ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

(CS 3491)

For B.E. IV Semester CSE and B.Tech., IT Branches

As per the Latest Syllabus of
Anna University, Chennai
(REGULATIONS - 2021)

with 12 Practical Exercises

Dr. S.N. SANGEETHA
S. JOTHIMANI

SUCHITRA PUBLICATIONS
(A GROUP OF LAKSHMI PUBLICATIONS)

SYLLABUS

ANNA UNIVERSITY, CHENNAI

For B.E., CSE and B.Tech., IT Branches

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

UNIT I: PROBLEM SOLVING 9

Introduction to AI - AI Applications - Problem solving agents - search algorithms - uninformed search strategies - Heuristic search strategies - Local search and optimization problems - adversarial search - constraint satisfaction problems (CSP).

UNIT II: PROBABILISTIC REASONING 9

Acting under uncertainty - Bayesian inference - naïve bayes models. Probabilistic reasoning - Bayesian networks - exact inference in BN - approximate inference in BN - causal networks.

UNIT III: SUPERVISED LEARNING 9

Introduction to machine learning - Linear Regression Models: Least squares, single & multiple variables, Bayesian linear regression, gradient descent, Linear Classification Models: Discriminant function - Probabilistic discriminative model - Logistic regression, Probabilistic generative model - Naive Bayes, Maximum margin classifier - Support vector machine, Decision Tree, Random forests.

UNIT IV: ENSEMBLE TECHNIQUES AND UNSUPERVISED LEARNING 9

Combining multiple learners: Model combination schemes, Voting, Ensemble Learning - bagging, boosting, stacking, Unsupervised learning: K-means, Instance Based Learning: KNN, Gaussian mixture models and Expectation maximization.

UNIT V: NEURAL NETWORKS 9

Perceptron - Multilayer perceptron, activation functions, network training - gradient descent optimization - stochastic gradient descent, error backpropagation, from shallow networks to deep networks - Unit saturation (aka the vanishing gradient problem) - ReLU, hyperparameter tuning, batch normalization, regularization, dropout.

PRACTICAL EXERCISES

1. Implementation of Uninformed Search Algorithms (BFS, DFS)
2. Implementation of Informed Search Algorithms (A*, memory-bounded A*)
3. Implement naïve Bayes Models
4. Implement Bayesian Networks
5. Build Regression Models
6. Build Decision Trees and Random Forests
7. Build SVM Models
8. Implement Ensembling Techniques
9. Implement Clustering Algorithms
10. Implement EM for Bayesian Networks
11. Build Simple NN Models
12. Build Deep Learning NN Models

References:

1. Stuart Russell and Peter Norvig, "Artificial Intelligence – A Modern Approach", Fourth Edition, Pearson Education, 2021.
2. Ethem Alpaydin, "Introduction to Machine Learning", MIT Press, Fourth Edition, 2020.

E-Resources:

https://www.simplilearn.com/
https://towardsdatascience.com/
https://www.kdnuggets.com/
https://www.javatpoint.com/
https://www.edureka.co/

CONTENTS

UNIT I

UNIT II

UNIT III

UNIT IV

UNIT V

NEURAL NETWORKS 5.1 - 5.104

UNIT I

PROBLEM SOLVING

Introduction to AI - AI Applications - Problem solving agents – search algorithms – uninformed search strategies – Heuristic search strategies – Local search and optimization problems – adversarial search – constraint satisfaction problems (CSP).

1.1. INTRODUCTION

Artificial intelligence is a branch of computer science that aims to create intelligent machines. It has become an essential part of the technology industry.

Research associated with artificial intelligence is highly technical and specialized. The core problems of artificial intelligence include programming computers for certain traits such as:

- Knowledge
- Reasoning
- Problem-solving
- Perception
- Learning
- Planning
- Ability to manipulate and move objects

❖ Artificial Intelligence is concerned with the design of intelligence in an artificial device. The term was coined by John McCarthy in 1956.

❖ Intelligence is the ability to acquire, understand and apply knowledge to achieve goals in the world.

❖ AI is the study of mental faculties through the use of computational models

❖ AI is the study of intellectual/mental processes as computational processes.

- AI programs will demonstrate a high level of intelligence to a degree that equals or exceeds the intelligence required of a human in performing some tasks.
- AI is unique, sharing borders with Mathematics, Computer Science, Philosophy, Psychology, Biology, Cognitive Science, and many others.
- Although there is no clear definition of AI or even Intelligence, it can be described as an attempt to build machines that like humans can think and act, and are able to learn and use knowledge to solve problems on their own.

1.1.1. DEFINITION

The study of how to make computers do things at which at the moment, people are b ..r. "Artificial Intelligence is the ability of a computer to act like a human being".

- Systems that think like humans
- Systems that act like humans
- Systems that think rationally.
- Systems that act rationally.

1.1.2. EXAMPLE OF AI

- **Automation:** What makes a system or process function automatically?
- **Machine Learning:** The science of getting a computer to act without programming.
- **Machine Vision:** The science of allowing computers to see.
- **Natural language processing (NLP):** The processing of human -- and not a computer -- language by a computer program.
- **Robotics:** A field of engineering focused on the design and manufacturing of robots.
- **Self-driving cars:** These use a combination of computer vision, image recognition, and deep learning to build automated skills at piloting a vehicle while staying in a given lane and avoiding unexpected obstructions, such as pedestrians.

1.1.3. HISTORY OF AI

- Important research that laid the groundwork for AI:
- In 1931, Goedel layed the foundation of Theoretical Computer Science1920-30s: He published the first universal formal language and showed that math itself is either flawed or allows for unprovable but true statements.
- In 1936, Turing reformulated Goedel's result and the church's extension thereof.
- In 1956, John McCarthy coined the term "Artificial Intelligence" as the topic of the Dartmouth Conference, the first conference devoted to the subject.
- In 1957, The General Problem Solver (GPS) demonstrated by Newell, Shaw & Simon
- In 1958, John McCarthy (MIT) invented the Lisp language.
- In 1959, Arthur Samuel (IBM) wrote the first game-playing program, for checkers, to achieve the sufficient skill to challenge a world champion.
- In 1963, Ivan Sutherland's MIT dissertation on Sketchpad introduced the idea of interactive graphics into computing.
- In 1966, Ross Quillian (Ph.D. dissertation, Carnegie Inst. of Technology; now CMU) demonstrated semantic nets
- In 1967, the Dendral program (Edward Feigenbaum, Joshua Lederberg, Bruce Buchanan, Georgia Sutherland at Stanford) demonstrated to interpret mass spectra on organic chemical compounds. First successful knowledge-based program for scientific reasoning.
- In 1967, Doug Engelbart invented the mouse at SRI
- In 1968, Marvin Minsky & Seymour Papert publish Perceptrons, demonstrating the limits of simple neural nets.
- In 1972, Prolog developed by Alain Colmerauer.
- In Mid '80s, Neural Networks become widely used with the Backpropagation algorithm (first described by Werbos in 1974).
- 1990, Major advances in all areas of AI, with significant demonstrations in machine learning, intelligent tutoring, case-based reasoning, multi-agent

planning, scheduling, uncertain reasoning, data mining, natural language understanding and translation, vision, virtual reality, games, and other topics.

- In 1997, Deep Blue beats the World Chess Champion Kasparov
- In 2002, iRobot, founded by researchers at the MIT Artificial Intelligence Lab, introduced Roomba, a vacuum cleaning robot. By 2006, two million had been sold.

1.2. AI APPLICATIONS

- Artificial Intelligence has made its way into a number of areas.
- **AI in healthcare.** The biggest bets are on improving patient outcomes and reducing costs. Companies are applying machine learning to make better and faster diagnoses than humans.
- **AI in business.** Robotic process automation is being applied to highly repetitive tasks normally performed by humans. Machine learning algorithms are being integrated into analytics and CRM platforms to uncover information on how to better serve customers.
- **AI in education.** AI can automate grading, giving educators more time. AI can assess students and adapt to their needs, helping them work at their own pace. AI tutors can provide additional support to students, ensuring they stay on track. AI could change where and how students learn, perhaps even replacing some teachers.
- **AI in finance.** AI in personal finance applications, such as Mint or Turbo Tax, is disrupting financial institutions. Applications such as these collect personal data and provide financial advice. Other programs, such as IBM Watson, have been applied to the process of buying a home. Today, the software performs much of the trading on Wall Street.
- **AI in law.** The discovery process, sifting through of documents, in law is often overwhelming for humans. Automating this process is a more efficient use of time. Start-ups are also building question-and-answer computer assistants that can sift programmed-to-answer questions by examining the taxonomy and ontology associated with a database.

- **AI in manufacturing.** This is an area that has been at the forefront of incorporating robots into the workflow. Industrial robots used to perform single tasks and were separated from human workers, but as technology advanced that changed.

The applications of AI are shown in Figure 1.1.

Consumer Marketing

- Have you ever used any kind of credit/ATM/store card while shopping?
- if so, you have very likely been "input" to an AI algorithm
- All of this information is recorded digitally
- Companies like Nielsen gather this information weekly and search for patterns
  - general changes in consumer behavior
  - tracking responses to new products
  - identifying customer segments: targeted marketing, e.g., they find out that consumers with sports cars who buy textbooks respond well to offers of new credit cards.
- Algorithms ("data mining") search data for patterns based on mathematical theories of learning

Identification Technologies

- ID cards e.g., ATM cards can be a nuisance and security risk: cards can be lost, stolen, passwords forgotten, etc
- Biometric Identification, walk up to a locked door
  - Camera
  - Fingerprint device
  - Microphone
  - Computer uses biometric signature for identification
  - Face, eyes, fingerprints, voice pattern
  - This works by comparing data from person at door with stored library
  - Learning algorithms can learn the matching process by analyzing a large library database off-line, can improve its performance.

- ❖ Intrusion Detection
- ❖ Computer security - we each have specific patterns of computer use times of day, lengths of sessions, command used, sequence of commands, etc
  - would like to learn the "signature" of each authorized user
  - can identify non-authorized users
- ❖ How can the program automatically identify users?
  - record user's commands and time intervals
  - characterize the patterns for each user
  - model the variability in these patterns
  - classify (online) any new user by similarity to stored patterns

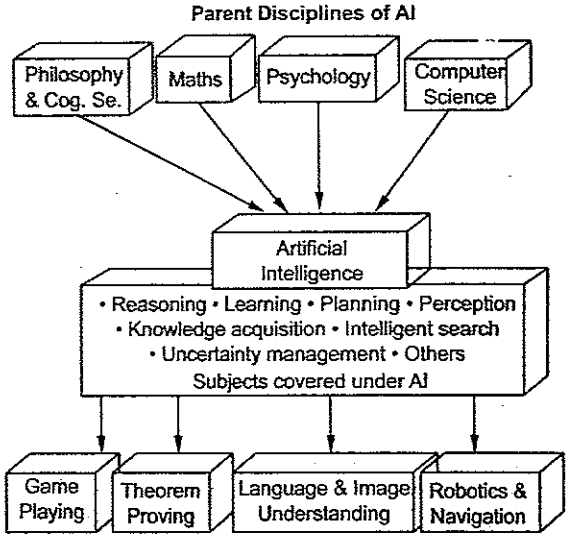


Fig. 1.1. Applications of AI

Machine Translation

- ❖ Language problems in international business
  - e.g., at a meeting of Japanese, Korean, Vietnamese and Swedish investors, no common language
  - If you are shipping your software manuals to 127 countries, the solution is ; hire translators to translate
  - would be much cheaper if a machine could do this!

- ❖ How hard is automated translation
  - very difficult!
  - e.g., English to Russian
  - not only must the words be translated, but their meaning also!

1.3. PROBLEM-SOLVING AGENTS

1.3.1. AGENTS AND ENVIRONMENTS:

An agent is anything that can be viewed as perceiving its environment through sensors and the sensor acts upon that environment through actuators. This simple idea is illustrated in Figure 1.2 .

- ❖ A human agent has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for actuators.
- ❖ A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators.
- ❖ A software agent receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

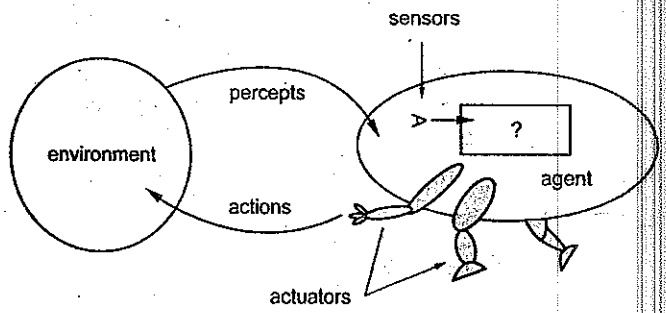


Fig. 1.2. Agents and environments

Percept

We use the term percept to refer to the agent's perceptual inputs at any given instant.

Percept Sequence

An agent's percept sequence is the complete history of everything the agent has ever perceived.

Agent function

Mathematically speaking, we say that an agent's behaviour is described by the agent function that maps any given percept sequence to an action.

$$f : P^* \rightarrow A$$

Agent Program

Internally, the agent function for an artificial agent will be implemented by an agent program. It is important to keep these two ideas distinct. The agent function is an abstract mathematical description; the agent program is a concrete implementation, running on the agent architecture.

To illustrate these ideas, we will use a very simple example-the vacuum-cleaner world shown in Figure 1.3.

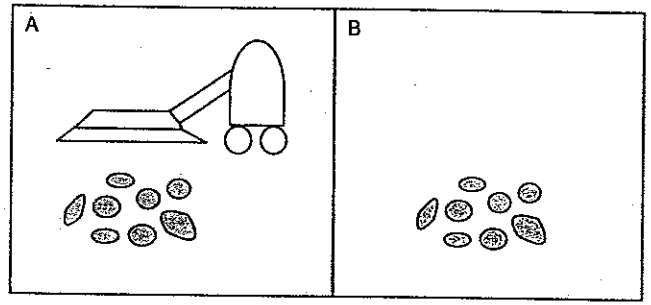


Fig. 1.3. The vacuum-cleaner world

This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck, otherwise move to the other square. A partial tabulation of this agent function is shown in Figure 1.4.

Percept Sequence	Action
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
...	

Fig. 1.4. A partial tabulation of agent function for the vacuum cleaner world

Example Agent Program

```
function Reflex-VACUUM-AGENT ([locations, status]) returns an action
    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

1.3.2. TYPES OF AGENTS

Agents can be grouped into five classes based on their degree of perceived intelligence and capability. All these agents can improve their performance and generate better action over time. These are given below:

- Simple Reflex Agent
- Model-based reflex agent
- Goal-based agents
- Utility-based agent
- Learning agent

Simple Reflex Agent

- The Simple reflex agents are the simplest agents as shown in figure 1.5. These agents take decisions on the basis of the current precepts and ignore the rest of the percept history.
- These agents only succeed in the fully observable environment.
- The Simple reflex agent does not consider any part of percept history during their decision and action process.
- The Simple reflex agent works on the Condition-action rule, which means it maps the current state to action. Such as a Room Cleaner agent, works only if there is dirt in the room.

Problems with the simple reflex agent design approach:

- They have very limited intelligence
- They do not have knowledge of non-perceptual parts of the current state
- Mostly too big to generate and store.
- Not adaptive to changes in the environment.

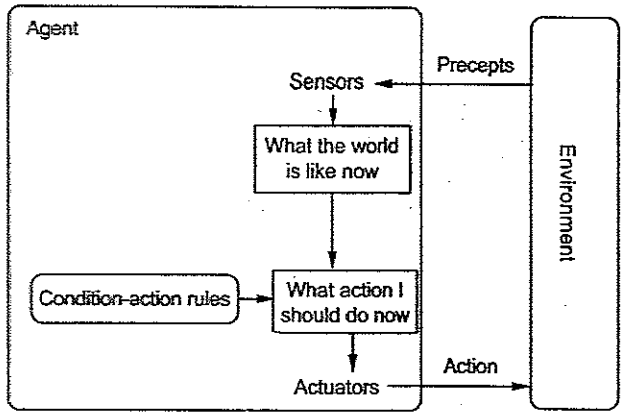


Fig. 1.5. Simple reflex agents

Model-based agent

- The Model-based agent as shown in figure 1.6 can work in a partially observable environment, and track the situation.

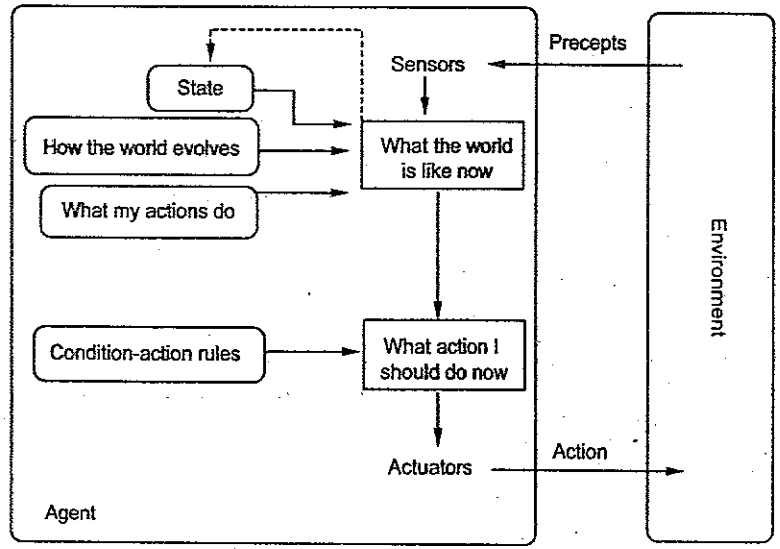


Fig. 1.6. Model-based agent

- A model-based agent has two important factors:
  - **Model:** It is knowledge about "how things happen in the world," so it is called a Model-based agent.

  - **Internal State:** It is a representation of the current state based on percept history.
- These agents have the model, "which is knowledge of the world" and based on the model they perform actions.
- Updating the agent state requires information about:
- How the world evolves
- How the agent's action affects the world.

The model-based reflex agent which keeps track of the current state of the world using an internal model. It then chooses an action in the same way as the reflex agent.

```
function REFLEX-AGENT-WITH-STATE(percept) returns an action
    static: rules, a set of condition-action rules
    state, a description of the current world state
    action, the most recent action.
    state← UPDATE-STATE(state, action, percept)
    rule← RULE-MATCH(state, rule)
    action← RULE-ACTION[rule]
    return action
```

Fig. 1.7. Model-based reflex agent

Goal-based agents

- The knowledge of the current state environment is not always sufficient to decide an agent what to do.
- The agent needs to know its goal which describes desirable situations.
- Goal-based agents as shown in Figure 1.8 expand the capabilities of the model-based agent by having the "goal" information.
- They choose action so that they can achieve the goal.
- These agents may have to consider a long sequence of possible actions before deciding whether the goal is achieved or not. Such considerations of different scenarios are called searching and planning, which makes an agent proactive.

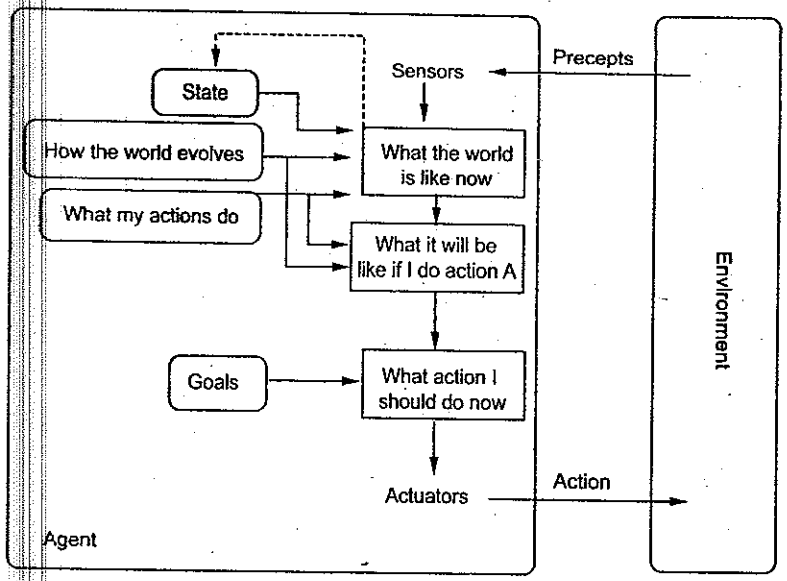


Fig. 1.8. Goal-based agents

Utility-based agent

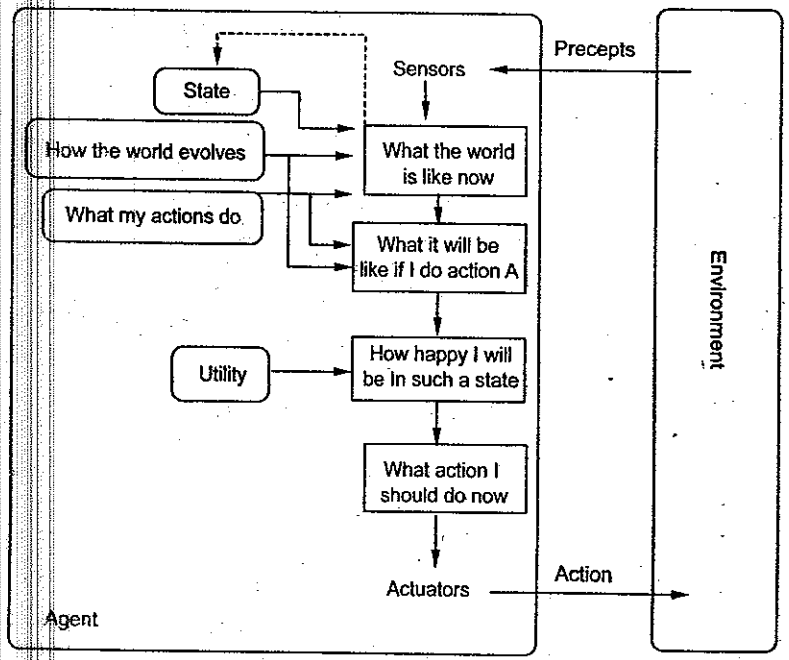


Fig. 1.9. Utility-based agent

- These agents are similar to the goal-based agent but provide an extra component of utility measurement which makes them different by providing a measure of success at a given state.
- Utility-based agent act based not only on goals but also on the best way to achieve the goal.
- The Utility-based agent is useful when there are multiple possible alternatives, and an agent has to choose in order to perform the best action.
- The utility function maps each state to a real number to check how efficiently each action achieves the goals as shown in figure 1.9.

Learning Agent

- A learning agent in AI is the type of agent that can learn from its past experiences, or it has learning capabilities. It is shown in the figure 1.10.
- It starts to act with basic knowledge and then is able to act and adapt automatically through learning.

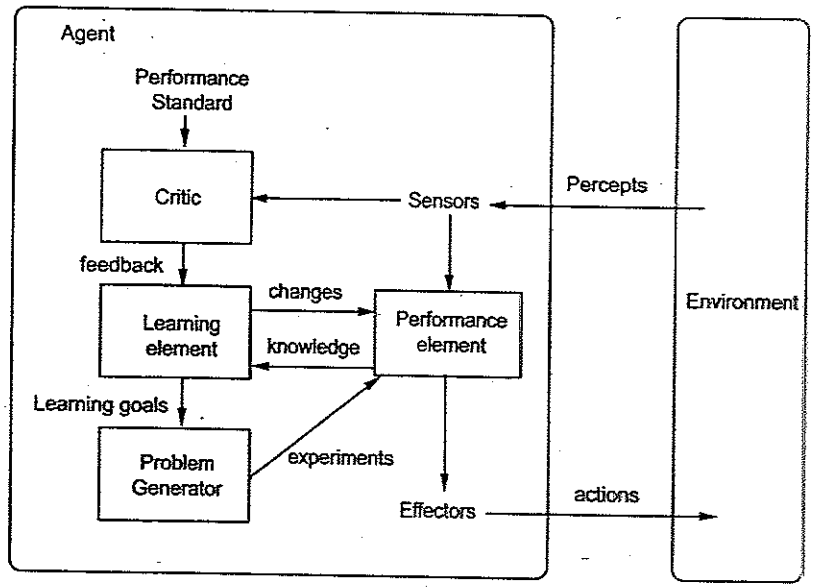


Fig. 1.10. learning agent

- A learning agent has mainly four conceptual components, which are:
  1. **Learning element:** It is responsible for making improvements by learning from the environment

2. **Critic:** The learning element takes feedback from the critic which describes that how well the agent is doing with respect to a fixed performance standard.
3. **Performance element:** It is responsible for selecting external action
4. **Problem generator:** This component is responsible for suggesting actions that will lead to new and informative experiences.

- Hence, learning agents are able to learn, analyze performance, and look for new ways to improve the performance.

1.3.3. PROBLEM-SOLVING AGENTS

Search is one of the operational tasks that characterize AI programs best. Almost every AI program depends on a search procedure to perform its prescribed functions. Problems are typically defined in terms of state, and solution corresponds to goal states. Problem-solving using the search technique performs two sequences of steps:

1. **Define the problem** – The given problem is identified with its required initial and goal state.
2. **Analyze the problem** - The best search technique for the given: the problem is chosen from different AI search techniques which derive one or more goal states in a minimum number of states.

1.3.3.1. Types of Problem

In general, the problem can be classified under any one of the following four types which depend on two important properties. They are

- Amount of knowledge, of the agent on the state and action description.
- How the agent is connected to its environment through its precepts and actions?

The four different types of problems are:

(*i*) Single-state problem

(*ii*) Multiple state problem

(*iii*) Contingency problem

(*iv*) Exploration problem

1.3.3.2. Problem-solving Agents

The problem-solving agent is one kind of goal-based agent, where the agent decides what to do by finding the sequence of actions that lead to desirable states.

Each action changes the state and the aim is to find the sequence of actions and states that lead from the initial (start) state to a final (goal) state. A well-defined problem can be described by:

- **Initial state**
- **Operator or successor function** - for any state x returns $s(x)$, the set of states reachable from x with one action
- **State space** - all states reachable from the initial by any sequence of actions
- **Path** - sequence through state space
- **Path cost** – the function that assigns a cost to a path. The cost of a path is the sum of the costs of individual actions along the path
- **Goal test** - test to determine if at the goal state

The complexity that arises here is the knowledge about the formulation process, (from current state to outcome action) of the agent. If the agent understood the definition of the problem, it is relatively straightforward to construct a search process for finding solutions, which implies that the problem-solving agent should be an intelligent agent to maximize the performance measure.

The sequence of steps done by the intelligent agent to maximize the performance measure:

(*i*) **Goal formulation** - based on the current situation is the first step in problem-solving. Actions that result in a failure case can be rejected without further consideration.

Example:

A problem is defined by four items:

initial state e.g., "at Arad"

successor function S(x) = set of action-state pairs

e.g., S(Arad) = {[Arad → Zerind;Zerind], ...}

goal test, can be

explicit, e.g., x = at Bucharest"

implicit, e.g., No Dirt(x)

path cost (additive)

e.g., sum of distances, number of actions executed, etc.

$e(x; a; y)$ is the step cost, assumed to be $\Rightarrow 0$

A solution is a sequence of actions leading from the initial state to a goal state.

(*ii*) **Problem formulation** - is the process of deciding what actions and states to consider and follows goal formulation.

Example: Route finding problem

On holiday in Romania : currently in Arad

Flight leaves tomorrow from Bucharest

Formulate goal: be in Bucharest

Formulate Problem:

States: various cities

actions: drive between cities

Find Solution:

Sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

(*iii*) **Search** - is the process of finding the different possible sequences of actions that lead to a state of known value, and choosing the best one from the states.

(*iv*) **Solution** - a search algorithm takes a problem as input and returns a solution in the form of an action sequence.

(*v*) **Execution phase** - if the solution exists, the action it recommends can be carried out.

1.3.4. SEARCH

An agent with several immediate options of unknown value can decide what to do by examining different possible sequences of actions that lead to the states of known value and then choosing the best sequence. The process of looking for a sequence of s actions from the current state to reach the goal state is called search.

A simple "formulate, search, execute" design for the agent is given below. Once the solution has been executed, the agent will formulate a new goal.

function SIMPLE-PROBLEM-SOLVING-AGENT (*percept*) **returns** an action

inputs: *percept,* a percept

static:*seq*, an action sequence, initially empty

state, some description of the current world state

goal, a goal, initially null

problem, a problem formulation

state UPDATE-STATE(*state, percept*)

if *seq* is empty **then do**

goal←FORMULATE-GOAL(*state*)

problem←FORMULATE-PROBLEM(*state, goal*)

seq← SEARCH(*problem*)

action←FIRST(*seq*);

seq←REST(*seq*)

return action

The search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the execution phase consists of carrying out the recommended action,

The agent design assumes the Environment is

- **Static:** The entire process is carried out without paying attention to changes that might be occurring in the environment.
- **Observable:** The initial state is known and the agent's sensor detects all aspects that are relevant to the choice of action
- **Discrete:** With respect to the state of the environment and percepts and actions so that alternate courses of action can be taken
- **Deterministic:** The next state of the environment is completely determined by the current state and the actions executed by the agent. Solutions to the problem are a single sequence of actions

An agent carries out their plan with eyes closed. This is called an open loop system because ignoring the percepts breaks the loop between the agent and the environment.

Well-defined problems and solutions

A **problem** can be formally defined by four components:

- The initial state that the agent starts in. The initial state for our agent of example problem is described by In(Arad)
- A Successor Function returns the possible actions available to the agent. Given a state
- x, SUCCESSOR-FN(x) returns a set of {action, successor} ordered pairs where each action is one of the legal actions in state x, and each successor is a state that can be reached from x by applying the action.

For example, from the state In (Arad),the successor function for the Romania problem would return

{ [Go(Sibiu),In(Sibiu)],[Go(Timisoara),In(Timisoara)],[Go(Zerind),In(Zerind)] }

- **State Space:** The set of all states reachable from the initial state. The state space forms a graph in which the nodes are states and the arcs between nodes are actions.

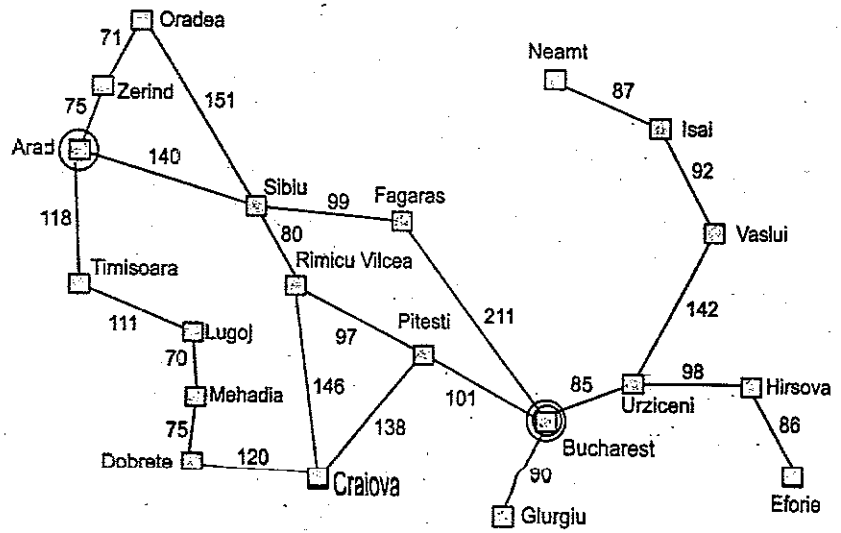


Fig. 1.11. A simplified Road Map of part of Romania

- A path in the state space is a sequence of states connected by a sequence of actions.
- The goal test determines whether the given state is a goal state.
- A path cost function assigns a numeric cost to each action. For the Romania problem, the cost of the path might be its length in kilometres.
- The step cost of taking action a to go from state x to state y is denoted by $c(x, a, y)$. The step cost for Romania is shown in figure 1.11. It is assumed that the step costs are non-negative
- A solution to the problem is a path from the initial state to a goal state.
- An optimal solution has the lowest path cost among all solutions

1.4. EXAMPLE PROBLEMS

The problem-solving approach has been applied to a vast array of task environments. Some best-known problems are summarized below. They are distinguished as a toy or real-world problems

A toy problem is intended to illustrate various problem-solving methods. It can be easily used by different researchers to compare the performance of algorithms.

A real-world problem is one whose solutions people actually care about.

1.4.1. TOY PROBLEM

1. Vacuum World Example

- **States:** The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states
- **Initial State:** Any state can be designated as the initial state.
- **Successor Function:** This generates the legal states that result from trying the three actions (left, right, suck). The complete state space is shown in figure 1.12.
- **Goal Test:** This tests whether all the squares are clean.
- **Path Test:** Each step costs one, so that the path cost is the number of steps in the path

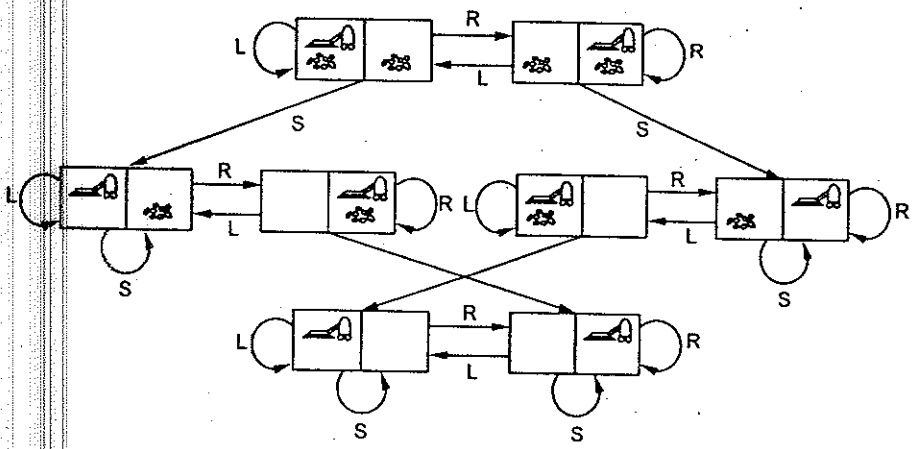


Fig. 1.12. The state space for the vacuum world.

Arcs denote actions: L = Left, R = Right, S = Suck

2. The 8-puzzle example

An 8-puzzle consists of a 3 × 3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach the goal state, as shown in figure 1.13.

Example: The 8-puzzle

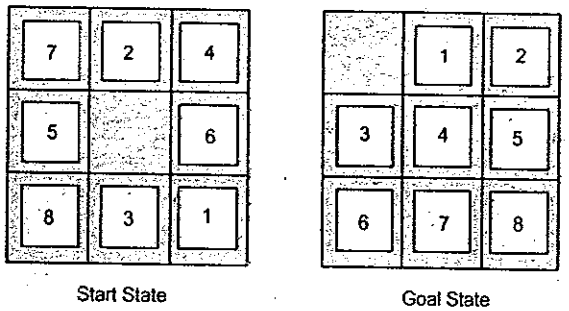


Fig. 1.13. A typical instance of 8-puzzle

The problem formulation is as follows :

- **States:** A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state:** Any state can be designated as the initial state. It can be noted that any given goal can be reached from exactly half of the possible initial states.

- **Successor function:** This generates the legal states that result from trying the four actions (blank moves Left, Right, Vp, or down).
- **Goal Test:** This checks whether the state matches the goal configuration
- Path cost: Each step costs 1, so the path cost is the number of steps in the path.
- The 8-puzzle belongs to the family of sliding-block puzzles, which are often used as test problems for new search algorithms in AI.
- This general class is known as NP-complete.
- The 8-puzzle has 9!/2 = 181,440 reachable states and is easily solved.
- The 15 puzzle (4 × 4 board) has around 1.3 trillion states, and the random instances can be solved optimally in few a milli seconds by the best search algorithms.
- The 24-puzzle (on a 5 × 5 board) has around 1025 states, and random instances are still quite difficult to solve optimally with current machines and algorithms.

3. 8-queens problem

- The goal of the 8-queens problem is to place 8 queens on the chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column, or diagonal).
- The following figure 1.14. shows an attempted solution that fails: the queen in the rightmost column is attacked by the queen at the top left.
- An Incremental formulation involves operators that augment the state description, starting with an empty state. for the 8-queens problem, this means each action adds a queen to the state.
- A complete-state formulation starts with all 8 queens on the board and moves them around.

In either case, the path cost is of no interest because only the final state counts.

- The first incremental formulation one might try is the following :
  - **States:** Any arrangement of 0 to 8 queens on board is a state.
  - **Initial state:** No queen on the board.

- **Successor function:** Add a queen to any empty square.
- **Goal Test:** 8 queens are on the board, and none attacked.

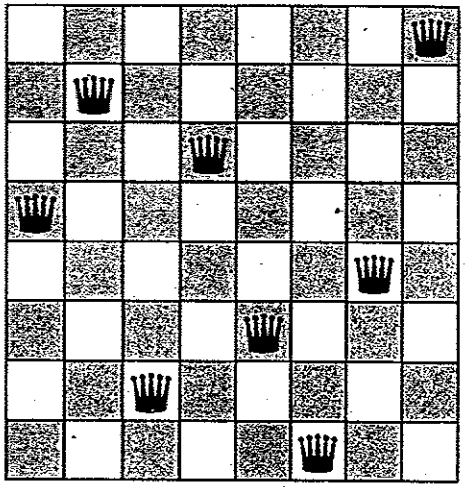


Fig. 1.14. The 8-queens problem

- In this formulation, we have $64.63...57 = 3 \times 10^{14}$ possible sequences to investigate.
- A better formulation would prohibit placing a queen in any square that is already attached:
  - **States:** Arrangements of n queens ($0 \Leftarrow n \Leftarrow 8$), one per column in the leftmost columns, with no queen attacking another state.
  - **Successor function:** Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.
- This formulation reduces the 8-queen state space from 3×10^{14} to just 2057, and solutions are easy to find.
- For the 100 queens, the initial formulation has roughly 10^{400} states whereas the improved formulation has about 1052 states.
- This is a huge reduction, but the improved state space is still too big for the algorithms to handle

1.4.2. REAL WORLD PROBLEMS

- A real-world problem is one whose solutions people actually care about.

- They tend not to have a single agreed-upon description, but attempt is made to give a general flavor of their formulation. The following are some real-world problems,
  - Route Finding Problem
  - Touring Problems
  - Travelling Salesman Problem
  - Robot Navigation

1. ROUTE-FINDING PROBLEM

- Route-finding problem is defined in terms of specified locations and transitions along links between them.
- Route-finding algorithms are used in a variety of applications, such as routing in computer networks, military operations planning, and airline travel planning systems.

2. AIRLINE TRAVEL PROBLEM

The airline travel problem is specified as follows:

- **States:** Each is represented by a location(e.g., an airport) and the current time.
- **Initial state:** This is specified by the problem.
- **Successor function:** This returns the states resulting from taking any scheduled flight(further specified by seat class and location),leaving later than the current time plus the within-airport transit time, from the current airport to another.
- **Goal Test:** Are we at the destination by some prespecified time?
- **Path Cost:** This depends upon the monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

3. TOURING PROBLEMS

- Touring problems are closely related to route-finding problems, but with an important difference.
- Consider, for example, the problem, "Visit every city at least once" as shown on the Romania map.

- As with route-finding, the actions correspond to trips between adjacent cities. The state space, however, is quite different.
  - **Initial state** would be "In Bucharest; visited {Bucharest}".
  - **Intermediate state** would be "In Vaslui; visited{Bucharest,Vrziceni,Vaslui}".
  - **Goal test** would check whether the agent is in Bucharest and whether all 20cities have been visited.

4. THE TRAVELLING SALESPERSON PROBLEM (TSP)

- TSP is a touring problem in which each city must be visited exactly once.
- The aim is to find the shortest tour. The problem is known to be NP-hard.
- Enormous efforts have been expended to improve the capabilities of TSP algorithms.
- These algorithms are also used in tasks such as planning movements of automatic circuit-board drills and of stocking machines on shop floors.

5. VLSI layout

A VLSI layout problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem is split into two parts: cell layout and channel routing.

6. ROBOT navigation

ROBOT navigation is a generalization of the route-finding problem. Rather than a discrete set of routes, a robot can move in a continuous space with an infinite set of possible actions and states. For a circular Robot moving on a flat surface, the space is essentially two dimensional. When the robot has arms and legs or wheels that also must be controlled, the search space becomes multi-dimensional. Advanced techniques are required to make the search space finite.

7. AUTOMATIC ASSEMBLY SEQUENCING

The example includes the assembly of intricate objects such as electric motors. The aim of assembly problems is to find the order in which to assemble the parts of some objects. If the wrong order is chosen, there will be no way to add some parts later without undoing some work already done. Another important assembly problem is protein design, in which the goal is to find a sequence of Amino acids that will be folded into a three-dimensional protein with the right properties to cure some disease.

8. INTERNET SEARCHING

In recent years there has been increased demand for software robots that perform Internet searching, looking for answers to questions, for related information, or for shopping deals. The searching techniques consider the internet as a graph of nodes (pages) connected by links.

9. MEASURING PROBLEM-SOLVING PERFORMANCE

- The output of the problem-solving algorithm is either a failure or a solution.(Some algorithms might be struck in an infinite loop and never return an output.)
- The algorithm's performance can be measured in four ways:
  - **Completeness:** Is the algorithm guaranteed to find a solution when there is one?
  - **Optimality:** Does the strategy find the optimal solution
  - **Time complexity:** How long does it take to find a solution?
  - **Space complexity:** How much memory is needed to perform the search?

1.5. SEARCH ALGORITHMS

Search Tree

Having formulated some problems, we now need to solve them. This is done by a search through the state space. A search tree is generated by the initial state and the successor function that together define the state space. In general, we may have a search graph rather than a search tree, when the same state can be reached from multiple paths. Figure 1.15 shows the Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded.; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed line

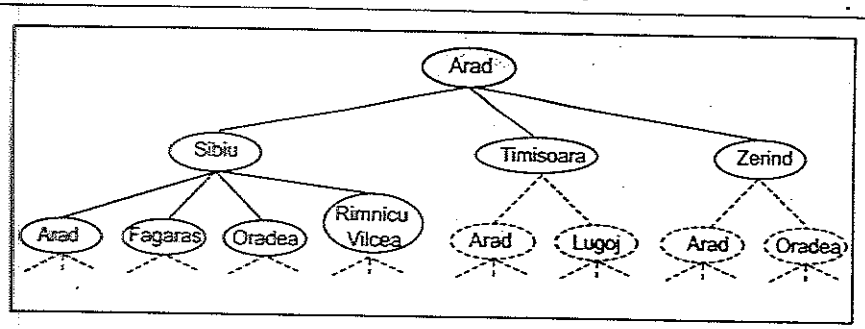


Fig. 1.15. Partial search trees for finding a route from Arad to Bucharest.

The root of the search tree is a search node corresponding to the initial state, In (Arad). The first step is to test whether this is a goal state. The current state is expanded by applying the successor function to the current state, thereby generating a new set of states. In this case, we get three new states: In (Sibiu), In (Timisoara), and In (Zerind). Now we must choose which of these three possibilities to consider further. This is the essence of search- following up on one option now and putting the others aside for latter, in case the first choice does not lead to a solution. Search strategy. The general tree-search algorithm is described informally in Figure 1.16.

```
function GRAPH-SEARCH (problem, fringe) returns a solution, or failure
    closed ←an empty set
    fringe←INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
      if EMPTY?(fringe) then return failure
      node ←REMOVE-FIRST(fringe)
      if GOAL-TEST[problem]⟨STATE[node]⟩ then return SOLUTION(node)
      if State[node] is not in closed then
        add STATE[node] to closed
        fringe ← INSERT-ALL(EXPAND(node, problem),fringe)
```

Fig. 1.16. general tree-search algorithm

The choice of which state to expand is determined by the search strategy. There is an infinite number paths in this state space, so the search tree has an infinite number of nodes.

A node is a data structure with five components:

- **STATE:** a state in the state space to which the node corresponds;
- **PARENT-NODE:** the node in the search tree that generated this node;
- **ACTION:** the action that was applied to the parent to generate the node;
- **PATH-COST:** the cost, denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers; and
- **DEPTH:** the number of steps along the path from the initial state.

It is important to remember the distinction between nodes and states. A node is a book keeping data structure used to represent the search tree. A state corresponds to configuration of the world.

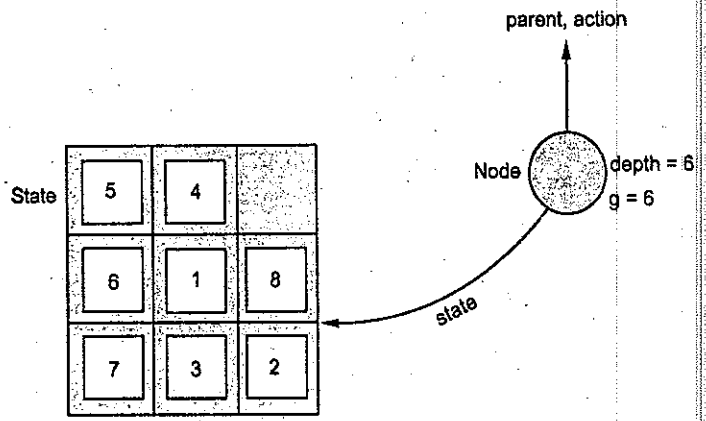


Fig. 1.17. Nodes are data structures from which the search tree is constructed. Each has a parent, and a state, Arrows point from child to parent

Fringe

A fringe is a collection of nodes that have been generated but have not yet been expanded. Each element of the fringe is a leaf node, that is, a node with no successors in the tree. The fringe of each tree consists of those nodes with bold outlines. The collection of these nodes is implemented as a queue. The general tree search algorithm is shown in Figure 1.18.

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
    fringe ←INSERT(MAKE-NODE(INITIAL_STATE[problem]),fringe)
    loop do
```

```
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GPA:-TEST[problem] applied to STATE(node) succeeds return node
    fringe ←INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND(node, problem) returns a set of nodes
  successors ← the empty set
  for each actions, result in SUCCESSOR-FN[problem](State[node]) do
    s← a new NODE
    PARENT-NODE[s] ← node, ACTION[s] ←action; STATE[s] ← result
    PATH-COST[s] ←PATH-COST[node] + STEP-COST(node, action, s)
    DEPTH[s] ← DEPTH[node] + 1
    add s to successors
  return successors
```

Fig. 1.18. general tree search algorithm

The operations specified in Figure 1.18 on a queue are as follows:

- **MAKE-QUEUE**(element…,) creates a queue with the given element(s).
- **EMPTY?** (queue) returns true only if there are no more elements in the queue.
- **FIRST**(queue) returns FIRST(queue) and removes it from the queue.
- **INSERT**(element, queue) inserts an element into the queue and returns the resulting queue.
- **INSERT-ALL**(elements, queue) inserts a set of elements into the queue and returns the resulting queue.

Measuring problem-solving performance

The output of the problem-solving algorithm is either a failure or a solution.(Some algorithms might be struck in an infinite loop and never return an output.

The algorithm's performance can be measured in four ways:

- **Completeness:** Is the algorithm guaranteed to find a solution when there is one?
- **Optimality:** Does the strategy find the optimal solution
- **Time complexity:** How long does it take to find a solution?
- **Space complexity:** How much memory is needed to perform the search?

1.5.1. UNINFORMED SEARCH STRATEGIES

Uninformed Search Strategies have no additional information about states beyond that provided in the problem definition. Strategies that know whether one non-goal state is "more promising" than another are called Informed search or heuristic search strategies. There are five uninformed search strategies as given below.

1. Breadth-first search
2. Uniform-cost search
3. Depth-first search
4. Depth-limited search
5. Iterative deepening search

1. Breadth-first search

Breadth-first search is a simple strategy in which the root node is expanded first, then all successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. Breath-first-search is implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out (FIFO) queue, assuring that the nodes that are visited first will be expanded first. In other words, calling TREE-SEARCH (problem, FIFO-QUEUE ()) results in breadth-first-search.

The FIFO queue puts all newly generated successors at the end of the queue, which means that Shallow nodes are expanded before deeper nodes.

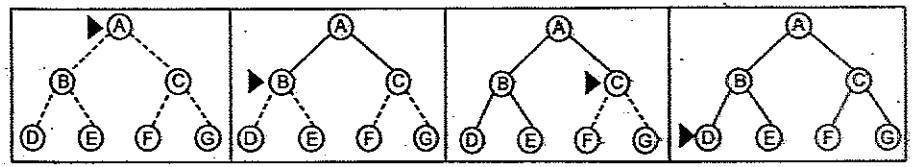


Fig. 1.19. Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Properties of breadth-first-search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp, in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? No, unless step costs are constant

Space is the big problem; can easily generate nodes at 100MB/sec so 24 hrs = 8640GB.

Example

- $b = 10$
- 10000 nodes/second
- Each node requires 1000 bytes of storage

Depth	Nodes	Time	Memory
2	1100	11 seconds	1 megabytes
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Table 1.1. Time and memory requirements for breadth-first-search.

Time Complexity for BFS

- Assume every state has b successors.
- The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level.
- Each of these generates b more nodes, yielding b^3 nodes at the third level, and so on.

- Now suppose, that the solution is at depth d.
- In the worst case, we would expand all but the last node at level d, generating $b^{d+1} - b$ nodes at level $d + 1$.
- Then the total number of nodes generated is $b + b^2 + b^3 + \ldots + b^d + (b^{d+1} + b) = O(b^{d+1})$.
- Every node that is generated must remain in memory, because it is either part of the fringe or is an ancestor of a fringe node.
- The space complexity is, therefore, the same as the time complexity

2. Uniform-cost search

- Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost.
- uniform-cost search does not care about the number of steps a path has, but only about their total cost.

Properties of Uniform-cost-search:

Completeness:

- Uniform-cost search is complete, such as if there is a solution, UCS will find it.

Time Complexity:

- Let C^* **is Cost of the optimal solution**, and ε is each step to get closer to the goal node. Then the number of steps is $= C^*/\varepsilon + 1$. Here we have taken $+ 1$, as we start from state 0 and end to C^*/ε.
- Hence, the worst-case time complexity of Uniform-cost search is $O(b^{1+\lceil C^*/\varepsilon \rceil})$.

Space Complexity:

- The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $O(b^{1+\lceil C^*/\varepsilon \rceil})$.

Optimal:

- Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

3. Depth-first-search

- Depth-first-search always expands the deepest node in the current fringe of the search tree.
- The progress of the search is illustrated in the figure 1.20.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.

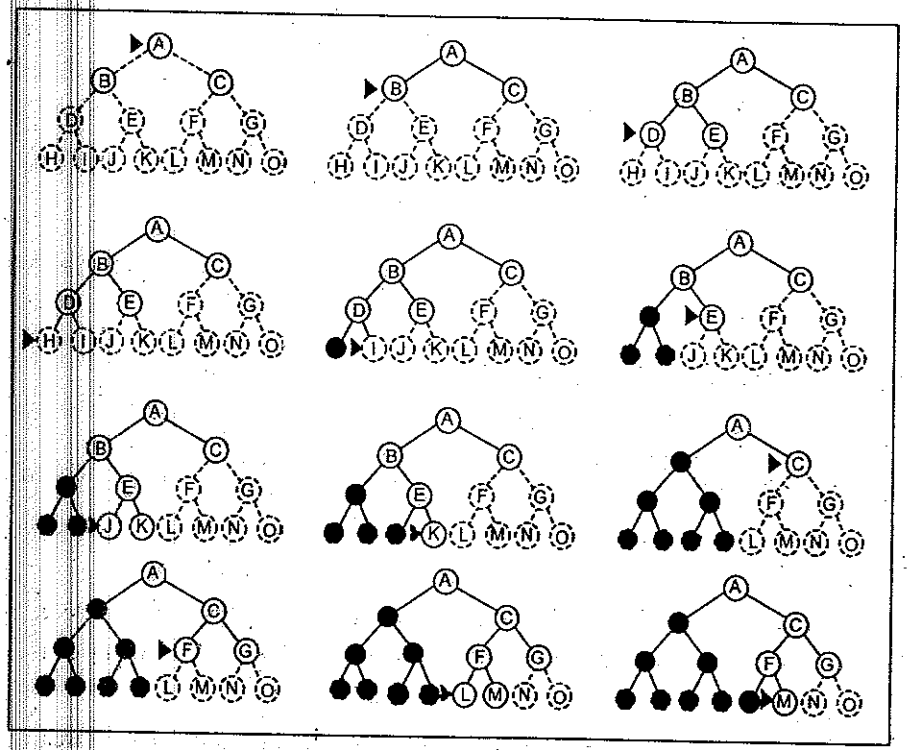


Fig. 1.20. Depth-first-search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from the memory; these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

- As those nodes are expanded, they are dropped from the fringe, so, then the search "backs up" to the next shallowest node that still has unexplored successors.
- This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue, also known as a stack.
- Depth-first-search has very modest memory requirements.

- It needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.
- Once the node has been expanded, it can be removed from the memory, as soon as its descendants have been fully explored.
- For a state space with a branching factor b and maximum depth m, depth-first-search requires storage of only $bm + 1$ nodes.

Drawback of Depth-first-search

The drawback of depth-first-search is that it can make a wrong choice and get stuck going down a very long (or even infinite) path when a different choice would lead to a solution near the root of the search tree.

For example, depth-first-search will explore the entire left subtree even if node C is a goal node.

Backtracking Search

A variant of depth-first search called backtracking search uses less memory and only one successor is generated at a time rather than all successors.; Only $O(m)$ memory is needed rather than $O(b^m)$.

4. Depth-limited-search

- The problem of unbounded trees can be alleviated by supplying depth-first-search with a pre-determined depth limit l.
- That is, nodes at depth l are treated as if they have no successors.
- This approach is called depth-limited-search.
- The depth limit solves the infinite path problem.
- Depth-limited search will be nonoptimal if we choose $l > d$. Its time complexity is $O(b^l)$ and its space complexity is $O(b^l)$.
- Depth-first-search can be viewed as a special case of depth-limited search with $l = oo$
- Sometimes, depth limits can be based on knowledge of the problem.
- For, example, on the map of Romania there are 20 cities.

- Therefore, we know that if there is a solution., it must be of length 19 at the longest, So l = 10 is a possible choice.
- However, it can be shown that any city can be reached from any other city in at most 9steps.
- This number known as the diameter of the state space, gives us a better depth limit.
- Depth-limited-search can be implemented as a simple modification to the general tree search algorithm or to the recursive depth-first-search algorithm.
- The pseudocode for recursive depth-limited-search is shown.
- It can be noted that the above algorithm can terminate with two kinds of failure: the standard failure value indicates no solution; the cut-off value indicates no solution within the depth limit.
- Depth-limited search = depth-first search with depth limit l, returns cut off if any path is cut off by depth limit
- Recursive implementation of Depth-limited-search:

```
function Depth-Limited-Search(problem, limit) returns a solution/fail/cut-off
return Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)
function Recursive-DLS(node, problem, limit) returns solution/fail/cutoff
cutoff-occurred?← false
if Goal-Test(problem,State[node]) then return Solution(node)
else if Depth[node] = limit then return cutoff
else for each successor in Expand(node, problem) do
result←Recursive-DLS(successor, problem, limit)
if result = cutoff then cutoff_occurred? ←true
else if result not = failure then return result
if cutoff_occurred? then return cutoff elsereturn failure
```


5. Iterative deepening depth-first search

- Iterative deepening search (or iterative-deepening-depth-first-search) is a general strategy often used in combination with depth-first-search, that finds the better depth limit.
- It does this by gradually increasing the limit - first 0, then 1, then 2, and so on - until a goal is found.
- This will occur when the depth limit reaches d, the depth of the shallowest goal node.
- Iterative deepening combines the benefits of depth-first and breadth-first-search
- Like depth-first-search, its memory requirements are modest; O(bd) to be precise.
- Like Breadth-first-search, it is complete when the branching factor is finite and optimal when the path cost is a non-decreasing function of the depth of the node.
- The following figure shows the four iterations of ITERATIVE-DEEPENING_SEARCH on a binary search tree, where the solution is found on the fourth iteration.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
    inputs: problem, a problem
    for depth ← 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
    end
```

Fig. 1.21. The iterative deepening search algorithm, which repeatedly applies depth-limited-search with increasing limits. It terminates when a solution is found or if the depth limited search returns failure, meaning that no solution exists.

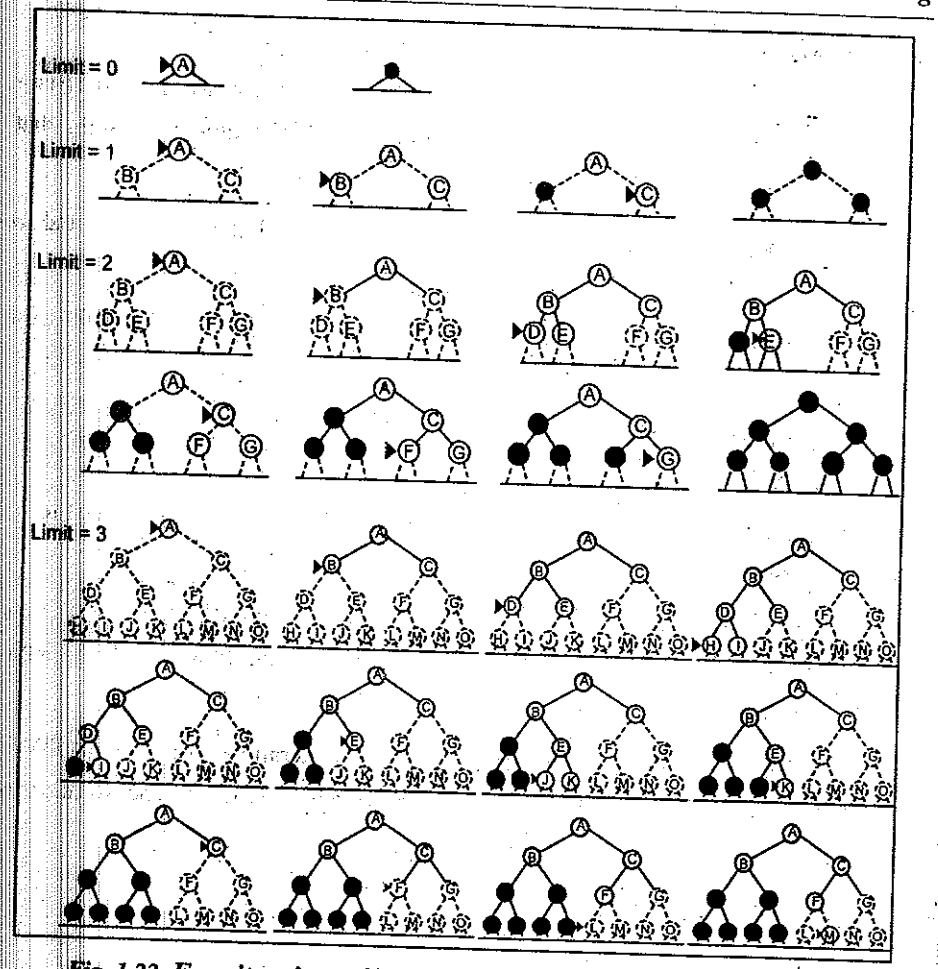


Fig. 1.22. Four iterations of iterative deepening search on a binary tree

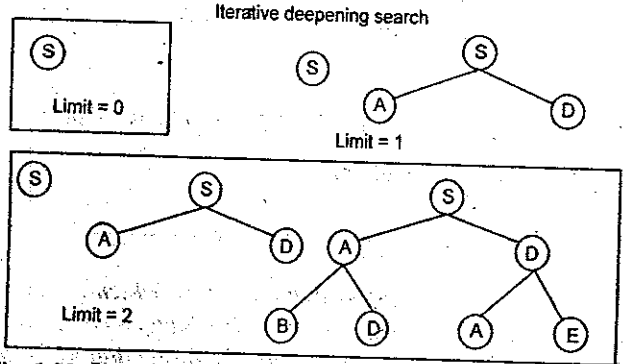


Fig. 1.23. Iterative search is not as wasteful as it might seem

- Iterative search is not as wasteful as it might seem

Properties of iterative deepening search

Complete?? Yes

Time?? $(d + 1)\, b^0 + d\, b^1 + (d - 1)b^2 + \ldots + b^d = o(b^d)$

Space?? $O(b^d)$

Optimal?? No, unless step costs are constant

Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450

N(BES) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100

IDS does better because other nodes at depth d are not expanded

BFS can be modified to apply goal test when a node is generated

In general, iterative deepening is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.

Bidirectional Search

- The idea behind bidirectional search is to run two simultaneous searches – one forward from the initial state and the other backward from the goal, stopping when the two searches meet in the middle (Figure 1.24)
- The motivation is that $b^{d/2} + b^{d/2}$ is much less than, or in the figure, the area of the two small circles is less than the area of one big circle centred on the start and reaching to the goal.

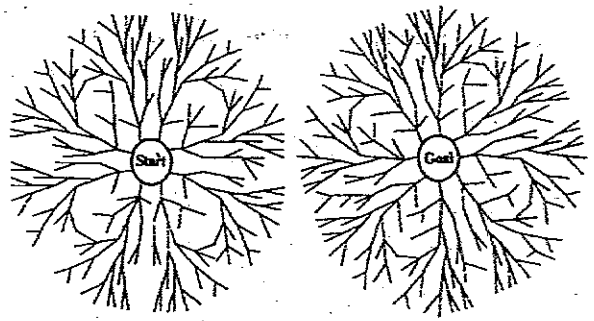


Fig. 1.24. A schematic view of a bidirectional search that is about to succeed, when a Branch from the Start node meets a Branch from the goal node.

Example:

In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph / tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.

The algorithm terminates at node 9 where two searches meet.

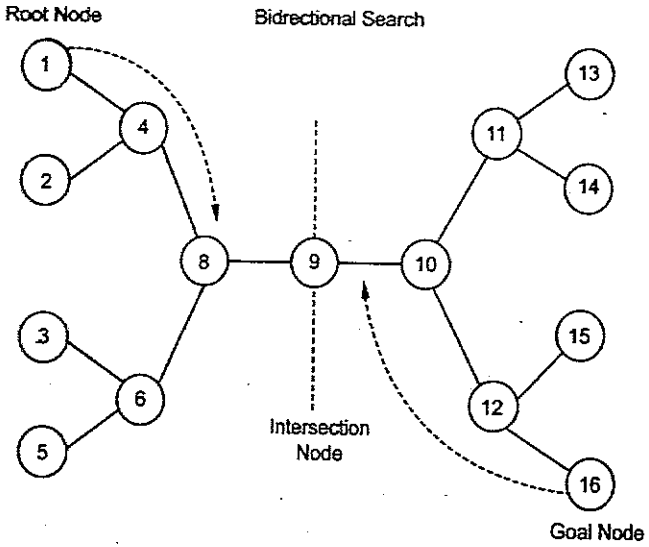


Fig. 1.25. bidirectional search algorithm

Comparing Uninformed Search Strategies

The following table compares search strategies in terms of the four evaluation criteria.

	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional Breadth-F
complete?	Yes	Yes	No	No	Yes	Yes
optimal?	Yes	Yes	No	No	Yes	Yes
time complexity	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
space complexity	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$

1.5.2. INFORMED (HEURISTIC) SEARCH STRATEGIES

The informed search strategy is one that uses problem-specific knowledge beyond the definition of the problem itself. It can find solutions more efficiently than an uninformed strategy.

1. Best-first search

Best-first search is an instance of a general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an evaluation function $f(n)$. The node with the lowest evaluation is selected for expansion because the evaluation measures the distance to the goal. This can be implemented using a priority queue, a data structure that will maintain the fringe in ascending order of f -values.

Heuristic functions

- A heuristic function or simply a heuristic is a function that ranks alternatives in various search algorithms at each branching step based on available information in order to make a decision on which branch is to be followed during a search.
- The key component of the Best-first search algorithm is a heuristic function, denoted by $h(n)$:
- $h(n)$ = estimated cost of the cheapest path from node n to a goal node.
- Figure 1.26 for example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via a straight-line distance from Arad to Bucharest.
- Heuristic function is the most common form in which additional knowledge is imparted to the search algorithm

2. Greedy Best-first search

A greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to be a solution quickly. It evaluates the nodes by using the heuristic function $f(n) = h(n)$. Taking the example of Route-finding problems in Romania, the goal is to reach Bucharest starting from the city of Arad. We need to know the straight-line distances to Bucharest from various cities as shown in Figure 1.26 for example, the initial state is In (Arad), and the straight line distance heuristic h_{SLD} (In(Arad)) is found to be 366. Using the straight-line distance heuristic h_{SLD}, the goal state can be reached faster.

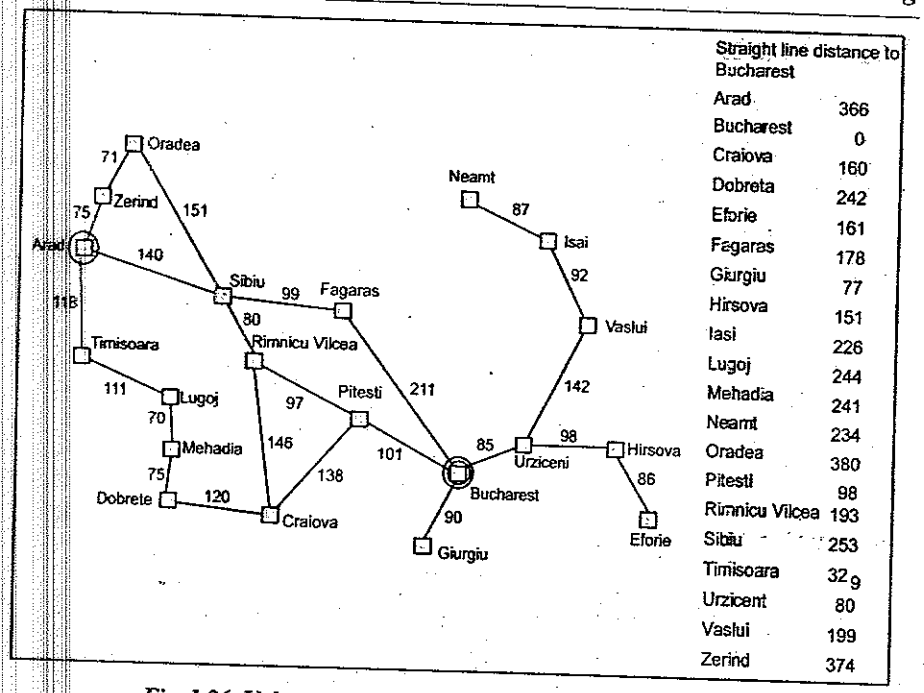


Fig. 1.26. Values of h_{SLD} - straight line distances to Bucharest

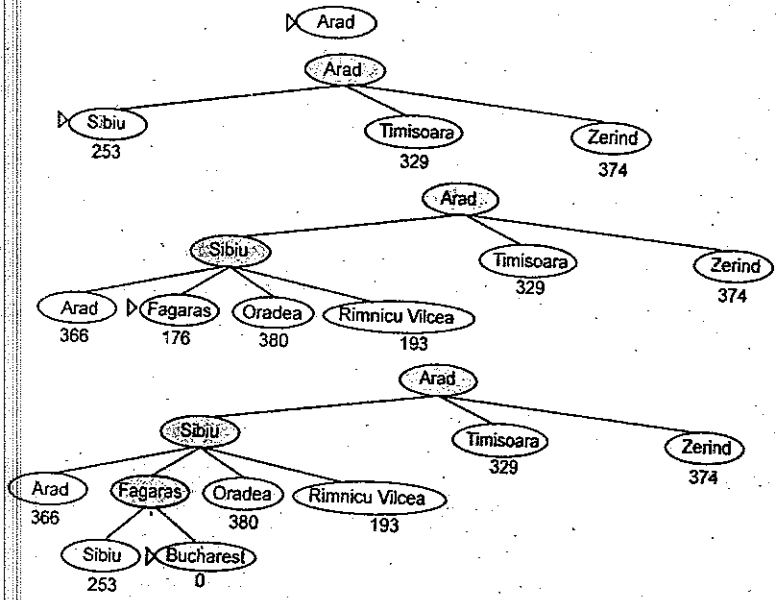


Fig. 1.27. Strategies in greedy best-first search for Bucharest using straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

- The above figure shows the progress of the greedy best-first search using h_{SLD} to find a path from Arad to Bucharest.
- The first node to be expanded from Arad will be Sibiu, because it is closer to Bucharest than either Zerind or Timisoara.
- The next node to be expanded will be Fagaras, because it is closest.
- Fagaras in turn generates Bucharest, which is the goal.

Properties of greedy search

- **Complete:** No one can get stuck in loops,
  - e.g., Iasi! Neamt! Iasi! Neamt!
  - Complete in finite space with repeated-state checking Time?? $O(b_m)$, but a good heuristic can give dramatic improvement
- **Space:** $O(b_m)$-keeps all nodes in memory
- **Optimal:** No Greedy best-first search is not optimal, and it is incomplete.
- The worst-case time and space complexity is $O(b_m)$, where m is the maximum depth of the search space.

3. A* Search

A* Search is the most widely used form of best-first search. The evaluation function f(n) is obtained by combining

1. $g(n)$ = the cost to reach the node, and
2. $h(n)$ = the cost to get from the node to the goal:

$$f(n) = g(n) + h(n).$$

A* Search is both optimal and complete. A* is optimal if $h(n)$ is an admissible heuristic. The obvious example of an admissible heuristic is the straight-line distance h_{SLD}. It cannot be overestimated.

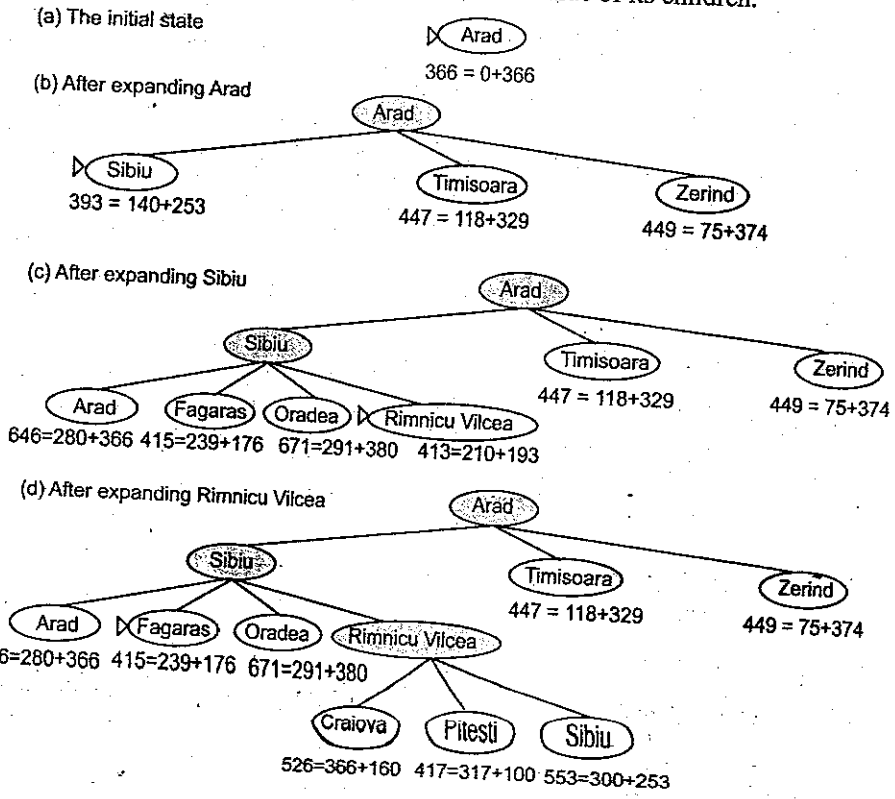
A* Search is optimal if $h(n)$ is an admissible heuristic – that is, provided that h(n) never overestimates the cost to reach the goal. An obvious example of an admissible heuristic is the straight-line distance h_{SLD} that we used in getting to Bucharest. The

progress of an A* tree search for Bucharest is shown in Figure 1.28. The values of 'g' are computed from the step costs shown in the Romania map (figure 1.27). Also, the values of h_{SLD} are given in Figure 1.27.

Recursive Best-first Search (RBFS)

Recursive best-first search is a simple recursive algorithm that attempts to mimic the operation of standard best-first search but uses only linear space.

- The algorithm is shown in the below figure.
- Its structure is similar to that of recursive depth-first search, but rather than continuing indefinitely down the current path, it keeps track of the f-value of the best alternative path available from any ancestor of the current node.
- If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with the best f-value of its children.



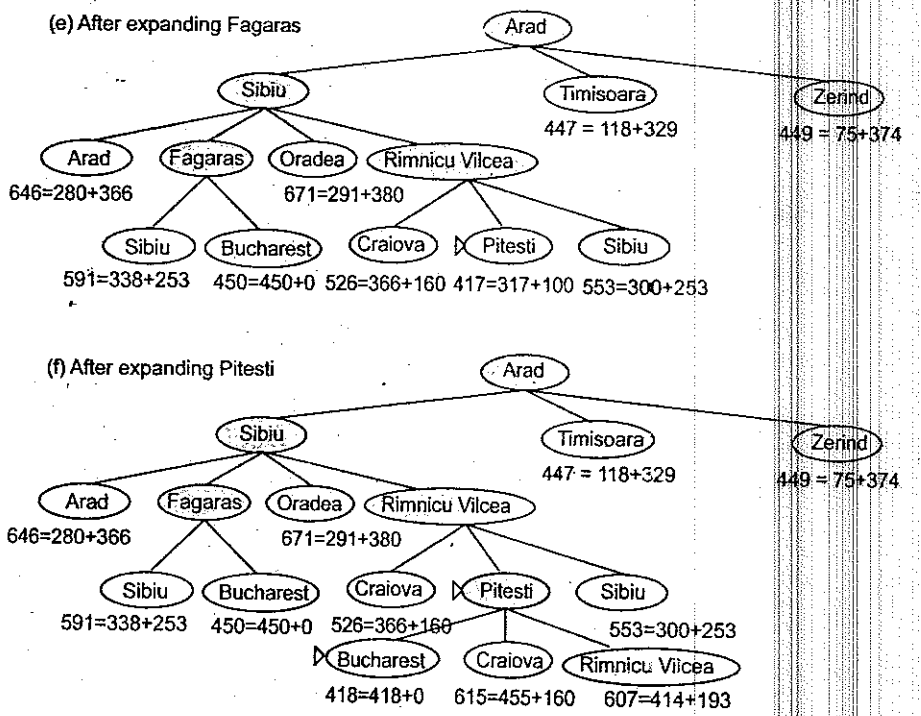


Fig. 1.28. Stages in A Search for Bucharest. Nodes are labelled with $f = g + h$. The h-values are the straight-line distances to Bucharest taken from figure Route map of Romania.*

The algorithm for recursive best-first search

function RECURSIVE-BEST-FIRST-SEARCH(problem) **return** a solution or failure

return RFBS(problem, MAKE-NODE(INITIAL-STATE[problem]), ∞)

function RFBS(problem, node f_limit) **return** a solution or failure and a new f-cost limit

if GOAL-TEST[problem], (STATE[node]) then return node

successors ← EXPAND (node, problem)

if successors is empty then **return** failure, ∞

for each s in successors **do**

$f[s] - \max(g(s) + h(s), f\,[\text{node}])$

repeat

best – the lowest f-value node in successors

```
if f [best] > f_limit then return failure, f[best]
alternative – the second lowest f-value among successors
result, f[best] – RBFS (problem, best, min(f_limit, alternative))
if result ↔ failure then return result
```

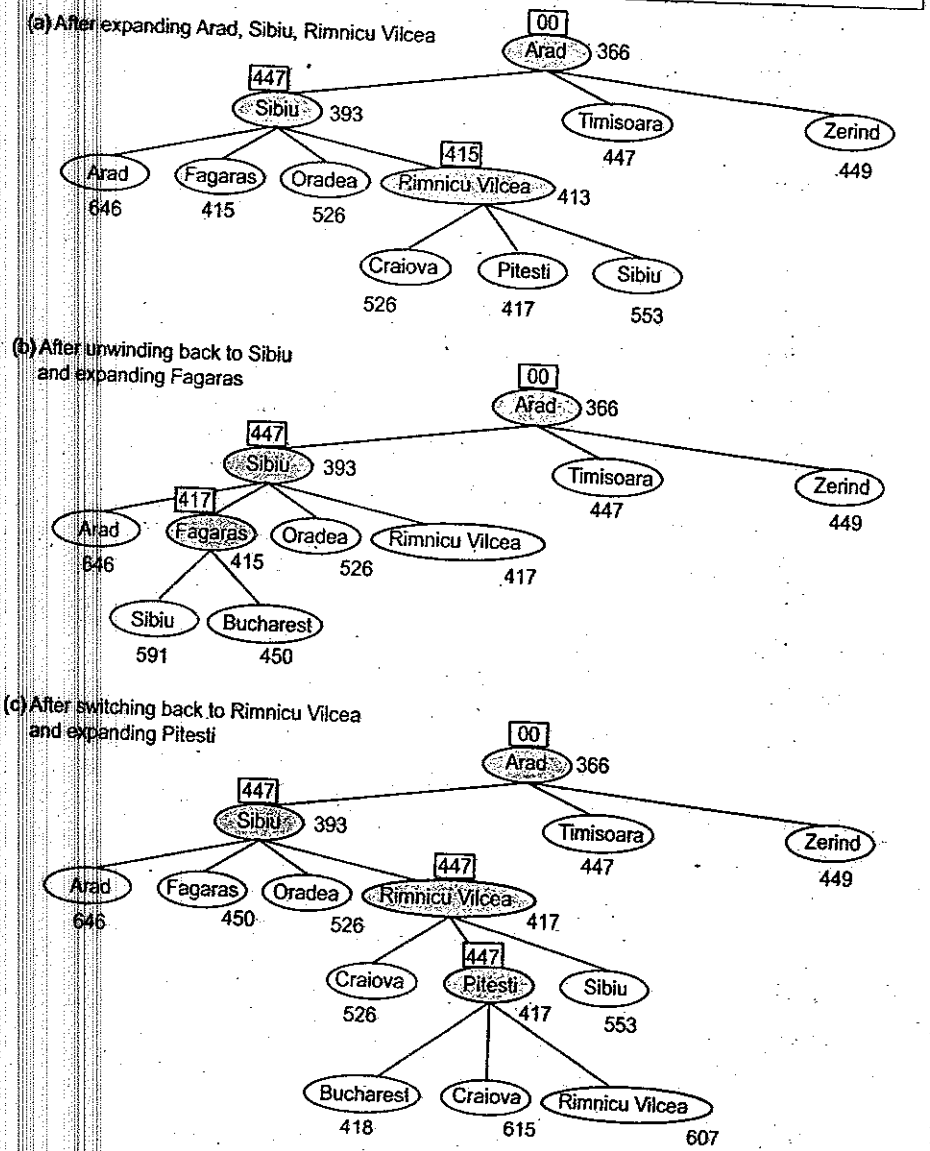


Fig. 1.29.

Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node.

(*a*) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras).

(*b*) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450.

(*c*) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimni Vicea is expanded.

This time because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

RBFS Evaluation :

- ❖ RBFS is a bit more efficient than IDA*
  - Still excessive node generation (mind changes)
- ❖ Like A*, optimal if $h(n)$ is admissible
- ❖ Space complexity is O(bd).
  - IDA* retains only one single number (the current f-cost limit)
- ❖ Time complexity difficult to characterize
  - Depends on accuracy if $h(n)$ and how often best path changes.
- ❖ IDA* en RBFS suffer from too little memory.

1.5.3. HEURISTIC FUNCTIONS

A heuristic function or simply a heuristic is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search

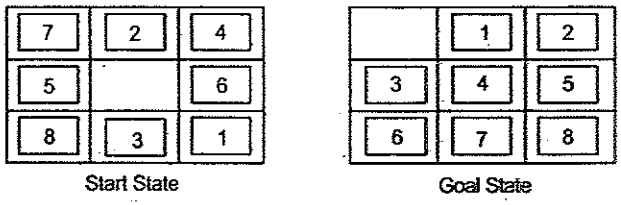


Fig. 1.30. A typical instance of the 8-puzzle.

The solution is 26 steps long,

The 8-puzzle

- The 8-puzzle is an example of a Heuristic search problem.
- The object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration
- The average cost for a randomly generated 8-puzzle instance is about 22 steps.
- The branching factor is about 3. (When the empty tile is in the middle, there are four
- possible moves; when it is in the corner there are two, and when it is along an edge there are three).
- This means that an exhaustive search to depth 22 would look at about 3^{22} approximately = 3.1×10^{10} states.
- By keeping track of repeated states, we could cut this down by a factor of about 170, 000, because there are only 9! / 2 = 181,440 distinct states that are reachable.
- This is a manageable number, but the corresponding number for the 15-puzzle is roughly 10^{13}
- If we want to find the shortest solutions by using A*, we need a heuristic function that never overestimates the number of steps to the goal.
- The two commonly used heuristic functions for the 15-puzzle are:
- h_1 = the number of misplaced tiles.
- In the above figure all of the eight tiles are out of position, so the start state would have $h_1 = 8$. h_1 is an admissible heuristic.
- h_2 = the sum of the distances of the tiles from their goal positions.
- This is called the city block distance or Manhattan distance. h_2 is admissible, because all any move can do is move one tile one step closer to the goal.
- Tiles 1 to 8 in start state give a Manhattan distance of $h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$.
- Neither of these overestimates the true solution cost, which is 26.

The Effective Branching factors

- One way to characterize the quality of a heuristic is the effective branching factor b^*.
- If the total number of nodes generated by A* for a particular problem is N, and the solution depth is d, then b* is the branching factor that a uniform tree of depth d would have to have in order to contain N + 1 nodes. Thus, $N + 1 = 1 + b^* + (b^*)2 + \ldots + (b^*)^d$
- For example, if A* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92.
- A well-designed heuristic would have a value of b^*close to 1, allowing the failure of large problems to be solved.
- To test the heuristic functions h_1 and h_2, 1200 random problems were generated with solution lengths from 2 to 24 and solved with iterative deepening search and with A*search using both h_1 and h_2.
- The following table gives the average number of nodes expanded by each strategy and the effective branching factor.
- The results suggest that h_2 is better than h_1, and is far better than using iterative deepening search.
- For a solution length of 14, A*with h_2 is 30,000 times more efficient than uninformed iterative deepening search.

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	-	539	113	-	1.44	1.23

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
16	-	1301	211	-	1.45	1.25
18	-	3056	363	-	1.46	1.26
20	-	7276	676	-	1.47	1.27
22	-	18094	1219	-	1.48	1.28
24	-	39135	1641	-	1.48	1.26

Fig. 1.31. Comparison of search costs and effective branching factors for the ITERATIVEDEEPENING-SEARCH and A* Algorithms with h_1, and h_2. Data are average over 100 instances of the 8-puzzle, for various solution lengths.

Inventing admissible heuristic functions

- Relaxed problems
- A problem with fewer restrictions on the actions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution

1.6. LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution

- For example, in the 8-queens problem, what matters is the final configuration of queens, not the order in which they are added.
- In such cases, we can use local search algorithms. They operate using a single current state (rather than multiple paths) and generally move only to neighbours of that state.
- The important applications of these class of problems are
  - (*a*) Integrated-circuit design,

  - (*b*) Factory-floor layout,
  - (*c*) Job-shop scheduling,
  - (*d*) Automatic programming,
  - (*e*) Telecommunications network optimization,
  - (*f*) Vehicle routing, and
  - (*g*) Portfolio management.

Key advantages of Local Search Algorithms

1. They use very little memory – usually a constant amount; and
2. They can often find reasonable solutions in large or infinite(continuous) state spaces for which systematic algorithms are unsuitable.

Optimization problems

In addition to finding goals, local search algorithms are useful for solving pure optimization problems, in which the aim is to find the best state according to an objective function.

State Space Landscape

- To understand local search, it is better explained using state space landscape as shown in figure 1.32.

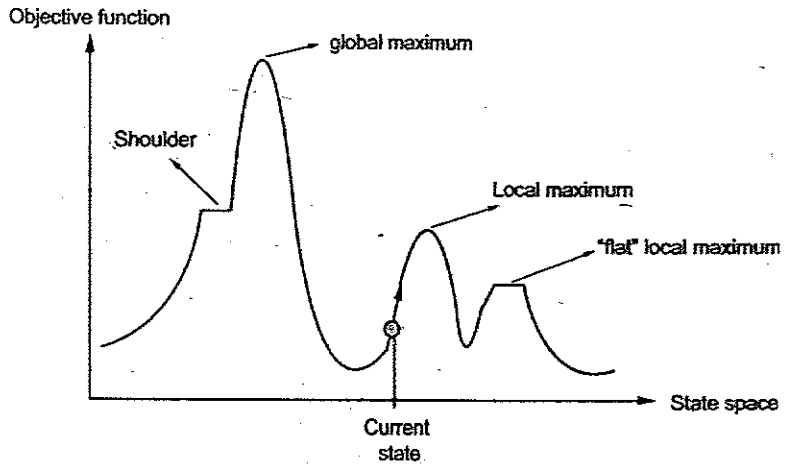


Fig. 1.32. A one-dimensional state space landscape in which elevation corresponds to the objective function.

- A landscape has both "location" (defined by the state) and "elevation"(defined by the value of the heuristic cost function or objective function).
- If elevation corresponds to cost, then the aim is to find the lowest valley – a global minimum; if elevation corresponds to an objective function, then the aim is to find the highest peak – a global maximum.
- Local search algorithms explore this landscape. A complete local search algorithm always finds a goal if one exists; an optimal algorithm always finds a global minimum/maximum.

The aim of a one-dimensional state space landscape in which elevation corresponds to the objective function to find the global maximum. Hill climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Hill-climbing search

The hill-climbing search algorithm as shown in figure 1.33, is simply a loop that continually moves in the direction of increasing value – that is, uphill. It terminates when it reaches a "peak" where no neighbour has a higher value.

