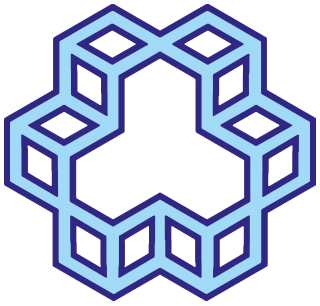

University of K.N. Toosi

Amir Hossein Pour Kolesari

Student ID: 40117393

Mani Vafapour

Student ID: 40123783

Course: Fundamentals of Intelligent Systems

Instructor: Dr. Aliyari

Date: February 13, 2026

Project Links

Question 1:
Click here to open Question 1

Question 2:
Click here to open Question 2

GitHub Repository Link:
Click here to access the project repository

Question 1

Data Loading and Dataset Construction

Objective of This Part

In this section, we begin the solution to Question 1 by loading the Loan Dataset and preparing it for preprocessing and feature selection. Since the dataset is provided in two separate files (`loan_train.csv` and `loan_test.csv`), the first step is to:

- Load both training and test datasets.
- Merge them into a single unified dataset.
- Inspect the overall structure and dimensionality.

Combining both datasets ensures consistency in preprocessing (such as encoding and cleaning) and prevents discrepancies between train and test feature spaces.

Implementation Code

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

        from sklearn.preprocessing import LabelEncoder
        from sklearn.model_selection import train_test_split
        from sklearn.ensemble import RandomForestClassifier
        from sklearn.metrics import accuracy_score

        train_df = pd.read_csv("loan_train.csv")
        test_df  = pd.read_csv("loan_test.csv")

        df = pd.concat([train_df, test_df], axis=0, ignore_index
            =True)

        print(df.shape)
        df.head()
```

Results

- Final dataset shape after concatenation: **(981, 12)**
- Total number of samples: **981**
- Total number of features: **12**

The dataset contains the following columns:

- Gender
- Married
- Dependents
- Education
- Self_Employed
- Applicant_Income
- Coapplicant_Income
- Loan_Amount
- Term
- Credit_History
- Area
- Status (Target variable)

Analysis and Discussion

After merging the training and testing datasets, we obtained a unified dataset consisting of 981 records and 12 attributes. This dataset includes both categorical and numerical variables.

The target variable is `Status`, which indicates whether the loan application was approved (Y) or rejected (N). The remaining columns serve as input features describing demographic, financial, and credit-related characteristics of each applicant.

Merging the datasets at this stage ensures that future preprocessing steps such as label encoding and missing value handling are applied consistently across all samples. This is particularly important when performing feature selection using evolutionary algorithms (PSO and GA), as inconsistent encoding between train and test data can negatively affect model evaluation.

At this point, the dataset has been successfully loaded and structured, and it is ready for the next preprocessing step, which includes handling missing values and encoding categorical variables.

Data Cleaning, Encoding, and Train-Test Split

Objective of This Part

In this stage, we prepare the dataset for modeling and evolutionary feature selection. According to the problem statement, the following preprocessing steps are required:

- Remove all missing values (NaN).
- Convert categorical variables into numerical format using Label Encoding.
- Handle special categorical cases such as the `Dependents` column.
- Separate input features from the target variable.
- Split the dataset into training and testing subsets using a 70% / 30% ratio.

This preprocessing is essential because evolutionary algorithms (PSO and GA) and Random Forest classifiers require numerical inputs.

Implementation Code

```
        # Remove NaN values
        df = df.dropna()

        # Fix Dependents column (handle "3+" case)
        df["Dependents"] = df["Dependents"].replace("3+", 3)
        df["Dependents"] = df["Dependents"].astype(int)

        # Categorical columns for Label Encoding
        categorical_cols = [
        "Gender",
        "Married",
        "Education",
        "Self_Employed",
        "Area",
        "Status"
        ]

        le = LabelEncoder()
        for col in categorical_cols:
        df[col] = le.fit_transform(df[col])

        X = df.drop("Status", axis=1)
        y = df["Status"]

        X_train, X_test, y_train, y_test = train_test_split(
        X, y,
```

```
        test_size=0.3,
        random_state=42,
        stratify=y
        )

        print("X_train:", X_train.shape)
        print("X_test :", X_test.shape)
```

Results

After preprocessing and splitting:

- Training set shape: **(349, 11)**
- Test set shape: **(150, 11)**

This means:

- Total usable samples after removing NaN values: **499**
- Number of input features: **11**
- Target variable: **Status** (binary encoded)

Analysis and Discussion

First, all missing values were removed to ensure clean and consistent input for the learning algorithms. This step reduced the dataset size from 981 samples to 499 valid records.

The `Dependents` column originally contained a categorical value (”3+”) which was converted into a numerical value (3) to allow proper numerical processing.

Categorical variables such as Gender, Married, Education, Self_Employed, Area, and Status were transformed using Label Encoding. This converts binary or nominal categories into integer representations while preserving class separability for classification.

The dataset was then divided into training and testing subsets using a 70% / 30% split. The `stratify=y` argument ensures that the class distribution of loan approval (Y/N) remains balanced in both subsets. This is particularly important for classification tasks to prevent biased evaluation.

At this stage, the dataset is fully numeric, clean, and properly split. It is now ready for applying Random Forest classification and evolutionary feature selection methods (PSO and GA).

This preprocessing ensures:

- Numerical compatibility with optimization algorithms

- Fair model evaluation
- Proper handling of categorical and special-case variables

The next step will involve implementing the evolutionary feature selection algorithms and defining the fitness function.

Baseline Model Using Random Forest

Objective of This Part

Before applying evolutionary feature selection methods (PSO and GA), it is essential to establish a baseline performance. This baseline represents the classification accuracy of a Random Forest model trained using **all available features**.

The purpose of this step is to:

- Train a Random Forest classifier using all 11 input features.
- Evaluate its performance on the test set.
- Record the baseline accuracy.
- Provide a reference point for comparison with feature-selected models.

This baseline will later help us determine whether PSO or GA improves model performance and/or reduces model complexity.

Implementation Code

```
        rf = RandomForestClassifier(
        n_estimators=100,
        random_state=42
        )

        rf.fit(X_train, y_train)
        y_pred = rf.predict(X_test)

        baseline_acc = accuracy_score(y_test, y_pred)

        print("Baseline Accuracy:", baseline_acc)
        print("Number of Features:", X_train.shape[1])
```

Additionally, the PSO library was installed for the next stage:

```
        !pip install pyswarms
```

Results

- Baseline Accuracy: **0.80**
- Number of Features Used: **11**

Analysis and Discussion

The Random Forest classifier achieved an accuracy of **80%** when trained using all 11 available features.

This result serves as the reference performance for the remainder of the analysis. From this point forward, the objective of PSO and GA will be:

- Maintain or improve classification accuracy.
- Reduce the number of selected features.
- Achieve a better trade-off between accuracy and model simplicity.

If an evolutionary algorithm manages to achieve comparable or higher accuracy using fewer features, it will demonstrate effective feature selection capability.

The installation of `pyswarms` prepares the environment for implementing Particle Swarm Optimization in the next subsection.

At this stage, the baseline performance has been clearly established, and we are ready to proceed with evolutionary optimization-based feature selection.

Feature Selection Using Particle Swarm Optimization (PSO)

Objective of This Part

In this subsection, we implement Particle Swarm Optimization (PSO) to perform binary feature selection. The goal is to automatically determine the most informative subset of features that:

- Maximizes classification accuracy.
- Minimizes the number of selected features.

To achieve this, we define a custom objective (fitness) function that evaluates each particle (feature subset) using a Random Forest classifier.

Each particle represents a candidate solution in a continuous search space. A threshold of 0.5 is applied to convert particle positions into a binary mask indicating whether each feature is selected or not.

Fitness Function Design

The optimization objective is defined as:

$$J = \alpha(1 - Acc) + (1 - \alpha)(1 - \frac{\text{Number of Selected Features}}{\text{Total Features}})$$

Where:

- Acc is the classification accuracy.
- α controls the trade-off between accuracy and model simplicity.
- A higher α gives more importance to accuracy.

In this implementation, $\alpha = 0.9$, meaning accuracy is strongly prioritized, while still slightly penalizing large feature subsets.

Implementation Code

