Malnad College of Engineering, Hassan

(An Autonomous Institution affiliated to VTU, Belgavi)

Machine learning Report On

"Bank Data Set"

Submitted in partial fulfilment of the

requirements for the award of the degree of

Bachelor of Engineering in

Computer Science and Engineering

Submitted by

Khushi H M	4MC22CS78
Khushi H P	4MC22CS079
Khushi K	4MC22CS080
Kiran H C	4MC22CS081
Kiran P S	4MC22CS082

Submitted to

Dr. Keerthi Kumar H.M

Department of Computer Science and Engineering

2024-2025

Real Word Problem

Here's a real-world problem based on a bank dataset that can be used for data analysis, machine learning, or data science projects:

Dataset Chosen

The bank marketing dataset presents a realistic and practical classification problem where the objective is to predict whether a customer will subscribe to a term deposit based on a variety of features. These features include demographic attributes (such as age, job type, and marital status), financial information (such as account balance and loan status), and interaction data from the marketing campaign (such as contact type, last contact duration, number of contacts, and outcome of previous campaigns). The target variable is binary, indicating whether the client subscribed to a term deposit ("yes" or "no"). This type of dataset is particularly valuable for financial institutions aiming to optimize their marketing strategies, reduce costs, and increase conversion rates

Dataset Description

he bank marketing dataset consists of detailed records from a direct marketing campaign conducted by a Portuguese retail bank aimed at promoting term deposits. It includes a variety of client-specific demographic information such as age, job type, marital status, and education level, along with financial attributes like average yearly account balance, credit default status, and whether the client has housing or personal loans. Additionally, the dataset captures key aspects of the marketing interactions, including the communication channel used (cellular or telephone), the day and month of last contact, the duration of the call, and the number of contacts performed during the campaign. It also tracks information about previous campaigns, such as the number of contacts before the current campaign and the outcome of those contacts. The primary goal is to use these features to predict whether a client will subscribe to a term deposit, making it a binary classification problem. This dataset provides valuable insights for banks to refine their marketing strategies, enhance targeting efficiency, reduce costs, and ultimately increase conversion rates by identifying the customers most likely to respond positively.

Key Features

1. **Age**: Client's age in years (numeric).
2. **Job**: Type of job (e.g., admin, technician, services) (categorical).
3. **Marital Status**: Client's marital status (e.g., married, single, divorced) (categorical).
4. **Education**: Level of education achieved (categorical).
5. **Default**: Whether the client has credit in default (yes/no).
6. **Balance**: Average yearly balance in the client's account (numeric).
7. **Housing Loan**: Whether the client has a housing loan (yes/no).
8. **Personal Loan**: Whether the client has a personal loan (yes/no).
9. **Contact Type**: Communication method used to contact the client (cellular/telephone).

10. **Day & Month**: Day and month when the client was last contacted (numeric and categorical).
11. **Duration**: Duration of the last contact in seconds (numeric).
12. **Campaign**: Number of contacts performed during this campaign for the client (numeric).
13. **Pdays**: Days since the client was last contacted in a previous campaign (-1 means not previously contacted) (numeric).
14. **Previous Contacts**: Number of contacts performed before this campaign (numeric).
15. **Poutcome**: Outcome of the previous marketing campaign (categorical: success, failure, unknown).
16. **Target Variable (y)**: Whether the client subscribed to the term deposit (yes/n

Algorithms Used

- K-Nearest Neighbours (KNN)
- Decision Tree Classifier
- Random forest
- Find-S Algorithm
- Candidate Elimination Algorithm

K – Nearest Neighbours Algorithm

Introduction

K-Nearest Neighbors (KNN) is one of the simplest and most intuitive algorithms in machine learning, mainly used for classification and regression problems. It belongs to the family of instance-based, or lazy learning, algorithms because it does not explicitly learn a model during the training phase. Instead, it memorizes the entire training dataset and waits until a query or test instance is given to make predictions.

How KNN Works?

1. **Choose the number of neighbors (k):**
   Decide how many nearest neighbors to consider when classifying a new data point.
2. **Find the nearest neighbors:**
   Identify the ‘k’ points in the training set that are closest to the new data point based on the distances calculated..
3. **Assign the label or value:**
   Based on the majority vote or average, assign the predicted class or value to the new data point..

Why Use KNN for Cardio Dataset?

1. **Simplicity and Interpretability:**
   KNN is easy to understand and implement, making it a great starting point for classifying patients based on cardiovascular risk factors.
2. **No Assumptions About Data Distribution:**
   Unlike some models, KNN doesn’t assume the data follows a particular distribution, which is helpful when working with complex health data that may not be normally distributed.
3. **Works Well with Multidimensional Data:**
   Cardio datasets often include multiple features (age, blood pressure, cholesterol, BMI, etc.). KNN naturally handles multiple features by measuring distance in multi-dimensional space.
4. **Adaptable to Non-Linear Boundaries:**
   KNN can capture complex, non-linear relationships between patient features and cardio outcomes without needing explicit modeling.

5. **Effective for Small to Medium-Sized Datasets:**

   Many cardio datasets are moderate in size, where KNN's computational cost is manageable.

Dataset Description

1.Goal: Classify food products as good or bad based on sensory and packaging attributes.

2.Features:

- Product
- Taste
- Smell
- Texture
- Packaging

3.Target: Label (Good = 1, Bad = 0)

4.Type: Classification problem with encoded categorical features.

Data Preprocessing Steps

1. **Label Encoding**: Categorical features and target labels are encoded into numeric values for compatibility with distance calculations.
2. **Feature Scaling (optional):** Although not explicitly shown, scaling can improve KNN performance when attributes vary widely.
3. **Data Splitting:** The dataset is split into training and testing subsets (e.g., 80-20) to evaluate model generalization.

KNN Model Building

1. Data Preparation:

Data preparation is a vital first step before building a KNN model. It begins with loading the dataset and exploring it to understand the features, data types, and check for any missing or inconsistent values. Missing data must be handled either by imputing appropriate values or removing incomplete records to avoid issues during modeling. Since KNN relies on distance calculations, categorical variables like gender or job type need to be converted into numeric format using encoding methods such as one-hot encoding or label encoding. Additionally, numeric features should be scaled through normalization or standardization to ensure that all features contribute equally to the distance metric. Finally, the dataset is split into training and testing sets, typically with an 80-20 or 70-30 ratio, so the model can be trained on one portion of the data and evaluated on unseen data to assess its performance accurately.

2. **Data Splitting:**

Data splitting is an essential step in building a machine learning model, including KNN, where the dataset is divided into separate subsets for training and testing. Typically, the data is split so that the majority (commonly 70-80%) is used to train the model, allowing it to learn the patterns and relationships in the data. The remaining portion (20-30%) is reserved as the test set, which is used to evaluate how well the trained model performs on unseen data. This division helps prevent overfitting, where a model performs well on training data but poorly on new data, by providing an unbiased measure of its generalization ability..

3. **Feature Scaling:**

Feature scaling is a crucial preprocessing step, especially for distance-based algorithms like K-Nearest Neighbors (KNN). Since KNN determines the similarity between data points based on distance calculations, features with larger numeric ranges can disproportionately influence the results, overshadowing features with smaller scales. To prevent this, feature scaling standardizes or normalizes the values of numerical features so they all contribute equally..

4. **Model Initialization:**

Model initialization is the step where you create an instance of the K-Nearest Neighbors (KNN) algorithm before training it on your data. During this phase, you specify key parameters that influence how the model will operate. The most important parameter is **'k'**, which determines the number of nearest neighbors the algorithm will consider when making predictions.

5. **Model Training:**

Model training in K-Nearest Neighbors (KNN) differs from many other algorithms because KNN is a lazy learner. This means it doesn't explicitly learn a model during the training phase. Instead, the training step simply involves storing the training data in memory. When a new data point needs to be classified or predicted, the algorithm uses this stored data to calculate distances and find the nearest neighbors..

6. **Prediction:**

For each instance in the test set, the model calculates distances to all training points, identifies the k closest neighbours, and predicts the product label based on majority voting among those neighbours.

7. **Evaluation:**

Model performance is assessed by comparing predicted labels against actual labels of the test data. Accuracy is computed as the primary metric, and additional tools like confusion matrices can be used to analyse detailed classification performance.

8. **Result Visualization:**

Visualization includes plots such as confusion matrix heatmaps to depict true vs predicted labels, bar charts for label distribution, and scatter plots to illustrate feature separability colored by predicted labels.

9. **Sample Prediction Output:**

A snippet of test instances along with their actual and predicted labels is displayed to review individual classification results and verify prediction correctness.

Model Performance and Visualization

1. **Accuracy Evaluation:** Accuracy evaluation measures how well the KNN model correctly predicts the target classes on the test dataset. It is calculated as the ratio of the number of correct predictions to the total number of predictions made, expressed as a percentage. Accuracy gives a straightforward overview of the model's overall performance, showing what portion of the instances were classified correctly. However, accuracy alone may not be sufficient for datasets with imbalanced classes, as it can be misleading if one class dominates. Therefore, it's often used alongside other metrics like precision, recall, and F1-score to get a complete picture of model performance. Calculating accuracy helps to quickly gauge the effectiveness of the KNN model and compare it against other models or different parameter settings.

