Assignment 6

Author: Ojas Phadake

Roll No: CH22B007

Summary of Assignment:

This assignment makes one apply linear and non-linear regression to impute missing values in a dataset. The effectiveness of the imputation methods will be measured indirectly by assessing the performance of a subsequent classification task, comparing the regression-based approach against simpler imputation strategies.

Part A: Data Preprocessing and Imputation

Load and Prepare Data

To aritificially introduce Missing At Random (MAR) values to simulate a real-world scenario with a substantial missing data problem.

```
import kagglehub
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix, ConfusionMatrixDisplay, RocCurveDisplay
from sklearn.decomposition import PCA

from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import LinearRegression

from sklearn.impute import SimpleImputer
from sklearn.impute import KNNImputer
from sklearn.neighbors import KNeighborsRegressor

from sklearn.preprocessing import StandardScaler

from collections import Counter

import warnings
warnings.filterwarnings('ignore')
```

```
path = kagglehub.dataset_download("uciml/default-of-credit-card-clients-dataset")

print("Path to dataset files:", path)
df = pd.read_csv(f"{path}/UCI_Credit_Card.csv")
print(df.shape)
print(df.head())
```

```
Downloading from https://www.kaggle.com/api/v1/datasets/download/uciml/default-of-credit-card-clients-dataset?dataset_version_r
100%|██████████| 0.98M/0.98M [00:00<00:00, 1.39MB/s]Extracting files...

Path to dataset files: /root/.cache/kagglehub/datasets/uciml/default-of-credit-card-clients-dataset/versions/1
(30000, 25)
   ID  LIMIT_BAL  SEX  EDUCATION  MARRIAGE  AGE  PAY_0  PAY_2  PAY_3  PAY_4  \
0   1    20000.0    2          2         1   24      2      2     -1     -1
1   2   120000.0    2          2         2   26     -1      2      0      0
2   3    90000.0    2          2         2   34      0      0      0      0
3   4    50000.0    2          2         1   37      0      0      0      0
4   5    50000.0    1          2         1   57     -1      0     -1      0

   ...  BILL_AMT4  BILL_AMT5  BILL_AMT6  PAY_AMT1  PAY_AMT2  PAY_AMT3  \
0  ...        0.0        0.0        0.0       0.0     689.0       0.0
1  ...     3272.0     3455.0     3261.0       0.0    1000.0    1000.0
2  ...    14331.0    14948.0    15549.0    1518.0    1500.0    1000.0
3  ...    28314.0    28959.0    29547.0    2000.0    2019.0    1200.0
4  ...    20940.0    19146.0    19131.0    2000.0   36681.0   10000.0

   PAY_AMT4  PAY_AMT5  PAY_AMT6  default.payment.next.month
0       0.0       0.0       0.0                          1
1    1000.0       0.0    2000.0                          1
2    1000.0    1000.0    5000.0                          0
3    1100.0    1069.0    1000.0                          0
4    9000.0     689.0     679.0                          0
```

```
[5 rows x 25 columns]
```

```
df.columns
```

```
Index(['ID', 'LIMIT_BAL', 'SEX', 'EDUCATION', 'MARRIAGE', 'AGE', 'PAY_0',
       'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6', 'BILL_AMT1', 'BILL_AMT2',
       'BILL_AMT3', 'BILL_AMT4', 'BILL_AMT5', 'BILL_AMT6', 'PAY_AMT1',
       'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5', 'PAY_AMT6',
       'default.payment.next.month'],
      dtype='object')
```

```
np.random.seed(42)

# columns to have missing values
cols_to_nan = ['AGE', 'BILL_AMT1', 'BILL_AMT2', 'BILL_AMT3']

# missing values (between 5-10%)
missing_fraction = np.random.uniform(0.05, 0.10)

for col in cols_to_nan:
    n_missing = int(missing_fraction * len(df))
    missing_indices = np.random.choice(df.index, n_missing, replace=False)
    df.loc[missing_indices, col] = np.nan

# Verify missingness
print(df[cols_to_nan].isnull().sum())
```

```
AGE          2061
BILL_AMT1    2061
BILL_AMT2    2061
BILL_AMT3    2061
dtype: int64
```

Now we have artificially introduced MAR missing values (roughly 6.8% of the number of values) in the 4 columns given by `AGE`, `BILL_AMT`, `BILL_AMT2` and `BILL_AMT3`. Now we proceed to carry out the imputation strategies.