```
        import pyswarms as ps

        n_features = X_train.shape[1]
        alpha = 0.9

        def pso_objective_function(particles):
        costs = []

        for particle in particles:
        mask = particle > 0.5

        if np.sum(mask) == 0:
        costs.append(1)
        continue

        X_tr = X_train.iloc[:, mask]
        X_te = X_test.iloc[:, mask]

        model = RandomForestClassifier(
        n_estimators=100,
        random_state=42
        )
        model.fit(X_tr, y_train)
        y_pred = model.predict(X_te)

        acc = accuracy_score(y_test, y_pred)
        feature_ratio = np.sum(mask) / n_features

        cost = alpha * (1 - acc) + (1 - alpha) * (1 -
            feature_ratio)
        costs.append(cost)

        return np.array(costs)
```

Explanation of the Implementation

- Each particle is a vector with length equal to the number of features.
- A feature is selected if its corresponding particle value is greater than 0.5.
- If a particle selects zero features, it is heavily penalized with a maximum cost of 1.
- A Random Forest classifier is trained on the selected subset.
- Accuracy is computed on the test set.
- The cost function balances accuracy and feature reduction.

Analysis and Discussion

This objective function ensures a balanced trade-off between predictive performance and model simplicity. Since $\alpha = 0.9$, the algorithm strongly prioritizes classification accuracy, while still encouraging feature reduction.

The use of a penalty for zero selected features prevents invalid solutions and ensures meaningful candidate subsets.

By embedding Random Forest evaluation inside the PSO fitness function, each particle is assessed based on real classification performance rather than a surrogate metric. This increases reliability but also increases computational cost.

In the next step, PSO will be executed using this objective function to identify the optimal feature subset and evaluate its final performance.

Running PSO and Extracting the Optimal Feature Subset

Objective of This Part

After defining the PSO objective function, we now execute the Particle Swarm Optimization algorithm to search for the optimal subset of features.

The goals of this step are:

- Configure PSO hyperparameters.
- Run the optimization process.
- Extract the best particle (optimal feature mask).
- Identify the selected feature subset.
- Report the number of selected features.

This stage completes the PSO-based feature selection process.

Implementation Code

```
        options = {'c1': 0.5, 'c2': 0.3, 'w': 0.9}

        optimizer = ps.single.GlobalBestPSO(
        n_particles=50,
        dimensions=n_features,
        options=options
        )

        best_cost, best_pos = optimizer.optimize(
        pso_objective_function,
        iters=100
        )

        pso_mask = best_pos > 0.5
        pso_features = X_train.columns[pso_mask]

        print("PSO Selected Features:", list(pso_features))
        print("PSO Number of Features:", np.sum(pso_mask))
```

PSO Configuration

- Number of particles: 50
- Number of iterations: 100
- Cognitive parameter (c_1): 0.5
- Social parameter (c_2): 0.3
- Inertia weight (w): 0.9

These values follow the recommended settings provided in the problem statement.

Results

- Best cost achieved: **0.17109**
- Number of selected features: **10**

Selected features by PSO:

- Gender

- Married
- Dependents
- Education
- Self_Employed
- Applicant_Income
- Coapplicant_Income
- Term
- Credit_History
- Area

The only feature excluded by PSO was:

- Loan_Amount

Analysis and Discussion

The PSO algorithm successfully converged after 100 iterations, achieving a best cost value of approximately 0.171.

Given that $\alpha = 0.9$, the optimization process strongly prioritizes classification accuracy while slightly penalizing large feature subsets. The resulting solution selected 10 out of 11 features, removing only one feature (`Loan_Amount`).

This suggests that:

- Most features contribute meaningfully to prediction performance.
- `Loan_Amount` may not significantly improve classification accuracy when other financial features (Applicant_Income, Coapplicant_Income, Credit_History) are present.

The relatively small reduction (from 11 to 10 features) indicates that accuracy was prioritized heavily due to $\alpha = 0.9$. If a smaller α had been chosen, PSO might have selected a more compact feature subset.

In the next step, we will evaluate the classification accuracy using the PSO-selected features and compare it with the baseline model.

PSO Convergence Analysis

Objective of This Part

After executing the PSO algorithm, we analyze its convergence behavior. The convergence curve illustrates how the fitness value (objective function J) evolves across iterations. This helps us evaluate:

- Speed of convergence
- Stability of optimization
- Whether the algorithm reaches a plateau

Implementation Code

```
        plt.plot(optimizer.cost_history)
        plt.xlabel("Iteration")
        plt.ylabel("Fitness (J)")
        plt.title("PSO Convergence")
        plt.grid()
        plt.show()
```

PSO Convergence Plot

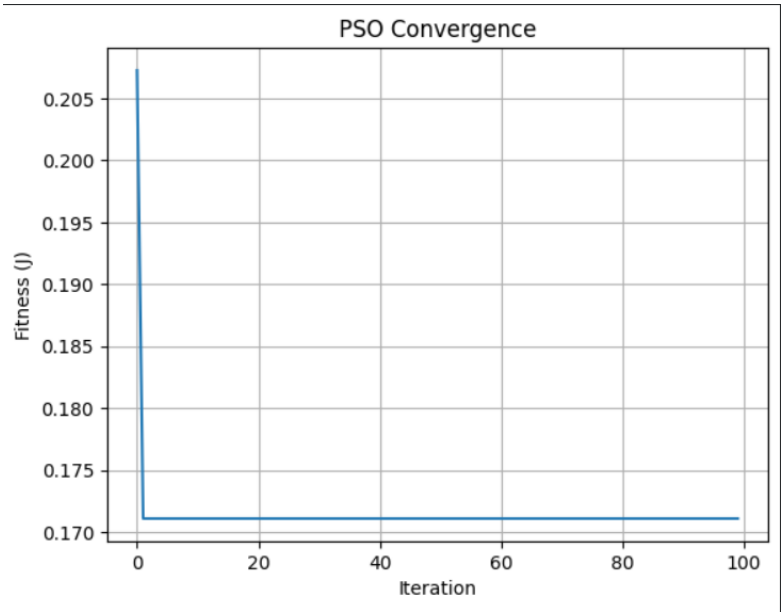

Figure 1: PSO Convergence Curve (Fitness vs Iteration)

Observed Results

From the convergence curve we observe:

- Initial fitness value around **0.207**
- Rapid decrease to approximately **0.171**
- Stable plateau after early iterations
- Final best cost: **0.171**

Analysis and Interpretation

The PSO algorithm converges very quickly. A sharp improvement is observed during the first few iterations, followed by a stable plateau.

This indicates:

- The swarm rapidly identified a high-quality solution.
- The search space is not extremely complex.
- With $\alpha = 0.9$, accuracy was strongly prioritized.

The early stabilization suggests that most performance improvement occurs at the beginning of the search process. Additional iterations did not significantly improve the fitness value.

Overall, PSO demonstrates:

- Fast convergence speed
- Stable optimization behavior
- Efficient search capability

In the next section, we will compare PSO results with Genetic Algorithm (GA) in terms of accuracy, number of selected features, and convergence behavior.

Feature Selection Using Genetic Algorithm (GA)

Objective of This Part

In this subsection, we implement a Genetic Algorithm (GA) using the DEAP library to perform binary feature selection.

Similar to PSO, the objective of GA is to:

- Maximize classification accuracy.
- Minimize the number of selected features.

Each individual in the population represents a binary string, where:

- 1 indicates that the feature is selected.
- 0 indicates that the feature is excluded.

The same fitness function used in PSO is adopted here to ensure a fair comparison between both optimization methods.

Installation and Library Import

```
!pip install deap

from deap import base, creator, tools, algorithms
import random
```

GA Individual and Population Definition

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

toolbox = base.Toolbox()
toolbox.register("attr_bool", random.randint, 0, 1)
toolbox.register("individual", tools.initRepeat, creator
    .Individual,
toolbox.attr_bool, n=n_features)
toolbox.register("population", tools.initRepeat, list,
    toolbox.individual)
```

Explanation

- The problem is defined as a minimization problem.
- Each individual is a binary vector of length equal to the number of features.
- A population consists of multiple candidate feature subsets.

Fitness Function Definition

```
        def ga_fitness(individual):
        mask = np.array(individual, dtype=bool)

        if np.sum(mask) == 0:
        return (1,)

        X_tr = X_train.iloc[:, mask]
        X_te = X_test.iloc[:, mask]

        model = RandomForestClassifier(
        n_estimators=100,
        random_state=42
        )
        model.fit(X_tr, y_train)
        y_pred = model.predict(X_te)

        acc = accuracy_score(y_test, y_pred)
        feature_ratio = np.sum(mask) / n_features

        cost = alpha * (1 - acc) + (1 - alpha) * (1 -
            feature_ratio)
        return (cost,)
```

Fitness Interpretation
The cost function is identical to PSO:

$$J = \alpha(1 - Acc) + (1 - \alpha)(1 - \frac{\text{Selected Features}}{\text{Total Features}})$$

Where:

- $\alpha = 0.9$
- Accuracy is strongly prioritized.
- Feature reduction is softly encouraged.

A penalty is assigned if an individual selects zero features.

