DTSC - 691 Capstone Project - Stroke Risk Prediction

Importing the Dataset

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

#import the dataset
health_data = pd.read_csv("StrockDataset.csv")
```

Exploratory Data Analysis

Understanding the Data

```
#Summary of the data
health_data.info()

#Display first 10 rows of the data
health_data.head(10)
```

Class Distribution of Categorical Variables

```
categorical_cols =
['gender','ever_married','work_type','Residence_type','smoking_status'
]

for col in categorical_cols:
    print(f"\n{col} distribution:")
    print(health_data[col].value_counts())
```

Descriptive Statistics

```
#Generate summary statistics of the data
health_data.describe()
```

By observing the summary statistics, I found that the minimum age is 0.08, which is not a realistic value. Similarly, the maximum BMI is 97, which is unusually high and should be addressed.

Data Cleaning

```
#Identify the missing values in each column
health_data.isnull().sum()
```

I have 201 missing values for BMI . We would handle the missing values by median imputation that would fill in the missing values with the median of BMI

```
#Find the median of BMI
health_data['bmi'].median()
```

I use scikit-learn tool Simple Imputer to fill in missing values with the median 28.1.

```
#Import sci-kit learn  and Simple Imputer
from sklearn.impute import SimpleImputer

#Imputers
bmi_imputer = SimpleImputer(strategy='median')

#Apply Imputator to fill in the missing values with median of BMI
health_data['bmi'] = bmi_imputer.fit_transform(health_data[['bmi']])
```

Verify that the dataset contains no missing values

```
health_data.isnull().sum()
```

Previously, in our summary statistics, we observed that BMI had a maximum value of 97 and age had a minimum value of 0.08, both of which are unrealistic. I need to address these discrepancies. Values above 60 are very rare and likely data errors or extreme outliers. So capping BMI at 60 is a reasonable data cleaning choice to handle extreme values, even if it's not a "healthy" or "approved" medical cutoff therefore I will cap the maximum BMI value at 60 by replacing all BMI values greater than 60 with 60.

```
#Modify the BMI values by giving 60 as maximum value
health_data.loc[health_data['bmi'] > 60,'bmi'] = 60
```

I will remove rows where the age is less than 1. Since the age column is currently of float datatype, I will convert it to an integer type.

```
#remove age below 1
health_data = health_data[health_data['age']>= 1]

#Convert age into integer type
health_data['age'] = health_data['age'].round().astype(int)
```

I will also remove the id column because it will not be relevant for my analysis

```
#drop id column
health_data = health_data.drop('id',axis=1)
```

Let's observe summary statistics again after cleaning. I can see that we dont have id column anymore , minimum age is 1, and the maximum BMI has been capped at 60.

```
health_data.describe()
```

Visualization

Box Plot of Age Grouped by Stroke Occurence

```
# import libraries
import seaborn as sns
import matplotlib.pyplot as plt

# Generate a box plot to visualize age distribution among individuals
who had stroke and who didnt
sns.boxplot(x='stroke', y='age', data=health_data)
plt.title('Age Grouped by Stroke Occurence')
plt.xlabel('Stroke (0 = No, 1 = Yes)')
plt.ylabel('Age')
plt.show()
```

The Above plot interprets that those with stroke , ages are generally higher , mostly concentrated between approximately 60 and 80 years, indicating that stroke patients tend to be older

Stacked Bar Chart

```
#Filter out unknown category
filtered_health_data = health_data[health_data['smoking_status'] !=
'Unknown']

# Calculate proportions
smoking_stroke = pd.crosstab(filtered_health_data['smoking_status'],
filtered_health_data['stroke'], normalize='index')

# Generate a Stacked Bar chart reperesenting Smoking Status based on
individuals who had stroke and not
ax = smoking_stroke.plot(kind='bar', stacked=True,
colormap='coolwarm', figsize=(8, 5))

# Add percentage labels
for p in ax.patches:
    height = p.get_height()
    if height > 0:
        ax.text(p.get_x() + p.get_width()/2,
                p.get_y() + height/2,
                f'{height*100:.1f}%',
                ha='center', va='center', fontsize=15, color='black')

# Add Labels and title to the plot
plt.title('Proportion of Stroke Occurence within Smoking Status
Groups')
plt.xlabel('Smoking Status')
plt.ylabel('Proportion')
plt.legend(title='Stroke', labels=['No', 'Yes'])
plt.xticks(rotation=15)
```

