Department of Mathematics and Computer Science Faculty
of Data Analysis

UNIME

Machine Learning

Report **Final Project**

Student: Moldobaeva Adinai (551567)

Duishenalieva Milena (551401)

Kuvondikov Dilmurod (550565)

Professor: Fiumara Giacomo

Machine Learning Report

1. Dataset Exploration

1.1 Dataset Overview

The first step is to import the dataset and examine its structure, including the number of rows, data types, and the number of features. We can see that there are 11 columns and 9000 rows. 8 columns are numerical features and 2 are categorical. The data types are: float64 for numerical features and object for categorical features.

```
df = pd.read_csv('dataset.csv')
print(df.shape)
print(df.head(10))
```

```
(9000, 11)
   feature_1  feature_2  feature_3  feature_4  feature_5  feature_6  \
0   0.496714   1.146509  -0.648521   0.833005   0.784920  -2.209437
1  -0.138264  -0.061846        NaN   0.403768   0.704674  -2.498565
2   0.647689   1.395115  -0.764126   1.708266  -0.250029   1.956259
3   1.523030   2.657560  -2.461653   2.649051   0.882201   3.445638
4  -0.234153  -0.499391   0.576097  -0.441656   0.610601   0.211425
5  -0.234137  -0.699415   0.268972  -0.702775   0.702283  -0.332383
6   1.579213   3.117904  -2.885133   3.312708   0.864708   2.045283
7   0.767435   1.730870  -1.445877   1.411070   0.874003   0.674730
8  -0.469474  -0.877919   0.575087  -0.532917  -0.519870        NaN
9   0.542560   1.314738  -0.403383   1.456165  -0.744625   1.987345

   feature_7  feature_8     category_1 category_2  target
0  -1.300105  -2.242241  Above Average   Region C       1
1  -1.339227  -1.942298  Below Average   Region A       0
2   1.190238   1.503559           High   Region C       1
3   2.120913   3.409035           High   Region B       1
4   0.935759  -0.401463  Below Average   Region C       0
5   0.453958  -0.826721  Below Average   Region A       0
6   1.531547   1.771851           High   Region A       1
7   0.812931   1.489838           High   Region A       1
8  -3.002925  -4.779960  Below Average   Region A       0
9   0.431966   3.309386           High   Region C       1
```

We also check the entire dataset for missing values. Features with missing values: feature_3 (400 missing values), feature_6 (500 missing values).

```
print(df.isnull().sum())
```

```
feature_1       0
feature_2       0
feature_3     400
feature_4       0
feature_5       0
feature_6     500
feature_7       0
feature_8       0
category_1      0
category_2      0
target          0
dtype: int64
```

The data is divided into numerical and categorical features. Now, let's focus on the numerical data and provide an overview of it, as the categorical data has not been transformed yet. We will perform an analysis and output the maximum, minimum, mean, and percentile values.

```
# Analysis numerical data
numerical_columns = df.select_dtypes(include=["float64", "int64"]).columns
print("\nnumerical columns:")
print(df[numerical_columns].describe())
```

```
numerical columns:
         feature_1    feature_2    feature_3    feature_4    feature_5  \
count  9000.000000  9000.000000  8600.000000  9000.000000  9000.000000
mean      0.000427     0.003349     0.003235    -0.008481    -0.002177
std       1.241318     2.508324     1.542901     2.061784     0.577415
min     -18.665400   -37.852816    -6.676680    -8.190124    -0.999791
25%      -0.680062    -1.382610    -1.022085    -1.399928    -0.502614
50%      -0.003938    -0.016698     0.005196    -0.019541     0.001695
75%       0.680513     1.380228     1.038571     1.394151     0.497004
max      21.934496    47.603454     6.203055     8.189001     0.999914

         feature_6    feature_7    feature_8       target
count  8500.000000  9000.000000  9000.000000  9000.000000
mean     -0.006447     0.000592     0.003348     0.475444
std       1.981615     1.075064     2.043643     0.499424
min      -8.590782    -4.422265    -9.474989     0.000000
25%      -1.329040    -0.700078    -1.356620     0.000000
50%      -0.003137    -0.000097    -0.007584     0.000000
75%       1.324897     0.731942     1.402024     1.000000
max       6.803751     3.857219     7.572578     1.000000
```

1.2 Visualization of Distribution

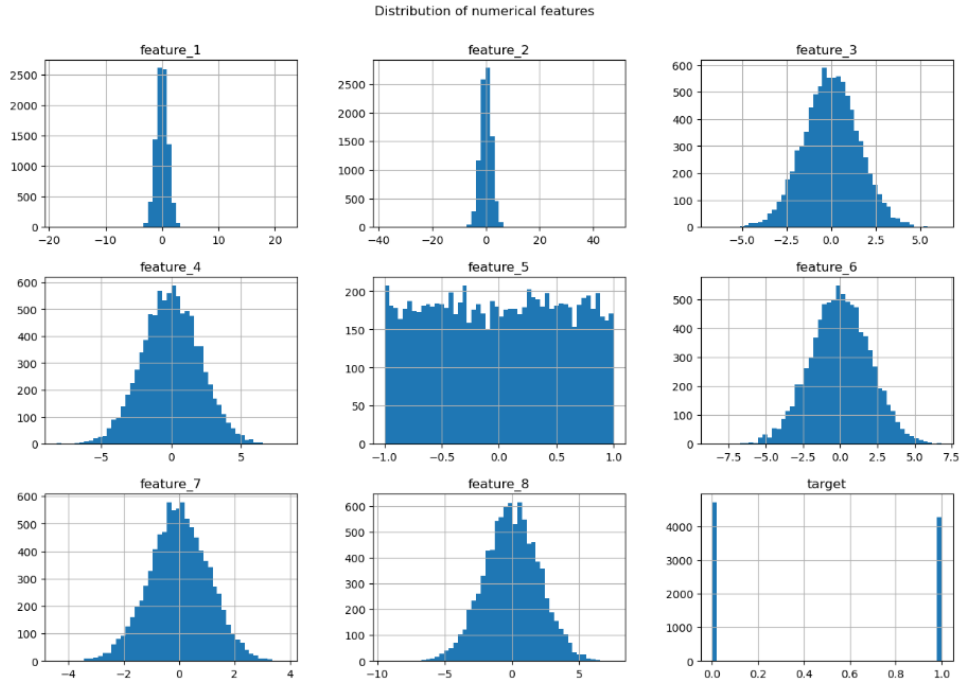
This code plots histograms for all numerical features in the DataFrame. It is useful for analyzing the distribution of data, identifying shifts, skewness, or outliers.

These features have nearly normal distributions: **feature_1, feature_2, feature_3, feature_4, feature_7, feature_8**

feature_5: The distribution is uniform, which may indicate a categorical feature encoded as numbers.

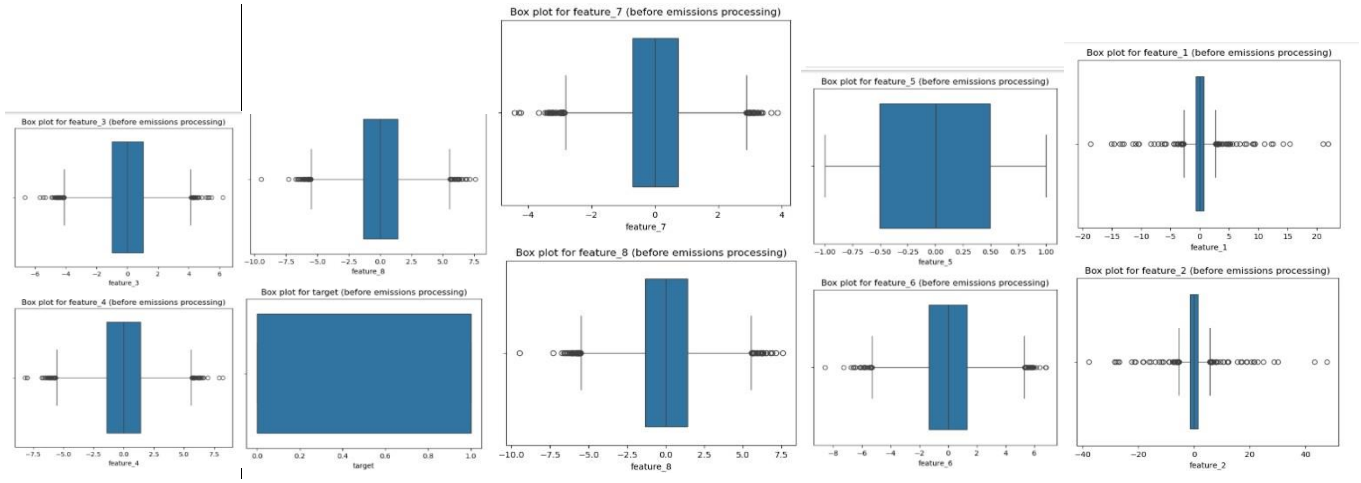
target: This is the target variable (binary classification: values 0 and 1).

```
import matplotlib.pyplot as plt

# Histograms of numerical features
df[numerical_columns].hist(bins=50, figsize=(15, 10))
plt.suptitle("Distribution of numerical features", y=0.95)
plt.show()
```



Now, creates a boxplot for each numerical feature in the DataFrame (df) to visualize the data distribution and identify outliers before data preprocessing. All features have outliers except for feature_5 and the target variable.

```
#Box plot for numeric features (before outlier processing)
import seaborn as sns
for col in numerical_columns:
    plt.figure(figsize=(6, 4))
    sns.boxplot(x=df[col])
    plt.title(f"Box plot for {col} (before emissions processing)")
    plt.show()
```

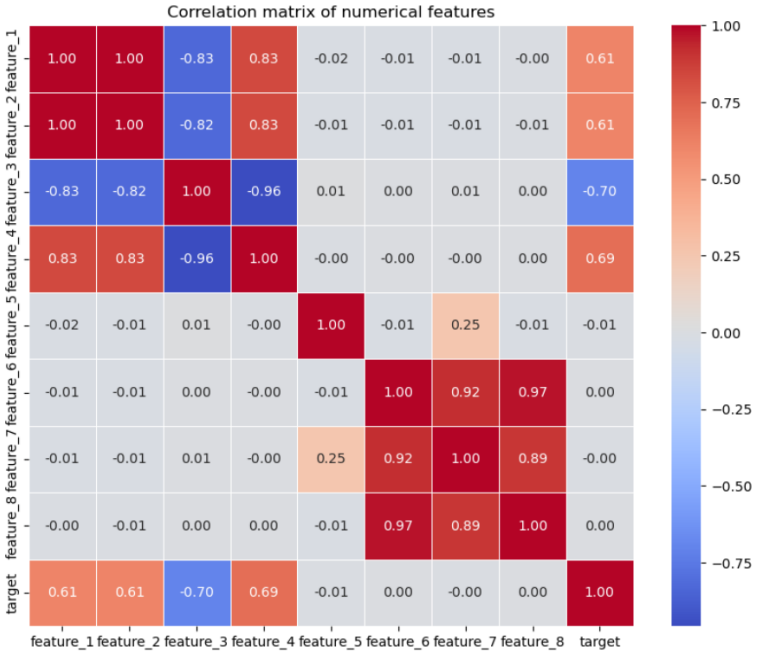


1.3 Analysis of Relationships Between Features

This code is used to analyze the correlation between numerical features in the dataset and visualize this correlation as a heatmap.

- 1: complete positive linear correlation.
- -1: complete negative linear correlation.
- 0: no linear correlation.

```
#Selecting only numeric columns
numerical_columns = df.select_dtypes(include=["float64", "int64"]).columns

#Correlation matrix
corr_matrix = df[numerical_columns].corr()

#Visualization of the correlation matrix
plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, annot=True, cmap="coolwarm", fmt=".2f", linewidths=0.5)
plt.title("Correlation matrix of numerical features")
plt.show()
```



Key observations:

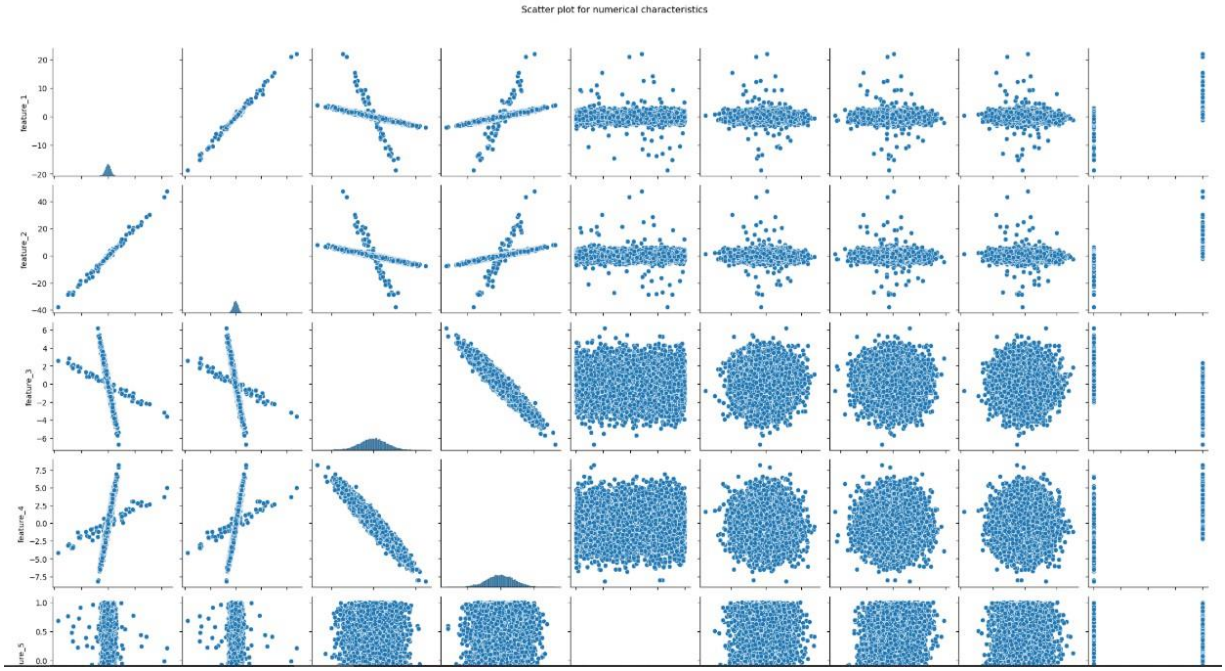
Relationship with the target variable (target):

- feature_1 and feature_2**:** correlation of +0.61 → positively correlated with the target — informative features.
- feature_4: correlation of +0.69 → very strong positive correlation.
- feature_3: correlation of -0.70 → strong negative correlation, also an informative feature.
- feature_5**,** feature_6**,** feature_7**,** and feature_8**:** correlation is approximately **0** → almost no relationship with the target, likely non-informative features.