```
function HILL-CLIMBING( problem) return a state that is a local maximum
    input: problem, a problem
    local variables: current, a node.
    neighbor, a node.
    current← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest valued successor of current
        if VALUE [neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
```

The hill-climbing search algorithm (steepest ascent version), is the most basic local search technique. At each step, the current node is replaced by the best neighbor; the neighbor with the highest VALUE. If the heuristic cost estimate *h* is used, we could find the neighbor with the lowest h.

Hill-climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next. Greedy algorithms often perform quite well.

Problems with hill-climbing

Hill-climbing often gets stuck for the following reasons:

- **Local maxima:** a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upwards towards the peak, but will then be stuck with nowhere else to go.
- **Ridges:** A ridge is shown in Figure 1.33. Ridges results in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
- **Plateaux:** A plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which it is possible to make progress.

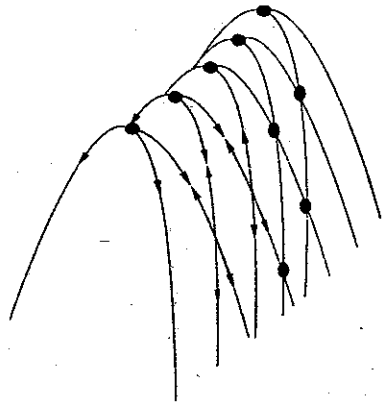


Fig. 1.33. Illustration of why ridges cause difficulties for hill-climbing.

The Figure 1.33 shows that the grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available options point downhill.

Hill-climbing variations

- Stochastic hill-climbing
  - Random selection among the uphill moves.

- The selection probability can vary with the steepness of the uphill move.

❖ First-choice hill-climbing

- CFR. stochastic hill climbing by generating successors randomly until a better one is found.

❖ Random-restart hill-climbing

- Tries to avoid getting stuck in local maxima.

Simulated annealing search

A hill-climbing algorithm that never makes "downhill" moves towards states with a lower value (or higher cost) is guaranteed to be incomplete, because it can stuck on a local maximum. In contrast, a purely random walk –that is, moving to a successor chosen uniformly at random from the set of successors – is complete, but extremely inefficient.

Simulated annealing is an algorithm that combines hill-climbing with a random walk in someway that yields both efficiency and completeness.

The below table shows the simulated annealing algorithm. It is quite similar to hill climbing. Instead of picking the best move, however, it picks the random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move – the amount ΔE by which the evaluation is worsened.

Simulated annealing was first used extensively to solve VLSI layout problems in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks.

function SIMULATED-ANNEALING(problem, schedule) returns a solution state

inputs: problem, a problem

schedule, a mapping from time to "temperature"

local variables: T, a "temperature" controlling the probability of downward steps

current – MAKE-NODE(problem.INITIAL-STATE)

for t = 1 to ∞ do

T ← schedule(t)

if T = 0 then **return** current

next ← a randomly selected successor of current

ΔE ← next. VALUE – current Value

if ΔE > 0 then current ← next

else current ← next only with probability $e^{\Delta E/T}$

The simulated annealing search algorithm, a version of stochastic hill climbing where some downhill moves are allowed

Genetic algorithms

A Genetic algorithm (or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states, rather than by modifying a single state.

Like beam search, Gas begins with a set of k randomly generated states, called the population. Each state, or individual, is represented as a string over a finite alphabet – most commonly, a string of 0s and 1s. For example, an 8 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires 8 × $\log_2 8 = 24$ bits.

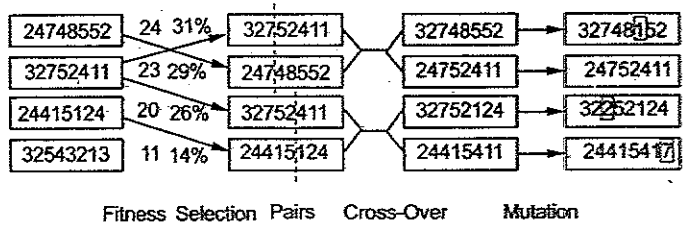


Fig. 1.34. The genetic algorithm. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subjected to mutation in (e).

function GENETIC_ALGORITHM (population, FITNESS-FN) return an individual

input: population, a set of individuals

FITNESS-FN, a function which determines the quality of the individual

repeat

```
        new_population ← empty set
        loop for i from 1 ro SIZE(population) do
            x ← RANDOM_SELECTION (population, FITNESS_FN)
    y ← RANDOM_SELECTION (population, FITNESS_FN)
            child ← REPRODUCES (x, y)
            if (small random probability) then child ← MUTATE (child)
                add child to new_population
        population ← new_population
until some individual is fit enough or enough time has elapsed
return the best individual
```

Fig. 1.35. A genetic algorithm

1.7. ADVERSARIAL SEARCH

Competitive environments, in which the agent's goals are in conflict, give rise to adversarial search problems – often known as games.

1.7.1. GAMES

Mathematical Game Theory, a branch of economics, views any multiagent environment as a game provided that the impact of each agent on the other is "significant", regardless of whether the agents are cooperative or competitive. In, AI," games" are deterministic, turn-taking, two-player, zero-sum games of perfect information. This means deterministic, fully observable environments in which there are two agents whose actions must alternate and in which the utility values at the end of the game are always equal and opposite. For example, if one player wins the game of chess (+1), the other player necessarily loses (–1). It is this opposition between the agents' utility functions that makes the situation adversarial.

Formal Definition of Game

We will consider games with two players, whom we will call MAX and MIN. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player, and penalties are given to the loser. A game can be formally defined as a search problem with the following components:

- ❖ The initial state, which includes the board position and identifies the player to move.
- ❖ A successor function, which returns a list of (move, state) pairs, each indicating a legal move and the resulting state.
- ❖ A terminal test, which describes when the game is over. States, where the game has ended, are called terminal states.
- ❖ A utility function (also called an objective function or payoff function), which gives a numeric value for the terminal states. In chess, the outcome is a win, loss, or draw, with values +1, –1, or 0. the payoffs in backgammon range from +192 to –192.

Game Tree

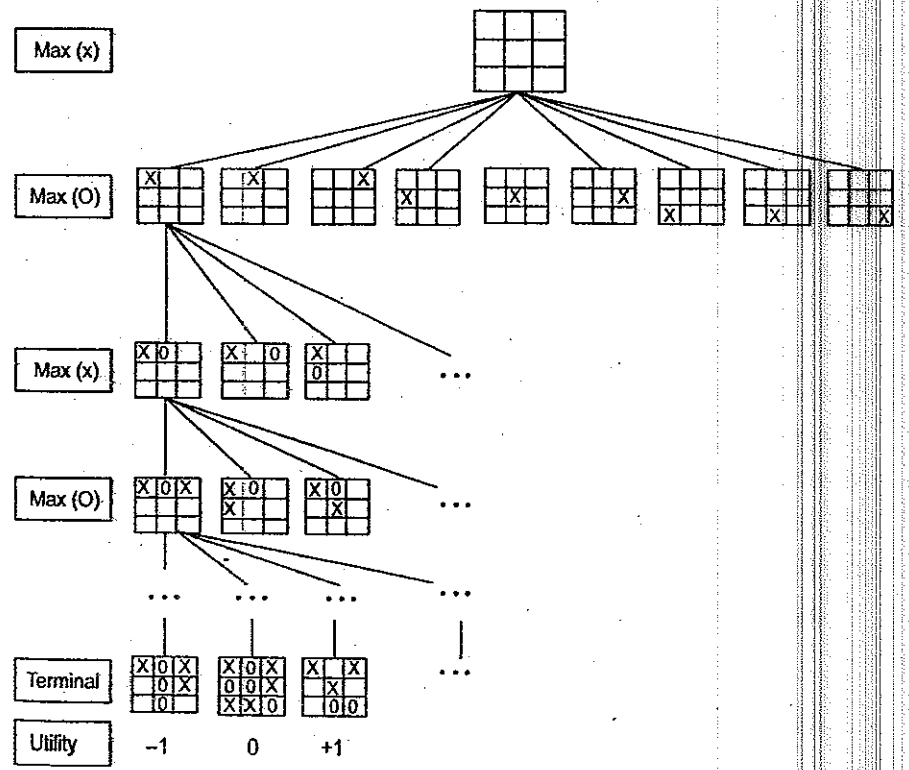


Fig. 1.36. A partial search tree. The top node is the initial state, and MAX moves first, placing an X in an empty square.

The initial state and legal moves for each side define the game tree for the game. Figure 1.36 shows the part of the game tree for tic-tac-toe (noughts and crosses). From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing a 0 until we reach leaf nodes corresponding to the terminal states such that one player has three in a row or all the squares are filled. He numbers on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN. It is the MAX's job to use the search tree (particularly the utility of terminal states) to determine the best move.

1.7.2. OPTIMAL DECISIONS IN GAMES

In a normal search problem, the optimal solution would be a sequence of moves leading to a goal state – a terminal state that is a win. In a game, on the other hand, MIN has something to say about it, MAX therefore must find a contingent strategy, which specifies MAX's move in the initial state, then MAX's moves in the states resulting from every possible response by MIN, then MAX's moves in the states resulting from every possible response by MIN those moves, and so on. An optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.

The nodes Δ are "MAX nodes", in which it is AMX's turn to move, and the ∇ nodes are "MIN nodes". The terminal nodes show the utility values for MAX; the other nodes are labelled with their minimax values. MAX's best move at the root is a_1, because it leads to the successor with the highest minimax value, and MIN's best reply is b_1, because it leads to the successor with the lowest minimax value.

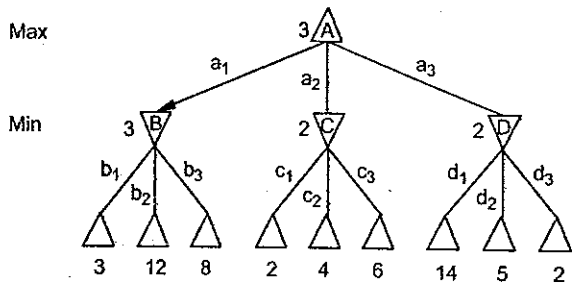


Fig. 1.37. A two-ply game tree

The minimax Algorithm

- The minimax algorithm (Figure 1.38) computes the minimax decision from the current state.
- It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations.
- The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are backed up through the tree as the recursion unwinds.
- For example, in Figure 1.38, the algorithm first recourses down to the three bottoms left nodes, and uses the utility function on them to discover that their values are 3, 12, and 8 respectively.
- Then it takes the minimum of these values, 3, and returns it as the backed-up value of node B. A similar process gives the backed-up values of 2 for C and 2 for D.
- Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 at the root node.
- The minimax algorithm performs a complete depth-first exploration of the game tree.
- If the maximum depth of the tree is m, and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b_m)$.
- The space complexity is $O(b_m)$ for an algorithm that generates successors at once.

```
function MINIMAX-DECISION(State) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS (state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v
```

```
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

Fig. 1.38. An algorithm for calculating minimax decisions.

It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state.

1.7.3. ALPHA-BETA PRUNING

The problem with minimax search is that the number of game states it has to examine is exponential in the number of moves. Unfortunately, we can't eliminate the exponent, but we can effectively cut it in half. By performing pruning, we can eliminate a large part of the tree from consideration. We can apply the technique known as alpha-beta pruning, when applied to a minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Alpha Beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:

- **α:** the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path of MAX.
- **β:** the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path of MIN.

Alpha Beta search updates the values of α and β as it goes along and prunes the remaining branches at the node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α and β value for MAX and MIN, respectively. The complete algorithm is given in Figure 1.39.

The effectiveness of alpha-beta pruning is highly dependent on the order in which the successors are examined. It might be worthwhile to try to examine first the successors that are likely to be the best. In such case, it turns out that alpha-beta needs to examine only $O(b^{d/2})$ nodes to pick the best move, instead of $O(b^d)$ for minimax. This means that the effective branching factor becomes sqrt(b) instead of b – for chess,6 instead of 35. Put another way alpha-beta cab look ahead roughly twice as far as minimax in the same amount of time.

```
function ALPHA-BETA-SEARCH(state) returns an action
    v ← MAX-VALUE(state, – ∞, + ∞)
    return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← – ∞
for each a in ACTIONS(state)do
    v ← MAX(v, MIN-VALUE(RESULT(s, a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
return v

function MIN-VALUE(state, α, β)returns a utility value
    if TERMINAL_TEST(state) then return UTILITY(state)
    v ← + ∞
    for each a in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(s, a), α., β))
        if v ≤ α the return v
        β ← MIN(β, v)
    return v
```

Fig. 1.39. The alpha beta search algorithm.

These routines are the same as the minimax routines in figure 1.38, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β.

1.7.4. IMPERFECT, REAL-TIME DECISIONS

The minimax algorithm generates the entire game search space, whereas the alpha-beta algorithm allows us to prune large parts of it. However, alpha-beta still has to search all the way to terminal states for at least a portion of the search space. Shannon's 1950 paper, programming a computer for playing chess, proposed that programs should cut off the search earlier and apply a heuristic evaluation function to states in the search, effectively turning nonterminal nodes into terminal leaves. The basic idea is to alter minimax or alpha-beta in two ways:

1. The utility function is replaced by a heuristic evaluation function EVAL, which gives an estimate of the position's utility, and
2. The terminal test is replaced by a cut-off test that decides when to apply EVAL.

1.7.5. GAMES THAT INCLUDE ELEMENT OF CHANCE

Evaluation functions. An evaluation function returns an estimate of the expected utility of the game from a given position, just as the heuristic function returns an estimate of the distance to the goal.

Games of imperfect information

- Minimax and alpha-beta pruning require too many leaf-node evaluations. May be impractical within a reasonable amount of time. o SHANNON (1950):
- Cut off search earlier (replace TERMINAL-TEST by CUTOFF-TEST)
- Apply heuristic evaluation function EVAL (replacing utility function of alpha-beta)

Cutting off search

Change:

- if TERMINAL-TEST(state) then return UTILITY(state) into
- if CUTOFF-TEST(state, depth) then return EVAL(state)
- Introduces a fixed-depth limit depth

- Is selected so that the amount of time will not exceed what the rules of the game allow.

When cuttoff occurs, the evaluation is performed

Heuristic EVAL

Idea: produce an estimate of the expected utility of the game from a given position. Performance depends on quality of EVAL.

Requirements:

- EVAL should order terminal nodes in the same way as UTILITY.
- Computation may not take too long.
- For non-terminal states the EVAL should be strongly correlated with the actual chance of winning.

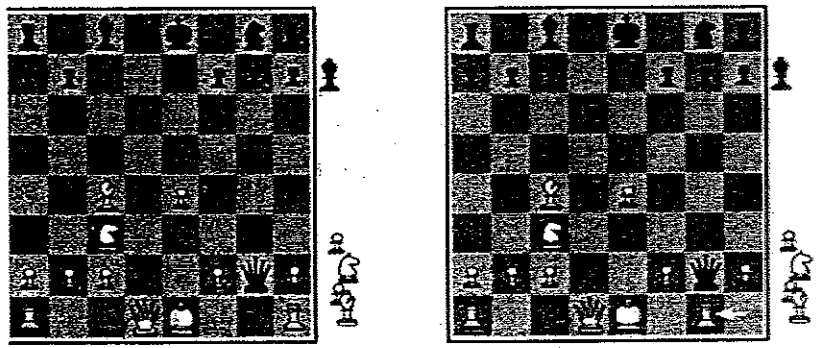
Only useful for quiescent (no wild swings in value in near future) states

Weighted Linear Function

The introductory chess books give an approximate material value for each piece: each pawn is worth 1, a knight or bishop is worth 3, a rook 3, and the queen 9. These feature values are then added up to obtain the evaluation of the position. Mathematically, this kind of evaluation function is called weighted linear function, and it can be expressed as:

$$\text{Eval(s)} = w_1 f_1(s) + w_2 f_2(s) + \ldots \div w_n f_n(s)$$

e.g., $w_1 = 9$ with $f_1(s)$ = (number of white queens) – (number of black queens), etc



(a) White to move (b) White to move

a) Black has an advantage of a knight and two pawns and will the game

(b) Black will lose after white captures the queen

Fig. 1.40. Weighted Linear Function

Games that include chance

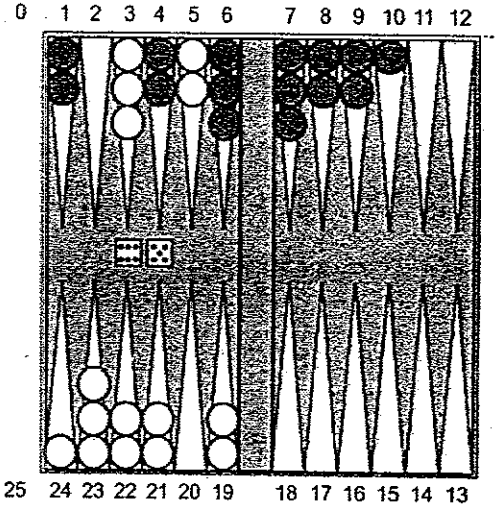


Fig. 1.41. A typical backgammon position.

In real life, there are many unpredictable external events that put us into unforeseen situations. Many games mirror this unpredictability by including a random element, such as throwing a dice. Backgammon is a typical game that combines luck and skill. Dice are rolled at the beginning of player's turn to determine the legal moves. The backgammon position of Figure 1.41, for example, white has rolled a 6-5, and has four possible moves.

The goal of the game is to move all one's pieces off the board. White moves clockwise toward 25, and black moves counter clockwise toward 0. A piece can move to any position unless there are multiple opponent pieces there; if there is one opponent, it is captured and must start over. In the position shown, white has rolled 6-5 and must choose among four legal moves (5-10,5-11), (5-11,19-24), (5-10,10-16), and (5-11,11-16)

- White moves clockwise toward 25
- Black moves counter clockwise toward 0
- A piece can move to any position unless there are multiple opponent pieces there; if there is one opponent, it is captured and must start over.
- White has rolled 6-5 and must choose among four legal moves: (5-10, 5-11), (5-11, 19-24) (5-10, 10-16), and (5-11, 11-16)

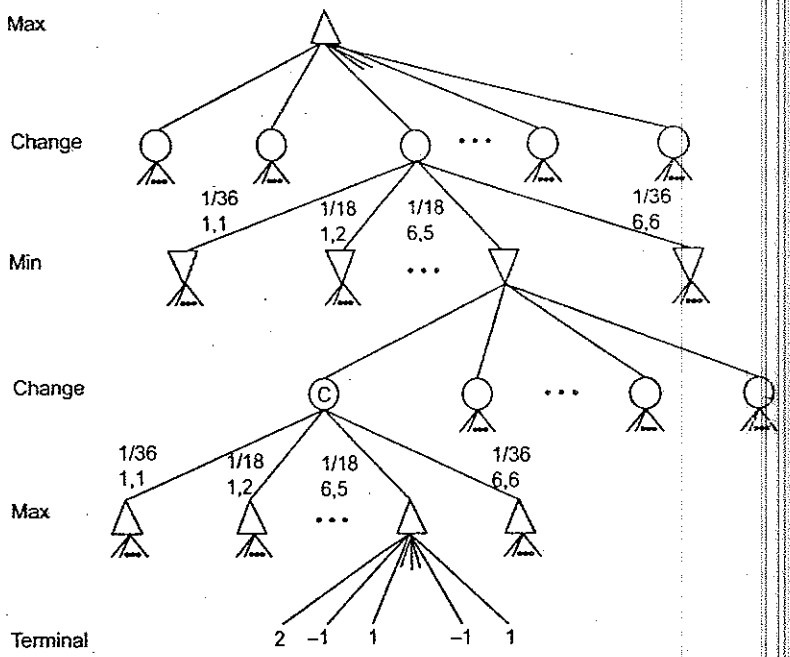


Fig. 1.42. Schematic game tree for a backgammon position

1.8. CONSTRAINT SATISFACTION PROBLEMS (CSP)

A Constraint Satisfaction Problem (or CSP) is defined by a set of variables, X_1, X_2,.... X_n, and a set of constraints C_1, C_2,..., C_m. Each variable Xi has a nonempty domain D, of possible values. Each constraint C_i involves some subset of variables and specifies the allowable combinations of values for that subset.

A State of the problem is defined by an assignment of values to some or all of the variables, $\{X_i = v_i , X_j = v_j ,...\}$. An assignment that does not violate any constraints is called a consistent or legal assignment. A complete assignment is one in which every variable is mentioned, and a solution to a CSP is a complete assignment that satisfies all the constraints.

Some CSPs also require a solution that maximizes an objective function.

Example of Constraint Satisfaction Problem:

- Figure 1.43 shows the map of Australia showing each of its states and territories. We are given the task of coloring each region either red, green, or blue in such a way that the neighboring regions have the same color.

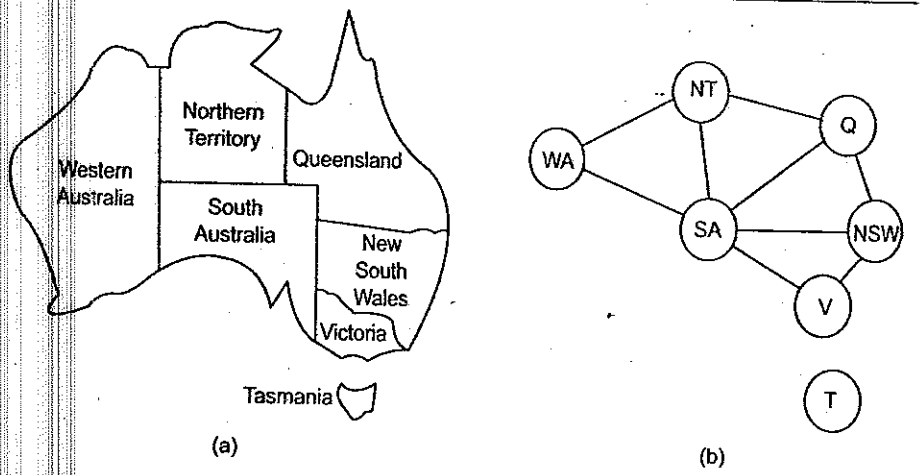


Fig. 1.43.(a) Principle states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map coloring problem represented as a constraint graph.

- To formulate this as CSP, we define the variable to be the regions: WA, NT, Q, NSW, V, SA, and T. The domain of each variable is the set {red, green, blue}.
- The constraints require neighboring regions to have distinct colors; for example, the allowable combinations for WA and NT are the pairs {(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)}.
- The constraint can also be represented more succinctly as the inequality WA not = NT, provided the constraint satisfaction algorithm has some way to evaluate such expressions.) There are many possible solutions such as {WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red}
- It is helpful to visualize a CSP as a constraint graph, as shown in Figure 1.43 (*b*). The nodes of the graph correspond to variables of the problem and the arcs correspond to constraints.

CSP can be viewed as a standard search problem as follows:

- **Initial state:** the empty assignment {}, in which all variables are unassigned.
- **Successor function:** a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.

- **Gol test:** the current assignment is complete.
- **Path cost:** a constant cost (E.g., 1) for every step.

Every solution must be a complete assignment and therefore appears at depth n if there are n variables. Depth-first search algorithms are popular for CSPs.

1.8.1. VARIETIES OF CSPS

(*i*) Discrete variables

Finite domains

The simplest kind of CSP involves variables that are discrete and have finite domains. Map coloring problems are of this kind. The 8-queens problem can also be viewed as finite-domain CSP, where the variables $Q_1, Q_2, \ldots, Q_8$ are the positions of each queen in columns 1,....8, and each variable has the domain {1,2,3,4,5,6,7,8}. If the maximum domain size of any variable in a CSP is d, then the number of possible complete assignments is O(d^n) – that is, exponential in the number of variables. Finite domain CSPs include **Boolean CSPs**, whose variables can be either true or false.

Infinite domains

Discrete variables can also have infinite domains – for example, a set of integers or a set of strings. With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combinations of values. Instead, a constraint language algebra inequality such as $\text{Startjob}_1 + 5 \leq \text{Startjob}_3$.

(*ii*) CSPs with continuous domains

CSPs with continuous domains are very common in the real world. For example, in the operation research field, the scheduling of experiments on the Hubble Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical, precedence and power constraints. The best-known category of continuous-domain CSPs is that of linear programming problems, where the constraints must be linear inequalities forming a convex region. Linear programming problems can be solved in time polynomial in the number of variables.

Varieties of constraints:

(*i*) unary constraints involve a single variable.

Example: SA # green

(*ii*) Binary constraints involve Pairs of variables.

Example: SA # WA

(*iii*) Higher order constraints involve 3 or more variables.

Example: cryptarithmetic puzzles.

Variables : F T U W R O X_1 X_2 X_3

Domains : {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Constraints : Alldiff(F, T, U, W, R, O}

$O + O = R = 10 \times X_1$

$X_1 + W + W = U + 10 \times X_2$

$X_2 + T + T = O + 10 \times X_3$

$X_3 = F, T \neq 0, F \neq 0$

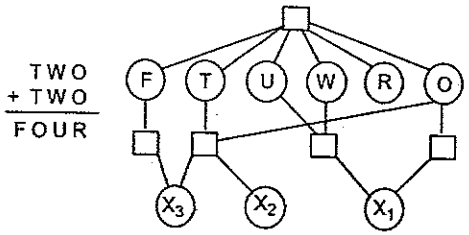


Fig. 1.44. Cryptarithmetic problem.

Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeros are allowed. (*b*) The constraint hypergraph for the cryptarithmetic problem, showing the Alldiff constraint as well as the column addition constraints. Each constraint is a square box connected to the variables it contains.

1.8.2. BACKTRACKING SEARCH FOR CSPS

The term backtracking search is used for depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in figure 1.45.


```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return BACKTRACK{},csp)
function BACKTRACK(assignment,csp) returns a solution, or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp)
    if assignment is complete then return assignment
    var ←SELECT-UNASSIGNED-VARIABLE(csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp)do
      if value is consistent with assignment then
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, value)
        if inferences ≠ failure then
          add inferences to assignment
          result ← BACKTRACK(assignment, csp)
          if result ≠ failure then
            return result
      remove {var = value} and inferences from assignment
    return failure
```

Fig. 1.45. (a) A simple backtracking algorithm for constraint satisfaction problem. The algorithm is modelled on the recursive depth-first search

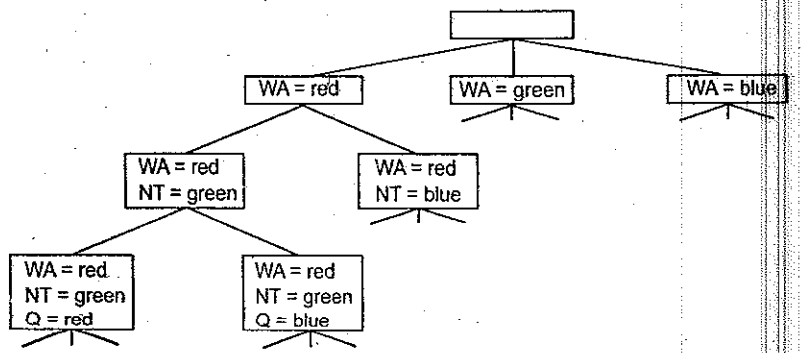


Fig. 1.45. (b) Part of search tree generated by simple backtracking for the map coloring problem

Propagating information through constraints

So far our search algorithm considers the constraints on a variable only at the time that the variable is chosen by SELECT-UNASSIGNED-VARIABLE. But by looking at some of the constraints earlier in the search, or even before the search has started, we can drastically reduce the search space.

Forward Checking

One way to make better use of constraints during search is called forward checking. Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y 's domain any value that is inconsistent with the value chosen for X. Figure 1.46 shows the progress of a map-coloring search with forward checking.

	WA	NT	Q	NSW	V	SA	T
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After WA = red	(R)	G B	R G B	R G B	R G B	G B	R G B
After Q = green	(R)	B	(G)	R B	R G B	B	R G B
After V = blue	(R)	B	(G)	R	(B)		R G B

Fig. 1.46. The progress of a map-coloring search with forward checking.

Constraint propagation

Although forward checking detects many inconsistencies, it does not detect all of them. Constraint propagation is the general term for propagating the implications of a constraint on one variable onto other variables.

Arc Consistency

The idea of arc consistency provides a fast method of constraint propagation that is substantially stronger than forward checking. Here, "arc" refers to a directed arc in the constraint graph, such as the arc from SA to NSW. Given the current domains of SA and NSW, the arc is consistent if, for every value x of SA, there is some value y of NSW that is consistent with x.

- One method of constraint propagation is to enforce arc consistency
  - Stronger than forward checking
  - Fast
- Arc refers to a directed arc in the constraint graph

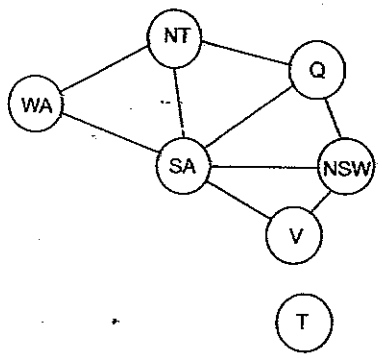


Fig. 1.47. Australian Territories

- Consider two nodes in the constraint graph (e.g. SA and NSW)
  - An arc is consistent if
  - For every value x of SA
  - There is some value y of NSW that is consistent with x
- Examine arcs for consistency in both directions

K-Consistency

- Can define stronger forms of consistency

k-Consistency

A CSP is k – consistent if, for any consistent assignment to $k-1$ variables there is a consistent assignment for the k- th variable

- 1-consistency (node consistency)
  - Each variable by itself is consistent (has a non-empty domain)
- 2-consistency (arc consistency)
- 3-consistency (path consistency)
  - Any pair of adjacent variables can be extended to a third

TWO MARKS QUESTIONS WITH ANSWERS (PART - A)

1. ***Define Artificial Intelligence (AI).***

The study of how to make computers do things at which at the moment, people are better. •Systems that think like humans • Systems that act like humans • Systems that think rationally • Systems that act rationally.

2. ***Differentiate between Intelligence and Artificial Intelligence.***

Intelligence	Artificial Intelligence
It is a natural process.	It is programmed by humans.
It is actually hereditary.	It is not hereditary.
Knowledge is required for intelligence.	kb and electricity are required to generate output.
No human is an expert. we may get better solutions from other humans.	Expert systems are made that aggregate many person's experiences and ideas

3. ***Is AI a science, or is it engineering? Or neither or both? Explain.***

Artificial Intelligence is most certainly a science. But it would be nothing with engineering. Computer Scientists need somewhere to place their programs, such as computers, servers, robots, cars, etc. But without engineers, they would have no outlet to test their Artificial Intelligence on. Science and Engineering go hand in hand, they both benefit each other. While the engineers build the machines, the scientists are writing code for their AI.

4. ***Explain why problem formulation must follow goal formulation.***

Well goal formulation is used to steer the agent in the right direction, thus ignoring any redundant actions. Problem formulation must follow this because it is based on Goal Formulation. It is the process of deciding what actions and states to consider given a certain goal. A goal may be set in stone, but how you achieve it can vary. Usually, the most optimal way is chosen though.

5. ***Define Artificial Intelligence in terms of rational acting.***

A field of study that seeks to explain and emulate intelligent behaviors in terms of computational processes-Schalk off. The branch of computer science that is concerned with the automation of intelligent behaviour-Luger & Stubblefield.

6. ***Define Artificial in terms of rational thinking.***

The study of mental faculties through the use of computational models-Charniak & McDermott. The study of the computations that make it possible to perceive, reason and act-Winston.

7. ***What is meant by the Turing test?***

A Turing Test is a method of inquiry in artificial intelligence (AI) for determining whether or not a computer is capable of thinking like a human being. The test is named after Alan Turing. To conduct this test, we need two people and one machine. One person will be an interrogator (i.e.) a questioner, who will be asking questions to one person and one machine. Three of them will be in a separate room. The interrogator knows them just as A and B. so it has to identify which is the person and machine. The goal of the machine is to make Interrogator believe that it is the person's answer. If the machine succeeds by fooling the Interrogator, the machine acts like a human. Programming a computer to pass the Turing test is very difficult.

8. ***What are the capabilities, the computer should possess to pass the Turing test?***

- ❖ natural language processing -to enable it to communicate successfully in English
- ❖ knowledge representation - to store what it knows or hears
- ❖ automated reasoning - to use the stored information to answer questions and to draw new conclusions
- ❖ machine learning - to adapt to new circumstances and to detect and extrapolate patterns.

9. ***What is meant by Total Turing Test?***

The test which includes a video signal so that the interrogator can test the perceptual abilities of the machine is termed Total Turing test.

10. ***What are the capabilities computers needs to pass total Turing test?***

Computer vision - to perceive objects. Robotics - to manipulate objects and move about.

11. ***Define an agent.***

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.

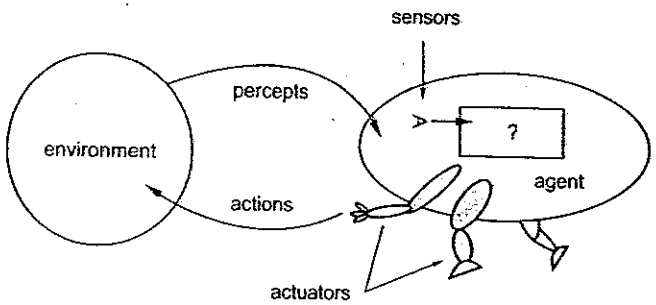


Fig. 1.48.

12. ***List the various type of agent pprograms***

- Simple reflex agent program.
- Agent that keeps track of the world.
- GGoal-based agent program.
- UUtility-based agent program

13. ***Define Agent Program.***

Agent function for an agent will be implemented by agent program

14. ***How to measure the performance of an agent?***

Performance measure of an agent is got by analyzing two tasks. They are How and When actions.

15. ***Define performance measures.***

Performance measure embodies the criterion for success of an agent's behaviour.

16. ***Define Ideal Rational Agent.***

For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built in knowledge the agent has.

17. ***List the steps involved in simple problem-solving technique.***

(*i*) Goal formulation
(*ii*) Problem formulation
(*iii*) Search
(*iv*) Solution
(*v*) Execution phase

18. ***What are the components of a problem?***

There are four components. They are

(*i*) Initial state
(*ii*) Successor function
(*iii*) Goal test
(*iv*) Path cost
(*v*) Operator
(*vi*) State-space
(*vii*) Path

19. ***Define State Space.***

The set of all possible states reachable from the initial state by any sequence of action is called state space.

20. ***Define Path.***

A path in the state space is a sequence of state connected by sequence of actions.

21. ***Define Path Cost.***

A function that assigns a numeric cost to each path, which is the sum of the cost of each action along the path.

22. ***Give example problems for Artificial Intelligence.***

(*i*) Toy problems
(*ii*) Real world problems

23. ***Give example for real world end toy problems.***

Real World Problem Examples:

(*i*) Airline travel problem.
(*ii*) Touring problem.
(*iii*) Traveling salesman problem.
(*iv*) VLSI Layout problem
(*v*) Robot navigation
(*vi*) Automatic Assembly
(*vii*) Internet searching

Toy Problem Examples:

Vacuum world problem.

8 – Queen problem

8 – Puzzle problem

24. ***Define search tree.***

The tree which is constructed for the search process over the state space is called search tree.

25. ***Define search node.***

The root of the search tree that is the initial state of the problem is called search node.

REVIEW QUESTIONS

1. Briefly explain the applications of AI.
2. List the types of agents and explain in detail.
3. Define the problem-solving agent and explain the types of problems.
4. Write down the "formulate, search, execute" design for the agent.
5. Draw the simplified Road Map of part of Romania.
6. Explain the 8-queen problem and write the solution.
7. Briefly explain the Travelling Salesman's Problem with finding the shortest path.
8. Define Fringe and write the function for the general search tree.
9. Explain the types of uninformed search strategies.
10. Explain the informed search strategies.
11. What are greedy search algorithms and explain with an example.
12. Explain local search algorithms and optimization problems.
13. Draw the Schematic game tree for a backgammon position and explain in detail.
14. Write an example of a Constraint Satisfaction Problem.

****

UNIT II

PROBABILISTIC REASONING

Acting under uncertainty – Bayesian inference – naïve Bayes models. Probabilistic reasoning – Bayesian networks – exact inference in BN – approximate inference in BN – causal network.

2.1. UNCERTAINTY

Till now, we have learned knowledge representation using first-order logic and propositional logic with certainty, which means we were sure about the predicates. With this knowledge representation, we might write A→ B, which means if A is true then B is true, but consider a situation where we are not sure about whether A is true or not then we cannot express this statement, this situation is called uncertainty.

So to represent uncertain knowledge, where we are not sure about the predicates, we need uncertain reasoning or probabilistic reasoning.

Causes of Uncertainty:

Following are some leading causes of uncertainty to occur in the real world.

- Information occurred from unreliable sources.
- Experimental Errors
- Equipment fault
- Temperature variation
- Climate change.

2.2. PROBABILISTIC REASONING

Probabilistic reasoning is a way of knowledge representation where we apply the concept of probability to indicate the uncertainty in knowledge. In probabilistic reasoning, we combine probability theory with logic to handle the uncertainty.

We use probability in probabilistic reasoning because it provides a way to handle the uncertainty that is the result of someone's laziness and ignorance.

In the real world, there are lots of scenarios, where the certainty of something is not confirmed, such as "It will rain today," "behavior of someone for some situations," "A match between two teams or two players." These are probable sentences for which we can assume that it will happen but not sure about it, so here we use probabilistic reasoning.

Need of probabilistic reasoning in AI:

- When there are unpredictable outcomes.
- When specifications or possibilities of predicates becomes too large to handle.
- When an unknown error occurs during an experiment.
- In probabilistic reasoning, there are two ways to solve problems with uncertain knowledge:
- Bayes' rule
- Bayesian Statistics

As probabilistic reasoning uses probability and related terms, so before understanding probabilistic reasoning, let's understand some common terms:

Probability:

Probability can be defined as a chance that an uncertain event will occur. It is the numerical measure of the likelihood that an event will occur. The value of probability always remains between 0 and 1 that represent ideal uncertainties.

- $0 \leq P(A) \leq 1$, where P(A) is the probability of an event A.
- P(A) = 0, indicates total uncertainty in an event A.
- P(A) =1, indicates total certainty in an event A.

We can find the probability of an uncertain event by using the below formula.

$$\text{Probability of Occurrence} = \frac{\text{Number of desired outcomes}}{\text{Total number of outcomes}}$$

- $P(\neg A)$ = probability of a not happening event.
- $P(\neg A) + P(A) = 1$.

Event:

Each possible outcome of a variable is called an event.

Sample Space:

The collection of all possible events is called sample space.

Random Variables:

Random variables are used to represent the events and objects in the real world.

Prior Probability:

The prior probability of an event is probability computed before observing new information.

Posterior Probability:

The probability that is calculated after all evidence or information has taken into account. It is a combination of prior probability and new information.

Conditional Probability:

Conditional probability is a probability of occurring an event when another event has already happened. Let's suppose, we want to calculate the event A when event B has already occurred, "the probability of A under the conditions of B", it can be written as:

$$P(A \mid B) = \frac{P(A \wedge B)}{P(B)}$$

Where $P(A \wedge B)$ = Joint probability of a and B

P(B) = Marginal probability of B.

If the probability of A is given and we need to find the probability of B, then it will be given as:

$$P(B \mid A) = \frac{P(A \wedge B)}{P(A)}$$

It can be explained by using the below Venn diagram, where B is occurred event, so sample space will be reduced to set B, and now we can only calculate event A when event B is already occurred by dividing the probability of $P(A \wedge B)$ by P(B).

Example:

In a class, there are 70% of the students who like English and 40% of the students who likes English and mathematics, and then what is the percent of students those who like English also like mathematics?

Solution::

Let, A is an event that a student likes Mathematics

B is an event that a student likes English.

$$P(A \mid B) = \frac{P(A \wedge B)}{P(B)} = \frac{0.4}{0.7} = 57\%$$

Hence, 57% are the students who like English also like Mathematics.

2.3. BAYES' THEOREM IN ARTIFICIAL INTELLIGENCE

Bayes' Theorem:

- Bayes' theorem is also known as Bayes' rule, Bayes' law, or Bayesian reasoning, which determines the probability of an event with uncertain knowledge. In probability theory, it relates the conditional probability and marginal probabilities of two random events.
- Bayes' theorem was named after the British mathematician Thomas Bayes. The Bayesian inference is an application of Bayes' theorem, which is fundamental to Bayesian statistics.
- It is a way to calculate the value of P(B|A) with the knowledge of P(A | B).
- Bayes' theorem allows updating the probability prediction of an event by observing new information of the real world.

Example: If cancer corresponds to one's age then by using Bayes' theorem, we can determine the probability of cancer more accurately with the help of age. Bayes' theorem can be derived using product rule and conditional probability of event A with known event B:

As from product rule we can write:

$$P(A \wedge B) = P(A|B)\, P(B) \text{ or}$$

Similarly, the probability of event B with known event A:

$$P(A \wedge B) = P(B|A)\, P(A)$$

Equating right hand side of both the equations, we will get:

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)} \quad \ldots(1)$$

The above equation (*a*) is called as Bayes' rule or Bayes' theorem. This equation is basic of most modern AI systems for probabilistic inference. It shows the simple relationship between joint and conditional probabilities. Here,

- P(A|B) is known as posterior, which we need to calculate, and it will be read as Probability of hypothesis A when we have occurred an evidence B.
- P(B|A) is called the likelihood, in which we consider that hypothesis is true, then we calculate the probability of evidence.
- P(A) is called the prior probability, probability of hypothesis before considering the evidence
- P(B) is called marginal probability, pure probability of an evidence.

In the equation (1), in general, we can write P (B) = P(A) * P(B | A_i), hence the Bayes' rule can be written as:

$$P(A_i \mid B) = \frac{P(A_i) * P(B \mid A_i)}{\sum_{i=1}^{k} P(A_i) * P(B \mid A_i)}$$

Where A_1 , A_2, A_3 ,....., A_n is a set of mutually exclusive and exhaustive events.

Applying Bayes' rule:

Bayes' rule allows us to compute the single term P(B | A) in terms of P(A | B), P(B), and P(A). This is very useful in cases where we have a good probability of these three terms and want to determine the fourth one. Suppose we want to perceive the effect of some unknown cause, and want to compute that cause, then the Bayes' rule becomes.

$$P(\text{cause} \mid \text{effect}) = \frac{P(\text{effect} \mid \text{cause})\, P(\text{cause})}{P(\text{effect})}$$

Example-1:

Question: what is the probability that a patient has diseases meningitis with a stiff neck?

Given Data:

A doctor is aware that disease meningitis causes a patient to have a stiff neck, and it occurs 80% of the time. He is also aware of some more facts, which are given as follows:

- The Known probability that a patient has meningitis disease is 1/30,000.
- The Known probability that a patient has a stiff neck is 2%.

Let a be the proposition that patient has stiff neck and b be the proposition that patient has meningitis. , so we can calculate the following as:

$$P(a \mid b) = 0.8$$

$$P(b) = 1/30000$$

$$P(a) = .02$$

$$P(b \mid a) = \frac{P(a \mid b)\,P(b)}{P(a)}$$

$$= \frac{0.8 * \left(\frac{1}{30000}\right)}{0.02}$$

$$= 0.001333333$$

Hence, we can assume that 1 patient out of 750 patients has meningitis disease with a stiff neck.

Example-2:

Question: From a standard deck of playing cards, a single card is drawn. The probability that the card is king is 4/52, then calculate posterior probability P(King|Face), which means the drawn face card is a king card.

Solution:

$$P(\text{king} \mid \text{face}) = \frac{P(\text{Face} \mid \text{King}) * P(\text{King})}{P(\text{Face})} \quad \ldots(i)$$

P(king): probability that the card is King = 4/52 = 1/13

P(face): probability that a card is a face card = 3/13

P(Face|King): probability of face card when we assume it is a king = 1

Putting all values in equation (*i*) we will get:

$$P(\text{king} \mid \text{face}) = \frac{1 * \left(\frac{1}{13}\right)}{\left(\frac{3}{13}\right)} = \frac{1}{3},$$

It is a probability that a face card is a king card.

Application of Bayes' theorem in Artificial intelligence:

Following are some applications of Bayes' theorem:

- It is used to calculate the next step of the robot when the already executed step is given.
- Bayes' theorem is helpful in weather forecasting.
- It can solve the Monty Hall problem.

2.4. BAYESIAN NETWORK

A Bayesian network is a directed graph in which each node is annotated with quantitative probability information. The full specification is as follows:

- Each node corresponds to a random variable, which may be discrete or continuous.
- A set of directed links or arrows connects pairs of nodes. If there is an arrow from node X to node Y, X is said to be a parent of Y. The graph has no directed cycles (and hence is a directed acyclic graph, or DAG.
- Each node X_i has a conditional probability distribution P (X_i) Parents (X_i) that quantifies the effect of the parents on the node.
- The topology of the network—the set of nodes and links - specifies the conditional independence relationships that hold in the domain, in a way that will be made precise shortly.
- The intuitive meaning of an arrow is typically that X has a direct influence on Y, which suggests that causes should be parents of effects.
- Once the topology of the Bayesian network is laid out, we need only specify a conditional probability distribution for each variable, given its parents.

Example

Now consider the following example, which is just a little more complex. You have a new burglar alarm installed at home. It is fairly reliable at detecting a burglary, but also responds on occasion to minor earthquakes. You also have two neighbor, John and Mary, who have promised to call you at work when they hear the alarm. John nearly always calls when he hears the alarm, but sometimes confuses the

telephone ringing with the alarm and calls then, too. Mary, on the other hand, likes rather loud music and often misses the alarm altogether. Given the evidence of who has or has not called, we would like to estimate the probability of a burglary

A simple Bayesian network in which Weather is independent of the other three variables and Toothache and Catch are conditionally independent, given Cavity.

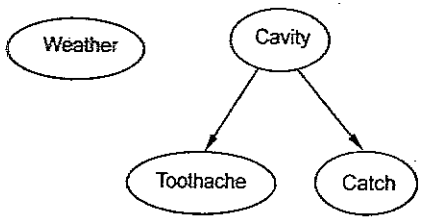


Fig. 2.1. A simple Bayesian network

A Bayesian network for this domain appears in Figure 2.2. The network structure shows that burglary and earthquakes directly affect the probability of the alarm's going off, but whether John and Mary call depends only on the alarm. The network thus represents our assumptions that they do not perceive burglaries directly, they do not notice minor earthquakes, and they do not confer before calling.

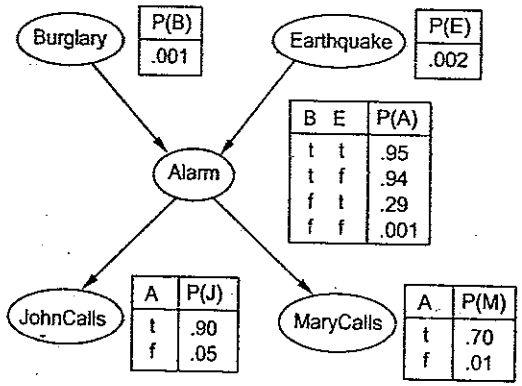


Fig. 2.2. A typical Bayesian network, showing both the topology and the conditional probability tables (CPTs). In the CPTs, the letters B, E, A, J, and M stand for Burglary, Earthquake, Alarm, JohnCalls, and MaryCalls , respectively

Conditional Probability Table

The conditional distributions in Figure 2.2 are shown as a conditional probability table, or CPT.

Conditioning Case

- Each row in a CPT contains the conditional probability of each node value for a conditioning case.
- A conditioning case is just a possible combination of values for the parent nodes - a miniature possible world, if you like.
- Each row must sum to 1, because the entries represent an exhaustive set of cases for the variable.
- For Boolean variables, once you know that the probability of a true value is p, the probability of false must be $1 - p$, so we often omit the second number, as in Figure 2.2.
- In general, a table for a Boolean variable with k Boolean parents contains 2^k independently specifiable probabilities.
- A node with no parents has only one row, representing the prior probabilities of each possible value of the variable

2.4.1. THE SEMANTICS OF BAYESIAN NETWORKS

There are two ways in which one can understand the semantics of Bayesian networks.

1. The first is to see the network as a representation of the joint probability distribution.
2. The second is to view it as an encoding of a collection of conditional independence statements.

The two views are equivalent, but the first turns out to be helpful in understanding how to construct networks, whereas the second is helpful in designing inference procedures.

Representing the Full Joint Distribution:

- A Bayesian network is a directed acyclic graph with some numeric parameters attached to each node.
- The semantics is to define the way in which it represents a specific joint distribution over all the variables
- Parameters correspond to conditional probabilities :

$$P(X_i \mid \text{Parents}(X_i));$$

- but until we assign semantics to the network as a whole, we should think of them just as numbers

$$\theta(X_i \mid \text{Parents}(X_i)).$$

A generic entry in the joint distribution is the probability of a conjunction of particular assignments to each variable, such as $P(X_1 = x_1 \wedge \ldots \wedge X_n = x_n)$. We use the notation $P(x_1, \ldots, x_n)$ as an abbreviation for this. The value of this entry is given by the formula,

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} \theta(x_i \mid \text{parents}(X_i))$$

where parents(X_i) denotes the values of P arents(X_i) that appear in $x_1, \ldots x_n$.

- Thus, each entry in the joint distribution is represented by the product of the appropriate elements of the conditional probability tables (CPTs) in the Bayesian network.
- From this definition, it is easy to prove that the parameters $\theta(X_i \mid \text{Parents}(X_i))$ are exactly the conditional probabilities $P(X_i \mid \text{Parents}(X_i))$ implied by the joint distribution

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid \text{parents}(X_i))$$

Example

We explain it, by calculating the probability that the alarm has sounded, but neither the burgulary nor an earthquake has occurred, and both Arisha and Bobby telephone you.

$$P(j, m, a, -b, -e) = P(j \mid a)P(m \mid a)\,P(a \mid -b \wedge -e)\,P(-b)\,P(-e)$$
$$= 0.90 \times 0.70 \times 0.001 \times 0.999 \times 0.998 = 0.000628$$

2.4.2. CONSTRUCTING BAYESIAN NETWORKS

We rewrite the entries in the joint distribution in terms of conditional probability, using the product rule.

$$P(x_1, \ldots, x_n) = P(x_n \mid x_{n-1}, \ldots, x_1)\,P(x_{n-1}, \ldots, x_1)$$

Then we repeat the process, reducing each conjunctive probability to a conditional probability and a smaller conjunction. We end up with one big product,

$$P(x_1, \ldots, x_n) = P(x_n \mid x_{n-1}, \ldots, x_1)\,P(x_{n-1} \mid x_{n-2}, \ldots, x_1) \ldots P(x_2 \mid x_1)P(x_1)$$
$$= \prod_{i=1}^{n} P(x_i \mid x_{i-1}, \ldots, x_1)$$

This identity is called the **chain rule**. The specification of the joint distribution is equivalent to the general assertion that, for every variable X_i in the network.

$$P(X_i \mid X_{i-1}, \ldots, X_1) = P(X_i \mid \text{Parents}(X_i)),$$

provided that $\text{Parents}(X_i) \subseteq \{X_{i-1}, \ldots, X_1\}$. This last condition is satisfied by numbering the nodes in a way that is consistent with the partial order implicit in the graph structure.

The Bayesian network is a correct representation of the domain only if each node is conditionally independent of its other predecessors in the node ordering, given its parents. We can satisfy this condition with this methodology:

1. Nodes:

First determine the set of variables that are required to model the domain. Now order them, $\{X_1, \ldots, X_n\}$

2. Links:

- For $i = 1$ to n do:
- Choose, from $X_1, \ldots, X_i - 1$, a minimal set of parents for X_i, such that Equation is satisfied.
- For each parent insert a link from the parent to X_i.
- CPTs: Write down the conditional probability table, $P(X_i \mid \text{Parents}(X_i))$.

2.4.3. COMPACTNESS AND NODE ORDERING

- As well as being a complete and nonredundant representation of the domain, a Bayesian network can often be far more compact than the full joint distribution.
- This property is what makes it feasible to handle domains with many variables.
- The compactness of Bayesian networks is an example of a general property of locally structured (also called sparse) systems.

- In a locally structured system, each subcomponent interacts directly with only a bounded number of other components, regardless of the total number of components.
- Local structure is usually associated with linear rather than exponential growth in complexity.
- In the case of Bayesian networks, it is reasonable to suppose that in most domains each random variable is directly influenced by at most k others, for some constant k.

If we assume n Boolean variables for simplicity, then the amount of information needed to specify each conditional probability table will be at most 2^k numbers, and the complete network can be specified by $n2^k$ numbers. In contrast, the joint distribution contains 2^n numbers. To make this concrete, suppose we have $n = 30$ nodes, each with five parents ($k = 5$). Then the Bayesian network requires 960 numbers, but the full joint distribution requires over a billion.

- There are domains in which each variable can be influenced directly by all the others so that the network is fully connected.
- Then specifying the conditional probability tables requires the same amount of information as specifying the joint distribution.
- In some domains, there will be slight dependencies that should strictly be included by adding a new link. But if these dependencies are tenuous, then it may not be worth the additional complexity in the network for the small gain in accuracy

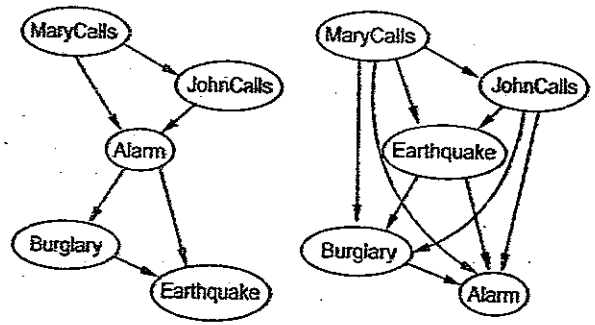


Fig. 2.3. Network structure depends on order of introduction. In each network, we have introduced nodes in top-to-bottom order.

For example, one might object to our burglary network on the grounds that if there is an earthquake, then John and Mary would not call even if they heard the alarm, because they assume that the earthquake is the cause. Whether to add the link from **Earthquake** to **JohnCalls** and **MaryCalls** (and thus enlarge the tables) depends on comparing the importance of getting more accurate probabilities with the cost of specifying the extra information.

Suppose we decide to add the nodes in the order MaryCalls, JohnCalls, Alarm, Burglary, Earthquake. We then get the somewhat more complicated network shown in Figure 2.3(a). The process goes as follows:

- **Adding MaryCalls:** No parents.
- **Adding JohnCalls:** If Mary calls, that probably means the alarm has gone off, which of course would make it more likely that John calls. Therefore, JohnCalls needsMaryCalls as a parent.
- **Adding Alarm:** Clearly, if both call, it is more likely that the alarm has gone off than if just one or neither calls, so we need both MaryCalls and JohnCalls as parents.
- **Adding Burglary:** If we know the alarm state, then the call from John or Mary mightgive us information about our phone ringing or Mary's music, but not about burglary:

P(Burglary | Alarm, JohnCalls , MaryCalls) = P(Burglary | Alarm)

Hence we need just Alarm as parent.

- **Adding Earthquake:** If the alarm is on, it is more likely that there has been an earthquake. (The alarm is an earthquake detector of sorts.) But if we know that there has been a burglary, then that explains the alarm, and the probability of an earthquake would be only slightly above normal. Hence, we need both Alarm and Burglary as parents

Figure 2.3 (b) shows a very bad node ordering: MaryCalls, JohnCalls, Earthquake, Burglary, Alarm. This network requires 31 distinct probabilities to be specified—exactly the same number as the full joint distribution. It is important to realize, however, that any of the three networks can represent exactly the same joint distribution. The last two versions simply fail to represent all the conditional independence relationships and hence end up specifying a lot of unnecessary numbers instead.

2.4.4. CONDITIONAL INDEPENDENCE RELATIONS IN BAYESIAN NETWORKS

- Using "numerical" semantics to derive a method for constructing Bayesian networks, we were led to the consequence that a node is conditionally independent of its other predecessors, given its parents.
- It turns out that we can also go in the other direction. We can start from a "topological" semantics that specifies the conditional independence relationships encoded by the graph structure, and from this we can derive the "numerical" semantics.
- The topological semantics specifies that each variable is conditionally independent of its non-descendants, given its parents.
- For example, in Figure 2.2, JohnCalls is independent of Burglary, Earthquake, and MaryCalls given the value of Alarm. The definition is illustrated in Figure 2.4 (*a*). In this sense, the "numerical" semantics and the "topological" semantics are equivalent.
- Another important independence property is implied by the topological semantics: a node is conditionally independent of all other nodes in the network, given its parents, children and children's parents—that is, given its Markov blanket.

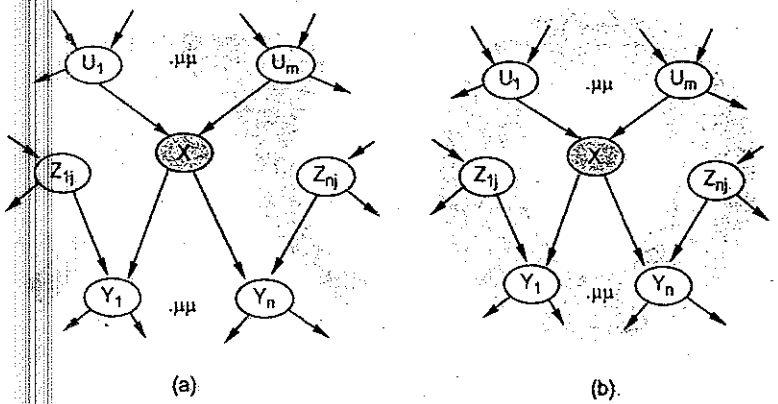


Fig. 2.4. (a) A node X is conditionally independent of its non-descendants (e.g., the Z_{ij}s) given its parents (the U_is shown in the gray area). (b) A node X is conditionally independent of all other nodes in the network given its Markov blanket (the gray area).

2.5. EXACT INFERENCE IN BAYESIAN NETWORKS

The basic task for any probabilistic inference system is to compute the posterior probability distribution for a set of query variables, given some observed event - that is, some assignment of values to a set of evidence variables.

X denotes the query variable; E denotes the set of evidence variables $E_1, ..., E_m$, and e is a particular observed event; Y will denote the nonevidence, non-query variables $Y_1, ..., Y_l$ (called the hidden variables). Thus, the complete set of variables is $X = \{X\} \cup E \cup Y$. A typical query asks for the posterior probability distribution $P(X \mid e)$.

In the burglary network, we might observe the event in which *JohnCalls = true* and *MaryCalls = true*. We could then ask for, say, the probability that a burglary has occurred:

P(Burglary | JoinCalls = true, MaryCalls = true) = (0.284, 0.716)

2.5.1. INFERENCE BY ENUMERATION

Any conditional probability can be computed by summing terms from the full joint distribution.

A query $P(X \mid e)$,

$$P(X \mid e) = \alpha P(X, e) = \alpha \sum_{y} P(X, e, y)$$

Now, a Bayesian network gives a complete representation of the full joint distribution. The terms P(x, e, y) in the joint distribution can be written as products of conditional probabilities from the network. Therefore, a query can be answered using a Bayesian network by computing sums of products of conditional probabilities from the network.

Consider the query P(Burglary | JohnCalls = true, MaryCalls = true). The hidden variables for this query are Earthquake and Alarm.

$$P(B \mid j, m) = \alpha P(B, j, m) = \alpha \sum_{e} \sum_{a} P(B, j, m, e, a)$$

The semantics of Bayesian networks then gives us an expression in terms of CPT entries. For simplicity, we do this just for Burglary = true:

$$P(B \mid j, m) = \alpha \sum_{e} \sum_{a} P(b)\, P(e)\, P(a \mid b, e)\, P(j \mid a)\, P(m \mid a)$$

To compute this expression, we have to add four terms, each computed by multiplying five numbers. In the worst case, where we have to sum out almost all the variables, the complexity of the algorithm for a network with n Boolean variables is $O(n\,2^n)$.

An improvement can be obtained from the following simple observations: the $P(b)$ term is a constant and can be moved outside the summations over a and e, and the $P(e)$ term can be moved outside the summation over a. Hence, we have,

$$P(b \mid j, m) = \alpha\, P(b) \sum_e P(e) \sum_a P(a \mid b, e)\, P(j \mid a)\, P(m \mid a)$$

Using the numbers from Figure 2.2, we obtain $P(b \mid j, m) = \alpha \times 0.00059224$. The corresponding computation for $\neg b$ yields $\alpha \times 0.0014919$; hence,

$$P(B \mid j, m) = \alpha\ \langle 0.00059224, 0.0014919 \rangle \approx \langle 0.284, 0.716 \rangle$$

That is, the chance of burglary, given calls from both neighbors, is about 28%

2.5.2. THE VARIABLE ELIMINATION ALGORITHM

The enumeration algorithm can be improved substantially by eliminating repeated calculations. The idea is simple:

- do the calculation once and save the results for later use.
- This is a form of dynamic programming. There are several versions of this approach; we present the variable elimination algorithm, which is the simplest.
- Variable elimination works by evaluating expressions in right-to-left order (Figure 2.5)
- Intermediate results are stored, and summations over each variable are done only for those portions of the expression that depend on the variable

Let us illustrate this process for the burglary network. We evaluate the expression,

$$P(A \mid j, m) = \alpha \underbrace{P(B)}_{f_1(B)} \sum_e \underbrace{P(e)}_{f_2(E)} \sum_a \underbrace{P(a \mid B, e)}_{f_3(A,B,E)} \underbrace{P(j \mid a)}_{f_4(A)} \underbrace{P(m \mid a)}_{f_5(A)}$$

We have annotated each part of the expression with the name of the corresponding factor; each factor is a matrix indexed by the values of its argument variables. For example, the factors $f_4(A)$ and $f_5(A)$ corresponding to $P(j \mid a)$ and $P(m \mid a)$ depend

just on A because J and M are fixed by the query. They are therefore two-element vectors.

$$f_4(A) = \begin{pmatrix} P(j \mid a) \\ P(j \mid \neg a) \end{pmatrix} = \begin{pmatrix} 0.90 \\ 0.05 \end{pmatrix}$$

$$f_5(A) = \begin{pmatrix} P(m \mid a) \\ P(m \mid \neg a) \end{pmatrix} = \begin{pmatrix} 0.70 \\ 0.01 \end{pmatrix}$$

$f_3(A, B, E)$ will be a $2 \times 2 \times 2$ matrix, which is hard to show on the printed page. (The "first" element is given by $P(a \mid b, e) = 0.95$ and the "last" by $P(a \mid b, e) = 0.999$.) In terms of factors, the query expression is written as

$$P(B \mid j, m) = \alpha f_1(B) \times \sum_e f_2(E) \times \sum_a f_3(A, B, E) \times f_4(A) \times f_5(A)$$

where the " $\times$" operator is not ordinary matrix multiplication but instead the pointwise product operation, to be described shortly.

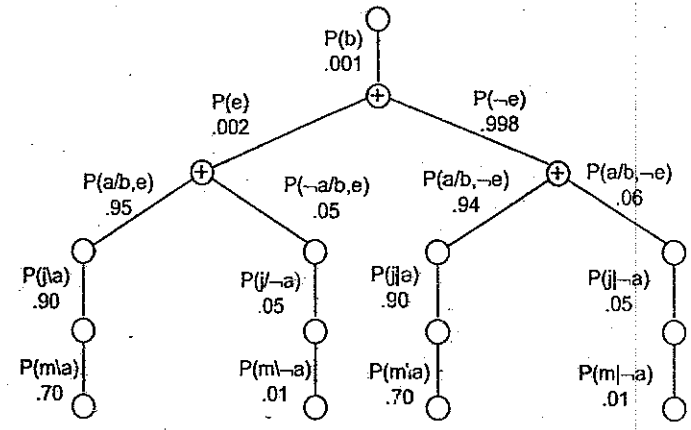


Fig. 2.5. The evaluation proceeds top-down, multiplying values along each path and summing at the "+" nodes. Notice the repetition of the paths for j and m.

function ENUMERATION-ASK(X, e, bn) returns a distribution over X

inputs: X, the query variable

e, observed values for variables E

bn, a Bayes net with variables {X} ∪ E ∪ Y /* Y = hidden variables */

Q(X) ← a distribution over X, initially empty

```
    for each value x_i of X do
        Q(x_i) ← ENUMERATE-ALL (bn.VARS, e_x_i)
            where e_x_i is e extended with X = x_i
    return NORMALIZE(Q(X))

function ENUMERATE-ALL(vars, e) returns a real number
    if EMPTY?(vars) then return 1.0
    Y ← FIRST(vars)
    if Y has value y in e
        then return P(y) | parents (Y)) × ENUMERATE-ALL(REST(vars).e)
        else return Σ_y P(y) | parents(Y)) × ENUMERATE-ALL REST(vars), e_y)
            where e_y is e extended with Y = y
```

Fig. 2.6. The enumeration algorithm for answering queries on Bayesian networks

The process of evaluation is a process of summing out variables (right to left) from pointwise products of factors to produce new factors, eventually yielding a factor that is the solution, i.e., the posterior distribution over the query variable. The steps are as follows:

- First, we sum out A from the product of f_3, f_4, and f_5. This gives us a new 2×2 factor f_6 (B,E) whose indices range over just B and E:

$$f_6(\text{B,E}) = \sum_a f_3(\text{A, B, E}) \times f_4(\text{A}) \times f_5(\text{A})$$

$$= (f_3(a, \text{B,E}) \times f_4(a) \times f_5(a)) + (f_3(\neg a, \text{B, E}) \times f_4(\neg a) \times f_5(\neg a))$$

Now we are left with the expression

$$\text{P(B} \mid j, m) = \alpha f_1(\text{B}) \times \sum f_2(\text{E}) \times f_6(\text{B,E})$$

Next, we sum out E from the product of f_2 and f_6:

$$f_7(\text{B}) = \sum_e f_2(\text{E}) \times f_6(\text{B,E})$$

$$= f_2(e) \times f_6(\text{B}, e) + f_2(\neg e) \times f_6(\text{B}, \neg e)$$

This leaves the expression

$$\text{P(B} \mid j, m) = \alpha f_1(\text{B}) \times f_7(\text{B})$$

which can be evaluated by taking the pointwise product and normalizing the result. Examining this sequence, we see that two basic computational operations are required:

1. pointwise product of a pair of factors,
2. summing out a variable from a product of factors.

Operations on Factors

- The pointwise product of two factors f_1 and f_2 yields a new factor f whose variables are the union of the variables in f_1 and f_2 and whose elements are given by the product of the corresponding elements in the two factors.
- Suppose the two factors have variables $Y_1, \ldots, Y_k$ in common. Then we have,

$$f(X_1 \ldots X_j, Y_1 \ldots Y_k, Z_1 \ldots Z_i) = f_1(X_1 \ldots X_j, Y_1 \ldots Y_k) f_2(Y_1 \ldots Y_k, Z, \ldots Z_i$$

- If all the variables are binary, then f_1 and f_2 have 2^{j+k} and 2^{k+1} entries, respectively, and the pointwise product has 2^{j+k+1} entries.
- For example, given two factors f_1(A, B) and f_2(B, C), the pointwise product $f_1 \times f_2 = f_3$(A, B, C) has $2^{1+1+1} = 8$ entries, as illustrated in Figure 2.7.

A	B	f_1(A, B)	B	C	f_2(B,C)	A	B	C	f_3 (A, B, C)
T	T	.3	T	T	.2	T	T	T	.3 × .2 = .06
T	F	.7	T	F	.8	T	T	F	.3 × .8 = .24
F	T	.9	F	T	.6	T	F	T	.7 × .6 = .42
F	F	.1	F	F	.4	T	F	F	.7 × .4 = .28
						F	T	T	.9 × .2 = .18
						F	T	F	.9 × .8 = .72
						F	F	T	.1 × .6 = .06
						F	F	F	.1 × .4 = .04

Fig. 2.7. Illustrating pointwise multiplication: f_1(A, B) × f_2(B,C) = f_3(A, B, C).

- Summing out a variable from a product of factors is done by adding up the submatrices formed by fixing the variable to each of its values in turn.

For example, to sum out A from f_3(A, B, C), we write,

$$f(B,C) = \sum_a f_3(A,B,C) = f_3(a,B,C) + f_3(-a,B,C)$$

$$= \begin{pmatrix} .06 & .24 \\ .42 & .28 \end{pmatrix} + \begin{pmatrix} .18 & .72 \\ .06 & .04 \end{pmatrix} = \begin{pmatrix} .24 & .96 \\ .48 & .32 \end{pmatrix}$$

- If we were to sum out E first in the burglary network, the relevant part of the expression would be,

$$\sum_e f_2(E) \times f_3(A,B,E) \times f_4(A) \times f_5(A) = f_4(A) \times f_5(A) \times \sum_e f_2(E) \times f_3(A,B,E)$$

Now the pointwise product inside the summation is computed, and the variable is summed out of the resulting matrix.

Variable ordering and variable relevance

- The algorithm in Figure 2.8 includes an unspecified ORDER function to choose an ordering for the variables.
- Every choice of ordering yields a valid algorithm, but different orderings cause different intermediate factors to be generated during the calculation.

```
function ELIMINATION-ASK (X, e, bn) returns a distribution over X
  inputs: X, the query variable
          e, observed values for variables E
          bn, a Bayesian network specifying joint distribution P(X1, ... Xn)
  factors ←[ ]
  for each var in ORDER(bn, VARS) do
      factors ←[MAKE-FACTOR(var, e)|factors]
      if var is a hidden variable then factors ←SUM-OUT(var, factors)
  return NORMALIZE(POINTWISE-PRODUCT(factors))
```

Fig. 2.8. The variable elimination algorithm for inference in Bayesian networks.

For example, in the calculation shown previously, we eliminated A before E; if we do it the other way, the calculation become

$$P(B \mid j, m) = \alpha f_1(B) \times \sum_a f_4(A) \times f_5(A) \times \sum_e f_2(E) \times f_3(A,B,E)$$

during which a new factor f_6 (A, B) will be generated.

- In general, the time and space requirements of variable elimination are dominated by the size of the largest factor constructed during the operation of the algorithm.
- This in turn is determined by the order of elimination of variables and by the structure of the network.
- It turns out to be intractable to determine the optimal ordering, but several good heuristics are available.
- One fairly effective method is a greedy one: eliminate whichever variable minimizes the size of the next factor to be constructed

Let us consider one more query: P(JohnCalls | Burglary = true). As usual, the first step is to write out the nested summation,

$$P(J \mid b) = \alpha P(b) \sum_e P(c) \sum_a P(a \mid b, c) P(J \mid a) \sum_m P(m \mid a)$$

2.5.3. THE COMPLEXITY OF EXACT INFERENCE

- The complexity of exact inference in Bayesian networks depends strongly on the structure of the network.
- The burglary network belongs to the family of networks in which there is at most one undirected path between any two nodes in the network.
- These are singly connected networks or polytrees, and they have a particularly nice property: The time and space complexity of exact inference in polytrees is linear in the size of the network. Here, the size is defined as the number of CPT entries; if the number of parents of each node is bounded by a constant, then the complexity will also be linear in the number of nodes.
- For multiply connected networks, such as that of Figure 2.9(*a*), variable elimination can have exponential time and space complexity in the worst case, even when the number of parents per node is bounded.

- This is not surprising when one considers that because it includes inference in propositional logic as a special case, inference in Bayesian networks is NP-hard.

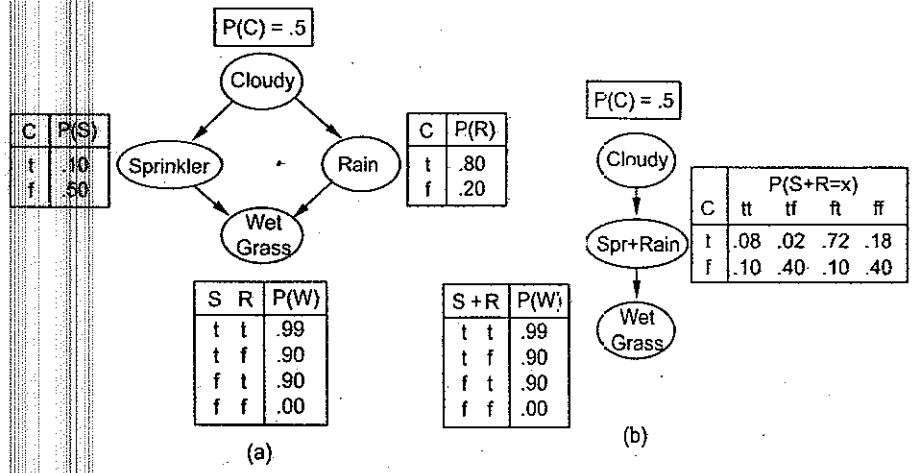


Fig. 2.9. (a) A multiply connected network with conditional probability tables. (b) Aclustered equivalent of the multiply connected network.

- The problem is as hard as that of computing the number of satisfying assignments for a propositional logic formula. This means that it **is #P-hard ("number -P hard")** - that is, strictly harder than NP-complete problems.
- There is a close connection between the complexity of Bayesian network inference and the complexity of constraint satisfaction problems (CSPs)
- The difficulty of solving a discrete CSP is related to how "treelike" its constraint graph is. Measures such as tree width, which bound the complexity of solving a CSP, can also be applied directly to Bayesian networks.
- Moreover, the variable elimination algorithm can be generalized to solve CSPs as well as Bayesian networks.

2.5.4. CLUSTERING ALGORITHMS

- The variable elimination algorithm is simple and efficient for answering individual queries. If we want to compute posterior probabilities for all the variables in a network, however, it can be less efficient.

- For example, in a polytree network, one would need to issue $O(n)$ queries costing $O(n)$ each, for a total of $O(n^2)$ time.
- Using clustering algorithms (also known as join tree algorithms), the time can be reduced to $O(n)$. For this reason, these algorithms are widely used in commercial Bayesian network tools.
- The basic idea of clustering is to join individual nodes of the network to form cluster nodes in such a way that the resulting network is a polytree.
- For example, the multiply connected network shown in Figure 2.8 (*a*) can be converted into a polytree by combining the Sprinkler and Rain node into a cluster node called Sprinkler + Rain, as shown in Figure 2.9 (*b*).
- The two Boolean nodes are replaced by a "meganode" that takes on four possible values: tt, tf, f t, and ff.
- The meganode has only one parent, the Boolean variable Cloudy, so there are two conditioning cases. Although this example doesn't show it, the process of clustering often produces meganodes that share some variables.
- Once the network is in polytree form, a special-purpose inference algorithm is required, because ordinary inference methods cannot handle meganodes that share variables with eachother

2.6. APPROXIMATE INFERENCE IN BAYESIAN NETWORKS

Monte Carlo algorithms, of which simulated annealing is an example, are used in many branches of science to estimate quantities that are difficult to calculate exactly. In this section, we are interested in sampling applied to the computation of posterior probabilities. We describe two families of algorithms: direct sampling and Markov chain sampling.

2.6.1. DIRECT SAMPLING METHODS

- The primitive element in any sampling algorithm is the generation of samples from a known probability distribution.
- For example, an unbiased coin can be thought of as a random variable Coin with values heads, tails and a prior distribution P(Coin) = 0.5, 0.5.

- Sampling from this distribution is exactly like flipping the coin: with probability 0.5 it will return heads, and with probability 0.5 it will return tails.
- Given a source of random numbers uniformly distributed in the range [0, 1], it is a simple matter to sample any distribution on a single variable, whether discrete or continuous.
- The simplest kind of random sampling process for Bayesian networks generates events from a network that has no evidence associated with it. The idea is to sample each variable in turn, in topological order.
- The probability distribution from which the value is sampled is conditioned on the values already assigned to the variable's parents.
- The algorithm is,

function PRIOR-SAMPLE(*bn*) returns an event sampled from the prior specified by *bn*

inputs: *bn*, a Bayesian network specifying joint distribution $P(X_1, ..., X_n)$

$x \leftarrow$ an event with n elements

for each variable X_i in $X_1 \ldots, X_n$ do

$x[i] \leftarrow$ a random sample from $P(X_i \mid \text{parents}(X_i))$

return x

Fig. 2.10. A sampling algorithm that generates events from a Bayesian network. Each variable is sampled according to the conditional distribution given the values already sampled for the variable's parents.

- assuming an ordering [Cloudy, Sprinkler, Rain, WetGrass]:
  1. Sample from P(Cloudy) = ⟨0.5, 0.5 ⟩$^{\langle 0.5, 0.5 \rangle}$, value is true.
  2. Sample from P(Sprinkler | Cloudy = true) = ⟨0.1, 0.9 ⟩$^{\langle 0.1, 0.9 \rangle}$, value is false.
  3. Sample from P(Rain | Cloudy = true) = ⟨0.8, 0.2⟩$^{\langle 0.8, 0.2 \rangle}$, value is true.
  4. Sample from P(WetGrass | Sprinkler = false, Rain = true) = ⟨0.9, 0.1⟩, value istrue.

In this case, PRIOR-SAMPLE returns the event [*true, false, true, true*]

- It is easy to see that PRIOR-SAMPLE generates samples from the prior joint distribution specified by the network.
- First, let SPS $(x_1, ..., x_n)$ be the probability that a specific event is generated by the PRIOR-SAMPLE algorithm. Just looking at the sampling process, we have,

$$S_{PS}(x_1 \ldots x_n) = \prod_{i=1}^{n} P(x_i \mid \text{parents}(X_i))$$

because each sampling step depends only on the parent values.

$S_{PS}(x_1 \ldots x_n) = P(x_1 \ldots x_n)$

- This simple fact makes it easy to answer questions by using samples. In any sampling algorithm, the answers are computed by counting the actual samples generated.
- Suppose there are N total samples, and let NPS $(x_1, \ldots x_n)$ be the number of times the specific event $x_1, \ldots x_n$ occurs in the set of samples. We expect this number, as a fraction of the total, to converge in the limit to its expected value according to the sampling probability:

$$\lim_{N \to \infty} \frac{N_{PS}(x_1, \ldots x_n)}{N} = S_{PS}(x_1, \ldots, x_n) = P(x_1, \ldots, x_n)$$

- For example, consider the event produced earlier: [true, false, true, true]. The sampling probability for this event is,

$S_{PS}(\text{true, false, true, true}) = 0.5 \times 0.9 \times 0.8 \times 0.9 = 0.324$

Hence, in the limit of large N, we expect 32.4% of the samples to be of this event.

- Whenever we use an approximate equality ("≈") in what follows, we mean it in exactly this sense - that the estimated probability becomes exact in the large-sample limit. Such an estimate is called **consistent**.
- For example, one can produce a consistent estimate of the probability of any partially specified event $x_1, \ldots, x_m$, where $m \leq n$, as follows:

$$P(x_1, \ldots, x_m) \approx N_{PS}(x_1, \ldots, x_m) / N$$

- That is, the probability of the event can be estimated as the fraction of all complete events generated by the sampling process that match the partially specified event.
- For example, if we generate 1000 samples from the sprinkler network, and 511 of them have Rain = true, then the estimated probability of rain, written as $\hat{P}$(Rain = true), is 0.511.

2.6.1.1. Rejection Sampling in Bayesian Networks

Rejection sampling is a general method for producing samples from a hard-to-sample distribution given an easy-to-sample distribution. In its simplest form, it can be used to compute conditional probabilities - that is, to determine P (X | *e*). The REJECTION-SAMPLING algorithm is,

function REJECTION-SAMPLING(X, *e*, *bn*, N) returns an estimate of P(X | *e*)
  inputs: X, the query variable
    e, observed values for variables E
    bn, a Bayesian network
    N, the total number of samples to be generated
  local variables: N, a vector of counts for each value of X, initially zero
  for *j* = 1 to N do
    x ← PRIOR SAMPLE(*bn*)
    if *x* is consistent with *e* then
      N[*x*] ←N[*x*] + 1 where *x* is the value of X in *x*
  return NORMALIZATION(N)

Fig. 2.11. The rejection-sampling algorithm for answering queries given evidence in a Bayesian network

- First, it generates samples from the prior distribution specified by the network. Then, it rejects all those that do not match the evidence.
- Finally, the estimate $\hat{P}(X = x \mid e)$ is obtained by counting how often X = *x* occurs in the remaining samples. Let $\hat{P}(X \mid e)$ be the estimated distribution that the algorithm returns. From the definition of the algorithm, we have,

$$P(X \mid e) = \alpha N_{PS}(X, e) = \frac{N_{PS}(X, e)}{N_{PS}(e)}$$

$$P(X \mid e) \approx \frac{P(X, e)}{P(e)} = P(X \mid e)$$

That is, rejection sampling produces a consistent estimate of the true probability.

2.6.1.2. Likelihood Weighting

- Likelihood weighting avoids the inefficiency of rejection sampling by generating only events that are consistent with the evidence *e*.
- It is a particular instance of the general statistical technique of importance sampling, tailored for inference in Bayesian networks.
- We begin by describing how the algorithm works; then we show that it works correctly - that is, generates consistent probability estimates.
- LIKELIHOOD-WEIGHTING fixes the values for the evidence variables E and samples only the nonevidence variables.
- This guarantees that each event generated is consistent with the evidence. Not all events are equal, however.
- Before tallying the counts in the distribution for the query variable, each event is weighted by the likelihood that the event accords to the evidence, as measured by the product of the conditional probabilities for each evidence variable, given its parents.
- Intuitively, events in which the actual evidence appears unlikely should be given less weight.
- Let us apply the algorithm to the network, shown in Figure 2.9, with the query P(Rain | Cloudy = true, WetGrass = true) and the ordering Cloudy, Sprinkler, Rain, WetGrass. (Any topological ordering will do.) The process goes as follows: First, the weight w is set to 1.0. Then an event is generated:
  1. Cloudy is an evidence variable with value true. Therefore, we set $w \leftarrow w \times P(\text{Cloudy} = \text{true}) = 0.5$.
  2. Sprinkler is not an evidence variable, so sample from P(Sprinkler | Cloudy = true) = (0.1,0.9) suppose this returns false.

3. Similarly, sample from P(Rain | Cloudy = true) (0.8, 0.2); suppose this returns true.
4. WetGrass is an evidence variable with value true. Therefore, we set $w \leftarrow w \times P(\text{WetGrass} = \text{true} \mid \text{Sprinkler} = \text{false}, \text{Rain} = \text{true}) = 0.45$.

- Here WEIGHTED-SAMPLE returns the event [true, false, true, true] with weight 0.45, and this is tallied under Rain = true.
- To understand why likelihood weighting works, we start by examining the sampling probability SWS for WEIGHTED-SAMPLE. The evidence variables E are fixed with values e. We call the nonevidence variables Z (including the query variable X). The algorithm samples each variable in Z given its parent values:

$$S_{WS}(z, e) = \prod_{i=1}^{l} P(z_i \mid \text{parents}(Z_i))$$

- The likelihood weight w makes up for the difference between the actual and desired sampling distributions.
- The weight for a given sample x, composed from z and e, is the product of the likelihoods for each evidence variable given its parents (some or all of which may be among the Z_i):

$$w(z, e) = \prod_{i=1}^{m} P(e_i \mid \text{parents}(E_i))$$

The weighted probability of a sample has the particularly convenient for,

$$S_{WS}(z, e)\, w(z, e) = \prod_{i=1}^{l} P(z_i \mid \text{parents}(Z_i)) \prod_{i=1}^{m} P(e_i \mid \text{parents}(E_i))$$
$$= P(z, e)$$

Now it is easy to show that likelihood weighting estimates are consistent. For any particular value x of X, the estimated posterior probability can be calculated as follows,

$$P(x, \mid e) = \alpha \sum_{y} N_{WS}(x, y, e)\, w(x, y, e)$$
$$\approx \alpha' \sum_{y} S_{WS}(x, y, e)\, w(x, y, e)$$

$$= \alpha' \sum_{y} P(x, y, e)$$
$$= \alpha' P(x, e) = P(x \mid e)$$

Hence, likelihood weighting returns consistent estimates

2.6.2. INFERENCE BY MARKOV CHAIN SIMULATION

- Markov chain Monte Carlo (MCMC) algorithms work quite differently from rejection sampling and likelihood weighting.
- Instead of generating each sample from scratch, MCMC algorithms generate each sample by making a random change to the preceding sample.
- It is therefore helpful to think of an MCMC algorithm as being in a particular current state specifying a value for every variable and generating a next state by making random changes to current state.
- Here we describe a particular form of MCMC called Gibbs sampling, which is especially well suited for Bayesian networks.

2.6.2.1. Gibbs sampling in Bayesian networks

- The Gibbs sampling algorithm for Bayesian networks starts with an arbitrary state (with the evidence variables fixed at their observed values) and generates a next state by randomly sampling a value for one of the nonevidence variables X_i.
- The sampling for X_i is done conditioned on the current values of the variables in the Markov blanket of X_i.
- The algorithm therefore wanders randomly around the state space—the space of possible complete assignments - flipping one variable at a time, but keeping the evidence variables fixed.
- Consider the query P(Rain | Sprinkler = true, WetGrass = true) applied to the network in Figure 2.9. The evidence variables Sprinkler and WetGrass are fixed to their observed values and the nonevidence variables Cloudy and Rain are initialized randomly - let us say to true and false respectively. Thus, the initial state is [true, true, false, true]. Now the nonevidence variables are sampled repeatedly in an arbitrary order.

- For example,
  1. Cloudy is sampled, given the current values of its Markov blanket variables: in this case, we sample from P(Cloudy | Sprinkler = true, Rain = false). Suppose the result is Cloudy = false. Then the new current state is [false, true, false, true].
  2. Rain is sampled, given the current values of its Markov blanket variables: in this case, we sample from P(Rain | Cloudy = false, Sprinkler = true, WetGrass = true). Suppose this yields Rain = true. The new current state is [false, true, true, true].
- Each state visited during this process is a sample that contributes to the estimate for the query variable Rain. If the process visits 20 states where Rain is true and 60 states where Rain is false, then the answer to the query is NORMALIZE (20, 60) = (0.25,0.75).
- The complete algorithm is,

function GIBBS-ASK(X, *e*, *bn*, N) returns an estimate of P(X | *e*)
  local variables: N, a vector of counts for each value of X, initially copied from *e*
    Z, the nonevidence variables in *bn*
    x, the current state of the network, initially copied from *e*
  initialize *x* with random values for the variables in Z
  for *j* = 1 to N **do**
    for each Z_i in Z **do**
      set the value of Z_i in *x* by sampling from P(Z_i | *mb*(Z_i))
      N[*x*] ← N[*x*] + 1 where *x* is the value of X in *x*
  return NORMALIZE(N)

Fig. 2.12. The Gibbs sampling algorithm for approximate inference in Bayesian networks; this version cycles through the variables, but choosing variables at random also works.

- Let $q(x \rightarrow x')$ be the probability that the process makes a transition from state x to state x'. This transition probability defines what is called a Markov chain on the state space.

- Now suppose that we run the Markov chain for t steps, and let $\pi_t(x)$ be the probability that the system is in state x at time t.
- Similarly, let $\pi_{t+1}(x')$ be the probability of being in state x' at time $t + 1$. Given $\pi_t(x)$, we can calculate $\pi_{t+1}(x')$ by summing, for all states the system could be in at time t, the probability of being in that state times the probability of making the transition to x' :

$$\pi_{t+1}(x') = \sum_x \pi_t(x)\, q(x \rightarrow x')$$

We say that the chain has reached its stationary distribution if $\pi_t = \pi_{t+1}$. Let us call this stationary distribution π; its defining equation is therefore,

$$\pi(x') = \sum_x \pi(x)\, q(x \rightarrow x') \qquad \text{for all } x'$$

Provided the transition probability distribution q is ergodic - that is, every state is reachable from every other and there are no strictly periodic cycles - there is exactly one distribution π satisfying this equation for any given q.

- The above equation can be read as saying that the expected "outflow" from each state (i.e., its current "population") is equal to the expected "inflow" from all the states.
- One obvious way to satisfy this relationship is if the expected flow between any pair of states is the same in both directions; that is

$$\pi(x)\, q(x \rightarrow x') = q(x' \rightarrow x) \qquad \text{for all } x, x'$$

When these equations hold, we say that $q(x \rightarrow x'$ DETAILED BALANCE) is in detailed balance with $\pi(x)$.

- We can show that detailed balance implies stationarity simply by summing over x. we have

$$\sum_x \pi(x)\, q(x \rightarrow x') = \sum_x \pi(x')\, q(x' \rightarrow x) = \pi(x') \sum_x q(x' \rightarrow x) = \pi(x')$$

where the last step follows because a transition from x' is guaranteed to occur.

- The transition probability $q(x \rightarrow x')$ defined by the sampling step in GIBBS-ASK is actually a special case of the more general definition of

Gibbs sampling, according to which each variable is sampled conditionally on the current values of all the other variables.

- We start by showing that this general definition of Gibbs sampling satisfies the detailed balance equation with a stationary distribution equal to $P(x \mid e)$, (the true posterior distribution on the nonevidence variables).
- Then, we simply observe that, for Bayesian networks, sampling conditionally on all variables is equivalent to sampling conditionally on the variable's Markov blanket.
- To analyze the general Gibbs sampler, which samples each X_i in turn with a transition probability q_i that conditions on all the other variables, we define X_i to be these other variables (except the evidence variables); their values in the current state are X_i.
- If we sample a new value x_i' for X_i conditionally on all the other variables, including the evidence, we have

$$q_i(x \to x') = q_i((x_i, \bar{x}_i) \to (x_i', \bar{x}_i)) = P(x_i' \mid \bar{x}_i, e)$$

Now we show that the transition probability for each step of the Gibbs sampler is in detailed balance with the true posterior:

$$\pi(x)\, q_i(x \to x') = P(x \mid e)\, P(x_i' \mid \bar{x}_i, e) = P(x_i, \bar{x}_i \mid e)\, P(x_i' \mid \bar{x}_i, e)$$

$= P(x_i \mid \bar{x}_i, e)\, P(\bar{x}_i \mid e) P(x_i' \mid \bar{x}_i, e)$ (using the chain rule on the first term)

$= P(x_i \mid \bar{x}_i, e)\, P(x_i', \bar{x}_i \mid e)$ (using the chain rule backward)

$= \pi(x')\, q_i(x' \to x)$

- The final step is to show how to perform the general Gibbs sampling step - sampling X_i from $P(X_i \mid x_i, e)$ -in a Bayesian network..

$$P(x_i' \mid \bar{x}_i, e) = P(x_i' \mid mb(X_i))$$

where $mb(X_i)$ denotes the values of the variables in X_i's Markov blanket, $MB(X_i)$.

- The probability of a variable given its Markov blanket is proportional to the probability of the variable given its parents times the probability of each child given its respective parents:

$$P(x_i' \mid mb(X_i)) = \alpha\, P(x_i' \mid \text{parents}(X_i)) \times \prod_{Yj \in Children(Xi)} P(y_j) \mid \text{parents}(Y_j))$$

2.6.3. CAUSAL NETWORKS

- Each model is a machine for turning inputs into outputs. Change the input and the output will change correspondingly.
- But this does not mean that nature works in the same way.
- Changing an input in the real world – administering polio vaccine, eating organic food, providing bed nets to the poor – may not cause the same change in the response variable as happens when you change the input to a model.
- Statisticians are careful to distinguish between two different interpretations of relationship: correlation and causal.
- Every successful prediction model Y ~ X is a demonstration that there is a correlation between the response Y and the explanatory variable X. But the performance of the model does not itself tell us that X causes Y in the real world.
- There are other possible configurations that will produce a correlation between X and Y. For instance, both X and Y may themselves have a common cause C without X being otherwise related to Y.
- In such a circumstance, a real-world intervention to change X will have no effect on Y. To put this in the form of a story, consider that the start of the school year and leaves changing color are correlated.
- But an intervention to start the school year in mid-winter will not result in leaves changing color. There's a common cause for the school year and colorful foliage that produces the relationship: the end of summer.
- Consider a simple network of causality involving three variables, generically called X, Y, and C. Always, we'll imagine that the modeler's interest is in anticipating how an intervention to change X will create to a change in Y. To accomplish this, the modeler has two basic choices for structuring a model, either

  Y ~ X, or

  Y ~ X + C.

- It's surprising to many people that models (1) and (2) can have utterly different, even contradictory implications for how a change in model input X will produce a change in the model output Y.
- To the modeler trying to capture how the real world works, there's a fundamental choice to be made between using model (1) or model (2) to anticipate the consequences of a real-world intervention on X.

EXERCISES

1. **We have a bag of three biased coins a, b, and c with probabilities of coming up heads of 20%, 60%, and 80%, respectively. One coin is drawn randomly from the bag (with equal likelihood of drawing each of the three coins), and then the coin is flipped three times to generate the outcomes X_1, X_2, and X_3.**

(*a*) Draw the Bayesian network corresponding to this setup and define the necessary CPTs.

With the random variable C denoting which coin $\{a, b, c\}$ we drew, the network has Cat the root and X_1, X_2, and X_3 as children.

The CPT for C is:

C	P(C)
a	1/3
b	1/3
c	1/3

The CPT for X_i given C are the same, and equal to:

C	X_1	P(C)
a	heads	0.2
b	heads	0.6
c	heads	0.8

(*b*) Calculate which coin was most likely to have been drawn from the bag if the observed flips come out heads twice and tails once.

The coin most likely to have been drawn from the bag given this sequence is the value of C with greatest posterior probability P(C|2 heads, 1 tails). Now,

$$P(C|2\text{ heads, }1\text{ tails}) = P(2\text{ heads, }1\text{ tails} \mid C)P(C) / P(2\text{ heads, }1\text{ tails})$$
$$\propto P(2\text{ heads, }1\text{ tails}|C)P(C)$$
$$\propto P(2\text{ heads, }1\text{ tails}|C)$$

where in the second line we observe that the constant of proportionality 1/P(2 heads, 1 tails) is independent of C, and in the last we observe that P(C) is also independent of the value of C since it is, by hypothesis, equal to 1/3.

From the Bayesian network we can see that X1, X2, and X3 are conditionally independent given C, so for example

$$P(X_1 = \text{tails}, X_2 = \text{heads}, X_3 = \text{heads}|C = a)$$
$$= P(X_1 = \text{tails}|C = a)P(X_2 = \text{heads}|C = a)P(X_3 = \text{heads}|C = a)$$
$$= 0.8 \times 0.2 \times 0.2 = 0.032$$

Note that since the CPTs for each coin are the same, we would get the same probability above for any ordering of 2 heads and 1 tails. Since there are three such orderings, we have

$$P(2\text{heads, }1\text{tails}|C = a) = 3 \times 0.032 = 0.096$$

Similar calculations to the above find that

$$P(2\text{heads, }1\text{tails}|C = b) = 0.432$$

P(2heads, 1tails|C = *c*) = 0.384 showing that coin *b* is most likely to have been drawn.

Alternatively, one could directly compute the value of P(C|2 heads, 1 tails).

TWO MARKS QUESTIONS WITH ANSWERS (PART - A)

1. ***What are the leading causes of uncertainty to occur in the real world?***
   1. Information occurred from unreliable sources.
   2. Experimental Errors
   3. Equipment fault
   4. Temperature variation
   5. Climate change.

2. ***Define Probabilistic reasoning***

Probabilistic reasoning is a way of knowledge representation where we apply the concept of probability to indicate the uncertainty in knowledge. In probabilistic reasoning, we combine probability theory with logic to handle the uncertainty.

3. ***List the ways to solve problems with uncertain knowledge***

1. Bayes' rule
2. Bayesian Statistics

4. ***Define joint probability distribution***

This completely specifies an agent's probability assignments to all propositions in the domain. The joint probability distribution $p(x_1, x_2, \ldots x_n)$ assigns probabilities to all possible atomic events; where $X_1, X_2 \ldots X_n$ 10 = variables.

