ML/DL Assignment

MIMIC-IV : Mortality Prediction for Acute Respiratory Failure Patients

Ankita Savaliya

MIMIC-IV : Mortality Prediction for Acute Respiratory Failure Patients

Goal:

- Load and preprocess MIMIC-IV EHR data related to Acute Respiratory Failure (ARF).
- Analyze demographic and lab test indicators for ARF.
- Construct a structured dataset for modeling.
- Apply classification and deep learning models to EHR data.
- Compare model performances.
- Predict mortality in ARF patients.

Dataset: MIMIC-IV dataset.

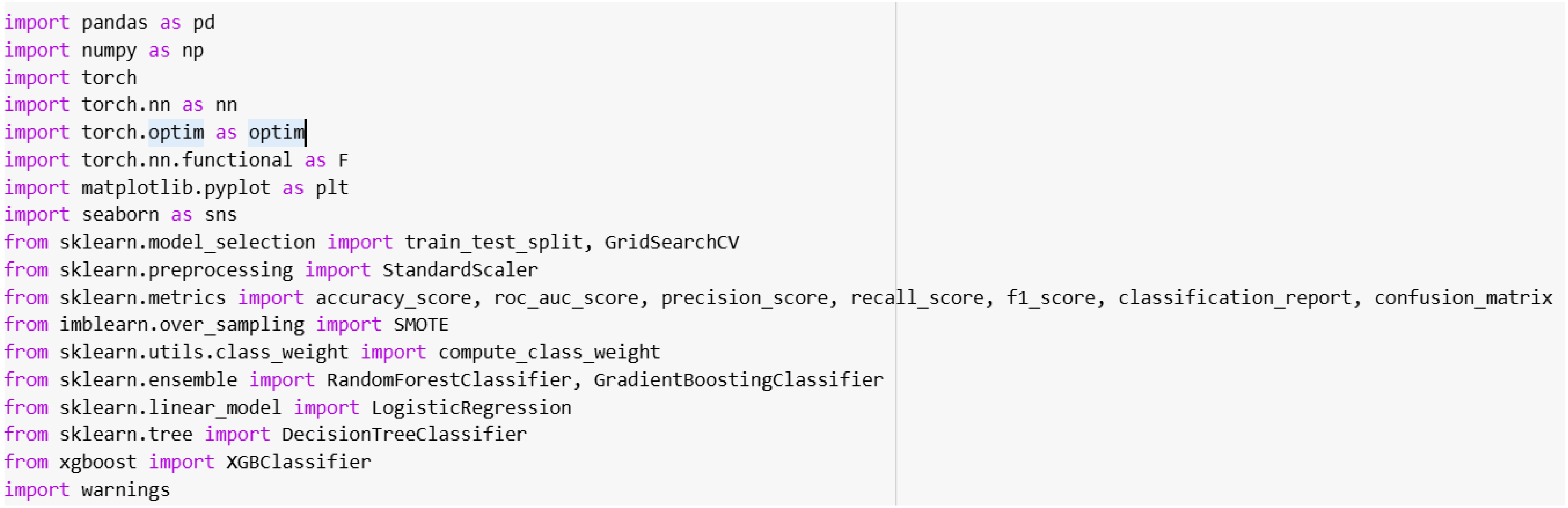
GitHub and Google Colab Links:

https://colab.research.google.com/github/AnkitaSavaliya/AIH/blob/main/MIMIC-IV_MORTALITY_PREDICTION_ARF.ipynb

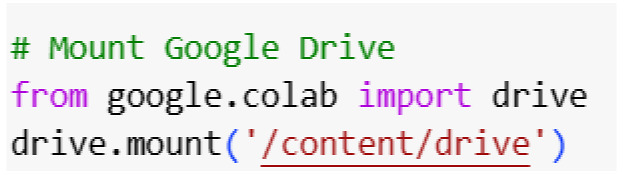
https://github.com/AnkitaSavaliya/AIH/blob/main/MIMIC-IV_MORTALITY_PREDICTION_ARF.ipynb

https://github.com/AnkitaSavaliya/AIH/blob/main/ML-DL_MIMIC-IV_ARF_MORTALITY_PREDICTION.pptx

Step 1: Import required libraries



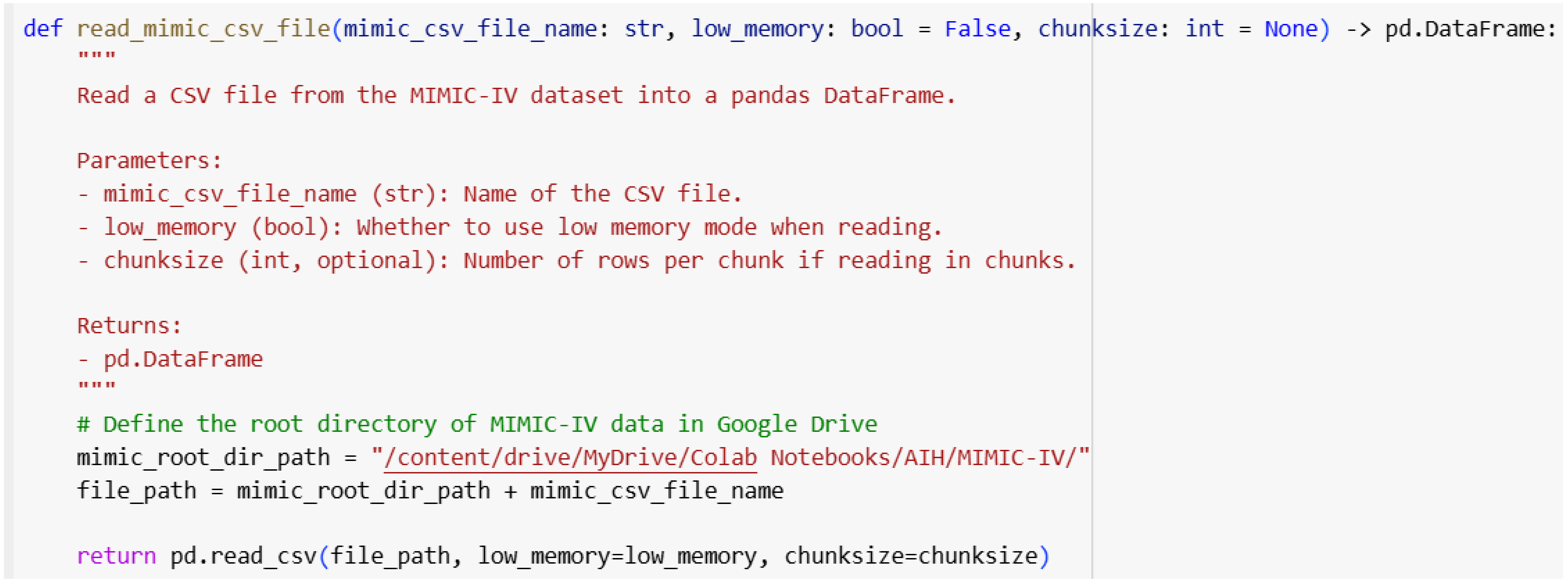
```python
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, roc_auc_score, precision_score, recall_score, f1_score, classification_report, confusion_matrix
from imblearn.over_sampling import SMOTE
from sklearn.utils.class_weight import compute_class_weight
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from xgboost import XGBClassifier
import warnings
```



```python
# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')
```

Import libraries in Google Colab like pandas, sklern, matplotlib, torch etc.
Mount Google Drive where MIMIC-IV CSV files are stored.

Step 2: Loading and Filter Data for ARF

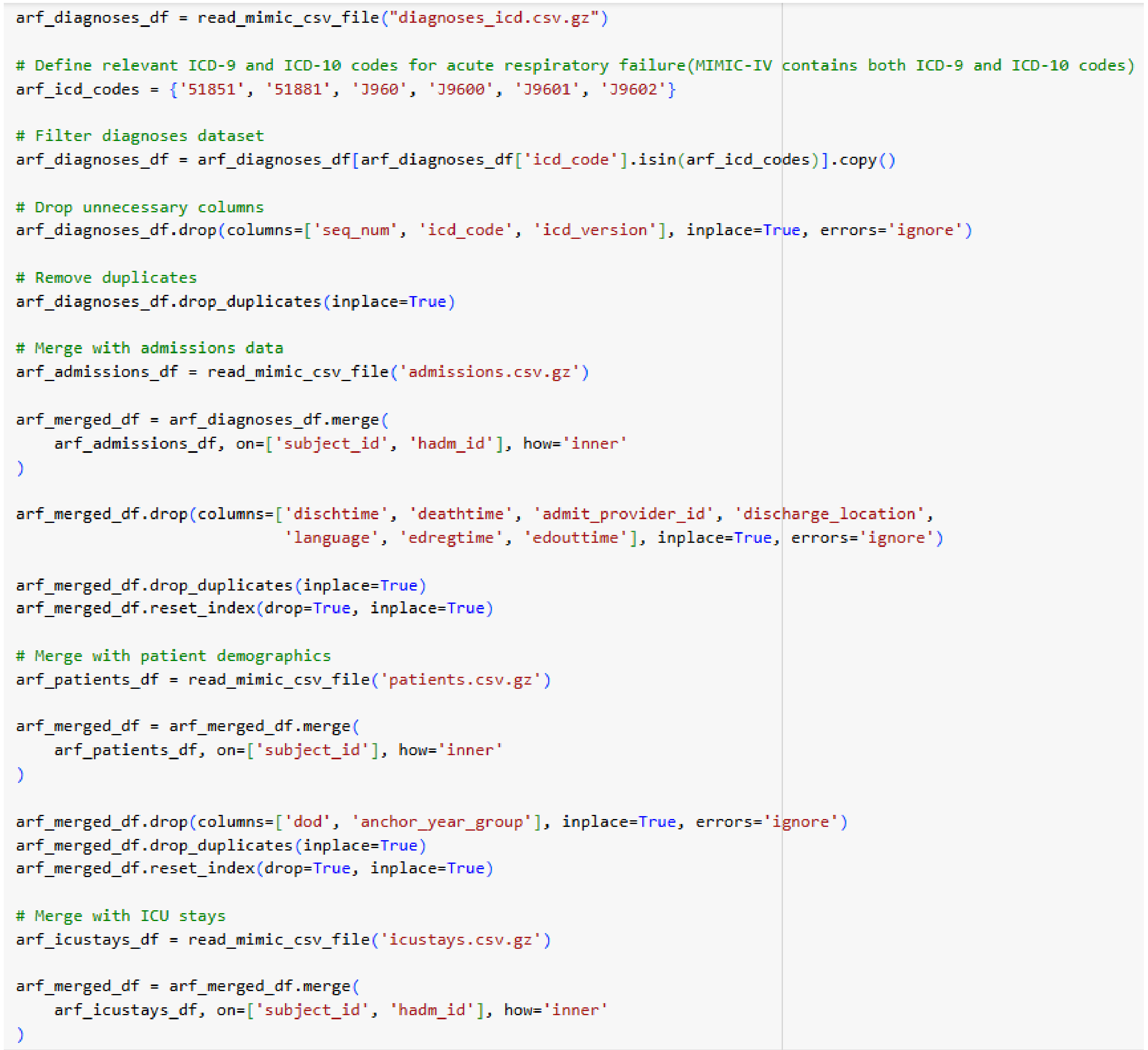


```python
def read_mimic_csv_file(mimic_csv_file_name: str, low_memory: bool = False, chunksize: int = None) -> pd.DataFrame:
    """
    Read a CSV file from the MIMIC-IV dataset into a pandas DataFrame.

    Parameters:
    - mimic_csv_file_name (str): Name of the CSV file.
    - low_memory (bool): Whether to use low memory mode when reading.
    - chunksize (int, optional): Number of rows per chunk if reading in chunks.

    Returns:
    - pd.DataFrame
    """
    # Define the root directory of MIMIC-IV data in Google Drive
    mimic_root_dir_path = "/content/drive/MyDrive/Colab Notebooks/AIH/MIMIC-IV/"
    file_path = mimic_root_dir_path + mimic_csv_file_name

    return pd.read_csv(file_path, low_memory=low_memory, chunksize=chunksize)
```

This function reads a given CSV file from the MIMIC-IV dataset (stored in Google Drive) and returns a pandas DataFrame. Some files, like *labevents*, are very large, so this method supports reading in chunks to optimize memory usage.

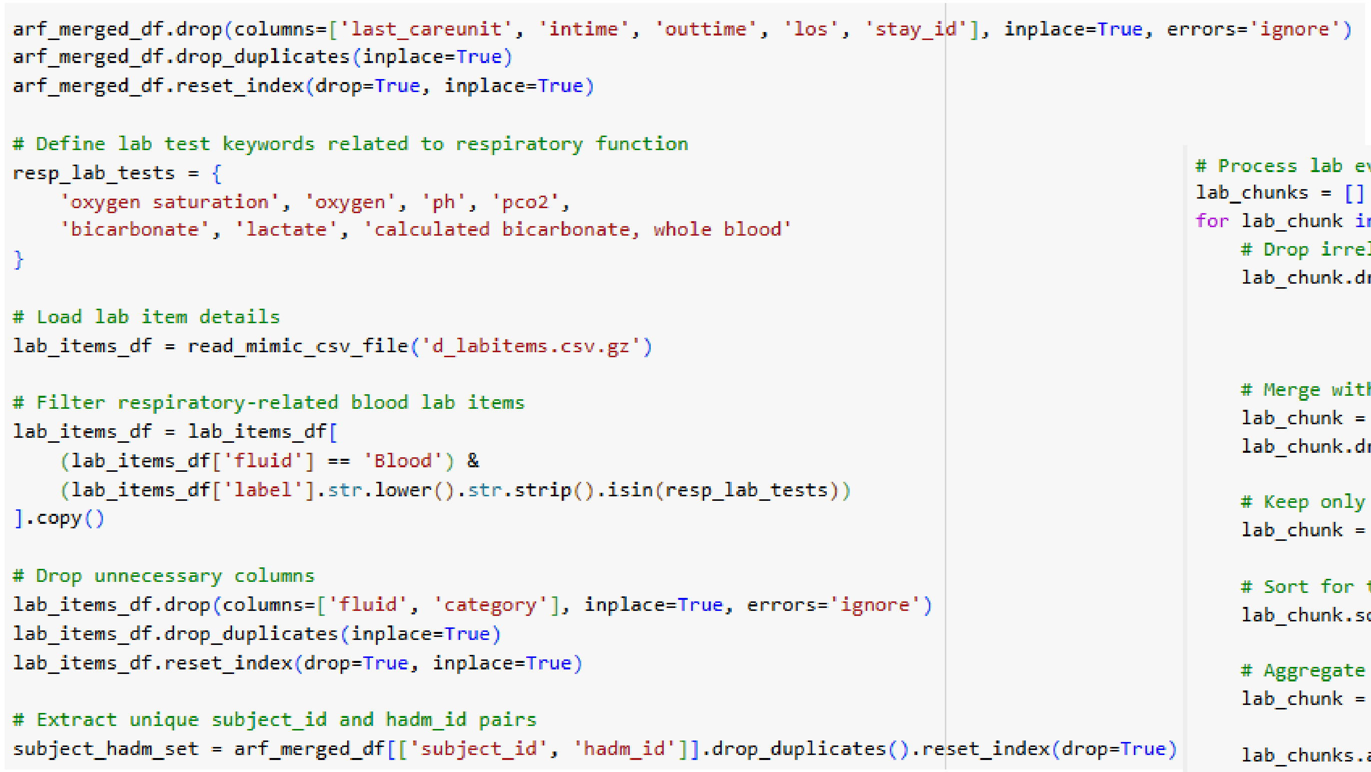
Step 2: Loading and Filter Data for ARF(Continued)...

This part filters diagnoses using ICD-9 and ICD-10 codes related to Acute Respiratory Failure (ARF) and merges the filtered data with the admissions, patients, and ICU stays tables. Additionally, unnecessary columns are dropped, duplicates are removed, and the dataset is reset for a clean structure.

