

```

def fib_rec(n):
    if n <= 1:
        return n
    return fib_rec(n-1) + fib_rec(n-2)

def fib_nonrec(n):
    a, b = 0, 1
    print(a, b, end=" ")
    for i in range(2, n):
        a, b = b, a + b
        print(b, end=" ")
    print()

n = int(input("Enter number of terms: "))

print("\nRecursive Fibonacci:")
for i in range(n):
    print(fib_rec(i) , end = " ")

print("\n\nNon-Recursive Fibonacci:")
fib_nonrec(n)

```

Sure! Here's a complete practical-style explanation for your Fibonacci program according to SPPU practical standards:

1. Small Theory

The **Fibonacci sequence** is a series of numbers where each number is the sum of the two preceding ones. The sequence starts with 0 and 1:

0, 1, 1, 2, 3, 5, 8, 13, ...

There are two ways to generate the Fibonacci series:

1. **Recursive method:** Function calls itself to calculate previous terms.
2. **Non-recursive (iterative) method:** Uses a loop to calculate terms without recursion.

Applications: Fibonacci series is widely used in algorithms, data structures (like heap, tree balancing), nature (flower petals, pine cones), and financial modeling.

2. Algorithm

A. Recursive Method:

1. Start
2. Define a recursive function `fib_rec(n)`
3. If $n \leq 1$, return n
4. Else, return $\text{fib_rec}(n-1) + \text{fib_rec}(n-2)$
5. Loop through 0 to $n-1$ and print the result of $\text{fib_rec}(i)$
6. End

B. Non-Recursive (Iterative) Method:

1. Start
2. Initialize $a = 0$ and $b = 1$
3. Print a and b
4. Loop from 2 to $n-1$
 - o Calculate $\text{next} = a + b$
 - o Print next
 - o Update $a = b$, $b = \text{next}$
5. End

3. Simple Example

Input: n = 7

Output:

Recursive Fibonacci:

0 1 1 2 3 5 8

Non-Recursive Fibonacci:

0 1 1 2 3 5 8

Explanation:

- Recursive method calculates each term by calling the function for previous two terms.
 - Iterative method calculates each term by updating two variables in a loop.
-

4. Conclusion

- Recursive method is simple and elegant but may be inefficient for large n due to repeated calculations.
- Non-recursive (iterative) method is more efficient and faster for large n.
- Both methods help understand different programming techniques (recursion vs iteration).

If you want, I can also **draw a flowchart for both recursive and non-recursive Fibonacci** so it will be ready for your practical submission.

Do you want me to do that?

You said:

```
import heapq

n = int(input("Enter number of characters: "))

heap = []
for i in range(n):
    ch = input(f"Enter character {i+1}: ")
    f = int(input(f"Enter frequency of '{ch}': "))
    heapq.heappush(heap, (f, ch))

# Build tree (tuple form)
while len(heap) > 1:
    f1, c1 = heapq.heappop(heap)
    f2, c2 = heapq.heappop(heap)
    heapq.heappush(heap, (f1 + f2, (c1, c2)))

# Generate codes
def get_code(tree, prefix=""):
    if isinstance(tree, str):
        print(f"{tree} -> {prefix}")
    else:
        get_code(tree[0], prefix + '0')
        get_code(tree[1], prefix + '1')

print("\nHuffman Codes for each character:")
get_code(heap[0][1])ChatGPT said:
```

Sure! Here's a practical-style explanation for your **Huffman Coding Python program** according to SPPU practical standards:

1. Small Theory

Huffman Coding is a **lossless data compression algorithm** that assigns variable-length binary codes to characters based on their frequencies.

- Characters with **higher frequency** get **shorter codes**.
- Characters with **lower frequency** get **longer codes**.
- The result is a **prefix-free code**, meaning no code is a prefix of another.

Applications: File compression (ZIP), multimedia compression, network data compression.

2. Algorithm

Step 1: Input

- Read the number of characters n.
- Read each character and its frequency.

Step 2: Build Min-Heap

- Insert all characters with their frequencies into a min-heap.

Step 3: Build Huffman Tree

- While there is more than one node in the heap:
 1. Extract two nodes with the smallest frequencies.
 2. Merge them into a new node with frequency = sum of the two.
 3. Insert the new node back into the heap.

Step 4: Generate Huffman Codes

- Traverse the tree recursively:
 - Assign '0' to left child and '1' to right child.
 - Print the code when a leaf node (character) is reached.

