SECOM- 1st Phase

1. Data Loading

Data understanding		
Purpose	**Library**	**Query**

Load dataset - **SECOM**	**import pandas as pd**	# Replace the file path with the actual path to your file
		
		file_path = "C:/Users/DhruviJayPatel/Documents/SECOMProject/secom_data/secom.data"
		
		# Read the data into a DataFrame
		
		secom = pd.read_csv(file_path, header=None, sep=" ")

Load dataset Label		file_path_2 = "C:/Users/DhruviJayPatel/Documents/SECOMProject/secom_data/secom_labels.data"
		
		label = pd.read_csv(file_path_2, header=None, sep=" ")

Merge datasets		merged_df = pd.concat([secom, label], axis=1)
		
		print(merged_df.columns)

Name predictors (**feature1**, **feature2**…)	**import pandas as pd**	import pandas as pd # Assuming secom is your DataFrame with 591 features # Replace secom with your actual DataFrame # Generate new column names using a list comprehension new_column_names = [f'feature{i}' for i in range(1, 593)]

		# Rename columns in the DataFrame
		merged_df.columns = new_column_names
		
		# Print the DataFrame to verify the changes
		
		print(merged_df)

2. Data Understanding

Data understanding		
Purpose	**Library**	**Query**

Histogram of missing values	import pandas as pd
	import matplotlib.pyplot as plt
		# Calculate the percentage of missing values for each feature
		missing_percentage = (secom.isnull().sum() / len(secom)) * 100
		
		# Plot histogram of percentage of missing values
		plt.figure(figsize=(10, 6))
		plt.hist(missing_percentage, bins=50, color='skyblue', edgecolor='black')
		plt.title('Histogram of Percentage of Missing Values')
		plt.xlabel('Percentage of Missing Values')
		plt.ylabel('Frequency')
		plt.grid(True)
		plt.show()

Histogram of variance	import pandas as pd
	import matplotlib.pyplot as plt
	import numpy as np # Add this line to import NumPy
		import pandas as pd
		import matplotlib.pyplot as plt
		import numpy as np # Add this line to import NumPy
		
		# Assuming df is your DataFrame with 591 features
		# Replace df with your actual DataFrame
		
		# Calculate variance for each feature
		variances = secom.var()
		
		# Generate bin edges with intervals of 0.5 starting from 0
		
		# Plot histogram of variances
		plt.figure(figsize=(20, 10))
		plt.hist(variances, bins=50, color='skyblue', edgecolor='black') # Use bin edges generated above
		plt.title('Histogram of Feature Variances')
		plt.xlabel('Variance')
		plt.ylabel('Frequency')
		plt.grid(True)
		plt.show()

Heatmap	import pandas as pd
	import seaborn as sns
	import matplotlib.pyplot as plt
		import pandas as pd
		import seaborn as sns
		import matplotlib.pyplot as plt
		
		# Assuming df contains your DataFrame with 592 variables
		# Calculate the correlation matrix
		correlation_matrix = secom.corr()
		
		# Plot the heatmap
		plt.figure(figsize=(20, 15))
		sns.heatmap(correlation_matrix, cmap="coolwarm", vmin=-1, vmax=1, center=0, annot=False)
		plt.title('Correlation Heatmap of 592 Variables')
		plt.xlabel('Features') # Add x-axis label
		plt.ylabel('Features') # Add y-axis label
		plt.show()

Duplicates		duplicate_rows = secom.duplicated()
		
		# Count the number of duplicate rows
		num_duplicates = duplicate_rows.sum()
		
		print("Number of duplicate rows:", num_duplicates)

Duplicate_features		total_duplicate_features = sum(secom.T.duplicated())
		
		# Print the total number of duplicate features
		print("Total number of duplicate features:", total_duplicate_features)

Pareto chart missing values	import pandas as pd
	import numpy as np
	import matplotlib.pyplot as plt
		import pandas as pd
		import numpy as np
		import matplotlib.pyplot as plt
		
		# Calculate the percentage of missing values for each column
		missing_percentage = (df.isnull().sum() / len(df)) * 100
		
		# Sort the columns based on the percentage of missing values in descending order
		sorted_indices = missing_percentage.sort_values(ascending=False).index
		
		# Create a histogram of the missing values
		plt.figure(figsize=(10, 6))
		plt.hist(missing_percentage, bins=20, color='skyblue', edgecolor='black')
		plt.xlabel('Percentage of Missing Values')
		plt.ylabel('Frequency')
		plt.title('Histogram of Missing Values')
		
		# Plot Pareto line on the same graph
		cumulative_percentage = (missing_percentage[sorted_indices].cumsum() / missing_percentage.sum() * 100).values
		plt.twinx()
		plt.xlim(0, 100)

2.1 Threshold Definition

Threshold definition	import matplotlib.pyplot as plt	import matplotlib.pyplot as plt
		
		# Define the threshold for missing values (e.g., 10%)
		threshold = 60
		
		# Calculate the percentage of missing values for each column in the training set
		train_missing_percentage = (X_train.isna().mean() * 100).round(2)
		
		# Calculate the percentage of missing values for each column in the testing set
		test_missing_percentage = (X_test.isna().mean() * 100).round(2)
		
		# Plot histogram of missing value percentages for the training set
		plt.figure(figsize=(12, 6))
		train_hist, train_bins, _ = plt.hist(train_missing_percentage, bins=10, range=(40, 100), color='skyblue', edgecolor='black')
		plt.axvline(x=threshold, color='red', linestyle='--', label='Threshold')
		
		# Add annotations for each bar in the training histogram
		for i, freq in enumerate(train_hist):
		plt.text(train_bins[i], freq, str(int(freq)), ha='center', va='bottom')
		
		plt.title('Histogram of Missing Value Percentages in Training Set')
		plt.xlabel('Percentage of Missing Values')
		plt.ylabel('Frequency')
		plt.legend()
		plt.tight_layout()
		plt.show()
		
		# Plot histogram of missing value percentages for the testing set
		plt.figure(figsize=(12, 6))
		test_hist, test_bins, _ = plt.hist(test_missing_percentage, bins=10, range=(40, 100), color='salmon', edgecolor='black')
		plt.axvline(x=threshold, color='red', linestyle='--', label='Threshold')
		
		# Add annotations for each bar in the testing histogram
		for i, freq in enumerate(test_hist):
		plt.text(test_bins[i], freq, str(int(freq)), ha='center', va='bottom')

		plt.title('Histogram of Missing Value Percentages in Testing Set')
		plt.xlabel('Percentage of Missing Values')
		plt.ylabel('Frequency')
		plt.legend()
		plt.tight_layout()
		plt.show()

2.2 Outlier Analysis

Outlier Analysis	import pandas as pd
	import numpy as np
	import matplotlib.pyplot as plt
		import pandas as pd
		import numpy as np
		import matplotlib.pyplot as plt
		
		# Calculate z-scores for each feature
		z_scores = (X_train - X_train.mean()) / X_train.std()
		z_scores_test = (X_test - X_test.mean()) / X_test.std()
		
		# Define threshold for identifying outliers
		threshold = 3
		
		# Find outliers for each feature
		outliers = np.abs(z_scores) > threshold
		outliers_test = np.abs(z_scores_test) > threshold
		
		# Count number of outliers for each feature
		num_outliers = np.sum(outliers, axis=0)
		
		# Calculate percentage of outliers for each feature
		percentage_outliers = (num_outliers / len(X_train)) * 100
		
		import matplotlib.pyplot as plt
		
		# Define bin edges
		bin_edges = [0, 0.001] + list(range(1, 6))
		
		# Plot histogram
		plt.figure(figsize=(10, 6))
		hist = plt.hist(percentage_outliers, bins=bin_edges, color='skyblue', edgecolor='black')
		plt.title("Histogram of Percentages of Outliers in Each Column")
		plt.xlabel("Percentage of Outliers")
		plt.ylabel("Frequency")
		plt.xticks(bin_edges)
		for bar in hist[2]:
		height = int(bar.get_height())
		plt.text(bar.get_x() + bar.get_width() / 2, height, height, ha='center', va='bottom')
		plt.grid(axis='y')
		plt.show()

3 Splitting

Split the data	from sklearn.model_selection import train_test_split	from sklearn.model_selection import train_test_split
		
		y = merged_df['feature591'] # Replace 'target_column_name' with the name of your target column
		
		# Dropping the target variable from the dataframe to get only the features
		X = merged_df.drop('feature591', axis=1)
		
		# Split pass cases into training and testing sets while preserving the distribution
		X_pass_train, X_pass_test, y_pass_train, y_pass_test = train_test_split(
		X[y == -1], y[y == -1], test_size=0.25, random_state=42, stratify=y[y == -1])
		
		# Split fail cases into training and testing sets while preserving the distribution
		X_fail_train, X_fail_test, y_fail_train, y_fail_test = train_test_split(
		X[y == 1], y[y == 1], test_size=0.25, random_state=42, stratify=y[y == 1])
		
		# Concatenate the pass and fail cases in training and testing sets
		X_train = pd.concat([X_pass_train, X_fail_train])
		y_train = pd.concat([y_pass_train, y_fail_train])
		X_test = pd.concat([X_pass_test, X_fail_test])
		y_test = pd.concat([y_pass_test, y_fail_test])
		
		print("Training set - Features:", X_train.shape, "Labels:", y_train.shape)
		print("Testing set - Features:", X_test.shape, "Labels:", y_test.shape)