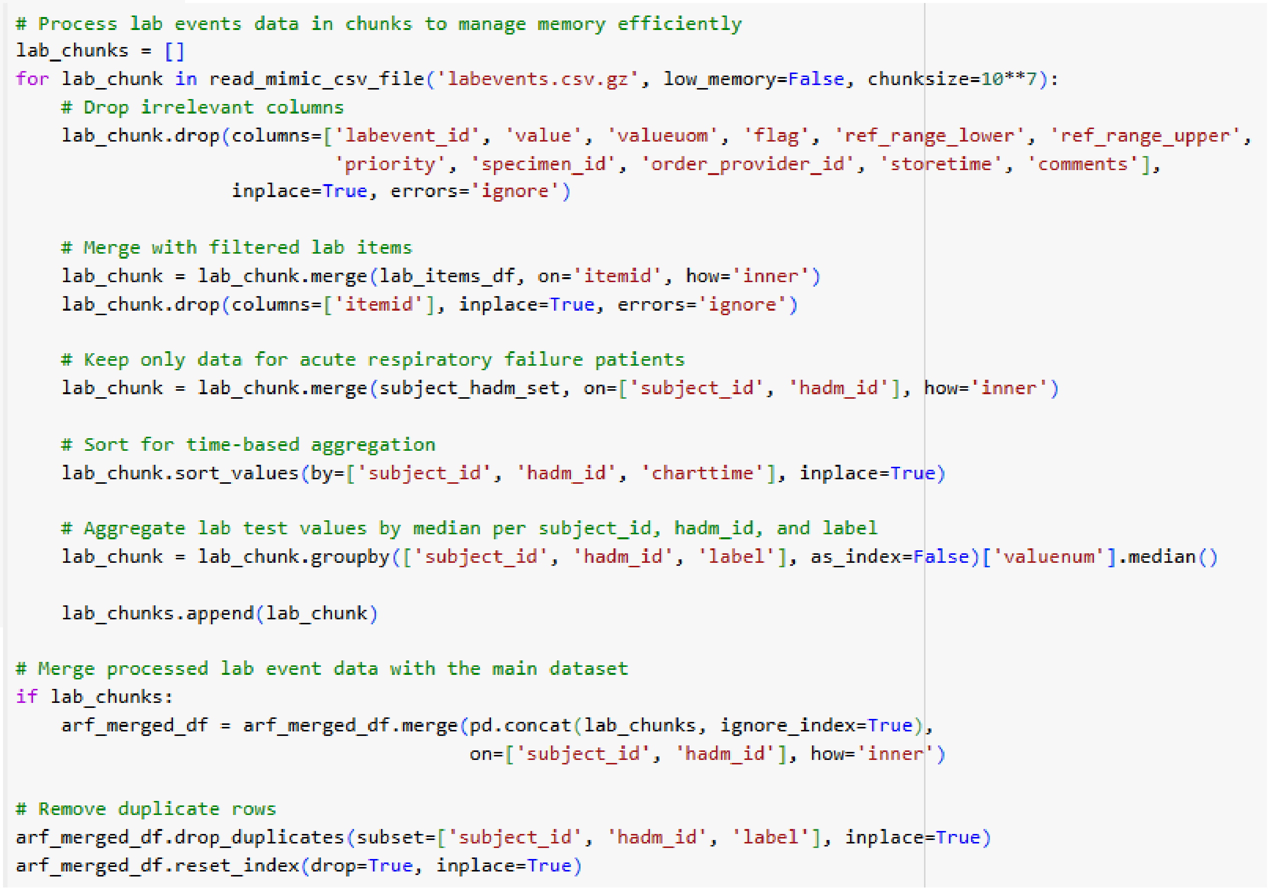


```python
arf_diagnoses_df = read_mimic_csv_file("diagnoses_icd.csv.gz")

# Define relevant ICD-9 and ICD-10 codes for acute respiratory failure(MIMIC-IV contains both ICD-9 and ICD-10 codes)
arf_icd_codes = {'51851', '51881', 'J960', 'J9600', 'J9601', 'J9602'}

# Filter diagnoses dataset
arf_diagnoses_df = arf_diagnoses_df[arf_diagnoses_df['icd_code'].isin(arf_icd_codes)].copy()

# Drop unnecessary columns
arf_diagnoses_df.drop(columns=['seq_num', 'icd_code', 'icd_version'], inplace=True, errors='ignore')

# Remove duplicates
arf_diagnoses_df.drop_duplicates(inplace=True)

# Merge with admissions data
arf_admissions_df = read_mimic_csv_file('admissions.csv.gz')

arf_merged_df = arf_diagnoses_df.merge(
    arf_admissions_df, on=['subject_id', 'hadm_id'], how='inner'
)

arf_merged_df.drop(columns=['dischtime', 'deathtime', 'admit_provider_id', 'discharge_location',
                            'language', 'edregtime', 'edouttime'], inplace=True, errors='ignore')

arf_merged_df.drop_duplicates(inplace=True)
arf_merged_df.reset_index(drop=True, inplace=True)

# Merge with patient demographics
arf_patients_df = read_mimic_csv_file('patients.csv.gz')

arf_merged_df = arf_merged_df.merge(
    arf_patients_df, on=['subject_id'], how='inner'
)

arf_merged_df.drop(columns=['dod', 'anchor_year_group'], inplace=True, errors='ignore')
arf_merged_df.drop_duplicates(inplace=True)
arf_merged_df.reset_index(drop=True, inplace=True)

# Merge with ICU stays
arf_icustays_df = read_mimic_csv_file('icustays.csv.gz')

arf_merged_df = arf_merged_df.merge(
    arf_icustays_df, on=['subject_id', 'hadm_id'], how='inner'
)
```

Step 2: Loading and Filter Data for ARF(Continued)...



```python
arf_merged_df.drop(columns=['last_careunit', 'intime', 'outtime', 'los', 'stay_id'], inplace=True, errors='ignore')
arf_merged_df.drop_duplicates(inplace=True)
arf_merged_df.reset_index(drop=True, inplace=True)

# Define lab test keywords related to respiratory function
resp_lab_tests = {
    'oxygen saturation', 'oxygen', 'ph', 'pco2',
    'bicarbonate', 'lactate', 'calculated bicarbonate, whole blood'
}

# Load lab item details
lab_items_df = read_mimic_csv_file('d_labitems.csv.gz')

# Filter respiratory-related blood lab items
lab_items_df = lab_items_df[
    (lab_items_df['fluid'] == 'Blood') &
    (lab_items_df['label'].str.lower().str.strip().isin(resp_lab_tests))
].copy()

# Drop unnecessary columns
lab_items_df.drop(columns=['fluid', 'category'], inplace=True, errors='ignore')
lab_items_df.drop_duplicates(inplace=True)
lab_items_df.reset_index(drop=True, inplace=True)

# Extract unique subject_id and hadm_id pairs
subject_hadm_set = arf_merged_df[['subject_id', 'hadm_id']].drop_duplicates().reset_index(drop=True)
```

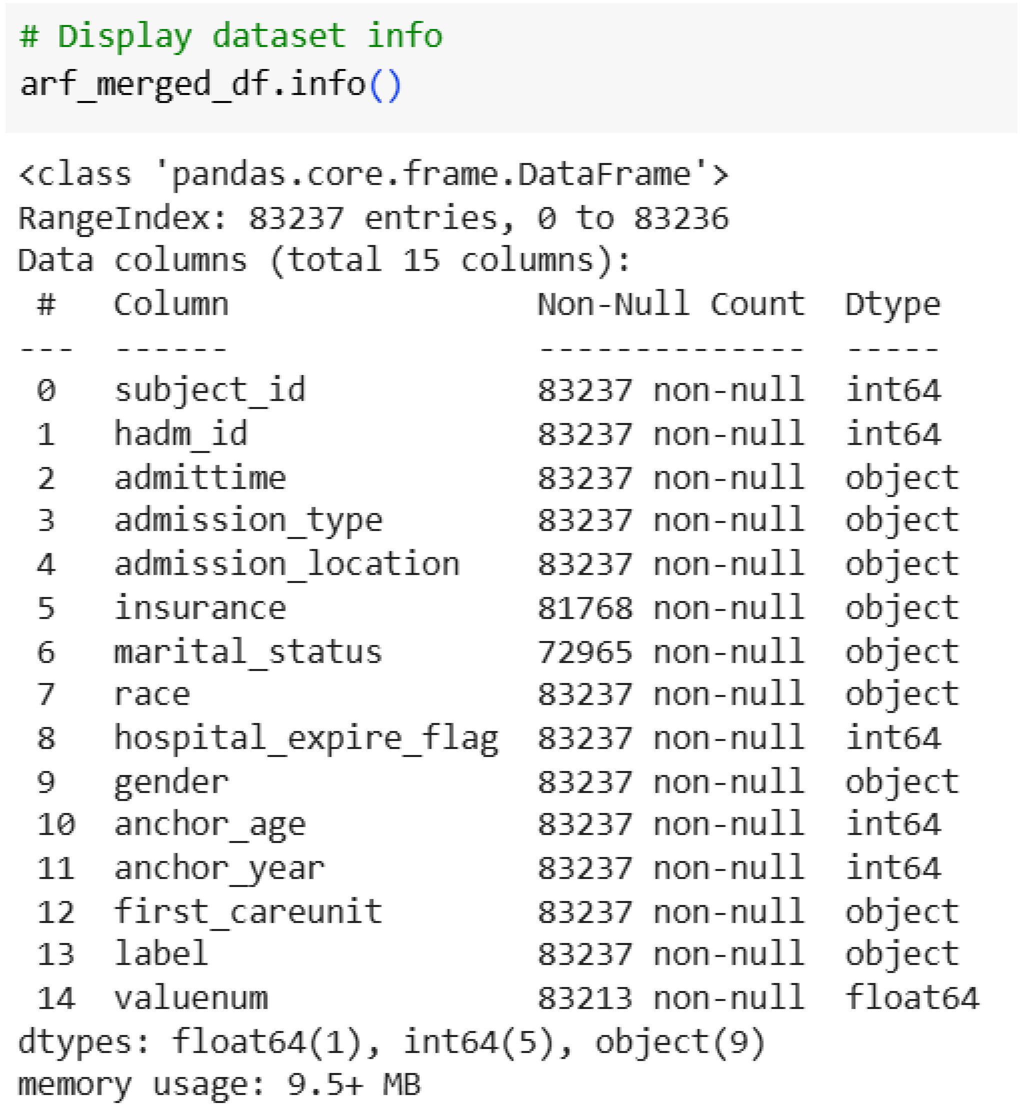


```python
# Process lab events data in chunks to manage memory efficiently
lab_chunks = []
for lab_chunk in read_mimic_csv_file('labevents.csv.gz', low_memory=False, chunksize=10**7):
    # Drop irrelevant columns
    lab_chunk.drop(columns=['labevent_id', 'value', 'valueuom', 'flag', 'ref_range_lower', 'ref_range_upper',
                            'priority', 'specimen_id', 'order_provider_id', 'storetime', 'comments'],
                   inplace=True, errors='ignore')

    # Merge with filtered lab items
    lab_chunk = lab_chunk.merge(lab_items_df, on='itemid', how='inner')
    lab_chunk.drop(columns=['itemid'], inplace=True, errors='ignore')

    # Keep only data for acute respiratory failure patients
    lab_chunk = lab_chunk.merge(subject_hadm_set, on=['subject_id', 'hadm_id'], how='inner')

    # Sort for time-based aggregation
    lab_chunk.sort_values(by=['subject_id', 'hadm_id', 'charttime'], inplace=True)

    # Aggregate lab test values by median per subject_id, hadm_id, and label
    lab_chunk = lab_chunk.groupby(['subject_id', 'hadm_id', 'label'], as_index=False)['valuenum'].median()

    lab_chunks.append(lab_chunk)

# Merge processed lab event data with the main dataset
if lab_chunks:
    arf_merged_df = arf_merged_df.merge(pd.concat(lab_chunks, ignore_index=True),
                                        on=['subject_id', 'hadm_id'], how='inner')

# Remove duplicate rows
arf_merged_df.drop_duplicates(subset=['subject_id', 'hadm_id', 'label'], inplace=True)
arf_merged_df.reset_index(drop=True, inplace=True)
```

- Lab tests, such as oxygen saturation, pH, pCO_2, and bicarbonate levels, are critical indicators of respiratory function and can serve as important features for mortality prediction.
- This code filters ARF-specific lab events and merges the datasets.
- To manage the large volume of lab event data efficiently, we processed it in chunks, ensuring memory optimization throughout the process.

Step 2: Loading and Filter Data for ARF(Continued)...

- After loading and filtering the data, the columns include patient identifiers, admission details, lab test results, and demographic information related to ARF.
- The next step involves understanding, processing, and cleaning the data to ensure it's ready for modeling. This may include handling missing values, transforming variables, and ensuring consistency across datasets.



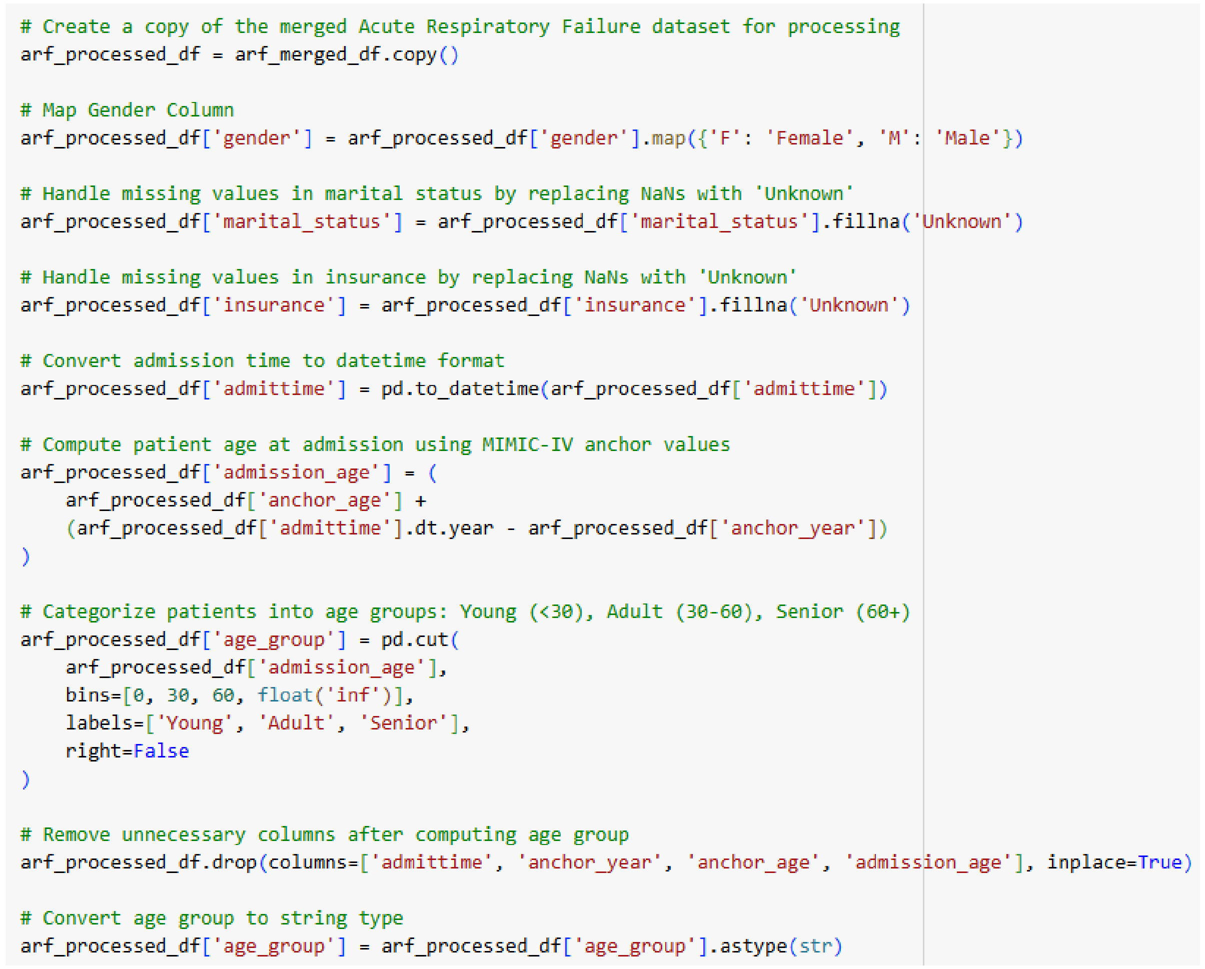
```
# Display dataset info
arf_merged_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 83237 entries, 0 to 83236
Data columns (total 15 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   subject_id           83237 non-null  int64  
 1   hadm_id              83237 non-null  int64  
 2   admittime            83237 non-null  object 
 3   admission_type       83237 non-null  object 
 4   admission_location   83237 non-null  object 
 5   insurance            81768 non-null  object 
 6   marital_status       72965 non-null  object 
 7   race                 83237 non-null  object 
 8   hospital_expire_flag 83237 non-null  int64  
 9   gender               83237 non-null  object 
 10  anchor_age           83237 non-null  int64  
 11  anchor_year          83237 non-null  int64  
 12  first_careunit       83237 non-null  object 
 13  label                83237 non-null  object 
 14  valuenum             83213 non-null  float64
dtypes: float64(1), int64(5), object(9)
memory usage: 9.5+ MB
```

