```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics
```

```
import pandas as pd
col_names = ['id', 'age', 'sex', 'region', 'income', 'married', 'children', 'car', 'save_act', 'current_ac', 'mortgage', 'pep']
bank = pd.read_csv("bank.csv", names=col_names)
```

```
print(bank.head())
```

```
        id  age     sex      region   income  married  children  car  \
0       id  age     sex      region   income  married  children  car
1  ID12101   48  FEMALE  INNER_CITY    17546       NO         1   NO
2  ID12102   40    MALE        TOWN  30085.1      YES         3  YES
3  ID12103   51  FEMALE  INNER_CITY  16575.4      YES         0  YES
4  ID12104   23  FEMALE        TOWN  20375.4      YES         3   NO

   save_act   current_ac  mortgage  pep
0  save_act  current_act  mortgage  pep
1        NO           NO        NO  YES
2        NO          YES       YES   NO
3       YES          YES        NO   NO
4        NO          YES        NO   NO
```

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

# Load the original dataset
data = pd.read_csv('/content/bank.csv')

# Initialize label encoder
label_encoder = LabelEncoder()

binary_columns = ['sex', 'married', 'car', 'save_act', 'current_act', 'mortgage']

# Apply label encoder to each binary column
for column in binary_columns:
    data[column] = label_encoder.fit_transform(data[column])

# Define the transformer for one-hot encoding
column_transformer = ColumnTransformer(
    transformers=[
        ('region', OneHotEncoder(), ['region'])
    ],
    remainder='passthrough'
)

# Apply the transformer to the DataFrame
data_encoded = column_transformer.fit_transform(data)

# Retrieve the one-hot encoded region column names
region_columns = column_transformer.named_transformers_['region'].get_feature_names_out()

# Define new column names after encoding (one-hot encoded columns + rest)
new_columns = list(region_columns) + [col for col in data.columns if col != 'region']

# Create a new DataFrame with the encoded data
encoded_df = pd.DataFrame(data_encoded, columns=new_columns)

# Save the encoded DataFrame back to csv without the index
encoded_df.to_csv('/content/bank_encoded.csv', index=False)

print(encoded_df.head())
```

```
  region_INNER_CITY region_RURAL region_SUBURBAN region_TOWN       id age sex  \
0               1.0          0.0             0.0         0.0  ID12101  48   0
1               0.0          0.0             0.0         1.0  ID12102  40   1
2               1.0          0.0             0.0         0.0  ID12103  51   0
3               0.0          0.0             0.0         1.0  ID12104  23   0
4               0.0          1.0             0.0         0.0  ID12105  57   0

    income married children car save_act current_act mortgage  pep
```

```
0  17546.0      0        1   0        0           0        0  YES
1  30085.1      1        3   1        0           1        1   NO
2  16575.4      1        0   1        1           1        0   NO
3  20375.4      1        3   0        0           1        0   NO
4  50576.3      1        0   0        1           0        0   NO
```

```
# Load the encoded dataset
data_encoded = pd.read_csv('/content/bank_encoded.csv')

# Since the file was previously encoded and saved, we assume 'region' was correctly transformed.
# We will prepare the feature columns list by excluding 'id' and 'pep' (target variable) from the dataframe columns.
feature_cols = [col for col in data_encoded.columns if col not in ('id', 'pep')]

# Split dataset in features and target variable
X = data_encoded[feature_cols]  # Features
y = data_encoded['pep']         # Target variable

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)  # 70% training and 30% test

# Create Decision Tree classifier object
clf = DecisionTreeClassifier()

# Train Decision Tree Classifier
clf = clf.fit(X_train, y_train)

# Predict the response for the test dataset
y_pred = clf.predict(X_test)

# Since the task does not specify to output anything, we will not display the predictions here.
```

```
# Create Decision Tree classifier object with entropy
clf_entropy = DecisionTreeClassifier(criterion='entropy')