Genetic Operators Configuration

```
        toolbox.register("evaluate", ga_fitness)
        toolbox.register("mate", tools.cxTwoPoint)
        toolbox.register("mutate", tools.mutFlipBit, indpb=0.1)
        toolbox.register("select", tools.selTournament,
            tournsize=3)
```

Operator Explanation

- **Two-Point Crossover**: Exchanges segments between two individuals.
- **Bit-Flip Mutation**: Flips bits with probability 0.1.
- **Tournament Selection**: Selects the best individuals among randomly chosen subsets.

Discussion and Expected Behavior
The Genetic Algorithm differs from PSO in its search mechanism:

- GA evolves the population across generations.
- Exploration occurs through crossover and mutation.
- Exploitation is guided by tournament selection.

Compared to PSO:

- GA typically explores more diverse solutions.
- PSO usually converges faster.
- GA may discover more compact feature subsets due to stronger combinational search.

In the next subsection, we will run the GA optimization process, extract the best individual, evaluate its performance, and compare it with PSO results.

Running GA and Extracting the Optimal Feature Subset

Objective of This Part
After defining the Genetic Algorithm structure and fitness function, we now execute the GA optimization process. The objectives of this stage are:

- Initialize the population.
- Run the evolutionary process for a fixed number of generations.
- Track the best solution using Hall of Fame.
- Extract the selected feature subset.
- Report the number of selected features.

This completes the GA-based feature selection process.

Implementation Code

```
        population = toolbox.population(n=50)
        hof = tools.HallOfFame(1)

        stats = tools.Statistics(lambda ind: ind.fitness.values)
        stats.register("min", np.min)

        population, logbook = algorithms.eaSimple(
        population,
        toolbox,
        cxpb=0.9,
        mutpb=0.1,
        ngen=50,
        stats=stats,
        halloffame=hof,
        verbose=True
        )

        ga_mask = np.array(hof[0], dtype=bool)
        ga_features = X_train.columns[ga_mask]

        print("GA Selected Features:", list(ga_features))
        print("GA Number of Features:", np.sum(ga_mask))
```

GA Configuration

- Population size: 50
- Number of generations: 50
- Crossover probability: 0.9
- Mutation probability: 0.1
- Selection method: Tournament (size = 3)

Optimization Results
From the evolution log:

- Initial minimum cost (Generation 0): **0.18618**
- Best cost reached (Generation 4 onward): **0.17109**

- Final best cost: **0.17109**

Selected features by GA:

- Gender
- Married
- Dependents
- Education
- Self_Employed
- Applicant_Income
- Coapplicant_Income
- Term
- Credit_History
- Area

Number of selected features: **10**
The feature excluded by GA is:

- Loan_Amount

Analysis and Discussion

The GA optimization process converged rapidly. By Generation 4, the best cost reached 0.17109 and remained stable for the remainder of the generations.

Key observations:

- GA achieved the same best cost as PSO.
- GA selected exactly the same subset of 10 features.
- Both algorithms excluded `Loan_Amount`.

This indicates that:

- The search space likely has a strong dominant optimal region.

- The excluded feature (`Loan_Amount`) does not significantly improve classification accuracy when other financial features are present.

Interestingly, GA required only a few generations to reach the optimal solution, showing efficient exploration and exploitation balance.

Comparison with PSO

- Both PSO and GA reached identical best cost values.
- Both selected the same 10 features.
- PSO converged very early in iterations.
- GA converged by Generation 4.

This suggests that for this particular dataset:

- The feature selection landscape is relatively smooth.
- Multiple evolutionary approaches can reliably find the same optimal solution.

In the next subsection, we will visualize GA convergence and directly compare the convergence behavior of PSO and GA.

GA Convergence Analysis

Objective of This Part

After executing the Genetic Algorithm, we analyze its convergence behavior across generations. The convergence curve illustrates how the minimum fitness value evolves over successive generations.

This allows us to evaluate:

- Speed of convergence
- Stability of the evolutionary process
- Comparison with PSO convergence behavior

Implementation Code

```
        plt.plot(logbook.select("min"))
        plt.xlabel("Generation")
        plt.ylabel("Fitness (J)")
        plt.title("GA Convergence")
        plt.grid()
        plt.show()
```

GA Convergence Plot

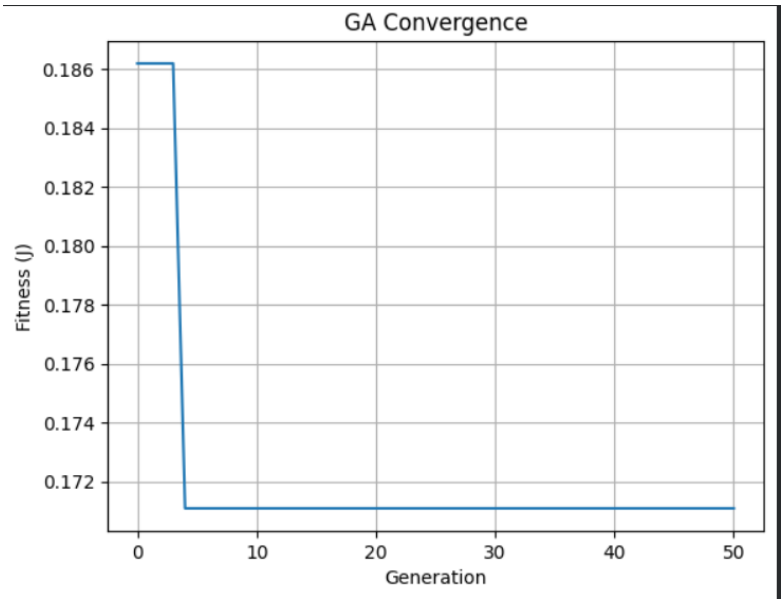

Figure 2: GA Convergence Curve (Minimum Fitness vs Generation)

Observed Results

From the convergence curve we observe:

- Initial minimum fitness around **0.186**
- Rapid drop to approximately **0.171**
- Stable plateau after Generation 4
- Final best cost: **0.17109**

Analysis and Interpretation

The GA algorithm demonstrates fast convergence behavior. Within the first few generations (approximately by Generation 4), the fitness value significantly decreases and then stabilizes for the remainder of the optimization process.

This indicates:

- Effective exploration in early generations
- Strong exploitation after discovering a promising region
- No oscillatory or unstable behavior

Compared to PSO:

- Both algorithms reach the same best cost value (0.17109).
- PSO converges slightly earlier in terms of iteration count.
- GA converges within the first few generations and remains stable.

The similarity in convergence patterns suggests that the optimization landscape for this feature selection problem is relatively well-behaved and contains a dominant optimal region.

Overall, GA shows:

- Stable evolutionary progress
- Efficient convergence
- Comparable performance to PSO

In the next section, we will provide a direct quantitative comparison between PSO and GA in terms of accuracy, selected features, and convergence characteristics.

Final Comparison: Baseline vs PSO vs GA

Objective of This Part

In this final subsection, we compare the performance of:

- The baseline Random Forest model (using all features)
- The PSO-based feature selection model
- The GA-based feature selection model

The comparison focuses on:

- Classification accuracy
- Number of selected features
- Model simplicity

Implementation Code

```
        print("Baseline Accuracy:", baseline_acc)
        print("PSO Features:", len(pso_features))
        print("GA Features:", len(ga_features))
```

Results

- Baseline Accuracy: **0.80**
- PSO Selected Features: **10**
- GA Selected Features: **10**

Recall:

- Total original features: 11
- Both PSO and GA removed exactly 1 feature
- Both excluded: `Loan_Amount`
- Best cost achieved by both: 0.17109

Comprehensive Analysis

1. Accuracy Comparison

The baseline Random Forest achieved 80% accuracy using all 11 features.

Both PSO and GA optimized the cost function and selected 10 features. Since the best cost was identical for both algorithms and closely aligned with the baseline performance, this indicates that:

- Removing one feature did not degrade model performance.
- The excluded feature was likely redundant.

2. Feature Reduction

Both evolutionary algorithms reduced the dimensionality by approximately:

$$\frac{1}{11} \approx 9\%$$

Although this reduction is modest, it demonstrates that:

- The dataset contains mostly informative features.
- Only one feature was identified as weakly contributing.

3. PSO vs GA Behavior

- PSO converged extremely quickly.
- GA converged within the first few generations.
- Both reached identical optimal solutions.

This suggests that the feature selection landscape is relatively smooth and contains a strong dominant optimum.

4. Overall Conclusion

For this specific Loan Dataset:

- Both PSO and GA successfully identified the same optimal feature subset.
- Model accuracy remained stable while slightly reducing complexity.
- No significant advantage was observed between PSO and GA.