```
plt.tight_layout()
plt.show()
```

Due to highly imbalanced stroke class we receive a higher proportion of non-stroke cases compared to stroke. From the plot i found that people who are formerly smoked have more chance of experiencing a stroke followed by current smokers and those never smoked.

KDE Plot

```
#Generate a KDE plot to visualize average glucose levels among
individuals who had stroke and those who did not
sns.kdeplot(health_data[health_data['stroke'] == 0]
['avg_glucose_level'], label='No Stroke', fill=True)
sns.kdeplot(health_data[health_data['stroke'] == 1]
['avg_glucose_level'], label='Stroke', fill=True)
plt.title('Glucose Level Distribution Among Stroke and Non-Stroke
Individuals')
plt.xlabel('Average Glucose Level')
plt.ylabel('Density')
plt.legend()
plt.show()
```

The plot compares the distribution of average glucose levels between individuals who had a stroke (orange) and those who did not (blue). We see a higher and narrow peak around 100 mg/dL which indicates large number of people are under normal range. A secondary peak around 200 mg/dL, suggesting a notable subgroup with significantly elevated glucose levels, possibly linked to diabetes or hyperglycemia. This distribution indicates that both normal and high glucose levels are observed in stroke patients, but higher glucose levels are more prevalent among them than in non-stroke individuals.

Correlation HeatMap

```
# identify the metrics of features that are related to stroke
corr = health_data.corr(numeric_only=True)

#Generate a correlation heatmap
plt.figure(figsize=(10,8))
sns.heatmap(corr,annot=True,cmap='coolwarm',fmt='.2f',linewidths=0.5)
plt.title("Correlation Heatmap")
plt.show()
```

Age has a correlation of 0.25 with stroke, indicating a moderate positive relationship. This means that as age increases, the likelihood of having a stroke tends to increase slightly.

Balance the Stroke Class

Lets check the class distribution of the Stroke column which is our target variable

```
#class distribution of stroke column
health_data['stroke'].value_counts()
```

I observes a high class imbalance in stroke cases, with the majority being non-stroke. This imbalance can lead to poor model performance, so it is important to apply techniques to balance the data

To Address the stroke class imbalance we use CTGAN(Conditional Tabular Generative Adversarial Network) a deep learning model that learns underlying distributions of data and generates synthetic samples that mimics the original data

```
import sdv
from sdv.metadata import Metadata
import pandas as pd
from sdv.single_table import CTGANSynthesizer
import random
import torch

# Set seeds for reproducibility . Usually all random generators are
called.
SEED = 42
# for Python in-built random-generators
random.seed(SEED)
# sets the seed for numpy and pandas random generator ,
np.random.seed(SEED)
# CTGAN primarily uses pytorch random generator because it is built on
top of pytorch
torch.manual_seed(SEED)

#Create a copy of health_data based on positive stroke cases
stroke_positive = health_data[health_data['stroke'] == 1].copy()

#Figures out the type of each column (e.g., categorical, numerical,
boolean) in our dataset
metadata = Metadata.detect_from_dataframe(data=stroke_positive)

synthesizer = CTGANSynthesizer(metadata)
synthesizer.fit(stroke_positive)

# Generate synthetic samples
synthetic_data = synthesizer.sample(num_rows=2409) # 50% of non-stroke
cases
synthetic_data['stroke'] = 1  # Add target label back

# Combine with original dataset
health_data_balanced = pd.concat([health_data, synthetic_data],
ignore_index=True)
```

The model may take some time to run based on the OS. Verify the stroke class again

```
health_data_balanced['stroke'].value_counts()
```

Model Training

Split our balanced dataset for training and testing

```
from sklearn.model_selection import train_test_split

features = health_data_balanced.drop('stroke',axis=1)
target = health_data_balanced['stroke']

x_train_val,x_test,y_train_val,y_test =
train_test_split(features,target,test_size=0.2,random_state=42,stratif
y=target)
```

Since our hypertension and heart_disease columns have only values between 0 and 1 we dont need to standardize the data .Encode categorical features using one-hot encoding and standardize numerical features using StandardScaler for model training

```
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer

#create a variable called numerical_cols to extract numerical columns
from the data
numerical_cols = ['age','avg_glucose_level','bmi']
#create a variable called numerical_cols to extract columns with
binary values  from the data
binary_cols = ['hypertension','heart_disease']

preprocessor = ColumnTransformer(
    transformers = [

('cat',OneHotEncoder(handle_unknown='ignore'),categorical_cols),
        ('num',StandardScaler(),numerical_cols),
        ('bin','passthrough',binary_cols)