# Train Decision Tree Classifier with entropy
clf_entropy = clf_entropy.fit(X_train, y_train)

# Predict the response for the test dataset using the entropy model
y_pred_entropy = clf_entropy.predict(X_test)

# Evaluate the entropy model
accuracy_entropy = metrics.accuracy_score(y_test, y_pred_entropy)
accuracy_entropy
```

```
0.8111111111111111
```

```
from sklearn import metrics

# Evaluate the model
accuracy = metrics.accuracy_score(y_test, y_pred)
accuracy
```

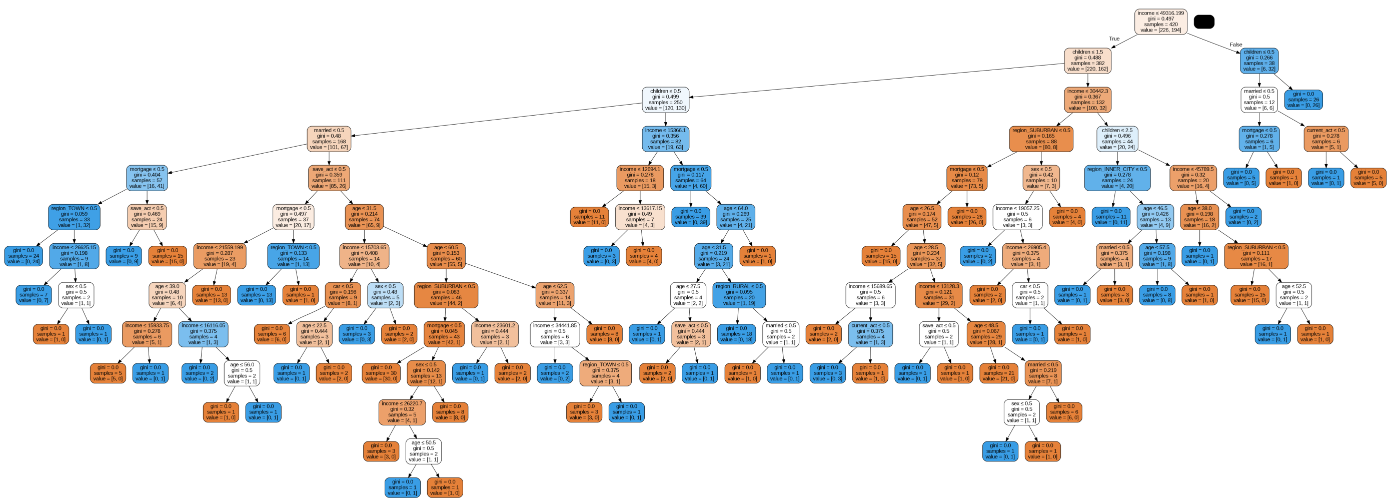
```
0.8666666666666667
```

```
from sklearn.tree import export_graphviz
from IPython.display import Image
import pydotplus
from io import StringIO

# Assuming clf is the classifier trained with the Gini impurity method
dot_data = StringIO()
export_graphviz(clf, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True, feature_names=feature_cols)

