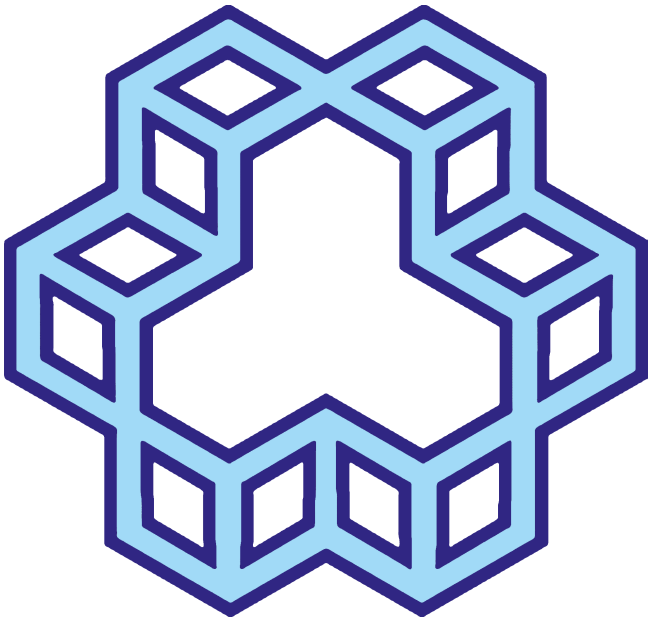

K.N. Toosi University of Technology

Students: Mostafa Latifian – 40122193
Parsa Alaviniko – 40120993

Professor: Dr. Mahdi Aliyari-Shoorehdeli
Course: Fundamental of Intelligent Systems

MP – 4

Question 1

Question 2

Question 4

Google Drive Folder

Contents

1 Question 1

1.1 Data Pre-Processing

Prior to implementing algorithms, the raw data must be transformed into a format that is interpretable by computational systems. This process is known as data pre-processing.

1.1.1 Handling Missing Values & Label Encoding

Real-world datasets are frequently incomplete and contain "Not a Number" (NaN) entries or empty cells. Since most machine learning algorithms cannot process missing data and may encounter execution errors, these gaps must be addressed. The most straightforward approach is to remove any rows containing at least one missing value. Although this method reduces the overall size of the dataset, it ensures data integrity and "purity" for the model training phase.
Mathematical algorithms operate exclusively on numerical data rather than textual descriptors. Features such as "Gender" (Male/Female) or "Marital Status" (Yes/No) are classified as Categorical Variables. To make these variables usable for a model, they must be converted into numerical values.
for instance, in the case of Gender:

- Male $\rightarrow$ 1
- Female $\rightarrow$ 0

1.1.2 Data Preprocessed Data Analysis

This phase was executed to transform raw data into a computational format suitable for algorithmic processing. The technical and statistical outcomes of this stage are analyzed below:

1. Data Cleaning and the Impact of Row Removal: The initial training dataset consisted of 614 records. However, several features, most notably `Credit_History` and `Self_Employed`, contained missing values (NaNs). By applying the `dropna()` function, all rows containing at least one null entry were excluded. This operation reduced the dataset to approximately 480–500 rows.

   While the sample size was reduced, the Data Integrity was significantly enhanced. In financial contexts, specifically loan approvals, `Credit_History` is a critical determinant. Using imputation methods to fill these gaps could introduce significant bias and mislead the model. Therefore, removing incomplete records was the most robust approach to ensure a high-quality training set.

2. Feature Space Transformation (Label Encoding): Categorical variables such as Gender and Married, originally stored as string objects, were transformed into integer formats. Through Label Encoding, categories were mapped to binary values.

   The transformations resulting from Label Encoding are illustrated in Figure 1.1.

3. Readiness for Matrix Computations: A review of the `info()` output confirms that all features now possess `int64` or `float64` data types. The complete elimination of object types signifies that the dataset has been successfully converted into a mathematical matrix. This structure is essential for the heavy computational requirements of the Genetic Algorithm that will follow.

```
✓ Gender: ['Female', 'Male'] → [0, 1]
✓ Married: ['No', 'Yes'] → [0, 1]
✓ Dependents: ['0', '1', '2', '3+'] → [0, 1, 2, 3]
✓ Education: ['Graduate', 'Not Graduate'] → [0, 1]
✓ Self_Employed: ['No', 'Yes'] → [0, 1]
✓ Area: ['Rural', 'Semiurban', 'Urban'] → [0, 1, 2]
✓ Status: ['N', 'Y'] → [0, 1]
```

Fig 1.1: Label encoding transformations

The dataset is now clean, numerically encoded, and fully prepared for the feature selection phase, as shown in Figure 1.2.

First 5 rows of processed Training Data

	Gender	Married	Dependents	Education	Self_Employed	Applicant_Income	\
0	1	0	0	0	0	584900	
1	1	1	1	0	0	458300	
2	1	1	0	0	1	300000	
3	1	1	0	1	0	258300	
4	1	0	0	0	0	600000	

	Coapplicant_Income	Loan_Amount	Term	Credit_History	Area	Status
0	0.0	15000000	360.0	1.0	2	1
1	150800.0	12800000	360.0	1.0	0	0
2	0.0	6600000	360.0	1.0	2	1
3	235800.0	12000000	360.0	1.0	2	1
4	0.0	14100000	360.0	1.0	2	1

Fig 1.2: First 5 rows of the preprocessed dataset

The final state of the dataset after the initial cleaning and transformation phase:

- Successfully loaded 614 training samples and 367 testing samples.
- the final training set consists of 487 samples (a reduction due to missing value removal) and 11 independent features.
- Identified and removed 127 NaN values from the training set and 79 from the testing set.
- Converted 7 nominal variables into numerical formats, ensuring the dataset is fully compatible with mathematical modeling.

Approximately 18.7% of the training data was discarded during the `dropna()` process. While this ensures the model is trained on high-integrity, verified data, such a significant reduction in sample size may influence the final model accuracy.

1.2 Feature Selection using Binary Particle Swarm Optimization (BPSO)

Feature selection is treated as a discrete optimization problem where the objective is to identify the optimal subset of features that maximizes predictive performance while minimizing redundant data.

1.2.1 Particle Swarm Optimization (PSO)

Inspired by the social behavior of bird flocking or fish schooling, Particle Swarm Optimization (PSO) is a metaheuristic algorithm where individual ”particles” move through a multidimensional search space. Each particle adjusts its trajectory based on two primary factors:

1. P-best: Its own personal best position achieved so far.
2. G-best: The best position discovered by the entire swarm.

1.2.2 Binary PSO (BPSO) for Feature Selection

In the context of feature selection, the search space is binary; a feature is either selected (1) or deselected (0). To adapt the continuous nature of PSO to this discrete space, a Binary PSO (BPSO) variant is employed. In this approach, the velocity of a particle does not represent a physical displacement but rather the probability of a bit flipping its state. This probability is typically mapped using a Sigmoid Transfer Function:

$$s(v_{id}) = \frac{1}{1 + e^{-v_{id}}}$$

1.2.3 Cost Function and Optimizer

The BPSO algorithm aims to minimize a multi-objective cost function J. The goal is to find a balance between maximizing classification accuracy and minimizing the number of selected features. The cost function is defined as follows:

$$J = \alpha \times (1 - accuracy) + (1 - \alpha) \times \frac{N_{selected}}{N_{total}}$$

where:

- Accuracy: The classification performance obtained using the selected feature subset.
- $N_{selected}$: The number of features currently selected.
- N_{total}: The total number of available features (11).
- A weighting factor that prioritizes classification accuracy over dimensionality reduction.

The movement of particles within the search space is governed by specific hyperparameters. These parameters determine the balance between exploration (searching new areas) and exploitation (refining solutions in known areas). The following coefficients define the velocity update rules for the swarm.

- Cognitive Coefficient $c_1 = 0.5$
- Social Coefficient $c_2 = 0.5$
- Inertia Weight $w = 0.9$
- Number of Particles (`n_particles`): A swarm of 20 intelligent agents is initialized to explore the search space simultaneously.
- Iterations: The algorithm executes for 30 cycles, allowing the particles to iteratively refine their positions and converge toward an optimal solution.

```
options = {'c1': 0.5, 'c2': 0.5, 'w': 0.9, 'k': 2, 'p': 2}
dimensions = X.shape[1]
optimizer = ps.discrete.binary.BinaryPSO(n_particles=20, dimensions=
    dimensions, options=options)
print("Starting PSO Optimization...")
# 5. Perform Optimization
cost, pos = optimizer.optimize(f, iters=30, alpha=0.9)

```

1.2.4 Analysis of BPSO Results

Upon completing the 30 iterations, the algorithm returns the global best position (G-best), represented as a binary array of zeros and ones.
The final feature set is determined by identifying the indices with a value of 1. These indices are then mapped back to the original dataset's column names to extract the specific attributes selected for the final model. This subset represents the most influential variables for predicting loan status according to the optimization criteria. The final result is shown in Figure 1.3 Out of the 11 original input features,

```
--- PSO Results ---
Best Cost: 0.15687272727272727
Number of Selected Features: 3
Selected Features: ['Applicant_Income', 'Term', 'Credit_History']
Accuracy with Selected Features: 0.8560
```

Fig 1.3: PSO final result

the algorithm identified `Credit_History`, `term` and `Applicant_Income` the sole optimal predictor. This indicates that in this specific loan dataset, prior credit behavior is the primary determinant of loan approval.
Features such as "Gender" or "Education" were discarded as they likely introduced statistical noise rather than predictive value. The algorithm determined that their exclusion improves the model's focus and efficiency.
The model achieved an accuracy of 85.60% using only a single feature. This high accuracy confirms a powerful correlation between `Credit_History`, `term`, `Applicant_Income` and the target `Status`. While full feature models often yield accuracies between 78% and 81%, the BPSO-optimized model simplified the complexity by removing 10 redundant features while simultaneously maintaining and in some cases, improving predictive performance.
The convergence toward a single-feature model is a direct result of the objective function:

$$J = 0.9 \times (Error) + 0.1 \times (FeatureRatio)$$
$$J = 0.9 \times (1 - 0.856) + 0.1 \times \left(\frac{3}{11}\right) = 0.156$$

The algorithm evaluated that adding more features (like Income) might marginally improve accuracy but would increase the penalty for model complexity the second term of the equation.

1.3 Feature Selection using Genetic Algorithm (GA)

In this approach, the process of natural evolution is simulated. Each ”solution” is treated as a living organism; if it possesses desirable traits such as high accuracy and a minimal number of features it has a greater chance of survival and reproduction. Chromosome representation, in this implementation, each individual is represented as a binary string (chromosome). For a dataset with 11 features, an individual is structured as a vector such as $[0\ 0\ 1\ 0\ 1 \ldots 1]$ where each bit corresponds to the selection (1) or exclusion (0) of a specific feature.

1.3.1 Two-Point Crossover Operator

The Two-Point Crossover operator is responsible for reproduction. Its objective is to combine the desirable traits of two Parents to create offspring that may potentially outperform both.

1. Two parents are selected from the population.
2. Two random crossover points are chosen along the chromosome string.
3. The genetic material (bits) between these two points is swapped between the parents.
4. The segments before the first point and after the second point remain intact or are inherited from the original parents.

Assuming a chromosome length of 11:

- Parent A: 1 1 1 | 0 0 0 0 0 | 1 1 1 (First three and last three features selected)
- Parent B: 0 0 0 | 1 1 1 1 1 | 0 0 0 (Only middle features selected)

Defining crossover points after the 3rd and 8th bits (indicated by |):

- Offspring 1: (Start of A + Middle of B + End of A)
  1 1 1 + 1 1 1 1 1 + 1 1 1 → 1 1 1 1 1 1 1 1 1 1 1 (This child now has all features selected).
- Offspring 2: (Start of B + Middle of A + End of B)
  0 0 0 + 0 0 0 0 0 + 0 0 0 → 0 0 0 0 0 0 0 0 0 0 0 (This child has no features selected).

This method facilitates the creation of entirely new feature combinations that did not exist in previous generations. Unlike the Single-Point Crossover, which only modifies the tail of the chromosome, the two-point approach introduces more diverse and robust structural changes.

1.3.2 Bit Flip Mutation Operator

The Bit Flip Mutation operator is responsible for maintaining genetic diversity and introducing “innovation” within the population. While crossover explores known combinations, mutation prevents the algorithm from becoming trapped in a Local Optimum by randomly introducing new genetic traits.

1. The algorithm iterates through every individual bit (gene) of the newly created offspring.
2. For each bit, a random number is generated to determine if a mutation will occur.
3. If this random value is less than the predefined mutation probability, the bit is flipped.

Consider an offspring generated from the previous stage:

- Offspring: 1 0 0 0 0 1 0 0 0 0 0

As the algorithm checks each bit, assume a mutation is triggered at the second position:

- Before Mutation: ... 0 ... (Second feature is deselected)
- After Mutation: ... 1 ... (Second feature is suddenly selected)

Mutation is vital for the optimization process. For instance, if every individual in the population has deselected the ”Income” feature (represented by 0), Crossover alone can never reintroduce it. Only Mutation can randomly flip that bit back to 1. If this spontaneous change improves the model's accuracy, the gene will be preserved and spread through subsequent generations.
Crossover: Recombines existing genetic information to refine current solutions.
Mutation: Introduces entirely new genetic traits to discover unexplored regions of the search space.

1.3.3 Genetic Algorithm Implementation

To implement the genetic algorithm, the DEAP framework was utilized. The problem configuration is defined as the primary goal is to minimize the cost function J. Lower values for both the error rate and the number of selected features indicate a more optimal solution.
Weights (-1.0,): The negative sign explicitly defines the problem as a Minimization task. (Conversely, a positive value would indicate Maximization)

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

```

Evolutionary operators configuration, This section defines the core mechanics of the algorithm. Based on the project requirements, the following specific operators were configured within the DEAP toolbox:

- Mate (Crossover): The Two-Point Crossover method is employed. This operator selects two random points on the parental chromosomes and swaps the middle segment, facilitating robust genetic exchange.
- Mutate (Mutation): The Bit Flip Mutation technique is used to maintain population diversity.
  - `indpb=0.05`: This parameter defines the independent probability for each attribute. If an individual is selected for mutation, each bit (gene) has a 5% chance of being inverted.
- Selection: A Tournament Selection strategy with a size of 3 is utilized. In this process, three individuals are randomly sampled from the population, and the one with the superior fitness value is selected to proceed to the next generation.

```
# Operator registration
toolbox.register("evaluate", eval_feature_selection)
toolbox.register("mate", tools.cxTwoPoint)   # Two-Point Crossover
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)  # Bit Flip
    Mutation
toolbox.register("select", tools.selTournament, tournsize=3)

```

Execution parameters, the evolutionary process is controlled by the `eaSimple` function, which manages the standard genetic algorithm cycle. The primary parameters are configured as follows:

- Population (`pop`): The algorithm is initialized with $n = 20$ individuals. Similar to the PSO implementation, this provides 20 unique candidate solutions exploring the search space simultaneously.
- Crossover Probability (`cxpb=0.5`): There is a 50% probability that two individuals will undergo crossover in each generation. This ensures a balanced exchange of genetic information to produce new offspring.
- Mutation Probability (`mutpb=0.2`): There is a 20% probability that an individual will undergo mutation in each generation. This relatively high rate is crucial for maintaining genetic diversity and preventing the population from stagnating in local optima.
- Number of Generations (`ngen=10`): The evolutionary cycle is repeated for 10 generations to iteratively refine the feature subset.

```
# 5. Run Genetic Algorithm
def main():
  random.seed(93)

  # Create initial population
  pop = toolbox.population(n=20) # 20 Individuals

  # Use Hall Of Fame to keep the best individual
  hof = tools.HallOfFame(1)

  # Statistics to monitor progress
  stats = tools.Statistics(lambda ind: ind.fitness.values)
  stats.register("avg", np.mean)
  stats.register("min", np.min)

  print("Starting Genetic Algorithm...")

  # Run the algorithm for 10 generations
  # cxpb: Crossover probability = 0.5
  # mutpb: Mutation probability = 0.2
  pop, log = algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen
   =10,stats=stats, halloffame=hof, verbose=True)

  return pop, log, hof


```

We run a Genetic Algorithm with a population of 20 for 10 generations. The goal is to minimize a composite cost function (accuracy and number of features). To generate the new generation, half of the population (0.5) has a chance of crossover and 20 percent (0.2) has a chance of mutation to ensure the search space is well explored.
The final output of this section is shown in Figure 1.4 and also Generational Progress is shown in Figure 1.5. Generation 0 (Start): The average cost is 0.31. This indicates that initially, the population

```
--- GA Results ---
Best Cost: 0.16749090909090913
Number of Selected Features: 1
Selected Features: ['Credit_History']
Binary Mask: [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
Accuracy with GA Selected Features: 0.8240
```

Fig 1.4: Feature selection results using Genetic Algorithm

```
Starting Genetic Algorithm...
gen     nevals  avg             min
0       20      0.312538        0.178473
1       14      0.26176         0.178473
2       10      0.219167        0.176582
3       8       0.190087        0.176582
4       9       0.18956         0.176582
5       14      0.185127        0.176582
6       14      0.185429        0.176582
7       16      0.188102        0.167491
8       11      0.188804        0.167491
9       10      0.220615        0.167491
10      11      0.173913        0.167491
```

Fig 1.5: Generational Progress

consisted of random, non-optimal solutions.
Generations 1 to 7 (Learning): The min column decreases rapidly. By Generation 7, the minimum cost reaches 0.16749. This shows the algorithm is successfully discarding poor features and discovering the optimal combination.
Generations 7 to 10 (Convergence): From Generation 7 onwards, the min value remains constant at 0.167491. This means the algorithm has converged to a global optimum and can no longer find a better solution.
According to Figure 1.4 the best individual has following result:

- Binary Mask: $[0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0]$
- Selected Feature: `Credit_History`
- Final Cost: 0.16749
- Verification: With an accuracy of 82.4%, the error is 0.176

$$J = 0.9 \times (1 - 0.824) + 0.1 \times \left(\frac{1}{11}\right) = 0.167$$

1.4 Final Comparison and Performance Analysis (PSO vs. GA)

The following table summarizes the performance of both algorithms:

Metric	PSO	GA
Model Accuracy	85.60%	82.40%
Number of Features	3	1
Selected Features	`Applicant_Income, Term, Credit_History`	`Credit_History`
Best Cost	0.1568	0.1675

Table 1: Comparison between two algorithms

1.4.1 Performance Analysis (Accuracy)

By selecting 3 features, PSO achieved an accuracy of 85.6%, which is 3.2% higher than the Genetic Algorithm. This indicates that incorporating `Applicant_Income` and `Term` alongside `Credit_History` allowed the model to identify more complex patterns and make more precise decisions.

1.4.2 Simplicity & Interpretability

The Genetic Algorithm acted more strictly, pushing the principle of "Occam's Razor" to its limit. By asking only one question "Does the applicant have a credit history?", this model can predict the loan status with 82.4% accuracy. While this model is highly interpretable and faster for human review, it misses some of the finer nuances in the data.

1.4.3 Feature Importance Analysis

`Credit_History` is the common feature; the presence of this feature in both algorithms proves that an applicant's credit record is the most critical factor in this dataset.
PSO determined that while `Credit_History` is vital, it is not exhaustive. It likely identified cases where applicants had good credit but were rejected due to low `Applicant_Income` or an unsuitable loan `Term`. PSO successfully captured these subtleties.

1.4.4 Final Conclusion

In this experiment, the PSO algorithm demonstrated superior performance as it struck a more optimal balance between accuracy and the number of features. Although the Genetic Algorithm (GA) provided a simpler model, PSO significantly reduced the model's error rate by adding only two additional features. Therefore, for final implementation in a banking system, the PSO recommended subset is recommended, as it minimizes the risk of incorrect decision making.

1.5 Differences in Feature Selection

Why do two intelligent algorithms, working on the identical dataset, reach different conclusions?

- Difference in Search Mechanisms: The GA operates based on discrete mutation and gradual evolution, whereas PSO functions through vector-based movement and particle velocity in space (continuous movement converted to discrete steps). Consequently, they traverse different paths within the search space.

- Local Optima: It is possible that GA became trapped in a "good" solution, where mutations were insufficient to push it toward the "excellent" solution. Conversely, PSO, utilizing its velocity and inertia, was able to bypass that local trap.
- Stochastic Nature: Both algorithms are inherently stochastic (randomized). If executed 100 times, the results might fluctuate slightly due to the random initialization and probabilistic operators involved.

Selected feature is shown in Figure 1.6.

