Stroke Prediction using Machine Learning Algorithm

```
# Importing necessary libraries

import pandas as pd
import numpy as np
import seaborn as sns
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score, roc_auc_score, roc_curve, precision_score,
recall_score, f1_score
import matplotlib.pyplot as plt
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from collections import Counter
from sklearn.ensemble import GradientBoostingClassifier
import warnings
warnings.filterwarnings("ignore")

# Loading the dataset
df = pd.read_csv('ModelFinalDataSet_normalised.csv')

# checking the dataset
print(df.head())
```

```
   Unnamed: 0    gender       age  hypertension  heart_disease
ever_married  \
0           3  1.198071  1.734960      0.283054      -4.953880
0.778368
1           7  1.198071  1.468824     -3.532094      -4.953880
0.778368
2           8 -0.832918  1.247044      0.283054       0.201816     -
1.284448
3           9 -0.832918  0.803483      0.283054       0.201816
0.778368
4          10 -0.832918  1.646248      0.283054       0.201816
0.778368

   work_type  Residence_type  avg_glucose_level       bmi    stroke
0   0.432417       -1.015944           0.637323  0.706801 -5.062062
1   0.432417       -1.015944          -0.943751 -0.064944 -5.062062
```

```
2   0.432417        0.984082           0.128537 -0.761028 -5.062062
3   0.432417       -1.015944          -0.676341  0.553924 -5.062062
4   0.432417        0.984082          -1.452095 -0.549176 -5.062062

# Checking the shape of the dataset
print(f"Dataset shape: {df.shape}")

# Checking for missing values
print("\nMissing values in each column:")
print(df.isnull().sum())

Dataset shape: (4394, 11)

Missing values in each column:
Unnamed: 0           0
gender               0
age                  0
hypertension         0
heart_disease        0
ever_married         0
work_type            0
Residence_type       0
avg_glucose_level    0
bmi                  0
stroke               0
dtype: int64

# Replacing values in the stroke column
df['stroke'] = np.where(df['stroke'] > -5, 0, 1)

# Verifying the changes
print("Class distribution before applying SMOTE:")
print(df['stroke'].value_counts())

Class distribution before applying SMOTE:
0    4229
1     165
Name: stroke, dtype: int64

# Dropping first column as id is not necessary
df = df.iloc[:, 1:]

df.head(5)

     gender       age  hypertension  heart_disease  ever_married
work_type  \
0  1.198071  1.734960      0.283054      -4.953880      0.778368
0.432417
1  1.198071  1.468824     -3.532094      -4.953880      0.778368
0.432417
2 -0.832918  1.247044      0.283054       0.201816     -1.284448
```

```
0.432417
3 -0.832918  0.803483      0.283054       0.201816      0.778368
0.432417
4 -0.832918  1.646248      0.283054       0.201816      0.778368
0.432417

   Residence_type  avg_glucose_level       bmi  stroke
0       -1.015944           0.637323  0.706801       1
1       -1.015944          -0.943751 -0.064944       1
2        0.984082           0.128537 -0.761028       1
3       -1.015944          -0.676341  0.553924       1
4        0.984082          -1.452095 -0.549176       1
```

```
# Separating features and target
X = df.drop('stroke', axis=1)
y = df['stroke']

# Splitting into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.3,
    random_state=42,
    stratify=y
)

print("\nTraining set class distribution before SMOTE:")
print(y_train.value_counts())
```

```
Training set class distribution before SMOTE:
0    2960
1     115
Name: stroke, dtype: int64
```

```
# Applying SMOTE ONLY to the training data
smote = SMOTE(random_state=42)
X_train_resmpled, y_train_resampled = smote.fit_resample(X_train,
y_train)

print("\nTraining set class distribution after SMOTE:")
print(pd.Series(y_train_resampled).value_counts())
```

```
Training set class distribution after SMOTE:
1    2960
0    2960
Name: stroke, dtype: int64
```

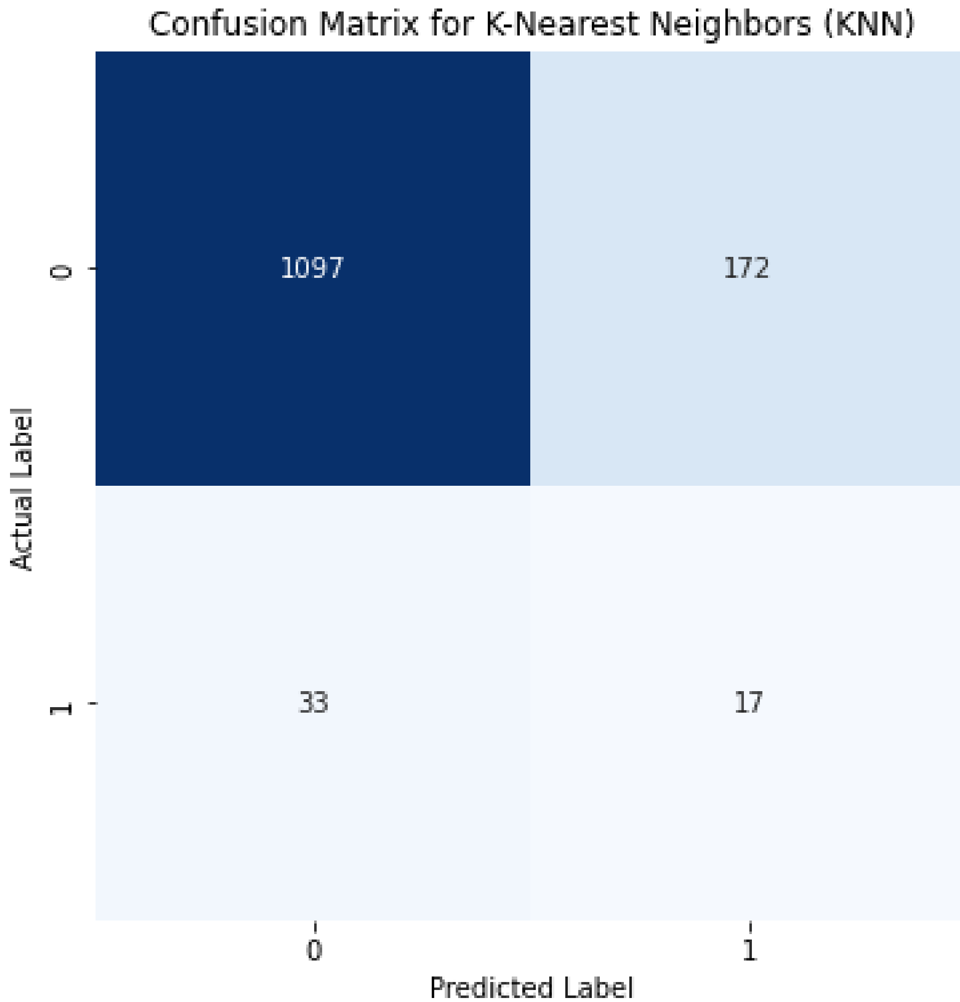
```
# Feature scaling
scaler = StandardScaler()
```

```
X_train_scaled = scaler.fit_transform(X_train_resmpled)
X_test_scaled = scaler.transform(X_test)

# function for model evaluation
def evaluate_model(y_test, y_pred, model_name):
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    auc = roc_auc_score(y_test, y_pred)

    print(f"\n{model_name} Performance:")
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1-score: {f1:.4f}")
    print(f"AUC-ROC: {auc:.4f}")

    return {
        "Model": model_name,
        "Accuracy": accuracy,
        "Precision": precision,
        "Recall": recall,
        "F1-score": f1,
        "AUC-ROC": auc
    }
```