# Generate graph from dot data
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())

# Save and display the graph
graph.write_png('decision_tree_gini.png')
Image(graph.create_png())
```

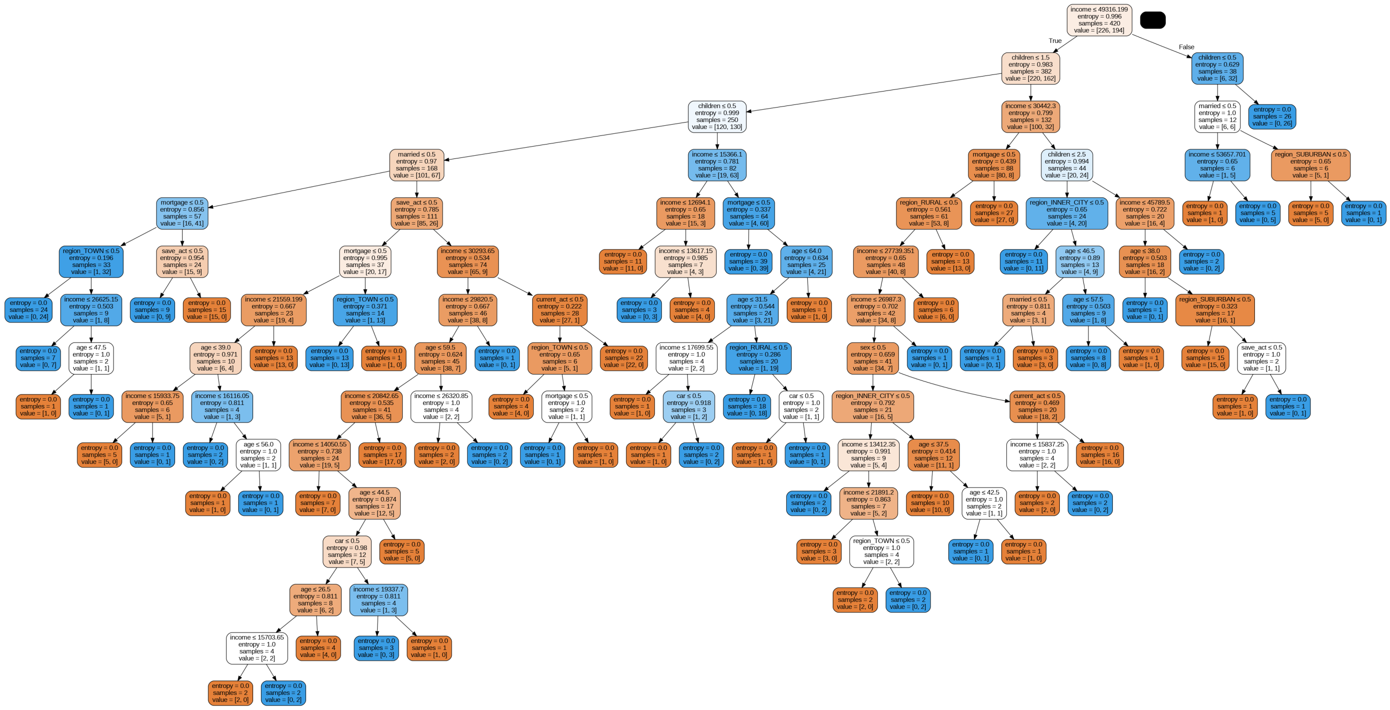


```
from sklearn.tree import export_graphviz
from IPython.display import Image
import pydotplus
from io import StringIO

# Visualize the tree using the entropy model
dot_data = StringIO()
export_graphviz(clf_entropy, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True, feature_names=feature_cols)

# Generate graph from dot data
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())

# Save and display the graph
graph.write_png('decision_tree_entropy.png')
Image(graph.create_png())
```



KNN

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn import metrics

# Load the dataset (assuming it's the same as for the decision tree)
data_encoded = pd.read_csv('/content/bank_encoded.csv')

# We will prepare the feature columns list by excluding 'id' and 'pep' (target variable) from the dataframe columns.
feature_cols = [col for col in data_encoded.columns if col not in ('id', 'pep')]

# Split dataset in features and target variable
X = data_encoded[feature_cols]  # Features
y = data_encoded['pep']         # Target variable

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.3, random_state=1)  # 70% training and 30% test

# Create KNN classifier object
knn = KNeighborsClassifier(n_neighbors=3)  # Using n_neighbors=5 as a common default

# Train KNN Classifier
knn.fit(X_train, y_train)

# Predict the response for the test dataset
y_pred_knn = knn.predict(X_test)

# Evaluate the model
accuracy_knn = metrics.accuracy_score(y_test, y_pred_knn)
accuracy_knn
```

```
0.6166666666666667
```

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.metrics import precision_score, recall_score, accuracy_score
from sklearn.preprocessing import StandardScaler

# Load the encoded dataset
data_encoded = pd.read_csv('/content/bank_encoded.csv')

# Prepare the feature columns list by excluding 'id' and 'pep' (target variable)
feature_cols = [col for col in data_encoded.columns if col not in ('id', 'pep')]