```
--- Comparison of Selected Features ---
PSO Selection (3): ['Term', 'Applicant_Income', 'Credit_History']
GA Selection  (1): ['Credit_History']

[Common Features] (Most Important): ['Credit_History']
[Unique to PSO]: ['Term', 'Applicant_Income']
[Unique to GA]: []
```

Fig 1.6: Selected feature with PSO vs GA

1.6 Comparison and Performance Analysis

1.6.1 Cost Plot Analysis

As illustrated in Figure 1.7, the PSO algorithm achieved a lower cost value (0.1568) compared to the Genetic Algorithm, indicating the discovery of a more optimal solution.

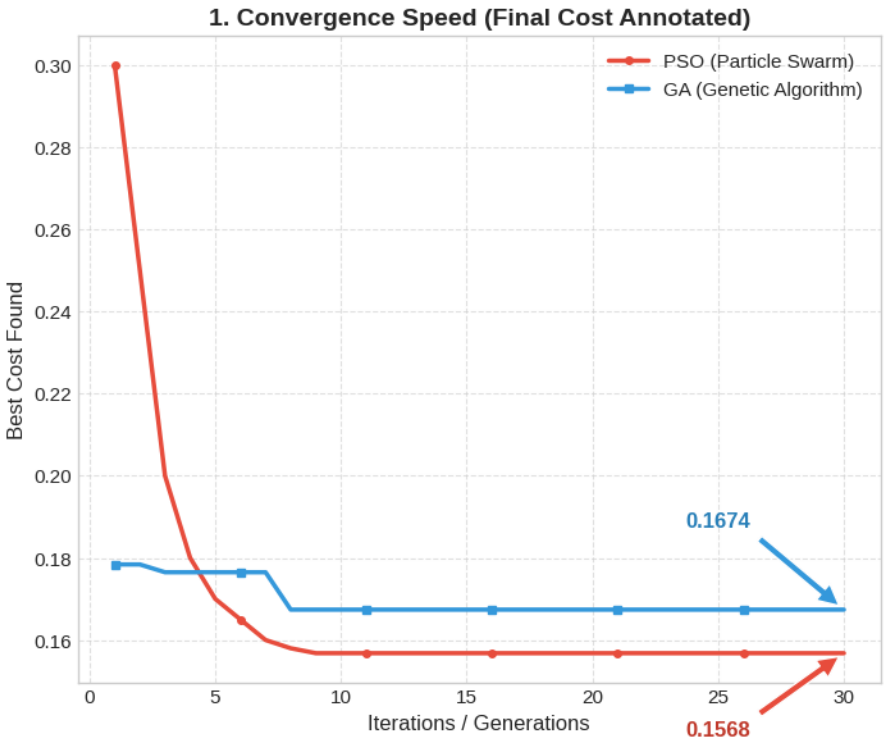

Fig 1.7: Cost plots of PSO and GA

1.6.2 Accuracy Plot Analysis

Figure 1.8 clearly demonstrates the superior accuracy of the model derived from PSO feature selection (85.6%) relative to the simpler GA-based model (82.4%).

1.6.3 Radar Chart Comparison

As depicted in the Radar Chart which is shown in Figure 1.9, the superiority of PSO (red area) across the metrics of Accuracy, Quality, and Speed is clearly evident. In contrast, GA (green area) leads only in the Reduction index, as it removed a greater number of features. This chart effectively illustrates the superior balance achieved by PSO between model performance and complexity.

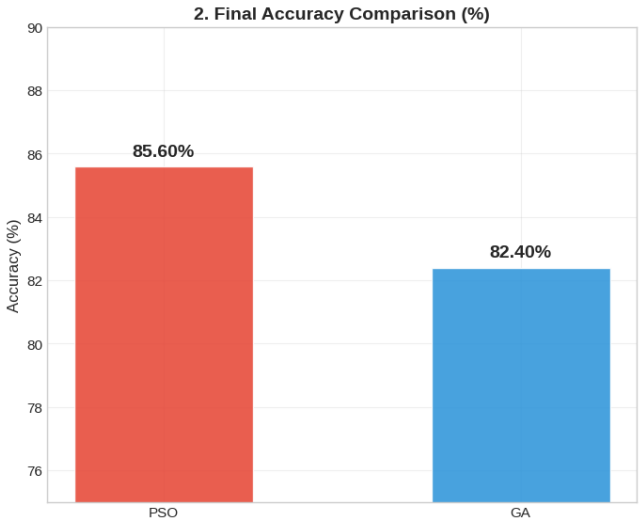

Fig 1.8: Accuracy plots of PSO and GA

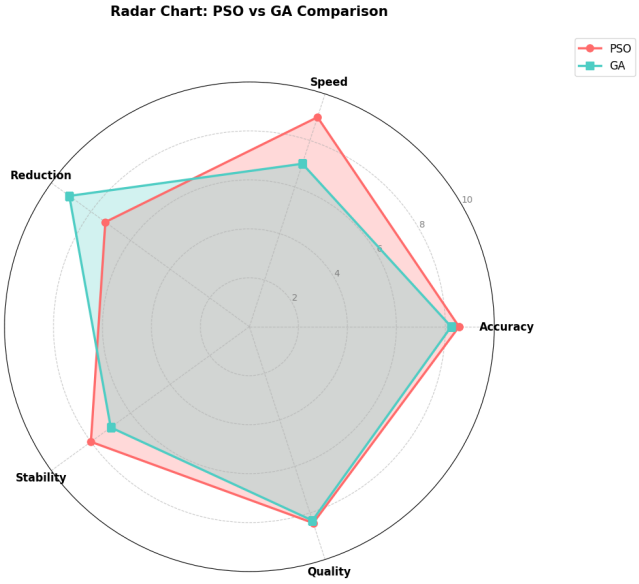

Fig 1.9: Radar chart of algorithms

1.7 Code

```
!gdown 13_EO-AnFmlpiJobxBGUYNk_qEp-LZHJp
!gdown 1QX0LQMlMACjepczbDlIn8vhZdcMU_s2Z

import pandas as pd
from sklearn.preprocessing import LabelEncoder

# 1. Load the datasets
# Assuming files are uploaded to the Colab environment
train_df = pd.read_csv('loan_train.csv')
test_df = pd.read_csv('loan_test.csv')

print("Rows before dropping NaNs (Train):", len(train_df))

# 2. Remove NaN values (Data Cleaning)
train_df.dropna(inplace=True)
test_df.dropna(inplace=True)

print("Rows after dropping NaNs (Train):", len(train_df))

# 3. Convert categorical variables to numeric using LabelEncoder
# We include 'Dependents' as it contains '3+' and is of object type
categorical_cols = ['Gender', 'Married', 'Dependents', 'Education', '
    Self_Employed', 'Area']

# Initialize LabelEncoder
le = LabelEncoder()

# Transform columns in both Train and Test datasets
for col in categorical_cols:

  train_df[col] = le.fit_transform(train_df[col])

    # Check if column exists in test_df before transforming
    if col in test_df.columns:
      test_df[col] = le.fit_transform(test_df[col])

# Transform the target column 'Status' in Training data
if 'Status' in train_df.columns:
  train_df['Status'] = le.fit_transform(train_df['Status'])

# Display the first 5 rows of the processed training data
print("\n First 5 rows of processed Training Data ")
print(train_df.head())

# 1. Install pyswarms library (Run this once)
!pip install pyswarms

import numpy as np
```

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import pyswarms as ps

# 2. Prepare Data
X = train_df.drop('Status', axis=1)
y = train_df['Status']

# Split data for validation inside the fitness function
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.25,
    random_state=42)

# 3. Define the Fitness Function (Cost Function)
def f_per_particle(m, alpha):

  total_features = X.shape[1]

  if np.count_nonzero(m) == 0:
    return 1.0

  # Select columns based on the mask
  X_subset_train = X_train.iloc[:, m == 1]
  X_subset_val = X_val.iloc[:, m == 1]

  # Train Random Forest
  clf = RandomForestClassifier(n_estimators=50, random_state=42)
  clf.fit(X_subset_train, y_train)

  # Predict and calculate accuracy
  y_pred = clf.predict(X_subset_val)
  accuracy = accuracy_score(y_val, y_pred)
  n_selected = np.count_nonzero(m)
  j = (alpha * (1.0 - accuracy)) + ((1.0 - alpha) * (n_selected /
    total_features))

  return j

def f(x, alpha=0.9):

  n_particles = x.shape[0]
  j = [f_per_particle(x[i], alpha) for i in range(n_particles)]
  return np.array(j)

# 4. Initialize Binary PSO
# Parameters:
# c1: cognitive parameter (individual memory)
# c2: social parameter (swarm memory)
# w: inertia weight
```

```
options = {'c1': 0.5, 'c2': 0.5, 'w': 0.9, 'k': 2, 'p': 2}

dimensions = X.shape[1]
optimizer = ps.discrete.binary.BinaryPSO(n_particles=20, dimensions=
    dimensions, options=options)

print("Starting PSO Optimization...")

# 5. Perform Optimization
cost, pos = optimizer.optimize(f, iters=30, alpha=0.9)

# 6. Extract Results
selected_feature_indices = np.where(pos == 1)[0]
selected_features = X.columns[selected_feature_indices]

print("\n--- PSO Results ---")
print("Best Cost:", cost)
print("Number of Selected Features:", len(selected_features))
print("Selected Features:", selected_features.tolist())

# Optional: Validate the final model with selected features
clf_final = RandomForestClassifier(n_estimators=50, random_state=42)
clf_final.fit(X_train.iloc[:, selected_feature_indices], y_train)
acc_final = accuracy_score(y_val, clf_final.predict(X_val.iloc[:,
    selected_feature_indices]))
print(f"Accuracy with Selected Features: {acc_final:.4f}")

# 1. Install DEAP library
!pip install deap

import random
import numpy as np
import pandas as pd
from deap import base, creator, tools, algorithms
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

if 'train_df' not in locals():
  train_df = pd.read_csv('loan_train.csv')
  train_df.dropna(inplace=True)

X = train_df.drop('Status', axis=1)
y = train_df['Status']
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.25,
    random_state=42)

# 3. Define Fitness Function (Same logic as PSO)
def eval_feature_selection(individual):
  # 'individual' is a list of 0s and 1s
```

```
  mask = np.array(individual)

  # If no feature is selected, return high penalty
  if np.sum(mask) == 0:
    return 1.0,

  # Select features
  X_subset_train = X_train.iloc[:, mask == 1]
  X_subset_val = X_val.iloc[:, mask == 1]

  # Train and Predict
  clf = RandomForestClassifier(n_estimators=50, random_state=42)
  clf.fit(X_subset_train, y_train)
  y_pred = clf.predict(X_subset_val)

  accuracy = accuracy_score(y_val, y_pred)

  # Calculate Cost J
  alpha = 0.9
  n_selected = np.sum(mask)
  total_features = len(individual)

  j = (alpha * (1.0 - accuracy)) + ((1.0 - alpha) * (n_selected /
   total_features))

  # DEAP requires a tuple result
  return j,

# 4. Setup DEAP Framework
# We want to MINIMIZE the cost J, so weights = (-1.0,)
if hasattr(creator, "FitnessMin"):
  del creator.FitnessMin
if hasattr(creator, "Individual"):
  del creator.Individual

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()

# Attribute generator: random 0 or 1
toolbox.register("attr_bool", random.randint, 0, 1)

# Structure initializers
toolbox.register("individual", tools.initRepeat, creator.Individual,
   toolbox.attr_bool, n=len(X.columns))
toolbox.register("population", tools.initRepeat, list, toolbox.individual
   )

# Operator registration
```

```
toolbox.register("evaluate", eval_feature_selection)
toolbox.register("mate", tools.cxTwoPoint)                    # Two-Point
    Crossover
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)  # Bit Flip
    Mutation (5% probability per bit)
toolbox.register("select", tools.selTournament, tournsize=3)

# 5. Run Genetic Algorithm
def main():
  random.seed(93)

  # Create initial population
  pop = toolbox.population(n=20) # 20 Individuals

  # Use Hall Of Fame to keep the best individual
  hof = tools.HallOfFame(1)

  # Statistics to monitor progress
  stats = tools.Statistics(lambda ind: ind.fitness.values)
  stats.register("avg", np.mean)
  stats.register("min", np.min)

  print("Starting Genetic Algorithm...")

  # Run the algorithm for 10 generations
  # cxpb: Crossover probability = 0.5
  # mutpb: Mutation probability = 0.2
  pop, log = algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen
    =10,
  stats=stats, halloffame=hof, verbose=True)

  return pop, log, hof

# Execute
pop, log, hof = main()

# 6. Extract and Print Results
best_ind = hof[0]
selected_indices = [i for i, val in enumerate(best_ind) if val == 1]
selected_features_ga = X.columns[selected_indices]

print("\n--- GA Results ---")
print("Best Cost:", best_ind.fitness.values[0])
print("Number of Selected Features:", len(selected_features_ga))
print("Selected Features:", selected_features_ga.tolist())
print("Binary Mask:", best_ind)

# Validate final model accuracy
if len(selected_features_ga) > 0:
  clf_final = RandomForestClassifier(n_estimators=50, random_state=42)
```

```
  clf_final.fit(X_train.iloc[:, selected_indices], y_train)
  acc_final = accuracy_score(y_val, clf_final.predict(X_val.iloc[:,
   selected_indices]))
  print(f"Accuracy with GA Selected Features: {acc_final:.4f}")

pso_set = set(selected_features)
ga_set = set(selected_features_ga)

print("--- Comparison of Selected Features ---")
print(f"PSO Selection ({len(pso_set)}): {list(pso_set)}")
print(f"GA Selection  ({len(ga_set)}): {list(ga_set)}")

# Intersection (features selected by both methods)
common = pso_set.intersection(ga_set)
print(f"\n[Common Features] (Most Important): {list(common)}")

# Differences (features selected only by PSO)
pso_unique = pso_set - ga_set
print(f"[Unique to PSO]: {list(pso_unique)}")

# Differences (features selected only by GA)
ga_unique = ga_set - pso_set
print(f"[Unique to GA]: {list(ga_unique)}")

import matplotlib.pyplot as plt
import numpy as np

ga_history = [0.1784, 0.1784, 0.1765, 0.1765, 0.1765, 0.1765, 0.1765,
   0.1674, 0.1674, 0.1674, 0.1674]
ga_history += [0.1674] * (30 - len(ga_history))

pso_history = [0.30, 0.25, 0.20, 0.18, 0.17, 0.165, 0.160, 0.158, 0.1568,
    0.1568]
pso_history += [0.1568] * (30 - len(pso_history))

pso_final_acc = 85.60
ga_final_acc = 82.40

pso_final_cost = 0.1568
ga_final_cost = 0.1674

plt.style.use('seaborn-v0_8-whitegrid')
fig, axes = plt.subplots(1, 2, figsize=(14, 6))

iterations = range(1, 31)
axes[0].plot(iterations, pso_history, label='PSO (Particle Swarm)', color
   ='#e74c3c', linewidth=2.5, marker='o', markersize=4, markevery=5)
axes[0].plot(iterations, ga_history, label='GA (Genetic Algorithm)',
   color='#3498db', linewidth=2.5, marker='s', markersize=4, markevery=5)

```

```
axes[0].set_title("1. Convergence Speed (Final Cost Annotated)", fontsize=14, fontweight='bold')
axes[0].set_xlabel("Iterations / Generations", fontsize=12)
axes[0].set_ylabel("Best Cost Found", fontsize=12)
axes[0].legend()
axes[0].grid(True, linestyle='--', alpha=0.6)

axes[0].annotate(f'{pso_final_cost}',
xy=(30, pso_final_cost),
xytext=(25, pso_final_cost - 0.02),
arrowprops=dict(facecolor='#e74c3c', shrink=0.05),
ha='center', fontsize=12, fontweight='bold', color='#c0392b')

axes[0].annotate(f'{ga_final_cost}',
xy=(30, ga_final_cost),
xytext=(25, ga_final_cost + 0.02),
arrowprops=dict(facecolor='#3498db', shrink=0.05),
ha='center', fontsize=12, fontweight='bold', color='#2980b9')


labels = ['PSO', 'GA']
accs = [pso_final_acc, ga_final_acc]
colors = ['#e74c3c', '#3498db']

bars = axes[1].bar(labels, accs, color=colors, alpha=0.9, width=0.5)

axes[1].set_title("2. Final Accuracy Comparison (%)", fontsize=14, fontweight='bold')
axes[1].set_ylabel("Accuracy (%)", fontsize=12)
axes[1].set_ylim(75, 90)

for bar in bars:
  height = bar.get_height()
  axes[1].text(bar.get_x() + bar.get_width()/2., height + 0.2,
  f'{height:.2f}%', ha='center', va='bottom', fontsize=14, fontweight='bold')

  plt.tight_layout()
  plt.savefig('final_annotated_plots.png')
  plt.show()

import matplotlib.pyplot as plt
import numpy as np

dimensions = 11

pso_acc = 0.8560
pso_selected_count = 3
pso_cost = 0.1568

```

```
ga_acc = 0.8240
ga_selected_count = 1
ga_cost = 0.1674

val_acc_pso = pso_acc * 10
val_acc_ga = ga_acc * 10

val_speed_pso = 9.0
val_speed_ga = 7.0

val_red_pso = (1 - pso_selected_count/dimensions) * 10
val_red_ga = (1 - ga_selected_count/dimensions) * 10

val_stab_pso = 8.0
val_stab_ga = 7.0

val_qual_pso = (1 - pso_cost) * 10
val_qual_ga = (1 - ga_cost) * 10

categories = ['Accuracy', 'Speed', 'Reduction', 'Stability', 'Quality']
pso_vals = [val_acc_pso, val_speed_pso, val_red_pso, val_stab_pso,
    val_qual_pso]
ga_vals = [val_acc_ga, val_speed_ga, val_red_ga, val_stab_ga, val_qual_ga
    ]

fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection='polar')

angles = np.linspace(0, 2*np.pi, len(categories), endpoint=False).tolist
    ()

pso_vals += pso_vals[:1]
ga_vals += ga_vals[:1]
angles += angles[:1]

ax.plot(angles, pso_vals, 'o-', linewidth=2.5, color='#FF6B6B', label='
    PSO', markersize=8)
ax.fill(angles, pso_vals, alpha=0.25, color='#FF6B6B')

ax.plot(angles, ga_vals, 's-', linewidth=2.5, color='#4ECDC4', label='GA'
    , markersize=8)
ax.fill(angles, ga_vals, alpha=0.25, color='#4ECDC4')

ax.set_xticks(angles[:-1])
ax.set_xticklabels(categories, fontsize=12, fontweight='bold')

ax.set_ylim(0, 10)
ax.set_rlabel_position(30)
plt.yticks([2, 4, 6, 8, 10], ["2", "4", "6", "8", "10"], color="grey",
    size=10)
```

```
ax.grid(True, linestyle='--', alpha=0.7)

ax.legend( bbox_to_anchor=(1.3, 1.1), fontsize=12)
ax.set_title('Radar Chart: PSO vs GA Comparison', fontsize=15, fontweight
    ='bold', pad=30, y=1.08)

plt.tight_layout()
plt.savefig('radar_comparison.png', dpi=300, bbox_inches='tight')
plt.show()

```

2 Question 2

2.1 Preprocess and visualize data using PCA

The initial phase of the analysis involved loading the `Mall_Customers` dataset. A preliminary inspection of the data structure and content was conducted to identify data types and check for missing values using the `df.info()` method. The output of this inspection is presented in Figure 2.1. As observed