Applying KNN Algorithm

```
# Basic KNN model
knn_model = KNeighborsClassifier()
knn_model.fit(X_train_scaled, y_train_resampled)
y_pred_knn = knn_model.predict(X_test_scaled)

# Model Evaluation
knn_performance = evaluate_model(y_test, y_pred_knn, "K-Nearest 
Neighbors (KNN)")


K-Nearest Neighbors (KNN) Performance:
Accuracy: 0.8446
Precision: 0.0899
Recall: 0.3400
F1-score: 0.1423
AUC-ROC: 0.6022

# Confusion matrix of KNN
labels = [0, 1]
cm = confusion_matrix(y_test, y_pred_knn, labels=labels)

# Plot the heatmap
```

```
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
xticklabels=labels,
            yticklabels=labels, square=True, vmin=0, vmax=np.max(cm))

plt.title('Confusion Matrix for K-Nearest Neighbors (KNN)')
plt.ylabel('Actual Label')
plt.xlabel('Predicted Label')
plt.show()
```



Hyperparameter Tuning in KNN

```
# defining parameter grid for KNN

param_grid  = {
    'n_neighbors': range(1, 31),
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan', 'minkowski']
```

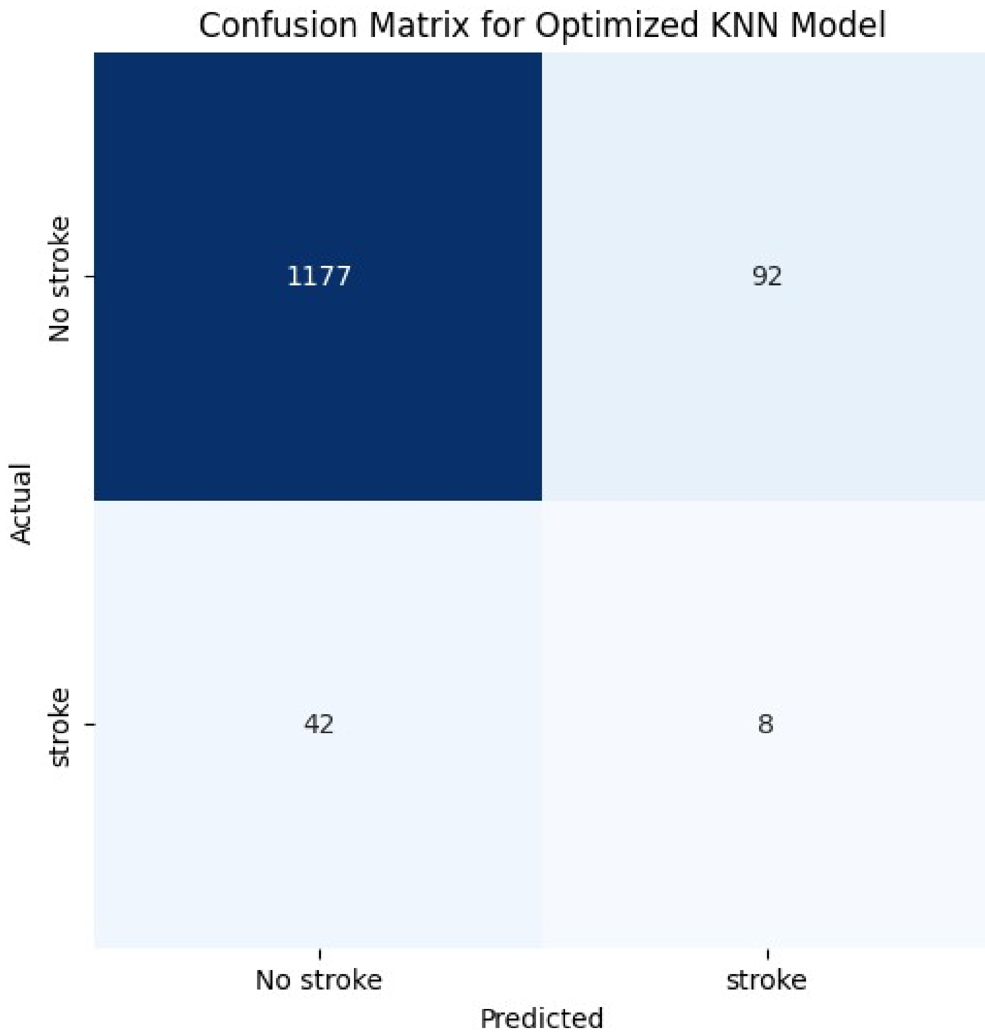
```
}

# Applying RandomizedSearchCV
grid_search = GridSearchCV(knn_model, param_grid, cv=5,
scoring='accuracy', n_jobs=-1)
grid_search.fit(X_train_scaled, y_train_resampled)


# Getting best parameters
best_params = grid_search.best_params_
print("Best Parameters:", best_params)
```

```
Best Parameters: {'metric': 'manhattan', 'n_neighbors': 1, 'weights':
'uniform'}
```

```
# Training KNN model with best parameters
best_knn_model = KNeighborsClassifier(**best_params)
best_knn_model.fit(X_train_scaled, y_train_resampled)

# Predictions
y_pred_knn_best = best_knn_model.predict(X_test_scaled)

# Model Evaluation
best_knn_performance = evaluate_model(y_test, y_pred_knn_best,
"Optimized KNN")
```

```
Optimized KNN Performance:
Accuracy: 0.8984
Precision: 0.0800
Recall: 0.1600
F1-score: 0.1067
AUC-ROC: 0.5438
```

