Homework 2: Trees and Calibration

Instructions

Please push the .ipynb, .py, and .pdf to Github Classroom prior to the deadline. Please include your UNI as well.

Make sure to use the dataset that we provide in CourseWorks/Classroom.

There are a lot of applied questions based on the code results. Please make sure to answer them all. These are primarily to test your understanding of the results your code generate (similar to any Data Science/ML case study interviews).

Name: Xinyu Li

UNI: xl3228

The Dataset

Description

This dataset contains details of individual relating to their health. The target is stored in **strokes** column which is binary variable indicating either 0 or 1. The goal of the assignment is a binary classification task to predict wether the person will get a stroke depending on other health factors

In [1]:
```
import warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import time
from sklearn.model_selection import train_test_split, StratifiedKFold, GridSearchCV, cross_val_score
```

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler, OrdinalEncoder, OneHotEncoder, LabelEncoder
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier, HistGradientBoostingClassifier
from sklearn.metrics import precision_score, make_scorer, accuracy_score, brier_score_loss, f1_score, recall_score
from sklearn.calibration import CalibrationDisplay, CalibratedClassifierCV, calibration_curve
import xgboost as xgb
```

Question 1: Decision Trees

1.1: Load the provided dataset

In [3]:
```
## YOUR CODE HERE
df = pd.read_csv("healthcare-dataset-stroke-data.csv")
df.head()
```

Out[3]:

	id	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	strok
0	9046	Male	67.0	0	1	Yes	Private	Urban	228.69	36.6	formerly smoked	
1	51676	Female	61.0	0	0	Yes	Self-employed	Rural	202.21	NaN	never smoked	
2	31112	Male	80.0	0	1	Yes	Private	Rural	105.92	32.5	never smoked	
3	60182	Female	49.0	0	0	Yes	Private	Urban	171.23	34.4	smokes	
4	1665	Female	79.0	1	0	Yes	Self-employed	Rural	174.12	24.0	never smoked	

1.2: Show or Plot % of missing values in each column. Would you consider dropping any columns or assuming we want to train a decision tree, would you consider imputing the missing values? If not, why? (Remove the columns that you consider dropping)

In [4]:
```
## YOUR CODE HERE
row_num = len(df)
for col in df.columns:
    missing_amount = df[col].isnull().sum()
```

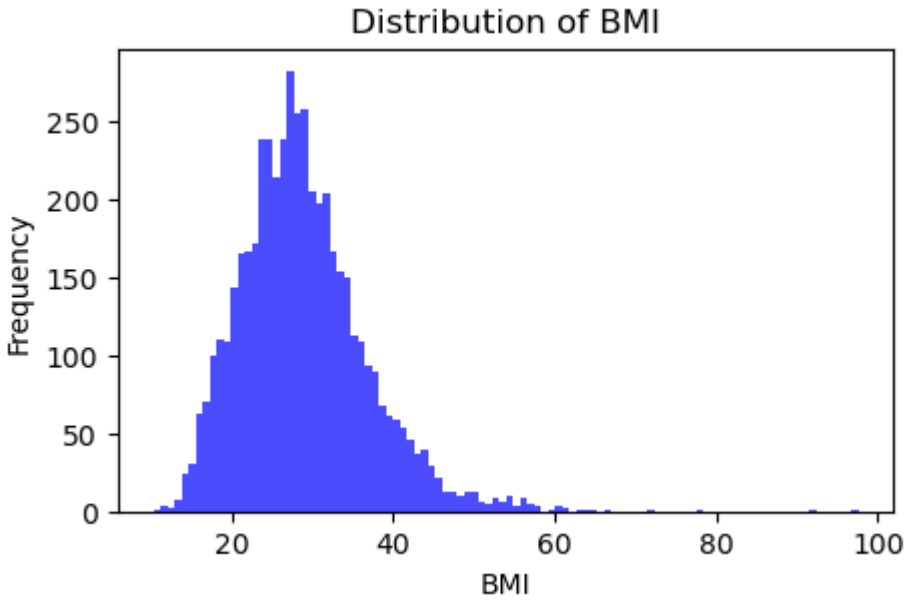
```
    missing_percentage = round(missing_amount/row_num,2)
    print(f"Percentage of missing values for {col}: {round(missing_percentage*100,2)}%")
```

```
Percentage of missing values for id: 0.0%
Percentage of missing values for gender: 0.0%
Percentage of missing values for age: 0.0%
Percentage of missing values for hypertension: 0.0%
Percentage of missing values for heart_disease: 0.0%
Percentage of missing values for ever_married: 0.0%
Percentage of missing values for work_type: 0.0%
Percentage of missing values for Residence_type: 0.0%
Percentage of missing values for avg_glucose_level: 0.0%
Percentage of missing values for bmi: 4.0%
Percentage of missing values for smoking_status: 0.0%
Percentage of missing values for stroke: 0.0%
```

Your Comments Here:

- (1) I decided not to drop any columns. I've noticed that the 'bmi' column is the only one with missing values in the dataset. Given this relatively small percentage of missing values (4% of dataset), I've chosen to retain the 'bmi' column. This decision is based on the belief that the 'bmi' feature still contains valuable information. Removing Removing 'bmi' could potentially have a negative impact on both the integrity and predictive capability of our model.
- (2) While decision trees can naturally handle missing values, I recommend imputing the missing values for the 'bmi' feature. First, Imputing missing values in a way that reflects the underlying data distribution can lead to more accurate predictions. If there is a systematic pattern to the missing values, imputation can help the decision tree model capture this pattern. Second, Imputing missing values can reduce the bias. Imputing missing values in decision trees helps prevent bias introduced by creating separate branches for instances with missing values and those without. It ensures all instances contribute to tree construction without introducing artificial splits.

In [5]:
```
## Distribution of BMI
plt.figure(figsize=(5, 3))
plt.hist(df['bmi'], bins=100, color='blue', alpha=0.7)
plt.xlabel('BMI')
plt.ylabel('Frequency')
plt.title('Distribution of BMI')
plt.show()
```



In [6]:
```
## Handling Missing Values
# Notice bmi is right skwed from the distribution of BMI we plotted. Thus, it is better to use the median.
df['bmi'].fillna(df['bmi'].median(), inplace=True)
```

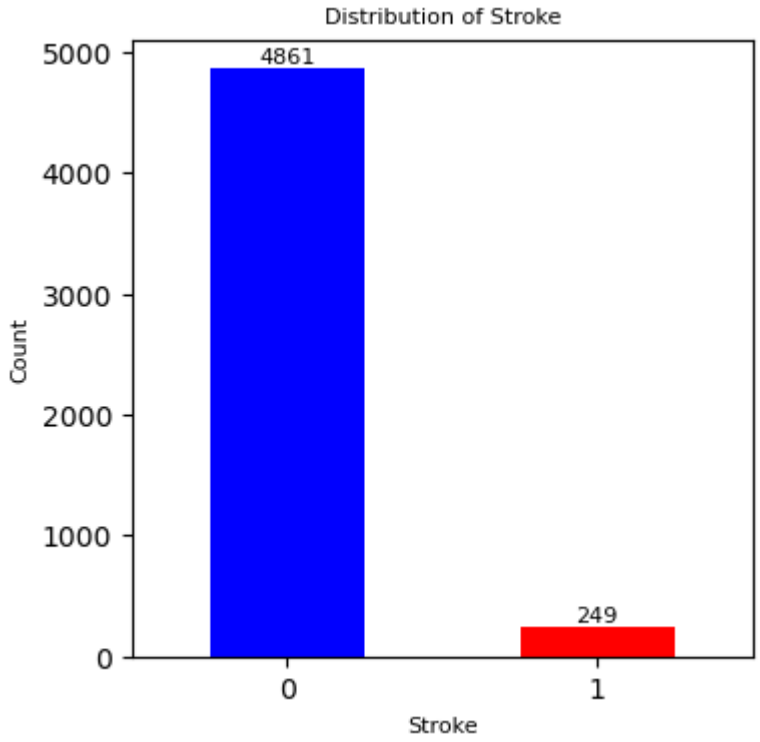
Check for the number of rows after handling null values, each column should have the same number of non-null count

In [7]:
```
## YOUR CODE HERE
row_num = len(df)
print(f"total number of rows in stroke_df: {row_num}\n\n")

for col in df.columns:
    not_null_amount = df[col].notnull().sum()
    print(f"number of non-null rows in {col}: {not_null_amount}")

print("\n\nall columns have the same nnumber of non-null count, which is 5110, same as number of rows.")
```