Fail and Pass Proportion		

original dataset		pass_original_proportion = (merged_df['feature591'] == -1).mean()
		fail_original_proportion = (merged_df['feature591'] == 1).mean()
		
		print("Original Dataset:")
		print("Pass cases proportion:", pass_original_proportion)
		print("Fail cases proportion:", fail_original_proportion)

Train and Test data		# Calculate the proportion of pass and fail cases in the training set
		pass_train_proportion = (y_train == -1).mean()
		fail_train_proportion = (y_train == 1).mean()
		
		# Calculate the proportion of pass and fail cases in the testing set
		pass_test_proportion = (y_test == -1).mean()
		fail_test_proportion = (y_test == 1).mean()
		
		print("Training Set:")
		print("Pass cases proportion:", pass_train_proportion)
		print("Fail cases proportion:", fail_train_proportion)
		print("\nTesting Set:")
		print("Pass cases proportion:", pass_test_proportion)
		print("Fail cases proportion:", fail_test_proportion)

SECOM - 2nd Phase

4 Data Preparation

4.1 Variance

DATA PREPARATION		
Purpose	**Library**	**Query**
Removal Zero Variance Features		

Train		"variance = X_train.var()
		columns_to_keep = variance[variance != 0].index
		X_train = X_train[columns_to_keep]"

Test		X_train = X_train[columns_to_keep]"

4.2 Missing Value Removal

Purpose	**Library**	**Query**

Defining Missing Value Threshold and removal		"missing_percentages = X_train.isnull().mean() * 100
		
		# Identify columns with more than 60% missing values
		columns_to_drop = missing_percentages[missing_percentages > 45].index
		
		# Drop those columns from the training set
		X_train = X_train.drop(columns=columns_to_drop) "

4.3 Outlier Handling

Purpose	**Library**	**Query**
Outlier Handling		

Remove	import pandas as pd
	import numpy as np
		# Replace outliers with NaNs in X_train DataFrame
		X_train = X_train.mask(outliers)
		X_test= X_test.mask(outliers_test)
		
		# Print the first few rows of the cleaned DataFrame to verify
		print("First few rows of X_train after replacing outliers with NaNs:")
		print(X_train.head())

Replace	import pandas as pd
	import numpy as np
		# Function to replace outliers based on 3 standard deviations and count them, excluding specified columns
		def replace_and_count_outliers(df, exclude_columns=[]):
		    outlier_counts = {}
		    for column in df.columns:
		if column not in exclude_columns: mean = df[column].mean()
		            std = df[column].std() lower_bound =
		            mean - 3 * std upper_bound = mean +
		            3 * std
		
		            # Count outliers before replacing
		            outliers = ((df[column] < lower_bound) | (df[column] >
		upper_bound)).sum()
		            outlier_counts[column] = outliers
		
		            # Replace outliers
		            df[column] = np.where(df[column] < lower_bound,
		lower_bound, df[column])
		            df[column] = np.where(df[column] > upper_bound,
		upper_bound, df[column])
		
		    return df, outlier_counts
		
		# Replace outliers in the original training set and count them X_test,
		outliers_count = replace_and_count_outliers(X_test)
		
		# Convert to a DataFrame for better readability outliers_df =
		pd.DataFrame.from_dict(outliers_count, orient='index',
		columns=['Outliers Count'])
		
		# Print the outlier counts after replacement print(outliers_df)

4.4 Imputation

Purpose	Library	Query
Missing Value Imputation		

Imputing	from sklearn.impute import KNNImputer	imputer = KNNImputer(n_neighbors=5)
		
		# Fit the imputer to your data and transform it
		data_imputed = imputer.fit_transform(X_train)
		
		# Convert the imputed data back to a DataFrame
		X_train = pd.DataFrame(data_imputed, columns=X_train.columns)
		from sklearn.impute import KNNImputer
		
		imputer = KNNImputer(n_neighbors=5)
		
		# Fit the imputer to your data and transform it
		data_imputed = imputer.fit_transform(X_test)
		
		# Convert the imputed data back to a DataFrame
		X_test = pd.DataFrame(data_imputed, columns=X_test.columns)

Comparison	import pandas as pd
	import matplotlib.pyplot as plt
	from sklearn.impute import KNNImputer
	from sklearn.experimental import enable_iterative_imputer
	from sklearn.impute import IterativeImputer, SimpleImputer
		import pandas as pd
		import matplotlib.pyplot as plt
		from sklearn.impute import KNNImputer
		from sklearn.experimental import enable_iterative_imputer
		from sklearn.impute import IterativeImputer, SimpleImputer
		
		# Calculate missing percentage per column
		missing_percentage = (X_train.isnull().sum() / len(data)) * 100
		
		# Select columns with missing values between 40% to 65%
		cols_to_plot = missing_percentage[(missing_percentage >= 40) & (missing_percentage <= 65)].index
		
		# Calculate volatility metrics before imputation for selected columns
		volatility_before = data[cols_to_plot].std() # Using standard deviation as an example
		print("\nVolatility Metrics Before Imputation:")
		print(volatility_before)
		
		# Imputation methods
		imputation_methods = ['mean', 'median', 'knn', 'mice']
		imputation_results = {}
		
		# Apply different imputation methods to selected columns only
		for method in imputation_methods:
		if method == 'mean':
		# Mean imputation
		imputer = SimpleImputer(strategy='mean')
		data_imputed = pd.DataFrame(imputer.fit_transform(X_train[cols_to_plot]), columns=cols_to_plot)
		elif method == 'median':
		# Median imputation
		imputer = SimpleImputer(strategy='median')
		data_imputed = pd.DataFrame(imputer.fit_transform(X_train[cols_to_plot]), columns=cols_to_plot)
		elif method == 'knn':
		# KNN imputation
		imputer = KNNImputer(n_neighbors=3)
		data_imputed = pd.DataFrame(imputer.fit_transform(X_train[cols_to_plot]), columns=cols_to_plot)
		elif method == 'mice':
		# MICE (IterativeImputer) imputation
		imputer = IterativeImputer()

```
data_imputed =
pd.DataFrame(imputer.fit_transform(X_train[cols_to_plot]),
columns=cols_to_plot)

# Calculate volatility metrics after imputation for selected columns
volatility_after = data_imputed.std() # Using standard deviation as an
example
imputation_results[method] = volatility_after

# Plot before and after volatility comparison for selected columns only
plt.figure(figsize=(10, 6))
x = range(len(cols_to_plot))

plt.bar(x, volatility_before, width=0.4, alpha=0.6, color='b',
label='Before Imputation')
plt.bar(x, volatility_after, width=0.4, alpha=0.6, color='r', label='After
Imputation')

# Add labels to the bars
for i in x:
plt.text(i, volatility_before[i], f'{volatility_before[i]:.2f}', ha='center',
va='bottom', color='blue')
plt.text(i, volatility_after[i], f'{volatility_after[i]:.2f}', ha='center',
va='bottom', color='red')

plt.title(f'Volatility Comparison - {method.capitalize()} Imputation')
plt.xlabel('Features')
plt.ylabel('Standard Deviation')
plt.xticks(x, cols_to_plot, rotation=45)
plt.legend()
plt.tight_layout()
plt.show()

# Print volatility metrics after each imputation method for selected
columns only
print("\nVolatility Metrics After Imputation:")
for method, volatility_after in imputation_results.items():
print(f"\nMethod: {method.capitalize()}")
print(volatility_after)
```

4.5 Feature Selection/ Reduction

Purpose	Library	Query

Scree Plot	from sklearn.preprocessing import MinMaxScaler from sklearn.decomposition import PCA
	from sklearn.impute import SimpleImputer
	import matplotlib.pyplot as plt
		# Scale the data using Min-Max scaling
		scaler = MinMaxScaler()
		X_train = scaler.fit_transform(X_train)
		
		# Fit PCA
		pca = PCA()
		pca.fit(X_train)
		
		# Plot scree plot
		plt.figure(figsize=(10, 6))
		plt.plot(range(1, len(pca.explained_variance_ratio_) + 1), pca.explained_variance_ratio_, marker='o', linestyle='--') plt.title('Scree Plot')
		plt.xlabel('Number of Components')
		plt.ylabel('Explained Variance Ratio')
		plt.xticks(range(1, len(pca.explained_variance_ratio_) + 1))
		plt.grid(True)
		plt.show()

KMO Test		# Calculate the correlation matrix
		corr_matrix = np.corrcoef(X_train, rowvar=False)
		print("\nCorrelation Matrix:")
		print(corr_matrix)
		
		# Calculate KMO statistic
		try:
		kmo_all, kmo_model = calculate_kmo(X_train)
		print(f"\nKMO statistic: {kmo_model}")
		except ValueError as ve:
		print(f"Error occurred while calculating KMO: {ve}")

Relation between features and target variable		# Calculate correlation with target variable (assuming `target` is your target variable)
		correlation_with_target = X_train.corrwith(y_train)
		
		# Sort correlation values from largest to smallest
		correlation_sorted = correlation_with_target.sort_values(ascending=False)
		