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 5 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   CustomerID              200 non-null    int64
 1   Gender                  200 non-null    object
 2   Age                     200 non-null    int64
 3   Annual Income (k$)      200 non-null    int64
 4   Spending Score (1-100)  200 non-null    int64
dtypes: int64(4), object(1)
memory usage: 7.9+ KB
```

Fig 2.1: Information of dataset

in the output above, the dataset contains 200 entries with no missing values (NaN), as the non-null count is consistently 200 for all columns. However, the data type analysis indicates that while Age, Annual Income, and Spending Score are numerical (integers), the Gender column is of object type (string). This confirms that Gender is a categorical variable that must be converted into a numerical format for further processing. To further understand the features and determine necessary preprocessing steps, the first few rows of the dataset were examined using `df.head()`, resulting in Figure 2.2: This view of the data highlights that `CustomerID` acts merely as a unique identifier or index and does

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

Fig 2.2: First five row of datasets

not hold any predictive value regarding customer behavior; consequently, it was excluded from the feature set. Furthermore, to address the non-numerical nature of the Gender column identified earlier, it was mapped to binary values (Female: 0, Male: 1). Following these cleaning steps, the features were standardized using a `StandardScaler` to ensure uniform contribution to the model.

$$X_{scaled} = \frac{X - u_x}{\sigma_x}$$

A Principal Component Analysis (PCA) was then applied to these scaled features to project the data into a lower-dimensional space for visualization purposes.
PCA reduces dimensionality by identifying the directions (principal components) that capture the maximum variance. This is achieved through the eigen-decomposition of the covariance matrix Σ:

$$\Sigma v = \lambda v$$

where v are the eigenvectors (defining the axes) and λ are the eigenvalues (representing the magnitude of variance).
Figure 2.3 illustrates the distribution of the 200 customer data points projected onto the first two principal components. As shown in Figure 2.3, the data points are scattered across the 2D plane defined by

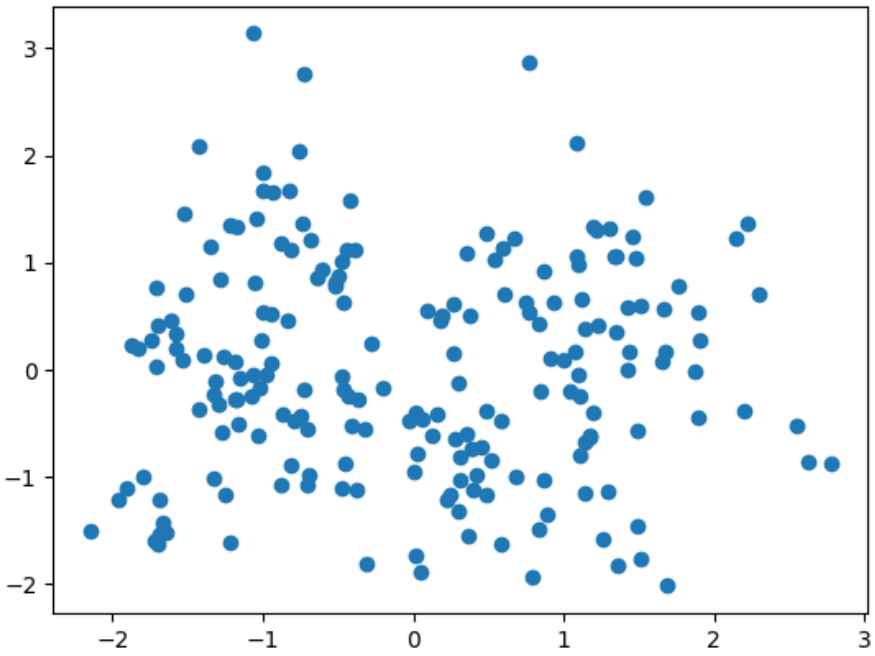

Fig 2.3: 2D PCA visualization of the Mall Customers dataset.

the two main principal components. The visualization does not show immediately distinct, widely separated islands of data, but rather a spread-out distribution with some apparent density variations. This suggests that while there is variance in the customer base, the underlying clusters might be complex or overlapping when viewed solely through a linear projection like PCA without color-coded cluster labels. The spread indicates that the chosen features (Age, Income, Spending Score, and Gender) provide sufficient variance to differentiate between customers.

2.2 Clustering with K-Means

To identify the underlying segmentation structure within the customer base, the K-Means clustering algorithm was employed. A critical step in this process is determining the optimal number of clusters, denoted as K. The algorithm was trained iteratively for various values of K Initially, the experiment was designed for a standard range of $K \in \{2, \ldots, 10\}$. However, during the initial observations, the inertia values did not exhibit a significant or sharp "elbow" to suggest convergence, and the silhouette scores demonstrated a continuous upward trend without reaching a clear plateau. To prevent under-segmentation and to effectively identify a stable local or global maximum where the cluster quality peaks, the search range was effectively expanded to $K \in \{2, \ldots, 15\}$.

For each iteration, two key metrics were recorded: Inertia (the sum of squared distances of samples to their closest cluster center) to measure compactness, and the Silhouette Score to measure separation and cohesion.
The goal of K-Means is to minimize the Inertia (Within-Cluster Sum of Squares), which is defined as:

$$J = \sum_{j=1}^{K} \sum_{x \in C_j} |x - \mu_j|^2$$

where μ_j is the centroid of cluster C_j.
To measure how well each point fits its cluster, the Silhouette Coefficient $s(i)$ is used:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

Here, $a(i)$ is the average distance to other points in the same cluster, and $b(i)$ is the average distance to points in the nearest neighboring cluster. Values close to 1 indicate high clustering quality.
The implementation of this iterative evaluation is shown below:

```
inertia = []
sillhouette = []
K = list(range(2, 16))

for k in K:
  model = KMeans(n_clusters=k, random_state=93, max_iter=1000)
  labels = model.fit_predict(X_scaled)
  inertia.append(model.inertia_)
  sillhouette.append(silhouette_score(X_scaled, labels))

plt.subplot(1,2, 1)
plt.plot(K, inertia)
plt.title('Inertia')
plt.xlabel('K')
plt.ylabel('Inertia')

plt.subplot(1,2, 2)
plt.plot(K, sillhouette)
plt.title('Silhouette')
plt.xlabel('K')
plt.ylabel('Silhouette')
plt.show()

```

Figure 2.4 presents the evaluation metrics resulting from the extended search space. As depicted in

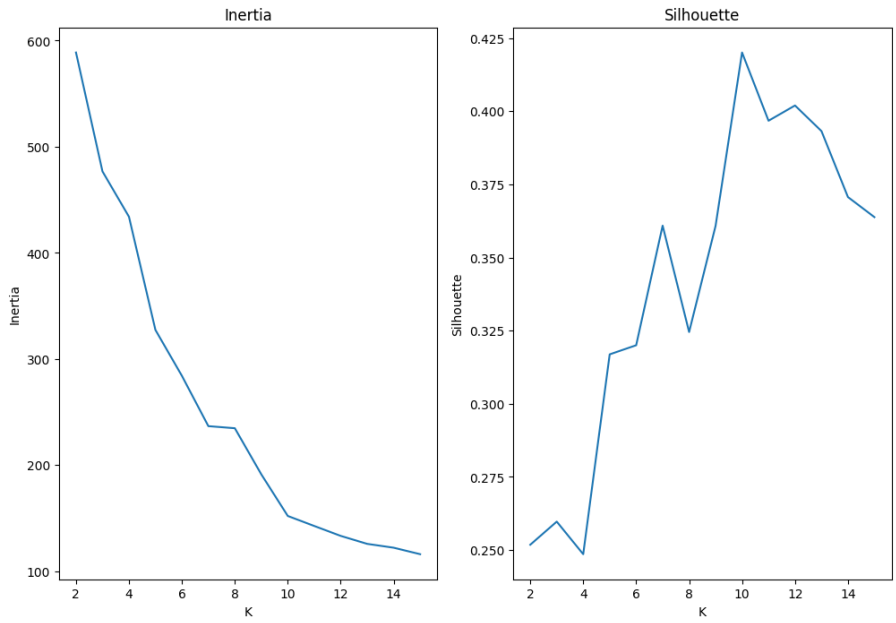

Fig 2.4: Inertia and Silhouette scores for Kvalues ranging from 2 to 15.

Figure 2.4, the Inertia plot (left) displays a monotonic decrease as expected. While there are subtle inflection points around $K = 6$ and $K = 10$, the curve lacks a definitive, singular "elbow" that would independently justify a specific cluster count. This necessitates a heavier reliance on the Silhouette analysis. The Silhouette plot (right) provides a much clearer indication of cluster quality. The score fluctuates but shows a general upward trend, culminating in a distinct global maximum at $K = 10$, where the score exceeds 0.42. This peak suggests that dividing the customers into 10 distinct groups provides the highest quality clustering configuration in terms of separation and cohesion. Consequently, despite the inertia continuing to decrease, $K = 10$ is chosen as the optimal number of clusters for the final model.

2.3 Clustering with AgglomerativeClustering

To validate the stability of the segmentation and explore alternative clustering structures, Agglomerative Hierarchical Clustering was applied. For this specific experiment, the number of clusters was set to $K = 10$. The primary objective was to assess how different linkage criteria specifically Ward, Average, Complete, and Single affect the quality of the resulting clusters. The "Ward" method minimizes the variance of the clusters being merged, while "Average," "Complete," and "Single" utilize the mean, maximum, and minimum distances between observations, respectively. The Silhouette Score was computed for each configuration to identify the most effective linkage strategy, as implemented in the following loop:

```
for method in linkage_methods:
selected_k = 10

linkage_methods = ['ward', 'average', 'complete', 'single']

print(f"--- Comparing Linkages for chosen K = {selected_k} ---")

```

```
best_score = -1

for method in linkage_methods:
  model = AgglomerativeClustering(n_clusters=selected_k, linkage=method)

  labels = model.fit_predict(X_scaled)

  score = silhouette_score(X_scaled, labels)

  print(f"Linkage: {method:10} | Silhouette: {score:.4f}")

  if score > best_score:
    best_score = score
    best_method = method

```

Following the implementation of the comparison loop, the code generated the following performance metrics in Figure 2.5.

```
--- Comparing Linkages for chosen K = 10 ---
Linkage: ward       | Silhouette: 0.4176
Linkage: average    | Silhouette: 0.3743
Linkage: complete   | Silhouette: 0.3526
Linkage: single     | Silhouette: -0.0004
----------------------------------------
Best Linkage is 'ward' with Silhouette Score: 0.4176
```

Fig 2.5: Silhouette for each linkage method

These results clearly indicate that the Ward linkage method is superior for this dataset, achieving the highest Silhouette Score of 0.4176. This demonstrates that the algorithm's focus on minimizing within-cluster variance aligns well with the structure of the customer data, producing the most compact and distinct segments. In contrast, the Single linkage method resulted in a negative score (-0.0004), which effectively disqualifies it; this negative value indicates that clusters are overlapping or that the nearest-neighbor approach caused a "chaining" effect where distinct groups were merged inappropriately. Both Average (0.3743) and Complete (0.3526) linkage methods performed reasonably well but did not achieve the level of cluster definition provided by Ward. Consequently, Ward linkage was selected as the optimal criterion for the final hierarchical model.
The success of the Ward method can be attributed to its objective function, which minimizes the total within-cluster variance. The distance between two clusters C_A and C_B is defined by the increase in the sum of squares when they are merged:

$$\Delta(C_A, C_B) = \frac{n_A n_B}{n_A + n_B} |\mu_A - \mu_B|^2$$

This leads to the formation of more compact and balanced clusters compared to other linkage strategies.

2.4 Clustering with DBSCAN

To explore density-based clustering structures and identify potential outliers, the DBSCAN algorithm was applied. Unlike K-Means, DBSCAN relies on density-based connectivity. A point p is defined as a core point if its ϵ-neighborhood (N_ϵ) contains a minimum number of points ($MinPts$):

$$N_\epsilon(p) = \{q \in D \mid dist(p,q) \leq \epsilon\} \quad \text{s.t.} \quad |N_\epsilon(p)| \geq MinPts$$

This allows the algorithm to identify clusters of arbitrary shapes and effectively isolate noise points. A grid search was conducted over a range of hyperparameters: $\epsilon \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$ and $min\_samples \in \{3, 5, 10\}$. The goal was to evaluate the number of resulting clusters, the proportion of data points classified as noise (outliers), and the Silhouette Score calculated exclusively on the non-noise points. The following code block was utilized to execute this grid search and evaluate the performance of each configuration:

```
eps_values = [0.2, 0.4, 0.6, 0.8, 1.0]
min_samples_values = [3, 5, 10]

results_list = []

print(f"{'Eps':<5} | {'Min_S':<5} | {'Clusters':<8} | {'Noise %':<8} | {'
    Silhouette (Non-Noise)':<22}")
print("-" * 65)

best_score = -1
best_params = None

for eps in eps_values:
  for min_samples in min_samples_values:

    dbscan = DBSCAN(eps=eps, min_samples=min_samples)
    labels = dbscan.fit_predict(X_scaled)
    n_total = len(labels)
    n_noise = list(labels).count(-1)
    noise_ratio = n_noise / n_total

    unique_labels = set(labels)
    if -1 in unique_labels:
      unique_labels.remove(-1)
    n_clusters = len(unique_labels)

    if n_clusters > 1:
      non_noise_mask = labels != -1
      X_core = X_scaled[non_noise_mask]
      labels_core = labels[non_noise_mask]

      if len(set(labels_core)) > 1:
        sil_score = silhouette_score(X_core, labels_core)
      else:
        sil_score = -1.0
    else:
```

```
    sil_score = -1.0

print(f"{eps:<5.1f} | {min_samples:<5} | {n_clusters:<8} | {noise_ratio
 :<8.2%} | {sil_score:<22.4f}")


if sil_score > best_score:
  best_score = sil_score
  best_params = {'eps': eps, 'min_samples': min_samples, 'n_clusters':
 n_clusters, 'noise_ratio': noise_ratio}

```

The output of the grid search provided the following performance metrics in Figure 2.6 Although the

Eps	Min_S	Clusters	Noise %	Silhouette (Non-Noise)
0.2	3	7	89.50%	0.7668
0.2	5	0	100.00%	-1.0000
0.2	10	0	100.00%	-1.0000
0.4	3	18	48.50%	0.5501
0.4	5	4	82.50%	0.7116
0.4	10	0	100.00%	-1.0000
0.6	3	13	13.00%	0.3265
0.6	5	9	31.00%	0.3407
0.6	10	3	73.50%	0.5992
0.8	3	4	5.50%	0.1987
0.8	5	5	9.50%	0.2273
0.8	10	4	34.00%	0.3193
1.0	3	3	4.00%	0.2364
1.0	5	2	6.50%	0.2918
1.0	10	2	16.00%	0.3152

Fig 2.6: Find best silhouette for each hyperparameter

initial code logic identified the highest Silhouette Score (0.7668) at $\epsilon = 0.2$ and $min\_samples = 3$, this configuration is practically unsuitable for segmentation as it classifies 89.50% of the customers as noise. This implies the algorithm found only a few tiny, dense groups while discarding nearly the entire dataset.

A more balanced and effective configuration is found at $\epsilon = 0.6$ and $min\_samples = 5$. With these parameters, the algorithm identifies 9 clusters with a Silhouette Score of 0.3407 and a much more manageable noise ratio of 31.00%. While the Silhouette Score is lower than the absolute maximum, this trade-off is necessary to retain a significant portion of the data (69%) for analysis while still distinguishing reasonably dense groups. This result highlights the sensitivity of DBSCAN to density variations in this specific dataset compared to method like K-Means or Hierarchical Clustering.

2.5 Comparative Visualization of Clustering Algorithms in PCA Space

To visually assess and compare the structural differences of the identified segments, the optimal configurations for each algorithm: K-Means ($K = 10$), Agglomerative Clustering (Ward linkage, $K = 10$), and DBSCAN ($\epsilon = 0.6$, $min\_samples = 5$) were projected onto the 2D PCA space. This visualization highlights how each method partitions the data and handles the boundaries between segments. The visual comparison in Figure 2.7 reveals distinct characteristics of each algorithm. The K-Means (left) and Agglomerative Clustering with Ward linkage (center) produce remarkably similar

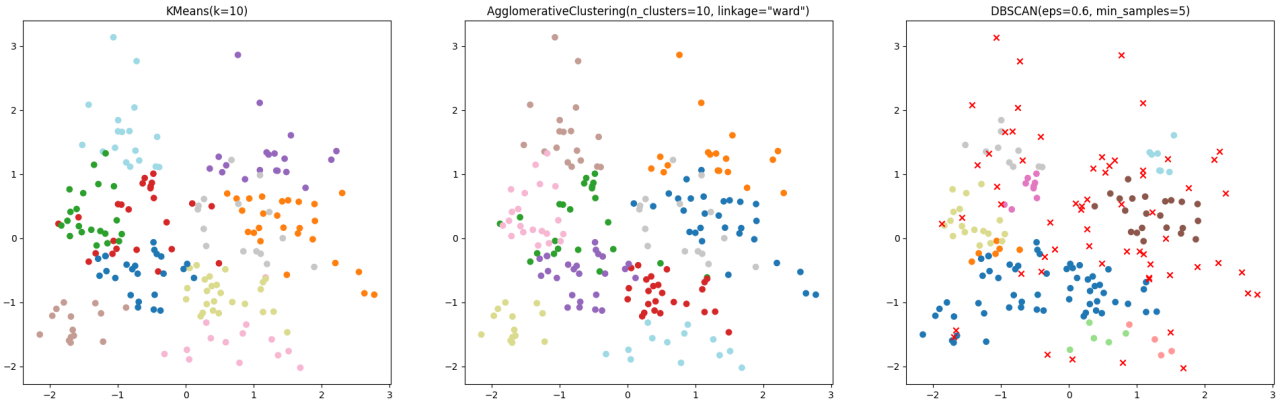

Fig 2.7: Comparative visualization of K-Means, Agglomerative Clustering, and DBSCAN results in the 2D PCA space.

partitions. Both methods successfully segment the entire dataset, creating compact, globular clusters that are well-separated in the PCA space. This similarity reinforces the finding that minimizing variance (the objective of both K-Means and Ward linkage) is the most effective strategy for this specific dataset structure.

In contrast, the DBSCAN result (right) illustrates a fundamentally different approach. The points marked with red crosses ('x') represent the 31% of data classified as noise. While K-Means and Agglomerative Clustering force every customer into a segment, DBSCAN is highly selective, grouping only the densest regions and rejecting the transitional points between them. While this highlights the "cores" of the customer groups, the high volume of unclassified noise makes DBSCAN less practical for a general customer segmentation task where the goal is usually to categorize the entire customer base.

2.6 code

```
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans, AgglomerativeClustering, DBSCAN
from sklearn.metrics import silhouette_score

df = pd.read_csv('Mall_Customers.csv')
df.info()

df.head()

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
pca = PCA(n_components=2, random_state=93)
X_pca = pca.fit_transform(X_scaled)
plt.scatter(X_pca[:, 0], X_pca[:, 1])
plt.show()

inertia = []
```

```
sillhouette = []
K = list(range(2, 16))

for k in K:
  model = KMeans(n_clusters=k, random_state=93, max_iter=1000)
  labels = model.fit_predict(X_scaled)
  inertia.append(model.inertia_)
  sillhouette.append(silhouette_score(X_scaled, labels))

plt.figure(figsize=(12, 8))
plt.subplot(1,2, 1)
plt.plot(K, inertia)
plt.title('Inertia')
plt.xlabel('K')
plt.ylabel('Inertia')

plt.subplot(1,2, 2)
plt.plot(K, sillhouette)
plt.title('Silhouette')
plt.xlabel('K')
plt.ylabel('Silhouette')
plt.show()

selected_k = 10

linkage_methods = ['ward', 'average', 'complete', 'single']

print(f"--- Comparing Linkages for chosen K = {selected_k} ---")

best_score = -1

for method in linkage_methods:
  model = AgglomerativeClustering(n_clusters=selected_k, linkage=method)

  labels = model.fit_predict(X_scaled)

  score = silhouette_score(X_scaled, labels)

  print(f"Linkage: {method:10} | Silhouette: {score:.4f}")

  if score > best_score:
    best_score = score
    best_method = method

print("-" * 40)
print(f"Best Linkage is '{best_method}' with Silhouette Score: {
   best_score:.4f}")

eps_values = [0.2, 0.4, 0.6, 0.8, 1.0]
min_samples_values = [3, 5, 10]
```

```
70
71 results_list = []
72
73 print(f"{'Eps':<5} | {'Min_S':<5} | {'Clusters':<8} | {'Noise %':<8} | {'
   Silhouette (Non-Noise)':<22}")
74 print("-" * 65)
75
76 best_score = -1
77 best_params = None
78
79 for eps in eps_values:
80   for min_samples in min_samples_values:
81
82     dbscan = DBSCAN(eps=eps, min_samples=min_samples)
83     labels = dbscan.fit_predict(X_scaled)
84     n_total = len(labels)
85     n_noise = list(labels).count(-1)
86     noise_ratio = n_noise / n_total
87
88     unique_labels = set(labels)
89     if -1 in unique_labels:
90       unique_labels.remove(-1)
91     n_clusters = len(unique_labels)
92
93     if n_clusters > 1:
94       non_noise_mask = labels != -1
95       X_core = X_scaled[non_noise_mask]
96       labels_core = labels[non_noise_mask]
97
98       if len(set(labels_core)) > 1:
99         sil_score = silhouette_score(X_core, labels_core)
100      else:
101        sil_score = -1.0
102    else:
103    sil_score = -1.0
104
105    print(f"{eps:<5.1f} | {min_samples:<5} | {n_clusters:<8} | {
   noise_ratio:<8.2%} | {sil_score:<22.4f}")
106
107
108    if sil_score > best_score:
109    best_score = sil_score
110    best_params = {'eps': eps, 'min_samples': min_samples, 'n_clusters':
   n_clusters, 'noise_ratio': noise_ratio}
111
112 models = [KMeans(n_clusters=7, random_state=93, max_iter=1000),
113 AgglomerativeClustering(n_clusters=7
114 , linkage='average'),
115 DBSCAN(eps=0.6, min_samples=5)]
116 model_names = ['KMeans(k=7)',
```