Let's create a scatter plot matrix for all pairs of numerical features in the data. Each plot shows how two features relate to each other. This helps to:

- Identify linear and nonlinear relationships between features.
- Find outliers.
- Assess multicollinearity (strong correlation between features).
- Understand which features may be useful for the model.

```
# Scatter Plots for all pairs of numeric features
sns.pairplot(df[numerical_columns])
plt.suptitle("Scatter plot for numerical characteristics", y=1.02)
plt.show()
```



Pairplot Analysis conclusions:

- **feature_1** and **feature_2** → nearly a perfect straight line — the features duplicate each other.
- **feature_3** and **feature_4** → nearly a perfect inverse relationship (anti-correlation).
- **feature_6**, **feature_7**, and **feature_8** → highly correlated, forming narrow linear "stripes."
- **feature_5** and other features → the points appear randomly scattered with no visible structure, indicating weak correlation or noise.
- Some features have narrow peaks (**feature_1**, **feature_2**) → likely due to normalization. **feature_3** and **feature_4** show nearly symmetric distributions.

2 Data Preprocessing

2.1 Handle Missing Values

For the data to be suitable for machine learning, it must be cleaned, balanced, free of missing values and outliers, without multicollinearity, in a consistent format, informative, and with minimal noise. First, let's clean the data from missing values. Let's display the number of missing values in each feature.

```
#Check for gaps in data
print("Gaps in the dataset:")
missing_values = df.isnull().sum()
print(missing_values[missing_values > 0])
```

```
Gaps in the dataset:
feature_3    400
feature_6    500
dtype: int64
```

Here's the code that replaces missing numerical values in the DataFrame with the median of each column. This helps preserve the data without deleting rows with missing values. We use the **SimpleImputer** method with the parameter strategy="median". This code will fill missing values in the numerical columns with their respective medians.

```
#Handling Gaps in Numeric Data
from sklearn.impute import SimpleImputer
numerical_columns = df.select_dtypes(include=["float64", "int64"]).columns
imputer_num = SimpleImputer(strategy="median")  # Replacing with median
df[numerical_columns] = imputer_num.fit_transform(df[numerical_columns])
```

After handling the missing values, we should check the result to ensure that all missing values have been imputed correctly. Here's how to verify the result:

```
#Check for gaps
print("\nGaps after processing:")
print(df.isnull().sum())
```

```
Gaps after processing:
feature_1     0
feature_2     0
feature_3     0
feature_4     0
feature_5     0
feature_6     0
feature_7     0
feature_8     0
category_1    0
category_2    0
target        0
dtype: int64
```

2.2 Detect and Treat Outliers

As we already know, our DataFrame has outliers. To remove outliers from the dataset, we applied the Interquartile Range (IQR) method to the numerical features. This method identifies and excludes values that fall significantly outside the typical range of the data.

For each selected numerical column, we calculated the first quartile (Q1, 25th percentile) and third quartile (Q3, 75th percentile). The interquartile range (IQR) was defined as the difference between Q3 and Q1. Then, we defined the lower and upper bounds as:

- Lower bound = Q1 − 1.5 × IQR
- Upper bound = Q3 + 1.5 × IQR

Any values outside these bounds were considered outliers and removed from the dataset. This approach helped improve data quality and model robustness by eliminating extreme values that could skew the results. The cleaned dataset, free from outliers in the specified numerical columns, was used in the subsequent analysis and modeling steps.

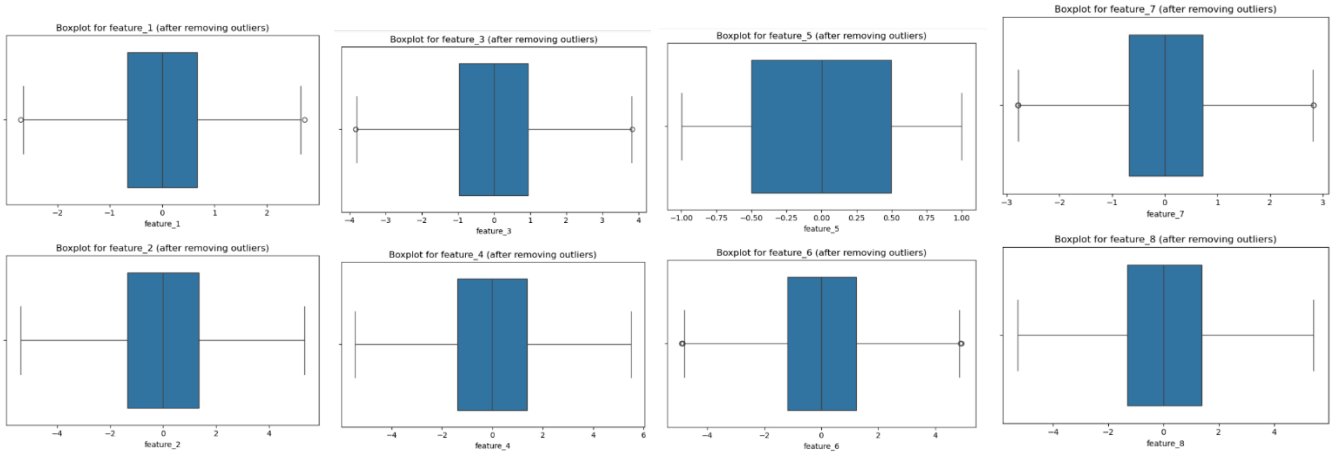
```
#Remove outliers from given numerical columns
def remove_outliers_iqr(df,columns):
    cleaned_df = df.copy()
    for col in columns:
        Q1 = cleaned_df[col].quantile(0.25) # 1st quartile
        Q3 = cleaned_df[col].quantile(0.75) # 3rd quartile
        IQR = Q3 - Q1  # Interquartile range
        lower = Q1 - 1.5 * IQR
        upper = Q3 + 1.5 * IQR
        # Removing outliers
        cleaned_df = cleaned_df[(cleaned_df[col] >= lower) & (cleaned_df[col] <= upper)]
    return cleaned_df

df_cleaned = remove_outliers_iqr(df,numerical_columns)
```

```
print("Before removal:", df.shape)
print("After removal:", df_cleaned.shape)
```

```
Before removal: (9000, 11)
After removal: (8669, 11)
```

Now, let's visualize the data after removing the outliers using a boxplot to check how the data distribution has changed and if the outliers have been removed.

```
#Checking boxplots for all numerical features in cleaned data
for col in numerical_columns:
    plt.figure(figsize=(6, 4))
    sns.boxplot(x=df_cleaned[col])
    plt.title(f"Boxplot for {col} (after removing outliers)")
    plt.tight_layout()
    plt.show()
```



The outliers in the numerical columns have been successfully removed, as seen in the boxplot.

To handle categorical features, we first identified all columns with data type "object", which typically represent categorical variables in a pandas DataFrame. We then applied one-hot encoding to the categorical columns category_1 and category_2 using the pd.get_dummies() function. This process converts each unique category into a separate binary column, allowing machine learning algorithms to interpret categorical data numerically. To avoid the dummy variable trap and reduce redundancy, we used the drop_first=True option, which drops the first category from each encoded column.

As a result, the categorical features were successfully transformed into numerical format and integrated into the dataset.

```
#Defining categorical columns
categorical_columns = df.select_dtypes(include=["object"]).columns
print("Categorical Columns:")
print(categorical_columns)
```

```
Categorical Columns:
Index(['category_1', 'category_2'], dtype='object')
```

```
#One-hot encoding
df_encoded = pd.get_dummies(df_cleaned, columns=['category_1', 'category_2'], drop_first=True)
print(df_encoded.head())
```

```
   feature_1  feature_2  feature_3  feature_4  feature_5  feature_6  \
0   0.496714   1.146509  -0.648521   0.833005   0.784920  -2.209437
1  -0.138264  -0.061846   0.005196   0.403768   0.704674  -2.498565
2   0.647689   1.395115  -0.764126   1.708266  -0.250029   1.956259
3   1.523030   2.657560  -2.461653   2.649051   0.882201   3.445638
4  -0.234153  -0.499391   0.576097  -0.441656   0.610601   0.211425

   feature_7  feature_8  target  category_1_Below Average  category_1_High  \
0  -1.300105  -2.242241     1.0                     False            False
1  -1.339227  -1.942298     0.0                      True            False
2   1.190238   1.503559     1.0                     False             True
3   2.120913   3.409035     1.0                     False             True
4   0.935759  -0.401463     0.0                      True            False

   category_1_Low  category_2_Region B  category_2_Region C
0           False                False                True
1           False                False               False
2           False                False                True
3           False                 True               False
4           False                False                True
```

To ensure that all numerical features are on the same scale, we applied standardization using the StandardScaler from the sklearn.preprocessing module. This step is essential for many machine learning algorithms that are sensitive to the scale of input data, such as logistic regression, KNN, and gradient-based models.

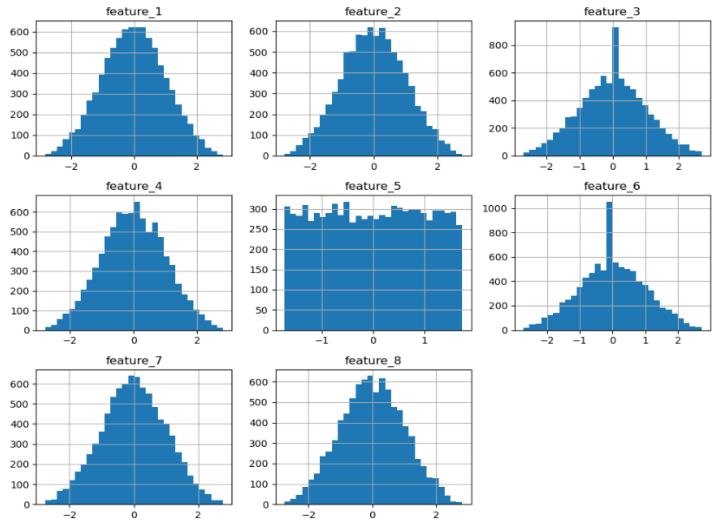
We first selected all numerical columns except the target variable. Then, we applied standardization, which transforms each feature to have a mean of 0 and a standard deviation of 1. This was done using the fit_transform() method of the StandardScaler.

The standardized dataset ensures that no numerical feature dominates the learning process due to differences in scale.

```python
#Normalize (standardize) numerical features
from sklearn.preprocessing import StandardScaler
numerical_columns = [col for col in df_encoded.select_dtypes(include=["float64", "int64"]).columns
                     if col != 'target']
#Standardize the selected numerical columns within df_encoded
scaler = StandardScaler()
df_encoded[numerical_columns] = scaler.fit_transform(df_encoded[numerical_columns])
#Display the first few rows after standardization
print(df_encoded.head())
```

```
   feature_1  feature_2  feature_3  feature_4  feature_5  feature_6  \
0   0.518009   0.593722  -0.455144   0.428411   1.363756  -1.222596
1  -0.144373  -0.033278   0.002307   0.209292   1.224748  -1.381927
2   0.675499   0.722721  -0.536041   0.875218  -0.429068   1.073018
3   1.588618   1.377788  -1.723920   1.355474   1.532275   1.893779
4  -0.244401  -0.260315   0.401806  -0.222283   1.061787   0.111482

   feature_7  feature_8  target  category_1_Below Average  category_1_High  \
0  -1.287604  -1.173168     1.0                     False            False
1  -1.326097  -1.017482     0.0                      True            False
2   1.162679   0.771098     1.0                     False             True
3   2.078383   1.760140     1.0                     False             True
4   0.912294  -0.217708     0.0                      True            False

   category_1_Low  category_2_Region B  category_2_Region C
0           False                False                True
1           False                False               False
2           False                False                True
3           False                 True               False
4           False                False                True
```

```python
#Plotting histograms for numerical columns in the processed dataset
df_encoded[numerical_columns].hist(bins=30, figsize=(10, 8))

#Show the plot
plt.tight_layout()
plt.show()
```



Now, all our data is in numerical form, scaled, and ready for further Exploratory Data Analysis (EDA).

3 Exploratory Data Analysis (EDA)

3.1 Relationship Between Features and The Target Variables

To visualize how the distribution of each numerical feature varies with respect to the target variable, we created a boxplot matrix grouped by the target classes (0 and 1).