```
total number of rows in stroke_df: 5110


number of non-null rows in id: 5110
number of non-null rows in gender: 5110
number of non-null rows in age: 5110
number of non-null rows in hypertension: 5110
number of non-null rows in heart_disease: 5110
number of non-null rows in ever_married: 5110
number of non-null rows in work_type: 5110
number of non-null rows in Residence_type: 5110
number of non-null rows in avg_glucose_level: 5110
number of non-null rows in bmi: 5110
number of non-null rows in smoking_status: 5110
number of non-null rows in stroke: 5110


all columns have the same nnumber of non-null count, which is 5110, same as number of rows.
```

1.3 Print the distribution of the target variable. Is the dataset balanced?

In [8]:
```python
## YOUR CODE HERE
stroke_counts = df['stroke'].value_counts()
plt.figure(figsize=(4, 4))
ax = stroke_counts.plot(kind='bar', color=['blue', 'red'])
plt.xlabel('Stroke', fontsize=8)
plt.ylabel('Count', fontsize=8)
plt.title('Distribution of Stroke',fontsize=8)
for i, count in enumerate(stroke_counts):
    plt.text(i, count + 10, str(count), ha='center', va='bottom', fontsize=8, color='black')
plt.xticks(rotation=0)
plt.show()
```

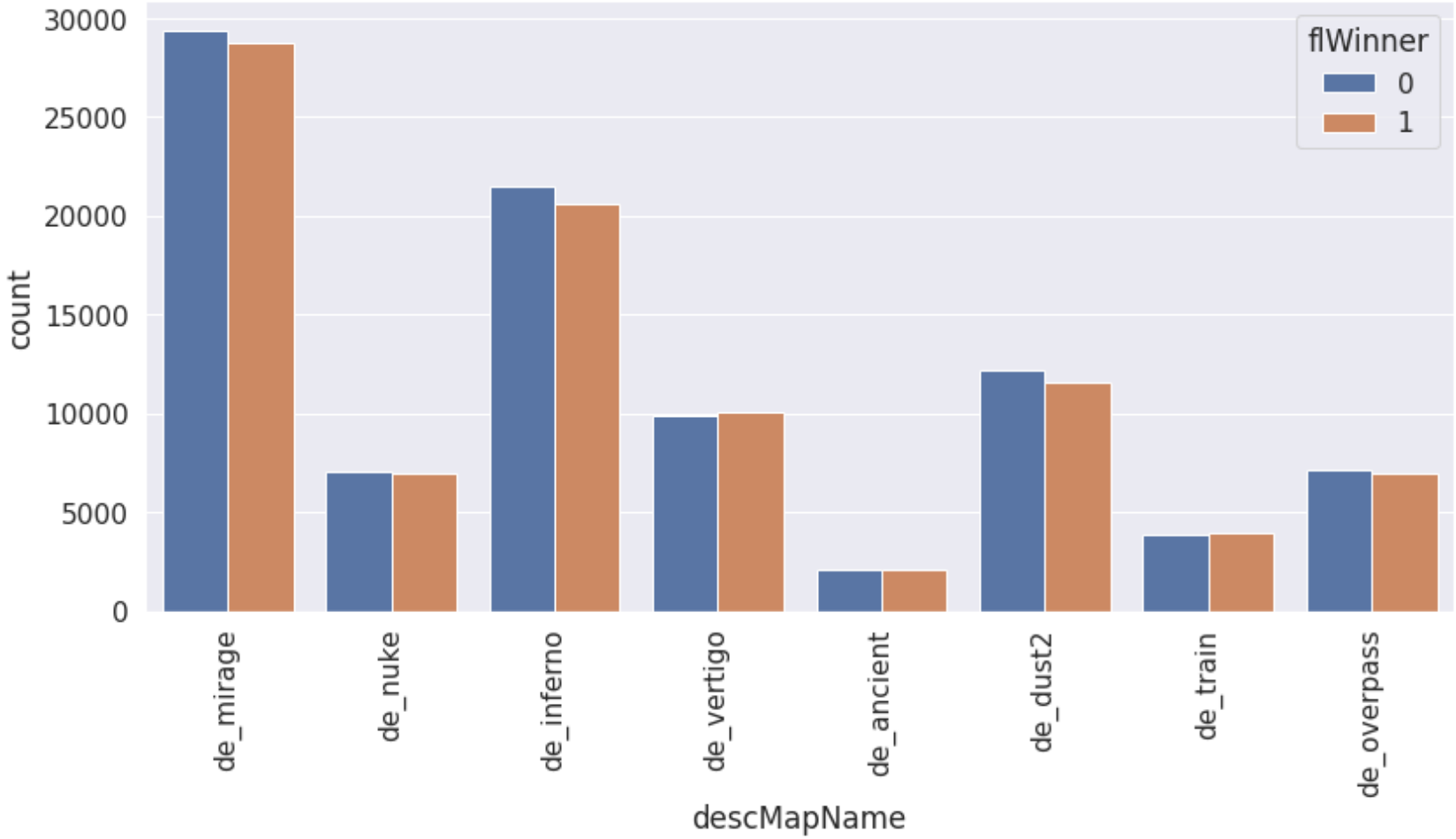


Your Comments Here:

- No, the dataset is not balanced with the minority class labeled with 1 (having stroke). The majority class (having stroke) outnumbers the minority class (not having stroke) by nearly a factor of 20.

1.4: Plot side-by-side bars of class distribtuion for each category for the categorical feature and the target categories.

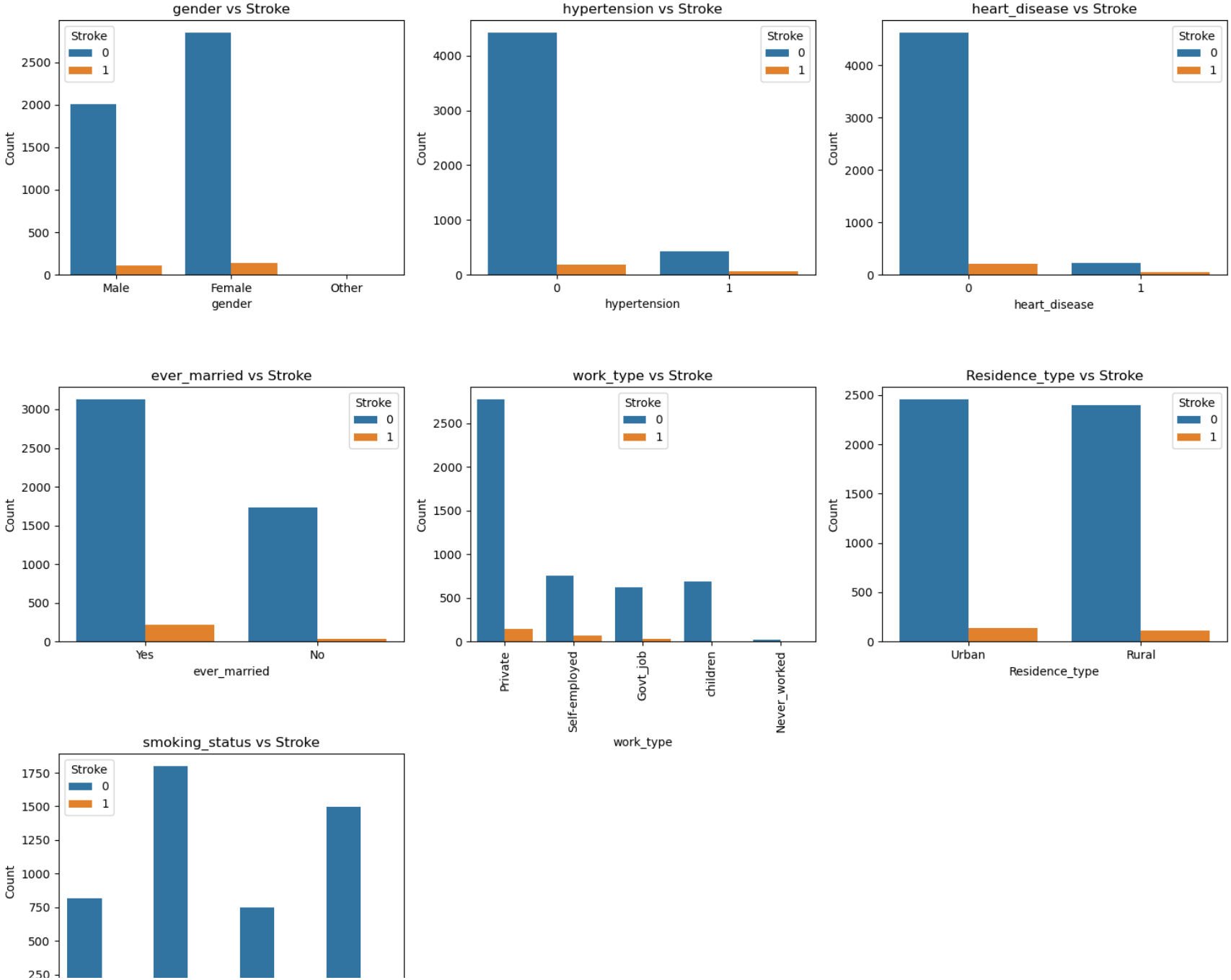
Clarification with Example below : Here `flWinner` is the Target Variable and `descMapName` is a categorical feature. You are required to make such side-by-side bar plot for each categorical feature with repect to it's class distribution with the target feature for our dataset.

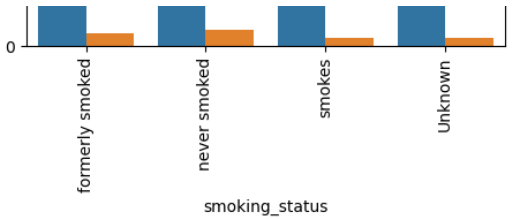


In [9]:
```
categorical_features = ['gender', 'hypertension', 'heart_disease', 'ever_married', 'work_type', 'Residence_type', 'smok

