Handling MISSING VALUES

Handling Missing Values

Replaced all “unknown” to NA `data.isna().sum()`

Imputation: The missing values are replaced with the mode of column

```
data.fillna(data.mode().iloc[0], inplace=True)
```

Why?

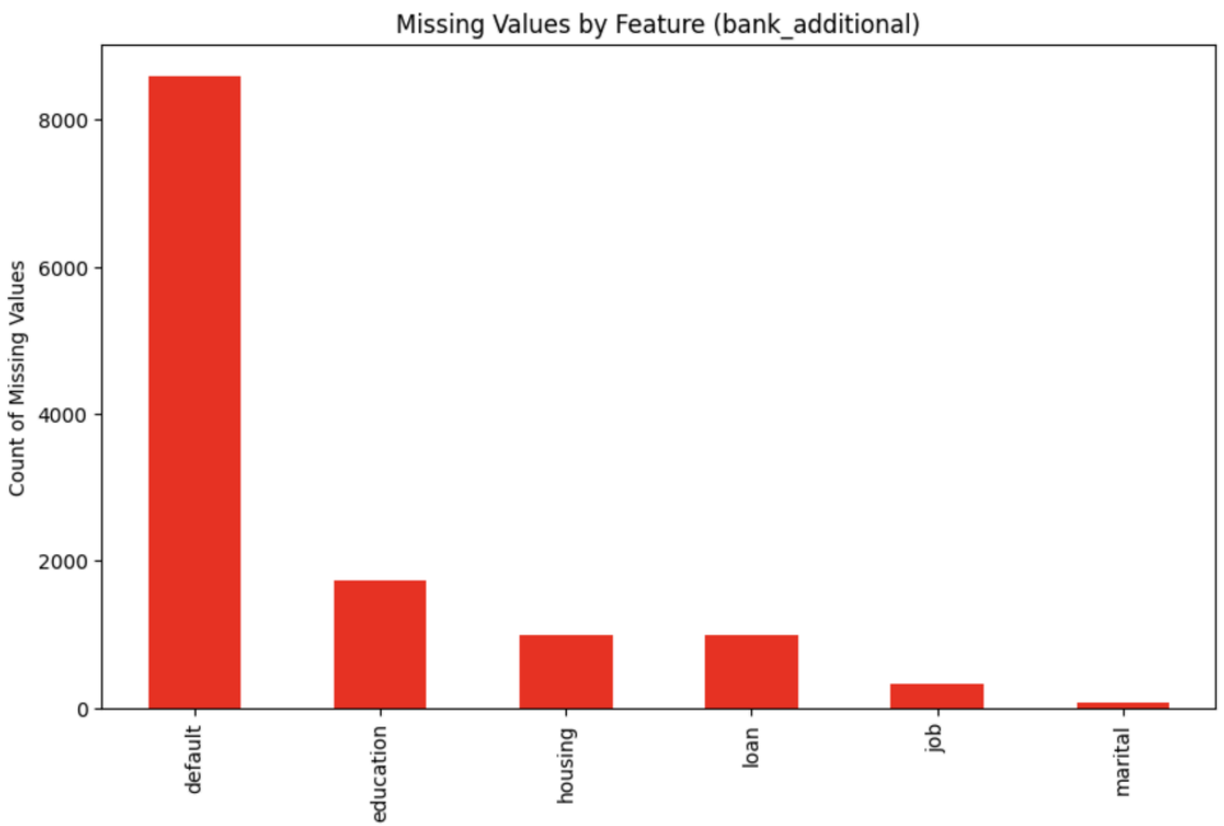
Machine Learning Algorithms Can't Handle Missing Data

