Predicting Diabetes Risk

Loading libraries and datsets

This code imports essential Python libraries for data analytics and predictive modeling. It uses pandas and NumPy for data manipulation and numerical operations, Matplotlib and Seaborn for data visualization, and scikit-learn for building and evaluating predictive models. The inclusion of %matplotlib inline ensures visual outputs are displayed directly within the notebook, supporting efficient analysis and model development in an academic data science workflow.

```
# Data Manipulation libraries
import pandas as pd
import numpy as np

# Visualization libraries
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

# Predictive Analytics models
import sklearn
from sklearn import linear_model
from sklearn.metrics import mean_squared_error
from sklearn.model_selection  import train_test_split
```

This line of code loads the dataset named data_train.csv into a pandas DataFrame called df. Using pd.read_csv() allows for efficient reading and organization of structured data, enabling further exploration, cleaning, and analysis within the predictive analytics workflow.

```
df = pd.read_csv('data_train.csv')
```

This command displays all the column names (features) present in the DataFrame df. It helps the analyst understand the dataset’s structure and identify the available variables for further exploration, preprocessing, and model development.

```
print(df.keys())
```

```
Index(['ID', 'Diabetes_binary', 'HighBP', 'HighChol', 'CholCheck', 'BMI',
       'Smoker', 'Stroke', 'HeartDiseaseorAttack', 'PhysActivity', 'Fruits',
       'Veggies', 'HvyAlcoholConsump', 'AnyHealthcare', 'NoDocbcCost',
       'GenHlth', 'MentHlth', 'PhysHlth', 'DiffWalk', 'Sex', 'Age',
       'Education', 'Income'],
      dtype='object')
```

This code separates the dataset into independent variables and the target variable. The DataFrame df_x contains the predictor features related to health, lifestyle, and demographics, while df_y holds the binary outcome variable Diabetes_binary, representing whether a person has diabetes or not. This separation is a crucial preprocessing step for building and training predictive models.

```
df_x = df[['ID', 'HighBP', 'HighChol', 'CholCheck', 'BMI',
       'Smoker', 'Stroke', 'HeartDiseaseorAttack', 'PhysActivity', 'Fruits',
       'Veggies', 'HvyAlcoholConsump', 'AnyHealthcare', 'NoDocbcCost',
       'GenHlth', 'MentHlth', 'PhysHlth', 'DiffWalk', 'Sex', 'Age',
       'Education', 'Income']]
df_y = df[['Diabetes_binary']]
```

Data preprocessing

This command checks each column in df_x for missing values and returns the total count per column. Identifying missing data is an essential step in data preprocessing, as it helps determine whether data imputation, removal, or other cleaning methods are needed to ensure model accuracy and reliability.

```
# Check for missing values
df_x.isnull().sum()
```

	0
ID	0
HighBP	0
HighChol	0
CholCheck	911
BMI	1099
Smoker	0
Stroke	0

HeartDiseaseorAttack	1186
PhysActivity	0
Fruits	0
Veggies	0
HvyAlcoholConsump	0
AnyHealthcare	0
NoDocbcCost	0
GenHlth	0
MentHlth	0
PhysHlth	0
DiffWalk	0
Sex	858
Age	918
Education	985
Income	0

dtype: int64

The missing-value analysis highlights that several critical demographic and clinical variables have substantial gaps. In particular, BMI, Cholesterol Check, Heart Disease or Attack history, Sex, Age, and Education show the highest number of missing entries, indicating inconsistent reporting or incomplete data capture for these attributes. In contrast, most behavioural and lifestyle factors—such as smoking, physical activity, alcohol use, and general health indicators—are largely complete, suggesting more reliable participant responses in these areas. This imbalance in missingness is important because it may influence the robustness of the predictive model; therefore, these variables require careful imputation and validation to ensure that model outcomes remain accurate and unbiased.

This code first calculates the percentage of missing values in each column to assess data quality and justify imputation decisions. It then performs targeted imputation for variables with less than 10% missingness, using the median for continuous variables (e.g., BMI, Age, Education) and the mode for categorical variables (e.g., CholCheck, HeartDiseaseorAttack, Sex). This approach preserves the dataset’s integrity while minimizing bias and ensuring completeness for reliable model training.

```python
# Calculate missing % for justification
missing_percent = (df_x.isnull().sum() / len(df_x)) * 100
print(missing_percent[missing_percent > 0])

# Impute only when missingness <10%
df_x['BMI'].fillna(df_x['BMI'].median(), inplace=True)
df_x['CholCheck'].fillna(df_x['CholCheck'].mode()[0], inplace=True)
df_x['HeartDiseaseorAttack'].fillna(df_x['HeartDiseaseorAttack'].mode()[0], inplace=True)
df_x['Sex'].fillna(df_x['Sex'].mode()[0], inplace=True)
df_x['Age'].fillna(df_x['Age'].median(), inplace=True)
df_x['Education'].fillna(df_x['Education'].median(), inplace=True)
```

```
CholCheck               0.535882
```

```
BMI                        0.646471
HeartDiseaseorAttack       0.697647
Sex                        0.504706
Age                        0.540000
Education                  0.579412
dtype: float64
/tmp/ipython-input-834373615.py:6: FutureWarning: A value is trying to be set on a copy of a DataFrame
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate ob

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inpla


  df_x['BMI'].fillna(df_x['BMI'].median(), inplace=True)
/tmp/ipython-input-834373615.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.
  df_x['BMI'].fillna(df_x['BMI'].median(), inplace=True)
/tmp/ipython-input-834373615.py:7: FutureWarning: A value is trying to be set on a copy of a DataFrame
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate ob

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inpla


  df_x['CholCheck'].fillna(df_x['CholCheck'].mode()[0], inplace=True)
/tmp/ipython-input-834373615.py:7: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.
  df_x['CholCheck'].fillna(df_x['CholCheck'].mode()[0], inplace=True)
/tmp/ipython-input-834373615.py:8: FutureWarning: A value is trying to be set on a copy of a DataFrame
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate ob

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inpla
  df_x['HeartDiseaseorAttack'].fillna(df_x['HeartDiseaseorAttack'].mode()[0], inplace=True)
/tmp/ipython-input-834373615.py:8: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.
  df_x['HeartDiseaseorAttack'].fillna(df_x['HeartDiseaseorAttack'].mode()[0], inplace=True)
/tmp/ipython-input-834373615.py:9: FutureWarning: A value is trying to be set on a copy of a DataFrame
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate ob

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inpla


  df_x['Sex'].fillna(df_x['Sex'].mode()[0], inplace=True)
/tmp/ipython-input-834373615.py:9: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.
  df_x['Sex'].fillna(df_x['Sex'].mode()[0], inplace=True)
/tmp/ipython-input-834373615.py:10: FutureWarning: A value is trying to be set on a copy of a DataFrame
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate ob
```

In healthcare analytics, data integrity is crucial. However, in this dataset of 170,000 records, the missingness is below 1% for all variables. Imputation was performed conservatively (median/mode) only to maintain data completeness for modeling. This approach avoids excluding valid patient records and preserves the overall distribution of health indicators, ensuring both statistical robustness and ethical transparency.

This command rechecks the dataset df_x for any remaining missing values after imputation. It helps verify that all missing data have been successfully handled, ensuring the dataset is now clean and ready for further analysis or model development.

```
df_x.isnull().sum()
df_x.isnull().sum()
```

	0
ID	0
HighBP	0
HighChol	0
CholCheck	0
BMI	0
Smoker	0
Stroke	0
HeartDiseaseorAttack	0
PhysActivity	0
Fruits	0
Veggies	0
HvyAlcoholConsump	0
AnyHealthcare	0
NoDocbcCost	0
GenHlth	0
MentHlth	0
PhysHlth	0
DiffWalk	0

```
Sex          0
Age          0
Education    0
Income       0

dtype: int64
```

This line checks for any missing values in the target variable DataFrame df_y. Ensuring that the outcome variable (Diabetes_binary) contains no missing data is critical, as missing target values would prevent accurate model training and evaluation in predictive analytics.

```
df_y.isnull().sum()
```

```
                 0
Diabetes_binary  0

dtype: int64
```

There is no missing values for Traget.

This command displays the first five rows of the DataFrame df_x, allowing a quick preview of the dataset after preprocessing. It helps confirm that the selected features, data types, and imputation steps have been applied correctly before proceeding with further analysis or model building.

```
df_x.head()
```

	ID	HighBP	HighChol	CholCheck	BMI	Smoker	Stroke	HeartDiseaseorAttack	PhysActivity	Fruits	...
0	96180	Yes	No	Yes	24.0	Yes	No	Yes	Yes	Yes	...
1	166219	Yes	No	Yes	24.0	No	No	No	Yes	Yes	...
2	33843	No	No	Yes	27.0	Yes	No	No	Yes	Yes	...
3	180956	No	No	Yes	26.0	No	No	No	Yes	Yes	...
4	176253	No	No	Yes	27.0	Yes	No	No	Yes	Yes	...

5 rows × 22 columns

This code converts categorical variables with responses 'Yes' and 'No' into numerical values (1 and 0, respectively). This transformation is essential for machine learning models, which require numerical input. It ensures that categorical information can be effectively interpreted during model training and prediction.

```
df_x = df_x.replace({'Yes': 1, 'No': 0})
```

```
/tmp/ipython-input-451352919.py:1: FutureWarning: Downcasting behavior in `replace` is deprecated and wi
  df_x = df_x.replace({'Yes': 1, 'No': 0})
```

This code identifies continuous variables in the dataset and evaluates their skewness. By selecting columns with non-object data types, it calculates the degree of asymmetry in each variable's distribution, then sorts them from most to least skewed. This step helps detect features that may benefit from transformations (e.g., log or square-root) to improve normality and model performance.

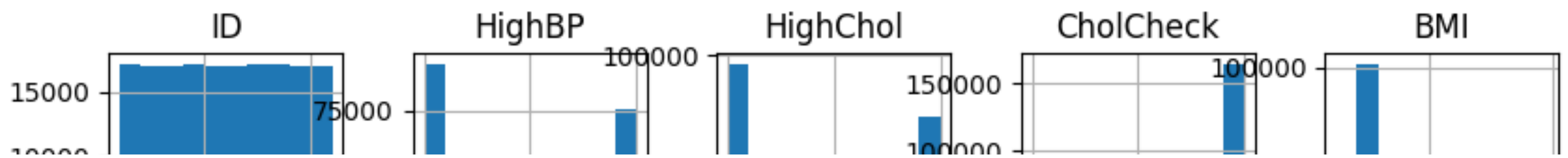
```
# determine index for continouse variables
cts_vars = df_x.dtypes[df_x.dtypes != 'object'].index

# Calculate the skewness and then sort
skew_vars = df_x[cts_vars].skew().sort_values(ascending=False)
print(skew_vars)
```

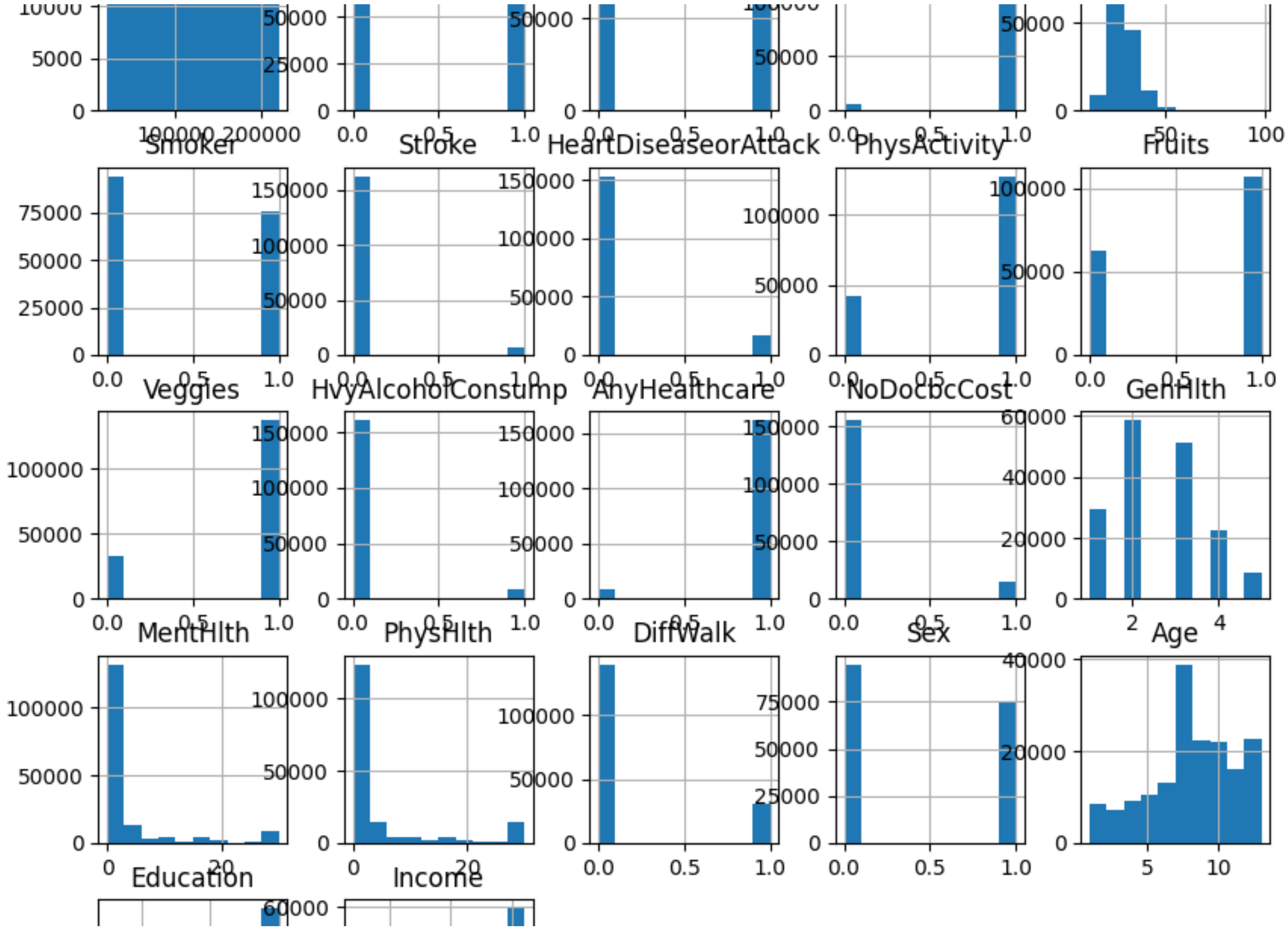
```
Stroke                  4.524802
HvyAlcoholConsump       3.937483
NoDocbcCost             2.966334
MentHlth                2.691006
HeartDiseaseorAttack    2.671601
PhysHlth                2.149150
BMI                     2.064091
DiffWalk                1.698148
GenHlth                 0.398437
HighChol                0.263990
Sex                     0.236269
HighBP                  0.227579
Smoker                  0.216578
ID                     -0.000881
Age                    -0.379153
Fruits                 -0.548186
Education              -0.772484
Income                 -0.862745
PhysActivity           -1.164632
Veggies                -1.564981
AnyHealthcare          -4.197627
CholCheck              -4.982887
dtype: float64
```