fig, axes = plt.subplots(3, 3, figsize=(15, 15))
fig.suptitle('Distribution of Stroke by Categorical Features', fontsize=16)

axes = axes.flatten()

for i, cat_feature in enumerate(categorical_features):
    sns.countplot(x=cat_feature, hue='stroke', data=df, ax=axes[i])
    axes[i].set_title(f'{cat_feature} vs Stroke')
    axes[i].set_xlabel(cat_feature)
    axes[i].set_ylabel('Count')
    axes[i].legend(title='Stroke')
```

```
    if i == 4 or i == 6:
        axes[i].tick_params(axis='x', rotation=90)

if len(categorical_features) < len(axes):
    for j in range(len(categorical_features), len(axes)):
        fig.delaxes(axes[j])

plt.tight_layout()
plt.subplots_adjust(top=0.9)

plt.show()
```

Distribution of Stroke by Categorical Features





1.5: Preprocess the data (Handle the Categorical Variable). Do we need to apply scaling? Briefly Justify.

In [10]:
```
df_X = df.drop(columns=['stroke','id'])
df_Y = df['stroke'].astype(int)

## YOUR CODE HERE
ohe_features = ['gender', 'hypertension', 'heart_disease', 'ever_married', 'work_type', 'Residence_type', 'smoking_stat
num_features = ['age', 'avg_glucose_level', 'bmi']
ohe = OneHotEncoder(drop = 'first')
X_cate = df_X[ohe_features]
X_cate_encoded = ohe.fit_transform(X_cate)
X_cate_encoded = pd.DataFrame(X_cate_encoded.toarray(), columns=ohe.get_feature_names_out(X_cate.columns))
df_X_encoded = pd.concat([X_cate_encoded, df_X[num_features]], axis=1)
display(df_X_encoded.head())
```

	gender_Male	gender_Other	hypertension_1	heart_disease_1	ever_married_Yes	work_type_Never_worked	work_type_Private	work_type_Self employe
0	1.0	0.0	0.0	1.0	1.0	0.0	1.0	0.
1	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.
2	1.0	0.0	0.0	1.0	1.0	0.0	1.0	0.
3	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.
4	0.0	0.0	1.0	0.0	1.0	0.0	0.0	1.

Your Comments Here:

- We do not need to apply scaling. The models constructed in this assignment are exclusively tree-based models. It's important to note that feature scaling has no discernible impact on the performance of these tree-based models. Tree-based models are

inherently robust to the scale of input features because they evaluate feature values in a relative manner during the splitting process.

1.6: Split the data into development and test datasets. Which splitting methodology did you choose and why?

In [11]:
```
## YOUR CODE HERE
df_X_dev, df_X_test, df_Y_dev, df_Y_test = train_test_split(df_X_encoded, df_Y, random_state = 123, test_size=0.2, stra
```

Your Comments Here:

- [1] Acknowledging the significant class imbalance within the dataset, I will implement stratified splitting. This approach guarantees that the distribution of instances for each class in the divided datasets closely reflects that of the original dataset.
- [2] Given the relatively small size of our dataset, comprising just 5110 rows, I intend to partition it into an 80% allocation for the development dataset and a 20% allocation for the test dataset. This partitioning ratio strikes a sensible balance, ensuring enough data for robust model training while reserving a substantial portion for meaningful evaluation.

1.7: Fit a Decision Tree on the training data until all leaves are pure. What is the performance of the tree on the development set and test set? Provide metrics you believe are relevant and briefly justify.

HINT : Think about the proportion of the class label

In [12]:
```
## YOUR CODE HERE
tree_model = DecisionTreeClassifier(random_state=123, criterion='entropy')
tree_model.fit(df_X_dev, df_Y_dev)

dev_predictions = tree_model.predict(df_X_dev)
test_predictions = tree_model.predict(df_X_test)

dev_precision_class_1 = precision_score(df_Y_dev, dev_predictions, pos_label=1)
dev_recall_class_1 = recall_score(df_Y_dev, dev_predictions, pos_label=1)

test_precision_class_1 = precision_score(df_Y_test, test_predictions, pos_label=1)
test_recall_class_1 = recall_score(df_Y_test, test_predictions, pos_label=1)

print(f"Precision for having stroke on development data set using decision tree: {dev_precision_class_1}")
print(f"Recall for having stroke on development data set using decision tree: {dev_recall_class_1}")