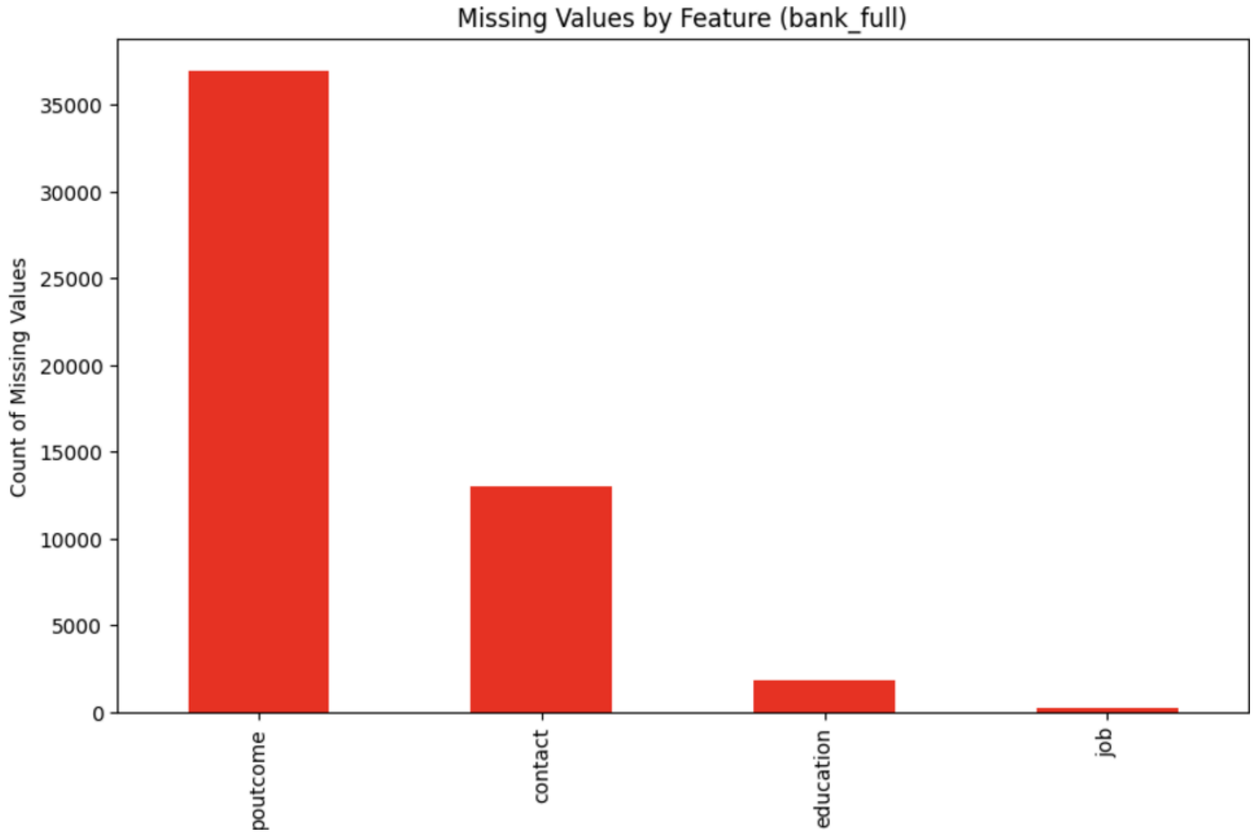
- In this case: KNN and Random Forest cant process missing values, leading to errors

Imputation Ensures Complete Data:

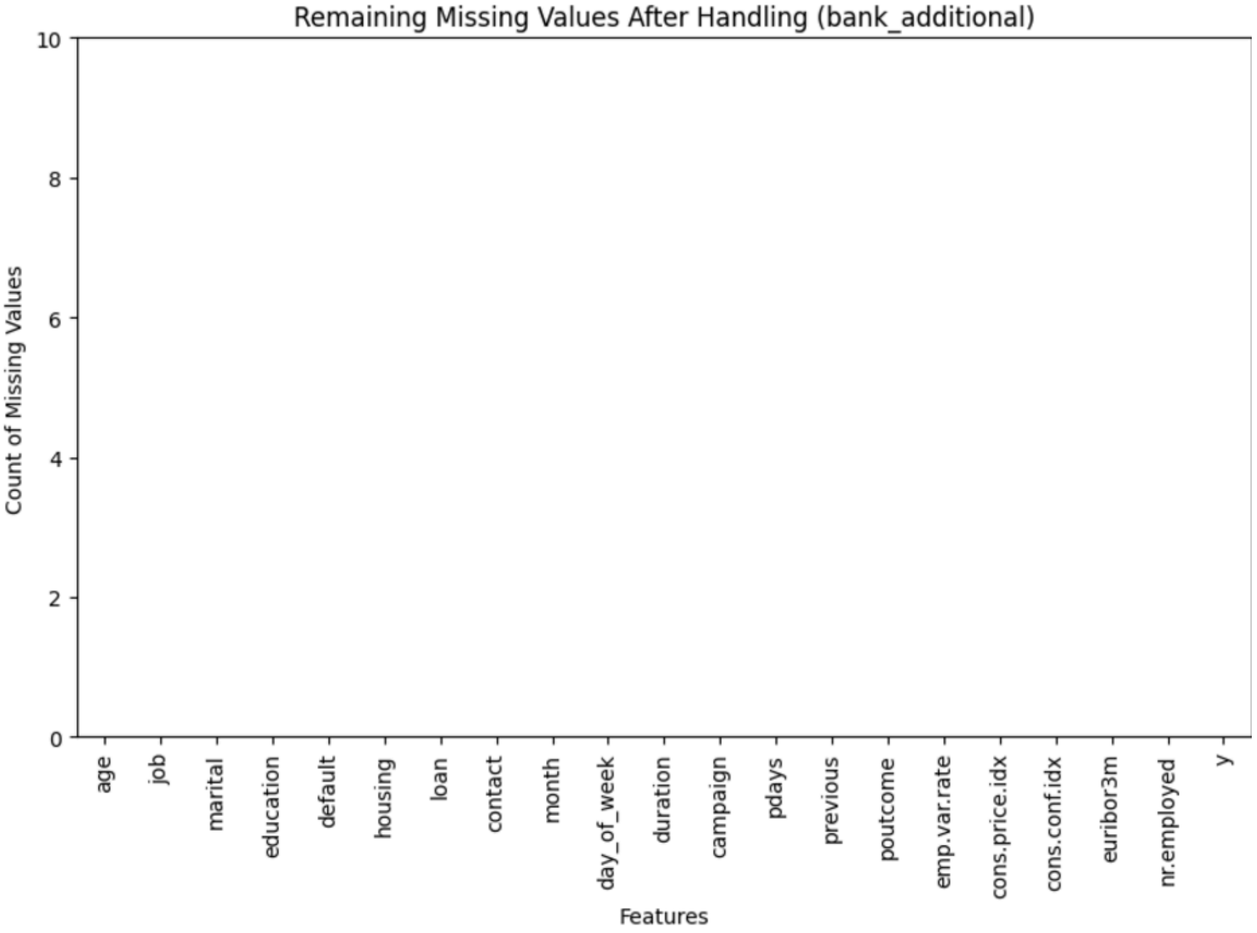
- Replaces missing values with substitutes (mean, median, mode