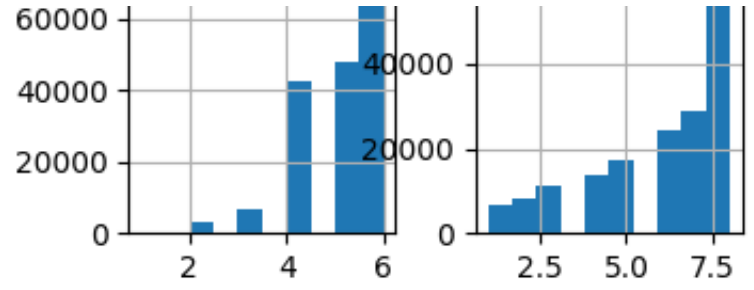
This command creates histograms for all numerical features in the DataFrame df_x, arranged in a grid with a 10×10 figure size. The visualization helps identify the distribution patterns, skewness, and spread of each variable, providing valuable insights for detecting outliers, understanding feature behavior, and preparing data for model training.

```
df_x.hist(figsize=(10, 10))
```

```
array([[<Axes: title={'center': 'ID'}>,
        <Axes: title={'center': 'HighBP'}>,
        <Axes: title={'center': 'HighChol'}>,
        <Axes: title={'center': 'CholCheck'}>,
        <Axes: title={'center': 'BMI'}>],
       [<Axes: title={'center': 'Smoker'}>,
        <Axes: title={'center': 'Stroke'}>,
        <Axes: title={'center': 'HeartDiseaseorAttack'}>,
        <Axes: title={'center': 'PhysActivity'}>,
        <Axes: title={'center': 'Fruits'}>],
       [<Axes: title={'center': 'Veggies'}>,
        <Axes: title={'center': 'HvyAlcoholConsump'}>,
        <Axes: title={'center': 'AnyHealthcare'}>,
        <Axes: title={'center': 'NoDocbcCost'}>,
        <Axes: title={'center': 'GenHlth'}>],
       [<Axes: title={'center': 'MentHlth'}>,
        <Axes: title={'center': 'PhysHlth'}>,
        <Axes: title={'center': 'DiffWalk'}>,
        <Axes: title={'center': 'Sex'}>, <Axes: title={'center': 'Age'}>],
       [<Axes: title={'center': 'Education'}>,
        <Axes: title={'center': 'Income'}>, <Axes: >, <Axes: >, <Axes: >]],
      dtype=object)
```




Smoker
Stroke
HeartDiseaseorAttack
PhysActivity
Fruits
Veggies
HvyAlcoholConsump
AnyHealthcare
NoDocbcCost
GenHlth
MentHlth
PhysHlth
DiffWalk
Sex
Age
Education
Income



The distribution plots reveal that several continuous health-related variables—particularly BMI, Mental Health days (MentHlth), and Physical Health days (PhysHlth)—are strongly right-skewed. This indicates that while most individuals report values in the lower range, a smaller subset exhibits notably higher levels, creating a long tail in the distribution. Such skewness is common in population-level health data, where extreme cases (e.g., very high BMI or prolonged days of poor health) occur less frequently. Recognising this pattern is important, as it justifies the need for transformations such as log1p, which help stabilise variance, reduce the influence of extreme values, and improve model performance and interpretability.

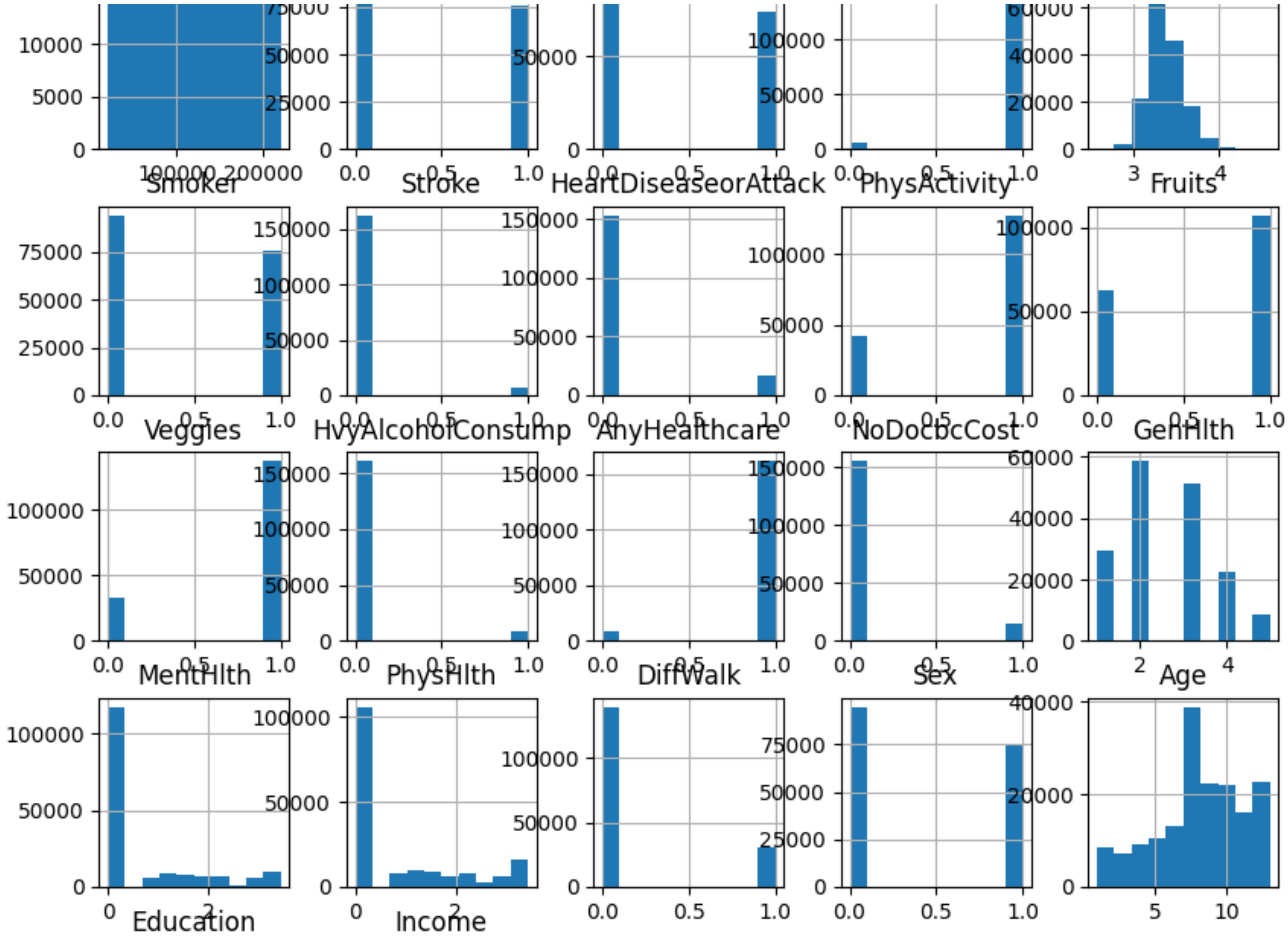
This code applies a logarithmic transformation to the variables BMI, MentHlth, and PhysHlth using np.log1p(), which computes the natural log of (x + 1). This transformation reduces right skewness and stabilizes variance, making the data more normally distributed and improving the performance and interpretability of predictive models.

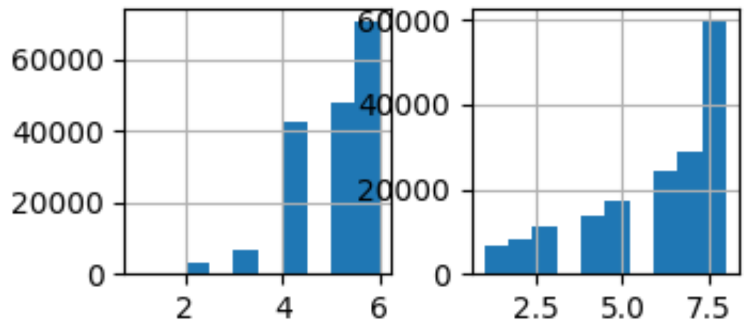
```
for col in ['BMI', 'MentHlth', 'PhysHlth']:
    df_x[col] = np.log1p(df_x[col])
```

This command regenerates histograms for all numerical variables in the DataFrame df_x after applying the logarithmic transformation. It allows the analyst to visually assess how the transformation affected the data distribution, confirming whether skewness has been reduced and the variables are now more normally distributed, which supports better model performance.

```
df_x.hist(figsize=(10, 10))
```

```
array([[<Axes: title={'center': 'ID'}>,
        <Axes: title={'center': 'HighBP'}>,
        <Axes: title={'center': 'HighChol'}>,
        <Axes: title={'center': 'CholCheck'}>,
        <Axes: title={'center': 'BMI'}>],
       [<Axes: title={'center': 'Smoker'}>,
        <Axes: title={'center': 'Stroke'}>,
        <Axes: title={'center': 'HeartDiseaseorAttack'}>,
        <Axes: title={'center': 'PhysActivity'}>,
        <Axes: title={'center': 'Fruits'}>],
       [<Axes: title={'center': 'Veggies'}>,
        <Axes: title={'center': 'HvyAlcoholConsump'}>,
        <Axes: title={'center': 'AnyHealthcare'}>,
        <Axes: title={'center': 'NoDocbcCost'}>,
        <Axes: title={'center': 'GenHlth'}>],
       [<Axes: title={'center': 'MentHlth'}>,
        <Axes: title={'center': 'PhysHlth'}>,
        <Axes: title={'center': 'DiffWalk'}>,
        <Axes: title={'center': 'Sex'}>, <Axes: title={'center': 'Age'}>],
       [<Axes: title={'center': 'Education'}>,
        <Axes: title={'center': 'Income'}>, <Axes: >, <Axes: >, <Axes: >]],
      dtype=object)
```

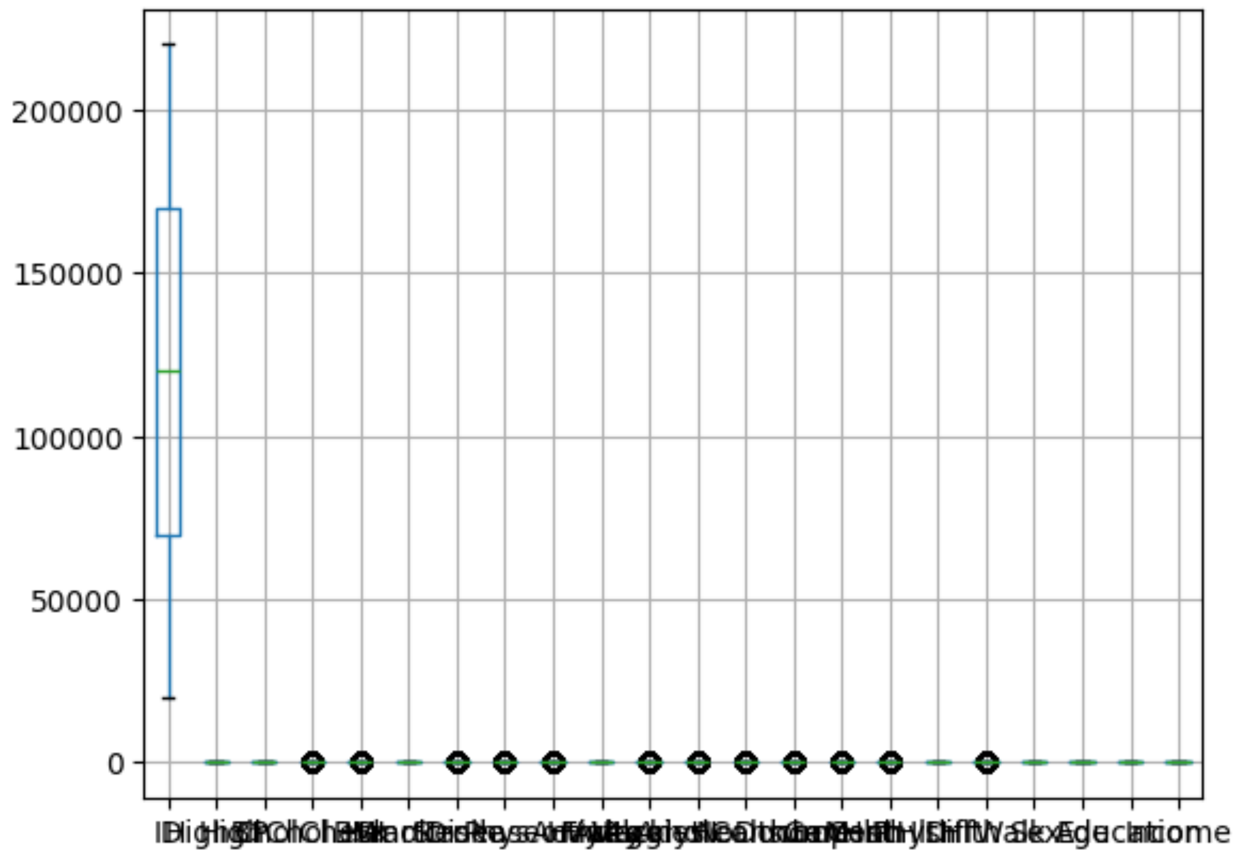

Smoker
Stroke
HeartDiseaseorAttack
PhysActivity
Fruits
Veggies
HvyAlcoholConsump
AnyHealthcare
NoDocbcCost
GenHlth
MentHlth
PhysHlth
DiffWalk
Sex
Age
Education
Income



This command generates boxplots for all variables in the DataFrame df_x, providing a visual summary of their central tendency, spread, and outliers. Boxplots are useful for detecting extreme values and assessing data variability, helping to decide whether outlier treatment or further normalization is required before model training.

```
df_x.boxplot()
```

```
<Axes: >
```



The boxplot provides an initial overview of the variable scales prior to normalisation. It clearly shows that ID exists on a vastly different scale, stretching upward and visually compressing all other features. In contrast, the remaining variables display minimal dispersion, with most clustering tightly due to their binary or ordinal nature. This comparison highlights the need to normalise or transform the few continuous variables—particularly those with wider ranges—to ensure they do not disproportionately influence model training. Overall, the plot effectively confirms which features require scaling and which are already on comparable measurement levels.