		# Print the sorted correlation Series
		print(correlation_sorted)

BORUTA	import pandas as pd
	import sys
		sys.path.append("C:/Users/DhruviJayPatel/AppData/Local/Programs/Python/Python312/Lib/site-packages")
		from boruta import BorutaPy
		from sklearn.ensemble import RandomForestClassifier
		
		# Assuming X_train and y_train are already defined
		# X_train: DataFrame with your features
		# y_train: Series or array with your target variable
		
		# Initialize RandomForestClassifier
		rf = RandomForestClassifier(n_estimators=100, random_state=42)
		
		# Initialize Boruta
		boruta_selector = BorutaPy(rf, n_estimators='auto', random_state=42)
		
		# Fit the Boruta model
		boruta_selector.fit(X_train.values, y_train)
		
		# Get the boolean mask of selected features
		selected_features = boruta_selector.support_
		
		# Get the column names of selected features
		selected_features_columns = X_train.columns[selected_features]
		
		# Print the selected features
		print("Selected Features:")
		print(selected_features_columns)

BORUTA feature ranking	from boruta import BorutaPy
	from sklearn.ensemble import RandomForestClassifier
	import matplotlib.pyplot as plt
	import numpy as np
		# Example: Initialize Boruta and fit it
		forest = RandomForestClassifier(n_estimators=100, random_state=42)
		boruta_selector = BorutaPy(forest, n_estimators='auto', random_state=42)
		boruta_selector.fit(X_train.values, y_train.values.ravel())
		
		# Get feature names and Boruta rankings
		feature_names = data.columns
		boruta_rankings = boruta_selector.ranking_
		
		# Create a list of tuples (feature, ranking) and sort by ranking
		features_with_ranking = list(zip(feature_names, boruta_rankings))
		features_with_ranking_sorted = sorted(features_with_ranking, key=lambda x: x[1])
		
		# Extract sorted feature names and rankings
		sorted_features = [feat for feat, rank in features_with_ranking_sorted]
		sorted_rankings = [rank for feat, rank in features_with_ranking_sorted]
		
		# Print sorted feature rankings
		print("Sorted Boruta feature rankings:")
		for feat, rank in features_with_ranking_sorted:
		print(f"{feat}: {rank}")

PCA	from sklearn.decomposition import PCA	pca = PCA(n_components=10) # Select the number of components
		X_train_pca = pca.fit_transform(X_train)
		X_test_pca = pca.transform(X_test)

4.6 Balancing

Purpose	**Library**	**Query**

ADASYN		# Step 2: ADASYN - Handle class imbalance
		adasyn = ADASYN(random_state=42)
		X_train_resampled, y_train_resampled = adasyn.fit_resample(X_train_pca, y_train)

SMOTE	from imblearn.over_sampling import SMOTE	from imblearn.over_sampling import SMOTE
		
		smote = SMOTE(random_state=42)
		X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)

Sampling - scatter plot	import pandas as pd
	import numpy as np
	import matplotlib.pyplot as plt
	import seaborn as sns
	from sklearn.model_selection import train_test_split
	from sklearn.preprocessing import StandardScaler
	from sklearn.impute import KNNImputer
	from sklearn.neighbors import KNeighborsClassifier
	from sklearn.metrics import accuracy_score, confusion_matrix
	from imblearn.over_sampling import SMOTE, ADASYN, RandomOverSampler
	from imblearn.under_sampling import RandomUnderSampler
	from collections import Counter
		import pandas as pd
		import numpy as np
		import matplotlib.pyplot as plt
		import seaborn as sns
		from sklearn.model_selection import train_test_split
		from sklearn.preprocessing import StandardScaler
		from sklearn.impute import KNNImputer
		from sklearn.neighbors import KNeighborsClassifier
		from sklearn.metrics import accuracy_score, confusion_matrix
		from imblearn.over_sampling import SMOTE, ADASYN, RandomOverSampler
		from imblearn.under_sampling import RandomUnderSampler
		from collections import Counter
		
		# Assuming X_train, y_train, X_test, y_test are already defined and preprocessed
		
		# 1. Remove features with more than 65% missing values from the training data
		threshold = 0.65
		missing_ratio = X_train.isnull().mean()
		features_to_drop = missing_ratio[missing_ratio > threshold].index
		X_train.drop(features_to_drop, axis=1, inplace=True)
		X_test.drop(features_to_drop, axis=1, inplace=True)
		
		# 2. Replace outliers using a 3-sigma boundary
		def replace_outliers(df):
		for col in df.select_dtypes(include=[np.number]).columns:
		mean = df[col].mean()
		std = df[col].std()
		upper_bound = mean + 3 * std
		lower_bound = mean - 3 * std
		df[col] = np.clip(df[col], lower_bound, upper_bound)
		return df
		
		X_train = replace_outliers(X_train)
		X_test = replace_outliers(X_test)
		
		# 3. Perform kNN imputation
		imputer = KNNImputer()
		X_train_imputed = imputer.fit_transform(X_train)
		X_test_imputed = imputer.transform(X_test)
		
		# Convert back to DataFrame
		X_train = pd.DataFrame(X_train_imputed, columns=X_train.columns)

```
X_test = pd.DataFrame(X_test_imputed, columns=X_test.columns)

# Function to create scatter plot
def create_scatter_plot(X, y, title, ax):
X_sample = X.iloc[:, :2] # selecting first two features for visualization
sns.scatterplot(x=X_sample.iloc[:, 0], y=X_sample.iloc[:, 1],
hue=y.map({1: 'blue', -1: 'red'}), ax=ax, palette=['blue', 'red'])
ax.set_title(title)

# Count the number of pass and fail cases
counts = Counter(y)
pass_count = counts.get(-1, 0)
fail_count = counts.get(1, 0)

# Add text annotations to the plot
textstr = f'Majority (Pass): {pass_count}\nMinority (Fail): {fail_count}'
props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)
ax.text(0.05, 0.95, textstr, transform=ax.transAxes, fontsize=12,
verticalalignment='top', bbox=props)

# Remove legend
ax.get_legend().remove()

# Sampling methods
sampling_methods = {
'Original': (X_train, y_train),
'Undersampling':
RandomUnderSampler(random_state=42).fit_resample(X_train,
y_train),
'Oversampling':
RandomOverSampler(random_state=42).fit_resample(X_train,
y_train),
'SMOTE': SMOTE(random_state=42).fit_resample(X_train, y_train),
'ADASYN': ADASYN(random_state=42).fit_resample(X_train,
y_train),
'ROSE':
RandomOverSampler(random_state=42).fit_resample(X_train,
y_train)
}

# Plotting scatter plots
fig, axes = plt.subplots(3, 2, figsize=(18, 18))
axes = axes.ravel()
for ax, (method, (X_resampled, y_resampled)) in zip(axes,
sampling_methods.items()):
create_scatter_plot(X_resampled, y_resampled, method, ax)
```

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

		# Training and evaluation
		knn = KNeighborsClassifier(n_neighbors=5)

		for method, (X_resampled, y_resampled) in
		sampling_methods.items():
		# Reversing class labels for training
		y_resampled_reversed = np.where(y_resampled == 1, -1, 1)
		knn.fit(X_resampled, y_resampled_reversed)
		y_pred = knn.predict(X_test)

		acc = accuracy_score(y_test, y_pred)
		conf_matrix = confusion_matrix(y_test, y_pred)

		print(f"Method: {method}")
		print(f"Accuracy: {acc}")
		print("Confusion Matrix:")
		print(conf_matrix)
		print("\n")

		# Print the number of pass and fail cases
		counts_resampled = Counter(y_resampled)
		pass_count_resampled = counts_resampled.get(-1, 0)
		fail_count_resampled = counts_resampled.get(1, 0)
		print(f"Resampled Pass (Majority): {pass_count_resampled},
		Resampled Fail (Minority): {fail_count_resampled}")
		print("\n")
		```

5. Building Basic Model

5.1 Model 1

```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.impute import KNNImputer
from boruta import BorutaPy
from imblearn.over_sampling import SMOTE

# Assuming 'X_train', 'y_train', 'X_test', 'y_test' are your training and test sets

# Step 1: Load and preprocess your data
# Example: Remove features with >65% missing values
threshold = 0.65
missing_counts = X_train.isnull().sum()
cols_to_remove = missing_counts[missing_counts / len(X_train) > threshold].index
X_train = X_train.drop(cols_to_remove, axis=1)
X_test = X_test.drop(cols_to_remove, axis=1)

# Step 2: Handle outliers using Z-score and replace with NaN
def handle_outliers_zscore(df, cols=None, threshold=3):
if cols is None:
cols = df.select_dtypes(include=[np.number]).columns

for col in cols:
z_scores = np.abs((df[col] - df[col].mean()) / df[col].std())
df[col] = np.where(z_scores > threshold, np.nan, df[col])

return df

# Apply outlier handling using Z-score to both X_train and X_test
X_train = handle_outliers_zscore(X_train)
X_test = handle_outliers_zscore(X_test)

# Step 3: KNN imputation to replace outliers and other missing values
knn_imputer = KNNImputer(n_neighbors=5)
X_train_imputed = pd.DataFrame(knn_imputer.fit_transform(X_train),
columns=X_train.columns)
X_test_imputed = pd.DataFrame(knn_imputer.transform(X_test),
columns=X_test.columns)