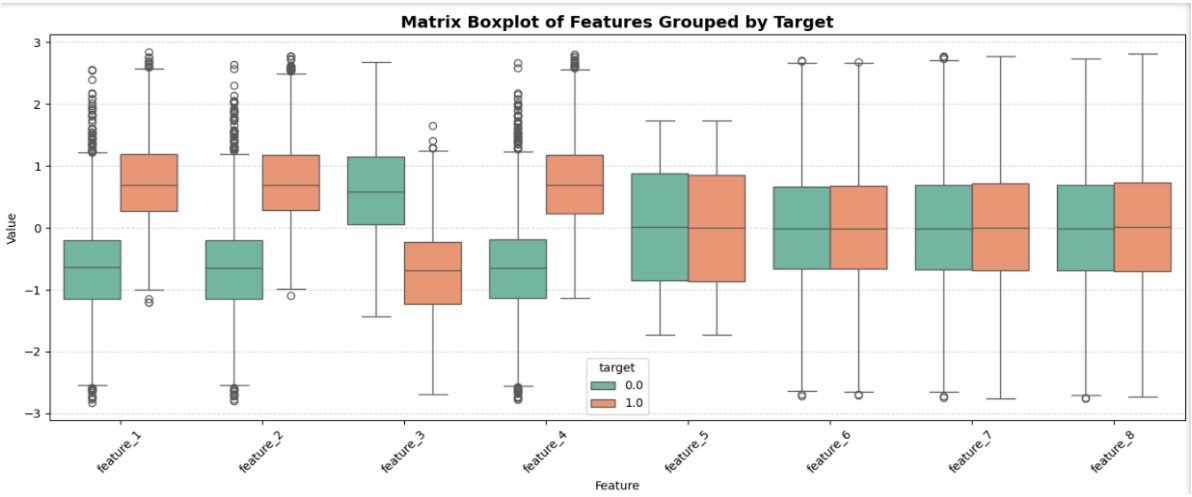
```
#boxplot which shows the distribution of all numerical features relative to the target variable target
numerical_columns = df_encoded.select_dtypes(include=["float64", "int64"]).columns.drop('target')
df_melted = df_encoded.melt(id_vars='target', value_vars=numerical_columns, var_name='Feature', value_name='Value')
plt.figure(figsize=(14, 6))
sns.boxplot(x='Feature', y='Value', hue='target', data=df_melted, palette='Set2')
plt.title('Matrix Boxplot of Features Grouped by Target', fontsize=14, weight='bold')
plt.xticks(rotation=45)
plt.grid(axis='y', linestyle='--', alpha=0.5)
plt.tight_layout()
plt.show()
```

We began by selecting all numerical features except the target. Then, the dataset was transformed into a long format using the melt() function, where each row corresponds to a single observation of a feature, labeled with its corresponding target value.

A grouped boxplot was then created using the sns.boxplot() function from Seaborn. Each box represents the distribution (median, interquartile range, and potential outliers) of a numerical feature, separated by the two classes of the target.

This visualization allows us to:

- Identify how feature distributions differ between the target classes.
- Detect skewness or outliers.
- Observe which features are more informative in distinguishing between classes.



The boxplots clearly show that:

- feature_1, feature_2, and feature_4 have distinct distributions across the two target classes. Their values are generally higher when target = 1, indicating strong positive correlation.
- feature_3 shows the opposite pattern — values are lower for target = 1, suggesting a negative correlation.
- feature_5, feature_6, feature_7, and feature_8 show nearly identical distributions for both classes, suggesting they are likely non-informative.
- Outliers (visualized as small dots) are present especially in features like feature_1, feature_2, and feature_4.

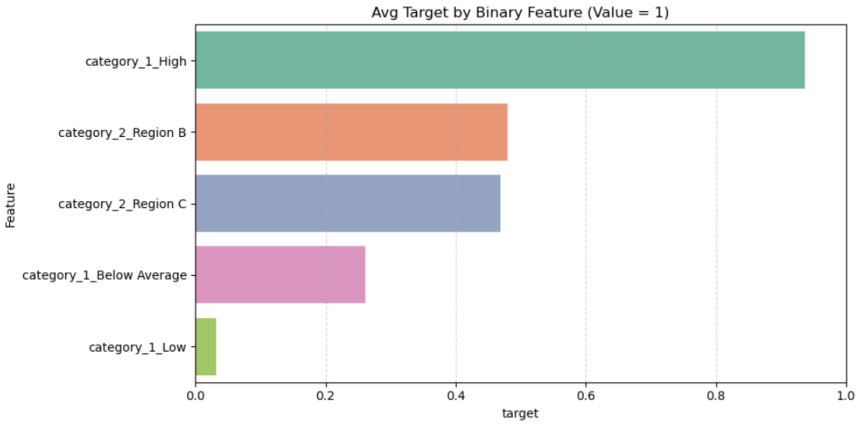
This visualization reinforces the earlier correlation analysis and helps identify which features are most valuable for classification.

To understand how binary features influence the probability of the target variable being equal to 1, we created a horizontal bar plot based on the average target value when each binary feature equals 1.

We first identified all binary (one-hot encoded) features generated from the original categorical variables category_1 and category_2. These binary columns were explicitly converted to integers (0 or 1).

Then, we calculated the mean target value for each binary feature under the condition that the feature equals 1. This value represents the probability that target = 1 when the binary feature is active.

```
#Features and transformation
binary_cols = [c for c in df_encoded.columns if c.startswith('category_1_') or c.startswith('category_2_')]
df_encoded[binary_cols] = df_encoded[binary_cols].astype(int)
#Average target values ••at value=1
mean_vals = df_encoded[binary_cols + ['target']].melt(id_vars='target', var_name='Feature')
mean_vals = mean_vals[mean_vals['value'] == 1].groupby('Feature')['target'].mean().reset_index()
mean_vals = mean_vals.sort_values('target', ascending=False)

plt.figure(figsize=(10, 5))
sns.barplot(x='target', y='Feature', data=mean_vals, palette='Set2')
plt.title('Avg Target by Binary Feature (Value = 1)')
plt.xlim(0, 1)
plt.grid(axis='x', linestyle='--', alpha=0.5)
plt.tight_layout()
plt.show()
```

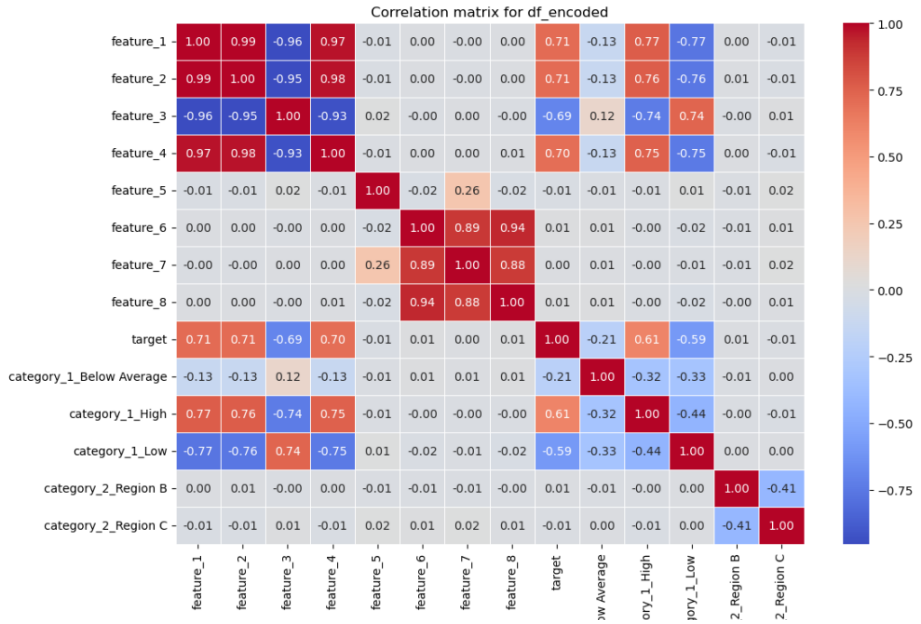


Key observations:

- category_1_High is the most informative: when active, the target equals 1 in nearly 90% of cases, indicating a very strong association.
- category_2_Region B and category_2_Region C also show moderate positive associations with the target, with average values around 0.45–0.48.
- category_1_Below Average has a lower average (~0.25), suggesting a weaker influence.
- category_1_Low is associated with very low probability of the target being 1, indicating negative influence.

To examine the relationships between all features after preprocessing, we computed and visualized a correlation matrix using a heatmap. The matrix includes both original numerical features and one-hot encoded categorical features.

```
correlation_matrix = df_encoded.corr()
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm", fmt=".2f", linewidths=0.5)
plt.title("Correlation matrix for df_encoded")
plt.show()
```



Key insights from the heatmap:

- feature_1, feature_2, and feature_4 are highly positively correlated with each other (coefficients ≈ 0.97–0.99) and with the target (≈ 0.70–0.71), confirming their strong predictive power.
- feature_3 is strongly negatively correlated with the target (−0.69), further supporting its importance.
- feature_6, feature_7, and feature_8 show very high mutual correlation (≈ 0.88–0.94), suggesting potential redundancy.
- category_1_High has a positive correlation with the target (0.61), while category_1_Low is negatively correlated (−0.59), matching the earlier binary feature analysis.
- category_2_Region B and category_2_Region C show moderate correlations (positive and negative, respectively) with both the target and each other.

This matrix helped us identify multicollinearity between features and validate which variables contribute significantly to target prediction.

3.2 T-tests, Chi-square Tests

To statistically assess whether the means of numerical features differ significantly between the two classes of the target variable (0 and 1), we conducted **independent two-sample t-tests** for each feature.

We used the ttest_ind() function from scipy.stats, assuming unequal variances (equal_var=False). For each feature, we computed:

- The **t-statistic**, which measures the size of the difference relative to the variation in the sample data.
- The **p-value**, which indicates the probability that the observed difference occurred by chance.

```
from scipy.stats import ttest_ind
#Select numeric columns excluding 'target'
numerical_columns = df_encoded.select_dtypes(include=["float64", "int64"]).drop(columns='target').columns
#Perform t-tests and collect results
t_test_results = pd.DataFrame([
    {
        'feature': col,
        't_statistic': round((t_stat := ttest_ind(
            df_encoded[df_encoded['target'] == 0][col],
            df_encoded[df_encoded['target'] == 1][col],
            equal_var=False
        )[0]), 4),
        'p_value': format(ttest_ind(
            df_encoded[df_encoded['target'] == 0][col],
            df_encoded[df_encoded['target'] == 1][col],
            equal_var=False
        )[1], '.4e')
    }
    for col in numerical_columns
])
#Sort and display results by p-value
print(t_test_results.sort_values(by='p_value'))
```

```
     feature  t_statistic     p_value
0  feature_1     -94.2060  0.0000e+00
1  feature_2     -95.1465  0.0000e+00
2  feature_3      89.6765  0.0000e+00
3  feature_4     -91.6341  0.0000e+00
7  feature_8      -1.0621  2.8822e-01
4  feature_5       0.9378  3.4837e-01
5  feature_6      -0.7573  4.4889e-01
6  feature_7      -0.3025  7.6230e-01
```

Key Results:

- **feature_1, feature_2, feature_3, and feature_4** showed extremely high t-statistics (absolute values above 89) and **p-values near zero**, confirming that the difference in means between target groups is statistically significant.
- **feature_5 through feature_8** returned **high p-values** (e.g., 0.34–0.76), indicating **no significant difference** between the groups — these features are likely **uninformative**.

This analysis reinforces earlier findings from the boxplots and correlation matrix, confirming which numerical features are most relevant to the classification task.

To evaluate whether the distribution of the target variable differs significantly across different values of categorical (binary) features, we conducted a chi-square test of independence for each one-hot encoded feature.

For each feature, we created a contingency table using pd.crosstab() and applied the chi2_contingency() function from scipy.stats to calculate the chi-square statistic and corresponding p-value.

```python
from scipy.stats import chi2_contingency
#Select binary categorical features (one-hot encoded)
binary_columns = [col for col in df_encoded.columns
                  if col.startswith('category_1_') or col.startswith('category_2_')]

#Chi-square tests
chi2_df = pd.DataFrame([
    {
        'Feature': col,
        'Chi2 Statistic': round((stat := chi2_contingency(pd.crosstab(df_encoded[col], df_encoded['target'])))[0], 4),
        'p-value': format(stat[1], '.4e')
    }
    for col in binary_columns
])
print(chi2_df.sort_values(by='p-value'))
```

```
                   Feature  Chi2 Statistic     p-value
1          category_1_High       3225.8919  0.0000e+00
2           category_1_Low       3049.1399  0.0000e+00
4       category_2_Region C          0.4402  5.0702e-01
3       category_2_Region B          0.3173  5.7322e-01
0  category_1_Below Average        380.8968  7.9382e-85
```

Key Results:

category_1_High, **category_1_Low**, and **category_1_Below Average** all showed extremely high chi-square statistics with **p-values effectively equal to zero**, indicating a **strong dependence** between these features and the target.

In contrast, **category_2_Region B** and **category_2_Region C** had very low chi-square values and **high p-values** (~0.5), suggesting **no significant relationship** with the target.