    ]
)


#Fit and tranform on training data and transform only on testing data
x_train_encoded = preprocessor.fit_transform(x_train_val)
x_test_encoded = preprocessor.transform(x_test)
```

I will be training and evaluating 7 different models and find the best performing model that would predict stroke risk . A good recall ,accuracy and better precision would be focus in our model evaluation.

ML Model Training and Evaluation

Random Forest : Training the Model

```
from sklearn.ensemble import RandomForestClassifier

random_f = RandomForestClassifier(n_estimators=100,random_state=42)
```

Hyperparameter Tuning using GridSearchCV

```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import  make_scorer,recall_score

recall_scorer= make_scorer(recall_score,average='binary')

rf_param_grid = {
    'n_estimators': [100],
    'max_depth': [17],
    'min_samples_split': [10],
}


rf_grid_search = GridSearchCV(
    estimator=random_f,
    param_grid=rf_param_grid,
    scoring=recall_scorer,
    cv=3,
    n_jobs=-1,
    verbose=2
)

rf_model = rf_grid_search.fit(x_train_encoded,y_train_val)

print("Best parameters:", rf_grid_search.best_params_)
print("Best recall score:", rf_grid_search.best_score_)

rf_best_model = rf_grid_search.best_estimator_
```

Random Forest: Testing the Model

```
y_rand_proba = rf_best_model.predict_proba(x_test_encoded)[:, 1]
y_rand_custom = (y_rand_proba >= 0.5).astype(int)
y_rand_custom
```

Random Forest : Model Evaluation Metrics

Confusion Matrix

```
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
cm = confusion_matrix(y_test, y_rand_custom)
```

```
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Random Forest - Confusion Matrix")
plt.show()
```

Classification Report

```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_rand_custom))
```

ROC-AUC Curve

```
from sklearn.metrics import roc_curve,auc
#Calculate ROC curve and AUC
fpr, tpr , thresholds = roc_curve(y_test,y_rand_custom)
roc_auc = auc(fpr,tpr)

#plot ROC-AUC curve
plt.figure(figsize=(8,6))
plt.plot(fpr,tpr,label=f'ROC curve(AUC= {roc_auc:.2f})',linewidth=2)
plt.plot([0, 1], [0, 1], 'k--', label='Chance')  # Diagonal line
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC-AUC Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```

Gradient Boost: Training the Model

```
from sklearn.ensemble import GradientBoostingClassifier

gboost_model = GradientBoostingClassifier(random_state=42)
```

Hyperparameter Tuning using GridSearchCV

```
from sklearn.model_selection import GridSearchCV


gb_param_grid = {
    'n_estimators': [100],
    'learning_rate': [0.05,0.1],
    'max_depth': [5],
}



gb_search = GridSearchCV(
```

```
    estimator=gboost_model,
    param_grid= gb_param_grid,
    scoring=recall_scorer,
    cv=3,
    n_jobs=-1,
    verbose=2,

)

gb_model = gb_search.fit(x_train_encoded,y_train_val)

print("Best parameters:",gb_search.best_params_)
print("Best recall score:", gb_search.best_score_)

gb_best_model = gb_search.best_estimator_
```

Testing the Model

```
y_gb_proba = gb_best_model.predict_proba(x_test_encoded)[:, 1]
y_gb_custom = (y_gb_proba >= 0.5).astype(int)
y_gb_custom
```

Classification Report

```
print(classification_report(y_test, y_gb_custom))
```

Confusion Matrix

```
cm = confusion_matrix(y_test, y_gb_custom)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Gradient Boost - Confusion Matrix")
plt.show()
```

ROC-AUC Curve

```
#Calculate ROC curve and AUC
fpr, tpr , thresholds = roc_curve(y_test,y_gb_custom)
roc_auc = auc(fpr,tpr)

#plot ROC-AUC curve
plt.figure(figsize=(8,6))
plt.plot(fpr,tpr,label=f'ROC curve(AUC= {roc_auc:.2f})',linewidth=2)
plt.plot([0, 1], [0, 1], 'k--', label='Chance')  # Diagonal line
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC-AUC Curve')
plt.legend(loc='lower right')
```

```
plt.grid(True)
plt.show()
```