# Step 4: Feature Selection using Boruta
rf = RandomForestClassifier(n_estimators=100)
boruta_selector = BorutaPy(rf, n_estimators='auto', verbose=2, random_state=1)
boruta_selector.fit(X_train_imputed.values, y_train.values)

selected_features = X_train_imputed.columns[boruta_selector.support_]

X_train_selected = X_train_imputed[selected_features]
X_test_selected = X_test_imputed[selected_features]

# Step 5: Handling imbalanced dataset using SMOTE
smote = SMOTE(random_state=1)
```

```
X_train_balanced, y_train_balanced = smote.fit_resample(X_train_selected, y_train)

# Step 6: Train the Random Forest model
rf_model = RandomForestClassifier(n_estimators=100, random_state=1)
rf_model.fit(X_train_balanced, y_train_balanced)

# Step 7: Model evaluation
# Predict on training set
y_train_pred = rf_model.predict(X_train_balanced)
train_accuracy = accuracy_score(y_train_balanced, y_train_pred)
train_error = 1 - train_accuracy
train_confusion_matrix = confusion_matrix(y_train_balanced, y_train_pred)

# Predict on test set
y_test_pred = rf_model.predict(X_test_selected)
test_accuracy = accuracy_score(y_test, y_test_pred)
test_error = 1 - test_accuracy
test_confusion_matrix = confusion_matrix(y_test, y_test_pred)

print("Train Confusion Matrix:")
print(train_confusion_matrix)
print("Train Accuracy:", train_accuracy)
print("Train Error:", train_error)

print("\nTest Confusion Matrix:")
print(test_confusion_matrix)
print("Test Accuracy:", test_accuracy)
print("Test Error:", test_error)

# Print model accuracy
print("\nModel Accuracy on Test Set:", test_accuracy)
```

5.2 Customized

```
import pandas as pd
import numpy as np
from sklearn.impute import KNNImputer
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from imblearn.over_sampling import SMOTE
import matplotlib.pyplot as plt
import seaborn as sns

# Assuming X_train, y_train, X_test, y_test is already defined

# 1. Remove features with more than 65% missing values
threshold = 0.65
missing_ratio = X_train.isnull().mean()
features_to_drop = missing_ratio[missing_ratio > threshold].index
X_train.drop(features_to_drop, axis=1, inplace=True)
X_test.drop(features_to_drop, axis=1, inplace=True)

# 2. Replace outliers using a 3-sigma boundary
def replace_outliers(df):
```

```
for col in df.select_dtypes(include=[np.number]).columns:
mean = df[col].mean()
std = df[col].std()
upper_bound = mean + 3 * std
lower_bound = mean - 3 * std
df[col] = np.clip(df[col], lower_bound, upper_bound)
return df

X_train = replace_outliers(X_train)
X_test = replace_outliers(X_test)

# 3. Perform kNN imputation
imputer = KNNImputer()
X_train_imputed = imputer.fit_transform(X_train)
X_test_imputed = imputer.transform(X_test)

# Convert back to DataFrame
X_train = pd.DataFrame(X_train_imputed, columns=X_train.columns)
X_test = pd.DataFrame(X_test_imputed, columns=X_test.columns


 # 4. Select specified features for model building
selected_features = ['feature60', 'feature65', 'feature66', 'feature342', 'feature351', 'feature478',
'feature540', 'feature563']
X_train = X_train[selected_features]
X_test = X_test[selected_features]

# Apply SMOTE for balancing the training data
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Train Random Forest model
rf = RandomForestClassifier(random_state=42)
rf.fit(X_train_smote, y_train_smote)

# Make predictions on the test set
y_pred = rf.predict(X_test)

# Evaluate the model
acc = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

print(f"Accuracy: {acc}")
print("Confusion Matrix:")
print(conf_matrix)
print("Classification Report:")
print(class_report)

# Calculate and print the loss cost
def calculate_loss_cost(conf_matrix, cost_fp, cost_fn):
# Confusion matrix format:
```

```
# [[TN, FP],
# [FN, TP]]
tn, fp, fn, tp = conf_matrix.ravel()
loss_cost = (fp * cost_fp) + (fn * cost_fn)
return loss_cost

cost_fp = 1000 # Example cost for False Positive
cost_fn = 5000 # Example cost for False Negative
loss_cost = calculate_loss_cost(conf_matrix, cost_fp, cost_fn)

print(f"Loss Cost: {loss_cost}")

# Plot feature importance
feature_importances = rf.feature_importances_
indices = np.argsort(feature_importances)[::-1]

plt.figure(figsize=(12, 6))
sns.barplot(x=feature_importances[indices], y=np.array(selected_features)[indices])
plt.title('Feature Importance')
plt.xlabel('Importance')
plt.ylabel('Features')
plt.show()
```

5.3 Pipeline

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score,
roc_auc_score
from sklearn.decomposition import PCA
from boruta import BorutaPy
from imblearn.over_sampling import SMOTE, RandomOverSampler
from sklearn.impute import KNNImputer, SimpleImputer, IterativeImputer

# Step 1: Remove features with missing values above the threshold
thresholds = [45, 50, 55, 60, 65]

def remove_features_with_missing_values(X, threshold):
return X.loc[:, X.isnull().mean() * 100 < threshold]

# Step 3: Handle outliers - replace with 3 standard deviation boundaries and put NA for each outlier
value
def handle_outliers(df):
for col in df.select_dtypes(include=[np.number]).columns:
upper_bound = df[col].mean() + 3 * df[col].std()
lower_bound = df[col].mean() - 3 * df[col].std()
df[col] = np.where((df[col] > upper_bound) | (df[col] < lower_bound), np.nan, df[col])
return df
```

```
# Step 4: Missing value imputation
def impute_missing_values(df, method):
if method == 'KNN':
imputer = KNNImputer()
elif method == 'MICE':
imputer = IterativeImputer()
elif method == 'Mean':
imputer = SimpleImputer(strategy='mean')


return pd.DataFrame(imputer.fit_transform(df), columns=df.columns)

# Step 5: Feature selection/reduction
# Boruta
def boruta_feature_selection(X, y):
rf = RandomForestClassifier(n_jobs=-1, class_weight='balanced', max_depth=5)
feat_selector = BorutaPy(rf, n_estimators='auto', verbose=0, random_state=42)
feat_selector.fit(X.values, y.values)
selected_features = X.columns[feat_selector.support_].tolist()
return selected_features, 'Boruta'

# PCA
def pca_reduction(X, n_components):
pca = PCA(n_components=n_components)
return pca.fit_transform(X), 'PCA'

# Step 6: Data sampling for imbalanced dataset
def handle_imbalance(X, y, method):
if method == 'SMOTE':
smote = SMOTE(random_state=42)
X_res, y_res = smote.fit_resample(X, y)
elif method == 'RandomOverSampler':
ros = RandomOverSampler(random_state=42)
X_res, y_res = ros.fit_resample(X, y)
return X_res, y_res

# Function to calculate Loss Cost based on Confusion Matrix
def calculate_loss_cost(conf_matrix, cost_fp, cost_fn):
total_fp = conf_matrix[0, 1]
total_fn = conf_matrix[1, 0]
loss_cost = total_fp * cost_fp + total_fn * cost_fn
return loss_cost

# List to store model performance
results = []
boruta_selected_features = [] # List to collect Boruta selected features

# Iterate through thresholds
for threshold in thresholds:
X_train_thresh = remove_features_with_missing_values(X_train, threshold)
```

```
X_test_thresh = X_test[X_train_thresh.columns]

# Handle outliers
X_train_outliers = handle_outliers(X_train_thresh)
X_test_outliers = handle_outliers(X_test_thresh)

# Iterate through feature selection/reduction methods

for feature_method_choice in ['Boruta', 'PCA']:
# Iterate through imputation methods
for impute_method in ['KNN', 'MICE', 'Mean']:
X_train_imputed = impute_missing_values(X_train_outliers, impute_method)
X_test_imputed = impute_missing_values(X_test_outliers, impute_method)

# Perform feature selection/reduction based on choice
if feature_method_choice == 'Boruta':
selected_features, feature_method = boruta_feature_selection(X_train_imputed, y_train)
X_train_selected = X_train_imputed[selected_features]
X_test_selected = X_test_imputed[selected_features]
elif feature_method_choice == 'PCA':
X_train_selected, feature_method = pca_reduction(X_train_imputed, n_components=30)
X_test_selected = pca_reduction(X_test_imputed, n_components=30)[0]

# Iterate through imbalance methods
for imbalance_method in ['SMOTE', 'RandomOverSampler']:
X_train_res, y_train_res = handle_imbalance(X_train_selected, y_train, imbalance_method)

# Train Random Forest model
rf_model = RandomForestClassifier(random_state=42)
rf_model.fit(X_train_res, y_train_res)

# Predictions
y_pred = rf_model.predict(X_test_selected)

# Evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
auc = roc_auc_score(y_test, y_pred)

# Calculate Loss Cost
cost_fp = 1000 # Example cost for False Positive
cost_fn = 5000 # Example cost for False Negative
loss_cost = calculate_loss_cost(conf_matrix, cost_fp, cost_fn)

# Prepare results
result = {
'threshold': threshold,
'impute_method': impute_method,
'imbalance_method': imbalance_method,
'accuracy': accuracy,
'confusion_matrix': conf_matrix,
'precision': precision,
'recall': recall,
'f1_score': f1,
'auc': auc,
'loss_cost': loss_cost,
'features_used': feature_method,
'selected_features': selected_features if feature_method == 'Boruta' else None
}

results.append(result)

# Collect Boruta selected features
if feature_method == 'Boruta':
boruta_selected_features.append(selected_features)

# Convert results to DataFrame
results_df = pd.DataFrame(results)

# Display top 10 models by loss cost
top_10_models_loss_cost = results_df.sort_values(by='loss_cost', ascending=True).head(10)
print("\nTop 10 Models by Loss Cost:")
print(top_10_models_loss_cost[['threshold', 'impute_method', 'imbalance_method', 'accuracy',
'confusion_matrix',
'precision', 'recall', 'f1_score', 'auc', 'loss_cost', 'features_used', 'selected_features']])

# Display top 10 models by minimizing False Positive and False Negative errors
top_10_models_fp_fn = results_df.sort_values(by=['confusion_matrix'], key=lambda x: x.apply(lambda
y: (y[0][1], y[1][0])), ascending=True).head(10)
print("\nTop 10 Models by False Positive and False Negative Errors:")
print(top_10_models_fp_fn[['threshold', 'impute_method', 'imbalance_method', 'accuracy',
'confusion_matrix',
'precision', 'recall', 'f1_score', 'auc', 'loss_cost', 'features_used', 'selected_features']])

# Print Boruta selected features if present in top models by loss cost
if boruta_selected_features:
print("\nBoruta Selected Features in Top Models:")
for idx, features in enumerate(boruta_selected_features, start=1):
print(f"Iteration {idx}: {features}")
else:
print("\nNo Boruta selected features in Top Models."
```