# Split dataset in features and target variable
X = data_encoded[feature_cols]  # Features
y = data_encoded['pep']         # Target variable

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.3, random_state=1)

# Define a range of k values to try
k_values = range(1, 31)

# Perform cross-validation and record the accuracy scores
scores = []
for k in k_values:
    knn_cv = KNeighborsClassifier(n_neighbors=k)
    cv_scores = cross_val_score(knn_cv, X_scaled, y, cv=5, scoring='accuracy')
    scores.append(np.mean(cv_scores))

# Find the best k value and accuracy
best_k_index = np.argmax(scores)
#best_k = k_values[best_k_index]
best_accuracy = scores[best_k_index]

best_k = 57
# Train a new KNN Classifier using the best k value
knn_best = KNeighborsClassifier(n_neighbors=best_k)
knn_best.fit(X_train, y_train)

# Make predictions
y_pred_best_knn = knn_best.predict(X_test)

# Calculate precision and recall
precision = precision_score(y_test, y_pred_best_knn, pos_label='YES')  # Assuming 'YES' is the positive class
recall = recall_score(y_test, y_pred_best_knn, pos_label='YES')  # Assuming 'YES' is the positive class

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred_best_knn)

(best_k, best_accuracy, precision, recall, accuracy)
```

```
(57, 0.6599999999999999, 0.6481481481481481, 0.4375, 0.6444444444444445)
```

```
# Test with a wider range of k values
k_range = range(1, 100, 2)  # Testing only odd numbers to avoid ties
k_scores = []

for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X_scaled, y, cv=10, scoring='accuracy')  # Using 10-fold cross-validation
    k_scores.append(scores.mean())

# Find the best k value
best_k_index = np.argmax(k_scores)
best_k = k_range[best_k_index]
print(best_k)
```

```
57
```

```
# Trying different distance metrics with the best k

# Manhattan distance
knn_manhattan = KNeighborsClassifier(n_neighbors=best_k, metric='manhattan')
manhattan_scores = cross_val_score(knn_manhattan, X_scaled, y, cv=5, scoring='accuracy')
manhattan_accuracy = manhattan_scores.mean()

# Euclidean distance
knn_euclidean = KNeighborsClassifier(n_neighbors=best_k, metric='euclidean')
euclidean_scores = cross_val_score(knn_euclidean, X_scaled, y, cv=5, scoring='accuracy')
euclidean_accuracy = euclidean_scores.mean()

manhattan_accuracy, euclidean_accuracy
```

```
(0.6833333333333332, 0.655)
```

```
from sklearn.preprocessing import MinMaxScaler

# Normalize features using MinMaxScaler instead of StandardScaler
minmax_scaler = MinMaxScaler()
X_minmax = minmax_scaler.fit_transform(X)

# Split dataset into training set and test set with normalized features
X_train_minmax, X_test_minmax, y_train, y_test = train_test_split(X_minmax, y, test_size=0.3, random_state=1)

# Create KNN classifier object with the best parameters found: k=57, distance metric 'euclidean', and weights='distance'
knn_final = KNeighborsClassifier(n_neighbors=23, metric='euclidean', weights='distance')

# Train KNN Classifier with normalized features
knn_final.fit(X_train_minmax, y_train)

# Predict the response for the test dataset with normalized features
y_pred_knn_final = knn_final.predict(X_test_minmax)

# Calculate precision, recall, and accuracy for the final model
precision_final = precision_score(y_test, y_pred_knn_final, pos_label='YES')  # Assuming 'YES' is the positive class
recall_final = recall_score(y_test, y_pred_knn_final, pos_label='YES')  # Assuming 'YES' is the positive class
accuracy_final = accuracy_score(y_test, y_pred_knn_final)