print(f"Precision for having stroke on test data Set using decision tree: {test_precision_class_1}")
print(f"Recall for having stroke on test data set using decision tree: {test_recall_class_1}")
```

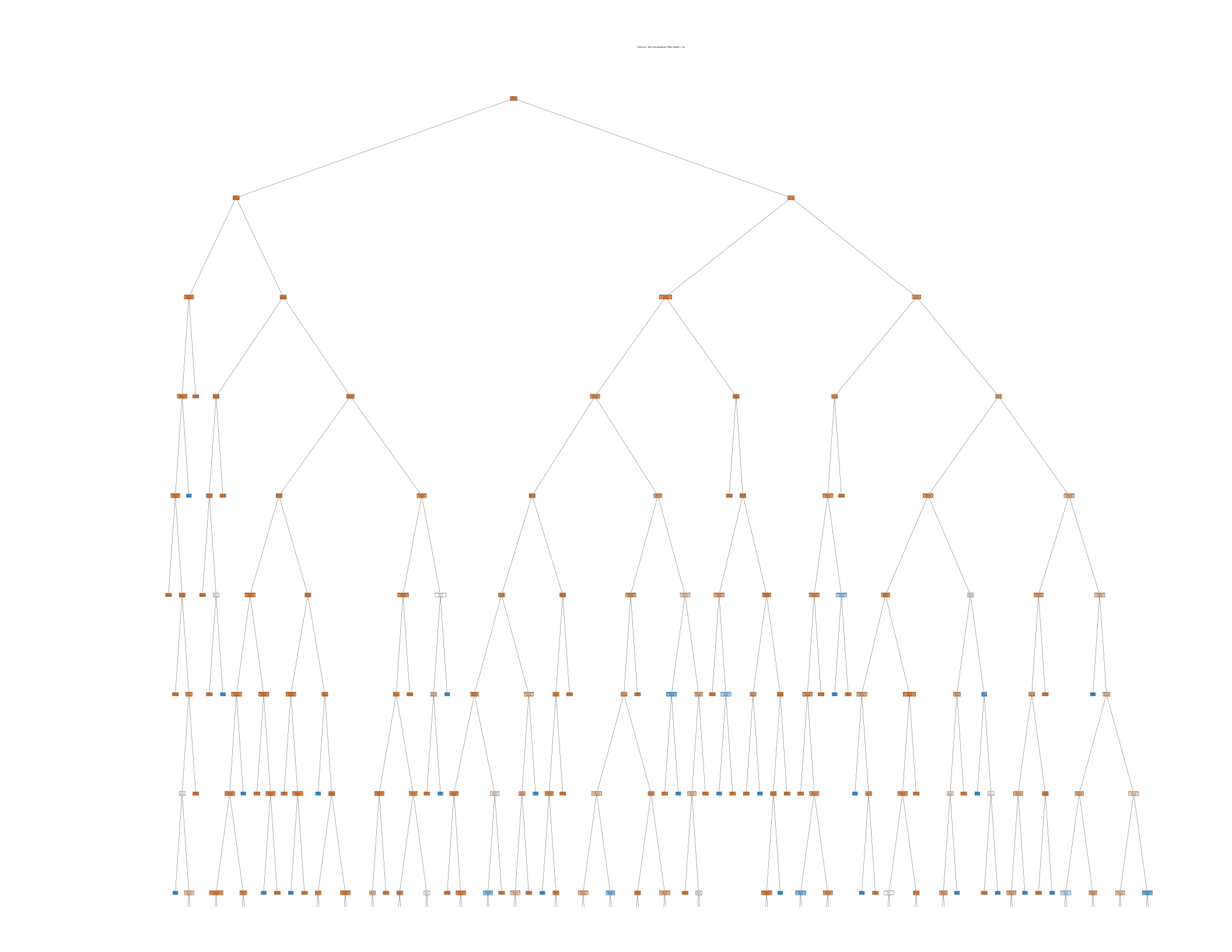
```
Precision for having stroke on development data set using decision tree: 1.0
Recall for having stroke on development data set using decision tree: 1.0
Precision for having stroke on test data Set using decision tree: 0.12244897959183673
Recall for having stroke on test data set using decision tree: 0.12
```

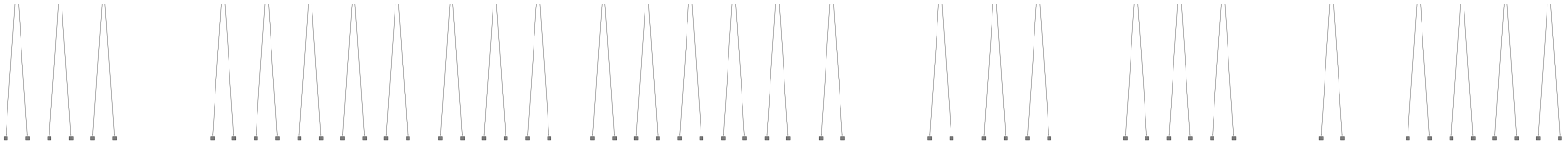
Your Comments Here

- Precision for having stroke on Development Set: 1.0
- Recall for having stroke on Development Set: 1.0
- Precision for having stroke on Test Set: 0.12244897959183673
- Recall for having stroke on Test Set: 0.12
- Given the highly imbalanced nature of the dataset, precision and recall with respect to the minority class (those having a stroke) stand out as essential metrics. In this assignment, precision with respect to having stroke will be the metrics used in cross validation for hyper-parameter tuning since this model is most likely to be used by the hospital. The reasoning for picking those metrics is shown below:
  - precision with respect to class 1: In this case, high precision indicates that when the model predicts someone has a stroke, it is likely to be correct. This is crucial because predicting someone has a stroke when they don't can lead to unnecessary stress, medical procedures, and costs. This is the most important metrics if the use case is in hospital or healthcare system.
  - recall with respect to class 1: High recall means the model is sensitive to identifying all individuals who had a stroke. It's important because missing a true stroke case can have serious consequences, so you want to minimize false negatives. Recall can be a critical metric when the model is intended for job applications where the job requirements necessitate individuals to be free from stroke.

1.8: Visualize the trained tree until the max_depth 8

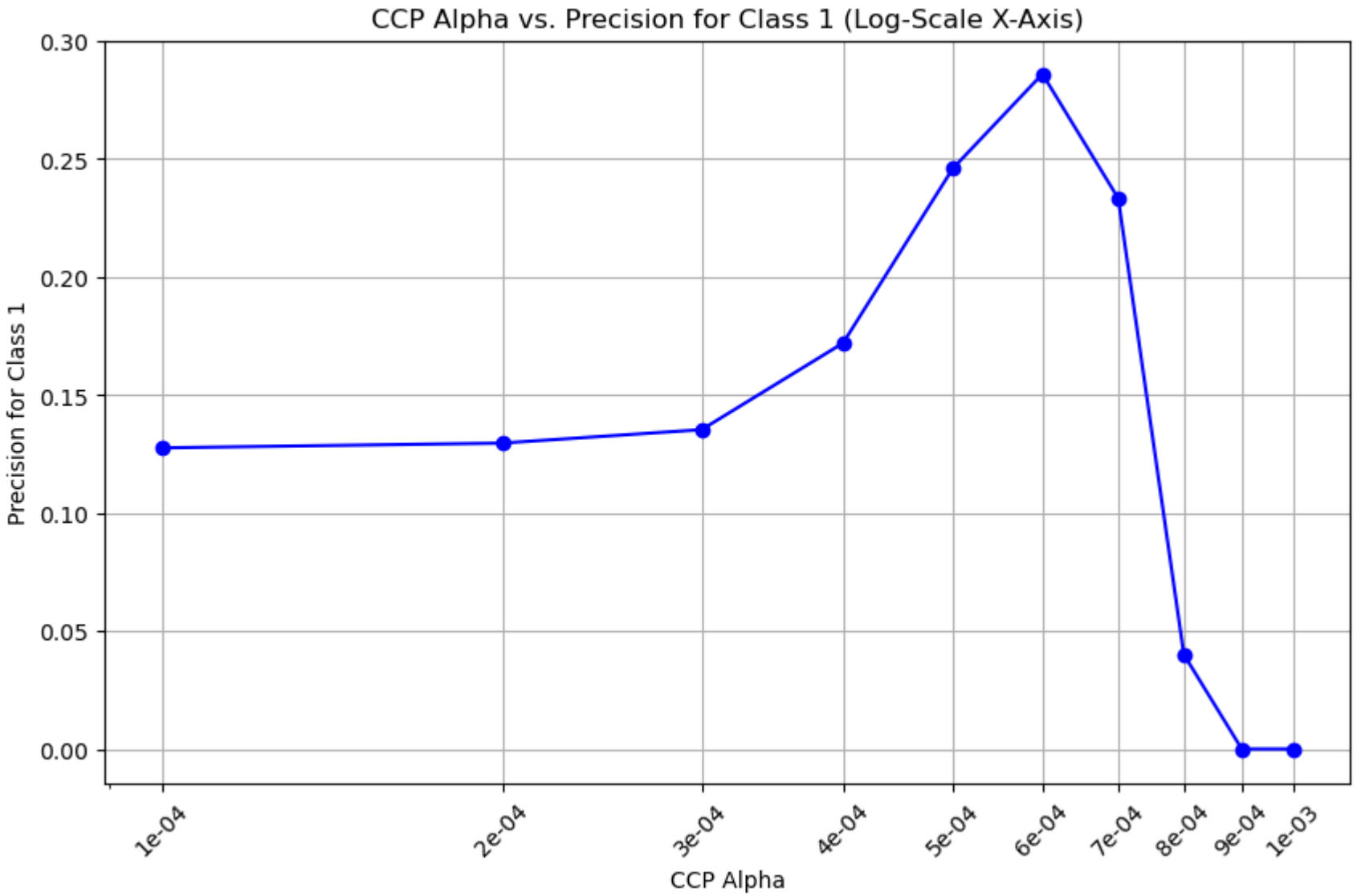
In [13]:
```
## YOUR CODE HERE
plt.figure(figsize=(100, 100))
plot_tree(tree_model, max_depth=8, filled=True, feature_names=df_X_dev.columns, class_names=["No Stroke", "Stroke"])
plt.title("Decision Tree Visualization (Max Depth = 8)")
plt.show()
```


Decision Tree Visualization (Max Depth = 8)



1.9: Prune the tree using one of the techniques discussed in class and evaluate the performance. Carefully consider which metric to use considering the imbalance in the class label.

In [14]:
```
ccp_alphas = [0.0001,0.0002, 0.0003, 0.0004, 0.0005, 0.0006, 0.0007, 0.0008, 0.0009, 0.001]
precision_scores = []
precision_scorer = make_scorer(precision_score, pos_label=1)
stratified_cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
for alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=42, ccp_alpha=alpha)
    precision = cross_val_score(clf, df_X_dev, df_Y_dev, cv=stratified_cv, scoring=precision_scorer)
    precision_scores.append(np.mean(precision))