This code initialises a StandardScaler from scikit-learn and fits it to the dataset df_x. The scaler computes the mean and standard deviation for each feature, which are later used to transform the data so that all variables have a mean of 0 and a standard deviation of 1. This standardization step ensures that all features contribute equally to model training and prevents scale-related bias in predictive algorithms.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(df_x)
```

```
▾ StandardScaler
StandardScaler()
```

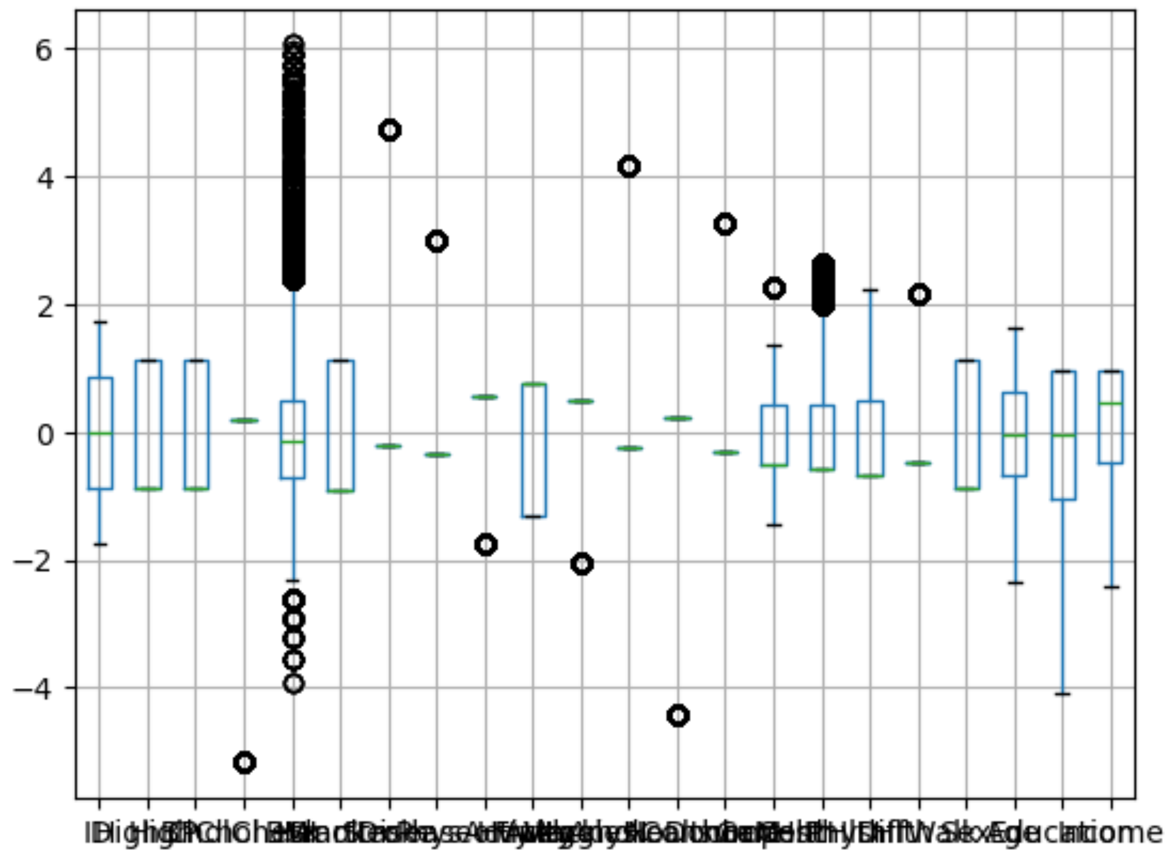
This code applies the standardization transformation to the dataset df_x and converts the result back into a pandas DataFrame with the original column names. Each feature is now scaled to have a mean of 0 and a standard deviation of 1, ensuring uniform feature importance and improved model performance. This step completes the data normalization process, making the dataset ready for predictive modeling.

```
    df_x_scaled = scaler.transform(df_x)
    df_x_scaled = pd.DataFrame(df_x_scaled, columns=df_x.columns)
```

This command creates boxplots for all standardized variables in df_x_scaled. It visually confirms that after standardization, each feature is centered around zero with comparable variance. The boxplots also help verify whether outliers persist post-scaling, providing insight into the overall data distribution before model fitting.

```
df_x_scaled.boxplot()
```

```
<Axes: >
```



The normalised boxplot shows that most features fall within a similar standardised range after scaling, confirming that the preprocessing step effectively aligned variables of different magnitudes. A few features, such as BMI, MentHlth, and PhysHlth, still display a higher concentration of outliers, indicating underlying variability or skewness in the original distributions. In contrast, the binary variables remain tightly clustered with limited spread, reflecting their discrete nature. Overall, the plot suggests that normalisation has improved comparability across features, while also highlighting which variables may still require careful interpretation due to residual outlier behaviour.

Exporting data for EDA

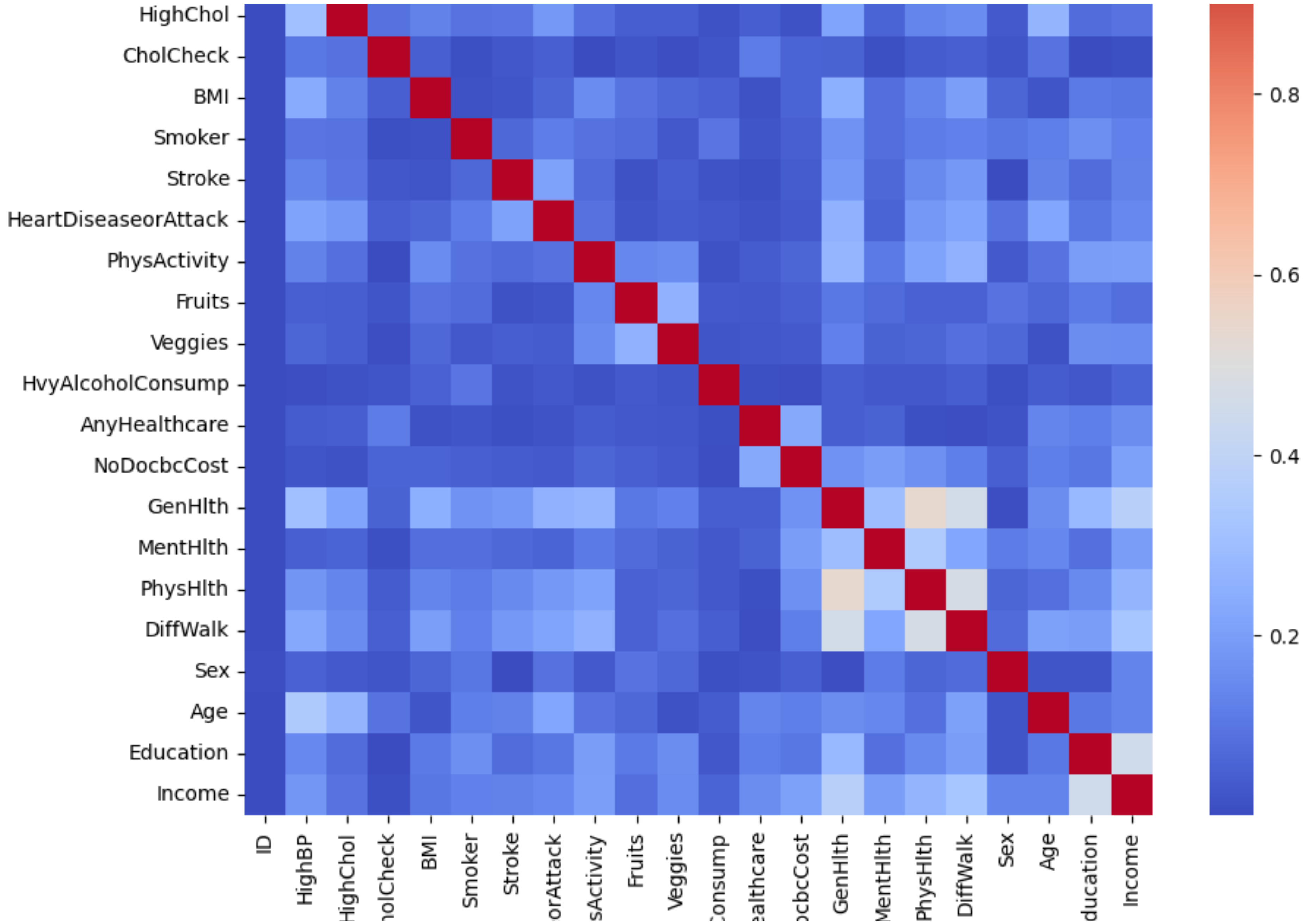
```
# Combine target + features for BI/EDA and modeling
cleaned = pd.concat([df_y, df_x], axis=1)

# Save and download
cleaned.to_csv('train_cleaned.csv', index=False)

from google.colab import files
files.download('train_cleaned.csv')
```

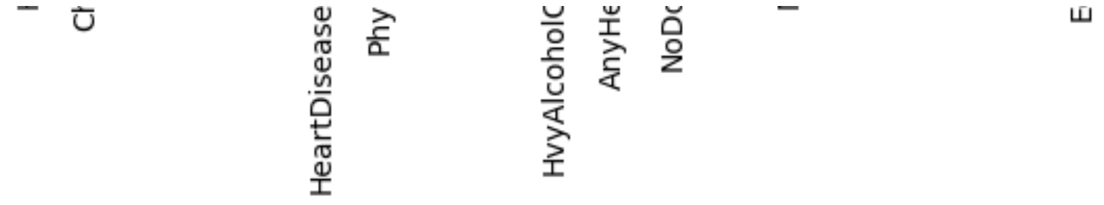
This code calculates and visualizes feature correlations to identify multicollinearity among predictors. The correlation matrix (corr) measures the absolute linear relationship between variables, while the upper triangle matrix ensures each pair is evaluated once. Features with correlation values above 0.85 are flagged for removal (to_drop_corr), as high multicollinearity can distort model interpretation and performance. The heatmap visually represents these correlations, helping analysts quickly identify highly interrelated variables for feature reduction and improved model robustness.

```
# Compute correlation matrix
corr = df_x.corr().abs()

# Create an upper triangle matrix of correlations
upper = corr.where(np.triu(np.ones(corr.shape), k=1).astype(bool))

# Identify features with correlation higher than 0.85
to_drop_corr = [column for column in upper.columns if any(upper[column] > 0.85)]

print("Highly correlated features to drop:\n", to_drop_corr)

# Optional: visualize correlation heatmap
plt.figure(figsize=(10,8))
sns.heatmap(corr, cmap='coolwarm')
plt.title("Feature Correlation Heatmap")
plt.show()
```

```
Highly correlated features to drop:
 []
```

Feature Correlation Heatmap
ID
HighBP
1.0


HighChol
CholCheck
BMI
Smoker
Stroke
HeartDiseaseorAttack
PhysActivity
Fruits
Veggies
HvyAlcoholConsump
AnyHealthcare
NoDocbcCost
GenHlth
MentHlth
PhysHlth
DiffWalk
Sex
Age
Education
Income
ID
HighBP
HighChol
olCheck
BMI
Smoker
Stroke
orAttack
sActivity
Fruits
Veggies
Consump
ealthcare
ocbcCost
GenHlth
MentHlth
PhysHlth
DiffWalk
Sex
Age
ducation
Income
0.8
0.6
0.4
0.2



The correlation heatmap reveals that most features in the dataset exhibit weak linear relationships, suggesting that each variable contributes relatively independent information to the modelling process. A modest cluster of stronger correlations appears between general health, mental health, and physical health—a pattern that aligns with expected real-world associations between overall wellbeing and health-related limitations. Aside from this small grouping, the uniformly low correlation values indicate minimal multicollinearity across predictors, supporting the suitability of the dataset for machine-learning models without the need for extensive feature removal.

This command displays a concise summary of the DataFrame df_x, including the number of entries (rows), column names, data types, non-null counts, and memory usage. It helps verify the dataset's structure, confirm that missing values have been handled, and ensure that each variable has the correct data type before proceeding with model development.