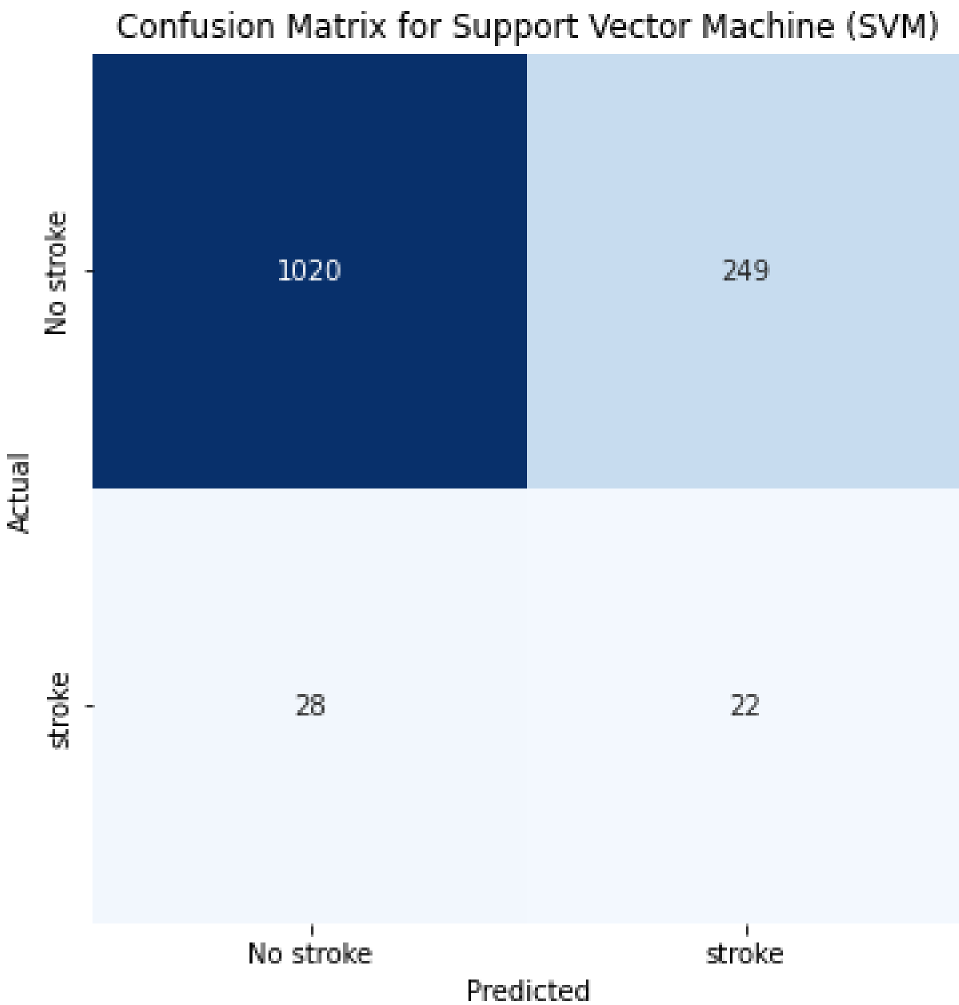
```
# Confusion matrix of optimized KNN
labels = [0, 1]
cm = confusion_matrix(y_test, y_pred_knn_best, labels=labels)

# Plot the heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
xticklabels=['No stroke', 'stroke'],
            yticklabels=['No stroke', 'stroke'], square=True, vmin=0,
vmax=np.max(cm))
plt.title('Confusion Matrix for Optimized KNN Model')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```



Applying SVM

```
# Basic SVM model
svm_model = SVC(kernel='rbf', random_state=42, probability=True)
svm_model.fit(X_train_scaled, y_train_resampled)
y_pred_svm = svm_model.predict(X_test_scaled)

# Model Evaluation
svm_performance = evaluate_model(y_test, y_pred_svm, "Support Vector
Machine (SVM)")


Support Vector Machine (SVM) Performance:
Accuracy: 0.7900
Precision: 0.0812
Recall: 0.4400
F1-score: 0.1371
AUC-ROC: 0.6219
```

```python
# Confusion matrix of SVM
labels = [0, 1]
cm = confusion_matrix(y_test, y_pred_svm, labels=labels)

# Plot the heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
xticklabels=['No stroke', 'stroke'],
            yticklabels=['No stroke', 'stroke'], square=True, vmin=0,
vmax=np.max(cm))
plt.title('Confusion Matrix for Support Vector Machine (SVM)')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```



Hyperparameter Tuning in SVM

```
# Defining hyperparameter grid for SVM
param_grid = {
    'C': [0.1, 1, 10, 100],
    'kernel': ['linear', 'rbf'],
    'gamma': ['scale', 0.01, 0.1]
}

# Performing Grid Search
grid_search = GridSearchCV(svm_model, param_grid, cv=5,
scoring='accuracy', n_jobs=-1)
grid_search.fit(X_train_scaled, y_train_resampled)

# Getting best parameters
best_params = grid_search.best_params_
print("Best Parameters for SVM:", best_params)
```

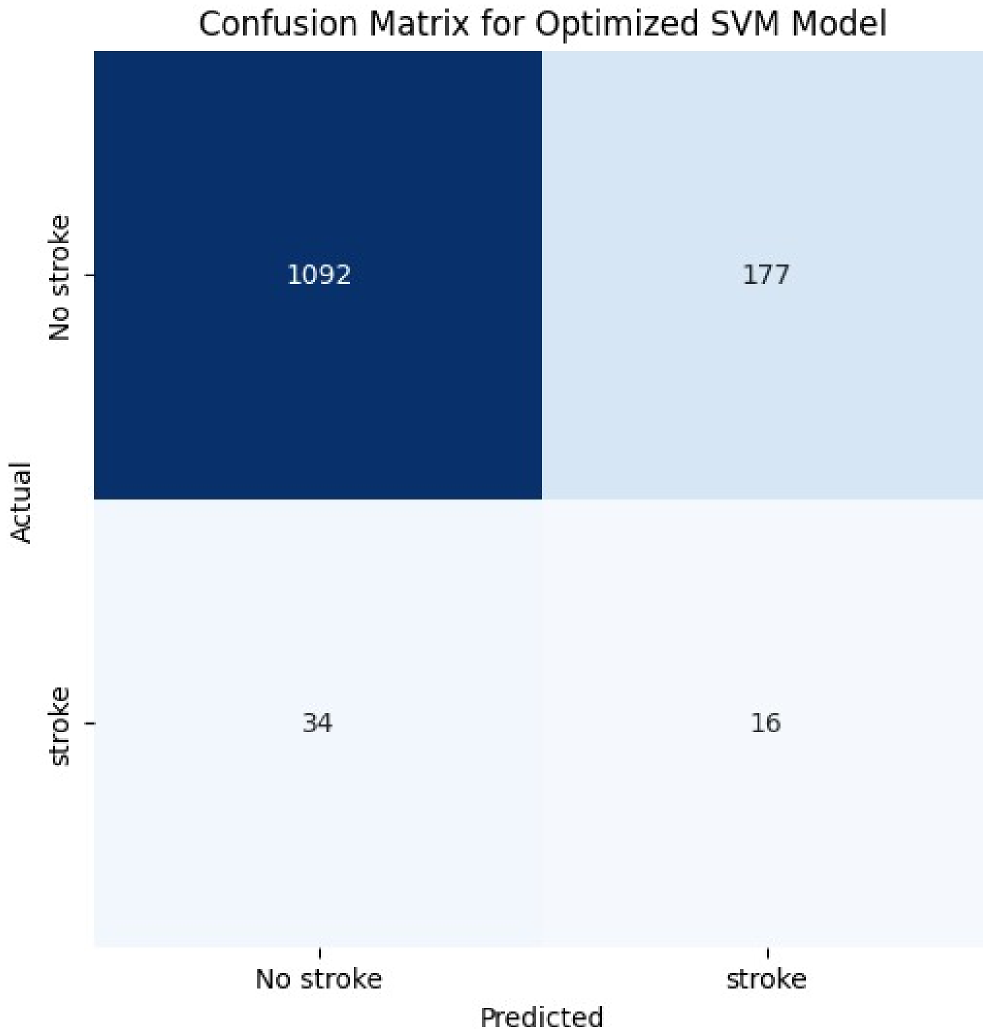
```
Best Parameters for SVM: {'C': 100, 'gamma': 'scale', 'kernel': 'rbf'}
```

```
# Training SVM model with best parameters
best_svm_model =SVC(**best_params, random_state=42, probability=True)
best_svm_model.fit(X_train_scaled, y_train_resampled)

# Predictions
y_pred_best_svm = best_svm_model.predict(X_test_scaled)

# Model Evaluation
best_svm_performance = evaluate_model(y_test, y_pred_best_svm,
"Optimized SVM")
```

```
Optimized SVM Performance:
Accuracy: 0.8400
Precision: 0.0829
Recall: 0.3200
F1-score: 0.1317
AUC-ROC: 0.5903
```

```
# Confusion matrix of optimized SVM
labels = [0, 1]
cm = confusion_matrix(y_test, y_pred_best_svm, labels=labels)

# Plot the heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
xticklabels=['No stroke', 'stroke'],
            yticklabels=['No stroke', 'stroke'], square=True, vmin=0,
vmax=np.max(cm))
plt.title('Confusion Matrix for Optimized SVM Model')
plt.ylabel('Actual')
```

```
plt.xlabel('Predicted')
plt.show()
```