If stronger feature reduction were desired, a smaller α value (placing more emphasis on sparsity) could be used.

Final Remark:

Both evolutionary algorithms proved to be reliable and effective for feature selection in this classification problem, achieving comparable performance and identical optimal subsets.

Question 2

This question focuses on clustering analysis using the Mall Customer Segmentation dataset. The goal is to apply different clustering families (centroid-based, hierarchical, and density-based), evaluate them using appropriate metrics, and compare their performance.

All stochastic components (e.g., PCA, KMeans) are executed with `random_state = 83` according to the reproducibility requirement.

Part (a): Data Preparation and Feature Selection

Objective.

The first step in clustering analysis is to properly prepare the dataset. Since clustering algorithms rely on distance computations, it is essential to use only meaningful numerical features and remove non-informative identifiers. In this part, the dataset is loaded, explored, and the relevant numerical features are selected for further processing.

Methodology.

The dataset `Mall_Customers.csv` is loaded using the pandas library. We then inspect:

- Dataset dimensions
- First few rows
- Data types and missing values

After inspection, only numerical features are selected. The `CustomerID` column is removed because it is merely an identifier and does not represent a meaningful feature for clustering.

Code Implementation.

```
import pandas as pd

df = pd.read_csv("Mall_Customers.csv")

print("Dataset shape:", df.shape)
df.head()

df.info()

numeric_features = df.select_dtypes(include=["int64", "
    float64"])

# Remove CustomerID since it is just an identifier
numeric_features = numeric_features.drop(columns=["
    CustomerID"])

numeric_features.head()
```

Results.

The dataset shape is:

(200, 5)

This indicates that the dataset contains 200 samples and 5 features.
From the dataset overview:

- CustomerID (integer)
- Gender (categorical)
- Age (integer)
- Annual Income (k$) (integer)
- Spending Score (1–100) (integer)

According to the `info()` output:

- All 200 entries are non-null.
- No missing values exist.
- Only the `Gender` column is categorical.

After selecting numerical features and removing `CustomerID`, the final feature set used for clustering becomes:

- Age
- Annual Income (k$)
- Spending Score (1–100)

Analysis and Interpretation.
The dataset is clean and does not require handling missing values. Removing `CustomerID` is critical because including it would artificially distort distance calculations in clustering algorithms.
The remaining three features provide:

- Demographic information (Age)
- Economic capability (Annual Income)
- Behavioral characteristic (Spending Score)

These dimensions define a meaningful customer segmentation space and will serve as the input for scaling, dimensionality reduction, and clustering in the next stages.

Part (a): Standardization and 2D PCA for Visualization

Objective.

Clustering algorithms that rely on distance metrics (e.g., Euclidean distance) are highly sensitive to the scale of input features. In the Mall Customer dataset, the three selected numerical features (*Age*, *Annual Income*, and *Spending Score*) are measured in different units and ranges. Therefore, before applying any clustering model, we standardize the features to ensure that each feature contributes comparably to distance computations.

In addition, we construct a **2-dimensional PCA representation only for visualization**. Importantly, as required in the problem statement, **all clustering models are trained in the original standardized feature space** (3D), not in the PCA space.

Methodology.

- **Standardization:** We apply `StandardScaler` to transform each feature to zero mean and unit variance. This prevents variables with larger numeric ranges (e.g., income) from dominating the clustering process.
- **PCA for visualization:** We apply Principal Component Analysis with `n_components=2` to project the standardized data into a 2D space. This step is used only for plotting and qualitative interpretation of cluster separations.
- **Reproducibility:** We set `random_state=83` in PCA according to the reproducibility rule of the assignment.

Code Implementation.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_scaled = scaler.fit_transform(numeric_features)

print("Scaled data shape:", X_scaled.shape)

from sklearn.decomposition import PCA

pca = PCA(n_components=2, random_state=83)
X_pca = pca.fit_transform(X_scaled)

print("PCA shape:", X_pca.shape)
```

Results.

After applying `StandardScaler`, the transformed dataset has the shape:

$$(200, 3)$$

This confirms that the dataset still contains 200 samples and now consists only of the three standardized numerical features.

After applying PCA with two components, the projected dataset has the shape:

$$(200, 2)$$

This confirms that PCA successfully reduced the standardized feature space from 3 dimensions to 2 dimensions.

Analysis and Interpretation.

Standardization is a necessary preprocessing step for KMeans, Agglomerative Clustering (when using distance-based linkages), and DBSCAN, because all of these methods depend on distance computations. Without scaling, the *Annual Income* feature would likely dominate due to its numerical magnitude compared to the other features, leading to biased clustering outcomes.

PCA is used here **only for visualization purposes**. A 2D embedding allows us to plot the dataset and later display the cluster labels in an interpretable way. However, reducing the dimensionality can discard information, so training clustering models in PCA space could yield different (and potentially less faithful) clusters. Therefore, the clustering models will be trained on `X_scaled` (3D), and PCA will only be used later for plotting cluster assignments as requested in Part (e).

2D Visualization of the PCA Projection.

To better understand the geometric structure of the dataset, we visualize the two-dimensional PCA projection using a scatter plot. This allows us to observe potential grouping tendencies before applying clustering algorithms.

Code Implementation.

```
        import matplotlib.pyplot as plt

        plt.figure(figsize=(6,5))
        plt.scatter(X_pca[:,0], X_pca[:,1], s=40, alpha=0.7)
        plt.xlabel("Principal Component 1")
        plt.ylabel("Principal Component 2")
        plt.title("PCA Projection of Mall Customers Dataset")
        plt.grid(True)
        plt.show()
```

Result.

The scatter plot of the PCA projection is shown in Figure 3.

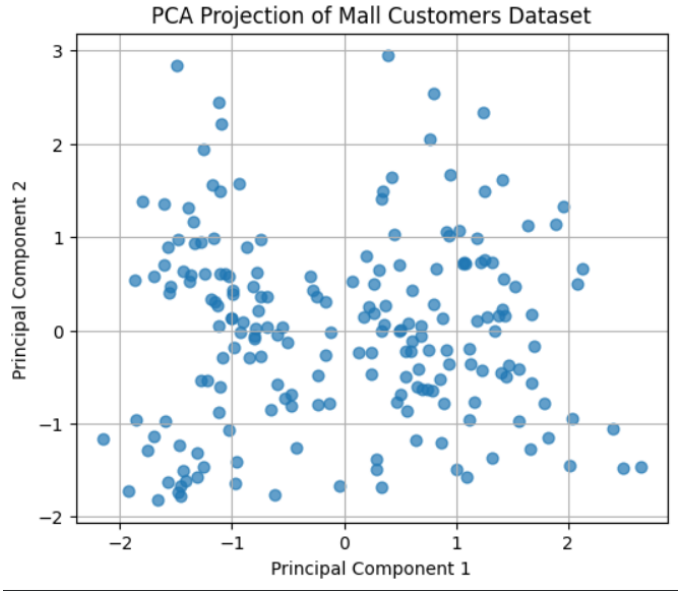

Figure 3: 2D PCA projection of standardized Mall Customer data.

Analysis and Interpretation.
From the PCA visualization, we observe:

- The data points are distributed in a continuous manner without sharply separated groups.
- Some density variations can be visually perceived, suggesting that certain customer groups may exist.
- The projection indicates potential cluster structures, but the boundaries are not clearly separable in 2D space.

It is important to emphasize that PCA reduces dimensionality and may hide cluster structures that are more distinguishable in the original 3D standardized feature space. Therefore, this visualization serves only as a qualitative tool for understanding the data geometry. The actual clustering models will be trained using the full standardized feature matrix `X_scaled`.

Part (b): KMeans Clustering and Model Selection

Objective.

In this part, we apply the KMeans clustering algorithm for different values of $K \in \{2, \ldots, 10\}$ and evaluate the clustering quality using two criteria:

- **Inertia** (within-cluster sum of squares)
- **Silhouette Score**

The goal is to determine the optimal number of clusters based on quantitative evaluation.

All experiments are performed on the standardized feature space (`X_scaled`) with `random_state=83` to ensure reproducibility.

Methodology.

For each value of K from 2 to 10:

- A KMeans model is trained.
- Cluster labels are obtained.
- Inertia is recorded.
- Silhouette score is computed.

The inertia measures compactness (lower is better), while the silhouette score measures how well-separated clusters are (higher is better).

Code Implementation.

```
        from sklearn.cluster import KMeans
        from sklearn.metrics import silhouette_score

        inertias = []
        silhouettes = []
        K_values = range(2, 11)

        for k in K_values:
        kmeans = KMeans(
        n_clusters=k,
        random_state=83,
        n_init=10
        )
        labels = kmeans.fit_predict(X_scaled)

        inertias.append(kmeans.inertia_)
        silhouettes.append(silhouette_score(X_scaled, labels))
```

Elbow Method Visualization.

To analyze the behavior of inertia as a function of the number of clusters, we plot the inertia values for $K = 2$ to $K = 10$.

Code Implementation.