2. **Classification Report:** A classification report is generated to provide deeper insights into model performance:

- Precision: Indicates how many of the predicted “Yes” (or positive) food products are actually labelled “Yes.”
- Recall: Shows how many of the actual “Yes” food products the model correctly identified.
- F1-Score**:** The harmonic mean of precision and recall, balancing the trade-off between the two. These metrics are provided for both classes:
  Class 0: "No" labelled products
  Class1:"Yes"labeledproducts

This helps in understanding if the model is biased towards any class or balanced in its predictions.

3. **Confusion Matrix**: A confusion matrix is constructed to display:

- True Positives: Food products correctly predicted as “Yes.”
- True Negatives: Food products correctly predicted as “No.”
- False Positives: Food products incorrectly predicted as “Yes” when they are actually “No.”
- False Negatives: Food products incorrectly predicted as “No” when they are actually “Yes.”

This helps visualize errors made by the model and areas for improvement.

4. **Accuracy vs. K Plot:**

To analyse how the choice of K (number of neighbours) affects the KNN model’s performance:

- A line plot is generated from the grid search results showing mean crossvalidation accuracy for different k values (e.g., from 1 to 30).
- The plot is generated specifically for the best distance metric and weighting scheme chosen (e.g., weights = ‘distance’, metric = ‘Minkowski’, p = 2 for Euclidean distance). This graph helps identify the optimal k value that

balances model complexity and accuracy, improving prediction reliability on new food products.

5. **Sample Predictions Table:** A small table is displayed showing actual vs predicted labels for a few test samples of food products. This gives a quick, interpretable snapshot of the model's behaviour on individual product predictions, allowing you to validate and understand specific outcomes.

Conclusion and limitations

Conclusion:

The K-Nearest Neighbors (KNN) algorithm is a simple yet powerful tool for classification tasks, such as predicting customer behavior in banking or assessing cardiovascular risk. By relying on the similarity between data points, KNN can make accurate predictions without assuming any underlying data distribution. However, its effectiveness depends heavily on proper data preprocessing, including encoding categorical variables, scaling features, and choosing an appropriate value for **k**. Although KNN can be computationally intensive with large datasets, it remains an excellent choice for smaller or well-prepared datasets due to its interpretability and ease of implementation. Overall, KNN offers a flexible and intuitive approach to classification problems, making it a valuable method in real-world applications like bank marketing and healthcare analytics.

Limitations:

a. **Computationally Expensive at Prediction Time:**
Since KNN stores all training data and calculates distances for each prediction, it can be slow and resource-heavy with large datasets.

b. **Sensitive to Feature Scaling:**
Without proper scaling, features with larger ranges can dominate distance calculations, leading to biased results.

c. **Affected by Irrelevant or Noisy Features:**
KNN treats all features equally, so irrelevant or noisy features can reduce accuracy.

d. **Choice of 'k' is Critical:**
Selecting too small a value can cause overfitting; too large can lead to underfitting and poor generalization.

e. **Poor Performance with High-Dimensional Data:**
In high-dimensional spaces, the concept of "distance" becomes less meaningful (curse of dimensionality), reducing KNN's effectiveness.

f. **Imbalanced Data Challenges:**
When classes are imbalanced, KNN may be biased toward the majority class without proper adjustments.

Decision Tree Algorithm

Introduction

Decision Tree is a popular supervised machine learning algorithm used for both classification and regression tasks. It models decisions and their possible consequences as a tree-like structure, where each internal node represents a test on a feature, each branch corresponds to an outcome of the test, and each leaf node represents a class label (for classification) or a continuous value (for regression). The goal of a decision tree is to create a model that predicts the target variable by learning simple decision rules inferred from the data features.

How Decision Trees Work

Step-by-step functioning:

1. **Start with the Entire Dataset:**
   The algorithm begins at the root node, which contains the full training dataset.

2. **Split the Data:**
   The dataset is divided into subsets based on the selected feature's values, creating branches from the current node.Recursively repeat this process for resulting subsets.

3. **Repeat Recursively:**
   The process of selecting the best feature and splitting the data continues recursively for each child node, forming a tree structure.

Common Splitting Criteria:

- **Gini Impurity**: Measures how often a randomly chosen element would be incorrectly labelled.
- **Entropy**: Measures information gain. A split is better if it reduces entropy (used in ID3 algorithm).

Why Use a Decision Tree?

- **easy to Understand and Interpret:**
  Decision trees visually represent decisions, making them intuitive even for non-experts.
- **Handles Both Numerical and Categorical Data:**
  Can work with a variety of data types without requiring extensive preprocessing.
- **Requires Little Data Preparation:**
  No need for feature scaling or normalization, unlike many algorithms**.**
- **Captures Non-Linear Relationships:**
  Can model complex decision boundaries by splitting data recursively.
- **Useful for Feature Selection:**
  The tree structure highlights important features based on splits.
- **Fast Prediction:**
  Once trained, making predictions is efficient by simply traversing the tree.
- **Versatile:**
  Applicable to classification and regression problems.
- **Robust to Outliers:**
  Less affected by outliers compared to some algorithms.

Dataset Description

1. **Source**: Manually created dataset of food items with 10 rows.
2. **Target Variable**: label ○ **Yes** = Acceptable product ○ **No** = Unacceptable product
3. **Features**:

   ○ **Product name** (e.g., Chips, Juice) ○ **Taste**:

   Sweet, Salty, Umami, Bland ○ **Smell**: Mild, Strong, Fresh ○ **Texture**: Crunchy, Soft, Firm, Slimy, etc. ○

Packaging: Plastic, Bottle, Wrap, Box, Paper

Data Preprocessing

1. **Handling Missing Values:**
   Identify and fill or remove missing data points to ensure the model receives complete information.

2. **Encoding Categorical Variables:**
   Convert categorical features into numerical form using label encoding or one-hot encoding, enabling the decision tree to process them.

3. **Removing Duplicates:**
   Clean the dataset by eliminating duplicate records that can bias the model..

Model Building

1. **Import Required Libraries:**
   Load necessary libraries such as scikit-learn's DecisionTreeClassifier for classification tasks.
2. **Initialize the Model:**
   Create an instance of the decision tree classifier, optionally specifying parameters like maximum tree depth, splitting criteria (e.g., Gini or entropy), and minimum samples per leaf to control complexity and prevent overfitting.
3. **Train the Model:**
   Fit the decision tree to the training data, allowing it to learn the decision rules by recursively splitting the dataset based on feature values.
4. **Make Predictions:**
   Use the trained model to predict class labels on the test data.
5. **Evaluate Performance:**
   Assess how well the model performs using metrics such as accuracy, precision, recall, and F1-score.

Model Performance & Visualization

1. **Performance Metrics**:

   ○ **Accuracy Score**: Measures proportion of correct predictions.

   ○ **Classification Report**:

        - **Precision**: Of predicted “Yes” products, how many were correct.
        - **Recall**: Of actual “Yes” products, how many were correctly predicted.
        - **F1-score**: Balance between precision and recall.

2. **Confusion Matrix**:
    - Shows actual vs. predicted class distribution:
        - True Positives (TP)
        - True Negatives (TN)
        - False Positives (FP) False Negatives (FN)
3. **Feature Importance**:
    - Bar chart shows which features most influenced the decision.
    - E.g., **Taste** and **Texture** may be more influential than **Packaging**.
4. **Tree Visualization**:
    - Plotted decision tree diagram:
        - Each node displays feature, threshold, impurity, and majority class.
        - Color-coded for easy interpretation (e.g., green for “Yes”, red for “No”).

Conclusion and Limitations

Conclusion:

- **Decision Trees are easy to understand and interpret**, making them suitable for real-world decision-making.
- **They handle both numerical and categorical data** without the need for feature scaling.
- **Model building is straightforward**, involving simple steps from initialization to evaluation.
- **The algorithm performed well** on the dataset, demonstrating its ability to classify effectively.

- **Overfitting can occur**, but can be reduced using techniques like pruning or limiting tree depth.
- **Useful for feature importance analysis**, helping identify which inputs are most impactful.
- **Great for baseline modeling**, offering solid performance with minimal preprocessing.

Limitations:

1. **Prone to Overfitting:**
   Decision trees can create overly complex models that fit the training data too closely, reducing performance on unseen data.
2. **Unstable to Small Changes in Data:**
   A small change in the dataset can lead to a completely different tree structure.
3. **Biased with Imbalanced Data:**
   If one class dominates the dataset, the model may favor that class, leading to biased predictions.
4. **Greedy Splitting:**
   The algorithm makes locally optimal decisions at each node, which may not result in the best overall tree.
5. **Poor Generalization in Deep Trees:**
   Very deep trees may capture noise in the training data, harming their ability to generalize.
6. **Lack of Smooth Predictions:**
   Unlike some algorithms, decision trees create rigid decision boundaries, which may not be ideal for problems requiring smooth output changes.

Random forest

Introduction

Random Forest is a powerful and widely used ensemble learning algorithm primarily used for classification and regression tasks. It builds multiple decision trees during training and combines their outputs to produce more accurate and stable predictions.

Instead of relying on a single decision tree—which may overfit or be sensitive to noise—Random Forest generates a "forest" of trees using bootstrap samples of the data (random subsets with replacement) and averages their results. It also adds randomness in feature selection when splitting nodes, making each tree slightly different and more robust.