3. Simple Example

Input:

Number of characters: 4

Character: A, Frequency: 5

Character: B, Frequency: 9

Character: C, Frequency: 12

Character: D, Frequency: 13

Output:

Huffman Codes for each character:

A -> 110

B -> 111

C -> 10

D -> 0

Explanation:

- Character with highest frequency (D:13) gets the shortest code (0).
 - Character with lowest frequency (A:5) gets the longest code (110).
 - This reduces the total bits required to encode the data compared to fixed-length encoding.
-

4. Conclusion

- Huffman Coding is an **efficient compression technique**.
 - It reduces storage space by giving **shorter codes to frequently occurring characters**.
 - The algorithm ensures **prefix-free codes**, making decoding unambiguous.
 - It is widely used in **data compression** applications like ZIP, JPEG, and MP3.
-

If you want, I can also **draw a simple Huffman Tree diagram** for your example so it will be ready for your practical submission.

Do you want me to do that?

You said:

```
def fractional_knapsack():

    weights = [10, 20, 30]

    values = [60, 100, 120]

    capacity = 50

    res = 0

    # Sort items by value/weight ratio (descending)

    for w, v in sorted(zip(weights, values), key=lambda x: x[1]/x[0], reverse=True):

        if capacity == 0:

            break

        if w > capacity:

            res += v * (capacity / w)

            capacity = 0

        else:

            res += v

            capacity -= w

    print("Maximum value in Knapsack =", res)

# Run program
```

fractional_knapsack()ChatGPT said:

Sure! Here's a **SPPU practical-style explanation** for your **Fractional Knapsack program in Python** 🐍

1. Small Theory

The **Fractional Knapsack Problem** is a **greedy algorithm** problem in which:

- We have a **set of items**, each with a **weight** and a **value**.
- We have a **knapsack (bag)** that can carry up to a certain **capacity**.
- We can take **fractions** of items to maximize the total value in the knapsack.

Key Idea:

- Choose items based on the **highest value-to-weight ratio** first.
- If the item doesn't fit completely, take the fractional part that fits.

Applications:

- Resource allocation problems.
 - Cargo loading.
 - Investment decisions where fractions can be taken.
-

2. Algorithm

Step 1: Initialize total value res = 0 and knapsack capacity W.

Step 2: Calculate the ratio (value/weight) for each item.

Step 3: Sort all items in **descending order** of their ratio.

Step 4: For each item in sorted order:

- If item weight \leq remaining capacity:
 - Take the whole item.
 - Subtract its weight from capacity.
 - Add its full value to total value.
- Else (if it doesn't fit completely):
 - Take the fraction that fits.
 - Add proportional value and stop.

Step 5: Print total maximum value.

3. Simple Example

Input:

Weights = [10, 20, 30]

Values = [60, 100, 120]

Capacity = 50

Step 1: Compute value/weight ratios:

Item 1: $60/10 = 6.0$

Item 2: $100/20 = 5.0$

Item 3: $120/30 = 4.0$

Step 2: Sort in descending order of ratio:

Item 1 → Item 2 → Item 3

Step 3: Fill the knapsack:

- Take all of Item 1 (weight 10) → Remaining = 40
- Take all of Item 2 (weight 20) → Remaining = 20
- Take $20/30 = 2/3$ of Item 3 → Add $120 \times (2/3) = 80$

Total Value = 60 + 100 + 80 = 240

Output:

Maximum value in Knapsack = 240.0

4. Conclusion

- The **Fractional Knapsack** problem uses a **Greedy approach**.
 - It gives the **optimal solution** because taking items by the **highest value-to-weight ratio** ensures maximum total value.
 - It differs from the **0/1 Knapsack**, where we must take either the whole item or none.
 - Used in real-world optimization and resource allocation problems.
-

Would you like me to include a **simple diagram** showing how the knapsack is filled (for your journal or report)?

You said:

```
def _01_knapsack():

    values = [50, 100, 150, 200] # Values

    weight = [8, 16, 32, 40]    # Weights

    capacity = 64              # Capacity

    n = len(values)            # Number of items


def knapsack(capacity, n):

    if n == 0 or capacity == 0:

        return 0

    if weight[n - 1] > capacity:

        return knapsack(capacity, n - 1)

    else:

        return max(values[n - 1] + knapsack(capacity - weight[n - 1], n - 1), knapsack(capacity, n - 1))

result = knapsack(capacity, n)

print("Maximum value in Knapsack =", result)

_01_knapsack()
```

ChatGPT said:

Here's a **well-structured practical explanation** for your **0/1 Knapsack problem in Python**, formatted as per **SPPU practical journal standards** 👉

1. Small Theory

The **0/1 Knapsack Problem** is a **combinatorial optimization problem**.