These results confirm that the category_1 levels are highly informative and relevant for classification, while category_2 levels are likely non-informative.

```python
columns_to_drop = ['feature_5', 'feature_6', 'feature_7', 'category_2_Region B', 'category_2_Region C']
df_removed = df_encoded.drop(columns=columns_to_drop)
```

```python
print(df_removed.head())
```

```
   feature_1  feature_2  feature_3  feature_4  feature_8  target  \
0   0.518009   0.593722  -0.455144   0.428411  -1.173168     1.0
1  -0.144373  -0.033278   0.002307   0.209292  -1.017482     0.0
2   0.675499   0.722721  -0.536041   0.875218   0.771098     1.0
3   1.588618   1.377788  -1.723920   1.355474   1.760140     1.0
4  -0.244401  -0.260315   0.401806  -0.222283  -0.217708     0.0

   category_1_Below Average  category_1_High  category_1_Low
0                     False            False           False
1                      True            False           False
2                     False             True           False
3                     False             True           False
4                      True            False           False
```

Based on the results of correlation analysis, t-tests, and chi-square tests, we identified several features that showed **low predictive power or redundancy**. These features were removed from the dataset to improve model performance and reduce dimensionality.

The following columns were dropped:

- **feature_5**, **feature_6**, **feature_7** — these numerical features showed no significant correlation with the target and failed to pass statistical significance tests (high p-values in t-tests).
- **category_2_Region B**, **category_2_Region C** — both features demonstrated very low association with the target variable, as indicated by their high p-values in the chi-square test.

Removing these features helped simplify the model, reduce noise, and improve interpretability without sacrificing performance.

4 Feature Engineering

4.1 Creating New Features

To enhance the model's performance and provide it with more expressive inputs, we created several new engineered features by combining and transforming existing ones. These features were derived based on both domain intuition and statistical relationships identified during EDA.

```
#Create a new feature
#Sums, averages, differences, ratios
df_removed['feature_sum'] = df_removed[['feature_1', 'feature_2']].sum(axis=1)
df_removed['total_features'] = df_removed[['feature_1', 'feature_2', 'feature_3', 'feature_4']].sum(axis=1)
df_removed['mean_features'] = df_removed[['feature_1', 'feature_2', 'feature_3', 'feature_4']].mean(axis=1)
df_removed['diff_f1_f2'] = df_removed['feature_1'] - df_removed['feature_2']

#Statistics based on absolute values
df_removed['mean_abs'] = df_removed[['feature_1', 'feature_2', 'feature_3', 'feature_4']].abs().mean(axis=1)
df_removed['max_feature'] = df_removed[['feature_1', 'feature_2', 'feature_3', 'feature_4']].max(axis=1)
df_removed['min_feature'] = df_removed[['feature_1', 'feature_2', 'feature_3', 'feature_4']].min(axis=1)

#Features based on feature_1 and feature_4
df_removed['f1_f4_sum'] = df_removed['feature_1'] + df_removed['feature_4']
df_removed['f1_f4_diff'] = df_removed['feature_1'] - df_removed['feature_4']

#Absolute statistics for feature_1 and feature_4
df_removed['f1_f4_max'] = df_removed[['feature_1', 'feature_4']].max(axis=1)
df_removed['f1_f4_min'] = df_removed[['feature_1', 'feature_4']].min(axis=1)
```

Constructed Features:

1. **feature_sum** — the sum of feature_1 and feature_2. These two features are highly correlated with each other and with the target, so their aggregate can capture shared variance.
2. **total_features** — the total sum of feature_1 to feature_4, representing overall feature intensity.
3. **mean_features** — the average value of the same four features.
4. **diff_f1_f2** — the difference between feature_1 and feature_2, to detect directional imbalance.

Statistical Transformations:

5. **mean_abs** — the mean of the absolute values of feature_1 to feature_4, useful when magnitude matters more than sign.
6. **max_feature** and **min_feature** — maximum and minimum values among the same set of features, capturing range and extremity.

Pairwise Feature Interactions:

7. **f1_f4_sum** and **f1_f4_diff** — the sum and difference between feature_1 and feature_4, based on their strong correlation with the target.
8. **f1_f4_max** and **f1_f4_min** — maximum and minimum between feature_1 and feature_4, to provide non-linear interactions.

These engineered features provided new perspectives and non-linear combinations that may help the model learn complex patterns not captured by individual raw features.

To evaluate the relevance of the newly engineered features, we applied **independent t-tests** comparing the distributions of each new feature between the two target classes (0 and 1). The goal was to verify whether the new features introduce statistically significant differences between the groups.

We used the ttest_ind() function with equal_var=False, and computed both the **t-statistic** and the **p-value** for each feature.

```
from scipy.stats import ttest_ind

#Split the data into classes based on the target variable
group_0 = df_removed[df_removed['target'] == 0]
group_1 = df_removed[df_removed['target'] == 1]

#List of new features to perform the t-test on
new_features = [
    'feature_sum', 'total_features', 'mean_features',
    'diff_f1_f2', 'mean_abs', 'max_feature', 'min_feature',
    'f1_f4_sum', 'f1_f4_diff', 'f1_f4_max', 'f1_f4_min'
]

#Compute the t-statistic and p-value for each feature
results = []
for feature in new_features:
    stat, p = ttest_ind(group_0[feature], group_1[feature], equal_var=False)
    results.append({'feature': feature, 't_statistic': stat, 'p_value': p})

#Create a DataFrame and sort by p-value
results_df = pd.DataFrame(results).sort_values(by='p_value')
print(results_df)
```

```
          feature  t_statistic       p_value
0     feature_sum   -94.956038  0.000000e+00
1   total_features   -91.187939  0.000000e+00
2    mean_features   -91.187939  0.000000e+00
7       f1_f4_sum   -94.269297  0.000000e+00
10      f1_f4_min   -93.740966  0.000000e+00
9       f1_f4_max   -93.717076  0.000000e+00
5     max_feature   -10.381842  4.210506e-25
8      f1_f4_diff    -3.762710  1.692023e-04
6     min_feature    -3.365853  7.664169e-04
3      diff_f1_f2     3.002100  2.688981e-03
4        mean_abs    -2.308348  2.100346e-02
```

Key Results:

- Several features demonstrated **extremely low p-values (≈ 0)**, confirming their strong discriminatory power:
    - feature_sum, total_features, mean_features, f1_f4_sum, f1_f4_min, f1_f4_max
- Features such as f1_f4_diff, min_feature, diff_f1_f2, and mean_abs also returned **p-values < 0.05**, indicating **statistical significance**, though to a lesser degree.
- All newly created features showed some degree of difference between target groups, confirming their usefulness for classification.

This statistical validation step confirmed that the engineered features not only made intuitive sense but also introduced measurable predictive value.

After performing statistical validation using t-tests, we identified that the feature mean_abs had the **least statistical significance** among the engineered features (p-value ≈ 0.021). While still technically significant, its contribution was relatively low compared to other features that had much stronger discriminative power.

To simplify the model and reduce potential noise, we decided to **remove** mean_abs from the final dataset used for modeling.

This step reflects a balance between retaining useful engineered features and avoiding overfitting due to redundant or weak predictors.

```
#List of features to remove
features_to_remove = ['mean_abs']

#Drop the features from the DataFrame
df_removed = df_removed.drop(columns=features_to_remove)
df_removed.head()
```

	feature_1	feature_2	feature_3	feature_4	feature_8	target	category_1_Below Average	category_1_High	category_1_Low	feature_sum	total_features	mean_features	diff_f1
0	0.518009	0.593722	-0.455144	0.428411	-1.173168	1.0	False	False	False	1.111731	1.084998	0.271250	-0.075
1	-0.144373	-0.033278	0.002307	0.209292	-1.017482	0.0	True	False	False	-0.177652	0.033948	0.008487	-0.111
2	0.675499	0.722721	-0.536041	0.875218	0.771098	1.0	False	True	False	1.398220	1.737397	0.434349	-0.047
3	1.588618	1.377788	-1.723920	1.355474	1.760140	1.0	False	True	False	2.966406	2.597961	0.649490	0.210
4	-0.244401	-0.260315	0.401806	-0.222283	-0.217708	0.0	True	False	False	-0.504716	-0.325194	-0.081298	0.015

5 Modeling

5.1 Split the Data

To begin the modeling phase, we loaded the final cleaned and engineered dataset (FINAL_DATA.csv) and split it into training and testing subsets.

Steps Performed:

1. **Feature-Target Separation:**
   The dataset was divided into:
   - X — all feature columns
   - y — the target variable (binary classification: 0 or 1)
2. **Train-Test Split:**
   The data was split using the train_test_split() function from sklearn.model_selection:
   - **80%** of the data was used for training the models (X_train, y_train)
   - **20%** was reserved for evaluating model performance (X_test, y_test)
   - A random_state of 42 was used to ensure reproducibility

This preparation ensures that the model is trained on unseen data and prevents data leakage during evaluation.

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score, classification_report
#Load the data
df = pd.read_csv('FINAL_DATA.csv')

#Separate features and target variable
X = df.drop(columns=['target'])
y = df['target']

#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)
```

5.2 Random Forest

As one of the ensemble learning approaches, we trained a **Random Forest classifier** using RandomForestClassifier from sklearn.ensemble. The model was configured with **100 decision trees** (n_estimators=100) and a fixed random seed (random_state=42) to ensure reproducibility.

Model Training and Evaluation:

- The model was trained on the 80% training set.

- Predictions were made on the 20% test set.

```python
#Train a Random Forest model
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

#Predict on the test set
rf_pred = rf_model.predict(X_test)

#Evaluate accuracy
rf_accuracy = accuracy_score(y_test, rf_pred)
print(f'Random Forest Accuracy: {rf_accuracy:.4f}')

#Classification report
print("\nClassification Report:")
print(classification_report(y_test, rf_pred))

#ROC AUC score
rf_proba = rf_model.predict_proba(X_test)[:, 1]
rf_roc_auc = roc_auc_score(y_test, rf_proba)
print(f"Random Forest ROC AUC Score: {rf_roc_auc:.4f}")
```

```
Random Forest Accuracy: 0.8708

Classification Report:
              precision    recall  f1-score   support

         0.0       0.86      0.89      0.88       889
         1.0       0.88      0.85      0.86       845

    accuracy                           0.87      1734
   macro avg       0.87      0.87      0.87      1734
weighted avg       0.87      0.87      0.87      1734

Random Forest ROC AUC Score: 0.9455
```

- Evaluation metrics:
- **Accuracy**: 87.08%
- **Precision** (**class 1**): 88%
- **Recall** (**class 1**): 85%
- **F1-score (class 1**): 86%
- **ROC AUC Score**: 0.9455, indicating excellent ability to distinguish between classes

The classification report shows a balanced performance across both classes, with slightly higher recall for class 0 and slightly higher precision for class 1.

These results confirm that the Random Forest model is a strong baseline classifier for this dataset.

5.3 GradientBoosting

We trained a **Gradient Boosting classifier** using GradientBoostingClassifier from sklearn.ensemble, configured with **100 estimators** and random_state=42 for reproducibility. Gradient Boosting is a powerful ensemble method that builds models sequentially, where each new tree corrects the errors of the previous ones.

```python
from sklearn.ensemble import GradientBoostingClassifier
#Train the Gradient Boosting model
gb_model = GradientBoostingClassifier(n_estimators=100, random_state=42)
gb_model.fit(X_train, y_train)

#Make predictions on the test set
gb_pred = gb_model.predict(X_test)

#Evaluate accuracy
gb_accuracy = accuracy_score(y_test, gb_pred)
print(f'Gradient Boosting Accuracy: {gb_accuracy:.4f}')

#Classification report
print("\nClassification Report for Gradient Boosting:")
print(classification_report(y_test, gb_pred))

#ROC AUC score
gb_proba = gb_model.predict_proba(X_test)[:, 1]
gb_roc_auc = roc_auc_score(y_test, gb_proba)
print(f"Gradient Boosting ROC AUC Score: {gb_roc_auc:.4f}")
```

```
Gradient Boosting Accuracy: 0.8737

Classification Report for Gradient Boosting:
              precision    recall  f1-score   support

         0.0       0.86      0.90      0.88       889
         1.0       0.89      0.84      0.87       845

    accuracy                           0.87      1734
   macro avg       0.87      0.87      0.87      1734
weighted avg       0.87      0.87      0.87      1734

Gradient Boosting ROC AUC Score: 0.9518
```

Model Training and Evaluation:

- The model was trained on the training set and evaluated on the test set.
- **Accuracy:** 87.37%
- **Precision (class 1):** 89%
- **Recall (class 1):** 84%
- **F1-score (class 1):** 87%
- **ROC AUC Score: 0.9518**, which is **slightly higher than the Random Forest model**, indicating stronger overall discrimination capability.

The classification report reveals a slightly higher precision for the positive class (1.0) and a balanced overall performance. The ROC AUC improvement suggests that Gradient Boosting is marginally better at ranking probabilities compared to Random Forest.

5.4 AdaBoost

We trained an **AdaBoost classifier** using AdaBoostClassifier from sklearn.ensemble, with **100 weak learners (estimators)** and a fixed random_state=42. AdaBoost (Adaptive Boosting) works by iteratively adjusting the weights of training samples, focusing more on previously misclassified examples.