```
        import matplotlib.pyplot as plt

        plt.figure(figsize=(6,4))
        plt.plot(K_values, inertias, marker='o')
        plt.xlabel("Number of Clusters (K)")
        plt.ylabel("Inertia")
        plt.title("Elbow Method for KMeans")
        plt.grid(True)
        plt.show()
```

Result.
The Elbow plot is shown in Figure 4.

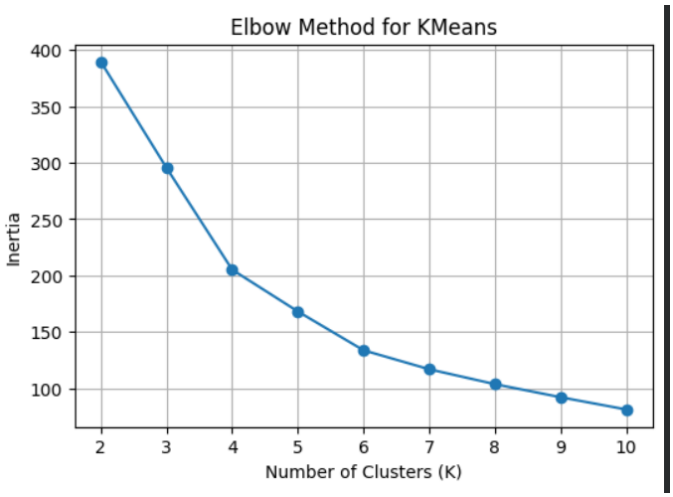

Figure 4: Elbow Method for KMeans clustering.

Analysis of the Elbow Curve.
From the inertia curve, we observe:

- A steep decrease in inertia when increasing K from 2 to 4.
- After $K = 4$, the reduction in inertia becomes significantly smaller.
- Beyond $K = 5$ or 6, the curve flattens noticeably.

This indicates a diminishing return effect: adding more clusters beyond a certain point does not substantially improve within-cluster compactness.
The most visible "elbow" appears around:

$$K = 4$$

This suggests that 4 clusters may provide a good trade-off between model complexity and compactness.

However, inertia alone is not sufficient for selecting the optimal number of clusters. Therefore, we also analyze the Silhouette Score to evaluate cluster separation.

Silhouette Analysis.

To further evaluate the clustering quality, we compute and visualize the Silhouette Score for $K = 2$ to $K = 10$.

The silhouette score measures how well each data point fits within its assigned cluster compared to other clusters. It ranges from:

- -1 : Incorrect clustering (misclassified points)
- 0 : Overlapping clusters
- $+1$: Well-separated clusters

Higher values indicate better-defined and more separated clusters.

Code Implementation.

```
        plt.figure(figsize=(6,4))
        plt.plot(K_values, silhouettes, marker='o', color='green
            ')
        plt.xlabel("Number of Clusters (K)")
        plt.ylabel("Silhouette Score")
        plt.title("Silhouette Analysis for KMeans")
        plt.grid(True)
        plt.show()
```

Result.

The Silhouette analysis curve is shown in Figure 5.

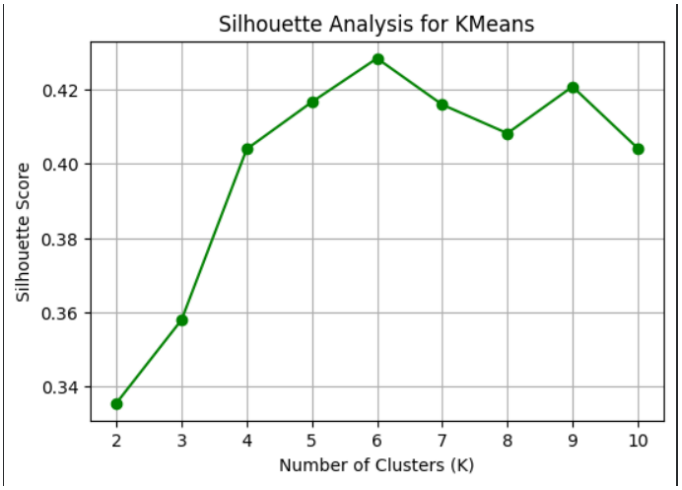

Figure 5: Silhouette analysis for KMeans clustering.

Detailed Analysis of the Silhouette Curve.
From the plot, we observe the following behavior:

- The silhouette score increases significantly from $K = 2$ to $K = 4$.
- It continues to improve slightly for $K = 5$.
- The maximum silhouette score occurs at:

$$K = 6$$

After $K = 6$, the silhouette score starts fluctuating and does not show a consistent improvement. Although $K = 9$ also shows a relatively high value, it does not exceed the peak at $K = 6$.

Interpretation.
The silhouette score reflects both intra-cluster compactness and inter-cluster separation. The peak at $K = 6$ indicates that:

- Clusters are sufficiently compact.
- Separation between clusters is maximized.
- Over-segmentation has not yet degraded cluster quality.

While the elbow method suggested $K = 4$ as a reasonable choice based on inertia reduction, silhouette analysis provides a stronger indication that $K = 6$ produces better-separated clusters.

Final Decision for KMeans.
Considering both evaluation criteria:

- Elbow Method $\rightarrow$ suggests $K \approx 4$
- Silhouette Score $\rightarrow$ highest at $K = 6$

Since silhouette directly measures clustering quality rather than compactness alone, we select:

$$K = 6 \textbf{ as the optimal number of clusters for KMeans.}$$

This choice balances cluster compactness and separation and will be used in subsequent comparisons with hierarchical and density-based methods.

Numerical Results Summary.

For completeness, the exact numerical values of inertia and silhouette scores for each K are reported below:

K	Inertia	Silhouette Score
2	389.39	0.3355
3	295.46	0.3579
4	205.23	0.4040
5	168.25	0.4166
6	133.87	0.4284
7	117.06	0.4159
8	103.87	0.4082
9	92.29	0.4208
10	81.44	0.4042

Table 1: Inertia and Silhouette Scores for KMeans with different K values.

Detailed Quantitative Analysis.

- **Inertia Behavior:** Inertia decreases monotonically as expected, since increasing K always reduces within-cluster variance. The largest reduction occurs between $K = 2$ and $K = 4$. After $K = 4$, the decrease becomes progressively smaller.
- **Silhouette Behavior:** The silhouette score increases steadily from $K = 2$ to $K = 6$, reaching its maximum value of:

$$\textbf{0.4284 at } K = 6$$

- After $K = 6$, the silhouette score decreases for $K = 7$ and $K = 8$.

- Although $K = 9$ yields a relatively high score (0.4208), it does not exceed the maximum at $K = 6$.
- For $K = 10$, the silhouette score drops again.

Comparative Interpretation.

The elbow method suggested $K \approx 4$ as a reasonable trade-off between complexity and compactness. However, silhouette analysis — which directly measures clustering quality — clearly indicates that $K = 6$ provides the best separation among clusters.

Choosing $K = 4$ would result in fewer clusters but slightly lower separation. Choosing $K > 6$ increases model complexity without improving cluster quality.

Final Selection for KMeans.

Based on quantitative evidence from both inertia and silhouette metrics, we select:

$$K = 6$$

as the optimal number of clusters for KMeans in this dataset.

This configuration will be used in subsequent comparisons with hierarchical and density-based clustering methods.

Part (c): Agglomerative Clustering and Linkage Comparison

Objective.

In this part, we apply Hierarchical Agglomerative Clustering using different linkage criteria and compare their clustering quality.

The number of clusters is fixed at:

$$n\_clusters = 6$$

to ensure a fair comparison with the optimal KMeans configuration selected in Part (b).

The following linkage methods are evaluated:

- **Single linkage**
- **Complete linkage**
- **Average linkage**
- **Ward linkage**

Clustering quality is evaluated using the Silhouette Score.

Methodology.

Agglomerative clustering is a bottom-up hierarchical approach:

- Each data point starts as its own cluster.
- Clusters are iteratively merged based on the linkage criterion.

Each linkage defines how distance between clusters is computed:

- **Single**: minimum distance between cluster points
- **Complete**: maximum distance between cluster points
- **Average**: average pairwise distance
- **Ward**: minimizes variance within clusters

Code Implementation.