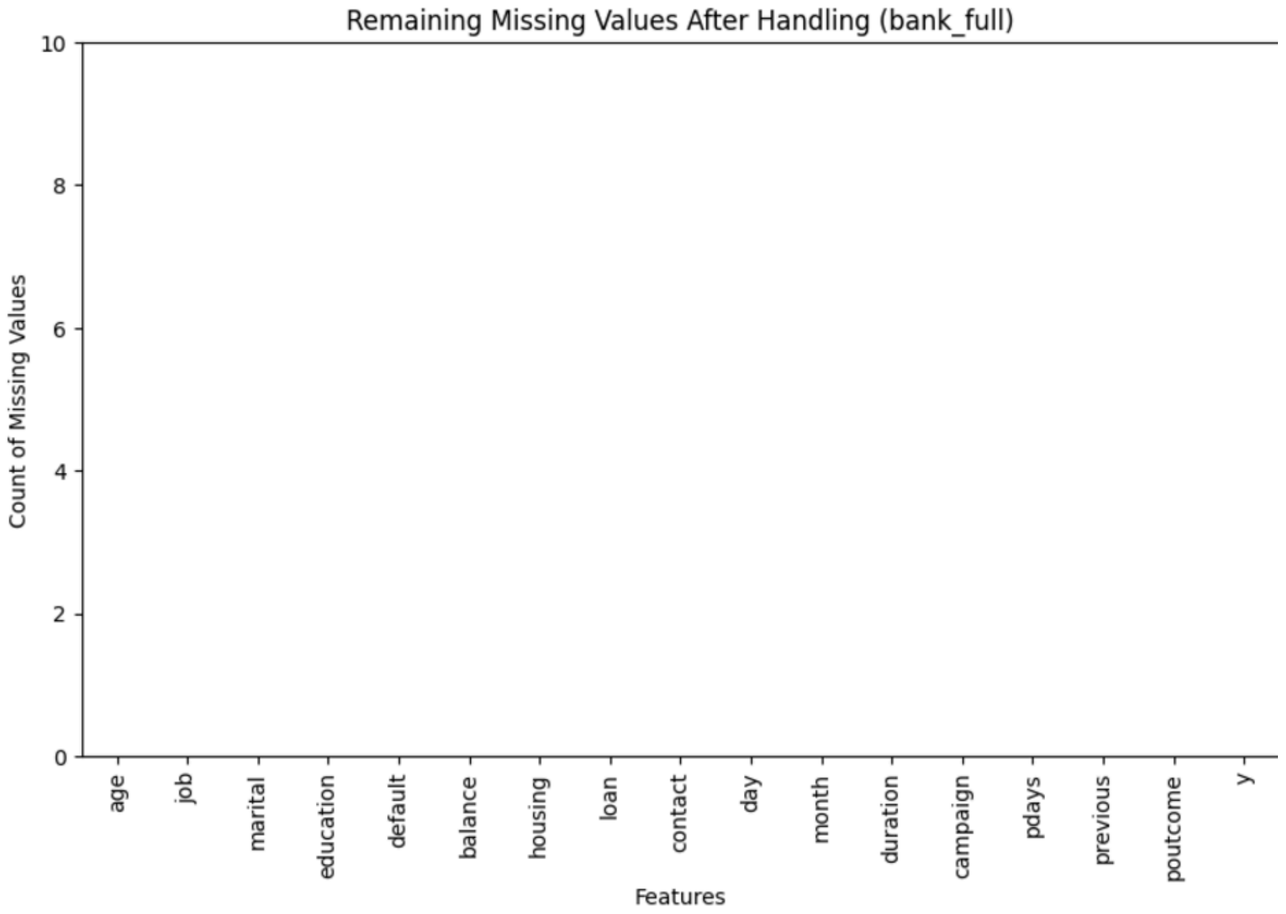
Visualization Before Imputation:





Visualization After Imputation:





ML Models Employed

K-Nearest Neighbour (KNN)

How it works

- Finding the nearest neighbors to a given data point and classifying it based on the majority class of those neighbors
- Customer’s features (age, job, balance, etc.) and compares them to other similar customers in the training data to predict whether they will subscribe to a term deposit or not

Pros

- Capture complex patterns in the data by using proximity between data points
- Capturing similarities between customers who might be similar in terms of their features and behaviors

Cons

- Dataset is large, k-NN is computationally expensive to make predictions
- Needs to compare every new test data point with all existing ones
- Long execution time

K-Nearest Neighbour (KNN)

Makes predictions based on the majority class of the closest data points to a given input.
It doesn't involve any learning during training but simply stores the dataset and uses the Euclidean distance to classify test data based on its proximity to the training data.
In the code, GridSearchCV is used to tune the hyperparameters (n_neighbors, weights, and p).
The scoring='roc_auc' parameter ensures that the best model is chosen based on the AUC-ROC score, and cross-validation (cv=5) is applied to evaluate the model's performance.

```
# Hyperparameter tuning for k-NN
knn_param_grid = {
    'n_neighbors': [3, 5, 7, 9],  # Number of neighbors
    'weights': ['uniform', 'distance'],  # Weight function
    'p': [1, 2]  # Distance metric: Manhattan (1) or Euclidean (2)
}
knn_grid_search = GridSearchCV(
    KNeighborsClassifier(),
    param_grid=knn_param_grid,
    scoring='roc_auc',
    cv=5,
    n_jobs=-1
)
knn_grid_search.fit(X_train, y_train)
```