SECOM – 3rd Phase

6 Modeling and Evaluation

6.1 Scaling

Purpose	Library	Query

Scaling	import numpy as np
	import matplotlib.pyplot as plt
	from sklearn.preprocessing import MinMaxScaler
	from scipy import stats
		# Assuming 'merged_df' is your DataFrame with 'feature17'
		feature17 = merged_df['feature17'].values.reshape(-1, 1)
		
		# Apply different transformations
		
		# Min-Max Scaling
		scaler_minmax = MinMaxScaler()
		feature17_minmax = scaler_minmax.fit_transform(feature17)
		
		# Log Transformation
		feature17_log = np.log(feature17 + 1) # Adding 1 to handle zeros
		
		# Linear Transformation
		feature17_linear = 0.5 * feature17 # Scaling by a factor of 0.5
		
		# Box-Cox Transformation
		feature17_boxcox, _ = stats.boxcox(feature17.flatten() + 1) # Adding 1 to handle zeros
		
		# Plotting
		fig, axes = plt.subplots(2, 2, figsize=(18, 12))
		
		# Min-Max Scaling
		axes[0, 0].hist(feature17_minmax.flatten(), bins=30, color='green', alpha=0.7)
		axes[0, 0].set_title('Min-Max Scaling (Feature 17)')
		
		# Log Transformation
		axes[0, 1].hist(feature17_log.flatten(), bins=30, color='purple', alpha=0.7)
		axes[0, 1].set_title('Log Transformation (Feature 17)')
		
		# Linear Transformation
		axes[1, 0].hist(feature17_linear.flatten(), bins=30, color='orange', alpha=0.7)
		axes[1, 0].set_title('Linear Transformation (Feature 17)')
		
		# Box-Cox Transformation
		axes[1, 1].hist(feature17_boxcox, bins=30, color='red', alpha=0.7)
		axes[1, 1].set_title('Box-Cox Transformation (Feature 17)')
		
		plt.tight_layout()
		plt.show()

6.2 Optimal Parameters

Purpose	**Library**	**Query**

Optimal Parameters	import numpy as np
	import pandas as pd
	import matplotlib.pyplot as plt
	from sklearn.decomposition import PCA
	from boruta import BorutaPy
	from sklearn.ensemble import RandomForestClassifier
	from imblearn.over_sampling import SMOTE, ADASYN
	from imblearn.combine import SMOTETomek
	from sklearn.tree import DecisionTreeClassifier
	from sklearn.linear_model import LogisticRegression
	from sklearn.naive_bayes import GaussianNB
	from sklearn.svm import SVC
	from sklearn.neighbors import KNeighborsClassifier
	from sklearn.preprocessing import StandardScaler
	from sklearn.metrics import f1_score, precision_score, recall_score, confusion_matrix
		# Define models
		models = {
		'RandomForest': RandomForestClassifier(),
		'DecisionTree': DecisionTreeClassifier(),
		'LogisticRegression': LogisticRegression(),
		'NaiveBayes': GaussianNB(),
		'SVM': SVC(),
		'KNN': KNeighborsClassifier()
		}
		
		# Define feature selection and balancing techniques
		feature_methods = ['PCA', 'Boruta']
		balance_methods = ['SMOTE', 'ADASYN', 'ROSE']
		
		# Function to apply PCA
		def apply_pca(X_train, X_test, n_components=10):
		pca = PCA(n_components=n_components)
		X_train_pca = pca.fit_transform(X_train)
		X_test_pca = pca.transform(X_test)
		return X_train_pca, X_test_pca
		
		# Function to apply Boruta
		def apply_boruta(X_train, y_train, X_test):
		rf = RandomForestClassifier(n_jobs=-1, class_weight='balanced', max_depth=5)
		boruta = BorutaPy(rf, n_estimators='auto', verbose=0, random_state=1)
		boruta.fit(X_train.values, y_train)
		X_train_boruta = boruta.transform(X_train.values)
		X_test_boruta = boruta.transform(X_test.values)
		return X_train_boruta, X_test_boruta
		
		# Balancing functions
		def balance_data(X_train, y_train, method):
		if method == 'SMOTE':
		smote = SMOTE(random_state=42)
		X_res, y_res = smote.fit_resample(X_train, y_train)
		elif method == 'ADASYN':
		adasyn = ADASYN(random_state=42)
		X_res, y_res = adasyn.fit_resample(X_train, y_train)
		elif method == 'ROSE':
		rose = SMOTETomek(random_state=42)
		X_res, y_res = rose.fit_resample(X_train, y_train)
		return X_res, y_res
		
