are required and how you should construct the model for each stage.

Train the model on a high-performance computing system with an appropriate batch-processing framework.

Usually, you partition the training data into segments that are processed sequentially, one after the other. You can do this by splitting the dataset using a sampling scheme (e.g., balanced sampling, stratified sampling) or via some online algorithm (e.g., map-reduce).

The partitions can be distributed to multiple machines, but they must all load the same set of features. Feature scaling is recommended. If you used unsupervised pre-training (e.g., autoencoders) for transfer learning, you must undo each partition.

After all the stages have been executed, you can predict unseen data with the resulting model by iterating sequentially over the partitions.

14.2.3 Deploying machine learning models on edge devices as embedded models [12]:

Computing on edge devices such as mobile and IoT has become very popular in recent years. The benefits of deploying a machine learning model on edge devices include, but are not limited to:

Reduced latency as the device is likely to be close to the user than a server far away.

Reduce data bandwidth consumption as we ship processed results back to the cloud instead of raw data that requires big size and eventually more bandwidth.

Edge devices such as mobile and IoT devices have limited computation power and storage capacity due to the nature of their hardware. We cannot simply deploy machine learning models to these devices directly, especially if our model is big or requires extensive computation to run inference on them.

Instead, we should simplify the model using techniques such as quantization and aggregation while maintaining accuracy. These simplified models can be deployed efficiently on edge devices with limited computation, memory, and storage.

We can use the TensorFlow Lite library on Android to simplify our TensorFlow model. TensorFlow Lite is an open-source software library for mobile and embedded devices that tries to do what the name says: run TensorFlow models in Mobile and Embedded platforms.

The following example converts a Keras TensorFlow model.

```
import tensorflow as tf

# create and train a keras neural network

classifier = tf.keras.models.Sequential([

   tf.keras.layers.Dense(units=1, input_shape=[1]),

   tf.keras.layers.Dense(units=28, activation='relu'),

   tf.keras.layers.Dense(units=1)

])

classifier.compile(optimizer='sgd', loss='mean_squared_error')

classifier.fit(x=[-1, 0, 1], y=[-3, -1, 1], epochs=5)

# Convert the model to a Tensorflow Lite object

converter = tf.lite.TFLiteConverter.from_keras_model(classifier)

tfl_classifier = converter.convert()

# Save the model as a .tflite file

with open('classifier.tflite', 'wb') as f:

  f.write(tfl_classifier)
```

REFERENCES

1. Quick Introduction to Boosting Algorithms in Machine Learning. https://www.analyticsvidhya.com/blog/2015/11/quick-introduction-boosting-algorithms-machine-learning/. [Last Accessed on 10.03.2022]

2. Boosting Algorithms Explained. https://towardsdatascience.com/boosting-algorithms-explained-d38f56ef3f30[Last Accessed on 10.03.2022]

3. A Comprehensive Guide To Boosting Machine Learning Algorithms. https://www.edureka.co/blog/boosting-machine-learning/[Last Accessed on 10.03.2022]

4. Essence of Boosting Ensembles for Machine Learning. https://machinelearningmastery.com/essence-of-boosting-ensembles-for-machine-learning/[Last Accessed on 10.03.2022]

5. Boosting in Machine Learning | Boosting and AdaBoost. https://www.geeksforgeeks.org/bagging-vs-boosting-in-machine-learning/[Last Accessed on 10.03.2022]

6. https://www.datacamp.com/. [Last Accessed on 10.03.2022]

7. Machine Learning Plus Platform . https://www.machinelearningplus.com/. [Last Accessed on 10.03.2022]

8. Weights & Biases with Gradient. https://blog.paperspace.com/. [Last Accessed on 10.03.2022]

9. Build a machine Learning Web App in 5 Minutes. https://www.aiproblog.com/. [Last Accessed on 10.03.2022]

10. AdaBoost Algorithm. https://www.educba.com/adaboost-algorithm/. [Last Accessed on 10.03.2022]

11. Implementing the AdaBoost Algorithm From Scratch. https://www.geeksforgeeks.org/implementing-the-adaboost-algorithm-from-scratch/?ref=gcse. [Last Accessed on 10.03.2022]

12. Optimisation algorithms for differentiable functions. https://www.seldon.io/algorithm-optimisation-for-machine-learning. [Last Accessed on 10.03.2022]

13. Quiz – Machine Learning. https://mcqmate.com/. [Last Accessed on 10.03.2022]

TUTORIALS

1. How to Develop a Weighted Average Ensemble for Deep Learning Neural Networks : https://machinelearningmastery.com/weighted-average-ensemble-for-deep-learning-neural-networks/ [Last Accessed on 10.03.2022]