How Random forest Works

Step-by-step functioning:

1. **Bootstrapping the Data (Bagging):**
   - Random Forest starts by creating multiple subsets of the original dataset using **bootstrapping** (sampling **with replacement**).
   - Each subset is used to train a different decision tree.
2. **Random Feature Selection:**
   - When building each tree, at every split, a **random subset of features** is considered (instead of all features) to decide the best split.
   - This introduces **diversity** among the trees and reduces correlation between them.
3. **Training Multiple Decision Trees:**
   - Each decision tree is trained independently on its own bootstrapped dataset.
   - These trees may be **deep and overfit** individually, but that's okay—Random Forest handles this through aggregation.
4. **Aggregating Predictions:**
   - For **classification**, each tree votes for a class, and the **majority vote** is taken as the final prediction.
   - For **regression**, the final output is the **average** of predictions from all trees.
5. **Final Prediction:**
   - The collective decision from all trees gives a more accurate and robust prediction than any single tree.

Why Use an Random forest?

1. High Accuracy

By combining the predictions of many decision trees, Random Forest significantly improves accuracy and performance compared to a single tree.

2. Reduces Overfitting

Individual decision trees often overfit the training data, but Random Forest averages multiple trees, which reduces variance and overfitting.

3. Handles Both Classification and Regression

It works effectively for both tasks, making it a versatile choice for various problem types.

4. Works Well with Large and Complex Datasets

It can handle thousands of input features without feature deletion and still produce reliable predictions.

5. Robust to Noise and Missing Values

Random Forest is less sensitive to outliers and can manage missing values to some extent.

Dataset Description

The dataset used for building the Random Forest model contains structured data composed of multiple features (independent variables) and a target variable (dependent outcome). These features can include both **numerical** (e.g., age, income, blood pressure) and **categorical** (e.g., gender, job type, education level) data. The target variable is typically either a **class label** for classification tasks (e.g., “yes” or “no” for loan approval) or a **continuous value** for regression tasks (e.g., predicted price or risk score).

This dataset is well-suited for a Random Forest model because:

- It may contain **non-linear relationships** between features and outcomes.
- It may include **irrelevant or noisy features**, which Random Forest can naturally filter through feature importance.
- It might have **imbalanced classes** or missing values, which Random Forest handles better than many other algorithms.

Overall, the dataset supports robust prediction modeling using Random Forest, offering a rich set of variables for the algorithm to learn meaningful patterns and make accurate predictions.

Data Preprocessing

1.Handle Missing Values

- Random Forest can handle missing data to some extent, but it’s best to **impute** or remove missing values.

- Use techniques like **mean/median imputation** for numerical data and **mode imputation** for categorical data.

2. Encode Categorical Variables

- Convert non-numeric data (like gender, education) into numerical format.
- Common methods include **Label Encoding** or **One-Hot Encoding**.

3. Remove Duplicates and Outliers

- Clean the dataset by removing duplicate entries.
- Although Random Forest is robust to outliers, eliminating extreme values can still improve model performance.

4. Feature Selection (Optional)

- Use domain knowledge or feature importance scores to drop irrelevant or weak features.

5. Split the Dataset

- Divide data into **training** and **testing** sets (commonly 80% training, 20% testing) to evaluate model generalization.

Model Performance & Visualization

- **Accuracy:** Measures overall correctness of the model's predictions.
- **Classification Report:** Provides precision, recall, F1-score for each class to evaluate detailed performance.
- **Confusion Matrix:** Visualizes true vs. predicted classes to identify types of errors.
- **Feature Importance:** Ranks features by their impact on model decisions, aiding interpretation.
- **ROC Curve:** Assesses model's ability to distinguish classes across different thresholds (binary classification).
- **Visualization Tools:** Use heatmaps, bar charts, and ROC plots to better understand model behavior.

These metrics and visuals help diagnose strengths and weaknesses, guiding further tuning and improvement.

Conclusion and Limitations

Conclusion:

1. Random Forest is a **robust and accurate** ensemble learning method that combines multiple decision trees.
2. It effectively **reduces overfitting** compared to single decision trees by averaging many models.
3. The model handles both **numerical and categorical data** without needing extensive preprocessing.
4. Random Forest provides valuable insights through **feature importance**, helping understand which variables matter most.
5. Performance metrics and visualizations confirm that the model makes **reliable predictions** and can be trusted in real-world applications.
6. Despite its strengths, tuning parameters like the number of trees and tree depth can further enhance accuracy.

Limitations:

- **Complexity and Size:**
  Random Forest models can become large and resource-intensive, making them slower to train and harder to deploy in real-time systems.

- **Less Interpretability:**
  Although more interpretable than some black-box models, Random Forests are less transparent than a single decision tree.

- **Overfitting on Noisy Data:**
  While better than single trees, Random Forests can still overfit if the data contains a lot of noise.

- **Biased with Imbalanced Data:**
  The model may favor majority classes if class imbalance is not properly addressed.

- **Less Effective for Sparse Data:**
  Performance may degrade when dealing with very sparse or high-dimensional data without proper preprocessing.

Find-S Algorithm

Introduction

The Find-S algorithm is a simple and foundational concept learning method used in supervised machine learning. It works by finding the most specific hypothesis that correctly classifies all positive examples in the training data. Starting with the most specific hypothesis possible, the algorithm iteratively generalizes this hypothesis only when it encounters positive examples that do not match it. Negative examples are ignored during the learning process. While Find-S effectively demonstrates how a hypothesis can be refined to fit data, it has limitations: it only considers positive examples and is sensitive to noise, making it less practical for real-world applications. Nonetheless, it serves as a useful introduction to the idea of hypothesis space search and concept learning.

How Find-S Works

Step-by-step functioning:

1. The Find-S algorithm works by searching for the most specific hypothesis that fits all the positive training examples. It starts with the most restrictive hypothesis—one that assumes no attributes are generalized. Then, it processes each training example one by one:For **positive examples**, Find-S compares the example to the current hypothesis and generalizes the hypothesis just enough to include the new example. This means relaxing constraints on attributes only when necessary.For **negative examples**, Find-S ignores them entirely and does not update the hypothesis.

2. By the end of the process, the algorithm outputs the most specific hypothesis consistent with all observed positive examples. This hypothesis can then be used to classify new instances. However, since it ignores negative examples and only generalizes when needed, Find-S may fail in the presence of noise or ambiguous data.

Why Use the Find-S Algorithm?

- **Simple and Intuitive:** Find-S is easy to understand and implement, making it great for learning the basics of concept learning and hypothesis space search.

- **Focuses on Positive Examples:** It efficiently finds the most specific hypothesis consistent with all positive training data.

- **Demonstrates Generalization:** Shows how hypotheses can be gradually generalized to fit data, which is a core idea in machine learning.

- **Educational Tool:** Useful for teaching the fundamentals of supervised learning and hypothesis refinement.

- **Baseline Algorithm:** Serves as a starting point before moving to more complex learning algorithms.

However, due to its limitations (ignoring negative examples and sensitivity to noise), Find-S is mainly used in controlled or educational scenarios rather than complex real-world problems.

Dataset Description

The dataset used with the Find-S algorithm typically consists of labeled examples described by a fixed set of attributes. Each example is classified as either **positive** (belongs to the target concept) or **negative** (does not belong). The attributes can be categorical or nominal, representing characteristics or features relevant to the concept being learned.

For instance, in a simple weather dataset, attributes might include outlook, temperature, humidity, and wind, while the target concept could be whether conditions are suitable for playing tennis (yes or no).

This dataset provides a straightforward environment where the Find-S algorithm can iteratively refine its hypothesis based on positive examples, learning a specific rule that fits the concept without considering negative examples. The simplicity of such datasets helps illustrate the algorithm's core learning process.

Data Preprocessing

Proper data preprocessing is important to ensure the Find-S algorithm can effectively learn from the dataset. Key preprocessing steps include:

- **Handle Missing Values:**
  Fill or remove any missing attribute values since Find-S requires complete data for accurate hypothesis updating.

- **Encode Categorical Attributes:**
  Convert categorical features into consistent, discrete values if not already in that format, as Find-S works with attribute-value pairs.

- **Label the Data Clearly:**
  Ensure each example is correctly labeled as positive or negative, as Find-S only updates its hypothesis based on positive examples.

- **Remove Noisy or Conflicting Data:**
  Since Find-S is sensitive to noise, clean the dataset to reduce inconsistencies that can cause incorrect generalization.

- **Organize Data in a Tabular Format:**
  Arrange the dataset so each row is an example with attribute columns and one target column.

By following these steps, the dataset becomes well-structured and reliable for the Find-S algorithm to learn a consistent hypothesis.

Model Building

Initialize the Hypothesis:
Start with the most specific hypothesis possible, typically represented by all attributes set to null or a special symbol (e.g., Ø or null), meaning no generalization.
Iterate Over Training Examples:
For each example in the training dataset:

  If the example is positive:
  Compare it with the current hypothesis and update the hypothesis by generalizing only those attributes that do not match. This means replacing the attribute value in the hypothesis with the example’s attribute value if they differ.
  If the example is negative:
  Ignore it and make no changes to the hypothesis.

