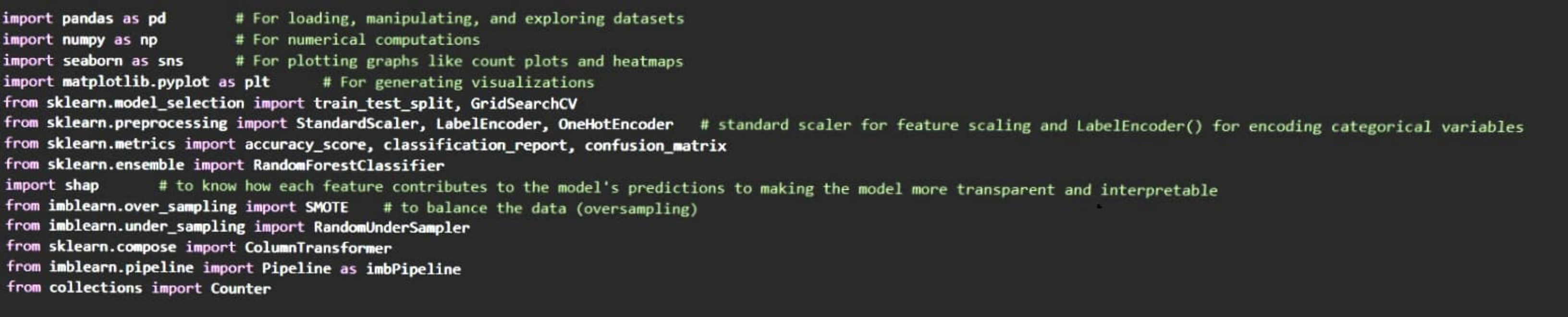
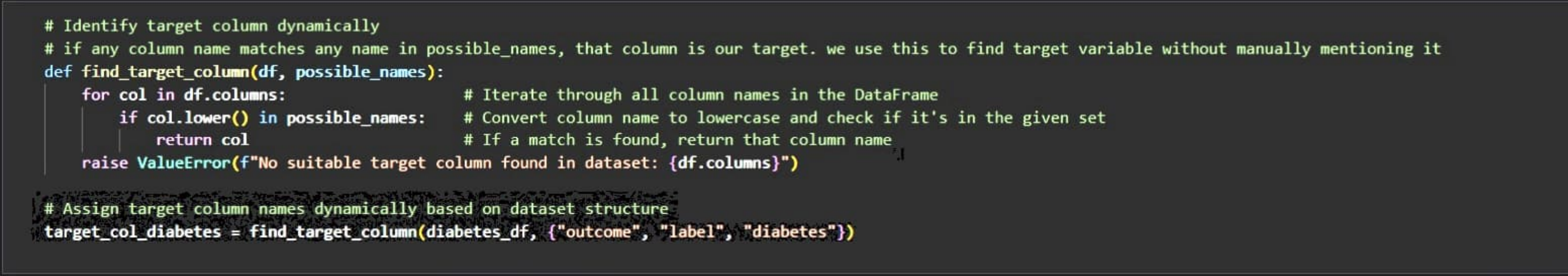
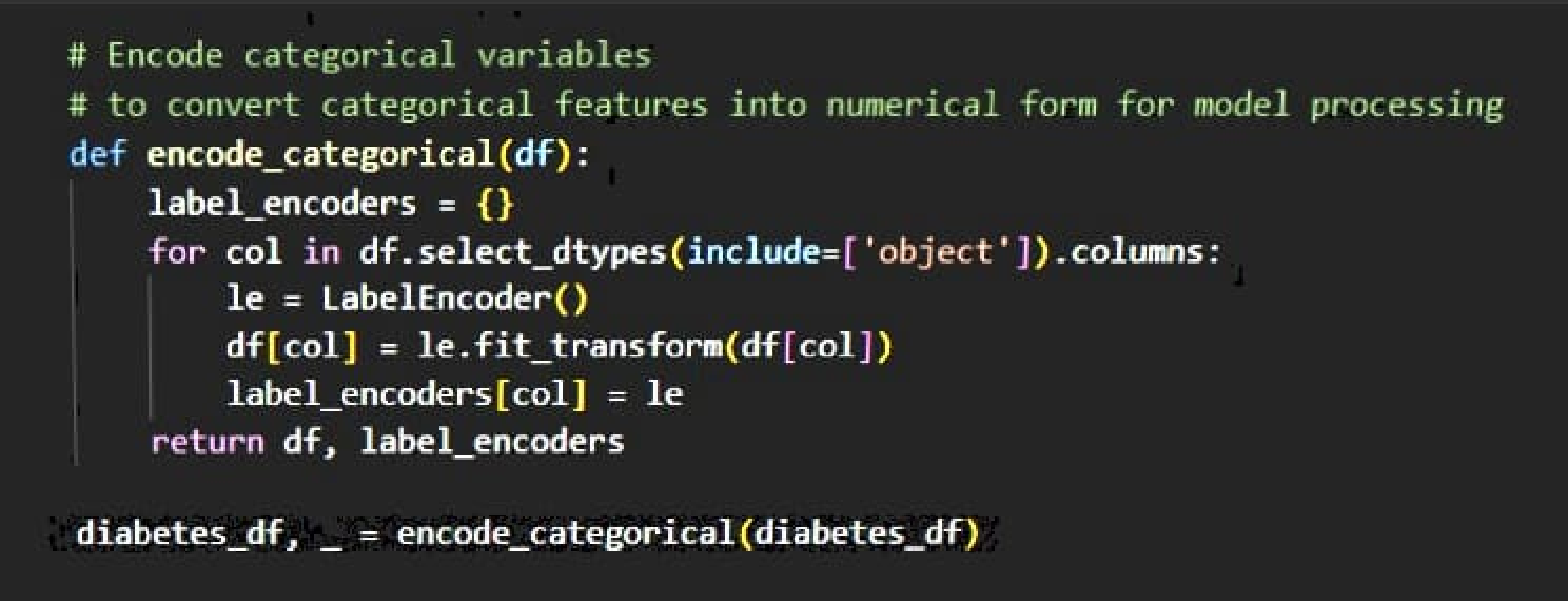


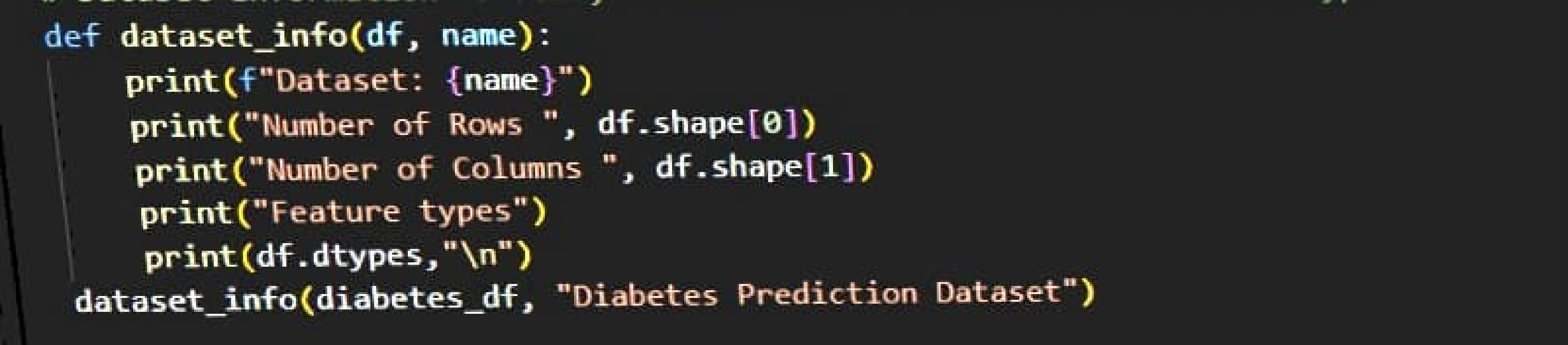
```python
import pandas as pd         # For loading, manipulating, and exploring datasets
import numpy as np          # For numerical computations
import seaborn as sns       # For plotting graphs like count plots and heatmaps
import matplotlib.pyplot as plt      # For generating visualizations
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler, LabelEncoder, OneHotEncoder   # standard scaler for feature scaling and LabelEncoder() for encoding categorical variables
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.ensemble import RandomForestClassifier
import shap       # to know how each feature contributes to the model's predictions to making the model more transparent and interpretable
from imblearn.over_sampling import SMOTE    # to balance the data (oversampling)
