FINAL v2

October 14, 2024

[32]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

df = pd.read_csv(
    "/Users/gabrielmancillas/Documents/GitHub/StudentPerformancePrediction/
  ↪dataset.csv"
)
```

[ ]:
```
curricular_units = df[
    [
        "Curricular units 1st sem (credited)",
        "Curricular units 1st sem (enrolled)",
        "Curricular units 1st sem (evaluations)",
        "Curricular units 1st sem (approved)",
        "Curricular units 1st sem (grade)",
        "Curricular units 1st sem (without evaluations)",
        "Curricular units 2nd sem (credited)",
        "Curricular units 2nd sem (enrolled)",
        "Curricular units 2nd sem (evaluations)",
        "Curricular units 2nd sem (approved)",
        "Curricular units 2nd sem (grade)",
        "Curricular units 2nd sem (without evaluations)",
    ]
]
curricular_units.head(20)
```

[ ]:
```
# numbers of students
df.shape
```

[35]:
```
df.rename(columns={"Nacionality": "Nationality"}, inplace=True)
```

[ ]:
```
df.describe().round(3)
```

[ ]:
```
df.info()
```

[ ]:
```
# Check if the column exists in the dataframe
if "Curricular units 1st sem (grade)" in df.columns:
    print(df["Curricular units 1st sem (grade)"])
else:
    print("Column not found. Available columns are:", df.columns)
```

[57]:
```
import warnings

warnings.filterwarnings("ignore")

# EDA for Debtor feature
plt.figure(figsize=(10, 6))
sns.countplot(data=df, x="Debtor", hue="Target", palette="viridis")
plt.title("Distribution of Debtor Status by Target")
plt.xlabel("Debtor Status")
plt.ylabel("Count")
plt.legend(title="Target")
plt.show()

# Ensure academic_features only includes existing columns
academic_features = [feature for feature in academic_features if feature in df.
 ↪columns]

# Compare the distribution of academic performance features
plt.figure(figsize=(18, 12))
for i, feature in enumerate(academic_features, 1):
    plt.subplot(2, 2, i)
    sns.boxplot(data=df, x="Target", y=feature, palette="viridis")
    plt.title(f"Distribution of {feature} by Target")
    plt.xlabel("Target")
    plt.ylabel(feature)

plt.tight_layout()
plt.show()

# Compare the distribution of financial features
plt.figure(figsize=(18, 12))
for i, feature in enumerate(financial_features, 1):
    plt.subplot(2, 2, i)
    sns.boxplot(data=df, x="Target", y=feature, palette="viridis")
    plt.title(f"Distribution of {feature} by Target")
    plt.xlabel("Target")
    plt.ylabel(feature)

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

[ ]:
```
import pandas as pd
import matplotlib.pyplot as plt

# Sample data based on the provided dataset
data_GDP = {
    "Unemployment rate": df["Unemployment rate"].tolist(),
    "Inflation rate": df["Inflation rate"].tolist(),
    "GDP": df["GDP"].tolist(),
    "Target": df["Target"].tolist(),
}
# Plotting the Unemployment rate, Inflation rate, and GDP for Dropout vs␣
 ↪Graduate
plt.figure(figsize=(10, 6))

# Plot Unemployment rate
plt.subplot(3, 1, 1)
df.groupby("Target")["Unemployment rate"].mean().plot(kind="bar",␣
 ↪color="skyblue")
plt.title("Average Unemployment Rate by Target")
plt.ylabel("Unemployment Rate")

# Plot Inflation rate
plt.subplot(3, 1, 2)
df.groupby("Target")["Inflation rate"].mean().plot(kind="bar",␣
 ↪color="lightgreen")
plt.title("Average Inflation Rate by Target")
plt.ylabel("Inflation Rate")

# Plot GDP
plt.subplot(3, 1, 3)
df.groupby("Target")["GDP"].mean().plot(kind="bar", color="salmon")
plt.title("Average GDP by Target")
plt.ylabel("GDP")

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

Here is a visualization of the macroeconomic indicators (Unemployment rate, Inflation rate, and GDP) at the time of students' enrollment, grouped by their target outcome (Dropout or Graduate).

The bar charts show the average values of these indicators for each group:

```
•   Unemployment rate: Higher for "Graduate" students in this small dataset.
•   Inflation rate: Lower for "Graduate" students, with some negative inflation observed.
•   GDP: Slightly higher for "Dropout" students in this dataset.
```

[ ]:
```
print(df.isna().sum())
print("Total Missing: ", df.isna().sum().sum())
```

we are working zero missing values

```
[ ]: print("Total Duplicates: ", df.duplicated().sum())
```

```
[ ]: df["Target"].value_counts()
```

```
[ ]: plt.figure(figsize=(10, 6))
     sns.countplot(data=df, x="Target", palette="viridis")
     plt.title("Distribution of Target Variable")
     plt.xlabel("Target")
     plt.ylabel("Count")
     plt.xticks(rotation=45)
     plt.show()
```

```
[44]: df = df[df.Target != "Enrolled"]
```

```
[ ]: df.shape
```

```
[ ]: freq_distribution = df["Target"].value_counts().to_frame(name="Count")
     freq_distribution["% of Total"] = (
         df["Target"].value_counts(normalize=True) * 100
     ).round(2)
     freq_distribution
```

```
[ ]: sns.set_style("whitegrid")
     plt.figure(figsize=(10, 6))
     sns.countplot(data=df, x="Target", palette="viridis")  # Changed palette to␣
      ↪'viridis'

     plt.ylabel("Total Students", fontsize=12)
     plt.xlabel(None)
     plt.title("Distribution of Target Variable", pad=20, fontsize=15)
     plt.xticks(fontsize=12)
     plt.yticks(fontsize=12)

     plt.show()
```

```
[ ]: sns.set_style("ticks")
     plt.figure(figsize=(10, 6))
     sns.countplot(data=df, x="Gender", hue="Target", palette="viridis")

     plt.xticks(ticks=[0, 1], labels=["Female", "Male"])
     plt.ylabel("Total Students", fontsize=12)
     plt.xlabel(None)
     plt.title("Distribution of Target by Gender", pad=20, fontsize=15)
     plt.xticks(fontsize=12)
     plt.yticks(fontsize=12)
plt.show()
```

[ ]:
```
# Calculate the crosstab of Target and Gender
ct_gender = pd.crosstab(df["Target"], df["Gender"])

# Rename columns for better readability
ct_gender.columns = ["Female", "Male"]

# Calculate the percentage distribution within each Target category
ct_gender_percentage = ct_gender.div(ct_gender.sum(axis=1), axis=0) * 100

# Display the crosstab with counts and percentages
ct_gender_combined = ct_gender.copy()
ct_gender_combined["Female (%)"] = ct_gender_percentage["Female"]
ct_gender_combined["Male (%)"] = ct_gender_percentage["Male"]

# Add a column for the total percentage
ct_gender_combined["Total (%)"] = (
    ct_gender_combined["Female (%)"] + ct_gender_combined["Male (%)"]
)
ct_gender_combined

# Plot the percentage distribution using a stacked bar plot
ct_gender_percentage.plot(
    kind="bar", stacked=True, figsize=(10, 6), color=["#1f77b4", "#ff7f0e"]
)
plt.title("Gender Distribution within Target Categories")
plt.xlabel("Target")
plt.ylabel("Percentage")
plt.legend(title="Gender")
plt.show()
```

[ ]:
```
import plotly.express as px

# Create an interactive histogram with more bins
fig = px.histogram(
    df,
    x="Age at enrollment",
    nbins=30,
    title="Distribution by Age",
    labels={"Age at enrollment": "Age at Enrollment", "count": "Total␣
 ↪Students"},
    color_discrete_sequence=["dodgerblue"],
)

# Customize the layout
fig.update_layout(
    title={"text": "Distribution by Age", "x": 0.5},
    xaxis_title="Age at Enrollment",
    yaxis_title="Total Students",
    bargap=0.1,
)