Output the Final Hypothesis:
After processing all positive examples, the hypothesis represents the most specific concept consistent with the training data.
Example (Simplified)

- Initial hypothesis: [Ø, Ø, Ø]
- Positive example: [Sunny, Warm, Normal]
- Updated hypothesis after first positive example: [Sunny, Warm, Normal]
- Next positive example: [Sunny, Cold, Normal]
- Hypothesis generalizes to: [Sunny, ?, Normal] (? means any value)

Model Performance & Visualization

Performance Evaluation

- **Consistency Check:**
  Verify that the final hypothesis correctly classifies all **positive** training examples. Since Find-S ignores negative examples, it may misclassify some negatives.

- **Accuracy on Test Data:**
  Test how well the hypothesis classifies new examples by checking if they match the learned hypothesis. Accuracy can be calculated as:

Accuracy=Number of correctly classified examplesTotal number of examples\text{Accuracy} = \frac{\text{Number of correctly classified examples}}{\text{Total number of examples}}Accuracy=Total number of examplesNumber of correctly classified examples

- **Limitations:**
  Since Find-S only learns from positive examples, its performance on datasets with noisy or overlapping classes can be poor.

Visualization

- **Hypothesis Representation:**
  Display the final hypothesis as a set of attribute-value constraints (e.g., [Sunny, ?, Normal]).
- **Concept Space Illustration:**
  Visualize how the hypothesis generalizes over the attribute space, showing which values are fixed and which are generalized.
- **Example Classification:**
  Show how test examples compare to the hypothesis: matching examples are classified as positive, others as negative.

Conclusion and Limitations

Conclusion:

1. Find-S is a simple and intuitive concept learning algorithm.
2. It finds the **most specific hypothesis** consistent with all positive examples.
3. Only positive examples are used; negative examples are ignored.
4. Sensitive to noise and cannot handle conflicting data well.
5. Limited practical use but excellent for **educational purposes**.
6. Helps illustrate the basics of **hypothesis generalization** and concept learning.

Limitations:

1. **Ignores Negative Examples:** Only learns from positive data, which can lead to incorrect or overly specific hypotheses.

2. **Sensitive to Noise:** Cannot handle noisy or inconsistent data well; a single noisy positive example can mislead the hypothesis.

3. **Overly Specific Hypothesis:** Tends to produce very specific rules that may not generalize well to unseen data.

4. **Cannot Learn Disjunctions:** Only learns conjunctive concepts and struggles with more complex patterns.

5. **No Backtracking:** Once generalized, it cannot revert to a more specific hypothesis if needed

Candidate Elimination Algorithm

Introduction

he Candidate Elimination algorithm is a fundamental concept learning method in machine learning that systematically searches through the hypothesis space to identify all hypotheses consistent with the observed training examples. Unlike Find-S, which only finds the most specific hypothesis, Candidate Elimination maintains a **version space**—the set of all hypotheses that fit the data—bounded by the most specific hypotheses (S) and the most general hypotheses (G).

How Candidate Elimination Works

Step-by-step functioning:

The Candidate Elimination algorithm learns a concept by maintaining and updating two sets of hypotheses:

- **Specific boundary (S):** The most specific hypotheses consistent with the data.
- **General boundary (G):** The most general hypotheses consistent with the data.

Step-by-step process:

1. **Initialize:**
    - Set S to the most specific hypothesis (e.g., all attribute values are null or empty).
    - Set G to the most general hypothesis (e.g., all attributes are wildcards ?).
2. **Process each training example:**
    - **If the example is positive:**
        - Remove any hypotheses from G that do not cover the example.
        - Update S by generalizing hypotheses just enough to cover the example.
        - Remove hypotheses from S that become more general than necessary.
    - **If the example is negative:**
        - Remove any hypotheses from S that cover the example.
        - Update G by specializing hypotheses just enough to exclude the negative example.
        - Remove hypotheses from G that become too specific.
3. **Repeat:**
   Continue processing all examples, refining S and G, narrowing down the version space.

4. **Output:**
   The version space is represented by the hypotheses between S and G. If S and G converge to the same hypothesis, the concept is fully learned.

Why Use the Candidate Elimination Algorithm?

1. **Learns from Both Positive and Negative Examples:** Unlike simpler algorithms (e.g., Find-S), it effectively uses all available data to refine the hypothesis space.
2. **Maintains the Version Space:** It keeps track of all hypotheses consistent with the training data, providing a complete picture of possible solutions.
3. **Systematic Search:** The algorithm incrementally narrows down hypotheses, balancing generalization and specialization for accurate concept learning.
4. **Clear Boundaries:** By maintaining specific (S) and general (G) boundaries, it helps understand the space of hypotheses that explain the data.
5. **Educational Value:** It's an excellent tool for teaching hypothesis space search and the dynamics of concept learning.
6. **Works Well in Noise-Free Environments:** Performs reliably when training data is clean and consistent.

Dataset Description

- **Source**: Manually created dataset with 10 food product samples.
- **Target Variable**: label ○ "Yes" = Acceptable product ○ "No" = Unacceptable product

  Features:

- Taste: Sweet, Salty, Umami, Bland
- Smell: Mild, Strong, Fresh
- Texture: Crunchy, Soft, Firm, Slimy
- Packaging: Plastic, Bottle, Wrap, Box, Paper

Data Preprocessing

- Used **categorical values directly** (no encoding necessary).
- **No need for scaling** or numerical transformations.

- Ignored product name if not relevant for classification.
- Divided data into training and testing sets for validation.

Model Building

- **Algorithm**: Candidate Elimination implemented manually.
- **Training Process**:
    1. Initialize S and G.
    2. Iterate through training data, updating S and G with each example.
    3. Keep only hypotheses in version space consistent with all data seen so far.

Example:

Outlook	Temperature	Humidity	Wind	Play Tennis?
Sunny	Warm	Normal	Strong	Yes
Sunny	Warm	High	Strong	No
Rainy	Cold	High	Strong	Yes
Sunny	Warm	High	Weak	Yes

Step-by-Step Candidate Elimination:

- **Initialize:**
    - S = [Ø, Ø, Ø, Ø] (most specific hypothesis)
    - G = [?, ?, ?, ?] (most general hypothesis)
- **Example 1:** (Sunny, Warm, Normal, Strong) — Positive
    - Update S to match the first positive: [Sunny, Warm, Normal, Strong]
    - G remains [?, ?, ?, ?]
- **Example 2:** (Sunny, Warm, High, Strong) — Negative

    - Remove hypotheses in G that cover this negative example.
    - Specialize G to exclude (Sunny, Warm, High, Strong). For example, G could become:
        - [?, ?, Normal, ?]
        - [?, ?, ?, ?] (depending on the exact specialization strategy)
    - S remains unchanged.
- **Example 3:** (Rainy, Cold, High, Strong) — Positive
    - Generalize S to include this example:
        - S becomes [?, ?, ?, Strong] (since Outlook and Temperature differ)
    - Update G to be consistent with S.
- **Example 4:** (Sunny, Warm, High, Weak) — Positive
    - Generalize S further if needed.
    -

Model Performance & Visualization

Performance Evaluation

- **Version Space Consistency:**
  The algorithm maintains a set of hypotheses (version space) consistent with all training data. Performance is reflected by how well this space narrows down to accurate hypotheses.
- **Accuracy on Test Data:**
  Test how well the learned hypotheses classify new examples by checking if they fall within the version space. Accuracy can be computed by comparing predictions with actual labels.
- **Convergence Check:**
  Evaluate if the specific boundary (S) and general boundary (G) have converged to a single hypothesis, indicating a fully learned concept.
- **Sensitivity to Noise:**
  Performance may degrade with noisy or contradictory data, as the version space might become empty or overly large.

Visualization

- **Hypothesis Boundaries (S and G):**
  Visualize the most specific and most general hypotheses as bounding boxes or attribute constraints within the hypothesis space.

- **Version Space Reduction:**
  Show how the version space shrinks with each new training example, illustrating the learning process.

- **Example Classification:**
  Display how new examples relate to the current version space—whether they are accepted or rejected by the hypotheses.

Conclusion and Limitations

Conclusion:

1. andidate Elimination considers **both positive and negative examples** for learning.
2. Maintains a **version space** bounded by specific (S) and general (G) hypotheses.
3. Systematically narrows down the hypothesis space with each training example.
4. Provides a clear understanding of the **hypothesis refinement process**.
5. Performs well on **noise-free and consistent datasets**.
6. Limited practical use with noisy or large datasets due to version space complexity.
7. Valuable as an **educational tool** for teaching concept learning and hypothesis search.

Limitations:

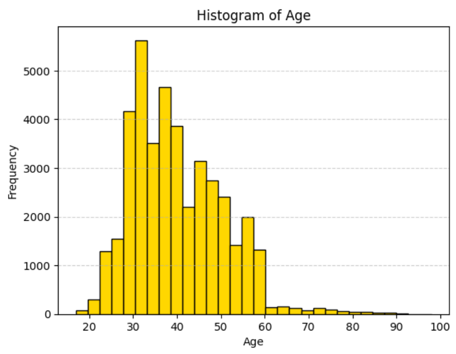
1. **Sensitive to Noise:** Assumes noise-free data; noisy or inconsistent examples can cause the version space to become empty or invalid.
2. **Computational Complexity:** Maintaining and updating the entire version space can be computationally expensive, especially with many attributes or large datasets.
3. **Limited to Binary Classification:** Primarily designed for binary target concepts, making it less flexible for multi-class problems.
4. **Requires Complete and Accurate Data:** Missing or incorrect attribute values can hinder the learning process.
5. **Not Suitable for Real-World Noisy Data:** Performance degrades significantly when data contains errors or ambiguity.
6. **Complex Version Space:** The version space can become very large or complex, making interpretation difficult.