5. ***Give the Baye's rule equation***

W.K.T $P(A \wedge B) = P(A/B)\, P(B)$ ...(1)

$P(A \wedge B) = P(B/A)\, P(A)$ ...(2)

DIVIDING BY P(A) ;

WE GET

$P(B/A) = P(A/B)\, P(B)$ ---------- $P(A)$

6. ***What is the basic task of a probabilistic inference?***

The basic task is to reason in terms of prior probabilities of conjunctions, but for the most part, we will use conditional probabilities as a vehicle for probabilistic inference.

7. ***Define Bayesian Learning.***

It calculates the probability of each hypothesis, given the data and makes predictions on that basis, (i.e.) predictions are made by using all the hypotheses, weighted by their probabilities rather than by using just single "best" hypotheses

8. ***Define Naïve Bayes model.***

In this model, the "class" variable C is the root and the "attribute" variable XI are the leaves. This model assumes that the attributes are conditionally independent of each other, given the class.

9. ***What is a Bayesian network, and why is it important in AI?***

Bayesian networks are the graphical models that are used to show the probabilistic relationship between a set of variables. It is a directed cycle graph that contains multiple edges, and each edge represents a conditional dependency.

Bayesian networks are probabilistic, because these networks are built from a probability distribution, and also use probability theory for prediction and anomaly detection. It is important in AI as it is based on Bayes theorem and can be used to answer the probabilistic questions.

10. ***Define Conditional probability:***

Conditional probability is a probability of occurring an event when another event has already happened. Let's suppose, we want to calculate the event A when event B has already occurred, "the probability of A under the conditions of B", it can be written as:

$$P(A \mid B) = \frac{P(A \wedge B)}{P(B)}$$

Where $P(A \wedge B)$ = Joint probability of *a* and B

$P(B)$ = Marginal probability of B.

11. ***In a class, there are 70% of the students who like English and 40% of the students who likes English and mathematics, and then what is the percent of students those who like English also like mathematics?***

Solution:

Let, A is an event that a student likes Mathematics

B is an event that a student likes English.

$$P(A \mid B) = \frac{P(A \wedge B)}{P(B)} = \frac{0.4}{0.7} = 57\%$$

Hence, 57% are the students who like English also like Mathematics

12. ***Define Prior probability.***

The prior probability of an event is probability computed before observing new information.

13. ***Define Posterior Probability.***

The probability that is calculated after all evidence or information has taken into account. It is a combination of prior probability and new information.

14. ***Define Gibbs Sampling.***

Gibbs sampling is commonly used as a means of statistical inference, especially Bayesian inference. It is a randomized algorithm (i.e., an algorithm that makes use of random numbers), and is an alternative to deterministic algorithms for statistical inference such as the expectation-maximization algorithm (EM).

15. ***Explain the Bayes' Box.***

The Bayes' box is a method of representing and solving probability through Bayes theorem.

Hypothesis	Prior	Likelihood	Likelihood × Prior	Posterior
A	0.75	1	0.75	0.857
B	0.25	0.5	0.125	0.143
Total			0.875	1

The prior probabilities are assumed values without additional factors.

The likelihood is nothing but the probability of A and B.

The posterior probabilities are results after considering added information. (For instance, rain in the above example). Bayesian Statistics PDF is a good resource to understand this concept with thorough details.

REVIEW QUESTIONS

1. In a class, there are 70% of the students like English, and 40% of the students like English and mathematics, and then what is the percentage of students who like English and also like mathematics?
2. What is the probability that a patient has the disease meningitis with a stiff neck?
3. From a standard deck of playing cards, a single card is drawn. The probability that the card is king is 4/52, then calculate posterior probability P(King|Face), which means the drawn face card is a king card.
4. Construct the Bayesian network.
5. Explain the conditional independence relations in Bayesian networks.
6. Write the enumeration algorithm for answering queries on Bayesian networks.
7. Write the variable elimination algorithm for inference in Bayesian networks.
8. Explain in detail the direct sampling methods.
9. Explain the inference by Markov chain simulation.
10. Write the Gibbs sampling algorithm for approximate inference in Bayesian networks.

****

UNIT III

SUPERVISED LEARNING

Introduction to machine learning - Linear Regression Models: Least squares, single & multiple variables, Bayesian linear regression, gradient descent, Linear Classification Models: Discriminant function - Probabilistic discriminative model - Logistic regression, Probabilistic generative model - Naive Bayes, Maximum margin classifier - Support vector machine, Decision Tree, Random forests

3.1. INTRODUCTION TO MACHINE LEARNING

Machine learning is programming computers to optimize a performance criterion using example data or experience. We have a model defined up to some parameters, and learning is the execution of a computer program to optimize the parameters of the model using the training data or experience. The model may be predictive to make predictions in the future, descriptive to gain knowledge from data or both.

A Machine Learning system learns from historical data, builds the prediction models, and whenever it receives new data, predicts the output for it. The accuracy of predicted output depends upon the amount of data, as the huge amount of data helps to build a better model which predicts the output more accurately.

Suppose we have a complex problem, where we need to perform some predictions, so instead of writing a code for it, we just need to feed the data to generic algorithms, and with the help of these algorithms, the machine builds the logic as per the data and predict the output. Machine learning has changed our way of thinking about problems. The below block diagram explains the working of the Machine Learning algorithm:

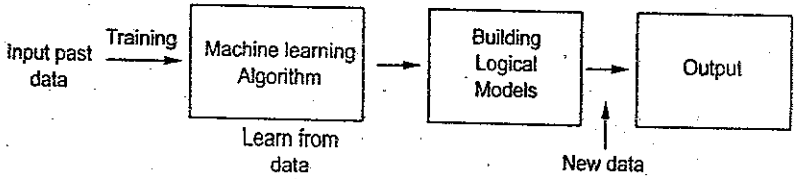


Fig. 3.1. Working of the Machine Learning algorithm

Features of Machine Learning:

- Machine learning uses data to detect various patterns in a given dataset.
- It can learn from past data and improve automatically.
- It is a data-driven technology.
- Machine learning is much similar to data mining as it also deals with a huge amount of data.

Importance of Machine Learning:

- Rapid increment in the production of data
- Solving complex problems, which are difficult for a human
- Decision-making in various sectors including finance
- Finding hidden patterns and extracting useful information from data.

3.1.1. CLASSIFICATION OF MACHINE LEARNING

At a broad level, machine learning can be classified into three types:

1. Supervised learning
2. Unsupervised learning
3. Reinforcement learning

1. Supervised Learning

- Supervised learning is a type of machine learning method in which we provide sample labeled data to the machine learning system to train it, and on that basis, it predicts the output.
- The system creates a model using labeled data to understand the datasets and learn about each data, once the training and processing are done then we test the model by providing sample data to check whether it is predicting the exact output or not.
- The goal of supervised learning is to map input data with the output data. Supervised learning is based on supervision, and it is the same as when a student learns things under the supervision of the teacher. An example of supervised learning is spam filtering.

Example

For instance, suppose you are given a basket filled with different kinds of fruits. Now the first step is to train the machine with all the different fruits one by one.

- If the shape of the object is rounded and has a depression at the top, is red in color, then it will be labeled as –Apple.
- If the shape of the object is a long curving cylinder having Green-Yellow color, then it will be labeled as –Banana.

Now suppose after training the data, you have given a new separate fruit, say a Banana from the basket, and asked to identify it.

Since the machine has already learned things from previous data and this time has to use it wisely. It will first classify the fruit with its shape and color and would confirm the fruit name as BANANA and put it in the Banana category. Thus, the machine learns things from training data (basket containing fruits) and then applies the knowledge to test data (new fruit).

Supervised learning is classified into two categories of algorithms:

(*a*) Classification:

- A classification problem is when the output variable is a category, such as "Red" or "blue", "disease" or "no disease".

(*b*) Regression:

- A regression problem is when the output variable is a real value, such as "dollars" or "weight".

Types Supervised Learning

- Regression
- Logistic Regression
- Classification
- Naive Bayes Classifiers
- K-NN (k nearest neighbor)
- Decision Trees
- Support Vector Machine

Advantages

- Supervised learning allows collecting data and produces data output from previous experiences.

- Helps to optimize performance criteria with the help of experience.
- Supervised machine learning helps to solve various types of real-world computation problems.

Disadvantages

- Classifying big data can be challenging.
- Training for supervised learning needs a lot of computation time. So, it requires a lot of time.

2. Unsupervised Learning

- Unsupervised learning is a learning method in which a machine learns without any supervision.
- The training is provided to the machine with the set of data that has not been labeled, classified, or categorized, and the algorithm needs to act on that data without any supervision.
- The goal of unsupervised learning is to restructure the input data into new features or a group of objects with similar patterns.
- In unsupervised learning, we don't have a predetermined result. The machine tries to find useful insights from a huge amount of data. It can be further classifieds into two categories of algorithms:
  (*a*) Clustering
  (*b*) Association

3. Reinforcement Learning

- Reinforcement learning is a feedback-based learning method, in which a learning agent gets a reward for each right action and gets a penalty for each wrong action.
- The agent learns automatically with this feedback and improves its performance.
- In reinforcement learning, the agent interacts with the environment and explores it.
- The goal of an agent is to get the most reward points, and hence, it improves its performance.

- The robotic dog, which automatically learns the movement of his arms, is an example of Reinforcement learning.

Supervised vs. Unsupervised Machine Learning

Parameters	Supervised machine learning	Unsupervised machine learning
Input Data	Algorithms are trained using labeled data.	Algorithms are used against data that is not labeled
Computational Complexity	Simpler method	Computationally complex
Accuracy	Highly accurate	Less accurate
No. of classes	No. of classes is known	No. of classes is not known
Data Analysis	Uses offline analysis	Uses real-time analysis of data
Algorithms used	Linear and Logistics regression, Random Forest, Support Vector Machine, Neural Networks, etc.	K-Means clustering, Hierarchical clustering, Apriori algorithm, etc.

3.2. LINEAR REGRESSION

Linear Regression is a supervised machine learning algorithm where the predicted output is continuous and has a constant slope. It's used to predict values within a continuous range, (e.g. sales, price) rather than trying to classify them into cat, dog).

3.2.1. TYPES OF LINEAR REGRESSION

Linear Regression can be broadly classified into two types of algorithms. There are three main types:

(*a*) Simple regression

Simple linear regression uses traditional slope-intercept form, where m and b are the variables, our algorithm will try to "learn" to produce the most accurate predictions. x represents our input data and y represents our prediction.

$$y = m x + b$$

(b) Multivariable regression

A more complex, multi-variable linear equation might look like this, where w represents the coefficients or weights, our model will try to learn.

$$f(x, y, z) = w_1 x + w_2 y + w_3 z$$

The variables x, y, and z represent the attributes or distinct pieces of information, we have about each observation. For sales predictions, these attributes might include a company's advertising spend on radio, TV, and newspapers.

$$\text{Sales C} = w_1 \text{ Radio} + w_2 \text{ TV} + w_3 \text{News}$$

(c) Non-Linear Regression

When the best-fitting line is not a straight line but a curve, it is referred to as Non-Linear Regression.

3.2.2. LINEAR REGRESSION TERMINOLOGIES

3.2.2.1. Cost Function

- The cost function measures the performance of a machine learning model for given data.
- The cost function quantifies the error between predicted and expected values and presents that error in the form of a single real number.
- Depending on the problem, the cost function can be formed in many different ways.
- The purpose of the cost function is to be either:
  - **Minimized:** The returned value is usually called cost, loss, or error. The goal is to find the values of model parameters for which the cost function returns as small a number as possible.
  - **Maximized:** In this case, the value it yields is named a reward. The goal is to find values of model parameters for which the returned number is as large as possible.
- Let's use MSE (L_2) as our cost function. MSE measures the average squared difference between an observation's actual and predicted values.

- The output is a single number representing the cost, or score, associated with our current set of weights. Our goal is to minimize MSE to improve the accuracy of our model.
- Given our simple linear equation $y = m x + b$, we can calculate MSE as:

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

  - N is the total number of observations (data points)
  - $\frac{1}{N} \sum_{i=1}^{n}$ is the mean
  - y_i is the actual value of an observation and $mx_i + b$ is our prediction

3.2.2.2. Gradient Descent:

Gradient Descent is an algorithm that finds the best-fit line for a given training dataset in a smaller number of iterations.

- Gradient descent is used to minimize the MSE by calculating the gradient of the cost function.
- A regression model uses gradient descent to update the coefficients of the line by reducing the cost function.
- It is done by a random selection of values of the coefficient and then iteratively updating the values to reach the minimum cost function.
- Gradient descent consists of looking at the error that our weight currently gives us, using the derivative of the cost function to find the gradient (The slope of the cost function using our current weight), and then changing our weight to move in the direction opposite of the gradient.
- We need to move in the opposite direction of the gradient since the gradient points up the slope instead of down it, so we move in the opposite direction to try to decrease our error.
- Here are two parameters (coefficients) in our cost function we can control: weight m and bias b. Since we need to consider the impact each one has on the final prediction, we use partial derivatives.

- To find the partial derivatives, we use the **Chain rule**. We need the chain rule because $(y-(mx+b))^2$ is really 2 nested functions: the inner function $y-(mx+b)$ and the outer function x^2.
- Returning to our cost function:

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

Using the following:

$$(y_i-(m x_i+b))^2 = A(B(m, b))$$

We can split the derivative into

$$A(x) = x^2$$
$$\frac{df}{dx} = A'(x) = 2x$$

And

$$B(m, b) = y_i-(mx_i+b) = y^i-mx_i-b$$
$$\frac{dx}{dm} = B'm = 0-x_i-0 = -x_i$$
$$\frac{dx}{db} = B'(b) = 0-0-1 = -1$$

And then using the Chain rule which states:

$$\frac{df}{dm} = \frac{df}{dx}\frac{dx}{dm}$$
$$\frac{df}{db} = \frac{df}{dx}\frac{dx}{db}$$

We then plug in each of the parts to get the following derivatives

$$\frac{df}{dm} = A'(B(m,f))B'(m) = 2(y_i-(mx_i+b))-x_i$$
$$\frac{df}{db} = A'(B(m,f))B'(b) = 2(y_i-(mx_i+b))-1$$

We can calculate the gradient of this cost function as:

$$f'(m, b) = \begin{bmatrix}\frac{df}{dm}\\ \frac{df}{db}\end{bmatrix} = \begin{bmatrix}\frac{1}{N}\sum -x_i\cdot 2(y_i-(mx_i+b))\\ \frac{1}{N}\sum -1\cdot 2(y_i-(m x_i+b))\end{bmatrix}$$

$$= \begin{bmatrix}\frac{1}{N}\sum -2x_i(y_i-(mx_i+b))\\ \frac{1}{N}\sum -2(y_i-(mx_i+b))\end{bmatrix}$$

To draw an analogy, imagine a pit in the shape of U and you are standing at the topmost point in the pit and your objective is to reach the bottom of the pit. There is a catch, you can only take a discrete number of steps to reach the bottom. If you decide to take one step at a time you would eventually reach the bottom of the pit but this would take a longer time. If you choose to take longer steps each time, you would reach sooner but, there is a chance that you could overshoot the bottom of the pit and not exactly at the bottom. In the gradient descent algorithm, the number of steps you take is the learning rate. This decides on how fast the algorithm converges to the minima.

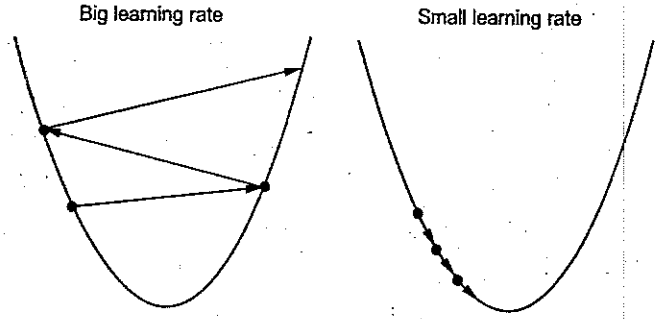


Fig. 3.2. gradient descent

3.2.2.3. Least Squares Regression Line

- If the data shows a leaner relationship between two variables, the line that best fits this linear relationship is known as a least-squares regression line, which minimizes the vertical distance from the data points to the regression line.
- The term "least squares" is used because it is the smallest sum of squares of errors, which is also called the "variance."
- In regression analysis, dependent variables are illustrated on the vertical y-axis, while independent variables are illustrated on the horizontal x-axis.
- These designations will form the equation for the line of best fit, which is determined from the least squares method.

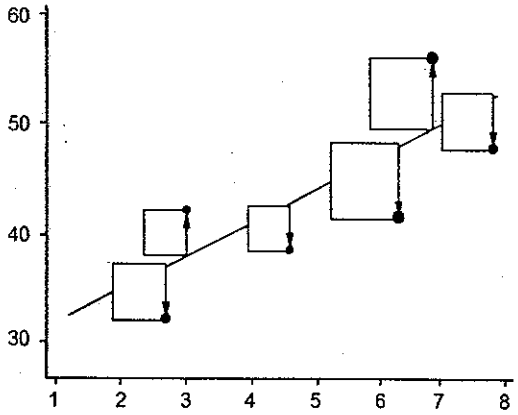


Fig. 3.3. Least Squares Regression Line

Line of Best Fit

Imagine you have some points, and want to have a line that best fits them like this:

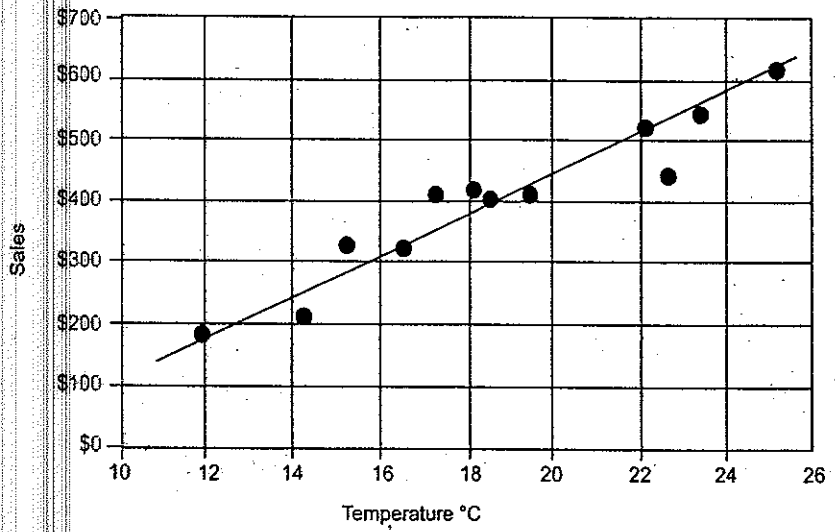


Fig. 3.4. Temperature vs sales

For better accuracy let's see how to calculate the line using Least Squares Regression.

To calculate the values m (slope) and b (y-intercept) in the equation of a line :

$$y = mx + b$$

Where:

y = how far up

x = how far along

m = Slope or Gradient (how steep the line is)

b = the Y Intercept (where the line crosses the Y axis)

To find the line of best fit for N points:

Step 1: For each (x, y) point calculate x^2 and xy

Step 2: Sum all x, y, x^2 and xy, which gives us Σx, Σy, Σx^2 and Σxy (Σ means "sum up")

Step 3: Calculate Slope m:

$$m = \frac{N\sum(xy) - \sum x \sum y}{N\sum(x^2) - (\sum x)^2}$$

(N is the number of points.)

Step 4: Calculate Intercept b:

$$b = \frac{\sum y - m\sum x}{N}$$

Step 5: Assemble the equation of a line

$$y = mx + b$$

Example:

Sam found how many hours of sunshine vs how many ice creams were sold at the shop from Monday to Friday:

"x" Hours of Sunshine	"y" Ice Creams Sold
2	4
3	5
5	7
7	10
9	15

Let us find the best m (slope) and b (y-intercept) that suits that data

$$y = mx + b$$

Step 1: For each (x, y) calculate x^2 and xy:

x	y	x^2	xy
2	4	4	8
3	5	9	15
5	7	25	35
7	10	49	70
9	15	81	135

Step 2: Sum x, y , x^2 and xy (gives us Σx, Σy, Σx^2 and Σxy):

x	y	x^2	xy
2	4	4	8
3	5	9	15
5	7	25	35
7	10	49	70
9	15	81	135
Σx: 26	Σy: 41	Σx^2: 168	Σxy: 263

Step 3: Calculate the Slope m

$$m = \frac{N\sum(xy) - \sum x \sum y}{N\sum(x^2) - (\sum x)^2}$$

$$= \frac{5 \times 263 - 26 \times 41}{5 \times 168 - 26^2}$$

$$= \frac{1315 - 1066}{840 - 676} = \frac{249}{164} = 1.5183\ldots$$

Step 4: Calculate Intercept b

$$b = \frac{\sum y - m\sum x}{N} = \frac{41 - 1.5183 \times 26}{5} = 0.3409\ldots$$

Step 5: Assemble the equation of a line

$$Y = mx + b$$

$$Y = 1.518 \times - 0.305$$

x	y	y = 1.518x + 0.305	error
2	4	3.34	−0.66
3	5	4.86	−0.14
5	7	7.89	0.89
7	10	10.93	0.93
9	15	13.97	−1.03

Here are the (x, y) points and the line $y = 1.518x + 0.305$ on a graph 3.5

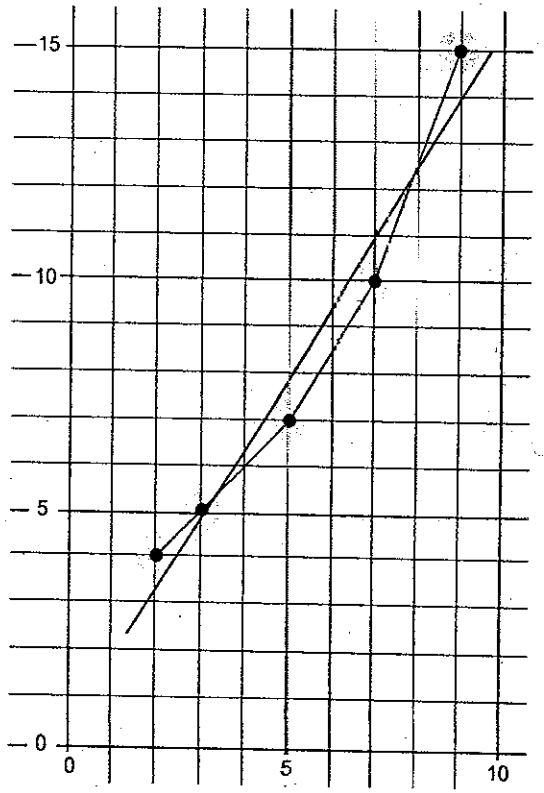


Fig. 3.5.line y = 1.518x + 0.305 on a graph

Sam hears the weather forecast which says "we expect 8 hours of sun tomorrow", so he uses the above equation to estimate that he will sell.

$$y = 1.518 \times 8 + 0.305 = 12.45 \text{ Ice Creams}$$

Sam makes fresh waffle cone mixture for 14 ice creams just in case

- It works by making the total of the **square of the errors** as small as possible (that is why it is called "least squares"):

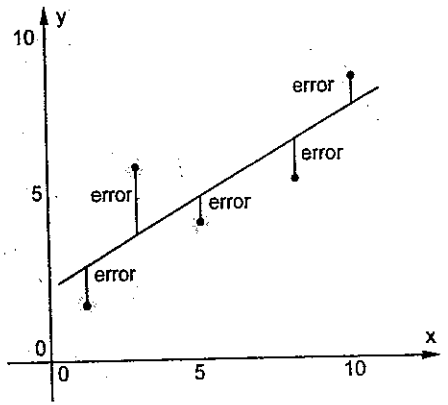


Fig. 3.6. error

- The straight line minimizes the sum of squared errors
- So, when we square each of those errors and add them all up, the total is as small as possible.
- You can imagine (but not accurately) each data point connected to a straight bar by springs:

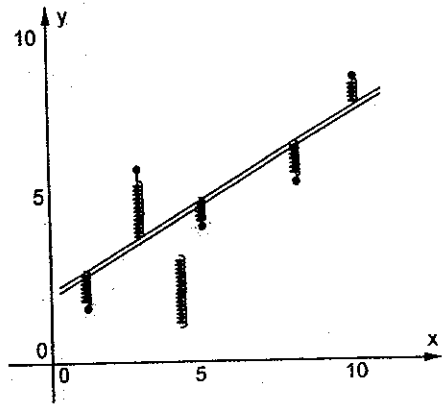


Fig. 3.7. each data point connected to a straight bar

3.3. SIMPLE AND MULTIVARIABLE LINEAR REGRESSION

3.3.1. SIMPLE LINEAR REGRESSION

Simple linear regression is a statistical method that allows us to summarize and study relationships between two continuous (quantitative) variables:

- One variable, denoted x, is regarded as the **predictor**, **explanatory**, or **independent** variable.
- The other variable, denoted y, is regarded as the **response**, **outcome**, or **dependent** variable.
- Because the other terms are used less frequently today, we'll use the "**predictor**" and "**response**" terms to refer to the variables encountered in this course.
- Simple linear regression gets its adjective "simple," because it concerns the study of only one predictor variable.
- In contrast, multiple linear regression, gets its adjective "multiple," because it concerns the study of two or more predictor variables.

Simple linear regression is used to estimate the relationship between two quantitative variables. You can use simple linear regression when you want to know:

- How strong the relationship is between two variables (e.g., the relationship between rainfall and soil erosion).
- The value of the dependent variable at a certain value of the independent variable (e.g., the amount of soil erosion at a certain level of rainfall).

Assumptions of simple linear regression

Simple linear regression is a parametric test, meaning that it makes certain assumptions about the data. These assumptions are:

- **Homogeneity of variance (homoscedasticity):** the size of the error in our prediction doesn't change significantly across the values of the independent variable.
- **Independence of observations:** the observations in the dataset were collected using statistically valid sampling methods, and there are no hidden relationships among observations.

- **Normality:** The data follows a normal distribution.
- The relationship between the independent and dependent variable is linear: the line of best fit through the data points is a straight line (rather than a curve or some sort of grouping factor)

The formula for a simple linear regression is:

$$y = \beta_0 + \beta_1 X + \epsilon$$

- y is the predicted value of the dependent variable (y) for any given value of the independent variable (x).
- B_0 is the **intercept**, the predicted value of y when the x is 0.
- B_1 is the regression coefficient – how much we expect y to change as x increases.
- x is the independent variable (the variable we expect is influencing y).
- e is the **error** of the estimate, or how much variation there is in our estimate of the regression coefficient.

Linear regression finds the line of the best fit line through your data by searching for the regression coefficient (B_1) that minimizes the total error (e) of the model.

3.3.2. MULTIVARIATE LINEAR REGRESSION

This is quite similar to the simple linear regression model we have discussed previously, but with multiple independent variables contributing to the dependent variable and hence multiple coefficients to determine and complex computation due to the added variables.

$$Y_i = \alpha + \beta_1 x_i^{(1)} + \beta_2 x_i^{(2)} + \ldots + \beta_n x_i^{(n)}$$

Y_i is the estimate of the i^{th} component of the dependent variable y, where we have n independent variables and $x_i^{(1)}$ denotes the i^{th} component of the j^{th} independent variable/feature. Similarly, the cost function is as follows,

$$E(\alpha, \beta_1, \beta_2, \ldots, \beta_n) = \frac{1}{2m} \sum_{i=1}^{m} (y_i - Y_i)$$

where we have m data points in training data and y is the observed data of the dependent variable

Computing Parameters

- Generally, when it comes to multivariate linear regression, we don't throw in all the independent variables at a time and start minimizing the error function.
- The First one should focus on selecting the best possible independent variables that contribute well to the dependent variable.
- For this, we go on and construct a correlation matrix for all the independent variables and the dependent variable from the observed data.
- The correlation value gives us an idea about which variable is significant and by what factor.
- From this matrix we pick independent variables in decreasing order of correlation value and run the regression model to estimate the coefficients by minimizing the error function.
- We stop when there is no prominent improvement in the estimation function by the inclusion of the next independent feature.
- This method can still get complicated when there is a large no. of independent features that have significant contributions in deciding our dependent variable.

Normal Equation

- If we have n independent variables in our training data, our matrix X has n +1 rows, where the first row is the 0^{th} term added to each vector of independent variables which has a value of 1 (this is the coefficient of the constant term α).
- So, X is as follows,

$$X = \begin{bmatrix} X_1 \\ \ldots \\ X_m \end{bmatrix}$$

- X^i contains n entries corresponding to each feature in the training data of the i^{th} entry. So, matrix **X** has m rows and n +1 columns (0^{th} column is all 1^{st} and rest for one independent variable each)

$$Y = \begin{bmatrix} Y_1 \\ Y_2 \\ \cdots \\ Y_m \end{bmatrix}$$

and coefficient matrix C,

$$C = \begin{bmatrix} \alpha \\ \beta_1 \\ \cdots \\ \beta_n \end{bmatrix}$$

and our final equation for our hypothesis is,

$$Y = XC$$

To calculate the coefficients, we need n +1 equations and we get them from the minimizing condition of the error function. Equating the partial derivative of E (α, β_1, β_2..., β_n) with each of the coefficients to 0 gives a system of n +1 equations. Solving these is a complicated step and gives the following nice result for matrix C,

$$C = (X^T X)^{-1} X^T y$$

where y is the matrix of the observed values of the dependent variable.

This method seems to work well when the n value is considerably small (approximately for 3-digit values of n). As n grows big the above computation of matrix inverse and multiplication takes a large amount of time. In future tutorials, let's discuss a different method that can be used for data with a large no. of features.

3.4. BAYESIAN LINEAR REGRESSION

- In Bayesian linear regression, the mean of one parameter is characterized by a weighted sum of other variables.
- This type of conditional modeling aims to determine the prior distribution of the regressors as well as other variables describing the allocation of the regress and) and eventually permits the out-of-sample forecasting of the regress and conditional on observations of the regression coefficients.
- The normal linear equation, where the distribution of display style YY given by display style XX is Gaussian, is the most basic and popular variant of this model.

- The future can be determined analytically for this model, and a specific set of prior probabilities for the parameters is known as conjugate priors.
- The posteriors usually have more randomly selected priors.
- When the dataset has too few or poorly dispersed data, Bayesian Regression might be quite helpful.
- In contrast to conventional regression techniques, where the output is only derived from a single number of each attribute, a Bayesian Regression model's output is derived from a probability distribution.
- The result, "*y*," is produced by a normal distribution (where the variance and mean are normalized).
- The goal of the Bayesian Regression Model is to identify the 'posterior' distribution again for model parameters rather than the model parameters themselves.
- The model parameters will be expected to follow a distribution in addition to the output *y*.
- The posterior expression is given below:

$$\text{Posterior} = (\text{Likelihood} * \text{Prior}) / \text{Normalization}$$

The expression parameters are explained below:

- **Posterior:** It is the likelihood that an event, such as H, will take place given the occurrence of another event, such as E, i.e., P(H | E).
- **Likelihood:** It is a likelihood function in which a marginalization parameter variable is used.
- **Priority:** This refers to the likelihood that event H happened before event A, i.e., P(H) (H)

This is the same as Bayes' Theorem, which states the following –

$$P(A \mid B) = (P(B \mid A)\, P(A)) / P(B)$$

- P(A) is the likelihood that event A will occur, while P(A|B) is the likelihood that event A will occur, provided that event B has already occurred. Here, A and B seem to be events. P(B), the likelihood of event B happening cannot be zero because it already has.
- According to the aforementioned formula, we get a prior probability for the model parameters that is proportional to the probability of the data

divided by the posterior distribution of the parameters, unlike Ordinary Least Square (OLS).

- The value of probability will rise as more data points are collected and eventually surpass the previous value. The parameter values converge to values obtained by OLS in the case of an unlimited number of data points.
- As we begin to include additional data points, the accuracy of our model improves. Therefore, to make a Bayesian Ridge Regression model accurate, a considerable amount of train data is required.
- Let's quickly review the mathematical side of the situation now. If 'y' is the expected value in a linear model, then,

$$y(w, x) = w_0 + w_1 x_1 + ... + w_p x_p$$

where, The vector "w" is made up of the elements w_0, w_1,... w_p. The weight value is expressed as 'x'.

$$w = (w_1 ... w_p)$$

- As a result, the output "y" is now considered to be the Gaussian distribution around X_w for Bayesian Regression to produce a completely probabilistic model, as demonstrated below:

$$p(y \mid X, w. \alpha) = N(y \mid X_w, \alpha)$$

where the Gamma distribution prior hyper-parameter alpha is present. It is handled as a probability calculated from the data. The Bayesian Ridge Regression implementation is provided below.

- The Bayesian Ridge Regression formula on which it is based is as follows:

$$p(y \mid \lambda) = N(w \mid 0, \lambda^{-1} Ip)$$

where alpha is the Gamma distribution's shape parameter before the alpha parameter and lambda is the distribution's shape parameter before the lambda parameter.

Real-life Application of Bayesian Linear Regression

Some of the real-life applications of Bayesian Linear Regression are given below:

- **Using Priors:** Consider a scenario in which your supermarkets carry a new product, and we want to predict its initial Christmas sales. For the new product's Christmas effect, we may merely use the average of comparable things as a previous one.

Additionally, once we obtain data from the new item's initial Christmas sales, the previous is immediately updated. As a result, the forecast for the next Christmas is influenced by both the prior and the new item's data.

- **Regularize Priors:** With the season, day of the week, trend, holidays, and a tonne of promotion indicators, our model is severely over-parameterized. Therefore regularization is crucial to keep the forecasts in check.

Advantages of Bayesian Regression

Some of the main advantages of Bayesian Regression are defined below:

- Extremely efficient when the dataset is tiny.
- Particularly well-suited for online learning as opposed to batch learning, when we know the complete dataset before we begin training the model. This is so that Bayesian Regression can be used without having to save data.
- The Bayesian technique has been successfully applied and is quite strong mathematically. Therefore, using this requires no additional prior knowledge of the dataset.

Disadvantages of Bayesian Regression

Some common disadvantages of using Bayesian Regression:

- The model's inference process can take some time.
- The Bayesian strategy is not worthwhile if there is a lot of data accessible for our dataset, and the regular probability approach does the task more effectively.

3.5. LINEAR CLASSIFICATION MODELS: DISCRIMINANT FUNCTION

A class of regression models with simple analytical/computational properties

- The analogous class of models for solving classification problems
- The goal in classification
  - Take a D-dimensional input vector x

- Assign it to one of K discrete classes C_k, $k = 1, \ldots, K$
- In the most common scenario, the classes are taken to be disjoint, each input is assigned to one and only one class

❖ The input space is divided into decision regions. The boundaries of the decision regions are

(a) decision boundaries

(b) decision surfaces

❖ With linear models for classification, the decision surfaces are linear functions

- These decision surfaces are linear functions of the input vector x
- (D − 1)-dimensional hyperplanes, in the D-dimensional input space

❖ Classes that can be separated well by linear surfaces are linearly separable

For regression problems, the target variable t was a vector of real numbers. In classification, there are various ways of representing class labels.

Two-class Problems:

In binary representation, there is a single target variable $t \in \{0, 1\}$

- $t = 1$ represents class C_1
- $t = 0$ represents class C_2

It is the probability of class C_1, with the probability only taking values of 0 and 1

Multi-class Problems:

1-of-K coding scheme, there is a K-long target vector t, such that

- If the class is C_j, all elements t_k of t are zero for k 6 = j and one for $k = j$
- t_k is the probability that the class is C_k K = 6 and $C_k = 4$, then $t = (0, 0, 0, 1, 0, 0)^T$

The simplest approach to classification problems is through the construction of a discriminant function that directly assigns each vector x to a specific class. More powerful is to model the conditional probability distribution $p(C_k | x)$ in an inference stage and use this distribution to make optimal decisions

Discriminative Modelling:

$p(C_k | x)$ can be modelled directly, using a parametric model and optimizing the parameters using a training set

Generative Modelling:

We model class-conditional densities $p(x | C_k)$ and the prior probabilities $p(C_k)$ for the classes, and we compute the posterior probabilities using Bayes' theorem

$$p(C_k | x) = \frac{p(x | C_k)\, p(C_k)}{p(x)}$$

3.5.1. DISCRIMINANT FUNCTIONS

❖ We start with the construction of classifiers based on **discriminant functions**. In linear regression models, the model prediction $y(x, w)$ is a linear function of parameters w. In the simplest case, the model is also linear in the inputs

$$y(x) = w^T x + w_0, \text{ with } y \text{ a real number}$$

❖ In classification problems, we would want to predict discrete class labels. More generally, posterior probabilities are in (0, 1). We can achieve this with a generalization of the linear regression model

$$y(x) = f(w^T x + w_0)(2)$$

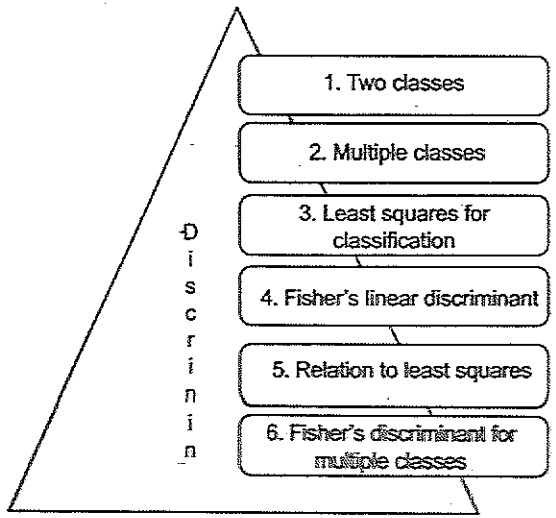


Fig. 3.8. Discriminant functions

- We transform the linear function of w using some nonlinear function $f(\cdot)$

$$y(x) = f(w^T x + w_0)$$

Function $f(\cdot)$ is the **activation function** and its inverse is the **link function**

- Decision surfaces correspond to $y(x)$ = constant so $w^T x + w_0$ = constant. Decision surfaces are linear functions of x, even if $f(\cdot)$ is nonlinear.
- This is the class of models known as **generalized linear models.** They are not linear in the parameters, because of $f(\cdot)$.

1. Two Classes Discriminant Functions

- A simple linear discriminant function is a linear function of the input vector

$$x\, y(x) = w^T x + w_0$$

  - w is the weight vector
  - w_0 is a biased term
  - $-w_0$ is a threshold
- An input vector x is assigned to class C_1 if $y(x) \geq 0$ and to class C_2 otherwise.
- The corresponding decision boundary is defined by the relationship $y(x) = 0$
  - (D − 1) - dimensional hyperplane within the D-dimensional input space
- Consider two points x_A and x_B on the decision boundary

$$\begin{cases} y(x_A) = w^T x_A = 0 \\ y(x_A) = w^T x_B = 0 \end{cases} \rightarrow \; w^T (x_A - x_B) = 0 \rightarrow w \perp (x_A - x_B)$$

- Vector w is orthogonal to every vector in the boundary. w sets the orientation of the boundary
- If x is a point of the decision surface, $y(x) = 0$ and $w^T x = -w_0$ and

$$\frac{w^T x}{\|w\|} = -\frac{w_0}{\|w\|}$$

- This is the normal distance from the origin to the decision surface. w_0 sets the location of the decision boundary.
- The value of $y(x)$ gives a signed measure of perpendicular distance too. The distance from the point x to the decision surface.

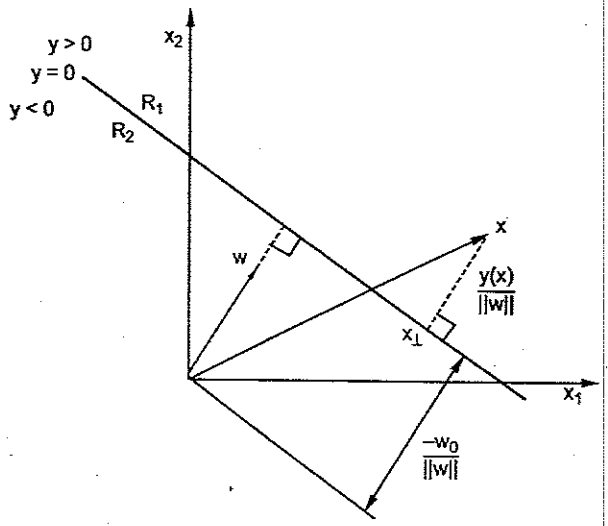


Fig. 3.9. Two classes discriminant functions

- Let x be any point and $x_\perp$ its orthogonal projection onto the boundary

$$X = X_\perp + r\frac{w}{\|w\|}$$

- Multiply both sides by w^T and adding w_0 with $y(x) = w^T x + w_0$ and $y(x_\perp) = w^T x_\perp + w_0 = 0$,

$$r = \frac{y(x)}{\|w\|}$$

- As with linear models for regression, it is sometimes convenient to use a more compact notation and introduce an additional dummy input value $x_0 = 1$.
- We define

$\tilde{w} = (w_0, w)$ and $\tilde{x} = (x_0, x)$, so that

$$y(x) = \tilde{w}^T \tilde{x}$$

- The decision surface is now a D-dimensional hyperplane passing through the origin of the (D + 1)-dimensional expanded input space

2. Multiple classes Discriminant functions

Now consider the extension of linear discriminants to the case of K > 2 classes.

- Consider the use of a K − 1 classifier, each of which solves a two-class problem
  - Separate points in class C_k from points not in C_k
  - It is a one-versus-the-rest classifier

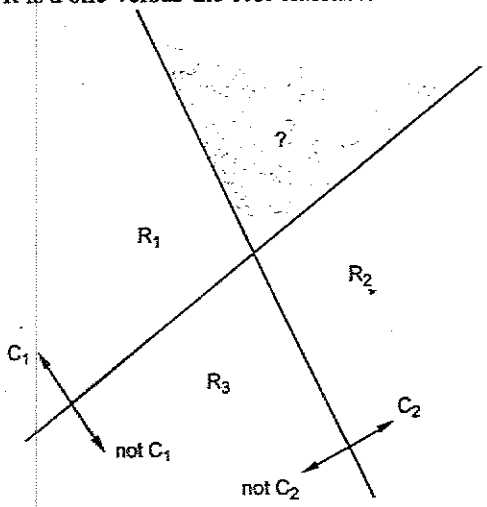


Fig. 3.10. Extension of linear discriminants to the case of K > 2 classes

This approach leads to regions of the input space that are ambiguously classified. By definition, the green area cannot be classified as both C_1 and C_2.

- Consider K(K − 1) / 2 classifiers, one for every possible pair of classes.
  - Separate points in class C_k from points in C_j $6 = k$, with $j = 1, \ldots,$ K
  - It is a one-versus-one classifier
  - Majority voting classifies them

Also, this approach leads to regions of the input space that are ambiguously classified.

- We can avoid these difficulties by considering a single K-class discriminant
  - with K linear functions of the form,

$$y_k(x) = w_k^T x + w_{k0}$$

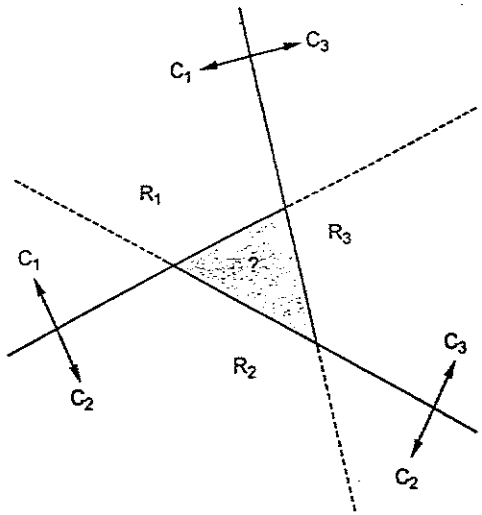


Fig. 3.11. K(K − 1)/2 classifiers

- A point x is then assigned to class C_k,

$$\text{if } y_k(x) > y_j(x), \text{ for all } j \neq k$$

- The boundary between class C_k and class C_j is $y_k(x) = y_j(x)$ or $(w_k - w_j)^T x + (w_{k0} - w_{j0}) = 0$
- A (D − 1)-dimensional hyperplane. It has the same form of the decision boundary for the two-classes case.

3. Least squares for classification

- In regression, models that are linear functions of the parameters could be solved for the parameters using a simple closed-form.
  - Minimisation of the sum-of-squares error function
- We consider a classification problem with K classes, using a 1-of-K binary encoding for the target vector t.

- Each class C_k is described by its own linear model in the form

$$y_k(x) = w_k^T x + w_{k0}, \; k = 1, \ldots, K$$

- The K models can be grouped using vector notation to obtain

$$y(x) = \tilde{W}^T \tilde{x}$$

- $\tilde{W}$ is a matrix whose k-th column comprises, the (D + 1)-dimensional vector

$$\tilde{w}_k = (w_{k0}, w_k^T)^T$$

- $\tilde{x}$ is the corresponding augmented input vector $(1, x^T)^T$ with the dummy input $x_0 = 1$.
- A new input x is assigned to the class for which, $\tilde{w}_k = (w_{k0}, w_k^T)^T$ is largest
- By minimising the sum-of-squares error function, get the parameter matrix $\tilde{W}$
- Consider a training data set $\{ x_n, t_n \}$ where $n = 1 \ldots N$ and define matrix T and matrix $\tilde{x}$
  - The n-th column of T is vector T_n^T
  - The n-th row of $\tilde{x}$ is vector $\tilde{x}_n^T$
- The sum-of-squares error function can be then written as

$$E_D(\tilde{X}) = \frac{1}{2} T_r((\tilde{X}\tilde{W} - T)^T (\tilde{X}\tilde{W} - T))$$

- By setting to zero the derivative of ED ($\tilde{W}$) w.r.t $\tilde{W}$ and rearranging

$$\tilde{W} = (\tilde{X}^T \tilde{X})^{-1} \tilde{X}^T T = \tilde{X}\dagger T$$

- The discriminant function is,

$$y(x) - \tilde{W}^T \tilde{x} - T^T (\tilde{X}\dagger)^T \tilde{x}$$

- Using 1-of-K coding for K classes, the elements of the predictions y(x) will sum to one for any value of x, though cannot be interpreted as probabilities
  - The elements of $y(x)$ are not constrained to be in (0, 1)

- It gives an exact closed-form solution for the discriminant function parameters.
- More worrying is that least-squares solutions lack of robustness to outliers. Outliers lead to large changes in the location of the decision boundary
- A synthetic set from two classes in a two-dimensional space (x_1, x_2)
- The magenta line is the decision boundary from least squares

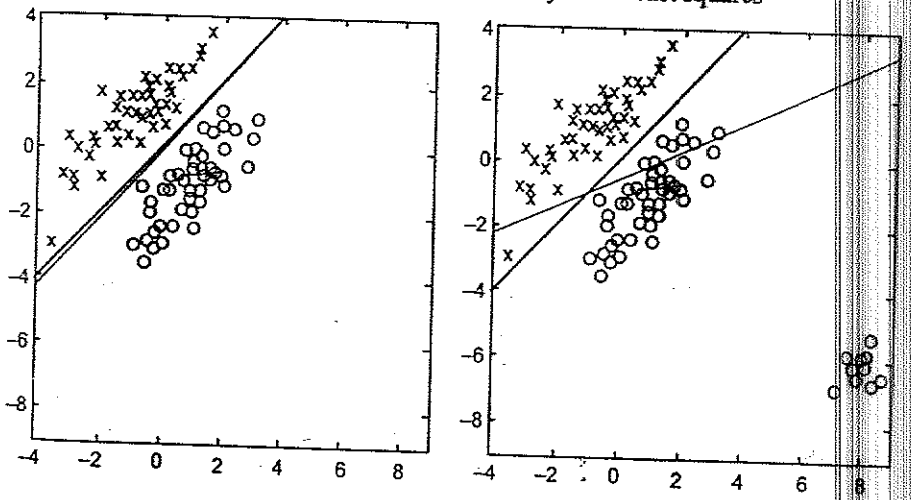


Fig. 3.12. Least squares for classification

In a synthetic set from three classes in a two-dimensional input space (x_1, x_2), Linear decision boundaries could separate classes well.

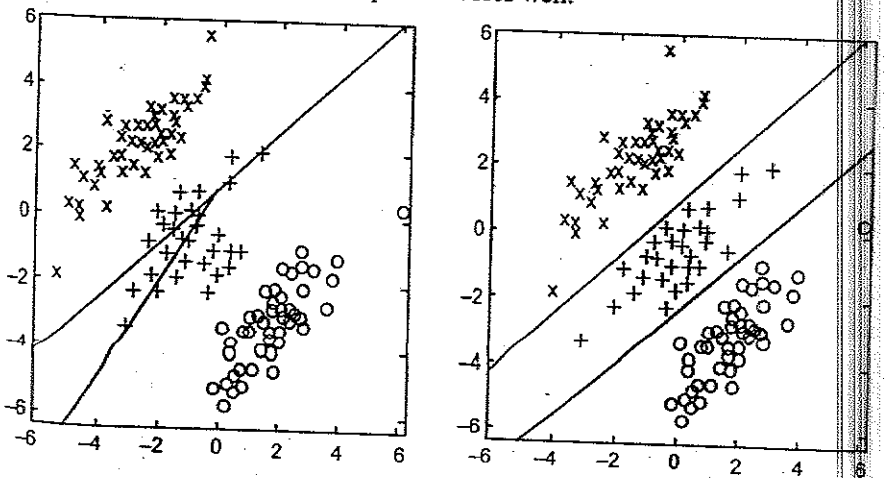


Fig. 3.13. Linear decision boundaries

Least squares correspond to maximum likelihood under the assumption of a Gaussian conditional distribution, but binary target vectors are not Gaussian.

4. Fisher's linear Discriminant

We view linear classification from the viewpoint of dimensionality reduction, Consider the two-classes case,

- We project the D-dimensional input vector x down onto 1D, $y = w_x^T$
- For classification, we place a threshold on y
  - $y \geq -w_0 , \rightarrow C_1$
  - otherwise, $\rightarrow C_2$

❖ Projection onto 1D leads to a considerable loss of information, in general classes that are well separated in the original space may become strongly overlapping in one dimension. Nevertheless, we can always adjust the components of the weight vector w.

❖ The basic idea: Set w, so that the projection maximises class separation,

❖ Consider a two-class problem

- N_1 points of class C_1
- N_2 points of class C_2

❖ The mean vectors of the two classes

$$m_1 = \frac{1}{N_1} \sum_{n \in C_1} x_n$$

$$m_2 = \frac{1}{N_2} \sum_{n \in C_2} x_n$$

❖ We need a measure of the separation of the classes, after projection onto w, An intuitive measure is separation of projected class means m_k,

$$m_2 - m_1 = w^T (m_2 - m_1)$$

❖ the mean of projected C_k data,

$$m_k = w^T m_k$$

$$m_2 - m_1 = w^T (m_2 - m_1)$$

❖ Projection onto the line joining the class means , Good separation in the original 2D space and Considerable class overlap in the projection 1D space

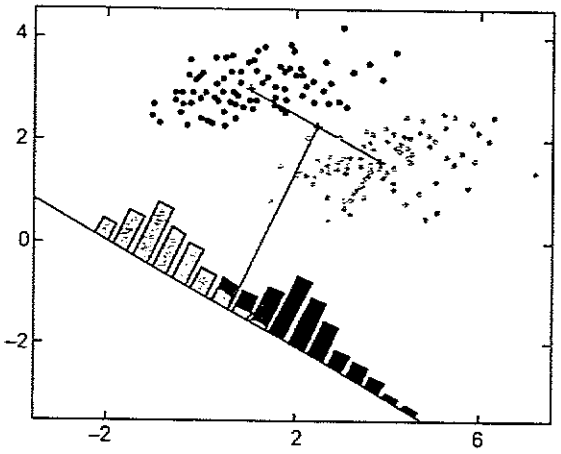


Fig. 3.14.

❖ Fisher's idea is to maximise a function that gives

- Large separation between projected class means
- Small variance within each projected class Or, find a direction that minimises class overlap

❖ The projection $y = w^T x$ transforms labelled points in x into a labelled set in y.

❖ The within-class variance of the projected data

$$s_k^2 = \sum_{n \in C_k} (y_n - m_k)^2$$

❖ The total within-class variance for the whole data (two-classes),

$$s_1^2 + s_2^2$$

❖ The between-class variance,

$$(m_2 - m_1)^2$$

❖ Fisher's criterion: The ratio of between-class and within-class variance,

$$J(w) = \frac{(m_2 - m_1)^2}{s_1^2 + s_2^2}$$

- To make the dependence on w explicit[2], we can write the Fisher's criterion as,

$$J(w) = \frac{w^T S_B w}{w^T S_W w}$$

- S_B is the between-class covariance matrix

$$S_B = (m_2 - m_1)(m_2 - m_1)^T$$

S_W is the total within-class covariance matrix

$$S_W = \sum_{n \in C_1} (x_n - m_1)(x_n - m_1)^T + \sum_{n \in C_2} (x_n - m_2)(x_n - m_2)^T$$

5. Relation to least squares

The least-squares approach to determining a linear discriminant is motivated by making model predictions as close as possible to a set of target values Fisher criterion pursues maximum class separation in the output space.

Consider a total number of patterns N Let N_1 be the number of patterns in class C_1.

- We take the target for class C_1 to be N / N_1 Let N_2 be the number of patterns in class C_2
- We take the target for class C_2 to be $-N / N_2$

The target value for class C_1 approximates the reciprocal of the prior probability for the class.

We write the sum-of-squares error function,

$$E(w, w_0) = \frac{1}{2} \sum_{n=1}^{N} \left[\underbrace{(w^T x_n + w_0)}_{y_n} - t_n \right]^2$$

We set derivatives w.r.t w_0 and w to zero,

$$\sum_{n=1}^{N} (w^T x_n + w_0 - t_n) = 0$$

$$\sum_{n=1}^{N} (w^T x_n + w_0 - t_n) x_n = 0$$

From,

$$\sum_{n=1}^{N} [(w^T x_n + w_0) - t_n] = 0$$

and using the target scheme encoding. The bias is given by w,

$$w_0 = -w^T m$$

where we have used,

$$\sum_{n=1}^{N} t_n = N_1 \frac{N}{N_1} - N_2 \frac{N}{N_2} = 0$$

$$m = \frac{1}{N} \sum_{n=1}^{N} x_n = \frac{1}{N}(N_1 m_1) + N_2 m_2)$$

m is the mean of the total data set.

Using the target encoding, from,

$$\sum_{n=1}^{N} [(w^T x_n + w_0) - t_n] x_n = 0$$

we get,

$$\left(S_W + \frac{N_1 N_2}{N} S_B \right) w = N (m_1 - m_2)$$

with $S_W = \sum_{n \in C_1} (x_n - m_1)(x_n - m_1)^T + \sum_{n \in C_2} (x_n - m_2)(x_n - m_2)^T$

with $S_B = (m_2 - m_1)(m_2 - m_1)^T$

with $w_0 = {}^0 w^T m$

$$S_B = (m_2 - m_1)(m_2 - m_1)^T$$

shows that $S_B w$ is in the direction of $m_2 - m_1$

$$w \propto S_w^{-1} (m_2 - m_1)$$

The weight vector w coincides with what found from the Fisher's criterion

- Vector x with $y(x) = w^T (x - m) > 0$ is classified as belonging to class C_1
- Vector x with $y(x) = w^T (x - m) \leq 0$ is classified as belonging to class C_2

6. Fisher's discriminant for multiple classes

- We consider a generalisation of the Fisher discriminant to $K > 2$ classes
- Assumption: Input dimensionality D is greater than class number K
- We firstly introduce $D' > 1$ linear features $y_k = w_k^T x$ with $k = 1, \ldots, D'$

$$y = w^T x$$

with y grouping $\{y_k\}$, with W grouping $\{w_k\}$

We are not including any bias parameter term in the definition of y, Generalise the within-class covariance matrix to K classes, N_k cases per class.

$$S_W = \sum_{k=1}^{K} S_k$$

$$S_k = \sum_{n \in C_k} (x_n - m_k)(x_n - m_k)^T$$

$$m_k = \frac{1}{N_k} \sum_{n \in C_k} x_n$$

Define the generalisation of the between-class covariance matrix to K classes. Consider first the total covariance matrix,

$$S_T = \sum_{n=1}^{N} (x_n - m)(x_n - m)^T$$

$$m = \frac{1}{N} \sum_{n=1}^{N} x_n$$

m above is the mean of the total data set.

Total covariance matrix can be decomposed into the sum of within-class covariance matrix S_W plus an additional matrix S_B.

$$S_T = S_W + S_B$$

We identify S_B as a measure of between-class covariance,

$$S_B = \sum_{k=1}^{K} N_k (m_k - m)(m_k - m)^T$$

Covariance matrices S_W and S_B are defined in the original x-space. We define similar matrices in the projected D′-dimensional y-space.

$$S_W = \sum_{k=1}^{K} \sum_{n \in C_k} (y_n - \mu_k)(y_n - \mu_k)^T$$

$$S_B = \sum_{k=1}^{K} N_k (\mu_k - \mu)(\mu_k - \mu)^T$$

Where the mean vectors μ_k and μ have been defined as always,

$$\mu_k = \frac{1}{N_k} \sum_{n \in C_k} y_n \qquad \mu = \frac{1}{N} \sum_{k=1}^{N} N_k \mu_k$$

7. The Perceptron

- Another example of a linear discriminant model is Rosenblatt's perceptron. It occupies an important place.
  - In the history of pattern recognition
- It corresponds to a two-class model in which the input vector x is transformed first by using a fixed nonlinear transformation, to give a feature vector $\varphi(x)$
- The feature vector is used to construct a generalised linear model of the form

$$y(x) = f(w^T \phi(x))$$

- The nonlinear activation function $f(\cdot)$ is given by a step function,

$$f(a) = \begin{cases} +1 & a \geq 0 \\ -1 & a < 0 \end{cases}$$

- The feature vector $\varphi(x)$ includes a bias component $\varphi_0(x) = 1$.
- Convenient to use target values $t = +1$ for class C_1 and $t = -1$ for class C_2
- The determination of w can be motivated by error function minimization
- A natural choice of the error function is the total number of misclassified patterns Not a simple algo because the error is a piecewise constant function of w
- Discontinuities wherever a change in w causes the decision boundary to move across one of the points
- Methods based on changing w using the gradient of the error function cannot then be applied, because the gradient is zero almost everywhere
- We consider an alternative error function, known as the perceptron criterion

$$y(w^T \phi(x_n)) = \begin{cases} +1 & w^T \phi(x_n) \geq 0 \\ -1 & w^T \phi(x_n) < 0 \end{cases}$$

- We are seeking a weight vector w such that
  - patterns x_n in class C_1 ($t = +1$) will have $w^T \varphi(x_n) > 0$
  - patterns x_n in class C_2 ($t = -1$) will have $w^T \varphi(x_n) < 0$
- We want all patterns satisfy $w^T \varphi(x_n) t_n > 0$

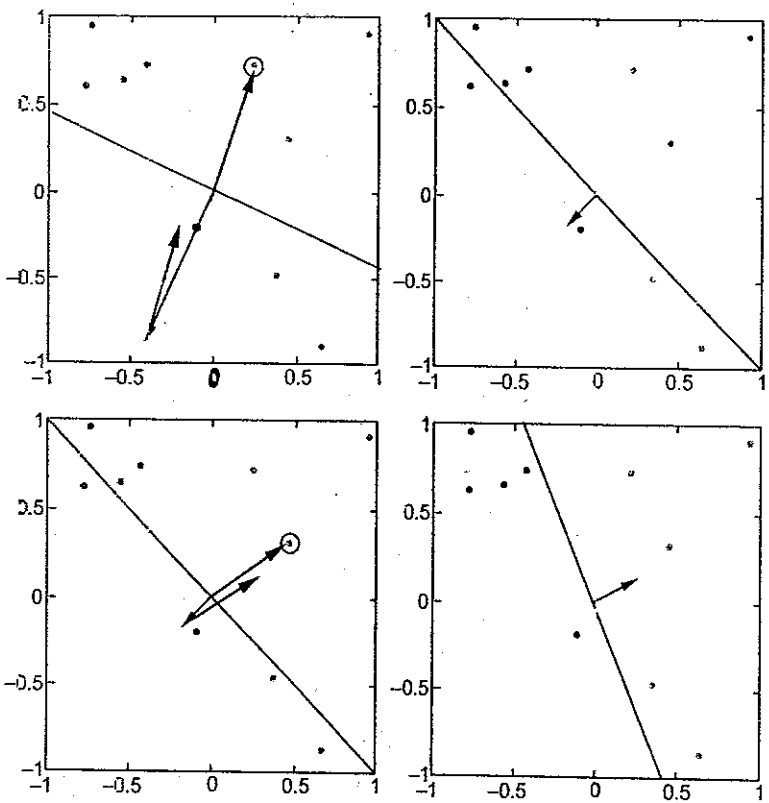


Fig. 3.15.

- The perceptron criterion associates zero error with correctly classified patterns, whereas for a misclassified patter x_n it tries to minimise quantity $-w^T \varphi(x_n) t_n$,

$$E_p(w) = -\sum_{n \in M} w^T \phi_n t_n$$

- $\varphi_n = \varphi(x_n)$ and M denotes the set of misclassified patters,

$$E_p(w) = -\sum_{n \in M} w^T \phi_n t_n$$

- Misclassified patterns contribute to the error with a linear function of w. We can apply a stochastic gradient algorithm to this error function.

$$w^{(T+1)} = w^{(T)} - \eta \nabla E_p(w) = w^{(T)} + \eta \phi_n t_n$$

It changes the weight vector using a learning rate η at each step τ

3.6. PROBABILISTIC DISCRIMINATIVE MODEL

- For the two-class classification problem, the posterior probability of class C_1 can be written as a logistic sigmoid acting on a linear function of x.

$$P(C_1 \mid x) = \sigma\left(ln \frac{p(x \mid C_1) p(C_1)}{p(x \mid C_2) p(C_2)} \right) = \sigma(w^T x + w_0)$$

- For the multi-class case, the posterior probability of class C_k is given by a softmax transformation of a linear function of x.

$$p(C_k \mid x) = \frac{p(x \mid C_k) p(C_k)}{\sum_{j=1}^{k} p(x \mid C_j) p(C_j)} = \frac{\exp(w_k^T x + w_{k0})}{\sum_{j=1}^{k} \exp(w_j^T + w_{j0})}$$

- For specific choices of class-conditionals $p(x \mid C_k)$, the maximum likelihood can be used to determine the parameters of the densities and the class priors $p(C_k)$, Bayes' theorem is then used to find posterior class probabilities $p(C_k \mid x)$.
- An alternative approach is to use the functional form of the generalized linear model explicitly and determine its parameters directly by maximum likelihood,
  - There is an efficient algorithm for finding such solutions
  - Iterative re-weighted least squares, (IRLS)
- The indirect approach to finding parameters of a generalized linear model, by fitting class-conditional densities and class priors separately and then by applying Bayes' theorem, represents an example of generative modelling
- We could take such a model and generate synthetic data by drawing values of x from the marginal distribution $p(x)$ In the direct approach, we

maximise a likelihood function defined through the conditional distribution $p(C_k \mid x)$, this is a form of discriminative training

- One advantage of the discriminative approach is that there will typically be fewer adaptive parameters to be determined
- It may also lead to improved predictive performance, particularly when the class-conditional density assumptions give a poor approximation to the true distributions.

1. Fixed Basis Functions

We considered classification models that work with the original input vector x However, all of the algorithms are equally applicable if we first make a fixed nonlinear transformation of the inputs using a vector of basis functions $\varphi(x)$ The resulting decision boundaries will be linear in the feature space φ, and these correspond to nonlinear decision boundaries in the original x space.

- Classes that are linearly separable in the feature space $\varphi(x)$ need not be linearly separable in the original observation space x.

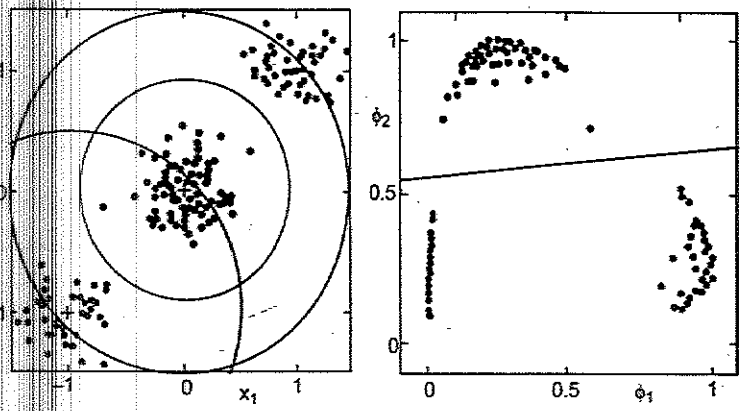


Fig. 3.16.

Original input space (x_1, x_2) together with points from two classes (red/blue)

- Two 'Gaussian' basis functions $\varphi_1(x)$ and $\varphi_2(x)$ are defined in this space with centers (green crosses) and with contours (green circles)

Feature space (φ_1, φ_2) together with the linear decision boundary (black line)

- Nonlinear decision boundary in the original input space (black curve)

Often, there is a significant overlap between class-conditional densities $p(x \mid C_k)$,

- This corresponds to posterior probabilities $p(C_k \mid x)$, which are not 0 or 1
- At least, for some values of x

In such cases, the optimal solution is obtained by modelling the posterior probabilities $p(C_k \mid x)$ accurately and then applying standard decision theory. However, suitable choices of nonlinearity can often make the process of modelling the posterior probabilities easier Not withstanding these limitations, models with fixed nonlinear basis functions play an important role.

2. Logistic Regression

When considering the two-class problem using a generative approach and under general assumptions, the posterior probability of class C_1 is re-written

- a logistic sigmoid on a linear function of the feature vector $\varphi = \varphi(x)$

$$p(C_1 \mid \phi) = y(\phi) = \sigma(w^T \phi) \text{ with } p(C_2 \mid \phi) = 1 - p(C_1 \mid \phi)$$

- The logistic sigmoid function is defined as,

$$\sigma(a) = \frac{1}{1 + \exp(-a)} \text{ with } a = \ln \frac{p(\phi \mid C_1)\, p(C_1)}{p(\phi \mid C_2)\, p(C_2)}$$

- In the terminology of statistics, this model is known as logistic regression, For an M-dimensional feature space φ, the model has M parameters.
- To fit Gaussian class conditional densities with maximum likelihood, we need, $2M + M(M + 1) / 2$ parameters for means and (shared) covariance matrix and a total of $M(M + 5) / 2 + 1$ parameters if we include the class prior $p(C_1)$.
- The number of parameters grows quadratically with M.
- For the M parameters of logistic regression model, we use maximum likelihood
- For data $\{\varphi_n, t_n\}$ where $n = 1....N$ with $t_n = \{0, 1\}$ and $\varphi_n = \varphi(x_n)$, the likelihood function

$$p(t \mid w) = \prod_{n=1}^{N} y_n^{t_n} (1 - y_n)^{1 - t_n}$$

is written for $t = (t_1, ..., t_N)T$ and $y_n = p(C_1 \mid \varphi_n)$.

- By taking the negative log of the likelihood, our error function is defined by,

$$E(w) = -\ln p(t \mid w) = -\sum_{n=1}^{N} (t_n \ln(y_n) + (1 - t_n)\, t_n(1 - y_n))$$

which is the cross-entropy error function with

$$y_n = \sigma(a_n) \text{ and } a_n = w^T \phi_n$$

$$y_n = \sigma(a_n) \text{ and } a_n = w^T \phi_n$$

- By taking the gradient of the error function with respect to w, we get,

$$\nabla E(w) = \sum_{n=1}^{N} (y_n - t_n)\, \phi_n$$

- The contribution to the gradient from point n comes from the error ($y_n - t_n$) between the target value and model prediction, times the basis function vector φ_n.
- The gradient takes the same form as the gradient of the sum-of-squares error function for linear regression models.

3. Iterative Reweighted least Squares

- In the case of the linear regression models, the maximum likelihood solution, on the assumption of a Gaussian noise model, leads to a closed-form solution,
  - A consequence of quadratic dependence of log-likelihood function on w.
- For logistic regression, due to the nonlinearity of the logistic sigmoid function, there is no longer a closed-form solution and departure from quadratic is not substantial
- Specifically, the error function is convex, and hence it has a unique minimum.
- The error function can be minimized by efficient iterative techniques such as those based on Newton-Raphson numerical optimisation
- The Newton-Raphson update, for minimising a function E(w), takes the form

$$w^{(new)} = w^{(old)} - H^{-1} \nabla E(w^{(old)})$$

- H is the Hessian matrix, with elements the second derivatives of E(w) w.r.t w, we apply the Newton-Raphson method to
  1. the sum-of-squares error function (linear regression model)

$$E_D(w) = \frac{1}{2} \sum_{n=1}^{N} (t_n - w^T \phi(x_n))^2$$

  2. the cross-entropy error function (logistic regression model)

$$E(w) = -\ln p(t \mid w) = -\sum_{n=1}^{N} (t_n \ln(y_n) + (1 - t_n)\, t_n(1 - y_n))$$

- Gradient and Hessian of the sum-of-squares error function are

$$\nabla E(w) = \sum_{n=1}^{N} (w^T \phi_n - t_n)\, \phi_n = \Phi^T \Phi w - \Phi^T t$$

$$H = \nabla\nabla E(w) = \sum_{n=1}^{N} \phi_n \phi_n^T = \Phi^T \Phi$$

where Φ is the N × N design matrix with $\varphi^T n$ in the n-th row.

- The Newton-Raphson update takes the form

$$w^{(new)} = w^{(old)} - (\Phi^T\Phi)^{-1} (\Phi^T \Phi w^{(old)} - \Phi^T t)$$

$$= (\Phi^T\Phi)^{-1} \Phi^T t$$

which is the classical least-squares solution

- The error function is quadratic, N-R formula gets the exact solution in one step
- Gradient and Hessian of the cross-entropy error function are

$$\nabla E(w) = \sum_{n=1}^{N} (y_n - t_n)\, \phi_n = \Phi^T(y - t)$$

$$H = \nabla\nabla E(w) = \sum_{n=1}^{N} y_n (1 - y_n \phi_n \phi_n^T) = \Phi^T R \Phi$$

where R(w) is a N × N diagonal matrix with (n, n) elements,

$$R_{nn} = y_n (1 - y_n)$$

- The Hessian is no longer constant, depends on w through weighting matrix R
- Because $0 < y_n < 1$, for an arbitrary vector u, we have that $u^T H_u > 0$, H is positive definite.
- The error function is concave in w and hence it has a unique minimum.
- The Newton-Raphson update formula becomes

$$w^{(new)} = w^{(old)} - (\Phi^T R\Phi)^{-1} \Phi^T(y - t)$$
$$= (\Phi^T R\Phi)^{-1} (\Phi^T R\Phi\, w^{(old)} - \Phi^T (y - t))$$
$$= (\Phi^T R\Phi)^{-1} \Phi^T Rz$$

where z is a N-vector with elements

$$z = \Phi w^{(old)} - R^{-1}(y - t)$$
$$w^{new} = (\Phi^T R\Phi)^{-1} (\Phi^T Rz \text{ with } z = \Phi w^{(old)} - R^{-1} (y - t)$$

- The update is the set of normal equations for a weighted least-squares problem
- Because the weighing matrix R is not constant but depends on the parameter vector w, we must apply the normal equations iteratively each time using the new weight vector w to compute revised weights R
- For this reason, the algorithm is iterative reweighted least squares, or IRLS

4. Multiclass Logistic Regression

- In the discussion of generative models for multiclass classification, we have seen that for a large class of distributions, the posterior probabilities are given by a softmax transformation of linear functions of feature variables.

$$p(C_k \mid \phi) = y_k(\phi) = \frac{\exp(a_k)}{\sum_j \exp(a_j)}$$

where the activations a_k are

$$a_k = w_k^T \phi$$

- We can use maximum likelihood to get parameters $\{w_k\}$ of this model directly. To do it, we need the derivatives of y_k with respect to all of the activations a_j

$$\frac{\partial y_k}{\partial a_j} = y_k (I_{kj} - y_j)$$

where I_{kj} are the elements of the identity matrix,

- Next we need to write the likelihood function using the 1-of-K coding scheme.
- The target vector t_n for feature vector φ_n belonging to class C_k is a binary vector with all elements zero except for element k
- The likelihood is then given by,

$$p(T \mid w_1, \ldots, w_k) = \prod_{n=1}^{N} \prod_{k=1}^{K} p(C_k \mid \phi_n)^{tnk} = \prod_{n=1}^{N} \prod_{k=1}^{K} y_k (\phi_n)^{tnk}$$

where t_{nk} is an element in the N × K matrix T of target variables.

- Taking the negative logarithm gives

$$E(w_1, \ldots, w_k) = -\ln p(T \mid w_1, \ldots, w_k) = -\prod_{n=1}^{N} \prod_{k=1}^{K} t_{nk} \ln (y_{nk})$$

- The cross-entropy error function for the multiclass classification problem.
- We now take the gradient of the error function w.r.t to one parameter vector w_j.

$$\nabla_{wj} E(w_1, \ldots, w_k) = \sum_{n=1}^{N} (y_{nj} - t_{nj}) \phi_n$$

- We used the result for derivatives of the softmax function,

$$\frac{\partial y_k}{\partial a_j} = y_k(I_{kj} - y_j)^1$$

- The same form for the gradient as found for the sum-of-squares error function with the linear model and the cross-entropy error for the logistic regression model
- The product of the error $(y_{nj} - t_{nj})$ times the basis function φ_n
- The derivative of the log-likelihood function for a linear regression model with respect to the parameter vector w for a data point n took the same form the error $(y_n - t_n)$ times the feature vector φ_n.

- Similarly, for the combination of the logistic sigmoid activation function and cross-entropy error function, and for the softmax activation function with the multiclass cross-entropy error function, we again obtain this same simple form.
- To find a batch algorithm, we can use the Newton-Raphson update to obtain the corresponding IRLS algorithm for the multiclass problem
- This requires evaluation of the Hessian matrix that comprises blocks of size M × M in which block (j, k) is given by,

$$\nabla_{wk} \nabla_{wj} E(w_1, \ldots, w_k) = -\sum_{n=1}^{N} y_{nk} (I_{kj} - y_{nj}) \phi_n \phi_n^T$$

- As with two-classes, the Hessian matrix for the multiclass logistic regression models is positive definite and the error function has a unique minimum

3.7. PROBABILISTIC GENERATIVE MODEL

- Models with linear decision boundaries arise from assumptions about the data
- In a generative approach to classification, we first model the class-conditional densities $p(x|\ C_k)$ and the class priors $p(C_k)$, and then we compute posterior probabilities $p(\ C_k \mid x)$ through Bayes' theorem.
- For the two-class problem, the posterior probability of class C_1 is

$$p(C_1 \mid x) = \underbrace{\frac{p(x \mid C_1)\, p(C_1)}{p(x \mid C_1)\, p(C_1) + p(x \mid C_2)\, p(C_2)}}_{p(x) = \sum_k p(x, c_k) = \sum_k p(x, c_k) p(c_k)} = \frac{1}{1 + \exp(-a)} = \sigma(a)$$

where we defined

$$a = ln \frac{p(x \mid C_1)\, p(C_1)}{p(x \mid C_2)\, p(C_2)}$$

- $\sigma(a)$ is the logistic sigmoid function (plotted in red)

or squashing function, because it maps R onto a finite interval.

- The logistic sigmoid satisfies the following symmetry property,

$$\sigma(-a) = 1 - \sigma(a) \quad (4)$$

- The inverse of the logistic sigmoid is known as log it function.

$$a = ln \left(\frac{\sigma}{1 - \sigma} \right)$$

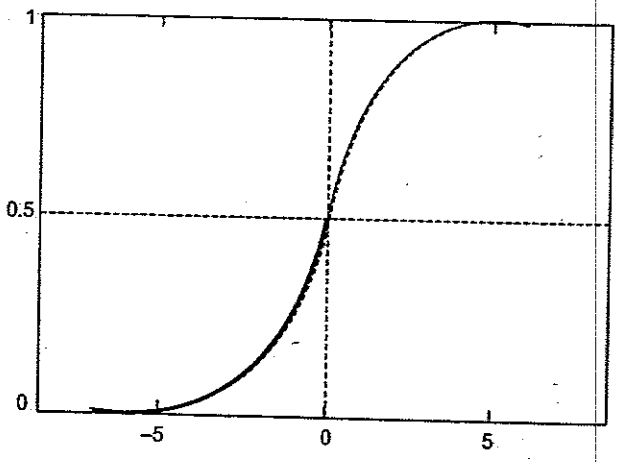


Fig. 3.17.

$$\sigma(a) = \frac{1}{1 + \exp(-a)}$$

- It reflects the log of the ratio of probabilities for two classes

$$ln\,(p\,(C_1 \mid x) / p\,(C_2 \mid x))$$

$$p(C_1 \mid x) = \frac{p(x \mid C_1)\, p(C_1)}{p(x \mid C_1)\, p(C_1) + p(x \mid C_2)\, p(C_2)}$$

$$= \frac{1}{1 + \exp\underbrace{\left(-ln \frac{p(x \mid C_1)\, p(C_1)}{p(x \mid C_2)\, p(C_2)} \right)}_{a}}$$

$$= \sigma\underbrace{\left(ln \frac{p(x \mid C_1)\, p(C_1)}{p(x \mid C_2)\, p(C_2)} \right)}_{a}$$

- We have written the posterior probabilities in an equivalent form that will have significance when $a(x)$ is a linear function of x. Here, the posterior probability is governed by a generalised linear model.
- For the case K > 2 classes, we have,

$$p(C_k \mid x) = \frac{p(x \mid C_k)\, p(C_k)}{\sum_{j=1}^{k} p(x \mid C_j)\, p(C_j)} = \frac{\exp(a_k)}{\sum_{j=1}^{k} \exp(a_j)}$$

known as normalised exponential.

- We have defined the quantity a_k as,

$$a_k = ln\,(p(x \mid C_k)\, p(C_k))$$

$$\text{If } a_k >> a_j, \text{ for all } j \neq k, \text{ then } \begin{cases} p(C_k \mid x) \simeq 1 \\ p(C_j \mid x) \simeq 0 \end{cases}$$

We analyse the consequences of choosing the form of class-conditional densities.

1. Continuous Inputs

- Let us assume that the class-conditional densities $p(x \mid C_k)$ are Gaussian,

$$p(x \mid C_k) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\sum|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_k)^T \sum^{-1} (x-\mu_k)\right)$$

and, we want to explore the form of the posterior probabilities $p(C_k \mid x)$

- The Gaussians have different means μ_k but share the same covariance matrix Σ, With 2 classes,

$$p(C_1 \mid x) = \frac{p(x \mid C_1)\, p(C_1)}{p(x \mid C_1)\, p(C_1) + p(x \mid C_2)\, p(C_2)} = \frac{1}{1+\exp(-a)} = \sigma(a)$$

and

$$a = ln \frac{p(x \mid C_1)\, p(C_1)}{p(x \mid C_2)\, p(C_2)}$$

we have

$$p(C_1 \mid x) = \sigma(w^T x + w_0)$$

Where,

$$w = \sum^{-1} (\mu_1 - \mu_2)$$

$$w_0 = -\frac{1}{2}\mu_1 \sum^{-1} \mu_1 + \frac{1}{2}\mu_1 \sum^{-1} \mu_2 + ln \frac{p(C_1)}{p(C_2)}$$

- The quadratic terms in x from the exponents of the Gaussian densities have cancelled (due to the assumption of common covariance matrices) leading to a linear function of x in the argument of the logistic sigmoid.
- The left-hand plot shows the class-conditional densities for two classes over 2D

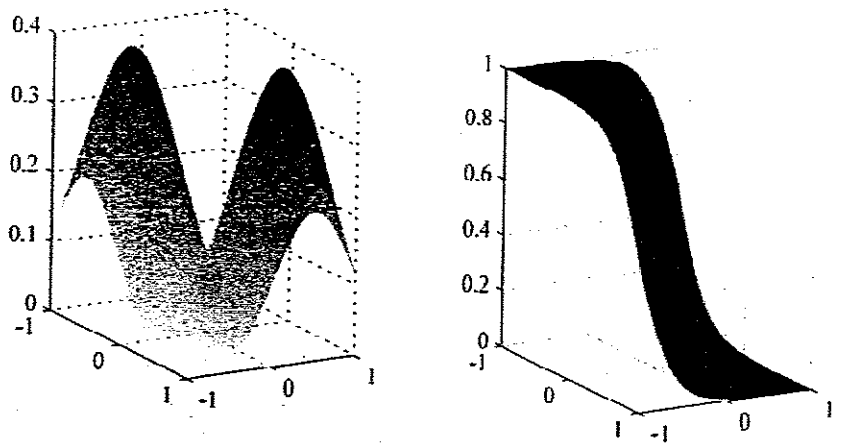


Fig. 3.18. 2D plot

- The posterior probability $p(C_1 \mid x)$ is a logistic sigmoid of a linear function of x The surface in the right-hand plot is coloured using a proportion of red given by $p(C_1 \mid x)$ and a proportion of blue given by $p(C_2 \mid x) = 1 - p(C_1 \mid x)$.
- Decision boundaries are surfaces with constant posterior probabilities $p(C_k \mid x)$
  - Linear functions of x
  - Linear in input space
- Prior probabilities $p(C_k)$ enter only through the bias parameter w_0 so changes in priors have the effect of making parallel shifts of the decision boundary.
- For the K-class case, using

$$p(C_k \mid x) = \frac{p(x \mid C_k)\, p(C_k)}{\sum_{j=1}^{k} p(x \mid C_j)\, p(C_j)} = \frac{\exp(a_k)}{\sum_{j=1}^{k} \exp(a_j)}$$

And

$$a_k = ln\,(p(x \mid C_k)\, p(C_k))$$

we have,

$$a_k(x0 = w_k^T x + w_{k0}$$

$$w_k = \sum^{-1} \mu_k$$

$$w_{k0} = -\frac{1}{2}\, \mu_k^T \sum^{-1} \mu_k + ln\, p(C_k)$$

- The $a_k(x)$ are again linear functions of x as a consequence of the cancellation of the quadratic terms due to the shared covariances.

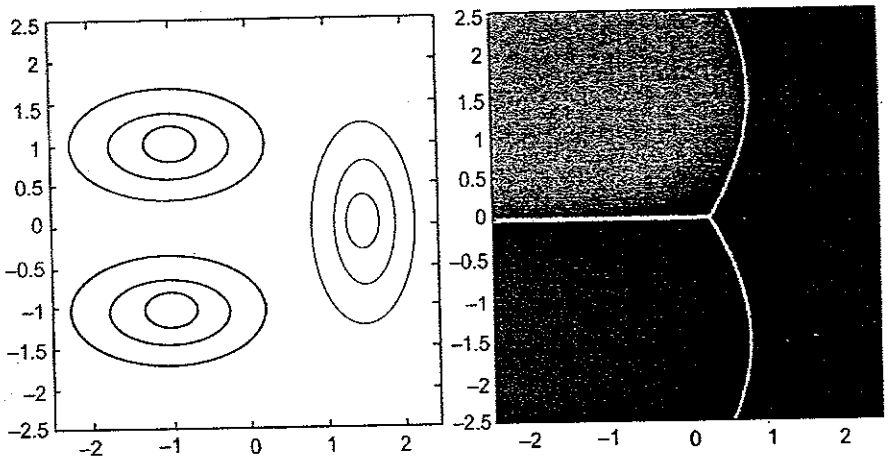


Fig. 3.19.

- The resulting decision boundaries (minimum misclassification rate) occur when two of the posterior probabilities (the two largest) are equal, and so they are defined by linear functions of x.
- If we relax the assumption of a shared covariance matrix and allow each class-conditional density $p(x \mid C_k)$ to have its own covariance matrix Σ_k, then the earlier cancellations will no longer occur, and we will obtain quadratic functions of x, giving rise to a quadratic discriminant.

- Class-conditional densities for three classes each having a Gaussian distribution
  - red and green classes have the same covariance matrix.
- The corresponding posterior probabilities and the decision boundaries
  - Linear boundary between red and green classes, same covariance matrix
  - Quadratic boundaries between other pairs, different covariance matrix

2. Maximum Likelihood Solution

- Once we specified a parametric functional form for class-conditional densities $p(x \mid C_k)$, we can determine parameters and prior class probabilities $p(C_k)$. This requires data comprising observations of x and corresponding class labels.
- Consider first the two-class case, each having a Gaussian density with shared covariance matrix Σ, and suppose we have data, $\{x_n, t_n\}_{n=1}^{N}$

$$\begin{cases} t_n = 1, & \text{for } C_1 \text{ with prior probability } p(C_1) = \pi \\ t_n = 0, & \text{for } C_2 \text{ with prior probability } p(C_2) = 1 - \pi \end{cases}$$

- For a data point x_n from class C_1 (C_2), we have $t_n = 1$ ($t_n = 0$) and thus

$$p(x_n, C_1) = p(C_1)\, p(x_n \mid C_1) = \pi N(x_n \mid \mu_1, \sum)$$

$$p(x_n, C_2) = p(C_2)\, p(x_n \mid C_2) = (1 - \pi)\, N(x_n \mid \mu_2, \sum)$$

- For $t = (t_1, \ldots, t_n)^T$, the likelihood function is given by

$$p(t, X \mid \pi, \mu_1, \mu_2, \sum) = \prod_{n=1}^{N} (\pi N(x_n \mid \mu_1, \sum))^{t_n} ((1 - \pi) N(x^n \mid \mu_2, \sum))^{1 - t_n}$$

- As usual, we maximise the log of the likelihood function

$$\sum_{n=1}^{N} \underbrace{t_n\, ln\,(\pi) + (1 - t_n)\, ln\,(1 - \pi)}_{\pi} + \underbrace{t_n\, ln\,(N(x_n \mid \mu_1, \sum))}_{\mu_1, \Sigma} + \underbrace{(1 + t_n)\, ln\,(N(x_n \mid \mu_2, \sum))}_{\mu_2, \Sigma}$$

$$\mu_1, \mu_2, \Sigma$$

- Consider first maximisation with respect to π, where the terms on π are

$$\sum_{n=1}^{N} (t_n \ln(\pi) + (1-t_n) \ln (1-\pi))$$

- Setting the derivative w.r.t π to zero and rearranging,

$$\pi = \frac{1}{N} \sum_{n=1}^{N} t_n = \frac{N_1}{N} = \frac{N_1}{N_1 + N_2}$$

- The maximum likelihood estimate for π is the fraction of points in C_1
- Now consider maximisation with respect to μ_1, where the terms on μ_1 are

$$\sum_{n=1}^{N} t_n \ln (N(x_n \mid \mu_1, \Sigma)) = \frac{1}{2} \sum_{n=1}^{N} t_n (x_n - \mu_1)^T \Sigma^{-1} (x_n - \mu_1) + \text{const}$$

- Setting the derivative w.r.t μ_1 to zero and rearranging

$$\mu_1 = \frac{1}{N_1} \sum_{n=1}^{N} t_n x_n$$

- The maximum likelihood estimate of μ_1 is the mean of inputs x_n in class C_1.

$$\mu_2 = \frac{1}{N_2} \sum_{n=1}^{N} t_n x_n$$

Lastly consider maximisation with respect to Σ, where the terms on Σ are

$$-\frac{1}{2} \sum_{n=1}^{N} t_n \ln |\Sigma| - \frac{1}{2} \sum_{n=1}^{N} t_n (x_n - \mu_1)^T \Sigma^{-1} (x_n - \mu_1)$$

$$-\frac{1}{2} \sum_{n=1}^{N} (1-t_n) \ln |\Sigma| - \frac{1}{2} \sum_{n=1}^{N} (1-t_n) (x_n - \mu_2)^T \Sigma^{-1} (x_n - \mu_2)$$

$$= -\frac{N}{2} \ln |\Sigma| - \frac{N}{2} \text{Tr} (\Sigma^{-1} S)$$

where

$$S = \frac{N_1}{N} S_1 + \frac{N_2}{N} S_2$$

$$S_1 = \frac{1}{N_1} \sum_{n \in C_1} (x_n - \mu_1) (x_n - \mu_1)^T$$

$$S_2 = \frac{1}{N_2} \sum_{n \in C_2} (x_n - \mu_2) (x_n - \mu_2)^T$$

$$\Sigma = S = \frac{N_1}{N} \frac{1}{N_1} \sum_{n \in C_1} (x_n - \mu_1) (x_n - \mu_1)^T + \frac{N_2}{N} \frac{1}{N_2} \sum_{n \in C_2} (x_n - \mu_2) (x_n - \mu_2)^T$$

It is the average of the covariance matrices associated with each class separately.

3.8. NAIVE BAYES CLASSIFIER ALGORITHM

- Naïve Bayes algorithm is a supervised learning algorithm, which is based on Bayes theorem and used for solving classification problems.
- It is mainly used in text classification that includes a high-dimensional training dataset.
- Naïve Bayes Classifier is one of the simple and most effective Classification algorithms which helps in building the fast machine learning models that can make quick predictions.
- It is a probabilistic classifier, which means it predicts on the basis of the probability of an object.
- Some popular examples of Naïve Bayes Algorithm are spam filtration, Sentimental analysis, and classifying articles.
- The Naïve Bayes algorithm is comprised of two words Naïve and Bayes, Which can be described as:
  - **Naive:** It is called Naïve because it assumes that the occurrence of a certain feature is independent of the occurrence of other features. Such as if the fruit is identified on the bases of color, shape, and taste, then red, spherical, and sweet fruit is recognized as an apple. Hence each feature individually contributes to identify that it is an apple without depending on each other.
  - **Bayes:** It is called Bayes because it depends on the principle of Bayes' Theorem.

3.8.1. BAYES' THEOREM:

- Bayes' theorem is also known as **Bayes' Rule** or **Bayes' law**, which is used to determine the probability of a hypothesis with prior knowledge. It depends on the conditional probability.
- The formula for Bayes' theorem is given as:

$$P(A \mid B) = \frac{P(B \mid A) P(A)}{P(B)}$$

Where,

- **P(A|B) is Posterior probability**: Probability of hypothesis A on the observed event B.
- **P(B|A) is Likelihood probability**: The probability of the evidence given that the probability of a hypothesis is true.
- **P(A) is Prior Probability**: The probability of the hypothesis before observing the evidence.
- **P(B) is Marginal Probability**: Probability of Evidence.