# Show the plot
fig.show()
```

[ ]:
```
# Create an interactive count plot
fig = px.histogram(
    df,
    x="Marital status",
    color="Target",
    barmode="group",
    title="Distribution of Target by Marital Status",
    labels={"Marital status": "Marital Status", "count": "Total Students"},
    color_discrete_sequence=["dodgerblue", "orange"],
)

# Customize the layout
fig.update_layout(
    title={"text": "Distribution of Target by Marital Status", "x": 0.5},
    xaxis_title="Marital Status",
    yaxis_title="Total Students",
    bargap=0.1,
)

# Change the x tick labels to the corresponding status
fig.update_xaxes(
    tickvals=[1, 2, 3, 4, 5, 6],
    ticktext=[
        "Single",
        "Married",
        "Widower",
        "Divorced",
        "Defacto union",
        "Legally separated",
    ],
)

# Show the plot
fig.show()
```

[ ]:
```
import plotly.express as px

# Group by Course and Target
```

```python
student_courses = (
    df.groupby(["Course", "Target"])
    .size()
    .reset_index()
    .pivot(columns="Target", index="Course", values=0)
)

# Rename the index with course names
student_courses = student_courses.rename(
    index={i + 1: category for i, category in enumerate(categories)}
)

# Ensure the 'Dropout' column exists
if "Dropout" not in student_courses.columns:
    student_courses["Dropout"] = student_courses[0]  # Assuming '0' represents␣
 ↪dropouts

# Calculate the percentage of Dropout and Graduate for each course
student_courses["Total"] = student_courses.sum(axis=1)
student_courses["Dropout (%)"] = (
    student_courses["Dropout"] / student_courses["Total"] * 100
).round(2)
student_courses["Graduate (%)"] = (
    student_courses["Graduate"] / student_courses["Total"] * 100
).round(2)

# Sort the data for plotting
student_courses_sorted = student_courses.sort_values(by="Total", ascending=True)

# Remove the 'Total' column
student_courses_sorted.drop(columns="Total", inplace=True)

# Generate the interactive plot
fig = px.bar(
    student_courses_sorted[["Dropout", "Graduate"]],
    orientation="h",
    title="Distribution of Dropout and Graduate by Course",
    labels={"value": "Total Students", "Course": "Course"},
    color_discrete_sequence=px.colors.qualitative.Pastel,
)

# Add percentage annotations inside the bars
for col in ["Dropout", "Graduate"]:
    for i, val in enumerate(student_courses_sorted[col]):
        percentage = student_courses_sorted[f"{col} (%)"].iloc[i]
        fig.add_annotation(
            x=(
                val / 2
                if col == "Dropout"
                else val + student_courses_sorted["Dropout"].iloc[i] / 2
            ),
            y=student_courses_sorted.index[i],
            text=f"{percentage}%",
            showarrow=False,
            xanchor="center",
            yanchor="middle",
            font=dict(size=12, color="black"),
        )

# Customize the layout
fig.update_layout(
    title={"text": "<b>Distribution of Dropout and Graduate by Course</b>", "x":
 ↪ 0.5},
    xaxis_title="<b>Total Students</b>",
    yaxis_title=None,
    barmode="stack",
    width=1200,  # Increase the width
    height=800,  # Increase the height
)

# Show the plot
fig.show()
```

[ ]:
```
# Create a new column 'Enrolled' that shows 0 for not enrolled and 1 for␣
 ↪enrolled
df["Enrolled"] = (
    (df["Curricular units 1st sem (enrolled)"] > 0)
    | (df["Curricular units 2nd sem (enrolled)"] > 0)
).astype(int)

# Display the first few rows to verify the new column
df.head()
```

[ ]:
```
# Group by Course and sum the 'Enrolled' column
enrolled_per_course = df.groupby("Course")["Enrolled"].sum()