		# Evaluation function
		def evaluate_model(model, X_train, y_train, X_test, y_test):
		model.fit(X_train, y_train)
		y_pred = model.predict(X_test)
		f1 = f1_score(y_test, y_pred)

```
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()
loss = fp * 1000 + fn * 5000
return f1, precision, recall, loss

# Collecting results
results = []
for feature_method in feature_methods:
if feature_method == 'PCA':
X_train_fs, X_test_fs = apply_pca(X_train_imputed,
X_test_imputed)
elif feature_method == 'Boruta':
X_train_fs, X_test_fs = apply_boruta(X_train_imputed, y_train,
X_test_imputed)

for balance_method in balance_methods:
X_train_bal, y_train_bal = balance_data(X_train_fs, y_train,
balance_method)

# Apply scaling where necessary
scaler = StandardScaler()
X_train_bal_scaled = scaler.fit_transform(X_train_bal)
X_test_fs_scaled = scaler.transform(X_test_fs)

for model_name, model in models.items():
if model_name in ['SVM', 'KNN', 'LogisticRegression']:
f1, precision, recall, loss = evaluate_model(model,
X_train_bal_scaled, y_train_bal, X_test_fs_scaled, y_test)
else:
f1, precision, recall, loss = evaluate_model(model, X_train_bal,
y_train_bal, X_test_fs, y_test)
results.append([f'{feature_method}+{balance_method}',
model_name, f1, precision, recall, loss])

# Convert results to DataFrame
results_df = pd.DataFrame(results, columns=['Method', 'Model',
'F1 Score', 'Precision', 'Recall', 'Loss'])

# Plotting
fig, axs = plt.subplots(2, 2, figsize=(15, 10))

# Plot F1 Score
x = np.arange(len(results_df['Method'].unique()))
width = 0.15

for i, model_name in enumerate(models.keys()):
subset = results_df[results_df['Model'] == model_name]
axs[0, 0].bar(x + i*width, subset['F1 Score'], width,
label=model_name)
axs[0, 0].set_title('F1 Score')
axs[0, 0].set_xticks(x + width*(len(models)/2))
axs[0, 0].set_xticklabels(results_df['Method'].unique(),
rotation=90)
axs[0, 0].legend()

# Plot Precision
for i, model_name in enumerate(models.keys()):
```

```
subset = results_df[results_df['Model'] == model_name]
axs[0, 1].bar(x + i*width, subset['Precision'], width,
label=model_name)
axs[0, 1].set_title('Precision')
axs[0, 1].set_xticks(x + width*(len(models)/2))
axs[0, 1].set_xticklabels(results_df['Method'].unique(),
rotation=90)
axs[0, 1].legend()

# Plot Recall
for i, model_name in enumerate(models.keys()):
subset = results_df[results_df['Model'] == model_name]
axs[1, 0].bar(x + i*width, subset['Recall'], width,
label=model_name)
axs[1, 0].set_title('Recall')
axs[1, 0].set_xticks(x + width*(len(models)/2))
axs[1, 0].set_xticklabels(results_df['Method'].unique(),
rotation=90)
axs[1, 0].legend()

# Plot Loss
for i, model_name in enumerate(models.keys()):
subset = results_df[results_df['Model'] == model_name]
axs[1, 1].bar(x + i*width, subset['Loss'], width,
label=model_name)
axs[1, 1].set_title('Loss')
axs[1, 1].set_xticks(x + width*(len(models)/2))
axs[1, 1].set_xticklabels(results_df['Method'].unique(),
rotation=90)
axs[1, 1].legend()

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

6.3 Finding best fit models among RF, NB and SVM

Modeling and Evaluation		
Purpose	**Library**	**Query**

Hyperparameter tuning + graphs	import pandas as pd
	import numpy as np
	from sklearn.model_selection import train_test_split, GridSearchCV, KFold
	from sklearn.impute import KNNImputer
	from sklearn.preprocessing import StandardScaler
	from sklearn.ensemble import RandomForestClassifier
	from sklearn.svm import SVC
	from sklearn.naive_bayes import GaussianNB
	from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, roc_curve, auc
	from imblearn.over_sampling import SMOTE
	import matplotlib.pyplot as plt
	from boruta import BorutaPy
		# Selecting specific features identified by Boruta
		selected_features = ['feature60', 'feature65', 'feature66', 'feature342', 'feature351', 'feature478', 'feature540', 'feature563']
		
		X_train_selected = X_train[selected_features]
		X_test_selected = X_test[selected_features]
		
		# SMOTE Data Balancing
		sm = SMOTE(random_state=42)
		X_train_res, y_train_res = sm.fit_resample(X_train_selected, y_train)
		
		# Scaling the Features for SVM only
		scaler = StandardScaler()
		X_train_res_scaled = scaler.fit_transform(X_train_res)
		X_test_scaled = scaler.transform(X_test_selected)
		
		# Function to evaluate model and return evaluation metrics
		def evaluate_model(model, X_train, y_train, X_test, y_test):
		model.fit(X_train, y_train)
		y_pred = model.predict(X_test)
		y_pred_prob = model.predict_proba(X_test)[:, 1]
		
		accuracy = accuracy_score(y_test, y_pred)
		precision = precision_score(y_test, y_pred)
		recall = recall_score(y_test, y_pred)
		f1 = f1_score(y_test, y_pred)
		conf_matrix = confusion_matrix(y_test, y_pred)
		
		return accuracy, precision, recall, f1, conf_matrix, y_pred_prob
		
		# Define models
		models = [
		SVC(probability=True),
		GaussianNB(),
		RandomForestClassifier(n_estimators=100, random_state=42)
		]
		
		# Define colors for each model consistently
		model_colors = {
		'RandomForestClassifier': 'green',
		'GaussianNB': 'lightgreen',
		'SVC': 'red'
		}
		
		# Lists to store results before and after tuning
		models_results_before = []
		models_results_after = []
		roc_curves_before = []
		roc_curves_after = []
		
		# Evaluate each model before tuning
		for model in models:

```
model_name = model.__class__.__name__
color = model_colors[model_name]

if isinstance(model, SVC):
accuracy, precision, recall, f1, conf_matrix, y_pred_prob =
evaluate_model(model, X_train_res_scaled, y_train_res,
X_test_scaled, y_test)
else:
accuracy, precision, recall, f1, conf_matrix, y_pred_prob =
evaluate_model(model, X_train_res, y_train_res, X_test_selected,
y_test)

models_results_before.append({
'Model': model_name,
'Accuracy': accuracy,
'Precision': precision,
'Recall': recall,
'F1 Score': f1,
'Confusion Matrix': conf_matrix,
'Color': color
})

fpr, tpr, _ = roc_curve(y_test, y_pred_prob)
roc_auc = auc(fpr, tpr)
roc_curves_before.append((fpr, tpr, roc_auc, model_name, color))

# Define parameter grids for tuning
param_grids = {
'SVC': {'C': [0.1, 1, 10], 'gamma': ['scale', 'auto'], 'kernel': ['linear',
'rbf']},
'GaussianNB': {'var_smoothing': [1e-09, 1e-08, 1e-07]},
'RandomForestClassifier': {'max_depth': [None, 5, 10]}
}

# Define k-fold Cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Evaluate each model after tuning with k-fold
for model in models:
model_name = model.__class__.__name__
param_grid = param_grids[model_name]
color = model_colors[model_name]

grid_search = GridSearchCV(model, param_grid, scoring='roc_auc',
cv=kf, n_jobs=-1)
if isinstance(model, SVC):
grid_search.fit(X_train_res_scaled, y_train_res)
best_model = grid_search.best_estimator_
accuracy, precision, recall, f1, conf_matrix, y_pred_prob =
evaluate_model(best_model, X_train_res_scaled, y_train_res,
X_test_scaled, y_test)
else:
grid_search.fit(X_train_res, y_train_res)
best_model = grid_search.best_estimator_
accuracy, precision, recall, f1, conf_matrix, y_pred_prob =
evaluate_model(best_model, X_train_res, y_train_res,
X_test_selected, y_test)
```

```
best_params = grid_search.best_params_
best_score = grid_search.best_score_

print(f"Best Parameters for {model_name}: {best_params}")
print(f"Best ROC AUC Score: {best_score}")

models_results_after.append({
'Model': model_name,
'Accuracy': accuracy,
'Precision': precision,
'Recall': recall,
'F1 Score': f1,
'Confusion Matrix': conf_matrix,
'Color': color
})

fpr, tpr, _ = roc_curve(y_test, y_pred_prob)
roc_auc = auc(fpr, tpr)
roc_curves_after.append((fpr, tpr, roc_auc, model_name, color))

# Plotting comparison graphs
metrics = ['Precision', 'Recall', 'F1 Score', 'Accuracy']
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

for i, metric in enumerate(metrics):
ax = axes[i//2, i%2]
index = np.arange(len(models_results_before))
bar_width = 0.35

before_values = [result[metric] for result in models_results_before]
after_values = [result[metric] for result in models_results_after]
colors = [result['Color'] for result in models_results_before]

bars1 = ax.bar(index, before_values, bar_width, color=colors,
alpha=0.6)
bars2 = ax.bar(index + bar_width, after_values, bar_width,
color=colors, alpha=1.0)

ax.set_xlabel('Models')
ax.set_ylabel(metric)
ax.set_title(f'{metric} Comparison')
ax.set_xticks(index + bar_width / 2)
ax.set_xticklabels([result['Model'] for result in
models_results_before])

for bar in bars1:
yval = bar.get_height()
ax.text(bar.get_x() + bar.get_width() / 2, yval / 2, f'{yval:.2f}',
ha='center', va='center', color='white')

for bar in bars2:
yval = bar.get_height()
ax.text(bar.get_x() + bar.get_width() / 2, yval / 2, f'{yval:.2f}',
ha='center', va='center', color='white')

fig.tight_layout()

# Add single legend for all subplots
```

```
handles = [plt.Rectangle((0,0),1,1, color='gray', alpha=0.6,
label='Before Tuning'),
plt.Rectangle((0,0),1,1, color='gray', alpha=1.0, label='After Tuning')]
fig.legend(handles=handles, loc='upper center',
bbox_to_anchor=(0.5, 1.05), ncol=2)
plt.show()

# Plotting TP, TN, FP, FN side by side with labels
fig, axes = plt.subplots(1, 4, figsize=(18, 6))
metrics_names = ['TN', 'FN', 'FP', 'TP']

# Confusion matrix plotting function
def plot_conf_matrix(metric_name, i, ax):
index = np.arange(len(models_results_before))
bar_width = 0.35

before_values = [result['Confusion Matrix'].ravel()[i] for result in
models_results_before]
after_values = [result['Confusion Matrix'].ravel()[i] for result in
models_results_after]
colors = [result['Color'] for result in models_results_before]

bars1 = ax.bar(index, before_values, bar_width, color=colors,
alpha=0.6)
bars2 = ax.bar(index + bar_width, after_values, bar_width,
color=colors, alpha=1.0)

ax.set_xlabel('Models')
ax.set_ylabel('Counts')
ax.set_title(f'{metric_name} Comparison')
ax.set_xticks(index + bar_width / 2)
ax.set_xticklabels([result['Model'] for result in
models_results_before])

for bar in bars1:
yval = bar.get_height()
ax.text(bar.get_x() + bar.get_width() / 2, yval / 2, f'{yval:.0f}',
ha='center', va='center', color='white')

for bar in bars2:
yval = bar.get_height()
ax.text(bar.get_x() + bar.get_width() / 2, yval / 2, f'{yval:.0f}',
ha='center', va='center', color='white')

for i, metric_name in enumerate(metrics_names):
plot_conf_matrix(metric_name, i, axes[i])

fig.tight_layout()
plt.show()

# Plot ROC Curves
plt.figure(figsize=(10, 8))
for fpr, tpr, roc_auc, model_name, color in roc_curves_before:
plt.plot(fpr, tpr, lw=2, linestyle='--', color=color, label=f'{model_name}
Before (AUC = {roc_auc:.2f})')

for fpr, tpr, roc_auc, model_name, color in roc_curves_after:
plt.plot(fpr, tpr, lw=2, color=color, label=f'{model_name} After (AUC
= {roc_auc:.2f})')