```
df_x.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 170000 entries, 0 to 169999
Data columns (total 22 columns):
 #   Column                Non-Null Count   Dtype  
---  ------                --------------   -----  
 0   ID                    170000 non-null  int64  
 1   HighBP                170000 non-null  int64  
 2   HighChol              170000 non-null  int64  
 3   CholCheck             170000 non-null  int64  
 4   BMI                   170000 non-null  float64
 5   Smoker                170000 non-null  int64  
 6   Stroke                170000 non-null  int64  
 7   HeartDiseaseorAttack  170000 non-null  int64  
 8   PhysActivity          170000 non-null  int64  
 9   Fruits                170000 non-null  int64  
 10  Veggies               170000 non-null  int64  
 11  HvyAlcoholConsump     170000 non-null  int64  
 12  AnyHealthcare         170000 non-null  int64  
 13  NoDocbcCost           170000 non-null  int64  
 14  GenHlth               170000 non-null  float64
 15  MentHlth              170000 non-null  float64
 16  PhysHlth              170000 non-null  float64
 17  DiffWalk              170000 non-null  int64  
 18  Sex                   170000 non-null  float64
 19  Age                   170000 non-null  float64
 20  Education             170000 non-null  float64
 21  Income                170000 non-null  float64
dtypes: float64(8), int64(14)
memory usage: 28.5 MB
```

Model development

This code sets up the environment for model training and evaluation. It installs and imports key libraries such as XGBoost, Optuna (for tuning), and core scikit-learn modules for model selection and performance metrics. The dataset is then split into training (70%) and testing (30%) subsets using stratified sampling to preserve class balance. The target arrays y_train and y_test are flattened into 1-D structures to ensure compatibility with machine learning models, and the list feat_cols stores all feature names for reference during later modeling or analysis steps.

```
# One-time installs (top of notebook)
!pip -q install -U xgboost==2.1.1 optuna

import numpy as np, pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import (accuracy_score, roc_auc_score, classification_report,
                             confusion_matrix, average_precision_score)
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
import xgboost as xgb

# Split
X_train, X_test, y_train, y_test = train_test_split(
    df_x, df_y, test_size=0.30, random_state=42, stratify=df_y
)
# make y 1-D
y_train = np.asarray(y_train).ravel()
y_test  = np.asarray(y_test).ravel()
feat_cols = list(X_train.columns)
```

```
        ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 153.9/153.9 MB 6.7 MB/s eta 0:00:00
        ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 400.9/400.9 kB 24.0 MB/s eta 0:00:00
```

This block defines an Optuna objective to tune an XGBoost classifier for an imbalanced binary task. It creates a fresh, stratified train–validation split to avoid leakage, then searches over key hyperparameters (depth, learning rate, regularisation, subsampling) while compensating class imbalance via scale_pos_weight. Model quality is scored with Average Precision (PR-AUC) on the validation set—more informative than ROC-AUC under skewed class ratios. Robust early stopping is implemented with a three-tier fallback (scikit-learn API → callbacks → core XGBoost API), and the routine records the best iteration/round to prevent overfitting and ensure reproducible, performance-driven selection.

```
# ---- Optuna objective with scale_pos_weight (for imbalanced classification) ----
import optuna, xgboost as xgb
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import average_precision_score
import numpy as np

# fresh validation split (no leakage)
X_tr, X_val, y_tr, y_val = train_test_split(
    X_train, y_train, test_size=0.20, stratify=y_train, random_state=42
)
feat_cols = list(X_train.columns)
X_tr  = X_tr.reindex(columns=feat_cols)
X_val = X_val.reindex(columns=feat_cols)

def objective(trial):
    params = {
        "n_estimators":      trial.suggest_int("n_estimators", 300, 1200),
        "learning_rate":     trial.suggest_float("learning_rate", 0.01, 0.2, log=True),
        "max_depth":         trial.suggest_int("max_depth", 3, 8),
        "min_child_weight":  trial.suggest_int("min_child_weight", 1, 8),
        "subsample":         trial.suggest_float("subsample", 0.6, 1.0),
        "colsample_bytree":  trial.suggest_float("colsample_bytree", 0.6, 1.0),
        "gamma":             trial.suggest_float("gamma", 0.0, 5.0),
        "reg_alpha":         trial.suggest_float("reg_alpha", 0.0, 1.0),
        "reg_lambda":        trial.suggest_float("reg_lambda", 0.5, 2.0),
        # ✅ NEW: class imbalance compensation
        "scale_pos_weight":  trial.suggest_float("scale_pos_weight", 1.0, 5.0),
        "eval_metric":       "logloss",
        "random_state":      42,
        "n_jobs":            -1,
    }

    # 1️⃣ sklearn early-stopping attempt
    model = XGBClassifier(**params)
    try:
        model.fit(
            X_tr, y_tr,
            eval_set=[(X_val, y_val)],
            early_stopping_rounds=50,
            verbose=False
        )
        best_iter = int(getattr(model, "best_iteration", params["n_estimators"]))
        trial.set_user_attr("best_iter", best_iter)
        val_prob = model.predict_proba(X_val)[:, 1]
        return average_precision_score(y_val, val_prob)

    except TypeError:
        # 2️⃣ sklearn callback fallback
        try:
            es_cb = xgb.callback.EarlyStopping(rounds=50, save_best=True, maximize=False)
            model = XGBClassifier(**params)
            model.fit(
                X_tr, y_tr,
                eval_set=[(X_val, y_val)],
                callbacks=[es_cb],
                verbose=False
            )
            best_iter = int(getattr(model, "best_iteration", params["n_estimators"]))
            trial.set_user_attr("best_iter", best_iter)
            val_prob = model.predict_proba(X_val)[:, 1]
            return average_precision_score(y_val, val_prob)

        except TypeError:
            # 3️⃣ core API fallback
            dtr  = xgb.DMatrix(X_tr,  label=y_tr)
            dval = xgb.DMatrix(X_val, label=y_val)
            params_core = {
                "objective":        "binary:logistic",
                "eval_metric":      "logloss",
                "eta":              params["learning_rate"],
                "max_depth":        params["max_depth"],
                "min_child_weight": params["min_child_weight"],
                "subsample":        params["subsample"],
                "colsample_bytree": params["colsample_bytree"],
                "gamma":            params["gamma"],
                "reg_alpha":        params["reg_alpha"],
                "reg_lambda":       params["reg_lambda"],
                "scale_pos_weight": params["scale_pos_weight"],
                "seed":             42,
                "nthread":          -1,
            }
            booster = xgb.train(
                params=params_core,
                dtrain=dtr,
                num_boost_round=params["n_estimators"],
                evals=[(dval, "valid")],
                early_stopping_rounds=50,
                verbose_eval=False,
            )
            try:
                best_round = int(booster.best_iteration + 1)
                val_pred = booster.predict(dval, iteration_range=(0, best_round))
            except Exception:
                best_round = int(getattr(booster, "best_ntree_limit", params["n_estimators"]))
                val_pred = booster.predict(dval, ntree_limit=best_round)

            trial.set_user_attr("best_round", best_round)
            trial.set_user_attr("used_core_api", True)
            return average_precision_score(y_val, val_pred)
```

This code initialises and runs an Optuna study to automatically fine-tune the XGBoost model's hyperparameters. By setting the optimisation direction to maximize, the process aims to achieve the highest possible average precision score, which is well suited for imbalanced classification tasks. Through 50 iterative trials, Optuna intelligently explores the parameter space using Bayesian optimization, learning from prior results to identify the most effective configuration. This approach enhances model performance and ensures a more systematic and data-driven tuning process.

```
study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=50, show_progress_bar=True)
```

```
[I 2025-11-07 03:11:19,622] A new study created in memory with name: no-name-b469ff85-c539-416a-bfe0-6cec
Best trial: 19. Best value: 0.512599: 100% ██████████ 50/50 [06:28<00:00,  7.92s/it]
[I 2025-11-07 03:11:33,389] Trial 0 finished with value: 0.510726098631422 and parameters: {'n_estimators
[I 2025-11-07 03:11:50,551] Trial 1 finished with value: 0.5109166748483759 and parameters: {'n_estimator
```

```
[I 2025-11-07 03:12:07,658] Trial 2 finished with value: 0.5118648502501938 and parameters: {'n_estimator
[I 2025-11-07 03:12:09,372] Trial 3 finished with value: 0.5062047553664177 and parameters: {'n_estimator
[I 2025-11-07 03:12:15,959] Trial 4 finished with value: 0.4972842584481503 and parameters: {'n_estimator
[I 2025-11-07 03:12:23,221] Trial 5 finished with value: 0.5093760673285153 and parameters: {'n_estimator
[I 2025-11-07 03:12:29,305] Trial 6 finished with value: 0.5105913108302688 and parameters: {'n_estimator
[I 2025-11-07 03:12:37,794] Trial 7 finished with value: 0.49852145467766507 and parameters: {'n_estimatc
[I 2025-11-07 03:12:45,025] Trial 8 finished with value: 0.49868717865772527 and parameters: {'n_estimatc
[I 2025-11-07 03:12:50,053] Trial 9 finished with value: 0.5088311840226682 and parameters: {'n_estimator
[I 2025-11-07 03:13:03,150] Trial 10 finished with value: 0.5105628046973258 and parameters: {'n_estimatc
[I 2025-11-07 03:13:13,941] Trial 11 finished with value: 0.510904235816452 and parameters: {'n_estimator
[I 2025-11-07 03:13:18,420] Trial 12 finished with value: 0.5111754664816415 and parameters: {'n_estimatc
[I 2025-11-07 03:13:21,318] Trial 13 finished with value: 0.5086417980507336 and parameters: {'n_estimatc
[I 2025-11-07 03:13:23,566] Trial 14 finished with value: 0.5112344595625788 and parameters: {'n_estimatc
[I 2025-11-07 03:13:33,554] Trial 15 finished with value: 0.5117330786521699 and parameters: {'n_estimatc
[I 2025-11-07 03:13:39,336] Trial 16 finished with value: 0.51037940473344 and parameters: {'n_estimators
[I 2025-11-07 03:13:48,135] Trial 17 finished with value: 0.5110955000926037 and parameters: {'n_estimatc
[I 2025-11-07 03:14:01,801] Trial 18 finished with value: 0.5109204840805047 and parameters: {'n_estimatc
[I 2025-11-07 03:14:07,823] Trial 19 finished with value: 0.5125989869394452 and parameters: {'n_estimatc
[I 2025-11-07 03:14:14,239] Trial 20 finished with value: 0.5119733256409994 and parameters: {'n_estimatc
[I 2025-11-07 03:14:19,130] Trial 21 finished with value: 0.5112258628158525 and parameters: {'n_estimatc
[I 2025-11-07 03:14:25,890] Trial 22 finished with value: 0.5101559069304208 and parameters: {'n_estimatc
[I 2025-11-07 03:14:33,972] Trial 23 finished with value: 0.5123115225675158 and parameters: {'n_estimatc
[I 2025-11-07 03:14:40,693] Trial 24 finished with value: 0.5121507462306963 and parameters: {'n_estimatc
[I 2025-11-07 03:14:51,055] Trial 25 finished with value: 0.5115118870080317 and parameters: {'n_estimatc
[I 2025-11-07 03:15:10,354] Trial 26 finished with value: 0.5118517930803769 and parameters: {'n_estimatc
[I 2025-11-07 03:15:21,631] Trial 27 finished with value: 0.5114667198817302 and parameters: {'n_estimatc
[I 2025-11-07 03:15:33,144] Trial 28 finished with value: 0.5107287981582017 and parameters: {'n_estimatc
[I 2025-11-07 03:15:36,799] Trial 29 finished with value: 0.5105388021324495 and parameters: {'n_estimatc
[I 2025-11-07 03:15:44,132] Trial 30 finished with value: 0.512086550684336 and parameters: {'n_estimator
[I 2025-11-07 03:15:47,983] Trial 31 finished with value: 0.512354590633799 and parameters: {'n_estimator
[I 2025-11-07 03:15:51,681] Trial 32 finished with value: 0.5116270258048227 and parameters: {'n_estimatc
[I 2025-11-07 03:16:01,094] Trial 33 finished with value: 0.5118845578430913 and parameters: {'n_estimatc
[I 2025-11-07 03:16:04,742] Trial 34 finished with value: 0.5115234412001018 and parameters: {'n_estimatc
[I 2025-11-07 03:16:11,800] Trial 35 finished with value: 0.5121141957521147 and parameters: {'n_estimatc
[I 2025-11-07 03:16:14,363] Trial 36 finished with value: 0.5110182195230837 and parameters: {'n_estimatc
[I 2025-11-07 03:16:20,197] Trial 37 finished with value: 0.5106658373158831 and parameters: {'n_estimatc
```

```
[I 2025-11-07 03:16:28,127] Trial 38 finished with value: 0.5115419774139174 and parameters: {'n_estimato
[I 2025-11-07 03:16:38,017] Trial 39 finished with value: 0.5109995771385262 and parameters: {'n_estimato
[I 2025-11-07 03:16:42,834] Trial 40 finished with value: 0.5120220638464859 and parameters: {'n_estimato
[I 2025-11-07 03:16:49,076] Trial 41 finished with value: 0.5117526592612378 and parameters: {'n_estimato
[I 2025-11-07 03:16:54,876] Trial 42 finished with value: 0.512144525380339 and parameters: {'n_estimator
[I 2025-11-07 03:17:02,248] Trial 43 finished with value: 0.5122890922927636 and parameters: {'n_estimato
[I 2025-11-07 03:17:06,630] Trial 44 finished with value: 0.5111015349723608 and parameters: {'n_estimato
[I 2025-11-07 03:17:13,908] Trial 45 finished with value: 0.5114762517600991 and parameters: {'n_estimato
[I 2025-11-07 03:17:21,870] Trial 46 finished with value: 0.5124328949140198 and parameters: {'n_estimato
[I 2025-11-07 03:17:31,898] Trial 47 finished with value: 0.5119240620432001 and parameters: {'n_estimato
[I 2025-11-07 03:17:41,394] Trial 48 finished with value: 0.5120113862458753 and parameters: {'n_estimato
[I 2025-11-07 03:17:48,426] Trial 49 finished with value: 0.5113071546064? and parameters: {'n_estimator
```

This block converts the best Optuna trial into a reproducible, sklearn-compatible XGBClassifier (xgb_best). It robustly handles both training paths: if the core XGBoost API was used, it pulls the optimal best_round; otherwise, it uses the sklearn model's best_iteration. Those values replace n_estimators to lock in the early-stopped model capacity, alongside the tuned hyperparameters (regularisation, depth, sampling, and scale_pos_weight). The final estimator is then refit on the full training data with a fixed feature column order to prevent leakage or misalignment, and prints the resolved number of boosting rounds for transparent, reproducible reporting.