Ada Boost : Training the Model

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

# Define base estimator (optional)
base_estimator = DecisionTreeClassifier(max_depth=1)

# Define AdaBoost model
adaboost_model =
AdaBoostClassifier(estimator=base_estimator,random_state=42)
```

Hyperparameter Tuning using GridSearchCV

```


# Define hyperparameter grid for AdaBoost
ada_param_grid = {
    'n_estimators': [100],
    'learning_rate': [0.01,0.05, 0.1],
    'estimator__max_depth': [5]  # tuning the base learner
}



# Grid Search
ada_search = GridSearchCV(
    estimator=adaboost_model,
    param_grid=ada_param_grid,
    scoring=recall_scorer,
    cv=3,
    n_jobs=-1,
    verbose=2
)

# Fit model
ada_model = ada_search.fit(x_train_encoded, y_train_val)

# Results
print("Best parameters:", ada_search.best_params_)
print("Best Recall score:", ada_search.best_score_)

# Best model
ada_best_model = ada_search.best_estimator_
```

Testing the Model

```
y_ab_proba = ada_best_model.predict_proba(x_test_encoded)[:, 1]
y_ab_custom = (y_ab_proba >= 0.5).astype(int)
y_ab_custom
```

Confusion Matrix

```
cm = confusion_matrix(y_test, y_ab_custom)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("AdaBoost - Confusion Matrix")
plt.show()
```

Classification Report

```
print(classification_report(y_test, y_ab_custom))
```

ROC-AUC Curve

```
#Calculate ROC curve and AUC
fpr, tpr , thresholds = roc_curve(y_test,y_ab_custom)
roc_auc = auc(fpr,tpr)

#plot ROC-AUC curve
plt.figure(figsize=(8,6))
plt.plot(fpr,tpr,label=f'ROC curve(AUC= {roc_auc:.2f})',linewidth=2)
plt.plot([0, 1], [0, 1], 'k--', label='Chance')  # Diagonal line
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC-AUC Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```

Classification Neural Network Model

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from sklearn.metrics import classification_report,confusion_matrix

#Define the model
NN_model = Sequential([

Dense(100,activation='relu',input_shape=(x_train_encoded.shape[1],)),
    Dropout(0.3),
    Dense(60,activation='relu'),
    Dropout(0.3),
```

```
    Dense(60,activation='relu'),
    Dropout(0.3),
    Dense(1,activation='sigmoid')
])

NN_model.compile(optimizer='adam',loss='binary_crossentropy',metrics=[
'accuracy'])

hist =
NN_model.fit(x_train_encoded,y_train_val,epochs=10,batch_size=32)
```

Testing the model

```
y_pred_prob = NN_model.predict(x_test_encoded)
neural_predict = (y_pred_prob >= 0.5).astype(int)
```

Classification Report

```
print(classification_report(y_test, neural_predict))
```

Confusion Matrix

```
cm = confusion_matrix(y_test, neural_predict)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("NeuralNetwork - Confusion Matrix")
plt.show()
```

ROC-AUC Curve

```
#Calculate ROC curve and AUC
fpr, tpr , thresholds = roc_curve(y_test,neural_predict)
roc_auc = auc(fpr,tpr)

#plot ROC-AUC curve
plt.figure(figsize=(8,6))
plt.plot(fpr,tpr,label=f'ROC curve(AUC= {roc_auc:.2f})',linewidth=2)
plt.plot([0, 1], [0, 1], 'k--', label='Chance')  # Diagonal line
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC-AUC Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```

Logistic Regression : Training the Model

```
from sklearn.linear_model import LogisticRegression

log_reg =
LogisticRegression(max_iter=1000,class_weight='balanced',random_state=
42)
```

Hyperparameter Tuning using GridSearchCV

```
from sklearn.model_selection import GridSearchCV


param_grid = {
    'C': [0.01],
    'penalty': ['l1', 'l2'],
    'solver': ['liblinear']
}



grid_search = GridSearchCV(
    estimator=log_reg,
    param_grid=param_grid,
    scoring=recall_scorer,
    cv=3,
    n_jobs=-1,
    verbose=2

)

lg_model = grid_search.fit(x_train_encoded,y_train_val)

print("Best parameters:", grid_search.best_params_)
print("Best Recall score:", grid_search.best_score_)

best_model = grid_search.best_estimator_
```