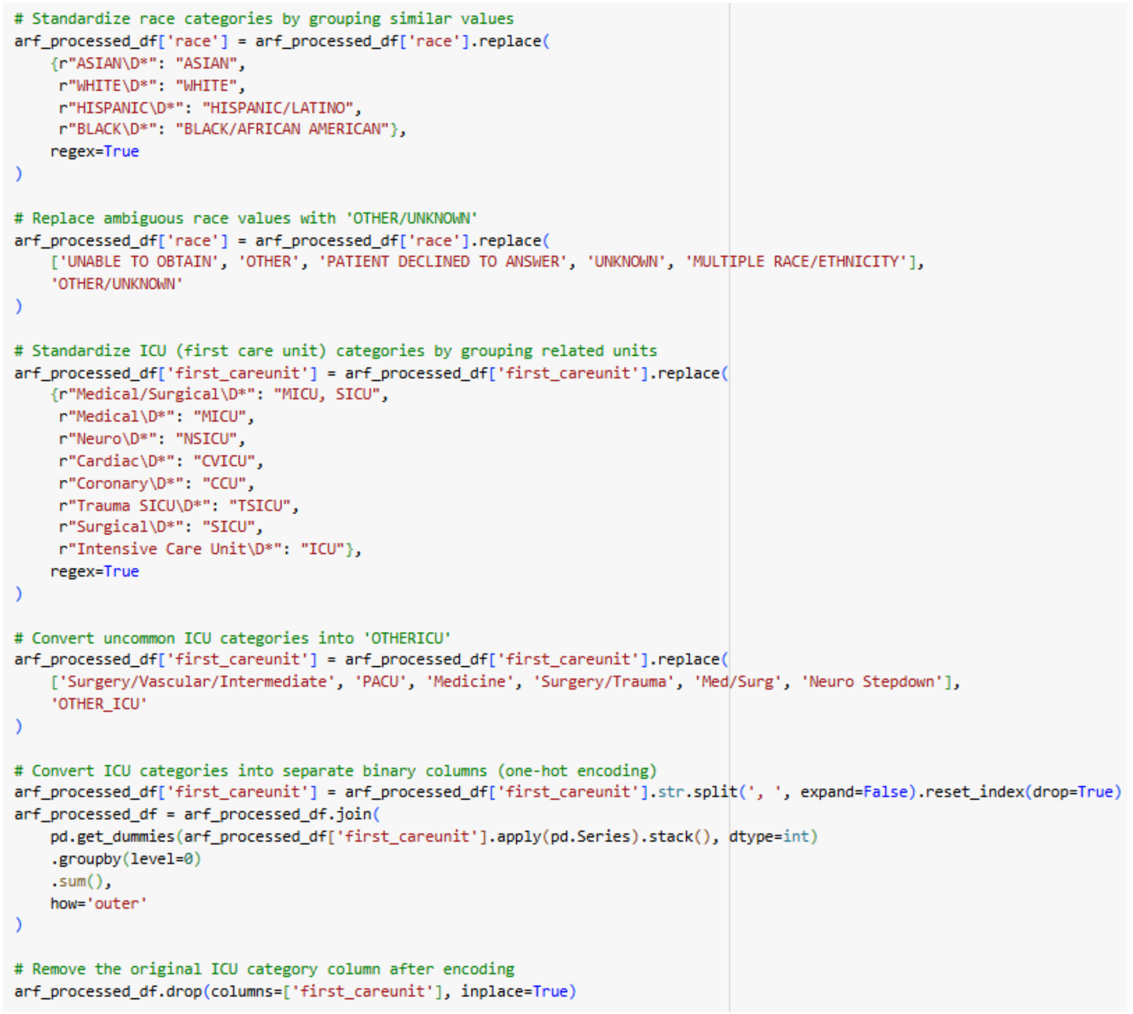
Step 3: Preprocessing and Feature Engineering for ARF

This step performs preprocessing (cleaning, mapping) and feature engineering on the dataset created earlier. The following processing is done:

- Mapping gender, handling missing values in marital status and insurance.
- Calculating age and categorizing patients into age groups.
- Standardizing race and ICU categories and creating separate columns for each ICU unit.
- Removing unnecessary columns to prepare the dataset for analysis.



```python
# Create a copy of the merged Acute Respiratory Failure dataset for processing
arf_processed_df = arf_merged_df.copy()

# Map Gender Column
arf_processed_df['gender'] = arf_processed_df['gender'].map({'F': 'Female', 'M': 'Male'})

# Handle missing values in marital status by replacing NaNs with 'Unknown'
arf_processed_df['marital_status'] = arf_processed_df['marital_status'].fillna('Unknown')

# Handle missing values in insurance by replacing NaNs with 'Unknown'
arf_processed_df['insurance'] = arf_processed_df['insurance'].fillna('Unknown')

# Convert admission time to datetime format
arf_processed_df['admittime'] = pd.to_datetime(arf_processed_df['admittime'])

# Compute patient age at admission using MIMIC-IV anchor values
arf_processed_df['admission_age'] = (
    arf_processed_df['anchor_age'] +
    (arf_processed_df['admittime'].dt.year - arf_processed_df['anchor_year'])
)

# Categorize patients into age groups: Young (<30), Adult (30-60), Senior (60+)
arf_processed_df['age_group'] = pd.cut(
    arf_processed_df['admission_age'],
    bins=[0, 30, 60, float('inf')],
    labels=['Young', 'Adult', 'Senior'],
    right=False
)

# Remove unnecessary columns after computing age group
arf_processed_df.drop(columns=['admittime', 'anchor_year', 'anchor_age', 'admission_age'], inplace=True)

# Convert age group to string type
arf_processed_df['age_group'] = arf_processed_df['age_group'].astype(str)
```

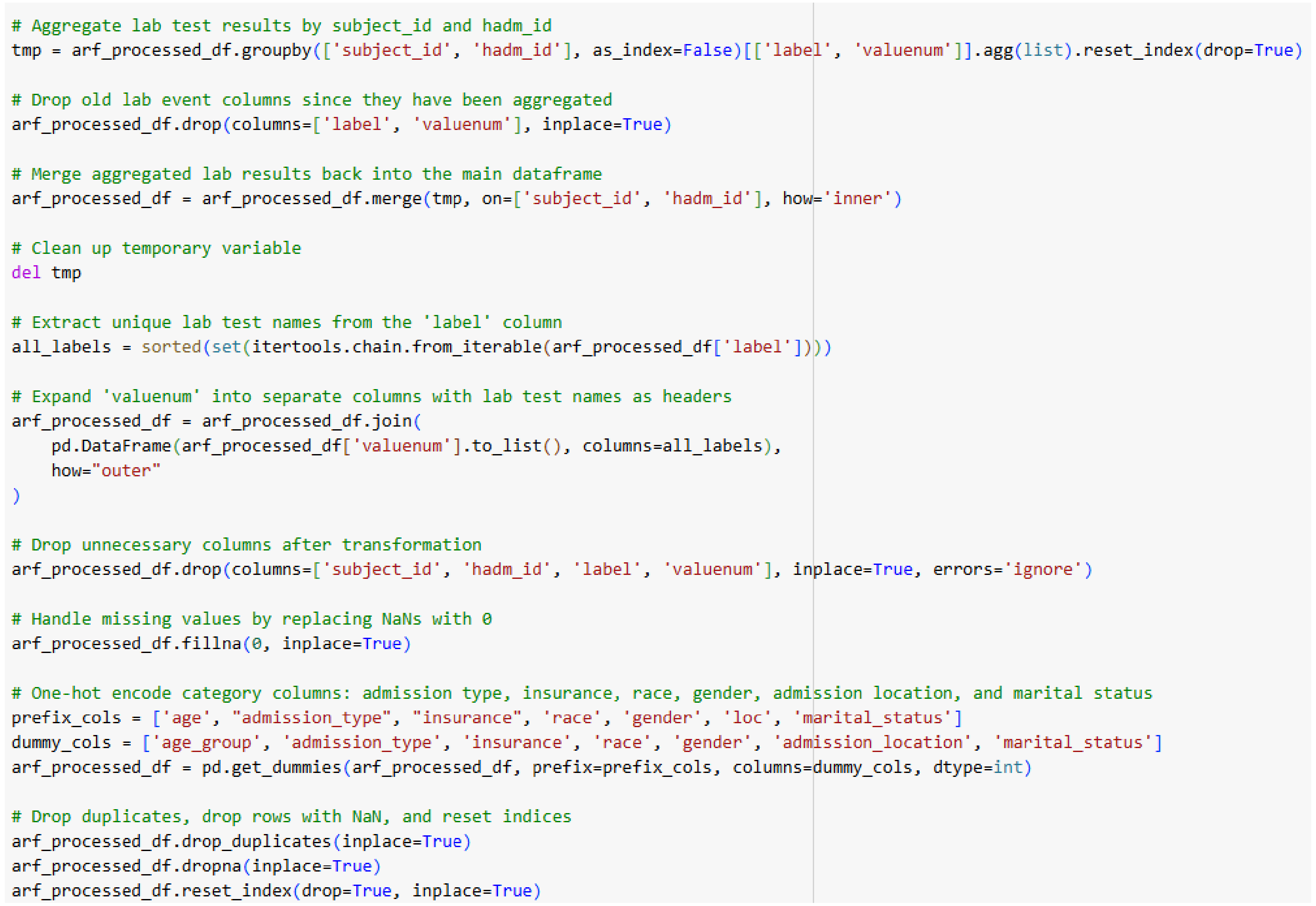


```python
# Standardize race categories by grouping similar values
arf_processed_df['race'] = arf_processed_df['race'].replace(
    {r"ASIAN\D*": "ASIAN",
     r"WHITE\D*": "WHITE",
     r"HISPANIC\D*": "HISPANIC/LATINO",
     r"BLACK\D*": "BLACK/AFRICAN AMERICAN"},
    regex=True
)

# Replace ambiguous race values with 'OTHER/UNKNOWN'
arf_processed_df['race'] = arf_processed_df['race'].replace(
    ['UNABLE TO OBTAIN', 'OTHER', 'PATIENT DECLINED TO ANSWER', 'UNKNOWN', 'MULTIPLE RACE/ETHNICITY'],
    'OTHER/UNKNOWN'
)

# Standardize ICU (first care unit) categories by grouping related units
arf_processed_df['first_careunit'] = arf_processed_df['first_careunit'].replace(
    {r"Medical/Surgical\D*": "MICU, SICU",
     r"Medical\D*": "MICU",
     r"Neuro\D*": "NSICU",
     r"Cardiac\D*": "CVICU",
     r"Coronary\D*": "CCU",
     r"Trauma SICU\D*": "TSICU",
     r"Surgical\D*": "SICU",
     r"Intensive Care Unit\D*": "ICU"},
    regex=True
)

# Convert uncommon ICU categories into 'OTHERICU'
arf_processed_df['first_careunit'] = arf_processed_df['first_careunit'].replace(
    ['Surgery/Vascular/Intermediate', 'PACU', 'Medicine', 'Surgery/Trauma', 'Med/Surg', 'Neuro Stepdown'],
    'OTHER_ICU'
)

# Convert ICU categories into separate binary columns (one-hot encoding)
arf_processed_df['first_careunit'] = arf_processed_df['first_careunit'].str.split(', ', expand=False).reset_index(drop=True)
arf_processed_df = arf_processed_df.join(
    pd.get_dummies(arf_processed_df['first_careunit'].apply(pd.Series).stack(), dtype=int)
    .groupby(level=0)
    .sum(),
    how='outer'
)

# Remove the original ICU category column after encoding
arf_processed_df.drop(columns=['first_careunit'], inplace=True)
```

Step 3: Preprocessing and Feature Engineering for ARF(Continued)...

Continuing data processing:

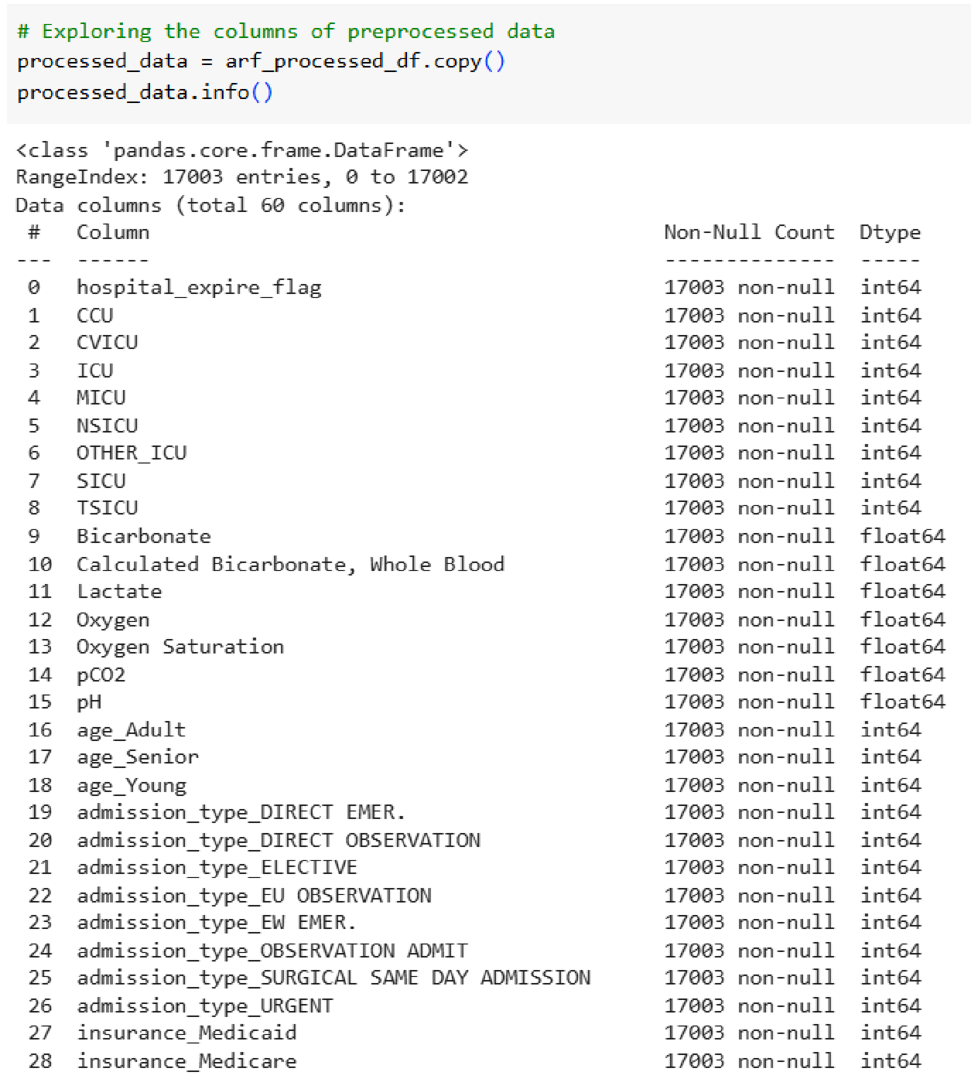
- Aggregate lab test results for each patient encounter.
- Transform lab test names into separate columns.
- Replace missing values (NaNs) with 0.
- Convert categorical features (admission type, insurance, race, gender, admission location, and marital status) into a binary format.
- Remove duplicates and reset indices to finalize the dataset for model training.



```python
# Aggregate lab test results by subject_id and hadm_id
tmp = arf_processed_df.groupby(['subject_id', 'hadm_id'], as_index=False)[['label', 'valuenum']].agg(list).reset_index(drop=True)

# Drop old lab event columns since they have been aggregated
arf_processed_df.drop(columns=['label', 'valuenum'], inplace=True)

# Merge aggregated lab results back into the main dataframe
arf_processed_df = arf_processed_df.merge(tmp, on=['subject_id', 'hadm_id'], how='inner')

# Clean up temporary variable
del tmp

# Extract unique lab test names from the 'label' column
all_labels = sorted(set(itertools.chain.from_iterable(arf_processed_df['label'])))

# Expand 'valuenum' into separate columns with lab test names as headers
arf_processed_df = arf_processed_df.join(
    pd.DataFrame(arf_processed_df['valuenum'].to_list(), columns=all_labels),
    how="outer"
)

# Drop unnecessary columns after transformation
arf_processed_df.drop(columns=['subject_id', 'hadm_id', 'label', 'valuenum'], inplace=True, errors='ignore')

# Handle missing values by replacing NaNs with 0
arf_processed_df.fillna(0, inplace=True)

# One-hot encode category columns: admission type, insurance, race, gender, admission location, and marital status
prefix_cols = ['age', "admission_type", "insurance", 'race', 'gender', 'loc', 'marital_status']
dummy_cols = ['age_group', 'admission_type', 'insurance', 'race', 'gender', 'admission_location', 'marital_status']
arf_processed_df = pd.get_dummies(arf_processed_df, prefix=prefix_cols, columns=dummy_cols, dtype=int)

# Drop duplicates, drop rows with NaN, and reset indices
arf_processed_df.drop_duplicates(inplace=True)
arf_processed_df.dropna(inplace=True)
arf_processed_df.reset_index(drop=True, inplace=True)
```