3.8.2. WORKING OF NAÏVE BAYES' CLASSIFIER

Working of Naïve Bayes' Classifier can be understood with the help of the below example:

Suppose we have a dataset of weather conditions and the corresponding target variable "Play". So, using this dataset we need to decide whether we should play or not on a particular day according to the weather conditions. So, to solve this problem, we need to follow the below steps:

1. Convert the given dataset into frequency tables.
2. Generate Likelihood table by finding the probabilities of given features.
3. Now, use Bayes theorem to calculate the posterior probability.

Problem: If the weather is sunny, then the Player should play or not?

Solution: To solve this, first consider the below dataset:

	Outlook	Play
0	Rainy	Yes
1	Sunny	Yes
2	Overcast	Yes
3	Overcast	Yes
4	Sunny	No
5	Rainy	Yes
6	Sunny	Yes
7	Overcast	Yes
8	Rainy	No
9	Sunny	No
10	Sunny	Yes
11	Rainy	No
12	Overcast	Yes
13	Overcast	Yes

Frequency table for the Weather Conditions:

Weather	Yes	No
Overcast	5	0
Rainy	2	2
Sunny	3	2
Total	10	5

Likelihood table weather condition:

Weather	No	Yes	
Overcast	0	5	5 / 14 = 0.35
Rainy	2	2	4 / 14 = 0.29
Sunny	2	3	5 / 14 = 0.35
All	4 / 14 = 0.29	10 / 14 = 0.71	

Applying Bayes' Theorem:

P(Yes|Sunny) = P(Sunny|Yes)*P(Yes) / P(Sunny)

P(Sunny|Yes) = 3 / 10 = 0.3

P(Sunny) = 0.35

P(Yes) = 0.71

So

P(Yes|Sunny) = 0.3*0.71 / 0.35 = **0.60**

P(No|Sunny) = P(Sunny|No) * P(No)/P(Sunny)

P(Sunny|NO) = 2 / 4 = 0.5

P(No) = 0.29

P(Sunny) = 0.35

So

P(No|Sunny) = 0.5*0.29 / 0.35 = **0.41**

So as we can see from the above calculation that **P(Yes|Sunny) > P(No|Sunny)**

Hence on a Sunny day, Player can play the game.

3.8.3. ADVANTAGES OF NAÏVE BAYES CLASSIFIER:

- Naïve Bayes is one of the fast and easy ML algorithms to predict a class of datasets.
- It can be used for Binary as well as Multi-class Classifications.
- It performs well in Multi-class predictions as compared to the other Algorithms.
- It is the most popular choice for **text classification problems.**

3.8.4. DISADVANTAGES OF NAÏVE BAYES CLASSIFIER:

- Naive Bayes assumes that all features are independent or unrelated, so it cannot learn the relationship between features

3.8.5. APPLICATIONS OF NAÏVE BAYES CLASSIFIER:

- It is used for **Credit Scoring.**
- It is used in **medical data classification.**
- It can be used in **real-time predictions** because Naïve Bayes Classifier is an eager learner.
- It is used in Text classification such as **Spam filtering and Sentiment analysis.**

3.8.6. TYPES OF NAÏVE BAYES MODEL:

There are three types of the Naive Bayes Models, which are given below:

- **Gaussian:** The Gaussian model assumes that features follow a normal distribution. This means if predictors take continuous values instead of discrete ones, then the model assumes that these values are sampled from the Gaussian distribution.
- **Multinomial:** The Multinomial Naïve Bayes classifier is used when the data is multinomial distributed. It is primarily used for document classification problems, it means a particular document belongs to which category such as Sports, Politics, education, etc. The classifier uses the frequency of words for the predictors.
- **Bernoulli:** The Bernoulli classifier works similar to the Multinomial classifier, but the predictor variables are the independent Booleans variables. Such as if a particular word is present or not in a document. This model is also famous for document classification tasks.

3.9. MAXIMUM MARGIN CLASSIFIER

- The maximal margin classifier is the optimal hyperplane defined in the (rare) case where two classes are linearly separable.
- Given an $n \times p$ data matrix X with a binary response variable defined as $y \in [-1,1]$ it might be possible to define a p-dimensional hyperplane.

$$h(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 \ldots + \beta_p X_p = x_i^T \beta + \beta_0 = 0$$

- such that all observations of each class fall on opposite sides of the hyperplane.
- This ***separating hyperplane*** has the property that if β is constrained to be a unit vector, $\|\beta\| = \sum\beta^2 = 1$, then the product of the hyperplane and response variables are positive perpendicular distances from the hyperplane, the smallest of which may be termed the hyperplane *margin*, M,
- The maximal margin classifier is the hyperplane with the maximum margin, max{M} subject to $\|\beta\|=1\|$.
- A separating hyperplane rarely exists. In fact, even if a separating hyperplane does exist, its resulting margin is probably undesirably narrow. Here is the maximal margin classifier.

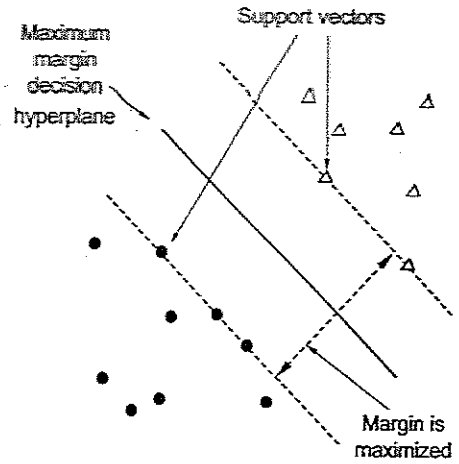


Fig. 3.20. Maximal margin classifier

The data set has two linearly separable classes, y ∈ [−1, 1] described by two features, X_1 and X_2^2. The code is unimportant - just trying to produce the visualization.

3.10. SUPPORT VECTOR MACHINE

- Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.
- The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a **hyperplane.**
- SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called **support vectors**, and hence algorithm is termed **Support Vector Machine.** Consider the below diagram in which there are two different categories that are classified using a decision boundary or hyperplane

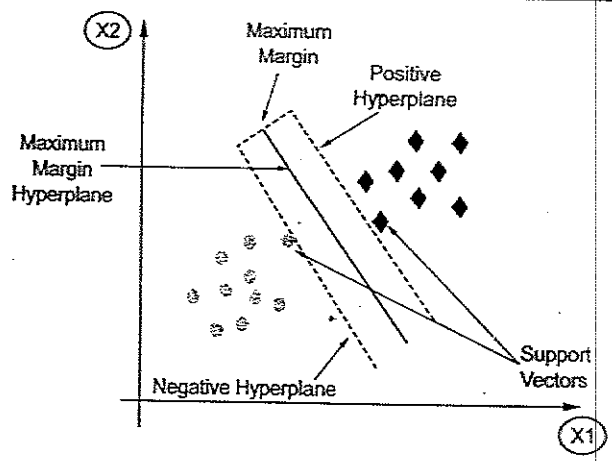


Fig. 3.21. Support Vector Machine

- SVM algorithm can be used for **Face detection, image classification, text categorization,** etc.
- The SVM algorithm is implemented in practice using a kernel.
- The learning of the hyperplane in linear SVM is done by transforming the problem using some linear algebra, which is out of the scope of this introduction to SVM.
- A powerful insight is that the linear SVM can be rephrased using the inner product of any two given observations, rather than the observations themselves. The inner product between two vectors is the sum of the multiplication of each pair of input values.
- For example, the inner product of the vectors [2, 3] and [5, 6] is 2*5 + 3*6 or 28.
- The equation for making a prediction for a new input using the dot product between the input (x) and each support vector (xi) is calculated as follows:

$$f(x) = B_0 + \text{sum}(a_i * (x, x_i))$$

- This is an equation that involves calculating the inner products of a new input vector (x) with all support vectors in training data. The coefficients B_0 and a_i (for each input) must be estimated from the training data by the learning algorithm.

3.10.1. TYPES OF SVM

SVM can be of two types:

1. **Linear SVM:** Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.
2. **Non-linear SVM:** Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data, and the classifier used is called a Non-linear SVM classifier.

3.10.2. HYPERPLANE AND SUPPORT VECTORS IN THE SVM ALGORITHM:

Hyperplane: There can be multiple lines/decision boundaries to segregate the classes in n-dimensional space, but we need to find out the best decision boundary that helps to classify the data points. This best boundary is known as the hyperplane of SVM.

The dimensions of the hyperplane depend on the features present in the dataset, which means if there are 2 features (as shown in the image), then the hyperplane will be a straight line. And if there are 3 features, then the hyperplane will be a 2-dimension plane.

We always create a hyperplane that has a maximum margin, which means the maximum distance between the data points.

Support Vectors:

The data points or vectors that are the closest to the hyperplane and which affect the position of the hyperplane are termed as Support Vector. Since these vectors support the hyperplane, hence called a Support vector.

3.10.3. LINEAR SVM

The working of the SVM algorithm can be understood by using an example. Suppose we have a dataset that has two tags (green and blue), and the dataset has two features x_1 and x_2. We want a classifier that can classify the pair(x_1, x_2) of coordinates in either green or blue. Consider the below image.

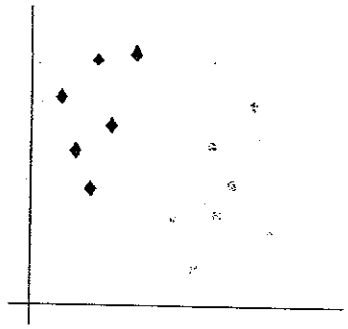


Fig. 3.22. Linear SVM

So as it is 2-d space so by just using a straight line, we can easily separate these two classes. But there can be multiple lines that can separate these classes. Consider the below image,

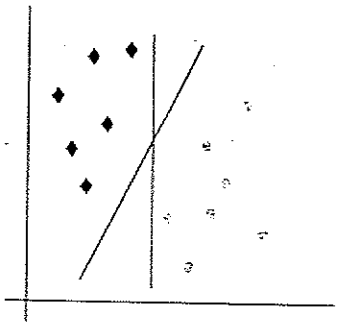


Fig. 3.23. Separate of two classes

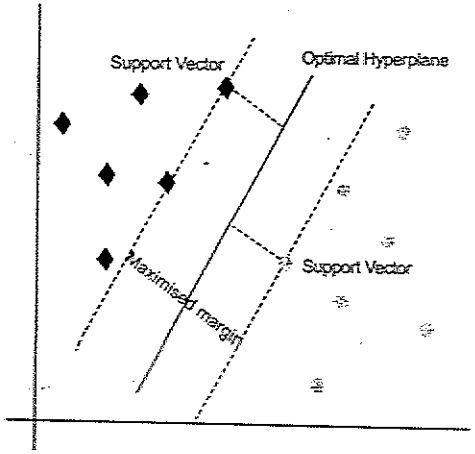


Fig. 3.24. Optional hyperplane

Hence, the SVM algorithm helps to find the best line or decision boundary; this best boundary or region is called a hyperplane. The SVM algorithm finds the closest point of the lines from both the classes. These points are called support vectors. The distance between the vectors and the hyperplane is called as margin. And the goal of SVM is to maximize this margin. The hyperplane with maximum margin is called the optimal hyperplane.

Non-Linear SVM:

If data is linearly arranged, then we can separate it by using a straight line, but for non-linear data, we cannot draw a single straight line. Consider the below image

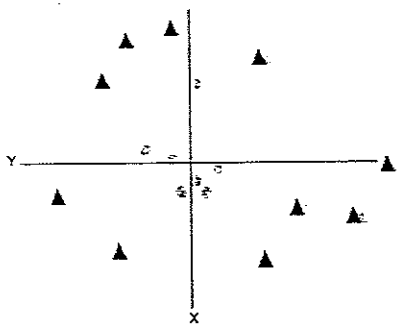


Fig. 3.25. Non-Linear SVM

So to separate these data points, we need to add one more dimension. For linear data, we have used two dimensions x and y, so for non-linear data, we will add a third-dimension z. It can be calculated as:

$$z = x^2 + y^2$$

By adding the third dimension, the sample space will become as below image

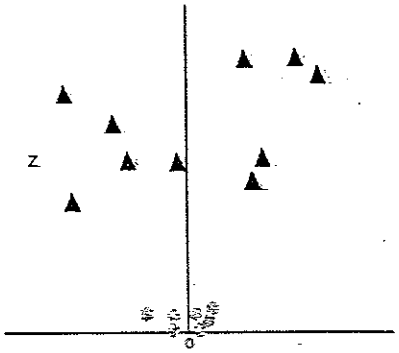


Fig. 3.26. Sample space

So now, SVM will divide the datasets into classes in the following way. Consider the below image:

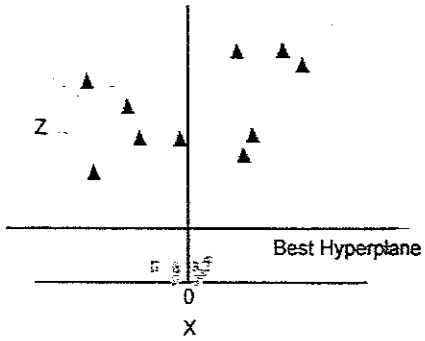


Fig. 3.27. Best Hyperplane

Since we are in 3-d Space, hence it is looking like a plane parallel to the x-axis. If we convert it in 2d space with $z = 1$, then it will become as:

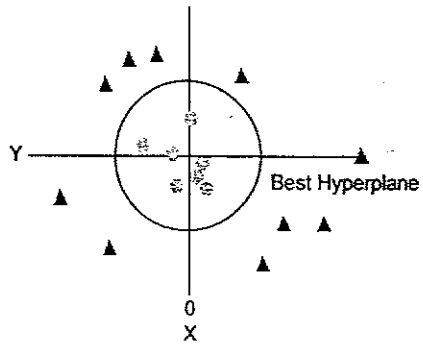


Fig. 3.28. Best Hyperplane in 2d space

Hence, we get a circumference of radius 1 in the case of non-linear data.

3.11. DECISION TREE

- Decision Tree Analysis is a general, predictive modelling tool that has applications spanning a number of different areas.
- In general, decision trees are constructed via an algorithmic approach that identifies ways to split a data set based on different conditions.
- It is one of the most widely used and practical methods for supervised learning.

- Decision Trees are a non-parametric supervised learning method used for both classification and regression tasks.
- The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.
- The decision rules are generally in form of if-then-else statements. The deeper the tree, the more complex the rules and fitter the model.

Before we dive deep, let's get familiar with some of the terminologies:

- **Instances:** Refer to the vector of features or attributes that define the input space
- **Attribute:** A quantity describing an instance
- **Concept:** The function that maps input to output
- **Target Concept:** The function that we are trying to find, i.e., the actual answer
- **Hypothesis Class:** Set of all the possible functions
- **Sample:** A set of inputs paired with a label, which is the correct output (also known as the Training Set)
- **Candidate Concept:** A concept that we think is the target concept
- **Testing Set:** Similar to the training set and is used to test the candidate concept and determine its performance

Below diagram explains the general structure of a decision tree:

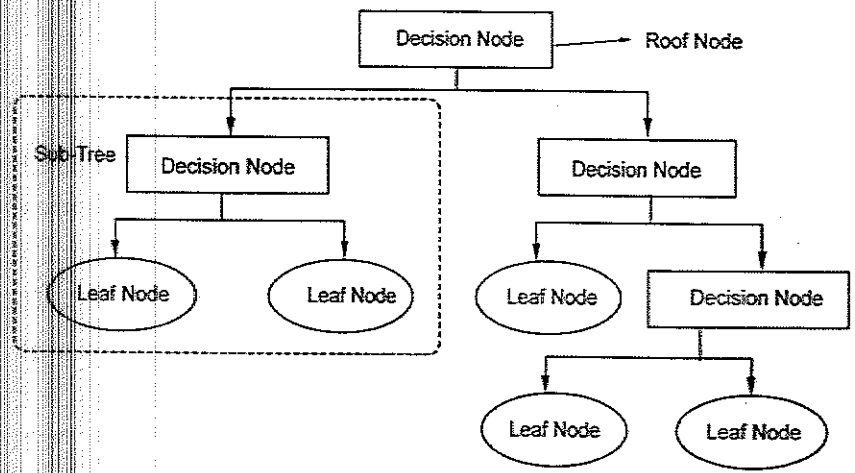


Fig. 3.29. General structure of a decision tree

Types of Decision Tree

There are 4 popular types of decision tree algorithms:

- ID3,
- CART (Classification and Regression Trees),
- Chi-Square and
- Reduction in Variance.

3.11.1. DECISION TREE TERMINOLOGIES

Root Node: Root node is from where the decision tree starts. It represents the entire dataset, which further gets divided into two or more homogeneous sets.

Leaf Node: Leaf nodes are the final output node, and the tree cannot be segregated further after getting a leaf node.

Splitting: Splitting is the process of dividing the decision node/root node into sub-nodes according to the given conditions.

Branch/Sub Tree: A tree formed by splitting the tree.

Pruning: Pruning is the process of removing unwanted branches from the tree.

Parent/Child node: The root node of the tree is called the parent node, and other nodes are called the child nodes.

In a decision tree, for predicting the class of the given dataset, the algorithm starts from the root node of the tree. This algorithm compares the values of the root attribute with the record (real dataset) attribute and, based on the comparison, follows the branch and jumps to the next node.

For the next node, the algorithm again compares the attribute value with the other sub-nodes and moves further. It continues the process until it reaches the leaf node of the tree. The complete process can be better understood using the below algorithm,

- **Step 1:** Begin the tree with the root node, says S, which contains the complete dataset.
- **Step 2:** Find the best attribute in the dataset using the Attribute Selection Measure (ASM).
- **Step 3:** Divide the S into subsets that contain possible values for the best attributes.

❖ **Step 4:** Generate the decision tree node, which contains the best attribute.

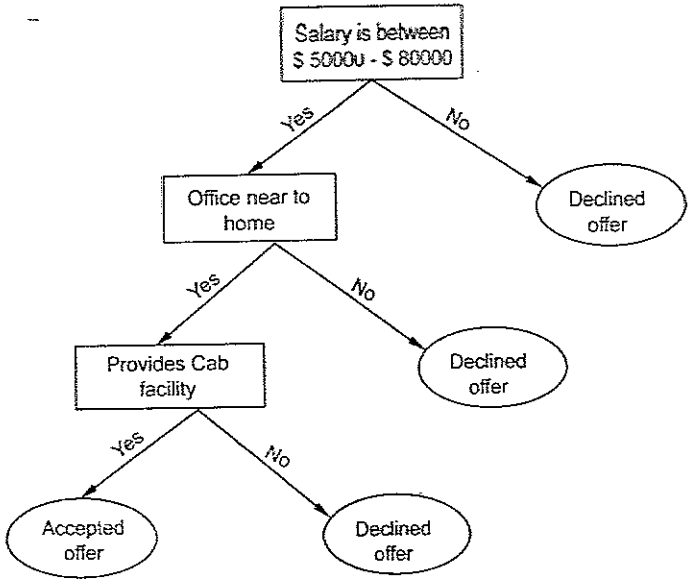


Fig. 3.30. Sample decision tree example

❖ **Step 5:** Recursively make new decision trees using the subsets of the dataset created in step -3. Continue this process until a stage is reached where you cannot further classify the nodes and called the final node as a leaf node.

Example: Suppose there is a candidate who has a job offer and wants to decide whether he should accept the offer or not. So, to solve this problem, the decision tree starts with the root node (Salary attribute by ASM). The root node splits further into the next decision node (distance from the office) and one leaf node based on the corresponding labels. The next decision node further gets split into one decision node (Cab facility) and one leaf node. Finally, the decision node splits into two leaf nodes (Accepted offers and Declined offer). Consider the above diagram.

3.11.2. ATTRIBUTE SELECTION MEASURES

❖ While implementing a Decision tree, the main issue arises that how to select the best attribute for the root node and for sub-nodes. So, to solve such problems there is a technique which is called an **Attribute selection measure or ASM.**

❖ By this measurement, we can easily select the best attribute for the nodes of the tree. There are two popular techniques for ASM, which are:

- Information Gain
- Gini Index

3.11.3. INFORMATION GAIN:

❖ Information gain is the measurement of changes in entropy after the segmentation of a dataset based on an attribute.

❖ It calculates how much information a feature provides us about a class.

❖ According to the value of information gain, we split the node and build the decision tree.

❖ A decision tree algorithm always tries to maximize the value of information gain, and a node/attribute having the highest information gain is split first.

$$IG(S, A) = H(S) - \sum_{i=0}^{n} P(x) * H(x)$$

❖ It can be calculated using the below formula:

Information Gain = Entropy(S) - [(Weighted Avg) * Entropy (each feature)

3.11.4. ENTROPY:

Entropy is a metric to measure the impurity in a given attribute. It specifies randomness in data.

$$H(S) = \sum_{x \in X} p(x) \log_2 \frac{1}{p(x)}$$

Entropy can be calculated as:

$$\text{Entropy(s)} = -P(\text{yes}) \log_2 P(\text{yes}) - P(\text{no}) \log_2 P(\text{no})$$

Where,

S = Total number of samples

P(yes) = Probability of yes

P(no) = Probability of no

3.11.5. GINI INDEX:

❖ Gini index is a measure of impurity or purity used while creating a decision tree in the CART (Classification and Regression Tree) algorithm.

- An attribute with the low Gini index should be preferred as compared to the high Gini index.
- It only creates binary splits, and the CART algorithm uses the Gini index to create binary splits.
- Gini index can be calculated using the below formula:

$$\text{Gini Index} = 1 - \sum_j P_j^2$$

3.11.6. PRUNING: GETTING AN OPTIMAL DECISION TREE

- ***Pruning is a process of deleting the unnecessary nodes from a tree in order to get the optimal decision tree.***
- A too-large tree increases the risk of overfitting, and a small tree may not capture all the important features of the dataset. Therefore, a technique that decreases the size of the learning tree without reducing accuracy is known as Pruning.
- There are mainly two types of trees **pruning** technology used:
  - Cost Complexity Pruning
  - Reduced Error Pruning.

3.11.7. ADVANTAGES OF THE DECISION TREE

- It is simple to understand as it follows the same process which a human follow while making any decision in real-life.
- It can be very useful for solving decision-related problems.
- It helps to think about all the possible outcomes for a problem.
- There is less requirement of data cleaning compared to other algorithms

3.11.8. DISADVANTAGES OF THE DECISION TREE

- The decision tree contains lots of layers, which makes it complex.
- It may have an overfitting issue, which can be resolved using the **Random Forest algorithm.**
- For more class labels, the computational complexity of the decision tree may increase.

3.12. RANDOM FOREST ALGORITHM

- Random Forest is a popular machine learning algorithm that belongs to the supervised learning technique.
- It can be used for both Classification and Regression problems in ML.
- It is based on the concept of ensemble learning, which is a process of combining multiple classifiers to solve a complex problem and to improve the performance of the model.
- As the name suggests, "Random Forest is a classifier that contains a number of decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset."
- Instead of relying on one decision tree, the random forest takes the prediction from each tree and based on the majority votes of predictions, and it predicts the final output.
- The greater number of trees in the forest leads to higher accuracy and prevents the problem of overfitting.
- The below diagram explains the working of the Random Forest algorithm:

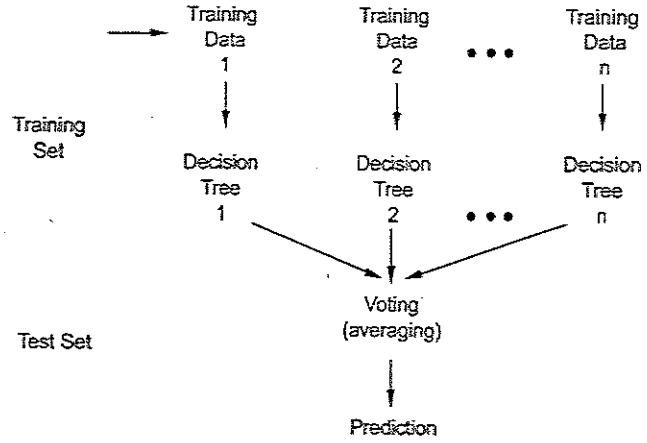


Fig. 3.31. Working of the Random Forest algorithm

- Since the random forest combines multiple trees to predict the class of the dataset, it is possible that some decision trees may predict the correct output, while others may not. But together, all the trees predict the correct

output. Therefore, below are two assumptions for a better Random Forest classifier:

- There should be some actual values in the feature variable of the dataset so that the classifier can predict accurate results rather than a guessed result.
- The predictions from each tree must have very low correlations.

Below are some points that explain why we should use the Random Forest algorithm:

- ❖ It takes less training time as compared to other algorithms.
- ❖ It predicts output with high accuracy, even for the large dataset it runs efficiently.
- ❖ It can also maintain accuracy when a large proportion of data is missing.

3.12.1. WORKING OF RANDOM FOREST

- ❖ Random Forest works in two-phase first is to create the random forest by combining N decision tree, and the second is to make predictions for each tree created in the first phase.

Steps involved in random forest algorithm:

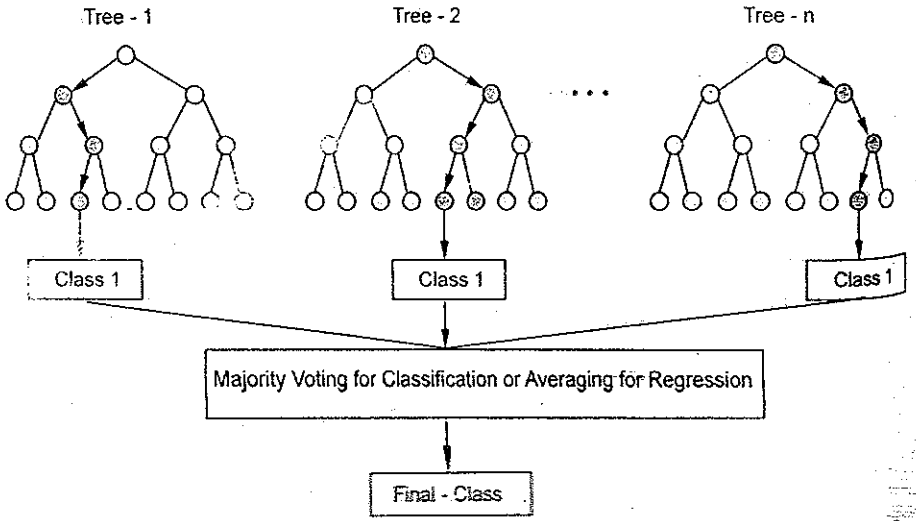


Fig. 3.32. Random forest algorithm

- ❖ **Step 1:** In Random forest *n* number of random records are taken from the data set having k number of records.
- ❖ **Step 2:** Individual decision trees are constructed for each sample.
- ❖ **Step 3:** Each decision tree will generate an output.
- ❖ **Step 4:** Final output is considered based on Majority Voting or Averaging for Classification and regression respectively.
- ❖ For example: consider the fruit basket as the data as shown in the figure below. Now n number of samples are taken from the fruit basket and an individual decision tree is constructed for each sample. Each decision tree will generate an output as shown in the figure. The final output is considered based on majority voting. In the below figure you can see that the majority decision tree gives output as an apple when compared to a banana, so the final output is taken as an apple.

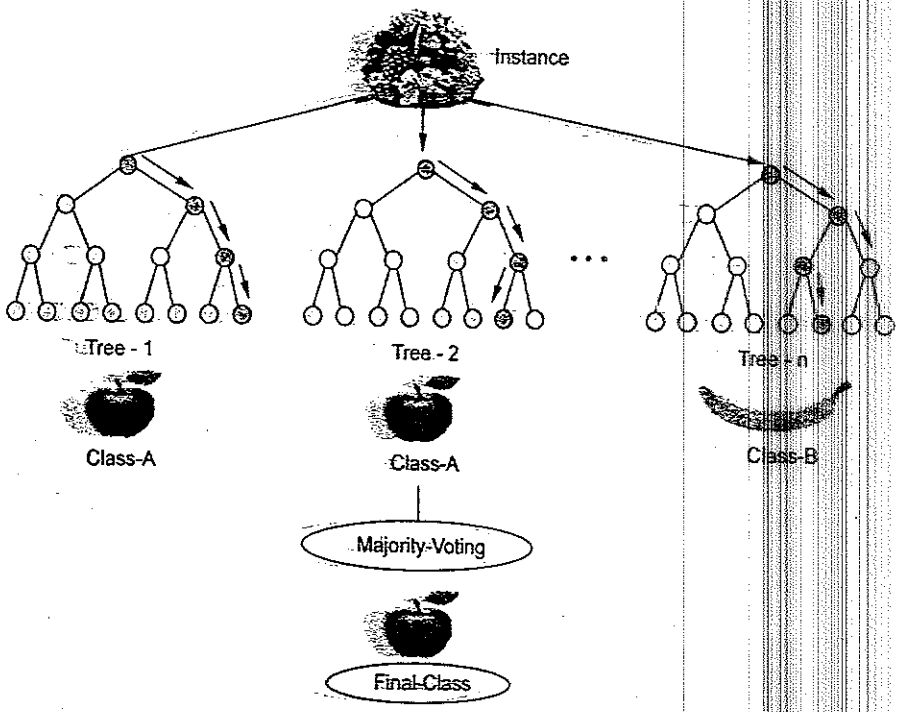


Fig. 3.33. Random forest example

This combination of multiple models is called Ensemble. Ensemble uses two methods:

Bagging: Creating a different training subset from sample training data with replacement is called Bagging. The final output is based on majority voting.

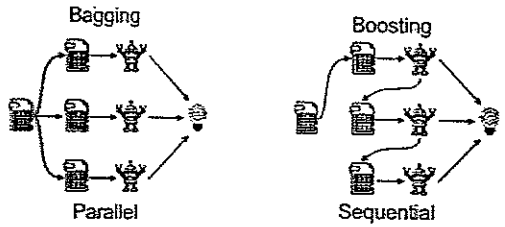


Fig. 3.34. Bagging and Boosting

Boosting: Combing weak learners into strong learners by creating sequential models such that the final model has the highest accuracy is called Boosting. Example: ADA BOOST, XG BOOST.

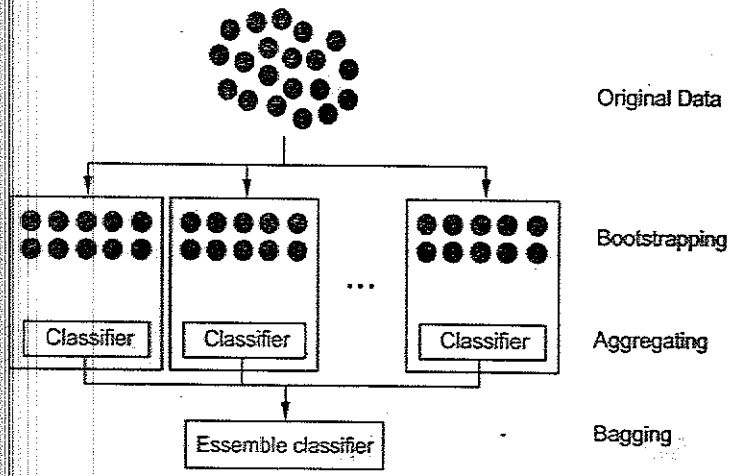


Fig. 3.35. Bootstrapping

Bagging: From the principle mentioned above, we can understand Random forest uses the Bagging code. Now, let us understand this concept in detail. Bagging is also known as Bootstrap Aggregation used by random forest. The process begins with any original random data. After arranging, it is organised into samples known as Bootstrap Sample. This process is known as Bootstrapping. Further, the models are trained individually, yielding different results known as Aggregation. In the last step,

all the results are combined, and the generated output is based on majority voting. This step is known as Bagging and is done using an Ensemble Classifier.

3.12.2. ESSENTIAL FEATURES OF RANDOM FOREST

1. **Diversity**- Not all attributes/variables/features are considered while making an individual tree, each tree is different.
2. **Immune to the curse of dimensionality**- Since each tree does not consider all the features, the feature space is reduced.
3. **Parallelization** - Each tree is created independently out of different data and attributes. This means that we can make full use of the CPU to build random forests.
4. **Train-Test split**- In a random forest we don't have to segregate the data for train and test as there will always be 30% of the data which is not seen by the decision tree.
5. **Stability**- Stability arises because the result is based on majority voting/ averaging.

3.12.3. DIFFERENCE BETWEEN DECISION TREE & RANDOM FOREST

Random forest is a collection of decision trees; still, there are a lot of differences in their behavior.

	Decision Trees	Random Forest
1.	Decision trees normally suffer from the problem of overfitting if it's allowed to grow without any control	Random forests are created from subsets of data and the final output is based on average or majority ranking and hence the problem of overfitting is taken care of.
2.	A single decision tree is faster in computation	It is comparatively slower.
3.	When a data set with features is taken as input by a decision tree it will formulate some set of rules to do prediction	Random forest randomly selects observations, builds a decision tree and the average result is taken. It doesn't use any set of formulas.

Thus random forests are much more successful than decision trees only if the trees are diverse and acceptable.

3.12.4. IMPORTANT HYPERPARAMETERS

Hyperparameters are used in random forests to either enhance the performance and predictive power of models or to make the model faster.

The following hyperparameters are used to enhance the predictive power:

- **n_estimators:** Number of trees built by the algorithm before averaging the products.
- **max_features:** Maximum number of features random forest uses before considering splitting a node.
- **mini_sample_leaf:** Determines the minimum number of leaves required to split an internal node.

The following hyperparameters are used to increase the speed of the model:

- **n_jobs:** Conveys to the engine how many processors are allowed to use. If the value is 1, it can use only one processor, but if the value is -1,, there is no limit.
- **random_state:** Controls randomness of the sample. The model will always produce the same results if it has a definite value of random state and if it has been given the same hyperparameters and the same training data.
- **oob_score:** OOB (Out Of the Bag) is a random forest cross-validation method. In this, one-third of the sample is not used to train the data but to evaluate its performance.

3.12.5. IMPORTANT TERMS TO KNOW

There are different ways that Random Forest algorithm makes data decisions, and consequently, there are some important related terms to know. Some of these terms include:

- **Entropy:** It is a measure of randomness or unpredictability in the data set.
- Information Gain: A measure of the decrease in the entropy after the data set is split is the information gain.
- **Leaf Node:** A leaf node is a node that carries the classification or the decision.
- **Decision Node:** A node that has two or more branches.

- **Root Node:** The root node is the topmost decision node, which is where you have all of your data.

Now that you have looked at the various important terms to better understand the random forest algorithm, let us next look at a case example.

3.12.6. CASE EXAMPLE

Let's say we want to classify the different types of fruits in a bowl based on various features, but the bowl is cluttered with a lot of options. You would create a training dataset that contains information about the fruit, including colors, diameters, and specific labels (i.e., apple, grapes, etc.) You would then need to split the data by sorting out the smallest piece so that you can split it in the biggest way possible. You might want to start by splitting your fruits by diameter and then by color. You would want to keep splitting until that particular node no longer needs it, and you can predict a specific fruit with 100 percent accuracy.

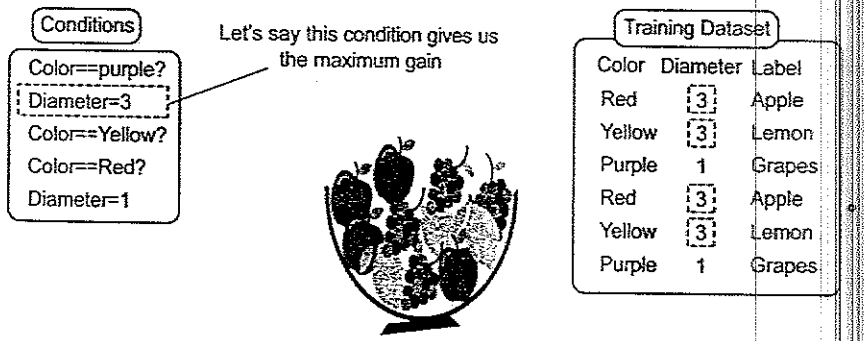


Fig. 3.36. Case example

Coding in Python – Random Forest

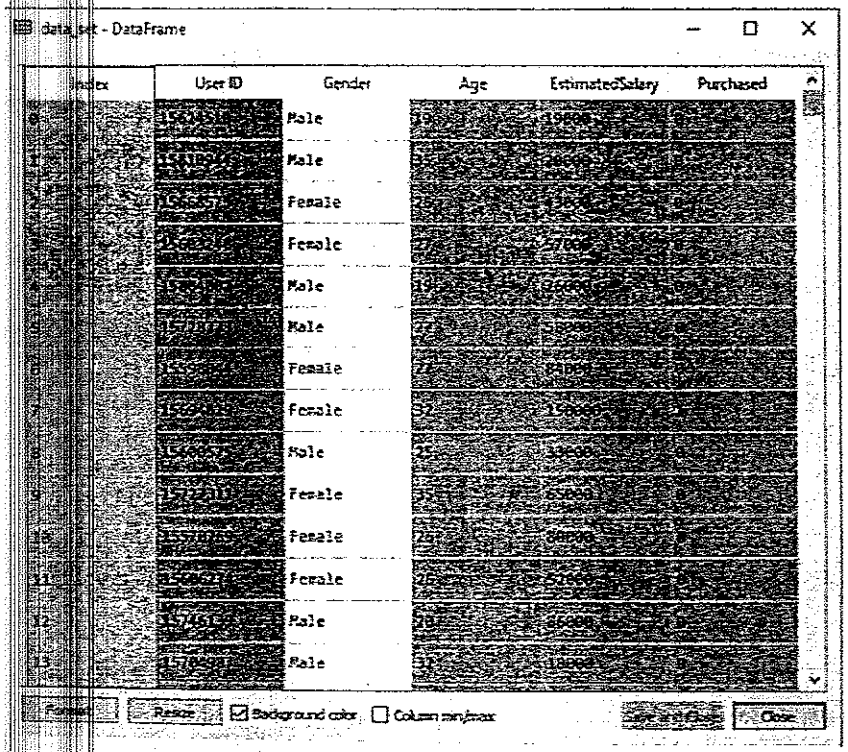
1. Data Pre-Processing Step:

The following is the code for the pre-processing step-

```
# importing libraries
import numpy as nm
import matplotlib.pyplot as mtp
import pandas as pd
# importing datasets
```

```
data set= pd.read_csv('user_data.csv')
#Extracting Independent and dependent Variable
x = data_set.iloc[:, [2,3]].values
y = data_set.iloc[:, 4].values
#Splitting the dataset into training and testset
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_train_test_split(x,y,test_size=
#feature Scaling
from sklearn.preprocessing import Standard Scaler
st_x= StandardScaler( )
x_train= st+x.fit+transform(x_train)
x_test= st_x.transform(x_test)
```

We have processed the data when we have loaded the dataset:



2. Fitting the Random Forest Algorithm:

Now, we will fit the Random Forest Algorithm in the training set. To do that, we will import RandomForestClassifier class from the sklearn. Ensemble library.

```
#Fitting Decision Tree classifier to the training set
from sklearn.ensemble import RandomForestClassifier
classifier=RandomForestClassifier(n_estimators= 10,criterion="entropy")
classifier.fit(x_train, y_train)
```

Here, the classifier object takes the following parameters:

1. **n_estimators:** The required number of trees in the Random Forest. The default value is 10.
2. **criterion:** It is a function to analyse the accuracy of the split.

Output

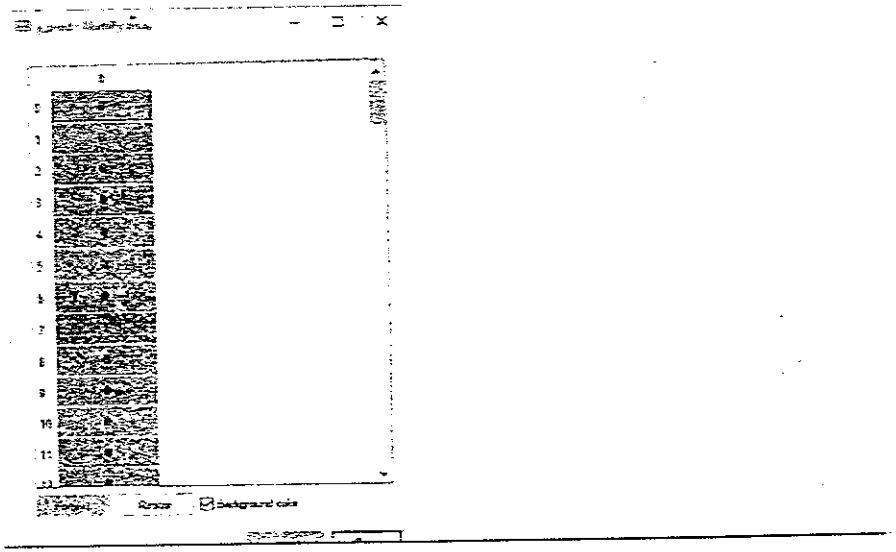
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy'
                       max_depth=None,max_features='auto',max_leaf_nodes=None,
                       min_impurity_decrease=0.0,min_impurity+split=None,
                       min_samples_leaf=1,min_samples_split=2,
                       min_weight_fraction_leaf=0.0,n_estimators=10,
                       n_jobs=None,oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

3. Predicting the Test Set result:

```
#Predicting the test set result
y_pred=classifier predict(x_test)
```

Output

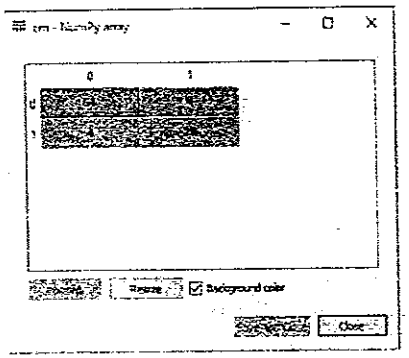
The Prediction vector is given as



4. Creating the Confusion Matrix

```
#Creating the Confusion matrix
from sklearn metrics import confusion_matrix
cm = confusion_maturity_test, y_pred)
```

Output



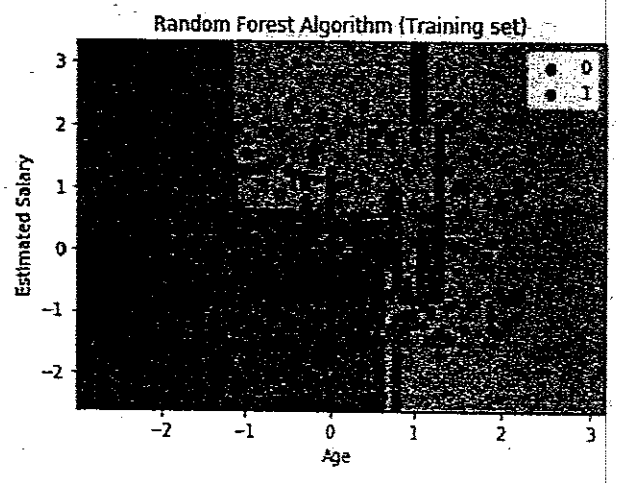
5. Visualizing the training Set result

```
from matplotlib.colors import Listed Colormap
x_set, y_set=x_train, y_train
```



```
x1,x2= nm.meshgris(nm.arange(start = x_set[:,0]min[ ] –1,stop = x_set[:,0]max( ) +
1,step = 0.01),
nm'arrange(start=x_set[;< 1],min( ) –1,stop = x_set[;,1]max( ) + 1, step = 0.01))
mtp.contourf(x1,x2,classifier.predict(nm.array((x1.ravel( ), x2.ravel( ),x2ravel(
))).T).reshape(x1.shape),
alpha = 0.75,cmap= ListedColormap(('purple','green')))
mtp.xlim(x1.min( ),x1.max( ))
mtp.ylim(x2.min( ),x2.max( ))
for i,j in enumerate(nm.unique(y_set)):
    mtp.scatter(x_setly_set = = j,0],x_setly_set = = j,1)
        c = ListedColormap(('purple','green'))(i),label = j)
mtp title('Random Forest Algorithm(Training set)')
mtp.xlabel('Age')
mtp.ylabel('Estimated Salary')
mtp.legend( )
mtp.show( )
```

Output



6. Visualizing the Test Set Result

```
#Visualizing the test set result
from matplotlib.colors import ListedColormap
```

```
x_set, y_set = x_test, y_test
x1, x2 = nmmeshgris(nmarrange(start = x_set[:,0]min( ) –1, stop=x_set[:,0]max ( ) +
1, step = 0.01),
nm.arrange(start = x_set[:, 1],min( ) – 1, stop = x_set[:, 1]max( ) + 1, step = 0.01))
mtp.conto arf(x1,x2,classifier.predict(nm.array([x1.ravel( ),x2 ravel(
)])T).reshape(x1:shape),
mtp.xlim(x1.min( ), x1max ( ))
mtp.ylim(x2.min( ),x2.max( ))
for i,j in enumerate (nm,uniquely_set)):
    mtp.scatter(x_set[yy_set = = j,1],
        c = ListedColormap(('purple', 'green'))(i), label = j)
mtp title('Random Forest Algorithm(Test set)')
mtp.xlabel('Age')
mtp.ylabel('Estimated Salary')
mtp.legend( )
mtp.show( )
```

3.12.7. APPLICATIONS OF RANDOM FOREST

There are mainly four sectors where Random Forest is mostly used:

1. **Banking:** The banking sector mostly uses this algorithm for the identification of loan risk.
2. **Medicine:** With the help of this algorithm, disease trends and risks of the disease can be identified.
3. **Land Use:** We can identify the areas of similar land use by this algorithm.
4. **Marketing:** Marketing trends can be identified using this algorithm.

3.12.8. ADVANTAGES OF RANDOM FOREST

- Random Forest is capable of performing both Classification and Regression tasks.
- It is capable of handling large datasets with high dimensionality.
- It enhances the accuracy of the model and prevents the overfitting issue.

3.12.9. DISADVANTAGES OF RANDOM FOREST

- Although random forest can be used for both classification and regression tasks, it is not more suitable for Regression tasks.

3.12.10. WHEN TO AVOID USING RANDOM FORESTS?

Random Forests Algorithms are not ideal in the following situations:

1. **Extrapolation:** Random Forest regression is not ideal in the extrapolation of data. Unlike linear regression, it uses existing observations to estimate values beyond the observation range.
2. **Sparse Data:** Random Forest does not produce good results when the data is sparse. In this case, the subject of features and bootstrapped sample will have an invariant space. This will lead to unproductive spills, which will affect the outcome.

TWO MARKS QUESTIONS WITH ANSWERS (PART - A)

1. ***What are the Applications of Supervised Machine Learning in Modern Businesses?***

Applications of supervised machine learning include:

Email Spam Detection

Here we train the model using historical data that consists of emails categorized as spam or not spam. This labeled information is fed as input to the model.

Healthcare Diagnosis

By providing images regarding a disease, a model can be trained to detect if a person is suffering from the disease or not.

Sentiment Analysis

This refers to the process of using algorithms to mine documents and determine whether they're positive, neutral, or negative in sentiment.

Fraud Detection

By training the model to identify suspicious patterns, we can detect instances of possible fraud.

2. ***What Is 'naive' in the Naive Bayes Classifier?***

The classifier is called 'naive' because it makes assumptions that may or may not turn out to be correct. The algorithm assumes that the presence of one feature of a class is not related to the presence of any other feature (absolute independence of features), given the class variable. For instance, a fruit may be considered to be a cherry if it is red in color and round in shape, regardless of other features. This assumption may or may not be right (as an apple also matches the description).

3. ***What is a Random Forest?***

A 'random forest' is a supervised machine learning algorithm that is generally used for classification problems. It operates by constructing multiple decision trees during the training phase. The random forest chooses the decision of the majority of the trees as the final decision.

4. ***What is Bias and Variance in a Machine Learning Model?***

Bias

Bias in a machine learning model occurs when the predicted values are further from the actual values. Low bias indicates a model where the prediction values are very close to the actual ones.

Underfitting:

High bias can cause an algorithm to miss the relevant relations between features and target outputs.

Variance

Variance refers to the amount the target model will change when trained with different training data. For a good model, the variance should be minimized.

Overfitting:

High variance can cause an algorithm to model the random noise in the training data rather than the intended outputs.

5. ***What is a Decision Tree Classification?***

A decision tree builds classification (or regression) models as a tree structure, with datasets broken up into ever-smaller subsets while developing the decision tree, literally in a tree-like way with branches and nodes. Decision trees can handle both categorical and numerical data.

6. ***What is Pruning in Decision Trees, and How Is It Done?***

Pruning is a technique in machine learning that reduces the size of decision trees. It reduces the complexity of the final classifier, and hence improves predictive accuracy by the reduction of overfitting.

Pruning can occur in:

- **Top-down fashion.** It will traverse nodes and trim subtrees starting at the root
- **Bottom-up fashion.** It will begin at the leaf nodes

There is a popular pruning algorithm called reduced error pruning, in which:

- Starting at the leaves, each node is replaced with its most popular class
- If the prediction accuracy is not affected, the change is kept
- There is an advantage of simplicity and speed

7. ***Briefly Explain Logistic Regression.***

Logistic regression is a classification algorithm used to predict a binary outcome for a given set of independent variables. The output of logistic regression is either a 0 or 1 with a threshold value of generally 0.5. Any value above 0.5 is considered as 1, and any point below 0.5 is considered as 0.

8. ***What is Kernel SVM?***

Kernel SVM is the abbreviated version of the kernel support vector machine. Kernel methods are a class of algorithms for pattern analysis, and the most common one is the kernel SVM.

9. ***How does the Support Vector Machine algorithm handle self-learning?***

The SVM algorithm has a learning rate and expansion rate which takes care of self-learning. The learning rate compensates or penalizes the hyperplanes for making all the incorrect moves while the expansion rate handles finding the maximum separation area between different classes.

10. ***What are the assumptions you need to take before starting with linear regression?***

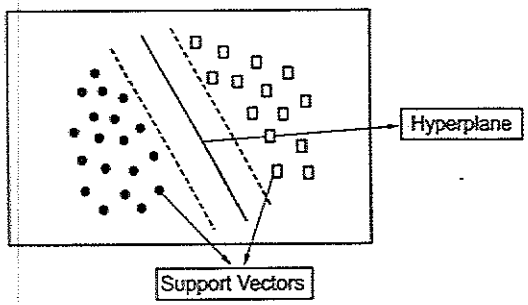
There are primarily 5 assumptions for a Linear Regression model:

- Multivariate normality
- No auto-correlation

- Homoscedasticity
- Linear relationship
- No or little multicollinearity

11. ***What are Support Vectors in SVM?***

Support Vectors are data points that are nearest to the hyperplane. It influences the position and orientation of the hyperplane. Removing the support vectors will alter the position of the hyperplane. The support vectors help us build our support vector machine model.



12. ***Define joint probability distribution***

This completely specifies an agent's probability assignments to all propositions in the domain. The joint probability distribution $p(x_1 x_2 ,..., x_n)$ assigns probabilities to all possible atomic events; where $X_1 ,X_2, ..., X_n$ 10 = variables.

13. ***What is Maximum – Likelihood hypotheses?***

ML – it is reasonable approach when there is no reason to prefer one hypotheses over another a prior

14. ***Define Naïve Bayes model.***

In this model, the "class" variable C is the root and the "attribute" variable XI are the leaves. This model assumes that the attributes are conditionally independent of each other, given the class.

15. ***Define sum of squared errors.***

The difference between the actual value yj and the predicated value ($\theta_1 x_j +$ θ_2) sc E is the sum of squared errors

16. ***What is the difference between Gini Impurity and Entropy in a Decision Tree?***

- **Gini Impurity and Entropy** are the metrics used for deciding how to split a Decision Tree.
- **Gini measurement** is the probability of a random sample being classified correctly if you randomly pick a label according to the distribution in the branch.
- **Entropy** is a measurement to calculate the lack of information. You calculate the Information Gain (difference in entropies) by making a split. This measure helps to reduce the uncertainty about the output label.

17. ***What is the difference between Entropy and Information Gain?***

- Entropy is an indicator of how messy your data is. It decreases as you reach closer to the leaf node.
- The Information Gain is based on the decrease in entropy after a dataset is split on an attribute. It keeps on increasing as you reach closer to the leaf node.

REVIEW QUESTIONS

1. Classify the machine learning algorithms with an example.
2. What are the categories of supervised machine learning algorithm.
3. Compare Supervised and Unsupervised Machine Learning algorithms.
4. What are the types of linear regression and explain in detail.
5. Explain in detail about gradient descent.
6. Construct the classifiers based on discriminant functions.
7. Define Fisher's linear Discriminant and explain in detail.
8. Explain in detail the probabilistic generative model.
9. Briefly explain the Naïve Bayes algorithm for solving the classification problem.
10. Explain in detail about the support vector machine with an example.
11. Compare decision tree & random forest.

****

UNIT IV

ENSEMBLE TECHNIQUES AND UNSUPERVISED LEARNING

Combining multiple learners: Model combination schemes, Voting, Ensemble Learning - bagging, boosting, stacking, Unsupervised learning: K-means, Instance Based Learning: KNN, Gaussian mixture models and Expectation maximization

4.1. INTRODUCTION

Generally, we develop machine learning models to solve a problem by using the given input data. When we work on a single algorithm, we are unable to distinguish the performance of the model for that particular statement, as there is nothing to compare it against. So, we feed the input data to other machine learning algorithms and then compare them with each other to know which the best algorithm that suits the given problem is. Every algorithm has its own mathematical computation and significance to deal with a specific problem to bring out the best results at the end.

4.2. COMBINING MULTIPLE LEARNERS

Why do we combine models?

While dealing with a specific problem with a machine learning algorithm we sometimes fail, because of the poor performance of the model. The algorithm that we have used may be well suited to the model, but we still fail in getting better outcomes at the end. In this situation, we might have many questions in our mind. How can we bring out better results from the model? What are the steps to be taken further in the model development? What are the hidden techniques that can help to develop an efficient model?

To overcome this situation there is a procedure called "Combining Models", where we mix one or two weaker machine learning models to solve a problem and get better outcomes. In machine learning, the combining of models is done by using two approaches namely "Ensemble Models" & "Hybrid Models".

Ensemble Models use multiple machine learning algorithms to bring out better predictive results, as compared to using a single algorithm. There are different approaches in Ensemble models to perform a particular task. There is another model called Hybrid model that is flexible and helps to create a more innovative model than an Ensemble model. While combining models we need to check how strong or weak a particular machine learning model is, to deal with a particular problem.

4.3. MODEL COMBINATION SCHEMES

There are also different ways the multiple base-learners are combined to generate the final output:

Multi expert combination methods have base-learners that work in parallel. These methods can in turn be divided into two:

- In the global approach, also called learner fusion, given an input, all base-learners generate an output and all these outputs are used. Examples are voting and stacking.
- In the local approach, or learner selection, for example, in mixture of experts, there is a gating model, which looks at the input and chooses one (or very few) of the learners as responsible for generating the output.

Multistage combination methods use a serial approach where the next base-learner is trained with or tested on only the instances where the previous base-learners are not accurate enough. The idea is that the base-learners (or the different representations they use) are sorted in increasing complexity so that a complex base-learner is not used (or its complex representation is not extracted) unless the preceding simpler base-learners are not confident. An example is cascading.

Let us say that we have L base-learners. We denote by $d_j(x)$ the prediction of base-learner M_j given the arbitrary dimensional input x. In the case of multiple representations, each M_j uses a different input representation x_j. The final prediction is calculated from the predictions of the base-learners:

$$y = f(d_1, d_2, \ldots d_L \mid \Phi)$$

where $f(\cdot)$ is the combining function with Φ denoting its parameters. When there are K outputs, for each learner there are $d_{ji}(x)$, $i = 1,\ldots,K, j = 1,\ldots,L$, and combining them, we also generate K values, y_i, $i = 1,\ldots,K$ and then for example in classification, we choose the class with the maximum y_i value:

$$\text{Choose } C_i \text{ if } y_i = \max_{k=1}^{K} y_k$$

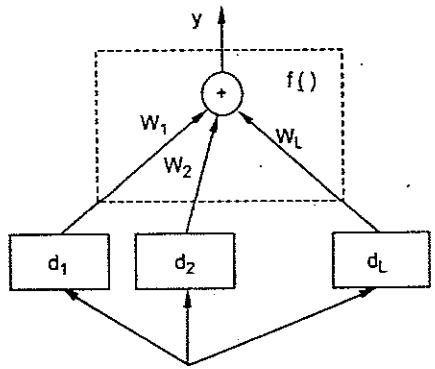


Fig. 4.1.

Figure 4.1 Base-learners are d_j and their outputs are combined using $f(\cdot)$. This is for a single output; in the case of classification, each base-learner has K outputs that are separately used to calculate y_i, and then we choose the maximum. Note that here, all learners observe the same input; it may be the case that different learners observe different representations of the same input object or event.

4.4. VOTING

Voting is an ensemble method that combines the performances of multiple models to make predictions. In this technique, the first step is to create multiple classification models using a training dataset. When the voting is applied to regression problems, the prediction is made with the average of multiple other regression models.

In the case of classification there are two types of voting,

- Hard Voting
- Soft Voting

The Hard Voting ensemble involves summing up the votes for crisp class labels from other models and predicting the class with the most votes. Soft Voting ensemble involves summing up the predicted probabilities for class labels and predicting the class label with the largest sum probability.

1. **Hard Voting:** In hard voting, the predicted output class is a class with the highest majority of votes i.e the class which had the highest probability of being predicted by each of the classifiers. Suppose three classifiers predicted the output class(A, A, B), so here the majority predicted A as output. Hence A will be the final prediction.
2. **Soft Voting:** In soft voting, the output class is the prediction based on the average of probability given to that class.

Suppose given some input to three models, the prediction probability for class A = (0.30, 0.47, 0.53) and B = (0.20, 0.32, 0.40). So the average for class A is 0.4333 and B is 0.3067, the winner is clearly class A because it had the highest probability averaged by each classifier.

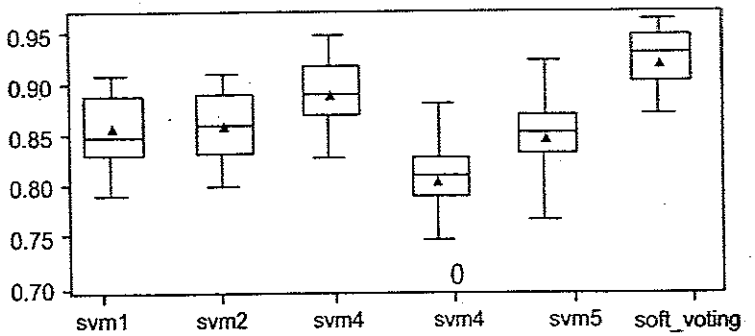


Fig. 4.2.

In short, for the Regression voting ensemble the predictions are the averages of contributing models, whereas for Classification voting ensemble, the predictions are the majority vote of contributing models.

There are other forms of voting like "Majority Voting" and "Weighted Voting". In the case of Majority Voting, the final output predictions are based on the number of votes it gets. If the count of votes is high, that model is taken into consideration. In some of the articles this method is also called as "Plurality Voting".

Unlike the technique of Majority voting, the weighted voting works based on the weights to increase the importance of one or more models. In the case of weighted voting, we count the prediction of the better models multiple times.

Note Make sure to include a variety of models to feed a Voting Classifier to be sure that the error made by one might be resolved by the other.

Code : Python code to implement Voting Classifier

```
# importing libraries
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

# loading iris dataset
iris = load_iris()
X = iris.data[:, :4]
Y = iris.target

# train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,
                                   Y,
                                   test_size = 0.20,
                                   random_state = 42)

# group / ensemble of models
estimator = []
estimator.append(('LR',
        LogisticRegression(solver ='lbfgs',
```

```
                    multi_class ='multinomial',
                    max_iter = 200)))
estimator.append(('SVC', SVC(gamma ='auto', probability = True)))
estimator.append(('DTC', DecisionTreeClassifier()))
 # Voting Classifier with hard voting
vot_hard = VotingClassifier(estimators = estimator, voting ='hard')
vot_hard.fit(X_train, y_train)
y_pred = vot_hard.predict(X_test)

# using accuracy_score metric to predict accuracy
score = accuracy_score(y_test, y_pred)
print("Hard Voting Score % d" % score)
 # Voting Classifier with soft voting
vot_soft = VotingClassifier(estimators = estimator, voting ='soft')
vot_soft.fit(X_train, y_train)
y_pred = vot_soft.predict(X_test)
 # using accuracy_score
score = accuracy_score(y_test, y_pred)
print("Soft Voting Score % d" % score)
```

Output :

```
Hard Voting Score 1
Soft Voting Score 1
```

Examples:

Input : 4.7, 3.2, 1.3, 0.2
Output : Iris Setosa

In practical the output accuracy will be more for soft voting as it is the average probability of the all estimators combined, as for our basic iris dataset we are already overfitting, so there won't be much difference in output.

4.5. ENSEMBLE LEARNING

When you want to purchase a new car, will you walk up to the first car shop and purchase one based on the advice of the dealer? It's highly unlikely.

You would likely browser a few web portals where people have posted their reviews and compare different car models, checking for their features and prices. You will also probably ask your friends and colleagues for their opinion. In short, you wouldn't directly reach a conclusion, but will instead make a decision considering the opinions of other people as well.

Ensemble models in machine learning operate on a similar idea. They combine the decisions from multiple models to improve the overall performance.

Let's understand the concept of ensemble learning with an example. Suppose you are a movie director and you have created a short movie on a very important and interesting topic. Now, you want to take preliminary feedback (ratings) on the movie before making it public. What are the possible ways by which you can do that?

A: You may ask one of your friends to rate the movie for you.

Now it's entirely possible that the person you have chosen loves you very much and doesn't want to break your heart by providing a 1-star rating to the horrible work you have created.

B: Another way could be by asking 5 colleagues of yours to rate the movie.

This should provide a better idea of the movie. This method may provide honest ratings for your movie. But a problem still exists. These 5 people may not be "Subject Matter Experts" on the topic of your movie. Sure, they might understand the cinematography, the shots, or the audio, but at the same time may not be the best judges of dark humour.

C: How about asking 50 people to rate the movie?

Some of which can be your friends, some of them can be your colleagues and some may even be total strangers.

The responses, in this case, would be more generalized and diversified since now you have people with different sets of skills. And as it turns out – this is a better approach to get honest ratings than the previous cases we saw.

With these examples, you can infer that a diverse group of people are likely to make better decisions as compared to individuals. Similar is true for a diverse set of models in comparison to single models. This diversification in Machine Learning is achieved by a technique called Ensemble Learning.

An Ensemble is made up of things that are grouped together, that take up a particular task. This method combines several algorithms together to bring out better predictive results, as compared to using a single algorithm. The objective behind the usage of an Ensemble method is that it decreases variance, bias and improves predictions in a developed model. Technically speaking, it helps in avoiding overfitting.

The models that contribute to an Ensemble are referred to as the Ensemble Members, which may be of the same type or different types, and may or may not be trained on the same training data.

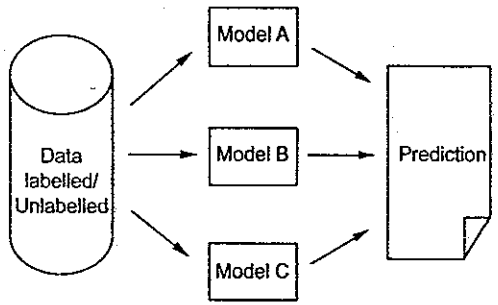


Fig. 4.3. Ensemble model

In the late 2000s, adoption of ensembles picked up due in part to their huge success in machine learning competitions, such as the Netflix Prize and other competitions on Kaggle.

These ensemble methods greatly increase the computational cost and complexity of the model. This increase comes from the expertise and time required to train and maintain multiple models rather than a single model.

Ensemble models are preferred because of two main reasons; namely Performance & Robustness. The ensemble methods majorly focus on improving the accuracy of the model by reducing variance component of the prediction error and by adding bias to the model.

Performance helps a Machine Learning model to make better predictions. Robustness reduces the spread or dispersion of the prediction and model performance.

The goal of a supervised machine learning algorithm is to have "low bias and low variance".

The Bias and the Variance are inversely proportional to each other i.e., if the bias is low then the variance is high, else the bias is high then the variance is low.

We explicitly use ensemble methods to seek better predictive performance, such as lower error on regression or higher accuracy for classification. They are also further used in Computer vision and are given utmost importance in academic competitions also.

Simple Ensemble Techniques

We will look at a few simple but powerful techniques, namely:

1. Max Voting
2. Averaging
3. Weighted Averaging

4.5.1. MAX VOTING

The max voting method is generally used for classification problems. In this technique, multiple models are used to make predictions for each data point. The predictions by each model are considered as a 'vote'. The predictions which we get from the majority of the models are used as the final prediction.

For example, when you asked 5 of your colleagues to rate your movie (out of 5); we'll assume three of them rated it as 4 while two of them gave it a 5. Since the majority gave a rating of 4, the final rating will be taken as 4. **You can consider this as taking the mode of all the predictions.**

The result of max voting would be something like this:

Colleague 1	Colleague 2	Colleague 3	Colleague 4	Colleague 5	Final rating
5	4	5	4	4	4

Sample Code:

Here *x_train* consists of independent variables in training data, *y_train* is the target variable for training data. The validation set is *x_test* (independent variables) and *y_test* (target variable).

you can use "VotingClassifier" module in sklearn as follows:

```
from sklearn.ensemble import VotingClassifier
model1 = LogisticRegression(random_state=1)
model2 = tree.DecisionTreeClassifier(random_state=1)
model = VotingClassifier(estimators=[('lr', model1), ('dt', model2)], voting='hard')
model.fit(x_train,y_train)
model.score(x_test,y_test)
```

4.5.2. AVERAGING

Similar to the max voting technique, multiple predictions are made for each data point in averaging. In this method, we take an average of predictions from all the models and use it to make the final prediction. Averaging can be used for making predictions in regression problems or while calculating probabilities for classification problems.

For example, in the below case, the averaging method would take the average of all the values.

i.e. (5 + 4 + 5 + 4 + 4) / 5 = 4.4

Colleague 1	Colleague 2	Colleague 3	Colleague 4	Colleague 5	Final rating
5	4	5	4	4	4.4

Sample Code:

```
model1 = tree.DecisionTreeClassifier()
model2 = KNeighborsClassifier()
model3= LogisticRegression()
model1.fit(x_train,y_train)
model2.fit(x_train,y_train)
model3.fit(x_train,y_train)
```



```
pred1=model1.predict_proba(x_test)
pred2=model2.predict_proba(x_test)
pred3=model3.predict_proba(x_test)
finalpred=(pred1+pred2+pred3)/3
```

4.5.3. WEIGHTED AVERAGE

This is an extension of the averaging method. All models are assigned different weights defining the importance of each model for prediction. For instance, if two of your colleagues are critics, while others have no prior experience in this field, then the answers by these two friends are given more importance as compared to the other people.

The result is calculated as [(5 × 0.23) + (4 × 0.23) + (5 × 0.18) + (4 × 0.18) + (4 × 0.18)] = 4.41.

	Colleague 1	Colleague 2	Colleague 3	Colleague 4	Colleague 5	Final rating
Weight	0.23	0.23	0.18	0.18	0.18	
Rating	5	4	5	4	4	4.41

Sample Code:

```
model1 = tree.DecisionTreeClassifier()
model2 = KNeighborsClassifier()
model3= LogisticRegression()

model1.fit(x_train,y_train)
model2.fit(x_train,y_train)
model3.fit(x_train,y_train)
pred1=model1.predict_proba(x_test)
pred2=model2.predict_proba(x_test)
pred3=model3.predict_proba(x_test)
finalpred=(pred1*0.3+pred2*0.3+pred3*0.4)
```

Advanced Ensemble Techniques

1. Stacking
2. Bagging
3. Boosting

4.6. STACKING

Stacking is one of the popular ensemble modeling techniques in machine learning. Various weak learners are ensembled in a parallel manner in such a way that by combining them with Meta learners, we can predict better predictions for the future.

This ensemble technique works by applying input of combined multiple weak learners' predictions and Meta learners so that a better output prediction model can be achieved.

In stacking, an algorithm takes the outputs of sub-models as input and attempts to learn how to best combine the input predictions to make a better output prediction.

Stacking is also known as **a stacked generalization** and is an extended form of the Model Averaging Ensemble technique in which all sub-models equally participate as per their performance weights and build a new model with better predictions. This new model is stacked up on top of the others; this is the reason why it is named stacking.

Stacking is an ensemble learning technique that uses predictions from multiple models (for example decision tree, knn or svm) to build a new model. This model is used for making predictions on the test set.

4.6.1. ARCHITECTURE OF STACKING

The architecture of the stacking model is designed in such as way that it consists of two or more base/learner's models and a meta-model that combines the predictions of the base models. These base models are called level 0 models, and the meta-model is known as the level 1 model. So, the Stacking ensemble method includes original (training) data, primary level models, primary level prediction, secondary level model, and final prediction. The basic architecture of stacking can be represented as shown below the image.

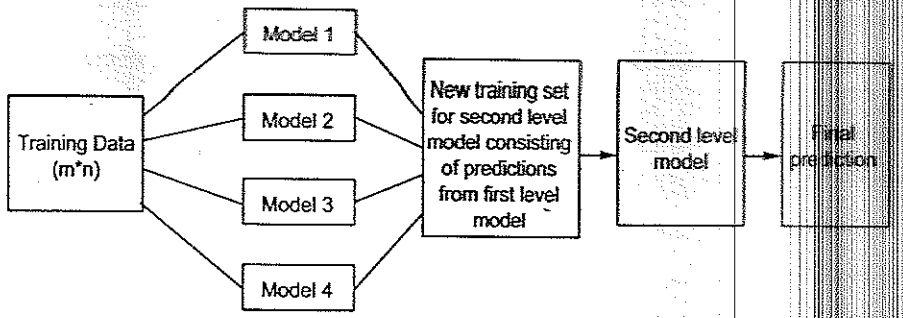


Fig. 4.4. Architecture of stacking

- **Original data:** This data is divided into n-folds and is also considered test data or training data.
- **Base models:** These models are also referred to as level-0 models. These models use training data and provide compiled predictions (level-0) as an output.
- **Level-0 Predictions:** Each base model is triggered on some training data and provides different predictions, which are known as **level-0 predictions.**
- **Meta Model:** The architecture of the stacking model consists of one meta-model, which helps to best combine the predictions of the base models. The meta-model is also known as the **level-1 model.**
- **Level-1 Prediction:** The meta-model learns how to best combine the predictions of the base models and is trained on different predictions made by individual base models, i.e., data not used to train the base models are fed to the meta-model, predictions are made, and these predictions, along with the expected outputs, provide the input and output pairs of the training dataset used to fit the meta-model.

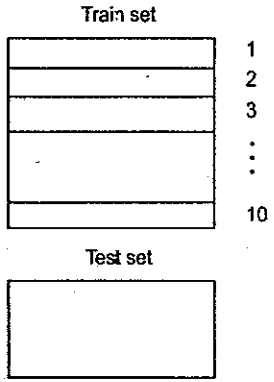
Meta-Model is made up of the predictions of a set of ML base models (i.e. weak learners) through the k-fold cross validation technique. Finally, the **Meta-Model** is trained with an additional ML model (which is commonly known as the "final estimator" or "final learner").

The **Stacking Generalization** method is commonly composed of 2 training stages, better known as "level 0" and "level 1". It is important to mention that it can be added as many levels as necessary. However, in practice it is common to use only

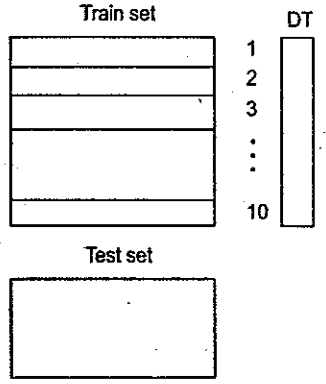
2 levels. The aim of the first stage (level 0) is to generate the training data for the meta-model, this is carried out by implementing k-fold cross validation for each "weak learner" defined in the first stage. The predictions of each one of these "weak learners" are "stacked" in order to build such "new training set" (the meta-model). The aim of the second stage (level 1) is to train the meta-model, such training is carried out through an already determined "final learner".

Below is a step-wise explanation for a simple stacked ensemble:

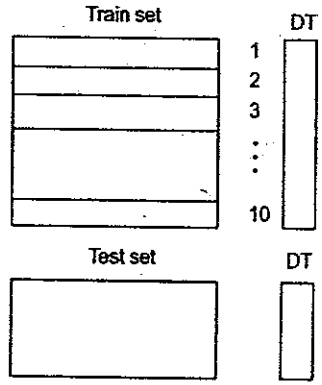
1. The train set is split into 10 parts.



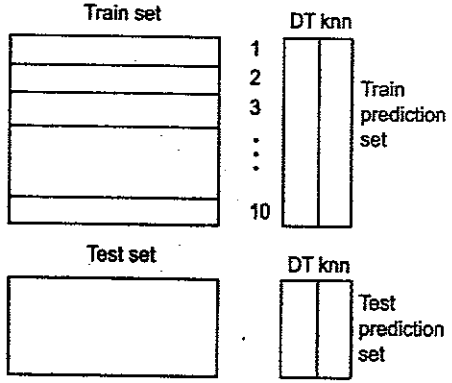
2. A base model (suppose a decision tree) is fitted on 9 parts and predictions are made for the 10th part. This is done for each part of the train set.



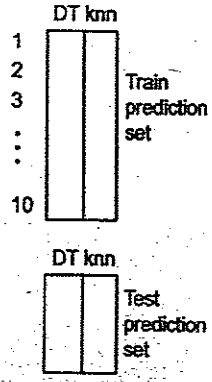
3. The base model (in this case, decision tree) is then fitted on the whole train dataset.
4. Using this model, predictions are made on the test set.



5. Steps 2 to 4 are repeated for another base model (say knn) resulting in another set of predictions for the train set and test set.



6. The predictions from the train set are used as features to build a new model.



7. This model is used to make final predictions on the test prediction set.

Sample Code:

We first define a function to make predictions on n-folds of train and test dataset. This function returns the predictions for train and test for each model.

```
def Stacking(model,train,y,test,n_fold):
  folds=StratifiedKFold(n_splits=n_fold,random_state=1)
  test_pred=np.empty((test.shape[0],1),float)
  train_pred=np.empty((0,1),float)
  for train_indices,val_indices in folds.split(train,y.values):
    x_train,x_val=train.iloc[train_indices],train.iloc[val_indices]
    y_train,y_val=y.iloc[train_indices],y.iloc[val_indices]

    model.fit(X=x_train,y=y_train)
    train_pred=np.append(train_pred,model.predict(x_val))
    test_pred=np.append(test_pred,model.predict(test))
  return test_pred.reshape(-1,1),train_pred
```

Now we'll create two base models – decision tree and knn.

```
model1 = tree.DecisionTreeClassifier(random_state=1)
test_pred1 ,train_pred1=Stacking(model=model1,n_fold=10,
train=x_train,test=x_test,y=y_train)
train_pred1=pd.DataFrame(train_pred1)
test_pred1=pd.DataFrame(test_pred1)
model2 = KNeighborsClassifier()
test_pred2,train_pred2=Stacking(model=model2,n_fold=10,train=x_train,test=x_t
est,y=y_train)
train_pred2=pd.DataFrame(train_pred2)
test_pred2=pd.DataFrame(test_pred2)
```


Create a third model, logistic regression, on the predictions of the decision tree and knn models.

```
df = pd.concat([train_pred1, train_pred2], axis=1)
df_test = pd.concat([test_pred1, test_pred2], axis=1)
model = LogisticRegression(random_state=1)
model.fit(df,y_train)
model.score(df_test, y_test)
```

In order to simplify the above explanation, the stacking model we have created has only two levels. The decision tree and knn models are built at level zero, while a logistic regression model is built at level one. Feel free to create multiple levels in a stacking model.

In figure 4.5 we see a graphical description of an architecture of a Stacking Generalization Classifier that is composed of 3 base models (weak learners) and a final estimator.

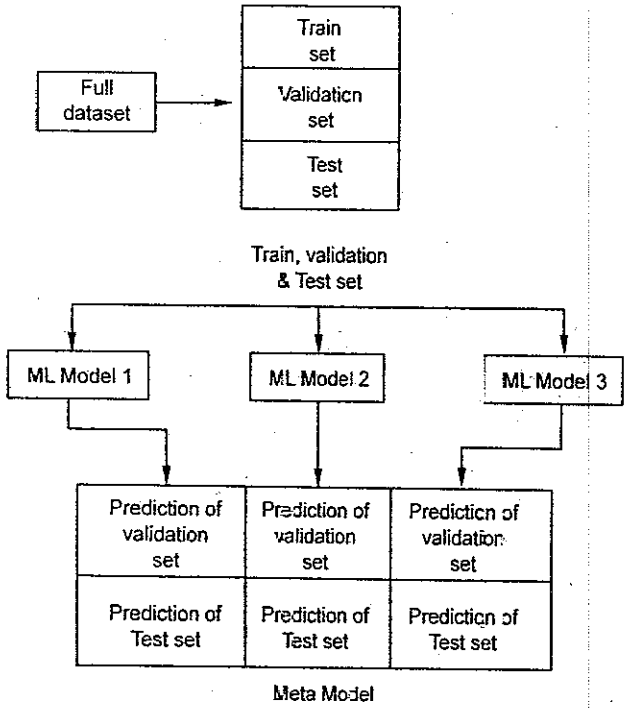


Fig. 4.5. Architecture of a Stacking Generalization Classifier

Code Snippet 1. Stacking Generalization function

```
    def StackingClassifier(self):
3       # Define weak learners
4       weak_learners = [('dt', DecisionTreeClassifier()),
5       ('knn', KNeighborsClassifier()),
6       ('rf', RandomForestClassifier()),
7       ('gb', GradientBoostingClassifier()),
8       ('gn', GaussianNB())]
9
10      # Finaler learner or meta model
11      final_learner = LogisticRegression()
12
13      train_meta_model = None
14      test_meta_model = None
15
16      # Start stacking
17      for clf_id, clf in weak_learners:
18      # Predictions for each classifier based on k-fold
19      predictions_clf = self.k_fold_cross_validation(clf)
20
21  # Predictions for test set for each classifier based on train of level 0
22      test_predictions_clf = self.train_level_0(clf)
23
24      # Stack predictions which will form
25      # the inputa data for the data model
26      if isinstance(train_meta_model, np.ndarray):
27      train_meta_model = np.vstack((train_meta_model,
    predictions_clf))
28      else:
29      train_meta_model = predictions_clf
```



```
30      # Stack predictions from test set
31      # which will form test data for meta model
32      if isinstance(test_meta_model, np.ndarray):
33      test_meta_model = np.vstack((test_meta_model,
    test_predictions_clf))
34      else:
35      test_meta_model = test_predictions_clf
36
37      # Transpose train_meta_model
38      train_meta_model = train_meta_model.T
39
40      # Transpose test_meta_model
41      test_meta_model = test_meta_model.T
42
43      # Training level 1
44      self.train_level_1(final_learner, train_meta_model,
    test_meta_model)
45      # Stack predictions from test set
```

Let's analyze the key parts, in line 4 we are defining 5 classifiers (weak learners) that will be the base models of our stack (which are trained at level 0). In line 11 we define the final classifier (which is the meta-model classifier). Now, level 0 training begins with the for loop defined in line 17. As we can see, in line 19 we are receiving the predictions of k-fold cross validation and in line 26 we are "stacking" these predictions (the which are forming the training data of the meta-model). On line 22 we are receiving the predictions from the test set which are "stacked" on line 33 to form the meta-model test data. Finally, in line 45 we carry out the level 1 training, that is, the meta-model training.

As we mentioned, one of the key parts of this method is the use of the k-fold cross validation for the generation of the meta-model training data. However, there is a variation, we can omit k-fold cross validation and only use "*one-holdout set*", this small but significant variation is called "Blending".

4.7. BLENDING

Blending is a technique derived from Stacking Generalization. The only difference is that in Blending, the k-fold cross validation technique is not used to generate the training data of the meta-model. Blending implements "one-holdout set", that is, a small portion of the training data (validation) to make predictions which will be "stacked" to form the training data of the meta-model. Also, predictions are made from the test data to form the meta-model test data.

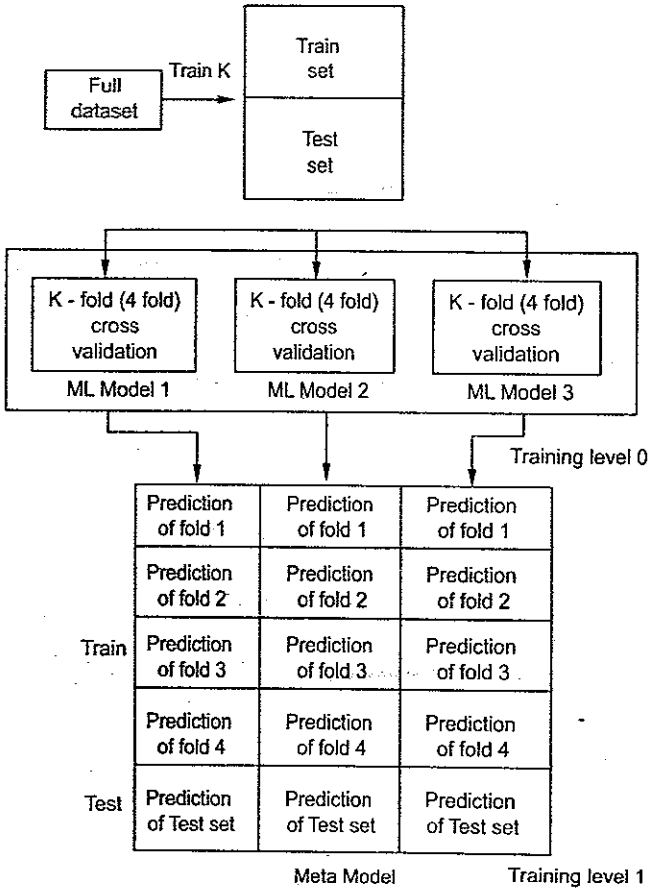


Fig. 4.6. Blending Architecture

In figure 4.6 we can see a Blending architecture using 3 base models (weak learners) and a final classifier. The blue boxes represent that portion of the training data that is used to generate predictions (yellow boxes) to form the meta-model.

The green boxes represent the test data which is used to generate predictions to form the meta-model test data (purple boxes).

Code Snippet 2: Blending Architecture

```
    def BlendingClassifier(self):
3       # Define weak learners
4       weak_learners = [('dt', DecisionTreeClassifier()),
5       ('knn', KNeighborsClassifier()),
6       ('rf', RandomForestClassifier()),
7       ('gb', GradientBoostingClassifier()),
8       ('gn', GaussianNB())]
9
10      # Finaler learner or meta model
11      final_learner = LogisticRegression()
12
13      train_meta_model = None
14      test_meta_model = None
15
16      # Start stacking
17      for clf_id, clf in weak_learners:
18
19      # Predictions for each classifier based on k-fold
20      val_predictions, test_predictions = self.train_level_0(clf)
21
22      # Stack predictions which will form
23      # the inputa data for the data model
24      if isinstance(train_meta_model, np.ndarray):
25      train_meta_model = np.vstack((train_meta_model,
        val_predictions))
26      else:
27      train_meta_model = val_predictions
```

```
28
29      # Stack predictions from test set
30      # which will form test data for meta model
31      if isinstance(test_meta_model, np.ndarray):
32          test_meta_model = np.vstack((test_meta_model,
        test_predictions))
33      else:
34          test_meta_model = test_predictions
35
36      # Transpose train_meta_model
37      train_meta_model = train_meta_model.T
38
39      # Transpose test_meta_model
40      test_meta_model = test_meta_model.T
41
42      # Training level 1
43      self.train_level_1(final_learner, train_meta_model,
        test_meta_model)
```

Let's analyze the key parts of this model. In line 4 we are defining the 5 base classifiers that we will use (weak learners), in line 11 we define the final classifier, as in the previous example, we will use Logistic Regression. Level 0 training begins on line 17. As we can see, on line 20 we are receiving the predictions of the validation set (which will form the training data of the meta-model) and the predictions of the test data (the which will form the meta-model test data). Also, on lines 24 and 31 we are "stacking" the predictions of each base classifier. Finally, on line 43 we are moving to level 1 training, and that is it!

As we can see, the Blending architecture is slightly simpler and more compact than Stack Generalization. Omitting *k*-fold cross validation can make us optimize the processing time.

4.8. VOTING

This type of ensemble is one of the most intuitive and easy to understand. The Voting Classifier is a homogeneous and heterogeneous type of Ensemble Learning, that is, the base classifiers can be of the same or different type. As mentioned earlier, this type of ensemble also works as an extension of bagging (e.g. Random Forest).

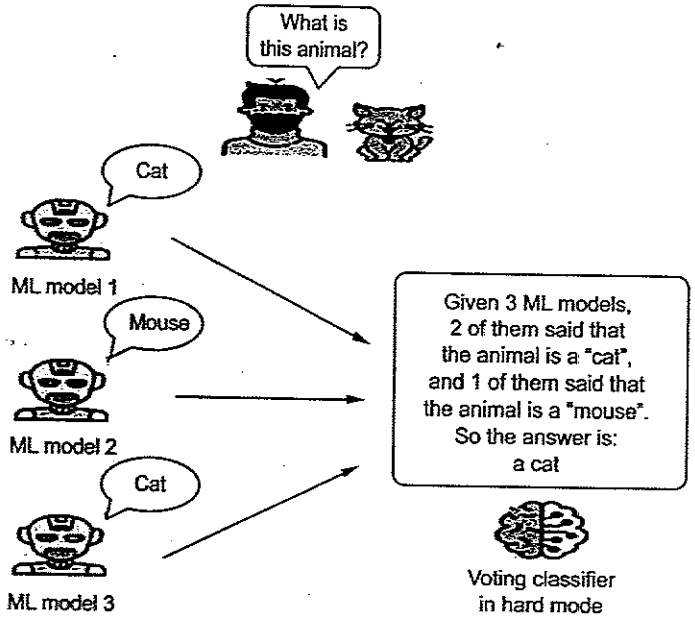
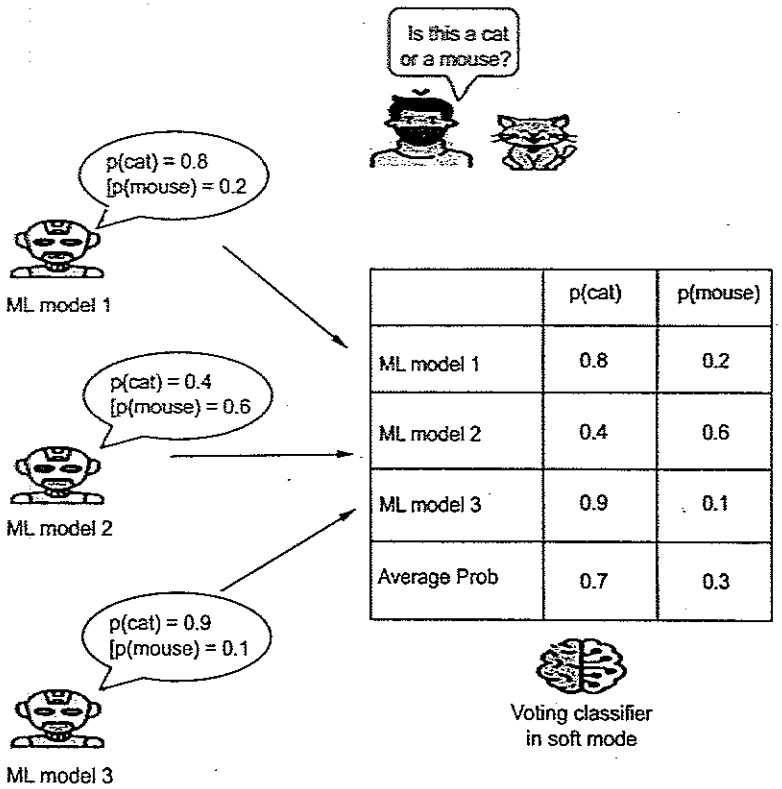


Fig. 4.7. Voting Classifier in "hard" mode

The architecture of a Voting Classifier is made up of a number "*n*" of ML models, whose predictions are valued in two different ways: hard and soft. In hard mode, the winning prediction is the one with "the most votes". In Figure 4.7 we see an example of how the Voting Classifier works in hard mode.

On the other hand, the Voting Classifier in soft mode considers the probabilities thrown by each ML model, these probabilities will be weighted and averaged, consequently the winning class will be the one with the highest weighted and averaged probability. In Figure 4.8 we see an example of how the Voting Classifier works in the soft mode.



	p(cat)	p(mouse)
ML model 1	0.8	0.2
ML model 2	0.4	0.6
ML model 3	0.9	0.1
Average Prob	0.7	0.3

Fig. 4.8. Voting Classifier in "soft" mode

Ok, now that we know how the Voting Classifier works, let's see how to do this in code. On this occasion, since it is a simple and intuitive ensemble technique (compared to Stacking or Blending), let's make use of the function provided by scikit-learn for the implementation of Voting, let's do it!

Code snippet 1. Voting Classifier

```
    Class Ensemble:
2       def __init__(self):
3           self.x_train = None
4           self.x_test = None
5           self.y_train = None
6           self.y_test = None
7
```



```
8       def load_data(self):
9           x, y = load_breast_cancer(return_X_y=True)
10          self.x_train, self.x_test, self.y_train, self.y_test =
        train_test_split(x, y, test_size=0.25, random_state=23)
11
12      @staticmethod
13      def __Classifiers__(name=None):
14          # See for reproducibility
15          random_state = 23
16
17          if name == 'decision_tree':
18              return DecisionTreeClassifier(random_state=random_state)
19          if name == 'kneighbors':
20              return KNeighborsClassifier()
21          if name == 'logistic_regression':
22              return LogisticRegression(random_state=random_state)
23
24      def __DecisionTreeClassifier__(self):
25
26          # Decision Tree Classifier
27          decision_tree = Ensemble.__Classifiers__(name='decision_tree')
28
29          # Train Decision Tree
30          decision_tree.fit(self.x_train, self.y_train)
31
32      def __KNearestNeighborsClassifier__(self):
33
34          # K-Nearest Neighbors Classifier
35          knn = Ensemble.__Classifiers__(name='kneighbors')
36
37          # Train K-Nearest Neighbos
```

```
38      knn.fit(self.x_train, self.y_train)
39
40      def __LogisticRegression__(self):
41
42      # Decision Tree Classifier
43      logistic_regression =
        Ensemble.__Classifiers__(name='logistic_regression')
44
45      # Train Logistic Regression
46      logistic_regression.fit(self.x_train, self.y_train)
47
48      def __VotingClassifier__(self):
49
50      # Instantiate classifiers
51      decision_tree = Ensemble.__Classifiers__(name='decision_tree')
52      knn = Ensemble.__Classifiers__(name='kneighbors')
53      logistic_regression =
        Ensemble.__Classifiers__(name='logistic_regression')
54
55      # Voting Classifier initialization
56      vc = VotingClassifier(estimators=[('decision_tree', decision_tree),
        ('knn', knn), ('logistic_regression', logistic_regression)], voting='soft')
57
58      # Train Voting Classifier
59      vc.fit(self.x_train, self.y_train)
```

In the code above we are creating a class which will contain different classifiers which are: Decision Tree, K-Nearest Neighbors, Logistic Regression and Voting Classifier (lines 24, 32, 40 and 48 respectively). To compare the effectiveness between "weak classifiers" and the Ensemble, we will make use of the "breast_cancer" toy dataset (line 8). We will use each classifier with its default values.