Random Forest

How it works

- Random Forest builds multiple decision trees
- Each tree is trained on a random subset of the data
- Each tree makes a prediction, and the final prediction is based on the majority vote from all trees.
- Random Forest captures non-linear relationships between features like age, balance, and contact history

Pros

- Suitable for large datasets like Bank Marketing (40,000+ rows)
- Works well with both numerical and categorical data
- Suitable for predicting customer behavior like term deposit subscription

Cons

- Computationally expensive if many trees

Random Forest

Random Forest builds multiple decision trees and combines their outputs to make predictions. Each tree is built using a random subset of the training data which helps in reducing overfitting. In the code, GridSearchCV is used for hyperparameter tuning, where important parameters like n_estimators (number of trees), max_depth (tree depth), and class_weight (to handle imbalanced classes) are optimized. The scoring='roc_auc' parameter ensures that the model is evaluated using AUC-ROC to measure its performance.

```
rf_param_grid = {
    'n_estimators': [50, 100, 200],  # Number of trees
    'max_depth': [None, 10, 20, 30],  # Maximum depth of trees
    'min_samples_split': [2, 5, 10],  # Minimum samples to split a node
    'min_samples_leaf': [1, 2, 4],  # Minimum samples per leaf
    'class_weight': ['balanced']  # Handle class imbalance
}
rf_grid_search = GridSearchCV(
    RandomForestClassifier(random_state=42),
    param_grid=rf_param_grid,
    scoring='roc_auc',
    cv=5,
    n_jobs=-1
)
rf_grid_search.fit(X_train, y_train)
```

Performance Metrics

AUC-ROC

AUC-ROC is ideal for binary classification
(e.g., predicting customer subscription: yes/no)

AUC measures the model’s ability to differentiate between positive and negative classes

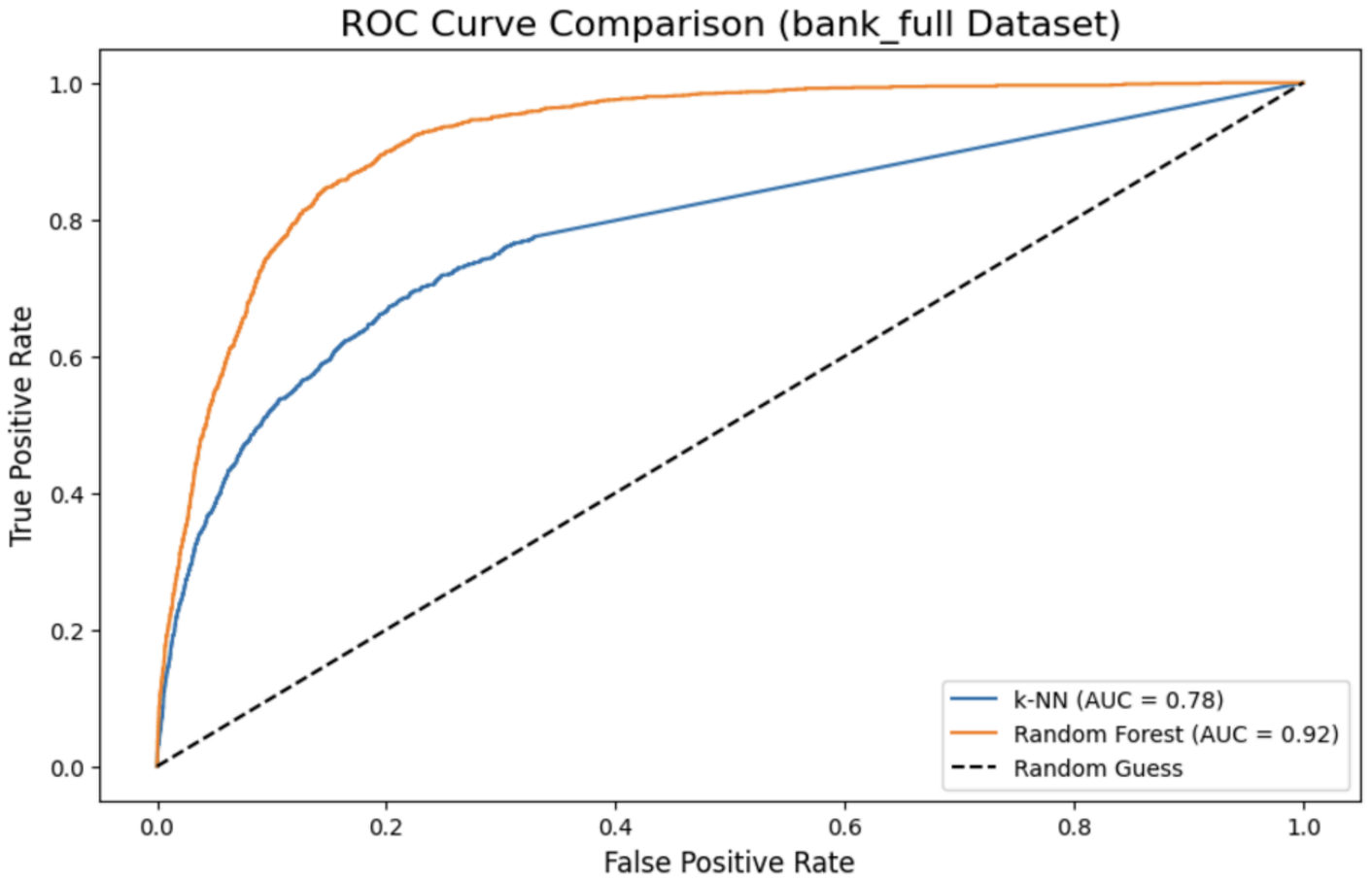
Class Imbalance: AUC-ROC is not affected by class imbalance, unlike accuracy (dataset has: more no’s than yes’)