(best_k, best_accuracy, precision_final, recall_final, accuracy_final)
```

```
(57, 0.6599999999999999, 0.6129032258064516, 0.475, 0.6333333333333333)
```

```
from sklearn.preprocessing import MinMaxScaler

# Normalize features using MinMaxScaler
minmax_scaler = MinMaxScaler()
X_minmax = minmax_scaler.fit_transform(X)

# Perform cross-validation with normalized features
scores_minmax = []
for k in k_values:
    knn_cv = KNeighborsClassifier(n_neighbors=k)
    cv_scores = cross_val_score(knn_cv, X_minmax, y, cv=5, scoring='accuracy')
    scores_minmax.append(np.mean(cv_scores))

# Find the best k value and accuracy for normalized features
best_k_index_minmax = np.argmax(scores_minmax)
best_k_minmax = k_values[best_k_index_minmax]
best_accuracy_minmax = scores_minmax[best_k_index_minmax]

best_k_minmax, best_accuracy_minmax
```

```
(23, 0.6716666666666666)
```

Naive Bayes

```
from sklearn.naive_bayes import GaussianNB

# Create a Gaussian Naive Bayes classifier object
gnb = GaussianNB()

# Train Naive Bayes Classifier
gnb.fit(X_train_minmax, y_train)

# Predict the response for the test dataset
y_pred_gnb = gnb.predict(X_test_minmax)

# Calculate accuracy, precision, and recall for Naive Bayes model
accuracy_gnb = accuracy_score(y_test, y_pred_gnb)
precision_gnb = precision_score(y_test, y_pred_gnb, pos_label='YES')
recall_gnb = recall_score(y_test, y_pred_gnb, pos_label='YES')

(accuracy_gnb, precision_gnb, recall_gnb)
```

```
(0.5722222222222222, 0.5238095238095238, 0.4125)
```

```
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE  # Requires installing imblearn

# Load the dataset
data_encoded = pd.read_csv('/content/bank_encoded.csv')

# Prepare the feature columns list by excluding 'id' and 'pep'
feature_cols = [col for col in data_encoded.columns if col not in ('id', 'pep')]

# Split dataset in features and target variable
X = data_encoded[feature_cols]
y = data_encoded['pep']

# Normalize the features
minmax_scaler = MinMaxScaler()
X_minmax = minmax_scaler.fit_transform(X)

# Handling class imbalance using SMOTE
smote = SMOTE(random_state=1)
X_smote, y_smote = smote.fit_resample(X_minmax, y)

# Split the dataset into training and test sets
X_train_smote, X_test_smote, y_train_smote, y_test = train_test_split(X_smote, y_smote, test_size=0.3, random_state=1)

# Create a Gaussian Naive Bayes classifier object
gnb = GaussianNB()

# Train the Naive Bayes Classifier
gnb.fit(X_train_smote, y_train_smote)

# Predict the response for the test dataset
y_pred_gnb = gnb.predict(X_test_smote)

# Calculate precision, recall, and accuracy
precision_gnb = precision_score(y_test, y_pred_gnb, pos_label='YES')
recall_gnb = recall_score(y_test, y_pred_gnb, pos_label='YES')
accuracy_gnb = accuracy_score(y_test, y_pred_gnb)