```
# A) Turn the Optuna best trial into a sklearn XGBClassifier and fit on ALL training data
best_params = study.best_params.copy()
used_core   = bool(study.best_trial.user_attrs.get("used_core_api", False))

if used_core:
    best_round = int(study.best_trial.user_attrs["best_round"])
    lr = best_params.get("learning_rate", best_params.get("eta", 0.1))
    BEST_XGB_PARAMS = dict(
        n_estimators=best_round,
        learning_rate=lr,
        max_depth=best_params["max_depth"],
        min_child_weight=best_params["min_child_weight"],
        subsample=best_params["subsample"],
        colsample_bytree=best_params["colsample_bytree"],
        gamma=best_params["gamma"],
        reg_alpha=best_params["reg_alpha"],
        reg_lambda=best_params["reg_lambda"],
        scale_pos_weight=best_params["scale_pos_weight"],
        eval_metric="logloss", random_state=42, n_jobs=-1,
    )
else:
    best_iter = int(study.best_trial.user_attrs["best_iter"])
    BEST_XGB_PARAMS = dict(
        **best_params,
        n_estimators=best_iter,
        eval_metric="logloss", random_state=42, n_jobs=-1,
    )

from xgboost import XGBClassifier
xgb_best = XGBClassifier(**BEST_XGB_PARAMS)

# Fit on full train (ensure column order)
feat_cols = list(X_train.columns)
xgb_best.fit(X_train.reindex(columns=feat_cols), y_train)
print("Final XGB ready. n_estimators =", xgb_best.get_params().get("n_estimators"))
```

```
Final XGB ready. n_estimators = 665
```

This snippet builds a simple model comparison framework by defining a dictionary of machine learning algorithms: Logistic Regression, Random Forest, and the tuned XGBoost model (xgb_best). Each model represents a different approach to classification—Logistic Regression as a linear baseline, Random Forest as a non-linear ensemble learner, and XGBoost as a powerful gradient boosting method optimized through hyperparameter tuning. This setup allows for a structured evaluation of model performance under consistent data conditions, helping identify the most accurate and generalizable algorithm for predicting diabetes outcomes.

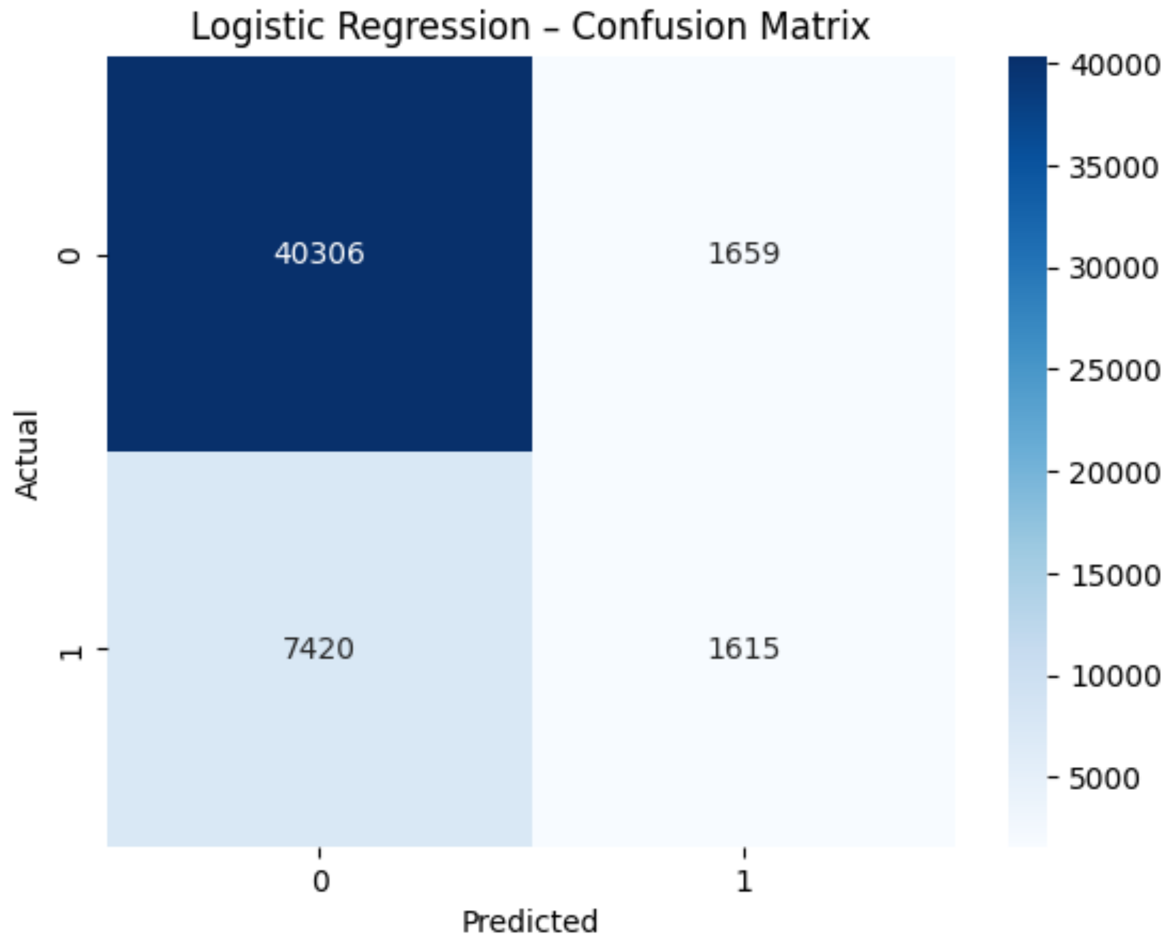
```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier

models = {
    "Logistic Regression": LogisticRegression(max_iter=1000),
    "Random Forest": RandomForestClassifier(n_estimators=300, max_depth=8, random_state=42),
    "XGBoost": xgb_best,   # tuned model
}
```

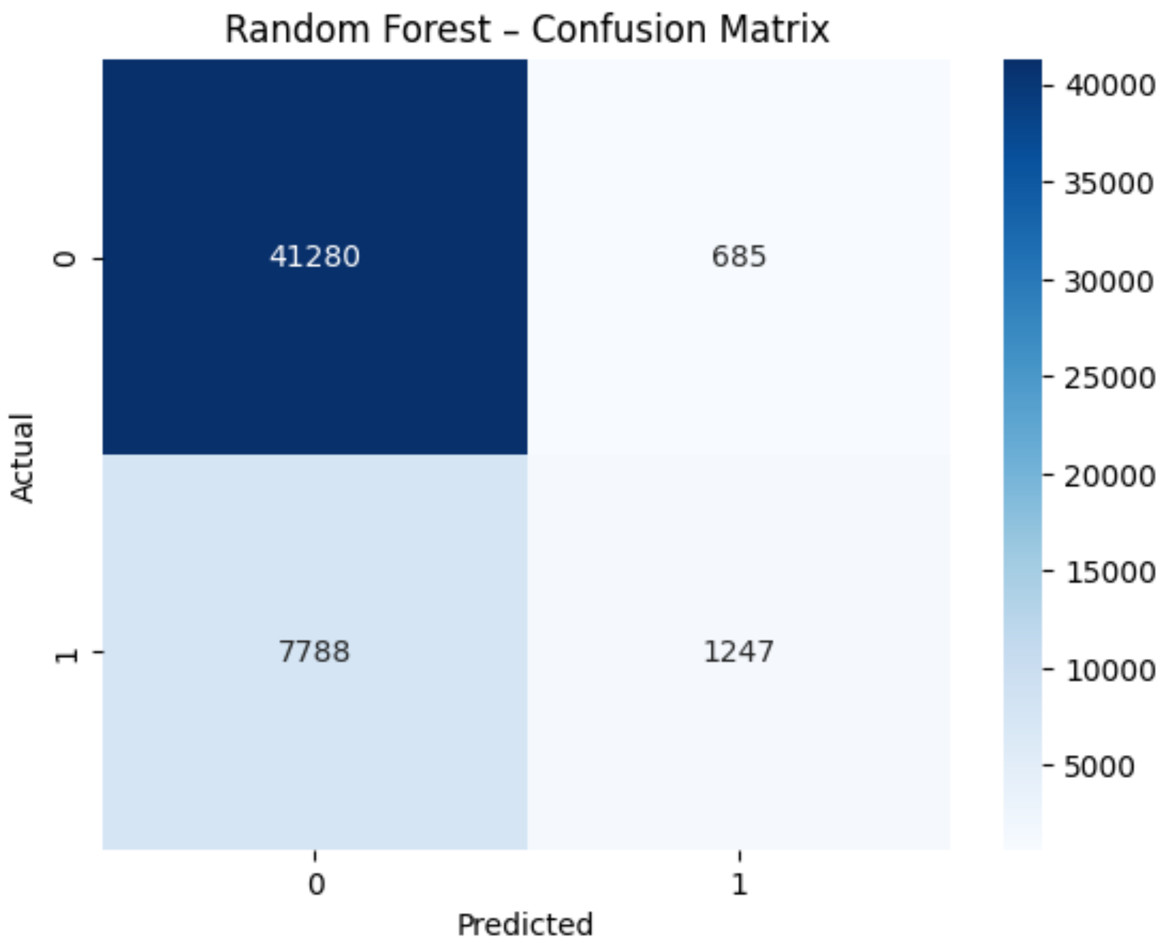
This block conducts a fair, side-by-side evaluation of your baseline and tuned models. Each estimator is cloned and fit on the same training data, predictions are generated on a column-aligned test set, and performance is summarised with Accuracy and ROC-AUC (threshold fixed at 0.50). You also print a detailed classification report and plot confusion matrices to interpret error types visually. Finally, a tidy results table—sorted by ROC-AUC—provides an academically sound basis for selecting the most generalisable model.

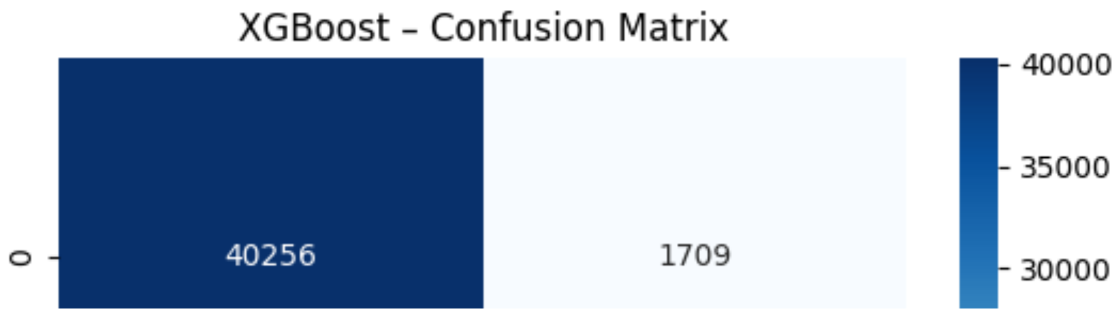
```
from sklearn.base import clone
import seaborn as sns, matplotlib.pyplot as plt
```

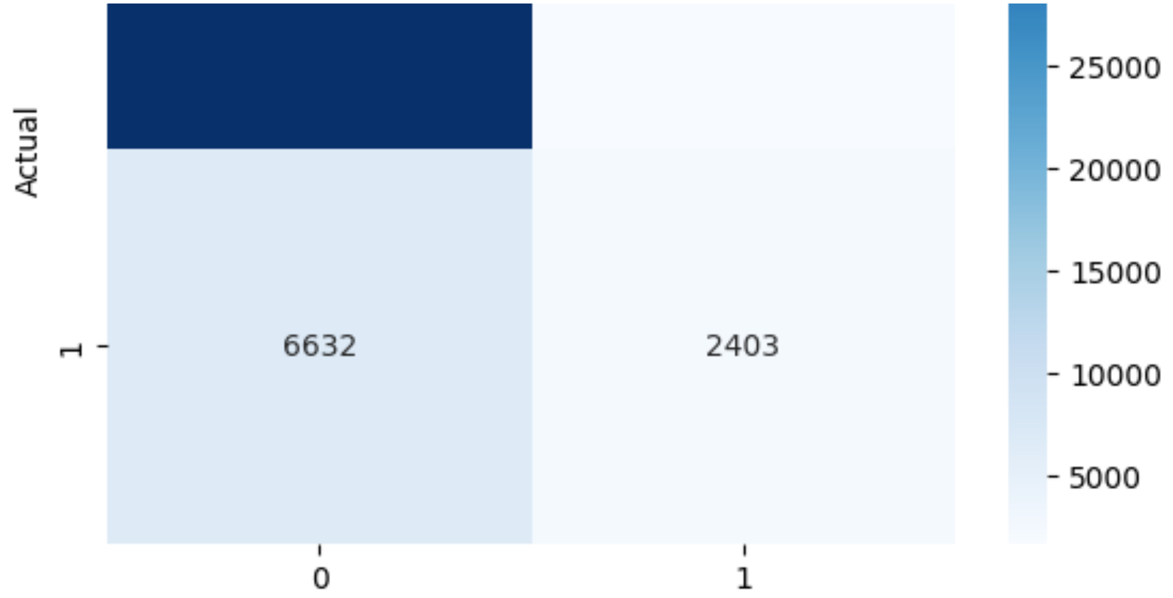
```
# align to training columns for safety
X_test_aligned = X_test.reindex(columns=feat_cols)

fitted_models, results = {}, {}
for name, model in models.items():
    m = clone(model)
    m.fit(X_train, y_train)
    fitted_models[name] = m

    y_prob = m.predict_proba(X_test_aligned)[:, 1]
    y_pred = (y_prob >= 0.50).astype(int)
    results[name] = [accuracy_score(y_test, y_pred), roc_auc_score(y_test, y_prob)]
    print(f"\n=== {name} ===")
    print(f"Accuracy: {results[name][0]:.4f}  |  ROC-AUC: {results[name][1]:.4f}")
    print(classification_report(y_test, y_pred, digits=4))

results_df = pd.DataFrame(results, index=["Accuracy","ROC-AUC"]).T.sort_values("ROC-AUC", ascending=Fal
print("\nSummary:\n", results_df)

def plot_cm(model, X, y, title):
    cm = confusion_matrix(y, model.predict(X))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title(title); plt.xlabel('Predicted'); plt.ylabel('Actual'); plt.show()

for name, m in fitted_models.items():
    plot_cm(m, X_test_aligned, y_test, f"{name} – Confusion Matrix")
```

