UNIT – 3

Optimization Techniques in Data Science

Greedy Algorithms

A greedy algorithm builds a solution step by step, choosing the best local option at each step, hoping it leads to a global optimum.

Key Properties

- Greedy choice property – a locally optimal choice leads to a global solution
- Optimal substructure – optimal solution contains optimal sub-solutions

Greedy Algorithm for Feature Selection

Feature selection chooses a subset of relevant features to improve model performance and reduce complexity.

Greedy Idea

At each step, add (or remove) the feature that gives the maximum improvement in model performance.

Forward Selection (Greedy)

Start with no features, add one feature at a time.

Steps

1. Start with an empty feature set
2. Evaluate each remaining feature
3. Select the feature that improves accuracy the most
4. Repeat until no improvement

Example

Features = {Age, Income, Education, Gender}

Step	Selected Feature	Reason
1	Income	Highest accuracy gain
2	Education	Best improvement with Income

3	Stop	No further gain

Greedy decision: pick best feature at each step

Backward Elimination (Greedy)

Start with all features, remove the least useful one.

Used Metrics

- Accuracy
- Information Gain
- Mutual Information
- Correlation

Advantage

- Simple
- Fast for large datasets

Limitation: may miss best global feature combination

Greedy Algorithm for Job Scheduling

Job scheduling aims to maximize profit or minimize lateness.

Job Sequencing with Deadlines

Each job has:

- Deadline
- Profit
- Unit execution time

Greedy Strategy

Sort jobs by decreasing profit
Schedule each job as late as possible before its deadline

Example

Job	Deadline	Profit
J1	2	100

J2	1	19
J3	2	27
J4	1	25

Step 1: Sort by profit

J1 → J3 → J4 → J2

Step 2: Schedule

- J1 at slot 2
- J3 at slot 1
- J4 (no slot)
- J2

Maximum profit = 127

Why Greedy Works Here

- Higher profit jobs should be scheduled first
- Delaying jobs preserves future slots

Greedy Algorithms in Clustering Initialization

Clustering initialization affects speed and quality of clustering.

Greedy Initialization for k-Means

Instead of random centers, choose informative centroids greedily.

k-Means++ (Greedy-based)

This is a well-known greedy initialization method.

Steps

1. Choose first centroid randomly
2. Compute distance of all points to nearest centroid
3. Select next centroid farthest from existing centroids
4. Repeat until k centroids chosen

Greedy choice: pick point with maximum distance

Why It's Greedy

- Each step locally maximizes distance
- Reduces poor cluster initialization

Example

For k = 3:

1. Pick first centroid randomly
2. Pick second farthest from first
3. Pick third farthest from both

Leads to:

- Faster convergence
- Better cluster quality

Comparison Table

Application	Greedy Choice	Goal
Feature Selection	Best feature at each step	Maximize accuracy
Job Scheduling	Highest profit job first	Maximize profit
Clustering Init	Farthest data point	Improve clustering

Advantages & Limitations

Advantages

- Simple
- Fast
- Easy to implement

Limitations

- No guarantee of global optimum
- Sensitive to initial choices

Dynamic Programming (DP)

Definition

Dynamic Programming is an algorithmic technique used to solve complex problems by:

- Breaking them into overlapping subproblems
- Solving each subproblem once
- Storing results to avoid recomputation

Key Properties

1. **Optimal substructure** – optimal solution depends on optimal sub-solutions
2. **Overlapping subproblems** – same subproblems appear repeatedly

Dynamic Programming in Sequence Prediction

What is Sequence Prediction?

Predicting the next element in a sequence using previous values.
Examples:

- Stock prices
- Weather data
- Sensor readings

How DP is Used

DP stores results of previous predictions and reuses them.

Example

Fibonacci-based sequence:

$$F(n) = F(n-1) + F(n-2)$$

Without DP → exponential time
With DP → linear time using a table

DP Approach

1. Define state: dp[t] = best prediction up to time t
2. Define recurrence relation

3. Store computed values

Used in:

- Hidden Markov Models (HMMs)
- Dynamic Time Warping (DTW)
- Sequential decision models

Dynamic Programming in Time Series Analysis

Time Series Analysis

Analysis of data points collected over time intervals.

DP Applications

1. Optimal Segmentation

Divide time series into meaningful segments.

$$dp[t] = \min_{k<t}(dp[k] + cost(k, t))$$

2. Change Point Detection

Detect when statistical properties change.

3. Dynamic Time Warping (DTW)

Align two time series with different speeds.

Why DP is Suitable

- Time dependencies
- Repeated sub-calculations
- Optimal decisions depend on past values

Dynamic Programming in NLP

NLP Tasks Using DP

Dynamic programming is core to NLP algorithms.