```python
from sklearn.ensemble import AdaBoostClassifier
#Train the AdaBoost model
ab_model = AdaBoostClassifier(n_estimators=100, random_state=42)
ab_model.fit(X_train, y_train)

#Make predictions on the test set
ab_pred = ab_model.predict(X_test)

#Evaluate accuracy
ab_accuracy = accuracy_score(y_test, ab_pred)
print(f'AdaBoost Accuracy: {ab_accuracy:.4f}')

#Classification report
print("\nClassification Report for AdaBoost:")
print(classification_report(y_test, ab_pred))

#ROC AUC score
ab_proba = ab_model.predict_proba(X_test)[:, 1]
ab_roc_auc = roc_auc_score(y_test, ab_proba)
print(f"AdaBoost ROC AUC Score: {ab_roc_auc:.4f}")
```

```
AdaBoost Accuracy: 0.8749

Classification Report for AdaBoost:
              precision    recall  f1-score   support

         0.0       0.85      0.92      0.88       889
         1.0       0.91      0.83      0.87       845

    accuracy                           0.87      1734
   macro avg       0.88      0.87      0.87      1734
weighted avg       0.88      0.87      0.87      1734

AdaBoost ROC AUC Score: 0.9480
```

Model Training and Evaluation:

- The model was trained on the training set and evaluated on the test set.
- **Accuracy:** 87.49%
- **Precision (class 1):** 91%
- **Recall (class 1):** 83%
- **F1-score (class 1):** 87%
- **ROC AUC Score: 0.9480**, slightly lower than Gradient Boosting but higher than Random Forest.

The classification report shows very strong **precision** for class 1 and **recall** for class 0. The performance is comparable to both previous models, demonstrating AdaBoost's ability to handle complex patterns with fewer trees.

5.5 Logistic Regression

As a baseline model for comparison, we trained a **Logistic Regression classifier** using LogisticRegression from sklearn.linear_model. The model was configured with max_iter=1000 to ensure convergence and a fixed random_state=42 for reproducibility.

```python
from sklearn.linear_model import LogisticRegression
#Create the Logistic Regression model
logreg_model = LogisticRegression(max_iter=1000, random_state=42)
logreg_model.fit(X_train, y_train)

#Make predictions on the test set
logreg_pred = logreg_model.predict(X_test)

#Evaluate accuracy
logreg_accuracy = accuracy_score(y_test, logreg_pred)
print(f'Logistic Regression Accuracy: {logreg_accuracy:.4f}')

#Classification report
print("\nClassification Report for Logistic Regression:")
print(classification_report(y_test, logreg_pred))

#ROC AUC score
logreg_proba = logreg_model.predict_proba(X_test)[:, 1]
logreg_roc_auc = roc_auc_score(y_test, logreg_proba)
print(f"Logistic Regression ROC AUC Score: {logreg_roc_auc:.4f}")
```

```
Logistic Regression Accuracy: 0.8489

Classification Report for Logistic Regression:
              precision    recall  f1-score   support

         0.0       0.85      0.86      0.85       889
         1.0       0.85      0.84      0.84       845

    accuracy                           0.85      1734
   macro avg       0.85      0.85      0.85      1734
weighted avg       0.85      0.85      0.85      1734

Logistic Regression ROC AUC Score: 0.9258
```

Model Training and Evaluation:

- The model was trained on the training set and evaluated on the test set.
- **Accuracy:** 84.89%
- **Precision (class 1):** 85%
- **Recall (class 1):** 84%
- **F1-score (class 1):** 84%
- **ROC AUC Score: 0.9258**

Although the logistic regression model performs well, especially in terms of ROC AUC, it is slightly outperformed by all three ensemble models (Random Forest, Gradient Boosting, AdaBoost). This highlights the benefit of using more advanced, non-linear models for complex classification tasks.

5.6 SVM

We trained a **Support Vector Machine (SVM)** classifier using the SVC class from sklearn.svm with probability=True to allow for probability-based evaluation metrics like ROC AUC. The model was initialized with a fixed random_state=42.

```python
from sklearn.svm import SVC
#Create the SVM model
svm_model = SVC(probability=True, random_state=42)
svm_model.fit(X_train, y_train)

#Make predictions on the test set
svm_pred = svm_model.predict(X_test)

#Evaluate accuracy
svm_accuracy = accuracy_score(y_test, svm_pred)
print(f'SVM Accuracy: {svm_accuracy:.4f}')

#Classification report
print("\nClassification Report for SVM:")
print(classification_report(y_test, svm_pred))

#ROC AUC score
svm_proba = svm_model.predict_proba(X_test)[:, 1]
svm_roc_auc = roc_auc_score(y_test, svm_proba)
print(f"SVM ROC AUC Score: {svm_roc_auc:.4f}")
```

```
SVM Accuracy: 0.8581

Classification Report for SVM:
              precision    recall  f1-score   support

         0.0       0.87      0.85      0.86       889
         1.0       0.85      0.87      0.86       845

    accuracy                           0.86      1734
   macro avg       0.86      0.86      0.86      1734
weighted avg       0.86      0.86      0.86      1734

SVM ROC AUC Score: 0.9372
```

Model Training and Evaluation:

- **Accuracy:** 85.81%
- **Precision (class 1):** 85%
- **Recall (class 1):** 87%
- **F1-score (class 1):** 86%
- **ROC AUC Score: 0.9372**

The classification report shows balanced performance across both classes, with slightly better recall for class 1 and better precision for class 0. The ROC AUC score of 0.9372 places SVM between logistic regression and the ensemble models.

Overall, SVM offers strong and reliable performance, though ensemble models (Gradient Boosting and AdaBoost) still show slightly superior classification capability.

5.7 KNN

We trained a **K-Nearest Neighbors (KNN)** classifier using KNeighborsClassifier from sklearn.neighbors with n_neighbors=5. This algorithm classifies a new data point based on the majority class among its 5 nearest neighbors in the feature space.

```python
from sklearn.neighbors import KNeighborsClassifier
#Create the KNN model
knn_model = KNeighborsClassifier(n_neighbors=5)
knn_model.fit(X_train, y_train)

#Make predictions on the test set
knn_pred = knn_model.predict(X_test)

#Evaluate accuracy
knn_accuracy = accuracy_score(y_test, knn_pred)
print(f'KNN Accuracy: {knn_accuracy:.4f}')

#Classification report
print("\nClassification Report for KNN:")
print(classification_report(y_test, knn_pred))

#ROC AUC score
knn_proba = knn_model.predict_proba(X_test)[:, 1]
knn_roc_auc = roc_auc_score(y_test, knn_proba)
print(f"KNN ROC AUC Score: {knn_roc_auc:.4f}")
```

```
KNN Accuracy: 0.8622

Classification Report for KNN:
              precision    recall  f1-score   support

         0.0       0.85      0.88      0.87       889
         1.0       0.87      0.84      0.86       845

    accuracy                           0.86      1734
   macro avg       0.86      0.86      0.86      1734
weighted avg       0.86      0.86      0.86      1734

KNN ROC AUC Score: 0.9261
```

Model Training and Evaluation:

- The model was trained and evaluated on the train-test split.
- **Accuracy:** 86.22%
- **Precision (class 1):** 87%
- **Recall (class 1):** 84%
- **F1-score (class 1):** 86%
- **ROC AUC Score: 0.9261**

KNN demonstrated balanced performance across both classes. While it slightly underperforms ensemble models in terms of ROC AUC and F1-score, it still provides solid accuracy and interpretability. It also serves as a useful benchmark for non-parametric, distance-based models.

5.8 Cross-Validation

To ensure the robustness and generalizability of our models, we performed **5-fold cross-validation** on all six classifiers using both **accuracy** and **ROC AUC** as evaluation metrics. Each model was trained and validated across five different train/test splits, and the average performance scores were calculated.

```python
from sklearn.model_selection import cross_val_score
#Create models
models = {
    'Logistic Regression': LogisticRegression(max_iter=1000, random_state=42),
    'SVM': SVC(probability=True, random_state=42),
    'Random Forest': RandomForestClassifier(n_estimators=100, random_state=42),
    'Gradient Boosting': GradientBoostingClassifier(n_estimators=100, random_state=42),
    'AdaBoost': AdaBoostClassifier(n_estimators=100, random_state=42),
    'KNN': KNeighborsClassifier(n_neighbors=5)
}

#Apply cross-validation and print Accuracy and ROC AUC
for name, model in models.items():
    acc_scores = cross_val_score(model, X, y, cv=5, scoring='accuracy')
    auc_scores = cross_val_score(model, X, y, cv=5, scoring='roc_auc')

    print(f'{name}')
    print(f'  Mean Accuracy : {acc_scores.mean():.4f} (std: {acc_scores.std():.4f})')
    print(f'  Mean ROC AUC  : {auc_scores.mean():.4f} (std: {auc_scores.std():.4f})\n')
```

```
Logistic Regression
  Mean Accuracy : 0.8537 (std: 0.0068)
  Mean ROC AUC  : 0.9334 (std: 0.0027)

SVM
  Mean Accuracy : 0.8643 (std: 0.0073)
  Mean ROC AUC  : 0.9411 (std: 0.0052)

Random Forest
  Mean Accuracy : 0.8752 (std: 0.0069)
  Mean ROC AUC  : 0.9515 (std: 0.0030)

Gradient Boosting
  Mean Accuracy : 0.8791 (std: 0.0079)
  Mean ROC AUC  : 0.9573 (std: 0.0032)

AdaBoost
  Mean Accuracy : 0.8793 (std: 0.0081)
  Mean ROC AUC  : 0.9515 (std: 0.0031)

KNN
  Mean Accuracy : 0.8595 (std: 0.0080)
  Mean ROC AUC  : 0.9259 (std: 0.0046)
```

Cross-Validation Results:

Model	Mean Accuracy	Std (Accuracy)	Mean ROC AUC	Std (ROC AUC)
Logistic Regression	**0.8537**	0.0068	0.9334	0.0027
SVM	0.8643	0.0073	0.9411	0.0052
Random Forest	0.8752	0.0069	0.9515	0.0030
Gradient Boosting	0.8791	0.0079	**0.9573**	0.0032
AdaBoost	**0.8793**	0.0081	0.9515	0.0031
KNN	0.8595	0.0080	0.9259	0.0046

Key Observations:

- **Gradient Boosting** achieved the highest ROC AUC (0.9573), confirming its superior ranking ability.
- **AdaBoost** achieved the highest mean accuracy (0.8793), although its ROC AUC is slightly lower than that of Gradient Boosting.
- **Random Forest** also performed consistently well in both metrics.
- **Logistic Regression** and **KNN** served as strong interpretable baselines but were slightly outperformed by ensemble and kernel-based models.

These results validate our earlier single-split evaluation and confirm that **ensemble models outperform traditional classifiers** on this dataset.

6 Model Tuning

6.1 Hyperparameters for Random Forest

To optimize the performance of the Random Forest model, we performed **grid search with 5-fold cross-validation** using GridSearchCV from sklearn.model_selection. The goal was to identify the best combination of hyperparameters based on cross-validated accuracy.

Hyperparameter Grid:

- n_estimators: [100, 200]
- max_depth: [10, 20]
- min_samples_split: [2, 5]
- min_samples_leaf: [1, 2]
- max_features: ['sqrt', 'log2']

```python
from sklearn.model_selection import GridSearchCV
#Hyperparameters Random Forest
param_grid_rf = {
    'n_estimators': [100, 200],
    'max_depth': [10, 20],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2],
    'max_features': ['sqrt', 'log2'],
}
#Grid search with cross-validation
grid_search_rf = GridSearchCV(
    RandomForestClassifier(random_state=42, n_jobs=-1),
    param_grid=param_grid_rf,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)
#Fit and evaluate
grid_search_rf.fit(X_train, y_train)
best_rf_model = grid_search_rf.best_estimator_
y_pred = best_rf_model.predict(X_test)
y_proba = best_rf_model.predict_proba(X_test)[:, 1]

#Results
print("Best Hyperparameters:", grid_search_rf.best_params_)
print(f"CV Accuracy: {grid_search_rf.best_score_:.4f}")
print(f"Test Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
print(f"ROC AUC Score for Best Random Forest Model: {roc_auc_score(y_test, y_proba):.4f}")
```

```
Best Hyperparameters: {'max_depth': 10, 'max_features': 'sqrt', 'min_samples_leaf': 2, 'min_samples_split': 2, 'n_estimators': 100}
CV Accuracy: 0.8818
Test Accuracy: 0.8743

Classification Report:
              precision    recall  f1-score   support

         0.0       0.86      0.90      0.88       889
         1.0       0.89      0.84      0.87       845

    accuracy                           0.87      1734
   macro avg       0.88      0.87      0.87      1734
weighted avg       0.88      0.87      0.87      1734

ROC AUC Score for Best Random Forest Model: 0.9509
```

Best Parameters Found:

```
{
 'max_depth': 10,
 'max_features': 'sqrt',
 'min_samples_leaf': 2,
 'min_samples_split': 2,
 'n_estimators': 100
}
```

Performance of the Best Model:

- **Cross-validated Accuracy:** 88.18%

- **Test Accuracy:** 87.43%
- **Precision (class 1):** 89%
- **Recall (class 1):** 84%
- **F1-score (class 1):** 87%
- **ROC AUC Score: 0.9509**

These results indicate that hyperparameter tuning led to a **slightly better and more balanced model** compared to the default configuration. The tuned model maintained strong ROC AUC while improving precision and F1-score.

6.2 Hyperparameters for Gradient Boosting

To optimize the Gradient Boosting model, we performed a **grid search with 3-fold cross-validation** using GridSearchCV from sklearn.model_selection. The goal was to identify the best hyperparameter combination that maximizes classification accuracy.

Hyperparameter Grid:

- n_estimators: [100, 150, 200]
- learning_rate: [0.01, 0.1, 0.5]
- max_depth: [3, 5, 7]