Next, we see Voting Classifier. As we can see, in lines from 51 to 53 we are defining the *weak classifiers* that we are going to use and, in a simple way, we are going to pass these classifiers as parameters to the Voting Classifier (line 56), in this example, we are using the "hard" mode. Finally, on line 59 we train the *Ensemble*, this is what we get:

Decision Tree, Train accuracy: 1

Decision Tree, Test accuracy: 0.958

K-Nearest Neighbors, Train accuracy: 0.930

K-Nearest Neighbors, Test accuracy: 0.946

Logistic Regression, Train accuracy: 0.953

Logistic Regression, Test accuracy: 0.951

Voting Classifier, Train accuracy: 0.981

Voting Classifier, Test accuracy: 0.965

As we can see, the Test accuracy of the Voting Classifier is slightly better than that of the weak classifiers. It is very important to mention that, although Voting Classifier is a great alternative to improve the accuracy of your models, it may not always be the best option due to various factors, including processing time.

4.9. BAGGING

Bagging, also known as bootstrap aggregating, is the aggregation of multiple versions of a predicted model. Each model is trained individually, and combined using an averaging process. The primary focus of bagging is to achieve less variance than any model has individually. To understand bagging, let's first understand the term bootstrapping.

4.9.1. BOOTSTRAPPING

Bootstrapping is the process of generating bootstrapped samples from the given dataset. The samples are formulated by randomly drawing the data points with replacement.

The resampled data contains different characteristics that are as a whole in the original data. It draws the distribution present in the data points, and also tend to remain different from each other, i.e. the data distribution has to remain intact whilst

maintaining the dissimilarity among the bootstrapped samples. This, in turn, helps in developing robust models.

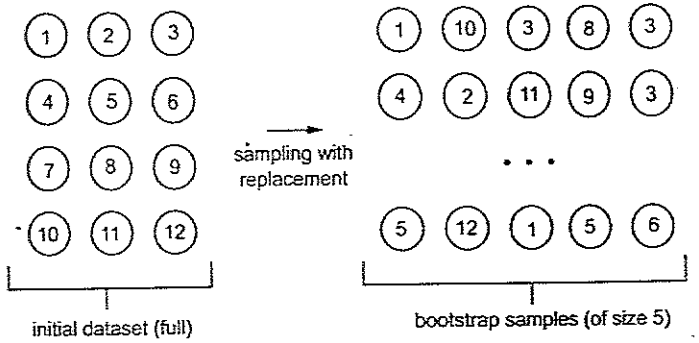


Fig. 4.9. Bootstrapping

Bootstrapping also helps to avoid the problem of overfitting. When different sets of training data are used in the model, the model becomes resilient to generating errors, and therefore, performs well with the test data, and thus reduces the variance by maintaining its strong foothold in the test data. Testing with different variations doesn't bias the model towards an incorrect solution.

Now, when we try to build fully independent models, it requires huge amounts of data. Thus, Bootstrapping is the option that we opt for by utilizing the approximate properties of the dataset under consideration.

4.9.2. BASIC CONCEPTS BEHIND BAGGING

It all started in the year 1994, when Leo Breiman proposed this algorithm, then known as "Bagging Predictors". In Bagging, the bootstrapped samples are first created. Then, either a regression or classification algorithm is applied to each sample. Finally, in the case of regression, an average is taken over all the outputs predicted by the individual learners. For classification either the most voted class is accepted (hard-voting), or the highest average of all the class probabilities is taken as the output (soft-voting). This is where aggregation comes into the picture.

Mathematically, Bagging is represented by the following formula,

$$\hat{f}_{bag} = \hat{f}_1(X) + \hat{f}_2(X) + \ldots + \hat{f}_b(X)$$

The term on the left hand side is the bagged prediction, and terms on the right hand side are the individual learners.

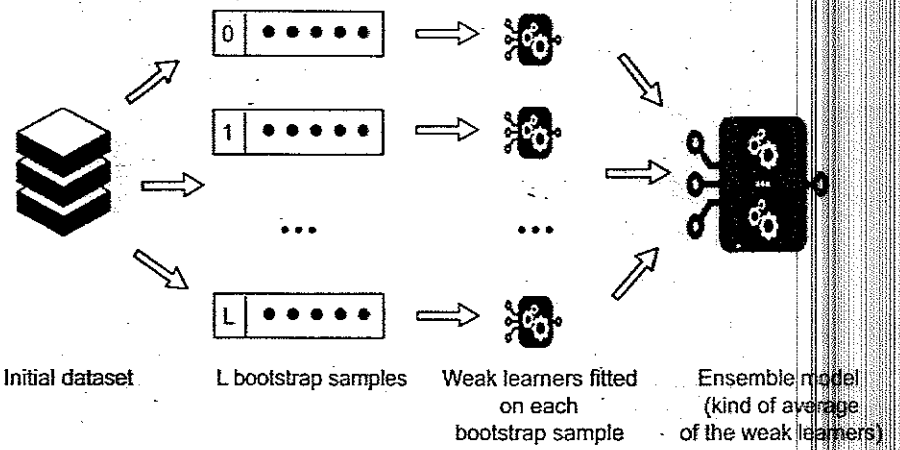


Fig. 4.10. Bagging

Bagging works especially well when the learners are unstable and tend to overfit, i.e. small changes in the training data lead to major changes in the predicted output. It effectively reduces the variance by aggregating the individual learners composed of different statistical properties, such as different standard deviations, means, etc. It works well for high variance models such as Decision Trees. When used with low variance models such as linear regression, it doesn't really affect the learning process. The number of base learners (trees) to be chosen depends on the characteristics of the dataset. Using too many trees doesn't lead to overfitting, but can consume a lot of computational power.

Bagging can be done in parallel to keep a check on excessive computational resources. This is a one good advantages that comes with it, and often is a booster to increase the usage of the algorithm in a variety of areas.

4.9.3. APPLICATIONS OF BAGGING

Bagging technique is used in a variety of applications. One main advantage is that it reduces the variance in prediction by generating additional data whilst applying different combinations and repetitions (replacements in the bootstrapped samples) in the training data. Below are some applications where the bagging algorithm is widely used:

- Banking Sector
- Medical Data Predictions

- High-dimensional data
- Land cover mapping
- Fraud detection
- Network Intrusion Detection Systems
- Medical fields like neuroscience, prosthetics, etc.

4.9.4. THE BAGGING ALGORITHM

- Bootstrapping comprises the first step in Bagging process flow wherein the data is divided into randomized samples.
- Then fit another algorithm (e.g. Decision Trees) to each of these samples. Training happens in parallel.
- Take an average of all the outputs, or in general, compute the aggregated output.

4.9.5. ADVANTAGES AND DISADVANTAGES

Let's discuss the advantages first. Bagging is a completely data-specific algorithm. The bagging technique reduces model over-fitting. It also performs well on high-dimensional data. Moreover, the missing values in the dataset do not affect the performance of the algorithm.

That being said, one limitation that it has is giving its final prediction based on the mean predictions from the subset trees, rather than outputting the precise values for the classification or regression model.

4.9.6. DECODING THE HYPERPARAMETERS

Scikit-learn has two classes for bagging, one for regression (sklearn. ensemble. Bagging Regressor) and another for classification (sklearn. ensemble. Bagging Classifier). Both accept various parameters which can enhance the model's speed and accuracy in accordance with the given data. Some of these include:

base_estimator: The algorithm to be used on all the random subsets of the dataset. Default value is a decision tree.

n_estimators: The number of base estimators in the ensemble. Default value is 10.

random_state: The seed used by the random state generator. Default value is None.

n_jobs: The number of jobs to run in parallel for both the fit and predict methods. Default value is None.

In the code below, we also use K-Folds cross-validation. It outputs the train/test indices to generate train and test data. The n_splits parameter determines the number of folds (default value is 3).

To estimate the accuracy, instead of splitting the data into training and test sets, we use K_Folds along with the cross_val_score. This method evaluates the score by cross-validation. Here's a peek into the parameters defined in cross_val_score:

estimator: The model to fit the data.

X: The input data to fit.

y: The target values against which the accuracy is predicted.

cv: Determines the cross-validation splitting strategy. Default value is 3.

4.9.7. IMPLEMENTING THE BAGGING ALGORITHM

Step 1: Import Modules

Of course, we begin by importing the necessary packages which are required to build our bagging model. We use the Pandas data analysis library to load our dataset. The dataset to be used is known as the Pima Indians Diabetes Database, used for predicting the onset of diabetes based on various diagnostic measures.

Next, let's import the Bagging Classifier from the sklearn.ensemble package, and the Decision Tree Classifier from the sklearn.tree package.

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

COPY

Step 2: Loading the Dataset

We'll download the dataset from the GitHub link provided below. Then we store the link in a variable called url. We name all the classes of the dataset in a list called names. We use the read_csv method from Pandas to load the dataset, and then set the class names using the parameter "names" to our variable, also called names.

Next, we load the features and target variables using the slicing operation and assign them to variables X and Y respectively.

```
url="https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
```

COPY

Step 3: Loading the Classifier

In this step, we set the seed value for all the random states, so that the generated random values shall remain the same until the seed value is exhausted. We now declare the KFold model wherein we set the *n*_splits parameter to 10, and the random_state parameter to the seed value. Here, we shall build the bagging algorithm with the Classification and Regression Trees algorithm. To do so, we import the Decision Tree Classifier, store it in the cart variable, and initiate the number of trees variable, num_trees to 100.

```
seed = 7
kfold = model_selection.KFold(n_splits=10, random_state=seed)
cart = DecisionTreeClassifier()
num_trees = 100
```

COPY

Step 4: Training and Results

In the last step, we train the model using the BaggingClassifier by declaring the parameters to the values defined in the previous step.

We then check the performance of the model by using the parameters, Bagging model, X, Y, and the kfold model. The output printed will be the accuracy of the model. In this particular case, it's around 77%.

```
model = BaggingClassifier(base_estimator=cart,n_estimators=num_trees, random_state=seed)
```



```
results = model_selection.cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Output:

0.770745044429255

COPY

When using a Decision Tree classifier alone, the accuracy noted is around 66%. Thus, Bagging is a definite improvement over the Decision Tree algorithm.

Example 2 :

Bagging Demonstration in Python Using IRIS Dataset

Import the libraries

```
#Import libraries
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

Load the dataset

```
dataset_train = pd.read_csv("Google_Stock_Price_Train.csv")
dataset_train.head( )
```

	Date	Open	High	Low	Close	Volume
0	1/3/2012	325.25	332.83	324.97	663.59	7,380,500
1	1/4/2012	331.27	333.87	329.08	666.45	5,749,400
2	1/5/2012	329.83	330.75	326.89	657.21	6,500,300
3	1/6/2012	328.34	328.77	323.68	648.24	5,405,900
4	1/9/2012	322.04	322.29	309.46	620.76	11,688,800

Split the dataset into training and testing

```
#Split data in training and testing set
```

```
X_fit, X_eval, y_fit, y_test=model_selection-train_test_split(X,Y, test _size=0.30,
random_state=1)
```

Creating sub samples to train models

```
#Create random sub samples to train multiple models
seed = 7
kfold = model_selection-Kfold(n-splits=10, random_states=seed)
```

Define a decision tree

```
#Define a decision tree classifier
cart = DecisionTreeClassifier( )
num_trees = 100
```

Classification model for bagging

```
#Create classification model for bagging
model=BaggingClassfier(base_estimator=cart,n_estimators=num_trees,
random_state=seed)
```

Train models and print their accuracy

```
#Train different models and print their accuracy
results = model_seelction-cross_val_score(model, X_fit, y_fit,cv=kfold)
for i in range(len(results)):
        print("Model:"+str(i)+"Accuracy is: "+str(results[i]))
Model: 0 Accuracy is :1.0
Model: 1 Accuracy is :1.0
Model: 2 Accuracy is :1.0
Model: 3 Accuracy is :0.909090909090091
Model: 4 Accuracy is :1.0
Model: 5 Accuracy is :1.0
Model: 6 Accuracy is :0.9
Model: 7 Accuracy is :1.0
Model: 8 Accuracy is :1.0
```



```
Model: 9 Accuracy is :0.7
```

Print the mean accuracy

```
print("Mean Accuracy is: "+str(results-mean( )))
Mean Accuracy is: 0.9509090909090908
```

Display the model's accuracy

```
model-fit(X_fit, y_fit)
pred_label = model-predict(X_eval)
nnz = np-shape(y_test)[0] - np-count_nonzero(pred_label - y_test)
acc = 100*nnz/np-shape(y_test)[0]
print("Accuracy is: '+str(acc))
Accuracy is: 95.55555555555556
```

From the above demonstration, you can conclude that the individual models (weak learners) overfit the data and have a high variance. But the aggregated result has a reduced variance and is trustworthy.

4.10. BOOSTING

4.10.1. WHY IS BOOSTING USED?

To solve convoluted problems we require more advanced techniques. Let's suppose that on given a data set of images containing images of cats and dogs, you were asked to build a model that can classify these images into two separate classes. Like every other person, you will start by identifying the images by using some rules, like given below:

1. The image has pointy ears: Cat
2. The image has cat shaped eyes: Cat
3. The image has bigger limbs: Dog
4. The image has sharpened claws: Cat
5. The image has a wider mouth structure: Dog

All these rules help us identify whether an image is a Dog or a cat, however, if we were to classify an image based on an individual (single) rule, the prediction would

be flawed. Each of these rules, individually, are called weak learners because these rules are not strong enough to classify an image as a cat or dog.

Therefore, to make sure that our prediction is more accurate, we can combine the prediction from each of these weak learners by using the majority rule or weighted average. This makes a strong learner model.

In the above example, we have defined 5 weak learners and the majority of these rules (i.e. 3 out of 5 learners predict the image as a cat) gives us the prediction that the image is a cat. Therefore, our final output is a cat.

So this brings us to the question,

4.10.2. WHAT IS BOOSTING?

Boosting is an ensemble learning technique that uses a set of Machine Learning algorithms to convert weak learner to strong learners in order to increase the accuracy of the model.

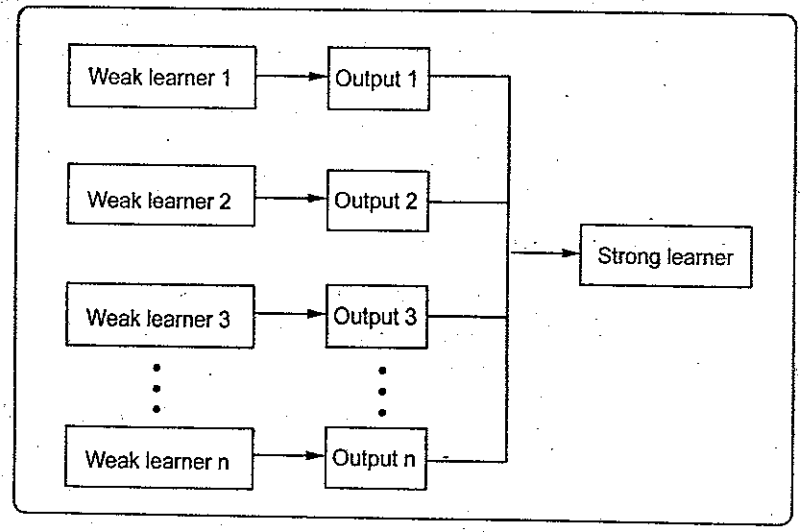


Fig. 4.11. Boosting

4.10.3. WHAT IS ENSEMBLE IN MACHINE LEARNING?

Ensemble learning is a method that is used to enhance the performance of Machine Learning model by combining several learners. When compared to a single model, this type of learning builds models with improved efficiency and accuracy.

This is exactly why ensemble methods are used to win market leading competitions such as the Netflix recommendation competition, Kaggle competitions and so on.

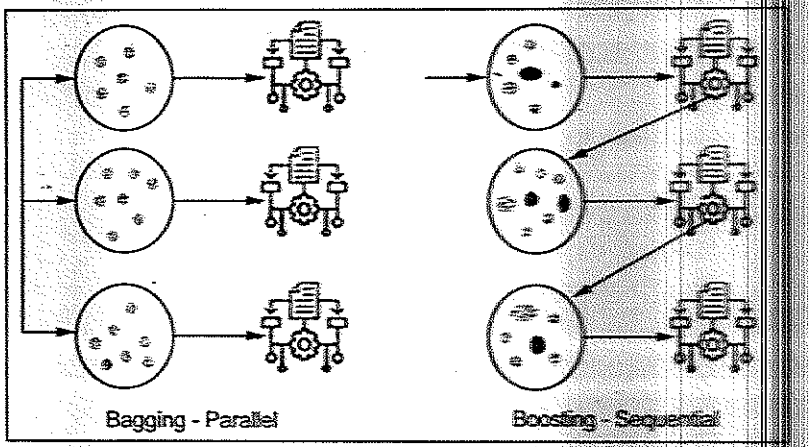


Fig. 4.12. Bagging Vs Boosting

4.10.4. BOOSTING VS BAGGING

Ensemble learning can be performed in two ways:

1. Sequential ensemble, popularly known as boosting, here the weak learners are sequentially produced during the training phase. The performance of the model is improved by assigning a higher weightage to the previous incorrectly classified samples. An example of boosting is the AdaBoost algorithm.
2. Parallel ensemble, popularly known as bagging, here the weak learners are produced parallelly during the training phase. The performance of the model can be increased by parallelly training a number of weak learners on bootstrapped data sets. An example of bagging is the Random Forest algorithm.

4.10.5. HOW BOOSTING ALGORITHM WORKS?

The basic principle behind the working of the boosting algorithm is to generate multiple weak learners and combine their predictions to form one strong rule. These weak rules are generated by applying base Machine Learning algorithms on different distributions of the data set. These algorithms generate weak rules for each iteration.

After multiple iterations, the weak learners are combined to form a strong learner that will predict a more accurate outcome.

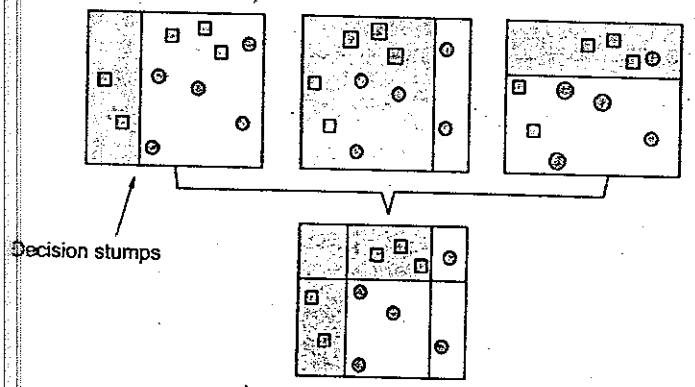


Fig. 4.13. Working of Boosting Algorithm

Here's how the algorithm works:

Step 1: The base algorithm reads the data and assigns equal weight to each sample observation.

Step 2: False predictions made by the base learner are identified. In the next iteration, these false predictions are assigned to the next base learner with a higher weightage on these incorrect predictions.

Step 3: Repeat step 2 until the algorithm can correctly classify the output.

Therefore, the main aim of Boosting is to focus more on miss-classified predictions.

4.10.6. TYPES OF BOOSTING

There are three main ways through which boosting can be carried out:

1. Adaptive Boosting or AdaBoost
2. Gradient Boosting
3. XGBoost

4.10.6.1. Adaptive Boosting

- AdaBoost is implemented by combining several weak learners into a single strong learner.

- The weak learners in AdaBoost take into account a single input feature and draw out a single split decision tree called the decision stump. Each observation is weighed equally while drawing out the first decision stump.
- The results from the first decision stump are analyzed and if any observations are wrongfully classified, they are assigned higher weights.
- Post this, a new decision stump is drawn by considering the observations with higher weights as more significant.
- Again if any observations are misclassified, they're given higher weight and this process continues until all the observations fall into the right class.
- Adaboost can be used for both classification and regression-based problems, however, it is more commonly used for classification purpose.

4.10.6.2. Gradient Boosting

Gradient Boosting is also based on sequential ensemble learning. Here the base learners are generated sequentially in such a way that the present base learner is always more effective than the previous one, i.e. the overall model improves sequentially with each iteration.

The difference in this type of boosting is that the weights for misclassified outcomes are not incremented, instead, Gradient Boosting method tries to optimize the loss function of the previous learner by adding a new model that adds weak learners in order to reduce the loss function.

The main idea here is to overcome the errors in the previous learner's predictions. This type of boosting has three main components:

1. Loss function that needs to be ameliorated.
2. Weak learner for computing predictions and forming strong learners.
3. An Additive Model that will regularize the loss function.

Like AdaBoost, Gradient Boosting can also be used for both classification and regression problems.

4.10.6.3. XGBoost

XGBoost is an advanced version of Gradient boosting method, it literally means eXtreme Gradient Boosting. XGBoost developed by Tianqi Chen, falls under the category of Distributed Machine Learning Community (DMLC).

The main aim of this algorithm is to increase the speed and efficiency of computation. The Gradient Descent Boosting algorithm computes the output at a slower rate since they sequentially analyze the data set, therefore XGBoost is used to boost or extremely boost the performance of the model.

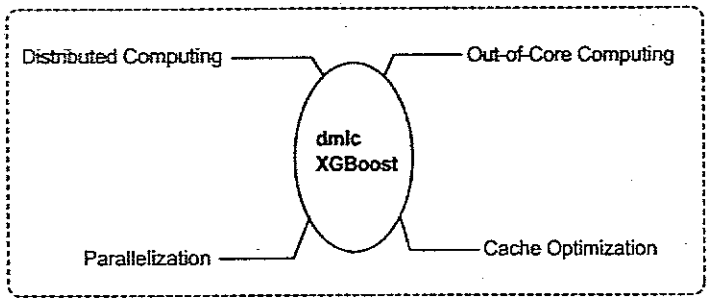


Fig. 4.14. XGBoost

XGBoost is designed to focus on computational speed and model efficiency. The main features provided by XGBoost are:

- Parallelly creates decision trees.
- Implementing distributed computing methods for evaluating large and complex models.
- Using Out-of-Core Computing to analyze huge datasets.
- Implementing cache optimization to make the best use of resources.

4.10.7. BOOSTING MACHINE LEARNING IN PYTHON

Problem Statement: To study a mushroom data set and build a Machine Learning model that can classify a mushroom as either poisonous or not, by analyzing its features.

Data Set Description: This data set provides a detailed description of hypothetical samples in accordance with 23 species of gilled mushrooms. Each species is classified as either edible mushrooms or non-edible (poisonous) ones.

Logic: To build a Machine Learning model by using one of the Boosting algorithms in order to predict whether or not a mushroom is edible.

Step 1: Import the required packages

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier
import pandas as pd
# Import train_test_split function
from sklearn.model_selection import train_test_split
#Import scikit-learn metrics module for accuracy calculation
from sklearn import metrics
```

Step 2: Import the data set

```
# Load in the data
dataset = pd.read_csv('C://Users//NeelTemp//Desktop//mushroomsdataset.csv')
```

Step 3: Data Processing

```
#Define the column names
    dataset.columns = ['target','cap-shape','cap-surface','cap-color','bruises','odor','gill-attachment','gill-spacing',
    'gill-size','gill-color','stalk-shape','stalk-root','stalk-surface-above-ring','stalk-surface-below-ring','stalk-color-above-ring',
    'stalk-color-below-ring','veil-type','veil-color','ring-number','ring-type','spore-print-color','population',
    'habitat']
    for label in dataset.columns:
    dataset[label] = LabelEncoder().fit(dataset[label]).transform(dataset[label])

    #Display information about the data set
    print(dataset.info())

    Int64Index: 8124 entries, 6074 to 686
    Data columns (total 23 columns):
    target 8124 non-null int32
    cap-shape 8124 non-null int32
```

```
19  cap-surface 8124 non-null int32
20  cap-color 8124 non-null int32
21  bruises 8124 non-null int32
22  odor 8124 non-null int32
23  gill-attachment 8124 non-null int32
24  gill-spacing 8124 non-null int32
25  gill-size 8124 non-null int32
26  gill-color 8124 non-null int32
27  stalk-shape 8124 non-null int32
28  stalk-root 8124 non-null int32
29  stalk-surface-above-ring 8124 non-null int32
30  stalk-surface-below-ring 8124 non-null int32
31  stalk-color-above-ring 8124 non-null int32
32  stalk-color-below-ring 8124 non-null int32
33  veil-type 8124 non-null int32
34  veil-color 8124 non-null int32
35  ring-number 8124 non-null int32
36  ring-type 8124 non-null int32
37  spore-print-color 8124 non-null int32
38  population 8124 non-null int32
    habitat 8124 non-null int32
    dtypes: int32(23)
    memory usage: 793.4 KB
```

Step 4: Data Splicing

```
1  X = dataset.drop(['target'], axis=1)
2  Y = dataset['target']
3  X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3)
```


Step 5: Build the model

```
1  model = DecisionTreeClassifier(criterion='entropy', max_depth=1)
2  AdaBoost = AdaBoostClassifier(base_estimator=model, n_estimators=400,
   learning_rate=1)
```

In the above code snippet, we have implemented the AdaBoost algorithm. The 'AdaBoostClassifier' function takes three important parameters:

- base_estimator: The base estimator (weak learner) is Decision Trees by default
- n_estimator: This field specifies the number of base learners to be used.
- learning_rate: This field specifies the learning rate, which we have set to the default value, i.e. 1.

```
1  #Fit the model with training data
2  boostmodel = AdaBoost.fit(X_train, Y_train)
```

Step 6: Model Evaluation

```
1  #Evaluate the accuracy of the model
2  y_pred = boostmodel.predict(X_test)
3  predictions = metrics.accuracy_score(Y_test, y_pred)
4  #Calculating the accuracy in percentage
5  print('The accuracy is: ', predictions * 100, '%')
6  The accuracy is: 100.0 %
```

We've received an accuracy of 100% which is perfect!

4.11. UNSUPERVISED LEARNING

Unsupervised learning is a machine learning technique in which models are not supervised using training dataset. Instead, models itself find the hidden patterns and insights from the given data. It can be compared to learning which takes place in the human brain while learning new things. It can be defined as:

Unsupervised learning is a type of machine learning in which models are trained using unlabeled dataset and are allowed to act on that data without any supervision.

Unsupervised learning cannot be directly applied to a regression or classification problem because unlike supervised learning, we have the input data but no corresponding output data. The goal of unsupervised learning is to find the underlying structure of dataset, group that data according to similarities, and represent that dataset in a compressed format.

Example: Suppose the unsupervised learning algorithm is given an input dataset containing images of different types of cats and dogs. The algorithm is never trained upon the given dataset, which means it does not have any idea about the features of the dataset. The task of the unsupervised learning algorithm is to identify the image features on their own. Unsupervised learning algorithm will perform this task by clustering the image dataset into the groups according to similarities between images.

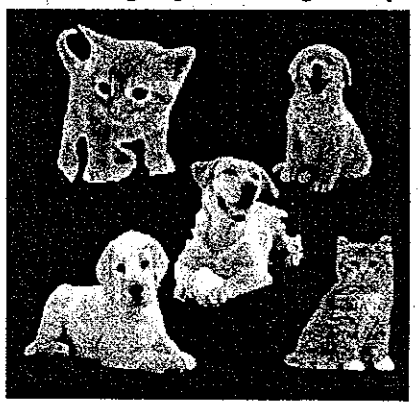


Fig. 4.15. Dataset

4.11.1. WHY USE UNSUPERVISED LEARNING?

Below are some main reasons which describe the importance of Unsupervised Learning:

- Unsupervised learning is helpful for finding useful insights from the data.
- Unsupervised learning is much similar as a human learns to think by their own experiences, which makes it closer to the real AI.
- Unsupervised learning works on unlabeled and uncategorized data which make unsupervised learning more important.
- In real-world, we do not always have input data with the corresponding output so to solve such cases, we need unsupervised learning.

4.11.2. WORKING OF UNSUPERVISED LEARNING

Here, we have taken an unlabeled input data, which means it is not categorized and corresponding outputs are also not given. Now, this unlabeled input data is fed to the machine learning model in order to train it. Firstly, it will interpret the raw data to find the hidden patterns from the data and then will apply suitable algorithm such as k-means clustering, Decision tree, etc.

Once it applies the suitable algorithm, the algorithm divides the data objects into groups according to the similarities and difference between the objects.

Working of unsupervised learning can be understood by the below diagram:

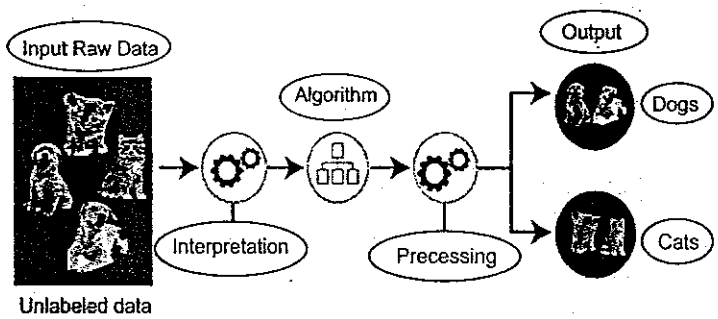


Fig. 4.16. Working of Unsupervised learning algorithm

4.11.3. TYPES OF UNSUPERVISED LEARNING ALGORITHM

The unsupervised learning algorithm can be further categorized into two types of problems:

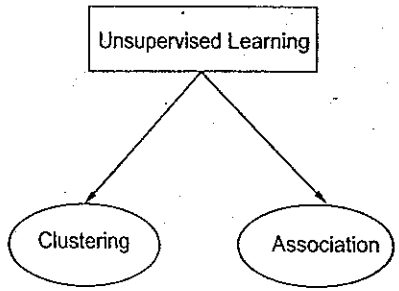


Fig. 4.17. Types of Unsupervised Algorithm

- **Clustering:** Clustering is a method of grouping the objects into clusters such that objects with most similarities remains into a group and has less

or no similarities with the objects of another group. Cluster analysis finds the commonalities between the data objects and categorizes them as per the presence and absence of those commonalities.

- **Association:** An association rule is an unsupervised learning method which is used for finding the relationships between variables in the large database. It determines the set of items that occurs together in the dataset. Association rule makes marketing strategy more effective. Such as people who buy X item (suppose a bread) are also tend to purchase Y

Unsupervised Learning algorithms:

Below is the list of some popular unsupervised learning algorithms:

- K-means clustering
- KNN (k-nearest neighbors)
- Hierarchal clustering
- Anomaly detection
- Neural Networks
- Principle Component Analysis
- Independent Component Analysis
- Apriori algorithm
- Singular value decomposition

4.11.4. ADVANTAGES OF UNSUPERVISED LEARNING

- Unsupervised learning is used for more complex tasks as compared to supervised learning because, in unsupervised learning, we don't have labeled input data.
- Unsupervised learning is preferable as it is easy to get unlabeled data in comparison to labeled data.

4.11.5. DISADVANTAGES OF UNSUPERVISED LEARNING

- Unsupervised learning is intrinsically more difficult than supervised learning as it does not have corresponding output.
- The result of the unsupervised learning algorithm might be less accurate as input data is not labeled, and algorithms do not know the exact output in advance.

4.12. K-MEANS

K-Means Clustering is an unsupervised learning algorithm that is used to solve the clustering problems in machine learning or data science. In this topic, we will learn what is K-means clustering algorithm, how the algorithm works, along with the Python implementation of k-means clustering.

4.12.1. WHAT IS K-MEANS ALGORITHM?

K-Means Clustering is an Unsupervised Learning algorithm, which groups the unlabeled dataset into different clusters. Here K defines the number of pre-defined clusters that need to be created in the process, as if K = 2, there will be two clusters, and for K = 3, there will be three clusters, and so on.

It is an iterative algorithm that divides the unlabeled dataset into k different clusters in such a way that each dataset belongs only one group that has similar properties.

It allows us to cluster the data into different groups and a convenient way to discover the categories of groups in the unlabeled dataset on its own without the need for any training.

It is a centroid-based algorithm, where each cluster is associated with a centroid. The main aim of this algorithm is to minimize the sum of distances between the data point and their corresponding clusters.

The algorithm takes the unlabeled dataset as input, divides the dataset into k-number of clusters, and repeats the process until it does not find the best clusters. The value of *k* should be predetermined in this algorithm.

The k-means clustering algorithm mainly performs two tasks:

- Determines the best value for K center points or centroids by an iterative process.
- Assigns each data point to its closest k-center. Those data points which are near to the particular k-center, create a cluster.

Hence each cluster has datapoints with some commonalities, and it is away from other clusters.

The below diagram explains the working of the K-means Clustering Algorithm:

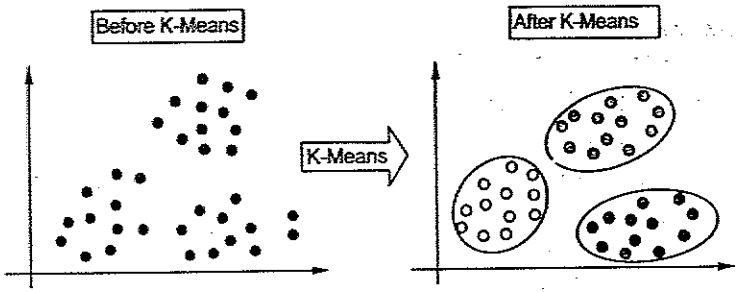


Fig. 4.18. K-means Clustering Algorithm

4.12.2. WORKING OF K-MEANS ALGORITHM

The working of the K-Means algorithm is explained in the below steps:

Step-1: Select the number K to decide the number of clusters.

Step-2: Select random K points or centroids. (It can be other from the input dataset).

Step-3: Assign each data point to their closest centroid, which will form the predefined K clusters.

Step-4: Calculate the variance and place a new centroid of each cluster.

Step-5: Repeat the third steps, which means reassign each datapoint to the new closest centroid of each cluster.

Step-6: If any reassignment occurs, then go to step-4 else go to FINISH.

Step-7: The model is ready.

Let's understand the above steps by considering the visual plots:

Suppose we have two variables M_1 and M_2. The *x-y* axis scatter plot of these two variables is given below:

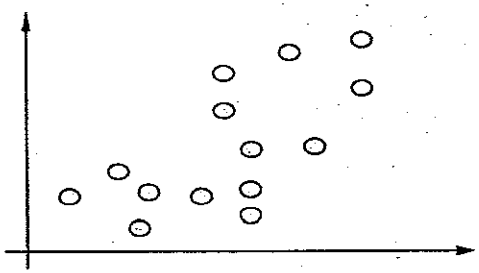


Fig. 4.19.

- Let's take number k of clusters, i.e., K = 2, to identify the dataset and to put them into different clusters. It means here we will try to group these datasets into two different clusters.
- We need to choose some random k points or centroid to form the cluster. These points can be either the points from the dataset or any other point. So, here we are selecting the below two points as k points, which are not the part of our dataset. Consider the below image:

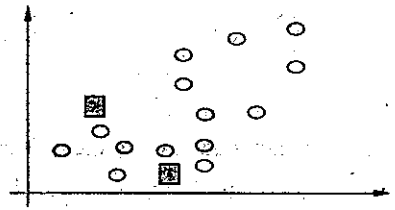


Fig. 4.20.

- Now we will assign each data point of the scatter plot to its closest K-point or centroid. We will compute it by applying some mathematics that we have studied to calculate the distance between two points. So, we will draw a median between both the centroids. Consider the below image:

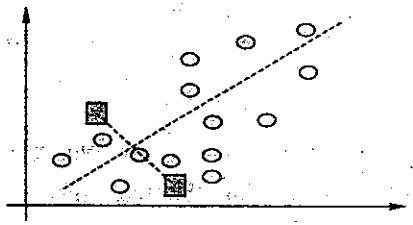


Fig. 4.21.

From the above image, it is clear that points left side of the line is near to the K_1 or blue centroid, and points to the right of the line are close to the yellow centroid. Let's color them as blue and yellow for clear visualization.

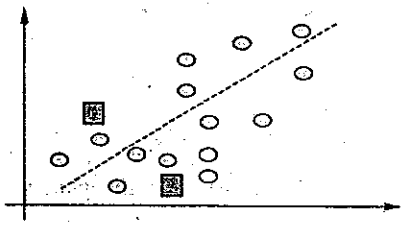


Fig. 4.22.

As we need to find the closest cluster, so we will repeat the process by choosing a **new centroid**. To choose the new centroids, we will compute the center of gravity of these centroids, and will find new centroids as below:

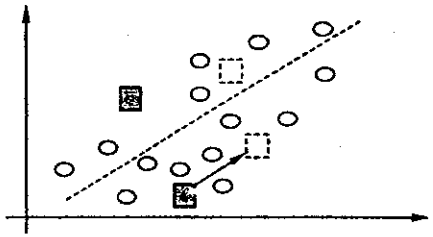


Fig. 4.23.

- Next, we will reassign each datapoint to the new centroid. For this, we will repeat the same process of finding a median line. The median will be like below image:

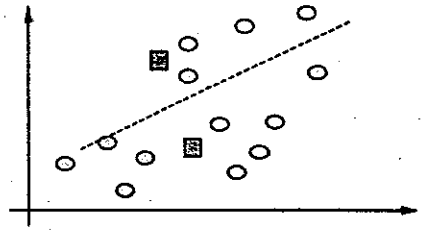


Fig. 4.24.

From the above image, we can see, one yellow point is on the left side of the line, and two blue points are right to the line. So, these three points will be assigned to new centroids.

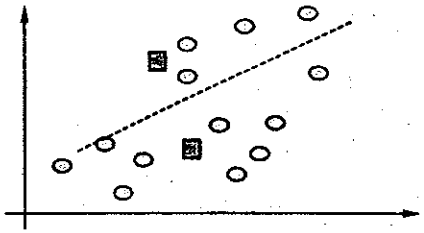


Fig. 4.25.

As reassignment has taken place, so we will again go to the step-4, which is finding new centroids or K-points.

- We will repeat the process by finding the center of gravity of centroids, so the new centroids will be as shown in the below image:

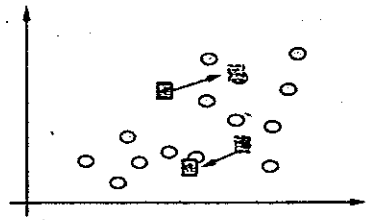


Fig. 4.26.

- As we got the new centroids so again will draw the median line and reassign the data points. So, the image will be:

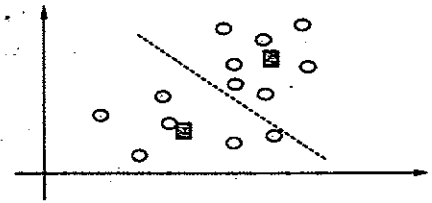


Fig. 4.27.

- We can see in the above image; there are no dissimilar data points on either side of the line, which means our model is formed. Consider the below image:

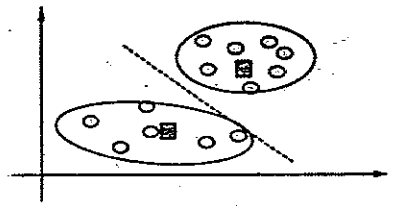


Fig. 4.28.

As our model is ready, so we can now remove the assumed centroids, and the two final clusters will be as shown in the below image:

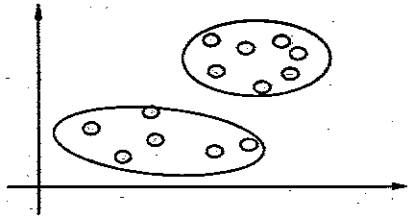


Fig. 4.29.

4.12.3. HOW TO CHOOSE THE VALUE OF "K NUMBER OF CLUSTERS" IN K-MEANS CLUSTERING?

The performance of the K-means clustering algorithm depends upon highly efficient clusters that it forms. But choosing the optimal number of clusters is a big task. There are some different ways to find the optimal number of clusters, but here we are discussing the most appropriate method to find the number of clusters or value of K. The method is given below:

4.12.3.1. Elbow Method

The Elbow method is one of the most popular ways to find the optimal number of clusters. This method uses the concept of WCSS value. **WCSS** stands for **Within Cluster Sum of Squares**, which defines the total variations within a cluster. The formula to calculate the value of WCSS (for 3 clusters) is given below:

$$WCSS = \sum_{Pi \text{ in cluster } 1} \text{distance } (P_i\ C_1)^2 + \sum_{Pi \text{ in cluster } 2} \text{distance } (P_i\ C_2)^2 + \sum_{Pi \text{ in cluster } 3} \text{distance } (P_i\ C_3)^2$$

In the above formula of WCSS,

$\sum_{Pi \text{ in cluster } 1}$ distance$(P_i\ C_1)^2$: It is the sum of the square of the distances between each data point and its centroid within a cluster1 and the same for the other two terms.

To measure the distance between data points and centroid, we can use any method such as Euclidean distance or Manhattan distance.

To find the optimal value of clusters, the elbow method follows the below steps:

- It executes the K-means clustering on a given dataset for different K values (ranges from 1-10).
- For each value of K, calculates the WCSS value.
- Plots a curve between calculated WCSS values and the number of clusters K.
- The sharp point of bend or a point of the plot looks like an arm, then that point is considered as the best value of K.

Since the graph shows the sharp bend, which looks like an elbow, hence it is known as the elbow method. The graph for the elbow method looks like the below image:

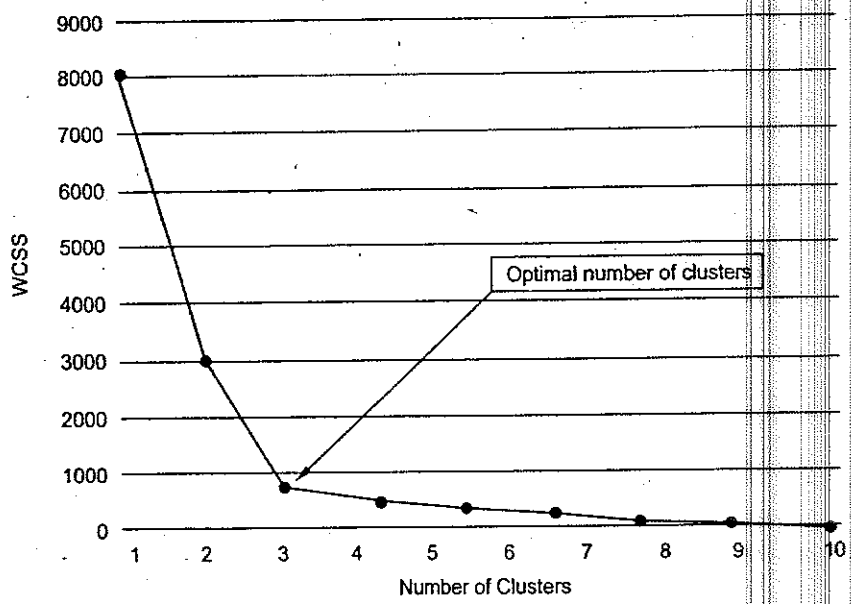


Fig. 4.30.

Note We can choose the number of clusters equal to the given data points. If we choose the number of clusters equal to the data points, then the value of WCSS becomes zero, and that will be the endpoint of the plot.

Example 1:

First, each data point is randomly assigned to one of the K clusters. Then, we compute the centroid (functionally the center) of each cluster, and reassign each data point to the cluster with the closest centroid. We repeat this process until the cluster assignments for each data point are no longer changing.

K-means clustering requires us to select K, the number of clusters we want to group the data into. The elbow method lets us graph the inertia (a distance-based metric) and visualize the point at which it starts decreasing linearly. This point is referred to as the "eblow" and is a good estimate for the best value for K based on our data.

Example

Start by visualizing some data points:

```
import matplotlib.pyplot as plt
x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
```

```
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]
plt.scatter(x, y)
plt.show()
```

Result

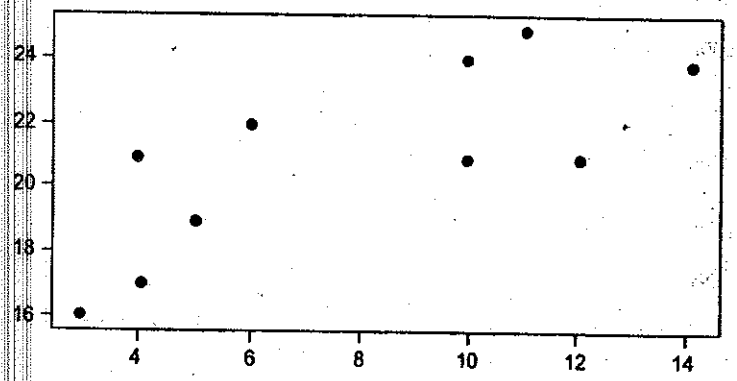


Fig. 4.31.

Now we utilize the elbow method to visualize the intertia for different values of K:

Example

```
from sklearn.cluster import KMeans
data = list(zip(x, y))
inertias = []
for i in range(1,11):
    kmeans = KMeans(n_clusters=i)
    kmeans.fit(data)
    inertias.append(kmeans.inertia_)
plt.plot(range(1,11), inertias, marker='o')
plt.title('Elbow method')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
plt.show()
```


Result

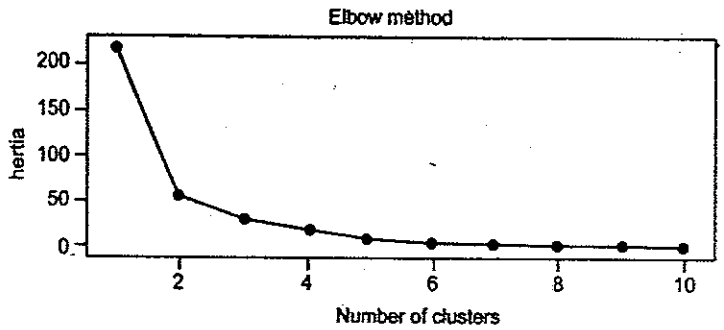


Fig. 4.32. Elbow method

The elbow method shows that 2 is a good value for K, so we retrain and visualize the result:

Example

```
kmeans = KMeans(n_clusters=2)
kmeans.fit(data)
plt.scatter(x, y, c=kmeans.labels_)
plt.show()
```

Result

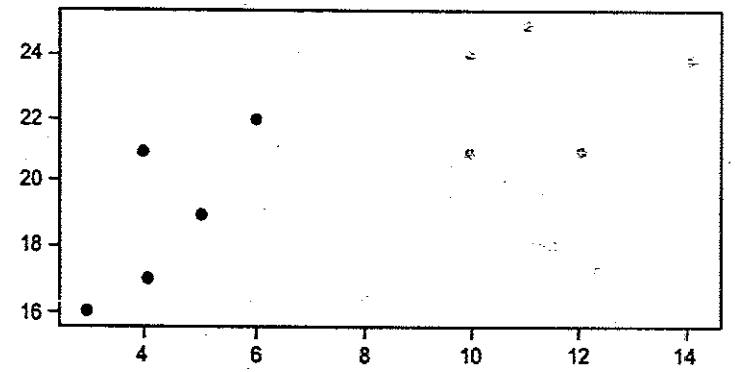


Fig. 4.33.

Example Explained

Import the modules you need.

```
import matplotlib.pyplot as plt
```

```
from sklearn.cluster import KMeans
```

You can learn about the Matplotlib module in our "Matplotlib Tutorial.

scikit-learn is a popular library for machine learning.

Create arrays that resemble two variables in a dataset. Note that while we only use two variables here, this method will work with any number of variables:

```
x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]
```

Turn the data into a set of points:

```
data = list(zip(x, y))
print(data)
```

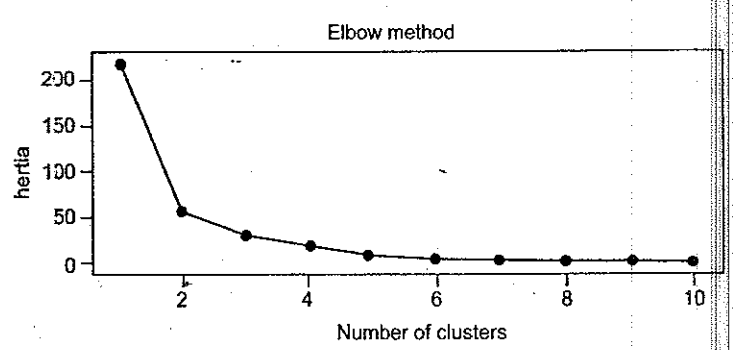
Result:

[(4, 21), (5, 19), (10, 24), (4, 17), (3, 16), (11, 25), (14, 24), (6, 22), (10, 21), (12, 21)]

In order to find the best value for K, we need to run K-means across our data for a range of possible values. We only have 10 data points, so the maximum number of clusters is 10. So for each value K in range(1,11), we train a K-means model and plot the intertia at that number of clusters:

```
inertias = []
for i in range(1,11):
    kmeans = KMeans(n_clusters=i)
    kmeans.fit(data)
    inertias.append(kmeans.inertia_)
plt.plot(range(1,11), inertias, marker='o')
plt.title('Elbow method')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
plt.show()
```

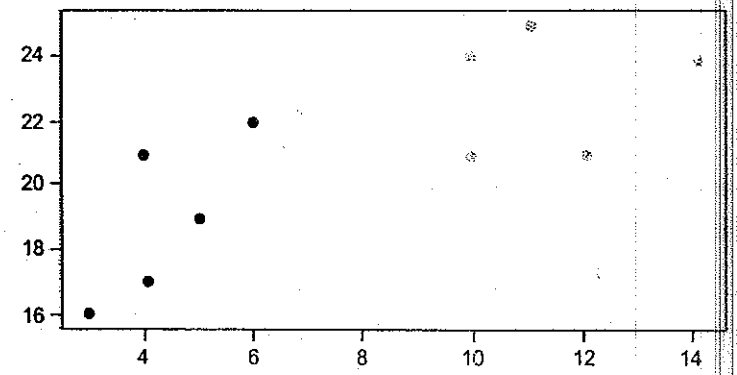

Result:



We can see that the "elbow" on the graph above (where the interia becomes more linear) is at K = 2. We can then fit our K-means algorithm one more time and plot the different clusters assigned to the data:

```
kmeans = KMeans(n_clusters=2)
kmeans.fit(data)
plt.scatter(x, y, c=kmeans.labels_)
plt.show()
```

Result:



Example 2:

Python Implementation of K-means Clustering Algorithm

We have a dataset of **Mall_Customers**, which is the data of customers who visit the mall and spend there.

In the given dataset, we have **Customer_Id, Gender, Age, Annual Income ($), and Spending Score** (which is the calculated value of how much a customer has spent in the mall, the more the value, the more he has spent). From this dataset, we need to calculate some patterns, as it is an unsupervised method, so we don't know what to calculate exactly.

The steps to be followed for the implementation are given below:

- Data Pre-processing
- Finding the optimal number of clusters using the elbow method
- Training the K-means algorithm on the training dataset
- Visualizing the clusters

Step-1: Data pre-processing Step

The first step will be the data pre-processing, as we did in our earlier topics of Regression and Classification. But for the clustering problem, it will be different from other models. Let's discuss it:

Importing Libraries :

As we did in previous topics, firstly, we will import the libraries for our model, which is part of data pre-processing. The code is given below:

```
# importing libraries
1. import numpy as nm
2. import matplotlib.pyplot as mtp
3. import pandas as pd
```

In the above code, **the numpy**

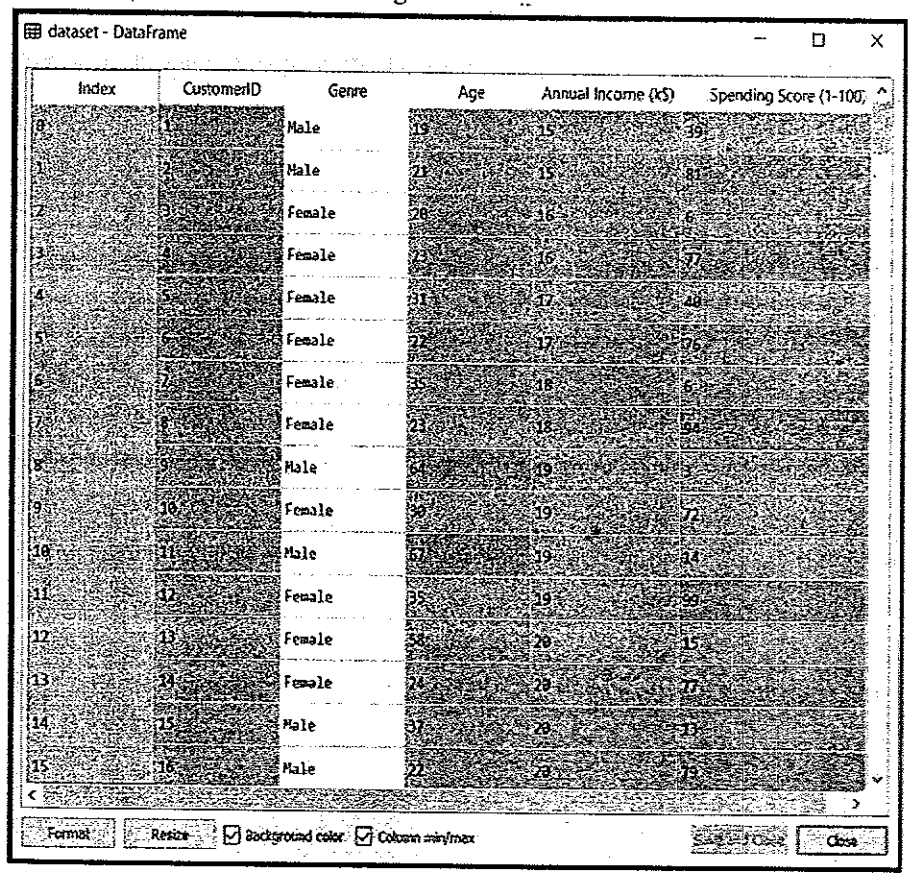
we have imported for the performing mathematics calculation, **matplotlib** is for plotting the graph, and **pandas** are for managing the dataset.

Importing the Dataset:

Next, we will import the dataset that we need to use. So here, we are using the Mall_Customer_data.csv dataset. It can be imported using the below code:

```
# Importing the dataset
1. dataset = pd.read_csv('Mall_Customers_data.csv')
```


By executing the above lines of code, we will get our dataset in the Spyder IDE. The dataset looks like the below image:



From the above dataset, we need to find some patterns in it.

Extracting Independent Variables

Here we don't need any dependent variable for data pre-processing step as it is a clustering problem, and we have no idea about what to determine. So we will just add a line of code for the matrix of features.

```
1. x = dataset.iloc[:, [3, 4]].values
```

As we can see, we are extracting only 3rd and 4th feature. It is because we need a 2d plot to visualize the model, and some features are not required, such as customer_id.

Step-2: Finding the optimal number of clusters using the elbow method

In the second step, we will try to find the optimal number of clusters for our clustering problem. So, as discussed above, here we are going to use the elbow method for this purpose.

As we know, the elbow method uses the WCSS concept to draw the plot by plotting WCSS values on the Y-axis and the number of clusters on the X-axis. So we are going to calculate the value for WCSS for different k values ranging from 1 to 10. Below is the code for it:

```
1. #finding optimal number of clusters using the elbow method
2. from sklearn.cluster import KMeans
3. wcss_list= []  #Initializing the list for the values of WCSS


4. #Using for loop for iterations from 1 to 10.
5. for i in range(1, 11):
6.     kmeans = KMeans(n_clusters=i, init='k-means++', random_state= 42)
7.     kmeans.fit(x)
8.     wcss_list.append(kmeans.inertia_)
9. mtp.plot(range(1, 11), wcss_list)
10. mtp.title('The Elobw Method Graph')
11. mtp.xlabel('Number of clusters(k)')
12. mtp.ylabel('wcss_list')
13. mtp.show()
```

As we can see in the above code, we have used **the KMeans** class of sklearn. cluster library to form the clusters.

Next, we have created the **wcss_list** variable to initialize an empty list, which is used to contain the value of wcss computed for different values of k ranging from 1 to 10.

After that, we have initialized the for loop for the iteration on a different value of k ranging from 1 to 10; since for loop in Python, exclude the outbound limit, so it is taken as 11 to include 10th value.

The rest part of the code is similar as we did in earlier topics, as we have fitted the model on a matrix of features and then plotted the graph between the number of clusters and WCSS.

Output: After executing the above code, we will get the below output:

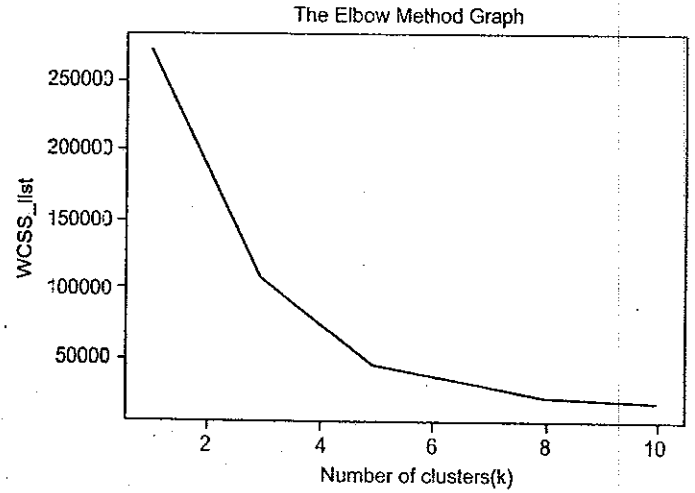
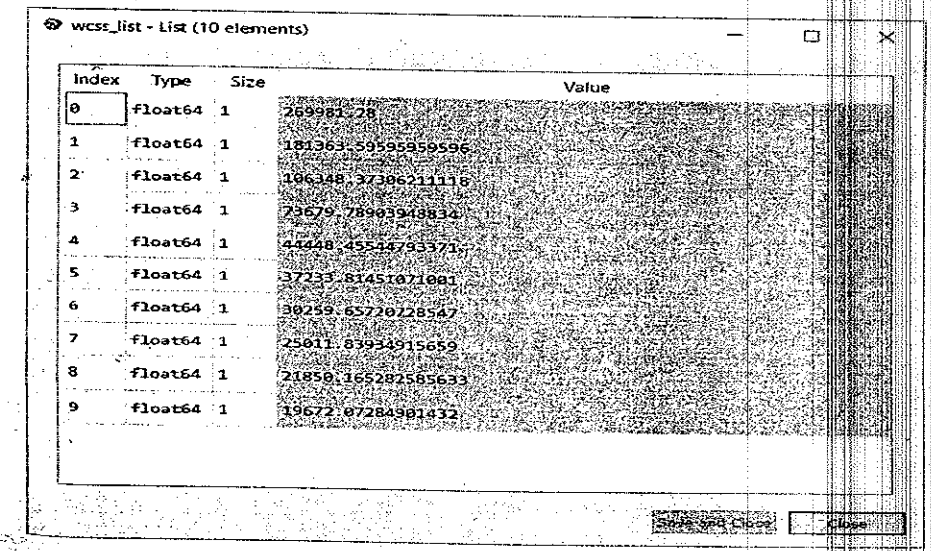


Fig. 4.34. The elbow method graph

From the above plot, we can see the elbow point is at **5. So the number of clusters here will be 5.**



Step- 3: Training the K-means algorithm on the training dataset

As we have got the number of clusters, so we can now train the model on the dataset.

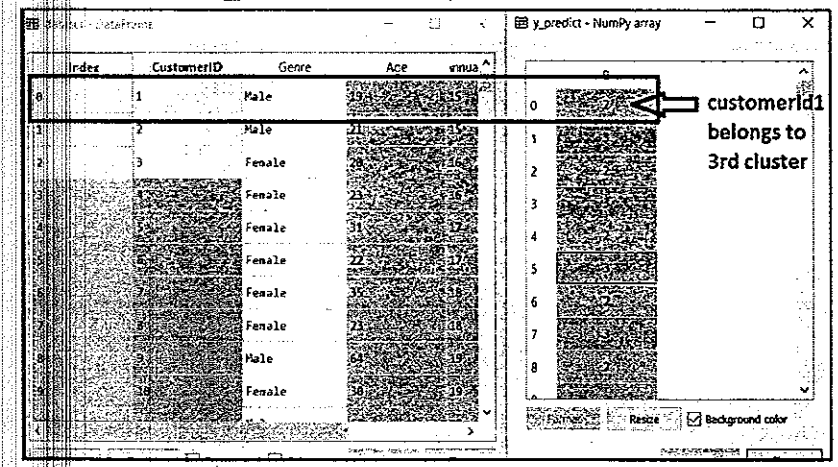
To train the model, we will use the same two lines of code as we have used in the above section, but here instead of using *i*, we will use 5, as we know there are 5 clusters that need to be formed. The code is given below:

```
#training the K-means model on a dataset
kmeans = KMeans(n_clusters=5, init='k-means++', random_state= 42)
y_predict= kmeans.fit_predict(x)
```

The first line is the same as above for creating the object of KMeans class.

In the second line of code, we have created the dependent variable **y_predict** to train the model.

By executing the above lines of code, we will get the y_predict variable. We can check it under **the variable explorer** option in the Spyder IDE. We can now compare the values of y_predict with our original dataset. Consider the below image:



From the above image, we can now relate that the CustomerID 1 belongs to a cluster 3(as index starts from 0, hence 2 will be considered as 3), and 2 belongs to cluster 4, and so on.

Step-4: Visualizing the Clusters

The last step is to visualize the clusters. As we have 5 clusters for our model, so we will visualize each cluster one by one.

To visualize the clusters will use scatter plot using mtp.scatter() function of matplotlib.

```
#visulaizing the clusters
mtp.scatter(x[y_predict == 0, 0], x[y_predict == 0, 1], s = 100,
c = 'blue', label = 'Cluster 1') #for first cluster
mtp.scatter(x[y_predict == 1, 0], x[y_predict == 1, 1], s = 100, c = 'green',
label = 'Cluster 2') #for second cluster
mtp.scatter(x[y_predict== 2, 0], x[y_predict == 2, 1], s = 100, c = 'red',
label = 'Cluster 3') #for third cluster
mtp.scatter(x[y_predict == , 0], x[y_predict == , 1], s = 100, c = 'cyan',
label = 'Cluster 4') #for fourth cluster
mtp.scatter(x[y_predict == 4, 0], x [y_predict == 4, 1], s = 100, c =
'magenta', label = 'Cluster 5') #for fifth cluster
mtp.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s
= 300, c = 'yellow', label = 'Centroid')
mtp.title('Clusters of customers')
mtp.xlabel('Annual Income (k$)')
mtp.ylabel('Spending Score (1-100)')
mtp.legend()
mtp.show()
```

In above lines of code, we have written code for each clusters, ranging from 1 to 5. The first coordinate of the mtp.scatter, i.e., x[y_predict == 0, 0] containing the x value for the showing the matrix of features values, and the y_predict is ranging from 0 to 1.

Output:

The output image is clearly showing the five different clusters with different colors. The clusters are formed between two parameters of the dataset; Annual income of customer and Spending. We can change the colors and labels as per the requirement or choice. We can also observe some points from the above patterns, which are given below:

- **Cluster1** shows the customers with average salary and average spending so we can categorize these customers as

- Cluster2 shows the customer has a high income but low spending, so we can categorize them as **careful**.
- Cluster3 shows the low income and also low spending so they can be categorized as sensible.
- Cluster4 shows the customers with low income with very high spending so they can be categorized as **careless**.
- Cluster 5 shows the customers with high income and high spending so they can be categorized as target, and these customers can be the most profitable customers for the mall owner.

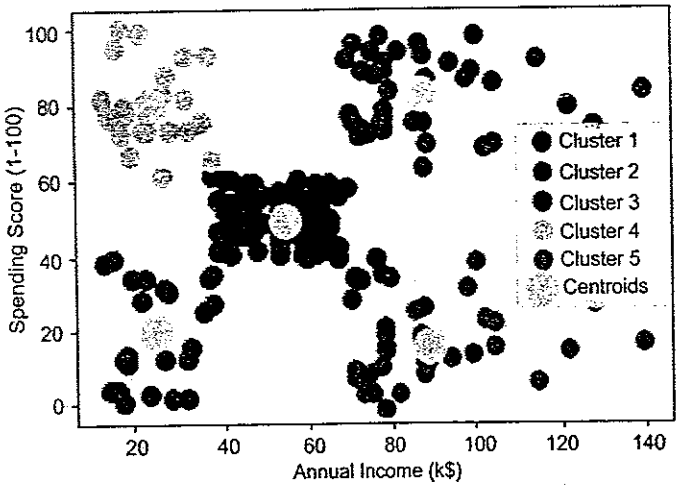


Fig. 4.35. Clusters of customers

4.13. INSTANCE BASED LEARNING

The Machine Learning systems which are categorized as instance-based learning are the systems that learn the training examples by heart and then generalize to new instances based on some similarity measure. It is called instance-based because it builds the hypotheses from the training instances. It is also known as memory-based learning or lazy-learning (because they delay processing until a new instance must be classified). The time complexity of this algorithm depends upon the size of training data. Each time whenever a new query is encountered, its previously stores data is examined. And assign to a target function value for the new instance.

Instance-based learning includes nearest neighbor, locally weighted regression and case-based reasoning methods.

A key advantage of lazy learning is that instead of estimating the target function once for the entire instance space, these methods can estimate it locally and differently for each new instance to be classified.

The worst-case time complexity of this algorithm is O (n), where n is the number of training instances. For example, If we were to create a spam filter with an instance-based learning algorithm, instead of just flagging emails that are already marked as spam emails, our spam filter would be programmed to also flag emails that are very similar to them. This requires a measure of resemblance between two emails. A similarity measure between two emails could be the same sender or the repetitive use of the same keywords or something else.

Instance-based learning refers to a family of techniques for classification and regression, which produce a class label/prediction based on the similarity of the query to its nearest neighbor(s) in the training set.

Functions are as follows:

1. **Similarity:** Similarity is a machine learning method that uses a nearest neighbor approach to identify the similarity of two or more objects to each other based on algorithmic distance functions.
2. **Classification:** Process of categorizing a given set of data into classes, It can be performed on both structured or unstructured data. The process starts with predicting the class of given data points. The classes are often referred to as target, label or categories.
3. **Concept Description:** Much of human learning involves acquiring general concepts from past experiences. This description can then be used to predict the class labels of unlabeled cases.

Advantages:

1. Instead of estimating for the entire instance set, local approximations can be made to the target function.
2. This algorithm can adapt to new data easily, one which is collected as we go.

Disadvantages:

1. Classification costs are high
2. Large amount of memory required storing the data, and each query involves starting the identification of a local model from scratch.

Some of the instance-based learning algorithms are:

1. K Nearest Neighbor (KNN)
2. Self-Organizing Map (SOM)
3. Learning Vector Quantization (LVQ)
4. Locally Weighted Learning (LWL)
5. Case-Based Reasoning

4.14. K NEAREST NEIGHBOR (KNN)

- K-Nearest Neighbour is one of the simplest Machine Learning algorithms based on Supervised Learning technique.
- K-NN algorithm assumes the similarity between the new case/data and available cases and put the new case into the category that is most similar to the available categories.
- K-NN algorithm stores all the available data and classifies a new data point based on the similarity. This means when new data appears then it can be easily classified into a well suite category by using K- NN algorithm.
- K-NN algorithm can be used for Regression as well as for Classification but mostly it is used for the Classification problems.
- K-NN is a non-parametric algorithm, which means it does not make any assumption on underlying data.
- It is also called a lazy learner algorithm because it does not learn from the training set immediately instead it stores the dataset and at the time of classification, it performs an action on the dataset.
- KNN algorithm at the training phase just stores the dataset and when it gets new data, then it classifies that data into a category that is much similar to the new data.
- **Example:** Suppose, we have an image of a creature that looks similar to cat and dog, but we want to know either it is a cat or dog. So for this identification, we can use the KNN algorithm, as it works on a similarity measure. Our KNN model will find the similar features of the new data set to the cats and dogs images and based on the most similar features it will put it in either cat or dog category.

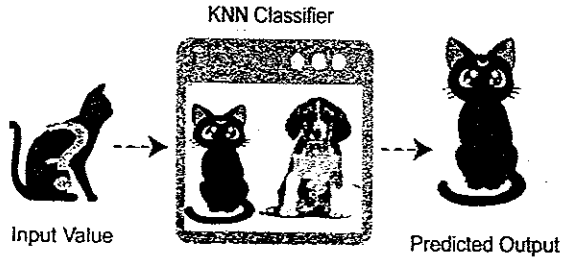


Fig. 4.36. KNN classifier

4.14.1. WHY DO WE NEED A KNN ALGORITHM?

Suppose there are two categories, i.e., Category A and Category B, and we have a new data point x_1, so this data point will lie in which of these categories. To solve this type of problem, we need a K-NN algorithm. With the help of K-NN, we can easily identify the category or class of a particular dataset. Consider the below diagram:

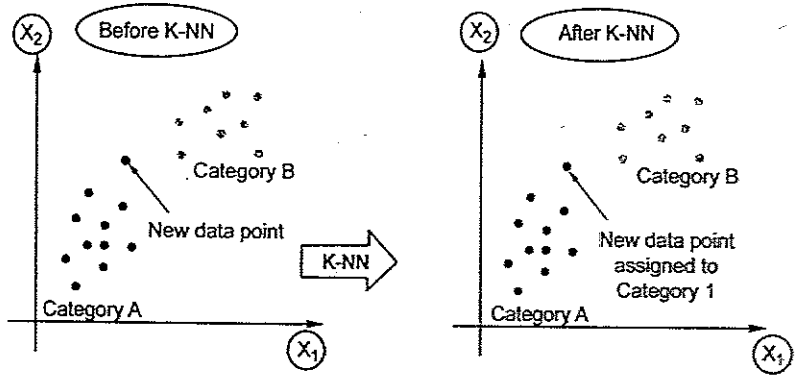


Fig. 4.37. Need of KNN

4.14.2. WORKING OF KNN

The K-NN working can be explained on the basis of the below algorithm:

Step-1: Select the number K of the neighbors

Step-2: Calculate the Euclidean distance of K number of neighbors

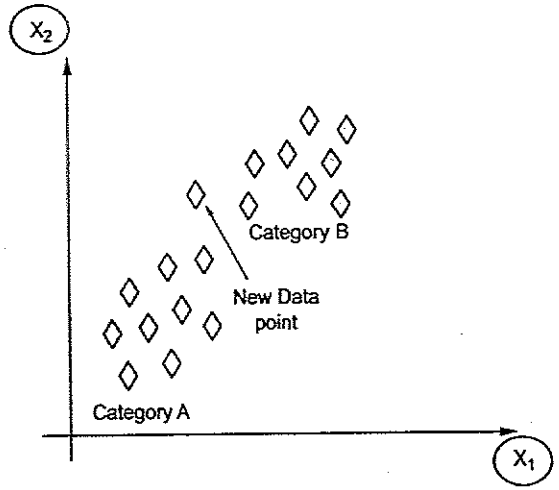
Step-3: Take the K nearest neighbors as per the calculated Euclidean distance.

Step-4: Among these k neighbors, count the number of the data points in each category.

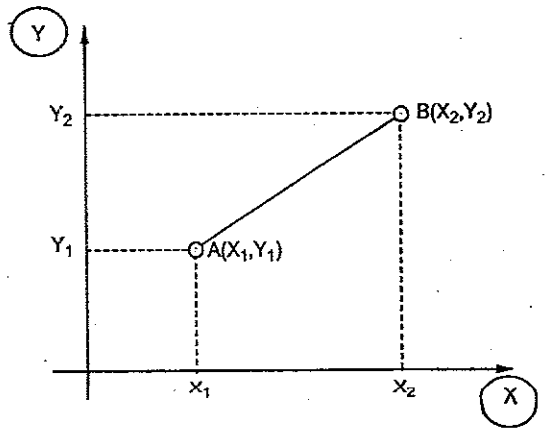
Step-5: Assign the new data points to that category for which the number of the neighbor is maximum.

Step-6: Our model is ready.

Suppose we have a new data point and we need to put it in the required category. Consider the below image:

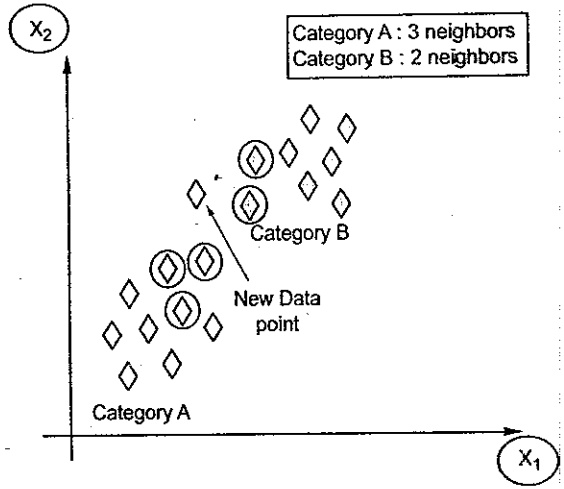


- Firstly, we will choose the number of neighbors, so we will choose the $k = 5$.
- Next, we will calculate the Euclidean distance between the data points. The Euclidean distance is the distance between two points, which we have already studied in geometry. It can be calculated as:



Euclidean Distance between A_1 and $B_2 = \sqrt{X_2 - X_1) + (Y_2 - Y_1)^2}$

- By calculating the Euclidean distance we got the nearest neighbors, as three nearest neighbors in category A and two nearest neighbors in category B. Consider the below image:



- As we can see the 3 nearest neighbors are from category A, hence this new data point must belong to category A.

4.14.3. HOW TO SELECT THE VALUE OF K IN THE KNN ALGORITHM?

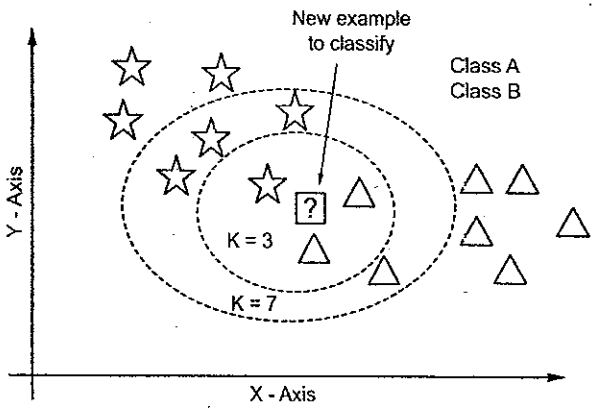
Below are some points to remember while selecting the value of K in the K-NN algorithm: A very low value for K such as K = 1 or K = 2, can be noisy and lead to the effects of outliers in the model.

Large values for K are good, but it may find some difficulties.

K value indicates the count of the nearest neighbors. We have to compute distances between test-points and trained labels points. Updating distance metrics with every iteration is computationally expensive, and that's why KNN is a lazy learning algorithm.

As you can verify from the above image, if we proceed with K = 3, then we predict that test input belongs to class B, and if we continue with K = 7, then we predict that test input belongs to class A.

That's how you can imagine that the K value has a powerful effect on KNN performance.



Then how to select the optimal K value?

- There are no pre-defined statistical methods to find the most favorable value of K.
- Initialize a random K value and start computing.
- Choosing a small value of K leads to unstable decision boundaries.
- The substantial K value is better for classification as it leads to smoothening the decision boundaries.
- Derive a plot between error rate and K denoting values in a defined range. Then choose the K value as having a minimum error rate.

Now you will get the idea of choosing the optimal K value by implementing the model.

Calculating Distance:

The first step is to calculate the distance between the new point and each training point. There are various methods for calculating this distance, of which the most commonly known methods are — Euclidian, Manhattan (for continuous) and Hamming distance (for categorical).

Euclidean Distance: Euclidean distance is calculated as the square root of the sum of the squared differences between a new point (x) and an existing point (y).

Manhattan Distance: This is the distance between real vectors using the sum of their absolute difference.

Distance functions

Euclidean $\sqrt{\sum_{i=1}^{k} (x_i - y_i)^2}$

Manhattan $\sum_{i=1}^{k} |x_1 - y_i|$

Hamming Distance: It is used for categorical variables. If the value (x) and the value (y) are the same, the distance D will be equal to 0 . Otherwise D = 1.

$$DH = \sum_{i=1}^{k} |x_1 - y_i|$$

$$x = y \Rightarrow D = 0$$

$$x \neq y \Rightarrow D = 1$$

4.14.4. WAYS TO PERFORM KNN

KNeighborsClassifier(n_neighbors=5, *, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None, **kwargs)

algorithm : {'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'

4.14.4.1. Brute Force

Lets consider for simple case with two dimension plot. If we look mathematically, the simple intuition is to calculate the euclidean distance from point of interest (of whose class we need to determine) to all the points in training set. Then we take class with majority points. This is called brute force method.

4.14.4.2. k-Dimensional Tree (kd tree)

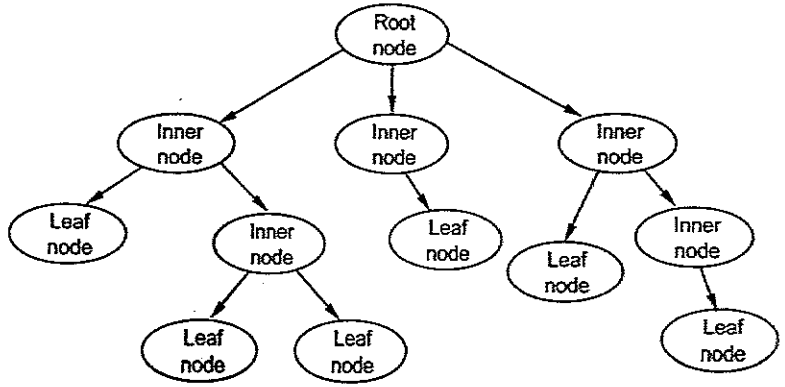


Fig. 4.38.

k-d tree is a hierarchical binary tree. When this algorithm is used for *k*-NN classification, it rearranges the whole dataset in a binary tree structure, so that when test data is provided, it would give out the result by traversing through the tree, which takes less time than brute search.

For a better understanding of how this can look like in a computer science topic, you can find below an HTML-code. A tree helps to structure a website and websites can normally be depicted using a tree.

```
<html>
<head>
  <meta charset=utf-8" />
  <title>Ball Tree vs. KD Tree</title>
  <nav>
  <a href="/r/">R</a>
  <a href="/js/">JavaScript</a>
  <a href="/python/">Python</a>
  </nav>
</head>
<body>
  <h1>What is a tree?</h1>
  <ul>
    <li>List item one</li>
    <li>List item two</li>
  </ul>
  <h2>How does a tree look like?</h2>
</body>
</html>
```

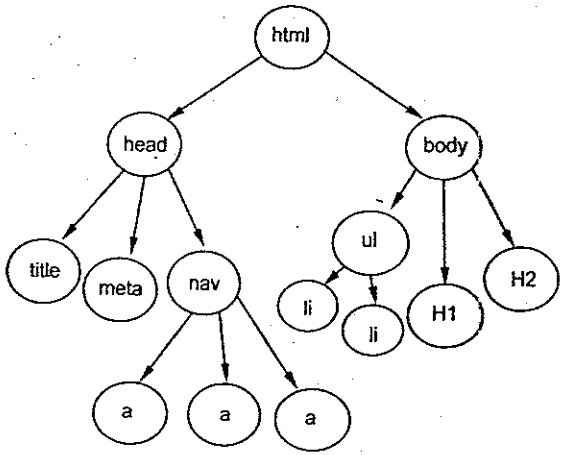



Fig. 4.39.

4.14.4.3. Ball Tree

Similar to k-d trees, Ball trees are also hierarchical data structure. These are very efficient specially in case of higher dimensions.

- Two clusters are created initially
- All the data points must belong to atleast one of the clusters.
- One point cannot be in both clusters.

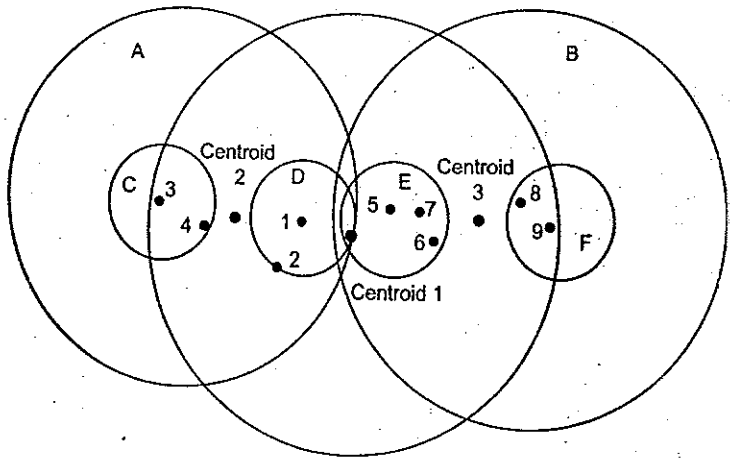


Fig. 4.40.

- Distance of the point is calculated from the centroid of the each cluster. The point closer to the centroid goes into that particular cluster.

- Each cluster is then divided into sub clusters again, and then the points are classified into each cluster on the basis of distance from centroid.
- This is how the clusters are kept to be divided till a certain depth.

The final resulting Ball Tree as follows,

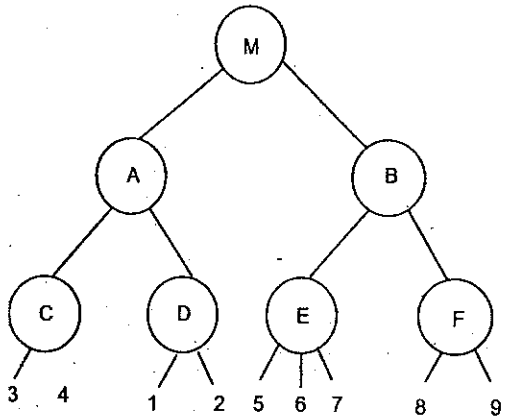


Fig. 4.41.

4.14.5. ADVANTAGES OF KNN ALGORITHM

- It is simple to implement.
- It is robust to the noisy training data
- It can be more effective if the training data is large.

4.14.6. DISADVANTAGES OF KNN ALGORITHM

- Always needs to determine the value of K which may be complex some time.
- The computation cost is high because of calculating the distance between the data points for all the training samples.

4.14.7. PYTHON IMPLEMENTATION OF THE KNN ALGORITHM

To do the Python implementation of the K-NN algorithm, we will use the same problem and dataset which we have used in Logistic Regression. But here we will improve the performance of the model. Below is the problem description:

Problem for K-NN Algorithm: There is a Car manufacturer company that has manufactured a new SUV car. The company wants to give the ads to the users who

are interested in buying that SUV. So for this problem, we have a dataset that contains multiple user's information through the social network. The dataset contains lots of information but the Estimated Salary and Age we will consider for the independent variable and the Purchased variable is for the dependent variable. Below is the dataset:

User ID	Gender	Age	Estimated Salary	Purchased
15624510	Male	19	19000	0
15810944	Male	35	20000	0
15668575	Female	26	43000	0
15603246	Female	27	57000	0
15804002	Male	19	76000	0
15728773	Male	27	58000	0
15598044	Female	27	84000	0
15694829	Female	32	150000	1
15600575	Male	25	33000	0
15727311	Female	35	65000	0
15570769	Female	26	80000	0
15606274	Female	26	52000	0
15746139	Male	20	86000	0
15704987	Male	32	18000	0
15628972	Male	18	82000	0
15697686	Male	29	80000	0
15733883	Male	47	25000	1
15617482	Male	45	26000	1
15704583	Male	46	28000	1
15621083	Female	48	29000	1
15649487	Male	45	22000	1
15736760	Female	47	49000	1

Steps to implement the K-NN algorithm:

- Data Pre-processing step
- Fitting the K-NN algorithm to the Training set
- Predicting the test result
- Test accuracy of the result(Creation of Confusion matrix)
- Visualizing the test set result.

Data Pre-Processing Step:

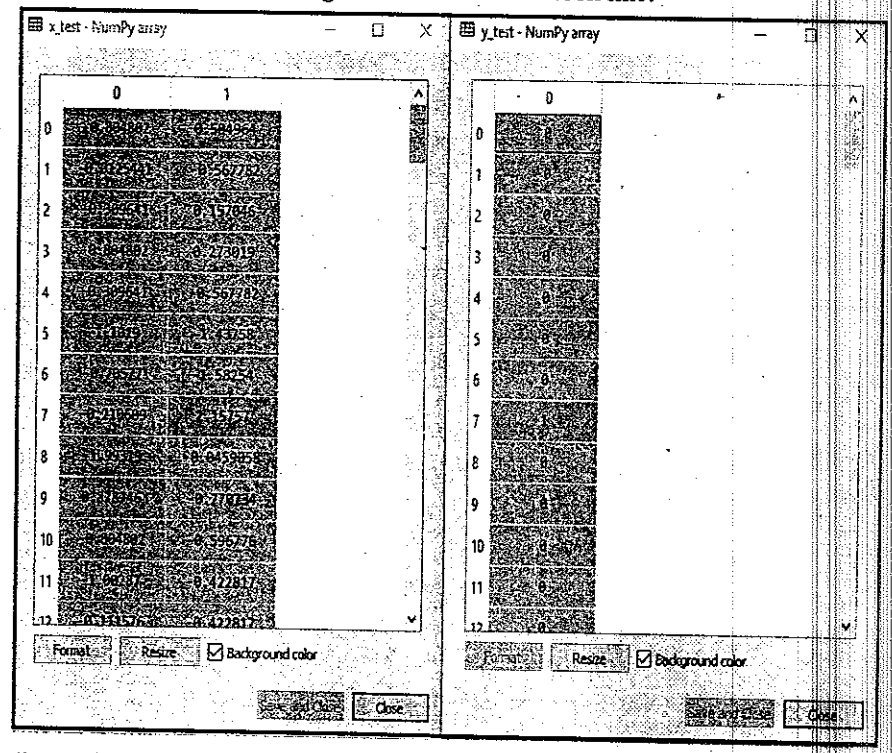
The Data Pre-processing step will remain exactly the same as Logistic Regression. Below is the code for it:

```
1. # importing libraries
2. import numpy as nm
3. import matplotlib.pyplot as mtp
4. import pandas as pd
5.
6. #importing datasets
7. data_set= pd.read_csv('user_data.csv')
8.
9. #Extracting Independent and dependent Variable
10. x= data_set.iloc[:, [2,3]].values
11. y= data_set.iloc[:, 4].values
12.
13. # Splitting the dataset into training and test set.
14. from sklearn.model_selection import train_test_split
15. x_train, x_test, y_train, y_test= train_test_split(x, y, test_size= 0.25, random_state=0)
16.
17. #feature Scaling
18. from sklearn.preprocessing import StandardScaler
19. st_x= StandardScaler()
```



```
20. x_train= st_x.fit_transform(x_train)
21. x_test= st_x.transform(x_test)
```

By executing the above code, our dataset is imported to our program and well pre-processed. After feature scaling our test dataset will look like:



From the above output image, we can see that our data is successfully scaled.

Fitting K-NN classifier to the Training data:

Now we will fit the K-NN classifier to the training data. To do this we will import the **KNeighborsClassifier** class of **Sklearn Neighbors** library. After importing the class, we will create the **Classifier** object of the class. The Parameter of this class will be

- **n_neighbors:** To define the required neighbors of the algorithm. Usually, it takes 5.
- **metric='minkowski':** This is the default parameter and it decides the distance between the points.

- **p=2:** It is equivalent to the standard Euclidean metric.

And then we will fit the classifier to the training data. Below is the code for it:

```
1. #Fitting K-NN classifier to the training set
2. from sklearn.neighbors import KNeighborsClassifier
3. classifier= . KNeighborsClassifier(n_neighbors=5, metric='minkowski',
   p=2 )
4. classifier.fit(x_train, y_train)
```

Output:

By executing the above code, we will get the output as:

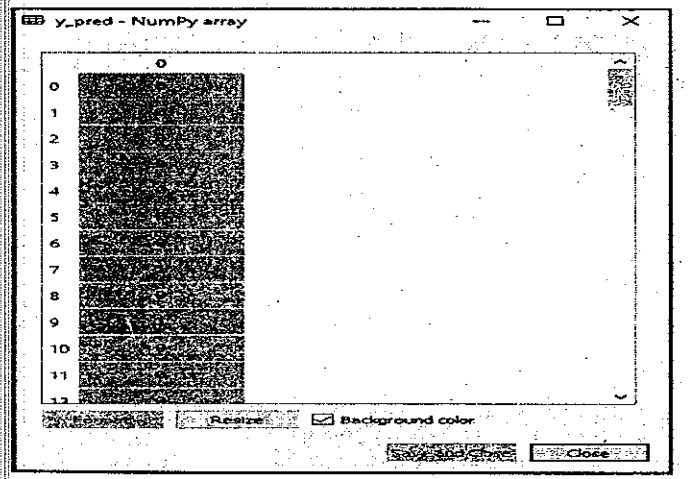
```
Out[10]:
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

Predicting the Test Result: To predict the test set result, we will create a **y_pred** vector as we did in Logistic Regression. Below is the code for it:

```
1. #Predicting the test set result
2. y_pred= classifier.predict(x_test)
```

Output:

The output for the above code will be:



Creating the Confusion Matrix:

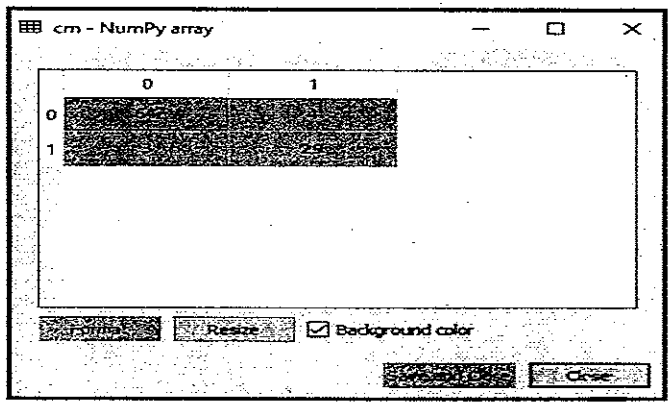
Now we will create the Confusion Matrix for our K-NN model to see the accuracy of the classifier. Below is the code for it:

```
#Creating the Confusion matrix
1. from sklearn.metrics import confusion_matrix
2. cm= confusion_matrix(y_test, y_pred)
```

In above code, we have imported the confusion_matrix function and called it using the variable cm.