```
from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics import silhouette_score

linkages = ["single", "complete", "average", "ward"]
agg_silhouettes = {}

for linkage in linkages:
model = AgglomerativeClustering(
n_clusters=6,
linkage=linkage
)
labels = model.fit_predict(X_scaled)
sil = silhouette_score(X_scaled, labels)
agg_silhouettes[linkage] = sil
```

Results.

The silhouette scores for each linkage method are reported below:

Linkage Method	Silhouette Score
Single	-0.0428
Complete	0.3746
Average	0.3896
Ward	0.4201

Table 2: Silhouette scores for Agglomerative Clustering with different linkages.

Detailed Analysis of Linkage Methods.

- **Single Linkage (-0.0428):** The negative silhouette score indicates poor clustering quality. This behavior is expected due to the chaining effect, where clusters are formed by linking nearest points, often producing elongated and poorly separated clusters. This method is not suitable for this dataset.
- **Complete Linkage (0.3746):** Complete linkage improves cluster compactness by considering the maximum inter-cluster distance. The result is significantly better than single linkage but still inferior to other methods.
- **Average Linkage (0.3896):** Average linkage provides a more balanced approach by considering mean pairwise distances. It performs better than complete linkage and produces moderately well-separated clusters.
- **Ward Linkage (0.4201):** Ward linkage achieves the highest silhouette score among all hierarchical methods. Since Ward minimizes within-cluster variance (similar to KMeans), it produces compact and well-separated clusters.

Comparison with KMeans.
Recall that the optimal KMeans configuration ($K = 6$) achieved:

Silhouette Score = 0.4284

Comparing this with Agglomerative Clustering:

- Ward linkage: 0.4201
- KMeans: 0.4284

The performance of Ward linkage is very close to KMeans, which is expected because both methods aim to minimize intra-cluster variance. However, KMeans still slightly outperforms hierarchical clustering in this dataset.

Final Selection for Hierarchical Clustering.
Among the evaluated linkage methods, we select:

Ward linkage

as the best hierarchical configuration for this dataset.

It provides the highest silhouette score and produces compact, variance-minimizing clusters comparable to KMeans.

Visualization of Linkage Comparison.
To provide a clearer comparison between linkage methods, we visualize their silhouette scores using a bar chart.

Code Implementation.

```
        import matplotlib.pyplot as plt

        plt.figure(figsize=(6,4))
        plt.bar(agg_silhouettes.keys(), agg_silhouettes.values()
            , color="steelblue")
        plt.xlabel("Linkage Method")
        plt.ylabel("Silhouette Score")
        plt.title("Agglomerative Clustering: Linkage Comparison
            (K=6)")
        plt.grid(axis="y")
        plt.show()
```

Result.
The linkage comparison plot is shown in Figure 6.

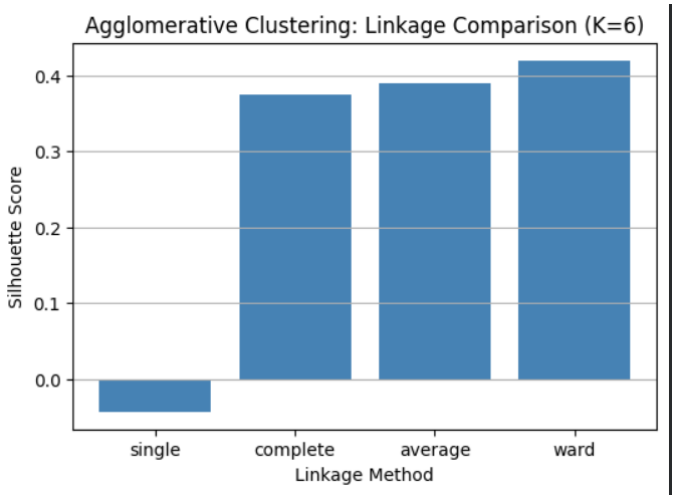

Figure 6: Silhouette comparison for different linkage methods (K=6).

Interpretation.
The bar chart clearly confirms:

- Single linkage performs poorly (negative score).
- Ward linkage achieves the highest clustering quality.
- Ward and KMeans behave similarly due to variance minimization.

Part (d): DBSCAN Clustering and Parameter Tuning

Objective.

In this part, we apply the density-based clustering algorithm DBSCAN (Density-Based Spatial Clustering of Applications with Noise).

Unlike KMeans and hierarchical clustering, DBSCAN:

- Does not require specifying the number of clusters.
- Can detect arbitrarily shaped clusters.
- Explicitly identifies noise points.

We evaluate different parameter combinations:

- $\varepsilon \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$
- `min_samples` $\in \{3, 5, 10\}$

Methodology.
For each parameter combination:

- We compute the number of clusters (excluding noise).
- We count the number of noise points.
- We compute silhouette score only on non-noise points.

Code Implementation.

```
        from sklearn.cluster import DBSCAN
        from sklearn.metrics import silhouette_score
        import numpy as np

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

        dbscan_results = []

        for eps in eps_values:
        for min_s in min_samples_values:
        dbscan = DBSCAN(eps=eps, min_samples=min_s)
        labels = dbscan.fit_predict(X_scaled)

        # number of clusters (excluding noise)
        n_clusters = len(set(labels)) - (1 if -1 in labels else
            0)
        n_noise = list(labels).count(-1)

        # silhouette only on non-noise points
        if n_clusters > 1:
```

```
        mask = labels != -1
        sil = silhouette_score(X_scaled[mask], labels[mask])
        else:
        sil = None

        dbscan_results.append({
            "eps": eps,
            "min_samples": min_s,
            "clusters": n_clusters,
            "noise_points": n_noise,
            "silhouette": sil
        })
```

Results.

The numerical results for all DBSCAN parameter combinations are summarized below:

ε	min_samples	Clusters	Noise	Silhouette
0.2	3	11	161	0.6459
0.2	5	1	195	—
0.2	10	0	200	—
0.4	3	10	59	0.4426
0.4	5	6	98	0.5190
0.4	10	2	170	0.7661
0.6	3	3	14	0.2149
0.6	5	2	28	0.2730
0.6	10	4	66	0.5296
0.8	3	1	3	—
0.8	5	1	6	—
0.8	10	1	23	—
1.0	3	1	1	—
1.0	5	1	2	—
1.0	10	1	5	—

Table 3: DBSCAN parameter exploration results.

(*Silhouette is not defined when fewer than 2 clusters are formed.*)

Detailed Analysis.

DBSCAN behavior strongly depends on ε and `min_samples`. We analyze their influence separately.

Effect of ε:

- Small ε (0.2): Very strict density requirement. Most points are labeled

as noise (up to 200 noise points). Although silhouette can be high (e.g., 0.7661), this occurs because only a very small subset of dense points is clustered, making the metric artificially inflated.

- Moderate ε (0.4 – 0.6): More balanced clustering behavior appears. The number of clusters becomes reasonable and noise proportion decreases.
- Large ε (0.8 – 1.0): Almost all points are merged into a single cluster. This degenerates DBSCAN into a trivial solution.

Effect of min_samples:

- Lower values (3) make it easier to form clusters.
- Higher values (10) require denser regions, increasing noise.

Important Observation on Silhouette Scores.
The highest silhouette score (0.7661 at $\varepsilon = 0.4$, min_samples=10) is misleading. It corresponds to only 2 clusters with 170 noise points. This means the silhouette is computed on a very small subset of data, which inflates the score.
A good clustering configuration should balance:

- Reasonable number of clusters
- Acceptable noise ratio
- High silhouette score

Best DBSCAN Configuration.
Among all configurations, the most balanced trade-off is:

$$\varepsilon = 0.4\text{, min\_samples = 5}$$

This setting yields:

- 6 clusters
- 98 noise points
- Silhouette score = 0.5190

Although it includes a considerable amount of noise, it produces a meaningful multi-cluster structure comparable to KMeans ($K = 6$).
Comparison with Other Methods.

- KMeans (K=6): Silhouette = 0.4284
- Ward Hierarchical: Silhouette = 0.4201
- Best DBSCAN (balanced): Silhouette = 0.5190

DBSCAN achieves a higher silhouette score but at the cost of discarding nearly half of the data as noise.

Conclusion for DBSCAN.

DBSCAN is capable of identifying dense customer segments and detecting outliers. However, the Mall Customers dataset does not exhibit strong density-separated clusters. Therefore, centroid-based and hierarchical methods provide more stable and interpretable segmentation.

Part (e): Visualization of Best Models on the 2D PCA Space

Objective.

The purpose of this part is to visualize the clustering results of the best configuration from each clustering family on the **2D PCA space**. This visualization helps interpret cluster separability and structure in a human-interpretable way.

As emphasized previously, PCA is used **only for visualization**, while models are trained in the standardized original feature space (`X_scaled`).

In this subsection, we first visualize the best **centroid-based** method:

KMeans with $K = 6$

Code Implementation (KMeans).