plt.figure(figsize=(10, 6))
plt.semilogx(ccp_alphas, precision_scores, '-o', color='b')
plt.xlabel('CCP Alpha')
plt.ylabel('Precision for Class 1')
plt.title('CCP Alpha vs. Precision for Class 1 (Log-Scale X-Axis)')
plt.xticks(ccp_alphas, [f"{a:.0e}" for a in ccp_alphas], rotation=45)
plt.grid(True)
plt.show()
best_alpha_index = np.argmax(precision_scores)
best_alpha = ccp_alphas[best_alpha_index]
best_classifier = DecisionTreeClassifier(random_state=42, ccp_alpha=best_alpha)
best_classifier.fit(df_X_dev, df_Y_dev)
test_predictions = best_classifier.predict(df_X_test)
precision_class_1 = precision_score(df_Y_test, test_predictions, pos_label=1)
recall_class_1 = recall_score(df_Y_test, test_predictions, pos_label=1)

print(f"Best alpha: {best_alpha}")
print(f"Precision for having stroke on test data set using pruned decision tree: {precision_class_1}")
print(f"Recall for having stroke on test data set using pruned decision tree: {recall_class_1}")
```




```
Best alpha: 0.0006
Precision for having stroke on test data set using pruned decision tree: 0.25
Recall for having stroke on test data set using pruned decision tree: 0.02
```

Your Comments Here

- I used cost complexity method to do tree pruning, and the best CCP Alpha in this case is 0.0006. Precision for having stroke on Test Set using pruned tree: 0.25. Recall for having stroke on Test Set using pruned tree: 0.02.

- I used precision for having stroke as the metrics for cross validation to choose the best CCP Alpha. The reason is stated as followings:
  - [1] The data is highly imbalanced.
  - [2] Since predicting someone has a stroke when they don't can lead to unnecessary stress, medical procedures, and costs, I decide to use precision for having stroke.

1.10: List the top 3 most important features for this trained tree? How would you justify these features being the most important?

In [15]:
```
## YOUR CODE HERE
feat_imps = zip(df_X_dev.columns, best_classifier.feature_importances_)
feats, imps = zip(*(sorted(list(filter(lambda x: x[1]!=0, feat_imps)),key=lambda x: x[1], reverse=True)))
print(f"Top 3 most important features are {feats[:3]}")
```

```
Top 3 most important features are ('age', 'avg_glucose_level', 'bmi')
```

Your Comments Here

- Top 3 most important features are ('age', 'avg_glucose_level', 'bmi'). Justifications are shown below:
  - [1] age
    - Older individuals are generally at a higher risk of stroke. Thus, age will be an important feature in predicting whether have stoke.
  - [2] avg_glucose_level
    - Elevated blood glucose levels are associated with diabetes, which is an important risk factor for stroke. Thus, avg_glucose_level will be an important feature in predicting whether have stroke.
  - [3] bmi
    - High BMI values are often associated with obesity, which is an important risk factor for stroke. Thus, bmi will be an important feature in predicting whether have stroke.

Question 2: Random Forests

2.1: Train a Random Forest model on the development dataset using RandomForestClassifier class in sklearn. Use the default parameters. Evaluate the performance of the model on test dataset. Does this perform better than Decision Tree on the test dataset (compare to results in Q 1.6)?

In [16]:
```
## YOUR CODE HERE
rf_model = RandomForestClassifier()
rf_model.fit(df_X_dev, df_Y_dev)
y_pred = rf_model.predict(df_X_test)
precision_rf_class_1 = precision_score(df_Y_test, y_pred, pos_label=1)
recall_rf_class_1 = recall_score(df_Y_test, y_pred, pos_label=1)
print(f"Precision for having stroke on test data set using random forest: {precision_rf_class_1}")
print(f"Recall for having stroke on test data set using random forest: {recall_rf_class_1}")
```

```
Precision for having stroke on test data set using random forest: 0.0
Recall for having stroke on test data set using random forest: 0.0
```

Your Comments Here

- No, the random forest with default hyper parameters does not perform better than the decision tree on the test datasets in terms of both precision_having_stroke and recall_having_stroke. The lack of hyperparameter tuning in the Random Forest model likely results in overfitting, contributing to this performance disparity.

2.2: Does all trees in the trained random forest model have pure leaves? How would you verify this?

In [17]:
```
#calculating the sum of gini index score across all leave nodes of all individual trees
total_gini_index = 0
for estimator in rf_model.estimators_:
    leaf_indices = estimator.apply(df_X_dev)
    sum_gini_indices_leaf = estimator.tree_.impurity[leaf_indices].sum()
    total_gini_index += sum_gini_indices_leaf
print(f"Sum of gini index score across all leave nodes of all individual trees is {total_gini_index}")
```

```
Sum of gini index score across all leave nodes of all individual trees is 0.0
```

Your Comments Here:

- Yes, all trees in the trained random forest model have pure leaves. This can be verified by the sum of gini index score across all leave nodes of all individual trees is 0.

2.3: Assume you want to improve the performance of this model. Also, assume that you had to pick two hyperparameters that you could tune to improve its performance. Which hyperparameters would you choose and why?

Your Comments Here

- two hyperparameters that I will choose will be max_depth and n_estimators, the reason is shown below:

- [1] max_depth: from 2.1 and 2.2, we find the the sum of gini index score across all leave nodes of all individual trees is 0, and the precision and recall for having stroke on test data set is 0. These information indicates that the individual tree is probablity overfitting. Thus, it will be good to limit the max_depth of each individual trees.
- [2] n_estimators: Increasing the number of trees can help the model generalize better and reduce overfitting.

2.4: Now, assume you had to choose up to 5 different values (each) for these two hyperparameters. How would you choose these values that could potentially give you a performance lift?

In [18]:
```
#getting the individual tree depth
estimators = rf_model.estimators_
tree_depths = []
for estimator in estimators:
    depth = estimator.tree_.max_depth
    tree_depths.append(depth)
print(f"Tree depths for each individual tree in random forest model: {tree_depths}")
```

```
Tree depths for each individual tree in random forest model: [19, 17, 20, 22, 17, 20, 18, 18, 16, 20, 16, 19, 23, 17, 1
7, 21, 22, 17, 17, 18, 17, 15, 19, 20, 18, 18, 17, 18, 19, 17, 21, 18, 17, 21, 20, 20, 16, 17, 21, 18, 17, 18, 17, 19,
23, 17, 17, 18, 16, 18, 18, 18, 19, 16, 17, 17, 20, 17, 19, 21, 17, 21, 23, 18, 18, 20, 18, 17, 18, 19, 17, 17, 18, 17,
19, 18, 18, 19, 18, 19, 18, 20, 19, 18, 17, 19, 18, 18, 25, 18, 18, 16, 18, 16, 18, 18, 17, 19, 18, 16]
```

Your Comments Here

- max_depth [2,5,10,17,20]: After analyzing the tree depths of individual trees within the default Random Forest model, it's evident that max_depth ranges from 15 to 25. Given the observed overfitting in the default model, I propose a strategy to explore max_depth lower than 25, specifically 2, 5, 10, 17, and 20. This selection allows for an investigation of a range of tree depths, from shallow trees to moderately deep ones. The objective is to find the optimal max_depth setting that strikes a balance between model complexity and generalization.
- n_estimators [200, 400, 600, 800, 1000]: Considering that we may deal with very deep trees, such as a maximum depth of 14, it's prudent to counterbalance this by increasing the number of individual trees in the Random Forest ensemble. This strategy aims to enhance generalization and mitigate the risk of overfitting. Since the default Random Forest model with n_estimators = 100 performs bad, it is advisable to consider adding more trees, exceeding 100. Thus, I will try out 200, 400, 600, 800, 1000 for n_estimators.

2.5: Perform model selection using the chosen values for the hyperparameters. Use cross-validation for finding the optimal hyperparameters. Report on the optimal hyperparameters. Estimate the performance of the optimal model (model trained

with optimal hyperparameters) on test dataset? Has the performance improved over your plain-vanilla random forest model trained in Q2.1?

In [19]:
```
## YOUR CODE HERE
param_grid = {
    'max_depth': [2, 5, 10, 17, 20],
    'n_estimators': [200, 400, 600, 800, 1000],
}
precision_scorer = make_scorer(precision_score, pos_label=1)
rf_classifier = RandomForestClassifier(random_state=42)
stratified_cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
grid_search = GridSearchCV(estimator=rf_classifier, param_grid=param_grid, cv=stratified_cv, scoring=precision_scorer,
grid_search.fit(df_X_dev, df_Y_dev)
print("Best Hyperparameters:", grid_search.best_params_)
best_rf_classifier = RandomForestClassifier(**grid_search.best_params_)
best_rf_classifier.fit(df_X_dev, df_Y_dev)
y_pred = best_rf_classifier.predict(df_X_test)
precision_score_class_1 = precision_score(df_Y_test, y_pred, pos_label=1)
recall_score_class_1 = recall_score(df_Y_test, y_pred, pos_label=1)
print(f"Precision for having stroke on test data set using random forest: {precision_score_class_1}")
print(f"Recall for having stroke on test data set using random forest: {recall_score_class_1}")
```