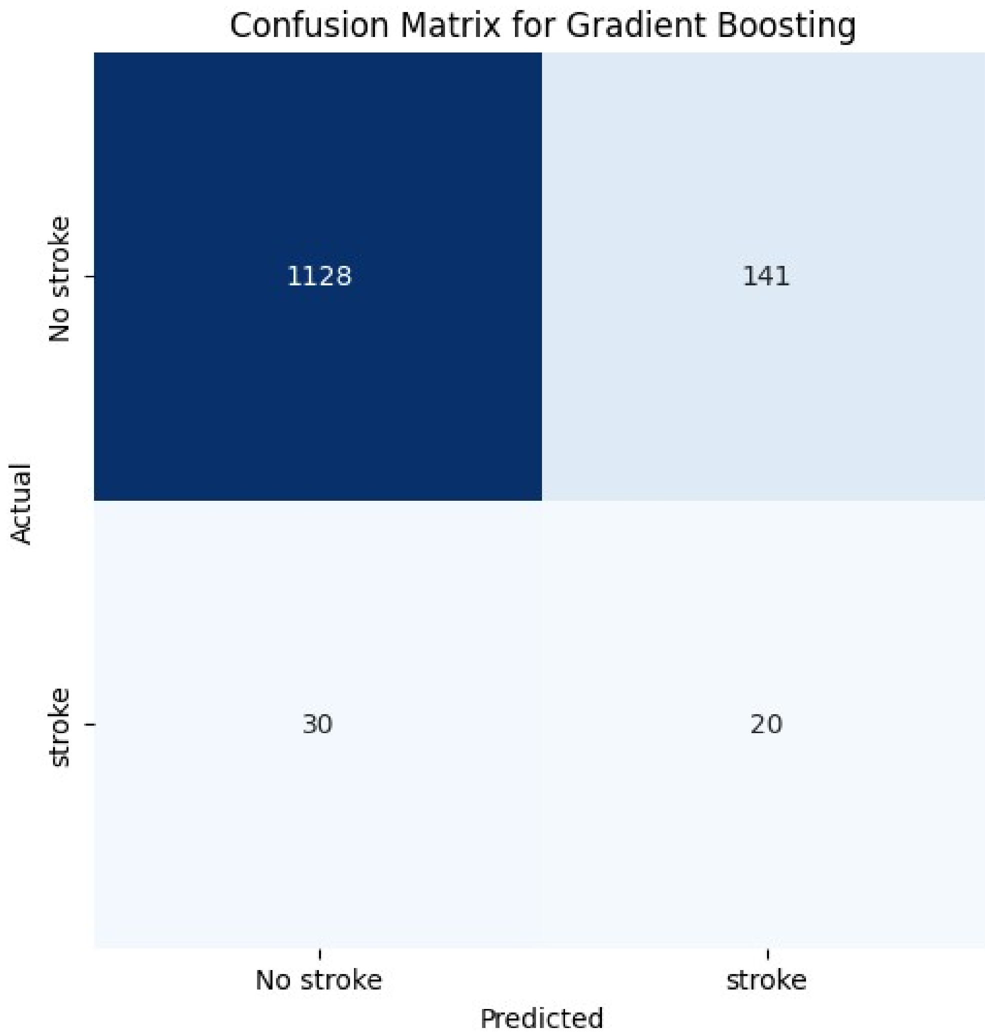


Applying Gradient Boosting Algorithm

```
# Train a basic Gradient Boosting model
gb_model =
GradientBoostingClassifier(random_state=42,n_estimators=100)

# Fit the model
gb_model.fit(X_train_scaled, y_train_resampled)

# Make predictions
y_pred_gb = gb_model.predict(X_test_scaled)

# Model Evaluation
gb_performance = evaluate_model(y_test, y_pred_gb, "Gradient
Boosting")
```

```
Gradient Boosting Performance:
Accuracy: 0.8704
Precision: 0.1242
Recall: 0.4000
F1-score: 0.1896
AUC-ROC: 0.6444
```

```
# Confusion matrix of Gradient Boosting
labels = [0, 1]
cm = confusion_matrix(y_test, y_pred_gb, labels=labels)

# Plot the heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
xticklabels=['No stroke', 'stroke'],
            yticklabels=['No stroke', 'stroke'], square=True, vmin=0,
vmax=np.max(cm))
plt.title('Confusion Matrix for Gradient Boosting')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```



Hyperparameter Tuning in Gradient Boosting

```
# Defining parameter grid for Gradient Boosting
param_grid = {
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7],
    'n_estimators': [50, 100, 200],
    'subsample': [0.8, 1.0],
    'max_features': ['sqrt', 'log2', None]
}