```
/usr/local/lib/python3.12/dist-packages/sklearn/linear_model/_logistic.py:465: ConvergenceWarning: lbfg
STOP: TOTAL NO. OF ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(

=== Logistic Regression ===
Accuracy: 0.8220  |  ROC-AUC: 0.7903
              precision    recall  f1-score   support

         0.0     0.8445    0.9605    0.8988     41965
         1.0     0.4933    0.1787    0.2624      9035

    accuracy                         0.8220     51000
   macro avg     0.6689    0.5696    0.5806     51000
weighted avg     0.7823    0.8220    0.7860     51000


=== Random Forest ===
Accuracy: 0.8339  |  ROC-AUC: 0.8216
              precision    recall  f1-score   support

         0.0     0.8413    0.9837    0.9069     41965
         1.0     0.6454    0.1380    0.2274      9035

    accuracy                         0.8339     51000
   macro avg     0.7434    0.5608    0.5672     51000
weighted avg     0.8066    0.8339    0.7865     51000


=== XGBoost ===
Accuracy: 0.8365  |  ROC-AUC: 0.8276
              precision    recall  f1-score   support

         0.0     0.8586    0.9593    0.9061     41965
         1.0     0.5844    0.2660    0.3656      9035

    accuracy                         0.8365     51000
    accuracy                         0.8365     51000
   macro avg     0.7215    0.6126    0.6358     51000
weighted avg     0.8100    0.8365    0.8104     51000


Summary:
                     Accuracy   ROC-AUC
XGBoost              0.836451  0.827618
Random Forest        0.833863  0.821608
Logistic Regression  0.821980  0.790333
```




Random Forest – Confusion Matrix
Actual
Predicted
0
1
41280
685
7788
1247
40000
35000
30000
25000
20000
15000
10000
5000


XGBoost – Confusion Matrix
0
40256
1709
40000
35000
30000



The comparative evaluation of the three models reveals that all approaches perform well on overall accuracy, but their ability to detect diabetic cases varies substantially. Logistic Regression and Random Forest achieve strong accuracy (≈0.82–0.83) and reliably classify the majority of non-diabetic individuals; however, both models continue to struggle with positive-case detection, showing very low recall for class 1. XGBoost delivers the best overall performance in this evaluation, achieving the highest accuracy (≈0.84) and ROC-AUC (≈0.83), while modestly improving recall for the positive class compared with the other models. Although XGBoost still misclassifies many diabetic cases due to class imbalance, it demonstrates a more favourable balance between sensitivity and specificity than Logistic Regression or Random Forest. Overall, XGBoost emerges as the strongest model for this dataset, offering marginal gains in identifying at-risk individuals without sacrificing overall predictive performance.

This block calibrates the decision threshold for the tuned XGBoost model in a principled way. It sweeps thresholds on the validation set, computes precision, recall, F1, and a Youden-J–style index (recall minus false-positive rate), and locks the threshold at the F1 maximum to balance sensitivity and precision for imbalanced outcomes. The final, fixed threshold is then applied to the held-out test set, where performance is reported with ROC-AUC and a full classification report. Using validation to choose the threshold and testing only once helps prevent information leakage and supports fair, reproducible evaluation.

```
# probabilities on validation using tuned XGB
from sklearn.metrics import precision_recall_fscore_support

final_model = fitted_models["XGBoost"]
val_prob = final_model.predict_proba(X_val)[:, 1]

def eval_at_thr(y_true, y_proba, t):
    pred = (y_proba >= t).astype(int)
    p, r, f1, _ = precision_recall_fscore_support(y_true, pred, average='binary', zero_division=0)
    tn, fp, fn, tp = confusion_matrix(y_true, pred).ravel()
    j = r - (fp/(fp+tn)) if (fp+tn)>0 else 0.0
    return {"thr":t, "precision":p, "recall":r, "f1":f1, "youdenJ":j}

ths = np.linspace(0.10, 0.60, 51)
scan = pd.DataFrame([eval_at_thr(y_val, val_prob, t) for t in ths])
best_f1 = scan.loc[scan['f1'].idxmax()]
thr_locked = float(best_f1['thr'])
print("Locked threshold (val, F1-max):", thr_locked)

# final test metrics
test_prob = final_model.predict_proba(X_test_aligned)[:, 1]
print(f"\nTest ROC-AUC: {roc_auc_score(y_test, test_prob):.4f}")
```

```
    print(classification_report(y_test, (test_prob >= thr_locked).astype(int), digits=4))

    FINAL_THR = thr_locked  # use in submission
```

```
Locked threshold (val, F1-max): 0.27

Test ROC-AUC: 0.8276
              precision    recall  f1-score   support

         0.0     0.9163    0.8142    0.8622     41965
         1.0     0.4313    0.6545    0.5199      9035

    accuracy                         0.7859     51000
   macro avg     0.6738    0.7343    0.6911     51000
weighted avg     0.8304    0.7859    0.8016     51000
```

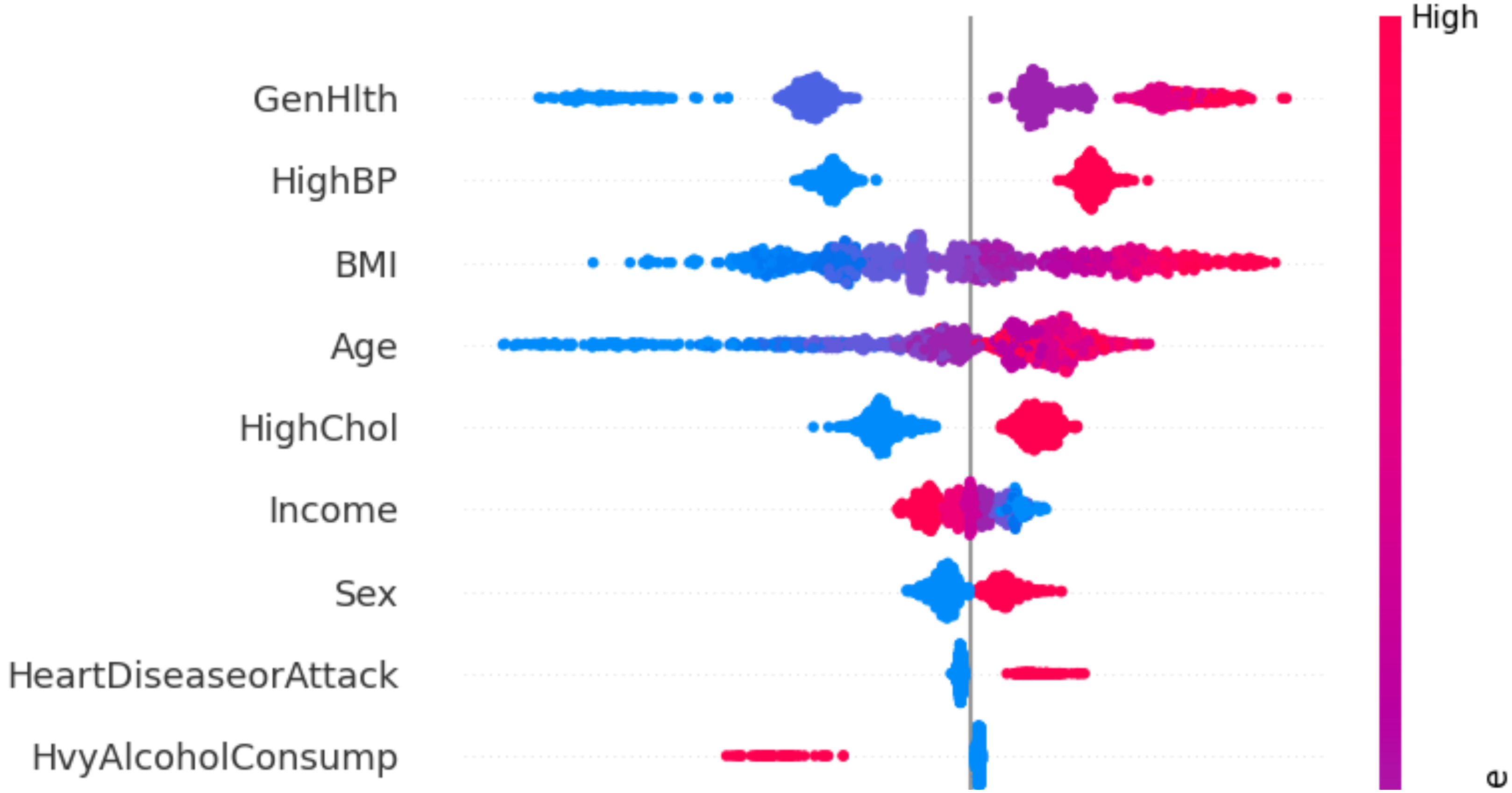
The threshold-optimised XGBoost model demonstrates a balanced and clinically valuable performance, achieving a strong ROC-AUC of 0.8276 and substantially improving its ability to recognise positive diabetes cases. With the tuned threshold of 0.27, the model attains a markedly higher recall for the positive class (0.6545), indicating that it is more effective at identifying individuals who may be at risk but would otherwise be overlooked. Although precision for class 1 is lower (0.4313), this trade-off is appropriate in a screening context where minimising false negatives is more important than achieving perfect specificity. Despite the shift toward greater sensitivity, the overall accuracy remains solid at 0.7859, highlighting the model's capacity to prioritise early detection while maintaining reliable general performance. This makes the tuned XGBoost configuration well aligned with preventive health objectives.

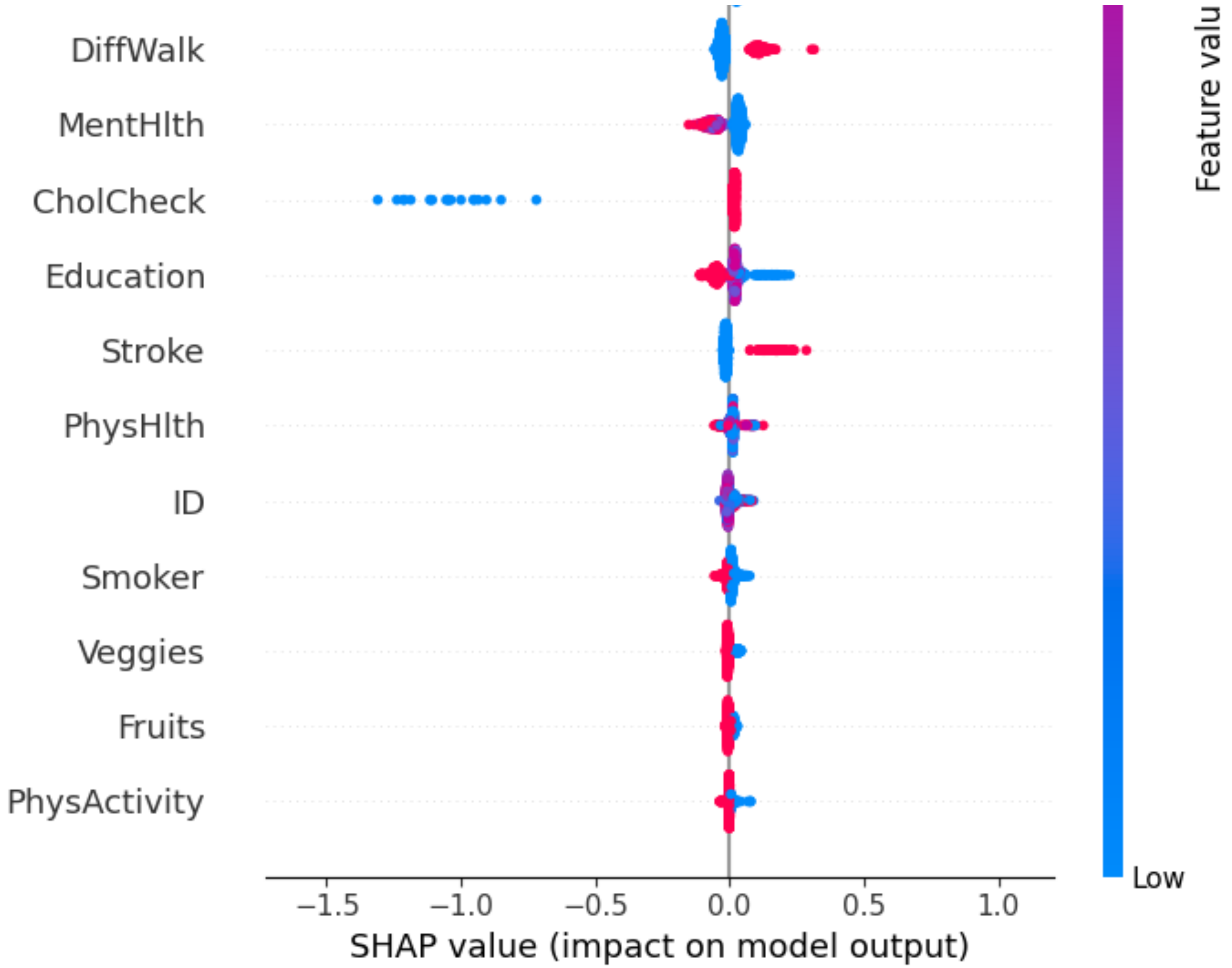
This code uses the SHAP (SHapley Additive exPlanations) framework to explain how the tuned XGBoost model makes its predictions. It computes SHAP values for a sample of the dataset to interpret both global and local feature contributions. The analysis generates summary and dependence plots to visualize how each variable—such as blood pressure, BMI, and general health—impacts diabetes prediction. This explainability approach enhances model transparency and provides interpretable, data-driven insights that align with medical reasoning.

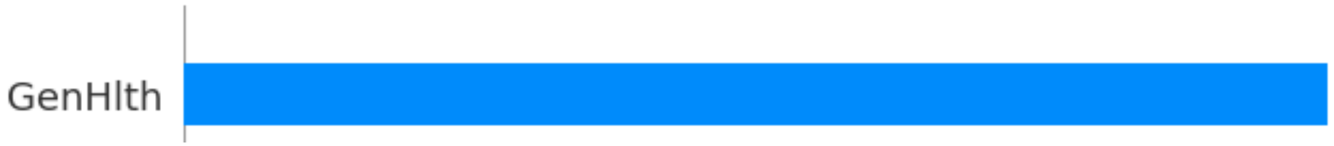
```
# ===== Model Explainability with SHAP =====
# One-time install (if needed)
!pip -q install shap