```
Best Hyperparameters: {'max_depth': 17, 'n_estimators': 600}
Precision for having stroke on test data set using random forest: 0.0
Recall for having stroke on test data set using random forest: 0.0
```

Your Comments Here

- Best Hyperparameters: {'max_depth': 17, 'n_estimators': 600}
- Precision for having stroke on test data set using random forest: 0.0
- Recall for having stroke on test data set using random forest: 0.0
- No, after hyper-parameter tuning, the performance of random forest is still 0 in terms of both Precision for having stroke and Recall for having stroke.

2.6: Can you find the top 3 most important features from the model trained in Q2.5? How do these features compare to the important features that you found from Q1.9? If they differ, which feature set makes more sense?

In [20]:
```
## YOUR CODE HERE
feat_imps = zip(df_X_dev.columns, best_rf_classifier.feature_importances_)
feats, imps = zip(*(sorted(list(filter(lambda x: x[1]!=0, feat_imps)),key=lambda x: x[1], reverse=True)))
print(f"Top 3 most important features are {feats[:3]}")
```

```
Top 3 most important features are ('avg_glucose_level', 'age', 'bmi')
```

Your Comments Here

- The top 3 most important features from the model trained in Q2.5: 'avg_glucose_level', 'age', 'bmi'.
- These features are the same as those foudn in Q1.9. However, the rank of top 3 features are different. In Q1.9, 'age' is priotized over 'avg_glucose_level'. However, in Q2.5, 'avg_glucose_level' is priotized over 'age'.
- I feel the top 3 feature rank in Q2.5 makes more sense. In this regard, 'avg_glucose_level' appears to be a more critical predictor than 'age.' This conclusion is grounded in the understanding that health factors, particularly elevated 'avg_glucose_level,' are potentially more directly linked to the occurrence of a stroke. While age can certainly be a contributing factor, it is often correlated with other health issues, such as high average glucose levels, which are the primary triggers for strokes. Therefore, emphasizing 'avg_glucose_level' as a prominent feature in stroke prediction models is a plausible and sensible choice.

Question 3: Gradient Boosted Trees

3.1: Choose three hyperparameters to tune GradientBoostingClassifier and HistGradientBoostingClassifier on the development dataset using 5-fold cross validation.
You can use GridSearchCV, however make sure to use appropriate metric for the scoring parameter of GridSearchCV.
Report on the time taken to do model selection for both the models. Also, report the performance of the test dataset from the optimal models.

In [21]:
```
## YOUR CODE HERE
param_grid = {
    'max_depth': [3, 10, 15, 20, 25],
    'n_estimators': [100, 200, 300, 400, 500],
    'learning_rate': [0.00001, 0.0001, 0.001, 0.01, 0.1]
}
precision_scorer = make_scorer(precision_score, pos_label=1)
gb_classifier = GradientBoostingClassifier(random_state=42)
grid_search_gb = GridSearchCV(estimator=gb_classifier, param_grid=param_grid, cv=5, scoring=precision_scorer, n_jobs =-
start_time_gb = time.time()
grid_search_gb.fit(df_X_dev, df_Y_dev)
end_time_gb = time.time()
time_taken_gb = end_time_gb - start_time_gb
best_hyperparameters_gb = grid_search_gb.best_params_
print("GradientBoostingClassifier - Best Hyperparameters:", best_hyperparameters_gb)
print("Time Taken for Model Selection (seconds):", time_taken_gb)
best_gb_classifier = GradientBoostingClassifier(**best_hyperparameters_gb)
```

```
best_gb_classifier.fit(df_X_dev, df_Y_dev)
y_pred = best_gb_classifier.predict(df_X_test)
precision_score_class_1 = precision_score(df_Y_test, y_pred, pos_label=1)
recall_score_class_1 = recall_score(df_Y_test, y_pred, pos_label=1)
print(f"Precision for having stroke on test data set using GradientBoostingClassifier: {precision_score_class_1}")
print(f"Recall for having stroke on test data set using GradientBoostingClassifier: {recall_score_class_1}")
```

```
GradientBoostingClassifier - Best Hyperparameters: {'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 500}
Time Taken for Model Selection (seconds): 407.41118001937866
Precision for having stroke on test data set using GradientBoostingClassifier: 0.375
Recall for having stroke on test data set using GradientBoostingClassifier: 0.06
```

In [22]:

```
## YOUR CODE HERE
param_grid = {
    'max_depth': [3, 10, 15, 20, 25],
    'max_iter': [100, 200, 300, 400, 500],
    'learning_rate': [0.00001, 0.0001, 0.001, 0.01, 0.1]
}
precision_scorer = make_scorer(precision_score, pos_label=1)
hgb_classifier = HistGradientBoostingClassifier()
grid_search_hgb = GridSearchCV(estimator=hgb_classifier, param_grid=param_grid, cv=5, scoring=precision_scorer, n_jobs
start_time_hgb = time.time()
grid_search_hgb.fit(df_X_dev, df_Y_dev)
end_time_hgb = time.time()
time_taken_hgb = end_time_hgb - start_time_hgb
best_hyperparameters_hgb = grid_search_hgb.best_params_
best_precision_hgb = grid_search_hgb.best_score_
print("HistGradientBoostingClassifier - Best Hyperparameters:", best_hyperparameters_hgb)
print("Time Taken for Model Selection (seconds):", time_taken_hgb)
best_hgb_classifier = HistGradientBoostingClassifier(**best_hyperparameters_hgb)
best_hgb_classifier.fit(df_X_dev, df_Y_dev)
y_pred = best_hgb_classifier.predict(df_X_test)
precision_score_class_1 = precision_score(df_Y_test, y_pred, pos_label=1)
recall_score_class_1 = recall_score(df_Y_test, y_pred, pos_label=1)
print(f"Precision for having stroke on test data set using HistGradientBoostingClassifier: {precision_score_class_1}")
print(f"Recall for having stroke on test data set using HistGradientBoostingClassifier: {recall_score_class_1}")
```

```
HistGradientBoostingClassifier - Best Hyperparameters: {'learning_rate': 0.01, 'max_depth': 15, 'max_iter': 300}
Time Taken for Model Selection (seconds): 131.10152220726013
Precision for having stroke on test data set using HistGradientBoostingClassifier: 0.25
Recall for having stroke on test data set using HistGradientBoostingClassifier: 0.02
```

Your Comments Here :

Model	Test_Precision_having_stroke	Test_Recall_having_stroke	Time_Taken_for_model_selection
HistGradientBoostingClassifier	0.25	0.02	131.10 seconds
GradientBoostingClassifier	0.375	0.06	407.41 seconds

3.2: Train an XGBoost model by tuning 3 hyperparameters using 10 fold cross-validation. Compare the performance of the trained XGBoost model on the test dataset against the performances obtained from 3.1

In [23]:
```
param_grid = {
    'max_depth': [3, 10, 15, 20, 25],
    'n_estimators': [100, 200, 300, 400, 500],
    'learning_rate': [0.00001, 0.0001, 0.001, 0.01, 0.1]
}
precision_scorer = make_scorer(precision_score, pos_label=1)
xgb_classifier = xgb.XGBClassifier(random_state=42)
grid_search_xgb = GridSearchCV(estimator=xgb_classifier, param_grid=param_grid, cv=10, scoring=precision_scorer, n_jobs
start_time_xgb = time.time()
grid_search_xgb.fit(df_X_dev, df_Y_dev)
end_time_xgb = time.time()
time_taken_xgb = end_time_xgb - start_time_xgb
best_hyperparameters_xgb = grid_search_xgb.best_params_
best_precision_xgb = grid_search_xgb.best_score_
print("XGBClassifier - Best Hyperparameters:", best_hyperparameters_xgb)
print("Time Taken for Model Selection (seconds):", time_taken_xgb)
best_xgb_classifier = xgb.XGBClassifier(**best_hyperparameters_xgb)
best_xgb_classifier.fit(df_X_dev, df_Y_dev)
y_pred = best_xgb_classifier.predict(df_X_test)
precision_score_class_1 = precision_score(df_Y_test, y_pred, pos_label=1)
recall_score_class_1 = recall_score(df_Y_test, y_pred, pos_label=1)
print(f"Precision for having stroke on test data set using XGBClassifier: {precision_score_class_1}")
print(f"Recall for having stroke on test data set using XGBClassifier: {recall_score_class_1}")
```