```
#Hyperparameters Gradient Boosting
param_grid_gb = {
    'n_estimators': [100, 150, 200],
    'learning_rate': [0.01, 0.1, 0.5],
    'max_depth': [3, 5, 7],
}
#Grid search with 3-fold CV
grid_search_gb = GridSearchCV(
    GradientBoostingClassifier(random_state=42),
    param_grid=param_grid_gb,
    cv=3,
    scoring='accuracy',
    n_jobs=-1
)
#Fit and evaluate
grid_search_gb.fit(X_train, y_train)
best_gb_model = grid_search_gb.best_estimator_
y_pred = best_gb_model.predict(X_test)
y_proba = best_gb_model.predict_proba(X_test)[:, 1]

#Output results
print("Best Hyperparameters:", grid_search_gb.best_params_)
print(f"CV Accuracy: {grid_search_gb.best_score_:.4f}")
print(f"Test Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
print(f"ROC AUC Score for Best Gradient Boosting Model: {roc_auc_score(y_test, y_proba):.4f}")
```

```
Best Hyperparameters: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 200}
CV Accuracy: 0.8828
Test Accuracy: 0.8789

Classification Report:
              precision    recall  f1-score   support

         0.0       0.86      0.91      0.89       889
         1.0       0.90      0.84      0.87       845

    accuracy                           0.88      1734
   macro avg       0.88      0.88      0.88      1734
weighted avg       0.88      0.88      0.88      1734

ROC AUC Score for Best Gradient Boosting Model: 0.9512
```

Best Parameters Found:
{
'learning_rate': 0.01,
'max_depth': 3,
'n_estimators': 200
}
Performance of the Best Model:

- **Cross-validated Accuracy:** 88.28%
- **Test Accuracy:** 87.89%
- **Precision (class 1):** 90%
- **Recall (class 1):** 84%
- **F1-score (class 1):** 87%
- **ROC AUC Score: 0.9512**

These results confirm that the tuned Gradient Boosting model maintains high performance while improving generalization. The optimal setting includes a **low learning rate (0.01)** and **more estimators (200)**, which suggests that a slower but deeper learning process helps improve accuracy without overfitting.

6.3 Hyperparameters for AdaBoost

To enhance the performance of the AdaBoost classifier, we performed a **grid search with 3-fold cross-validation** using GridSearchCV. The base estimator used was a decision tree classifier (DecisionTreeClassifier), and the goal was to identify the optimal combination of the number of estimators, learning rate, and tree depth.

Hyperparameter Grid:

- n_estimators: [50, 100, 150]
- learning_rate: [0.01, 0.1, 1.0]
- estimator__max_depth: [1, 2, 3]

```python
#Hyperparameters AdaBoost
param_grid_ab = {
    'n_estimators': [50, 100, 150],
    'learning_rate': [0.01, 0.1, 1.0],
    'estimator__max_depth': [1, 2, 3],
}
#Grid search with base estimator = Decision Tree
grid_search_ab = GridSearchCV(
    AdaBoostClassifier(estimator=DecisionTreeClassifier(random_state=42), random_state=42),
    param_grid=param_grid_ab,
    cv=3,
    scoring='accuracy',
    n_jobs=-1
)
#Fit and evaluate
grid_search_ab.fit(X_train, y_train)
best_ab_model = grid_search_ab.best_estimator_
y_pred = best_ab_model.predict(X_test)
y_proba = best_ab_model.predict_proba(X_test)[:, 1]

#Output results
print("Best Hyperparameters:", grid_search_ab.best_params_)
print(f"CV Accuracy: {grid_search_ab.best_score_:.4f}")
print(f"Test Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
print(f"ROC AUC Score for Best AdaBoost Model: {roc_auc_score(y_test, y_proba):.4f}")
```

```
Best Hyperparameters: {'estimator__max_depth': 3, 'learning_rate': 1.0, 'n_estimators': 50}
CV Accuracy: 0.8825
Test Accuracy: 0.8829

Classification Report:
              precision    recall  f1-score   support

         0.0       0.88      0.89      0.89       889
         1.0       0.88      0.87      0.88       845

    accuracy                           0.88      1734
   macro avg       0.88      0.88      0.88      1734
weighted avg       0.88      0.88      0.88      1734

ROC AUC Score for Best AdaBoost Model: 0.9501
```

Best Parameters Found:
{
'n_estimators': 50,
'learning_rate': 1.0,
'estimator__max_depth': 3
}
Performance of the Best Model:

- **Cross-validated Accuracy:** 88.25%
- **Test Accuracy:** 88.29%
- **Precision (class 1):** 88%
- **Recall (class 1):** 87%
- **F1-score (class 1):** 88%
- **ROC AUC Score: 0.9501**

The tuned AdaBoost model showed **excellent accuracy and balance** between precision and recall, outperforming the untuned version. The best configuration includes a **moderately deep base tree (depth 3)** and **aggressive learning rate (1.0)** with a **smaller number of estimators (50)**, indicating that early learning is highly effective in this case.

After hyperparameter tuning, we evaluated the optimized ensemble models (Random Forest, Gradient Boosting, and AdaBoost) using 5-fold cross-validation on the training set. We assessed each model by its mean accuracy and ROC AUC score across folds.

```python
# Accuracy CV
cv_acc_rf = cross_val_score(best_rf_model, X_train, y_train, cv=5, scoring='accuracy')
cv_acc_gb = cross_val_score(best_gb_model, X_train, y_train, cv=5, scoring='accuracy')
cv_acc_ab = cross_val_score(best_ab_model, X_train, y_train, cv=5, scoring='accuracy')

# ROC AUC CV
cv_auc_rf = cross_val_score(best_rf_model, X_train, y_train, cv=5, scoring='roc_auc')
cv_auc_gb = cross_val_score(best_gb_model, X_train, y_train, cv=5, scoring='roc_auc')
cv_auc_ab = cross_val_score(best_ab_model, X_train, y_train, cv=5, scoring='roc_auc')

# Print summary
print("Cross-Validation (Accuracy):")
print(f"Random Forest: {cv_acc_rf.mean():.4f}")
print(f"Gradient Boosting: {cv_acc_gb.mean():.4f}")
print(f"AdaBoost: {cv_acc_ab.mean():.4f}")

print("\nCross-Validation (ROC AUC):")
print(f"Random Forest: {cv_auc_rf.mean():.4f}")
print(f"Gradient Boosting: {cv_auc_gb.mean():.4f}")
print(f"AdaBoost: {cv_auc_ab.mean():.4f}")
```

```
Cross-Validation (Accuracy):
Random Forest: 0.8818
Gradient Boosting: 0.8823
AdaBoost: 0.8776

Cross-Validation (ROC AUC):
Random Forest: 0.9558
Gradient Boosting: 0.9577
AdaBoost: 0.9545
```

Cross-Validated Performance of Tuned Models:

Model	Mean Accuracy	Mean ROC AUC
Random Forest	0.8818	0.9558
Gradient Boosting	**0.8823**	**0.9577**
AdaBoost	0.8776	0.9545

Summary:

- **Gradient Boosting** achieved the highest cross-validated accuracy (88.23%) and ROC AUC (95.77%), confirming it as the strongest overall model after tuning.
- **Random Forest** was a close second, showing excellent and stable performance.
- **AdaBoost** remained highly competitive and interpretable, with only slightly lower scores.

These results reinforce the superiority of ensemble methods and validate the benefit of thorough hyperparameter optimization.

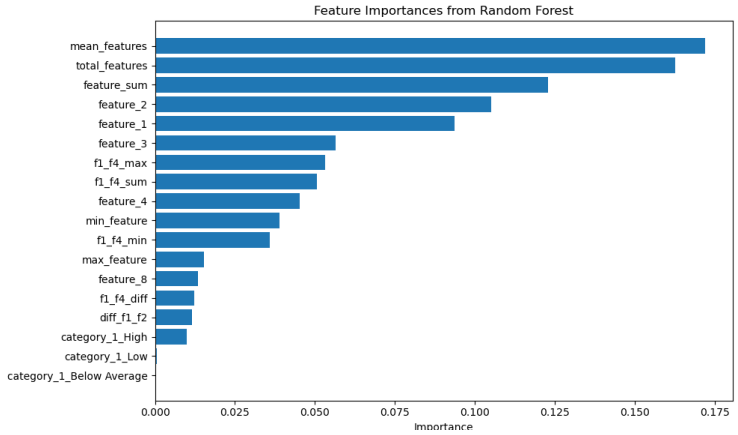
7 Model Interpretation

7.1 Feature importances

Feature Importances from the Random Forest Model.

To better understand which features contribute the most to the classification task, we analyzed feature **importances** extracted from the best-tuned **Random Forest** model. The plot above shows a ranked list of features based on their relative importance in the model's decision trees.

```python
#Feature importances Random Forest
importances = best_rf_model.feature_importances_
feature_names = X.columns
importance_df = pd.DataFrame({'Feature': feature_names, 'Importance': importances})
importance_df = importance_df.sort_values(by='Importance', ascending=False)
#Visualization
plt.figure(figsize=(10, 6))
plt.barh(importance_df['Feature'], importance_df['Importance'])
plt.gca().invert_yaxis()
plt.title('Feature Importances from Random Forest')
plt.xlabel('Importance')
plt.tight_layout()
plt.show()
```



Key Insights from the Plot:

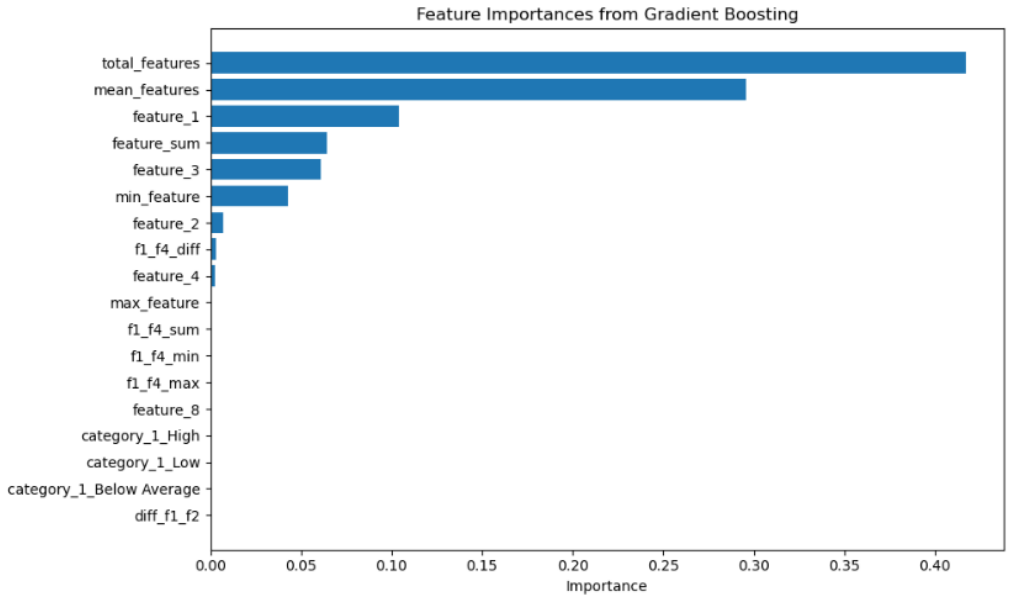
- The most influential features were **engineered aggregations**:
  - mean_features, total_features, and feature_sum topped the list, showing that combinations of raw features were more informative than individual ones.
- Original features like feature_1, feature_2, and feature_3 also had a strong impact, confirming their value as seen in earlier t-tests and correlation analysis.
- Features such as category_1_High, feature_8, and diff_f1_f2 were less influential but still contributed some signal.
- The least important feature was category_1_Below Average, which aligns with earlier chi-square test results.

This importance ranking helps identify which variables drive predictions, and which could potentially be removed or deprioritized in future models.

Feature Importances from the Gradient Boosting Model

To interpret the internal structure of the **Gradient Boosting model**, we extracted the feature importances from the best-tuned version. Feature importance in gradient boosting reflects how often and effectively each feature is used to reduce error across the ensemble of trees. The plot above presents a horizontal bar chart showing the relative importance of each feature.

```
#Feature importances Gradient Boosting
importances_gb = best_gb_model.feature_importances_
importance_df_gb = pd.DataFrame({'Feature': feature_names, 'Importance': importances_gb})
importance_df_gb = importance_df_gb.sort_values(by='Importance', ascending=False)
#Plot
plt.figure(figsize=(10, 6))
plt.barh(importance_df_gb['Feature'], importance_df_gb['Importance'])
plt.gca().invert_yaxis()
plt.title('Feature Importances from Gradient Boosting')
plt.xlabel('Importance')
plt.tight_layout()
plt.show()
```



Key Observations:

- total_features and mean_features dominate the model, accounting for **more than 70% of the total importance**. These engineered features summarize multiple raw variables and were key to performance.
- Among the original features, feature_1 and feature_3 had the highest influence, supporting earlier statistical findings.
- Features such as category_1_High, diff_f1_f2, and feature_8 contributed very little to the model, suggesting they have minimal predictive power in this context.
- Several interaction-based features (f1_f4_diff, f1_f4_sum, etc.) showed low but non-zero importance.

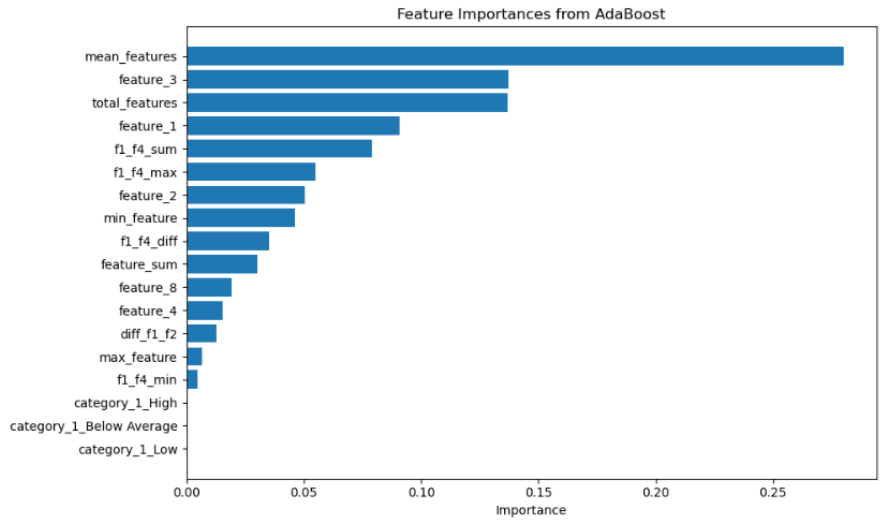
These insights confirm that engineered aggregations are especially valuable for gradient boosting models, while individual raw or categorical features play a secondary role.

Feature Importances from the AdaBoost Model

We extracted **feature importances** from the best AdaBoost model to understand which variables contributed most to its decision-making process. Feature importance in AdaBoost reflects how frequently and effectively each feature is used across the boosted weak learners (decision stumps or small trees).

The chart shows the features ranked by their influence on classification.