Imputation Strategy 1: Simple Imputation

```
df.columns[df.isnull().any()]
```

```
Index(['AGE', 'BILL_AMT1', 'BILL_AMT2', 'BILL_AMT3'], dtype='object')
```

```
df_A = df.copy()

cols_with_missing = df_A.columns[df_A.isnull().any()]

# Perform median imputation for each column with missing values
for col in cols_with_missing:
    median_value = df_A[col].median()
    df_A[col].fillna(median_value, inplace=True)

# Verify imputation
print("Missing values after median imputation:")
print(df_A[cols_with_missing].isnull().sum())
```

```
Missing values after median imputation:
AGE          0
BILL_AMT1    0
BILL_AMT2    0
BILL_AMT3    0
dtype: int64
```

Explanation: Why Median Is Often Preferred Over Mean for Imputation

The **median** is often preferred over the **mean** when imputing missing numerical values because:

1. **Robustness to Outliers:** The median is not affected by extreme values. In datasets like the credit card default data, financial features (e.g., `AGE`, `BILL_AMT1`, ...) can have very large outliers due to a few clients with exceptionally high bills or payments. → Using the mean would **shift the imputed value** toward these extremes, distorting the data distribution. → The median, on the other hand, provides a **more stable central value**.
2. **Preservation of Data Distribution:** Median imputation tends to preserve the ori
than mean imputation, which can artificially reduce variance.

3. **More Realistic for Non-Normal Data:** Many financial and demographic variables (like `AGE`, `BILL_AMTx`) are **not normally distributed**. In such cases, the median is a **better measure of central tendency** than the mean.

Imputation Strategy 2: Regression Imputation (Linear)

We wil be selecting the column `AGE` to perform LinearRegression on.

```
df_B = df.copy()
target_col = 'AGE'

# Separate rows with and without missing AGE
df_train = df_B[df_B[target_col].notnull()]
df_missing = df_B[df_B[target_col].isnull()]

# Select numerical predictor columns (excluding the target and dependent variable)
predictor_cols = df_B.select_dtypes(include=[np.number]).columns.drop(
    [target_col, 'default.payment.next.month']
)

# Prepare training and prediction sets
X_train = df_train[predictor_cols]
y_train = df_train[target_col]
X_missing = df_missing[predictor_cols]

# Handle any missing values in predictors by simple imputation (median)
X_train = X_train.fillna(X_train.median())
X_missing = X_missing.fillna(X_train.median())

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_missing_scaled = scaler.transform(X_missing)

# Train Linear Regression model
lin_reg = LinearRegression()
lin_reg.fit(X_train_scaled, y_train)

predicted_age = lin_reg.predict(X_missing_scaled)

# Fill in the predicted values
df_B.loc[df_B[target_col].isnull(), target_col] = predicted_age

# Verify imputation
print(f"Missing values in '{target_col}' after regression imputation:",
      df_B[target_col].isnull().sum())
```

```
Missing values in 'AGE' after regression imputation: 0
```

Explanation behind Linear Regression Imputation

Underlying Assumption: Missing At Random (MAR)

The **Regression Imputation** method assumes that the missing data mechanism is **Missing At Random (MAR)**.

What MAR means:

> A variable is *Missing At Random* if the probability that a value is missing **depends only on other observed variables** in the dataset, **not on the missing value itself**.

Formally,

$$P(\text{Missing in X} \mid X, Y) = P(\text{Missing in X} \mid Y)$$

where (Y) are the observed variables.

That is, after controlling for (Y), missingness in (X) is independent of the true (unobserved) values of (X).

Example in this dataset

The column **`AGE`** has some missing values.