```
XGBClassifier - Best Hyperparameters: {'learning_rate': 0.001, 'max_depth': 15, 'n_estimators': 400}
Time Taken for Model Selection (seconds): 367.5286421775818
Precision for having stroke on test data set using XGBClassifier: 0.2222222222222222
Recall for having stroke on test data set using XGBClassifier: 0.04
```

Your Comments Here :

Model	Test_Precision_having_stroke	Test_Recall_having_stroke	Time_Taken_for_model_selection
XGBoost	0.22	0.04	367.53 seconds

Model	Test_Precision_having_stroke	Test_Recall_having_stroke	Time_Taken_for_model_selection
HistGradientBoostingClassifier	0.25	0.02	131.10 seconds
GradientBoostingClassifier	0.375	0.06	407.41 seconds

- In terms of time taken for model selection, HistGradientBoostingClassifier performs best.
- In terms of Test_Precision_having_stroke, GradientBoostingClassifier performs best.
- In terms of Test_Recall_having_stroke, GradientBoostingClassifier performs best.

3.3: Can you list the top 3 features from the trained XGBoost model? How do they differ from the features found from Random Forest and Decision Tree? Which one would you trust the most?

In [24]:
```
## YOUR CODE HERE
feat_imps = zip(df_X_dev.columns, best_xgb_classifier.feature_importances_)
print(f"Top 3 most important features are {feats[:3]}")
```

```
Top 3 most important features are ('avg_glucose_level', 'age', 'bmi')
```

Your Comments Here

- top 3 features from XGBoost model is the same as Random Forest and Decision Tree. Rank of top 3 features from XGBoost model is the same as Random Forest while different from the decision tree. In Decision Tree, 'age' is priotized over 'avg_glucose_level'. In XGBoost and Random Forest, 'avg_glucose_level' is priotized over 'age'.
- I feel the feature set rank of XGBoost and Random Forest. In this regard, 'avg_glucose_level' appears to be a more critical predictor than 'age.' This conclusion is grounded in the understanding that health factors, particularly elevated 'avg_glucose_level,' are potentially more directly linked to the occurrence of a stroke. While age can certainly be a contributing factor, it is often correlated with other health issues, such as high average glucose levels, which are the primary triggers for strokes. Therefore, emphasizing 'avg_glucose_level' as a prominent feature in stroke prediction models is a plausible and sensible choice.

3.4: Can you choose the top 7 features (as given by feature importances from XGBoost) and repeat Q3.2? Does this model perform better than the one trained in Q3.2? Why or why not is the performance better?

In [25]:
```
print(f"Top 7 most important features are {feats[:7]}")
```

```
Top 7 most important features are ('avg_glucose_level', 'age', 'bmi', 'Residence_type_Urban', 'gender_Male', 'hypertens
ion_1', 'work_type_Private')
```

In [26]:
```
## YOUR CODE HERE
param_grid = {
    'max_depth': [3, 10, 15, 20, 25],
    'n_estimators': [100, 200, 300, 400, 500],
    'learning_rate': [0.00001, 0.0001, 0.001, 0.01, 0.1]
}
selected_features = list(feats[:7])
precision_scorer = make_scorer(precision_score, pos_label=1)
xgb_classifier = xgb.XGBClassifier(random_state=42)
grid_search_xgb = GridSearchCV(estimator=xgb_classifier, param_grid=param_grid, cv=10, scoring=precision_scorer, n_jobs
start_time_xgb = time.time()
grid_search_xgb.fit(df_X_dev[selected_features], df_Y_dev)
end_time_xgb = time.time()
time_taken_xgb = end_time_xgb - start_time_xgb
best_hyperparameters_xgb = grid_search_xgb.best_params_
best_precision_xgb = grid_search_xgb.best_score_
print("XGBClassifier - Best Hyperparameters:", best_hyperparameters_xgb)
print("Time Taken for Model Selection (seconds):", time_taken_xgb)
best_xgb_classifier = xgb.XGBClassifier(**best_hyperparameters_xgb)
best_xgb_classifier.fit(df_X_dev[selected_features], df_Y_dev)
y_pred = best_xgb_classifier.predict(df_X_test[selected_features])
precision_score_class_1 = precision_score(df_Y_test, y_pred, pos_label=1)
recall_score_class_1 = recall_score(df_Y_test, y_pred, pos_label=1)
print(f"Precision for having stroke on test data set using XGBClassifier: {precision_score_class_1}")
print(f"Recall for having stroke on test data set using XGBClassifier: {recall_score_class_1}")
```