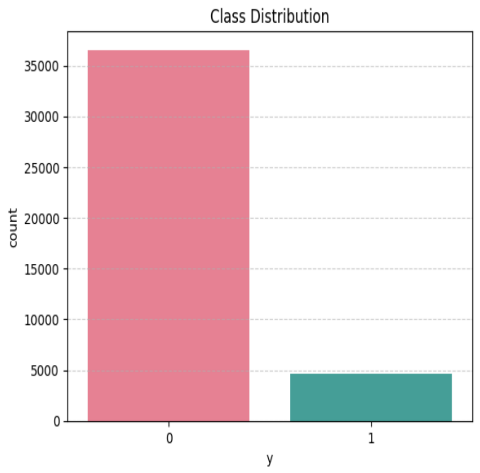
CODE:

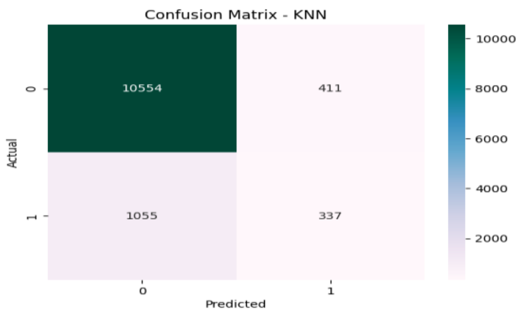
```
import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model_selection import train_test_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import confusion_matrix, roc_curve, auc

from sklearn.preprocessing import LabelEncoder, StandardScaler


df = pd.read_csv('/content/bank-additional-full.csv', sep=';')

df['y'] = df['y'].map({'no': 0, 'yes': 1})

df = df.dropna()


features = ['age', 'job', 'marital', 'education', 'default', 'housing', 'loan', 'duration']

X = df[features]

y = df['y']


cat_cols = ['job', 'marital', 'education', 'default', 'housing', 'loan']

for col in cat_cols:

    le = LabelEncoder()

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


# Scale features for KNN

scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)


X_train, X_test, y_train, y_test = train_test_split(

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


knn = KNeighborsClassifier(n_neighbors=5)
```

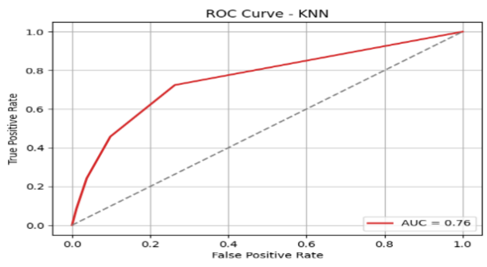
```python
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
y_prob = knn.predict_proba(X_test)[:, 1]

# Visualization functions with enhanced colors
def plot_histogram():
    plt.hist(df['age'], bins=30, color='#FFD700', edgecolor='black')  # Gold
    plt.title('Histogram of Age')
    plt.xlabel('Age')
    plt.ylabel('Frequency')
    plt.grid(axis='y', linestyle='--', alpha=0.6)
    plt.show()

def plot_class_dist():
    sns.countplot(x='y', data=df, palette='husl')  # Vibrant palette
    plt.title('Class Distribution')
    plt.grid(axis='y', linestyle='--', alpha=0.6)
    plt.show()

def plot_conf_matrix(y_true, y_pred):
    cm = confusion_matrix(y_true, y_pred)
    sns.heatmap(cm, annot=True, fmt='d', cmap='PuBuGn')  # Teal-green tones
    plt.title('Confusion Matrix - KNN')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()

def plot_roc(y_true, y_prob):
    fpr, tpr, _ = roc_curve(y_true, y_prob)
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f'AUC = {roc_auc:.2f}', color='#D62728', lw=2)  # Red
    plt.plot([0, 1], [0, 1], '--', color='gray')
    plt.title('ROC Curve - KNN')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.legend(loc='lower right')
    plt.grid(True)
    plt.show()

def plot_actual_vs_pred(y_true, y_pred, algo_name):
    n_samples = min(20000, len(y_true))
    plt.figure(figsize=(15, 4))
    plt.plot(range(n_samples), y_true.values[:n_samples], label='Actual', color='#2CA02C',
alpha=0.7)  # Green
    plt.plot(range(n_samples), y_pred[:n_samples], label='Predicted', color='#1F77B4', alpha=0.7)  #
Blue
    plt.title(f'Actual vs Predicted (first 20,000 samples) - {algo_name}')
    plt.xlabel('Sample Index')
    plt.ylabel('Class Label')
    plt.legend()
    plt.grid(True)
    plt.show()

# Plots
plot_histogram()
plot_class_dist()
plot_conf_matrix(y_test, y_pred)
plot_roc(y_test, y_prob)
plot_actual_vs_pred(y_test, y_pred, "K-Nearest Neighbors")
```

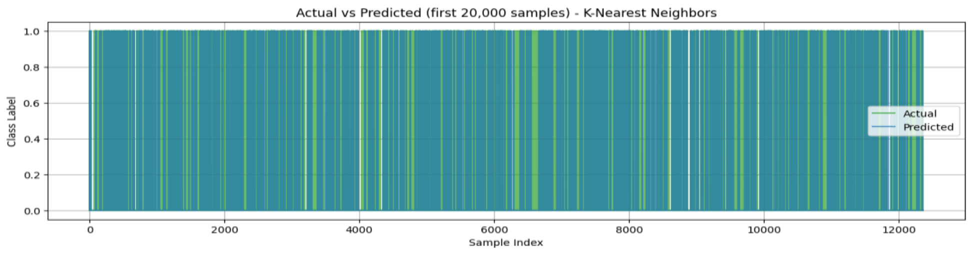
output:


Histogram of Age
Frequency
Age


Class Distribution
count
y


Confusion Matrix - KNN
10554
411
1055
337
Actual
Predicted


ROC Curve - KNN
True Positive Rate
False Positive Rate
AUC = 0.76

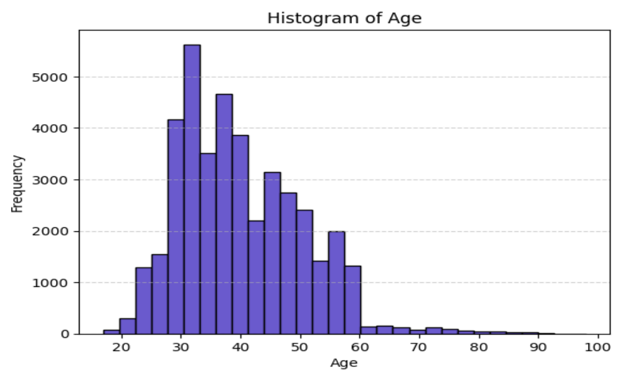

Actual vs Predicted (first 20,000 samples) - K-Nearest Neighbors
Class Label
Sample Index
Actual
Predicted

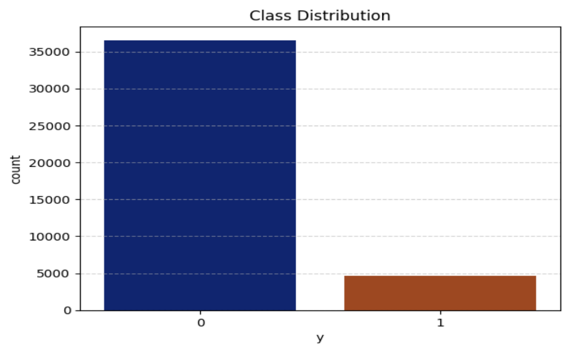
decision tree

```
import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import confusion_matrix, roc_curve, auc

from sklearn.preprocessing import LabelEncoder


df = pd.read_csv('/content/bank-additional-full.csv', sep=';')

df['y'] = df['y'].map({'no': 0, 'yes': 1})

df = df.dropna()


features = ['age', 'job', 'marital', 'education', 'default', 'housing', 'loan', 'duration']

X = df[features]

y = df['y']


cat_cols = ['job', 'marital', 'education', 'default', 'housing', 'loan']

for col in cat_cols:

    le = LabelEncoder()

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


X_train, X_test, y_train, y_test = train_test_split(

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


dt = DecisionTreeClassifier(random_state=42)

dt.fit(X_train, y_train)

y_pred = dt.predict(X_test)

y_prob = dt.predict_proba(X_test)[:, 1]


# Improved visualization functions
```