Output:

By executing the above code, we will get the matrix as below:



In the above image, we can see there are 64+29= 93 correct predictions and 3 + 4 = 7 incorrect predictions, whereas, in Logistic Regression, there were 11 incorrect predictions. So we can say that the performance of the model is improved by using the K-NN algorithm.

Visualizing the Training set result:

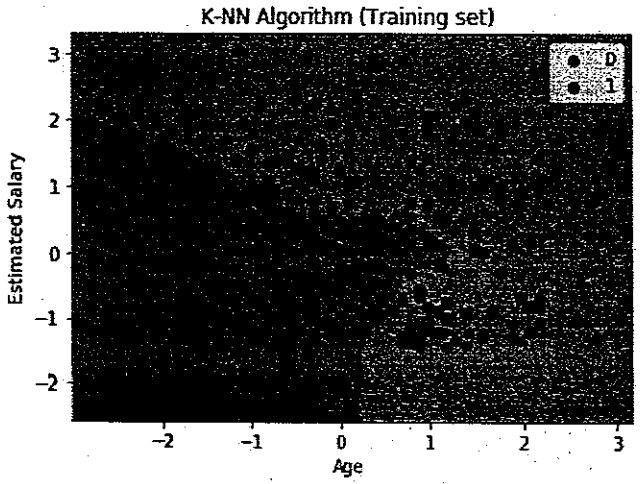
Now, we will visualize the training set result for K-NN model. The code will remain same as we did in Logistic Regression, except the name of the graph. Below is the code for it:

```
1. #Visulaizing the trianing set result
2. from matplotlib.colors import ListedColormap
3. x_set, y_set = x_train, y_train
```

```
4. x1, x2 = nm.meshgrid(nm.arange(start = x_set[:, 0].min() -
   1, stop = x_set[:, 0].max() + 1, step =0.01),
5. nm.arange(start = x_set[:, 1].min() -
   1, stop = x_set[:, 1].max() + 1, step = 0.01))
6. mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(),
   x2.ravel()]).T).reshape(x1.shape),
7. alpha = 0.75, cmap = ListedColormap(('red','green' )))
8. mtp.xlim(x1.min(), x1.max())
9. mtp.ylim(x2.min(), x2.max())
10. for i, j in enumerate(nm.unique(y_set)):
11.    mtp.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1],
12.        c = ListedColormap(('red', 'green'))(i), label = j)
13. mtp.title('K-NN Algorithm (Training set)')
14. mtp.xlabel('Age')
15. mtp.ylabel('Estimated Salary')
16. mtp.legend()
17. mtp.show()
```

Output:

By executing the above code, we will get the below graph:



The output graph is different from the graph which we have occurred in Logistic Regression. It can be understood in the below points:

- As we can see the graph is showing the red point and green points. The green points are for Purchased(1) and Red Points for not Purchased(0) variable.
- The graph is showing an irregular boundary instead of showing any straight line or any curve because it is a K-NN algorithm, i.e., finding the nearest neighbor.
- The graph has classified users in the correct categories as most of the users who didn't buy the SUV are in the red region and users who bought the SUV are in the green region.
- The graph is showing good result but still, there are some green points in the red region and red points in the green region. But this is no big issue as by doing this model is prevented from overfitting issues.
- Hence our model is well trained.

Visualizing the Test set result:

After the training of the model, we will now test the result by putting a new dataset, i.e., Test dataset. Code remains the same except some minor changes: such as **x_train and y_train** will be replaced by **x_test and y_test**.

Below is the code for it:

```
1. #Visualizing the test set result
2. from matplotlib.colors import ListedColormap
3. x_set, y_set = x_test, y_test
4. x1, x2 = nm.meshgrid(nm.arange(start = x_set[:, 0].min() -
   1, stop = x_set[:, 0].max() + 1, step =0.01),
5. nm.arange(start = x_set[:, 1].min() -
   1, stop = x_set[:, 1].max() + 1, step = 0.01))
6. mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(),
   x2.ravel()]).T).reshape(x1.shape),
7. alpha = 0.75, cmap = ListedColormap(('red','green' )))
```

```
mtp.xlim(x1.min(), x1.max())
mtp.ylim(x2.min(), x2.max())
for i, j in enumerate(nm.unique(y_set)):
mtp.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1],
        c = ListedColormap(('red', 'green'))(i), label = j)
mtp.title('K-NN algorithm(Test set)')
mtp.xlabel('Age')
mtp.ylabel('Estimated Salary')
mtp.legend()
mtp.show()
```

Output:

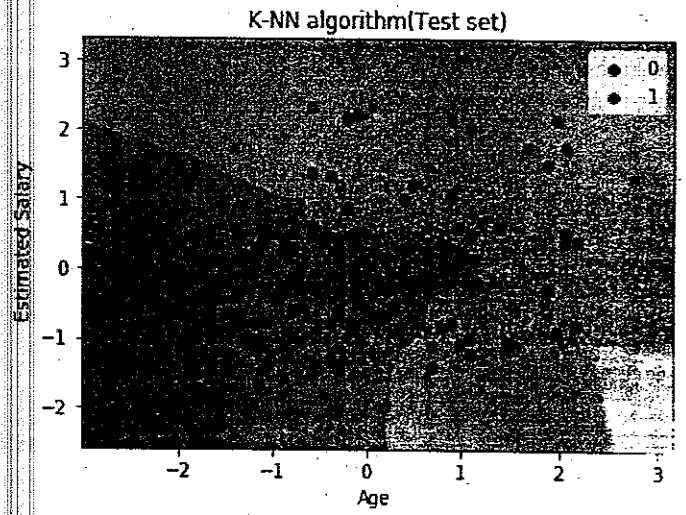


Fig. 4.42.

The above graph is showing the output for the test data set. As we can see in the graph, the predicted output is well good as most of the red points are in the red region and most of the green points are in the green region.

However, there are few green points in the red region and a few red points in the green region. So these are the incorrect observations that we have observed in the confusion matrix(7 Incorrect output).

4.15. GAUSSIAN MIXTURE MODELS

Gaussian Mixture Models (GMMs) assume that there are a certain number of Gaussian distributions, and each of these distributions represent a cluster. Hence, a Gaussian Mixture Model tends to group the data points belonging to a single distribution together.

Let's say we have three Gaussian distributions (more on that in the next section) – GD1, GD2, and GD3. These have a certain mean (μ1, μ2, μ3) and variance (σ1, σ2, σ3) value respectively. For a given set of data points, our GMM would identify the probability of each data point belonging to each of these distributions.

Gaussian Mixture Models are probabilistic models and use the soft clustering approach for distributing the points in different clusters.

Here, we have three clusters that are denoted by three colors – Blue, Green, and Cyan. Let's take the data point highlighted in red. The probability of this point being a part of the blue cluster is 1, while the probability of it being a part of the green or cyan clusters is 0.

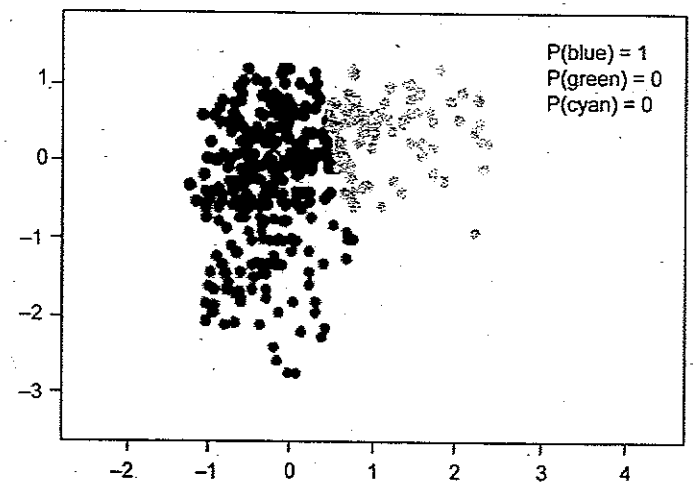


Fig. 4.43.

Now, consider another point – somewhere in between the blue and cyan (highlighted in the below figure). The probability that this point is a part of cluster green is 0, right? And the probability that this belongs to blue and cyan is 0.2 and 0.8 respectively.

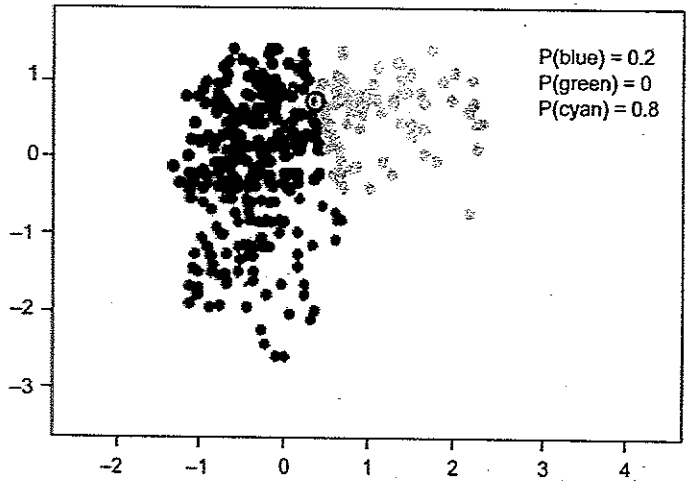


Fig. 4.44.

Gaussian Mixture Models use the soft clustering technique for assigning data points to Gaussian distributions.

4.15.1. THE GAUSSIAN DISTRIBUTION

I'm sure you're familiar with Gaussian Distributions (or the Normal Distribution). It has a bell-shaped curve, with the data points symmetrically distributed around the mean value.

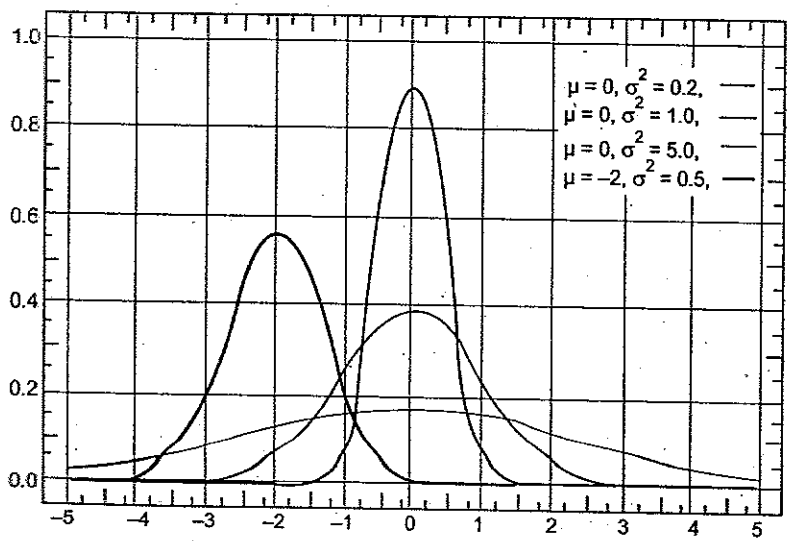


Fig. 4.45.

The above image has a few Gaussian distributions with a difference in mean (μ) and variance (σ^2). Remember that the higher the σ value more would be the spread.

In a one dimensional space, the probability density function of a Gaussian distribution is given by:

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where μ is the mean and σ^2 is the variance.

But this would only be true for a single variable. In the case of two variables, instead of a 2D bell-shaped curve, we will have a 3D bell curve as shown below:

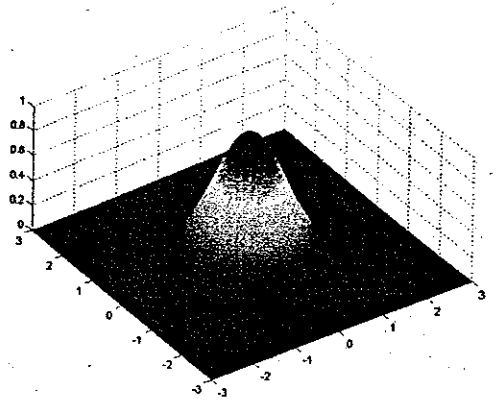


Fig. 4.46.

The probability density function would be given by:

$$f(x \mid \mu, \Sigma) = \frac{1}{\sqrt{2\pi\,|\Sigma|}} \exp\left[-\frac{1}{2}(x-\mu)^t\, \Sigma^{-1}(x-\mu) \right]$$

where x is the input vector, μ is the 2D mean vector, and Σ is the 2×2 covariance matrix. The covariance would now define the shape of this curve. We can generalize the same for d-dimensions.

Thus, this multivariate Gaussian model would have x and μ as vectors of length d, and Σ would be a $d \times d$ covariance matrix.

Hence, for a dataset with d features, we would have a mixture of k Gaussian distributions (where k is equivalent to the number of clusters), each having a certain mean vector and variance matrix. But wait – how is the mean and variance value for each Gaussian assigned?

These values are determined using a technique called Expectation-Maximization (EM). We need to understand this technique before we dive deeper into the working of Gaussian Mixture Models.

A Gaussian is a type of distribution, and it is a popular and mathematically convenient type of distribution. A distribution is a listing of outcomes of an experiment and the probability associated with each outcome. Let's take an example to understand. We have a data table that lists a set of cyclist's speeds.

Speed (Km/h)	Frequency
1	4
2	9
3	6
4	7
5	3
6	2

Here, we can see that a cyclist reaches the speed of 1 Km/h four times, 2Km/h nine times, 3 Km/h and so on. We can notice how this follows, the frequency goes up and then it goes down. It looks like it follows a kind of bell curve the frequencies go up as the speed goes up and then it has a peak value and then it goes down again, and we can represent this using a bell curve otherwise known as a Gaussian distribution.

A Gaussian distribution is a type of distribution where half of the data falls on the left of it, and the other half of the data falls on the right of it. It's an even distribution, and one can notice just by the thought of it intuitively that it is very mathematically convenient.

So, what do we need to define a Gaussian or Normal Distribution? We need a mean which is the average of all the data points. That is going to define the centre of the curve, and the standard deviation which describes how to spread out the data is. Gaussian distribution would be a great distribution to model the data in those cases where the data reaches a peak and then decreases. Similarly, in Multi Gaussian Distribution, we will have multiple peaks with multiple means and multiple standard deviations.

4.15.2. PROBABILITY DENSITY FUNCTION

The formula for Gaussian distribution using the mean and the standard deviation called the Probability Density Function:

$$y = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

where μ = Mean

σ = Standard Deviation

For a given point X, we can compute the associated Y values. Y values are the probabilities for those X values. So, for any X value, we can calculate the probability of that X value being a part of the curve or being a part of the dataset.

This is a function of a continuous random variable whose integral across an interval gives the probability that the value of the variable lies within the same interval.

4.15.3. WHAT IS A GAUSSIAN MIXTURE MODEL?

Sometimes our data has multiple distributions or it has multiple peaks. It does not always have one peak, and one can notice that by looking at the data set. It will look like there are multiple peaks happening here and there. There are two peak points and the data seems to be going up and down twice or maybe three times or four times. But if there are Multiple Gaussian distributions that can represent this data, then we can build what we called a Gaussian Mixture Model.

In other words we can say that, if we have three Gaussian Distribution as GD1, GD2, GD3 having mean as $\mu 1$, $\mu 2$, $\mu 3$ and variance 1,2,3 than for a given set of data points GMM will identify the probability of each data point belonging to each of these distributions.

It is a probability distribution that consists of multiple probability distributions and has Multiple Gaussians.

The probability distribution function of d-dimensions Gaussian Distribution is defined as:

$$N(\mu, \Sigma) = \frac{1}{(2\pi)^2\sqrt{|\Sigma|}} \exp\left[-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right]$$

where μ = Mean

Σ = Covariance Matrix of the Gaussian

d = The numbers of features in our dataset

x = the number of datapoints

4.15.4. WHY DO WE USE THE VARIANCE-COVARIANCE MATRIX?

The Covariance is a measure of how changes in one variable are associated with changes in a second variable. It's not about the independence of variation of two variables but how they change depending on each other. The variance-covariance matrix is a measure of how these variables are related to each other, and in that way it's very similar to the standard deviation except when we have more dimension, the covariance matrix against the standard deviation gives us a better more accurate result.

$$V = \begin{bmatrix} \frac{\Sigma x_1^2}{N} & \frac{\Sigma x_1 x_2}{N} & \cdots & \frac{\Sigma x_1 x_c}{N} & \frac{\Sigma x_2 x_1}{N} & \frac{\Sigma x_2^2}{N} & \cdots & \frac{\Sigma x_2 x_c}{N} & \frac{\Sigma x_c x_1}{N} & \cdots & \frac{\Sigma x_c x_2}{N} & \cdots & \frac{\Sigma x_c^2}{N} \end{bmatrix}$$

Where,

V = $c \times c$ variance-covariance matrix

N = the number of scores in each of the c datasets

x_i = is a deviation score from the i^{th} dataset

x_i^2 / N = is the variance of element from the i^{th} dataset

$x_i \; x_j / N$ = is the covariance for the elements from the i^{th} and j^{th} datasets and the probability given in a mixture of K Gaussian where K is a number of distributions:

$$p(x) = \sum_{j=1}^{K} w_j \, N(x \mid \mu_j, \Sigma_j)$$

where w_j is the prior probability of the j^{th} Gaussian

$$\sum_{j=1}^{K} w_j = 1 \text{ and } 0 \Leftarrow w_j \Leftarrow 1$$

Once we multiply the probability distribution function of d-dimension by W, the prior probability of each of our gaussians, it will give us the probability value X for a given X data point. If we were to plot multiple Gaussian distributions, it would be multiple bell curves. What we really want is a single continuous curve that consists of multiple bell curves. Once we have that huge continuous curve then for the given data points, it can tell us the probability that it is going to belong to a specific class.

Now, we would like to find the maximum likelihood estimate of X (the data point we want to predict the probability) i.e. we want to maximize the likelihood that X belongs to a particular class or we want to find a class that this data point X is most likely to be part of.

It is very similar to the k-means algorithm. It uses the same optimization strategy which is the expectation maximization algorithm.

4.15.5. K-MEANS VS GAUSSIAN MIXTURE MODEL

K-means finds k to minimize $(x - \mu_k)^2$

The Gaussian Mixture Model is, $\frac{(x - \mu_k)^2}{\sigma^2}$

The reason that standard deviation is added into this because in the denominator the 2 takes variation into consideration when it calculates its measurement but K means only calculates conventional Euclidean distance. i.e K-means calculates distance and GM calculates weights.

This means that the k-means algorithm gives you a hard assignment: it either says this is going to be this data point is a part of this class or it's a part of this class. In a lot of cases we just want that hard assignment but in a lot of cases it's better to have a soft assignment. Sometimes we want the maximum probability like: This is going to be 70% likely that it's a part of this class but we also want the probability that it's going to be a part of other classes. It is a list of probability values that it could be a part of multiple distributions, it could be in the middle, it could be 60% likely this class and 40% likely of this class. That's why we incorporate the standard deviation.

4.16. EXPECTATION MAXIMIZATION

The EM algorithm is considered a latent variable model to find the local maximum likelihood parameters of a statistical model, proposed by Arthur Dempster, Nan Laird, and Donald Rubin in 1977. The EM (Expectation-Maximization) algorithm is one of the most commonly used terms in machine learning to obtain maximum likelihood estimates of variables that are sometimes observable and sometimes not. However, it is also applicable to unobserved data or sometimes called latent. It has various real-world applications in statistics, including

obtaining the mode of the posterior marginal distribution of parameters in machine learning and data mining applications.

In most real-life applications of machine learning, it is found that several relevant learning features are available, but very few of them are observable, and the rest are unobservable. If the variables are observable, then it can predict the value using instances. On the other hand, the variables which are latent or directly not observable, for such variables Expectation-Maximization (EM) algorithm plays a vital role to predict the value with the condition that the general form of probability distribution governing those latent variables is known to us. In this topic, we will discuss a basic introduction to the EM algorithm, a flow chart of the EM algorithm, its applications, advantages, and disadvantages of EM algorithm, etc.

4.16.1. WHAT IS AN EM ALGORITHM?

The Expectation-Maximization (EM) algorithm is defined as the combination of various unsupervised machine learning algorithms, which is used to determine the local maximum likelihood estimates (MLE) or maximum a posteriori estimates (MAP) for unobservable variables in statistical models. Further, it is a technique to find maximum likelihood estimation when the latent variables are present. It is also referred to as the latent variable model.

A latent variable model consists of both observable and unobservable variables where observable can be predicted while unobserved are inferred from the observed variable. These unobservable variables are known as latent variables.

Key Points:

- It is known as the latent variable model to determine MLE and MAP parameters for latent variables.
- It is used to predict values of parameters in instances where data is missing or unobservable for learning, and this is done until convergence of the values occurs.

4.16.2. EM ALGORITHM

The EM algorithm is the combination of various unsupervised ML algorithms, such as the k-means clustering algorithm. Being an iterative approach, it consists of two modes. In the first mode, we estimate the missing or latent variables. Hence it is referred to as the Expectation/estimation step (E-step). Further, the other mode is

used to optimize the parameters of the models so that it can explain the data more clearly. The second mode is known as the maximization-step or M-step.

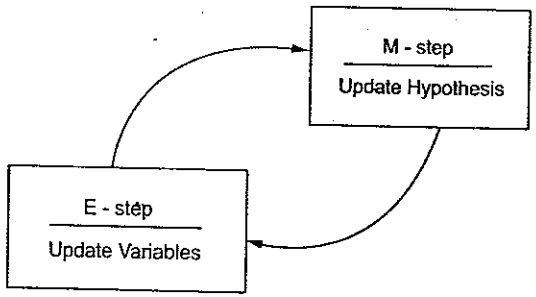


Fig. 4.47.

- **Expectation step (E - step):** It involves the estimation (guess) of all missing values in the dataset so that after completing this step, there should not be any missing value.
- **Maximization step (M - step):** This step involves the use of estimated data in the E-step and updating the parameters.
- **Repeat** E-step and M-step until the convergence of the values occurs.

The primary goal of the EM algorithm is to use the available observed data of the dataset to estimate the missing data of the latent variables and then use that data to update the values of the parameters in the M-step.

4.16.3. CONVERGENCE IN THE EM ALGORITHM

Convergence is defined as the specific situation in probability based on intuition, e.g., if there are two random variables that have very less difference in their probability, then they are known as converged. In other words, whenever the values of given variables are matched with each other, it is called convergence.

4.16.4. STEPS IN EM ALGORITHM

The EM algorithm is completed mainly in 4 steps, which include Initialization Step, Expectation Step, Maximization Step, and convergence Step. These steps are explained as follows:

- **1st Step:** The very first step is to initialize the parameter values. Further, the system is provided with incomplete observed data with the assumption that data is obtained from a specific model.

- **2nd Step:** This step is known as Expectation or E-Step, which is used to estimate or guess the values of the missing or incomplete data using the observed data. Further, E-step primarily updates the variables.
- **3rd Step:** This step is known as Maximization or M-step, where we use complete data obtained from the 2nd step to update the parameter values. Further, M-step primarily updates the hypothesis.
- **4th step:** The last step is to check if the values of latent variables are converging or not. If it gets "yes", then stop the process; else, repeat the process from step 2 until the convergence occurs.

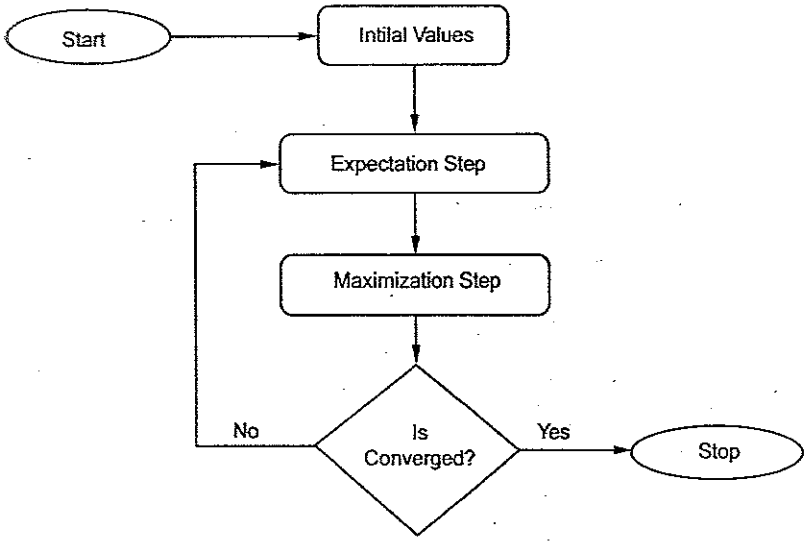


Fig. 4.48.

4.16.5. GAUSSIAN MIXTURE MODEL (GMM)

The Gaussian Mixture Model or GMM is defined as a mixture model that has a combination of the unspecified probability distribution function. Further, GMM also requires estimated statistics values such as mean and standard deviation or parameters. It is used to estimate the parameters of the probability distributions to best fit the density of a given training dataset. Although there are plenty of techniques available to estimate the parameter of the Gaussian Mixture Model (GMM), the Maximum Likelihood Estimation is one of the most popular techniques among them.

Let's understand a case where we have a dataset with multiple data points generated by two different processes. However, both processes contain a similar Gaussian probability distribution and combined data. Hence it is very difficult to discriminate which distribution a given point may belong to.

The processes used to generate the data point represent a latent variable or unobservable data. In such cases, the Estimation-Maximization algorithm is one of the best techniques which helps us to estimate the parameters of the gaussian distributions. In the EM algorithm, E-step estimates the expected value for each latent variable, whereas M-step helps in optimizing them significantly using the Maximum Likelihood Estimation (MLE). Further, this process is repeated until a good set of latent values, and a maximum likelihood is achieved that fits the data.

Even though there are a lot of techniques to estimate the parameters for a Gaussian Mixture Model, the most common technique is the Maximum Likelihood estimation.

Let us consider a case, where the data points are generated by two different processes and each process has a Gaussian probability distribution. But it is unclear, which distribution a given data point belongs to since the data is combined and distributions are similar. And the processes used for generating the data points represent the latent variables and influence the data. The EM algorithm seems like the best approach to estimate the parameters of the distributions.

In the EM algorithm, the E-STEP would estimate the expected value for each latent variable and the M-STEP would optimize the parameters of the distribution using the Maximum Likelihood.

4.16.6. EXAMPLE 1:

Let's say we have a data set where points are generated from one of the two Gaussian processes. The points are one dimensional, the mean is 20 and 40 respectively with a standard deviation 5.

We will draw 4000 points from the first process and 8000 points from the second process and mix them together.

```
from numpy import hstack
from numpy.random import normal
import matplotlib.pyplot as plt

sample1 = normal(loc=20, scale=5 , size=4000)
sample2 = normal(loc=40, scale=5 , size=8000)
sample = hstack((sample1,sample2))
plt.hist(sample, bins=50, density=True)
plt.show()
```

Output:

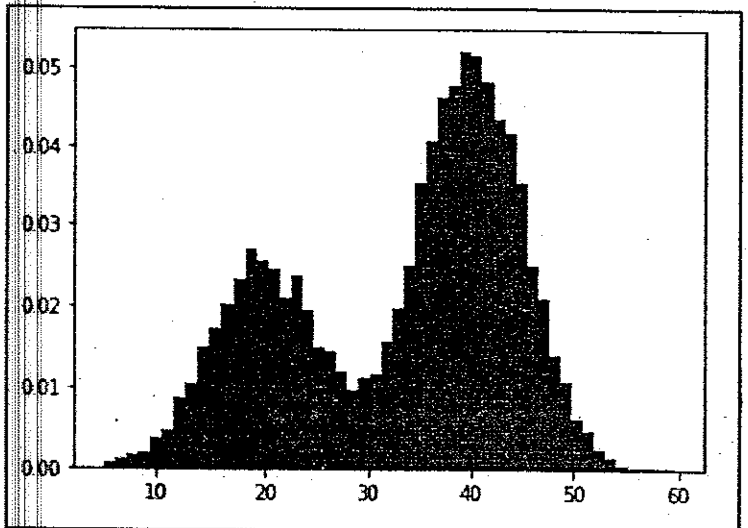


Fig. 4.49.

The plot clearly shows the expected distribution with the peak for the first process is 20 and the second process is 40. And for many points in the middle of the distribution, it is unclear as to which distribution they are picked up from.

We can model the problem of estimating the density of this data set using the Gaussian Mixture Model.


```
# example of fitting a gaussian mixture model with expectation maximization
from numpy import hstack
from numpy.random import normal
from sklearn.mixture import GaussianMixture
# generate a sample
sample1 = normal(loc=20, scale=5, size=4000)
sample2 = normal(loc=40, scale=5, size=8000)
sample = hstack((sample1, sample2))
# reshape into a table with one column
sample = sample.reshape((len(sample), 1))
# fit model
model = GaussianMixture(n_components=2, init_params='random')
model.fit(sample)
# predict latent values
yhat = model.predict(sample)
# check latent value for first few points
print(yhat[:80])
# check latent value for last few points
print(yhat[-80:])
```

Output:

```
(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 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)
(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 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1
 1 0 0 0 0 0)
```

The above example fits the Gaussian mixture model on the data set using the EM algorithm. In this case, we can see that for the first few and the last few examples in the data set, the model mostly predicts the accurate value for the latent variable.

Now that we are clear with the implementation of the EM algorithm using the Gaussian mixture model, let us take a look at other EM algorithm applications as well.

Example 2:

Implementation of GMM in Python

It's time to dive into the code! Here for implementation, we use the **Sklearn Library** of Python.

From sklearn, we use the Gaussian Mixture class which implements the EM algorithm for fitting a mixture of Gaussian models. After object creation, by using the **GaussianMixture.fit** method we can learns a Gaussian Mixture Model from the training data.

Step-1: Import necessary Packages and create an object of the Gaussian Mixture class

Python Code:

Step-2: Fit the created object on the given dataset

```
gmm.fit(np.expand_dims(data, 1))
```

Step-3: Print the parameters of 2 input Gaussians

```
Gaussian_nr = 1
print('Input Normal_distb {:}: μ = {:.2}, σ = {:.2}'.format("1", Mean1, Standard_dev1))
print('Input Normal_distb {:}: μ = {:.2}, σ = {:.2}'.format("2", Mean2, Standard_dev2))
```

Output:

```
Input Normal_distb 1: μ = 2.0, σ = 4.0
Input Normal_distb 2: μ = 9.0, σ = 2.0
```

Step-4: Print the parameters after mixing of 2 Gaussians

```
for mu, sd, p in zip(gmm.means_.flatten(), np.sqrt(gmm.covariances_.flatten()), gmm.weights_):
    print('Normal_distb {:}: μ = {:.2}, σ = {:.2}, weight = {:.2}'.format(Gaussian_nr, mu, sd, p))
    g_s = stats.norm(mu, sd).pdf(x) * p
    plt.plot(x, g_s, label='gaussian sklearn');
    Gaussian_nr += 1
```

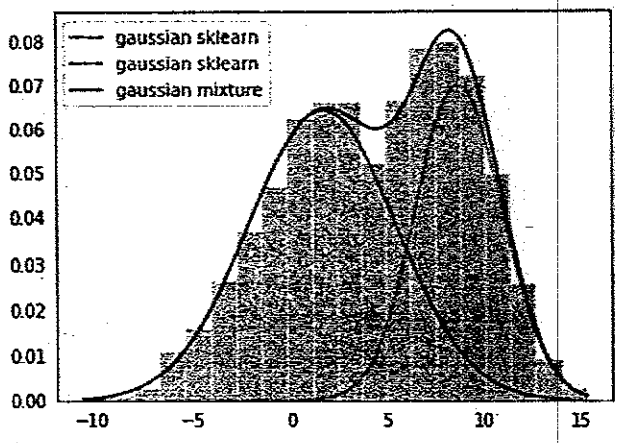

Output:

```
Normal_distb 1: μ = 1.7, σ = 3.8, weight = 0.61
Normal_distb 2: μ = 8.8, σ = 2.2, weight = 0.39
```

Step-5: Plot the distribution plots

```
sns.distplot(data, bins=20, kde=False, norm_hist=True)
gmm_sum = np.exp([gmm.score_samples(e.reshape(-1, 1)) for e in x])
plt.plot(x, gmm_sum, label='gaussian mixture');
plt.legend();
```

Output:



4.16.7. APPLICATIONS OF EM ALGORITHM

The primary aim of the EM algorithm is to estimate the missing data in the latent variables through observed data in datasets. The EM algorithm or latent variable model has a broad range of real-life applications in machine learning. These are as follows:

- The EM algorithm is applicable in data clustering in machine learning.

- It is often used in computer vision and NLP (Natural language processing).
- It is used to estimate the value of the parameter in mixed models such as the Gaussian Mixture Model and quantitative genetics.
- It is also used in psychometrics for estimating item parameters and latent abilities of item response theory models.
- It is also applicable in the medical and healthcare industry, such as in image reconstruction and structural engineering.
- It is used to determine the Gaussian density of a function.

4.16.8. ADVANTAGES OF EM ALGORITHM

- It is very easy to implement the first two basic steps of the EM algorithm in various machine learning problems, which are E-step and M- step.
- It is mostly guaranteed that likelihood will enhance after each iteration.
- It often generates a solution for the M-step in the closed form.

4.16.9. DISADVANTAGES OF EM ALGORITHM

- The convergence of the EM algorithm is very slow.
- It can make convergence for the local optima only.
- It takes both forward and backward probability into consideration. It is opposite to that of numerical optimization, which takes only forward probabilities.

In real-world applications of machine learning, the expectation-maximization (EM) algorithm plays a significant role in determining the local maximum likelihood estimates (MLE) or maximum a posteriori estimates (MAP) for unobservable variables in statistical models. It is often used for the latent variables, i.e., to estimate the latent variables through observed data in datasets. It is generally completed in two important steps, i.e., the expectation step (E-step) and the Maximization step (M-Step), where E-step is used to estimate the missing data in datasets, and M-step is used to update the parameters after the complete data is generated in E-step. Further, the importance of the EM algorithm can be seen in various applications such as data clustering, natural language processing (NLP), computer vision, image reconstruction, structural engineering, etc.

TWO MARKS QUESTIONS WITH ANSWERS (PART - A)

1. ***Define voting.***

   Voting is an ensemble method that combines the performances of multiple models to make predictions.

2. ***List the type of voting.***

   - **Hard Voting:** In hard voting, the predicted output class is a class with the highest majority of votes i.e the class which had the highest probability of being predicted by each of the classifiers. Suppose three classifiers predicted the output class(A, A, B), so here the majority predicted A as output. Hence A will be the final prediction.
   - **Soft Voting:** In soft voting, the output class is the prediction based on the average of probability given to that class.

3. ***What is Plurality Voting.***

   In the case of Majority Voting, the final output predictions are based on the number of votes it gets. If the count of votes is high, that model is taken into consideration.

4. ***What is ensemble learning?***

   Ensemble learning is one of the most powerful machine learning techniques that use the combined output of two or more models/weak learners and solve a particular computational intelligence problem. E.g., a Random Forest algorithm is an ensemble of various decision trees combined.

   Ensemble learning is primarily used to improve the model performance, such as classification, prediction, function approximation, etc. In simple words, we can summarise the ensemble learning as follows:

   "An ensembled model is a machine learning model that combines the predictions from two or more models."

   There are 3 most common ensemble learning methods in machine learning. These are as follows:

   - Bagging
   - Boosting
   - Stacking

5. ***Define Bootstrapping.***

It is a random sampling method that is used to derive samples from the data using the replacement procedure. In this method, first, random data samples are fed to the primary model, and then a base learning algorithm is run on the samples to complete the learning process.

6. ***Define Stacking.***

Stacking is one of the popular ensemble modeling techniques in machine learning. Various weak learners are ensembled in a parallel manner in such a way that by combining them with Meta learners, we can predict better predictions for the future.

7. ***What is max voting?***

The max voting method is generally used for classification problems. In this technique, multiple models are used to make predictions for each data point. The predictions by each model are considered as a _vote'. The predictions which we get from the majority of the models are used as the final prediction

8. ***Define averaging.***

The multiple predictions are made for each data point in averaging. In this method, we take an average of predictions from all the models and use it to make the final prediction. Averaging can be used for making predictions in regression problems or while calculating probabilities for classification problems.

9. ***Define Meta-Model.***

It is made up of the predictions of a set of ML base models (i.e. weak learners) through the k-fold cross validation technique. Finally, the Meta-Model is trained with an additional ML model (which is commonly known as the -final estimator or - final learner).

10. ***What is Stacking Generalization?***

This method is commonly composed of 2 training stages, better known as - level 0 and - level 1. It is important to mention that it can be added as many levels as necessary. However, in practice it is common to use only 2 levels. The aim of the first stage (level 0) is to generate the training data for the meta-model, this is carried out by implementing k-fold cross validation for each weak learner

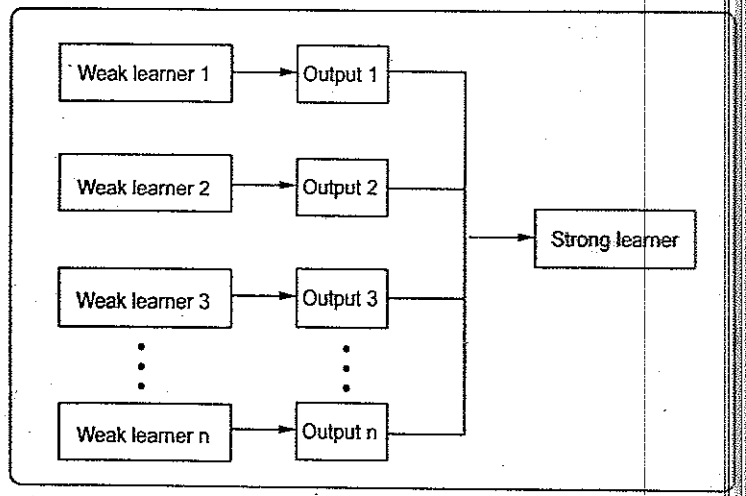
defined in the first stage. The predictions of each one of these weak learners are stacked in order to build such new training set (the meta-model). The aim of the second stage (level 1) is to train the meta-model, such training is carried out through an already determined final learner.

11. ***Define Blending.***

It is a technique derived from Stacking Generalization. The only difference is that in Blending, the k-fold cross validation technique is not used to generate the training data of the meta-model. Blending implements "one-holdout set", that is, a small portion of the training data (validation) to make predictions which will be "stacked" to form the training data of the meta-model. Also, predictions are made from the test data to form the meta-model test data.

12. ***What is boosting?***

Boosting is an ensemble learning technique that uses a set of Machine Learning algorithms to convert weak learner to strong learners in order to increase the accuracy of the model.



13. ***How boosting algorithm works?***

The basic principle behind the working of the boosting algorithm is to generate multiple weak learners and combine their predictions to form one strong rule. These weak rules are generated by applying base Machine Learning algorithms on different distributions of the data set. These algorithms generate weak rules for each iteration. After multiple iterations, the weak learners are combined to form a strong learner that will predict a more accurate outcome.

14. *List the types of boosting.*

There are three main ways through which boosting can be carried out:

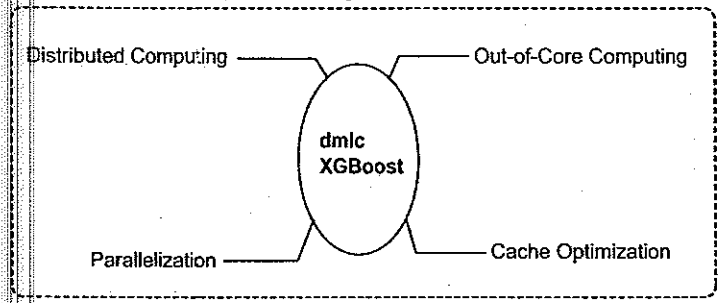
1. Adaptive Boosting or AdaBoost
2. Gradient Boosting
3. XGBoost

15. *Define Gradient Boosting.*

Gradient Boosting is also based on sequential ensemble learning. Here the base learners are generated sequentially in such a way that the present base learner is always more effective than the previous one, i.e. the overall model improves sequentially with each iteration.

16. *Define XGBoost .*

The main aim of this algorithm is to increase the speed and efficiency of computation. The Gradient Descent Boosting algorithm computes the output at a slower rate since they sequentially analyze the data set, therefore XGBoost is used to boost or extremely boost the performance of the model.



17. *What is Unsupervised learning?*

It is a type of machine learning in which models are trained using unlabeled dataset and are allowed to act on that data without any supervision. Unsupervised learning cannot be directly applied to a regression or classification problem because unlike supervised learning, we have the input data but no corresponding output data. The goal of unsupervised learning is to find the underlying structure of dataset, group that data according to similarities, and represent that dataset in a compressed format.

18. *Why use unsupervised learning?*

- Unsupervised learning is helpful for finding useful insights from the data.

- Unsupervised learning is much similar as a human learns to think by their own experiences, which makes it closer to the real AI.
- Unsupervised learning works on unlabeled and uncategorized data which make unsupervised learning more important.
- In real-world, we do not always have input data with the corresponding output so to solve such cases, we need unsupervised learning.

19. *Define Clustering?*

Clustering is a method of grouping the objects into clusters such that objects with most similarities remains into a group and has less or no similarities with the objects of another group. Cluster analysis finds the commonalities between the data objects and categorizes them as per the presence and absence of those commonalities.

20. *Define Association.*

An association rule is an unsupervised learning method which is used for finding the relationships between variables in the large database. It determines the set of items that occurs together in the dataset.

21. *List the advantages of unsupervised learning.*

Unsupervised learning is used for more complex tasks as compared to supervised learning because, in unsupervised learning, we don't have labeled input data.

Unsupervised learning is preferable as it is easy to get unlabeled data in comparison to labeled data.

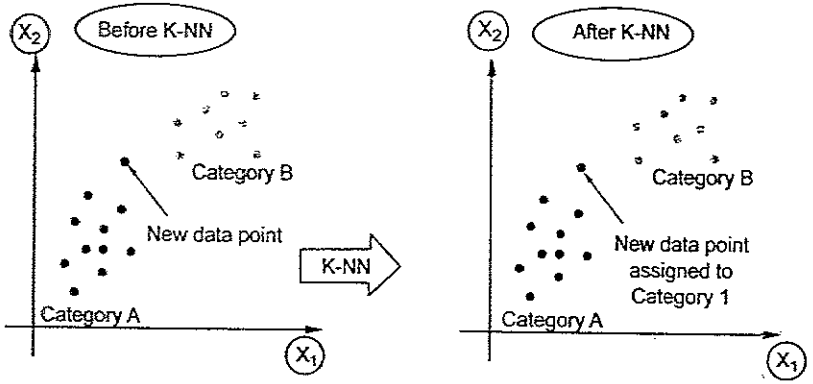
22. *What is k-means algorithm?*

K-Means Clustering is an Unsupervised Learning algorithm, which groups the unlabeled dataset into different clusters. Here K defines the number of pre-defined clusters that need to be created in the process, as if K = 2, there will be two clusters, and for K = 3, there will be three clusters, and so on.

23. *Why do we need a k-NN algorithm?*

Suppose there are two categories, i.e., Category A and Category B, and we have a new data point x_1 , so this data point will lie in which of these categories.

To solve this type of problem, we need a K-NN algorithm. With the help of K-NN, we can easily identify the category or class of a particular dataset. Consider the below diagram



24. ***How to select the optimal K value?***

- There are no pre-defined statistical methods to find the most favorable value of K.
- Initialize a random K value and start computing.
- Choosing a small value of K leads to unstable decision boundaries.
- The substantial K value is better for classification as it leads to smoothening the decision boundaries.
- Derive a plot between error rate and K denoting values in a defined range. Then choose the K value as having a minimum error rate.

25. ***Why do we use the variance-covariance matrix?***

The Covariance is a measure of how changes in one variable are associated with changes in a second variable. It's not about the independence of variation of two variables but how they change depending on each other. The variance-covariance matrix is a measure of how these variables are related to each other, and in that way it's very similar to the standard deviation except when we have more dimension, the covariance matrix against the standard deviation gives us a better more accurate result.

$$V = \begin{bmatrix} \frac{\sum x_1^2}{N} & \frac{\sum x_1 x_2}{N} & \cdots & \frac{\sum x_1 x_c}{N} & \frac{\sum x_2 x_1}{N} & \frac{\sum x_2^2}{N} & \cdots & \frac{\sum x_2 x_c}{N} & \frac{\sum x_c x_1}{N} & \cdots & \frac{\sum x_c x_2}{N} & \cdots & \frac{\sum x_c^2}{N} \end{bmatrix}$$

26. ***What is an EM algorithm?***

The Expectation-Maximization (EM) algorithm is defined as the combination of various unsupervised machine learning algorithms, which is used to determine the local maximum likelihood estimates (MLE) or maximum a posterior estimates (MAP) for unobservable variables in statistical models. Further, it is a technique to find maximum likelihood estimation when the latent variables are present. It is also referred to as the latent variable model.

27. ***What is convergence in the EM algorithm?***

Convergence is defined as the specific situation in probability based on intuition, e.g., if there are two random variables that have very less difference in their probability, then they are known as converged. In other words, whenever the values of given variables are matched with each other, it is called convergence.

REVIEW QUESTIONS

1. Discuss briefly about model combination schemes.
2. Explain in brief about voting.
3. Discuss in detail about ensemble learning.
4. Write short notes on i) Bagging ii) Boosting iii) Stacking
5. Discuss in detail about K-means algorithm.
6. Explain: Instance Based Learning
7. Write short notes on KNN classifier.
8. Discuss in detail about Gaussian mixture models.
9. Discuss briefly about Expectation maximization.

****

UNIT V

NEURAL NETWORKS

Perceptron - Multilayer perceptron, activation functions, network training - gradient descent optimization - stochastic gradient descent, error back propagation, from shallow networks to deep networks – Unit saturation (aka the vanishing gradient problem) - ReLU, hyperparameter tuning, batch normalization, regularization, dropout.

5.1. INTRODUCTION

The term "Artificial Neural Network" is derived from Biological neural networks that develop the structure of a human brain. Similar to the human brain that has neurons interconnected to one another, artificial neural networks also have neurons that are interconnected to one another in various layers of the networks. These neurons are known as nodes.

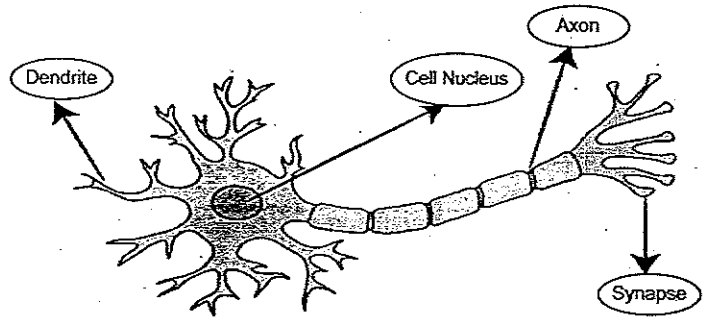


Fig. 5.1. Biological Neural Network

The figure 5.1 illustrates the typical diagram of Biological Neural Network. Figure 5.2 illustrates the typical Artificial Neural Network.

Dendrites from Biological Neural Network represent inputs in Artificial Neural Networks, cell nucleus represents Nodes, synapse represents Weights, and Axon represents Output.

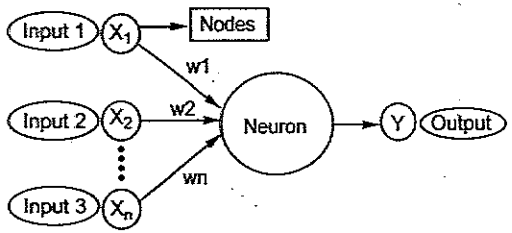


Fig. 5.2. Artificial Neural Network

Relationship between Biological neural network and artificial neural network:

Biological Neural Network	Artificial Neural Network
Dendrites	Inputs
Cell nucleus	Nodes
Synapse	Weights
Axon	Output

Deep Learning

Deep Learning is an advanced field of Machine Learning that uses the concepts of Neural Networks to solve highly-computational use cases that involve the analysis of multi-dimensional data. It automates the process of feature extraction, making sure that very minimal human intervention is needed.

Deep Learning is an advanced sub-field of Machine Learning that uses algorithms inspired by the structure and function of the brain called Artificial Neural Networks.

5.1.1. DIFFERENCE BETWEEN AI, ML, AND DL (ARTIFICIAL INTELLIGENCE VS MACHINE LEARNING VS DEEP LEARNING)

People often tend to think that Artificial Intelligence, Machine Learning, and Deep Learning are the same since they have common applications. For example, Siri is an application of AI, Machine learning and Deep learning.

- **Artificial Intelligence** is the science of getting machines to mimic the behavior of humans.
- **Machine learning** is a subset of Artificial Intelligence (AI) that focuses on getting machines to make decisions by feeding them data.
- **Deep learning** is a subset of Machine Learning that uses the concept of neural networks to solve complex problems.

To sum it up AI, Machine Learning and Deep Learning are interconnected fields. Machine Learning and Deep learning aids Artificial Intelligence by providing a set of algorithms and neural networks to solve data-driven problems.

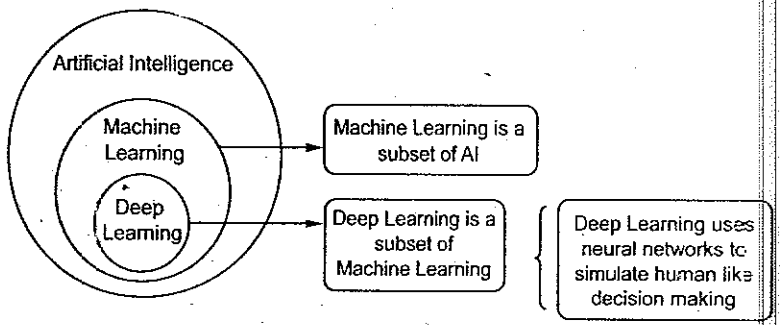


Fig. 5.3. Relationship : AI, DL, ML

5.1.2. NEED FOR DEEP LEARNING: LIMITATIONS OF TRADITIONAL MACHINE LEARNING ALGORITHMS AND TECHNIQUES

Machine Learning was a major breakthrough in the technical world, it led to the automation of monotonous and time-consuming tasks, it helped in solving complex problems and making smarter decisions. However, there were a few drawbacks in Machine learning that led to the emergence of Deep Learning.

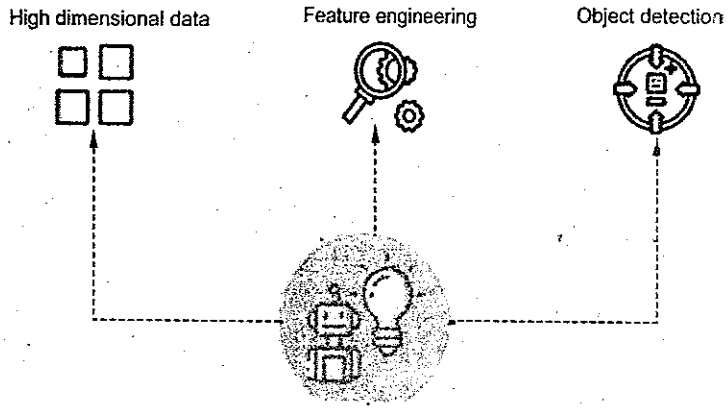


Fig. 5.4.

Here are some limitations of Machine Learning:

1. **Unable to process high dimensional data:** Machine Learning can process only small dimensions of data that contain a small set of variables. If you

want to analyze data containing 100s of variables, then Machine Learning cannot be used.

2. **Feature engineering is manual:** Consider a use case where you have 100 predictor variables and you need to narrow down only the significant ones. To do this you have to manually study the relationship between each of the variables and figure out which ones are important in predicting the output. This task is extremely tedious and time-consuming for a developer.

3. **Not ideal for performing object detection and image processing:** Since object detection requires high-dimensional data, Machine Learning cannot be used to process image data sets, it is only ideal for data sets with a restricted number of features.

An Artificial Neural Network in the field of Artificial intelligence where it attempts to mimic the network of neurons makes up a human brain so that computers will have an option to understand things and make decisions in a human-like manner. The artificial neural network is designed by programming computers to behave simply like interconnected brain cells.

There are around 1000 billion neurons in the human brain. Each neuron has an association point somewhere in the range of 1,000 and 100,000. In the human brain, data is stored in such a manner as to be distributed, and we can extract more than one piece of this data when necessary from our memory parallelly. We can say that the human brain is made up of incredibly amazing parallel processors.

We can understand the artificial neural network with an example, consider an example of a digital logic gate that takes an input and gives an output. "OR" gate, which takes two inputs. If one or both the inputs are "On," then we get "On" in output. If both the inputs are "Off," then we get "Off" in output. Here the output depends upon input. Our brain does not perform the same task. The outputs to inputs relationship keep changing because of the neurons in our brain, which are "learning."

5.1.3. WORKING OF NEURAL NETWORK

To understand neural networks, we need to break it down and understand the most basic unit of a Neural Network, i.e. a Perceptron.

(a) Perceptron

A Perceptron is a single layer neural network that is used to classify linear data. It has 4 important components:

1. Inputs
2. Weights and Bias
3. Summation Function
4. Activation or transformation Function

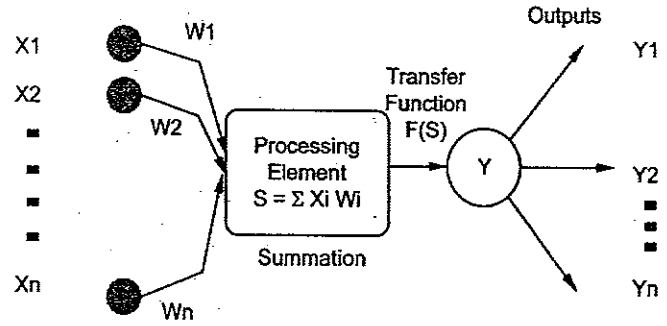


Fig. 5.5. Artificial Neural Network

The basic logic behind a Perceptron is as follows:

The inputs (x) received from the input layer are multiplied with their assigned weights w. The multiplied values are then added to form the Weighted Sum. The weighted sum of the inputs and their respective weights are then applied to a relevant Activation Function. The activation function maps the input to the respective output.

(b) Weights and Bias in Deep Learning

Once an input variable is fed to the network, a randomly chosen value is assigned as the weight of that input. The weight of each input data point indicates how important that input is in predicting the outcome.

The bias parameter, on the other hand, allows you to adjust the activation function curve in such a way that a precise output is achieved.

(c) Summation Function

Once the inputs are assigned some weight, the product of the respective input and weight is taken. Adding all these products gives us the Weighted Sum. This is done by the summation function.

(d) Activation Function

The main aim of the activation functions is to map the weighted sum to the output. Activation functions such as tanh, ReLU, sigmoid and so on are examples of transformation functions.

Example

Have you ever asked Siri a question? The device answers accurately. Let's take a closer look and see how the virtual assistant accomplishes this feat of speech recognition.

There are five recognized types of neural networks.

- Single-layer feed-forward network.
- Multilayer feed-forward network.
- Single node with its own feedback.
- Single-layer recurrent network.
- Multilayer recurrent network.

Consider, for instance, the neural network shown below, an example of a single-layer recurrent network:

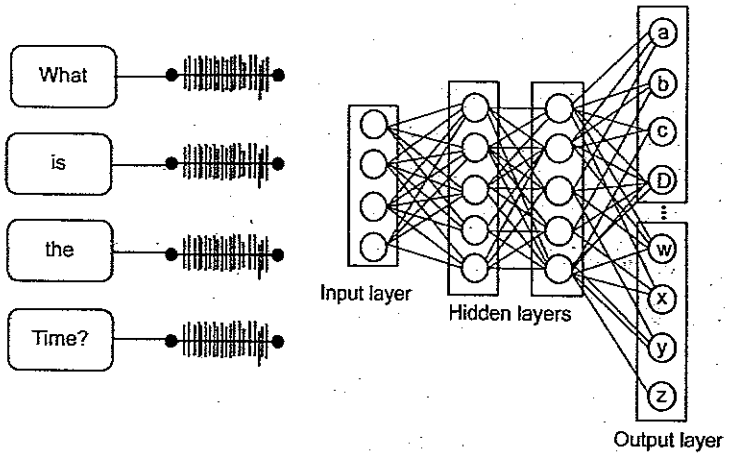


Fig. 5.6.

There are input, hidden, and output layers on the network. But, first, the network needs to recognize the sentence: What is the time?

Here, each word comes in as a pattern of sound. Then, the sentence gets sampled into discrete sound waves.

Let's consider the first word: What

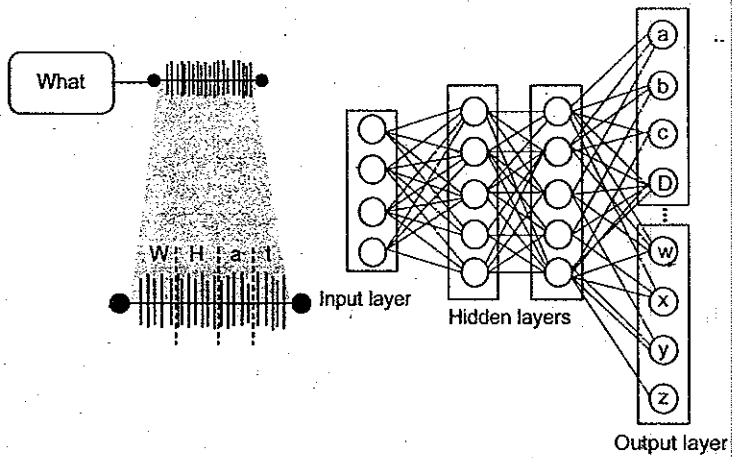


Fig. 5.7.

You can see the waveform is split based on every letter. Now we will split the sound wave for the letter W into smaller segments.

The amplitude varies in the sound wave when we analyze the letter 'W,' as shown below.

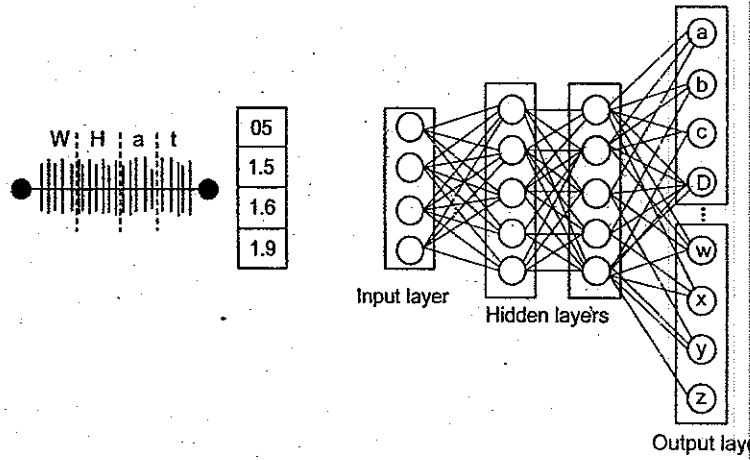


Fig. 5.8.

We collect the values at intervals and form an array. Then, different amplitudes come in for other letters, and we feed the variety of amplitudes to the input layer.

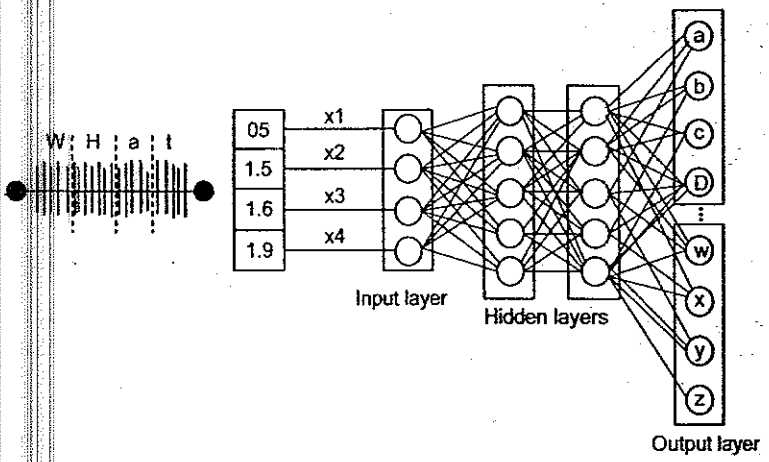


Fig. 5.9.

Random weights get assigned to each interconnection between the input and hidden layers.

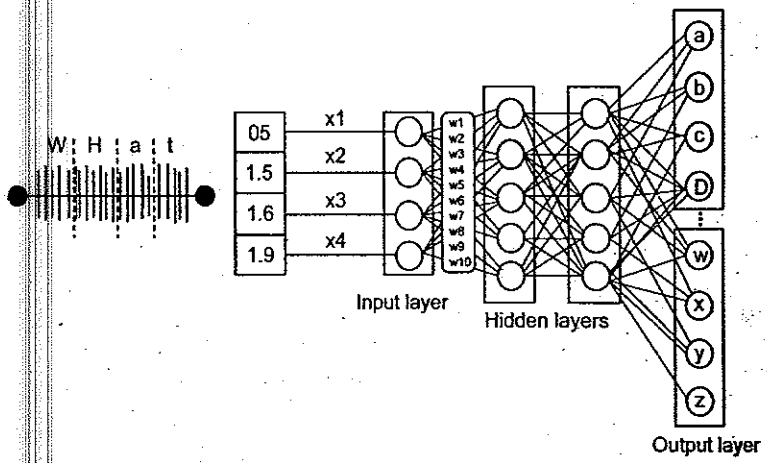


Fig. 5.10.

We always start with the random key, as assigning a preset value to the weights takes a significant amount of time when training the model.

The weights get multiplied with the inputs, and a bias is added to form the transfer function.

$$\sum_{i=1}^{n} w_i \times x_i + b$$

Weights get assigned to the interconnection between the hidden layers. The output of the transfer function is fed as an input to the activation function. The work from one hidden layer becomes the input to the next.

The acoustic model contains the statistical representation of each sound that makes a word. So we start building these acoustic models, and as these layers separate them, they'll start learning what the different models represent for other letters.

The lexicon contains the data for different pronunciations of every word. The lexicon is at the end, where we end up with the ABCD, identifying the other letters there.

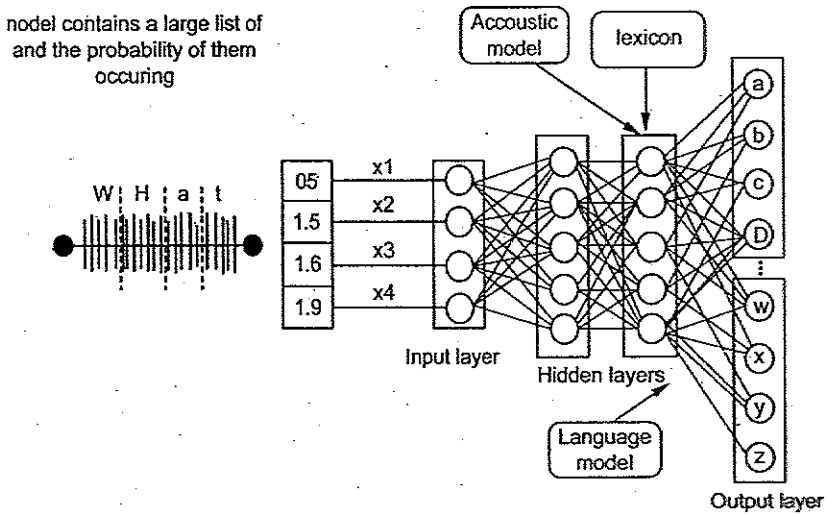


Fig. 5.11.

Finally, we get our output letter. Following the same process for every word and letter, the neural network recognizes the sentence you said or your question.

Note that the terms "acoustic model" and "lexicon" are specific to the domain of understanding speech. When dealing with other input formats, you'll have different labels, but the process remains the same.

Example 2

Consider a scenario where you are to build an Artificial Neural Network (ANN) that classifies images into two classes:

- **Class A:** Containing images of non-diseased leaves

❖ **Class B:** Containing images of diseased leaves

Problem Definition: Create a Neural network that classifies the leaves into diseased and non-diseased crops

The process always begins with processing and transforming the input in such a way that it can be easily processed. In our case, each leaf image will be broken down into pixels depending on the dimension of the image.

For example, if the image is composed of 30 by 30 pixels, then the total number of pixels will be 900. These pixels are represented as matrices, which are then fed into the input layer of the Neural Network.

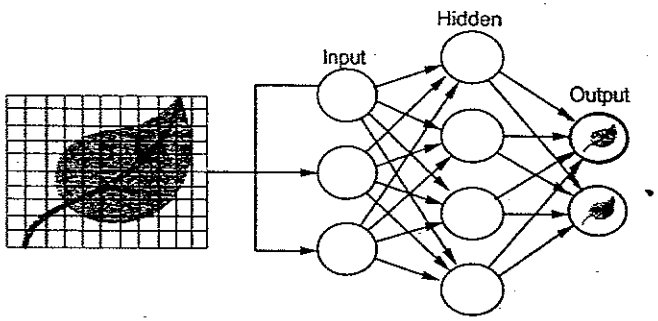


Fig. 5.12.

Just like how our brains have neurons that help in building and connecting thoughts, an ANN has perceptrons that accept inputs and process them by passing them on from the input layer to the hidden and finally the output layer.

As the input is passed from the input layer to the hidden layer, an initial random weight is assigned to each input. The inputs are then multiplied with their corresponding weights and their sum is sent as input to the next hidden layer.

Here, a numerical value called bias is assigned to each perceptron, which is associated with the weightage of each input. Further, each perceptron is passed through activation or a transformation function that determines whether a particular perceptron gets activated or not.

An activated perceptron is used to transmit data to the next layer. In this manner, the data is propagated (Forward propagation) through the neural network until the perceptrons reach the output layer.

At the output layer, a probability is derived which decides whether the data belongs to class A or class B.

5.2. PERCEPTRON

It is the primary step to learn Machine Learning and Deep Learning technologies, which consists of a set of weights, input values or scores, and a threshold. Perceptron is a building block of an Artificial Neural Network. Initially, in the mid of 19th century, Mr. Frank Rosenblatt invented the Perceptron for performing certain calculations to detect input data capabilities or business intelligence. Perceptron is a linear Machine Learning algorithm used for supervised learning for various binary classifiers. This algorithm enables neurons to learn elements and processes them one by one during preparation.

Perceptron is Machine Learning algorithm for supervised learning of various binary classification tasks. Further, Perceptron is also understood as an Artificial Neuron or neural network unit that helps to detect certain input data computations in business intelligence.

Perceptron model is also treated as one of the best and simplest types of Artificial Neural networks. However, it is a supervised learning algorithm of binary classifiers. Hence, we can consider it as a single-layer neural network with four main parameters, i.e., input values, weights and Bias, net sum, and an activation function.

5.2.1. BINARY CLASSIFIER

In Machine Learning, binary classifiers are defined as the function that helps in deciding whether input data can be represented as vectors of numbers and belongs to some specific class.

Binary classifiers can be considered as linear classifiers. In simple words, we can understand it as a classification algorithm that can predict linear predictor function in terms of weight and feature vectors.

5.2.2. PERCEPTRON FUNCTION

Perceptron function "$f(x)$" can be achieved as output by multiplying the input 'x' with the learned weight coefficient 'w'.

Mathematically, we can express it as follows:

$$f(x) = 1; \text{if } w \cdot x + b > 0$$

otherwise, $f(x) = 0$

- 'w' represents real-valued weights vector
- 'b' represents the bias
- 'x' represents a vector of input x values.

5.2.3. BASIC COMPONENTS OF PERCEPTRON

Mr. Frank Rosenblatt invented the perceptron model as a binary classifier which contains three main components. These are as follows:

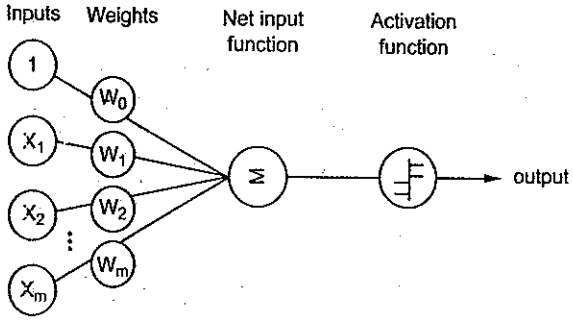


Fig. 5.13. Perceptron

Input Nodes or Input Layer:

This is the primary component of Perceptron which accepts the initial data into the system for further processing. Each input node contains a real numerical value.

Wight and Bias:

Weight parameter represents the strength of the connection between units. This is another most important parameter of Perceptron components. Weight is directly proportional to the strength of the associated input neuron in deciding the output. Further, Bias can be considered as the line of intercept in a linear equation.

Activation Function:

These are the final and important components that help to determine whether the neuron will fire or not. Activation Function can be considered primarily as a step function.

Types of Activation functions:

- Sign function

- Step function, and
- Sigmoid function

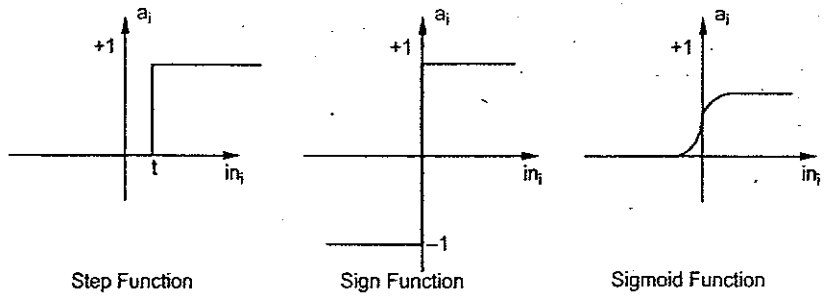


Fig. 5.14.

The data scientist uses the activation function to take a subjective decision based on various problem statements and forms the desired outputs. Activation function may differ (e.g., Sign, Step, and Sigmoid) in perceptron models by checking whether the learning process is slow or has vanishing or exploding gradients.

5.2.4. WORKING OF PERCEPTRON

In Machine Learning, Perceptron is considered as a single-layer neural network that consists of four main parameters named input values (Input nodes), weights and Bias, net sum, and an activation function. The perceptron model begins with the multiplication of all input values and their weights, then adds these values together to create the weighted sum. Then this weighted sum is applied to the activation function 'f' to obtain the desired output. This activation function is also known as the step function and is represented by 'f'.

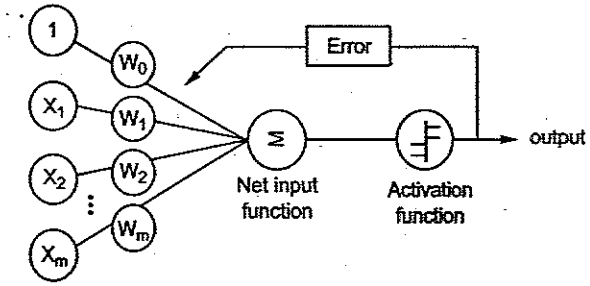


Fig. 5.15.

This step function or Activation function plays a vital role in ensuring that output is mapped between required values (0,1) or (–1,1). It is important to note that the

weight of input is indicative of the strength of a node. Similarly, an input's bias value gives the ability to shift the activation function curve up or down.

Perceptron model works in two important steps as follows:

Step-1

In the first step first, multiply all input values with corresponding weight values and then add them to determine the weighted sum. Mathematically, we can calculate the weighted sum as follows:

$$\sum w_i * x_i = x_1 * w_1 + x_2 * w_2 + \ldots w_n * x_n$$

Add a special term called bias 'b' to this weighted sum to improve the model's performance.

$$\sum w_i * x_i + b$$

Step-2

In the second step, an activation function is applied with the above-mentioned weighted sum, which gives us output either in binary form or a continuous value as follows:

5.2.5. CHARACTERISTICS OF PERCEPTRON

The perceptron model has the following characteristics.

1. Perceptron is a machine learning algorithm for supervised learning of binary classifiers.
2. In Perceptron, the weight coefficient is automatically learned.
3. Initially, weights are multiplied with input features, and the decision is made whether the neuron is fired or not.
4. The activation function applies a step rule to check whether the weight function is greater than zero.
5. The linear decision boundary is drawn, enabling the distinction between the two linearly separable classes +1 and –1.
6. If the added sum of all input values is more than the threshold value, it must have an output signal; otherwise, no output will be shown.

Example

Imagine a perceptron (in your brain).

The perceptron tries to decide if you should go to a concert.

Is the artist good? Is the weather good?

What weights should these facts have?

Criteria	Input	Weight
Artists is Good	x_1 = 0 or 1	w_1 = 0.7
Weather is Good	x_2 = 0 or 1	w_2 = 0.6
Friend will Come	x_3 = 0 or 1	w_3 = 0.5
Food is Served	x_4 = 0 or 1	w_4 = 0.3
Alcohol is Served	x_5 = 0 or 1	w_5 = 0.4

The Perceptron Algorithm

Frank Rosenblatt suggested this algorithm:

1. Set a threshold value
2. Multiply all inputs with its weights
3. Sum all the results
4. Activate the output

1. Set a threshold value:

- Threshold = 1.5

2. Multiply all inputs with its weights:

- $x_1 * w_1 = 1 * 0.7 = 0.7$
- $x_2 * w_2 = 0 * 0.6 = 0$
- $x_3 * w_3 = 1 * 0.5 = 0.5$
- $x_4 * w_4 = 0 * 0.3 = 0$
- $x_5 * w_5 = 1 * 0.4 = 0.4$

3. Sum all the results:

- 0.7 + 0 + 0.5 + 0 + 0.4 = 1.6 (The Weighted Sum)

4. Activate the Output:

- Return true if the sum > 1.5 ("Yes I will go to the Concert")

Note

If the weather weight is 0.6 for you, it might different for someone else. A higher weight means that the weather is more important to them.

If the threshold value is 1.5 for you, it might be different for someone else. A lower threshold means they are more wanting to go to the concert.

5.2.6. TYPES OF PERCEPTRON MODELS

Based on the layers, Perceptron models are divided into two types. These are as follows:

1. Single-layer Perceptron Model
2. Multi-layer Perceptron model

Single Layer Perceptron Model:

This is one of the easiest Artificial neural networks (ANN) types. A single-layered perceptron model consists feed-forward network and also includes a threshold transfer function inside the model. The main objective of the single-layer perceptron model is to analyze the linearly separable objects with binary outcomes.

In a single layer perceptron model, its algorithms do not contain recorded data, so it begins with inconstantly allocated input for weight parameters. Further, it sums up all inputs (weight). After adding all inputs, if the total sum of all inputs is more than a pre-determined value, the model gets activated and shows the output value as +1.

If the outcome is same as pre-determined or threshold value, then the performance of this model is stated as satisfied, and weight demand does not change. However, this model consists of a few discrepancies triggered when multiple weight inputs values are fed into the model. Hence, to find desired output and minimize errors, some changes should be necessary for the weights input.

"Single-layer perceptron can learn only linearly separable patterns."

Multi-Layered Perceptron Model:

Like a single-layer perceptron model, a multi-layer perceptron model also has the same model structure but has a greater number of hidden layers.

The multi-layer perceptron model is also known as the Backpropagation algorithm, which executes in two stages as follows:

- **Forward Stage:** Activation functions start from the input layer in the forward stage and terminate on the output layer.

- **Backward Stage:** In the backward stage, weight and bias values are modified as per the model's requirement. In this stage, the error between actual output and demanded originated backward on the output layer and ended on the input layer.

Hence, a multi-layered perceptron model has considered as multiple artificial neural networks having various layers in which activation function does not remain linear, similar to a single layer perceptron model. Instead of linear, activation function can be executed as sigmoid, TanH, ReLU, etc., for deployment.

A multi-layer perceptron model has greater processing power and can process linear and non-linear patterns. Further, it can also implement logic gates such as AND, OR, XOR, NAND, NOT, XNOR, NOR.

5.2.7. ADVANTAGES OF MULTI-LAYER PERCEPTRON:

- A multi-layered perceptron model can be used to solve complex non-linear problems.
- It works well with both small and large input data.
- It helps us to obtain quick predictions after the training.
- It helps to obtain the same accuracy ratio with large as well as small data.

5.2.8. DISADVANTAGES OF MULTI-LAYER PERCEPTRON:

- In Multi-layer perceptron, computations are difficult and time-consuming.
- In multi-layer Perceptron, it is difficult to predict how much the dependent variable affects each independent variable.
- The model functioning depends on the quality of the training.

5.2.9. LIMITATIONS OF PERCEPTRON MODEL

A perceptron model has limitations as follows:

- The output of a perceptron can only be a binary number (0 or 1) due to the hard limit transfer function.
- Perceptron can only be used to classify the linearly separable sets of input vectors. If input vectors are non-linear, it is not easy to classify them properly.

5.3. MULTILAYER PERCEPTRON

5.3.1. LIMITATIONS OF SINGLE-LAYER PERCEPTRON:

There are two major problems:

- Single-Layer Percpetrons cannot classify non-linearly separable data points.
- Complex problems, that involve a lot of parameters cannot be solved by Single-Layer Perceptrons.

Single-Layer Percpetrons cannot Classify Non-Linearly Separable Data Points

Let us understand this by taking an example of XOR gate. Consider the diagram below:

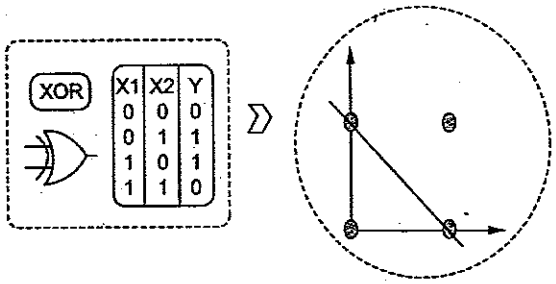


Fig. 5.16.

Here, you cannot separate the high and low points with a single straight line. But, we can separate it by two straight lines. Consider the diagram below:

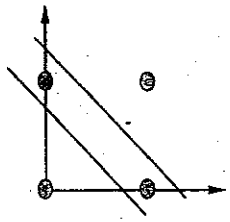


Fig. 5.17.

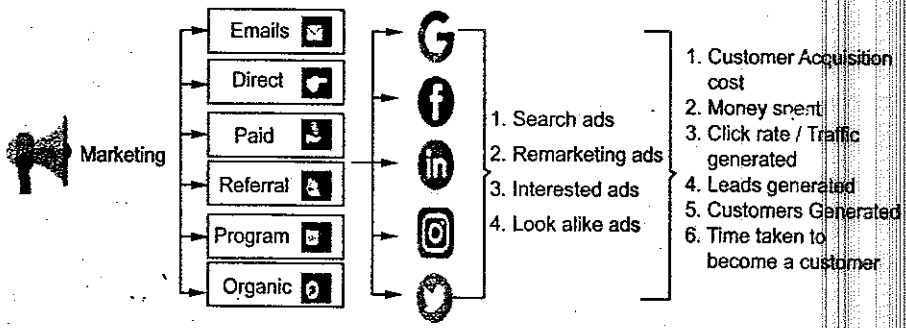
Complex problems, that involve a lot of parameters cannot be solved by Single-Layer Perceptrons:

As an E-commerce firm, you have noticed a decline in your sales. Now, you try to form a marketing team who would market the products for increasing the sales.

The marketing team can market your product through various ways, such as:

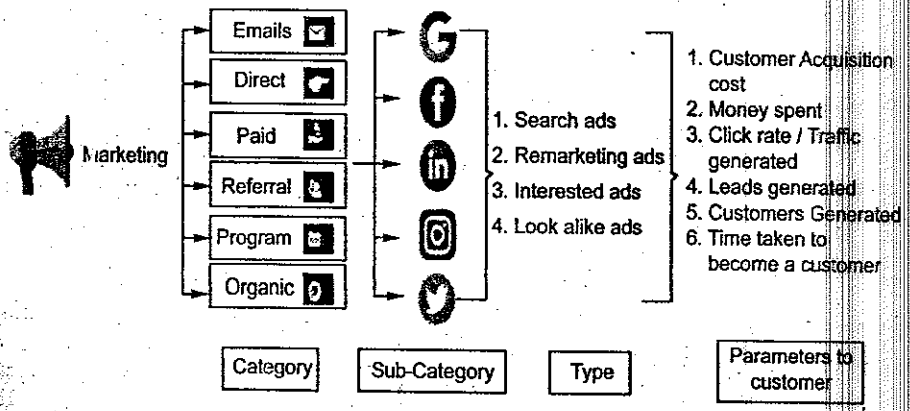
- Google Ads
- Personal emails
- Sale advertisement on relevant sites
- Reference program
- Blogs and so on . . .

Considering all the factors and options available, marketing team has to decide a strategy to do optimal and efficient marketing, but this task is too complex for a human to analyse, because number of parameters are quite high. This problem will have to be solved using Deep Learning. Consider the diagram below:



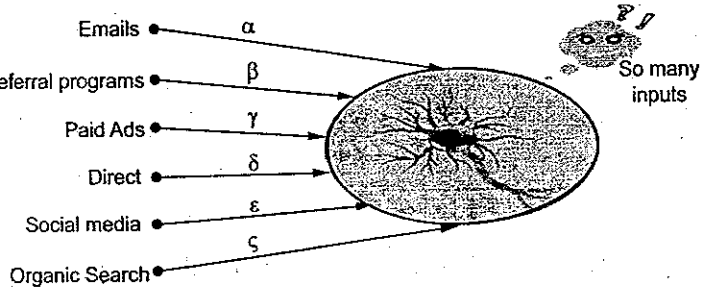
They can either use just one means to market their products or use a variety of them.

Each way would have different advantages and disadvantages as well, they will have to focus on a variety of factors and options such as:



Number of sales that would happen would be dependent on different categorical inputs, their sub categories and their parameters. However, computing and calculating from so many inputs and their sub parameters is not possible just through one neuron (Perceptron).

That is why more than one neuron would be used to solve this problem. Consider the diagram below:



Because of all these reasons, Single-Layer Perceptron cannot be used for complex non-linear problems.

5.3.2. HISTORY OF MULTI-LAYER ANN

Deep Learning deals with training multi-layer artificial neural networks, also called Deep Neural Networks. After Rosenblatt perceptron was developed in the 1950s, there was a lack of interest in neural networks until 1986, when Dr.Hinton and his colleagues developed the backpropagation algorithm to train a multilayer neural network.

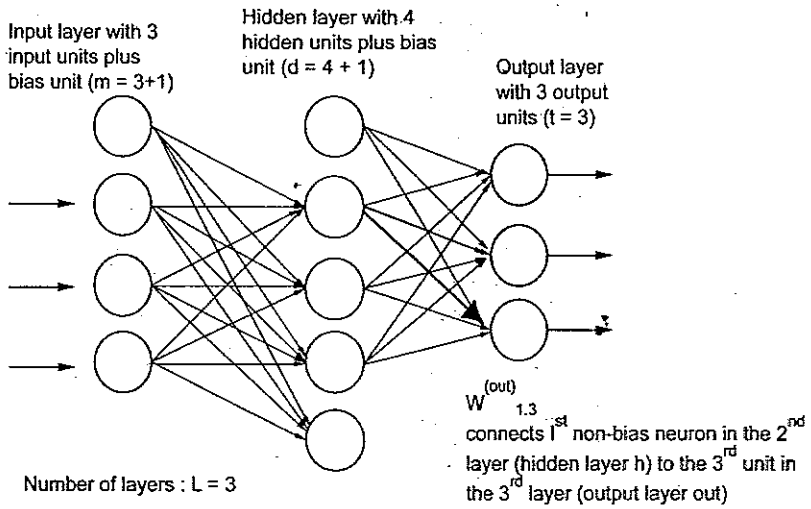
5.3.3. MULTI-LAYER ANN

A fully connected multi-layer neural network is called a Multilayer Perceptron (MLP).

It has 3 layers including one hidden layer. If it has more than 1 hidden layer, it is called a deep ANN. An MLP is a typical example of a feedforward artificial neural network. In this figure 5.21, the i^{th} activation unit in the l^{th} layer is denoted as ai(l).

The number of layers and the number of neurons are referred to as hyperparameters of a neural network, and these need tuning. Cross-validation techniques must be used to find ideal values for these.

The weight adjustment training is done via backpropagation. Deeper neural networks are better at processing data. However, deeper layers can lead to vanishing gradient problems. Special algorithms are required to solve this issue.



Notations

In the representation below:

$$a^{(in)} = \begin{bmatrix} a_0^{(in)} \\ a_1^{(in)} \\ \vdots \\ a_m^{(in)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(in)} \\ \vdots \\ x_m^{(in)} \end{bmatrix}$$

- a_i(in) refers to the i^{th} value in the input layer
- a_i (h) refers to the i^{th} unit in the hidden layer
- a_i (out) refers to the i^{th} unit in the output layer
- a_0(in) is simply the bias unit and is equal to 1; it will have the corresponding weight w_0
- The weight coefficient from layer l to layer l + 1 is represented by $w_{k,j}$(l)

A simplified view of the multilayer is presented here. This image shows a fully connected three-layer neural network with 3 input neurons and 3 output neurons. A bias term is added to the input vector.

5.3.4. IMPLEMENTATION

Step 1: Import the necessary libraries.

```
# importing modules
import tensorflow as tf
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Activation
import matplotlib.pyplot as plt
```

Step 2: Download the dataset.

TensorFlow allows us to read the MNIST dataset and we can load it directly in the program as a train and test dataset.

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

Output:

11493376/11490434 [==============================] – 2s 0us/step

Step 3: Now we will convert the pixels into floating-point values.

```
# Cast the records into float values
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

# normalize image pixel values by dividing
# by 255
gray_scale = 255
x_train /= gray_scale
x_test /= gray_scale
```


We are converting the pixel values into floating-point values to make the predictions. Changing the numbers into grayscale values will be beneficial as the values become small and the computation becomes easier and faster. As the pixel values range from 0 to 256, apart from 0 the range is 255. So dividing all the values by 255 will convert it to range from 0 to 1

Step 4: Understand the structure of the dataset

```
print("Feature matrix:", x_train.shape)
print("Target matrix:", x_test.shape)
print("Feature matrix:", y_train.shape)
print("Target matrix:", y_test.shape)
```

Output:

Feature matrix: (60000, 28, 28)

Target matrix: (10000, 28, 28)

Feature matrix: (60000,)

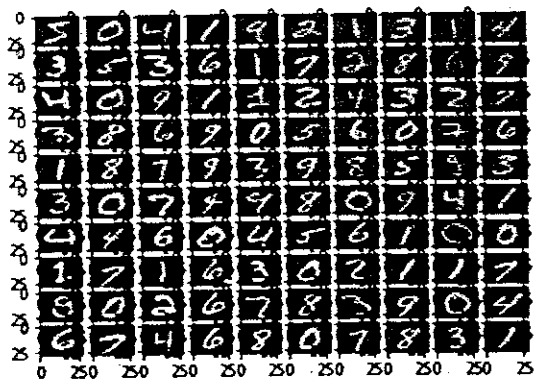
Target matrix: (10000,)

Thus we get that we have 60,000 records in the training dataset and 10,000 records in the test dataset and Every image in the dataset is of the size 28 × 28.

Step 5: Visualize the data.

```
fig, ax = plt.subplots(10, 10)
k = 0
for i in range(10):
    for j in range(10):
        ax[i][j].imshow(x_train[k].reshape(28, 28),
                        aspect='auto')
        k += 1
plt.show()
```

Output



Step 6: Form the Input, hidden, and output layers

```
model = Sequential([
    # reshape 28 row * 28 column data to 28*28 rows
    Flatten(input_shape=(28, 28)),

    # dense layer 1
    Dense(256, activation='sigmoid'),

    # dense layer 2
    Dense(128, activation='sigmoid'),

    # output layer
    Dense(10, activation='sigmoid'),
])
```

Some important points to note:

- The **Sequential model** allows us to create models layer-by-layer as we need in a multi-layer perceptron and is limited to single-input, single-output stacks of layers.

- **Flatten** flattens the input provided without affecting the batch size. For example, If inputs are shaped (batch_size,) without a feature axis, then flattening adds an extra channel dimension and output shape is (batch_size, 1).
- **Activation** is for using the sigmoid activation function.
- The first two **Dense** layers are used to make a fully connected model and are the hidden layers.
- The **last Dense layer** is the output layer which contains 10 neurons that decide which category the image belongs to.

Step 7: Compile the model.

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Compile function is used here that involves the use of loss, optimizers, and metrics. Here loss function used is sparse_categorical_crossentropy, optimizer used is adam.

Step 8: Fit the model.

```
model.fit(x_train, y_train, epochs=10,
          batch_size=2000,
          validation_split=0.2)
```

Output:

```
Epoch 1/10
24/24 [==============================] - 2s 43ms/step - loss: 2.0848 - accuracy: 0.3766 - val_loss: 1.7302 – val_accuracy: 0.6564
Epoch 2/10
24/24 [==============================] - 1s 37ms/step - loss: 1.3903 - accuracy: 0.7211 - val_loss: 1.0328 – val_accuracy: 0.8033
```

```
Epoch 3/10
24/24 [==============================] - 1s 28ms/step - loss: 0.8634 - accuracy: 0.8146 - val_loss: 0.6720 – val_accuracy: 0.8603
Epoch 4/10
24/24 [==============================] - 1s 31ms/step - loss: 0.6047 - accuracy: 0.8661 - val_loss: 0.4967 – val_accuracy: 0.8877
Epoch 5/10
24/24 [==============================] - 1s 28ms/step - loss: 0.4724 - accuracy: 0.8869 - val_loss: 0.4066 – val_accuracy: 0.8993
Epoch 6/10
24/24 [==============================] - 1s 30ms/step - loss: 0.4006 - accuracy: 0.8977 - val_loss: 0.3559 – val_accuracy: 0.9074
Epoch 7/10
24/24 [==============================] - 1s 26ms/step - loss: 0.3564 - accuracy: 0.9051 - val_loss: 0.3221 – val_accuracy: 0.9147
Epoch 8/10
24/24 [==============================] - 1s 30ms/step - loss: 0.3256 - accuracy: 0.9116 - val_loss: 0.2989 – val_accuracy: 0.9195
Epoch 9/10
24/24 [==============================] - 1s 29ms/step - loss: 0.3029 - accuracy: 0.9164 - val_loss: 0.2807 – val_accuracy: 0.9233
Epoch 10/10
24/24 [==============================] - 1s 28ms/step - loss: 0.2847 - accuracy: 0.9205 - val_loss: 0.2662 – val_accuracy: 0.9264
<tensorflow.python.keras.callbacks.History at 0x25dcad04048>
```

Some important points to note:

- **Epochs** tell us the number of times the model will be trained in forwarding and backward passes.
- **Batch Size** represents the number of samples, If it's unspecified, batch_size will default to 32.