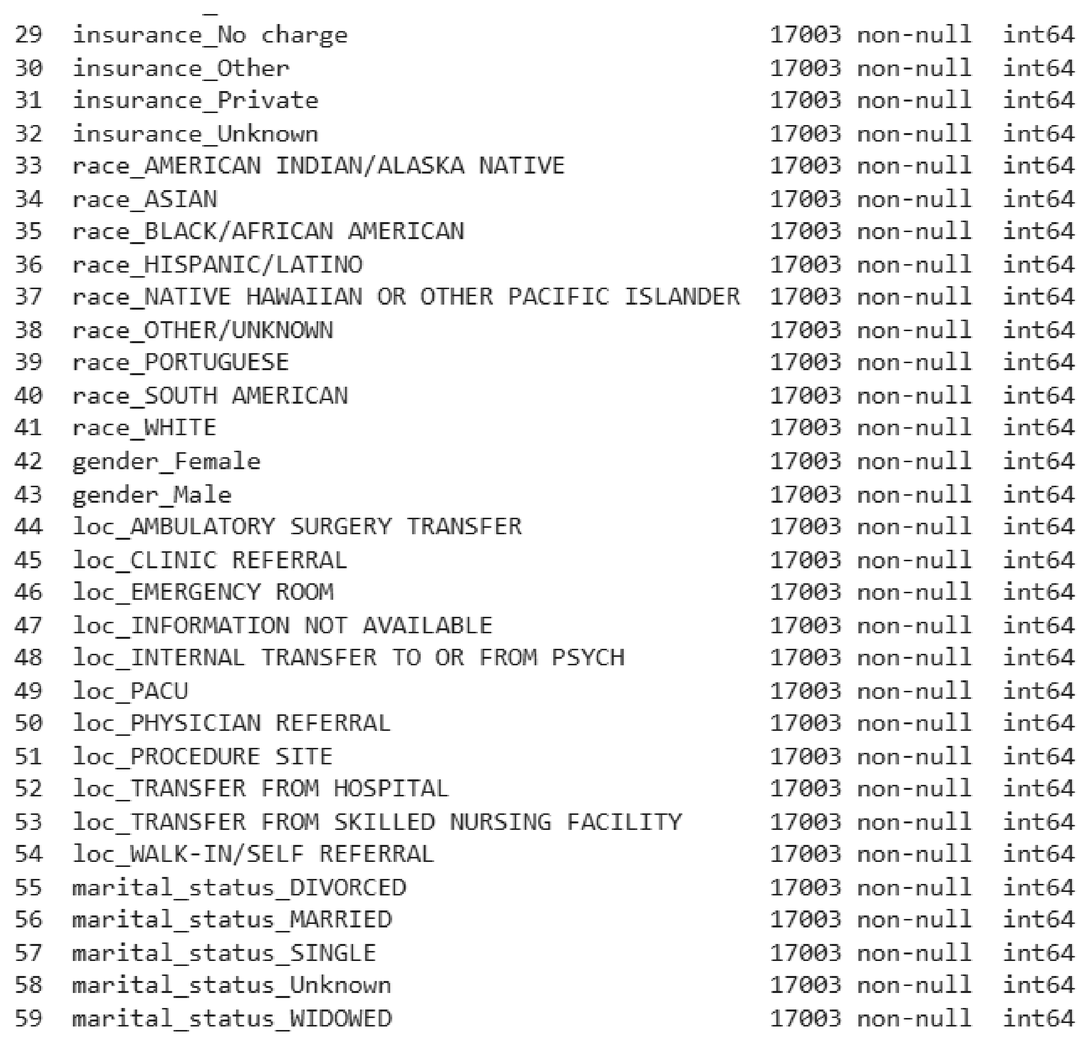
Step 3: Preprocessing and Feature Engineering for ARF(Continued)…

After processing and feature engineering, the dataset now contains the following columns. It is now ready for model training.



```
# Exploring the columns of preprocessed data
processed_data = arf_processed_df.copy()
processed_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17003 entries, 0 to 17002
Data columns (total 60 columns):
 #   Column                                  Non-Null Count  Dtype
---  ------                                  --------------  -----
 0   hospital_expire_flag                    17003 non-null  int64
 1   CCU                                     17003 non-null  int64
 2   CVICU                                   17003 non-null  int64
 3   ICU                                     17003 non-null  int64
 4   MICU                                    17003 non-null  int64
 5   NSICU                                   17003 non-null  int64
 6   OTHER_ICU                               17003 non-null  int64
 7   SICU                                    17003 non-null  int64
 8   TSICU                                   17003 non-null  int64
 9   Bicarbonate                             17003 non-null  float64
 10  Calculated Bicarbonate, Whole Blood     17003 non-null  float64
 11  Lactate                                 17003 non-null  float64
 12  Oxygen                                  17003 non-null  float64
 13  Oxygen Saturation                       17003 non-null  float64
 14  pCO2                                    17003 non-null  float64
 15  pH                                      17003 non-null  float64
 16  age_Adult                               17003 non-null  int64
 17  age_Senior                              17003 non-null  int64
 18  age_Young                               17003 non-null  int64
 19  admission_type_DIRECT EMER.             17003 non-null  int64
 20  admission_type_DIRECT OBSERVATION       17003 non-null  int64
 21  admission_type_ELECTIVE                 17003 non-null  int64
 22  admission_type_EU OBSERVATION           17003 non-null  int64
 23  admission_type_EW EMER.                 17003 non-null  int64
 24  admission_type_OBSERVATION ADMIT        17003 non-null  int64
 25  admission_type_SURGICAL SAME DAY ADMISSION  17003 non-null  int64
 26  admission_type_URGENT                   17003 non-null  int64
 27  insurance_Medicaid                      17003 non-null  int64
 28  insurance_Medicare                      17003 non-null  int64
```

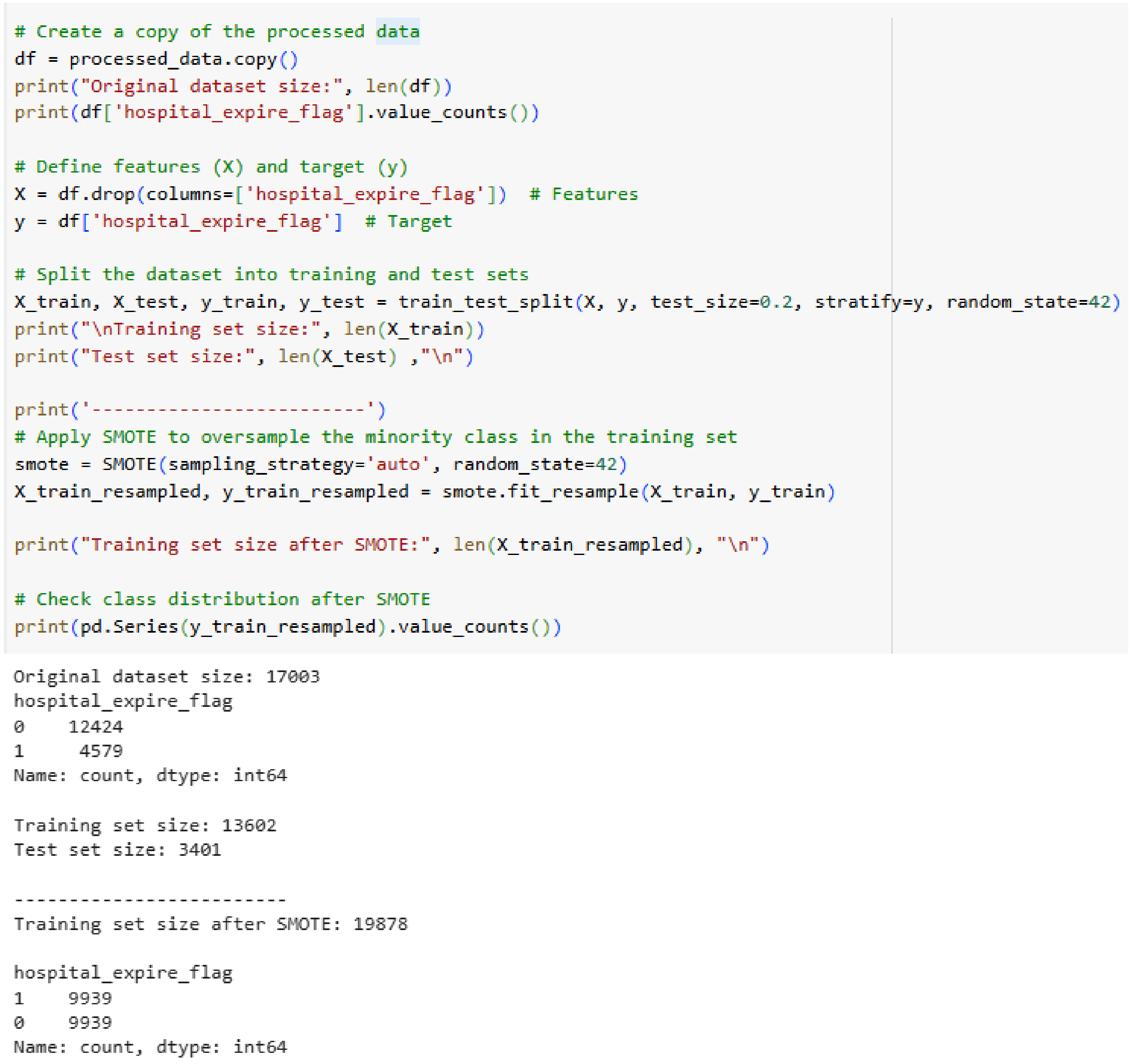


```
 29  insurance_No charge                              17003 non-null  int64
 30  insurance_Other                                  17003 non-null  int64
 31  insurance_Private                                17003 non-null  int64
 32  insurance_Unknown                                17003 non-null  int64
 33  race_AMERICAN INDIAN/ALASKA NATIVE               17003 non-null  int64
 34  race_ASIAN                                       17003 non-null  int64
 35  race_BLACK/AFRICAN AMERICAN                      17003 non-null  int64
 36  race_HISPANIC/LATINO                             17003 non-null  int64
 37  race_NATIVE HAWAIIAN OR OTHER PACIFIC ISLANDER   17003 non-null  int64
 38  race_OTHER/UNKNOWN                               17003 non-null  int64
 39  race_PORTUGUESE                                  17003 non-null  int64
 40  race_SOUTH AMERICAN                              17003 non-null  int64
 41  race_WHITE                                       17003 non-null  int64
 42  gender_Female                                    17003 non-null  int64
 43  gender_Male                                      17003 non-null  int64
 44  loc_AMBULATORY SURGERY TRANSFER                  17003 non-null  int64
 45  loc_CLINIC REFERRAL                              17003 non-null  int64
 46  loc_EMERGENCY ROOM                               17003 non-null  int64
 47  loc_INFORMATION NOT AVAILABLE                    17003 non-null  int64
 48  loc_INTERNAL TRANSFER TO OR FROM PSYCH           17003 non-null  int64
 49  loc_PACU                                         17003 non-null  int64
 50  loc_PHYSICIAN REFERRAL                           17003 non-null  int64
 51  loc_PROCEDURE SITE                               17003 non-null  int64
 52  loc_TRANSFER FROM HOSPITAL                       17003 non-null  int64
 53  loc_TRANSFER FROM SKILLED NURSING FACILITY       17003 non-null  int64
 54  loc_WALK-IN/SELF REFERRAL                        17003 non-null  int64
 55  marital_status_DIVORCED                          17003 non-null  int64
 56  marital_status_MARRIED                           17003 non-null  int64
 57  marital_status_SINGLE                            17003 non-null  int64
 58  marital_status_Unknown                           17003 non-null  int64
 59  marital_status_WIDOWED                           17003 non-null  int64
```

Step 4: Splitting the Data into Training and Test Sets

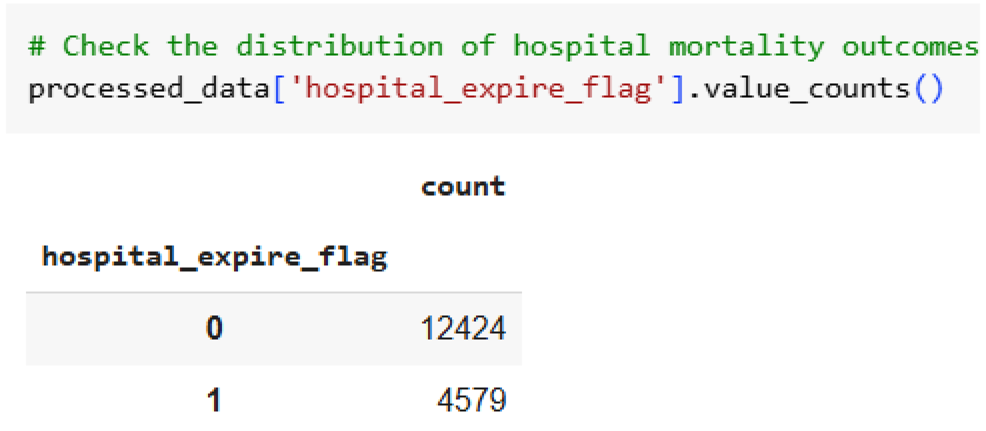
This step prepares the dataset for modeling:

- Defines features (X) and target (y).
- Splits the dataset into training (80%) and test (20%) sets while preserving class distribution.
- As the data is imbalanced(12424 negative case and 4579 positive case), applies SMOTE to oversample the minority class in the training set, ensuring a balanced dataset for better model performance.



```
# Create a copy of the processed data
df = processed_data.copy()
print("Original dataset size:", len(df))
print(df['hospital_expire_flag'].value_counts())

# Define features (X) and target (y)
X = df.drop(columns=['hospital_expire_flag'])  # Features
y = df['hospital_expire_flag']  # Target

# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)
print("\nTraining set size:", len(X_train))
print("Test set size:", len(X_test) ,"\n")

print('-------------------------')
# Apply SMOTE to oversample the minority class in the training set
smote = SMOTE(sampling_strategy='auto', random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)

print("Training set size after SMOTE:", len(X_train_resampled), "\n")

# Check class distribution after SMOTE
print(pd.Series(y_train_resampled).value_counts())
```

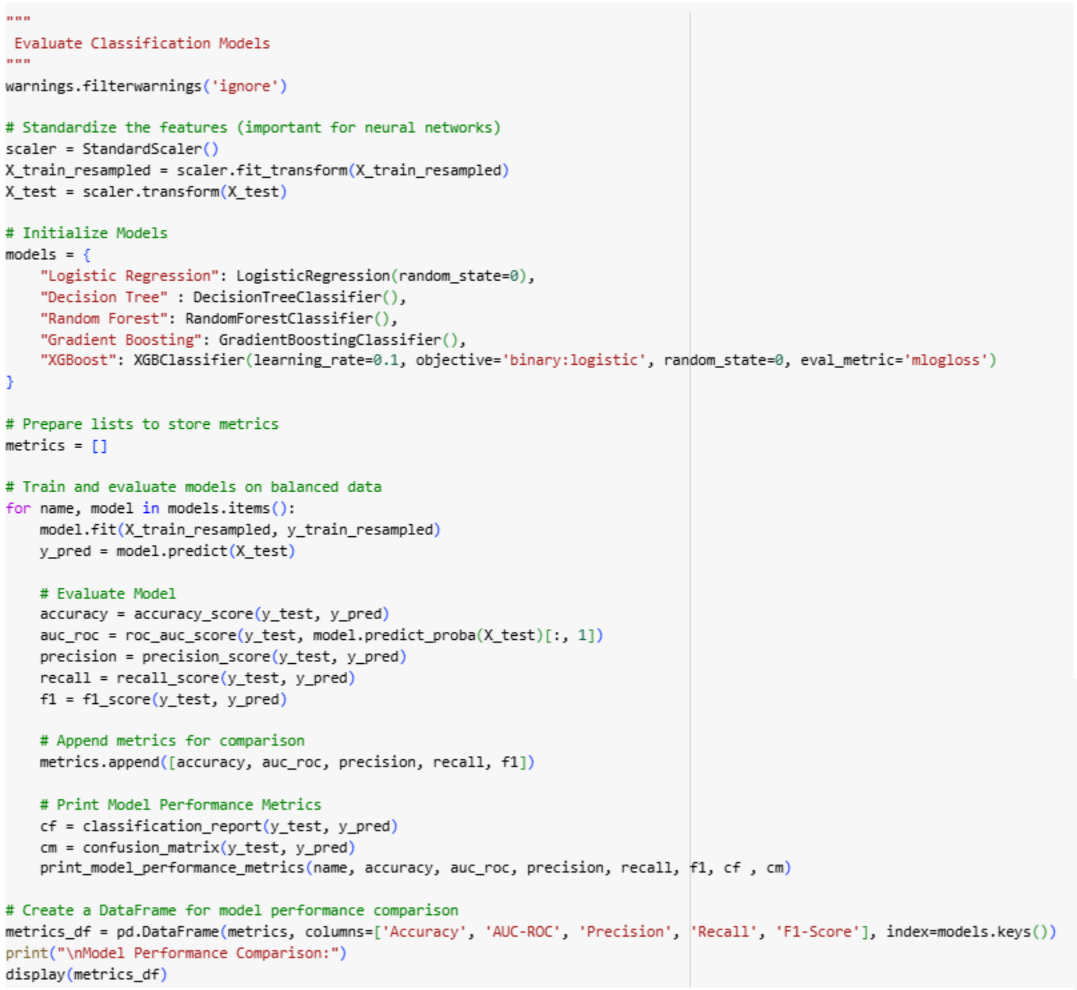
```
Original dataset size: 17003
hospital_expire_flag
0    12424
1     4579
Name: count, dtype: int64

Training set size: 13602
Test set size: 3401

-------------------------
Training set size after SMOTE: 19878

hospital_expire_flag
1    9939
0    9939
Name: count, dtype: int64
```



```
# Check the distribution of hospital mortality outcomes
processed_data['hospital_expire_flag'].value_counts()
```

hospital_expire_flag	count
0	12424
1	4579

Step 5: Model Evaluation and Comparison (sklearn models and XGBoost)