Interpretability: The ROC curve plots True Positive Rate (Sensitivity/Recall) vs. False Positive Rate (1 - Specificity) across thresholds to visualize model performance

AUC-ROC Bank Dataset

Random Forest (orange curve):
AUC = 0.92, showing high classification performance and the ability to distinguish between classes effectively.

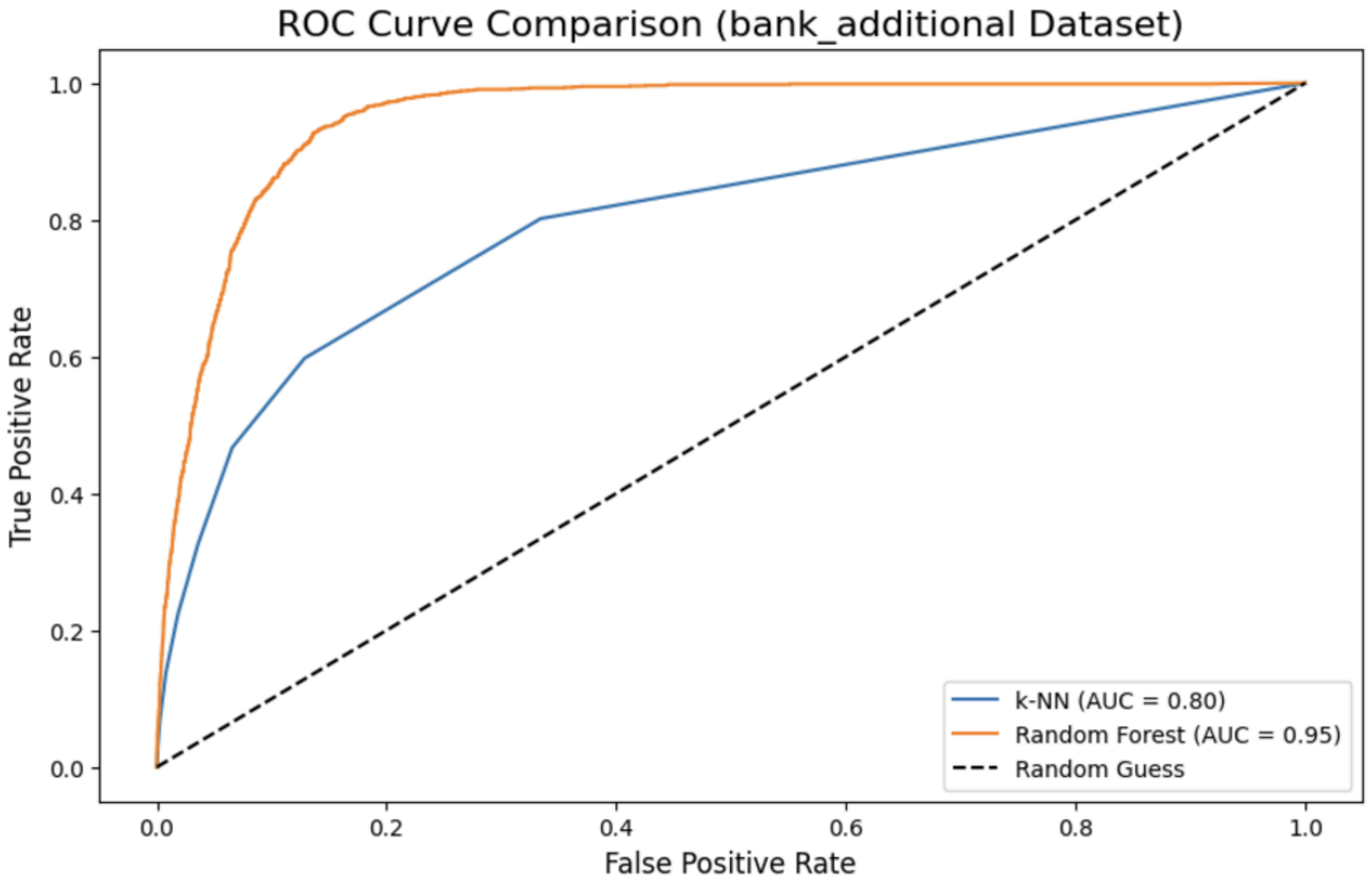
k-NN (blue curve):
AUC = 0.78, which indicates moderate performance but is significantly weaker compared to Random Forest.