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
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')

# Create a single legend outside the plot for ROC Curves
plt.legend(loc='lower right')
plt.tight_layout()
plt.show()

# Function to calculate loss cost from confusion matrix
def calculate_loss_cost(conf_matrix):
# Define the costs for each type of misclassification
cost_fp = 1000 # Cost of False Positive
cost_fn = 5000 # Cost of False Negative

# Extract values from confusion matrix
FP = conf_matrix[0, 1]
FN = conf_matrix[1, 0]

# Calculate total loss cost
total_cost = FP * cost_fp + FN * cost_fn
return total_cost



# Extracting model names and corresponding colors
model_names = [result['Model'] for result in models_results_before]
colors = [result['Color'] for result in models_results_before]

# Loss costs before and after tuning
loss_costs_before = [calculate_loss_cost(result['Confusion Matrix'])
for result in models_results_before]
loss_costs_after = [calculate_loss_cost(result['Confusion Matrix']) for
result in models_results_after]

# Setting up the figure
fig, ax = plt.subplots(figsize=(10, 6))
bar_width = 0.35
index = np.arange(len(model_names))

bars1 = ax.bar(index - bar_width/2, loss_costs_before, bar_width,
color=colors, alpha=0.6, label='Before Tuning')
bars2 = ax.bar(index + bar_width/2, loss_costs_after, bar_width,
color=colors, alpha=1.0, label='After Tuning')

ax.set_xlabel('Models')
ax.set_ylabel('Loss Cost')
ax.set_title('Loss Cost Comparison Before and After Tuning')
ax.set_xticks(index)
ax.set_xticklabels(model_names)
ax.legend()

# Adding text labels for values on top of bars
def autolabel(bars):
```

		for bar in bars:
		yval = bar.get_height()
		ax.text(bar.get_x() + bar.get_width()/2, yval / 2, f'{yval:.0f}', ha='center', va='center', color='white')
		
		autolabel(bars1)
		autolabel(bars2)
		
		plt.tight_layout()
		plt.show()

Learning curve	from sklearn.model_selection import cross_val_score, learning_curve
	import matplotlib.pyplot as plt
		kf = KFold(n_splits=5, shuffle=True, random_state=42)
		
		# Cross-validation scores
		def cross_val_evaluation(model, X, y, cv=kf):
		accuracy = cross_val_score(model, X, y, cv=cv, scoring='accuracy')
		precision = cross_val_score(model, X, y, cv=cv, scoring='precision')
		recall = cross_val_score(model, X, y, cv=cv, scoring='recall')
		f1 = cross_val_score(model, X, y, cv=cv, scoring='f1')
		return accuracy, precision, recall, f1
		
		# Learning curves
		def plot_learning_curve(estimator, title, X, y, cv=kf):
		plt.figure()
		plt.title(title)
		plt.xlabel("Training examples")
		plt.ylabel("Score")
		train_sizes, train_scores, test_scores = learning_curve(estimator, X, y, cv=cv, n_jobs=-1)
		train_scores_mean = np.mean(train_scores, axis=1)
		test_scores_mean = np.mean(test_scores, axis=1)
		plt.grid()
		
		plt.plot(train_sizes, train_scores_mean, 'o-', color="r", label="Training score")
		plt.plot(train_sizes, test_scores_mean, 'o-', color="g", label="Cross-validation score")
		
		plt.legend(loc="best")
		plt.show()
		
		# usage for cross-validation
		models = [
		(SVC(probability=True), X_train_res_scaled, y_train_res),
		(GaussianNB(), X_train_res, y_train_res),
		(RandomForestClassifier(n_estimators=100, random_state=42), X_train_res, y_train_res)
		]
		
		for model, X, y in models:
		acc, prec, rec, f1 = cross_val_evaluation(model, X, y)
		print(f"{model.__class__.__name__} - Accuracy: {np.mean(acc):.2f}, Precision: {np.mean(prec):.2f}, Recall: {np.mean(rec):.2f}, F1 Score: {np.mean(f1):.2f}")
		
		# Example usage for learning curves
		plot_learning_curve(SVC(probability=True), "Learning Curve (SVM)", X_train_res_scaled, y_train_res)

		plot_learning_curve(GaussianNB(), "Learning Curve (Naive Bayes)", X_train_res, y_train_res)
		plot_learning_curve(RandomForestClassifier(n_estimators=100, random_state=42), "Learning Curve (Random Forest)", X_train_res, y_train_res)

6.4 Feature Engineering

Modeling and Evaluation		
Purpose	**Library**	**Query**

Feature Engineering	import pandas as pd
	import numpy as np
		# Assuming 'merged_df' is your DataFrame and 'feature592' is the column with date-time values
		
		# Convert the 'feature592' column to datetime format
		merged_df['feature592'] = pd.to_datetime(merged_df['feature592'], format='%d/%m/%Y %H:%M:%S')
		
		# Extract date and time components
		merged_df['year'] = merged_df['feature592'].dt.year
		merged_df['month'] = merged_df['feature592'].dt.month
		merged_df['day'] = merged_df['feature592'].dt.day
		merged_df['hour'] = merged_df['feature592'].dt.hour
		merged_df['minute'] = merged_df['feature592'].dt.minute
		merged_df['second'] = merged_df['feature592'].dt.second
		
		# Day of the week (0=Monday, 6=Sunday)
		merged_df['day_of_week'] = merged_df['feature592'].dt.dayofweek
		
		# Part of the day (morning, afternoon, evening, night)
		def part_of_day(hour):
		if 5 <= hour < 12:
		return 1 # morning
		elif 12 <= hour < 17:
		return 2 # afternoon
		elif 17 <= hour < 21:
		return 3 # evening
		else:
		return 0 # night
		
		merged_df['part_of_day'] = merged_df['hour'].apply(part_of_day)
		
		# Elapsed time since first entry in minutes
		merged_df['elapsed_time'] = (merged_df['feature592'] - merged_df['feature592'].min()).dt.total_seconds() / 60
		
		# Seasonal features
		merged_df['quarter'] = merged_df['feature592'].dt.quarter
		
		# Flag for weekend
		merged_df['is_weekend'] = (merged_df['day_of_week'] >= 5).astype(int)
		
		# Time differences between consecutive entries in minutes
		merged_df['time_diff'] = merged_df['feature592'].diff().dt.total_seconds() / 60
		merged_df['time_diff'].fillna(0, inplace=True)

		# Display the DataFrame with new features
		print(merged_df.head())
		
		# Drop the original datetime column if not needed
		merged_df.drop('feature592', axis=1, inplace=True)

6.5 Building models after Feature Engineering and Evaluation

Modeling and Evaluation		
Purpose	**Library**	**Query**

Before tuning-Model 1 (RF with General Boruta)	import pandas as pd
	import numpy as np
	from boruta import BorutaPy
	from sklearn.ensemble import RandomForestClassifier
	from imblearn.over_sampling import SMOTE
	from sklearn.preprocessing import MinMaxScaler
	from sklearn.model_selection import train_test_split
	from sklearn.metrics import confusion_matrix, accuracy_score, f1_score, precision_score, recall_score
		# Assuming 'merged_df' is your DataFrame with the relevant data
		X = merged_df.drop('feature591', axis=1) # Replace 'feature591' with your actual target column name
		y = merged_df['feature591']
		
		# Split data into training and testing sets
		X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
		
		# Boruta Feature Selection
		rf_clf = RandomForestClassifier(n_jobs=-1, class_weight='balanced', max_depth=5, random_state=42)
		boruta_selector = BorutaPy(rf_clf, n_estimators='auto', verbose=0, random_state=42) # Set verbose to 0
		boruta_selector.fit(X_train.values, y_train)
		
		# Selected features
		selected_features = X_train.columns[boruta_selector.support_].tolist()
		print(""Selected Features: "", selected_features)
		
		# Update X_train and X_test with selected features
		X_train_selected = X_train[selected_features]
		X_test_selected = X_test[selected_features]
		
		# SMOTE Balancing on training data
		smote = SMOTE(random_state=42)
		X_train_res, y_train_res = smote.fit_resample(X_train_selected, y_train)
		
		# Min-Max Scaling
		scaler = MinMaxScaler()
		X_train_scaled = scaler.fit_transform(X_train_res)
		X_test_scaled = scaler.transform(X_test_selected)
		
		# Train Random Forest Classifier
		rf_clf.fit(X_train_scaled, y_train_res)
		y_pred = rf_clf.predict(X_test_scaled)
		
		# Confusion Matrix
		conf_matrix = confusion_matrix(y_test, y_pred)
		tn, fp, fn, tp = conf_matrix.ravel()
		
		# Calculate Metrics

		accuracy = accuracy_score(y_test, y_pred)
		f1 = f1_score(y_test, y_pred, average='weighted')
		precision = precision_score(y_test, y_pred, average='weighted')
		recall = recall_score(y_test, y_pred, average='weighted')
		specificity = tn / (tn + fp)
		
		# Calculate Loss Cost
		cost_fp = 1000
		cost_fn = 5000
		loss_cost = cost_fp * fp + cost_fn * fn
		
		print(""Accuracy Score: "", accuracy)
		print(""F1 Score: "", f1)
		print(""Precision: "", precision)
		print(""Recall (Sensitivity): "", recall)
		print(""Specificity: "", specificity)
		print(""Loss Cost: "", loss_cost)
		
		print(""Confusion Matrix:"")
		print(conf_matrix)
		print(f""True Positives (TP): {tp}"")
		print(f""True Negatives (TN): {tn}"")
		print(f""False Positives (FP): {fp}"")
		print(f""False Negatives (FN): {fn}"")

Before tuning-Model 2 (RF selected features)	import pandas as pd
	import numpy as np
	from sklearn.model_selection import train_test_split
	from sklearn.preprocessing import MinMaxScaler
	from sklearn.ensemble import RandomForestClassifier
	from sklearn.neighbors import KNeighborsClassifier
	from sklearn.naive_bayes import GaussianNB
	from sklearn.tree import DecisionTreeClassifier
	from sklearn.linear_model import LogisticRegression
	from sklearn.svm import SVC
	from imblearn.over_sampling import SMOTE
	from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score, confusion_matrix
	import matplotlib.pyplot as plt
		# Assuming 'merged_df' is your DataFrame with the selected features and target
		selected_features = ['feature17', 'feature41', 'feature60', 'feature66', 'feature342', 'feature427', 'feature442', 'feature563', 'elapsed_time']
		X = merged_df[selected_features]
		y = merged_df['feature591'] # Replace 'feature591' with your actual target column name
		
		# Min-Max scaling
		scaler = MinMaxScaler()
		X_train_scaled = scaler.fit_transform(X_train)
		X_test_scaled = scaler.transform(X_test)
		
		# SMOTE Data Balancing
		smote = SMOTE(random_state=42)
		X_train_res, y_train_res = smote.fit_resample(X_train_scaled, y_train)
		
		# Define classifiers
		classifiers = {
		'Random Forest': RandomForestClassifier(random_state=42),
		'KNN': KNeighborsClassifier(),
		'Naive Bayes': GaussianNB(),
		'Decision Tree': DecisionTreeClassifier(random_state=42),
		'Logistic Regression': LogisticRegression(random_state=42),
		'SVM': SVC(probability=True, random_state=42)
		}
		
		# Dictionary to store evaluation metrics
		metrics = {
		'Accuracy': [],
		'Precision': [],
		'Recall': [],