# Calculate the total enrolled students
total_enrolled_students = df["Enrolled"].sum()

# Rename the courses for better readability
enrolled_per_course = enrolled_per_course.rename(
    index={
        1: "Biofuel Production Technologies",
        2: "Animation and Multimedia Design",
        3: "Social Service (evening attendance)",
        4: "Agronomy",
        5: "Communication Design",
        6: "Veterinary Nursing",
        7: "Informatics Engineering",
        8: "Equinculture",
        9: "Management",
        10: "Social Service",
        11: "Tourism",
        12: "Nursing",
        13: "Oral Hygiene",
        14: "Advertising and Marketing Management",
        15: "Journalism and Communication",
        16: "Basic Education",
        17: "Management (evening attendance)",
    }
)

# Display the results
print("Enrolled Students per Course:")
print(enrolled_per_course)
print("\nTotal Enrolled Students:", total_enrolled_students)
```

[ ]:
```
# Calculate the Dropout Rate and Graduate Rate
student_courses_sorted["Dropout Rate"] = (
    student_courses_sorted["Dropout"] / student_courses_sorted.sum(axis=1) * 100
).round(3)
student_courses_sorted["Graduate Rate"] = (
    student_courses_sorted["Graduate"] / student_courses_sorted.sum(axis=1) *␣
 ↪100
).round(3)

# Create a new DataFrame with only Dropout Rate and Graduate Rate
dropout_graduate_rates = student_courses_sorted[
    ["Dropout Rate", "Graduate Rate"]
].copy()

# Display the new DataFrame
dropout_graduate_rates
```

0.0.1 Feature Selection

[ ]:
```
df = pd.get_dummies(df, columns=["Target"])
df.head()
```

[ ]:
```
dummies_to_drop = ["Target_Graduate"]
df.drop(columns=dummies_to_drop, inplace=True)
df.rename(columns={"Target_Dropout": "Target"}, inplace=True)

df.head()
```

[58]:
```
# Set display options to show all columns and rows
pd.set_option("display.max_columns", None)
pd.set_option("display.max_rows", None)

# Calculate the correlation matrix and round it to 2 decimal places
correlation_matrix = df.corr().round(2)

# Display the correlation matrix
correlation_matrix

# Reset display options to default values
pd.reset_option("display.max_columns")
pd.reset_option("display.max_rows")
```

[ ]:
```
sns.set(rc={"figure.figsize": (24, 20)})  # Increased the figure size
sns.heatmap(correlation_matrix, annot=True, cmap="viridis", fmt=".2f")
plt.title("Correlation Matrix")
plt.show()
```

[60]:
```
# Demographic
demographics = df[
    [
        "Marital status",
        "Nationality",
        "Displaced",
        "Gender",
        "Age at enrollment",
        "International",
        "Target",
    ]
]
# Academic
academic_path = df[
    [
        "Curricular units 1st sem (credited)",
        "Curricular units 1st sem (enrolled)",
        "Curricular units 1st sem (evaluations)",
        "Curricular units 1st sem (approved)",
        "Curricular units 1st sem (grade)",
        "Curricular units 1st sem (without evaluations)",
        "Curricular units 2nd sem (credited)",
        "Curricular units 2nd sem (enrolled)",
        "Curricular units 2nd sem (evaluations)",
        "Curricular units 2nd sem (approved)",
        "Curricular units 2nd sem (grade)",
        "Curricular units 2nd sem (without evaluations)",
        "Target",
    ]
]
```

[ ]:
```
sns.set(rc={"figure.figsize": (10, 8)})
sns.heatmap(
    demographics.corr().round(2),
    linewidths=0.5,
    annot=True,
    annot_kws={"size": 10},
    cmap="viridis",
    cbar_kws={"shrink": 0.8},
    fmt=".2f",
)