from imblearn.under_sampling import RandomUnderSampler
from sklearn.compose import ColumnTransformer
from imblearn.pipeline import Pipeline as imbPipeline
from collections import Counter
```



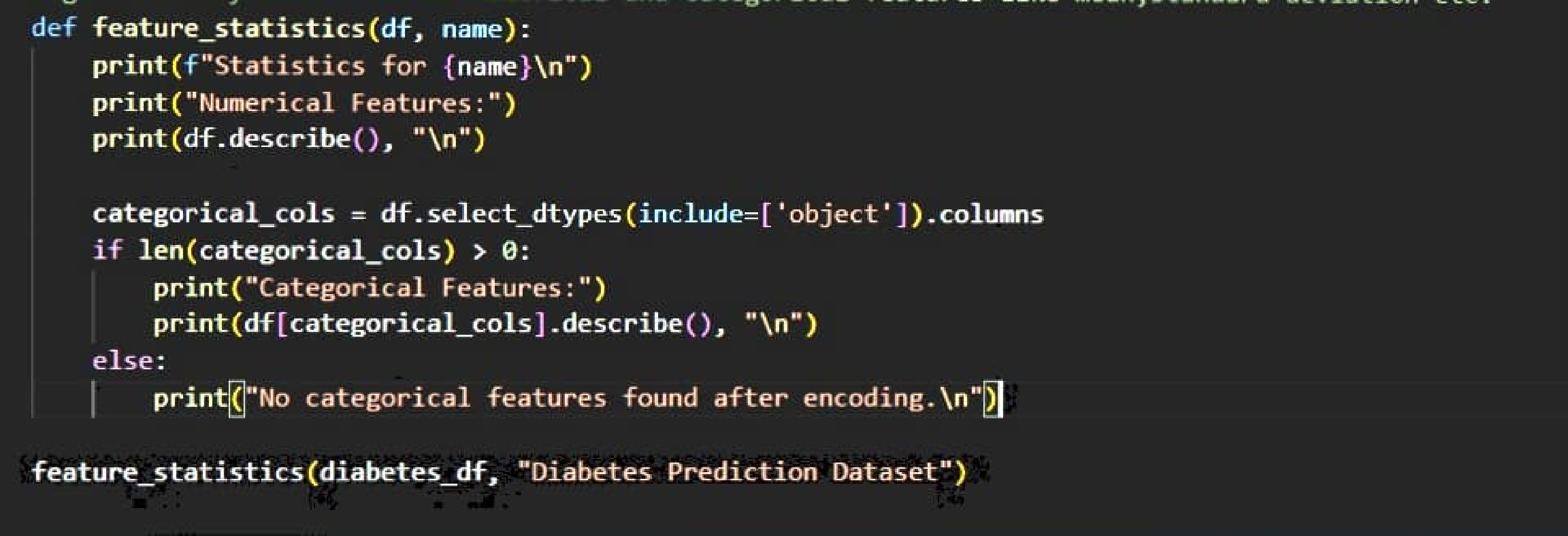
```python
# Identify target column dynamically
# if any column name matches any name in possible_names, that column is our target. we use this to find target variable without manually mentioning it
def find_target_column(df, possible_names):
    for col in df.columns:                   # Iterate through all column names in the DataFrame
        if col.lower() in possible_names:    # Convert column name to lowercase and check if it's in the given set
            return col                       # If a match is found, return that column name
    raise ValueError(f"No suitable target column found in dataset: {df.columns}")