# Performing Grid Search
grid_search = GridSearchCV(
    GradientBoostingClassifier(random_state=42),
    param_grid,
    cv=5,
    scoring='roc_auc',
    n_jobs=-1
```

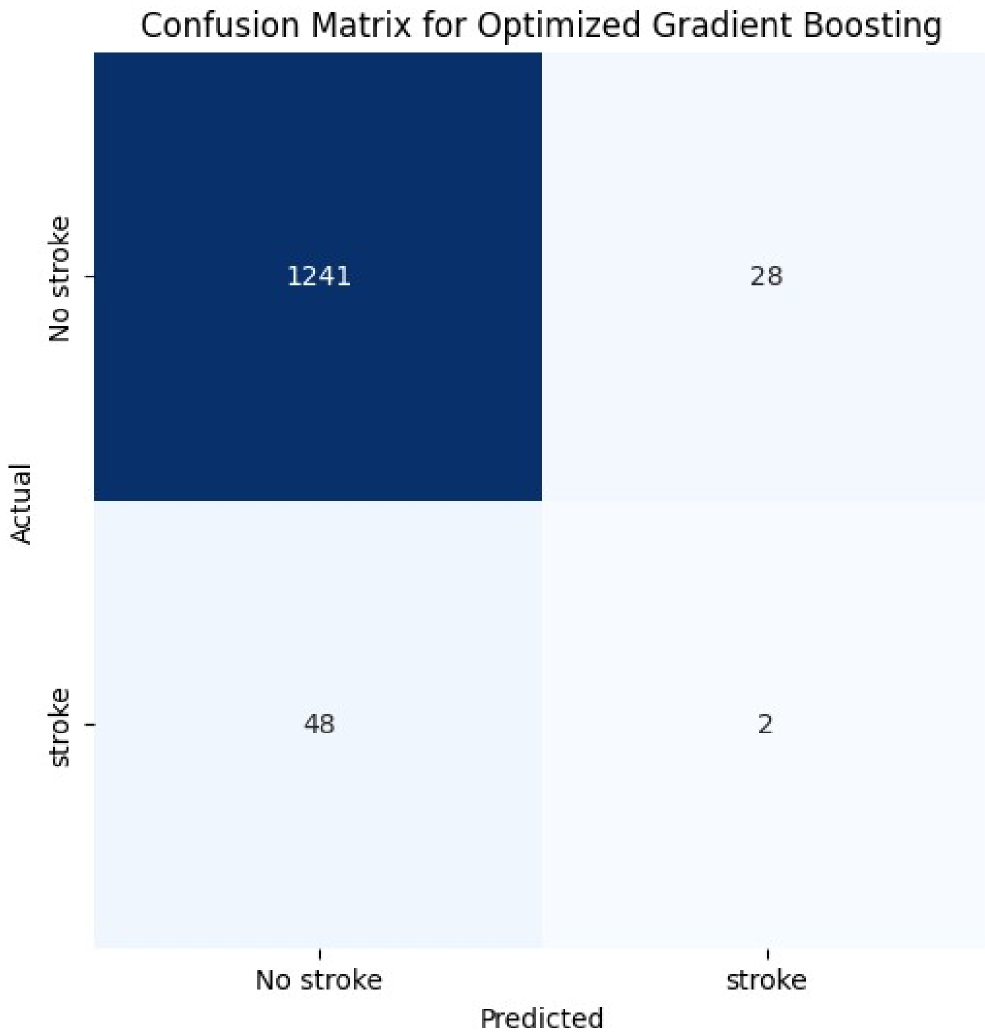
```
)
grid_search.fit(X_train_scaled, y_train_resampled)

# Getting best parameters
best_params = grid_search.best_params_
print("Best Parameters for GB:", best_params)
```

```
Best Parameters for GB: {'learning_rate': 0.2, 'max_depth': 7,
'max_features': None, 'n_estimators': 200, 'subsample': 1.0}
```

```
# Get the best model
best_gb = grid_search.best_estimator_

# Make predictions with the tuned model
y_pred_best_gb = best_gb.predict(X_test_scaled)

# Model Evaluation
best_gb_performance = evaluate_model(y_test, y_pred_best_gb,
"Optimized Gradient Boosting")
```

```
Optimized Gradient Boosting Performance:
Accuracy: 0.8984
Precision: 0.0800
Recall: 0.1600
F1-score: 0.1067
AUC-ROC: 0.5438
```

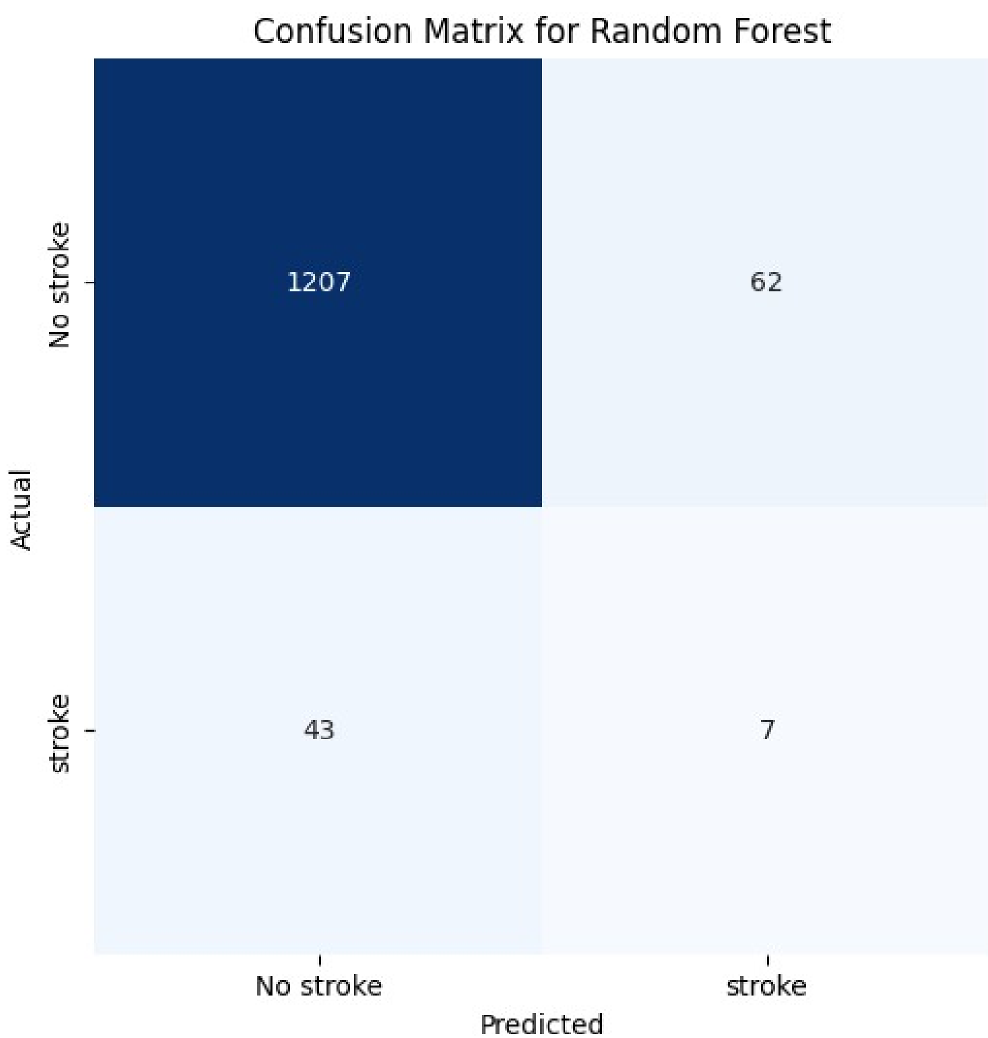
```
# Confusion matrix of Optimized Gradient Boosting
labels = [0, 1]
cm = confusion_matrix(y_test, y_pred_best_gb, labels=labels)

# Plot the heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
xticklabels=['No stroke', 'stroke'],
            yticklabels=['No stroke', 'stroke'], square=True, vmin=0,
vmax=np.max(cm))
plt.title('Confusion Matrix for Optimized Gradient Boosting')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```