```
XGBClassifier - Best Hyperparameters: {'learning_rate': 0.001, 'max_depth': 15, 'n_estimators': 500}
Time Taken for Model Selection (seconds): 244.03818345069885
Precision for having stroke on test data set using XGBClassifier: 0.4
Recall for having stroke on test data set using XGBClassifier: 0.04
```

Your comments here

- Top 7 most important features are ('avg_glucose_level', 'age', 'bmi', 'Residence_type_Urban', 'gender_Male', 'hypertension_1', 'work_type_Private')
- Yes, The model with Top 7 most important features performances better than usig the full datasets because
  - [1] Reduce Overfitting: Training the XGBoost using the full features may result in overfiting, and less relevant features can introduce noise into the model. By removing insignificant features, we can reduce the model complexity to avoid overfitting and limit the noise captured by the model.
  - [2] Reduce Multicollinearity: There may be some high correlation between the top 7 features and the remaining features. By removing those less relevent features, we are able to remove these high correlatted pairs of features that confuse the model.

- [3] Improved Generalization: By using fewer features, the model may generalize better to unseen data. This is because it is less likely to learn the peculiarities of the training data and is more likely to capture the underlying patterns that apply to a broader range of instances.

3.5: Compare the results on the test dataset from XGBoost, HistGradientBoostingClassifier, GradientBoostingClassifier with results from Q1.6 and Q2.1. Which model tends to perform the best and which one does the worst? How big is the difference between the two? Which model would you choose among these 5 models and why?

Model	Precision_having_stroke	Recall_having_stroke
Decision Tree	0.25	0.02
Random Forest	0	0
XGBoost	0.4	0.04
HistGradientBoostingClassifier	0.25	0.02
GradientBoostingClassifier	0.375	0.06

- GradientBoostingClassifier performs best in terms of Recall_having_stroke.
- XGBoost performs best in terms of Precision_having_stroke.
- Random Forest performs worst in terms of both Precision_having_stroke and Recall_having_stroke.
- XGBoost outperforms random forest in terms of Precision_having_stroke by 0.4.
- GradientBoostingClassifier outperforms random forest in terms of Recall_having_stroke by 0.06.
- Since predicting someone has a stroke when they don't can lead to unnecessary stress, medical procedures, and costs, precision_having_stoke will be the most important metrics. I will choose XGBoost Since it has the highest Precision_having_stroke on test data set.

Question 4: Calibration

4.1: Estimate the brier score for the XGBoost model (trained with optimal hyperparameters from Q3.2) scored on the test dataset.

In [31]:
```
## YOUR CODE HERE
best_xgb_classifier = best_xgb_classifier.fit(df_X_dev, df_Y_dev)
y_pred_proba = best_xgb_classifier.predict_proba(df_X_test)[:, 1]
```

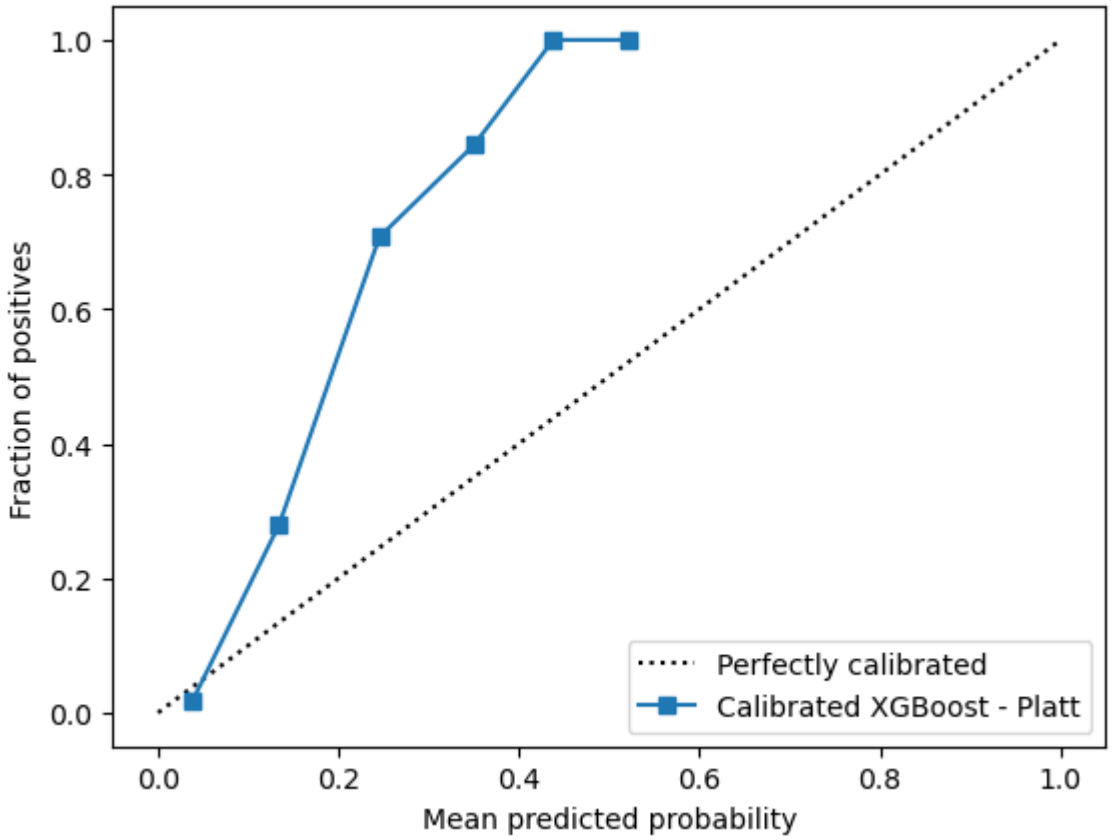
```
brier_score = brier_score_loss(df_Y_test, y_pred_proba)
print(brier_score)
```

```
0.1191626159681238
```

4.2: Calibrate the trained XGBoost model using isotonic regression as well as Platt scaling. Plot predicted v.s. actual on test datasets from both the calibration methods. Report brier score after calibration.

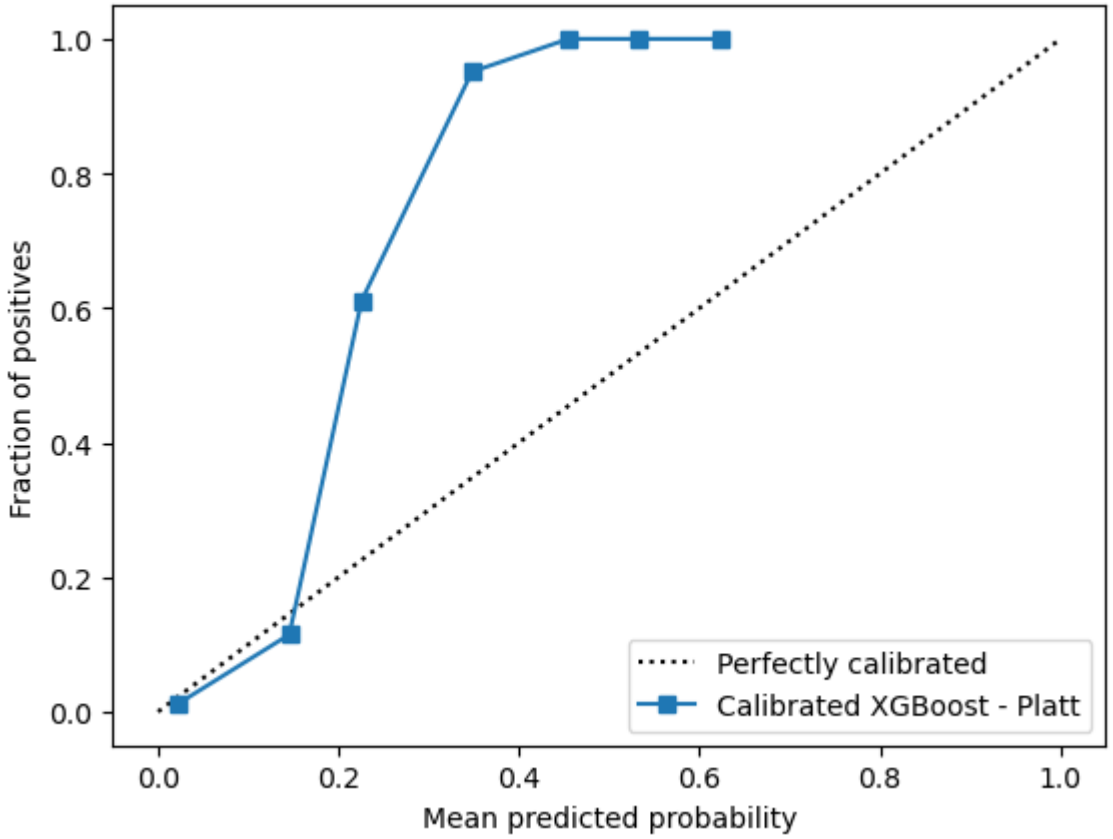
In [36]:
```
## YOUR CODE HERE
cal_platt = CalibratedClassifierCV(best_xgb_classifier, cv=5, method='sigmoid')
cal_platt.fit(df_X_dev, df_Y_dev)
display = CalibrationDisplay.from_estimator(cal_platt, df_X_dev, df_Y_dev, n_bins = 10, name = "Calibrated XGBoost - Pl
y_pred_proba = cal_platt.predict_proba(df_X_dev)[:, 1]
brier_score_platt_scaling = brier_score_loss(df_Y_dev, y_pred_proba)
print(f"brier_score for Platt scaling: {brier_score_platt_scaling}")
```

```
brier_score for Platt scaling: 0.03484430187783169
```



In [33]:
```
cal_platt = CalibratedClassifierCV(best_xgb_classifier, cv=5, method='isotonic')
cal_platt.fit(df_X_dev, df_Y_dev)
display = CalibrationDisplay.from_estimator(cal_platt, df_X_dev, df_Y_dev, n_bins = 10, name = "Calibrated XGBoost - Pl
y_pred_proba = cal_platt.predict_proba(df_X_dev)[:, 1]
brier_score_isotonic = brier_score_loss(df_Y_dev, y_pred_proba)
print(f"brier_score for isotonic regression: {brier_score_isotonic}")
```

```
brier_score for isotonic regression: 0.035384027859951775
```



4.3: Report brier scores from both the calibration methods. Do the calibration methods help in having better predicted probabilities?

In [30]:
```
## YOUR CODE HERE
print(f"brier_score without calibration: {brier_score}")
print(f"brier_score with Platt scaling calibration: {brier_score_platt_scaling}")
print(f"brier_score with isotonic regression calibration: {brier_score_isotonic}")
```

```
brier_score without calibration: 0.1191626159681238
brier_score with Platt scaling calibration: 0.03484430187783169
brier_score with isotonic regression calibration: 0.035384027859951775
```

Your comments here

- Yes. Calibration methods help in having better predicted probabilities. Both Platt scaling and isotonic regression have demonstrated their ability to reduce the Brier score compared to the uncalibrated model.