plt.title("Demographics Collinearity Heatmap", pad=20, fontsize=15)
plt.xticks(rotation=45, ha="right", fontsize=12)
plt.yticks(fontsize=12)
plt.show()
```

[ ]:
```
features_to_drop = ["Nationality", "International"]
features_to_drop
```

[ ]:
```
features_to_drop.extend(
    [
        "Curricular units 1st sem (credited)",
        "Curricular units 1st sem (enrolled)",
        "Curricular units 1st sem (evaluations)",
        "Curricular units 1st sem (approved)",
        "Curricular units 1st sem (grade)",
        "Curricular units 1st sem (without evaluations)",
        "Curricular units 2nd sem (credited)",
        "Curricular units 2nd sem (without evaluations)",
    ]
)
features_to_drop
```

[ ]:
```
df.drop(features_to_drop, axis=1, inplace=True)
df.head()
```

[ ]:
```
df.corr()["Target"]
```

0.0.2 Logistic regression

[66]:
```
X = df.drop(columns="Target", axis=1)
y = df["Target"]
```

[ ]:
```
X.shape
```

[ ]:
```
print("X: ", type(X))
print("y: ", type(y))
```

[ ]:
```
# Step 1: Import necessary libraries
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,␣
 ↪classification_report

# Step 2: Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Step 3: Standardize the feature variables (scaling)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Step 4: Define hyperparameter grid
param_grid = {
    "C": [0.01, 0.1, 1, 10, 100],
    "penalty": ["l1", "l2"],
    "solver": ["liblinear", "saga"],
}

# Step 5: Perform Grid Search with cross-validation
logreg = LogisticRegression(max_iter=500, random_state=42)
grid_search = GridSearchCV(logreg, param_grid, cv=5, scoring="accuracy")
grid_search.fit(X_train_scaled, y_train)

# Display best hyperparameters
print(f"Best Hyperparameters: {grid_search.best_params_}")

# Step 6: Train the logistic regression model with the best hyperparameters
best_logreg = grid_search.best_estimator_
best_logreg.fit(X_train_scaled, y_train)

# Step 7: Make predictions on the test set
```

```
y_pred = best_logreg.predict(X_test_scaled)

# Step 8: Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Test Set Accuracy: {accuracy:.2f}")

# Confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(conf_matrix)

# Classification report
class_report = classification_report(y_test, y_pred)
print("Classification Report:")
print(class_report)
```

[ ]:
```
from sklearn.metrics import ConfusionMatrixDisplay

# Import necessary library for plotting

# Plot the confusion matrix
ConfusionMatrixDisplay.from_predictions(y_test, y_pred, cmap="viridis")
plt.title("Confusion Matrix")
plt.show()
```

[ ]:
```
from sklearn.metrics import accuracy_score, recall_score, precision_score,␣
 ↪f1_score

# Calculate and print the evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print(f"Accuracy for testing data: {accuracy:.3f}")
print(f"Recall for testing data: {recall:.3f}")
print(f"Precision for testing data: {precision:.3f}")
print(f"F1 Score for testing data: {f1:.3f}")
```

[ ]:
```
# Step 1: Import necessary libraries
from sklearn.model_selection import train_test_split, GridSearchCV,␣
 ↪cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix,␣
 ↪classification_report
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
```

```python
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier

# Step 2: Load and preprocess the data
# Assuming df is your DataFrame and 'Target' is the column you want to predict
X = df.drop(columns="Target")  # Features
y = df["Target"]  # Target variable

# Step 3: Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Step 4: Standardize the feature variables
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Step 5: Define the models and their hyperparameter grids
models = {
    "Logistic Regression": {
        "model": LogisticRegression(max_iter=500, random_state=42),
        "params": {
            "C": [0.01, 0.1, 1, 10, 100],
            "penalty": ["l1", "l2"],
            "solver": ["liblinear", "saga"],
        },
    },
    "Random Forest": {
        "model": RandomForestClassifier(random_state=42),
        "params": {
            "n_estimators": [50, 100, 200],
            "max_depth": [None, 10, 20, 30],
            "min_samples_split": [2, 5, 10],
        },
    },
    "Support Vector Machine": {
        "model": SVC(random_state=42),
        "params": {
            "C": [0.1, 1, 10, 100],
            "kernel": ["linear", "rbf", "poly"],
            "gamma": ["scale", "auto"],
        },
    },
    "Gradient Boosting": {
        "model": GradientBoostingClassifier(random_state=42),
        "params": {
            "n_estimators": [50, 100, 200],
            "learning_rate": [0.01, 0.1, 0.2],
            "max_depth": [3, 5, 7],
        },
    },
    "K-Nearest Neighbors": {
        "model": KNeighborsClassifier(),
        "params": {
            "n_neighbors": [3, 5, 7, 9],
            "weights": ["uniform", "distance"],
            "metric": ["euclidean", "manhattan"],
        },
    },
    "Decision Tree": {
        "model": DecisionTreeClassifier(random_state=42),
        "params": {
            "max_depth": [None, 10, 20, 30],
            "min_samples_split": [2, 5, 10],
            "criterion": ["gini", "entropy"],
        },
    },
}