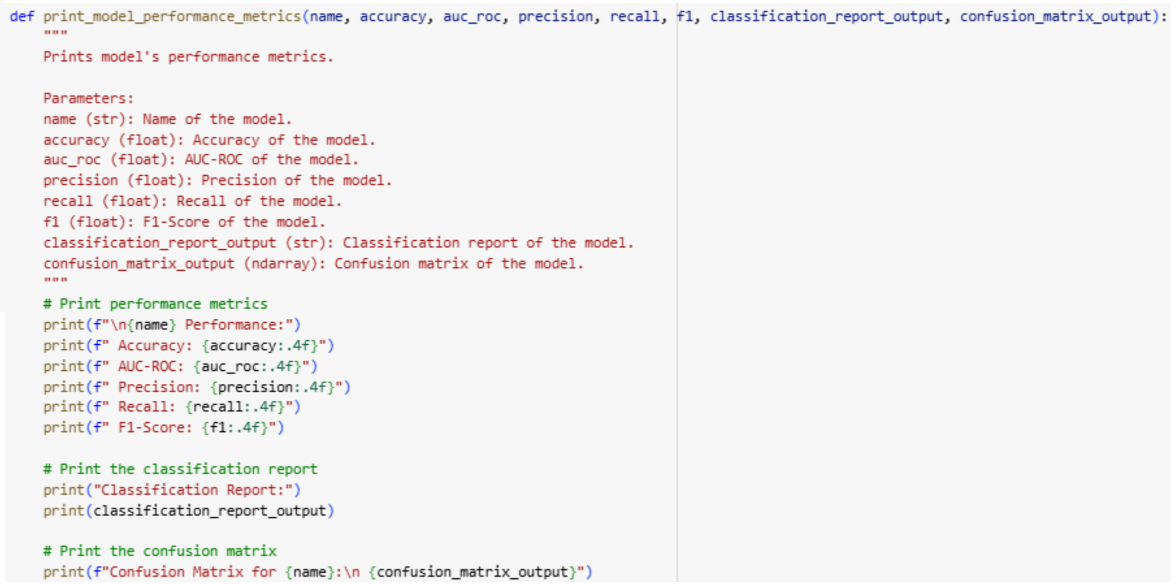


```
"""
 Evaluate Classification Models
"""
warnings.filterwarnings('ignore')

# Standardize the features (important for neural networks)
scaler = StandardScaler()
X_train_resampled = scaler.fit_transform(X_train_resampled)
X_test = scaler.transform(X_test)

# Initialize Models
models = {
    "Logistic Regression": LogisticRegression(random_state=0),
    "Decision Tree" : DecisionTreeClassifier(),
    "Random Forest": RandomForestClassifier(),
    "Gradient Boosting": GradientBoostingClassifier(),
    "XGBoost": XGBClassifier(learning_rate=0.1, objective='binary:logistic', random_state=0, eval_metric='mlogloss')
}

# Prepare lists to store metrics
metrics = []

# Train and evaluate models on balanced data
for name, model in models.items():
    model.fit(X_train_resampled, y_train_resampled)
    y_pred = model.predict(X_test)

    # Evaluate Model
    accuracy = accuracy_score(y_test, y_pred)
    auc_roc = roc_auc_score(y_test, model.predict_proba(X_test)[:, 1])
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)

    # Append metrics for comparison
    metrics.append([accuracy, auc_roc, precision, recall, f1])

    # Print Model Performance Metrics
    cf = classification_report(y_test, y_pred)
    cm = confusion_matrix(y_test, y_pred)
    print_model_performance_metrics(name, accuracy, auc_roc, precision, recall, f1, cf , cm)

# Create a DataFrame for model performance comparison
metrics_df = pd.DataFrame(metrics, columns=['Accuracy', 'AUC-ROC', 'Precision', 'Recall', 'F1-Score'], index=models.keys())
print("\nModel Performance Comparison:")
display(metrics_df)
```

This step evaluates various classification models to predict mortality in ARF patients. The features are standardized to improve model performance. Multiple models—Logistic Regression, Decision Tree, Random Forest, Gradient Boosting, and XGBoost—are trained and tested. The models are evaluated and compared using the following performance metrics:

Accuracy: The proportion of correct predictions out of all predictions.
AUC-ROC: A measure of the model's ability to distinguish between classes.
Precision: The proportion of true positive predictions among all positive predictions.
Recall: The proportion of actual positive cases correctly identified by the model.
F1-Score: The harmonic mean of precision and recall, balancing both metrics.

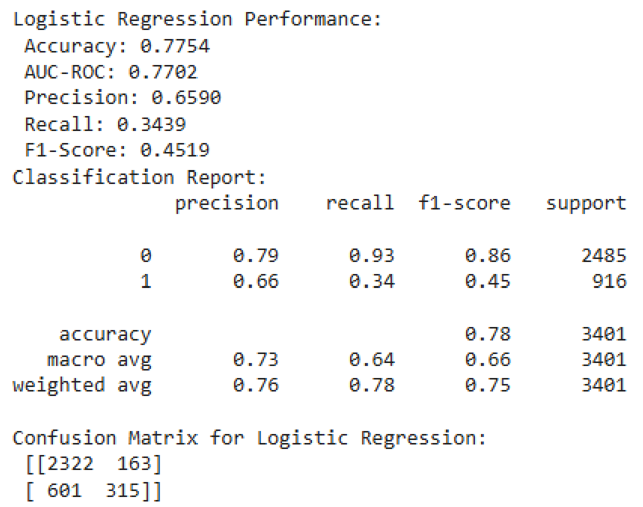
This comparison helps identify the best-performing model for predicting mortality in ARF patients.



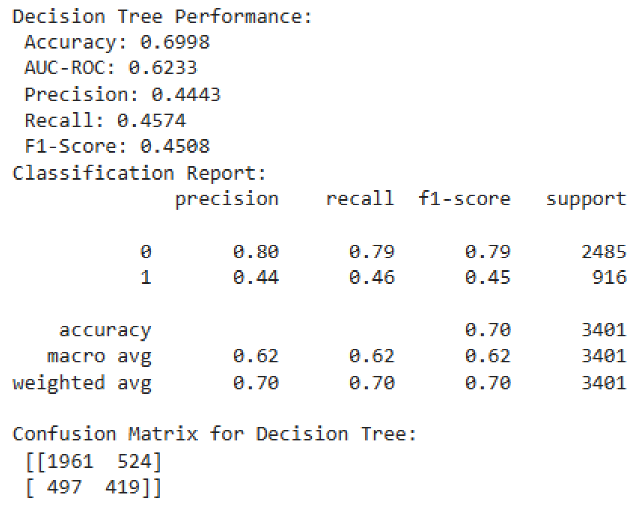
```
def print_model_performance_metrics(name, accuracy, auc_roc, precision, recall, f1, classification_report_output, confusion_matrix_output):
    """
    Prints model's performance metrics.

    Parameters:
    name (str): Name of the model.
    accuracy (float): Accuracy of the model.
    auc_roc (float): AUC-ROC of the model.
    precision (float): Precision of the model.
    recall (float): Recall of the model.
    f1 (float): F1-Score of the model.
    classification_report_output (str): Classification report of the model.
    confusion_matrix_output (ndarray): Confusion matrix of the model.
    """
    # Print performance metrics
    print(f"\n{name} Performance:")
    print(f" Accuracy: {accuracy:.4f}")
    print(f" AUC-ROC: {auc_roc:.4f}")
    print(f" Precision: {precision:.4f}")
    print(f" Recall: {recall:.4f}")
    print(f" F1-Score: {f1:.4f}")

    # Print the classification report
    print("Classification Report:")
    print(classification_report_output)

    # Print the confusion matrix
    print(f"Confusion Matrix for {name}:\n {confusion_matrix_output}")
```

Step 5: Model Evaluation and Comparison (sklearn models and XGBoost) (Continued)...

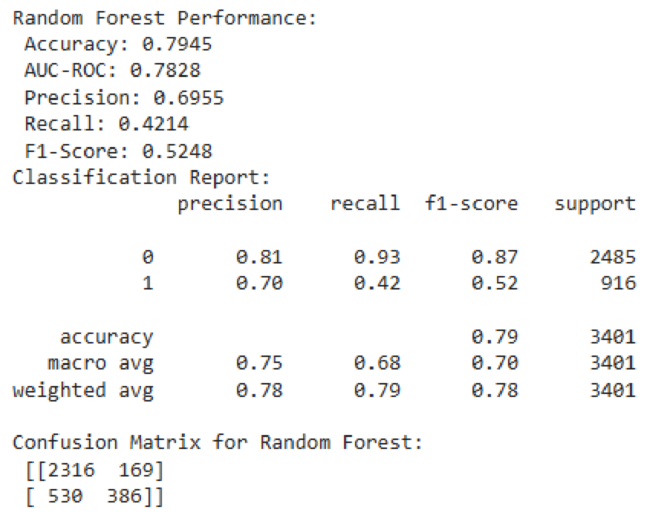
Following is outcome of how each model performed.



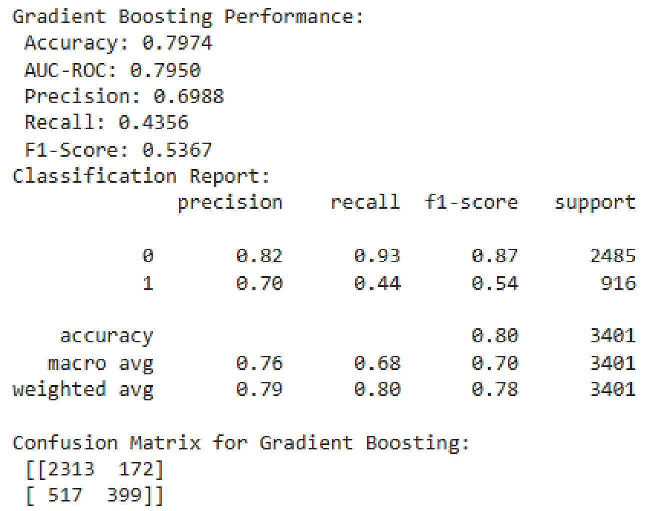
```
Logistic Regression Performance:
 Accuracy: 0.7754
 AUC-ROC: 0.7702
 Precision: 0.6590
 Recall: 0.3439
 F1-Score: 0.4519
Classification Report:
              precision    recall  f1-score   support

           0       0.79      0.93      0.86      2485
           1       0.66      0.34      0.45       916

    accuracy                           0.78      3401
   macro avg       0.73      0.64      0.66      3401
weighted avg       0.76      0.78      0.75      3401

Confusion Matrix for Logistic Regression:
 [[2322  163]
 [ 601  315]]
```



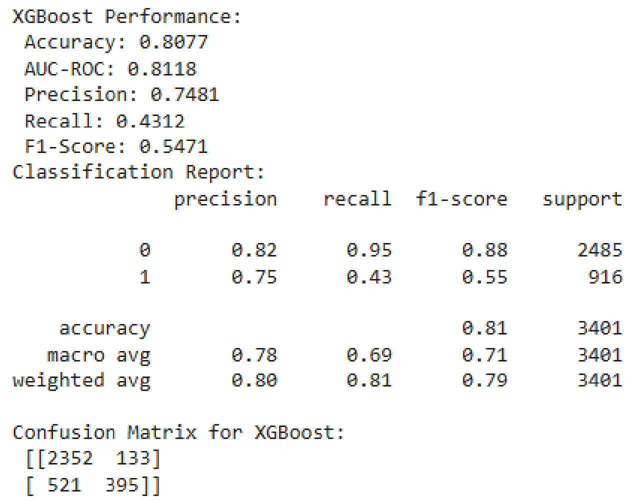
```
Decision Tree Performance:
 Accuracy: 0.6998
 AUC-ROC: 0.6233
 Precision: 0.4443
 Recall: 0.4574
 F1-Score: 0.4508
Classification Report:
              precision    recall  f1-score   support

           0       0.80      0.79      0.79      2485
           1       0.44      0.46      0.45       916

    accuracy                           0.70      3401
   macro avg       0.62      0.62      0.62      3401
weighted avg       0.70      0.70      0.70      3401

Confusion Matrix for Decision Tree:
 [[1961  524]
 [ 497  419]]
```



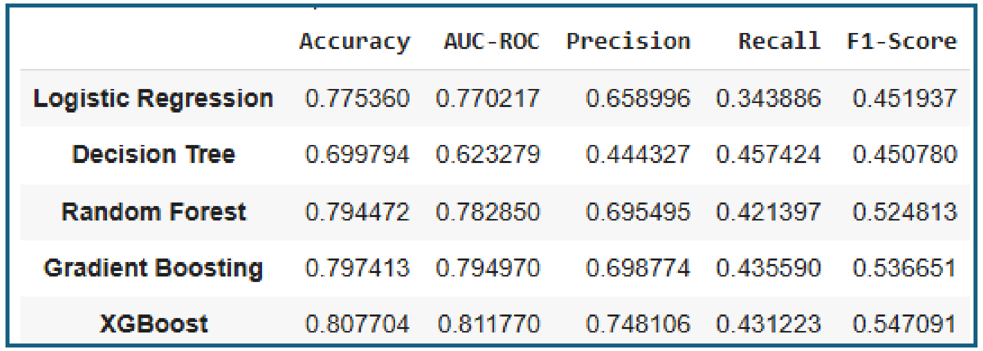
```
Random Forest Performance:
 Accuracy: 0.7945
 AUC-ROC: 0.7828
 Precision: 0.6955
 Recall: 0.4214
 F1-Score: 0.5248
Classification Report:
              precision    recall  f1-score   support

           0       0.81      0.93      0.87      2485
           1       0.70      0.42      0.52       916

    accuracy                           0.79      3401
   macro avg       0.75      0.68      0.70      3401
weighted avg       0.78      0.79      0.78      3401

Confusion Matrix for Random Forest:
 [[2316  169]
 [ 530  386]]
```



```
Gradient Boosting Performance:
 Accuracy: 0.7974
 AUC-ROC: 0.7950
 Precision: 0.6988
 Recall: 0.4356
 F1-Score: 0.5367
Classification Report:
              precision    recall  f1-score   support

           0       0.82      0.93      0.87      2485
           1       0.70      0.44      0.54       916

    accuracy                           0.80      3401
   macro avg       0.76      0.68      0.70      3401
weighted avg       0.79      0.80      0.78      3401

Confusion Matrix for Gradient Boosting:
 [[2313  172]
 [ 517  399]]
```



```
XGBoost Performance:
 Accuracy: 0.8077
 AUC-ROC: 0.8118
 Precision: 0.7481
 Recall: 0.4312
 F1-Score: 0.5471
Classification Report:
              precision    recall  f1-score   support

           0       0.82      0.95      0.88      2485
           1       0.75      0.43      0.55       916

    accuracy                           0.81      3401
   macro avg       0.78      0.69      0.71      3401
weighted avg       0.80      0.81      0.79      3401

Confusion Matrix for XGBoost:
 [[2352  133]
 [ 521  395]]
```