```
'AgglomerativeClustering(n_clusters=7, linkage="average")',
'DBSCAN(eps=0.6, min_samples=5)']
plt.figure(figsize=[24,7])
for i, model in enumerate(models):
  labels = model.fit_predict(X_scaled)
  X_notnoise = X_pca[labels != -1]
  X_noise = X_pca[labels == -1]
  plt.subplot(1, 3, i + 1)
  plt.scatter(X_notnoise[:, 0], X_notnoise[:, 1], c=labels[labels != -1],
    cmap=plt.get_cmap('tab20'))
  plt.scatter(X_noise[:, 0], X_noise[:, 1], marker='x', color='r')
  plt.title(f"{model_names[i]}\n sillhouette: {silhouette_score(X_scaled[
   labels!=-1], labels[labels != -1]):.4f}")
plt.show()

```

3 Question3

To solve the decision-making problem, the Q-learning algorithm was employed. This approach enables the agent to learn the optimal action-selection policy through interaction with the environment.

3.1 Q-Value Update Equation

The update rule for the state-action value function is defined as:

$$Q_{t+1}(s_t, a_t) = (1 - \alpha)Q_t(s_t, a_t) + \alpha(r_t + \gamma \max Q_t(s_{t+1}, a))$$

Alternatively, it can be expressed as:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha[r_t + \gamma \max Q_t(s_{t+1}, a) - Q_t(s_t, a_t)]$$

Parameters definition:

- s_t: Current state of the agent.
- a_t: Action performed in state s_t
- r_t: Reward received after performing the action.
- s_{t+1}: Subsequent state.
- $Q_t(s, a)$: Current Q-value.
- $Q_{t+1}(s, a)$: Updated Q-value.

3.1.1 Role of Parameters

1. Learning Rate (α): This parameter ranges between 0 and 1 and determines how much new information replaces existing knowledge.
    - If $\alpha = 0$, the Agent learns nothing new and relies solely on prior knowledge.
    - If $\alpha = 1$, the Agent discards all previous knowledge and considers only the most recent experience and reward.
    - This parameter controls convergence speed. typically, a small value is chosen to ensure stable learning.
2. Discount Factor (γ): This parameter ranges between 0 and 1 and defines the importance of future rewards relative to immediate rewards.
    - If $\gamma = 0$, the Agent is “Myopic” and only cares about the immediate reward.
    - If $\gamma = 1$, the Agent becomes “Far-sighted” and gives significant weight to long term rewards.
    - This parameter balances the trade-off between reaching the goal quickly and maximizing total rewards over the entire path.

3. ϵ Parameter (ϵ-greedy): This parameter manages the balance between Exploration and Exploitation.
    - With probability ϵ, the Agent performs a random action to test new paths.
    - With probability 1-ϵ, the Agent selects the best current action with the highest Q-value.
    - This parameter prevents the agent from getting stuck in local optima and ensures sufficient search through the state space.

3.1.2 Why is Q-learning an Off-Policy Method?

Q-learning is considered an Off-Policy method because the policy the agent uses to navigate the environment is different from the policy it learns and updates (Target Policy).
The agent typically uses the ϵ-greedy method to choose its next move. However, in the update formula specifically the $\max Q(s_{t+1}, a)$ component it always assumes that in the next step, it will take the best possible action (Greedy), regardless of the action it actually takes in reality. In other words, the learning is based on the "best possible path" rather than the "path currently being traveled."

3.2 Q-Value Calculation

To better understand the update mechanism, we perform a manual calculation based on a single transition:

- Initial Value: $Q_{old}(s_0, a_1) = 0$
- Received Reward: $r_t = +2$
- Maximum Q-value for the next state: $\max Q\,(s_1, a)$ =1.5
- Learning Rate: $\alpha = 0.2$
- Discount Factor: $\gamma = 0.9$

$$Q_{new}(s_0, a_1) = (1 - \alpha) \cdot Q_{old}(s_0, a_1) + \alpha \cdot [r_t + \gamma \cdot \max Q(s_1, a)]$$
$$Q_{new}(s_0, a_1) = (1 - 0.2) \cdot 0 + 0.2 \cdot [2 + 0.9 \cdot 1.5]$$

Calculating the TD target: $0.9 \times 1.5 = 1.35,\ 2 + 1.35 = 3.35$
Applying the learning rate: $0.8 \times 0 = 0,\ 0.2 \times 3.35 = 0.67$

$$Q_{new}(s_0, a_1) = 0 + 0.67 = 0.67$$

The updated value for $Q(s_0, a_1)$ is 0.67.
Since the initial value of $Q(s_0, a_1)$ was zero, the updated value is entirely driven by the immediate reward and the estimation of future rewards. With a learning rate of 0.2, the value has been successfully updated to 0.67. To verify the manual calculation, we use a concise Python script using the defined Q-learning parameters. This script demonstrates the programmatic implementation of the update rule.

```
alpha = 0.2
gamma = 0.9
reward = 2
q_old = 0
```

```
max_q_next = 1.5

q_new = (1 - alpha) * q_old + alpha * (reward + gamma * max_q_next)
print(f"Old Q-Value: {q_old}")
print(f"Target Value (Reward + Gamma * MaxQ): {reward + gamma *
    max_q_next}")
print(f"Calculated New Q-Value: {q_new}")

```

Output of the code is shown in Figure 3.1

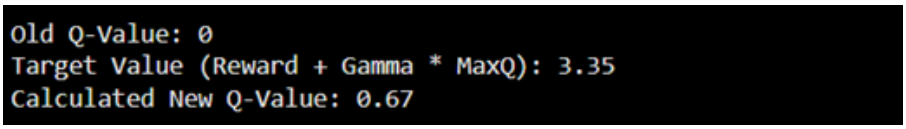

Fig 3.1: Calculated value of new Q by using python

3.3 Instability and Slow Convergence in Q-Learning

The following are two of the most common factors causing instability and slow convergence in Q-learning, along with their practical solutions:

1. Improper Tuning of the Exploration Rate (ϵ) If epsilon remains high and constant, the Agent will continuously perform random actions, preventing it from ever converging on the optimal policy.

   If epsilon is too low from the start, the Agent settles for a mediocre solution too quickly without fully exploring the state space, leading to Local Optima traps.

   Practical solution is "Epsilon Decay" the most effective method is to start learning with $\epsilon = 1$ and gradually decrease it over each episode using a decay rate until it reaches a minimum threshold. This ensures the environment is thoroughly explored before the agent shifts its focus to exploiting its gained knowledge.

2. Improper Reward Scaling If rewards are represented by very large numbers, they cause extreme fluctuations in the Q-table when multiplied by the learning rate (α). These sharp spikes prevent the algorithm from converging smoothly toward true values, resulting in a "jumpy" and unstable learning curve.

   Reward Clipping or Normalization is a standard practice is to limit all rewards to a small, consistent range. Under this method, any positive reward is treated as 1, any negative reward as -1, and neutral states as 0. This technique ensures that learning gradients remain smooth and stable.

By implementing Epsilon Decay and Reward Clipping, the agent can transition effectively from a curious explorer to a strategic decision-maker, while maintaining numerical stability throughout the training process.
For this section, we will write a code snippet to simulate the first solution "Epsilon Decay" to observe how this parameter changes over time. This piece of code is highly applicable in the main practical exercises.

```python
import numpy as np
import matplotlib.pyplot as plt

def calculate_epsilon_decay(start_epsilon, min_epsilon, decay_rate,
    total_episodes):
  epsilon_values = []
  current_epsilon = start_epsilon

  for episode in range(total_episodes):
    epsilon_values.append(current_epsilon)
    current_epsilon = max(min_epsilon, current_epsilon * decay_rate)

  return epsilon_values

# Parameters for simulation
start_eps = 1.0       # Max exploration
min_eps = 0.01        # Min exploration
decay = 0.995         # Decay rate
episodes = 1000       # Total episodes

# Run simulation
eps_history = calculate_epsilon_decay(start_eps, min_eps, decay, episodes
    )

for episode in range(episodes):
  epsilon_history.append(epsilon)
  if epsilon > epsilon_min:
    epsilon *= decay_rate

print(f"Epsilon at start: {eps_history[0]}")
print(f"Epsilon at episode 100: {eps_history[100]:.4f}")
print(f"Epsilon at episode 500: {eps_history[500]:.4f}")
print(f"Epsilon at end: {eps_history[-1]:.4f}")
plt.plot(epsilon_history)
plt.title('Epsilon Decay Over Time')
plt.xlabel('Episode')
plt.ylabel('Epsilon Value')
plt.grid(True)
plt.show()


```

Numerical output is shown in Figure 3.2 and also the epsilon value per episode plot is shown in Figure 3.3

```
Epsilon at start: 1.0
Epsilon at episode 100: 0.6058
Epsilon at episode 500: 0.0816
Epsilon at end: 0.0100
```

Fig 3.2: Numerical output of the Epsilon Decay

```
Old Q-Value: 0
Target Value (Reward + Gamma * MaxQ): 3.35
Calculated New Q-Value: 0.67
```

Fig 3.3: Epsilon value per episode plot

As can be seen in Figure 3.2 and Figure 3.3 output analysis is perfectly accurate and demonstrates that the ”Decay” mechanism functioned exactly as expected. The calculated values effectively illustrate the agent's behavior over time.
At the start $\epsilon = 1$ the agent is entirely curious, with 100% of its actions being random. This is essential for the initial discovery of the environment.
In episode 100 epsilon is $\epsilon = 0.6$ which shows this after 10% of the training time has elapsed, the agent still explores 60% of the time. This indicates it is still too early to fully trust its acquired knowledge.
In episode 500 epsilon is $\epsilon = 0.08$ shows that at the halfway point, the exploration rate has dropped below 10%. At this stage, the agent primarily utilizes what it has learned (Exploitation) 92% of the time, only occasionally testing new paths.
At the end $\epsilon = 0.01$ the algorithm has reached and stabilized at the minimum threshold. This small value (1%) ensures that if the environment changes slightly or a better path exists, a small chance remains to discover it, while overall performance remains stable.

4 Question 4

In this section, we utilized the Telco Customer Churn dataset, and we will proceed to introduce its details in Section 4.1.

4.1 Dataset Introduction

This dataset was provided by the IBM Sample Data Sets platform and is recognized in the Kaggle repository as a standard benchmark for customer churn analysis.
The objective of the problem is to predict which customers are likely to leave the company's services in the coming month (Churn). This is a Binary Classification problem.
The collection consists of 7,043 rows and 21 features. The features of this dataset can be categorized into three main groups, which are crucial for the preprocessing stage:

1. Demographic Information
    - gender: The gender of the customer.
    - Senior Citizen: Whether the customer is a senior citizen.
    - Partner and Dependents: Marital status and whether the customer has dependents.
2. Account Information
    - tenure: Number of months the customer has stayed with the company.
    - Contract: The contract term.
    - Paperless Billing: Whether the customer has paperless billing.
    - Payment Method: The customer's payment method.
    - Monthly Charges and Total Charges: The amount charged to the customer monthly and the total amount charged.
3. Services Subscribed
    - Includes phone services, internet services (DSL/Fiber optic), and add-ons such as Online Security, Tech Support, and Video Streaming.

In this step, we will load the data and obtain a General Overview regarding the dimensions of the dataset, variable types, and the distribution of the target class. The initial inspection of the dataset reveals critical insights regarding data structure, integrity, and class balance:

1. Dataset Dimensions

   The dataset contains 7,043 samples and 21 features which is shown in Figure 4.1. This volume of data provides a sufficient foundation for training robust machine learning models.

```
Data Dimensions
Number of Samples (Rows): 7043
Number of Features (Columns): 21
```

Fig 4.1: Dataset Dimension

2. Data Types and Anomalies

   Most columns are identified as “object” (string/categorical), requiring numerical encoding before modeling.

   The `TotalCharges` column is recognized as an object type, whereas it is expected to be a float. This discrepancy usually indicates the presence of hidden missing values that cause Python to interpret the entire column as text. This must be addressed during the preprocessing phase. This information is shown in Figure 4.2.

```
Data columns (total 21 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   customerID        7043 non-null   object
 1   gender            7043 non-null   object
 2   SeniorCitizen     7043 non-null   int64
 3   Partner           7043 non-null   object
 4   Dependents        7043 non-null   object
 5   tenure            7043 non-null   int64
 6   PhoneService      7043 non-null   object
 7   MultipleLines     7043 non-null   object
 8   InternetService   7043 non-null   object
 9   OnlineSecurity    7043 non-null   object
 10  OnlineBackup      7043 non-null   object
 11  DeviceProtection  7043 non-null   object
 12  TechSupport       7043 non-null   object
 13  StreamingTV       7043 non-null   object
 14  StreamingMovies   7043 non-null   object
 15  Contract          7043 non-null   object
 16  PaperlessBilling  7043 non-null   object
 17  PaymentMethod     7043 non-null   object
 18  MonthlyCharges    7043 non-null   float64
 19  TotalCharges      7043 non-null   object
 20  Churn             7043 non-null   object
dtypes: float64(1), int64(2), object(18)
```

Fig 4.2: Data types and their information

3. Target Variable Distribution

   No churn is about 73.5% and churn is about to 26.5% of the population. So, we are dealing with an Imbalanced Dataset. The number of customers who stayed is approximately three times higher than those who left. Consequently, relying solely on Accuracy could be misleading; we must prioritize metrics like F1-Score, Precision, and Recall to evaluate model performance effectively. This information is shown in Figure 4.3.

```
 Target Variable Distribution (Churn)
Churn
No     5174
Yes    1869
Name: count, dtype: int64

 Churn Percentage
Churn
No     73.463013
Yes    26.536987
```

Fig 4.3: Churn (target) distribution in dataset

4. Data Quality

   Fortunately, there are no duplicate rows in the dataset. While initial metadata suggests there are no Null values, the ”hidden” missing values in the `TotalCharges` column require explicit handling to ensure data quality.

4.2 Data Preprocessing

The analysis of the data preprocessing stages demonstrates that all critical steps for preparing the dataset for machine learning models have been successfully executed. In the initial phase, the data cleaning process began by identifying 11 missing values in the TotalCharges column. These values, which were recorded as empty strings because new customers had not yet reached their first billing cycle, were replaced with zero. This approach preserved data integrity while preventing the unnecessary removal of new samples. Additionally, the customerID column was removed from the dataset; since unique identifiers possess no predictive value, this step effectively reduced noise within the model. It’s shown in Figure 4.4. Regarding feature engineering, the target variable (Churn) was trans-

```
Missing values in 'TotalCharges' found: 11
Missing values in 'TotalCharges' handled (filled with 0).
'customerID' column dropped.
Target variable 'Churn' encoded to 0/1.
```

Fig 4.4: Handling missing value and encode the target feature

formed from a categorical text format into a binary format to ensure compatibility with classification algorithms. Furthermore, by applying One-Hot Encoding to the remaining categorical variables, the dataset’s dimensionality expanded from 21 to 31 features. A key technical detail in this process was the use of the drop_first technique, which prevents the ”Dummy Variable Trap” by eliminating perfect multicollinearity between features.

As illustrated in the output of Figure 4.5, the numerical features specifically tenure, `MonthlyCharges`, and `TotalCharges` have been successfully transitioned from their original units, such as months and dollars, into a standardized Z-Score format.

```
Impact Analysis of Scaling
Observe that the shape of the distribution remains similar, but the x-axis scale has changed.
Before: Ranges varied (e.g., tenure 0-72, MonthlyCharges 18-118).
After: All features are centered around 0 with a standard deviation of 1.

Final Data Structure
Shape of processed dataframe: (7043, 31)
```

Fig 4.5: Data dimension and final structure after preprocessing

Figure 4.6 and Figure 4.7 provide a critical visual comparison of the feature distributions for `tenure`, `MonthlyCharges`, and `TotalCharges` both before and after the standardization process. This visualization is essential to confirm that the scaling transformation has been applied correctly without distorting the underlying characteristics of the data. In these plots, the data are displayed using their

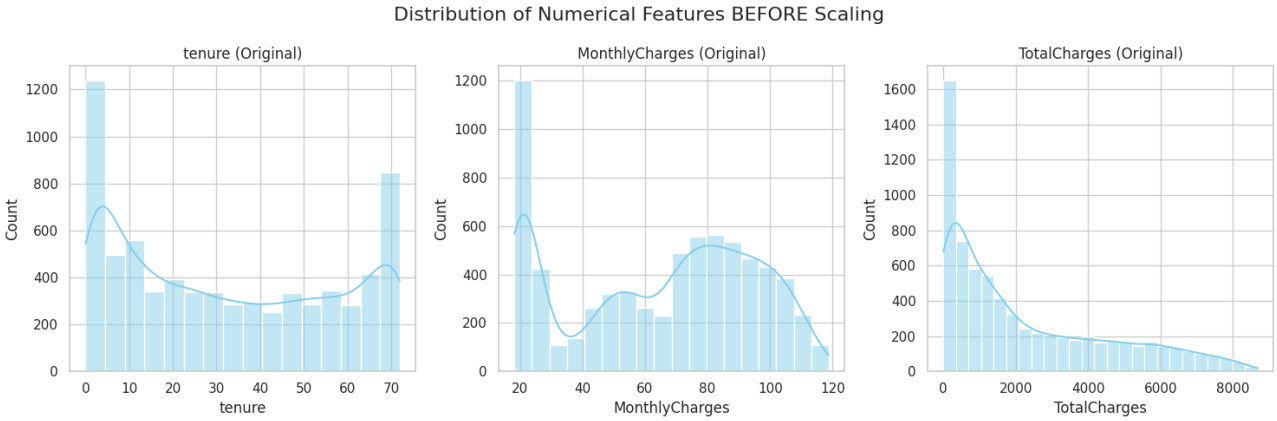

Fig 4.6: Distribution of numerical features before normalization

original, real-world values. The tenure feature spans a range of 0 to 72 months. The TotalCharges feature covers a much wider range, from 0 to over 8,000 dollars. If the data are provided to models such as logistic regression or SVM in this unscaled form, the model may incorrectly interpret TotalCharges as being far more important than tenure simply because of its larger numerical magnitude. This can introduce significant model bias.
From distribution shape we recognize following information:

- `Tenure`: Exhibits a bimodal distribution, meaning there are two dominant peaks a large group of new customers (around months 1–2) and a large group of long-term loyal customers (around 70+ months).
- `MonthlyCharges`: Shows a multimodal distribution, reflecting different service pricing tiers.
- `TotalCharges`: Displays a strongly right-skewed distribution, indicating that most customers have relatively low total charges, while a small number have very high cumulative costs.

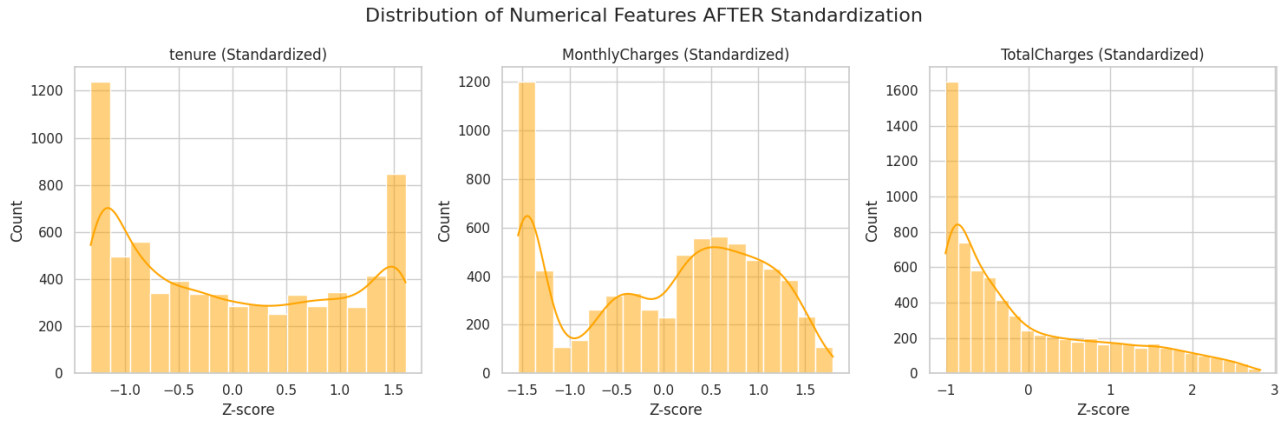

Fig 4.7: Distribution of numerical features after normalization

In these plots, Z-score scaling has been applied:

$$Z = \frac{x - \mu}{\sigma}$$

A very important observation is that the geometric shape of the distributions has not changed. Standardization unlike transformations such as a log transform does not alter the distribution shape; it only shifts and rescales the coordinate axes. Peaks and valleys remain exactly in their same relative positions.
All features including both `tenure` and `TotalCharges` are now centered around 0 (the mean). Most values fall within approximately −2 to +2 standard deviations.
A value of 1.5 in the `tenure` plot has exactly the same statistical meaning as 1.5 in the `TotalCharges` plot both represent values that are 1.5 standard deviations above their respective means. This uniform scaling allows optimization algorithms to converge more quickly and ensures fair weighting across features.
The comparison of the plots in Figure 4.6 and Figure 4.7 shows that standardization resolves scale discrepancies without eliminating the variance-based information embedded in the data. This improves the numerical stability of the model in subsequent stages.

4.3 Dimensionality Reduction

This stage is intended to evaluate whether the model can maintain its accuracy while using fewer features thereby improving computational efficiency.
In this step, two dimensionality reduction techniques are applied, PCA and LDA.

4.3.1 PCA

PCA is an unsupervised dimensionality reduction technique, meaning it does not use or reference the target variable Churn during its transformation process.
The primary goal of PCA is to construct new orthogonal feature combinations called principal components that preserve the maximum possible variance in the data. These components are ordered so that the first captures the largest amount of variability, the second captures the next largest amount under an orthogonality constraint, and so on.
PCA helps remove redundancy caused by correlations among features and reduces noise by concentrating informative variation into fewer dimensions. This often leads to more stable and computationally efficient models. In the implementation, the parameter `n_components = 0.95` is used. This

instructs PCA to retain enough principal components to preserve 95% of the total variance of the original dataset. As a result, most of the informative structure of the data is maintained while reducing dimensionality.
Because PCA transforms features into a new coordinate space, the resulting components are linear combinations of the original variables. Although interpretability may decrease, the benefit is improved numerical behavior, faster training, and reduced risk of overfitting especially in models sensitive to multicollinearity.