AUC-ROC Bank Additional Dataset

Random Forest (orange curve): Outperforms k-NN with an AUC of 0.95, indicating excellent performance.

k-NN (blue curve): Has an AUC of 0.80, which shows good but relatively weaker performance compared to Random Forest.



Accuracy

- Measures the percentage of correct predictions

$$\mathrm{CA} = \frac{\mathrm{TP}+\mathrm{TN}}{\mathrm{TP}+\mathrm{TN}+\mathrm{FP}+\mathrm{FN}}$$

- But in this imbalanced dataset, predicting the majority class ("no subscription") can yield high accuracy but poor performance for the minority class (“yes")

Precision

- Measures how many of the predicted "yes" (subscribers) were actually correct

$$\mathrm{PR} = \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}}$$

- **Marketing:** Ensures we target customers who are likely to subscribe and not waste efforts on false positives

Recall

- Measures how many actual "yes" cases were correctly identified

$$\mathrm{RE} = \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}}$$

- **Marketing:** High recall ensures that most potential subscribers are captured and minimizing missed opportunities

F1-Score

- Mean of precision and recall to balance it (useful for imbalanced dataset)

$$\mathrm{F}_1 = \frac{2\mathrm{TP}}{2\mathrm{TP}+\mathrm{FP}+\mathrm{FN}}$$

- Balance between correctly identifying subscribers (recall) and targeting the right ones (precision) in marketing campaigns

Bank Additional Dataset

k-NN Model:
AUC-ROC: 0.78 — Reasonable

Recall for Subscribers (Class 1): 0.22 — Low, indicating many potential subscribers are missed

Accuracy: 0.90 — High accuracy but not reliable in imbalanced datasets like this one

Random Forest Model:
AUC-ROC : 0.95 — Excellent distinction between classes

Recall for Subscribers (Class 1): 0.68 — Better at identifying potential subscribers compared to k-NN

Accuracy: 0.90 — Similar to k-NN, but the model performs better in identifying minority class (Class 1)

```
k-NN Classification Report:
              precision    recall  f1-score   support

           0       0.91      0.98      0.94     10965
           1       0.61      0.22      0.33      1392

    accuracy                           0.90     12357
   macro avg       0.76      0.60      0.64     12357
weighted avg       0.87      0.90      0.87     12357

k-NN Test AUC-ROC: 0.80

Random Forest Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.91      0.94     10965
           1       0.54      0.84      0.65      1392

    accuracy                           0.90     12357
   macro avg       0.76      0.87      0.80     12357
weighted avg       0.93      0.90      0.91     12357

Random Forest Test AUC-ROC: 0.95
```