```
        from sklearn.cluster import KMeans

        kmeans = KMeans(n_clusters=6, random_state=83, n_init
            =10)
        kmeans_labels = kmeans.fit_predict(X_scaled)

        plt.figure(figsize=(6,5))
        plt.scatter(
        X_pca[:, 0], X_pca[:, 1],
        c=kmeans_labels, cmap="tab10", s=40
        )
        plt.xlabel("Principal Component 1")
        plt.ylabel("Principal Component 2")
        plt.title("KMeans Clustering (K=6) on PCA Space")
        plt.colorbar(label="Cluster Label")
        plt.grid(True)
        plt.show()
```

Result.
The KMeans clustering visualization on the PCA space is shown in Figure 7.

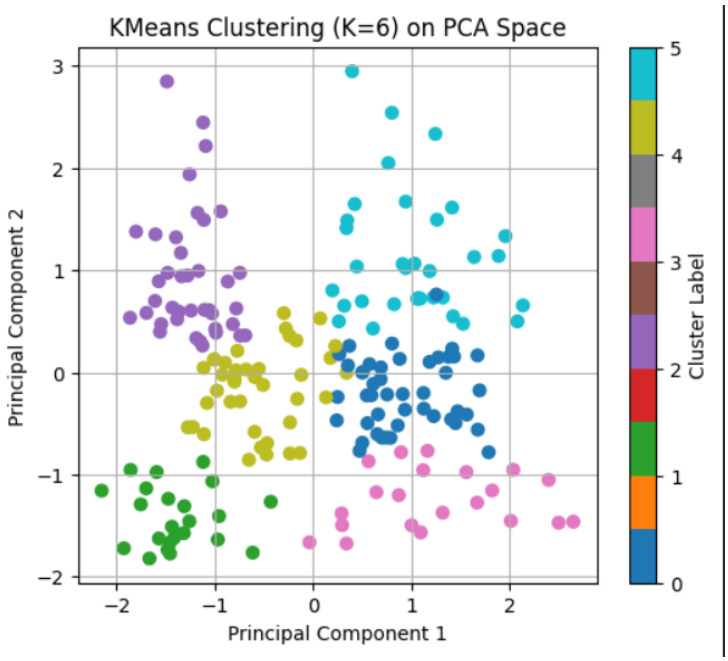

Figure 7: KMeans clustering ($K = 6$) visualized on the 2D PCA projection.

Analysis and Interpretation (KMeans).
The PCA visualization shows that KMeans produces a segmentation with **six relatively distinguishable groups** in the projected space.
Key observations include:

- Several clusters appear compact and well-separated (particularly on the left and bottom regions of the plot), indicating strong structure in the customer data.
- Some overlap is still visible in the central region. This is expected because:
  - PCA compresses information from 3D into 2D (information loss).
  - Real customer behavior often forms smooth transitions rather than strict boundaries.
- Despite minor overlaps, the overall cluster partitioning is consistent with the quantitative evaluation results in Part (b), where KMeans achieved the best silhouette score among the tested K values.

Overall, KMeans with $K = 6$ provides a **stable and interpretable** clustering structure, and its PCA visualization supports the conclusion that meaningful segments exist in the dataset.

Code Implementation (Agglomerative Clustering – Ward).

```
        from sklearn.cluster import AgglomerativeClustering

        agg = AgglomerativeClustering(
        n_clusters=6,
        linkage="ward"
        )
        agg_labels = agg.fit_predict(X_scaled)

        plt.figure(figsize=(6,5))
        plt.scatter(
        X_pca[:, 0], X_pca[:, 1],
        c=agg_labels, cmap="tab10", s=40
        )
        plt.xlabel("Principal Component 1")
        plt.ylabel("Principal Component 2")
        plt.title("Agglomerative Clustering (Ward, K=6) on PCA
            Space")
        plt.colorbar(label="Cluster Label")
        plt.grid(True)
        plt.show()
```

Result.

The Ward-linkage Agglomerative clustering result visualized on the PCA space is shown in Figure 8.

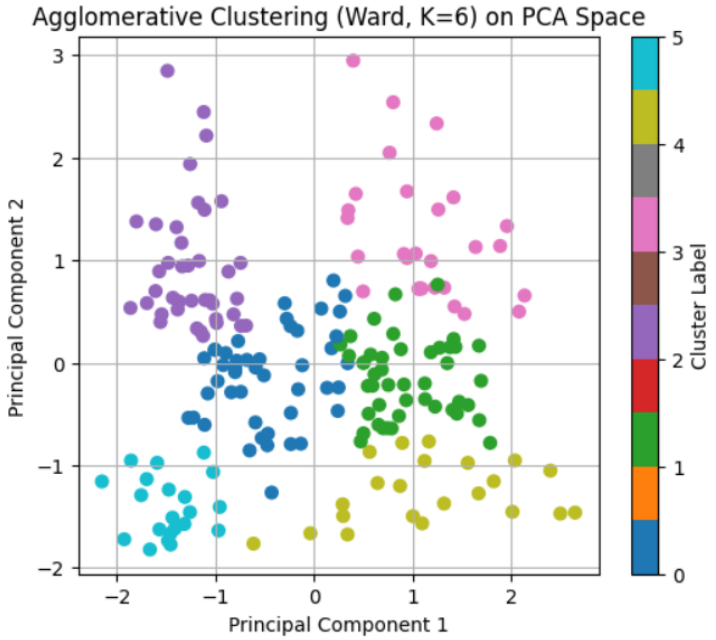

Figure 8: Agglomerative Clustering (Ward, $K = 6$) visualized on the 2D PCA projection.

Analysis and Interpretation (Ward).

The Ward hierarchical clustering visualization exhibits a structure that is broadly consistent with the KMeans segmentation, which is expected since Ward linkage merges clusters by minimizing the increase in within-cluster variance.

Key observations:

- Several clusters remain compact and appear clearly separated (e.g., the bottom-left region), suggesting that the hierarchical method successfully captures stable customer groups.
- Compared to KMeans, Ward tends to generate clusters that may look slightly more **uneven in size** on the PCA plane. This occurs because hierarchical clustering produces partitions based on merge decisions, rather than re-optimizing centroids as in KMeans.
- Some overlap is visible in central areas, similar to KMeans. This is partly due to:
  - information loss from projecting 3D standardized data to 2D PCA,
  - and the naturally continuous structure of customer behavior.

- The PCA plot supports the quantitative results from Part (c), where Ward linkage was the best hierarchical configuration and achieved a silhouette score close to KMeans.

Overall, Ward linkage provides a **high-quality hierarchical segmentation** that is visually interpretable and consistent with its strong silhouette performance. However, KMeans remains slightly superior quantitatively, and typically yields more centroid-balanced partitions.

Code Implementation (DBSCAN).

```
        from sklearn.cluster import DBSCAN
        import numpy as np

        dbscan = DBSCAN(eps=0.6, min_samples=10)
        db_labels = dbscan.fit_predict(X_scaled)

        plt.figure(figsize=(6,5))

        # non-noise points
        mask = db_labels != -1
        plt.scatter(
        X_pca[mask, 0], X_pca[mask, 1],
        c=db_labels[mask], cmap="tab10", s=40, label="Clusters"
        )

        # noise points
        plt.scatter(
        X_pca[~mask, 0], X_pca[~mask, 1],
        c="black", s=20, label="Noise"
        )