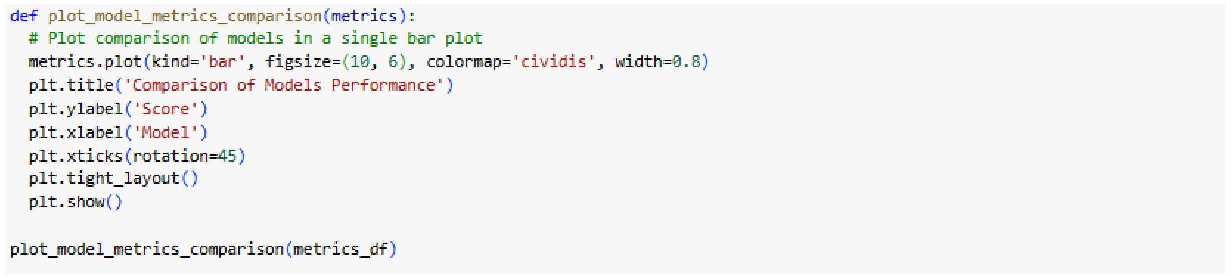


	Accuracy	AUC-ROC	Precision	Recall	F1-Score
Logistic Regression	0.775360	0.770217	0.658996	0.343886	0.451937
Decision Tree	0.699794	0.623279	0.444327	0.457424	0.450780
Random Forest	0.794472	0.782850	0.695495	0.421397	0.524813
Gradient Boosting	0.797413	0.794970	0.698774	0.435590	0.536651
XGBoost	0.807704	0.811770	0.748106	0.431223	0.547091

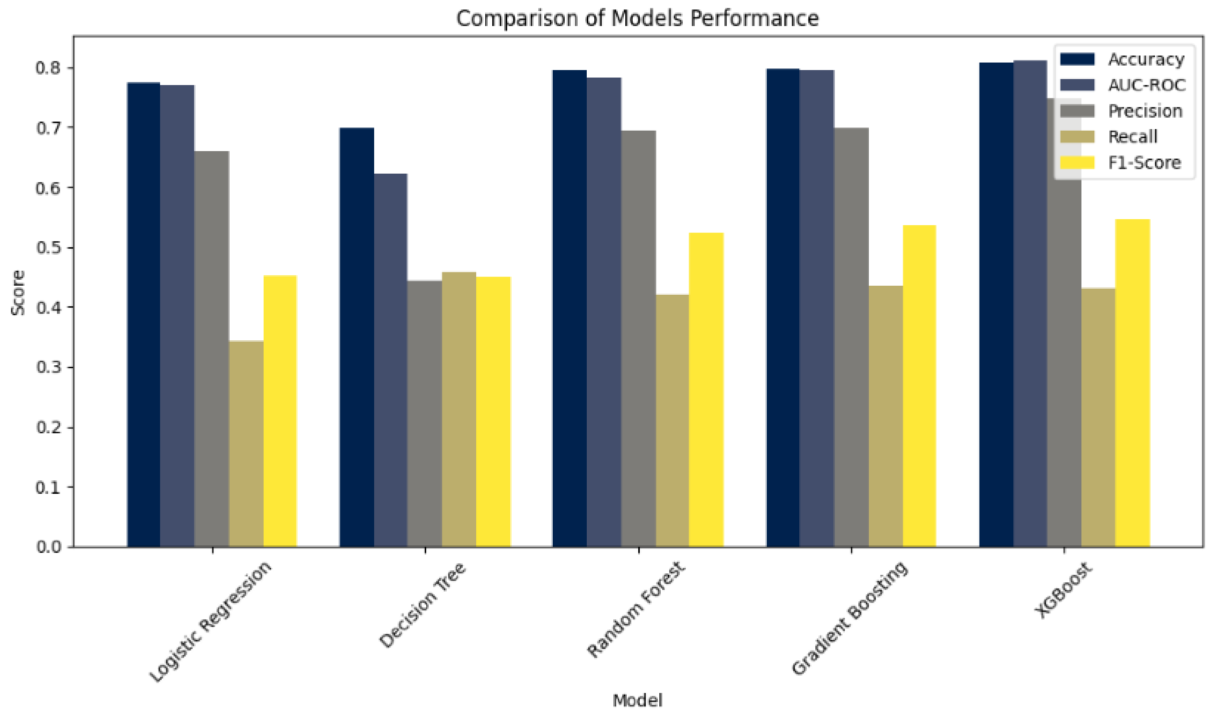
Step 5: Model Evaluation and Comparison (sklearn models and XGBoost) (Continued)...

The right-side plot visualizes the model comparison, highlighting the performance differences across various metrics. As observed from the metrics and visuals:

- Logistic Regression struggles with recall (34.4%), missing many mortality cases, but maintains decent precision (65.9%).
- Decision Tree performs the worst overall, with the lowest AUC-ROC (62.3%) and precision (44.3%).
- Random Forest and Gradient Boosting show improvements in recall (42.1% and 43.5%, respectively), meaning they identify more mortality cases.
- XGBoost performs the best, achieving the highest accuracy (80.8%) and AUC-ROC (81.1%), demonstrating a better balance of precision (74.8%) and recall (43.1%) compared to the other models.
- So overall XGBoost is best performing model.

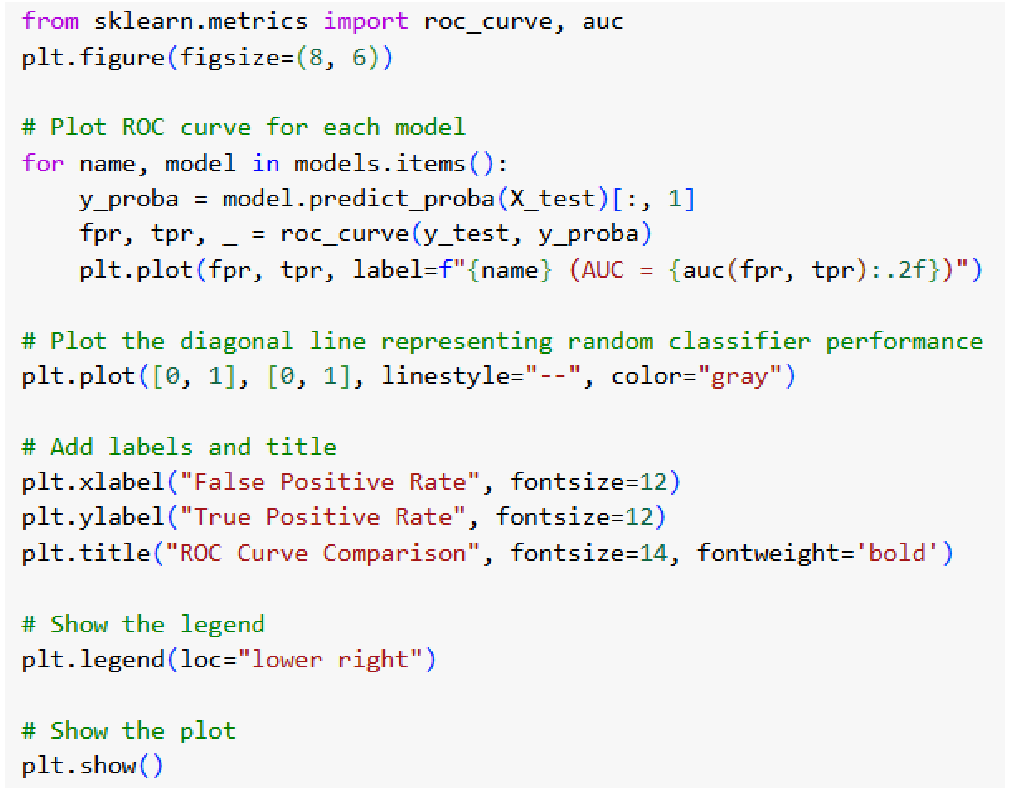


```
def plot_model_metrics_comparison(metrics):
  # Plot comparison of models in a single bar plot
  metrics.plot(kind='bar', figsize=(10, 6), colormap='cividis', width=0.8)
  plt.title('Comparison of Models Performance')
  plt.ylabel('Score')
  plt.xlabel('Model')
  plt.xticks(rotation=45)
  plt.tight_layout()
  plt.show()

plot_model_metrics_comparison(metrics_df)
```

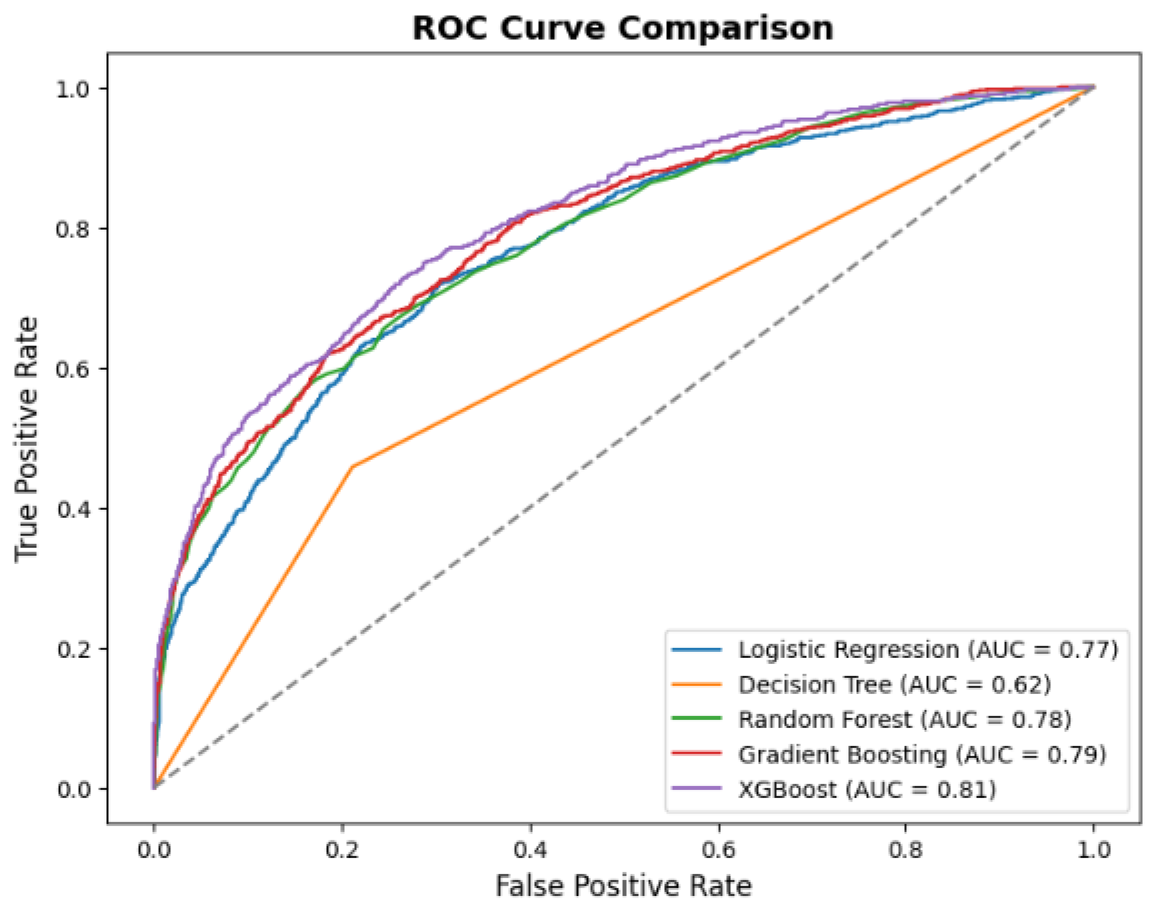


Step 5: Model Evaluation and Comparison (sklearn models and XGBoost) (Continued)...

This shows the ROC curves, which visualize the True Positive Rate (TPR) and False Positive Rate (FPR) for each classification model. By comparing the curves, we can identify which model best balances the trade-off between false positives and true positives. A higher AUC indicates better performance in distinguishing between the mortality and survival classes.

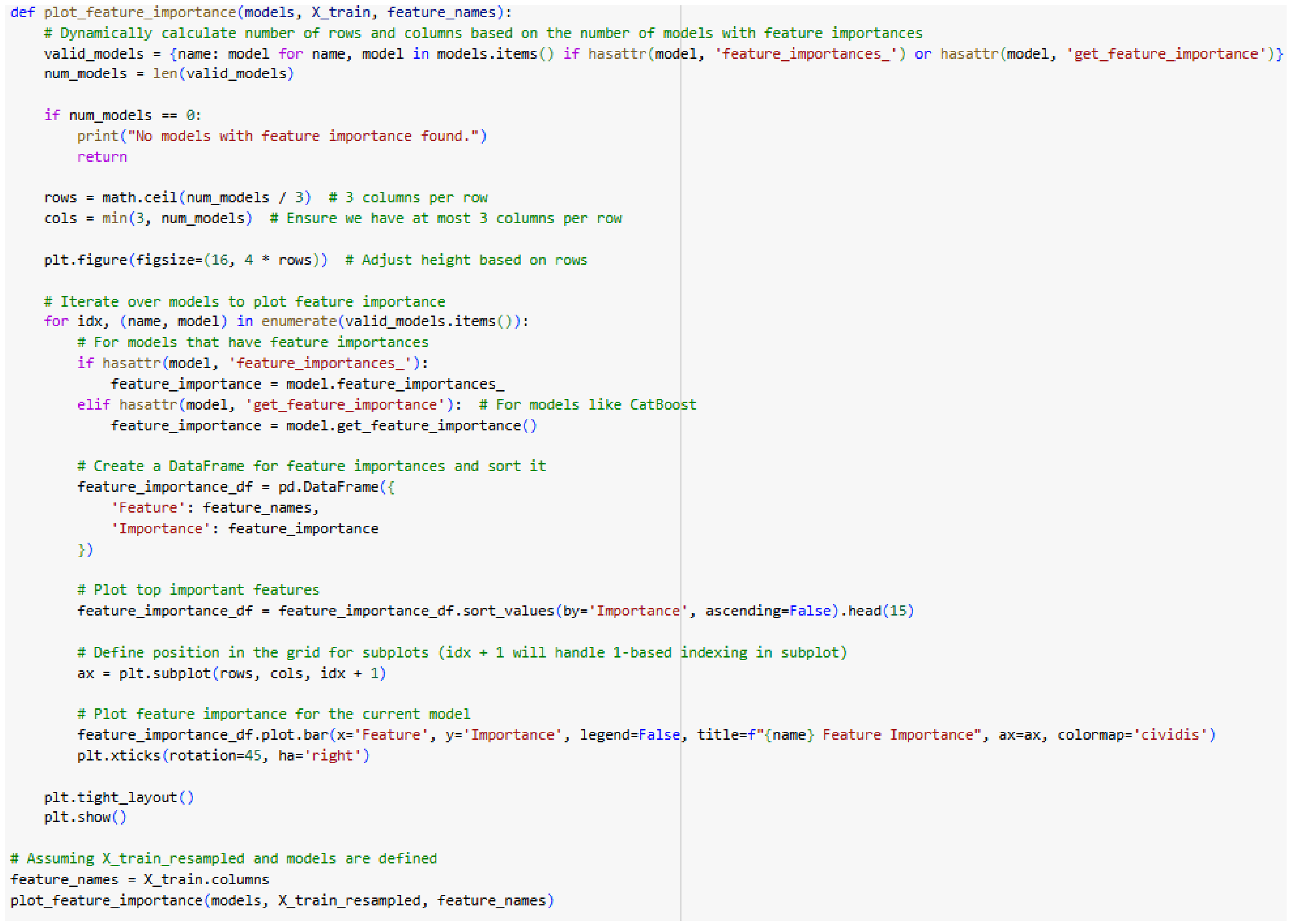


```python
from sklearn.metrics import roc_curve, auc
plt.figure(figsize=(8, 6))

# Plot ROC curve for each model
for name, model in models.items():
    y_proba = model.predict_proba(X_test)[:, 1]
    fpr, tpr, _ = roc_curve(y_test, y_proba)
    plt.plot(fpr, tpr, label=f"{name} (AUC = {auc(fpr, tpr):.2f})")

# Plot the diagonal line representing random classifier performance
plt.plot([0, 1], [0, 1], linestyle="--", color="gray")

# Add labels and title
plt.xlabel("False Positive Rate", fontsize=12)
plt.ylabel("True Positive Rate", fontsize=12)
plt.title("ROC Curve Comparison", fontsize=14, fontweight='bold')

# Show the legend
plt.legend(loc="lower right")

# Show the plot
plt.show()
```



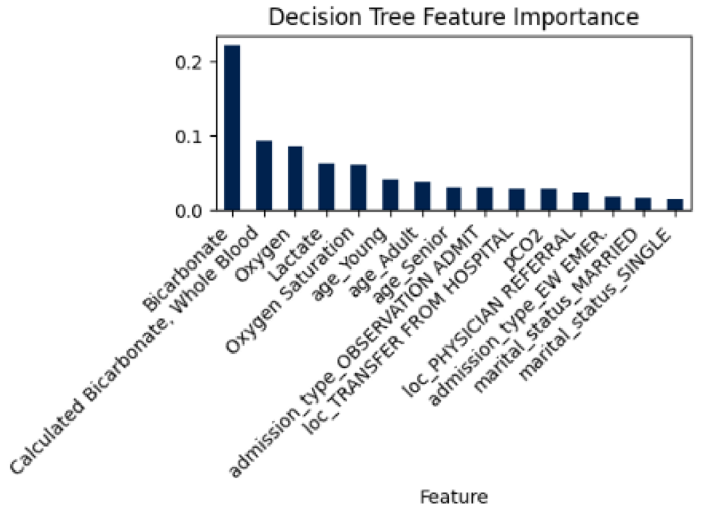
Step 5: Model Evaluation and Comparison (sklearn models and XGBoost) (Continued)...