# Step 6: Perform Grid Search with cross-validation for each model
best_models = {}
for name, model_info in models.items():
    grid_search = GridSearchCV(
        model_info["model"], model_info["params"], cv=5, scoring="accuracy"
    )
    grid_search.fit(X_train_scaled, y_train)
    best_models[name] = grid_search.best_estimator_
    print(f"{name} - Best Hyperparameters: {grid_search.best_params_}")

# Step 7: Train each model on the training data and evaluate on the test set
for name, model in best_models.items():
    model.fit(X_train_scaled, y_train)
    y_pred = model.predict(X_test_scaled)

    # Evaluate the model
    accuracy = accuracy_score(y_test, y_pred)
    print(f"\n{name} - Test Set Accuracy: {accuracy:.2f}")

    # Confusion matrix
    conf_matrix = confusion_matrix(y_test, y_pred)
    print(f"{name} - Confusion Matrix:")
    print(conf_matrix)
    # Classification report
    class_report = classification_report(y_test, y_pred)
    print(f"{name} - Classification Report:")
    print(class_report)
```

[ ]:
```
# Step 1: Import necessary libraries
from sklearn.metrics import roc_curve, auc, roc_auc_score
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.metrics import precision_recall_curve
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
import pandas as pd

# Assuming X_train_scaled, y_train, X_test_scaled, and y_test are already␣
↪defined

# Train the RandomForestClassifier
rf_clf = RandomForestClassifier(random_state=42)
rf_clf.fit(X_train_scaled, y_train)

# Predict probabilities
y_pred_prob = rf_clf.predict_proba(X_test_scaled)[:, 1]

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

# Plot the ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color="blue", label=f"ROC Curve (AUC = {roc_auc:.2f})")
plt.plot([0, 1], [0, 1], color="red", linestyle="--")  # Diagonal line
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Receiver Operating Characteristic (ROC)")
plt.legend(loc="lower right")
plt.show()


# Step 7: Cumulative Gains Curve
def plot_cumulative_gains(y_true, y_pred_proba):
    data = pd.DataFrame({"true": y_true, "probability": y_pred_proba}).
↪sort_values(
        by="probability", ascending=False
    )

    total_positive = np.sum(data["true"])
    cumulative_gains = np.cumsum(data["true"]) / total_positive
    cumulative_percentage = np.arange(1, len(data) + 1) / len(data)

    plt.figure(figsize=(8, 6))
    plt.plot(
        cumulative_percentage,
        cumulative_gains,
        label="Cumulative Gains Curve",
        color="blue",
    )
    plt.plot([0, 1], [0, 1], linestyle="--", color="red", label="Baseline")
    plt.xlabel("Percentage of Samples")
    plt.ylabel("Cumulative True Positives")
    plt.title("Cumulative Gains Curve")
    plt.legend(loc="lower right")
    plt.grid(True)
    plt.show()


# Call the cumulative gains function
plot_cumulative_gains(y_test, y_pred_prob)
```

[ ]:
```
import joblib

# Save the trained logistic regression model
joblib.dump(logreg, "logistic_regression_model.pkl")

# Save the scaler
joblib.dump(scaler, "scaler.pkl")
```

End of Document