- **Validation Split** is a float value between 0 and 1. The model will set apart this fraction of the training data to evaluate the loss and any model metrics at the end of each epoch. (The model will not be trained on this data)

Step 9: Find Accuracy of the model.

```
results = model.evaluate(x_test, y_test, verbose = 0)
print('test loss, test acc:', results)
```

Output:

```
test loss, test acc: [0.27210235595703125, 0.9223999977111816]
```

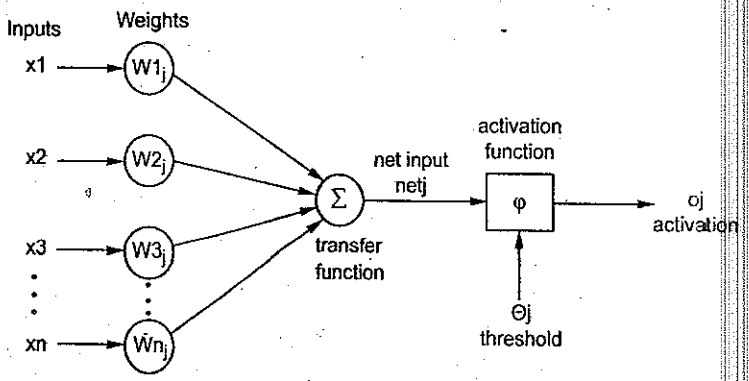
We got the accuracy of our model 92% by using model.evaluate() on the test samples.

5.4. ACTIVATION FUNCTIONS

Activation functions are mathematical equations that determine the output of a neural network model. Activation functions also have a major effect on the neural network's ability to converge and the convergence speed, or in some cases, activation functions might prevent neural networks from converging in the first place. Activation function also helps to normalize the output of any input in the range between 1 to –1 or 0 to 1.

Activation function must be efficient and it should reduce the computation time because the neural network sometimes trained on millions of data points.

Let's consider the simple neural network model without any hidden layers.



Here is the output-

$$Y = \sum (\text{weights} * \text{input} + \text{bias})$$

and it can range from -infinity to +infinity. So it is necessary to bound the output to get the desired prediction or generalized results.

$$Y = \text{Activation function}(\sum (\text{weights}*\text{input} + \text{bias}))$$

So the activation function is an important part of an artificial neural network. They decide whether a neuron should be activated or not and it is a non-linear transformation that can be done on the input before sending it to the next layer of neurons or finalizing the output.

5.4.1. PROPERTIES OF ACTIVATION FUNCTIONS

1. Non Linearity
2. Continuously differentiable
3. Range
4. Monotonic
5. Approximates identity near the origin

5.4.2. TYPES OF ACTIVATION FUNCTIONS

The activation function can be broadly classified into 3 categories.

1. Binary Step Function
2. Linear Activation Function
3. Non-Linear Activation Function

5.4.2.1. Binary Step Function

A binary step function is generally used in the Perceptron linear classifier. It thresholds the input values to 1 and 0, if they are greater or less than zero, respectively.

The step function is mainly used in binary classification and it can't classify the multi-class problems.

The input fed to the activation function is compared to a certain threshold; if the input is greater than it, then the neuron is activated, else it is deactivated, meaning that its output is not passed on to the next hidden layer.

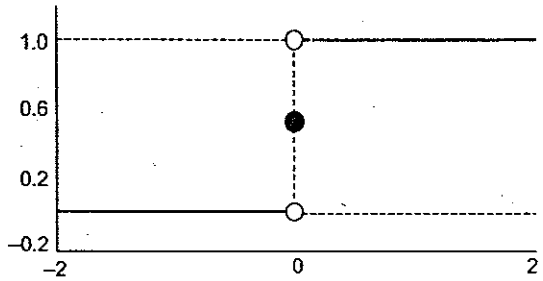


Fig. 5.18. Binary Step Function

Mathematically it can be represented as:

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

Here are some of the limitations of binary step function:

- ❖ It cannot provide multi-value outputs - for example, it cannot be used for multi-class classification problems.
- ❖ The gradient of the step function is zero, which causes a hindrance in the backpropagation process.

5.4.2.2. Linear Activation Function

The linear activation function, also known as "no activation," or "identity function" (multiplied $x_1.0$), is where the activation is proportional to the input.

The function doesn't do anything to the weighted sum of the input, it simply spits out the value it was given.

The equation for Linear activation function is:

$$f(x) = a.\mathbf{x}$$

When $a = 1$ then $f(x) = x$ and this is a special case known as identity.

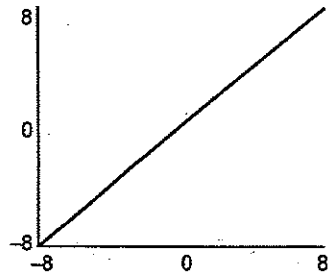


Fig. 5.19. Linear Activation Function

Properties:

1. Range is -infinity to +infinity
2. Provides a convex error surface so optimisation can be achieved faster
3. $df(x) / dx$ = a which is constant. So cannot be optimised with gradient descent

Limitations:

1. It's not possible to use backpropagation as the derivative of the function is a constant and has no relation to the input x.
2. All layers of the neural network will collapse into one if a linear activation function is used. No matter the number of layers in the neural network, the last layer will still be a linear function of the first layer. So, essentially, a linear activation function turns the neural network into just one layer.

5.4.2.3. Non-Linear Activation Functions

Modern neural network models use non-linear activation functions. They allow the model to create complex mappings between the network's inputs and outputs, such as images, video, audio, and data sets that are non-linear or have high dimensionality.

Majorly there are 3 types of Non-Linear Activation functions.

1. Sigmoid Activation Functions
2. Rectified Linear Units or ReLU
3. Complex Nonlinear Activation Functions

(a) Sigmoid / Logistic Activation Function

This function takes any real value as input and outputs values in the range of 0 to 1.

The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to 0.0, as shown in figure 5.25.

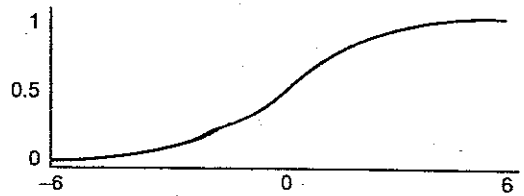


Fig. 5.20. Sigmoid / Logistic Activation Function

Mathematically it can be represented as:

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

Here's why sigmoid/logistic activation function is one of the most widely used functions:

- ❖ It is commonly used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice because of its range.
- ❖ The function is differentiable and provides a smooth gradient, i.e., preventing jumps in output values. This is represented by an S-shape of the sigmoid activation function.

The limitations of sigmoid function are discussed below:

- ❖ The derivative of the function is $f'(x)$ = sigmoid(x)*(1-sigmoid(x)).

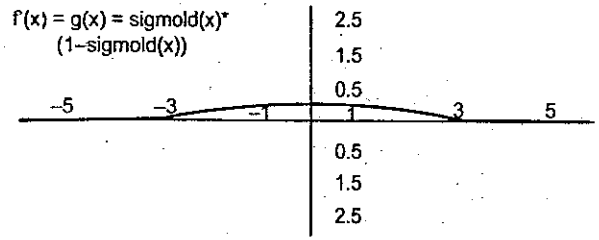


Fig. 5.21. The derivative of the Sigmoid Activation Function

As we can see from the above figure 5.26, the gradient values are only significant for range – 3 to 3, and the graph gets much flatter in other regions.

It implies that for values greater than 3 or less than –3, the function will have very small gradients. As the gradient value approaches zero, the network ceases to learn and suffers from the Vanishing gradient problem.

- ❖ The output of the logistic function is not symmetric around zero. So the output of all the neurons will be of the same sign. This makes the training of the neural network more difficult and unstable.

(b) Tanh Function (Hyperbolic Tangent)

Tanh function is very similar to the sigmoid/logistic activation function, and even has the same S-shape with the difference in output range of –1 to 1. In Tanh, the larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to -1.0.

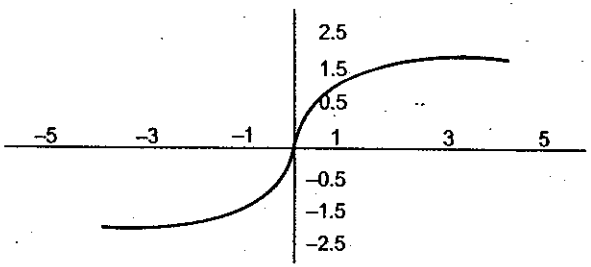


Fig. 5.22. Tanh Function (Hyperbolic Tangent)

Mathematically it can be represented as:

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

Advantages of using this activation function are:

- The output of the tanh activation function is Zero centered; hence we can easily map the output values as strongly negative, neutral, or strongly positive.
- Usually used in hidden layers of a neural network as its values lie between –1 to; therefore, the mean for the hidden layer comes out to be 0 or very close to it. It helps in centering the data and makes learning for the next layer much easier.

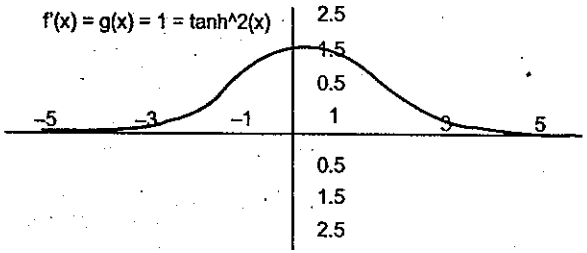


Fig. 5.23. Gradient of the Tanh Activation Function

Limitations

As you can see - it also faces the problem of vanishing gradients similar to the sigmoid activation function. Plus the gradient of the tanh function is much steeper as compared to the sigmoid function.

Note Although both sigmoid and tanh face vanishing gradient issue, tanh is zero centered, and the gradients are not restricted to move in a certain direction. Therefore, in practice, tanh nonlinearity is always preferred to sigmoid nonlinearity.

(c) ReLU Function

ReLU stands for Rectified Linear Unit.

Although it gives an impression of a linear function, ReLU has a derivative function and allows for backpropagation while simultaneously making it computationally efficient.

The main catch here is that the ReLU function does not activate all the neurons at the same time.

The neurons will only be deactivated if the output of the linear transformation is less than 0.

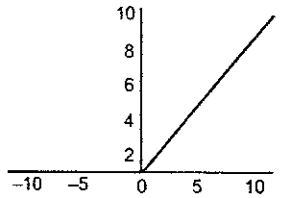


Fig. 5.24. ReLU Activation Function

Mathematically it can be represented as:

$$f(x) = \max(0, x)$$

The advantages of using ReLU as an activation function are as follows:

- Since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh functions.
- ReLU accelerates the convergence of gradient descent towards the global minimum of the loss function due to its linear, non-saturating property.

The limitations faced by this function are:

- The Dying ReLU problem, which I explained below.

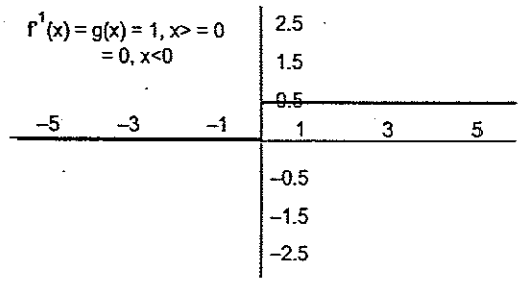


Fig. 5.25. The Dying ReLU problem

The negative side of the graph makes the gradient value zero. Due to this reason, during the backpropagation process, the weights and biases for some neurons are not updated. This can create dead neurons which never get activated.

- ❖ All the negative input values become zero immediately, which decreases the model's ability to fit or train from the data properly.

Note For building the most reliable ML models, split your data into train, validation, and test sets.

(d) Leaky ReLU Function

Leaky ReLU is an improved version of ReLU function to solve the Dying ReLU problem as it has a small positive slope in the negative area.

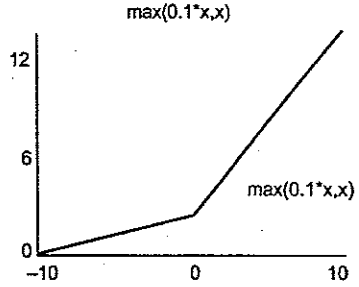


Fig. 5.26. Leaky ReLU

Mathematically it can be represented as:

$$f(x) = \max(0.1x, x)$$

The advantages of Leaky ReLU are same as that of ReLU, in addition to the fact that it does enable backpropagation, even for negative input values.

By making this minor modification for negative input values, the gradient of the left side of the graph comes out to be a non-zero value. Therefore, we would no longer encounter dead neurons in that region.

Here is the derivative of the Leaky ReLU function.

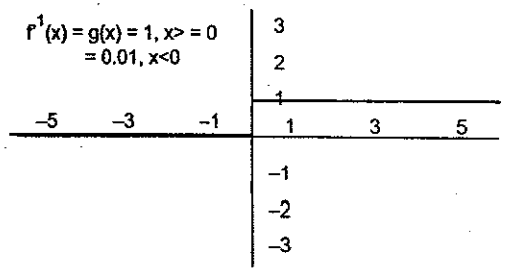


Fig. 5.27. The derivative of the Leaky ReLU function

The limitations that this function faces include:

- ❖ The predictions may not be consistent for negative input values.
- ❖ The gradient for negative values is a small value that makes the learning of model parameters time-consuming.

(e) Parametric ReLU Function

Parametric ReLU is another variant of ReLU that aims to solve the problem of gradient's becoming zero for the left half of the axis.

This function provides the slope of the negative part of the function as an argument *a*. By performing backpropagation, the most appropriate value of *a* is learnt.

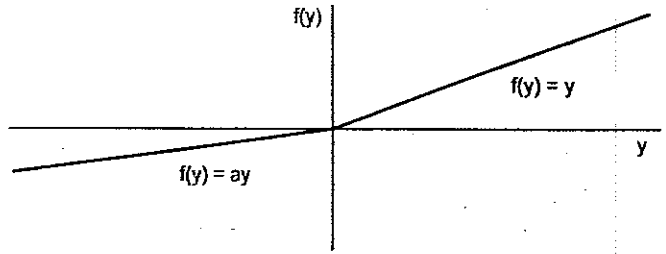


Fig. 5.28. Parametric ReLU

Mathematically it can be represented as:

$$f(x) = \max(ax, x)$$

Where *"a"* is the slope parameter for negative values.

The parameterized ReLU function is used when the leaky ReLU function still fails at solving the problem of dead neurons, and the relevant information is not successfully passed to the next layer.

This function's limitation is that it may perform differently for different problems depending upon the value of slope parameter *a*.

(f) Exponential Linear Units (ELUs) Function

Exponential Linear Unit, or ELU for short, is also a variant of ReLU that modifies the slope of the negative part of the function.

ELU uses a log curve to define the negative values unlike the leaky ReLU and Parametric ReLU functions with a straight line.

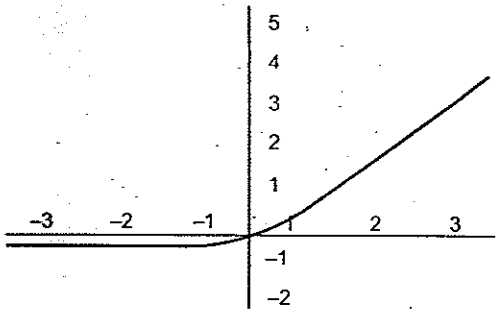


Fig. 5.29. ELU Activation Function

Mathematically it can be represented as:

$$\begin{cases} x & \text{for } x \geq 0 \\ \alpha(e^x - 1) & \text{for } x < 0 \end{cases}$$

ELU is a strong alternative for f ReLU because of the following advantages:

- ELU becomes smooth slowly until its output equal to -α whereas RELU sharply smoothes.
- Avoids dead ReLU problem by introducing log curve for negative values of input. It helps the network nudge weights and biases in the right direction.

The limitations of the ELU function are as follow:

- It increases the computational time because of the exponential operation included
- No learning of the 'a' value takes place
- Exploding gradient problem

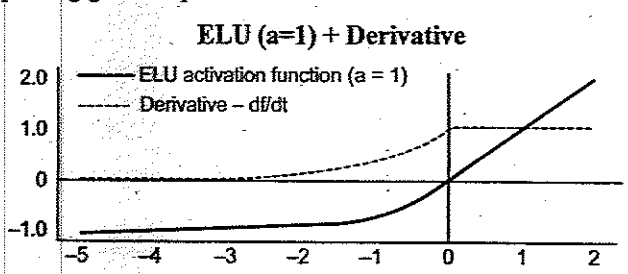


Fig. 5.30. ELU Activation Function and its derivative

Mathematically it can be represented as:

$$f'(x) = \begin{cases} 1 & \text{for } x \geq 0 \\ f(x) + \alpha & \text{for } x < 0 \end{cases}$$

(g) Softmax Function

Before exploring the ins and outs of the Softmax activation function, we should focus on its building block - the sigmoid/logistic activation function that works on calculating probability values.

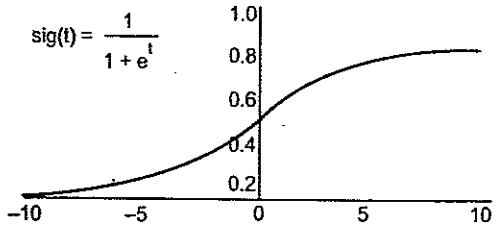


Fig. 5.31. Probability

The output of the sigmoid function was in the range of 0 to 1, which can be thought of as probability.

But this function faces certain problems.

Let's suppose we have five output values of 0.8, 0.9, 0.7, 0.8, and 0.6, respectively. How can we move forward with it?

The answer is: We can't.

The above values don't make sense as the sum of all the classes/output probabilities should be equal to 1.

You see, the Softmax function is described as a combination of multiple sigmoids.

It calculates the relative probabilities. Similar to the sigmoid/logistic activation function, the SoftMax function returns the probability of each class.

It is most commonly used as an activation function for the last layer of the neural network in the case of multi-class classification.

Mathematically it can be represented as:

$$\text{softmax}(Z_i) = \frac{\exp(Z_i)}{\sum_j \exp(Z_i)}$$

Let's go over a simple example together.

Assume that you have three classes, meaning that there would be three neurons in the output layer. Now, suppose that your output from the neurons is [1.8, 0.9, 0.68].

Applying the softmax function over these values to give a probabilistic view will result in the following outcome: [0.58, 0.23, 0.19].

The function returns 1 for the largest probability index while it returns 0 for the other two array indexes. Here, giving full weight to index 0 and no weight to index 1 and index 2. So the output would be the class corresponding to the 1st neuron(index 0) out of three.

You can see now how softmax activation function make things easy for multi-class classification problems.

(h) Swish

It is a self-gated activation function developed by researchers at Google.

Swish consistently matches or outperforms ReLU activation function on deep networks applied to various challenging domains such as image classification, machine translation etc.

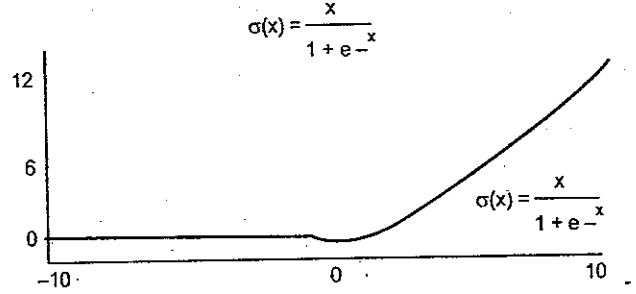


Fig. 5.32. Swish Activation Functions

This function is bounded below but unbounded above i.e. Y approaches to a constant value as X approaches negative infinity but Y approaches to infinity as X approaches infinity.

Mathematically it can be represented as:

$$f(x) = x * \text{sigmoid}(x)$$

Here are a few advantages of the Swish activation function over ReLU.

- Swish is a smooth function that means that it does not abruptly change direction like ReLU does near x = 0. Rather, it smoothly bends from 0 towards values < 0 and then upwards again.
- Small negative values were zeroed out in ReLU activation function. However, those negative values may still be relevant for capturing patterns underlying the data. Large negative values are zeroed out for reasons of sparsity making it a win-win situation.
- The swish function being non-monotonous enhances the expression of input data and weight to be learnt.

(i) Gaussian Error Linear Unit (GELU)

The Gaussian Error Linear Unit (GELU) activation function is compatible with BERT, ROBERTa, ALBERT, and other top NLP models. This activation function is motivated by combining properties from dropout, zoneout, and ReLUs.

ReLU and dropout together yield a neuron's output. ReLU does it deterministically by multiplying the input by zero or one (depending upon the input value being positive or negative) and dropout stochastically multiplying by zero.

RNN regularizer called zoneout stochastically multiplies inputs by one.

We merge this functionality by multiplying the input by either zero or one which is stochastically determined and is dependent upon the input. We multiply the neuron input x by $m \sim \text{Bernoulli}(\Phi(x))$, where $\Phi(x) = P(X \leq x)$, $X \sim N(0, 1)$ is the cumulative distribution function of the standard normal distribution.

This distribution is chosen since neuron inputs tend to follow a normal distribution, especially with Batch Normalization.

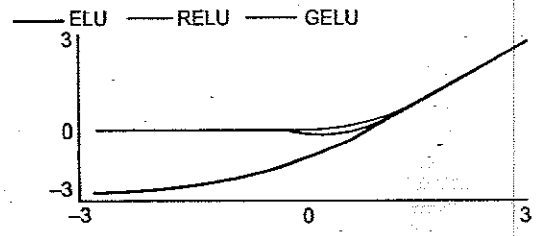


Fig. 5.33. Gaussian Error Linear Unit (GELU) Activation Function

Mathematically it can be represented as:

$$f(x) = xP(X \le x) = x\phi(x)$$
$$= 0.5\,x\left[1 + \tanh\left[\sqrt{2/\pi}\,(x + 0.044715\,x^3)\right]\right]$$

GELU nonlinearity is better than ReLU and ELU activations and finds performance improvements across all tasks in domains of computer vision, natural language processing, and speech recognition.

(j) Scaled Exponential Linear Unit (SELU)

SELU was defined in self-normalizing networks and takes care of internal normalization which means each layer preserves the mean and variance from the previous layers. SELU enables this normalization by adjusting the mean and variance.

SELU has both positive and negative values to shift the mean, which was impossible for ReLU activation function as it cannot output negative values.

Gradients can be used to adjust the variance. The activation function needs a region with a gradient larger than one to increase it.

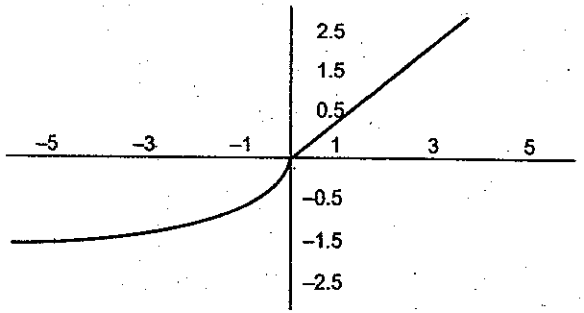


Fig. 5.34. SELU Activation Function

Mathematically it can be represented as:

$$f(\alpha,x) = \lambda\begin{cases}\alpha(e^x - 1) & \text{for } x < 0\\ x & \text{for } x \ge 0\end{cases}$$

SELU has values of alpha α and lambda λ predefined.

Here's the main advantage of SELU over ReLU:

- Internal normalization is faster than external normalization, which means the network converges faster.

SELU is a relatively newer activation function and needs more papers on architectures such as CNNs and RNNs, where it is comparatively explored.

5.5. NETWORK TRAINING

5.5.1. GRADIENT DESCENT OPTIMIZATION

Gradient Descent is known as one of the most commonly used optimization algorithms to train machine learning models by means of minimizing errors between actual and expected results. Further, gradient descent is also used to train Neural Networks.

In mathematical terminology, Optimization algorithm refers to the task of minimizing/maximizing an objective function f(x) parameterized by x. Similarly, in machine learning, optimization is the task of minimizing the cost function parameterized by the model's parameters. The main objective of gradient descent is to minimize the convex function using iteration of parameter updates. Once these machine learning models are optimized, these models can be used as powerful tools for Artificial Intelligence and various computer science applications.

Gradient descent was initially discovered by "Augustin-Louis Cauchy" in mid of 18th century. Gradient Descent is defined as one of the most commonly used iterative optimization algorithms of machine learning to train the machine learning and deep learning models. It helps in finding the local minimum of a function.

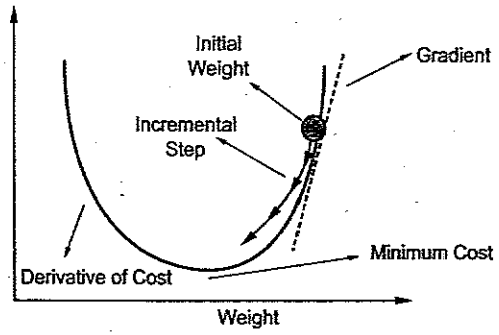


Fig. 5.35.

The best way to define the local minimum or local maximum of a function using gradient descent is as follows:

- If we move towards a negative gradient or away from the gradient of the function at the current point, it will give the local minimum of that function.

- Whenever we move towards a positive gradient or towards the gradient of the function at the current point, we will get the local maximum of that function.

This entire procedure is known as Gradient Ascent, which is also known as steepest descent. The main objective of using a gradient descent algorithm is to minimize the cost function using iteration. To achieve this goal, it performs two steps iteratively:

- Calculates the first-order derivative of the function to compute the gradient or slope of that function.
- Move away from the direction of the gradient, which means slope increased from the current point by alpha times, where Alpha is defined as Learning Rate. It is a tuning parameter in the optimization process which helps to decide the length of the steps.

Cost-function

The cost function is defined as the measurement of difference or error between actual values and expected values at the current position and present in the form of a single real number. It helps to increase and improve machine learning efficiency by providing feedback to this model so that it can minimize error and find the local or global minimum. Further, it continuously iterates along the direction of the negative gradient until the cost function approaches zero. At this steepest descent point, the model will stop learning further. Although cost function and loss function are considered synonymous, also there is a minor difference between them. The slight difference between the loss function and the cost function is about the error within the training of machine learning models, as loss function refers to the error of one training example, while a cost function calculates the average error across an entire training set.

The cost function is calculated after making a hypothesis with initial parameters and modifying these parameters using gradient descent algorithms over known data to reduce the cost function.

For completeness, below are examples of cost functions used within machine learning:

- Mean Squared Error

- Categorical Cross-Entropy
- Binary Cross-Entropy
- Logarithmic Loss

5.5.1.1. Working of Gradient Descent

In many machine learning models, our ultimate goal is to find the best parameter values that reduce the cost associated with the predictions. To do this, we initially start with random values of these parameters and try to find the optimal ones. To find the optimal values, we use the gradient descent algorithm.

1. Start with random initial values for the parameters.
2. Predict the value of the target variable using the current parameters.

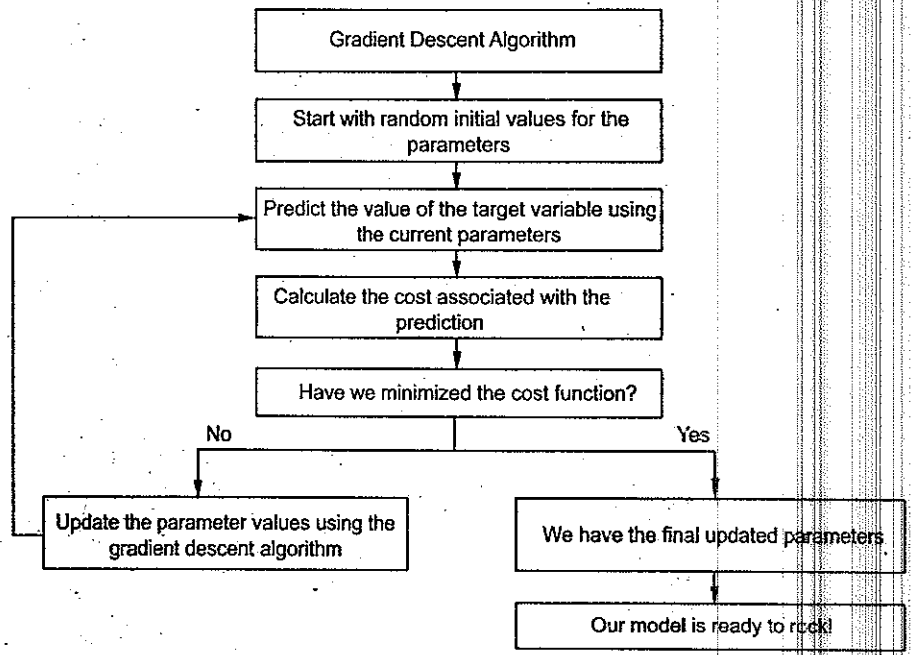


Fig. 5.36. How the gradient descent algorithm works

3. Calculate the cost associated with the prediction.
4. Have we minimized the cost? If yes, then go to step -6. If no, then go to step - 5.
5. Update the parameter values using the gradient descent algorithm and return to step -2.

6. We have our final updated parameters.
7. Our model is ready to roll (down the mountain)!

Before starting the working principle of gradient descent, we should know some basic concepts to find out the slope of a line from linear regression. The equation for simple linear regression is given as:

$$Y = mX + c$$

Where 'm' represents the slope of the line, and 'c' represents the intercepts on the y-axis.

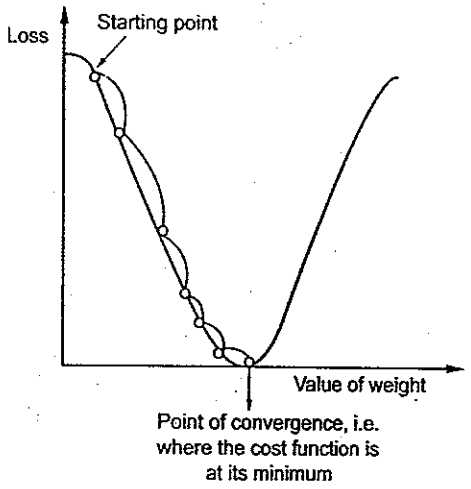


Fig. 5.37.

The starting point(shown in above figure 5.42) is used to evaluate the performance as it is considered just as an arbitrary point. At this starting point, we will derive the first derivative or slope and then use a tangent line to calculate the steepness of this slope. Further, this slope will inform the updates to the parameters (weights and bias).

The slope becomes steeper at the starting point or arbitrary point, but whenever new parameters are generated, then steepness gradually reduces, and at the lowest point, it approaches the lowest point, which is called a point of convergence.

The main objective of gradient descent is to minimize the cost function or the error between expected and actual. To minimize the cost function, two data points are required:

Direction & Learning Rate

These two factors are used to determine the partial derivative calculation of future iteration and allow it to the point of convergence or local minimum or global minimum. Let's discuss learning rate factors in brief;

Learning Rate:

It is defined as the step size taken to reach the minimum or lowest point. This is typically a small value that is evaluated and updated based on the behavior of the cost function. If the learning rate is high, it results in larger steps but also leads to risks of overshooting the minimum. At the same time, a low learning rate shows the small step sizes, which compromises overall efficiency but gives the advantage of more precision.

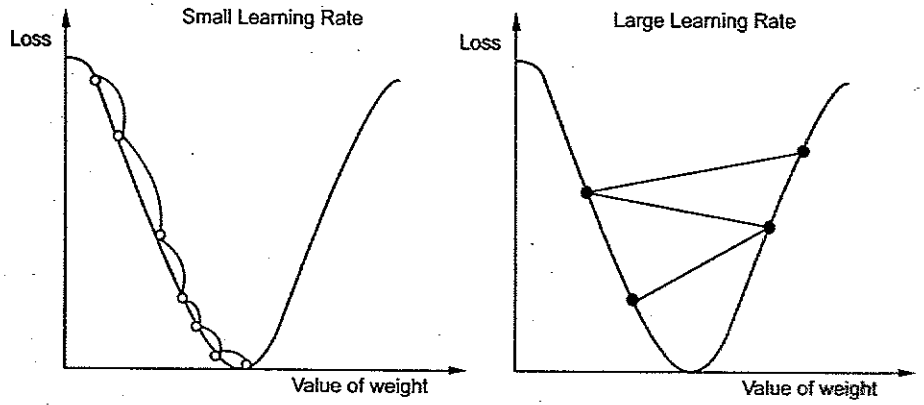


Fig. 5.38.

5.5.1.2. Types of gradient descent

There are three types of gradient descent learning algorithms: batch gradient descent, stochastic gradient descent and mini-batch gradient descent.

(a) Batch gradient descent

Batch gradient descent sums the error for each point in a training set, updating the model only after all training examples have been evaluated. This process referred to as a training epoch.

While this batching provides computation efficiency, it can still have a long processing time for large training datasets as it still needs to store all of the data into memory. Batch gradient descent also usually produces a stable error gradient and

convergence, but sometimes that convergence point isn't the most ideal, finding the local minimum versus the global one.

(b) Stochastic gradient descent

Stochastic gradient descent (SGD) runs a training epoch for each example within the dataset and it updates each training example's parameters one at a time. Since you only need to hold one training example, they are easier to store in memory. While these frequent updates can offer more detail and speed, it can result in losses in computational efficiency when compared to batch gradient descent. Its frequent updates can result in noisy gradients, but this can also be helpful in escaping the local minimum and finding the global one.

(c) Mini-batch gradient descent

Mini-batch gradient descent combines concepts from both batch gradient descent and stochastic gradient descent. It splits the training dataset into small batch sizes and performs updates on each of those batches. This approach strikes a balance between the computational efficiency of batch gradient descent and the speed of stochastic gradient descent.

5.5.1.3. Challenges with gradient descent

While gradient descent is the most common approach for optimization problems, it does come with its own set of challenges. Some of them include:

(a) Local minima and saddle points

For convex problems, gradient descent can find the global minimum with ease, but as nonconvex problems emerge, gradient descent can struggle to find the global minimum, where the model achieves the best results.

Recall that when the slope of the cost function is at or close to zero, the model stops learning. A few scenarios beyond the global minimum can also yield this slope, which are local minima and saddle points. Local minima mimic the shape of a global minimum, where the slope of the cost function increases on either side of the current point. However, with saddle points, the negative gradient only exists on one side of the point, reaching a local maximum on one side and a local minimum on the other. Its name inspired by that of a horse's saddle.

Noisy gradients can help the gradient escape local minimums and saddle points.

(b) Vanishing and Exploding Gradients

In deeper neural networks, particular recurrent neural networks, we can also encounter two other problems when the model is trained with gradient descent and backpropagation.

- **Vanishing gradients:** This occurs when the gradient is too small. As we move backwards during backpropagation, the gradient continues to become smaller, causing the earlier layers in the network to learn more slowly than later layers. When this happens, the weight parameters update until they become insignificant - i.e. 0 - resulting in an algorithm that is no longer learning.
- **Exploding gradients:** This happens when the gradient is too large, creating an unstable model. In this case, the model weights will grow too large, and they will eventually be represented as NaN. One solution to this issue is to leverage a dimensionality reduction technique, which can help to minimize complexity within the model.

5.6 STOCHASTIC GRADIENT DESCENT

There are a few downsides of the gradient descent algorithm. We need to take a closer look at the amount of computation we make for each iteration of the algorithm.

Say we have 10,000 data points and 10 features. The sum of squared residuals consists of as many terms as there are data points, so 10000 terms in our case. We need to compute the derivative of this function with respect to each of the features, so in effect we will be doing 10000 * 10 = 100,000 computations per iteration. It is common to take 1000 iterations, in effect we have 100,000 * 1000 = 100000000 computations to complete the algorithm. That is pretty much an overhead and hence gradient descent is slow on huge data.

Stochastic gradient descent comes to our rescue !! "Stochastic", in plain terms means "random".

Where can we potentially induce randomness in our gradient descent algorithm??

Yes, you might have guessed it right !! It is while selecting data points at each step to calculate the derivatives. SGD randomly picks one data point from the whole data set at each iteration to reduce the computations enormously.

In SGD, it uses only a single sample, i.e., a batch size of one, to perform each iteration. The sample is randomly shuffled and selected for performing the iteration.

5.6.1. SGD ALGORITHM:

for i in range (m)

$$\theta_3 = \theta_3 - \alpha\,(\tilde{y} - y^i)\,x_j^i$$

So, in SGD, we find out the gradient of the cost function of a single example at each iteration instead of the sum of the gradient of the cost function of all the examples.

In SGD, since only one sample from the dataset is chosen at random for each iteration, the path taken by the algorithm to reach the minima is usually noisier than your typical Gradient Descent algorithm. But that doesn't matter all that much because the path taken by the algorithm does not matter, as long as we reach the minima and with a significantly shorter training time.

The path is taken by Batch Gradient Descent as shown below as follows:

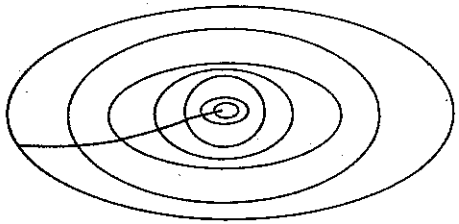


Fig. 5.39.

A path has been taken by Stochastic Gradient Descent –

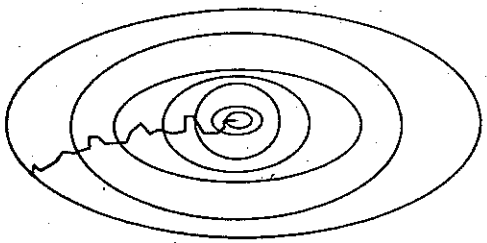


Fig. 5.40.

It is also common to sample a small number of data points instead of just one point at each step and that is called "mini-batch" gradient descent. Mini-batch tries to strike a balance between the goodness of gradient descent and speed of SGD.

The optimizer is an algorithm that adjusts the weights to minimize the loss.

Virtually all of the optimization algorithms used in deep learning belong to a family called stochastic gradient descent. They are iterative algorithms that train a network in steps. One step of training goes like this:

1. Sample some training data and run it through the network to make predictions.
2. Measure the loss between the predictions and the true values.
3. Finally, adjust the weights in a direction that makes the loss smaller.

Then just do this over and over until the loss is as small as you like (or until it won't decrease any further.)

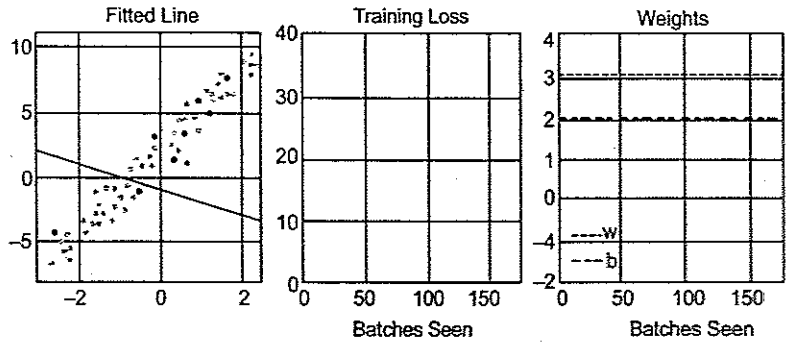


Fig. 5.41.

5.6.2. TRAINING A NEURAL NETWORK WITH STOCHASTIC GRADIENT DESCENT.

Each iteration's sample of training data is called a minibatch (or often just "batch"), while a complete round of the training data is called an epoch. The number of epochs you train for is how many times the network will see each training example.

The animation shows the linear model from Lesson 1 being trained with SGD. The pale red dots depict the entire training set, while the solid red dots are the minibatches. Every time SGD sees a new minibatch, it will shift the weights (w the slope and b the y-intercept) toward their correct values on that batch. Batch after batch, the line eventually converges to its best fit. You can see that the loss gets smaller as the weights get closer to their true values.

5.6.3. LEARNING RATE AND BATCH SIZE

Notice that the line only makes a small shift in the direction of each batch (instead of moving all the way). The size of these shifts is determined by the learning rate. A smaller learning rate means the network needs to see more minibatches before its weights converge to their best values.

The learning rate and the size of the minibatches are the two parameters that have the largest effect on how the SGD training proceeds. Their interaction is often subtle and the right choice for these parameters isn't always obvious. (We'll explore these effects in the exercise.)

Fortunately, for most work it won't be necessary to do an extensive hyperparameter search to get satisfactory results. Adam is an SGD algorithm that has an adaptive learning rate that makes it suitable for most problems without any parameter tuning (it is "self tuning", in a sense). Adam is a great general-purpose optimizer.

Adding the Loss and Optimizer

After defining a model, you can add a loss function and optimizer with the model's compile method:

```
model.compile(
  optimizer="adam",
  loss="mae",
)
```

5.6.4. EXAMPLE - RED WINE QUALITY

Dataset : Red Wine Quality dataset.

This dataset consists of physiochemical measurements from about 1600 Portuguese red wines. Also included is a quality rating for each wine from blind taste-tests. We have to predict a wine's perceived quality from these measurements

We've put all of the data preparation into this next hidden cell. One thing you might note for now though is that we've rescaled each feature to lie in the interval [0,1][0,1].

	Fixed acidity	Volatile acidity	Citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
1109	10.8	0.470	0.43	2.10	0.171	27.0	66.0	0.99820	3.17	0.76	10.8	6
1032	8.1	0.820	0.00	4.10	0.095	5.0	14.0	0.99854	3.36	0.53	9.6	5
1002	9.1	0.290	0.33	2.05	0.063	13.0	27.0	0.99516	3.26	0.84	11.7	7
487	10.2	0.645	0.36	1.80	0.053	5.0	14.0	0.99820	3.17	0.42	10.0	6

Be sure not to include the target ('quality') here -- only the input features.

In [2]:

```
print(X_train.shape)
```

(1119, 11)

Eleven columns means eleven inputs.

We've chosen a three-layer network with over 1500 neurons. This network should be capable of learning fairly complex relationships in the data.

In [3]:

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
  layers.Dense(512, activation='relu', input_shape=[11]),
  layers.Dense(512, activation='relu'),
  layers.Dense(512, activation='relu'),
  layers.Dense(1),
])
```

Creating new thread pool with default inter op setting: 2. Tune using inter_op_parallelism_threads for best performance.

Deciding the architecture of your model should be part of a process. Start simple and use the validation loss as your guide. You'll learn more about model development in the exercises.

After defining the model, we compile in the optimizer and loss function.

In [4]:

```
model.compile(
    optimizer='adam',
    loss='mae',
)
```

Now we're ready to start the training! We've told Keras to feed the optimizer 256 rows of the training data at a time (the batch_size) and to do that 10 times all the way through the dataset (the epochs).

In [5]:

```
history = model.fit(
    X_train, y_train,
    validation_data=(X_valid, y_valid),
    batch_size=256,
    epochs=10,
)
```

```
None of the MLIR Optimization Passes are enabled (registered 2)
Epoch 1/10
5/5 [==============================] - 1s 73ms/step - loss: 0.2626 - val_loss: 0.1382
Epoch 2/10
5/5 [==============================] - 0s 24ms/step - loss: 0.1382 - val_loss: 0.1243
Epoch 3/10
5/5 [==============================] - 0s 17ms/step - loss: 0.1246 - val_loss: 0.1180
Epoch 4/10
```



```
5/5 [==============================] - 0s 17ms/step - loss: 0.1141 - val_loss: 0.1096
Epoch 5/10
5/5 [==============================] - 0s 29ms/step - loss: 0.1127 - val_loss: 0.1091
Epoch 6/10
5/5 [==============================] - 0s 24ms/step - loss: 0.1096 - val_loss: 0.1044
Epoch 7/10
5/5 [==============================] - 0s 22ms/step - loss: 0.1049 - val_loss: 0.1025
Epoch 8/10
5/5 [==============================] - 0s 17ms/step - loss: 0.1054 - val_loss: 0.1021
Epoch 9/10
5/5 [==============================] - 0s 21ms/step - loss: 0.1016 - val_loss: 0.1010
Epoch 10/10
5/5 [==============================] - 0s 17ms/step - loss: 0.1004 - val_loss: 0.0975
```

You can see that Keras will keep you updated on the loss as the model trains.

Often, a better way to view the loss though is to plot it. The fit method in fact keeps a record of the loss produced during training in a History object. We'll convert the data to a Pandas dataframe, which makes the plotting easy.

In [6]:

```
import pandas as pd
# convert the training history to a dataframe
history_df = pd.DataFrame(history.history)
# use Pandas native plot method
history_df['loss'].plot();
```

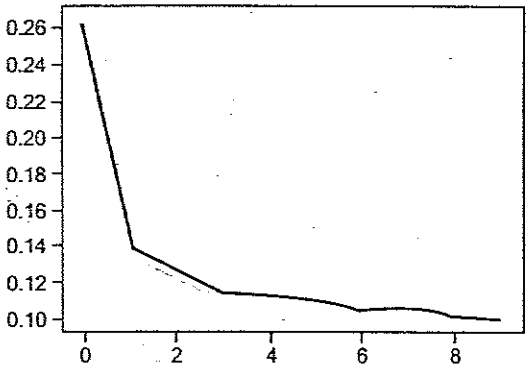


Fig. 5.42.

Notice how the loss levels off as the epochs go by. When the loss curve becomes horizontal like that, it means the model has learned all it can and there would be no reason continue for additional epochs.

5.7. ERROR BACKPROPAGATION

Need of Backpropagation

While designing a Neural Network, in the beginning, we initialize weights with some random values or any variable for that fact. So, it's not necessary that whatever weight values we have selected will be correct, or it fits our model the best. We have selected some weight values in the beginning, but our model output is way different than our actual output i.e. the error value is huge.

Basically, what we need to do, we need to somehow explain the model to change the parameters (weights), such that error becomes minimum.

Let's put it in an another way, we need to train our model.

Let me summarize the steps for you:

- **Calculate the error** – How far is your model output from the actual output.
- **Minimum Error** – Check whether the error is minimized or not.
- **Update the parameters** – If the error is huge then, update the parameters (weights and biases). After that again check the error. Repeat the process until the error becomes minimum.

- **Model is ready to make a prediction** – Once the error becomes minimum, you can feed some inputs to your model and it will produce the output.

One way to train our model is called as Backpropagation. Consider the diagram below:

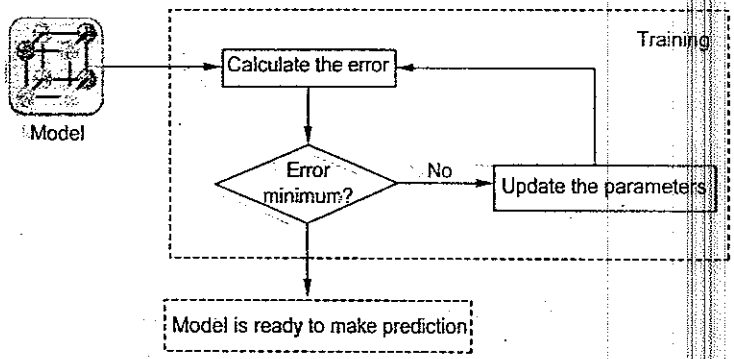


Fig. 5.43.

Backpropagation is one of the important concepts of a neural network. Our task is to classify our data best. For this, we have to update the weights of parameter and bias, but how can we do that in a deep neural network? In the linear regression model, we use gradient descent to optimize the parameter. Similarly here we also use gradient descent algorithm using Backpropagation.

For a single training example, Backpropagation algorithm calculates the gradient of the error function. Backpropagation can be written as a function of the neural network. Backpropagation algorithms are a set of methods used to efficiently train artificial neural networks following a gradient descent approach which exploits the chain rule.

The main features of Backpropagation are the iterative, recursive and efficient method through which it calculates the updated weight to improve the network until it is not able to perform the task for which it is being trained. Derivatives of the activation function to be known at network design time is required to Backpropagation.

The Backpropagation algorithm looks for the minimum value of the error function in weight space using a technique called the delta rule or gradient descent. The weights that minimize the error function is then considered to be a solution to the learning problem.

Let's understand how it works with an example:

You have a dataset, which has labels.

Consider the below table:

Input	Desired Output
0	0
1	2
2	4

Now the output of your model when 'W" value is 3:

Input	Desired Output	Model output (W = 3)
0	0	0
1	2	3
2	4	6

Notice the difference between the actual output and the desired output:

Input	Desired Output	Model output (W = 3)	Absolute Error	Square Error
0	0	0	0	0
1	2	3	1	1
2	4	6	2	4

Let's change the value of 'W'. Notice the error when 'W' = '4'

Input	Desired Output	Model output (W = 3)	Absolute Error	Square Error	Model output (W = 4)	Square Error
0	0	0	0	0	0	0
1	2	3	1	1	4	4
2	4	6	2	4	8	16

Now if you notice, when we increase the value of 'W' the error has increased. So, obviously there is no point in increasing the value of 'W' further. But, what happens if I decrease the value of 'W'? Consider the table below:

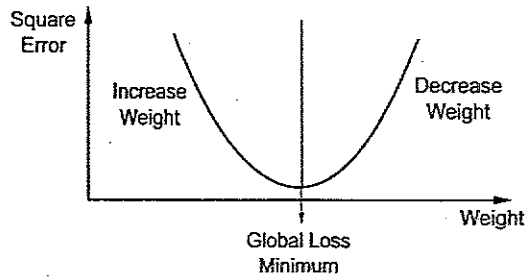
Input	Desired Output	Model output (W = 3)	Absolute Error	Square Error	Model output (W = 2)	Square Error
0	0	0	0	0	0	0
1	2	3	2	4	3	0
2	4	6	2	4	4	0

Now, what we did here:

- We first initialized some random value to 'W' and propagated forward.
- Then, we noticed that there is some error. To reduce that error, we propagated backwards and increased the value of 'W'.
- After that, also we noticed that the error has increased. We came to know that, we can't increase the 'W' value.
- So, we again propagated backwards and we decreased 'W' value.
- Now, we noticed that the error has reduced.

So, we are trying to get the value of weight such that the error becomes minimum. Basically, we need to figure out whether we need to increase or decrease the weight value. Once we know that, we keep on updating the weight value in that direction until error becomes minimum. You might reach a point, where if you further update the weight, the error will increase. At that time you need to stop, and that is your final weight value.

Consider the graph below:



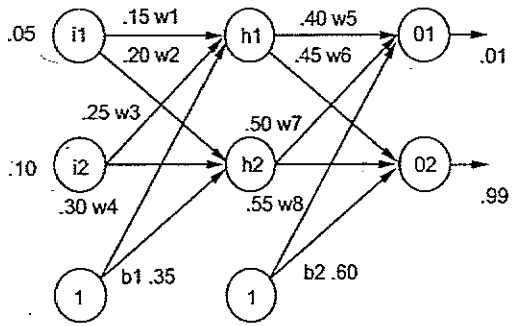
We need to reach the 'Global Loss Minimum'.

This is nothing but Backpropagation.

Let's now understand the math behind Backpropagation.

5.7.1. WORKING OF BACKPROPAGATION

Consider the below Neural Network:



The above network contains the following:

- two inputs
- two hidden neurons
- two output neurons
- two biases

Below are the steps involved in Backpropagation:

- **Step – 1:** Forward Propagation
- **Step – 2:** Backward Propagation
- **Step – 3:** Putting all the values together and calculating the updated weight value

Step – 1: Forward Propagation

We will start by propagating forward.

Net Input For h1:

net h1 = w1*i1 + w2*i2 + b1*1 → net h1 = 0.15*0.05 + 0.2*0.1 + 0.35*1 = 0.3775

Output Of h1:

out h1 = 1 / 1 + $e^{-net\ h1}$ → 1 / 1 + $e^{-.3775}$ = 0593269992

Output Of h2:

out h2 = 0.596884378

We will repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.

Output Of h1:

net o1 = w5* out h1 + w6*out h2 + b2*1 → 0.4*0.593269992 + 0.45*0.596884378 + 0.6*1 = 1.105905967

out 01 = 1 / 1 – $e^{-net\ o1}$ → 1 / 1 + $e^{-1.105905967}$ = 075136507

Output Of h2:

out o2 = 0.772928465

Now, let's see what is the value of the error:

Error For o1:

E o1 = ∑1/2 (target – output)^2 → 1/2(0.01 – 0.75136507)^2 = 0.274811083

Error For o2:

E o2 = 0.023560026

Total Error:

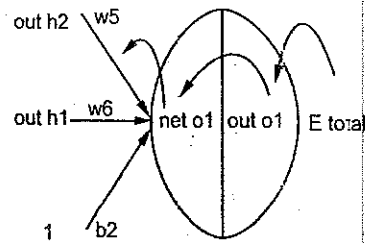
Etotal = E o1 + E o2 → 0.274811083 + 0.023560026 = 0.298371109

Step – 2: Backward Propagation

Now, we will propagate backwards. This way we will try to reduce the error by changing the values of weights and biases.

Consider W5, we will calculate the rate of change of error w.r.t change in weight W5.

$$\frac{\delta Etotal}{\delta w5} = \frac{\delta Etotal}{\delta out\ o1} * \frac{\delta out\ o1}{\delta net\ o1} * \frac{\delta net\ o1}{\delta w5}$$



Since we are propagating backwards, first thing we need to do is, calculate the change in total errors w.r.t the output O_1 and O_2.

$$E_{total} = 1/2(\text{target } 01 - \text{out } 01)^2 + 1/2(\text{target}02 - \text{out } 02)^2$$

$$\frac{\delta E\text{total}}{\delta \text{out } 01} = -(\text{target } 01 - \text{out } 01) = -(0.01 - 0.75136507) = 0.74136507$$

Now, we will propagate further backwards and calculate the change in output O_1 w.r.t to its total net input.

$$\text{out } 01 = 1/1 + e^{-netol}$$

$$\frac{\delta \text{out } 01}{\delta \text{net } 01} = \text{out } 01\ (1 - \text{out } 01) = 0.75136507(1 - 0.75136507) = 0.186815602$$

Let's see now how much does the total net input of O1 changes w.r.t W5?

$$\text{net } 01 = W5 * \text{out h1} + W6 * \text{out h2} + b2 * 1$$

$$\frac{\delta \text{net } 01}{\delta W5} = 1 * \text{out h1}\ W5^{(1-1)} + 0 + 0 = 0.593269992$$

Step – 3: Putting all the values together and calculating the updated weight value

Now, let's put all the values together:

$$\frac{\delta E\text{total}}{\delta w5} = \frac{\delta E\text{total}}{\delta \text{out } o1} * \frac{\delta \text{out } o1}{\delta \text{net } o1} * \frac{\delta \text{net } o1}{\delta w5} \rightarrow 0.082167041$$

Let's calculate the updated value of W5:

$$w5^{+} = w5 - n\frac{\delta E\text{total}}{\delta w5} \rightarrow w5^{+} = 0.4 - 0.5 * 0.082167041$$

Updated w5 → 0.35891648

Similarly, we can calculate the other weight values as well.

- After that we will again propagate forward and calculate the output. Again, we will calculate the error.
- If the error is minimum we will stop right there, else we will again propagate backwards and update the weight values.
- This process will keep on repeating until error becomes minimum.

Backpropagation Key Points

- Simplifies the network structure by elements weighted links that have the least effect on the trained network

- You need to study a group of input and activation values to develop the relationship between the input and hidden unit layers.
- It helps to assess the impact that a given input variable has on a network output. The knowledge gained from this analysis should be represented in rules.
- Backpropagation is especially useful for deep neural networks working on error-prone projects, such as image or speech recognition.
- Backpropagation takes advantage of the chain and power rules allows backpropagation to function with any number of outputs.

Disadvantages of using Backpropagation

- The actual performance of backpropagation on a specific problem is dependent on the input data.
- Back propagation algorithm in data mining can be quite sensitive to noisy data
- You need to use the matrix-based approach for backpropagation instead of mini-batch.

5.8. FROM SHALLOW NETWORKS TO DEEP NETWORKS

Shallow neural networks give us basic idea about deep neural network which consist of only 1 or 2 hidden layers. Understanding a shallow neural network gives us an understanding into what exactly is going on inside a deep neural network A neural network is built using various hidden layers. The figure 5.49 below shows a shallow neural network with 1 hidden layer, 1 input layer and 1 output layer.

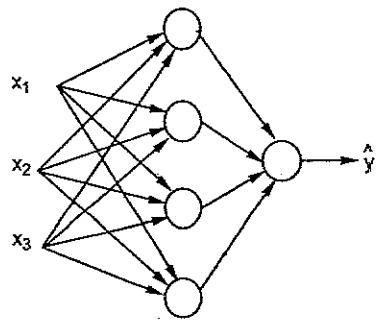


Fig. 5.44.

Now that we know the computations that occur in a particular layer, let us understand how the whole neural network computes the output for a given input X. These can also be called the forward-propagation equations.

$$Z^{[1]} = W^{[1]T} X + b^{[1]}$$

$$A^{[1]} = \sigma(Z^{[1]})$$

$$Z^{[2]} = W^{[2]T} A^{[1]} + b^{[2]}$$

$$\hat{y} = A^{[2]} = \sigma(Z^{[2]})$$

1. The first equation calculates the intermediate output **Z[1]**of the first hidden layer.
2. The second equation calculates the final output **A[1]**of the first hidden layer.
3. The third equation calculates the intermediate output **Z[2]**of the output layer.
4. The fourth equation calculates the final output **A[2]**of the output layer which is also the final output of the whole neural network.

5.8.1. DEEP NETS AND SHALLOW NETS

There is no clear threshold of depth that divides shallow learning from deep learning; but it is mostly agreed that for deep learning which has multiple non-linear layers, CAP must be greater than two.

Basic node in a neural net is a perception mimicking a neuron in a biological neural network. Then we have multi-layered Perception or MLP. Each set of inputs is modified by a set of weights and biases; each edge has a unique weight and each node has a unique bias.

The prediction accuracy of a neural net depends on its weights and biases.

The process of improving the accuracy of neural network is called training. The output from a forward prop net is compared to that value which is known to be correct.

The cost function or the loss function is the difference between the generated output and the actual output.

The point of training is to make the cost of training as small as possible across millions of training examples. To do this, the network tweaks the weights and biases until the prediction matches the correct output.

Once trained well, a neural net has the potential to make an accurate prediction every time.

When the pattern gets complex and you want your computer to recognise them, you have to go for neural networks. In such complex pattern scenarios, neural network out performs all other competing algorithms.

There are now GPUs that can train them faster than ever before. Deep neural networks are already revolutionizing the field of AI.

Computers have proved to be good at performing repetitive calculations and following detailed instructions but have been not so good at recognising complex patterns.

If there is the problem of recognition of simple patterns, a support vector machine (svm) or a logistic regression classifier can do the job well, but as the complexity of pattern increases, there is no way but to go for deep neural networks.

Therefore, for complex patterns like a human face, shallow neural networks fail and have no alternative but to go for deep neural networks with more layers. The deep nets are able to do their job by breaking down the complex patterns into simpler ones.

For example, human face; a deep net would use edges to detect parts like lips, nose, eyes, ears and so on and then re-combine these together to form a human face

The accuracy of correct prediction has become so accurate that recently at a Google Pattern Recognition Challenge, a deep net beat a human.

This idea of a web of layered perceptrons has been around for some time; in this area, deep nets mimic the human brain. But one downside to this is that they take long time to train, a hardware constraint

5.8.2. CHOOSING A DEEP NET

We have to decide if we are building a classifier or if we are trying to find patterns in the data and if we are going to use unsupervised learning. To extract patterns from a set of unlabelled data, we use a Restricted Boltzman machine or an Auto encoder.

Consider the following points while choosing a deep net –

- For text processing, sentiment analysis, parsing and name entity recognition, we use a recurrent net or recursive neural tensor network or RNTN;

- For any language model that operates at character level, we use the recurrent net.
- For image recognition, we use deep belief network DBN or convolutional network.
- For object recognition, we use a RNTN or a convolutional network.
- For speech recognition, we use recurrent net.

In general, deep belief networks and multilayer perceptrons with rectified linear units or RELU are both good choices for classification.

For time series analysis, it is always recommended to use recurrent net.

Neural nets have been around for more than 50 years; but only now they have risen into prominence. The reason is that they are hard to train; when we try to train them with a method called back propagation, we run into a problem called vanishing or exploding gradients. When that happens, training takes a longer time and accuracy takes a back-seat. When training a data set, we are constantly calculating the cost function, which is the difference between predicted output and the actual output from a set of labelled training data. The cost function is then minimized by adjusting the weights and biases values until the lowest value is obtained. The training process uses a gradient, which is the rate at which the cost will change with respect to change in weight or bias values.

5.8.3. RESTRICTED BOLTZMAN NETWORKS OR AUTOENCODERS - RBNS

In 2006, a breakthrough was achieved in tackling the issue of vanishing gradients. Geoff Hinton devised a novel strategy that led to the development of Restricted Boltzman Machine - RBM, a shallow two layer net.

The first layer is the visible layer and the second layer is the hidden layer. Each node in the visible layer is connected to every node in the hidden layer. The network is known as restricted as no two layers within the same layer are allowed to share a connection.

Autoencoders are networks that encode input data as vectors. They create a hidden, or compressed, representation of the raw data. The vectors are useful in dimensionality reduction; the vector compresses the raw data into smaller number of essential dimensions. Autoencoders are paired with decoders, which allows the reconstruction of input data based on its hidden representation.

RBM is the mathematical equivalent of a two-way translator. A forward pass takes inputs and translates them into a set of numbers that encodes the inputs. A backward pass meanwhile takes this set of numbers and translates them back into reconstructed inputs. A well-trained net performs back prop with a high degree of accuracy.

In either steps, the weights and the biases have a critical role; they help the RBM in decoding the interrelationships between the inputs and in deciding which inputs are essential in detecting patterns. Through forward and backward passes, the RBM is trained to re-construct the input with different weights and biases until the input and there-construction are as close as possible. An interesting aspect of RBM is that data need not be labelled. This turns out to be very important for real world data sets like photos, videos, voices and sensor data, all of which tend to be unlabelled. Instead of manually labelling data by humans, RBM automatically sorts through data; by properly adjusting the weights and biases, an RBM is able to extract important features and reconstruct the input. RBM is a part of family of feature extractor neural nets, which are designed to recognize inherent patterns in data. These are also called auto-encoders because they have to encode their own structure.

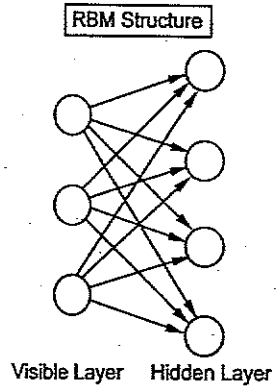


Fig. 5.45.

5.8.4. DEEP BELIEF NETWORKS - DBNS

Deep belief networks (DBNs) are formed by combining RBMs and introducing a clever training method. We have a new model that finally solves the problem of vanishing gradient. Geoff Hinton invented the RBMs and also Deep Belief Nets as alternative to back propagation.

A DBN is similar in structure to a MLP (Multi-layer perceptron), but very different when it comes to training. It is the training that enables DBNs to outperform their shallow counterparts

A DBN can be visualized as a stack of RBMs where the hidden layer of one RBM is the visible layer of the RBM above it. The first RBM is trained to reconstruct its input as accurately as possible.

The hidden layer of the first RBM is taken as the visible layer of the second RBM and the second RBM is trained using the outputs from the first RBM. This process is iterated till every layer in the network is trained.

In a DBN, each RBM learns the entire input. A DBN works globally by fine-tuning the entire input in succession as the model slowly improves like a camera lens slowly focussing a picture. A stack of RBMs outperforms a single RBM as a multi-layer perceptron MLP outperforms a single perceptron.

At this stage, the RBMs have detected inherent patterns in the data but without any names or label. To finish training of the DBN, we have to introduce labels to the patterns and fine tune the net with supervised learning.

We need a very small set of labelled samples so that the features and patterns can be associated with a name. This small-labelled set of data is used for training. This set of labelled data can be very small when compared to the original data set.

The weights and biases are altered slightly, resulting in a small change in the net's perception of the patterns and often a small increase in the total accuracy.

The training can also be completed in a reasonable amount of time by using GPUs giving very accurate results as compared to shallow nets and we see a solution to vanishing gradient problem too.

5.8.5. GENERATIVE ADVERSARIAL NETWORKS - GANS

Generative adversarial networks are deep neural nets comprising two nets, pitted one against the other, thus the "adversarial" name.

GANs were introduced in a paper published by researchers at the University of Montreal in 2014. Facebook's AI expert Yann LeCun, referring to GANs, called adversarial training "the most interesting idea in the last 10 years in ML."

GANs' potential is huge, as the network-scan learn to mimic any distribution of data. GANs can be taught to create parallel worlds strikingly similar to our own in

any domain: images, music, speech, prose. They are robot artists in a way, and their output is quite impressive.

In a GAN, one neural network, known as the generator, generates new data instances, while the other, the discriminator, evaluates them for authenticity.

Let us say we are trying to generate hand-written numerals like those found in the MNIST dataset, which is taken from the real world. The work of the discriminator, when shown an instance from the true MNIST dataset, is to recognize them as authentic.

Now consider the following steps of the GAN –

- The generator network takes input in the form of random numbers and returns an image.
- This generated image is given as input to the discriminator network along with a stream of images taken from the actual dataset.
- The discriminator takes in both real and fake images and returns probabilities, a number between 0 and 1, with 1 representing a prediction of authenticity and 0 representing fake.
- So you have a double feedback loop –
  - The discriminator is in a feedback loop with the ground truth of the images, which we know.
  - The generator is in a feedback loop with the discriminator.

5.8.6. RECURRENT NEURAL NETWORKS - RNNS

RNNS are neural networks in which data can flow in any direction. These networks are used for applications such as language modelling or Natural Language Processing (NLP).

The basic concept underlying RNNs is to utilize sequential information. In a normal neural network it is assumed that all inputs and outputs are independent of each other. If we want to predict the next word in a sentence we have to know which words came before it.

RNNs are called recurrent as they repeat the same task for every element of a sequence, with the output being based on the previous computations. RNNs thus can be said to have a "memory" that captures information about what has been

previously calculated. In theory, RNNs can use information in very long sequences, but in reality, they can look back only a few steps.

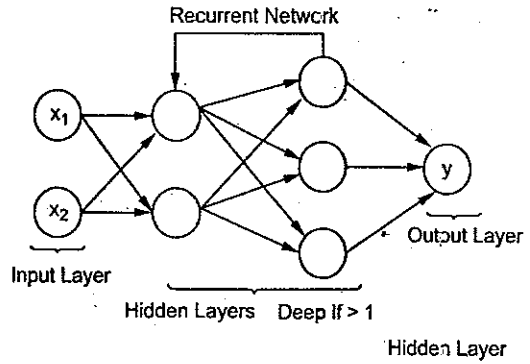


Fig. 5.46.

Long short-term memory networks (LSTMs) are most commonly used RNNs.

Together with convolutional Neural Networks, RNNs have been used as part of a model to generate descriptions for unlabelled images. It is quite amazing how well this seems to work.

5.8.7. CONVOLUTIONAL DEEP NEURAL NETWORKS - CNNS

If we increase the number of layers in a neural network to make it deeper, it increases the complexity of the network and allows us to model functions that are more complicated. However, the number of weights and biases will exponentially increase. As a matter of fact, learning such difficult problems can become impossible for normal neural networks. This leads to a solution, the convolutional neural networks.

CNNs are extensively used in computer vision; have been applied also in acoustic modelling for automatic speech recognition.

The idea behind convolutional neural networks is the idea of a "moving filter" which passes through the image. This moving filter, or convolution, applies to a certain neighbourhood of nodes which for example may be pixels, where the filter applied is 0.5 x the node value –

Noted researcher Yann LeCun pioneered convolutional neural networks. Facebook as facial recognition software uses these nets. CNN have been the go to solution for machine vision projects. There are many layers to a convolutional

network. In Imagenet challenge, a machine was able to beat a human at object recognition in 2015.

In a nutshell, Convolutional Neural Networks (CNNs) are multi-layer neural networks. The layers are sometimes up to 17 or more and assume the input data to be images.

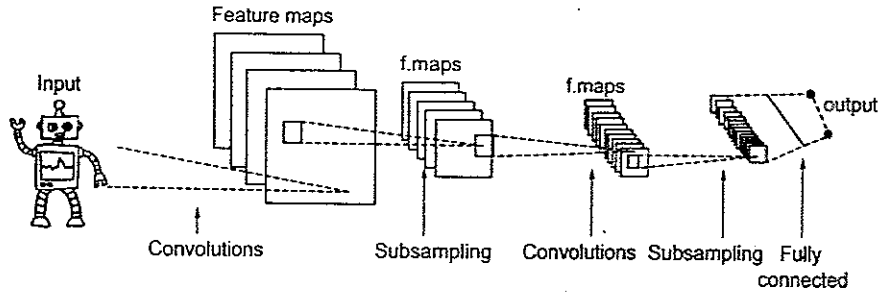


Fig. 5.47.

CNNs drastically reduce the number of parameters that need to be tuned. So, CNNs efficiently handle the high dimensionality of raw images.

5.9. UNIT SATURATION (AKA THE VANISHING GRADIENT PROBLEM)

The sigmoid function is one of the most popular activations functions used for developing deep neural networks. The use of sigmoid function restricted the training of deep neural networks because it caused the vanishing gradient problem. This caused the neural network to learn at a slower pace or in some cases no learning at all.

5.9.1. SIGMOID FUNCTION

Sigmoid functions are used frequently in neural networks to activate neurons. It is a logarithmic function with a characteristic S shape. The output value of the function is between 0 and 1. The sigmoid function is used for activating the output layers in binary classification problems. It is calculated as follows:

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

On the graph below you can see a comparison between the sigmoid function itself and its derivative. First derivatives of sigmoid functions are bell curves with values ranging from 0 to 0.25.

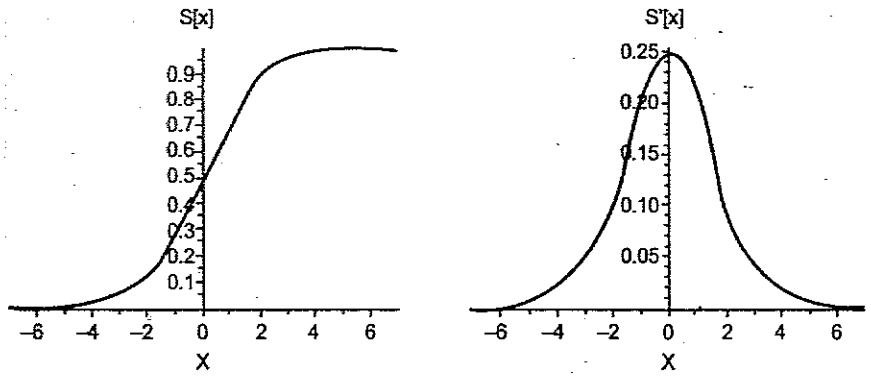


Fig. 5.48.

5.9.2. FORWARD PROPAGATION

The basic structure of a neural network is an input layer, one or more hidden layers, and a single output layer.

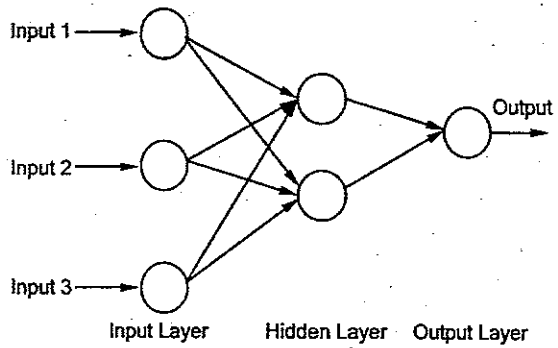


Fig. 5.49.

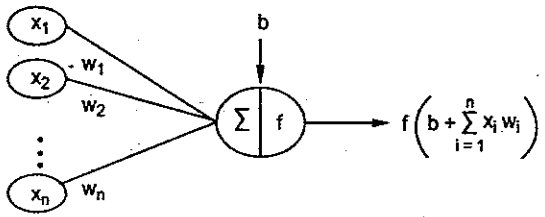


Fig. 5.50.

The weights of the network are randomly initialized during forward propagation. The input features are multiplied by the corresponding weights at each node of the hidden layer, and a bias is added to the net sum at each node. This value is then

transformed into the output of the node using an activation function. To generate the output of the neural network, the hidden layer output is multiplied by the weights plus bias values, and the total is transformed using another activation function. This will be the predicted value of the neural network for a given input value.

5.9.3. BACK PROPAGATION

As the network generates an output, the loss function(C) indicates how well it predicted the output. The network performs back propagation to minimize the loss. A back propagation method minimizes the loss function by adjusting the weights and biases of the neural network. In this method, the gradient of the loss function is calculated with respect to each weight in the network.

In back propagation, the new weight(w_{new}) of a node is calculated using the old weight(w_{old}) and product of the learning rate(η) and gradient of the loss function $\left(\frac{\partial C}{\partial w}\right)$.

$$w_{new} = w_{old} - \eta \times \frac{\partial C}{\partial w}$$

With the chain rule of partial derivatives, we can represent gradient of the loss function as a product of gradients of all the activation functions of the nodes with respect to their weights. Therefore, the updated weights of nodes in the network depend on the gradients of the activation functions of each node.

For the nodes with sigmoid activation functions, we know that the partial derivative of the sigmoid function reaches a maximum value of 0.25. When there are more layers in the network, the value of the product of derivative decreases until at some point the partial derivative of the loss function approaches a value close to zero, and the partial derivative vanishes. We call this the vanishing gradient problem.

With shallow networks, sigmoid function can be used as the small value of gradient does not become an issue. When it comes to deep networks, the vanishing gradient could have a significant impact on performance. The weights of the network remain unchanged as the derivative vanishes. During back propagation, a neural network learns by updating its weights and biases to reduce the loss function. In a network with vanishing gradient, the weights cannot be updated, so the network cannot learn. The performance of the network will decrease as a result.

5.9.4. METHOD TO OVERCOME THE PROBLEM

The vanishing gradient problem is caused by the derivative of the activation function used to create the neural network. The simplest solution to the problem is to replace the activation function of the network. Instead of sigmoid, use an activation function such as ReLU.

Rectified Linear Units (ReLU) are activation functions that generate a positive linear output when they are applied to positive input values. If the input is negative, the function will return zero.

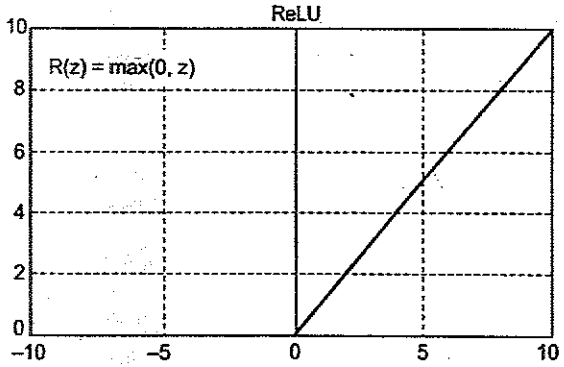


Fig. 5.51.

The derivative of a ReLU function is defined as 1 for inputs that are greater than zero and 0 for inputs that are negative. The graph shared below indicates the derivative of a ReLU function

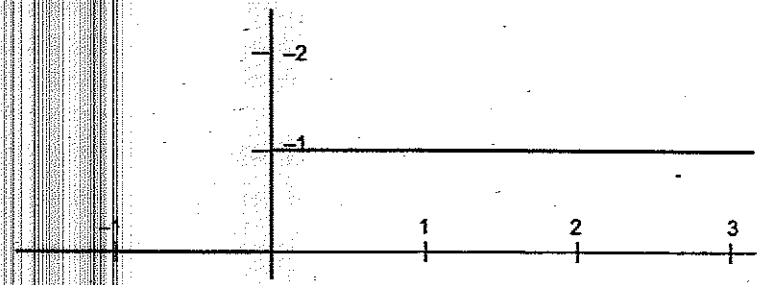


Fig. 5.52.

If the ReLU function is used for activation in a neural network in place of a sigmoid function, the value of the partial derivative of the loss function will be having values of 0 or 1 which prevents the gradient from vanishing. The use of ReLU function thus prevents the gradient from vanishing. The problem with the use

of ReLU is when the gradient has a value of 0. In such cases, the node is considered as a dead node since the old and new values of the weights remain the same. This situation can be avoided by the use of a leaky ReLU function which prevents the gradient from falling to the zero value.

Another technique to avoid the vanishing gradient problem is weight initialization. This is the process of assigning initial values to the weights in the neural network so that during back propagation, the weights never vanish.

In conclusion, the vanishing gradient problem arises from the nature of the partial derivative of the activation function used to create the neural network. The problem can be worse in deep neural networks using Sigmoid activation function. It can be significantly reduced by using activation functions like ReLU and leaky ReLU.

5.10. RELU

Activation functions give out the final value given out from a neuron, but what is activation function and why do we need it?

So, an activation function is basically just a simple function that transforms its inputs into outputs that have a certain range. There are various types of activation functions that perform this task in a different manner, For example, the sigmoid activation function takes input and maps the resulting values in between 0 to 1.

One of the reasons that this function is added into an artificial neural network in order to help the network learn complex patterns in the data. These functions introduce nonlinear real-world properties to artificial neural networks. Basically, in a simple neural network, x is defined as inputs, w weights, and we pass $f(x)$ that is the value passed to the output of the network. This will then be the final output or the input of another layer.

If the activation function is not applied, the output signal becomes a simple linear function. A neural network without activation function will act as a linear regression with limited learning power. But we also want our neural network to learn non-linear states as we give it complex real-world information such as image, video, text, and sound.

5.10.1. RELU ACTIVATION FUNCTION

ReLU stands for rectified linear activation unit and is considered one of the few milestones in the deep learning revolution. It is simple yet really better than its

predecessor activation functions such as sigmoid or tanh. The diagram below with the blue line is the representation of the Rectified Linear Unit (ReLU), whereas the green line is a variant of ReLU called Softplus. The other variants of ReLU include leaky ReLU, exponential linear unit (ELU) and Sigmoid linear unit (SiLU), etc., which are used to improve performances in some tasks.

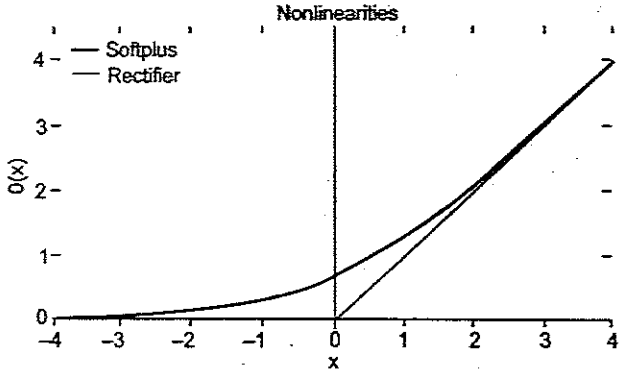


Fig. 5.53. Example of a ReLU activation function and Softplus

5.10.2. RELU ACTIVATION FUNCTION FORMULA

Now how does ReLU transform its input? It uses this simple formula:

$$f(x) = \max(0, x)$$

ReLU function is its derivative both are monotonic. The function returns 0 if it receives any negative input, but for any positive value x, it returns that value back. Thus it gives an output that has a range from 0 to infinity.

Now let us give some inputs to the ReLU activation function and see how it transforms them and then we will plot them also.

First, let us define a ReLU function

```
def ReLU(x):
  if x>0:
    return x
  else:
    return 0
```

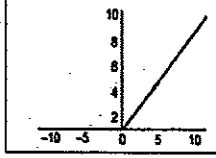
Next, we store numbers from -19 to 19 in a list called input_series and next we apply ReLU to all these numbers and plot them


```
from matplotlib import pyplot
pyplot.style.use('ggplot')
pyplot.figure(figsize=(10,5))
# define a series of inputs
input_series = [x for x in range(-19, 19)]
# calculate outputs for our inputs
output_series = [ReLU(x) for x in input_series]
# line plot of raw inputs to rectified outputs
pyplot.plot(input_series, output_series)
pyplot.show()
```

ReLU is used as a default activation function and nowadays and it is the most commonly used activation function in neural networks, especially in CNNs.

5.10.3. WHY IS RELU A GOOD ACTIVATION FUNCTION?

The main reason ReLU wasn't used until more recently is because it was not differentiable at the point zero. Researchers tended to use differentiable functions like sigmoid and tanh. However, it's now determined that ReLU is the best activation function for deep learning.



$$f(x) = \begin{cases} 0 & \text{for } x \le 0 \\ x & \text{for } x > 0 \end{cases} = f(x) = \begin{cases} 0 & \text{for } x \le 0 \\ x & \text{for } x > 0 \end{cases} \quad (0, 00)$$

$$\max(0, z) = x1_{x>0}$$

Equation for the ReLU activation function.

The ReLU activation function is differentiable at all points except at zero. For values greater than zero, we just consider the max of the function. This can be written as:

$$f(x) = \max(0, z)$$

In simple terms, this can also be written as follows:

```
if input > 0:
  return input
else:
  return 0
```

All the negative values default to zero, and the maximum for the positive number is taken into consideration.

For the computation of the backpropagation of neural networks, the differentiation for the ReLU is relatively easy. The only assumption we will make is the derivative at the point zero, which will also be considered as zero. This is usually not such a big concern, and it works well for the most part. The derivative of the function is the value of the slope. The slope for negative values is 0.0, and the slope for positive values is 1.0.

5.10.4. ADVANTAGES OF THE RELU ACTIVATION FUNCTION

1. **Convolutional layers and deep learning:** It is the most popular activation function for training convolutional layers and deep learning models.
2. **Computational simplicity:** The rectifier function is trivial to implement, requiring only a max() function.
3. **Representational sparsity:** An important benefit of the rectifier function is that it is capable of outputting a true zero value.
4. **Linear behavior:** A neural network is easier to optimize when its behavior is linear or close to linear.

5.10.5. DISADVANTAGES OF THE RELU ACTIVATION FUNCTION

The main issue with ReLU is that all the negative values become zero immediately, which decreases the ability of the model to fit or train from the data properly.

That means any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in turn affects the resulting graph by not mapping the negative values appropriately. This can however be easily fixed by using the different variants of the ReLU activation function, like the leaky ReLU and other functions.

5.11. HYPERPARAMETER TUNING

A learning algorithm learns or estimates model parameters for the given data set, then continues updating these values as it continues to learn. After learning is complete, these parameters become part of the model. For example, each weight and bias in a neural network is a parameter.

Hyperparameters, on the other hand, are specific to the algorithm itself, so we can't calculate their values from the data. We use hyperparameters to calculate the model parameters. Different hyperparameter values produce different model parameter values for a given data set.

Hyperparameter tuning consists of finding a set of optimal hyperparameter values for a learning algorithm while applying this optimized algorithm to any data set. That combination of hyperparameters maximizes the model's performance, minimizing a predefined loss function to produce better results with fewer errors. Note that the learning algorithm optimizes the loss based on the input data and tries to find an optimal solution within the given setting. However, hyperparameters describe this setting exactly.

For instance, if we work on natural language processing (NLP) models, we probably use neural networks, support-vector machines (SVMs), Bayesian networks, and Extreme Gradient Boosting (XGB) for tuning parameters.

5.11.1. STEPS TO PERFORM HYPERPARAMETER TUNING

- Select the right type of model.
- Review the list of parameters of the model and build the HP space
- Finding the methods for searching the hyperparameter space
- Applying the cross-validation scheme approach
- Assess the model score to evaluate the model

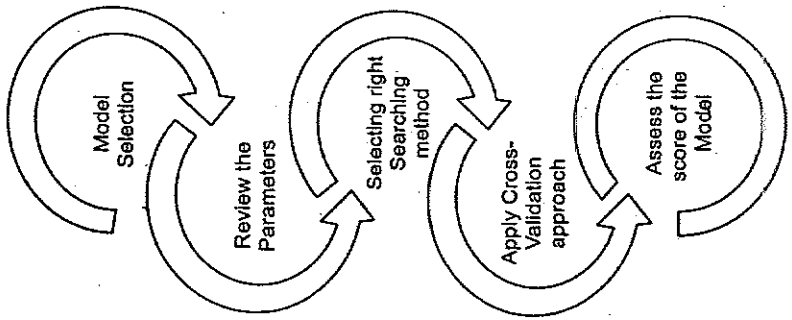


Fig. 5.54.

5.11.2. HYPERPARAMETER TYPES

Some important hyperparameters that require tuning in neural networks are:

- **Number of hidden layers:** It's a trade-off between keeping our neural network as simple as possible (fast and generalized) and classifying our

input data correctly. We can start with values of four to six and check our data's prediction accuracy when we increase or decrease this hyperparameter.

- **Number of nodes/neurons per layer**: More isn't always better when determining how many neurons to use per layer. Increasing neuron count can help, up to a point. But layers that are too wide may memorize the training dataset, causing the network to be less accurate on new data.
- **Learning rate**: Model parameters are adjusted iteratively — and the learning rate controls the size of the adjustment at each step. The lower the learning rate, the lower the changes to parameter estimates are. This means that it takes a longer time (and more data) to fit the model — but it also means that it is more likely that we actually find the minimum loss.
- **Momentum**: Momentum helps us avoid falling into local minima by resisting rapid changes to parameter values. It encourages parameters to keep changing in the direction they were *already* changing, which helps prevent zig-zagging on every iteration. Aim to start with low momentum values and adjust upward as needed.

We consider these essential hyperparameters for tuning SVMs:

- **C**: A trade-off between a smooth decision boundary (more generic) and a neat decision boundary (more accurate for the training data). A low value may cause the model to incorrectly classify some training data, while a high value may cause the model to incur overfitting. Overfitting creates an analysis too specific for the current data set and possibly unfit for future data and unreliable for future observations.
- **Gamma**: The inverse of the influence radius of data samples we selected as support vectors. High values indicate the small radius of influence and small decision boundaries that do not consider relatively close data samples. These high values cause overfitting. Low values indicate the significant effect of distant data samples, so the model can't capture the correct decision boundaries from the data set.

Important hyperparameters that need tuning for XGBoost are:

- **max_depth** and **min_child_weight**: This controls the tree architecture. max_depth defines the maximum number of nodes from the root to the farthest leaf (the default number is 6). min_child_weight is the minimum weight required to create a new node in the tree.
- **learning_rate**: This determines the amount of correction at each step, given that each boosting round corrects the previous round's errors. learning_rate takes values from 0 to 1, and the default value is 0.3.
- **n_estimators**: This defines the number of trees in the ensemble. The default value is 100. Note that if we were using vanilla XGBoost instead of scikit-learn, we'd use num_boost_rounds instead of n_estimators.
- **colsample_bytree** and **subsample**: This controls the data set samples that each round uses. These hyperparameters are helpful to avoid overfitting. subsample is the fraction of samples used, with a value from 0 to 1 and a default value of 1. colsample_bytree defines the fraction of columns (features) and takes numbers from 0 to 1, with a default value of 1.

In these examples, hyperparameters tending toward one extreme or the other can negatively affect our model's ability to make predictions. The trick is to find just the right value for each hyperparameter, so our model performs well and produces the best possible results.

5.11.3. DATA LEAKAGE

Data Leakage is when the model somehow knows the patterns in the test data during its training phase. In other words, the data that you are using to train your ML algorithm happens to have the information you are trying to predict.

Data leakage prevents the model to generalize well. It's very difficult for a data scientist to identify data leakage. Some of the reasons for data leakage are

- Outlier and missing value treatment with central values before splitting
- Scaling the data before splitting into training and testing
- train your model with both train and test data.

Hyper-Parameter Tuning is the process of finding the best set of hyper-parameters of the ML algorithm that delivers best performance.

5.11.4. METHODS FOR TUNING HYPERPARAMETERS

Now that we understand what hyperparameters are and the importance of tuning them, we need to know how to choose their optimal values. We can find these optimal hyperparameter values using manual or automated methods.

When tuning hyperparameters manually, we typically start using the default recommended values or rules of thumb, then search through a range of values using trial-and-error. But manual tuning is a tedious and time-consuming approach. It isn't practical when there are many hyperparameters with a wide range.

Automated hyperparameter tuning methods use an algorithm to search for the optimal values. Some of today's most popular automated methods are grid search, random search, and Bayesian optimization. Let's explore these methods in detail.

Grid Search

Grid search is a sort of "brute force" hyperparameter tuning method. We create a grid of possible discrete hyperparameter values then fit the model with every possible combination. We record the model performance for each set then select the combination that has produced the best performance.

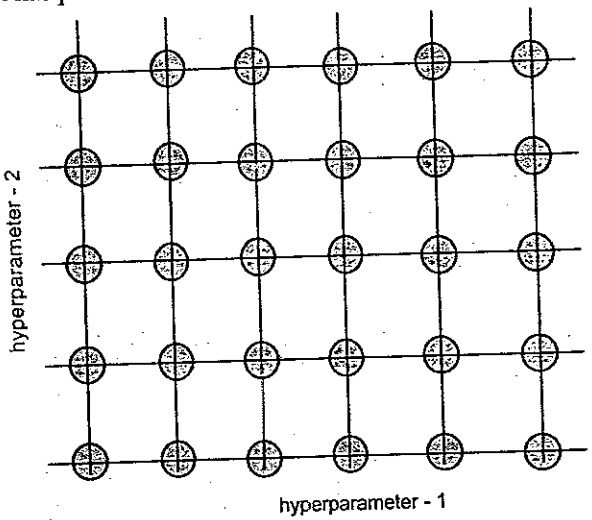


Fig. 5.55.

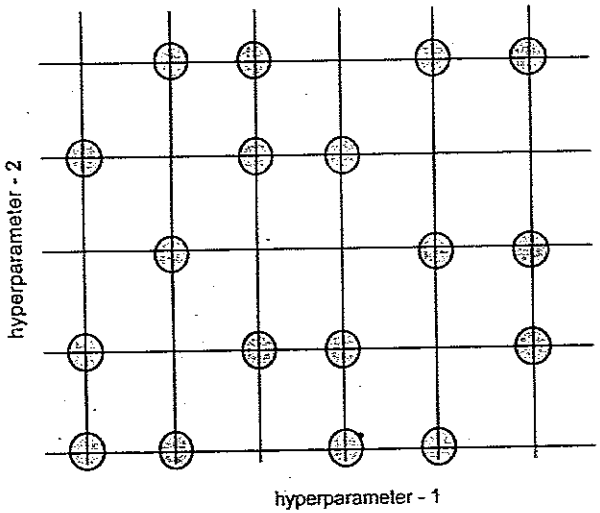
Grid search is a hyperparameter tuning method in which we create a grid of possible discrete hyperparameter values, then fit the model with every possible combination.

Grid search is an exhaustive algorithm that can find the best combination of hyperparameters. However, the drawback is that it's slow. Fitting the model with every possible combination usually requires a high computation capacity and significant time, which may not be available.

Random Search

The random search method (as its name implies) chooses values randomly rather than using a predefined set of values like the grid search method.

Random search tries a random combination of hyperparameters in each iteration and records the model performance. After several iterations, it returns the mix that produced the best result.



Random search tries a random combination of hyperparameters in each iteration and records the model performance. After several iterations, it returns the mix that produced the best result.

Random search is appropriate when we have several hyperparameters with relatively large search domains. We can make discrete ranges (for instance, [5-100] in steps of 5) and still get a reasonably good set of combinations.

The benefit is that random search typically requires less time than grid search to return a comparable result. It also ensures we don't end up with a model that's biased toward value sets arbitrarily chosen by users. Its drawback is that the result may not be the best possible hyperparameter combination.

Bayesian Optimization

Grid search and random search are relatively inefficient because they often evaluate many unsuitable hyperparameter combinations. They don't take into account the previous iterations' results when choosing the next hyperparameters.

The Bayesian optimization method takes a different approach. This method treats the search for the optimal hyperparameters as an optimization problem. When choosing the next hyperparameter combination, this method considers the previous evaluation results. It then applies a probabilistic function to select the combination that will probably yield the best results. This method discovers a fairly good hyperparameter combination in relatively few iterations.

Data scientists choose a probabilistic model when the objective function is unknown. That is, there is no analytical expression to maximize or minimize. The data scientists apply the learning algorithm to a data set, use the algorithm's results to define the objective function, and take the various hyperparameter combinations as the input domain.

The probabilistic model is based on past evaluation results. It estimates the probability of a hyperparameter combination's objective function result:

P(result | hyperparameters)

This probabilistic model is a "surrogate" of the objective function. The objective function can be, for instance, the root-mean-square error (RMSE). We calculate the objective function using the training data with the hyperparameter combination. We try to optimize it (maximize or minimize, depending on the objective function selected).

Applying the probabilistic model to the hyperparameters is computationally inexpensive compared to the objective function, so this method typically updates and improves the surrogate probability model every time the objective function runs. Better hyperparameter predictions decrease the number of objective function evaluations we need to achieve a good result.

5.12. BATCH NORMALIZATION

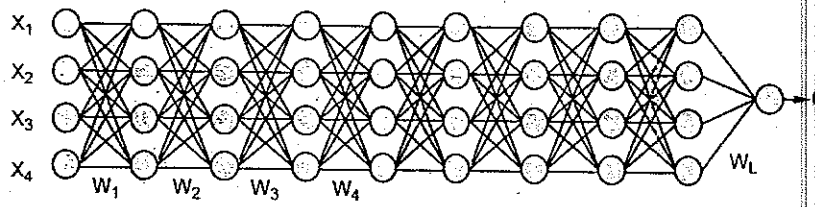
Normalization is a data pre-processing tool used to bring the numerical data to a common scale without distorting its shape.

Generally, when we input the data to a machine or deep learning algorithm we tend to change the values to a balanced scale. The reason we normalize is partly to ensure that our model can generalize appropriately.

Now coming back to Batch normalization, it is a process to make neural networks faster and more stable through adding extra layers in a deep neural network. The new layer performs the standardizing and normalizing operations on the input of a layer coming from a previous layer.

But what is the reason behind the term "Batch" in batch normalization? A typical neural network is trained using a collected set of input data called batch. Similarly, the normalizing process in batch normalization takes place in batches, not as a single input.

Let's understand this through an example, we have a deep neural network as shown in the following image.



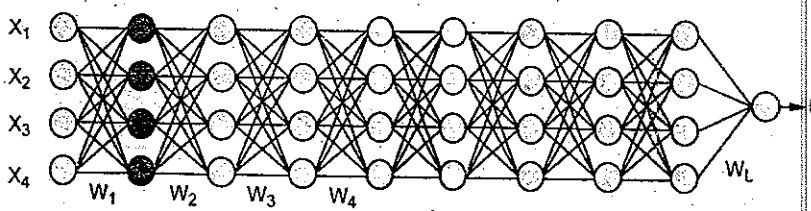
L = Number of layers

Bias = 0

Activation Function : Sigmoid

Fig. 5.56.

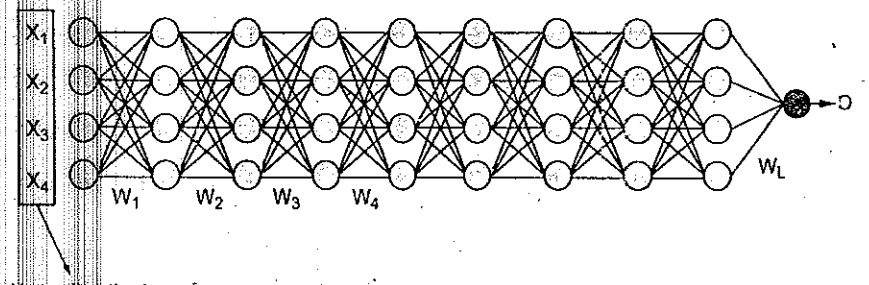
Initially, our inputs X_1, X_2, X_3, X_4 are in normalized form as they are coming from the pre-processing stage. When the input passes through the first layer, it transforms, as a sigmoid function applied over the dot product of input X and the weight matrix W.



$$h_1 = \sigma(W_1 X)$$

Fig. 5.57.

Similarly, this transformation will take place for the second layer and go till the last layer L as shown in the following image.



$$h_1 = \sigma(W_1 X)$$
$$h_2 = \sigma(W_2 h_1) = \sigma(W_2 \sigma(W_1 X))$$
$$O = \sigma(W_L h_{L-1})$$

Fig. 5.58.

Although, our input X was normalized with time the cutput will no longer be on the same scale. As the data go through multiple layers of the neural network and L activation functions are applied, it leads to an internal co-variate shift in the data.

Batch normalization is computed differently during the training and the testing phase.

5.12.1. TRAINING

At each hidden layer, Batch Normalization transforms the signal as follow :

1. $\mu = \frac{1}{n}\sum_i Z^{(i)}$
2. $\sigma^2 = \frac{1}{n}\sum_i (Z^{(i)} - \mu)^2$
3. $Z^{(i)}_{norm} = \frac{Z^{(i)} - \mu}{\sqrt{\sigma^2 - \epsilon}}$
4. $\hat{Z} = \gamma * Z^{(i)}_{norm} + \beta$

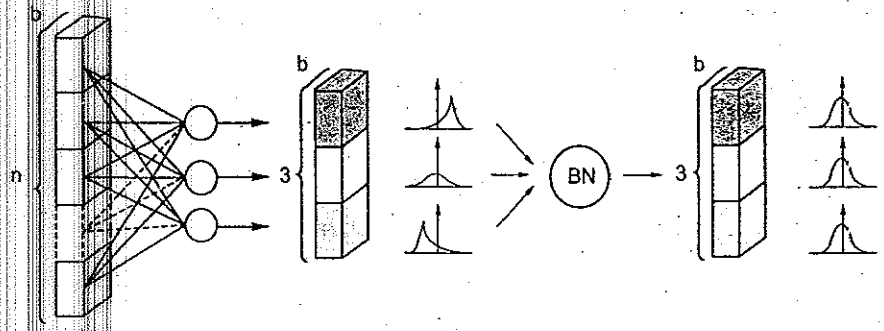


Fig. 5.59.

The BN layer first determines the **mean** μ and **the variance** σ^2 of the activation values across the batch, using (1) and (2).

It then normalizes the activation vector Z^(*i*) with (3). That way, each neuron's output follows a standard normal distribution across the batch. (ϵ is a constant used for numerical stability)

Batch Normalization first step. Example of a 3-neurons hidden layer, with a batch of size *b*. Each neuron follows a standard normal distribution.

it finally calculates the **layer's output** $\hat{Z}(i)$ by applying a linear transformation with γ and β, two trainable parameters (4). Such step allows the model to choose the optimum distribution for each hidden layers, by adjusting those two parameters :

- γ allows to adjust the standard deviation ;
- β allows to adjust the bias, shifting the curve on the right or on the left side.

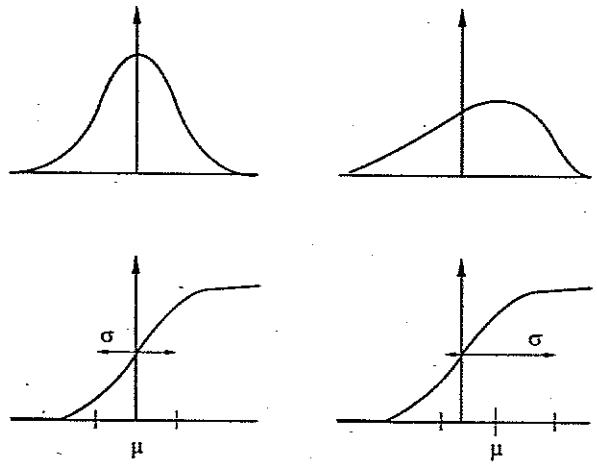


Fig. 5.60.

Benefits of γ *and* β arameters. Modifying the distribution (on the top) allows us to use different regimes of the nonlinear functions (on the bottom).

5.12.2. WORKING OF BATCH NORMALIZATION

Since by now we have a clear idea of why we need Batch normalization, let's understand how it works. It is a two-step process. First, the input is normalized, and later rescaling and offsetting is performed.

Normalization of the Input

Normalization is the process of transforming the data to have a mean zero and standard deviation one. In this step we have our batch input from layer h, first, we need to calculate the mean of this hidden activation.

$$\mu = \frac{1}{m}\sum h_i$$

Here, m is the number of neurons at layer h.

Once we have meant at our end, the next step is to calculate the standard deviation of the hidden activations.

$$\sigma = \left[\frac{1}{m}\sum (h_i - \mu)^2\right]^{1/2}$$

Further, as we have the mean and the standard deviation ready. We will normalize the hidden activations using these values. For this, we will subtract the mean from each input and divide the whole value with the sum of standard deviation and the smoothing term (ε).

The smoothing term(ε) assures numerical stability within the operation by stopping a division by a zero value.

$$h_{i(norm)} = \frac{(h_i - \mu)}{\sigma + \varepsilon}$$

Rescaling of Offsetting

In the final operation, the re-scaling and offsetting of the input take place. Here two components of the BN algorithm come into the picture, γ (gamma) and β (beta). These parameters are used for re-scaling (γ) and shifting(β) of the vector containing values from the previous operations.

$$h_i = y\, h_{i(norm)} + \beta$$

These two are learnable parameters, during the training neural network ensures the optimal values of γ and β are used. That will enable the accurate normalization of each batch.

5.12.3. ADVANTAGES OF BATCH NORMALIZATION

Now let's look into the advantages the BN process offers.

Speed Up the Training

By Normalizing the hidden layer activation the Batch normalization speeds up the training process.

Handles internal covariate shift

Smoothens the Loss Function

Batch normalization smoothens the loss function that in turn by optimizing the model parameters improves the training speed of the model.

5.13. REGULARIZATION

Overfitting is a phenomenon that occurs when a Machine Learning model is constraint to training set and not able to perform well on unseen data.

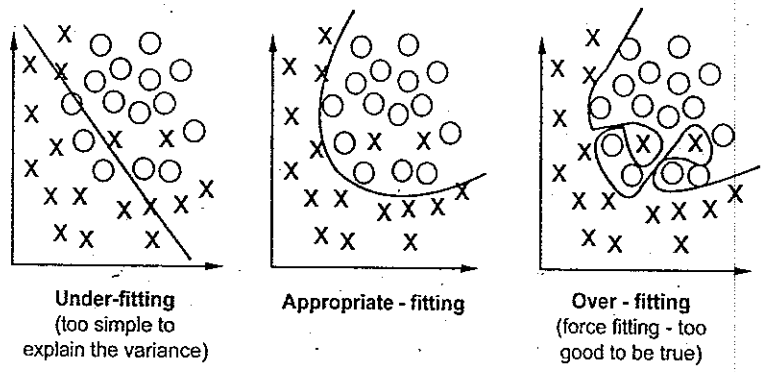


Fig. 5.61.

Regularization refers to techniques that are used to calibrate machine learning models in order to minimize the adjusted loss function and prevent overfitting or underfitting.

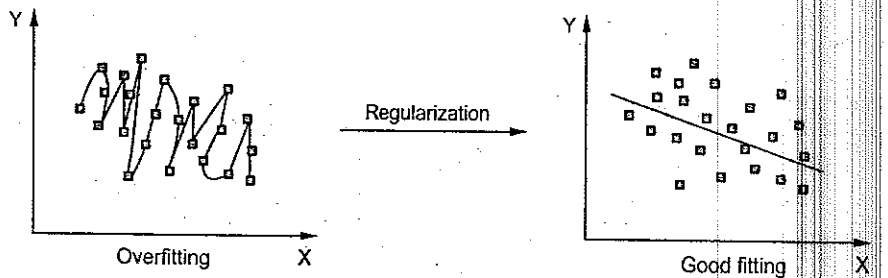


Fig. 5.62. Regularization on an over-fitted model

Using Regularization, we can fit our machine learning model appropriately on a given test set and hence reduce the errors in it.

This technique can be used in such a way that it will allow to maintain all variables or features in the model by reducing the magnitude of the variables. Hence, it maintains accuracy as well as a generalization of the model.

It mainly regularizes or reduces the coefficient of features toward zero. In simple words, "In regularization technique, we reduce the magnitude of the features by keeping the same number of features."

5.13.1. REGULARIZATION TECHNIQUES

The commonly used regularization techniques are :

1. L1 regularization
2. L2 regularization
3. Dropout regularization

A regression model which uses L_1 Regularization technique is called LASSO (Least Absolute Shrinkage and Selection Operator) regression.

A regression model that uses L_2 regularization technique is called Ridge regression.

Lasso Regression adds "absolute value of magnitude" of coefficient as penalty term to the loss function(L).

5.13.2. WORKING OF REGULARIZATION

Regularization works by adding a penalty or complexity term to the complex model. Let's consider the simple linear regression equation:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \cdots + \beta_n x_n + b$$

In the above equation, Y represents the value to be predicted

$X_1, X_2, \ldots X_n$ are the features for Y.

$\beta_0, \beta_1, \ldots \beta_n$ are the weights or magnitude attached to the features, respectively. Here represents the bias of the model, and b represents the intercept.

Linear regression models try to optimize the β_0 and b to minimize the cost function. The equation for the cost function for the linear model is given below:

$$\sum_{i=0}^{M} (y_i - y_i')^2 = \sum_{i=0}^{M} \left(y_i - \sum_{j=0}^{n} \beta_j * X_{ij} \right)^2$$

Now, we will add a loss function and optimize parameter to make the model that can predict the accurate value of Y. The loss function for the linear regression is called as **RSS or Residual sum of squares.**

5.13.3. RIDGE REGRESSION

- Ridge regression is one of the types of linear regression in which a small amount of bias is introduced so that we can get better long-term predictions.
- Ridge regression is a regularization technique, which is used to reduce the complexity of the model. It is also called as L_2 regularization.
- In this technique, the cost function is altered by adding the penalty term to it. The amount of bias added to the model is called Ridge Regression penalty. We can calculate it by multiplying with the lambda to the squared weight of each individual feature.
- The equation for the cost function in ridge regression will be:

$$\sum_{i=1}^{M} (y_i - y_i')^2 = \sum_{i=1}^{M} \left(y_i - \sum_{j=0}^{n} \beta_j * X_{ij} \right)^2 + \lambda \sum_{j=0}^{n} \beta_j^2$$

- In the above equation, the penalty term regularizes the coefficients of the model, and hence ridge regression reduces the amplitudes of the coefficients that decreases the complexity of the model.

Cost function = **Loss** + $\lambda \times \sum \|w\|^2$

Here,

Loss = Sum of the squared residuals

λ = Penalty for the errors

w = Slope of the curve / line

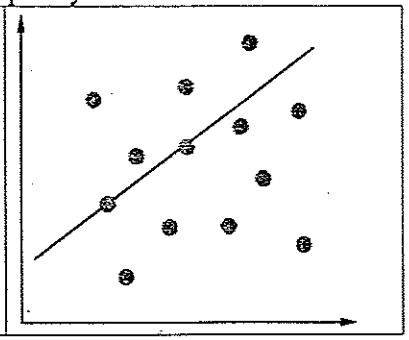


Fig. 5.63. Cost Function of Ridge Regression

- As we can see from the above equation, if the values of λ tend to zero, the equation becomes the cost function of the linear regression model. Hence, for the minimum value of λ, the model will resemble the linear regression model.

- A general linear or polynomial regression will fail if there is high collinearity between the independent variables, so to solve such problems, Ridge regression can be used.
- It helps to solve the problems if we have more parameters than samples.

In the cost function, the penalty term is represented by Lambda λ. By changing the values of the penalty function, we are controlling the penalty term. The higher the penalty, it reduces the magnitude of coefficients. It shrinks the parameters. Therefore, it is used to prevent multicollinearity, and it reduces the model complexity by coefficient shrinkage.

Consider the graph illustrated below which represents Linear regression :

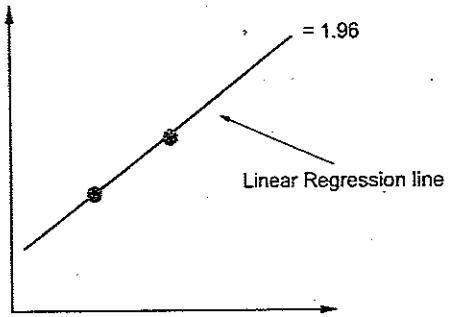


Fig. 5.64. Linear regression model

Cost function = Loss + λ × ∑|w|^2

For Linear Regression line, let's consider two points that are on the line,

Loss = 0 (considering the two points on the line)

λ = 1

w = 1.4

Then, Cost function = 0 + 1 × 1.42 = 1.96

For Ridge Regression, let's assume,

Loss = 0.32 + 0.22 = 0.13

λ = 1

w = 0.7

Then, Cost function = 0.13 + 1 × 0.72

= 0.62

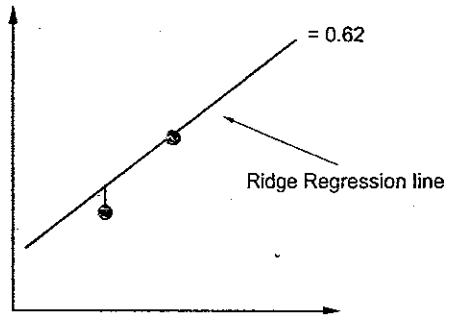


Fig. 5.65. Ridge regression model

Comparing the two models, with all data points, we can see that the Ridge regression line fits the model more accurately than the linear regression line.

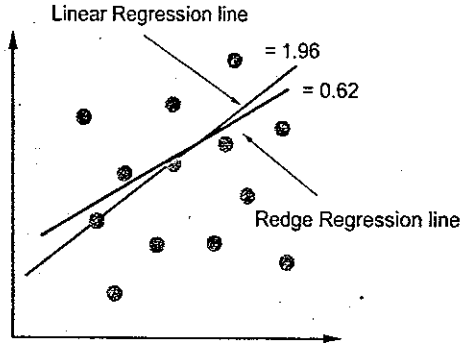


Fig. 5.66. Optimization of model fit using Ridge Regression

5.13.4. LASSO REGRESSION:

- Lasso regression is another regularization technique to reduce the complexity of the model. It stands for Least Absolute and Selection Operator.
- It is similar to the Ridge Regression except that the penalty term contains only the absolute weights instead of a square of weights.
- Since it takes absolute values, hence, it can shrink the slope to 0, whereas Ridge Regression can only shrink it near to 0.
- It is also called as **L_1 regularization.** The equation for the cost function of Lasso regression will be:

$$\sum_{i=1}^{M} (y_i - y_i')^2 = \sum_{i=1}^{M} \left(y_i - \sum_{j=0}^{n} \beta_j * X_{ij} \right)^2 + \lambda \sum_{j=0}^{n} |\beta_j|$$

- Some of the features in this technique are completely neglected for model evaluation.
- Hence, the Lasso regression can help us to reduce the overfitting in the model as well as the feature selection.

Lasso regression also performs coefficient minimization, but instead of squaring the magnitudes of the coefficients, it takes the true values of coefficients. This means that the coefficient sum can also be 0, because of the presence of negative coefficients. Consider the cost function for Lasso regression :

Cost function = **Loss** + $\lambda \times \sum \|w\|$

Here,

Loss = Sum of the squared residuals

λ = Penalty for the errors

w = Slope of the curve / line

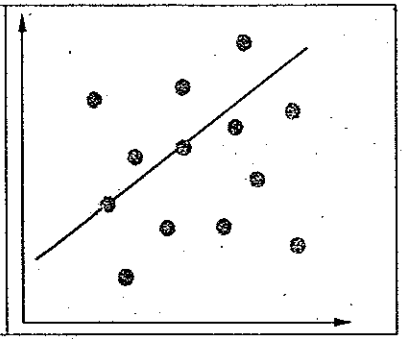


Fig. 5.67. Cost function for Lasso Regression

We can control the coefficient values by controlling the penalty terms, just like we did in Ridge Regression. Again consider a Linear Regression model :

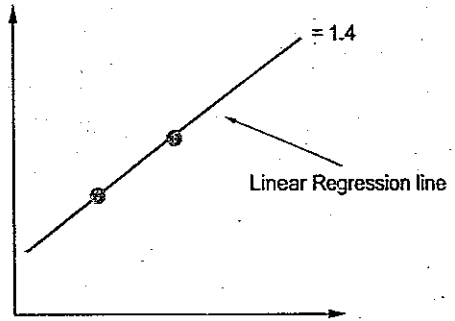


Fig. 5.68. Linear Regression Model

Cost function = Loss + λ x $\sum$ |w|

For Linear Regression line, let's assume,

Loss = 0 (considering the two points on the line)

λ = 1

w = 1.4

Then, Cost function = $0 + 1 \times 1.4 = 1.4$

For Ridge Regression, let's assume,

Loss = $0.32 + 0.12 = 0.1$

λ = 1

w = 0.7

Then, Cost function = $0.1 + 1 \times 0.7 = 0.8$

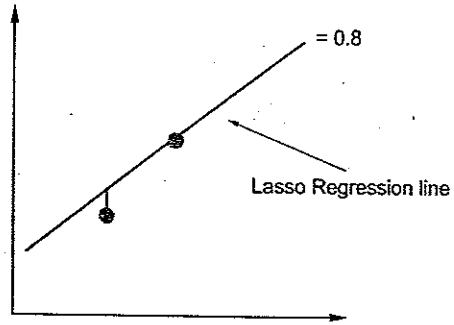


Fig. 5.69. Lasso Regression

Comparing the two models, with all data points, we can see that the Lasso regression line fits the model more accurately than the linear regression line.

Key Difference between Ridge Regression and Lasso Regression

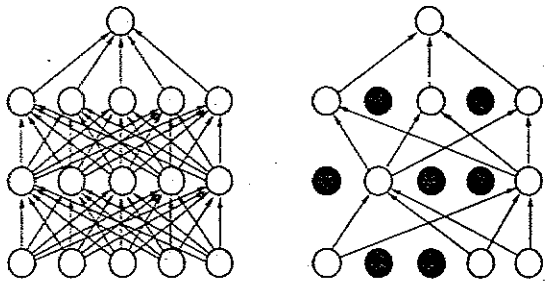
- **Ridge regression** is mostly used to reduce the overfitting in the model, and it includes all the features present in the model. It reduces the complexity of the model by shrinking the coefficients.
- **Lasso regression** helps to reduce the overfitting in the model as well as feature selection.

5.14. DROPOUT

"Dropout" in machine learning refers to the process of randomly ignoring certain nodes in a layer during training.

In the figure below, the neural network on the left represents a typical neural network where all units are activated. On the right, the red units have been dropped

out of the model — the values of their weights and biases are not considered during training.



(a) Standard Neural Net (b) After applying dropout

Dropout is used as a regularization technique — it prevents overfitting by ensuring that no units are codependent.

The term "dropout" refers to dropping out the nodes (input and hidden layer) in a neural network (as seen in Figure (a)). All the forward and backwards connections with a dropped node are temporarily removed, thus creating a new network architecture out of the parent network. The nodes are dropped by a dropout probability of p.

Let's try to understand with a given input x: {1, 2, 3, 4, 5} to the fully connected layer. We have a dropout layer with probability p = 0.2 (or keep probability = 0.8). During the forward propagation (training) from the input x, 20% of the nodes would be dropped, i.e. the x could become {1, 0, 3, 4, 5} or {1, 2, 0, 4, 5} and so on. Similarly, it applied to the hidden layers.

For instance, if the hidden layers have 1000 neurons (nodes) and a dropout is applied with drop probability = 0.5, then 500 neurons would be randomly dropped in every iteration (batch).

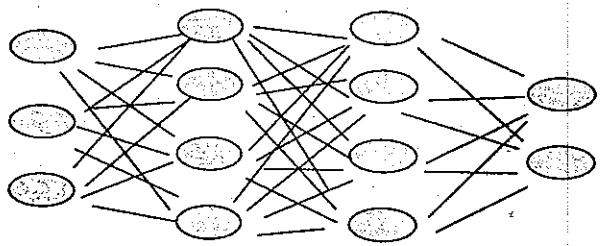
Generally, for the input layers, the keep probability, i.e. 1- drop probability, is closer to 1, 0.8 being the best as suggested by the authors. For the hidden layers, the greater the drop probability more sparse the model, where 0.5 is the most optimised keep probability, that states dropping 50% of the nodes.

Other Common Regularization Methods

When it comes to combating overfitting, dropout is definitely not the only option. Common regularization techniques include:

1. **Early stopping:** stop training automatically when a specific performance measure (eg. Validation loss, accuracy) stops improving
2. **Weight decay:** incentivize the network to use smaller weights by adding a penalty to the loss function (this ensures that the norms of the weights are relatively evenly distributed amongst all the weights in the networks, which prevents just a few weights from heavily influencing network output)
3. **Noise:** allow some random fluctuations in the data through augmentation (which makes the network robust to a larger distribution of inputs and hence improves generalization)
4. **Model combination:** average the outputs of separately trained neural networks (requires a lot of computational power, data, and time)

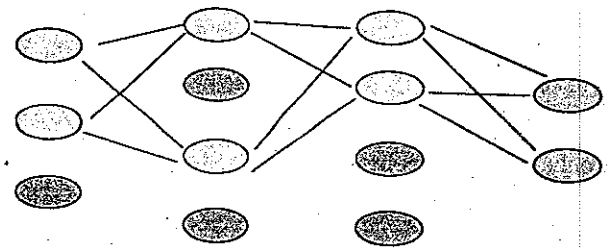
5.14.1. WORKING OF DROPOUT



(a) Fully connected layers

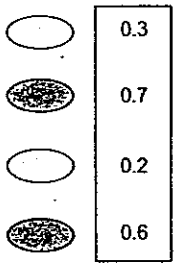
To apply dropout, you need to set a retention probability for each layer. The retention probability specifies the probability that a unit is not dropped. For example, if you set the retention probability to 0.8, the units in that layer have an 80% chance of remaining active and a 20% chance of being dropped.

Standard practice is to set the retention probability to 0.5 for hidden layers and to something close to 1, like 0.8 or 0.9 on the input layer. Output layers generally do not apply dropout.

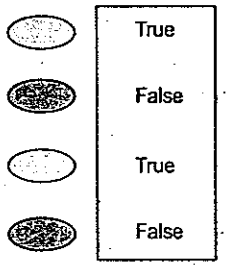


A network with dropout applied to randomly eliminate nodes.

In practice, dropout is applied by creating a mask for each layer and filling it with values between 0 and 1 generated by a random number generator according to the retention probability. Each neuron with a corresponding retention probability below the specified threshold is kept, while the other ones are removed. For example, for the first hidden layer in the network above, we would create a mask with four entries.



Alternatively, we could also fill the mask with random boolean values according to the retention probability. Neurons with a corresponding "True" entry are kept while those with a "False" value are discarded.

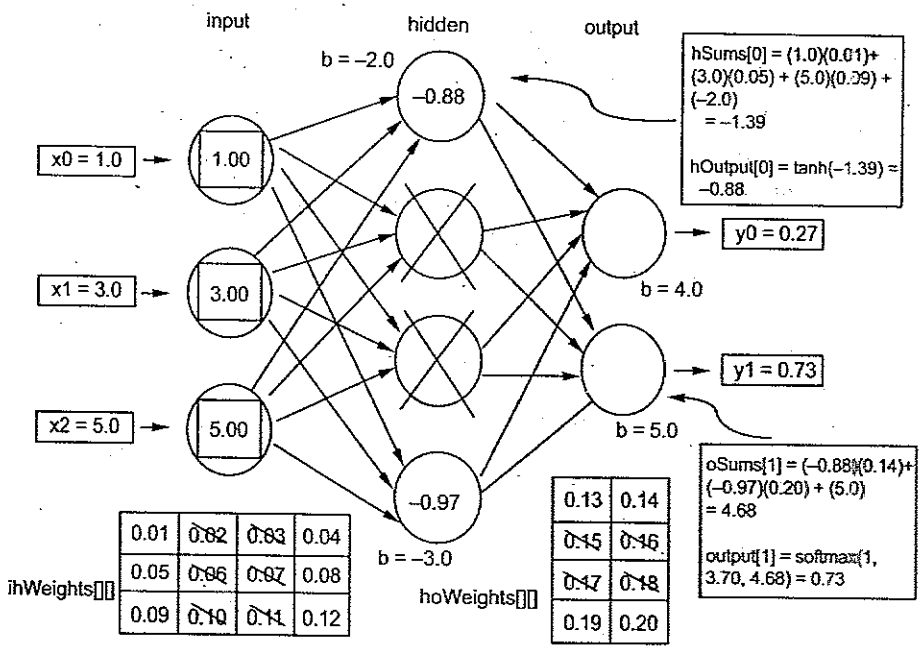


(b) Dropout at Test Time

Dropout is only used during training to make the network more robust to fluctuations in the training data. At test time, however, you want to use the full network in all its glory. In other words, you do not apply dropout with the test data and during inference in production.

But that means your neurons will receive more connections and therefore more activations during inference than what they were used to during training. For example, if you use a dropout rate of 50% dropping two out of four neurons in a layer during training, the neurons in the next layer will receive twice the activations during inference and thus become overexcited. Accordingly, the values produced by

these neurons will, on average, be too large by 50%. To correct this over activation at test and inference time, you multiply the weights of the overexcited neurons by the retention probability (1 – dropout rate) and thus scale them down.



(c) Inverted Dropout

An alternative to scaling the activations at test and inference time by the retention probability is to scale them at training time.

You do this by dropping out the neurons and immediately afterward scaling them by the inverse retention probability.

$$\text{activation} \times \frac{1}{\text{retention probability}}$$

This operation scales the activations of the remaining neurons up to make up for the signal from the other neurons that were dropped.

This corrects the activations right at training time. Accordingly, it is often the preferred option.

5.14.2. DROPOUT REGULARIZATION IN TENSORFLOW

When it comes to applying dropout in practice, you are most likely going to use it in the context of a deep learning framework. In deep learning frameworks, you

usually add an explicit dropout layer after the hidden layer to which you want to apply dropout with the dropout rate (1 – retention probability) set as an argument on the layer. The framework will take care of the underlying details, such as creating the mask.

Using TensorFlow, we start by importing the dropout layer, along with the dense layer and the Sequential API from Tensorflow in Python.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
```

In a simple neural network that consists of a sequence of dense layers, you add dropout to a dense layer by adding an additional "Dropout" layer right after the dense layer. The following code creates a neural network of two dense layers. We add dropout with a rate of 0.2 to the first dense layer and dropout with a rate of 0.5 to the second dense layer. We assume that our dataset has six dimensions which are why we set the input shape parameter equal to 6.

```
model = Sequential([
Dense(64, activation='relu', input_shape=(6,)),
Dropout(0.2),
Dense(64, activation='relu'),
Dropout(0.5),
Dense(3, activation='softmax')
])
```


TWO MARKS QUESTIONS WITH ANSWERS (PART - A)

1. ***How are AI,DL and ML technologies related?***
   - **Artificial Intelligence** is the science of getting machines to mimic the behavior of humans.
   - **Machine learning** is a subset of Artificial Intelligence (AI) that focuses on getting machines to make decisions by feeding them data.
   - **Deep learning** is a subset of Machine Learning that uses the concept of neural networks to solve complex problems.
2. ***Why do we have to assign weights to each input?***
   - Once an input variable is fed to the network, a randomly chosen value is assigned as the weight of that input. The weight of each input data point indicates how important that input is in predicting the outcome.
   - The bias parameter, on the other hand, allows you to adjust the activation function curve in such a way that a precise output is achieved.
3. ***What is Binary classifier in Machine Learning?***

   In Machine Learning, binary classifiers are defined as the function that helps in deciding whether input data can be represented as vectors of numbers and belongs to some specific class.
4. ***List the types of Activation Function.***

   These are the final and important components that help to determine whether the neuron will fire or not. Activation Function can be considered primarily as a step function.

   Types of Activation functions:
   - Sign function
   - Step function, and
   - Sigmoid function
5. ***List the Limitations of Perceptron Model.***

   A perceptron model has limitations as follows:
   - The output of a perceptron can only be a binary number (0 or 1) due to the hard limit transfer function.

- Perceptron can only be used to classify the linearly separable sets of input vectors. If input vectors are non-linear, it is not easy to classify them properly.

6. ***List the Advantages of Multi-Layer Perceptron:***
   - A multi-layered perceptron model can be used to solve complex non-linear problems.
   - It works well with both small and large input data.
   - It helps us to obtain quick predictions after the training.
   - It helps to obtain the same accuracy ratio with large as well as small data.

7. ***Define Linear Activation Function.***
   - The linear activation function, also known as "no activation," or "identity function" (multiplied × 1.0), is where the activation is proportional to the input.
   - The function doesn't do anything to the weighted sum of the input, it simply spits out the value it was given.
   - The equation for Linear activation function is:
   - $f(x) = a.\ x$
   - When $a = 1$ then $f(x) = x$ and this is a special case known as identity.

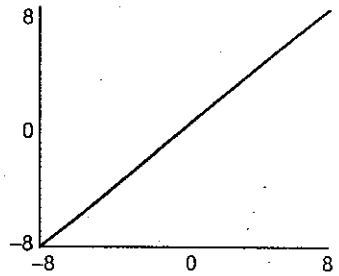


Fig. 5.70. Linear Activation Function

8. ***What is ReLU Function?***

ReLU stands for Rectified Linear Unit.

Although it gives an impression of a linear function, ReLU has a derivative function and allows for backpropagation while simultaneously making it computationally efficient.

The main catch here is that the ReLU function does not activate all the neurons at the same time.

The neurons will only be deactivated if the output of the linear transformation is less than 0.

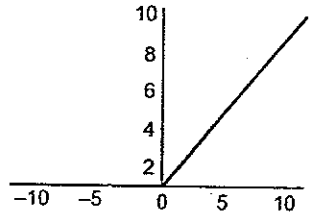


Fig. 5.71. ReLU Activation Function

Mathematically it can be represented as:

$$f(x) = \max(0, x)$$

9. ***Why Is ReLU a Good Activation Function?***

The main reason ReLU wasn't used until more recently is because it was not differentiable at the point zero. Researchers tended to use differentiable functions like sigmoid and tanh. However, it's now determined that ReLU is the best activation function for deep learning.

	$f(x) = \begin{cases} 0 & \text{for } x \le 0 \\ x & \text{for } x > 0 \end{cases}$ $= \max(0, z) = x1_{x>0}$	$f(x) = \begin{cases} 0 & \text{for } x \le 0 \\ x & \text{for } x > 0 \end{cases}$	(0, 00)

Equation for the ReLU activation function. | Image: Wikipedia

The ReLU activation function is differentiable at all points except at zero. For values greater than zero, we just consider the max of the function. This can be written as:

$$f(x) = \max\{0, z\}$$

In simple terms, this can also be written as follows:

```
if input > 0:
  return input
else:
  return 0
```

All the negative values default to zero, and the maximum for the positive number is taken into consideration.

10. ***Define Gradient Descent .***

It is defined as one of the most commonly used iterative optimization algorithms of machine learning to train the machine learning and deep learning models. It helps in finding the local minimum of a function.

11. ***Define Cost-function.***

The cost function is defined as the measurement of difference or error between actual values and expected values at the current position and present in the form of a single real number.

12. ***Define Learning Rate.***

It is defined as the step size taken to reach the minimum or lowest point. This is typically a small value that is evaluated and updated based on the behavior of the cost function. If the learning rate is high, it results in larger steps but also leads to risks of overshooting the minimum. At the same time, a low learning rate shows the small step sizes, which compromises overall efficiency but gives the advantage of more precision.

13. ***List the types of gradient descent.***

(*a*) Batch gradient descent

(*b*) Stochastic gradient descent

(*c*) Mini-batch gradient descent

14. ***Differentiate Vanishing gradients and Exploding gradients.***

- **Vanishing gradients :** This occurs when the gradient is too small. As we move backwards during backpropagation, the gradient continues to become smaller, causing the earlier layers in the network to learn more slowly than later layers. When this happens, the weight parameters update until they become insignificant—i.e. 0—resulting in an algorithm that is no longer learning.
- **Exploding gradients:** This happens when the gradient is too large, creating an unstable model. In this case, the model weights will grow too large, and they will eventually be represented as NaN. One solution to this issue is to leverage a dimensionality reduction technique, which can help to minimize complexity within the model.

15. ***Why do we need Backpropogation and how will you reduce the error?***

While designing a Neural Network, in the beginning, we initialize weights with some random values or any variable for that fact. So, it's not necessary that whatever weight values we have selected will be correct, or it fits our model the best. We have selected some weight values in the beginning, but our model output is way different than our actual output i.e. the error value is huge.

Basically, what we need to do, we need to somehow explain the model to change the parameters (weights), such that error becomes minimum.

Let's put it in an another way, we need to train our model.

One way to train our model is called as Backpropagation.

16. ***Define Restricted Boltzman Networks/ Autoencoders.***

Autoencoders are networks that encode input data as vectors. They create a hidden, or compressed, representation of the raw data. The vectors are useful in dimensionality reduction; the vector compresses the raw data into smaller number of essential dimensions. Autoencoders are paired with decoders, which allows the reconstruction of input data based on its hidden representation.

17. ***List the Steps to Perform Hyperparameter Tuning***

- Select the right type of model.
- Review the list of parameters of the model and build the HP space
- Finding the methods for searching the hyperparameter space
- Applying the cross-validation scheme approach
- Assess the model score to evaluate the model

18. ***What is Data Leakage?***

Data Leakage is when the model somehow knows the patterns in the test data during its training phase. In other words, the data that you are using to train your ML algorithm happens to have the information you are trying to predict.

Data leakage prevents the model to generalize well. It's very difficult for a data scientist to identify data leakage. Some of the reasons for data leakage are

- Outlier and missing value treatment with central values before splitting
- Scaling the data before splitting into training and testing
- Train your model with both train and test data.

19. ***What are Key Difference between Ridge Regression and Lasso Regression?***
    - **Ridge regression** is mostly used to reduce the overfitting in the model, and it includes all the features present in the model. It reduces the complexity of the model by shrinking the coefficients.
    - **Lasso regression** helps to reduce the overfitting in the model as well as feature selection.

20. ***Define dropout.***
    - "Dropout" in machine learning refers to the process of randomly ignoring certain nodes in a layer during training.

REVIEW QUESTIONS

1. Explain briefly about Perceptron and its types
2. Discuss gradient descent optimization
3. Describe briefly about stochastic gradient descent
4. Write short notes on backpropogation
5. Explain the working principle of the ReLu activation function.
6. Explain in brief about regularization and its types.

****

PRACTICAL EXERCISES

Exercise No: 1

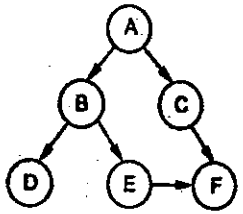
Implementation of Uninformed Search Algorithms (BFS, DFS)

The BFS Algorithm

1. Pick any node, visit the adjacent unvisited vertex, mark it as visited, display it, and insert it in a queue.
2. If there are no remaining adjacent vertices left, remove the first vertex from the queue.
3. Repeat step 1 and step 2 until the queue is empty or the desired node is found.

Implementation

Consider the graph, which is implemented in the code below:




```
graph = {
 'A' : ['B','C'],
 'B' : ['D', 'E'],
 'C' : ['F'],
 'D' : [],
 'E' : ['F'],
 'F' : []
}
visited = [] # List to keep track of visited nodes.
queue = []     #Initialize a queue
def bfs(visited, graph, node):
visited.append(node)
```

```
    queue.append(node)
    while queue:
      s = queue.pop(0)
      print (s, end = " ")
      for neighbour in graph[s]:
        if neighbour not in visited:
    visited.append(neighbour)
    queue.append(neighbour)
    # Driver Code
    bfs(visited, graph, 'A')
```

Output

A B C D E F

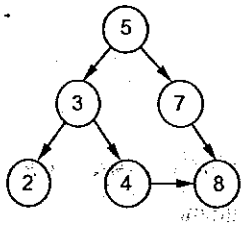
DSF Algorithm

The DSF algorithm follows as:

1. We will start by putting any one of the graph's vertex on top of the stack.
2. After that take the top item of the stack and add it to the visited list of the vertex.
3. Next, create a list of that adjacent node of the vertex. Add the ones which aren't in the visited list of vertexes to the top of the stack.
4. Lastly, keep repeating steps 2 and 3 until the stack is empty.

Implementation

Consider the graph, which is implemented in the code below:




```
# Using a Python dictionary to act as an adjacency list
graph = {
  '5' : ['3','7'],
```



```
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
visited = set() # Set to keep track of visited nodes of graph.
def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
visited.add(node)
        for neighbour in graph[node]:
dfs(visited, graph, neighbour)
# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

Output

Following is the Depth-First Search

5 3 2 4 8 7

Exercise No: 2

Implementation of Informed Search Algorithms (A*, memory-bounded A*)

Algorithm of A* search:

Step1: Place the starting node in the OPEN list.

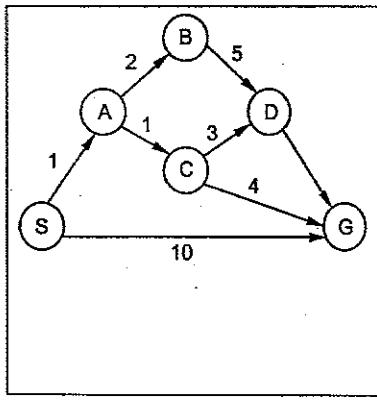
Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function ($g + h$), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest $g(n')$ value.

Step 6: Return to Step 2.



State	$h(n)$
S	5
A	3
B	4
C	2
D	6
G	0

Implementation

```
def heuristic(a: GridLocation, b: GridLocation) -> float:
    (x1, y1) = a
    (x2, y2) = b
    return abs(x1 - x2) + abs(y1 - y2)
def a_star_search(graph: WeightedGraph, start: Location, goal: Location):
    frontier = PriorityQueue()
frontier.put(start, 0)
came_from: dict[Location, Optional[Location]] = {}
cost_so_far: dict[Location, float] = {}
```



```
came_from[start] = None
cost_so_far[start] = 0
while not frontier.empty():
    current: Location = frontier.get()
    if current == goal:
        break
for next in graph.neighbors(current):
new_cost = cost_so_far[current] + graph.cost(current, next)
        if next not in cost_so_far or new_cost<cost_so_far[next]:
cost_so_far[next] = new_cost
            priority = new_cost + heuristic(next, goal)
frontier.put(next, priority)
came_from[next] = current
    return came_from, cost_so_far
```

Output

S →A→C→G it provides the optimal path with cost 6

Memory-bounded A*

SMA* has the following properties

- It works with a heuristic, just as A*
- It is complete if the allowed memory is high enough to store the shallowest solution
- It is optimal if the allowed memory is high enough to store the shallowest optimal solution, otherwise it will return the best solution that fits in the allowed memory
- It avoids repeated states as long as the memory bound allows it
- It will use all memory available
- Enlarging the memory bound of the algorithm will only speed up the calculation

- When enough memory is available to contain the entire search tree, then calculation has an optimal speed

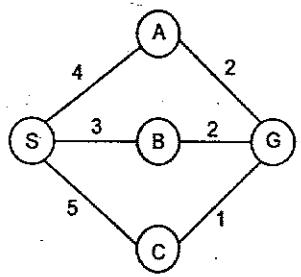
Implementation

```
function SMA-star(problem): path
 queue: set of nodes, ordered by f-cost;
begin
queue.insert(problem.root-node);

 while True do begin
  if queue.empty() then return failure; //there is no solution that fits in the given
memory
node := queue.begin(); // min-f-cost-node
  if problem.is-goal(node) then return success;

s := next-successor(node)
if !problem.is-goal(s) && depth(s) == max_depth then
    f(s) := inf;
    // there is no memory left to go past s, so the entire path is useless
  else
    f(s) := max(f(node), g(s) + h(s));
    // f-value of the successor is the maximum of
    //    f-value of the parent and
    //    heuristic of the successor + path length to the successor
  endif
  if no more successors then
    update f-cost of node and those of its ancestors if needed
  if node.successors⊆queue then queue.remove(node);
  // all children have already been added to the queue via a shorter way
```



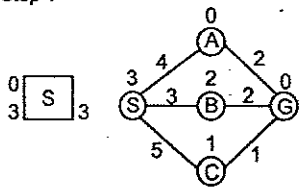
```
  if memory is full then begin
badNode := shallowest node with highest f-cost;
   for parent in badNode.parents do begin
parent.successors.remove(badNode);
    if needed then queue.insert(parent);
endfor
  endif
queue.insert(s);
endwhile
end
```

Example

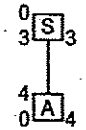


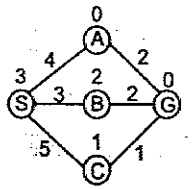
	S	A	B	C	G
heuristic	3	0	2	1	0

Step 1



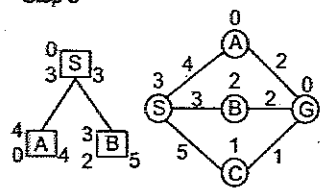
Step 2





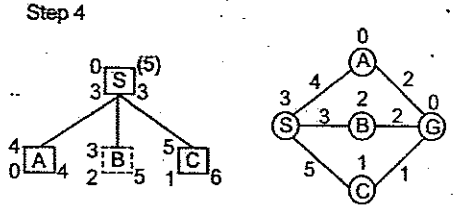
Generate children (One by one)

Step 3



Generate children (One by one)

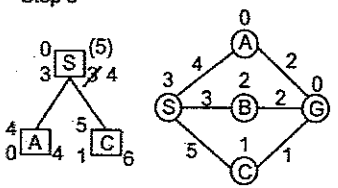
Step 4



Generate children (One by one)

Memory full

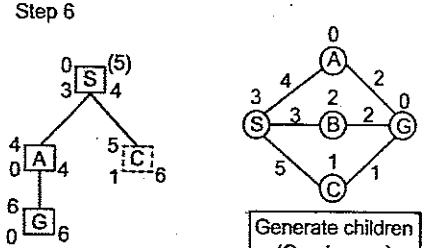
Step 5



All children are explored

Adjust f-values

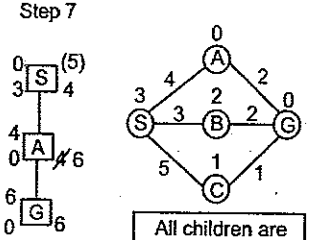
Step 6



Generate children (One by one)

Memory full

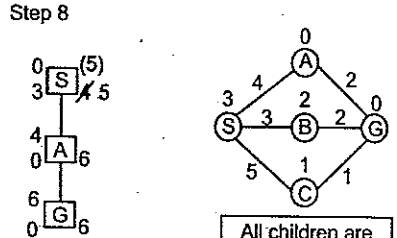
Step 7



All children are explored

Adjust f-values

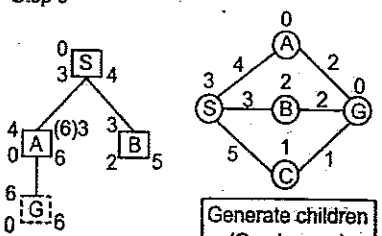
Step 8



All children are explored (update)

Adjust f-values

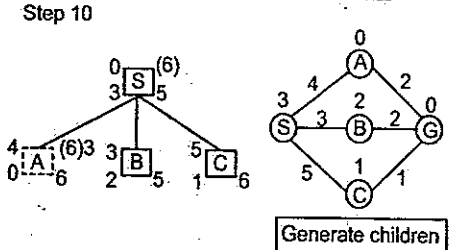
Step 9



Generate children (One by one)

Memory full

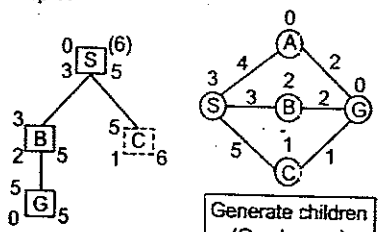
Step 10



Generate children (One by one)

Memory full

Step 11



Generate children (One by one)

Memory full

Exercise No: 3

Implement Naïve Bayes models

We are using the Advertisement clicking dataset (about users clicking the ads or not)

```
#importing libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import Sklearn
#loading the dataset
data = pd.read_csv('/content/drive/MyDrive/ML_datasets/advertising.csv')
#head of the dataset
data.head()


#describing the data
```

```
data.describe()
#drop 'Ad Line Topic' ,'City', 'Country' and Timestamp.
data.drop(['Ad Topic Line','City','Country','Timestamp'],axis = 1,inplace = True)
data.head()
#train test split the data
X = data.iloc[:,0:4].values
Y = data['Clicked on Ad'].values
from sklearn.model_selection import train_test_split
X_train,X_test,Y_train,Y_test = train_test_split(X,Y,test_size = 1/3,random_state
= 0)
print('X_train shape:',X_train.shape)
print('X_test shape:',X_test.shape)
print('Y_train shape:',Y_train.shape)
print('Y_test shape:',X_test.shape)
#feature scaling
from sklearn.preprocessing import StandardScaler
stdscaler = StandardScaler()
X_train = stdscaler.fit_transform(X_train)
X_test = stdscaler.transform(X_test)
#naive bayes model
from sklearn.naive_bayes import GaussianNB
clf = GaussianNB()
clf.fit(X_train,Y_train
#predicting the result
y_pred = clf.predict(X_test)
```



```
print(y_pred)
#confusion matrix
from sklearn.metrics import confusion_matrix,accuracy_score
cm = confusion_matrix(Y_test, y_pred)
print(cm)
#accuracy score of the model
print('Accuracy score :',accuracy_score(Y_test,y_pred))
#plotting the confusion matrix
plt.figure(figsize=(10,5))
plt.title('Confusion matrix')
sns.heatmap(cm,annot=True,fmt='d',cmap='inferno_r')
```

Output

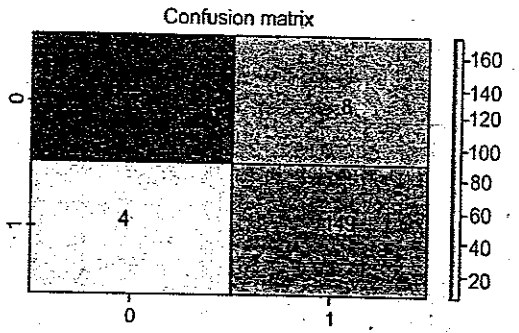
	Daily Time spent on site	Age	Area Income	Daily Internet Usage	Ad Topic Line	City	Male	Country	Timestamp
0	68.95	35	61833.90	256.09	Cloned 5th generation orchestration	Wrightburgh	0	Tunisia	2018-03-27 00:53:11
1	80.23	31	68441.85	193.77	Monitored national standardization	West Jodi	1	Nauru	2016-04-04 01:39:02
2	69.47	26	59785.94	236.50	Organic bottom-line service-desk	Davidton	0	San Marino	2016-03-13 20:35:42
3	74:15	29	54806.18	245.89	Triple-buffered reciprocal time-frame	West Terrifurt	1	Italy	2016-01-10 02:31:19
4	68.37	35	73889.99	225.58	Robust logistical utilization	South Manuel	0	Iceland	2016-06-03 03:36:18

	Daily Time Spent on Site	Age	Area Income	Daily Internet Usage	Male	Clicked on Ad
count	1000,000000	1000,000000	1000,000000	1000,000000	1000,000000	1000,000000
mean	65.000200	36.009000	55000.000080	180.000100	0.481000	0.50000
std	15.853615	8.785562	13414.634022	43.902339	0.499889	0.50025
min	32.600000	19,000000	13996.500000	104.780000	0.000000	0.00000
25%	51.360000	29.000000	47031.802500	138.830000	0.000000	0.00000
50%	68.215000	35.000000	57012.300000	183.130000	0.000000	0.50000
75%	78.547500	42.000000	65470.635000	218.792500	1.000000	1.00000
max	91.430000	61.000000	79484.800000	269.960000	1.000000	1.00000

	Daily Time Spent on Site	Age	Area Income	Daily Internet Usage	Male	Clicked on Ad
0	68.95	35	61833.90	256.09	0	0
1	80.23	31	68441.85	193.77	1	0
2	69.47	26	59785.94	236.50	0	0
3	74.15	29	54806.18	245.89	1	0
4	68.37	35	73889.99	225.58	0	0

[[173 8]
[4 149]]

Accuracy score : 0.9640718562874252



Exercise No: 4

Implement Bayesian Networks

Data Set:

Title: Heart Disease Databases

The Cleveland database contains 76 attributes, but all published experiments refer to using a subset of 14 of them. In particular, the Cleveland database is the only one that has been used by ML researchers to this date. The "Heart Disease" field refers to the presence of heart disease in the patient. It is an integer value from 0 (no presence) to 4.

Database: 0 1 2 3 4 Total

Cleveland: 164 55 36 35 13 303

```
import numpy as np
import csv
import pandas as pd
from pgmpy.models import BayesianModel
from pgmpy.estimators import MaximumLikelihoodEstimator
from pgmpy.inference import VariableElimination
#read Cleveland Heart Disease data
heartDisease = pd.read_csv('heart.csv')
heartDisease = heartDisease.replace('?',np.nan)
#display the data
print('Few examples from the dataset are given below')
print(heartDisease.head())
#Model Bayesian Network
Model=BayesianModel([('age','trestbps'),('age','fbs'),('sex','trestbps'),('exang','trestbps'),('trestbps','heartdisease'),('fbs','heartdisease'),('heartdisease','restecg'),('heartdisease','thalach'),('heartdisease','chol')])
```

```
#Learning CPDs using Maximum Likelihood Estimators
print('\n Learning CPD using Maximum likelihood estimators')
model.fit(heartDisease,estimator=MaximumLikelihoodEstimator)
# Inferencing with Bayesian Network
print('\n Inferencing with Bayesian Network:')
HeartDisease_infer = VariableElimination(model)
#computing the Probability of HeartDisease given Age
print('\n 1. Probability of HeartDisease given Age=30')
q=HeartDisease_infer.query(variables=['heartdisease'],evidence={'age':28})
print(q['heartdisease'])
#computing the Probability of HeartDisease given cholesterol
print('\n 2. Probability of HeartDisease given cholesterol=100')
q=HeartDisease_infer.query(variables=['heartdisease'],evidence={'chol':100})
print(q['heartdisease'])
```

Output:

Few examples from the dataset are given below

	age	sex	cp	trestbps	...	slope	ca	thal	heartdisease
0	63	1	1	145	...	3	0	6	0
1	67	1	4	160	...	2	3	3	2
2	67	1	4	120	...	2	2	7	1
3	37	1	3	130	...	3	0	3	0
4	41	0	2	130	...	1	0	3	0

[5 rows × 14 columns]

Learning CPD using Maximum likelihood estimators

Inferencing with Bayesian Network:

1. **Probability of Heart Disease given Age = 28**

heartdisease	phi (heartdisease)
heartdisease_0	0.6791
heartdisease_1	0.1212
heartdisease_2	0.0810
heartdisease_3	0.0939
heartdisease_4	0.0247

2. **Probability of Heart Disease given cholesterol = 100**

heartdisease	phi (heartdisease)
heartdisease_0	0.5400
heartdisease_1	0.1513
heartdisease_2	0.1303
heartdisease_3	0.1259
heartdisease_4	0.0506

Exercise No: 5

5. Build Regression Models

- Simple linear regression is an approach for predicting a response using a single feature.
- It is assumed that the two variables are linearly related. Hence, we try to find a linear function that predicts the response value(y) as accurately as possible as a function of the feature or independent variable(x).

Let us consider a dataset where we have a value of response y for every feature x:

x	0	1	2	3	4	5	6	7	8	9
y	1	3	2	5	7	8	8	9	10	12

Implementation

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

def estimate_coef(x, y):
    # number of observations/points
    n = np.size(x)

    # mean of x and y vector
    m_x = np.mean(x)
    m_y = np.mean(y)

    # calculating cross-deviation and deviation about x
    SS_xy = np.sum(y*x) - n*m_y*m_x
    SS_xx = np.sum(x*x) - n*m_x*m_x

    # calculating regression coefficients
    b_1 = SS_xy / SS_xx
    b_0 = m_y - b_1*m_x
    return (b_0, b_1)

def plot_regression_line(x, y, b):
    # plotting the actual points as scatter plot
    plt.scatter(x, y, color = "m",
                marker = "o", s = 30)

    # predicted response vector
    y_pred = b[0] + b[1]*x
```



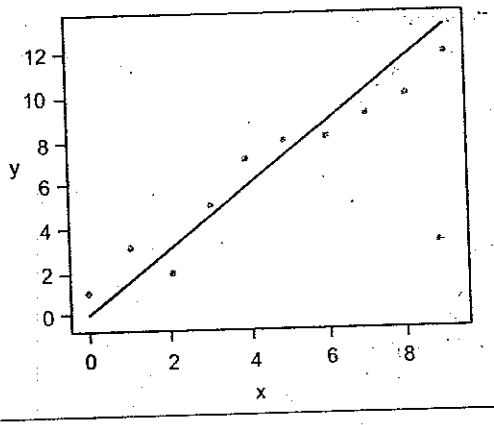
```
    # plotting the regression line
    plt.plot(x, y_pred, color = "g")

    # putting labels
    plt.xlabel('x')
    plt.ylabel('y')

    # function to show plot
    plt.show()

def main():
    # observations/data
    x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])

    # estimating coefficients
    b = estimate_coef(x, y)
    print("Estimated coefficients:\nb_0 = {} \
        \nb_1 = {}".format(b[0], b[1]))

    # plotting regression line
    plot_regression_line(x, y, b)

if __name__ == "__main__":
    main()
```

Output:

Estimated coefficients:

b_0 = -0.0586206896552

b_1 = 1.45747126437



Exercise No: 6

Build Decision Trees and Random Forests

Decision Trees

```
# Run this program on your local python interpreter, provided you have installed the required #libraries.
# Importing the required packages
import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix
from sklearn.model_selection 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('https://archive.ics.uci.edu/ml/machine-learning-'+'databases/balance-scale/balance-scale.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()
```

Data Infomation:

Dataset Length: 625

Dataset Shape: (625, 5)

Dataset:

```
    0 1 2 3 4
0   B 1 1 1 1
1   R 1 1 1 2
2   R 1 1 1 3
3   R 1 1 1 4
4   R 1 1 1 5
```

Results Using Gini Index:

Predicted values:

```
['R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'R' 'L'
 'L' 'R' 'L' 'R' 'L' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'L'
 'L' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'R' 'L' 'R'
 'R' 'L' 'R' 'R' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'R'
 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'R' 'L'
```

'R' 'R' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'R' 'L' 'R' 'R' 'R' 'R' 'R'
'R' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L'
'L' 'L' 'L' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R'
'L' 'L' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R'
'L' 'L' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'R' 'R'
'L' 'R' 'R' 'L' 'L' 'R' 'R' 'R']

Confusion Matrix: [[0 6 7]
[0 67 18]
[0 19 71]]

Accuracy : 73.4042553191

Report :

	precision	recall	f1-score	support
B	0.00	0.00	0.00	13
L	0.73	0.79	0.76	85
R	0.74	0.79	0.76	90
avg/total	0.68	0.73	0.71	188

Results Using Entropy:

Predicted values:

['R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'L'
'L' 'R' 'L' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'L' 'L'
'L' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'L' 'R' 'L' 'L' 'R' 'L' 'L'
'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'L' 'R' 'L' 'L' 'L' 'R'
'R' 'L' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'R' 'L'
'R' 'R' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'R' 'R' 'R' 'R' 'R'
'R' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L'
'L' 'L' 'L' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R'
'L' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R'
'R' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'L' 'R'
'R' 'R' 'L' 'L' 'L' 'R' 'R' 'R']

Confusion Matrix: [[0 6 7]
[0 63 22]
[0 20 70]]

Accuracy : 70.7446808511

Report :

	precision	recall	f1-score	support
B	0.00	0.00	0.00	13
L	0.71	0.74	0.72	85
R	0.71	0.78	0.74	90
avg / total	0.66	0.71	0.68	188

Random Forests

We will be using a data set of kyphosis patients and building a random forest algorithm to predict whether or not patients have the disease.

```
#Numerical computing libraries
import pandas as pd
import numpy as np
#Visalization libraries
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
raw_data = pd.read_csv('kyphosis-data.csv')
raw_data.columns
#Exploratory data analysis
raw_data.info()
sns.pairplot(raw_data, hue = 'Kyphosis')
#Split the data set into training data and test data
from sklearn.model_selection import train_test_split
x = raw_data.drop('Kyphosis', axis = 1)
y = raw_data['Kyphosis']
```

```
x_training_data, x_test_data, y_training_data, y_test_data = train_test_split(x, y, test_size = 0.3)
#Train the decision tree model
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier()
model.fit(x_training_data, y_training_data)
predictions = model.predict(x_test_data)
#Measure the performance of the decision tree model
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
print(classification_report(y_test_data, predictions))
print(confusion_matrix(y_test_data, predictions))
#Train the random forests model
from sklearn.ensemble import RandomForestClassifier
random_forest_model = RandomForestClassifier()
random_forest_model.fit(x_training_data, y_training_data)
random_forest_predictions = random_forest_model.predict(x_test_data)
#Measure the performance of the random forest model
print(classification_report(y_test_data, random_forest_predictions))
print(confusion_matrix(y_test_data, random_forest_predictions))
```

Output

	precision	recall	f1-score	support
absent	0.85	0.89	0.87	19
present	0.60	0.50	0.55	6
accuracy			0.80	25
macro avg	0.72	0.70	0.71	25
weighted avg	0.79	0.80	0.79	25

```
[[17  2]
 [ 3  3]]
```


Exercise No: 7

Build SVM Models

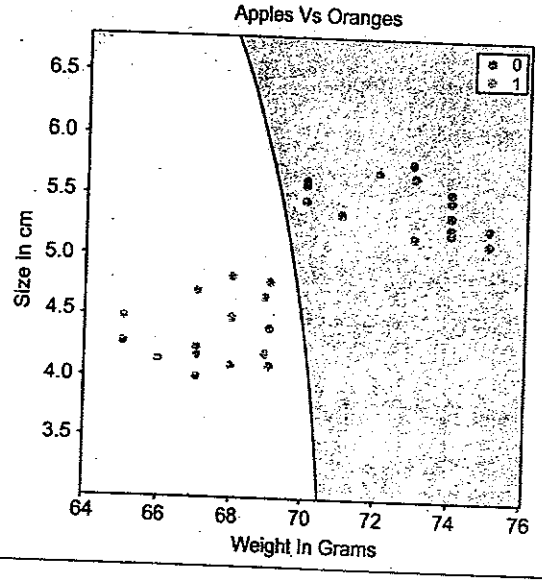
Let's use the same dataset of apples and oranges. We will consider the Weights and Size for 20 each

```
import pandas as pd
data = pd.read_csv("apples_and_oranges.csv")
from sklearn.model_selection import train_test_split
training_set, test_set = train_test_split(data, test_size = 0.2, random_state = 1)
#Classifying the predictors and target
X_train = training_set.iloc[:,0:2].values
Y_train = training_set.iloc[:,2].values
X_test = test_set.iloc[:,0:2].values
Y_test = test_set.iloc[:,2].values
#Initializing Support Vector Machine and fitting the training data
from sklearn.svm import SVC
classifier = SVC(kernel='rbf', random_state = 1)
classifier.fit(X_train,Y_train)
#Predicting the classes for test set
Y_pred = classifier.predict(X_test)
#Attaching the predictions to test set for comparing
test_set["Predictions"] = Y_pred
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test,Y_pred)
accuracy = float(cm.diagonal().sum())/len(Y_test)
```

```
print("\nAccuracy Of SVM For The Given Dataset : ", accuracy)
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
Y_train = le.fit_transform(Y_train)
from sklearn.svm import SVC
classifier = SVC(kernel='rbf', random_state = 1)
classifier.fit(X_train,Y_train)
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
plt.figure(figsize = (7,7))
X_set, y_set = X_train, Y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:,
0].max() + 1, step = 0.01), np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:,
1].max() + 1, step = 0.01))
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(),
X2.ravel()]).T).reshape(X1.shape), alpha = 0.75, cmap = ListedColormap(('black',
'white')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = ListedColormap(('red',
'orange'))(i), label = j)
plt.title('Apples Vs Oranges')
plt.xlabel('Weight In Grams')
plt.ylabel('Size in cm')
plt.legend()
plt.show()
```


Output:

Accuracy Of SVM For The Given Dataset : 0.875



Exercise No: 8

Implement Ensembling Techniques

(*i*) Averaging method: It is mainly used for regression problems. The method consists of building multiple models independently and returning the average of the prediction of all the models. In general, the combined output is better than an individual output because variance is reduced.

In the below example, three regression models (linear regression, xgboost, and random forest) are trained and their predictions are averaged. The final prediction output is pred_final.

```
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

# 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 model objects with default parameters
model_1 = LinearRegression()
model_2 = xgb.XGBRegressor()
model_3 = RandomForestRegressor()

# training all the model on the training dataset
model_1.fit(X_train, y_target)
model_2.fit(X_train, y_target)
```



```
model_3.fit(X_train, y_target)

# predicting the output on the validation dataset
pred_1 = model_1.predict(X_test)
pred_2 = model_2.predict(X_test)
pred_3 = model_3.predict(X_test)

# final prediction after averaging on the prediction of all 3 models
pred_final = (pred_1+pred_2+pred_3)/3.0

# printing the mean squared error between real value and predicted
value
print(mean_squared_error(y_test, pred_final))
```

Output:

4560

(*ii*) Max voting: It is mainly used for classification problems. The method consists of building multiple models independently and getting their individual output called 'vote'. The class with maximum votes is returned as output.

In the below example, three classification models (logistic regression, xgboost, and random forest) are combined using sklearnVotingClassifier, that model is trained and the class with maximum votes is returned as output. The final prediction output is pred_final. Please note it's a classification, not regression, so the loss may be different from other types of ensemble methods.

```
# 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
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()
model_3 = RandomForestClassifier()

# 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))
```

Output:

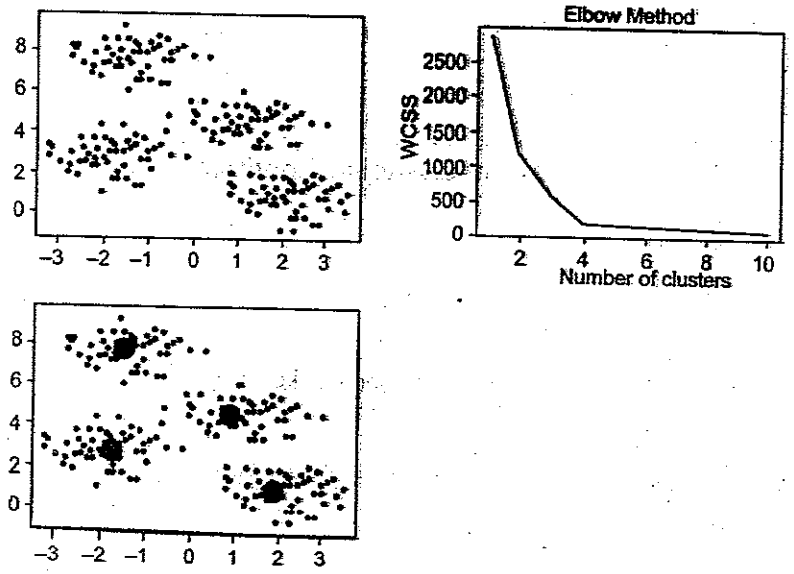
231

Exercise No: 9

Implement Clustering Algorithms

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.datasets.samples_generator import make_blobs
from sklearn.cluster import KMeans
X, y = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)
plt.scatter(X[:,0], X[:,1])
wcss = []
for i in range(1, 11):
```

```
kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10, random_state=0)
kmeans.fit(X)
wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.title('Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
kmeans = KMeans(n_clusters=4, init='k-means++', max_iter=300, n_init=10, random_state=0)
pred_y = kmeans.fit_predict(X)
plt.scatter(X[:,0], X[:,1])
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=300, c='red')
plt.show()
```



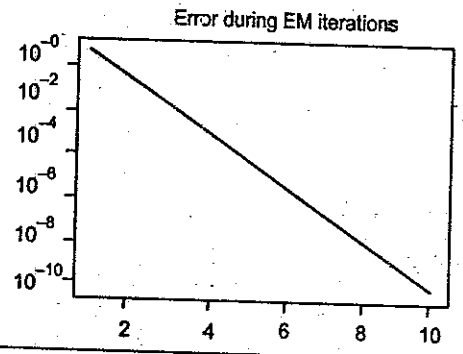
Exercise No: 10

Implement EM for Bayesian Networks

1. Given a set of incomplete data, start with a set of initialized parameters.
2. Expectation step (E – step): In this expectation step, by using the observed available data of the dataset, we can try to estimate or guess the values of the missing data. Finally, after this step, we get complete data having no missing values.
3. Maximization step (M – step): Now, we have to use the complete data, which is prepared in the expectation step, and update the parameters.
4. Repeat step 2 and step 3 until we converge to our solution.

```
#import
import pandas as pd
importnumpyas np
importpyAgrumas gum

# Read data that does not contain any missing values
data = pd.read_csv("asia10K.csv")
# not exactly the same names
data = pd.DataFrame(data, columns=["smoking", "lung_cancer", "positive_XraY"])
test_data = data[:2000]
new_data = data[2000:].copy()

# Learn structure of initial model from data
learner=gum.BNLearner(test_data)
learner.useScoreBIC()
learner.useGreedyHillClimbing()
model=learner.learnBN()
```

```
# create some missing values
new_data["smoking"][:500] = "?"# instead of NaN

# learn parameterization of BN
bn = gum.BayesNet(model)
learner2=gum.BNLearner(new_data,model)
learner2.useEM(1e-10)
learner2.fitParameters(bn)
  1 import matplotlib.pyplot as plt
  2 plt.plot(np.arange(1, 1 + 1learner2.nbrIteration( )),learner2.history( ))
  3 plt.semilogy( )
  4. plt.title("Error during EM iterations");
```



Exercise No: 11

Build simple NN models

```
# Import python libraries required in this example:
from keras.models import Sequential
from keras.layers import Dense, Activation
import numpy as np
```



```
# Use numpy arrays to store inputs (x) and outputs (y):
x = np.array([[0,0], [0,1], [1,0], [1,1]])
y = np.array([[0], [1], [1], [0]])
# Define the network model and its arguments.
# Set the number of neurons/nodes for each layer:
model = Sequential()
model.add(Dense(2, input_shape=(2,)))
model.add(Activation('sigmoid'))
model.add(Dense(1))
model.add(Activation('sigmoid'))
# Compile the model and calculate its accuracy:
model.compile(loss='mean_squared_error', optimizer='sgd', metrics=['accuracy'])
# Print a summary of the Keras model:
model.summary()
```

```
>>> # Print a summary of the Keras model:
>>> model.summary ( )
Model: "sequential_4"
```

Layer (type)	Output Shape	Param #
dense_7 (Dense)	(None, 2)	6
activation_7 (Activation)	(None, 2)	0
dense_8 (Dense)	(None, 1)	3
activation_8 (Activation)	(None, 1)	0

```
Total params: 9
Trainable params; 9
Non-Trainable params: 0
```

Exercise No: 12

Build Deep Learning NN Models

```
import numpy as np
import tensorflow as tf
from time import time
import math

from include.data import get_data_set
from include.model import model, lr

train_x, train_y = get_data_set("train")
test_x, test_y = get_data_set("test")
tf.set_random_seed(21)
x, y, output, y_pred_cls, global_step, learning_rate = model()
global_accuracy = 0
epoch_start = 0

# PARAMS
_BATCH_SIZE = 128
_EPOCH = 60
_SAVE_PATH = "./tensorboard/cifar-10-v1.0.0/"

# LOSS AND OPTIMIZER
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=output,
labels=y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate,
                                   beta1=0.9,
                                   beta2=0.999,
```



```
                                   epsilon=1e-08).minimize(loss, global_step=global_step)

# PREDICTION AND ACCURACY CALCULATION
correct_prediction = tf.equal(y_pred_cls, tf.argmax(y, axis=1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# SAVER
merged = tf.summary.merge_all()
saver = tf.train.Saver()
sess = tf.Session()
train_writer = tf.summary.FileWriter(_SAVE_PATH, sess.graph)


try:
    print("
Trying to restore last checkpoint ...")
    last_chk_path = tf.train.latest_checkpoint(checkpoint_dir=_SAVE_PATH)
    saver.restore(sess, save_path=last_chk_path)
    print("Restored checkpoint from:", last_chk_path)
except ValueError:
    print("
Faiied to restore checkpoint. Initializing variables instead.")
    sess.run(tf.global_variables_initializer())

def train(epoch):
    global epoch_start
    epoch_start = time()
    batch_size = int(math.ceil(len(train_x) / _BATCH_SIZE))
    i_global = 0
```

```
    for s in range(batch_size):
      batch_xs = train_x[s*_BATCH_SIZE: (s+1)*_BATCH_SIZE]
      batch_ys = train_y[s*_BATCH_SIZE: (s+1)*_BATCH_SIZE]
      start_time = time()
      i_global, _, batch_loss, batch_acc = sess.run(
        [global_step, optimizer, loss, accuracy],
        feed_dict={x: batch_xs, y: batch_ys, learning_rate: lr(epoch)})
      duration = time() - start_time
      if s % 10 == 0:
        percentage = int(round((s/batch_size)*100))
        bar_len = 29
        filled_len = int((bar_len*int(percentage))/100)
        bar = '=' * filled_len + '>' + '-' * (bar_len - filled_len)
        msg = "Global step: {:>5} - [{}] {:>3}% - acc: {:.4f} - loss: {:.4f} - {:.1f}
sample/sec"
        print(msg.format(i_global, bar, percentage, batch_acc, batch_loss,
_BATCH_SIZE / duration))
      test_and_save(i_global, epoch)

  def test_and_save(_global_step, epoch):
    global global_accuracy
    global epoch_start
    i = 0
    predicted_class = np.zeros(shape=len(test_x), dtype=np.int)
    while i<len(test_x): j = min(i + _BATCH_SIZE, len(test_x)) batch_xs =
test_x[i:j, :] batch_ys = test_y[i:j, :] predicted_class[i:j] = sess.run( y_pred_cls,
feed_dict={x: batch_xs, y: batch_ys, learning_rate: lr(epoch)} ) i = j correct =
(np.argmax(test_y, axis=1) == predicted_class) acc = correct.mean()*100
correct_numbers = correct.sum() hours, rem = divmod(time() - epoch_start, 3600)
```



```
minutes, seconds = divmod(rem, 60) mes = " Epoch {} - accuracy: {:.2f}% ({}/{}) -
time: {:0>2}:{:0>2}:{:05.2f}"
    print(mes.format((epoch+1), acc, correct_numbers, len(test_x), int(hours),
int(minutes), seconds))
    if global_accuracy != 0 and global_accuracy<acc: summary =
tf.Summary(value=[ tf.Summary.Value(tag="Accuracy/test", simple_value=acc), ])
train_writer.add_summary(summary, _global_step) saver.save(sess,
save_path=_SAVE_PATH, global_step=_global_step) mes = "This epoch receive
better accuracy: {:.2f} > {:.2f}. Saving session..."
      print(mes.format(acc, global_accuracy))
      global_accuracy = acc
    elifglobal_accuracy == 0:
      global_accuracy = acc
    print("#################################################################
##########################################")
  def main():
    train_start = time()

    for i in range(_EPOCH):
      print("
Epoch: {}/{}
".format((i+1), _EPOCH))
      train(i)
    hours, rem = divmod(time() - train_start, 3600)
    minutes, seconds = divmod(rem, 60)
    mes = "Best accuracy pre session: {:.2f}, time: {:0>2}:{:0>2}:{:05.2f}"
    print(mes.format(global_accuracy, int(hours), int(minutes), seconds))
if __name__ == "__main__":
  main()
sess.close()
```

Output:

Epoch: 60/60

Global step: 23070 - [>----------] 0% - acc: 0.9531 - loss: 1.5081 - 7045.4 sample/sec

Global step: 23080 - [>----------] 3% - acc: 0.9453 - loss: 1.5159 - 7147.6 sample/sec

Global step: 23090 - [=>----------] 5% - acc: 0.9844 - loss: 1.4764 - 7154.6 sample/sec

Global step: 23100 - [=>----------] 8% - acc: 0.9297 - loss: 1.5307 - 7104.4 sample/sec

Global step: 23110 - [=>----------] 10% - acc: 0.9141 - loss: 1.5462 - 7091.4 sample/sec

Global step: 23120 - [==>----------] 13% - acc: 0.9297 - loss: 1.5314 - 7162.9 sample/sec

Global step: 23130 - [==>----------] 15% - acc: 0.9297 - loss: 1.5307 - 7174.8 sample/sec

Global step: 23140 - [==>----------] 18% - acc: 0.9375 - loss: 1.5231 - 7140.0 sample/sec

Global step: 23150 - [==>----------] 20% - acc: 0.9297 - loss: 1.5301 - 7152.8 sample/sec

Global step: 23160 - [===>----------] 23% - acc: 0.9531 - loss: 1.5080 - 7112.3 sample/sec

Global step: 23170 - [===>----------] 26% - acc: 0.9609 - loss: 1.5000 - 7154.0 sample/sec

Global step: 23180 - [===>----------] 28% - acc: 0.9531 - loss: 1.5074 - 6862.2 sample/sec

Global step: 23190 - [===>----------] 31% - acc: 0.9609 - loss: 1.4993 - 7134.5 sample/sec

Global step: 23200 - [====>----------] 33% - acc: 0.9609 - loss: 1.4995 - 7166.0 sample/sec

Global step: 23210 - [====>----------] 36% - acc: 0.9375 - loss: 1.5231 - 7116.7 sample/sec

Global step: 23220 - [====>----------] 38% - acc: 0.9453 - loss: 1.5153 - 7134.1 sample/sec

Global step: 23230 - [====>----------] 41% - acc: 0.9375 - loss: 1.5233 - 7074.5 sample/sec

Global step: 23240 - [=====>----------] 43% - acc: 0.9219 - loss: 1.5387 - 7176.9 sample/sec

Global step: 23250 - [=====>----------] 46% - acc: 0.8828 - loss: 1.5769 - 7144.1 sample/sec

Global step: 23260 - [=====>----------] 49% - acc: 0.9219 - loss: 1.5383 - 7059.7 sample/sec

Global step: 23270 - [=====>----------] 51% - acc: 0.8984 - loss: 1.5618 - 6638.6 sample/sec

Global step: 23280 - [=====>----------] 54% - acc: 0.9453 - loss: 1.5151 - 7035.7 sample/sec

Global step: 23290 - [=====>----------] 56% - acc: 0.9609 - loss: 1.4996 - 7129.0 sample/sec

Global step: 23300 - [======>----------] 59% - acc: 0.9609 - loss: 1.4997 - 7075.4 sample/sec

Global step: 23310 - [======>----------] 61% - acc: 0.8750 - loss: 1.5842 - 7117.8 sample/sec

Global step: 23320 - [======>----------] 64% - acc: 0.9141 - loss: 1.5463 - 7157.2 sample/sec

Global step: 23330 - [======>----------] 66% - acc: 0.9062 - loss: 1.5549 - 7169.3 sample/sec

Global step: 23340 - [======>----------] 69% - acc: 0.9219 - loss: 1.5389 - 7164.4 sample/sec

Global step: 23350 - [======>----------] 72% - acc: 0.9609 - loss: 1.5002 - 7135.4 sample/sec

Global step: 23360 - [==========>--------] 74% - acc: 0.9766 - loss: 1.4842 - 7124.2 sample/sec

Global step: 23370 - [==========>-------] 77% - acc: 0.9375 - loss: 1.5231 - 7168.5 sample/sec

Global step: 23380 - [==========>-------] 79% - acc: 0.8906 - loss: 1.5695 - 7175.2 sample/sec

Global step: 23390 - [==========>------] 82% - acc: 0.9375 - loss: 1.5225 - 7132.1 sample/sec

Global step: 23400 - [==========>-----] 84% - acc: 0.9844 - loss: 1.4768 - 7100.1 sample/sec

Global step: 23410 - [==========>----] 87% - acc: 0.9766 - loss: 1.4840 - 7172.0 sample/sec

Global step: 23420 - [==========>---] 90% - acc: 0.9062 - loss: 1.5542 - 7122.1 sample/sec

Global step: 23430 - [==========>---] 92% - acc: 0.9297 - loss: 1.5313 - 7145.3 sample/sec

Global step: 23440 - [==========>--] 95% - acc: 0.9297 - loss: 1.5301 - 7133.3 sample/sec

Global step: 23450 - [==========>-] 97% - acc: 0.9375 - loss: 1.5231 - 7135.7 sample/sec

Global step: 23460 - [==========>] 100% - acc: 0.9250 - loss: 1.5362 - 10297.5 sample/sec

Epoch 60 - accuracy: 78.81% (7881/10000)

This epoch receive better accuracy: 78.81 > 78.78. Saving session...

##

MODEL QUESTION PAPERS

Model Question Paper - 1

B.E./B.Tech DEGREE EXAMINATION.,

Fourth Semester

CS3491 – ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

(Regulations 2021)

Time: Three Hours **Maximum: 100 Marks**

Answer ALL Questions

PART – A (10 × 2 = 20 Marks)

1. ***Is AI a science, or is it engineering? Or neither or both? Explain.***

Artificial Intelligence is most certainly a science. But it would be nothing with engineering. Computer Scientists need somewhere to place their programs, such as computers, servers, robots, cars, etc. But without engineers, they would have no outlet to test their Artificial Intelligence on. Science and Engineering go hand in hand, they both benefit each other. While the engineers build the machines, the scientists are writing code for their AI.

2. ***State the problem formulation for 8-puzzle problem.***

The problem formulation is as follows:

States: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.

Initial State: Any state can be designated as the initial state. It can be noted that any given goal can be reached from exactly half of the possible initial states.

Successor Function: This generates the legal states that result from trying the four actions (blank moves Left, Right, Vp, or down).

Goal Test: This checks whether the state matches the goal configuration

Path Cost: Each step costs 1, so the path cost is the number of steps in the path.

3. ***List few leading causes of uncertainty to occur in the real world.***

- Information occurred from unreliable sources.
- Experimental Errors

- ❖ Equipment fault
- ❖ Temperature variation
- ❖ Climate change.

4. ***Define Likelihood Weighting.***

Likelihood weighting avoids the inefficiency of rejection sampling by generating only events that are consistent with the evidence e. It is a particular instance of the general statistical technique of importance sampling, tailored for inference in Bayesian networks.

5. ***Differentiate supervised and unsupervised machine learning.***

Parameters	Supervised machine learning	Unsupervised machine learning
Input Data	Algorithms are trained using labeled data.	Algorithms are used against data that is not labeled
Computational Complexity	Simpler method	Computationally complex
Accuracy	Highly accurate	Less accurate
No. of classes	No. of classes is known	No. of classes is not known
Data Analysis	Uses offline analysis	Uses real-time analysis of data
Algorithms used	Linear and Logistics regression, Random Forest, Support Vector Machine, Neural Networks, etc.	K-Means clustering, Hierarchical clustering, Apriori algorithm, etc.

6. ***State Bayes theorem.***

Bayes' theorem is also known as Bayes' Rule or Bayes' law, which is used to determine the probability of a hypothesis with prior knowledge. It depends on the conditional probability.

The formula for Bayes' theorem is given as:

$$P(A \mid B) = \frac{P(B \mid A)\,P(A)}{P(B)}$$

7. ***What is boosting?***

Boosting is an ensemble learning technique that uses a set of Machine Learning algorithms to convert weak learner to strong learners in order to increase the accuracy of the model.

8. ***Mention some of the instance-based learning algorithms.***

1. K Nearest Neighbor (KNN)
2. Self-Organizing Map (SOM)
3. Learning Vector Quantization (LVQ)
4. Locally Weighted Learning (LWL)
5. Case-Based Reasoning

9. ***List the Limitations of Perceptron Model.***

A perceptron model has limitations as follows:

- ❖ The output of a perceptron can only be a binary number (0 or 1) due to the hard limit transfer function.
- ❖ Perceptron can only be used to classify the linearly separable sets of input vectors. If input vectors are non-linear, it is not easy to classify them properly.

10. ***Why do we need Backpropogation and how will you reduce the error?***

While designing a Neural Network, in the beginning, we initialize weights with some random values or any variable for that fact. So, it's not necessary that whatever weight values we have selected will be correct, or it fits our model the best. We have selected some weight values in the beginning, but our model output is way different than our actual output i.e. the error value is huge. Basically, what we need to do, we need to somehow explain the model to change the parameters (weights), such that error becomes minimum. Let's put it in an another way, we need to train our model. One way to train our model is called as Backpropagation.

PART B (5 x 13 = 65 Marks)

11. **(a)** ***Discuss briefly about types of agent.***

Ans: Refer Section No. 1.5 Page No. 1.25

[OR]

(b) Explain briefly about search algorithms.

Ans: Refer Section No.1.5 Page No. 1.25

12. ***(a) Write short notes on***

1. Inference by enumeration

2. Variable elimination algorithm

Ans: Refer Section No. 2.5 Page No. 2.15

[OR]

(b) Describe in detail about Bayesian network.

Ans: Refer Section No. 2.4 Page No. 2.7

13. ***(a) Discuss simple and multivariable linear regression.***

Ans: Refer Section No. 3.3 Page No. 3.15

[OR]

(b) Discuss about support vector machine and its types.

Ans: Refer Section No. 3.10 Page No. 3.56

14. ***(a) Discuss in detail about K-means algorithm.***

Ans: Refer Section No. 4.12 Page No. 4.47

[OR]

(b) Discuss briefly about Expectation maximization.

Ans: Refer Section No. 4.16 Page No. 4.89

15. ***(a) Discuss gradient descent optimization.***

Ans: Refer Section No. 5.5 Page No. 5.41

[OR]

(b) Explain the working principle of the ReLu activation function.

Ans: Refer Section No. 5.10 Page No. 5.73

PART C (1 × 15 = 15 Marks)

16. ***(a) Let's say we want to classify the different types of fruits in a bowl based on various features, but the bowl is cluttered with a lot of options. You would create a training dataset that contains information about the fruit,***

including colors, diameters, and specific labels (i.e., apple, grapes, etc.) You would then need to split the data by sorting out the smallest piece so that you can split it in the biggest way possible. You might want to start by splitting your fruits by diameter and then by color. You would want to keep splitting until that particular node no longer needs it, and you can predict a specific fruit with 100 percent accuracy. Implement the following using random forest algorithm.

(a) ***Implement data preprocessing, fitting for random forest classifier algorithm.***

(b) ***Create the confusion matrix, visualize the training set result, test set result by applying the random forest classifier algorithm.***

Ans: Refer Section No. 3.12.6 Page No. 3.73

[OR]

(b) ***Briefly explain the applications of AI.***

Ans: Refer Section No.1.4 Page No. 1.4

****

MODEL QUESTION PAPERS

Model Question Paper - 2

B.E./B.Tech DEGREE EXAMINATION.,

Fourth Semester

CS3491 – ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

(Regulations 2021)

Time: Three Hours **Maximum: 100 Marks**

Answer ALL Questions

PART – A (10 × 2 = 20 Marks)

1. ***What are the components of a problem?***

There are four components. They are

(*i*) Initial state

(*ii*) Successor function

(*iii*) Goal test

(*iv*) Path cost

(*v*) Operator

(*vi*) State-space

(*vii*) Path

2. ***Give examples of real-world end toy problems.***

Real World Problem

Examples: (*i*) Airline travel problem. (*ii*) Touring problem. (*iii*) Traveling salesman problem. (*iv*) VLSI Layout problem (*v*) Robot navigation (*vi*) Automatic Assembly (*vii*) Internet searching

Toy Problem

Examples:

Vacuum world problem.

8 – Queen problem

8 – Puzzle problem

3. ***What is a Bayesian network, a Real-World it important in AI?***

Bayesian networks are the graphical models that are used to show the probabilistic relationship between a set of variables. It is a directed cycle graph

that contains multiple edges, and each edge represents a conditional dependency. Bayesian networks are probabilistic, because these networks are built from a probability distribution, and also use probability theory for prediction and anomaly detection. It is important in AI as it is based on Bayes theorem and can be used to answer the probabilistic questions.

4. ***Define joint probability distribution.***

This completely specifies an agent's probability assignments to all propositions in the domain. The joint probability distribution $p(x_1, x_2, \ldots x_n)$ assigns probabilities to all possible atomic events; where $X_1, X_2 \ldots X_n$ 10 variables.

5. ***What is the difference between Gini Impurity and Entropy in a Decision Tree.***

- **Gini Impurity and Entropy** are the metrics used for deciding how to spli a Decision Tree.
- **Gini measurement** is the probability of a random sample being classifie correctly if you randomly pick a label according to the distribution in th branch. ϖ Entropy is a measurement to calculate the lack of information You calculate the Information Gain (difference in entropies) by making split. This measure helps to reduce the uncertainty about the output label.

6. ***What is the difference between Entropy and Information Gain?***

- Entropy is an indicator of how messy your data is. It decreases as yo reach closer to the leaf node.
- The Information Gain is based on the decrease in entropy after a dataset i split on an attribute. It keeps on increasing as you reach closer to the lea node.

7. ***Define Blending.***

It is a technique derived from Stacking Generalization. The only difference is that in Blending, the k-fold cross-validation technique is not used to generate the training data of the meta-model. Blending implements a "one-holdout set", that is, a small portion of the training data (validation) to make predictions that will be "stacked" to form the training data of the meta-model. Also, predictions are made from the test data to form the meta-model test data.

8. ***How boosting algorithm works?***

The basic principle behind the working of the boosting algorithm is to generate multiple weak learners and combine their predictions to form one strong rule. These weak rules are generated by applying base Machine Learning