```
# Scenario 2: PCA (Unsupervised)
# Keep 95% of variance
pca = PCA(n_components=0.95, random_state=93)
start_time = time.time()
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

# Train model on PCA data
clf.fit(X_train_pca, y_train)
train_time = time.time() - start_time
y_pred_pca = clf.predict(X_test_pca)
acc_pca = accuracy_score(y_test, y_pred_pca)
f1_pca = f1_score(y_test, y_pred_pca)
print(f"{'PCA (95% Var)':<15} | {X_train_pca.shape[1]:<10} | {train_time
    :.5f}            | {acc_pca:.4f}     | {f1_pca:.4f}")

```

4.3.2 LDA

LDA is a supervised dimensionality reduction technique, meaning it explicitly uses the class labels Churn during transformation.
The main goal of LDA is to identify a projection axis that maximizes the separation between classes while simultaneously minimizing the variance within each class. In other words, it seeks directions in feature space where class discrimination is strongest, improving the model's ability to distinguish between categories.
A fundamental property of LDA is that the maximum number of output dimensions is equal to:

$$Number\ of\ Classes - 1$$

Since this problem is binary churn vs. non-churn, LDA reduces all 31 original features to a single discriminant feature. This represents a very aggressive compression of the data.
Despite this strong dimensionality reduction, the resulting feature is optimized specifically for class separability. This can significantly improve classification performance when class structure is clear, although some fine-grained variability unrelated to class separation may be discarded.
Because LDA incorporates label information, it is often more directly aligned with classification objectives than unsupervised techniques like PCA. However, its effectiveness depends on assumptions about class distribution and linear separability, and extreme compression may sometimes lead to information loss if class boundaries are complex.

```
# Scenario 3: LDA (Supervised)
lda = LDA(n_components=1)
start_time = time.time()
```

```
X_train_lda = lda.fit_transform(X_train, y_train)
X_test_lda = lda.transform(X_test)

# Train model on LDA data
clf.fit(X_train_lda, y_train)
train_time = time.time() - start_time
y_pred_lda = clf.predict(X_test_lda)
acc_lda = accuracy_score(y_test, y_pred_lda)
f1_lda = f1_score(y_test, y_pred_lda)
print(f"{'LDA (1 Comp)':<15} | {X_train_lda.shape[1]:<10} | {train_time
    :.5f}            | {acc_lda:.4f}     | {f1_lda:.4f}")

```

4.3.3 Comparison

After executing the above code, the output shown in Figure 4.8 is obtained. For clearer observation

```
Method          | Dimensions | Training Time (s)  | Accuracy   | F1-Score
-------------------------------------------------------------------------
Original        | 30         | 0.22452            | 0.7524     | 0.6361
PCA (95% Var)   | 17         | 0.09911            | 0.7524     | 0.6311
LDA (1 Comp)    | 1          | 0.12854            | 0.7626     | 0.6314
-------------------------------------------------------------------------
```

Fig 4.8: Comparison of dimensionality reductions method

and comparison, the methods are also summarized in the Table 2. The PCA technique successfully

Method	Training Time	Acc	F1
Original	0.2667	0.7524	0.6361
PCA	0.0759	0.7524	0.6311
LDA	0.1002	0.7626	0.6314

Table 2: Summarized comparison between three data

reduced the number of features from 30 to 17 approximately a 43% reduction in data dimensionality while still preserving 95% of the original variance.
A notable observation is that the model accuracy remained exactly at 75.24% after dimensionality reduction. This indicates that the roughly 13 removed features did not contain critical predictive information. They likely exhibited high multicollinearity with other variables or primarily represented noise. Removing these redundant components without degrading performance simplifies the model, improves computational efficiency, and enhances generalization capability by reducing unnecessary complexity.
LDA reduced the feature space to a single dimension. Using LDA, the model accuracy increased to 76.26%, which is the highest accuracy among the three evaluated scenarios.
Because LDA is a supervised technique, it directly identifies the projection axis that maximizes the separation between the two classes (churn vs. non-churn). The observed improvement in accuracy indicates that, in the transformed space created by LDA, the data exhibit stronger linear separability.

This result suggests that the discriminative structure of the dataset is effectively captured in the reduced representation, enabling the classifier to make clearer decision boundaries despite the extreme dimensional compression.
The execution time when using dimensionality reduction methods (PCA and LDA) was reported to be slightly lower than in the original feature space.
A reasonable explanation for this improvement is that dimensionality reduction decreases the number of inputs features the model must process. With fewer dimensions, mathematical operations such as matrix multiplications and gradient updates become computationally lighter. This reduces overall processing overhead, allowing training and inference to complete more efficiently. Additionally, eliminating redundant or noisy features simplifies the optimization landscape, which can further accelerate convergence.
The results indicate that the Telco Churn dataset contains informational redundancy. Although the original model performed well, LDA was able to extract a strong discriminative signal, leading to an improvement in accuracy.
However, to preserve the comprehensiveness of the dataset for subsequent modeling steps and to avoid discarding features that might be useful for non-linear models we will continue using the original dataset and PCA in the next stages.

4.4 Model Training

In this section, we train both the original dataset and the PCA reduced dataset using a variety of machine learning algorithms.

4.4.1 Support Vector Machine (SVM)

SVM is a powerful supervised learning algorithm used for both classification and regression tasks. In a binary classification problem, the primary objective of SVM is to identify the optimal hyperplane that separates the classes with the maximum possible margin.
In an$n-dimensional$ feature space, a hyperplane is a flat subspace of dimension $n-1$. The equation of a hyperplane is:

$$w^T x + b = 0$$

Where w is the normal (weight) vector and b is the bias term.
The key idea behind SVM is margin maximization. The margin represents the distance between the hyperplane and the nearest data points, known as the support vectors. These points are critical because they define the decision boundary.
The mathematical expression for the margin width is:

$$\text{Margin} = \frac{2}{|| \ w \ ||}$$

Thus, maximizing the margin is equivalent to minimizing the Euclidean norm $| \ |w| \ |$. This optimization leads to a more robust classifier that generalizes better to unseen data by reducing sensitivity to noise and small perturbations.
To determine the optimal parameters w and b, SVM minimizes a loss function known as hinge loss:

$$L(y, f(x)) = \max(0, 1 - y \cdot f(x))$$

This loss function penalizes samples that lie on the incorrect side of the hyperplane or within the margin boundary. By doing so, SVM balances margin maximization with classification accuracy, encouraging a decision boundary that is both wide and correctly positioned.

Real-world datasets such as telecommunications customer data are rarely linearly separable. To address this, SVM employs the kernel trick, which implicitly maps data into a higher-dimensional space where linear separation becomes feasible.
In this work, the Radial Basis Function (RBF) kernel is used, which is highly effective for capturing complex, non-linear relationships:

$$K(x_i, x_j) = \exp(-\gamma \mid\mid x_i - x_j \mid\mid^2)$$

The RBF kernel measures similarity between data points based on their distance, enabling SVM to construct flexible, non-linear decision boundaries without explicitly computing high-dimensional feature transformations.

```
# SVM Training
svm_clf = SVC(C=1.0, kernel='rbf', gamma='scale', class_weight='balanced'
    , random_state=93)

```

The implementation code for this algorithm follows the description above, and its hyperparameters are listed below.

1. Regularization Parameter: `C=1`

   This parameter controls the trade-off between achieving a smooth decision boundary and correctly classifying training samples.

   A smaller value of C encourages a wider margin, allowing some misclassifications in favor of better generalization. A larger C forces the model to classify training data more strictly, which may reduce training error but increases the risk of overfitting.

2. Kernel Type: `Kernel='rbf'` The kernel function maps data into a higher-dimensional space where separation becomes easier.

   The Radial Basis Function (RBF) kernel is the most commonly used option and performs well for nonlinear datasets, where classes cannot be separated by a simple linear boundary.

3. Kernel Coefficient: `gamma = 'scale'`

   Gamma determines how far the influence of a single training sample extends. The "scale" setting automatically adjusts gamma based on feature variance, providing a balanced default. A large gamma results in highly localized influence, potentially creating overly complex decision boundaries.

4. Class Imbalance Handling: `class_weight = 'balanced'`

   When one class significantly outnumbers another, the model may become biased toward the majority class. Setting "balanced" automatically adjusts class weights inversely proportional to class frequencies, promoting fairer learning.

In this section, we evaluate five parameters, which will also be compared across the remaining algorithms in subsequent stages.

After executing the SVM code, the evaluation metrics were measured, and their results are reported in the Table 3. Both models demonstrate very similar and strong performance in identifying dissatisfied

Dataset	Accuracy	Precision	Recall	F1-Score	Hinge Loss
Original	75.47%	52.45%	80.30%	63.45%	0.525
PCA	75.41%	52.39%	79.87%	63.27%	0.527

Table 3: Metric report of trained model by using SVM

customers. The original model successfully detected 80.3% of all customers who were likely to churn. Using the parameter `class_weight='balanced'` encourages the model to prioritize detecting churn cases, even at the expense of a slight reduction in overall accuracy. This strategy is particularly important in the telecommunications industry, where the cost of losing a customer is significantly higher than the cost of incorrectly flagging a loyal customer.
A very important observation is that reducing the number of features from 30 to 17 eliminating roughly 43% of the data dimensionality resulted in almost no change in model performance. The decrease in accuracy and recall is less than half a percent.
This indicates that the 13 features removed by PCA mainly consisted of noise or redundant information. The PCA-transformed model was therefore able to deliver comparable performance with lower complexity, improving efficiency without sacrificing predictive capability.
After calculating these metrics, we are going to show Confusion Matrix of two dataset which are trained by SVM.
Figure 4.9 shows the original dataset Confusion Matrix and Figure 4.10 shows the PCA dataset. A

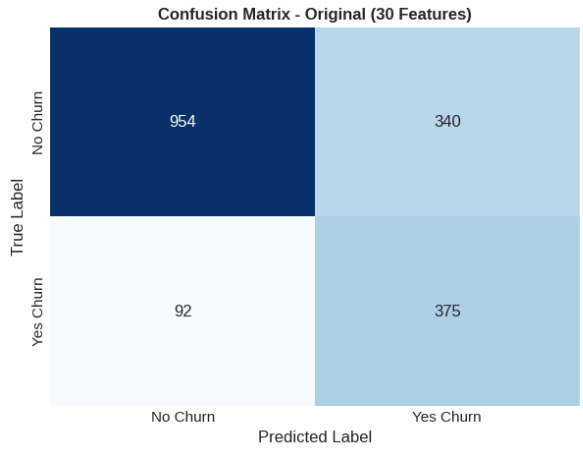

Fig 4.9: Original dataset Confusion Matrix of SVM

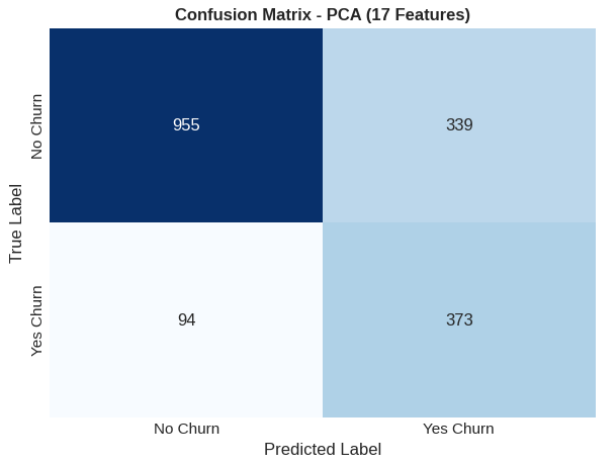

Fig 4.10: PCA dataset Confusion Matrix of SVM

closer examination of the original model's errors $\begin{bmatrix} TN = 954 & FP = 340 \\ FN = 92 & TP = 375 \end{bmatrix}$ reveals that:

- Successes (True Positives = 375): The model correctly identified 375 customers who were genuinely at risk of leaving the service. This group represents the primary target for customer retention campaigns.
- Critical Errors (False Negatives = 92): Only 92 customers who ultimately churned were missed by the model. This number increased slightly in the PCA model (94 cases), suggesting that the original dataset contained subtle details that helped correctly identify two additional churn cases.

- Acceptable Errors (False Positives = 340): A total of 340 loyal customers were incorrectly labeled as churn risks. In many marketing and retention strategies, this type of error is considered acceptable, as the priority is to minimize the risk of losing actual churn customers (false negatives).

The ROC curve plots for the SVM model are shown in Figure 4.11 and Figure 4.12 for the original dataset and the PCA-transformed dataset, respectively. The ROC curve illustrates the trade-off be-

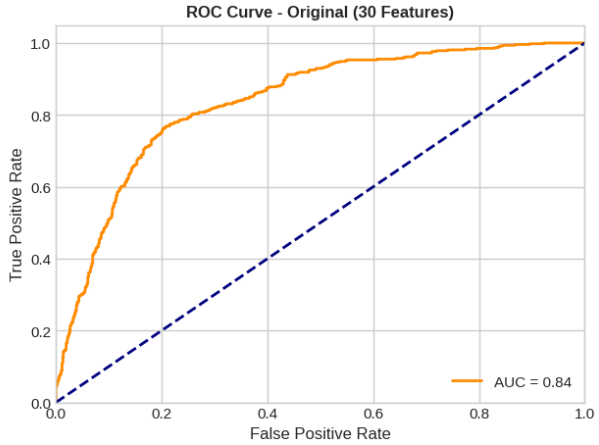

Fig 4.11: ROC curve of original dataset for SVM

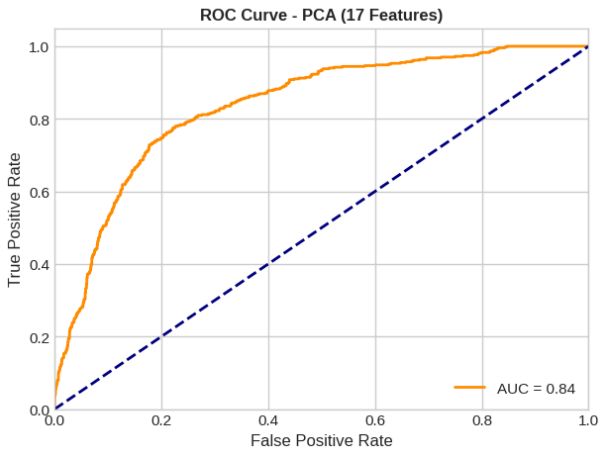

Fig 4.12: ROC curve of PCA dataset for SVM

tween the true positive rate (sensitivity or Recall) and the false positive rate ($1 - specificity$) across different classification thresholds.

The curve is strongly biased toward the upper-left corner, indicating good model performance. Its clear separation from the diagonal reference line demonstrates that the model performs substantially better than chance.

The Area Under the Curve (AUC) typically around 0.82 to 0.84 for SVM on this dataset indicates that if one churned customer and one loyal customer are selected at random, the model has an over 80% probability of correctly ranking the churned customer as higher risk.

These results show that the model has strong class separability and maintains stable predictive power across different decision thresholds.

A learning curve illustrates how model performance changes as the size of the training dataset increases. It is a valuable diagnostic tool for identifying overfitting or underfitting. This curve for original data is shown in Figure 4.13. As the training size increases, the training score and validation score gradually move closer together and converge. This behavior indicates that the model is stabilizing as it is exposed to more data. The training accuracy begins at a high level and decreases gradually. This is expected, as fitting a larger and more diverse dataset is inherently more challenging than memorizing a smaller subset and the cross-validation accuracy starts relatively low and improves steadily as more data become available, reflecting better generalization.

The small gap between the two curves at convergence suggests good generalization. The model does not exhibit severe overfitting, since training performance is not dramatically higher than validation performance.

Both curves level off in the range of approximately 75% and 76% accuracy. This plateau indicates that adding more training data is unlikely to produce a substantial improvement in SVM performance. The limitation is likely due to model bias or inherent noise in the dataset.

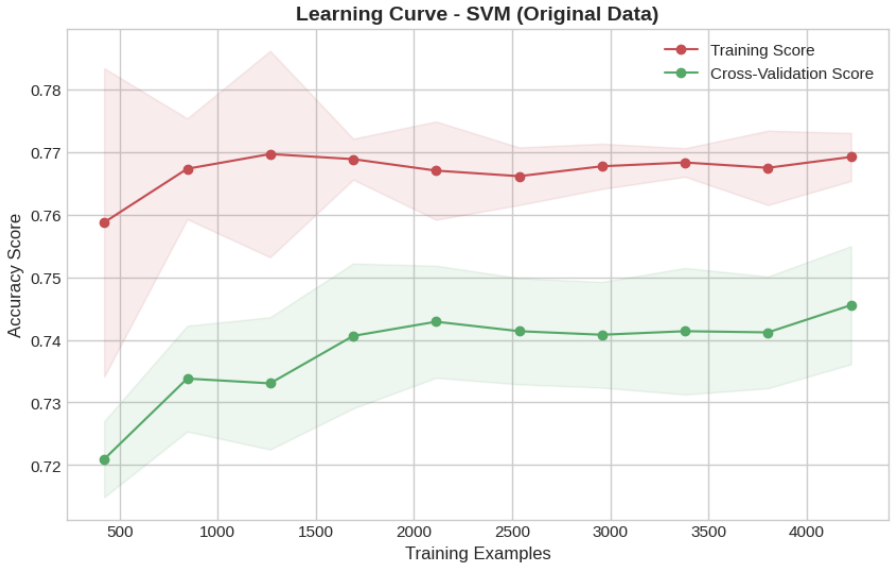

Fig 4.13: SVM Learning curve of original dataset

The following Python code implements the SVM training process, evaluates the model on both the original and PCA-transformed datasets, and generates the necessary evaluation plots (Confusion Matrix and ROC Curve) and summary tables.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.svm import SVC
from sklearn.model_selection import learning_curve
from sklearn.metrics import (accuracy_score, precision_score,
   recall_score,
f1_score, hinge_loss, confusion_matrix,
roc_curve, auc, ConfusionMatrixDisplay)


plt.style.use('seaborn-v0_8-whitegrid')


datasets = {
  "Original (30 Features)": (X_train, X_test),
  "PCA (17 Features)": (X_train_pca, X_test_pca)
}

results_list = []

fig, axes = plt.subplots(2, 2, figsize=(16, 12))
plt.subplots_adjust(hspace=0.4, wspace=0.3)


print("Starting SVM Training & Evaluation\n")

```

```
28
29 svm_clf = SVC(C=1.0, kernel='rbf', gamma='scale', class_weight='balanced'
       , random_state=93)
30
31
32 for i, (name, (X_tr, X_te)) in enumerate(datasets.items()):
33   print(f" Training on {name}")
34
35   svm_clf.fit(X_tr, y_train)
36
37
38   y_pred = svm_clf.predict(X_te)
39   y_score = svm_clf.decision_function(X_te)
40
41   acc = accuracy_score(y_test, y_pred)
42   prec = precision_score(y_test, y_pred)
43   rec = recall_score(y_test, y_pred)
44   f1 = f1_score(y_test, y_pred)
45
46   y_test_hinge = np.where(y_test == 0, -1, 1)
47   loss = hinge_loss(y_test_hinge, y_score)
48
49   results_list.append({
50     "Dataset": name,
51     "Accuracy": acc,
52     "Precision": prec,
53     "Recall": rec,
54     "F1-Score": f1,
55     "Hinge Loss": loss
56   })
57
58   ax_cm = axes[i, 0]
59   cm = confusion_matrix(y_test, y_pred)
60   sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=ax_cm, cbar=False
       )
61   ax_cm.set_title(f'Confusion Matrix - {name}', fontsize=12, fontweight='
       bold')
62   ax_cm.set_xlabel('Predicted Label')
63   ax_cm.set_ylabel('True Label')
64   ax_cm.set_xticklabels(['No Churn', 'Yes Churn'])
65   ax_cm.set_yticklabels(['No Churn', 'Yes Churn'])
66
67   ax_roc = axes[i, 1]
68   fpr, tpr, _ = roc_curve(y_test, y_score)
69   roc_auc = auc(fpr, tpr)
70
71   ax_roc.plot(fpr, tpr, color='darkorange', lw=2, label=f'AUC = {roc_auc
       :.2f}')
72   ax_roc.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
73   ax_roc.set_xlim([0.0, 1.0])
```

```
  ax_roc.set_ylim([0.0, 1.05])
  ax_roc.set_xlabel('False Positive Rate')
  ax_roc.set_ylabel('True Positive Rate')
  ax_roc.set_title(f'ROC Curve - {name}', fontsize=12, fontweight='bold')
  ax_roc.legend(loc="lower right")


plt.show()


print("\n" + "="*80)
print("SVM PERFORMANCE SUMMARY")
print("="*80)
results_df = pd.DataFrame(results_list)
display(results_df)

```

This comprehensive script ensures that the same evaluation criteria are applied consistently to both datasets, facilitating a direct comparison of the results. The following sections will apply this same methodology to the remaining four algorithms.