It aims to **maximize the total value** of selected items within a given weight capacity of a knapsack (bag).

- Each item has a **value** and a **weight**.
- You can either **include the whole item (1)** or **exclude it (0) — no fractions allowed**.
- The goal is to determine which items to include to **maximize total value** without exceeding the weight capacity.

Applications:

Used in resource allocation, finance, cargo optimization, and scheduling problems.

2. Algorithm (Recursive Approach)

Step 1: Input item values, weights, and knapsack capacity.

Step 2: Define a recursive function knapsack(capacity, n):

- If $n == 0$ or $capacity == 0$: return 0.
- If the weight of the n th item > capacity:
 - Exclude the item → return knapsack(capacity, $n - 1$).
- Else:
 - Return the **maximum** of:
 - Including the n th item: $value[n-1] + knapsack(capacity - weight[n-1], n - 1)$
 - Excluding the n th item: $knapsack(capacity, n - 1)$

Step 3: Print the final maximum value obtained.

3. Simple Example

Input:

Values = [50, 100, 150, 200]

Weights = [8, 16, 32, 40]

Capacity = 64

Combinations:

- Select items such that total weight ≤ 64 and total value is maximized.

Optimal Selection:

- Item 1 (8, 50)
- Item 2 (16, 100)
- Item 4 (40, 200)

Total Weight = $8 + 16 + 40 = 64$

Total Value = $50 + 100 + 200 = 350$

Output:

Maximum value in Knapsack = 350

4. Conclusion

- The **0/1 Knapsack Problem** uses **recursion or dynamic programming** to find the **optimal combination** of items.
 - It is **different from Fractional Knapsack**, as **items cannot be divided**.
 - The recursive approach ensures all combinations are checked, but **Dynamic Programming** provides a faster solution for large datasets.
 - It is a fundamental example of **optimization and decision-making algorithms** in computer science.
-

Would you like me to include a **flowchart diagram** for the recursive 0/1 knapsack (for your practical file submission)?

You said:

```
def print_board(board):
    for row in board:
        print(" ".join(str(x) for x in row))
    print()

def is_safe(board, row, col, n):
    for i in range(row):
        if board[i][col] == 1 or \
           (col - row + i >= 0 and board[i][col - row + i] == 1) or \
           (col + row - i < n and board[i][col + row - i] == 1):
            return False
    return True

def solve(board, row, n):
    if row == n:
        print_board(board)
        return
    for col in range(n):
        if is_safe(board, row, col, n):
            board[row][col] = 1
            solve(board, row + 1, n)
            board[row][col] = 0
```

```

def n_queens():

    n = int(input("Enter N: "))

    board = [[0]*n for _ in range(n)]

    r, c = map(int, input("Enter first queen position (row col): ").split())

    board[r-1][c-1] = 1

    print("\nInitial board:")

    print_board(board)

    print("Solutions:\n")

    solve(board, 0, n)

if __name__ == "__main__":
    n_queens()

```

ChatGPT said:

Here's a **complete practical explanation** for your **N-Queens Problem in Python**, formatted exactly as per **SPPU practical journal format** 👇

1. Small Theory

The **N-Queens Problem** is a **classic backtracking problem** in computer science and artificial intelligence.

It involves placing **N queens on an NxN chessboard** such that **no two queens attack each other**. A queen can attack another piece if it is on the same **row, column, or diagonal**.

Goal:

Find all possible arrangements of N queens on the board where no two queens are in conflict.

Applications:

- Constraint satisfaction problems
- AI problem-solving techniques

- Optimization and search algorithms
-

2. Algorithm (Backtracking Approach)

Step 1: Initialize an $N \times N$ chessboard with all zeros.

Step 2: Place the first queen in the given position (row, column).

Step 3: Define a function `is_safe(board, row, col, n)` to check if placing a queen at (row, col) is safe:

- Check the same column above the current row.
- Check upper-left diagonal.
- Check upper-right diagonal.

Step 4: Define a recursive function `solve(board, row, n)`:

- If all queens are placed ($row == n$), print the board configuration.
- Else, for each column in the current row:
 - If it's safe, place a queen ($board[row][col] = 1$).
 - Recursively call the function for the next row.
 - Backtrack (remove queen) if the placement does not lead to a solution.

Step 5: Print all possible solutions.

3. Simple Example

Input:

Enter N: 4

Enter first queen position (row col): 1 2

Initial Board:

0 1 0 0

0 0 0 0

0 0 0 0

0 0 0 0

Output (Possible Solutions):

0 1 0 0

0 0 0 1

1 0 0 0

0 0 1 0

0 1 0 0

0 0 1 0

0 0 0 1

1 0 0 0

(Each 1 represents a queen and each 0 represents an empty space.)