Sequence Tagging

Examples:

- Part-of-Speech (POS) tagging
- Named Entity Recognition (NER)

Viterbi Algorithm (DP-based)

Finds the most probable sequence of tags.

$$dp[t][s] = \max(dp[t-1][s'] \times P(s \mid s') \times P(w \mid s))$$

String Similarity

- Edit Distance (Levenshtein Distance)
- Spell checking
- DNA sequence comparison

Edit Distance DP Table

$$dp[i][j] = \min \begin{cases} dp[i-1][j] + 1 \\ dp[i][j-1] + 1 \\ dp[i-1][j-1] + cost \end{cases}$$

Parsing & Language Modeling

- CKY parsing
- Word segmentation
- Text summarization

Advantages of DP in Data Science

- Efficient for sequential problems
- Ensures optimal solution
- Reduces time complexity drastically

Divide & Conquer (D&C)

Definition

Divide & Conquer solves a problem by:

1. **Dividing** it into smaller subproblems
2. **Conquering** (solving) each subproblem independently
3. **Combining** the solutions

Divide & Conquer for Data Partitioning

Data Partitioning

Splitting large datasets into manageable chunks.

Examples

- Divide dataset into subsets
- Process each subset independently
- Merge results

Common Algorithms

Algorithm	Role
Merge Sort	Data partitioning + merging
Quick Sort	Partition-based processing
KD-Trees	Multidimensional data partitioning

Benefit

- Reduces memory usage
- Improves cache performance

Divide & Conquer for Parallel Computation

Why D&C Works Well with Parallelism

- Subproblems are independent

- Can be executed simultaneously

Examples in Data Science

1. Parallel Sorting

- Each processor sorts a subset
- Results merged later

2. MapReduce

- **Map** = divide
- **Reduce** = combine

3. Parallel Matrix Multiplication

Split matrices into blocks and compute in parallel.

Example Flow

1. Split dataset into n parts
2. Process each part on different processors
3. Merge partial results

9. Dynamic Programming vs Divide & Conquer

Aspect	Dynamic Programming	Divide & Conquer
Subproblems	Overlapping	Independent
Storage	Uses memoization	Usually no storage
Best For	Sequential problems	Parallel tasks
Examples	NLP, time series	Sorting, MapReduce

Application

1.Shortest Sequence Alignment (Dynamic Programming)

What is Sequence Alignment?

Sequence alignment is the process of arranging two sequences (strings, DNA, words) to identify similarities by allowing:

- Matches
- Mismatches
- Gaps (insertions/deletions)

Goal: Find the alignment with minimum cost (or maximum similarity).
This is also called minimum edit distance or optimal alignment.

Why Dynamic Programming?

- Alignment decisions depend on previous characters
- Same sub-alignments are computed repeatedly
- DP ensures optimal global alignment

Shortest (Minimum-Cost) Sequence Alignment Problem

Given two sequences:

- $X = x_1 x_2 \dots x_m$
- $Y = y_1 y_2 \dots y_n$

Define costs:

- Match = 0
- Mismatch = 1
- Gap (insertion/deletion) = 1

DP Formulation

State Definition

$$dp[i][j] = \text{minimum cost to align } X[1..i] \text{ and } Y[1..j]$$

Recurrence Relation

$$dp[i][j] = \min \begin{cases} dp[i-1][j] + 1 & \text{(deletion)} \\ dp[i][j-1] + 1 & \text{(insertion)} \\ dp[i-1][j-1] + cost & \text{(match/mismatch)} \end{cases}$$

Where:

- cost = 0 if $x_i = y_j$
- cost = 1 if $x_i \neq y_j$

Base Conditions

- $dp[0][j] = j$(insert all characters)
- $dp[i][0] = i$(delete all characters)

Example

Align:

- **X = "CAT"**
- **Y = "CUT"**

Final DP value:

$$dp[3][3] = 1$$

Only one substitution (A → U) needed

Shortest sequence alignment achieved

Applications

- Bioinformatics (DNA / protein alignment)
- Spell checking
- Plagiarism detection
- NLP string similarity
- Speech recognition

2. Optimal Substructure Identification

What is Optimal Substructure?

A problem has optimal substructure if:

The optimal solution of the problem contains optimal solutions to its subproblems.

This property is mandatory for applying Dynamic Programming.

Identifying Optimal Substructure

Steps

1. Define the global problem
2. Break it into smaller subproblems
3. Show that solving subproblems optimally leads to a global optimum

Optimal Substructure in Sequence Alignment

Explanation

If the best alignment of:

- $X[1..i]$and $Y[1..j]$

ends with:

- a match, insertion, or deletion

then the rest of the alignment must be the best possible alignment of:

- $X[1..i-1]$and $Y[1..j-1]$
- or $X[1..i-1]$and $Y[1..j]$
- or $X[1..i]$and $Y[1..j-1]$

Otherwise, the solution would not be optimal.

Hence, sequence alignment satisfies optimal substructure

Other Examples of Optimal Substructure

Problem	**Substructure**
Shortest Path	Shortest subpaths
Knapsack	Optimal weight combinations

Edit Distance	Optimal prefix alignment
Matrix Chain Multiplication	Optimal matrix grouping

Relationship Between Both Concepts

Concept	**Role**
Optimal Substructure	Justifies DP use
Sequence Alignment	Practical DP application
DP Table	Stores optimal sub-solutions
Recurrence Relation	Exploits substructure