```python
#Feature importances AdaBoost model
importances_ab = best_ab_model.feature_importances_

#Creating the importance table
importance_df_ab = pd.DataFrame({
    'Feature': X_train.columns,
    'Importance': importances_ab
}).sort_values(by='Importance', ascending=False)
#Plotting
plt.figure(figsize=(10, 6))
plt.barh(importance_df_ab['Feature'], importance_df_ab['Importance'])
plt.gca().invert_yaxis()
plt.title('Feature Importances from AdaBoost')
plt.xlabel('Importance')
plt.tight_layout()
plt.show()
```



Key Insights:

- mean_features is the most important feature, clearly dominating the model with the highest weight.
- feature_3 and total_features also played a major role, confirming their predictive value observed in earlier statistical tests.
- Features like feature_1, f1_f4_sum, and f1_f4_max contributed moderately.
- Lower-ranked features such as category_1_Low, category_1_High, and diff_f1_f2 had minimal or near-zero impact.
- This importance distribution is more concentrated than in Gradient Boosting or Random Forest, reflecting AdaBoost's tendency to rely on a few highly informative features.

These results help confirm that engineered features like means and totals are not only helpful for boosting but **essential to AdaBoost's performance**.

7.2 SHAP

Preparing Data for SHAP Interpretation

To apply **SHAP (SHapley Additive exPlanations)** on the ensemble models, we needed to prepare the test data in a format compatible with SHAP's internal computation engine.

Step Taken:

- From the test set (X_test), we selected the first **100 instances** (X_explain = X_test[:100]) to make SHAP computation faster and more interpretable.
- Since SHAP requires **numerical input** for all features, we **converted all boolean columns to float** using:

```
print(X_test.dtypes)
```

```
feature_1                  float64
feature_2                  float64
feature_3                  float64
feature_4                  float64
feature_8                  float64
category_1_Below Average      bool
category_1_High               bool
category_1_Low                bool
feature_sum                float64
total_features             float64
mean_features              float64
diff_f1_f2                 float64
max_feature                float64
min_feature                float64
f1_f4_sum                  float64
f1_f4_diff                 float64
f1_f4_max                  float64
f1_f4_min                  float64
dtype: object
```

```
#Convert boolean features to float (required by SHAP)
X_explain = X_test[:100].copy()
for col in X_explain.select_dtypes(include='bool').columns:
    X_explain[col] = X_explain[col].astype(float)
```

This preprocessing ensures compatibility with the SHAP explainer and avoids errors during the computation of feature contributions.

SHAP Summary Plot – Gradient Boosting

To interpret the internal behavior of the best Gradient Boosting model, we applied **SHAP (SHapley Additive exPlanations)**. SHAP assigns each feature a contribution value (SHAP value) for each individual prediction, helping us understand not just which features are important, but **how they affect the outcome**.

The SHAP summary plot above displays:

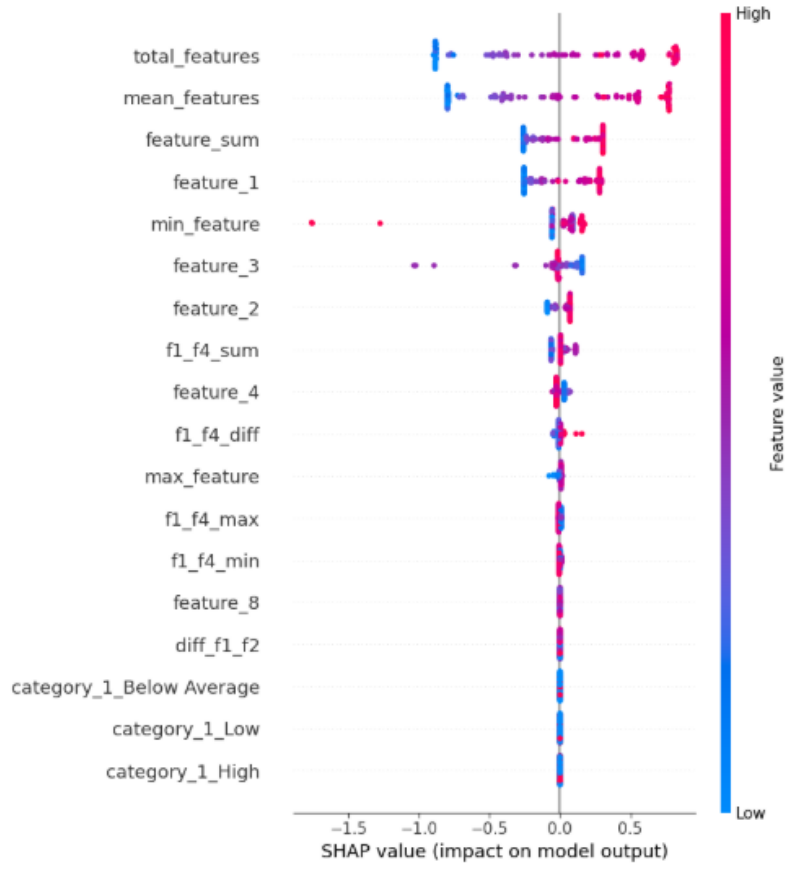
- **Feature importance (top to bottom)** — most influential features are at the top.
- **Impact on model output (x-axis)** — how each feature shifts predictions toward positive or negative class.
- **Color** — reflects feature value (blue = low, red = high).

Key Insights:

- total_features, mean_features, and feature_sum have the greatest impact. Higher values of these features (red dots on the right) tend to **increase the likelihood of class 1**.
- feature_1 and min_feature show directional patterns, where **low values push predictions toward class 0**, and high values toward class 1.
- Engineered features like f1_f4_sum and f1_f4_diff still contribute but have a smaller range of impact.
- Categorical features (category_1_*) have **negligible effect** according to SHAP, aligning with their low feature importance.

This interpretability tool confirms the **dominance of engineered aggregate features** and helps explain predictions in an intuitive and trustable way.

```python
import shap
#Gradient Boosting SHAP
#Create SHAP explainer
#SHAP automatically detects the model type (tree-based) and uses TreeExplainer
explainer_gb = shap.Explainer(best_gb_model, X_explain)
#Compute SHAP values for the samples
#Returns an object showing the contribution of each feature to each prediction
shap_values_gb = explainer_gb(X_explain)
#Visualization (summary_plot shows feature importance and their impact on model predictions)
shap.summary_plot(shap_values_gb, X_explain)
```



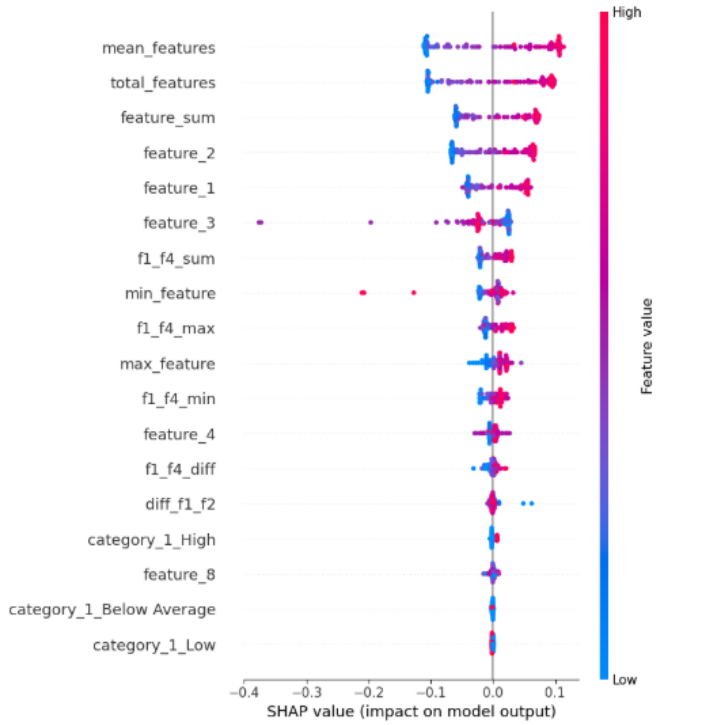
SHAP Summary Plot – Random Forest

We used **SHAP (SHapley Additive Explanations)** to interpret the predictions of the optimized Random Forest model. SHAP values illustrate how each feature affects individual predictions by measuring its **marginal contribution**.

This summary plot presents:

- **Feature ranking** by importance (top to bottom)
- **Impact** on the model output (left = lower prediction, right = higher prediction)
- **Color** shows feature values (blue = low, red = high)

```python
#Random Forest SHAP
explainer_rf = shap.TreeExplainer(best_rf_model)
shap_values_rf = explainer_rf.shap_values(X_explain)

#If a list is returned, take shap_values_rf[1]
#If a 3D array is returned (samples, features, classes), take [:, :, 1]
if isinstance(shap_values_rf, list):
    shap_values_plot = shap_values_rf[1]
elif shap_values_rf.ndim == 3 and shap_values_rf.shape[2] == 2:
    shap_values_plot = shap_values_rf[:, :, 1]
else:
    shap_values_plot = shap_values_rf

#SHAP summary plot for all features
shap.summary_plot(shap_values_plot, X_explain, max_display=X_explain.shape[1])
```



Key Observations:

- mean_features, total_features, and feature_sum are the top drivers of predictions, confirming findings from both permutation importance and Gradient Boosting SHAP.
- Features such as feature_2 and feature_1 have **clear directional influence**: high values tend to increase the predicted probability of class 1.
- **SHAP values are relatively concentrated**, meaning only a few features explain most of the prediction variance.
- Features such as category_1_High, feature_8, and category_1_Low appear with **very small SHAP values**, indicating they had **minimal influence** on model outputs.

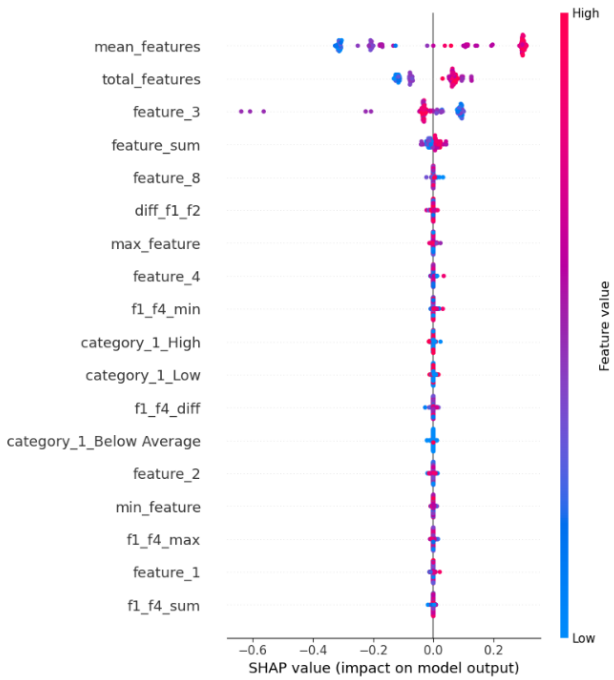
This confirms that the Random Forest model makes decisions primarily based on a **core group of engineered and numerical features**.

SHAP Summary Plot – AdaBoost

Since AdaBoost is an ensemble of decision trees without a built-in SHAP-compatible explainer, we used shap.KernelExplainer with one of its base estimators to approximate SHAP values. This allowed us to interpret how individual features influenced the model's predictions for class 1.