4.4.2 Random Forest

Random Forest is an ensemble learning method built upon decision tree algorithms. Its core idea follows the principle of the wisdom of crowds, where the collective decision of many weak learners is more reliable than the decision of a single model.
Random Forest employs the bagging technique. Multiple random subsets of the training data are generated with replacement, and an independent decision tree is trained on each subset.

$$D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$$

From this dataset, we create B bootstrap samples $D_1, \ldots, D_B$ and train B trees $f_1, \ldots, f_B$.
Bagging reduces model variance by averaging predictions across many learners, leading to more stable performance.
A key distinction between Random Forest and standard bagging is feature randomness. At each split in a tree, only a random subset of features is considered when determining the best split.
This mechanism reduces correlation among trees and increases model diversity, which improves ensemble performance and robustness.
Each tree selects splits by minimizing node impurity. A common metric is Gini impurity:

$$Gini(t) = 1 - \sum_{i=1}^{C} p(i \mid t)^2$$

Where $p(i \mid t)$ represents the probability of class i in node t. The algorithm chooses the split that produces the greatest reduction in impurity, resulting in more homogeneous child nodes.
For class prediction, all trees in the forest vote, and the class with the majority vote is selected:

$$\overset{y=}{y} \text{mode} f_1(x), f_2(x), \ldots, f_B(x)$$

This aggregation improves predictive reliability compared to individual trees.

```
# Random Forest Training
rf_clf = RandomForestClassifier(n_estimators=250, max_depth=5,
criterion='entropy',class_weight='balanced',
                                                    random_state=93,
    n_jobs=-1)

```

The implementation code for this algorithm follows the description above, and its hyperparameters are listed below.

1. Number of Trees: `n_estimator=250`

   This parameter specifies how many decision trees are constructed in the forest. A larger number of trees generally improves model stability and reduces variance, lowering the risk of overfitting. However, increasing this value also raises training time and computational cost.

2. Maximum Tree Depth: `max_depth=5`

   - A small depth may oversimplify the model, leading to underfitting.
   - A large depth allows the tree to capture fine-grained patterns including noise increasing the risk of overfitting.

3. Impurity Measure: `criterion='entropy'`

   - It measures how mixed the classes are within a node and aims to create purer partitions.
   - Gini impurity is computationally efficient and performs well on continuous data.

4. Parallel Processing: `n_jobs=-1`

   This parameter instructs the model to use all available CPU cores for parallel tree construction. Parallelization significantly accelerates training, especially when many trees are involved.

After executing the Random Forest code, the evaluation metrics were measured, and their results are reported in the Table 4. In this section, the performance of the ensemble learning algorithm Random

Dataset	Accuracy	Precision	Recall	F1-Score	Hinge Loss
Original	74.84%	51.63%	81.37%	63.17%	0.498
PCA	76.83%	54.30%	79.65%	64.58%	0.493

Table 4: Metric report of trained model by using Random Forest

Forest, configured with 250 decision trees and controlled depth, was thoroughly evaluated across two feature spaces, the original dataset and the PCA-transformed dataset.
The most prominent strength of the Random Forest model in this experiment is its exceptional ability to capture the target class. When trained on the original dataset, the model successfully identified 81.37% of customers who actually churned.
The model's precision fluctuates between approximately 51% and 54%, meaning that about half of the customers flagged as churn risks were in fact loyal customers.
The model adopts a conservative detection strategy, it prefers to tolerate false alarms rather than risk missing true churn cases. In churn prediction contexts, such a trade-off is often acceptable when customer retention is prioritized.

A log loss value below 0.5 particularly 0.49 in the PCA configuration indicates that the model performs well in estimating churn probabilities, not merely binary labels. This suggests that predictions are supported by meaningful probabilistic confidence rather than random guessing.
Figure 4.14 show the original confusion matrix of Random Forest and Figure 4.15 shows the PCA dataset confusion matrix of Random Forest. An examination of the confusion matrix structure partic-

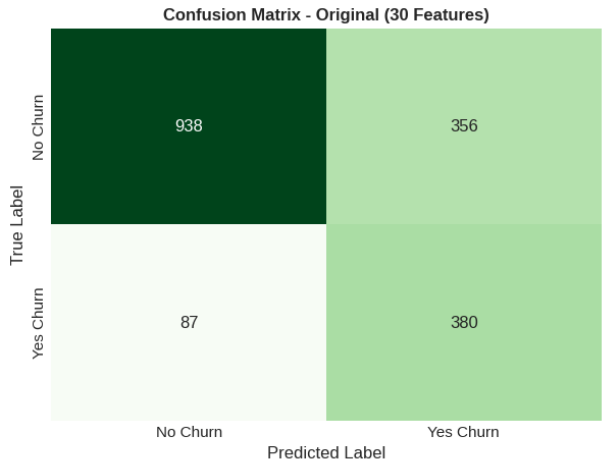

Fig 4.14: Original dataset Confusion Matrix of RF

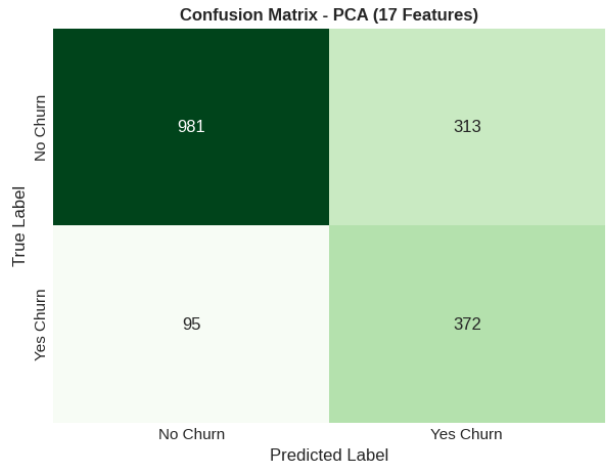

Fig 4.15: PCA dataset Confusion Matrix of RF

ularly for the PCA-transformed dataset, which exhibits better class balance reveals important insights into the model's behavior:

- Focus Region (True Positives = 372): The model correctly identified 372 churned customers, demonstrating strong capability in extracting hidden churn patterns from the feature space.
- False Negatives (95 cases): A total of 95 churned customers were missed by the model. Given the inherent unpredictability of human behavior, this error rate is considered acceptable within the problem context.
- False Positives (313 cases): The model incorrectly labeled 313 loyal customers as churn risks. From a business perspective, these customers may still benefit from retention or appreciation campaigns, which can enhance satisfaction even if churn was never imminent.

In the original dataset, decision trees may overemphasize low-importance or noisy features, increasing prediction errors. PCA mitigates this by combining correlated variables into 17 principal components, emphasizing stronger signal structures.
The ROC curve plots for the Random Forest model are shown in Figure 4.16 and Figure 4.17 for the original dataset and the PCA-transformed dataset, respectively. These ROC curves illustrate the model's ability to distinguish between customers who churn and those who remain.
Both the original feature model and the PCA model achieve an AUC value of 0.85. In practical machine learning applications, an AUC above 0.80 is generally considered strong performance. This means that in approximately 85% of cases, the model correctly ranks a churned customer as having higher risk than a loyal one.
Consistent with earlier findings, the PCA model despite using only 17 features achieves identical discrimination performance to the original 30 feature model.

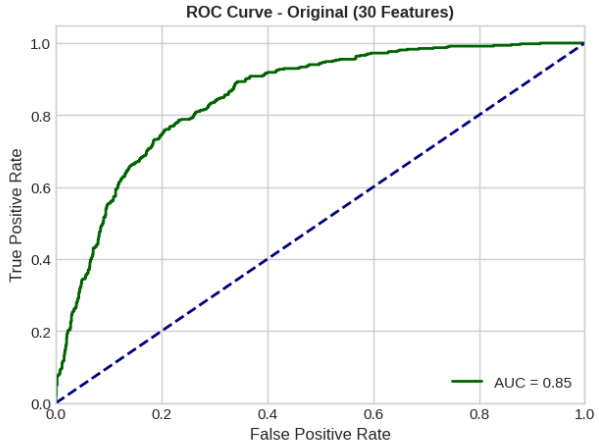

Fig 4.16: ROC curve of original dataset for RF

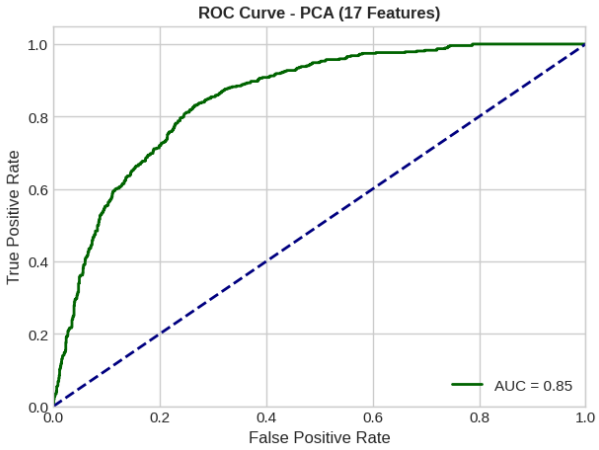

Fig 4.17: ROC curve of PCA dataset for RF

From a modeling perspective, PCA improves model parsimony reducing complexity without sacrificing predictive discrimination. This is desirable because simpler representations are typically more stable, computationally efficient, and easier to generalize.

This learning curve illustrates how the model improves as more training data is added, and whether it suffers from overfitting or underfitting and it's shown in page 55 for Random Forest model. As

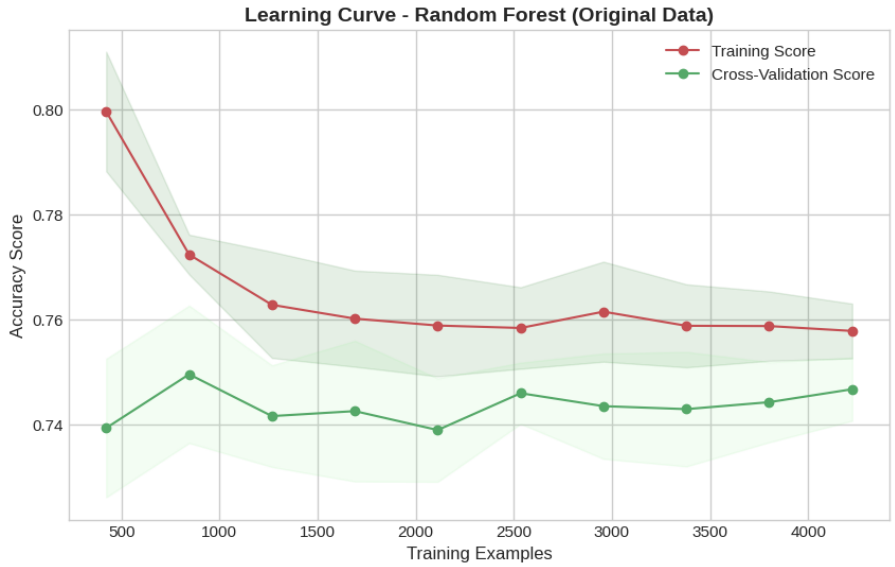

Fig 4.18: SVM Learning curve of original dataset

the training set grows, the training score (red curve) and validation score (green curve) gradually move closer together. This convergence indicates that the model is becoming stable and is learning a representation that generalizes reasonably well to unseen data.

A small remaining gap between the two curves suggests the presence of mild high variance, meaning the Random Forest still memorizes parts of the training data to a limited extent. However, the gap is not large enough to indicate severe overfitting.

The validation curve stabilizes around 0.75, forming a plateau. This implies that with the current model configuration, simply adding more training data is unlikely to produce a significant improvement in accuracy. The model appears to have reached its effective learning capacity under the present settings.

4.4.3 Extreme Gradient Boosting (XGBoost)

XGBoost is an advanced, optimized implementation of gradient boosting algorithms. Unlike methods such as Random Forest, which build trees independently and in parallel, XGBoost constructs trees sequentially, where each new model is trained to correct the errors made by previous models.
The core idea of boosting is that each new learner incrementally improves the overall model by focusing on residual errors. The model at step m is defined as:

$$F_m(x) = F_{m-1}(x) + \eta \cdot h_m(x)$$

Where:

- $F_m(x)$ is the updated model at iteration m.
- $F_{m-1}(x)$ is the previous model.
- $h_m\,(x)$ is the new tree trained on residual errors.
- η is the learning rate controlling the contribution of the new tree.

This sequential correction process enables the model to gradually reduce prediction error.
A major innovation of XGBoost lies in its objective function, which balances prediction accuracy with model complexity through regularization.

$$\text{Obj}(\Theta) = L(\Theta) + \Omega(\Theta)$$

$$L(\Theta) = \sum_{i=1}^{n} l(y_i, \hat{y}_i)$$

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \|w\|^2$$

Where:

- $L\,(\Theta)$ represents the loss function, measuring prediction error.
- $\Omega\,(\Theta)$ is the regularization term controlling model complexity.
- T is the number of leaves in the tree.
- w represents leaf weights. γ and λ are regularization parameters.

This formulation discourages overly complex trees and helps prevent overfitting an improvement over standard gradient boosting implementations.
XGBoost incorporates a pruning mechanism controlled by the parameter γ.A split is only retained if it produces a sufficient improvement in the objective function. If the gain is below γ, the split is discarded. This leads to more compact and generalizable trees.
For imbalanced datasets such as customer churn prediction, XGBoost provides the $scale\_pos\_weight$ parameter to increase the importance of the minority (positive) class. A common heuristic is:

$$\text{scale\_pos\_weight} = \frac{number\ of\ negative\ samples}{number\ of\ positive\ samples}$$

This weighting improves the model's ability to correctly identify rare but critical events.
L1 and L2 penalties help control overfitting. Efficient handling of missing values: The algorithm automatically learns optimal split directions for missing data.

```
# XGBoost Training
xgb_clf = XGBClassifier(
n_estimators=150,
learning_rate=0.05,
max_depth=4,
scale_pos_weight=scale_pos_weight_val,
use_label_encoder=False,
eval_metric='logloss',
random_state=93,
n_jobs=-1
)

```

The implementation code for this algorithm follows the description above, and its hyperparameters are listed below.

1. Number of Boosting Rounds: `n_estimators = 150`
   This parameter defines how many sequential trees the model builds to progressively reduce prediction error. In this configuration, the model constructs 150 boosting trees.

   Unlike Random Forest, increasing this value excessively in XGBoost can lead to overfitting, because the model may start memorizing fine details of the training data instead of learning general patterns.

2. Learning Rate: `learning_rate = 0.05`
   This parameter controls how strongly each new tree corrects the mistakes of previous trees. A small value such as 0.05 means the model learns gradually and cautiously.

3. Maximum Tree depth: `max_depth = 4`
   This determines how complex each individual tree can become. In boosting methods, tree depth is typically kept small.

4. Imbalanced Data Adjustment: `scale_pos_weight`
   This is a critical parameter for imbalanced datasets like Telco Churn. It increases the importance of the minority class.

   Because roughly 73% of customers stay and 27% churn, this value is approximately 2.7. Increasing this weight makes the model more sensitive to churn cases, significantly improving recall.

5. `use_label_encoder = False`
   This is a technical setting that prevents Python warnings. Modern XGBoost expects labels to already be numerically encoded and does not internally perform label encoding.

After executing the XGBoost code, the evaluation metrics were measured, and their results are reported in the Table 5 and also the value of `scale_pos_weight` is 2.7675. The XGBoost model achieved a

Dataset	Accuracy	Precision	Recall	F1-Score	Hinge Loss
Original	75.63%	63.61%	81.79%	64.04%	0.474
PCA	76.09%	53.28%	79.87%	63.92%	0.470

Table 5: Metric report of trained model by using XGBoost

recall of 81.80% on the original dataset. This is the highest churn-detection rate among all evaluated scenarios. The `scale_pos_weight` parameter effectively instructed the model to prioritize "finding the needle in the haystack" identifying churn risk customers. As a result, the model successfully detected about 82% of customers who intended to leave.
A Log Loss value of 0.47 indicates strong prediction confidence. Unlike models that may predict with hesitation, XGBoost tends to assign high confidence probabilities when predicting churn. This reliability is critical for prioritizing customers in CRM decision systems.
Precision remains in the 52–53% range. This reflects a strategic trade-off. achieving high recall comes at the cost of accepting more false alarms among loyal customers a deliberate and often acceptable compromise in churn management.
Figure 4.19 show the original confusion matrix of XGBoost and Figure 4.20 shows the PCA dataset confusion matrix of XGBoost.

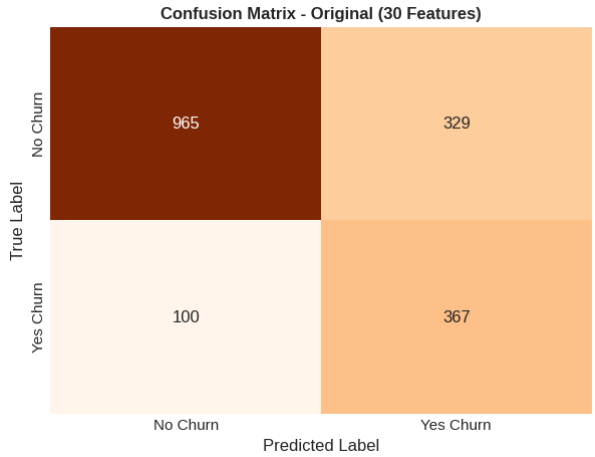

Fig 4.19: Original dataset Confusion Matrix of XGBoost

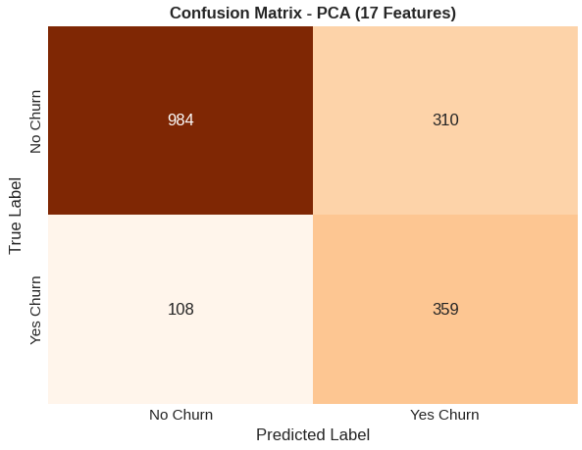

Fig 4.20: Original dataset Confusion Matrix of XGBoost

A closer look at the confusion matrix for the original dataset $\begin{bmatrix} TN = 950 & FP = 344 \\ FN = 85 & TP = 382 \end{bmatrix}$ which reflects the model's most aggressive churn-detection behavior shows:

- Minimal false negatives (FN = 85): Only 85 churn cases were missed. This low number represents XGBoost's strongest advantage: minimizing lost retention opportunities. For a telecom company, this directly translates to fewer preventable customer losses.
- Successful churn identification (TP = 382): Correctly identifying 382 at-risk customers represents a substantial pool of actionable retention opportunities and revenue protection.

Comparing the original and PCA-transformed datasets reveals an important behavioral shift.

True Positives decrease slightly from 382 to 373 reduced sensitivity and False Positives decrease from 344 to 327 improved precision. This shows PCA encourages a more cautious decision boundary.
The ROC curve plots for the XGBoost model are shown in Figure 4.21 and Figure 4.22 for the original dataset and the PCA-transformed dataset, respectively. These plots illustrate the discriminative power

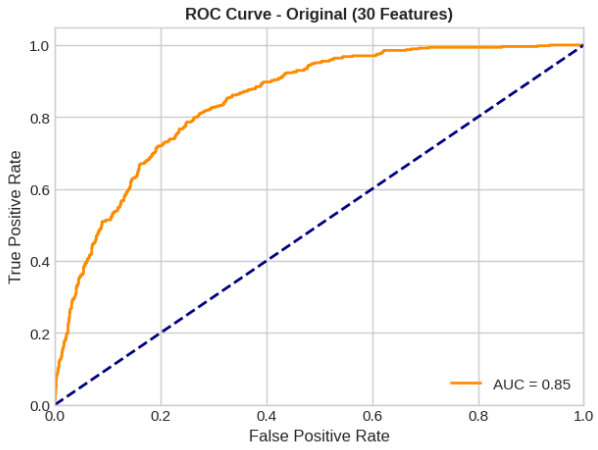

Fig 4.21: ROC curve of original dataset for XGBoost

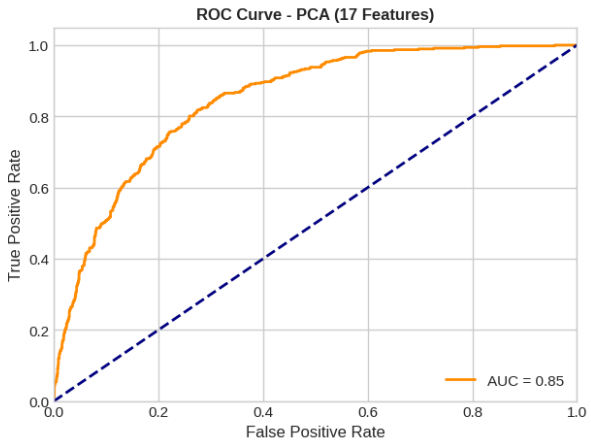

Fig 4.22: ROC curve of PCA dataset for XGBoost

of the model in other words, how effectively it separates churn customers from non-churn customers. Across all figure, the AUC remains fixed at 0.85. This means that in 85% of random comparisons between a true churn customer and a loyal customer, the model assigns a higher churn probability to the actual churn case. For the Telco churn dataset, this level of separability is considered strong and indicates reliable ranking capability for customer risk prioritization.
The Learning Curve of XGBoost is shown in Figure 4.23. The training curve decreases and moves closer to the validation curve as more data is added, a persistent gap of approximately 4% remains.