2. How to Develop a Stacking Ensemble for Deep Learning Neural Networks in Python With Keras : https://machinelearningmastery.com/stacking-ensemble-for-deep-learning-neural-networks/ [Last Accessed on 10.03.2022]

BOOKS

1. Schapire RE, Freund Y. Boosting: Foundations and algorithms. Kybernetes. 2013 Jan 4.
2. Zhou ZH. Ensemble methods: foundations and algorithms. CRC press; 2012 Jun 6.
3. Mohri M, Rostamizadeh A, Talwalkar A. Foundations of machine learning. MIT press; 2018 Dec 25.
4. Zhou ZH. Ensemble methods: foundations and algorithms. CRC press; 2012 Jun 6.
5. Data Mining: Practical Machine Learning Tools and Techniques, 2016.

MOOCS

Machine Learning: Classification. https://www.coursera.org/lecture/ml-classification/boosting-rV0iX

Advanced Machine Learning and Signal Processing. https://www.coursera.org/lecture/advanced-machine-learning-signal-processing/boosting-and-gradient-boosted-trees-8MEjw?redirectTo=%2Flearn%2Fadvanced-machine-learning-signal-processing%3Faction%3Denroll

Boosting Machine Learning Models in Python. https://www.udemy.com/course/boosting-machine-learning-models-in-python/

Boosting Algorithm in Python. https://python-course.eu/machine-learning/boosting-algorithm-in-python.php

Gradient Boosting Algorithm. https://www.educba.com/gradient-boosting-algorithm/.

Learning: Boosting. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/lecture-videos/lecture-17-learning-boosting/.

Bagging and Boosting. https://www.mygreatlearning.com/academy/learn-for-free/courses/bagging-and-boosting.

APIS

1. Ensemble methods scikit-learn API.
2. sklearn.ensemble.VotingClassifier API.
3. sklearn.ensemble.VotingRegressor API.

VIDEO LECTURES

1. Boosting Machine Learning Tutorial | Adaptive Boosting, Gradient Boosting, XGBoost | Edureka. https://www.youtube.com/watch?v=kho6oANGu_A [Last Accessed on 10.03.2022]

2. A Quick Guide to Boosting in Machine Learning. https://www.youtube.com/watch?v=sfVms30Ulxw

3. Introduction To Gradient Boosting algorithm (simplistic n graphical) - Machine Learning. https://www.youtube.com/watch?v=ErDgauqnTHk

4. Gradient Boosting In Depth Intuition- Part 1 Machine Learning. https://www.youtube.com/watch?v=Nol1hVtLOSg

5. Gradient Boosting - Math Clearly Explained Step By Step | Machine Learning Step By Step. https://www.youtube.com/watch?v=sANhnV-2Mo8

6. Visual Guide to Gradient Boosted Trees (xgboost). https://www.youtube.com/watch?v=TyvYZ26alZs

7. Xgboost Classification Indepth Maths Intuition- Machine Learning Algorithms. https://www.youtube.com/watch?v=gPciUPwWJQQ

8. Trevor Hastie - Gradient Boosting Machine Learning. https://www.youtube.com/watch?v=wPqtzj5VZus

9. Machine Learning Lecture 32 "Boosting" -Cornell CS4780 SP17. https://www.youtube.com/watch?v=dosOtgSdbnY

QUIZ

1. Ensemble learning can only be applied to supervised learning methods.

   A. True

   B. False

2. Ensembles will yield bad results when there is significant diversity among the models.

   Note: All individual models have meaningful and good predictions.

   A. true

   B. false

3. Which of the following is / are true about weak learners used in ensemble model?

   1. They have low variance and they don't usually overfit

   2. They have high bias, so they can not solve hard learning problems

   3. They have high variance and they don't usually overfit

   A. 1 and 2

   B. 1 and 3

   C. 2 and 3

   D. none of these

4. Ensemble of classifiers may or may not be more accurate than any of its individual model.

   A. true

   B. false

5. If you use an ensemble of different base models, is it necessary to tune the hyper parameters of all base models to improve the ensemble performance?

   A. yes

   B. no

   C. can't say

6. Generally, an ensemble method works better, if the individual base models have _____?

Note: Suppose each individual base models have accuracy greater than 50%.

A. less correlation among predictions

B. high correlation among predictions

C. correlation does not have any impact on ensemble output

D. none of the above

7. In an election, N candidates are competing against each other and people are voting for either of the candidates. Voters don't communicate with each other while casting their votes. Which of the following ensemble method works similar to above-discussed election procedure?

   Hint: Persons are like base models of ensemble method.

   A. bagging

   B. boosting

   C. a or b

   D. none of these

8. Suppose there are 25 base classifiers. Each classifier has error rates of e = 0.35.

   Suppose you are using averaging as ensemble technique. What will be the probabilities that ensemble of above 25 classifiers will make a wrong prediction?