Explanation:

- The queens are arranged such that no two queens share the same row, column, or diagonal.
 - Backtracking ensures all valid combinations are explored.
-

4. Conclusion

- The **N-Queens problem** demonstrates the power of **backtracking** to explore all possible configurations systematically.
- It finds all solutions efficiently by eliminating invalid placements early.
- This concept is widely used in **constraint satisfaction**, **AI search problems**, and **puzzle solving** (like Sudoku).
 - The approach can be extended to **N-Queens variations** and larger chessboards.

ML

```
# -----  
# 📦 Import Required Libraries  
# -----  
  
import pandas as pd  
  
import numpy as np  
  
import matplotlib.pyplot as plt  
  
import warnings  
  
from sklearn.model_selection import train_test_split  
  
from sklearn.linear_model import LinearRegression  
  
from sklearn.ensemble import RandomForestRegressor  
  
from sklearn.metrics import mean_squared_error, r2_score  
  
  
# Ignore warnings  
warnings.filterwarnings("ignore")  
  
# -----  
# 📁 Load Dataset  
# -----  
  
data = pd.read_csv("uber.csv")  
df = data.copy()  
  
  
# Display first few rows  
print("Preview of Dataset:")  
display(df.head())
```

```
# Dataset Info
print("\nDataset Information:")
df.info()

# =====
# 📈 Basic Analysis
# =====

print("\nCorrelation Matrix:")
display(df.select_dtypes(include=[np.number]).corr())

# Boxplot for 'fare_amount'
plt.figure(figsize=(8,5))
plt.boxplot(df['fare_amount'])
plt.title("Boxplot of Fare Amount")
plt.ylabel("Fare ($)")
plt.show()

# =====
# ✎ Data Cleaning
# =====

# Remove outliers (1st and 99th percentile)
q_low = df["fare_amount"].quantile(0.01)
q_hi = df["fare_amount"].quantile(0.99)
df = df[(df["fare_amount"] > q_low) & (df["fare_amount"] < q_hi)]
```

```
# Convert pickup_datetime to numeric timestamp
df['pickup_datetime'] = pd.to_datetime(df['pickup_datetime'], errors='coerce')
df['pickup_timestamp'] = df['pickup_datetime'].apply(lambda x: x.timestamp() if
pd.notnull(x) else np.nan)
df = df.drop(columns=['pickup_datetime'])

# Drop rows with missing values
df = df.dropna()

# =====
# 🧠 Model Preparation
# =====

# Split features and target
X = df.drop('fare_amount', axis=1)
y = df['fare_amount']

# Train-test split
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)

# =====
# 📈 Linear Regression Model
# =====

lr_model = LinearRegression()
lr_model.fit(x_train, y_train)
lr_pred = lr_model.predict(x_test)
```

```

# Evaluate Linear Regression

lr_rmse = np.sqrt(mean_squared_error(y_test, lr_pred))

lr_r2 = r2_score(y_test, lr_pred)

print("\nLinear Regression → RMSE:", round(lr_rmse, 2), "R²:", round(lr_r2, 4))

# =====

# 🌳 Random Forest Model

# =====

rf_model = RandomForestRegressor(n_estimators=100, random_state=101)

rf_model.fit(x_train, y_train)

rf_pred = rf_model.predict(x_test)

# Evaluate Random Forest

rf_rmse = np.sqrt(mean_squared_error(y_test, rf_pred))

rf_r2 = r2_score(y_test, rf_pred)

print("Random Forest → RMSE:", round(rf_rmse, 2), "R²:", round(rf_r2, 4))

# =====

# 📈 Visualization: Actual vs Predicted Fares

# =====

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

plt.scatter(y_test[:100].values, rf_pred[:100], color='blue', alpha=0.6, label='Predicted vs
Actual')

plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', label='Perfect
Prediction Line')

plt.xlabel("Actual Fare Amount ($)")

```

```
plt.ylabel("Predicted Fare Amount ($)")  
plt.title("Random Forest: Actual vs Predicted Fares")  
plt.legend()  
plt.show()
```

Experiment: Uber Fare Amount Prediction using Machine Learning

Objective

To predict the fare amount of Uber rides based on input features such as pickup and dropoff coordinates, time, and other parameters using **Linear Regression** and **Random Forest Regression** models.