Applying Random Forest

```
# Training random forest
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)

# Fit the model
rf_model.fit(X_train_scaled, y_train_resampled)

# Make predictions
y_pred_rf = rf_model.predict(X_test_scaled)

# Model Evaluation
rf_performance = evaluate_model(y_test, y_pred_rf, "Random Forest")


Random Forest Performance:
Accuracy: 0.9204
Precision: 0.1014
Recall: 0.1400
```

```
F1-score: 0.1176
AUC-ROC: 0.5456

# Confusion matrix of Random Forest
labels = [0, 1]
cm = confusion_matrix(y_test, y_pred_rf, labels=labels)

# Plot the heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
xticklabels=['No stroke', 'stroke'],
            yticklabels=['No stroke', 'stroke'], square=True, vmin=0,
vmax=np.max(cm))
plt.title('Confusion Matrix for Random Forest')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```



Hyperparameter boosting for Random forest

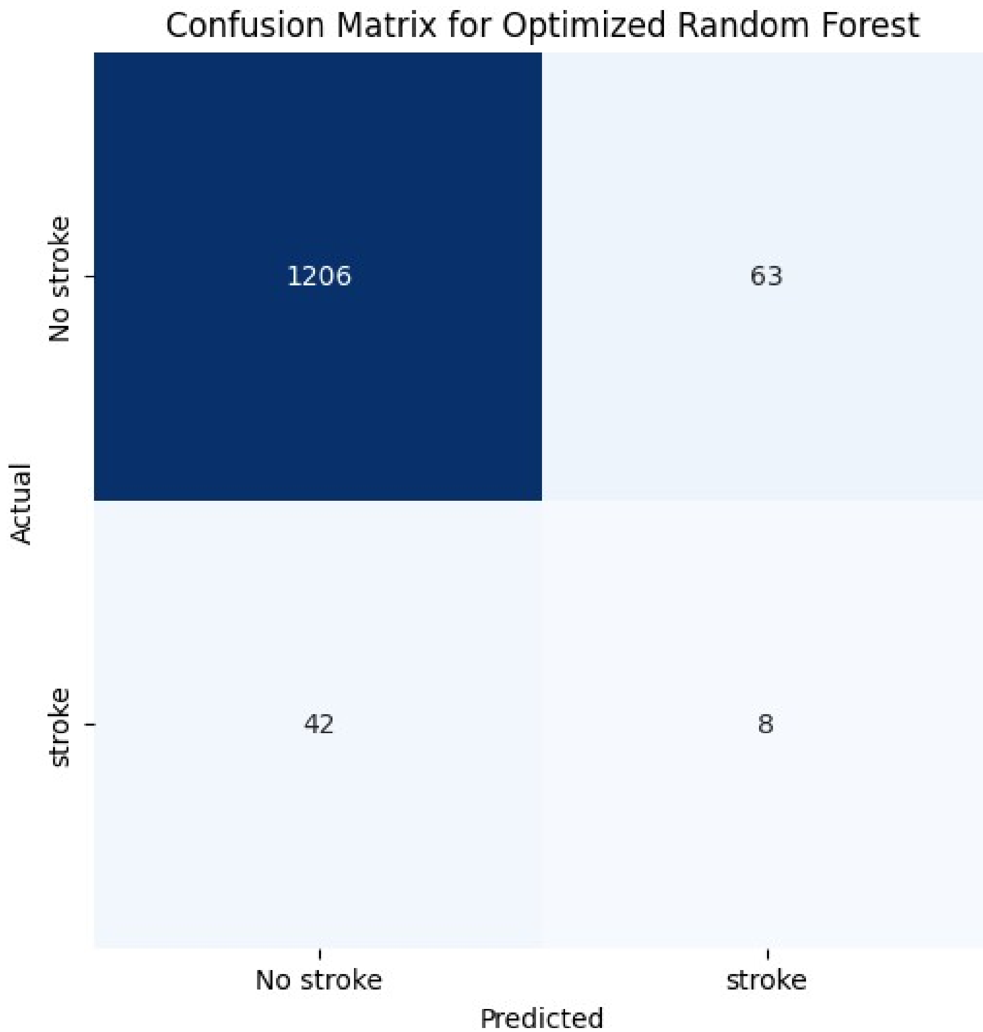
```
rf_param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2]
}
rf_grid_search = GridSearchCV(RandomForestClassifier(random_state=42),
rf_param_grid, cv=5, scoring='roc_auc', n_jobs=-1)
rf_grid_search.fit(X_train_scaled, y_train_resampled)
best_rf = rf_grid_search.best_estimator_
print("Best Random Forest Params:", rf_grid_search.best_params_)
```

```
Best Random Forest Params: {'max_depth': None, 'min_samples_leaf': 1,
'min_samples_split': 2, 'n_estimators': 200}
```

```
# Make predictions with the tuned model
y_pred_best_rf = best_rf.predict(X_test_scaled)

# Model Evaluation
best_rf_performance = evaluate_model(y_test, y_pred_tuned, "Optimized
Random Forest")
```

```
Optimized Random Forest Performance:
Accuracy: 0.9204
Precision: 0.1127
Recall: 0.1600
F1-score: 0.1322
AUC-ROC: 0.5552
```

```
# Confusion matrix of optimized Random Forest
labels = [0, 1]
cm = confusion_matrix(y_test, y_pred_best_rf, labels=labels)

# Plot the heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
xticklabels=['No stroke', 'stroke'],
            yticklabels=['No stroke', 'stroke'], square=True, vmin=0,
vmax=np.max(cm))
plt.title('Confusion Matrix for Optimized Random Forest')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