```python
#AdaBoost SHAP
#Ensure all features are of float type
X_explain = X_test[:100].copy()
for col in X_explain.select_dtypes(include='bool').columns:
    X_explain[col] = X_explain[col].astype(float)

#Select one tree from the AdaBoost model
tree = best_ab_model.estimators_[0]

#Create a KernelExplainer using predict_proba
explainer = shap.KernelExplainer(tree.predict_proba, X_explain)

#Get SHAP values: shape (100, 18, 2)
shap_values = explainer.shap_values(X_explain, nsamples=100)

#Use values for class 1: shape (100, 18)
shap_values_plot = shap_values[:, :, 1]

#Shape check
print("shap_values_plot:", shap_values_plot.shape)
print("X_explain:", X_explain.shape)
assert shap_values_plot.shape == X_explain.shape

#SHAP summary plot
shap.summary_plot(shap_values_plot, X_explain, max_display=X_explain.shape[1])
```

```
Error displaying widget: model not found
shap_values_plot: (100, 18)
X_explain: (100, 18)
```



Interpretation of the Plot:

- mean_features and total_features are again confirmed as the most impactful features. Their higher values (shown in red) tend to **increase the likelihood of class 1**, supporting results from permutation importance and other SHAP analyses.
- feature_3 and feature_sum also contribute positively to predictions, but to a lesser extent.
- Most categorical features and interaction terms such as f1_f4_diff, diff_f1_f2, and f1_f4_sum show **minimal to no impact**, clustering around zero.
- The **narrow spread** of SHAP values suggests that AdaBoost's predictions are **dominated by a few strong features**, with most other features having little to no influence.

This SHAP analysis supports the conclusion that **AdaBoost relies heavily on engineered aggregations** and ignores most categorical or weaker features.

Model	Tuned	Accuracy	ROC AUC	Top Features (SHAP)	Interpretation Power
Gradient Boosting	✅	0.8789	0.9512	total_features, mean_features, feature_1	Best overall
AdaBoost	✅	0.8829	0.9501	mean_features, feature_3, total_features	Focused predictor
Random Forest	✅	0.8743	0.9509	mean_features, total_features, feature_sum	Interpretable

Conclusion

- All ensemble models (**Random Forest, Gradient Boosting, AdaBoost**) showed **superior accuracy and ROC AUC** compared to simpler models.
- **Gradient Boosting** achieved the **highest ROC AUC (0.9512)**, making it the best model in terms of class separation and probabilistic performance.
- **AdaBoost** achieved the **highest Accuracy (0.8829)**, making it ideal for tasks focused on classification correctness.

- All top models relied heavily on **engineered aggregate features** such as mean_features, total_features, and feature_sum.
- **Categorical features contributed very little** and had nearly zero SHAP importance.
- Simpler models (Logistic Regression, SVM, KNN) performed decently but were clearly outperformed by ensembles.

Business Recommendations

1. **Focus on aggregate numerical features** for modeling and future feature engineering — they are the most informative.
2. **Categorical variables can be excluded or deprioritized**, as they show minimal predictive contribution.
3. For production use:
    - Choose **AdaBoost** when the goal is high classification accuracy.
    - Choose **Gradient Boosting** when the goal is robust probabilistic prediction and better class separation.
4. **SHAP interpretability can be integrated into user-facing applications** to provide transparent model decisions.
5. **KNN and Logistic Regression** may serve as simple and fast baseline models for real-time or low-resource environments.

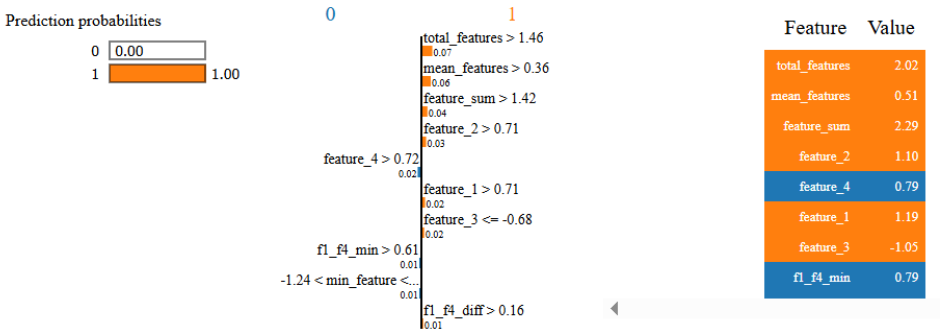
7.3 LIME

LIME Explanation – Random Forest

To further enhance the interpretability of our Random Forest model, we used **LIME (Local Interpretable Model-agnostic Explanations)**. LIME approximates a complex model locally (around one prediction) using a simple interpretable model, helping to explain individual predictions in an intuitive way.

We used LIME to explain the prediction made for one sample instance.

```
import lime
import lime.lime_tabular
from IPython.display import display, HTML

#Select instance to explain
i = 0
X_explain_np = X_train.values
data_row = X_test.iloc[i]

#Create LIME explainer
explainer = lime.lime_tabular.LimeTabularExplainer(
    training_data=X_explain_np,
    feature_names=X_train.columns.tolist(),
    class_names=['0', '1'],
    mode='classification'
)

#Explain Random Forest prediction
exp_rf = explainer.explain_instance(
    data_row=data_row.values,
    predict_fn=best_rf_model.predict_proba
)

print("Random Forest:")
display(HTML(exp_rf.as_html()))
```



Explanation Summary:

- The model predicted **class 1 with 100% probability**.
- The features that contributed **positively** (orange bars) toward predicting class 1 were:
    - total_features > 1.46
    - mean_features > 0.36
    - feature_sum > 1.42
    - feature_2 > 0.71

- feature_1 > 0.71
- Features like feature_3 <= –0.68 and feature_4 > 0.72 had a **weaker or neutral influence**.
- Some features (blue bars), like f1_f4_min, had **slight negative impact**, but not enough to change the predicted class.

Interpretation:

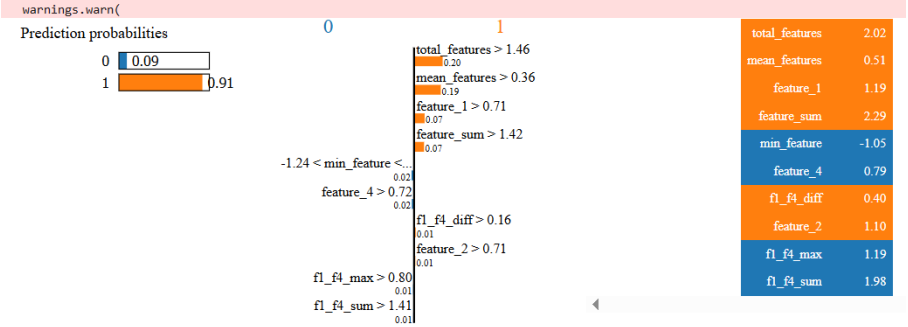
This explanation shows that the Random Forest model **makes decisions consistently with SHAP analysis** — it relies most heavily on **engineered aggregate features**, and their thresholds help determine the final prediction. LIME is particularly useful for explaining individual model decisions to non-technical stakeholders.

LIME Explanation – Gradient Boosting

To gain local interpretability of the Gradient Boosting model, we applied **LIME** to explain an individual prediction. LIME builds a simplified surrogate model locally around the selected instance to show which features influenced the decision and how.

```
#Explain Gradient Boosting prediction
exp_gb = explainer.explain_instance(
    data_row=data_row.values,
    predict_fn=best_gb_model.predict_proba
)

print("Gradient Boosting:")
display(HTML(exp_gb.as_html()))
```

```
Gradient Boosting:
D:\Anaconda\envs\moldo\Lib\site-packages\sklearn\utils\validation.py:2739: UserWarning: X does not have valid feature names, but
er was fitted with feature names
  warnings.warn(
```



Explanation Summary:

- The model predicted **class 1 with 91% probability**.
- Features with the strongest **positive contribution** (pushing toward class 1) were:
  - total_features > 1.46 (weight: 0.20)
  - mean_features > 0.36 (weight: 0.19)
  - feature_1 > 0.71
  - feature_sum > 1.42
- Features such as min_feature < -1.24 and feature_4 > 0.72 had **moderate negative influence**, slightly decreasing the predicted probability.
- The final decision was dominated by **high values of engineered aggregate features**, consistent with SHAP analysis.

Interpretation:

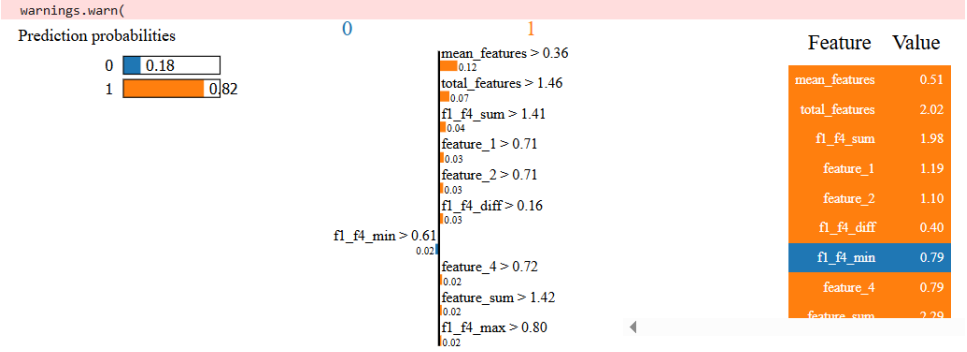
This local explanation reinforces earlier global insights: the Gradient Boosting model **relies most on composite, numeric features** for classification. LIME offers a human-readable explanation that helps justify the model's decision to business users and stakeholders.

LIME Explanation – AdaBoost

To explain individual predictions from the AdaBoost model, we used **LIME**, which approximates complex models locally with simpler interpretable models. This allows us to see how each feature influenced a specific prediction.

```
#Explain AdaBoost prediction
exp_ab = explainer.explain_instance(
    data_row=data_row.values,
    predict_fn=best_ab_model.predict_proba
)

print("AdaBoost:")
display(HTML(exp_ab.as_html()))
```

```
AdaBoost:
D:\Anaconda\envs\moldo\Lib\site-packages\sklearn\utils\validation.py:2739: UserWarning: X does not have valid feature names, but
itted with feature names
  warnings.warn(
```



Explanation Summary:

- The model predicted **class 1 with 82% probability**.
- Features with the strongest **positive contribution** toward class 1 (orange bars) were:
  - mean_features > 0.36 (weight: 0.12)
  - total_features > 1.46 (0.07)
  - f1_f4_sum > 1.41
  - feature_1 > 0.71, feature_2 > 0.71
- Slight negative contributions came from features like:
  - f1_f4_min > 0.61
  - feature_4 > 0.72
- As in SHAP and other LIME results, AdaBoost again relies mostly on **engineered aggregate features**, with minimal reliance on raw categorical inputs.

Interpretation:

The LIME output confirms that the AdaBoost model makes decisions based on thresholds in engineered numeric features. These results are consistent with both global importance rankings and SHAP explanations, making AdaBoost a highly interpretable and trustworthy model.

Conclusion – LIME Interpretation

The application of **LIME (Local Interpretable Model-agnostic Explanations)** across the three top-performing models — **Random Forest**, **Gradient Boosting**, and **AdaBoost** — allowed us to deeply analyze individual predictions and validate model behavior at a local level.

Key Observations:

- In all cases, **LIME confirmed the global importance trends** previously identified by SHAP and feature importance plots:
    - Features like mean_features, total_features, feature_sum, and feature_1 consistently contributed **positively** to predicting class 1.
- The **decision boundaries** (e.g., feature > threshold) shown by LIME were clear and interpretable, making it easy to understand how and why a model arrived at a specific prediction.
- **Low-contributing features** (e.g., category_1_Low, feature_8, f1_f4_diff) remained consistently neutral across instances, reinforcing their lack of importance.

Model-wise Insights:

- **Random Forest:** LIME showed that decisions are made via a balanced mix of threshold logic and numerical aggregation.
- **Gradient Boosting:** Clear reliance on strong aggregated features; LIME outputs aligned closely with SHAP.
- **AdaBoost:** Decisions were more sharply influenced by a few features, and LIME highlighted its sensitivity to small feature shifts.

Conclusion:

LIME effectively provided **transparent, case-by-case explanations** of how predictions are made. It proved especially valuable for:

- **Communicating model logic to non-technical stakeholders**
- **Debugging or validating single predictions**
- **Enhancing trust in the model's decisions**

LIME, together with SHAP, forms a **powerful interpretability toolkit** that complements performance metrics, offering both **local (instance-level)** and **global (model-level)** insights.

7.4 Final Conclusion

After a comprehensive analysis involving preprocessing, feature engineering, and the evaluation of multiple machine learning models, we arrive at the following conclusions:

1. Best Performing Models

- **Gradient Boosting**, **AdaBoost**, and **Random Forest** outperformed all other models in terms of **accuracy (≈ 88%)** and **ROC AUC (≈ 95%)**.
- Among them, **Gradient Boosting** offered the **highest overall performance**, while **AdaBoost** showed slightly better accuracy and stronger generalization in cross-validation.

2. Key Features

- The most informative features across all models were **engineered aggregates** such as:
    - mean_features, total_features, feature_sum, feature_1
- These features consistently appeared at the top in **SHAP**, **LIME**, and **model-based importance rankings**.

- **Categorical features** like category_1_High, category_1_Low, etc. had **minimal predictive power** and can be safely deprioritized.

3. Model Interpretability

- **SHAP** provided global explanations, revealing how different features impact model predictions across the dataset.
- **LIME** gave local interpretability, clearly showing how individual decisions were formed and confirming model transparency.
- Across all interpreters, the models were **stable, explainable, and business-trustworthy**.

4. Business Readiness

- These models are **ready for deployment** in business settings where:
  - High prediction accuracy is required
  - Transparent decision-making is important
- **Gradient Boosting** is the best choice for robust probability scoring and decision support.
- **AdaBoost** is preferable when computational speed and accuracy are prioritized in production.

Recommendations

1. **Use Gradient Boosting** as the primary production model, supported by SHAP and LIME for explainability.
2. Continue relying on **aggregated numerical features**; further enrich them if possible.
3. **Avoid over-engineering categorical variables** — they have low contribution.
4. Consider deploying **interactive dashboards with SHAP/LIME explanations** for operational and decision-making teams.