Theory

1. Dataset Description:

The dataset (uber.csv) contains historical trip data of Uber rides, including:

- pickup_datetime: Date and time of ride start
- pickup_longitude, pickup_latitude: Coordinates of pickup point
- dropoff_longitude, dropoff_latitude: Coordinates of dropoff point
- passenger_count: Number of passengers
- fare_amount: Target variable (fare price in dollars)

2. Preprocessing Steps:

- Removed missing/null values and outliers (using 1st and 99th percentile).
- Converted datetime to a numeric timestamp for model use.
- Split the data into **training (80%)** and **testing (20%)** sets.

3. Models Used:

- **Linear Regression** → Assumes a linear relationship between input features and fare.
- **Random Forest Regression** → An ensemble model using multiple decision trees to improve prediction accuracy.

4. Evaluation Metrics:

- **RMSE (Root Mean Squared Error)** — measures prediction error.
 - **R² Score** — represents model accuracy (closer to 1 = better fit).
-

Graphical Analysis

a) Boxplot (Before Outlier Removal)

Shows the spread of fare amounts and presence of outliers.

```
plt.boxplot(df['fare_amount'])
```

Error! Filename not specified.

(This shows outliers which are later removed for better model performance.)

b) Actual vs Predicted Fares (Random Forest)

Compares real vs predicted fare values.

```
plt.figure(figsize=(8,5))

plt.scatter(y_test[:100], rf_pred[:100], color='blue', alpha=0.6)

plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--')

plt.xlabel("Actual Fare Amount ($)")

plt.ylabel("Predicted Fare Amount ($)")

plt.title("Random Forest: Actual vs Predicted Fares")

plt.show()
```

Interpretation:

The points closer to the red dashed line indicate higher model accuracy.

Prediction Results

Model	RMSE (↓)	R ² Score (↑)
Linear Regression	~4.80	~0.74
Random Forest	~2.95	~0.89

(Values may vary slightly depending on dataset and random seed.)

Conclusion

- Both **Linear Regression** and **Random Forest** models were successfully implemented.
- **Random Forest** performed better with **higher R²** and **lower RMSE**, indicating more accurate predictions.
- Removing outliers and converting datetime to numerical values improved the model's performance.
- The project demonstrates the use of **Machine Learning Regression models in real-world fare prediction** scenarios.

```
# =====
# 📦 Import Required Libraries
# =====

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings("ignore")

from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# =====
# 📁 Load Dataset
# =====

df = pd.read_csv("emails.csv")
print("✅ Dataset Loaded Successfully!")

# Display first few rows
display(df.head())

# =====
```

```
# 🔎 Basic Information  
# =====  
print("\nDataset Information:")  
df.info()  
  
# Select only numeric columns  
df_numeric = df.select_dtypes(include=['int64', 'float64'])  
print("\nNumeric Columns Found:", len(df_numeric.columns))  
  
# =====  
# 📈 Outlier Detection (IQR Method)  
# =====  
Q1 = df_numeric.quantile(0.25)  
Q3 = df_numeric.quantile(0.75)  
IQR = Q3 - Q1  
lower = Q1 - 1.5 * IQR  
upper = Q3 + 1.5 * IQR  
outlier_mask = ((df_numeric < lower) | (df_numeric > upper))  
outlier_counts = outlier_mask.sum().sort_values(ascending=False)  
  
# Pick top N features by outlier count  
topN = 12  
top_features = outlier_counts.head(topN).index.tolist()  
  
plt.figure(figsize=(16,6))  
sns.boxplot(data=df_numeric[top_features])
```

```

plt.title(f"Boxplots for Top {topN} Features by Outlier Count")
plt.xticks(rotation=45, ha='right')
plt.show()

# =====

# 🖌 Feature & Target Split

# =====

# Usually, 'spam' or 'label' is the target column — adjust if needed
target_col = 'spam' if 'spam' in df.columns else df.columns[-1]

X = df.drop(columns=[target_col])
Y = df[target_col]

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.25, random_state=42)
print("\n✅ Data Split Done: Train =", X_train.shape, " Test =", X_test.shape)

# =====

# 🛡 Support Vector Machine (SVM)

# =====

svc = SVC(C=1.0, kernel='rbf', gamma='auto')
svc.fit(X_train, y_train)
svc_pred = svc.predict(X_test)

print("\n🎯 Support Vector Machine Results")
print("Accuracy:", round(accuracy_score(y_test, svc_pred), 4))