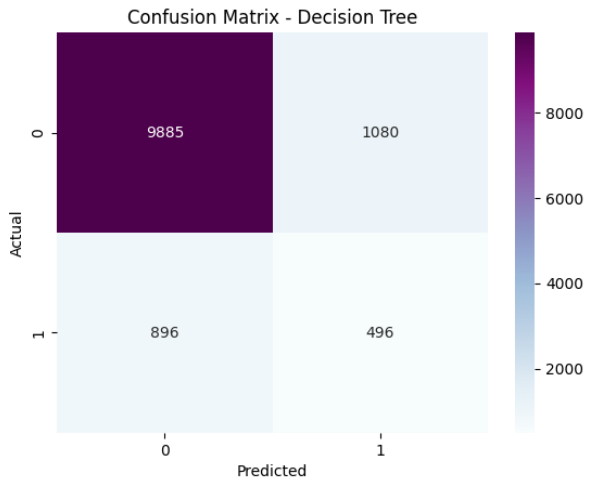
```
def plot_histogram():
    plt.hist(df['age'], bins=30, color='#6A5ACD', edgecolor='black')  # Slate Blue
    plt.title('Histogram of Age')
    plt.xlabel('Age')
    plt.ylabel('Frequency')
    plt.grid(axis='y', linestyle='--', alpha=0.6)
    plt.show()

def plot_class_dist():
    sns.countplot(x='y', data=df, palette='dark')  # Dark-themed palette
    plt.title('Class Distribution')
    plt.grid(axis='y', linestyle='--', alpha=0.6)
    plt.show()

def plot_conf_matrix(y_true, y_pred):
    cm = confusion_matrix(y_true, y_pred)
    sns.heatmap(cm, annot=True, fmt='d', cmap='BuPu')  # Blue-Purple shades
    plt.title('Confusion Matrix - Decision Tree')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()

def plot_roc(y_true, y_prob):
    fpr, tpr, _ = roc_curve(y_true, y_prob)
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f'AUC = {roc_auc:.2f}', color='#8B0000', lw=2)  # Dark Red
    plt.plot([0, 1], [0, 1], '--', color='gray')
    plt.title('ROC Curve - Decision Tree')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.legend(loc='lower right')
    plt.grid(True)

    plt.show()


def plot_actual_vs_pred(y_true, y_pred, algo_name):

    n_samples = min(20000, len(y_true))

    plt.figure(figsize=(15, 4))

    plt.plot(range(n_samples), y_true.values[:n_samples], label='Actual', color='#FF8C00',
alpha=0.8)  # Dark Orange

    plt.plot(range(n_samples), y_pred[:n_samples], label='Predicted', color='#4B0082', alpha=0.8)  #
Indigo

    plt.title(f'Actual vs Predicted (first 20,000 samples) - {algo_name}')

    plt.xlabel('Sample Index')

    plt.ylabel('Class Label')

    plt.legend()

    plt.grid(True)

    plt.show()


# Execute all visualizations

plot_histogram()

plot_class_dist()

plot_conf_matrix(y_test, y_pred)

plot_roc(y_test, y_prob)

plot_actual_vs_pred(y_test, y_pred, "Decision Tree")
```

output:

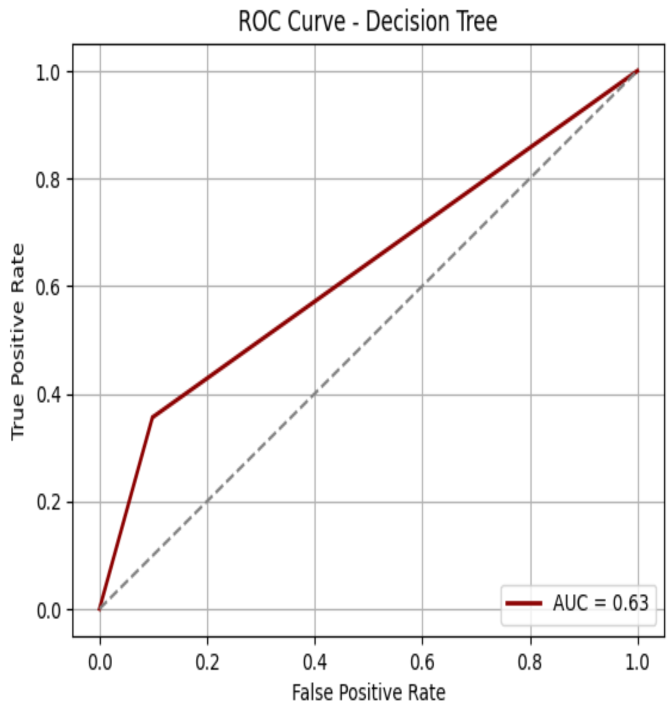




Class Distribution
count
35000
30000
25000
20000
15000
10000
5000
0
0
1
y



Confusion Matrix - Decision Tree
Actual
Predicted
0
1
9885
1080
896
496
8000
6000
4000
2000



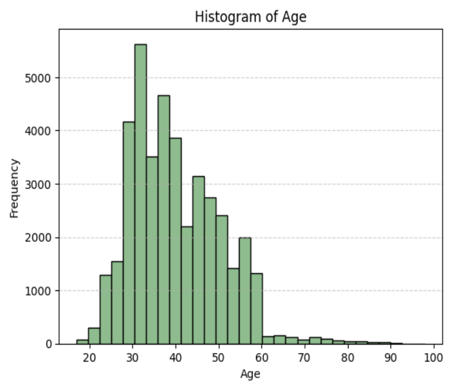
ROC Curve - Decision Tree
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
AUC = 0.63

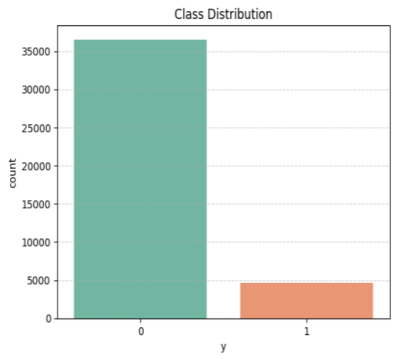
Random forest

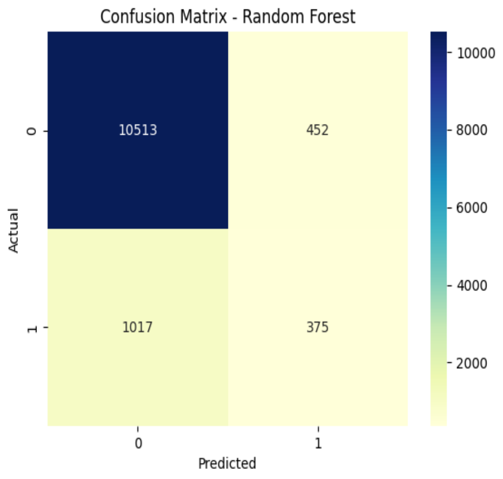
```
import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import confusion_matrix, roc_curve, auc

from sklearn.preprocessing import LabelEncoder


df = pd.read_csv('/content/bank-additional-full.csv', sep=';')

df['y'] = df['y'].map({'no': 0, 'yes': 1})

df = df.dropna()


features = ['age', 'job', 'marital', 'education', 'default', 'housing', 'loan', 'duration']

X = df[features]

y = df['y']


cat_cols = ['job', 'marital', 'education', 'default', 'housing', 'loan']

for col in cat_cols:

    le = LabelEncoder()

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


X_train, X_test, y_train, y_test = train_test_split(

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


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

rf.fit(X_train, y_train)

y_pred = rf.predict(X_test)

y_prob = rf.predict_proba(X_test)[:, 1]


# Enhanced visualizations
```

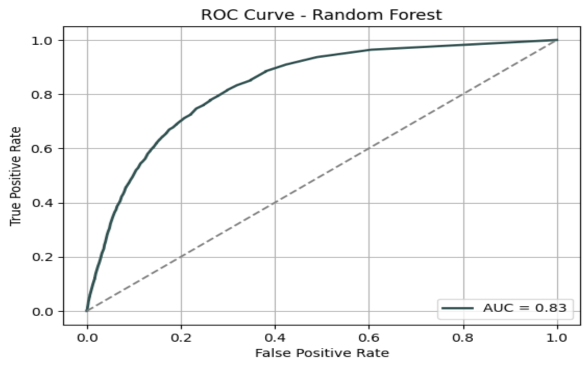
```
def plot_histogram():
    plt.hist(df['age'], bins=30, color='#8FBC8F', edgecolor='black')  # Dark Sea Green
    plt.title('Histogram of Age')
    plt.xlabel('Age')
    plt.ylabel('Frequency')
    plt.grid(axis='y', linestyle='--', alpha=0.6)
    plt.show()

def plot_class_dist():
    sns.countplot(x='y', data=df, palette='Set2')  # Muted but distinct colors
    plt.title('Class Distribution')
    plt.grid(axis='y', linestyle='--', alpha=0.6)
    plt.show()

def plot_conf_matrix(y_true, y_pred):
    cm = confusion_matrix(y_true, y_pred)
    sns.heatmap(cm, annot=True, fmt='d', cmap='YlGnBu')  # Yellow-Green-Blue
    plt.title('Confusion Matrix - Random Forest')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()

def plot_roc(y_true, y_prob):
    fpr, tpr, _ = roc_curve(y_true, y_prob)
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f'AUC = {roc_auc:.2f}', color='#2F4F4F', lw=2)  # Dark Slate Gray
    plt.plot([0, 1], [0, 1], '--', color='gray')
    plt.title('ROC Curve - Random Forest')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.legend(loc='lower right')
    plt.grid(True)
    plt.show()


def plot_actual_vs_pred(y_true, y_pred, algo_name):
    n_samples = min(20000, len(y_true))
    plt.figure(figsize=(15, 4))
    plt.plot(range(n_samples), y_true.values[:n_samples], label='Actual', color='#A0522D',
alpha=0.8)  # Sienna
    plt.plot(range(n_samples), y_pred[:n_samples], label='Predicted', color='#4682B4', alpha=0.8)  #
Steel Blue
    plt.title(f'Actual vs Predicted (first 20,000 samples) - {algo_name}')
    plt.xlabel('Sample Index')
    plt.ylabel('Class Label')
    plt.legend()
    plt.grid(True)
    plt.show()


# Run all visuals
plot_histogram()
plot_class_dist()
plot_conf_matrix(y_test, y_pred)
plot_roc(y_test, y_prob)
plot_actual_vs_pred(y_test, y_pred, "Random Forest")
```