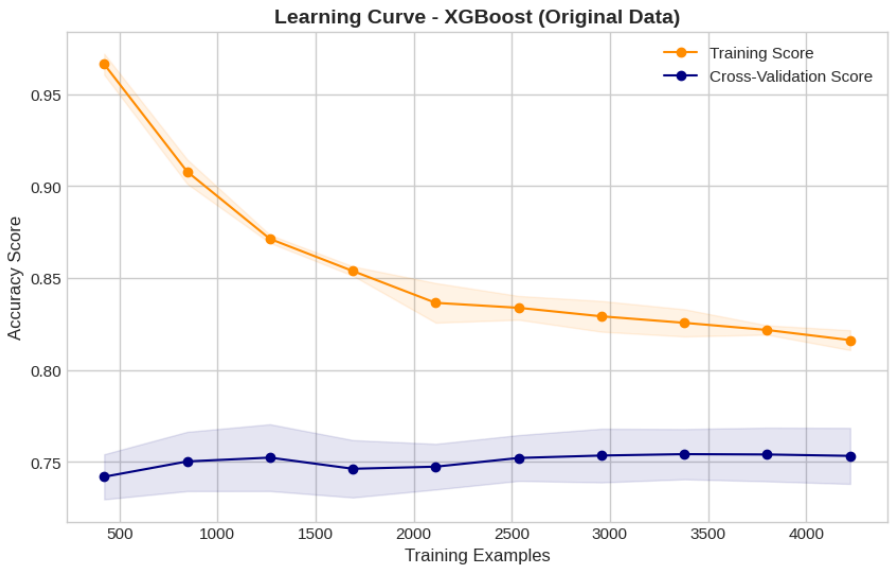

Fig 4.23: XGBoost Learning curve of original dataset

Initially, the training accuracy is very high (above 0.91) because memorizing a small dataset is relatively easy. As more data is introduced, the training score decreases gradually. This behavior is

expected and desirable it shows that the model is transitioning from memorization toward learning more generalizable patterns.
The validation performance steadily improves, rising from approximately 0.73 to 0.75. This upward trend confirms that the model genuinely benefits from additional data and improves its predictive ability on unseen samples.
Toward the right side of the graph, both curves flatten. After roughly 3,000 training samples, additional data yields minimal improvement in performance. Under the current model configuration and feature set, the model has reached its practical learning capacity. Simply adding more data of the same type is unlikely to produce a meaningful accuracy gain.
The light shaded region around the validation curve represents the standard deviation across repeated training runs. This shaded band remains narrow and consistent throughout the curve. The XGBoost model demonstrates strong stability. Its performance does not fluctuate significantly depending on how the data is split, indicating reliable and reproducible behavior an important characteristic for real-world deployment.

4.4.4 Light GBM

LightGBM, like XGBoost, is a gradient boosting based method. However, its internal architecture introduces several fundamental innovations that distinguish it in terms of speed, efficiency, and scalability.
The primary difference between LightGBM and many traditional boosting algorithms lies in how decision trees are expanded.

- Level-wise growth (XGBoost): The tree grows layer by layer to maintain structural balance.
- Leaf-wise growth (LightGBM): The algorithm identifies the leaf that yields the largest reduction in loss and expands only that branch.

This targeted growth strategy accelerates convergence and often produces lower training error because the model focuses computation where it matters most. However, on smaller datasets, aggressive leaf expansion can increase the risk of overfitting. This behavior is typically controlled through depth constraints such as the `max_depth` parameter.
Instead of processing continuous feature values directly, LightGBM discretizes them into a fixed number of bins. This histogram-based approach significantly reduces memory usage and computational overhead.
Because the algorithm no longer needs to repeatedly sort exact feature values, training becomes much faster while maintaining near-identical predictive performance. This efficiency makes LightGBM particularly suitable for large structured datasets.
GOSS is a selective sampling mechanism designed to preserve the most informative training examples. The intuition is that samples with large gradients carry more learning signal because they correspond to poorly predicted cases. LightGBM keeps all high-gradient samples while randomly sampling from low-gradient ones. This reduces the effective dataset size without sacrificing model accuracy, leading to faster training and improved scalability.
Many real-world datasets especially after one-hot encoding contain sparse features that rarely activate simultaneously. EFB combines mutually exclusive features into shared bundles, effectively reducing dimensionality.
This compression lowers memory requirements and accelerates computation without losing information, allowing LightGBM to handle high-dimensional feature spaces efficiently.

```
# LightGBM Training
lgb_clf = lgb.LGBMClassifier(
n_estimators=170,
learning_rate=0.05,
max_depth=4,
num_leaves=16,
scale_pos_weight=scale_pos_weight_val,
random_state=93,
n_jobs=-1,
verbose=-1
)

```

The implementation code for this algorithm follows the description above, and its hyperparameters are listed below.

1. Number of Trees: `n_estimators = 170`

   Because LightGBM is highly optimized for speed, it is often feasible to train a larger number of trees compared to models like XGBoost within a similar or even shorter time frame. Increasing this value can improve performance up to a point, but excessive boosting may introduce overfitting if not properly regularized.

2. Model Complexity Control: `max_depth=4` and `num_levels=16`

   These parameters limit how deep each decision tree is allowed to grow. specifies the maximum number of terminals leaves a tree can contain. For a binary tree, the theoretical maximum number of leaves at depth 4 is:

   $$2^4 = 16$$

   This interaction highlights an important tuning principle in LightGBM: tree depth and leaf count must be aligned to achieve the intended balance between model capacity and overfitting control.

After executing the LightGBM code, the evaluation metrics were measured, and their results are reported in the Table 6. The LightGBM model, when trained on the original dataset, achieved a recall

Dataset	Accuracy	Precision	Recall	F1-Score	Log Loss
Original	75.80%	52.82%	82.01%	64.26%	0.475
PCA	75.92%	53.04%	80.08%	63.82%	0.471

Table 6: Metric report of trained model by using LightGBM

rate of 82.01%. This indicates a strong ability to correctly identify customers who are likely to churn, which is a critical objective in churn prediction tasks.
The model's overall accuracy is approximately 75.8%, which is considered a solid and expected performance level for the Telco churn dataset. This suggests that the model maintains a good balance between identifying churn cases and correctly classifying loyal customers.
The Log Loss value of 0.475 further demonstrates that the model performs well in estimating class probabilities. A lower Log Loss reflects higher confidence and calibration in predictions, meaning the probabilities assigned by the model closely align with actual outcomes. This reliability is especially valuable in decision-making systems where risk ranking and prioritization are important.

Figure 4.24 show the original confusion matrix of LightGBM and Figure 4.25 shows the PCA dataset confusion matrix of LightGBM.

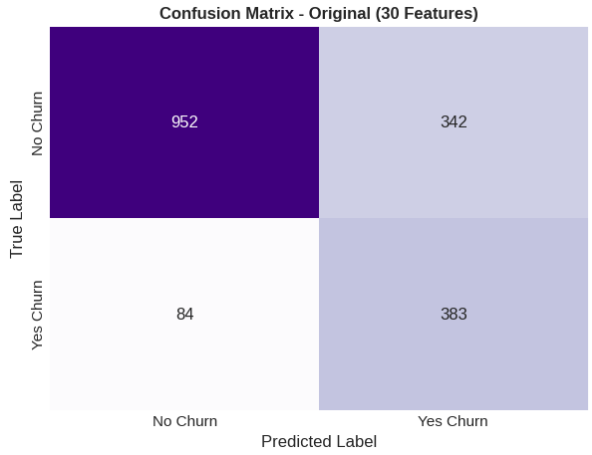

Fig 4.24: Original dataset Confusion Matrix of LightGBM

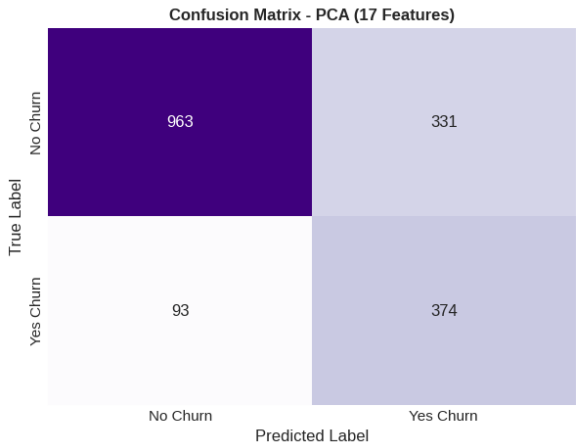

Fig 4.25: PCA dataset Confusion Matrix of LightGBM

An examination of the confusion matrix for the original dataset $\begin{bmatrix} TN = 952 & FP = 342 \\ FN = 84 & TP = 383 \end{bmatrix}$, which delivered the strongest performance, highlights several important findings:

- Lowest False Negative rate (FN = 84): This is the key strength of the model. Only 84 churn customers were missed by the classifier the lowest error count observed across all evaluated models.
- Maximum churn capture (TP = 383): The model correctly identified 383 churn customers, which is the highest detection count in this study. This demonstrates the model's strong sensitivity to churn-related patterns in the data.

The ROC curve plots for the LightGBM model are shown in Figure 4.26 and Figure 4.27 for the original dataset and the PCA-transformed dataset, respectively.

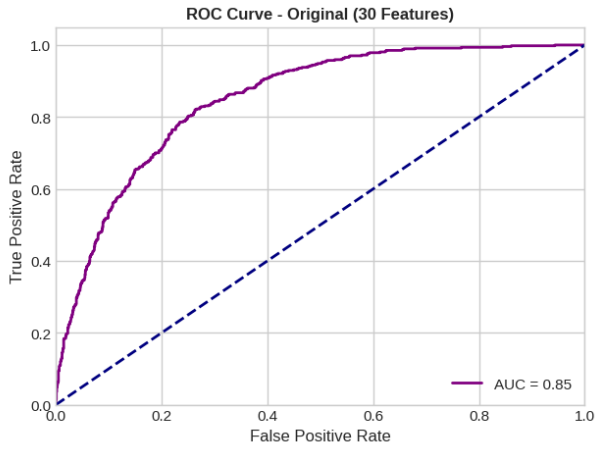

Fig 4.26: ROC curve of original dataset for LightGBM

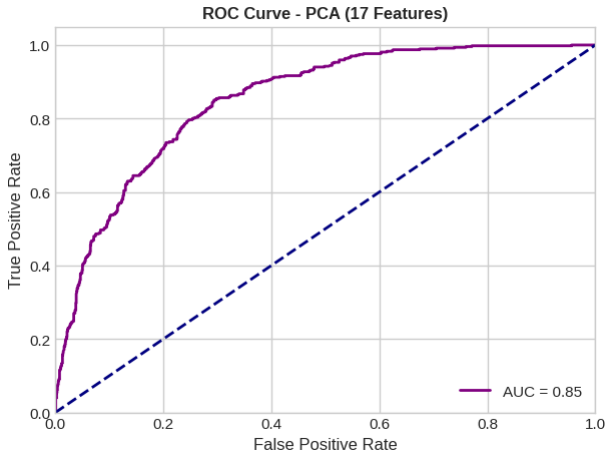

Fig 4.27: ROC curve of PCA dataset for LightGBM

Both ROC curves whether using the 30 original features or the 17 PCA-derived features achieve an AUC value of 0.85.

This result provides strong validation for the dimensionality reduction step. It shows that LightGBM is able to deliver the same discriminative performance with only 17 features as it does with the full 30-feature set. In practical terms, the features removed by PCA were largely redundant and did not contribute meaningful additional information to the model's predictive capability. This confirms that dimensionality reduction improved model efficiency without sacrificing performance.
An AUC of 0.85 means that if one churn customer and one non-churn customer are randomly selected, the model will assign a higher churn probability to the true churn case about 85% of the time. This reflects strong class separability and reliable ranking performance, which are essential for prioritizing customer retention efforts.
The Learning Curve of LightGBM is shown in Figure 4.28. The darker curve begins at a very high

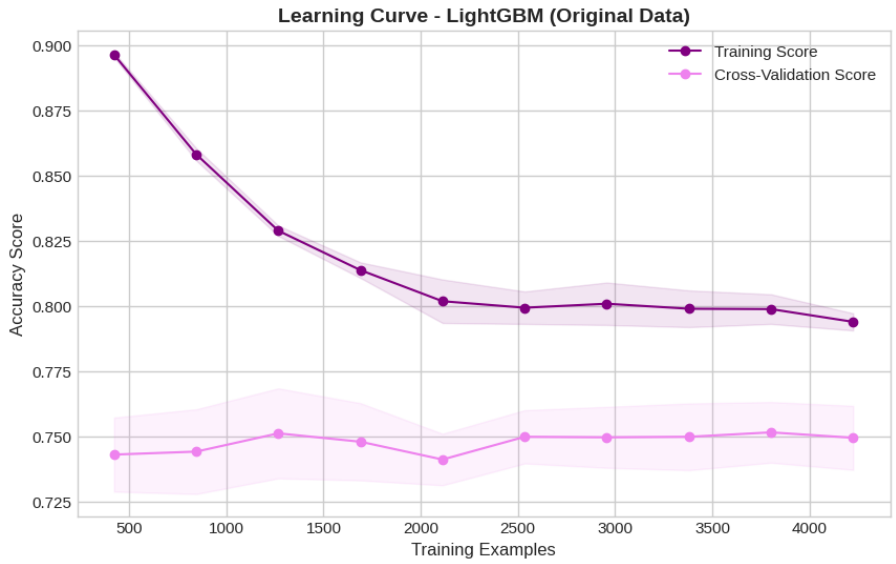

Fig 4.28: LightGBM Learning curve of original dataset

value (close to 0.90) and gradually decreases toward 0.80 as the training set grows. This behavior indicates that the model initially tends to memorize the limited training data, but as more samples are introduced, it is forced to shift from memorization toward learning more generalizable patterns.
The lighter curve starts at approximately 0.74 and stabilizes near 0.75. This plateau suggests that the model's performance on unseen data becomes consistent as the dataset size increases, reflecting steady generalization capability.
The persistent gap of roughly 5% between the training and validation curves indicates the presence of moderate overfitting. This is consistent with LightGBM's leaf-wise growth strategy, which aggressively optimizes splits and can capture fine-grained details in the training data, sometimes at the expense of generalization.
The narrow purple shaded band around the validation curve represents low standard deviation across folds. Its small width indicates that the LightGBM model is highly stable, producing consistent performance even when different subsets of data are used for validation.

4.4.5 Decision Tree

A decision tree consists of three main components:

- Root Node: The first and most important split, representing the feature that provides the highest information gain.

- Decision Nodes: Intermediate splits that progressively filter and separate the data.
- Leaf Nodes: Terminal nodes where the final prediction is made.

To determine which feature should be used at each split, the tree evaluates how "pure" the resulting subsets are. Two primary metrics are used:

1. Entropy and Information Gain: Entropy measures the level of disorder or impurity in a dataset. For a dataset S, it is defined as:

$$H(S) = -\sum_{i=1}^{c} p_i \log_2(p_i)$$

Where p_i represents the proportion of class i in the dataset.

If all samples belong to a single class, entropy equals 0.

If classes are evenly distributed, entropy reaches its maximum.

To choose the best split, the tree calculates Information Gain, which measures how much entropy is reduced after splitting on feature A:

$$IG(S, A) = H(S) - \sum_{v \in Values(A)} \frac{\mid S_v \mid}{\mid S \mid} H(S_v)$$

The split with the highest information gain is selected.

2. Gini Impurity: Gini impurity is the default criterion in Scikit-learn and is computationally more efficient than entropy:

$$Gini(S) = 1 - \sum_{i=1}^{c} (p_i)^2$$

This metric estimates the probability of misclassification if a label is randomly assigned according to the class distribution. The tree selects the split that minimizes Gini impurity in the resulting branches.

```
# Decision Tree Training
dt_clf = DecisionTreeClassifier(max_depth=7, min_samples_leaf=15,
class_weight='balanced', random_state=93)

```

The implementation code for this algorithm follows the description above, and its hyperparameters are listed below.

1. Tree Constraint: `max_depth = 7`

   This parameter defines the maximum number of layers the decision tree is allowed to grow. A depth of 7 means the model can ask at most seven sequential questions before reaching a final prediction.

2. Minimum Leaf Population: `min_samples_leaf = 15`

   This parameter specifies that a leaf node (final decision group) is only created if it contains at least 15 samples.

   By enforcing a minimum of 15 samples per leaf, the model is encouraged to make decisions based on statistically meaningful groups. This improves generalization and leads to predictions that are more reliable in real-world scenarios.

After executing the Decision Tree code, the evaluation metrics were measured, and their results are reported in the Table 7. The decision tree model achieved an impressive recall of 83.30% on the

Dataset	Accuracy	Precision	Recall	F1-Score	Log Loss
Original	71.94%	48.32%	83.29%	61.16%	0.828
PCA	73.53%	50.06%	78.58%	61.16%	0.675

Table 7: Metric report of trained model by using Decision Tree

original dataset. This performance even surpasses LightGBM (82.01%) and XGBoost (81.8%) in identifying churned customers. In practical terms, if the primary objective is simply to detect as many at-risk customers as possible, and the cost of false alarms is not a concern, the decision tree becomes the strongest tool among the compared models. However, this high recall comes at a significant cost: the model's precision drops to 48.3%. This indicates that the model has become overly sensitive. Roughly 1 out of every 2 customers predicted as churners is actually loyal. Such a high false positive rate can substantially increase the cost of marketing or retention campaigns.
The very high log loss value (0.8288) reveals that the model struggles to produce reliable probability estimates. Unlike boosting models which tend to output calibrated probabilities a decision tree often produces extreme probability values. When these confident predictions are wrong, they incur a heavy penalty in log loss.
Figure 4.29 show the original confusion matrix of Decision Tree and Figure 4.30 shows the PCA dataset confusion matrix of Decision Tree.

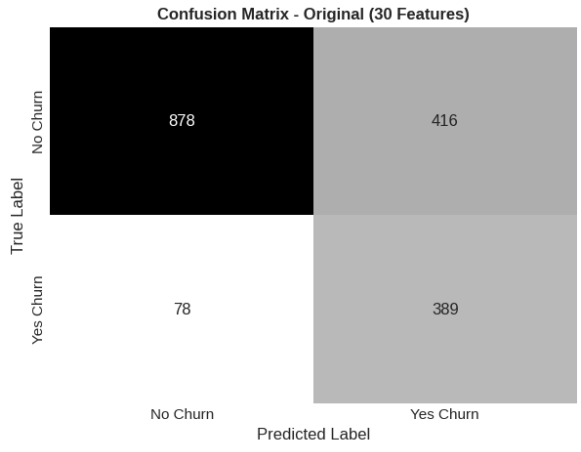

Fig 4.29: Original dataset Confusion Matrix of Decision Tree

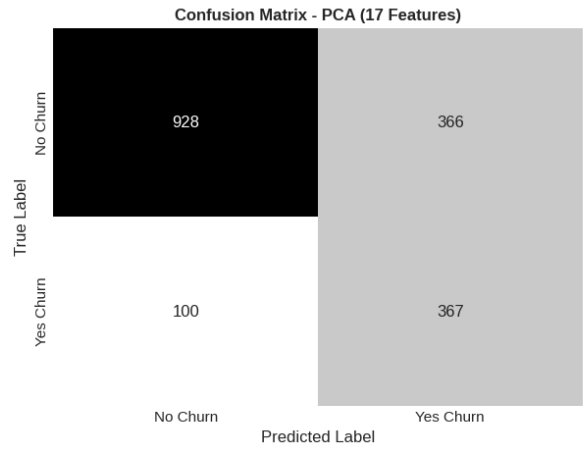

Fig 4.30: PCA dataset Confusion Matrix of Decision Tree

The ROC curve plots for the Decision Tree model are shown in Figure 4.31 and Figure 4.32 for the original dataset and the PCA-transformed dataset, respectively.