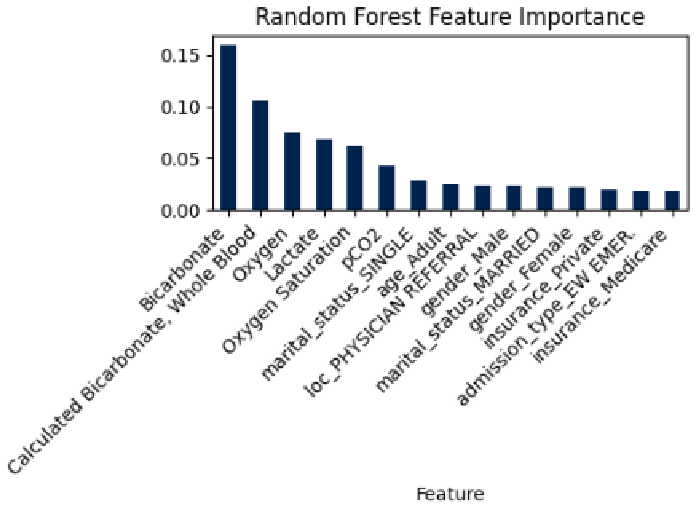
- This code visualizes the feature importance for each model, helping us understand which features have the most influence on the model's predictions.
- Code plots the top 15 most important features for each model that supports featue_importances_.

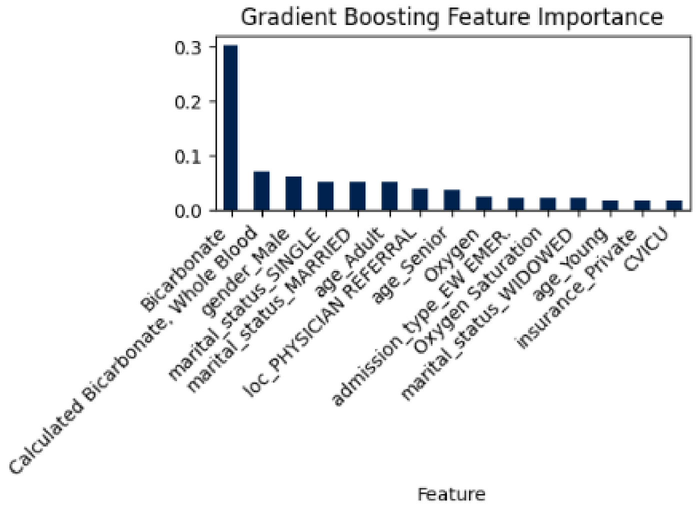


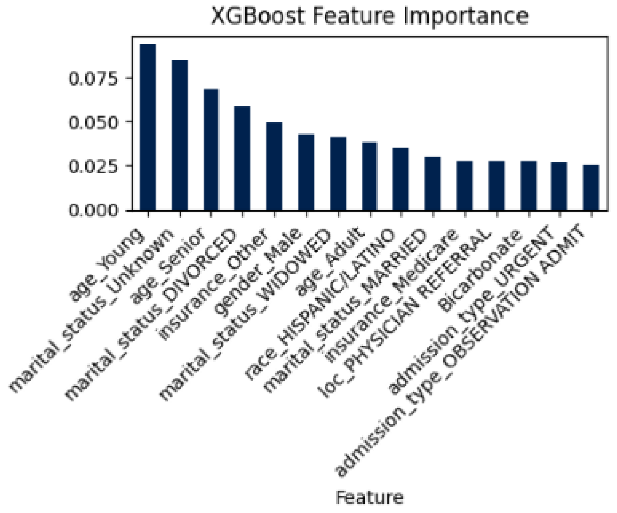
```python
def plot_feature_importance(models, X_train, feature_names):
    # Dynamically calculate number of rows and columns based on the number of models with feature importances
    valid_models = {name: model for name, model in models.items() if hasattr(model, 'feature_importances_') or hasattr(model, 'get_feature_importance')}
    num_models = len(valid_models)

    if num_models == 0:
        print("No models with feature importance found.")
        return

    rows = math.ceil(num_models / 3)  # 3 columns per row
    cols = min(3, num_models)  # Ensure we have at most 3 columns per row

    plt.figure(figsize=(16, 4 * rows))  # Adjust height based on rows

    # Iterate over models to plot feature importance
    for idx, (name, model) in enumerate(valid_models.items()):
        # For models that have feature importances
        if hasattr(model, 'feature_importances_'):
            feature_importance = model.feature_importances_
        elif hasattr(model, 'get_feature_importance'):  # For models like CatBoost
            feature_importance = model.get_feature_importance()

        # Create a DataFrame for feature importances and sort it
        feature_importance_df = pd.DataFrame({
            'Feature': feature_names,
            'Importance': feature_importance
        })

        # Plot top important features
        feature_importance_df = feature_importance_df.sort_values(by='Importance', ascending=False).head(15)

        # Define position in the grid for subplots (idx + 1 will handle 1-based indexing in subplot)
        ax = plt.subplot(rows, cols, idx + 1)

        # Plot feature importance for the current model
        feature_importance_df.plot.bar(x='Feature', y='Importance', legend=False, title=f"{name} Feature Importance", ax=ax, colormap='cividis')
        plt.xticks(rotation=45, ha='right')

    plt.tight_layout()
    plt.show()

# Assuming X_train_resampled and models are defined
feature_names = X_train.columns
plot_feature_importance(models, X_train_resampled, feature_names)
```

Step 5: Model Evaluation and Comparison (sklearn models and XGBoost) (Continued)…



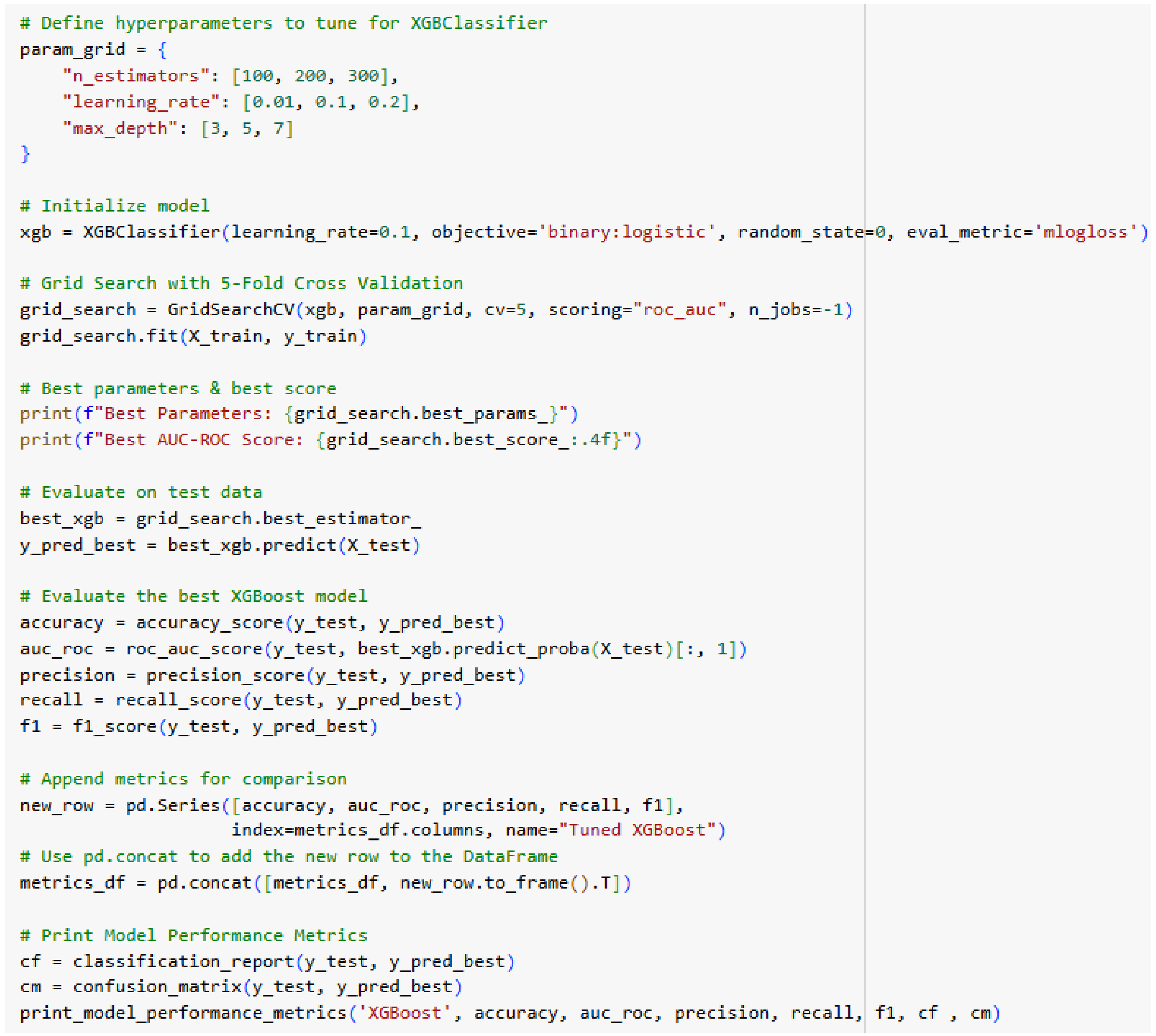




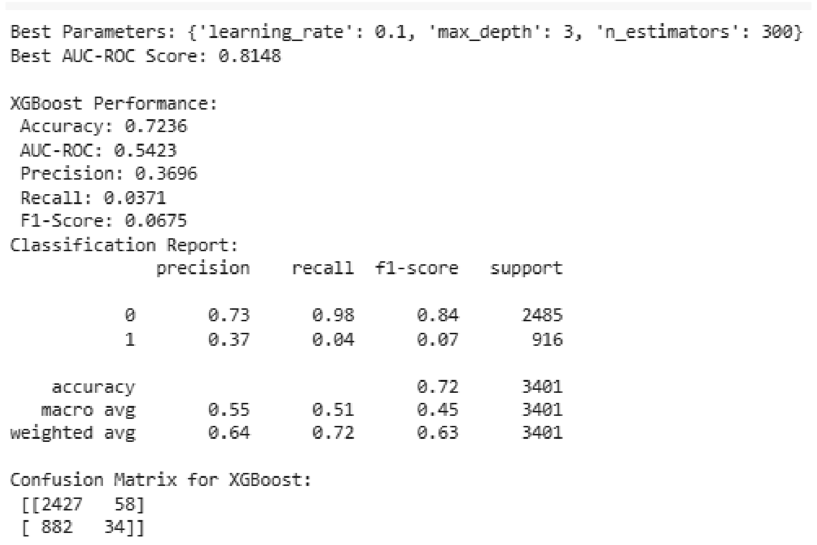


The feature importance plots strongly suggest that lab tests, particularly oxygen-related measures and bicarbonate levels, are crucial predictors across all the models.

Step 5: Model Evaluation and Comparison (sklearn models and XGBoost) (Continued)…



```python
# Define hyperparameters to tune for XGBClassifier
param_grid = {
    "n_estimators": [100, 200, 300],
    "learning_rate": [0.01, 0.1, 0.2],
    "max_depth": [3, 5, 7]
}

# Initialize model
xgb = XGBClassifier(learning_rate=0.1, objective='binary:logistic', random_state=0, eval_metric='mlogloss')

# Grid Search with 5-Fold Cross Validation
grid_search = GridSearchCV(xgb, param_grid, cv=5, scoring="roc_auc", n_jobs=-1)
grid_search.fit(X_train, y_train)

# Best parameters & best score
print(f"Best Parameters: {grid_search.best_params_}")
print(f"Best AUC-ROC Score: {grid_search.best_score_:.4f}")

# Evaluate on test data
best_xgb = grid_search.best_estimator_
y_pred_best = best_xgb.predict(X_test)

# Evaluate the best XGBoost model
accuracy = accuracy_score(y_test, y_pred_best)
auc_roc = roc_auc_score(y_test, best_xgb.predict_proba(X_test)[:, 1])
precision = precision_score(y_test, y_pred_best)
recall = recall_score(y_test, y_pred_best)
f1 = f1_score(y_test, y_pred_best)

# Append metrics for comparison
new_row = pd.Series([accuracy, auc_roc, precision, recall, f1],
                    index=metrics_df.columns, name="Tuned XGBoost")
# Use pd.concat to add the new row to the DataFrame
metrics_df = pd.concat([metrics_df, new_row.to_frame().T])

# Print Model Performance Metrics
cf = classification_report(y_test, y_pred_best)
cm = confusion_matrix(y_test, y_pred_best)
print_model_performance_metrics('XGBoost', accuracy, auc_roc, precision, recall, f1, cf , cm)
```



```
Best Parameters: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 300}
Best AUC-ROC Score: 0.8148

XGBoost Performance:
 Accuracy: 0.7236
 AUC-ROC: 0.5423
 Precision: 0.3696
 Recall: 0.0371
 F1-Score: 0.0675
Classification Report:
              precision    recall  f1-score   support

           0       0.73      0.98      0.84      2485
           1       0.37      0.04      0.07       916

    accuracy                           0.72      3401
   macro avg       0.55      0.51      0.45      3401
weighted avg       0.64      0.72      0.63      3401

Confusion Matrix for XGBoost:
 [[2427   58]
 [ 882   34]]
```

This shows hyperparameter tuning of best performing model(XGBoost) we found in earlier steps.

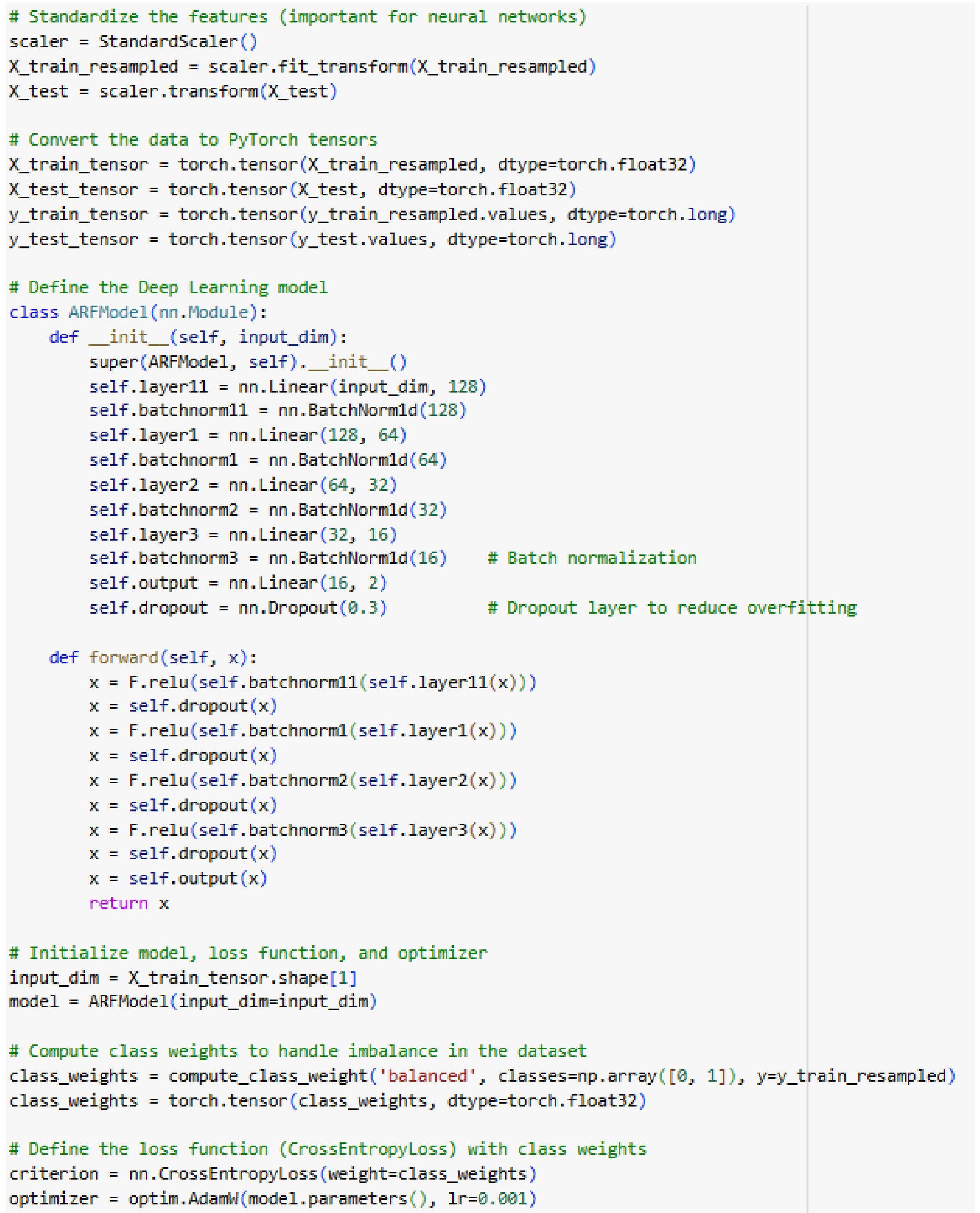
- After tuning, unfortunately model's performance worsened. The model's accuracy dropped from 80.7% to 72.3%, and more importantly, its recall for mortality cases significantly declined to 3.7%.
- This suggests that the tuned hyperparameters may have overfitted to training data or altered the balance between precision and recall, making the model less effective at identifying critical cases. So out earlier model was better.