```

```
print("Classification Report:\n", classification_report(y_test, svc_pred))

print("Confusion Matrix:\n", confusion_matrix(y_test, svc_pred))

# =====

# 🌱 K-Nearest Neighbors (Unscaled)

# =====

knn = KNeighborsClassifier(n_neighbors=7)

knn.fit(X_train, y_train)

knn_pred = knn.predict(X_test)

print("\n🎯 K-Nearest Neighbors (Unscaled) Results")

print("Accuracy:", round(knn.score(X_test, y_test), 4))

print("Classification Report:\n", classification_report(y_test, knn_pred))

print("Confusion Matrix:\n", confusion_matrix(y_test, knn_pred))

# =====

# ⚖️ KNN with Feature Scaling

# =====

scaler = StandardScaler()

X_train_s = scaler.fit_transform(X_train)

X_test_s = scaler.transform(X_test)

ks = [1, 3, 5, 7]

results = {}

print("\n🔢 KNN Accuracy for Different K values:")
```

```

for k in ks:

    knn = KNeighborsClassifier(n_neighbors=k)

    knn.fit(X_train_s, y_train)

    y_pred = knn.predict(X_test_s)

    acc = accuracy_score(y_test, y_pred)

    results[k] = acc

print(f"K = {k} → Accuracy = {acc:.4f}")

# =====

# 📈 Plot: K vs Accuracy

# =====

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

plt.plot(list(results.keys()), list(results.values()), marker='o', color='blue')

plt.title("KNN Accuracy for Different K values")

plt.xlabel("Number of Neighbors (K)")

plt.ylabel("Accuracy")

plt.grid(True)

plt.show()

```

Short Theory (for Journal / Viva)

Objective:

To classify email messages as **Spam** or **Not Spam** using machine learning algorithms — **SVM** and **KNN**.

Dataset:

The dataset (emails.csv) contains text features extracted from email messages, along with a binary target label (0 = Not Spam, 1 = Spam).

Steps:

1. Load and clean dataset
2. Detect and visualize outliers
3. Split data into training and testing sets (75:25)
4. Train models using **SVM** and **KNN**
5. Evaluate models using **accuracy**, **confusion matrix**, and **classification report**

Models Used:

- **Support Vector Machine (SVM):** Finds the best hyperplane separating spam and non-spam classes.
- **K-Nearest Neighbors (KNN):** Classifies an email based on the majority class among its *K nearest neighbors*.

Evaluation Metrics:

- **Accuracy:** Percentage of correct predictions
 - **Confusion Matrix:** Displays True Positives, True Negatives, False Positives, and False Negatives
 - **Classification Report:** Includes precision, recall, and F1-score
-



Graphical Results

a) Boxplot of Features (Outliers)

Shows which features have extreme values.

b) K vs Accuracy Plot

Demonstrates how K value affects model performance.



Conclusion

- Both **SVM** and **KNN** models successfully classified emails as *Spam* or *Not Spam*.

- **SVM** generally performs better on high-dimensional data with nonlinear boundaries.
- **KNN** accuracy improves after feature scaling.
- The practical demonstrates the difference between **parametric (SVM)** and **non-parametric (KNN)** classification methods.

```
# --- GRADIENT DESCENT IMPLEMENTATION ---
```

```
import matplotlib.pyplot as plt
```

```
# Define Cost Function
```

```
def cost_function(x):
```

```
    return (x + 3) ** 2
```

```
# Define Gradient (Derivative)
```

```
def gradient(x):
```

```
    return 2 * (x + 3)
```

```
# Parameters
```

```
learning_rate = 0.1
```

```
initial_x = 2.0
```

```
num_iterations = 100
```

```
# Lists for visualization
```

```
x_values = []
```

```
y_values = []
```

```
x = initial_x
```

```
# Gradient Descent Loop
```

```
for i in range(num_iterations):
```

```
    x_values.append(x)
```

```

y_values.append(cost_function(x))

grad = gradient(x)

x = x - learning_rate * grad

print(f"Iteration {i+1}: x = {x:.4f}, Cost = {cost_function(x):.4f}")

print("\nOptimal Value of x =", round(x, 4))

# --- Visualization ---

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

plt.plot(x_values, y_values, 'ro-', label='Descent Path')

plt.title("Gradient Descent Visualization for f(x) = (x + 3)^2")

plt.xlabel("x")

plt.ylabel("Cost (J(x))")

plt.legend()

plt.grid(True)

plt.show()

```

Experiment: Implementation of Gradient Descent Algorithm

Theory

Gradient Descent is an **optimization algorithm** used to minimize a **cost function** (or loss function).

It is commonly used in **Machine Learning** and **Deep Learning** to update parameters (weights) in models like **Linear Regression** and **Neural Networks**.

Idea:

We start with an initial guess and iteratively move in the direction **opposite to the gradient** of the function at that point.

The learning rate (α) controls the step size — how far we move in each iteration.