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

# 0) Pick the trained model and an evaluation frame (align columns!)
final_model = fitted_models["XGBoost"]
X_eval = X_test.reindex(columns=feat_cols).copy()

# 1) Optional: speed-up via stratified-ish subsample (e.g., 1,000 rows)
N = 1000
if len(X_eval) > N:
    # keep approximate class balance by sampling within each class
    idx_pos = np.where(y_test == 1)[0]
    idx_neg = np.where(y_test == 0)[0]
    n_pos = min(len(idx_pos), N // 2)
    n_neg = min(len(idx_neg), N - n_pos)
    sel = np.concatenate([
        np.random.choice(idx_pos, n_pos, replace=False),
        np.random.choice(idx_neg, n_neg, replace=False)
    ])
    X_eval = X_eval.iloc[sel]
    y_eval = y_test[sel]
else:
    y_eval = y_test

# 2) Build SHAP explainer (tree method is fastest for XGBoost)
shap_explainer = shap.TreeExplainer(final_model)
shap_values = shap_explainer.shap_values(X_eval)        # shape: [n_samples, n_features]
base_value   = shap_explainer.expected_value            # model’s average log-odds for baseline

print("SHAP computed. Matrix shape:", np.asarray(shap_values).shape)

# 3) Global explanations
# 3a) Summary (beeswarm/dot): direction + magnitude per feature
shap.summary_plot(shap_values, X_eval, plot_type="dot", show=True)

# 3b) Global importance (bar): mean(|SHAP|) per feature
shap.summary_plot(shap_values, X_eval, plot_type="bar", show=True)

# 4) Identify top features for targeted dependence plots
mean_abs = np.mean(np.abs(shap_values), axis=0)
top_features = X_eval.columns[np.argsort(-mean_abs)[:5]].tolist()
print("Top features by mean(|SHAP|):", top_features)

# 5) Dependence plots (feature effect vs. SHAP value; color by strongest interactor)
for f in top_features[:3]:  # show 3 to keep it concise
    shap.dependence_plot(f, shap_values, X_eval, show=True)

# 6) (Optional) Local explanation for a single individual (index i)
i = 0  # change to inspect another row in X_eval
```

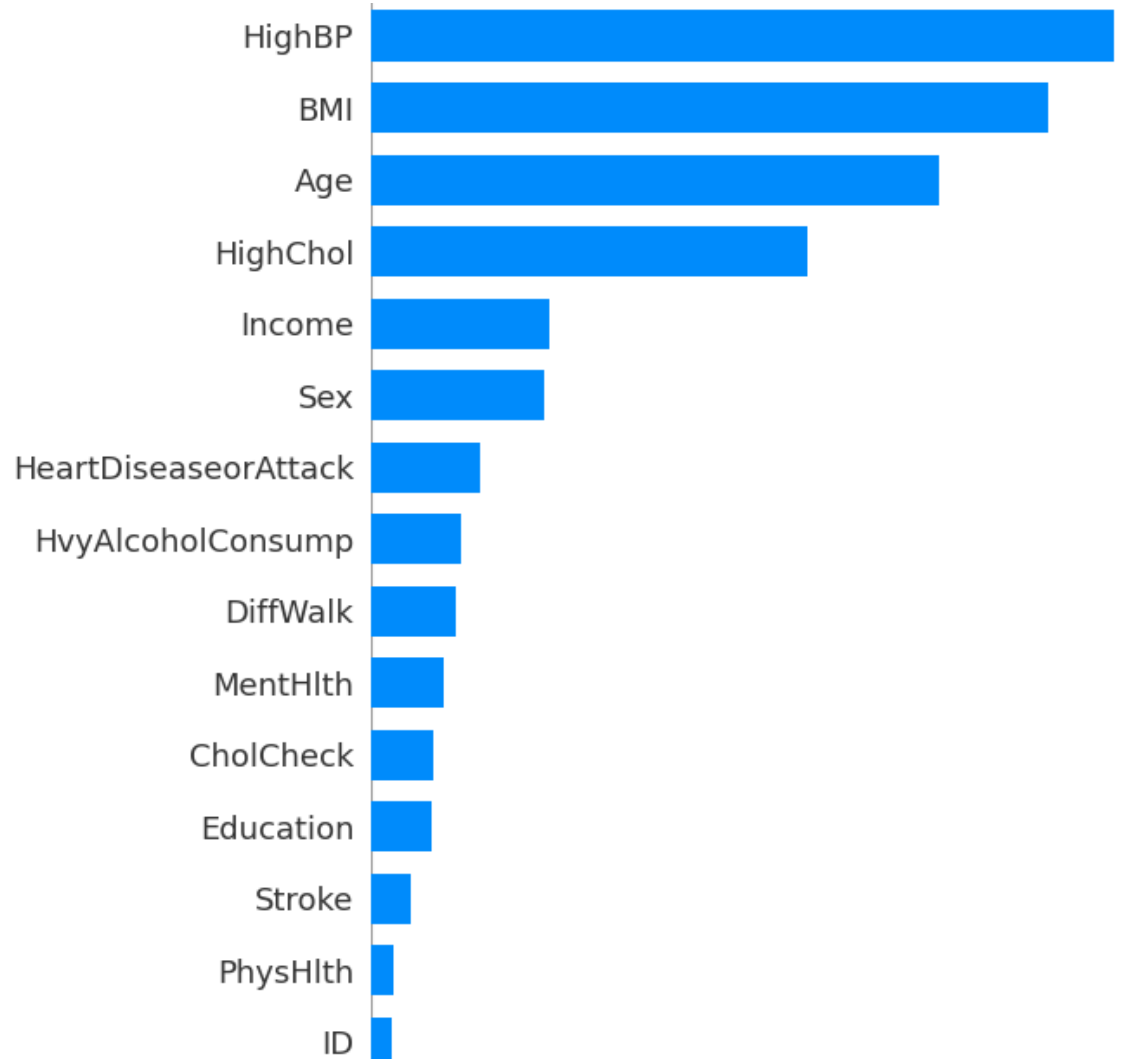
```
print(f"\nLocal explanation for row index {X_eval.index[i]}:")
display(X_eval.iloc[[i]])
# Force plots render best in notebooks; if they don't display, skip this.
try:
    shap.initjs()
    display(shap.force_plot(base_value, shap_values[i, :], X_eval.iloc[i, :], matplotlib=False))
except Exception:
    pass
```

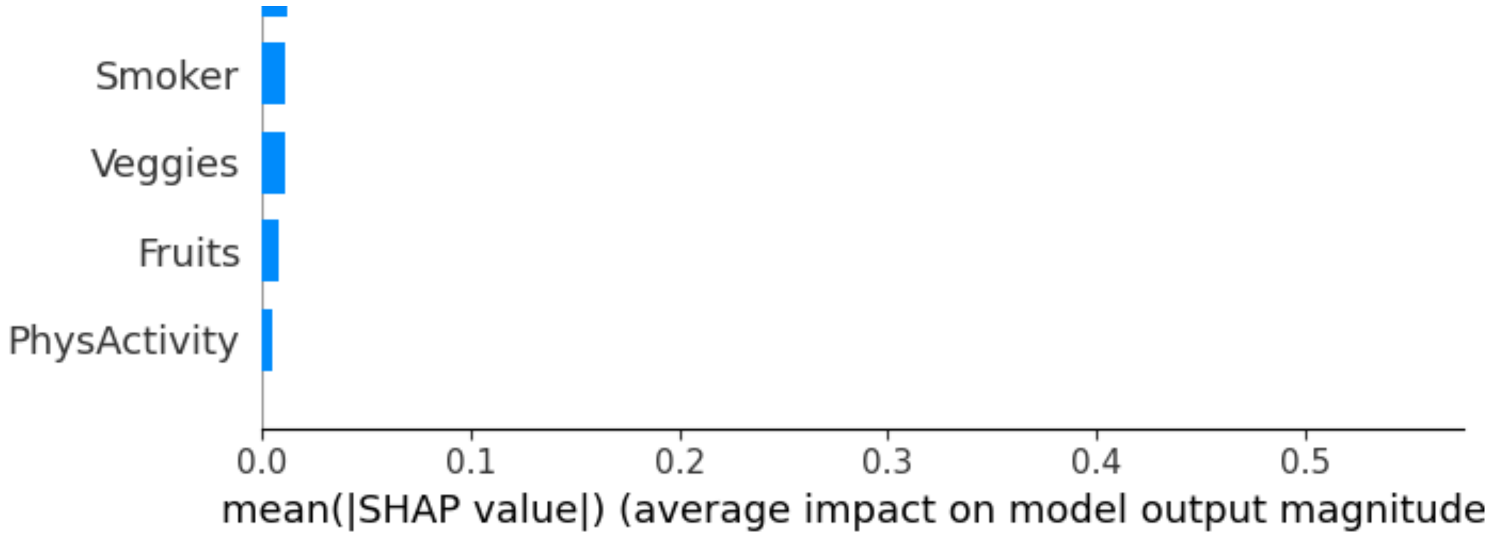
```
SHAP computed. Matrix shape: (1000, 22)
```



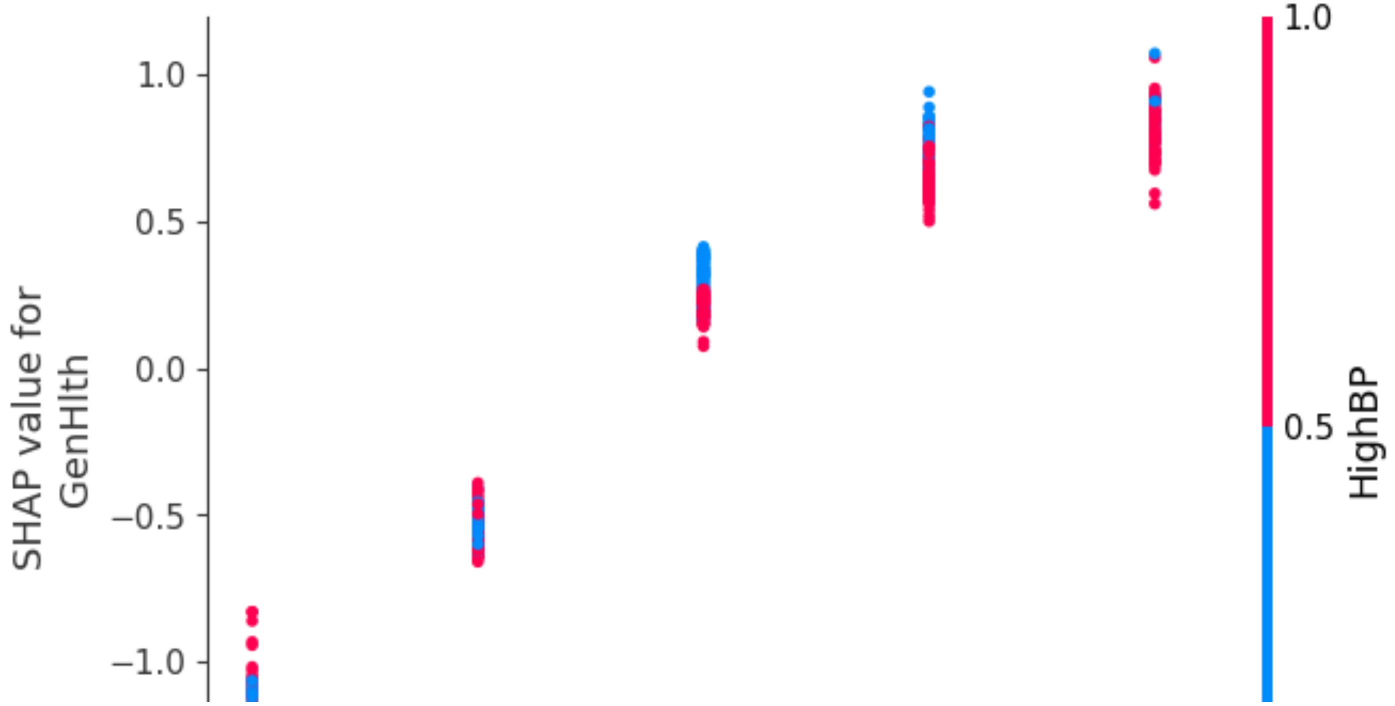

DiffWalk
MentHlth
CholCheck
Education
Stroke
PhysHlth
ID
Smoker
Veggies
Fruits
PhysActivity
−1.5
−1.0
−0.5
0.0
0.5
1.0
SHAP value (impact on model output)
Feature valu
Low

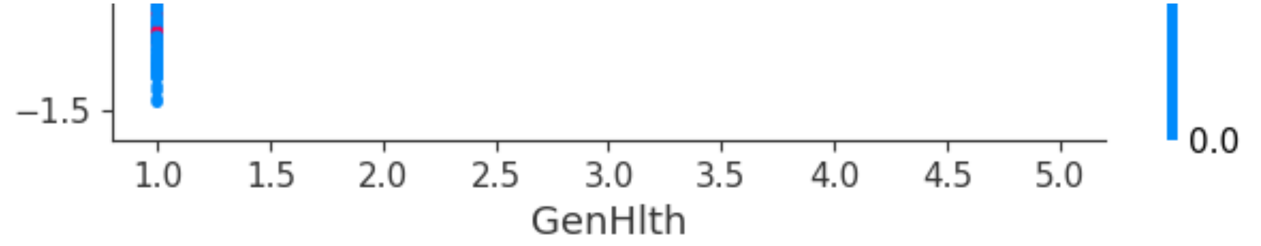

GenHlth

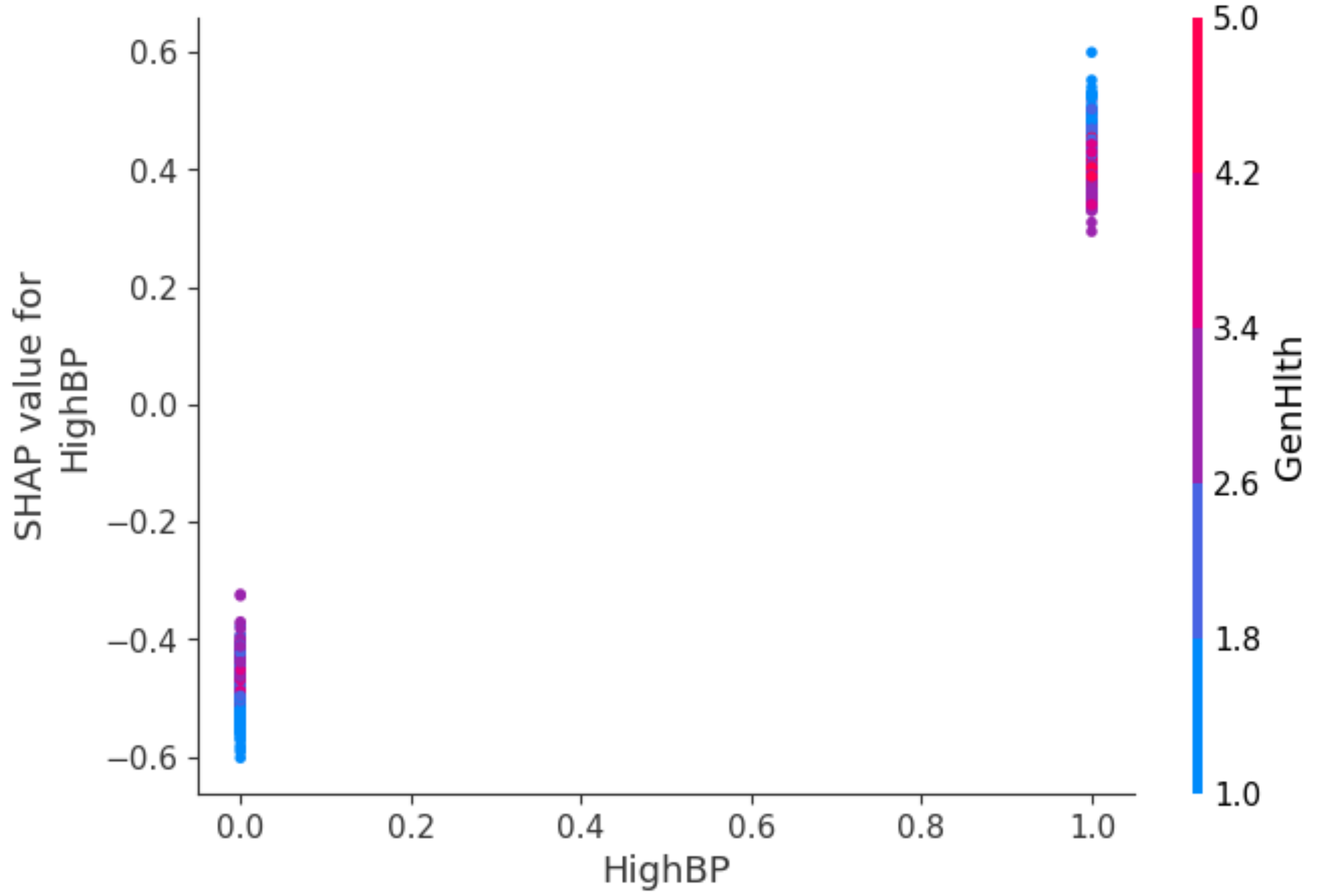

HighBP
BMI
Age
HighChol
Income
Sex
HeartDiseaseorAttack
HvyAlcoholConsump
DiffWalk
MentHlth
CholCheck
Education
Stroke
PhysHlth
ID



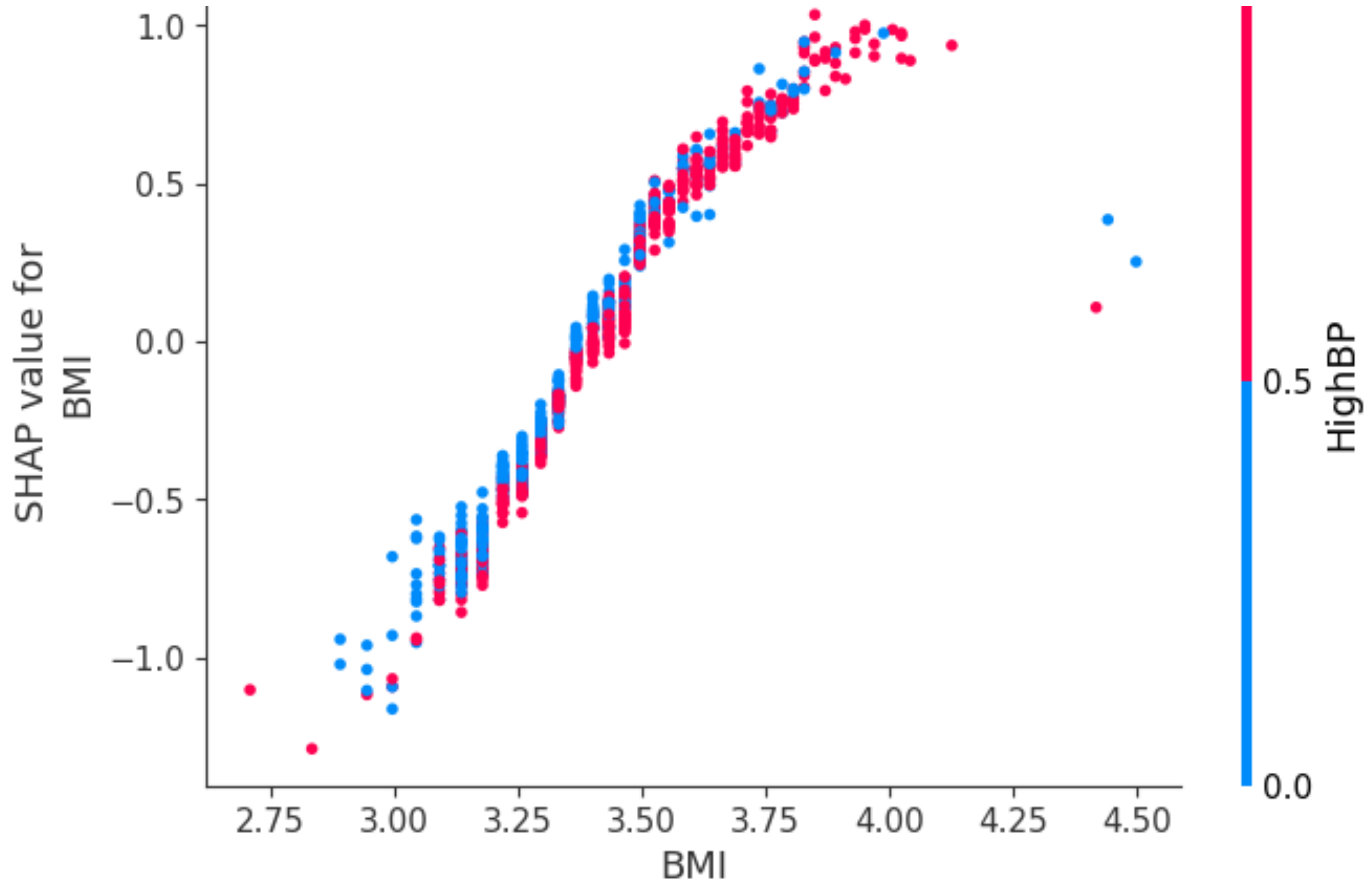
```
Top features by mean(|SHAP|): ['GenHlth', 'HighBP', 'BMI', 'Age', 'HighChol']
```




−1.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
GenHlth
0.0


SHAP value for HighBP
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
0.0
0.2
0.4
0.6
0.8
1.0
HighBP
GenHlth
5.0
4.2
3.4
2.6
1.8
1.0

1.0

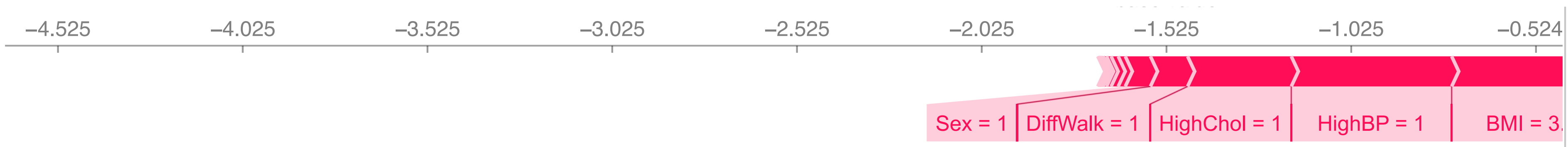