- Under the **MAR assumption**, the fact that a person's **AGE** is missing could be related to other observed information — for example:
  - Their **education level**, **bill amount**, or **payment history**.
- But it is **not directly related to their actual age value** (the missing value itself).

So, if two clients have the same observed data (same `LIMIT_BAL`, `BILL_AMT1`, etc.) their `AGE` missing — regardless of what their true age is.

Why Regression Imputation Relies on MAR

Regression imputation uses other **observed variables** to predict the missing ones. This approach only works well if those observed variables truly capture the pattern behind the missingness.

If the missingness instead depends on the **unobserved value itself** — i.e., data are **Missing Not At Random (MNAR)** — then regression imputation will produce **biased estimates** because the model cannot account for the hidden dependency.

Summary

Missingness Type	Description	Can Regression Imputation Handle It?
MCAR (Missing Completely At Random)	Missingness is random and unrelated to any variable	✅ Works fine
MAR (Missing At Random)	Missingness depends on observed data	✅ Works well (assumed case)
MNAR (Missing Not At Random)	Missingness depends on the missing value itself	❌ Leads to bias

```
# Create a clean dataset copy
df_C = df.copy()

# Column to impute
target_col = 'AGE'
target_variable = 'default.payment.next.month'

# Separate rows with and without missing AGE
df_train = df_C[df_C[target_col].notnull()]
df_missing = df_C[df_C[target_col].isnull()]

# Select predictor columns
predictor_cols = df_C.select_dtypes(include=[np.number]).columns.drop([target_col, target_variable])

# Prepare train/test sets
X_train = df_train[predictor_cols]
y_train = df_train[target_col]
X_missing = df_missing[predictor_cols]

# Handle missing predictor values with median
X_train = X_train.fillna(X_train.median())
X_missing = X_missing.fillna(X_train.median())

# Standardize predictors
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_missing_scaled = scaler.transform(X_missing)

# Train KNN Regressor
knn_reg = KNeighborsRegressor(n_neighbors=5, weights='distance')
knn_reg.fit(X_train_scaled, y_train)

# Predict missing AGE values
predicted_age_knn = knn_reg.predict(X_missing_scaled)

# Fill missing AGE values in dataset
df_C.loc[df_C[target_col].isnull(), target_col] = predicted_age_knn

# Verify completion
print(f"Missing values in '{target_col}' after KNN regression imputation:",
      df_C[target_col].isnull().sum())
```

```
Missing values in 'AGE' after KNN regression imputation: 0
```

Part B: Model Training and Performance Assessment

Data Split

Let us now split the training sets of the models A,B,C and D into training and testing sets.

```
target_variable = 'default.payment.next.month'

# -------------------------------
# Dataset A: Median Imputation
# -------------------------------
X_A = df_A.drop(columns=[target_variable])
y_A = df_A[target_variable]

X_train_A, X_test_A, y_train_A, y_test_A = train_test_split(
    X_A, y_A, test_size=0.2, random_state=42, stratify=y_A
)
```

```
# ------------------------------
# Dataset B: Linear Regression Imputation
# ------------------------------
X_B = df_B.drop(columns=[target_variable])
y_B = df_B[target_variable]

X_train_B, X_test_B, y_train_B, y_test_B = train_test_split(
    X_B, y_B, test_size=0.2, random_state=42, stratify=y_B
)

# ------------------------------
# Dataset C: Non-linear Regression (KNN) Imputation
# ------------------------------
X_C = df_C.drop(columns=[target_variable])
y_C = df_C[target_variable]

X_train_C, X_test_C, y_train_C, y_test_C = train_test_split(
    X_C, y_C, test_size=0.2, random_state=42, stratify=y_C
)

# ------------------------------
# Dataset D: Listwise Deletion (remove any rows with NaN)
# ------------------------------
df_D = df.dropna().copy()

X_D = df_D.drop(columns=[target_variable])
y_D = df_D[target_variable]

X_train_D, X_test_D, y_train_D, y_test_D = train_test_split(
    X_D, y_D, test_size=0.2, random_state=42, stratify=y_D
)

# ------------------------------
# Verify shapes of splits
# ------------------------------
print("Dataset A (Median Imputation):", X_train_A.shape, X_test_A.shape)
print("Dataset B (Linear Imputation):", X_train_B.shape, X_test_B.shape)
print("Dataset C (Non-linear Imputation):", X_train_C.shape, X_test_C.shape)
print("Dataset D (Listwise Deletion):", X_train_D.shape, X_test_D.shape)
```

```
Dataset A (Median Imputation): (24000, 24) (6000, 24)
Dataset B (Linear Imputation): (24000, 24) (6000, 24)
Dataset C (Non-linear Imputation): (24000, 24) (6000, 24)
Dataset D (Listwise Deletion): (18047, 24) (4512, 24)
```