(accuracy_gnb, precision_gnb, recall_gnb)
```

```
(0.6122448979591837, 0.6666666666666666, 0.5098039215686274)
```

Logistic Regression

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.model_selection import train_test_split

# Load the encoded dataset
data_encoded = pd.read_csv('/content/bank_encoded.csv')

# Prepare the feature columns list by excluding 'id' and 'pep'
feature_cols = [col for col in data_encoded.columns if col not in ('id', 'pep')]

# Split dataset in features and target variable
X = data_encoded[feature_cols]
y = data_encoded['pep']

# Normalize the features using MinMaxScaler
minmax_scaler = MinMaxScaler()
X_minmax = minmax_scaler.fit_transform(X)

# Split dataset into training set and test set with normalized features
X_train_minmax, X_test_minmax, y_train, y_test = train_test_split(X_minmax, y, test_size=0.3, random_state=1)


lr = LogisticRegression(solver='saga', penalty='l1', random_state=1)

# Train Logistic Regression Classifier
lr.fit(X_train_minmax, y_train)

# Predict the response for the test dataset
y_pred_lr = lr.predict(X_test_minmax)

# Calculate precision, recall, and accuracy for Logistic Regression model
accuracy_lr = accuracy_score(y_test, y_pred_lr)
precision_lr = precision_score(y_test, y_pred_lr, pos_label='YES')
recall_lr = recall_score(y_test, y_pred_lr, pos_label='YES')

(accuracy_lr, precision_lr, recall_lr)
```

```
(0.5888888888888889, 0.5416666666666666, 0.4875)
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.preprocessing import PolynomialFeatures, MinMaxScaler
import pandas as pd
import numpy as np

# Assuming X_minmax and y are already defined and available from the previous steps
# Create polynomial features
poly = PolynomialFeatures(degree=2, interaction_only=True, include_bias=False)
X_poly = poly.fit_transform(X_minmax)

# Split dataset into training set and test set with polynomial features
X_train_poly, X_test_poly, y_train, y_test = train_test_split(X_poly, y, test_size=0.3, random_state=1)

# Set up the parameters to test for GridSearch on Logistic Regression
param_grid = {'C': np.logspace(-4, 4, 20)}

# Perform GridSearch with Logistic Regression and polynomial features
grid_search = GridSearchCV(LogisticRegression(class_weight='balanced', max_iter=1000, random_state=1),
                           param_grid=param_grid, scoring='accuracy', cv=5)
grid_search.fit(X_train_poly, y_train)

# Get the best C value from grid search
best_C = grid_search.best_params_['C']

# Update the Logistic Regression model with the best C value
lr_best = LogisticRegression(C=best_C, class_weight='balanced', max_iter=1000, random_state=1)
lr_best.fit(X_train_poly, y_train)

# Predict using the updated model
y_pred_lr_best = lr_best.predict(X_test_poly)

# Calculate updated metrics
updated_accuracy = accuracy_score(y_test, y_pred_lr_best)
updated_precision = precision_score(y_test, y_pred_lr_best, pos_label='YES')
updated_recall = recall_score(y_test, y_pred_lr_best, pos_label='YES')

(updated_accuracy, updated_precision, updated_recall, best_C)
```

```
(0.65, 0.6164383561643836, 0.5625, 11.288378916846883)
```

```
# Perform GridSearch with Logistic Regression using saga solver and polynomial features
grid_search_saga = GridSearchCV(LogisticRegression(solver='saga', class_weight='balanced', max_iter=10000, random_state=1),
                                param_grid=param_grid, scoring='accuracy', cv=5)
grid_search_saga.fit(X_train_poly, y_train)

# Get the best C value from grid search
best_C_saga = grid_search_saga.best_params_['C']

# Update the Logistic Regression model with the best C value and saga solver
lr_best_saga = LogisticRegression(solver='saga', C=best_C_saga, class_weight='balanced', max_iter=10000, random_state=1)
lr_best_saga.fit(X_train_poly, y_train)

# Predict using the updated model with saga solver
y_pred_lr_best_saga = lr_best_saga.predict(X_test_poly)

# Calculate updated metrics with saga solver
updated_accuracy_saga = accuracy_score(y_test, y_pred_lr_best_saga)
updated_precision_saga = precision_score(y_test, y_pred_lr_best_saga, pos_label='YES')
updated_recall_saga = recall_score(y_test, y_pred_lr_best_saga, pos_label='YES')

# The best 'C' parameter found, and the updated accuracy, precision, and recall with saga solver
best_C_saga, updated_accuracy_saga, updated_precision_saga, updated_recall_saga
```

```
(11.288378916846883, 0.65, 0.6164383561643836, 0.5625)
```

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

# Assuming y_test and y_pred_lr are already defined from the previous logistic regression model
# Calculate confusion matrix
cnf_matrix = confusion_matrix(y_test, y_pred_lr_best, labels=['YES', 'NO'])

# Plotting the confusion matrix
class_names=['YES', 'NO']  # the order of classes here is important for the matrix
fig, ax = plt.subplots()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names)
plt.yticks(tick_marks, class_names)

# Create heatmap
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu", fmt='g')
```