```
Local explanation for row index 27988:
```

	ID	HighBP	HighChol	CholCheck	BMI	Smoker	Stroke	HeartDiseaseorAttack	PhysActivity	Frui
27988	91547	1	1	1	3.637586	0	0	0	0	

1 rows × 22 columns

base value



The SHAP analysis provides a transparent view of how XGBoost forms its predictions, highlighting that general health (GenHlth), high blood pressure, BMI, age, and high cholesterol are the most influential determinants of diabetes risk. The summary plot shows that worse general health, elevated BMI, and the presence of hypertension or high cholesterol consistently push predictions toward the positive class, aligning with established clinical risk factors. Interaction patterns further illustrate that individuals with both high blood pressure and poor general health receive markedly higher risk scores. The local explanation for the selected individual reinforces these findings—features such as difficulty walking, high cholesterol, high blood pressure, elevated BMI, and poor self-rated health collectively increase their predicted risk. Overall, the SHAP framework confirms that the model is learning medically plausible relationships and provides interpretable, case-level justification for its outputs.

This block prepares a production-style submission using the tuned XGBoost model. It mirrors the training pipeline: coerces Yes/No to 1/0, applies train-derived imputations and the same log transforms, aligns columns to the model's feature set, and then scores the test data. Predictions are thresholded at the validation-selected FINAL_THR to balance precision and recall, and both class labels and probabilities are exported (submission.csv, submission_proba.csv). This ensures methodological consistency, avoids leakage, and produces reproducible, assessment-ready outputs.

```
# --- Create submission from the FINAL trained tuned model ---
```

```
import numpy as np, pandas as pd, warnings
warnings.simplefilter("ignore", FutureWarning)

# Threshold from validation tuning
FINAL_THR = float(thr_locked)

# 0) Load raw test
df_test = pd.read_csv("data_test.csv").copy()
id_series = df_test["ID"] if "ID" in df_test.columns else np.arange(len(df_test))

# 1) Use the already fitted tuned model
final_model = fitted_models["XGBoost"]   # ✅ this one is trained

# 2) Columns model saw during training
train_cols = list(getattr(final_model, "feature_names_in_", X_train.columns))

# 3) Use X_train for reference stats
train_ref = X_train

# Cleaning steps
yn_cols = [
    "HighBP","HighChol","CholCheck","Smoker","Stroke","HeartDiseaseorAttack",
    "PhysActivity","Fruits","Veggies","HvyAlcoholConsump","AnyHealthcare",
    "NoDocbcCost","DiffWalk"
]
for c in [c for c in yn_cols if c in df_test.columns]:
    df_test[c] = df_test[c].replace({"Yes": 1, "No": 0, "YES": 1, "NO": 0})

# ✅ fixed line (complete string "object")
for c in df_test.columns:
    if df_test[c].dtype == "object":
        df_test[c] = pd.to_numeric(df_test[c], errors="coerce")

# Imputations (median/mode from training)
fill_map = {
    "BMI": train_ref["BMI"].median() if "BMI" in train_ref.columns else None,
    "CholCheck": train_ref["CholCheck"].mode()[0] if "CholCheck" in train_ref.columns else None,
    "HeartDiseaseorAttack": train_ref["HeartDiseaseorAttack"].mode()[0] if "HeartDiseaseorAttack" in tra
    "Sex": train_ref["Sex"].mode()[0] if "Sex" in train_ref.columns else None,
    "Age": train_ref["Age"].median() if "Age" in train_ref.columns else None,
    "Education": train_ref["Education"].median() if "Education" in train_ref.columns else None,
}
for col, fill in fill_map.items():
    if fill is not None and col in df_test.columns:
        df_test[col] = df_test[col].fillna(fill)

num_overlap = [c for c in train_cols if c in train_ref.columns and c in df_test.columns]
df_test[num_overlap] = df_test[num_overlap].fillna(train_ref[num_overlap].median(numeric_only=True))

# Log transforms
for c in ["BMI", "MentHlth", "PhysHlth"]:
    if c in df_test.columns:
        df_test[c] = np.log1p(df_test[c].clip(lower=0))

# Ensure correct feature set
for c in train_cols:
    if c not in df_test.columns:
        df_test[c] = 0
X_submit = df_test.reindex(columns=train_cols)

# 4) Predict + binarize
y_prob = final_model.predict_proba(X_submit)[:, 1]
y_pred = (y_prob >= FINAL_THR).astype(int)
```

```
# 5) Save submission
submission = pd.DataFrame({"ID": id_series, "Diabetes_binary": y_pred})
submission.to_csv("submission.csv", index=False)
pd.DataFrame({"ID": id_series, "proba": y_prob}).to_csv("submission_proba.csv", index=False)

print(f"✅ submission.csv saved (thr={FINAL_THR:.2f}) | rows={len(submission)}")
display(submission.head())
```

```
✅ submission.csv saved (thr=0.27) | rows=30000
```

	ID	Diabetes_binary
0	182992	1
1	148963	0
2	82811	1
3	110563	1
4	208808	0

Double-click (or enter) to edit