# Assign target column names dynamically based on dataset structure
target_col_diabetes = find_target_column(diabetes_df, {"outcome", "label", "diabetes"})
```



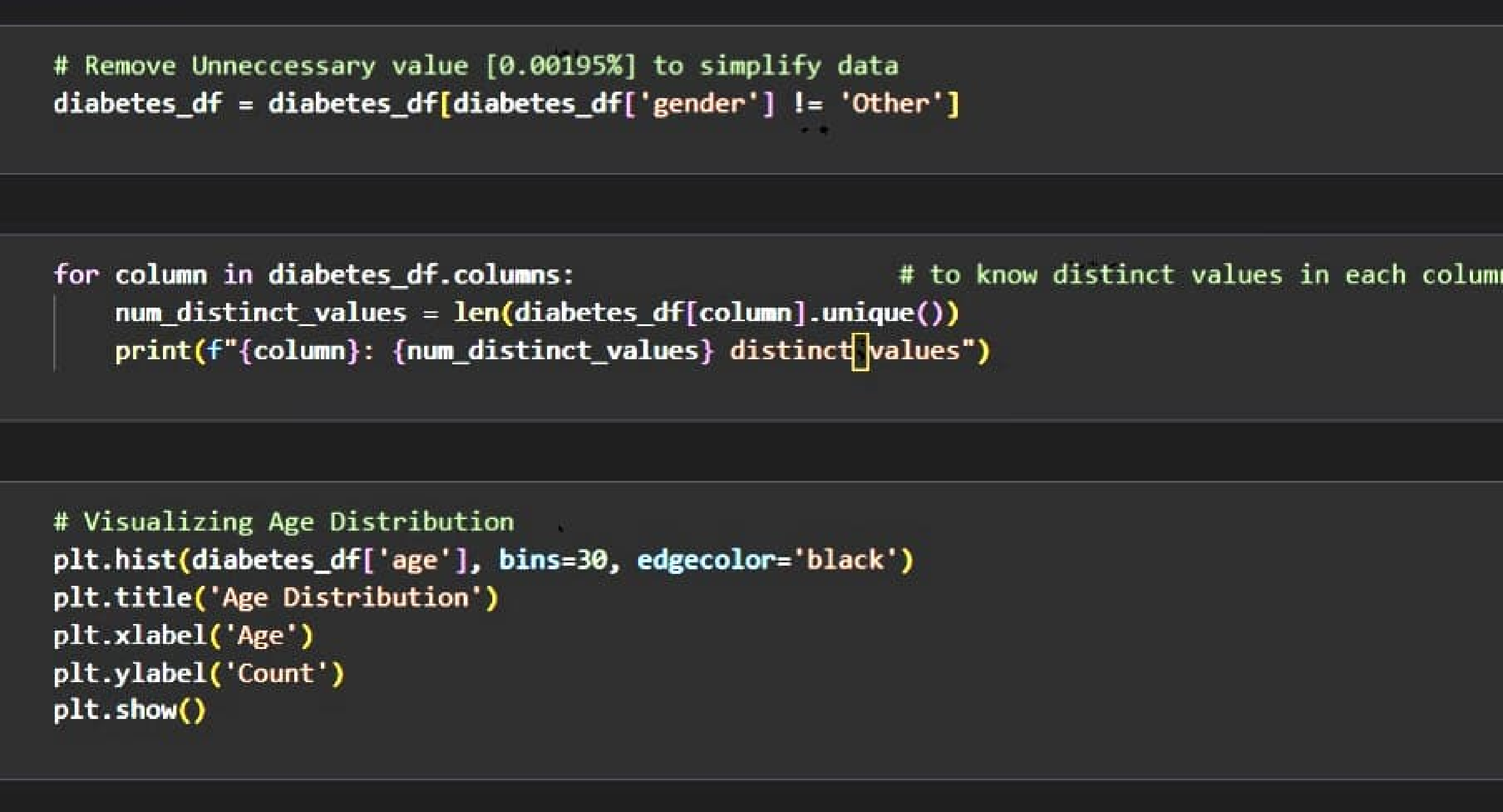
```python
# Encode categorical variables
# to convert categorical features into numerical form for model processing
def encode_categorical(df):
    label_encoders = {}
    for col in df.select_dtypes(include=['object']).columns:
        le = LabelEncoder()
        df[col] = le.fit_transform(df[col])
        label_encoders[col] = le
    return df, label_encoders

diabetes_df, _ = encode_categorical(diabetes_df)
```



```python
def dataset_info(df, name):
    print(f"Dataset: {name}")
    print("Number of Rows ", df.shape[0])
    print("Number of Columns ", df.shape[1])
    print("Feature types")
    print(df.dtypes,"\n")
dataset_info(diabetes_df, "Diabetes Prediction Dataset")
```



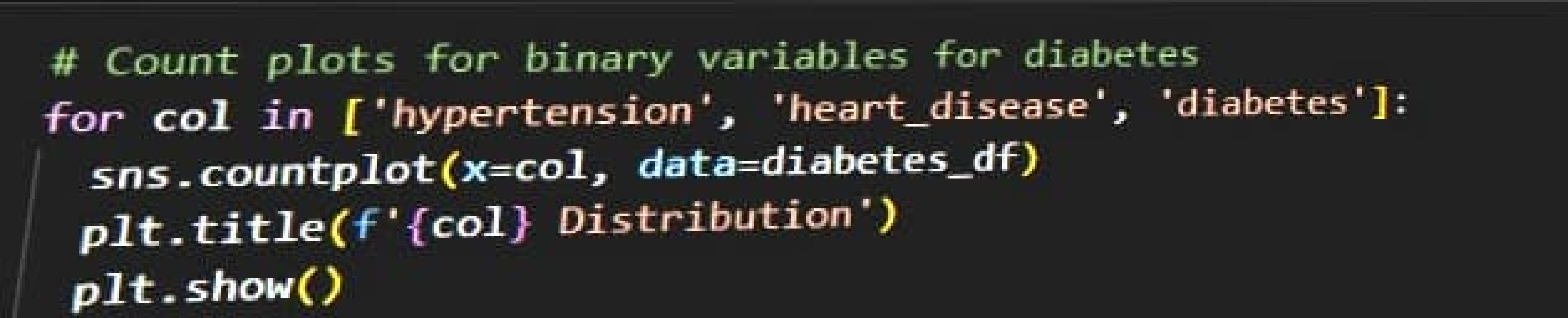
```python
def feature_statistics(df, name):