   Note: All classifiers are independent of each other

   A. 0.05

   B. 0.06

   C. 0.07

   D. 0.09

9. In machine learning, an algorithm (or learning algorithm) is said to be unstable if a small change in training data cause the large change in the learned classifiers.True or False: Bagging of unstable classifiers is a good idea

   A. true

   B. false

10. Which of the following parameters can be tuned for finding good ensemble model in bagging based algorithms?

1. Max number of samples

2. Max features

3. Bootstrapping of samples

4. Bootstrapping of features

A. 1 and 3

B. 2 and 3

C. 1 and 2

D. all of above

11. How is the model capacity affected with dropout rate (where model capacity means the ability of a neural network to approximate complex functions)?

A. model capacity increases in increase in dropout rate

B. model capacity decreases in increase in dropout rate

C. model capacity is not affected on increase in dropout rate

D. none of these

12. Dropout is computationally expensive technique w.r.t. bagging

A. true

B. false

13. Suppose, you want to apply a stepwise forward selection method for choosing the best models for an ensemble model. Which of the following is the correct order of the steps?

Note: You have more than 1000 models predictions

1. Add the models predictions (or in another term take the average) one by one in the ensemble which improves the metrics in the validation set.

2. Start with empty ensemble

3. Return the ensemble from the nested set of ensembles that has maximum performance on the validation set

A. 1-2-3

B. 1-3-4

C. 2-1-3

D. none of above

14. Suppose, you have 2000 different models with their predictions and want to ensemble predictions of best x models. Now, which of the following can be a possible method to select the best x models for an ensemble?

    A. step wise forward selection

    B. step wise backward elimination

    C. both

    D. none of above

15. Below are the two ensemble models:

    1. E1(M1, M2, M3) and

    2. E2(M4, M5, M6)

    Above, Mx is the individual base models.

    Which of the following are more likely to choose if following conditions for E1 and E2 are given?

E1: Individual Models accuracies are high but models are of the same type or in another term less diverse

E2: Individual Models accuracies are high but they are of different types in another term high diverse in nature

    A. e1

    B. e2

    C. any of e1 and e2

    D. none of these

16. In boosting, individual base learners can be parallel.

    A. true

    B. false

17. Which of the following is true about bagging?

    1. Bagging can be parallel

    2. The aim of bagging is to reduce bias not variance

    3. Bagging helps in reducing overfitting

    A. 1 and 2

    B. 2 and 3

    C. 1 and 3

D. all of these

18. Suppose you are using stacking with n different machine learning algorithms with k folds on data.

    Which of the following is true about one level (m base models + 1 stacker) stacking?

    Note: Here, we are working on binary classification problem

    All base models are trained on all features

    You are using k folds for base models

    A. you will have only k features after the first stage

    B. you will have only m features after the first stage

    C. you will have k+m features after the first stage

    D. you will have k*n features after the first stage

19. Which of the following is the difference between stacking and blending?

    A. stacking has less stable cv compared to blending

    B. in blending, you create out of fold prediction

    C. stacking is simpler than blending

    D. none of these

20. Which of the following can be one of the steps in stacking?

    1. Divide the training data into k folds

    2. Train k models on each k-1 folds and get the out of fold predictions for remaining one fold

    3. Divide the test data set in "k" folds and get individual fold predictions by different algorithms

    A. 1 and 2

    B. 2 and 3

    C. 1 and 3

    D. all of above

21. Which of the following are advantages of stacking?

    1) More robust model

    2) better prediction

3) Lower time of execution

A. 1 and 2

B. 2 and 3

C. 1 and 3

D. all of the above

22. Which of the following are correct statement(s) about stacking?

A machine learning model is trained on predictions of multiple machine learning models

A Logistic regression will definitely work better in the second stage as compared to other classification methods

First stage models are trained on full / partial feature space of training data

A. 1 and 2

B. 2 and 3

C. 1 and 3

D. all of above

23. Which of the following is true about weighted majority votes?

1. We want to give higher weights to better performing models

2. Inferior models can overrule the best model if collective weighted votes for inferior models is higher than best model

3. Voting is special case of weighted voting

A. 1 and 3

B. 2 and 3

C. 1 and 2

D. 1, 2 and 3

24. Which of the following is true about averaging ensemble?

A. it can only be used in classification problem

B. it can only be used in regression problem

C. it can be used in both classification as well as regression

D. none of these

25. How can we assign the weights to output of different models in an ensemble?

1. Use an algorithm to return the optimal weights

2. Choose the weights using cross validation

3. Give high weights to more accurate models

A. 1 and 2

B. 1 and 3

C. 2 and 3

D. all of above

*****