Testing the Model

```
y_proba = best_model.predict_proba(x_test_encoded)[:, 1]
y_pred_custom = (y_proba >= 0.5).astype(int)
y_pred_custom
```

Logistic Regression: Model Evaluation Metrics

Confusion Matrix

```
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt

# Plot the confusion matrix
```

```
cm = confusion_matrix(y_test, y_pred_custom)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Logistic Regression- Confusion Matrix")
plt.show()
```

Classification Report including Accuracy ,Precision , Recall and F1 Score of the Model

```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred_custom))
```

ROC-AUC Curve

```
from sklearn.metrics import roc_curve,auc

#Calculate ROC curve and AUC
fpr, tpr , thresholds = roc_curve(y_test,y_pred_custom)
roc_auc = auc(fpr,tpr)

#plot ROC-AUC curve
plt.figure(figsize=(8,6))
plt.plot(fpr,tpr,label=f'ROC curve(AUC= {roc_auc:.2f})',linewidth=2)
plt.plot([0, 1], [0, 1], 'k--', label='Chance')  # Diagonal line
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC-AUC Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```

Graph Neural Network Model

```
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import NearestNeighbors
import torch
from torch_geometric.data import Data

feature_cols =
['age','hypertension','heart_disease','avg_glucose_level']
graph_features = health_data_balanced[feature_cols].values

scaler = StandardScaler()
graph_features = scaler.fit_transform(graph_features)

target_var = health_data_balanced['stroke'].values

x = torch.tensor(graph_features,dtype=torch.float)
```

```
y = torch.tensor(target_var,dtype=torch.long)

#Create edges using KNN
k = 3
nbrs = NearestNeighbors(n_neighbors=k+1).fit(graph_features)
_, indices = nbrs.kneighbors(graph_features)


#Build edge list
edge_sources = []
edge_targets = []


for i, neighbors in enumerate(indices):
    for j in neighbors[1:]:
        edge_sources.append(i)
        edge_targets.append(j)

edge_index =
torch.tensor([edge_sources,edge_targets],dtype=torch.long)

print(edge_index.shape)
```

Training the Model

```
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn import GCNConv

class StrokeGNN(nn.Module):
    def __init__(self, input_dim, hidden_dim=32):
        super(StrokeGNN,self).__init__()
        self.conv1 = GCNConv(input_dim,hidden_dim)
        self.conv2 = GCNConv(hidden_dim,hidden_dim)
        self.classifier = nn.Linear(hidden_dim,2)

    def forward(self,data):
        x,edge_index = data.x,data.edge_index

        x = self.conv1(x,edge_index)
        x = F.relu(x)
        x = self.conv2(x,edge_index)
        x= F.relu(x)

        return self.classifier(x)

from torch_geometric.utils import train_test_split_edges

data = Data(x=x,edge_index=edge_index,y=y)

# Split manually using sklearn
```

```
train_mask, test_mask = train_test_split(torch.arange(len(y)),
test_size=0.2, stratify=y)

data.train_mask = torch.zeros(data.num_nodes, dtype=torch.bool)
data.test_mask = torch.zeros(data.num_nodes, dtype=torch.bool)

data.train_mask[train_mask] = True
data.test_mask[test_mask] = True

from sklearn.utils.class_weight import compute_class_weight
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
gnn_model = StrokeGNN(input_dim=x.size(1)).to(device)
data = data.to(device)

optimizer = torch.optim.Adam(gnn_model.parameters(), lr=0.01)

# Compute weights based on training labels only
# Extract training labels
train_labels = data.y[data.train_mask].cpu().numpy()

# Convert classes to NumPy array
classes = np.array([0, 1])

# Compute weights
class_weights = compute_class_weight(class_weight='balanced',
classes=classes, y=train_labels)

# Convert to tensor and move to device
weights = torch.tensor(class_weights, dtype=torch.float).to(device)

# Define weighted loss
criterion = nn.CrossEntropyLoss(weight=weights)

def train():
    gnn_model.train()
    optimizer.zero_grad()
    out = gnn_model(data)
    loss = criterion(out[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()
    return loss.item()

def test():
    gnn_model.eval()
    out = gnn_model(data)
    pred = out.argmax(dim=1)

    correct = pred[data.test_mask] == data.y[data.test_mask]
    acc = int(correct.sum()) / int(data.test_mask.sum())
    return acc
```