    print(f"Statistics for {name}\n")
    print("Numerical Features:")
    print(df.describe(), "\n")

    categorical_cols = df.select_dtypes(include=['object']).columns
    if len(categorical_cols) > 0:
        print("Categorical Features:")
        print(df[categorical_cols].describe(), "\n")
    else:
        print("No categorical features found after encoding.\n")

feature_statistics(diabetes_df, "Diabetes Prediction Dataset")
```



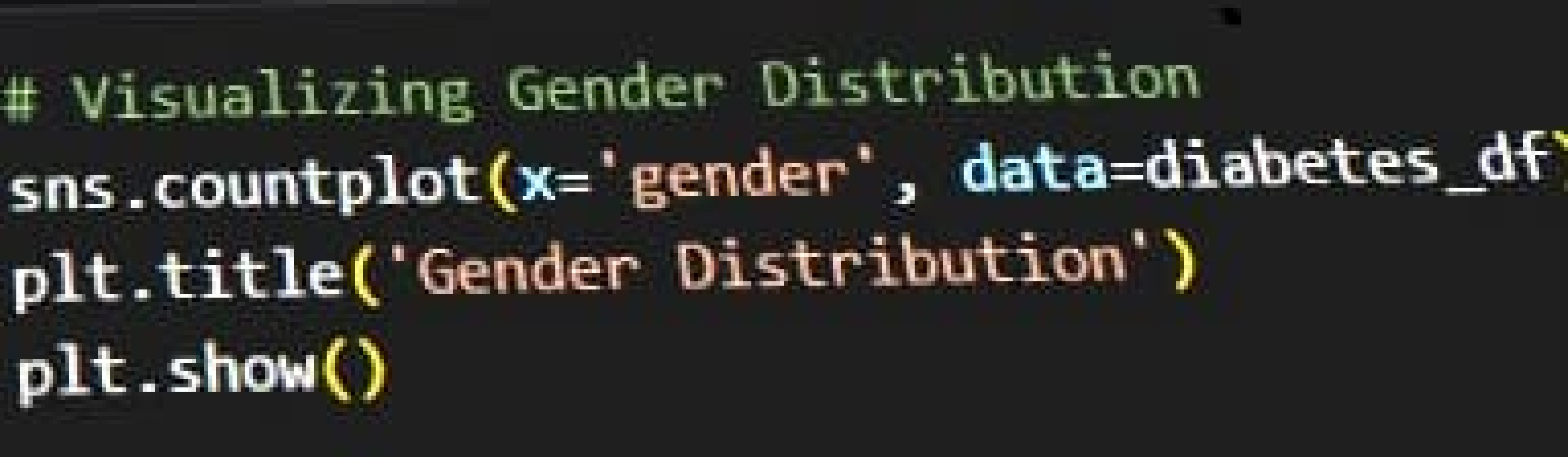
```python
# Remove Unneccessary value [0.00195%] to simplify data
diabetes_df = diabetes_df[diabetes_df['gender'] != 'Other']
```

```python
for column in diabetes_df.columns:                    # to know distinct values in each colum
    num_distinct_values = len(diabetes_df[column].unique())
    print(f"{column}: {num_distinct_values} distinct values")
```

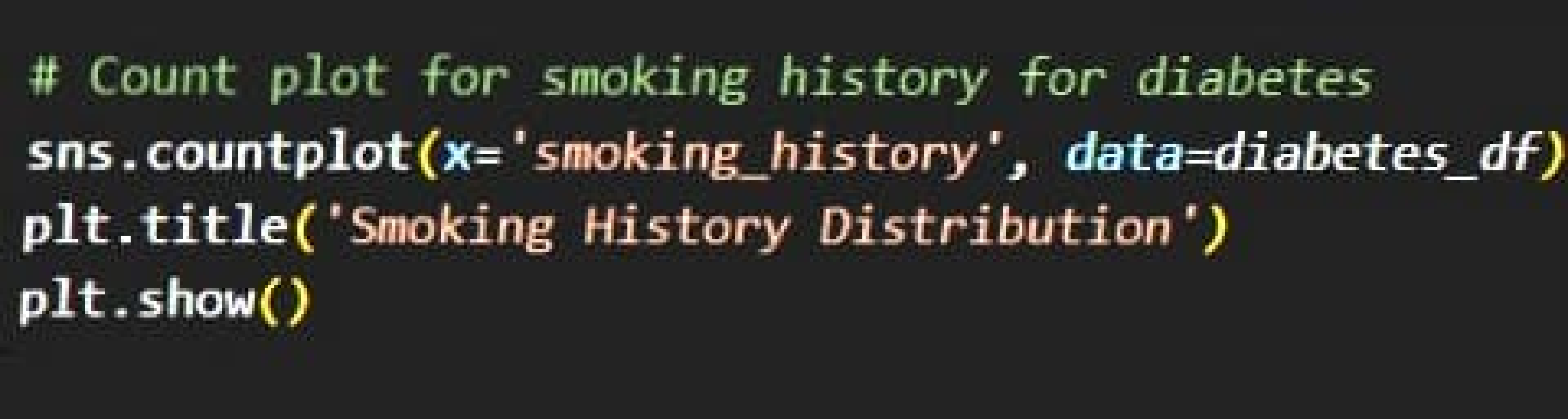
```python
# Visualizing Age Distribution
plt.hist(diabetes_df['age'], bins=30, edgecolor='black')