Classifier Setup

Standardize the dataset using StandardScaler() from sklearn library.

```
# Create separate scalers for each dataset
scaler_A = StandardScaler()
scaler_B = StandardScaler()
scaler_C = StandardScaler()
scaler_D = StandardScaler()

# ------------------------------
# Dataset A
# ------------------------------
X_train_A_scaled = scaler_A.fit_transform(X_train_A)
X_test_A_scaled = scaler_A.transform(X_test_A)

# ------------------------------
# Dataset B
# ------------------------------
X_train_B_scaled = scaler_B.fit_transform(X_train_B)
X_test_B_scaled = scaler_B.transform(X_test_B)

# ------------------------------
# Dataset C
# ------------------------------
X_train_C_scaled = scaler_C.fit_transform(X_train_C)
X_test_C_scaled = scaler_C.transform(X_test_C)

# ------------------------------
# Dataset D
# ------------------------------
X_train_D_scaled = scaler_D.fit_transform(X_train_D)
X_test_D_scaled = scaler_D.transform(X_test_D)
```

```
# --------------------------------
# Verification
# --------------------------------
print("Feature standardization complete for all four datasets.")
print("Shapes (train/test):")
print("A:", X_train_A_scaled.shape, X_test_A_scaled.shape)
print("B:", X_train_B_scaled.shape, X_test_B_scaled.shape)
print("C:", X_train_C_scaled.shape, X_test_C_scaled.shape)
print("D:", X_train_D_scaled.shape, X_test_D_scaled.shape)
```

```
Feature standardization complete for all four datasets.
Shapes (train/test):
A: (24000, 24) (6000, 24)
B: (24000, 24) (6000, 24)
C: (24000, 24) (6000, 24)
D: (18047, 24) (4512, 24)
```

```
def preprocess_features(X_train, X_test):
    # Fill any remaining NaNs using median
    imputer = SimpleImputer(strategy='median')
    X_train_imputed = imputer.fit_transform(X_train)
    X_test_imputed = imputer.transform(X_test)

    # Standardize features
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train_imputed)
    X_test_scaled = scaler.transform(X_test_imputed)

    return X_train_scaled, X_test_scaled

# Preprocess all datasets
X_train_A_scaled, X_test_A_scaled = preprocess_features(X_train_A, X_test_A)
X_train_B_scaled, X_test_B_scaled = preprocess_features(X_train_B, X_test_B)
X_train_C_scaled, X_test_C_scaled = preprocess_features(X_train_C, X_test_C)
X_train_D_scaled, X_test_D_scaled = preprocess_features(X_train_D, X_test_D)
```