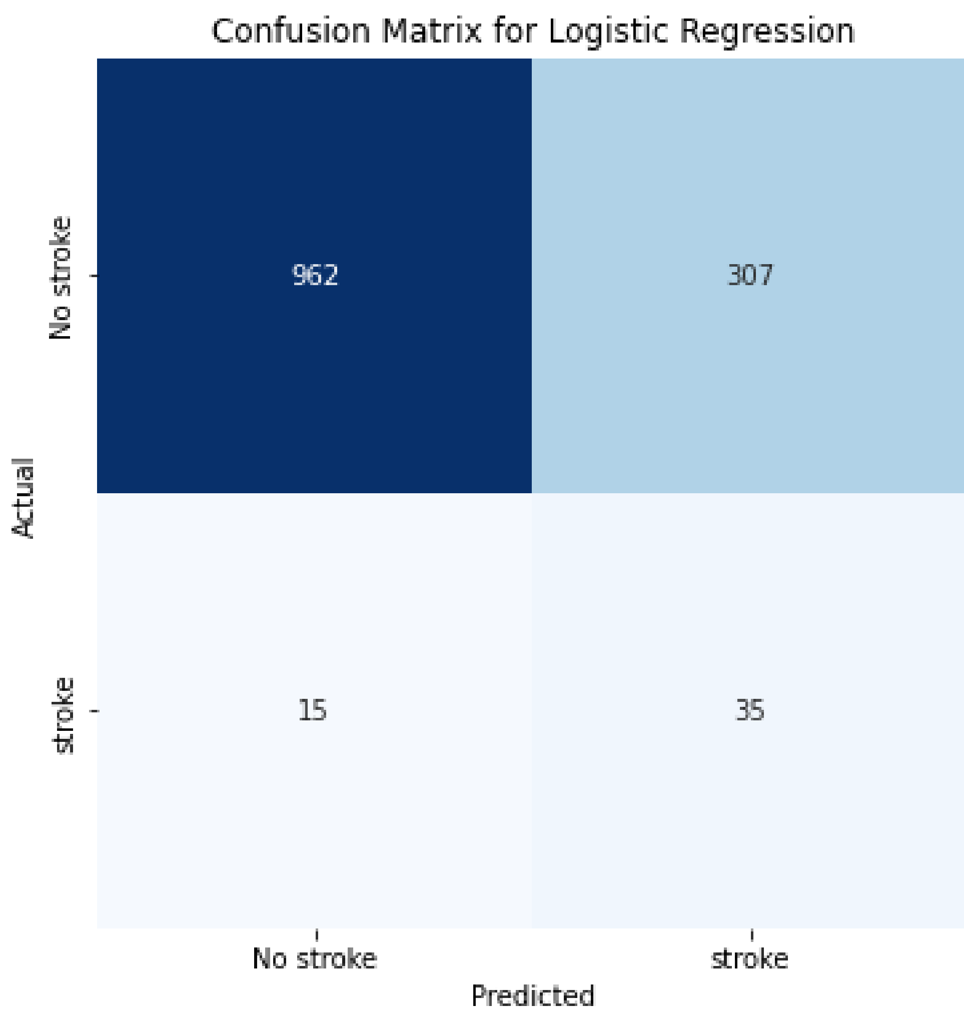


Applying Logistic Regression

```
# Training random forest
lr_model = LogisticRegression()

# Fit the model
lr_model.fit(X_train_scaled, y_train_resampled)

# Make predictions
y_pred_lr = lr_model.predict(X_test_scaled)

# Model Evaluation
lr_performance = evaluate_model(y_test, y_pred_lr, "Logistic
Regression")
```

```
Logistic Regression Performance:
Accuracy: 0.7559
Precision: 0.1023
```

```
Recall: 0.7000
F1-score: 0.1786
AUC-ROC: 0.7290

# Confusion matrix of Logistic Regression
labels = [0, 1]
cm = confusion_matrix(y_test, y_pred_lr, labels=labels)

# Plot the heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
xticklabels=['No stroke', 'stroke'],
            yticklabels=['No stroke', 'stroke'], square=True, vmin=0,
vmax=np.max(cm))
plt.title('Confusion Matrix for Logistic Regression')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```



Hyperparameter tuning for Logistic regression

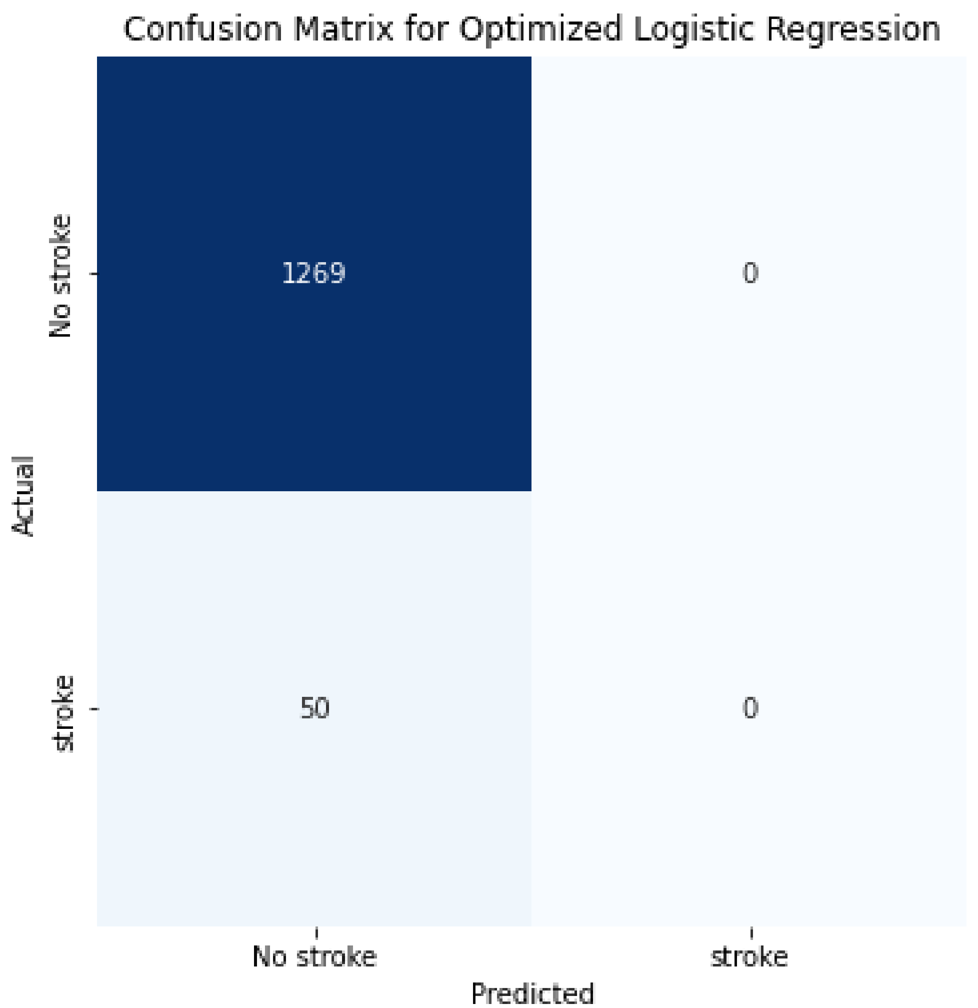
```
# Hyperparameter tuning for Logistic Regression
lr_param_grid = {
    'C': [0.1, 1, 10, 100],
    'solver': ['liblinear', 'lbfgs']
}
lr_grid_search = GridSearchCV(LogisticRegression(max_iter=1000),
lr_param_grid, cv=5, scoring='roc_auc', n_jobs=-1)
lr_grid_search.fit(X_train, y_train)
best_lr = lr_grid_search.best_estimator_
print("Best Logistic Regression Params:", lr_grid_search.best_params_)
```

```
Best Logistic Regression Params: {'C': 100, 'solver': 'lbfgs'}
```

```
# Make predictions with the tuned model
y_pred_best_lr = best_lr.predict(X_test_scaled)

# Model Evaluation
best_lr_performance = evaluate_model(y_test, y_pred_best_lr,
"Optimized Logistic Regression")
```

```
Optimized Logistic Regression Performance:
Accuracy: 0.9621
Precision: 0.0000
Recall: 0.0000
F1-score: 0.0000
AUC-ROC: 0.5000
```

```
# Confusion matrix of Optimized Logistic Regression
labels = [0, 1]
cm = confusion_matrix(y_test, y_pred_best_lr, labels=labels)

# Plot the heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
xticklabels=['No stroke', 'stroke'],
            yticklabels=['No stroke', 'stroke'], square=True, vmin=0,
vmax=np.max(cm))
plt.title('Confusion Matrix for Optimized Logistic Regression')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```