plt.title('Age Distribution')
plt.xlabel('Age')
plt.ylabel('Count')
plt.show()
```



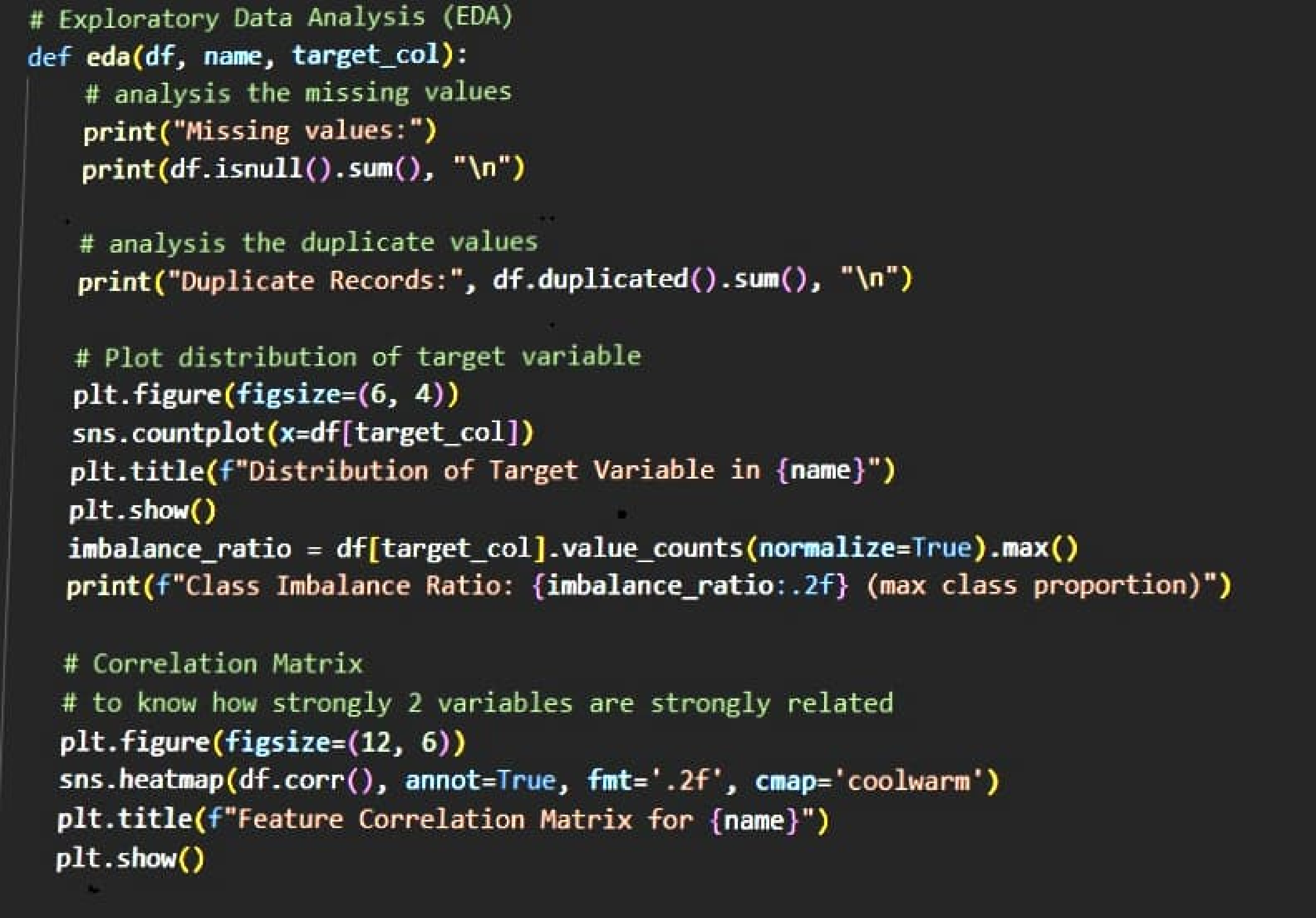
```python
# Count plots for binary variables for diabetes
for col in ['hypertension', 'heart_disease', 'diabetes']:
    sns.countplot(x=col, data=diabetes_df)
    plt.title(f'{col} Distribution')
    plt.show()
```



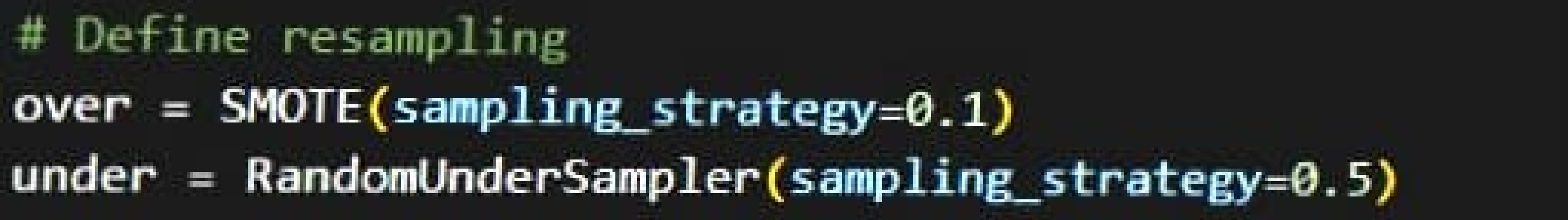
```python
# Visualizing Gender Distribution
sns.countplot(x='gender', data=diabetes_df
plt.title('Gender Distribution')
plt.show()
```



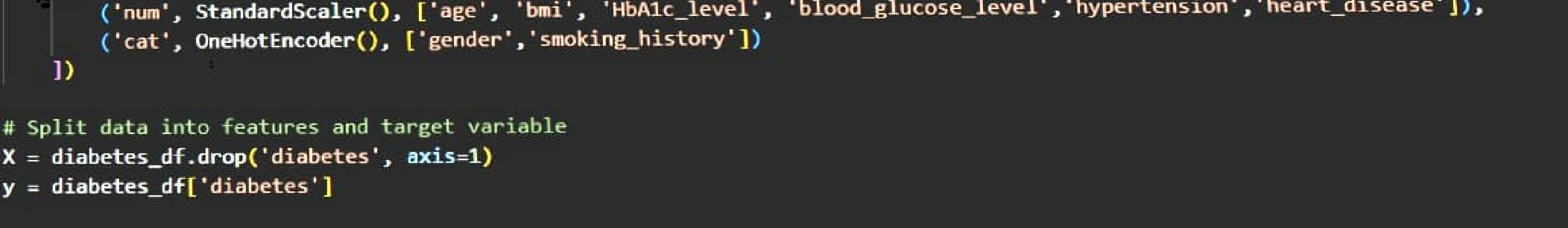
```python
# Count plot for smoking history for diabetes
sns.countplot(x='smoking_history', data=diabetes_df)
plt.title('Smoking History Distribution')
plt.show()
```



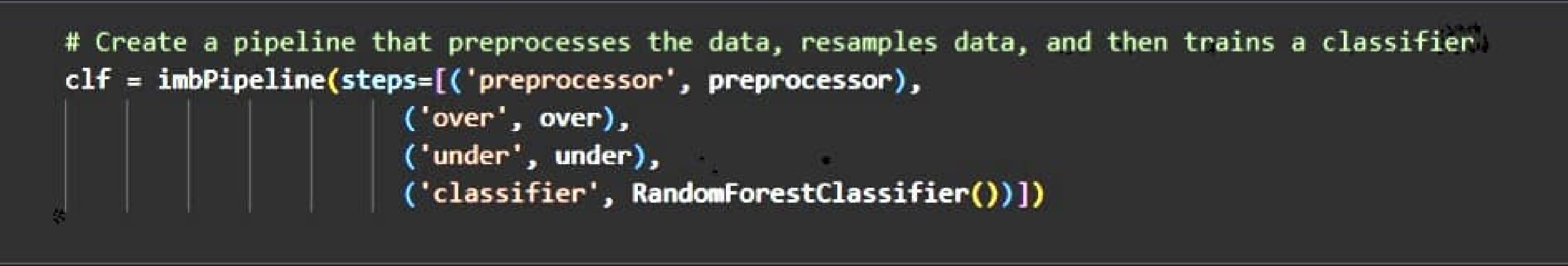
```python
# Exploratory Data Analysis (EDA)
def eda(df, name, target_col):
    # analysis the missing values