Bank Dataset

k-NN Model:
AUC-ROC: 0.78 — Reasonable

Recall for Subscribers: 0.23 — Low recall, meaning a significant number of actual subscribers are missed by the model

Accuracy: 0.89 — Accuracy high, the low recall for Class 1 (subscribers) suggests that the model might be biased towards predicting "no subscription"

Random Forest Model:
AUC-ROC: 0.92 — Strong performance, model good at distinguishing between the two classes

Recall for Subscribers: 0.55 — Much better than k-NN

Accuracy: 0.90 — High accuracy like k-NN, but Random Forest more reliable in identifying subscribers with a better recall

```
k-NN Classification Report:
              precision    recall  f1-score   support

           0       0.91      0.98      0.94     11977
           1       0.62      0.23      0.33      1587

    accuracy                           0.89     13564
   macro avg       0.76      0.60      0.64     13564
weighted avg       0.87      0.89      0.87     13564

k-NN Test AUC-ROC: 0.78

Random Forest Classification Report:
              precision    recall  f1-score   support

           0       0.94      0.95      0.94     11977
           1       0.59      0.55      0.57      1587

    accuracy                           0.90     13564
   macro avg       0.76      0.75      0.76     13564
weighted avg       0.90      0.90      0.90     13564

Random Forest Test AUC-ROC: 0.92
```

Improving Performance

Feature Engineering:

Create New Features

- Combining the existing info in dataset to create new features
- Eg: Customer’s age + account balance to create a new feature that represents age-to-balance ratio
- Helps the model see more patterns in the data

Hyperparameter Tuning:

Grid Search or Random Search

- Perform hyperparameter optimization using GridSearchCV or RandomizedSearchCV to find the best combination of parameters for models like k-NN and Random Forest.

Random Forest Tuning

- Increasing the number of trees or adjusting tree depth to reduce overfitting or underfitting

Exploratory Data Analysis (EDA)

Checking for Missing Values

Step 1: Visualizing Missing Values
Goal: To identify and visualize which features have missing values in the dataset.

1) Calculates the number of missing values per column using data.isna().sum().
2) Filters only the columns that have missing values (missing_values > 0).
3 If missing values exist, a bar chart is plotted with features on the x-axis and the count of missing values on the y-axis

```
def visualize_missing_values(data, dataset_name):

    # Calculate missing values per column
    missing_values = data.isna().sum()
    missing_values = missing_values[missing_values > 0]  # Filter columns with missing values only

    if not missing_values.empty:
        plt.figure(figsize=(10, 6))
        missing_values.sort_values(ascending=False).plot(kind='bar', color='red')
        plt.title(f"Missing Values by Feature ({dataset_name})")
        plt.xlabel("Features")
        plt.ylabel("Count of Missing Values")
        plt.show()
    else:
        print(f"No missing values in the {dataset_name} dataset.")
```

Checking for Missing Values

Step 2: Visualizing Missing Values

After imputing missing values, it’s important to visualize the results to ensure that no missing values remain. This is done with the function visualize_remaining_missing()

```
def visualize_remaining_missing(data, dataset_name):
    """
    Visualize remaining missing values after handling.
    Shows a bar chart with zero missing values if none remain.
    """
    missing_values = data.isna().sum()

    plt.figure(figsize=(10, 6))
    missing_values.plot(kind='bar', color='green')
    plt.title(f"Remaining Missing Values After Handling ({dataset_name})")
    plt.xlabel("Features")
    plt.ylabel("Count of Missing Values")
    plt.ylim(0, 10)
    plt.axhline(0, color="black", linewidth=0.8)
    plt.show()
```

Checking for Class Distribution

Assessing the distribution of the target variable, y, in both datasets to understand how balanced or imbalanced the data is. The target variable represents whether a customer subscribes to a term deposit, with values of 1 indicating a subscription ("Yes") and 0 indicating no subscription ("No").