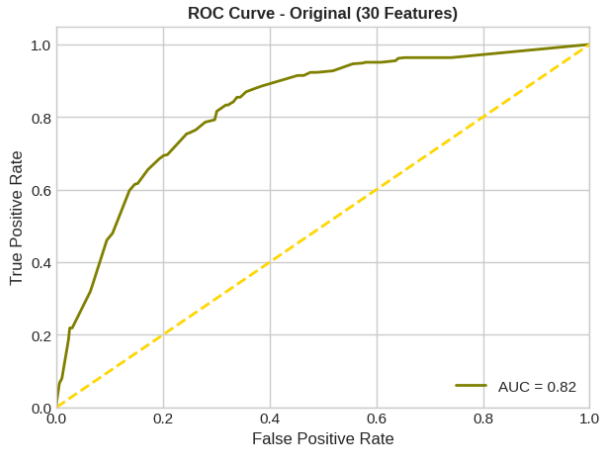

Fig 4.31: ROC curve of original dataset for Decision Tree

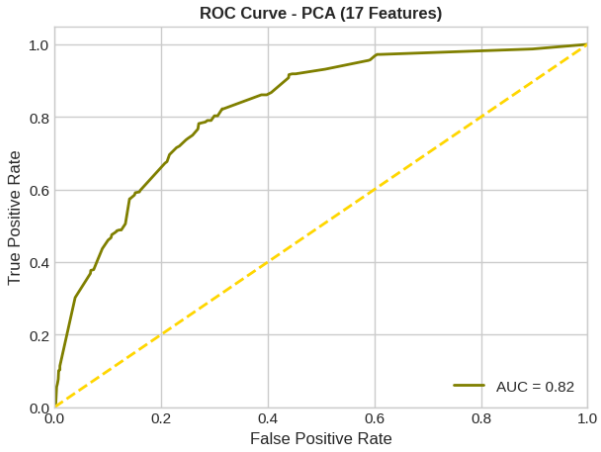

Fig 4.32: ROC curve of PCA dataset for Decision Tree

As observed earlier, the previous models (XGBoost, Random Forest, and LightGBM) all achieved an AUC of 0.85. The drop to 0.82 indicates that a single decision tree cannot capture complex class boundaries as effectively as ensemble models. Ensemble methods combine multiple learners, allowing them to model more nuanced decision surfaces and achieve stronger discrimination performance.
An interesting observation is that even for the decision tree, reducing the feature space from 30 to 17 dimensions using PCA did not degrade the AUC it remained at 0.82. This suggests that the tree model is able to extract all necessary predictive information from the compressed feature set, meaning the removed features were largely redundant rather than informative. The Learning Curve of Decision Tree is shown in figure 4.33.

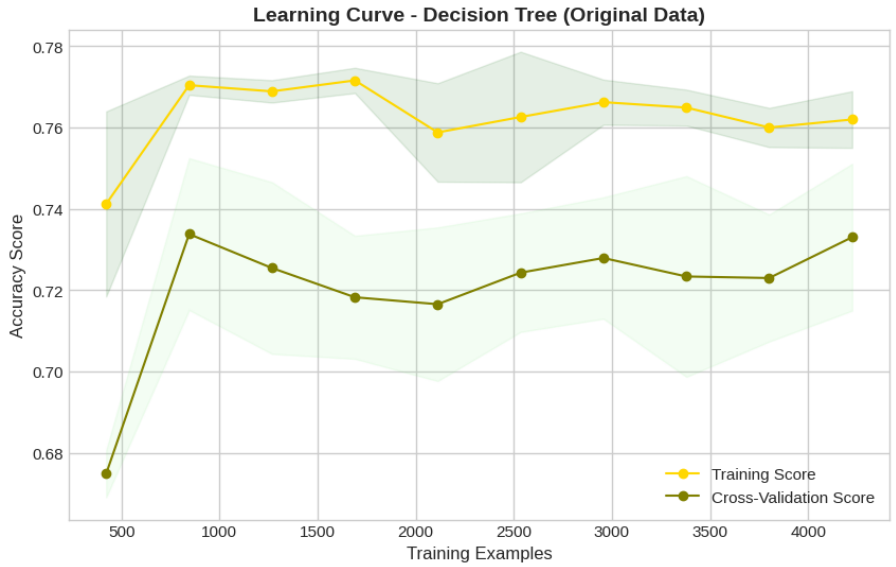

Fig 4.33: Decision Tree Learning curve of original dataset

Unlike XGBoost or LightGBM, which showed smooth learning curves, this plot exhibits noticeable fluctuations in accuracy especially in the early stages. This indicates that the model is highly sensitive to which samples happen to be included in the training split.

The separation between the training curve and the validation curve is clearly visible. Even with the depth limited to 7, the model performs significantly better on training data (around 77%) than on test data (about 73%). This suggests that the tree still tends to memorize noisy patterns in the training set rather than learning fully generalizable rules.
The shaded region around the validation curve is relatively wide, indicating higher variance across folds. In practical terms, this means the model's performance is less stable and more dependent on how the data is partitioned.

4.4.6 Final Conclusion

In this study, five machine learning algorithms were evaluated for customer churn prediction. Given the nature of the problem class imbalance and the high importance of detecting churn recall was chosen as the primary decision metric.

Model	Accuracy	Recall	Precision	F1-Score	Loss
LightGBM	75.81%	82.01%	52.83%	64.26%	0.4754
Decision Tree	71.95%	83.30%	48.32%	61.16%	0.828
XGBoost	75.64%	81.80%	52.62%	64.04%	0.473
Random Forest	74.84%	81.37%	51.63%	63.17%	0.498
SVM	75.47%	80.30%	52.45%	63.45%	0.525

Table 8: Final comparison of all classification models based on performance metrics

1. Decision Tree

   Most aggressive model and highest Recall at 83.3%, very low precision 48% and poor stability (high log loss) are the weakness of this model. This model hits many targets, but also produces many false alarms.

   A Decision Tree is a greedy learner it keeps splitting the feature space to fit the training data as closely as possible. This behavior often leads to very high recall, but low precision, since the model also learns noise and mistakenly generalizes it to new data.

   Decision Trees are highly sensitive to the training set. Even a small change in the data can produce a very different tree structure. This instability means the model lacks robustness and may not generalize consistently to unseen samples.

2. LightGBM

   Most intelligent and balanced model, it's Recall is very close to first place 82.01%, unlike the decision tree, it maintains high precision and strong probabilistic stability. This model achieves a golden balance between churn detection and prediction reliability.

   Most traditional boosting implementations grow trees level by level to keep them balanced. LightGBM instead grows trees leaf-wise. At each step, it expands only the leaf that yields the largest reduction in loss. This asymmetric growth lets the model focus its capacity on the hardest-to-model customer subgroups, capturing complex churn patterns more precisely.

   GOSS prioritizes samples with large gradients the cases the model is currently predicting poorly while down-sampling easier examples. These concentrates learning on informative errors, which can improve sensitivity to minority classes such as churn without a major speed penalty.

3. XGBoost

   Close competitor to LightGBM, Performance is very similar to LightGBM, but it ranks slightly lower overall. Unlike Random Forest, the second tree in XGBoost is trained specifically on the errors of the first tree. This process continues iteratively, forcing the model to focus on hard-to-predict customers and progressively reduce residual errors.

   XGBoost includes built in L1 and L2 regularization terms in its objective function. These penalties control model complexity, preventing trees from growing excessively and helping the model avoid overfitting a common issue in standalone decision trees.

4. Random Forest

   A strong and reliable model whose main limitation compared to boosting models is slower training and slightly less flexibility with noisy data. Random Forest addresses the main weakness of a single decision tree high variance by averaging the predictions of 100 trees. This stabilizes the model and reduces overfitting.

   Random Forest is inherently rule-based, which aligns well with the conditional and segmented nature of customer behavior data.

   The trees in Random Forest are built independently and do not learn from each other's mistakes. As a result, the model is less effective at correcting hard to classify cases compared to boosting methods, where each new tree explicitly focuses on prior errors.

5. SVM While acceptable, tree-based models demonstrated superior performance on this structured/tabular dataset compared to vector-based methods.

   SVM tries to find a geometric hyperplane that separates the data. Customer (tabular) data usually doesn't have a smooth, continuous geometric structure; instead, it is full of discrete conditional rules.

   SVM attempts to maximize the margin. The presence of outliers in customer behavior can shift the decision boundary, which may reduce accuracy.

   So, SVM performs very well on image or text data, but for tabular datasets that mix numerical and categorical variables, it is often outperformed by tree-based models.

Considering both technical performance and business impact, LightGBM is selected as the final model for this project.

Figure 4.34 also presents a graphical comparison of the models.

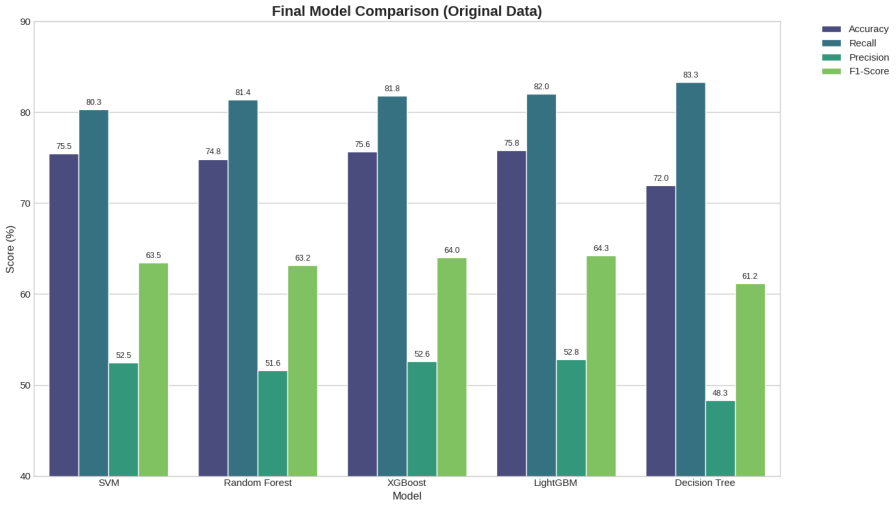

Fig 4.34: Models comparison bar chart

After comparing the models and selecting LightGBM as the best model, we present the 10 most important features in Figure 4.35.

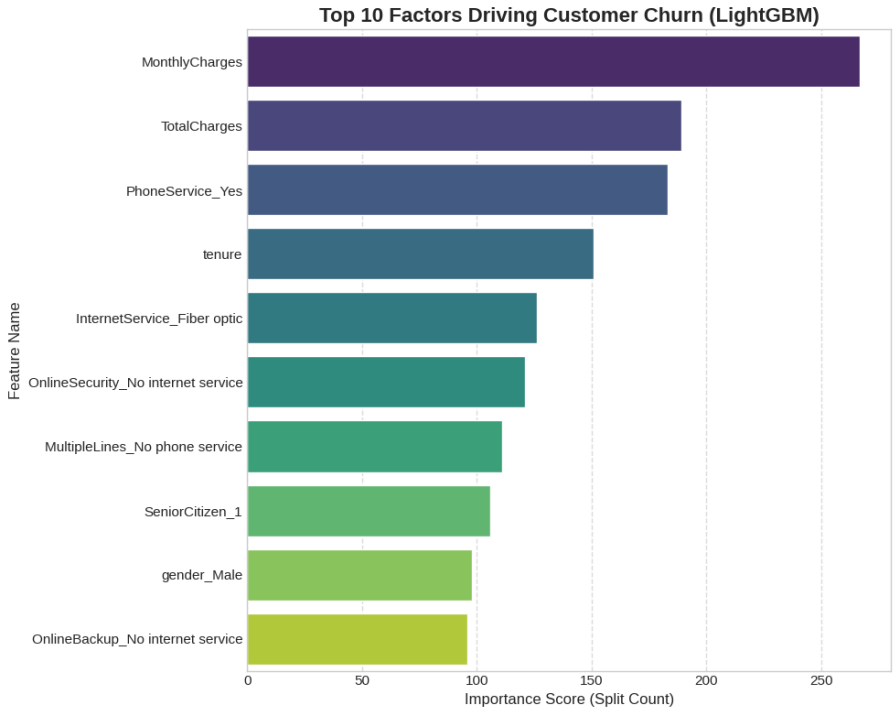

Fig 4.35: Top 10 most important features in Churn

The top two features, `MonthlyCharges` and `TotalCharge`, lead the ranking by a wide margin. This indicates that money is the primary factor in customer loyalty. Customers with high monthly charges are significantly more likely to churn. The company should focus on discounted plans or incentives for customers whose expenses exceed certain thresholds to reduce churn risk.
The feature `tenure` ranks fourth in importance. This indicates that time acts as a strong filter customer who pass the initial months of their contract tend to stabilize and remain loyal. The critical period for

churn is the early months of the contract, so interventions should focus on this stage.
The presence of `PhoneService_Yes` and `InternetService_Fiber` optic among the top features signals a red flag. Customers with Fiber optic internet tend to have a higher churn rate, this usually happens for two main reasons. Fiber optic plans are more expensive, making price-sensitive customers more likely to leave. In some areas, fiber quality may fluctuate, causing frustration and increasing the likelihood of churn.
Features like `OnlineSecurity_No` internet service and `OnlineBackup_No` internet service also appear in the top list. Customers who lack these add-on services are more likely to churn. These services act like a "sticky bond" when customers store their data or rely on your company for protection, it becomes harder for them to leave for a competitor.
The following Python scripts were used to generate the final comparative visualizations and feature importance analysis. The first script produces a grouped bar chart comparing the primary metrics across all five models, while the second script extracts and visualizes the top 10 features from the selected LightGBM model.

```
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

model_performance = {
  'Model': ['SVM', 'Random Forest', 'XGBoost', 'LightGBM', 'Decision Tree
   '],
  'Accuracy': [75.47, 74.84, 75.64, 75.81, 71.95],
  'Recall': [80.30, 81.37, 81.80, 82.01, 83.30],
  'Precision': [52.45, 51.63, 52.62, 52.83, 48.32],
  'F1-Score': [63.45, 63.17, 64.04, 64.26, 61.16]
}

df_results = pd.DataFrame(model_performance)

plt.figure(figsize=(14, 8))
plt.style.use('seaborn-v0_8-whitegrid')

df_melted = df_results.melt(id_vars="Model", var_name="Metric",
   value_name="Score")

ax = sns.barplot(x="Model", y="Score", hue="Metric", data=df_melted,
   palette="viridis")

plt.title('Final Model Comparison (Original Data)', fontsize=16,
   fontweight='bold')
plt.ylabel('Score (%)', fontsize=12)
plt.xlabel('Model', fontsize=12)
plt.ylim(40, 90)
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)

for container in ax.containers:
ax.bar_label(container, fmt='%.1f', padding=3, fontsize=9)

plt.tight_layout()
```

```
plt.show()

print("\n Ranking Models by Recall (Churn Detection Power):")
display(df_results.sort_values(by="Recall", ascending=False).reset_index(
    drop=True))

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

feature_imp = pd.DataFrame(sorted(zip(lgb_clf.feature_importances_,
    X_train.columns)), columns=['Value','Feature'])
feature_imp = feature_imp.sort_values(by="Value", ascending=False)

plt.figure(figsize=(10, 8))
sns.barplot(x="Value", y="Feature", data=feature_imp.head(10), palette="
    viridis")

plt.title('Top 10 Factors Driving Customer Churn (LightGBM)', fontsize
    =16, fontweight='bold')
plt.xlabel('Importance Score (Split Count)', fontsize=12)
plt.ylabel('Feature Name', fontsize=12)
plt.grid(axis='x', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

print("\nTop 10 Features Table:")
display(feature_imp.head(10).reset_index(drop=True))

```

4.5 Implementation using Hyperparameter Tuning

In Section 4.4, the models were trained without using hyperparameter search methods. In this section, we will implement that approach as well by applying the available tuning methods to optimize the hyperparameters.

4.5.1 Grid Search

This method is essentially a brute-force approach. You provide a list of values for each parameter, and the model tries all possible combinations.
If we supply values $[3, 5]$ for `max_depth` and $[100, 200]$ for `n_estimator` the method evaluates 4 cases: $(3, 100), (3, 200), (5, 100), (5, 200)$
If you have k hyperparameters and define n values for each, the total number of runs is:

$$\text{Total\_Runs} = n^k$$

If the optimal values lie within the grid, this method is guaranteed to find them.
This method is very slow. As the number of parameters grows, runtime increases exponentially (the curse of dimensionality).

4.5.2 Random Search

Instead of trying every single cell in the grid, this method samples randomly from the hyperparameter space.
We define ranges of values (for example, any number between 10 and 1000) and instruct the model to test only a fixed number of random combinations say, 100 trials.
Researches showed that many hyperparameters have little impact on performance. Random search avoids wasting time exhaustively exploring unimportant parameters and has a higher chance of finding good values for the parameters that actually matter.
It is much faster than grid search and often reaches similar or even better results with far fewer evaluations.

4.5.3 Implementation

Because the objective of this project is to maximize Recall while maintaining acceptable performance across the other evaluation metrics, the hyperparameter tuning process is conducted with recall as the optimization criterion.
First, we apply grid search to three models, LightGBM, SVM, and Decision Tree and applying Random Search just for LightGBM.
We should always be combined with cross-validation. In this approach, each parameter combination is evaluated across multiple data splits to ensure that the obtained performance is not due to chance and that the model generalizes reliably to unseen data.
Code of Grid Search for LightGBM is as follows.

```
import lightgbm as lgb
from sklearn.model_selection import GridSearchCV

neg_count = np.sum(y_train == 0)
pos_count = np.sum(y_train == 1)
scale_pos_weight_val = neg_count / pos_count
print(f"Base scale_pos_weight: {scale_pos_weight_val:.4f}\n")
param_grid = {
  'n_estimators': [50, 500],
  'learning_rate': [0.01, 1],
  'max_depth': [3, 7],
  'num_leaves': [10, 60],
  'scale_pos_weight': [scale_pos_weight_val, scale_pos_weight_val * 1.1]
}

```

After running the code, the optimal hyperparameters for the LightGBM model are presented in bellow:

- `n_estmator = 500`
- `learning_rate = 0.01`
- `max_depth = 3`
- `num_level = 10`

Code of Random Search for LightGBM is as follows.

```
import lightgbm as lgb
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint, uniform

neg_count = np.sum(y_train == 0)
pos_count = np.sum(y_train == 1)
scale_pos_weight_val = neg_count / pos_count
print(f"Base scale_pos_weight: {scale_pos_weight_val:.4f}\n")

param_dist = {
  'n_estimators': randint(100, 500),
  'learning_rate': [0.01, 0.05, 0.1, 0.2],
  'max_depth': [-1, 3, 5, 7, 10],
  'num_leaves': randint(20, 50),
  'min_child_samples': randint(10, 50),
  'scale_pos_weight': [scale_pos_weight_val, 3.0, 3.5, 4.0]
}

```

After running the code, the optimal hyperparameters for the LightGBM model are presented in bellow:

- `n_estmator = 216`
- `learning_rate = 0.01`
- `max_depth = 5`
- `num_level = 33`

For the remaining mentioned models, the same procedure is followed, and the results are reported in Table 9.

4.5.4 Final Conclusion for Hyperparameter Tuning

Model	Accuracy	Recall	Precision	F1-Score	Loss
LightGBM (Grid Search)	74.46%	84.36%	51.30%	63.80%	0.493
LightGBM (Random Search)	73.25%	86.08%	49.75%	63.05%	0.512
Decision Tree	71.89%	84.15%	48.28%	61.35%	0.560
SVM	74.90%	81.15%	51.70%	63.16%	0.425

Table 9: Metric report of all trained models

After performing hyperparameter tuning on the four final candidate models, the LightGBM model optimized via Grid Search is selected as the definitive winner.
Although this configuration achieves the highest recall (86.08%), its precision drops below 50% (49.75%). This means the number of false alarms (false positives) exceeds the number of correct positive predictions. In a real business setting, such behavior may reduce the marketing team's trust in the model.

This configuration achieves an excellent recall of 84.36%, while maintaining precision above 50%. This model provides the best overall balance, meaning it is both an effective “churn detector” and maintains an acceptable error rate.
The SVM model achieves a recall of 81.15%, roughly 3 percentage points lower. In a churn prediction context where losing customers carries significant cost this gap represents dozens of customers and is therefore highly meaningful. While SVM shows relatively low log loss, it lacks the ability of tree-based models to capture complex nonlinear patterns.
Although the decision tree achieves a competitive recall (84.15%), it suffers from the lowest precision (48.28%) and the weakest stability (loss = 0.560). This makes it a high-risk and unstable option.
The selected model achieves a log loss of 0.493 (below 0.5). This indicates that its predictions are not merely coincidental; rather, the model assigns churn probabilities with strong mathematical confidence and reliability.

4.6 Final Conclusion

After completing all the previous stages, the best-performing model was selected as LightGBM (Grid Search), we successfully developed a model capable of identifying 84.4% of customers who are likely to leave the company before the churn event occurs.