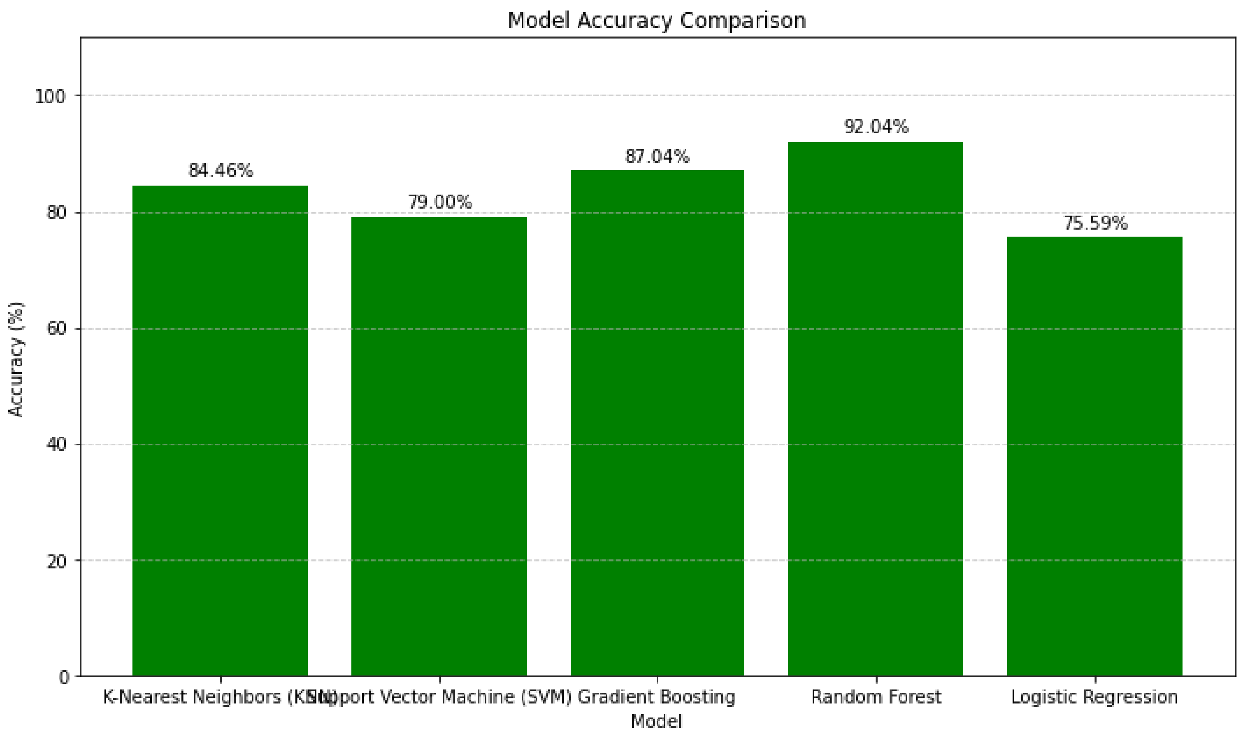
Model Comparison

```
# defining model lists in a data frame
models = pd.DataFrame([
    knn_performance,
    svm_performance,
    gb_performance,
    rf_performance,
    lr_performance
])

metrics = ["Accuracy", "Precision", "Recall", "F1-score", "AUC-ROC"]
for metric in metrics:
    models[f"{metric} (%)"] = models[metric] * 100
```

Accuracy Comparison

```
plt.figure(figsize=(10, 6))
bars = plt.bar(models["Model"], models["Accuracy (%)"], color='green')
```

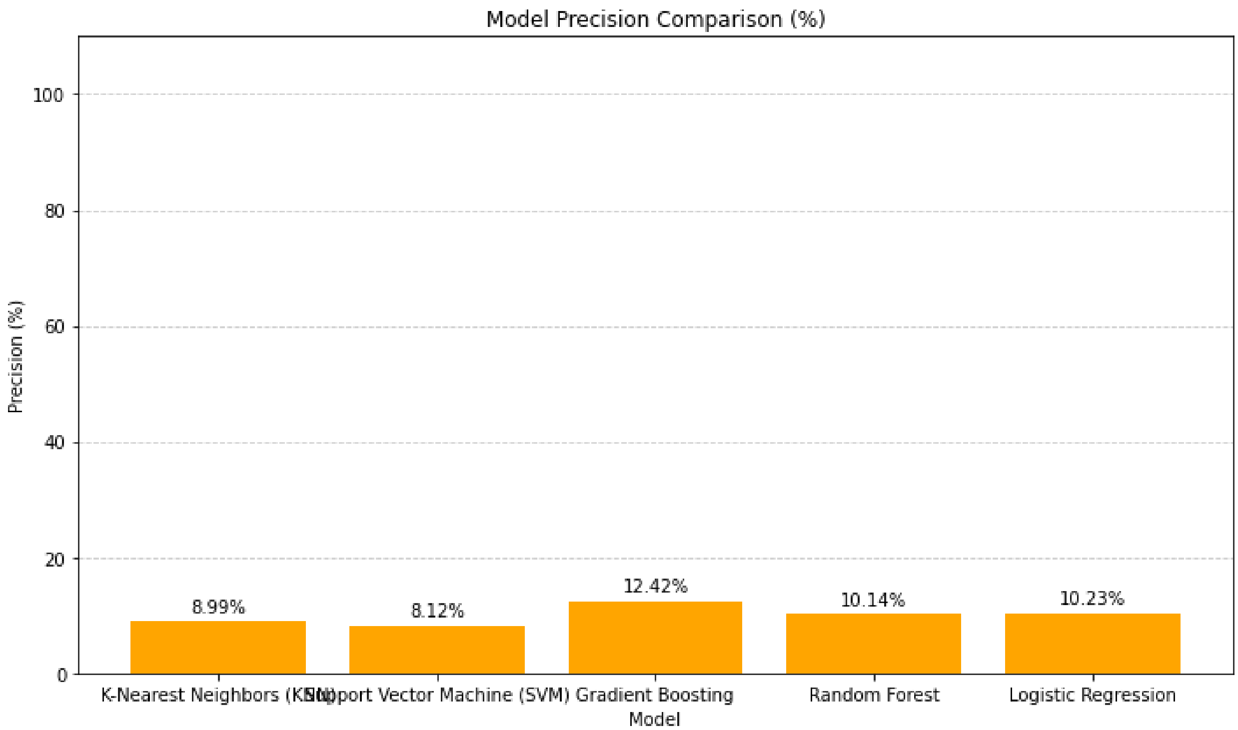
```
plt.title("Model Accuracy Comparison")
plt.xlabel("Model")
plt.ylabel("Accuracy (%)")
plt.ylim(0, 110)
plt.grid(axis='y', linestyle='--', alpha=0.7)

for bar in bars:
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, height + 1,
f"{height:.2f}%", ha='center', va='bottom')

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



Precision Comparison

```
plt.figure(figsize=(10, 6))
bars = plt.bar(models["Model"], models["Precision (%)"],
color='orange')
plt.title("Model Precision Comparison (%)")
plt.xlabel("Model")
plt.ylabel("Precision (%)")
plt.ylim(0, 110)
plt.grid(axis='y', linestyle='--', alpha=0.7)

for bar in bars:
    height = bar.get_height()
```

```
    plt.text(bar.get_x() + bar.get_width()/2, height + 1,
f"{height:.2f}%", ha='center', va='bottom')

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



Comparison of all the metrics

```
# Set up for grouped bar chart
bar_metrics = [f"{m} (%)" for m in metrics]
x = np.arange(len(models["Model"]))  # label locations
bar_width = 0.15

# Plot setup
plt.figure(figsize=(12, 6))

# Plot each metric as a bar group
for i, metric in enumerate(bar_metrics):
    plt.bar(x + i * bar_width, models[metric], width=bar_width,
label=metric)

# X-axis and labels
plt.xticks(x + (bar_width * 2), models["Model"])
plt.xlabel("Model")
plt.ylabel("Percentage")
plt.title("Model Performance Comparison")
plt.ylim(0, 110)
```

```
plt.legend()
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```