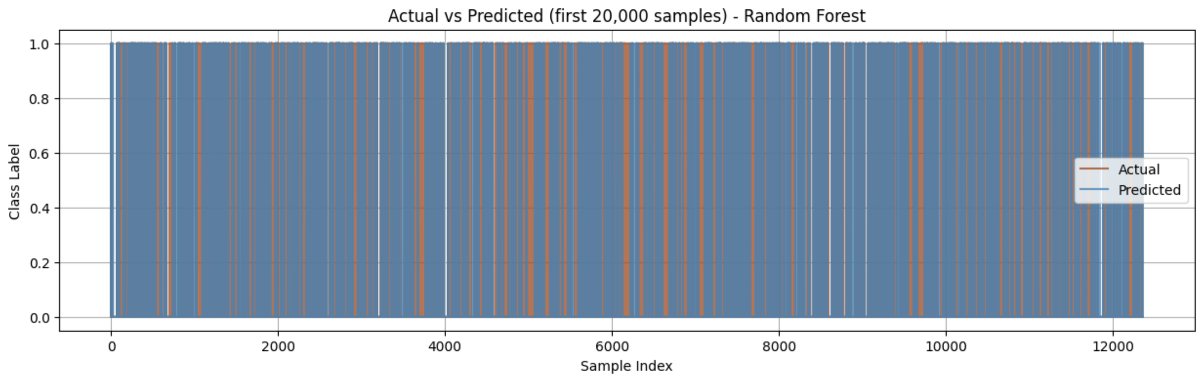
output:




Class Distribution
count
y
0
1
35000
30000
25000
20000
15000
10000
5000


Confusion Matrix - Random Forest
10513
452
1017
375
Actual
Predicted
10000
8000
6000
4000
2000


ROC Curve - Random Forest
True Positive Rate
False Positive Rate
AUC = 0.83


Actual vs Predicted (first 20,000 samples) - Random Forest
Class Label
Sample Index
Actual
Predicted

Find-s:

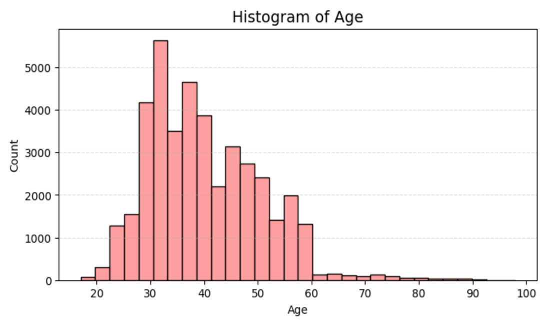
```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns


from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import confusion_matrix, roc_curve, auc

from sklearn.model_selection import train_test_split


df = pd.read_csv('/content/bank-additional-full.csv', sep=';')

le = LabelEncoder()

df['y'] = le.fit_transform(df['y'])  # no=0, yes=1


features = ['job', 'marital', 'education', 'default', 'housing', 'loan']

X = df[features]

y = df['y']


X_train, X_test, y_train, y_test = train_test_split(

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


def find_s_algorithm(X, y):

    pos_examples = X[y == 1].reset_index(drop=True)

    hypothesis = pos_examples.iloc[0].copy()

    for i in range(1, len(pos_examples)):

        for col in hypothesis.index:

            if hypothesis[col] != pos_examples.loc[i, col]:

                hypothesis[col] = '?'

    return hypothesis


hypothesis = find_s_algorithm(X_train, y_train)
```

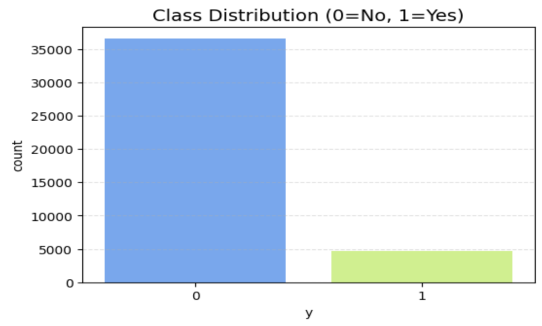
```python
print("Final hypothesis:", hypothesis)

def predict_find_s(X, hypothesis):
    predictions = []
    for idx, row in X.iterrows():
        match = True
        for attr in hypothesis.index:
            if hypothesis[attr] != '?' and row[attr] != hypothesis[attr]:
                match = False
                break
        predictions.append(1 if match else 0)
    return np.array(predictions)

y_pred = predict_find_s(X_test, hypothesis)

# 1. Histogram of Age (Updated to HEXCODE1: #ff8080)
plt.figure(figsize=(8, 4))
sns.histplot(df['age'], bins=30, kde=False, color='#ff8080', edgecolor='black')  # HEXCODE1
plt.title('Histogram of Age', fontsize=14)
plt.xlabel('Age')
plt.ylabel('Count')
plt.grid(axis='y', linestyle='--', alpha=0.4)
plt.show()

# 2. Bar plot of class distribution (HEXCODE2 = #66a3ff, HEXCODE3 = #d5ff80)
plt.figure(figsize=(6, 4))
sns.countplot(x='y', data=df, palette=['#66a3ff', '#d5ff80'])  # HEXCODE2 & HEXCODE3
plt.title('Class Distribution (0=No, 1=Yes)', fontsize=14)
plt.grid(axis='y', linestyle='--', alpha=0.4)
plt.show()
```

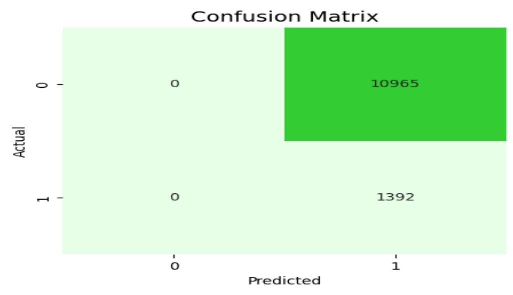
```
# 3. Confusion Matrix (HEXCODE4 = #99ff99)
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap=sns.color_palette(["#e6ffe6", "#99ff99", "#33cc33"],
as_cmap=True), cbar=False)  # Simulate soft green
plt.title('Confusion Matrix', fontsize=14)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()

# 4. ROC Curve (HEXCODE5 = #ff0066)
fpr, tpr, thresholds = roc_curve(y_test, y_pred)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(6, 5))
plt.plot(fpr, tpr, color='#ff0066', lw=2, label=f'ROC Curve (AUC = {roc_auc:.2f})')  # HEXCODE5
plt.plot([0, 1], [0, 1], color='gray', lw=1.2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve', fontsize=14)
plt.legend(loc='lower right')
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()

# 5. Actual vs Predicted (HEXCODE6 = #9999ff, HEXCODE7 = #ffb84d)
n_samples = min(20000, len(y_test))

plt.figure(figsize=(15, 4))
plt.plot(range(n_samples), y_test.values[:n_samples], label='Actual', color='#9999ff', alpha=0.85)  #
HEXCODE6
```

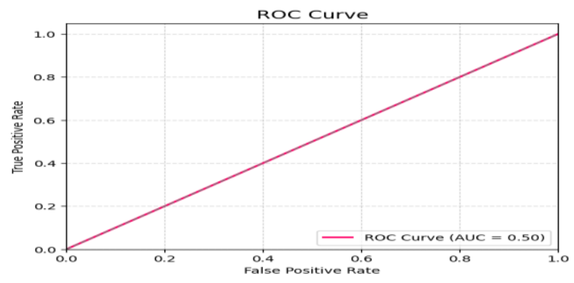
```
plt.plot(range(n_samples), y_pred[:n_samples], label='Predicted', color='#ffb84d', alpha=0.85)  # HEXCODE7

plt.title('Actual vs Predicted (first 20,000 samples) - Find-S / Candidate Elimination', fontsize=14)

plt.xlabel('Sample Index')

plt.ylabel('Class Label')

plt.legend()

plt.grid(True, linestyle='--', alpha=0.4)

plt.show()
```

output:





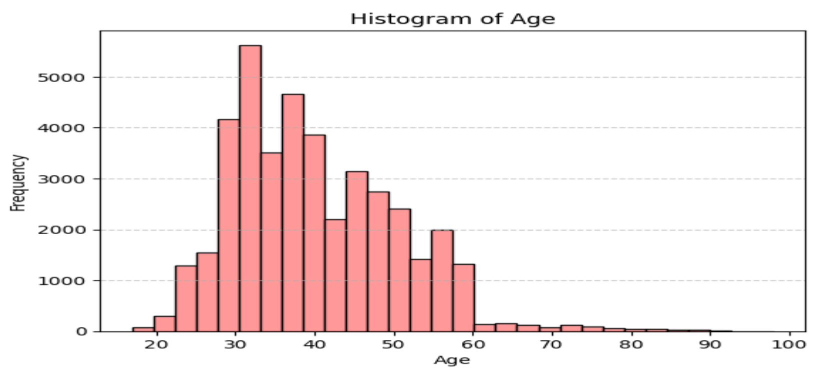


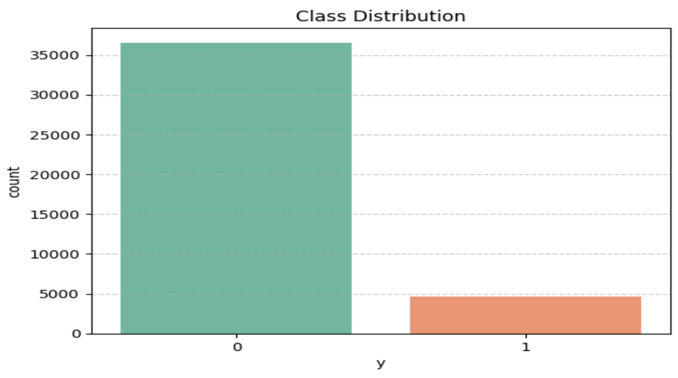


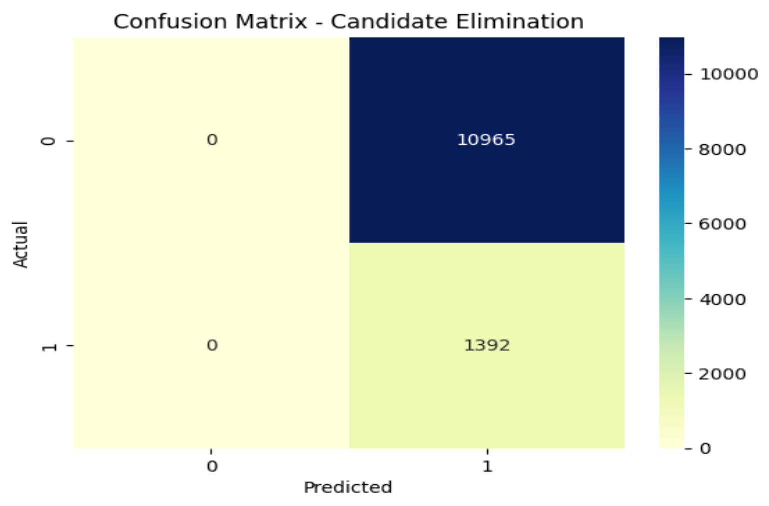
Candidate elimination:

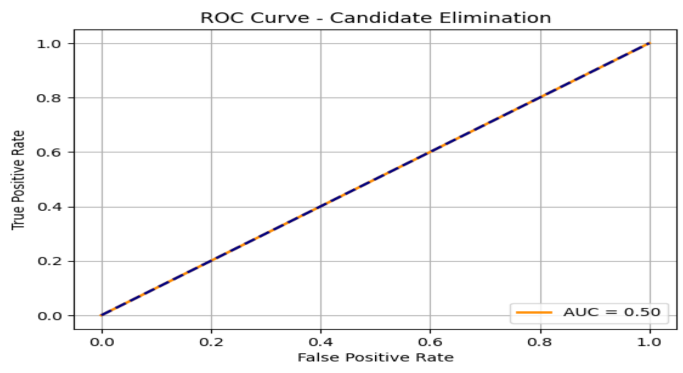
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, roc_curve, auc

df = pd.read_csv('/content/bank-additional-full.csv', sep=';')

df['y'] = df['y'].map({'no': 0, 'yes': 1})

df = df.dropna()

features = ['age', 'job', 'marital', 'education', 'default', 'housing', 'loan', 'duration']
X = df[features]
y = df['y']

from sklearn.preprocessing import LabelEncoder
cat_cols = ['job', 'marital', 'education', 'default', 'housing', 'loan']
for col in cat_cols:
    le = LabelEncoder()
    X[col] = le.fit_transform(X[col])

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, stratify=y, random_state=42)

# Candidate Elimination simplified (Find-S style)
def candidate_elimination(X_train, y_train):
    ce_features = ['job', 'marital', 'education', 'default', 'housing', 'loan']
    pos_data = X_train[y_train == 1][ce_features].reset_index(drop=True)
    hypothesis = pos_data.loc[0].copy()
    for i in range(1, len(pos_data)):
        for feature in ce_features:
            if hypothesis[feature] != pos_data.loc[i, feature]:
                hypothesis[feature] = '?'
    return hypothesis

def predict_ce(hypothesis, X_test):
    ce_features = hypothesis.index.tolist()
    y_pred = []
    for idx, row in X_test.iterrows():
        match = True
        for feature in ce_features:
            if hypothesis[feature] != '?' and hypothesis[feature] != row[feature]:
                match = False
                break
        y_pred.append(1 if match else 0)
    return np.array(y_pred)

hypothesis = candidate_elimination(X_train, y_train)
y_pred = predict_ce(hypothesis, X_test)
y_prob = y_pred  # For ROC approx

# Visualization functions
def plot_histogram():
    plt.hist(df['age'], bins=30, color='#FF9999', edgecolor='black')
    plt.title('Histogram of Age')
    plt.xlabel('Age')
    plt.ylabel('Frequency')
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.show()

def plot_class_dist():
    sns.countplot(x='y', data=df, palette='Set2')
    plt.title('Class Distribution')
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.show()

def plot_conf_matrix(y_true, y_pred, algo):
    cm = confusion_matrix(y_true, y_pred)
    sns.heatmap(cm, annot=True, fmt='d', cmap='YlGnBu')
    plt.title(f'Confusion Matrix - {algo}')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()

def plot_roc(y_true, y_prob, algo):
    fpr, tpr, _ = roc_curve(y_true, y_prob)
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'AUC = {roc_auc:.2f}')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.title(f'ROC Curve - {algo}')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.legend(loc='lower right')
    plt.grid(True)
    plt.show()

def plot_actual_vs_pred(y_true, y_pred, algo_name):
    n_samples = min(20000, len(y_true))
    plt.figure(figsize=(15, 4))
    plt.plot(range(n_samples), y_true.values[:n_samples], label='Actual', color='#66C2A5', alpha=0.7)
    plt.plot(range(n_samples), y_pred[:n_samples], label='Predicted', color='#FC8D62', alpha=0.7)
    plt.title(f'Actual vs Predicted (first 20,000 samples) - {algo_name}')
    plt.xlabel('Sample Index')
    plt.ylabel('Class Label')
    plt.legend()
    plt.grid(True)
    plt.show()

# Plots
plot_histogram()
plot_class_dist()
plot_conf_matrix(y_test, y_pred, "Candidate Elimination")
plot_roc(y_test, y_prob, "Candidate Elimination")
plot_actual_vs_pred(y_test, y_pred, "Candidate Elimination")
```

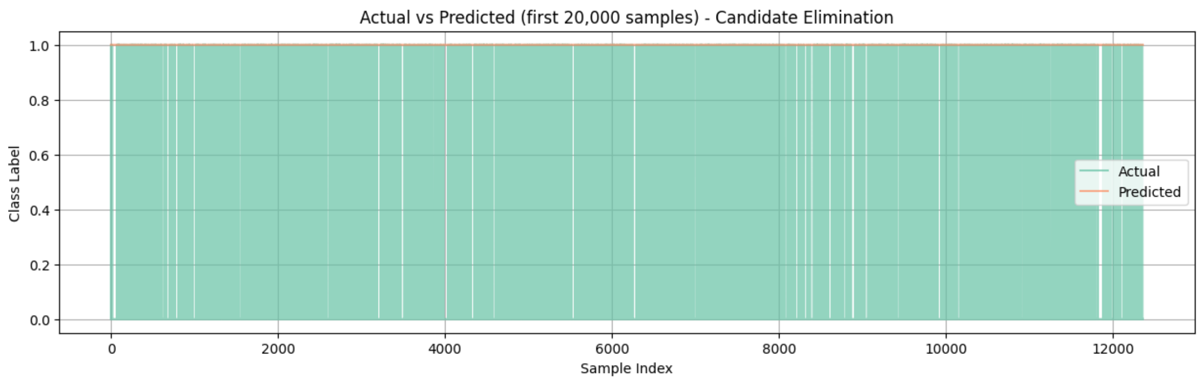
output:






Confusion Matrix - Candidate Elimination
Actual
0
1
0
10965
0
1392
Predicted
0
1
10000
8000
6000
4000
2000
0


ROC Curve - Candidate Elimination
True Positive Rate
False Positive Rate
AUC = 0.50
0.0
0.2
0.4
0.6
0.8
1.0


Actual vs Predicted (first 20,000 samples) - Candidate Elimination
Class Label
Sample Index
Actual
Predicted
0
2000
4000
6000
8000
10000
12000
0.0
0.2
0.4
0.6
0.8
1.0