    print("Missing values:")
    print(df.isnull().sum(), "\n")

    # analysis the duplicate values
    print("Duplicate Records:", df.duplicated().sum(), "\n")

    # Plot distribution of target variable
    plt.figure(figsize=(6, 4))
    sns.countplot(x=df[target_col])
    plt.title(f"Distribution of Target Variable in {name}")
    plt.show()
    imbalance_ratio = df[target_col].value_counts(normalize=True).max()
    print(f"Class Imbalance Ratio: {imbalance_ratio:.2f} (max class proportion)")

    # Correlation Matrix
    # to know how strongly 2 variables are strongly related
    plt.figure(figsize=(12, 6))
    sns.heatmap(df.corr(), annot=True, fmt='.2f', cmap='coolwarm')
    plt.title(f"Feature Correlation Matrix for {name}")
    plt.show()
```



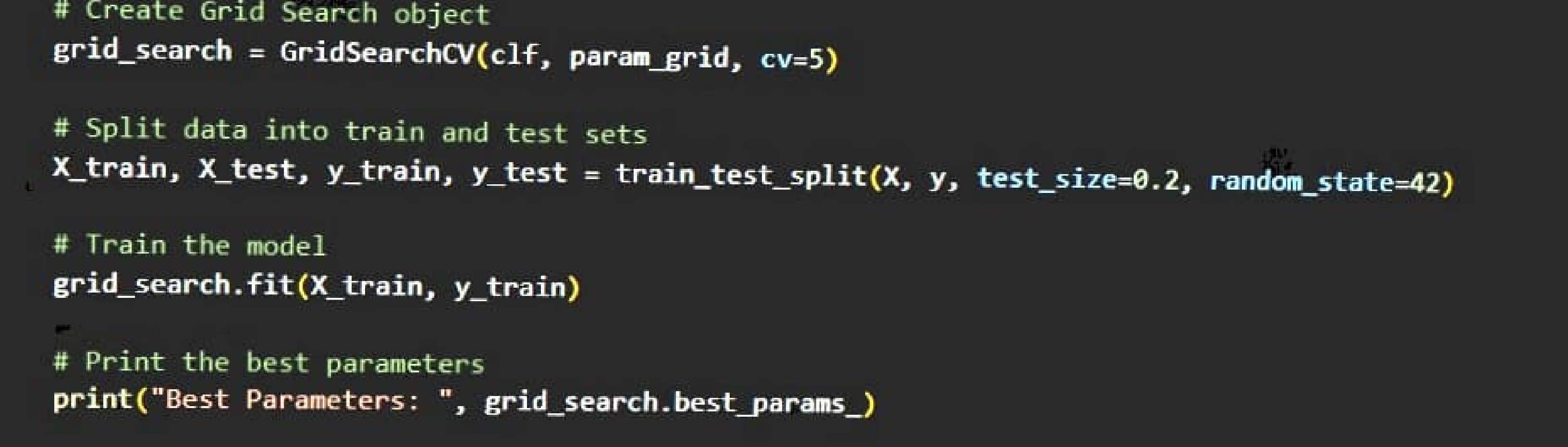
```
# Define resampling
over = SMOTE(sampling_strategy=0.1)
under = RandomUnderSampler(sampling_strategy=0.5)
```



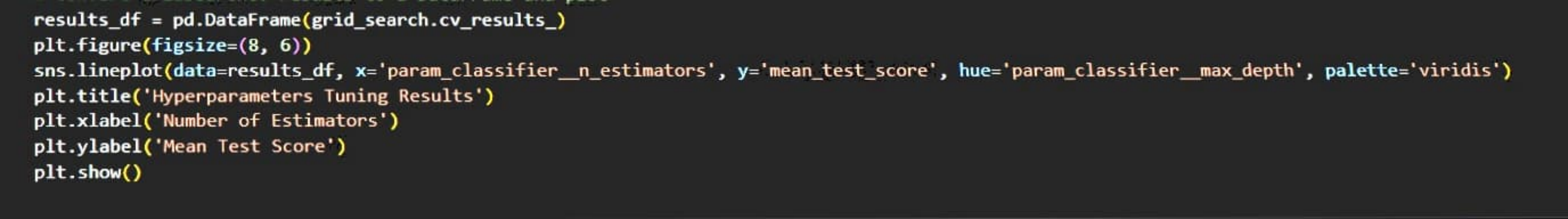