Model Evaluation

Train `LogisticRegression` on each of the training dataset and use it to predict on the corresponding test dataset.

```
# Define a function to train and evaluate Logistic Regression
def train_and_evaluate(X_train, X_test, y_train, y_test, label):
    print(f"\n===== Logistic Regression Results for {label} =====")
    model = LogisticRegression(max_iter=1000, random_state=42)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    print(classification_report(y_test, y_pred))

# Train and evaluate on all datasets
train_and_evaluate(X_train_A_scaled, X_test_A_scaled, y_train_A, y_test_A, "Dataset A (Mean Imputation)")
train_and_evaluate(X_train_B_scaled, X_test_B_scaled, y_train_B, y_test_B, "Dataset B (Linear Regression Imputation)")
train_and_evaluate(X_train_C_scaled, X_test_C_scaled, y_train_C, y_test_C, "Dataset C (Non-linear Regression Imputation)")
train_and_evaluate(X_train_D_scaled, X_test_D_scaled, y_train_D, y_test_D, "Dataset D (Listwise Deletion)")
```

```
===== Logistic Regression Results for Dataset A (Mean Imputation) =====
              precision    recall  f1-score   support

           0       0.82      0.97      0.89      4673
           1       0.69      0.24      0.35      1327

    accuracy                           0.81      6000
   macro avg       0.75      0.60      0.62      6000
weighted avg       0.79      0.81      0.77      6000


===== Logistic Regression Results for Dataset B (Linear Regression Imputation) =====
              precision    recall  f1-score   support

           0       0.82      0.97      0.89      4673
           1       0.69      0.24      0.35      1327

    accuracy                           0.81      6000
   macro avg       0.75      0.60      0.62      6000
weighted avg       0.79      0.81      0.77      6000


===== Logistic Regression Results for Dataset C (Non-linear Regression Imputa
              precision    recall  f1-score   support
```

```
              precision    recall  f1-score   support

           0       0.82      0.97      0.89      4673
           1       0.68      0.24      0.35      1327

    accuracy                           0.81      6000
   macro avg       0.75      0.60      0.62      6000
weighted avg       0.79      0.81      0.77      6000


===== Logistic Regression Results for Dataset D (Listwise Deletion) =====
              precision    recall  f1-score   support

           0       0.82      0.97      0.89      3511
           1       0.73      0.27      0.39      1001

    accuracy                           0.82      4512
   macro avg       0.78      0.62      0.64      4512
weighted avg       0.80      0.82      0.78      4512
```

Part C: Comparative Analysis

Results Comparison

Given below is a comparitive analysis between the results and various metrics for all the 4 methods, followed by a comprehensive discussion about the efficacy discussion.

```
# Function to extract metrics into a dictionary
def get_metrics(y_true, y_pred):
    report = classification_report(y_true, y_pred, output_dict=True)
    # We take the 'weighted avg' row to summarize performance across both classes
    metrics = {
        'Accuracy': report['accuracy'],
        'Precision': report['weighted avg']['precision'],
        'Recall': report['weighted avg']['recall'],
        'F1-score': report['weighted avg']['f1-score']
    }
    return metrics

# Train models and collect metrics
def train_and_collect_metrics(X_train, X_test, y_train, y_test):
    model = LogisticRegression(max_iter=1000, random_state=42)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    return get_metrics(y_test, y_pred)

# Collect metrics for all datasets
metrics_A = train_and_collect_metrics(X_train_A_scaled, X_test_A_scaled, y_train_A, y_test_A)
metrics_B = train_and_collect_metrics(X_train_B_scaled, X_test_B_scaled, y_train_B, y_test_B)
metrics_C = train_and_collect_metrics(X_train_C_scaled, X_test_C_scaled, y_train_C, y_test_C)
metrics_D = train_and_collect_metrics(X_train_D_scaled, X_test_D_scaled, y_train_D, y_test_D)

# Create summary DataFrame
import pandas as pd

summary_df = pd.DataFrame([metrics_A, metrics_B, metrics_C, metrics_D],
                          index=['Model A (Median Imputation)',
                                 'Model B (Linear Regression Imputation)',
                                 'Model C (Non-linear Regression Imputation)',
                                 'Model D (Listwise Deletion)'])

# Display summary table
print("===== Summary of Model Performance =====")
print(summary_df)
```

```
===== Summary of Model Performance =====
                                    Accuracy  Precision    Recall  \
Model A (Median Imputation)         0.807667   0.789085  0.807667
Model B (Linear Regression Imputation)  0.807333   0.788280  0.807333
Model C (Non-linear Regression Imputation)  0.807333   0.788192  0.807333
Model D (Listwise Deletion)         0.815824   0.803023  0.815824

                                    F1-score
Model A (Median Imputation)         0.768959
Model B (Linear Regression Imputation)  0.768845
Model C (Non-linear Regression Imputation)  0.768987
Model D (Listwise Deletion)         0.780685
```