```
for epoch in range(1, 101):
    loss = train()
    acc = test()
    if epoch % 10 == 0:
        print(f"Epoch {epoch:3d} | Loss: {loss:.4f} | Test Acc:
{acc:.4f}")
```

Testing the model

```
gnn_model.eval()

# Forward pass
out = gnn_model(data)

# Apply softmax to get probabilities
probs = torch.softmax(out, dim=1)

# Predicted class labels
preds = probs.argmax(dim=1)

# Filter for test nodes
y_true = data.y[data.test_mask].cpu().numpy()
y_pred_gnn = preds[data.test_mask].cpu().numpy()

y_pred_gnn
```

Classification Report

```
print(classification_report(y_true, y_pred_gnn))
```

Confusion Matrix

```
cm = confusion_matrix(y_true, y_pred_gnn)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Graph Neural Network - Confusion Matrix")
plt.show()
```

ROC-AUC Curve

```
#Calculate ROC curve and AUC
fpr, tpr , thresholds = roc_curve(y_true,y_pred_gnn)
roc_auc = auc(fpr,tpr)

#plot ROC-AUC curve
plt.figure(figsize=(8,6))
plt.plot(fpr,tpr,label=f'ROC curve(AUC= {roc_auc:.2f})',linewidth=2)
plt.plot([0, 1], [0, 1], 'k--', label='Chance')  # Diagonal line
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
```

```
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC-AUC Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```

XGBoost: Training the Model

```
import xgboost as xgb
xgb_model = xgb.XGBClassifier( eval_metric='aucpr', random_state=42)
```

Hyperparameter Tuning using GridSearchCV

```
xgb_param_grid = {
    'max_depth': [6],
    'learning_rate': [0.05],
    'n_estimators': [300],
    'scale_pos_weight': [2]
}

xgb_grid_search = GridSearchCV(estimator=xgb_model,
param_grid=xgb_param_grid, scoring='recall', cv=3, verbose=2)
xgb_grid_search.fit(x_train_encoded, y_train_val)

print("Best params:", xgb_grid_search.best_params_)
print("Best recall:", xgb_grid_search.best_score_)

xgb_best_model = xgb_grid_search.best_estimator_
```

Testing the Model

```
xgb_proba = xgb_best_model.predict_proba(x_test_encoded)[:, 1]
xgb_custom = (xgb_proba >= 0.5).astype(int)
xgb_custom
```

Confusion Matrix

```
cm = confusion_matrix(y_test, xgb_custom)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("XGBoost - Confusion Matrix")
plt.show()
```

Classification Report

```
print(classification_report(y_test, xgb_custom))
```

ROC-AUC Curve

```
#Calculate ROC curve and AUC
fpr, tpr , thresholds = roc_curve(y_test,xgb_custom)
roc_auc = auc(fpr,tpr)

#plot ROC-AUC curve
plt.figure(figsize=(8,6))
plt.plot(fpr,tpr,label=f'ROC curve(AUC= {roc_auc:.2f})',linewidth=2)
plt.plot([0, 1], [0, 1], 'k--', label='Chance')  # Diagonal line
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC-AUC Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```

SHAP

```
import shap
# Get the fitted OneHotEncoder and StandardScaler
ohe = preprocessor.named_transformers_['cat']
scaler = preprocessor.named_transformers_['num']

# Get encoded categorical feature names
encoded_cat_features = ohe.get_feature_names_out(categorical_cols)

# Combine with numerical column names
all_feature_names = np.concatenate([encoded_cat_features,
numerical_cols])

passthrough_cols = [col for col in x_train_val.columns if col not in
categorical_cols + numerical_cols]
all_feature_names = np.concatenate([encoded_cat_features,
numerical_cols, passthrough_cols])

explainer = shap.Explainer(xgb_best_model,x_test_encoded)
shap_values = explainer(x_test_encoded,check_additivity=False)

shap.summary_plot(shap_values,
x_test_encoded,feature_names=all_feature_names)
```

To deploy the model to streamlit , i will use a model pipeline that would preprocess data and fit the best model

```
from sklearn.pipeline import Pipeline
model_pipeline = Pipeline([
    ('preprocessor',preprocessor),
    ('classifier',xgb_best_model)
```

```
])

model_pipeline.fit(x_train_val,y_train_val)

print(model_pipeline)
```

Deployment

```
#for scikitlearn
import joblib
joblib.dump(model_pipeline,"stroke_model_sklearn.pkl")
```