```
        ('num', StandardScaler(), ['age', 'bmi', 'HbA1c_level', 'blood_glucose_level','hypertension','heart_disease']),
        ('cat', OneHotEncoder(), ['gender','smoking_history'])
    ])

# Split data into features and target variable
X = diabetes_df.drop('diabetes', axis=1)
y = diabetes_df['diabetes']
```



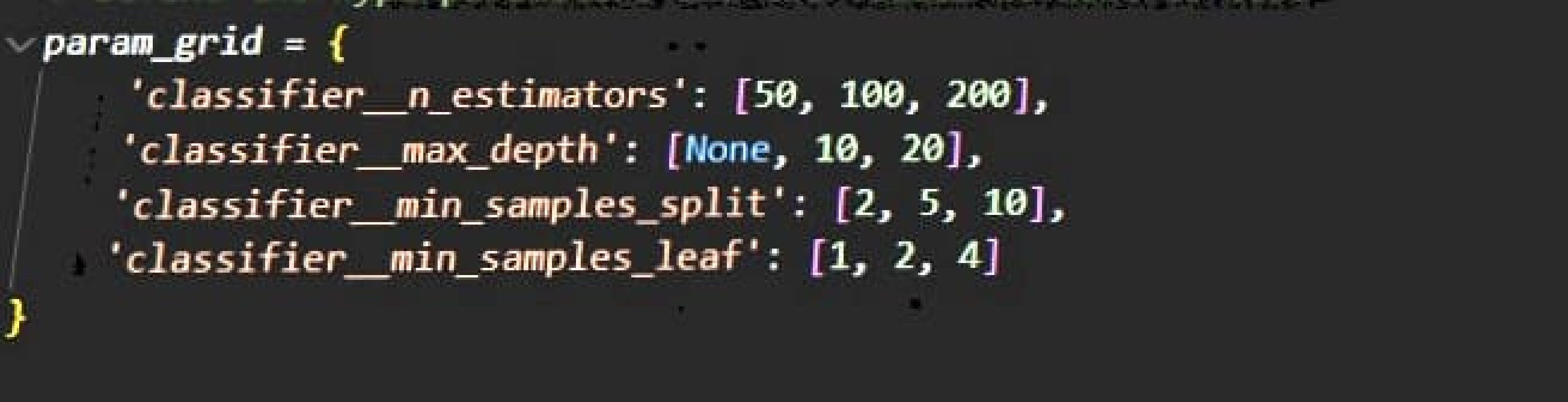
```python
# Create a pipeline that preprocesses the data, resamples data, and then trains a classifier
clf = imbPipeline(steps=[('preprocessor', preprocessor),
                         ('over', over),
                         ('under', under),
                         ('classifier', RandomForestClassifier())])
```



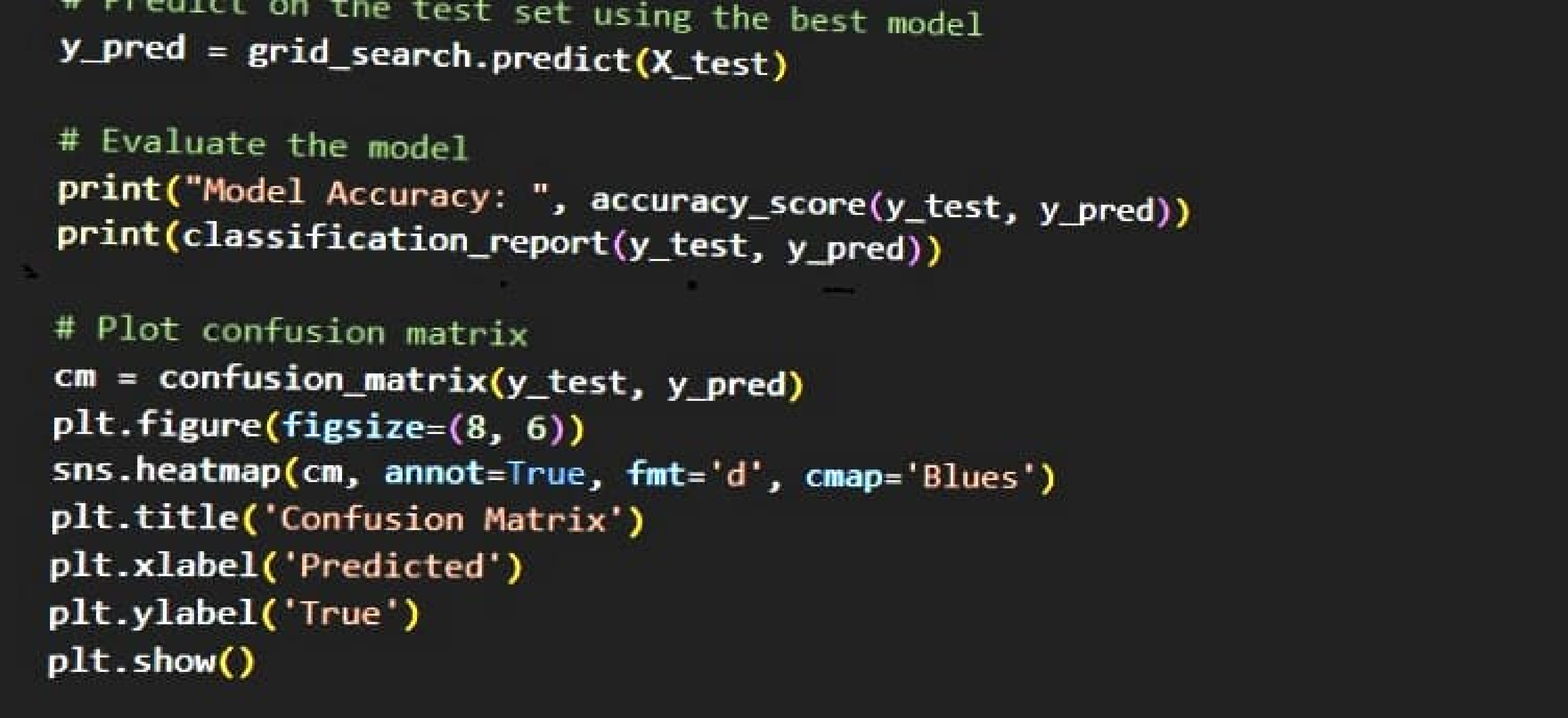
```python
# Create Grid Search object
grid_search = GridSearchCV(clf, param_grid, cv=5)

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train the model
grid_search.fit(X_train, y_train)

# Print the best parameters
print("Best Parameters: ", grid_search.best_params_)
```



```python
results_df = pd.DataFrame(grid_search.cv_results_)
plt.figure(figsize=(8, 6))
sns.lineplot(data=results_df, x='param_classifier__n_estimators', y='mean_test_score', hue='param_classifier__max_depth', palette='viridis')
plt.title('Hyperparameters Tuning Results')
plt.xlabel('Number of Estimators')
plt.ylabel('Mean Test Score')
plt.show()
```



```python
param_grid = {
    'classifier__n_estimators': [50, 100, 200],
    'classifier__max_depth': [None, 10, 20],
    'classifier__min_samples_split': [2, 5, 10],
    'classifier__min_samples_leaf': [1, 2, 4]
}
```



```python
# Predict on the test set using the best model
y_pred = grid_search.predict(X_test)

# Evaluate the model
print("Model Accuracy: ", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))

# Plot confusion matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```