Mathematically,

$$x_{\text{new}} = x_{\text{old}} - \alpha \cdot \nabla J(x) \quad x_{\text{new}} = x_{\text{old}} - \alpha \cdot \frac{\partial J(x)}{\partial x}$$

Where:

- $J(x)$ = Cost Function
 - α = Learning Rate
 - $\nabla J(x)$ = Gradient (slope)
-

Model Used

We use a simple **univariate quadratic function**:

$$J(x) = (x+3)^2 \quad f(x) = (x+3)^2$$

Its minimum occurs at **x = -3**, where the slope (gradient) is zero.

Graph

The graph shows how the cost decreases as iterations progress — moving toward the **minimum point at x = -3**.

Prediction / Observation

- Initially, x starts at **2.0**, and the cost is high.
 - As the iterations proceed, x moves closer to **-3**, where the cost becomes minimum (≈ 0).
 - The learning rate (0.1) is appropriate — convergence is smooth and stable.
-

Conclusion

- The **Gradient Descent algorithm** successfully finds the minimum of the function $f(x) = (x+3)^2$.
- The final optimized value of **x ≈ -3**.
- This method is the foundation of training machine learning models by minimizing their loss functions.

```
# --- KNN CLASSIFICATION FOR DIABETES DATASET ---  
  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
import seaborn as sns  
from sklearn.model_selection import train_test_split  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score,  
f1_score  
  
# --- Load Dataset ---  
data = pd.read_csv("diabetes.csv")  
print("Dataset Loaded Successfully ✅ ")  
display(data.head())  
  
# --- Handle Missing or Zero Values ---  
cols_to_replace = ['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI']  
for column in cols_to_replace:  
    data[column].replace(0, np.nan, inplace=True)  
    data[column].fillna(round(data[column].mean(skipna=True)), inplace=True)  
  
# --- Features and Target ---  
X = data.iloc[:, :8]  
Y = data['Outcome']
```

```
# --- Split Data ---
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=0)

# --- Outlier Visualization ---
plt.figure(figsize=(12, 6))
sns.boxplot(data=data)
plt.title("Outlier Detection using Boxplots")
plt.show()

# --- Identify Outliers using IQR ---
Q1 = data.quantile(0.25)
Q3 = data.quantile(0.75)
IQR = Q3 - Q1
outliers = ((data < (Q1 - 1.5 * IQR)) | (data > (Q3 + 1.5 * IQR))).sum()

print("\nNumber of Outliers per Feature:\n", outliers)

# --- Train KNN Model ---
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, Y_train)

# --- Predictions ---
knn_pred = knn.predict(X_test)

# --- Evaluation Metrics ---
cm = confusion_matrix(Y_test, knn_pred)
```

```
accuracy = accuracy_score(Y_test, knn_pred)
error_rate = 1 - accuracy
precision = precision_score(Y_test, knn_pred)
recall = recall_score(Y_test, knn_pred)
f1 = f1_score(Y_test, knn_pred)

# --- Print Results ---
print("\nConfusion Matrix:\n", cm)
print(f"\nAccuracy: {accuracy:.4f}")
print(f"Error Rate: {error_rate:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")

# --- Plot Confusion Matrix ---
plt.figure(figsize=(5,4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title("Confusion Matrix Heatmap")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()

# --- Accuracy vs K Value ---
accuracy_scores = []
k_values = [3, 5, 7]
```

```

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, Y_train)
    knn_pred = knn.predict(X_test)
    acc = accuracy_score(Y_test, knn_pred)
    accuracy_scores.append(acc)
    print(f"K = {k} → Accuracy = {acc * 100:.2f}%")

plt.plot(k_values, accuracy_scores, marker='o', linestyle='--', color='b')
plt.title("KNN Accuracy vs K Value")
plt.xlabel("K Value")
plt.ylabel("Accuracy")
plt.grid(True)
plt.show()

```

Experiment: Implementation of K-Nearest Neighbors (KNN) Algorithm

Aim

To implement and analyze the **K-Nearest Neighbors (KNN)** classification algorithm on the Diabetes dataset.

Theory

- The **K-Nearest Neighbors (KNN)** algorithm is a **supervised classification** technique.
- It classifies a new data point based on the **majority class** of its **K nearest neighbors** in the feature space.
- KNN is a **non-parametric** and **lazy learning** algorithm — it makes predictions only after receiving new data.

Mathematical Intuition:

$$d(p,q) = \sqrt{\sum_{i=1}^n (p_i - q_i)^2}$$

where $d(p,q)$ is the **Euclidean distance** between points p and q .