Efficacy Discussion

1. Trade-off Between Listwise Deletion (Model D) and Imputation (Models A, B, C)

Listwise Deletion (Model D):

- **Removes** all rows containing missing values, ensuring the dataset used for training is completely clean.
- **Pros:**
  - Simplifies model training since no missing data remain.
  - Avoids potential bias introduced by incorrect imputation if the data were **Missing Completely At Random (MCAR)**.
- **Cons:**
  - **Reduces dataset size**, leading to a loss of potentially valuable information.
  - Decreases **statistical power** and can result in biased estimates if missingness is related to other observed variables (MAR).

Imputation Methods (Models A, B, C):

- Replace missing values with estimated ones, preserving the number of samples.
- **Pros:**
  - Retains all data points, helping the model learn from the full dataset.
  - Regression-based imputations (Models B and C) use relationships between features to produce more realistic estimates.
- **Cons:**
  - May introduce small inaccuracies or bias if the imputation assumptions are violated.
  - Simple median imputation (Model A) ignores relationships between features, leading to less precise replacements.

Observed Results: Although **Model D** (Listwise Deletion) achieved the **highest F1-score (0.7807)** and **accuracy (0.8158)**, this result does not necessarily mean deletion is always superior. The performance gain might reflect that the deleted subset contained noisier or inconsistent samples. However, in real-world cases, listwise deletion can **drastically reduce dataset size**, which typically harms model generalization — especially when missingness is widespread.

In contrast, **Models A, B, and C** (with F1-scores around **0.769**) preserved all samples, providing a more robust and complete training base, even if their short-term performance is slightly lower.

---

2. Linear vs. Non-Linear Regression Imputation (Models B vs. C)

Observation:

- Model B (Linear Regression Imputation): F1-score = **0.7688**
- Model C (Non-linear Regression Imputation): F1-score = **0.7690**

Interpretation:

- The difference between the two is **minimal**, but the **non-linear regression model (C)** performed slightly better.
- This small improvement indicates that the relationship between the imputed feature (e.g., `AGE`) and predictors (`BILL_AMT1-6`, `LIMIT_BAL`) is **not strictly linear**.
- Non-linear models (like KNN or Decision Tree regression) capture **non-linear dependencies and local variations** better than linear models, which assume a straight-line relationship between predictors and the target.

Conceptual Link: This aligns with the **Missing At Random (MAR)** assumption — the missing values depend on other observed features (like billing amounts), and modeling those relationships flexibly helps produce more accurate imputations.

---

3. Recommendation and Conclusion

Strategy	Pros	Cons	F1-score (Observed)
Median Imputation (Model A)	Simple, fast, robust to outliers	Ignores feature relationships	0.7689
Linear Regression Imputation (Model B)	Utilizes linear correlations, preserves variance	Cannot capture non-linear dependencies	0.7688
Non-linear Regression Imputation (Model C)	Captures complex interactions, realistic imputations	Slightly more computationally intensive	0.7690
Listwise Deletion (Model D)	Simplifies model, removes noise	Data loss, possible bias	0.7807

Final Recommendation: While **Listwise Deletion (Model D)** achieved the best numerical performance here, it is **not a generally reliable strategy** — its success likely stems from reduced noise after dropping incomplete rows rather than a fundamentally better approach. For most real-world scenarios with significant missingness, **Non-linear Regression Imputation (Model C)** is the **most balanced and conceptually sound** method:

- It **preserves all data** and avoids information loss.
- It **models complex dependencies** between variables, improving imputation qua
- It aligns with the **MAR assumption**, ensuring that missing values are replaced bas

Thus, the recommended approach for handling missing data in this context is **non-linear regression imputation**, as it provides a strong balance between completeness, theoretical robustness, and predictive accuracy.

```
Start coding or generate with AI.
```