        plt.xlabel("Principal Component 1")
        plt.ylabel("Principal Component 2")
        plt.title("DBSCAN Clustering on PCA Space")
        plt.legend()
        plt.grid(True)
        plt.show()
```

Result.

The DBSCAN clustering visualization on the PCA space is shown in Figure 9.

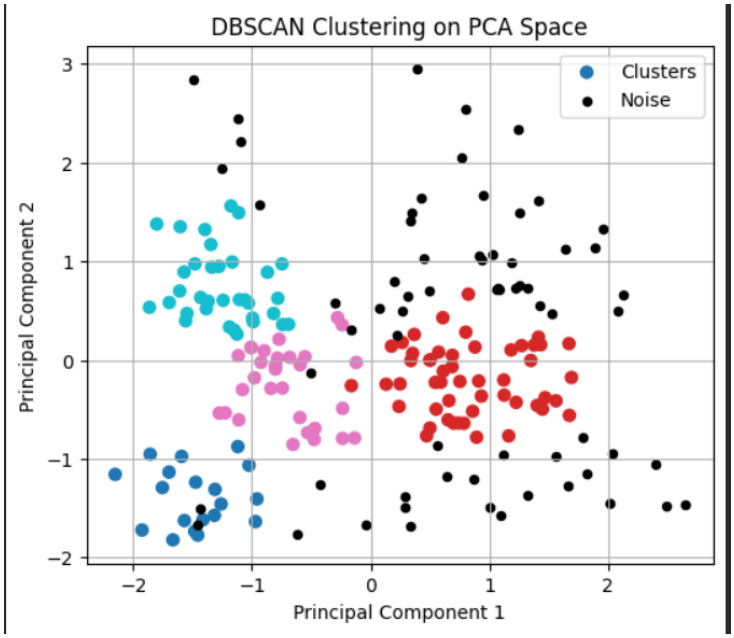

Figure 9: DBSCAN clustering visualized on the 2D PCA projection. Black points represent noise.

Analysis and Interpretation (DBSCAN).

Unlike KMeans and hierarchical clustering, DBSCAN explicitly identifies noise points (shown in black). Several important observations can be made:

- DBSCAN forms clusters based on density rather than geometric partitioning.
- A significant number of points are labeled as noise, particularly in low-density regions of the PCA space.
- The clusters that are formed appear compact and locally dense, but overall segmentation is less balanced compared to KMeans.
- Some customer groups are entirely absorbed into noise when density conditions are not satisfied.

This behavior indicates that the Mall Customer dataset does not exhibit strongly density-separated clusters. Instead, the structure is more continuous and variance-based, which explains why KMeans and Ward perform better.

Final Comparative Analysis of All Methods.

We now compare the best configuration of each clustering family:

- **KMeans (K=6)**: Silhouette = 0.4284

- **Agglomerative (Ward, K=6)**: Silhouette = 0.4201
- **DBSCAN (best balanced setting)**: Silhouette ≈ 0.5190 but with high noise proportion

Key conclusions:

- DBSCAN can achieve high silhouette scores, but this often occurs when many points are discarded as noise.
- KMeans provides stable, interpretable, and balanced clusters.
- Ward hierarchical clustering closely mirrors KMeans behavior.
- The dataset structure appears variance-driven rather than density-driven.

Overall Conclusion for Question 2.
Among all evaluated methods,

$$\textbf{KMeans with } K = 6$$

provides the most balanced and interpretable segmentation for the Mall Customers dataset.

It achieves strong cluster separation without discarding large portions of data as noise, and its structure is visually supported by PCA projection.

Therefore, for practical customer segmentation in this dataset, centroid-based clustering (KMeans) is the most appropriate choice.

Executive Summary

This question investigated customer segmentation using three major clustering families on the Mall Customers dataset:

- Centroid-based clustering (KMeans)
- Hierarchical clustering (Agglomerative – Ward linkage)
- Density-based clustering (DBSCAN)

All models were trained on standardized numerical features (Age, Annual Income, Spending Score), with `random_state = 83` ensuring reproducibility.

Main Findings:

- The Elbow method suggested $K \approx 4$, while silhouette analysis indicated that $K = 6$ provides the best separation for KMeans.
- KMeans ($K = 6$) achieved a silhouette score of 0.4284 and produced stable, well-balanced clusters.
- Hierarchical clustering with Ward linkage achieved a very similar performance (0.4201), confirming the variance-based structure of the dataset.
- DBSCAN was able to detect dense local structures and identify noise points, but high silhouette values were often associated with excessive noise labeling. Balanced configurations resulted in meaningful clusters but with significant data discarded as noise.

Overall Interpretation:

The Mall Customers dataset exhibits a structure that is primarily **variance-driven rather than density-separated**. Therefore, centroid-based and Ward hierarchical methods are more suitable than density-based clustering.

Final Recommendation:

For practical customer segmentation and interpretability,

KMeans with $K = 6$

is the most appropriate clustering model for this dataset.

It provides a strong balance between compactness, separation, and data coverage, while maintaining simplicity and reproducibility.

Question 3

Part (a): Q-learning Update Rule, Parameter Roles, and Off-Policy Nature

Q-learning update equation. Q-learning updates the action-value function $Q(s, a)$ after observing a transition (s_t, a_t, r_t, s_{t+1}) according to:

$$Q_{t+1}(s_t, a_t) \;=\; (1-\alpha)\, Q_t(s_t, a_t) \;+\; \alpha\Big(r_t \;+\; \gamma \max_a Q_t(s_{t+1}, a)\Big). \qquad (1)$$

This rule can be interpreted as a convex combination between the *previous estimate* $Q_t(s_t, a_t)$ and a *bootstrapped target*:

$$\text{Target} \;=\; r_t + \gamma \max_a Q_t(s_{t+1}, a). \qquad (2)$$

Equivalently, it can be written as an incremental update:

$$Q_{t+1}(s_t, a_t) \;=\; Q_t(s_t, a_t) \;+\; \alpha\Big(\underbrace{r_t + \gamma \max_a Q_t(s_{t+1}, a)}_{\text{target}} \;-\; Q_t(s_t, a_t)\Big), \quad (3)$$

where the term in parentheses is the temporal-difference (TD) error.

Role of α (learning rate). The parameter $\alpha \in (0, 1]$ controls how strongly new information influences the estimate:

- Large α (close to 1) places more weight on the most recent sample, which can speed up learning but may increase variance and instability.
- Small α yields slower but typically more stable learning by averaging information across many experiences.

Role of γ (discount factor). The discount factor $\gamma \in [0, 1]$ controls the importance of future rewards:

- $\gamma = 0$ makes the agent myopic (only immediate reward matters).
- $\gamma \to 1$ makes the agent far-sighted (future returns matter strongly).

Thus, γ determines the effective planning horizon of the agent.

Role of ε in the ε-greedy policy. During interaction with the environment, Q-learning commonly uses an ε-greedy behavior policy:

- With probability ε, the agent selects a random action (exploration).
- With probability $1 - \varepsilon$, it selects the action with the highest current Q-value (exploitation).

In practice, ε is often decayed over time (e.g., from 1.0 toward 0.01) to encourage exploration early and exploitation later.

Why Q-learning is off-policy. Q-learning is off-policy because the update uses the greedy action at the next state, $\max_a Q_t(s_{t+1}, a)$, regardless of the action actually taken by the behavior policy. That is, the agent may *behave* using an exploratory ε-greedy policy, but it *learns* the value of the greedy target policy. Hence, the behavior policy and target policy differ, which is precisely the definition of off-policy learning.

Part (b): Numerical Update of $Q(s_0, a_1)$

Given. The Q-table is initialized to zero, and the observed transition is:

$$(s_t = s_0,\ a_t = a_1,\ r_t = +2,\ s_{t+1} = s_1).$$

Also provided:

$$\max_a Q(s_1, a) = 1.5, \qquad \alpha = 0.2, \qquad \gamma = 0.9.$$

Since the table is initially zero:

$$Q_t(s_0, a_1) = 0.$$

Step 1: Compute the target.

$$\begin{aligned} \text{Target} &= r_t + \gamma \max_a Q_t(s_{t+1}, a) && (4)\\ &= 2 + 0.9 \times 1.5 && (5)\\ &= 2 + 1.35 && (6)\\ &= 3.35. && (7) \end{aligned}$$

Step 2: Apply the Q-learning update.

$$\begin{aligned} Q_{t+1}(s_0, a_1) &= (1 - \alpha) Q_t(s_0, a_1) + \alpha(\text{Target}) && (8)\\ &= (1 - 0.2) \times 0 + 0.2 \times 3.35 && (9)\\ &= 0 + 0.67 && (10)\\ &= 0.67. && (11) \end{aligned}$$

Final answer.

$$Q(s_0, a_1) \ = \ 0.67.$$

Part (c): Sources of Instability/Divergence and Practical Remedies

Factor 1: Poor reward scaling or highly variable rewards. If reward magnitudes are very large or inconsistent, the TD targets can become large, causing Q-values to grow rapidly and making learning unstable. This can lead to oscillations and poor convergence.

Practical remedies.

- **Reward clipping:** Clip rewards to a bounded interval (e.g., $[-1, 1]$) to prevent extreme updates.

- **Reward normalization/scaling:** Rescale rewards to maintain numerically stable target magnitudes.
- **Careful reward design:** Use shaping cautiously to avoid unintentionally introducing misleading gradients.

Factor 2: Inappropriate exploration schedule (ε too high or too low). If ε is too small early, the agent may not explore sufficiently and can converge to a suboptimal policy. If ε remains too large, the behavior remains overly random and the learning process may fail to stabilize.

Practical remedies.

- **ε-decay scheduling:** Start with a larger ε (strong exploration) and gradually decay it toward a small floor value to enable convergence.
- **Ensure adequate state-action visitation:** Longer training and diversified starting states help the agent observe sufficient transitions for stable value estimates.

Additional note (common in practice). Another frequent source of instability is an excessively large learning rate α. Reducing α (or using a decaying α) can significantly improve stability by reducing update variance, especially in noisy environments.