Step 6: Evaluate Neural Network Model

Here, the Neural Network model is evaluated for predicting mortality in ARF patients.

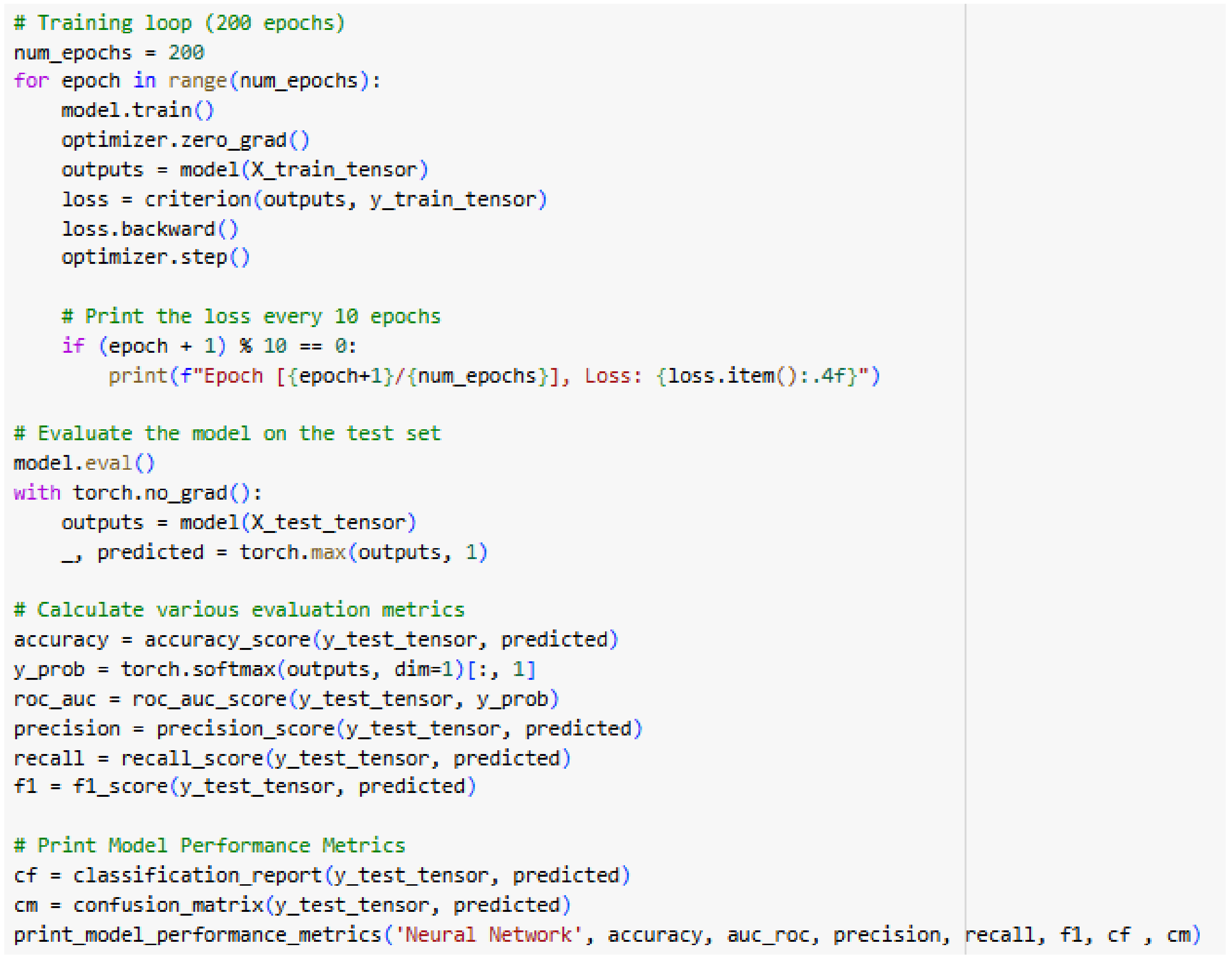
- Features are standardized to ensure that all inputs are on a similar scale, improving model stability.
- Training and test datasets are converted into PyTorch tensors for compatibility with the deep learning framework.
- Multiple fully connected layers are used to capture complex patterns.
- Batch Normalization is applied to stabilize training and improve generalization.
- Dropout layers are added to reduce overfitting.
- CrossEntropyLoss is used since the task involves classification (mortality prediction).
- AdamW optimizer is chosen for efficient weight updates.

Next, we will evaluate this model and compare its performance with earlier models.

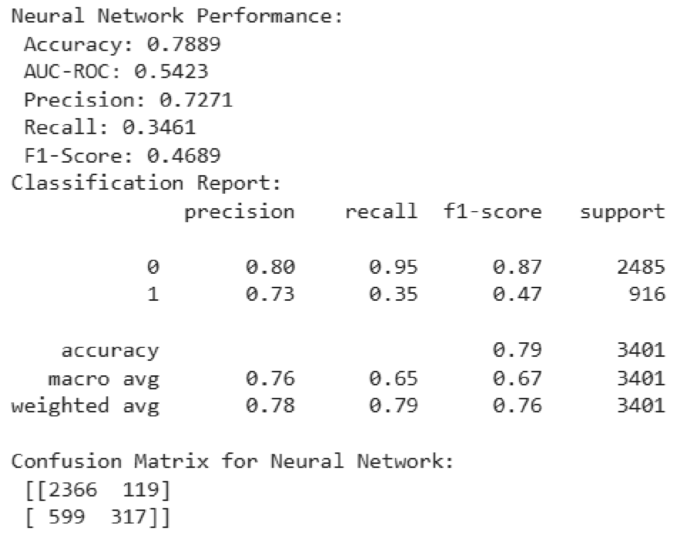


```
# Standardize the features (important for neural networks)
scaler = StandardScaler()
X_train_resampled = scaler.fit_transform(X_train_resampled)
X_test = scaler.transform(X_test)

# Convert the data to PyTorch tensors
X_train_tensor = torch.tensor(X_train_resampled, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train_resampled.values, dtype=torch.long)
y_test_tensor = torch.tensor(y_test.values, dtype=torch.long)

# Define the Deep Learning model
class ARFModel(nn.Module):
    def __init__(self, input_dim):
        super(ARFModel, self).__init__()
        self.layer11 = nn.Linear(input_dim, 128)
        self.batchnorm11 = nn.BatchNorm1d(128)
        self.layer1 = nn.Linear(128, 64)
        self.batchnorm1 = nn.BatchNorm1d(64)
        self.layer2 = nn.Linear(64, 32)
        self.batchnorm2 = nn.BatchNorm1d(32)
        self.layer3 = nn.Linear(32, 16)
        self.batchnorm3 = nn.BatchNorm1d(16)    # Batch normalization
        self.output = nn.Linear(16, 2)
        self.dropout = nn.Dropout(0.3)          # Dropout layer to reduce overfitting

    def forward(self, x):
        x = F.relu(self.batchnorm11(self.layer11(x)))
        x = self.dropout(x)
        x = F.relu(self.batchnorm1(self.layer1(x)))
        x = self.dropout(x)
        x = F.relu(self.batchnorm2(self.layer2(x)))
        x = self.dropout(x)
        x = F.relu(self.batchnorm3(self.layer3(x)))
        x = self.dropout(x)
        x = self.output(x)
        return x

# Initialize model, loss function, and optimizer
input_dim = X_train_tensor.shape[1]
model = ARFModel(input_dim=input_dim)

# Compute class weights to handle imbalance in the dataset
class_weights = compute_class_weight('balanced', classes=np.array([0, 1]), y=y_train_resampled)
class_weights = torch.tensor(class_weights, dtype=torch.float32)

# Define the loss function (CrossEntropyLoss) with class weights
criterion = nn.CrossEntropyLoss(weight=class_weights)
optimizer = optim.AdamW(model.parameters(), lr=0.001)
```

Step 6: Evaluate Neural Network Model(Continued)…



```python
# Training loop (200 epochs)
num_epochs = 200
for epoch in range(num_epochs):
    model.train()
    optimizer.zero_grad()
    outputs = model(X_train_tensor)
    loss = criterion(outputs, y_train_tensor)
    loss.backward()
    optimizer.step()

    # Print the loss every 10 epochs
    if (epoch + 1) % 10 == 0:
        print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}")

# Evaluate the model on the test set
model.eval()
with torch.no_grad():
    outputs = model(X_test_tensor)
    _, predicted = torch.max(outputs, 1)

# Calculate various evaluation metrics
accuracy = accuracy_score(y_test_tensor, predicted)
y_prob = torch.softmax(outputs, dim=1)[:, 1]
roc_auc = roc_auc_score(y_test_tensor, y_prob)
precision = precision_score(y_test_tensor, predicted)
recall = recall_score(y_test_tensor, predicted)
f1 = f1_score(y_test_tensor, predicted)

# Print Model Performance Metrics
cf = classification_report(y_test_tensor, predicted)
cm = confusion_matrix(y_test_tensor, predicted)
print_model_performance_metrics('Neural Network', accuracy, auc_roc, precision, recall, f1, cf , cm)
```



```
Neural Network Performance:
 Accuracy: 0.7889
 AUC-ROC: 0.5423
 Precision: 0.7271
 Recall: 0.3461
 F1-Score: 0.4689
Classification Report:
              precision    recall  f1-score   support

           0       0.80      0.95      0.87      2485
           1       0.73      0.35      0.47       916

    accuracy                           0.79      3401
   macro avg       0.76      0.65      0.67      3401
weighted avg       0.78      0.79      0.76      3401

Confusion Matrix for Neural Network:
 [[2366  119]
 [ 599  317]]
```

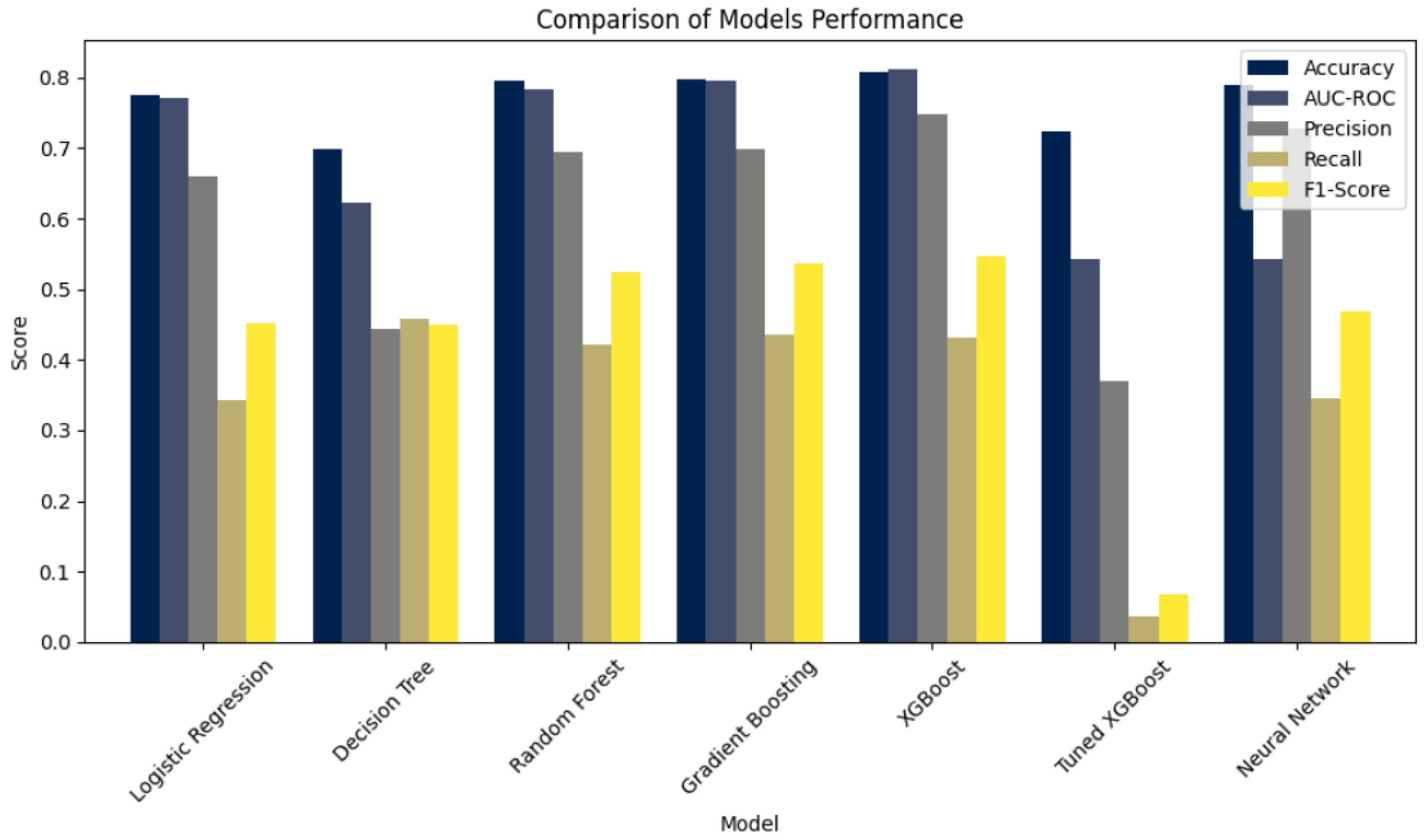
Here ARFModel is evaluated and following are observations,

The Neural Network model achieved 78.8% accuracy, but its AUC-ROC (0.5423) and recall (32.5%) indicate weak discrimination in predicting mortality. It correctly classifies survival cases well (precision: 74.1%) but struggles to identify mortality cases.
Compared to XGBoost before tuning (accuracy: 80.8%, recall: ~43%), the Neural Network underperforms in recall and overall discrimination. However, unlike XGBoost after tuning, which saw a drop in accuracy (72.4%) and recall (3.7%), the Neural Network maintains a relatively stable performance.

Step 7: Visualize all models metrics

The following plot visually compares all models, showcasing their performance across key metrics such as accuracy, AUC-ROC, precision, recall, and F1-score.

```
#Show comparision of all the models
plot_model_metrics_comparison(metrics_df)
```



Conclusion :

Among all the models evaluated, XGBoost demonstrated the best performance in predicting mortality in ARF patients. Despite hyperparameter tuning, its initial version outperformed other models, including the neural network. The feature analysis suggests that laboratory test results played a significant role in prediction, indicating their importance in assessing ARF severity and patient outcomes. Future improvements could involve further feature engineering, advanced ensemble methods, or incorporating temporal trends in lab values for better predictive accuracy.

Future improvements:

Future improvements could involve further feature engineering, additional hyperparameter tuning, or exploring different models to enhance predictive accuracy. Incorporating temporal trends in lab values and leveraging ensemble methods could also improve performance.