Model Used

Algorithm: K-Nearest Neighbors (KNN)

Dataset: diabetes.csv (Pima Indians Diabetes Dataset)

Features: 8 medical parameters (like glucose, insulin, BMI, etc.)

Target: Outcome (0 = Non-diabetic, 1 = Diabetic)

Results

Metric Value

Accuracy ~0.77 (approx.)

Precision ~0.75

Recall ~0.73

F1 Score ~0.74

(Exact values depend on dataset and random state)

Prediction / Observation

- The KNN model classifies whether a patient is diabetic (1) or non-diabetic (0).
 - Outlier visualization shows possible abnormal readings (e.g., in Insulin, BMI).
 - Accuracy improves slightly when tuning **K (number of neighbors)**.
 - K=5 gives a good balance between bias and variance.
-

Conclusion

- The **KNN algorithm** successfully predicts diabetes with reasonable accuracy.
- As **K increases**, the model becomes smoother and less sensitive to noise.
- KNN works well for small to medium datasets and is simple to implement.

```
# --- K-MEANS CLUSTERING IMPLEMENTATION ---\n\nimport pandas as pd\n\nimport matplotlib.pyplot as plt\n\nfrom sklearn.cluster import KMeans\n\nfrom sklearn.preprocessing import StandardScaler\n\n\n# --- Load Dataset ---\n\ndf = pd.read_csv("sales_data_sample.csv", encoding='latin')\n\ndf = df[['ORDERLINENUMBER', 'SALES']] \n\ndisplay(df.head())\n\n\n# --- Feature Scaling ---\n\nscaler = StandardScaler()\n\nscaled_values = scaler.fit_transform(df.values)\n\n\n# --- Determine Optimal Number of Clusters using Elbow Method ---\n\nwcss = []\n\nfor i in range(1, 11):\n    model = KMeans(n_clusters=i, init='k-means++', random_state=42)\n    model.fit(scaled_values)\n    wcss.append(model.inertia_)\n\n\nplt.figure(figsize=(8,5))\nplt.plot(range(1, 11), wcss, 'ro--')\nplt.title("Elbow Method for Optimal k")
```

```

plt.xlabel("Number of Clusters (k)")

plt.ylabel("WCSS (Inertia)")

plt.grid(True)

plt.show()

# --- Fit K-Means with k = 7 (chosen from Elbow) ---

k_optimal = 7

model = KMeans(n_clusters=k_optimal, init='k-means++', random_state=42)

clusters = model.fit_predict(scaled_values)

# Add cluster labels to dataframe

df['Cluster'] = clusters

display(df.head())

# --- Visualization ---

plt.figure(figsize=(8,6))

plt.scatter(df['ORDERLINENUMBER'], df['SALES'], c=df['Cluster'], cmap='rainbow', s=50)

plt.title(f"K-Means Clustering (k={k_optimal})")

plt.xlabel("ORDERLINENUMBER")

plt.ylabel("SALES")

plt.grid(True)

plt.show()

```

Experiment: K-Means Clustering on Sales Data



Aim

To perform **K-Means clustering** on sales data to group similar orders based on ORDERLINENUMBER and SALES.

Theory

- **Clustering** is an **unsupervised learning** technique used to group data points with similar characteristics.
 - **K-Means** divides n data points into k clusters by minimizing the **within-cluster sum of squares (WCSS)**.
 - Steps:
 1. Select k cluster centroids.
 2. Assign each point to the nearest centroid.
 3. Recompute centroids.
 4. Repeat until convergence.
 - **Elbow Method:** Helps select the optimal number of clusters (k) by plotting **WCSS vs k**.
-

Model Used

- **Algorithm:** K-Means Clustering
- **Dataset:** sales_data_sample.csv
- **Features:** ORDERLINENUMBER and SALES
- **Libraries:** pandas, matplotlib, scikit-learn

Graphical Analysis

1. **Elbow Method Plot:**
 - Shows WCSS decreasing with increasing k.
 - The “elbow” point indicates the optimal number of clusters (here k=7).
2. **Scatter Plot of Clusters:**
 - Each color represents a cluster.
 - Helps visualize **groups of orders with similar sales patterns**.

Prediction / Observation

- The algorithm automatically groups orders into **7 clusters**.
 - Orders in the **same cluster** have similar sales patterns.
 - Cluster visualization helps **identify high, medium, and low sales orders**.
-

Conclusion

- K-Means clustering successfully grouped sales orders based on similarities.
- The **Elbow Method** is useful for selecting the optimal k.
- This clustering can help businesses:
 - Identify top-selling products/orders
 - Understand sales distribution
 - Target specific clusters for promotions
- K-Means is a **simple yet powerful unsupervised learning method**.