```
'F1 Score': [],
'ROC AUC': [],
'Loss Cost': [],
'Confusion Matrix': []
}

# Iterate over classifiers
for clf_name, clf in classifiers.items():
# Train the model
clf.fit(X_train_res, y_train_res)

# Predictions
y_pred = clf.predict(X_test_scaled)
y_pred_prob = clf.predict_proba(X_test_scaled)[:, 1]

# Evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
roc_auc = roc_auc_score(y_test, y_pred_prob)
conf_matrix = confusion_matrix(y_test, y_pred)

# Calculate loss cost
cost_fp = 1000
cost_fn = 5000
FP = conf_matrix[0, 1]
FN = conf_matrix[1, 0]
loss_cost = cost_fp * FP + cost_fn * FN

# Store metrics
metrics['Accuracy'].append(accuracy)
metrics['Precision'].append(precision)
metrics['Recall'].append(recall)
metrics['F1 Score'].append(f1)
metrics['ROC AUC'].append(roc_auc)
metrics['Loss Cost'].append(loss_cost)
metrics['Confusion Matrix'].append(conf_matrix)

# Print results
print(f"Classifier: {clf_name}")
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")
print(f"ROC AUC: {roc_auc}")
print(f"Confusion Matrix:\n{conf_matrix}")
print(f"Loss Cost: {loss_cost}")
print("\n")

# Plotting ROC curves
plt.figure(figsize=(8, 6))
for clf_name, clf in classifiers.items():
y_pred_prob = clf.predict_proba(X_test_scaled)[:, 1]
fpr, tpr, _ = roc_curve(y_test, y_pred_prob)
roc_auc = auc(fpr, tpr)
plt.plot(fpr, tpr, label=f'{clf_name} (AUC = {roc_auc:.2f})')
```

		plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
		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')
		plt.legend(loc="lower right")
		plt.show()

Before tuning- Model 3 (Gaussian NB)	import pandas as pd
	import numpy as np
	from boruta import BorutaPy
	from sklearn.ensemble import RandomForestClassifier
	from imblearn.over_sampling import SMOTE
	from sklearn.preprocessing import MinMaxScaler
	from sklearn.model_selection import train_test_split
	from sklearn.metrics import confusion_matrix, accuracy_score, f1_score, precision_score, recall_score
	from sklearn.naive_bayes import GaussianNB
		# Assuming X_train_imputed, y_train, X_test_imputed, y_test are already defined
		X_train = X_train_imputed[['feature17', 'feature41', 'feature60', 'feature66', 'feature342', 'feature427', 'feature442', 'feature563', 'elapsed_time']].copy()
		y_train = y_train.copy()
		X_test = X_test_imputed[['feature17', 'feature41', 'feature60', 'feature66', 'feature342', 'feature427', 'feature442', 'feature563', 'elapsed_time']].copy()
		y_test = y_test.copy()
		
		# Update X_train and X_test with selected features
		X_train_selected = X_train[selected_features]
		X_test_selected = X_test[selected_features]
		
		# SMOTE Balancing on training data
		smote = SMOTE(random_state=42)
		X_train_res, y_train_res = smote.fit_resample(X_train_selected, y_train)
		
		# Min-Max Scaling
		scaler = MinMaxScaler()
		X_train_scaled = scaler.fit_transform(X_train_res)
		X_test_scaled = scaler.transform(X_test_selected)
		
		# Define parameter grid for GaussianNB
		param_grid = {
		'var_smoothing': np.logspace(0,-9, num=100)
		}
		
		# Perform parameter tuning
		best_f1_score = -1
		best_params = None
		
		for var_smoothing in param_grid['var_smoothing']:
		# Train GaussianNB with current parameters
		nb_clf = GaussianNB(var_smoothing=var_smoothing)
		nb_clf.fit(X_train_scaled, y_train_res)
		
		# Predict on test data
		y_pred = nb_clf.predict(X_test_scaled)
		
		# Calculate F1 score
		f1 = f1_score(y_test, y_pred, average='weighted')
		
		# Check if current model is better than previous ones
		if f1 > best_f1_score:
		best_f1_score = f1
		best_params = {
		'var_smoothing': var_smoothing
		}

		# Train final GaussianNB with best parameters
		best_nb_clf =
		GaussianNB(var_smoothing=best_params['var_smoothing'])
		best_nb_clf.fit(X_train_scaled, y_train_res)
		
		# Predict on test data with the best model
		y_pred = best_nb_clf.predict(X_test_scaled)
		
		# Confusion Matrix
		conf_matrix = confusion_matrix(y_test, y_pred)
		tn, fp, fn, tp = conf_matrix.ravel()
		
		# Calculate Metrics
		accuracy = accuracy_score(y_test, y_pred)
		f1 = f1_score(y_test, y_pred, average='weighted')
		precision = precision_score(y_test, y_pred, average='weighted')
		recall = recall_score(y_test, y_pred, average='weighted')
		specificity = tn / (tn + fp)
		
		# Calculate Loss Cost
		cost_fp = 1000
		cost_fn = 5000
		loss_cost = cost_fp * fp + cost_fn * fn
		
		print("Best Parameters:", best_params)
		print("Best F1 Weighted Score:", best_f1_score)
		
		print("Accuracy Score: ", accuracy)
		print("F1 Score: ", f1)
		print("Precision: ", precision)
		print("Recall (Sensitivity): ", recall)
		print("Specificity: ", specificity)
		print("Loss Cost: ", loss_cost)
		
		print("Confusion Matrix:")
		print(conf_matrix)
		print(f"True Positives (TP): {tp}")
		print(f"True Negatives (TN): {tn}")
		print(f"False Positives (FP): {fp}")
		print(f"False Negatives (FN): {fn}")

After tuning-Model 1	import pandas as pd
	import numpy as np
	from boruta import BorutaPy
	from sklearn.ensemble import RandomForestClassifier
	from imblearn.over_sampling import SMOTE
	from sklearn.preprocessing import MinMaxScaler
	from sklearn.model_selection import train_test_split
	from sklearn.metrics import confusion_matrix, accuracy_score, f1_score,
		# Assuming 'merged_df' is your DataFrame with the relevant data
		X = merged_df.drop('feature591', axis=1) # Replace 'feature591' with your actual target column name
		y = merged_df['feature591']
		
		# Split data into training and testing sets
		X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
		
		# Boruta Feature Selection
		rf_clf = RandomForestClassifier(n_jobs=-1, class_weight='balanced', max_depth=5, random_state=42)
		boruta_selector = BorutaPy(rf_clf, n_estimators='auto', verbose=0, random_state=42) # Set verbose to 0
		boruta_selector.fit(X_train.values, y_train)

	precision_score, recall_score
	from sklearn.naive_bayes
	import GaussianNB
		
		# Selected features
		selected_features = X_train.columns[boruta_selector.support_].tolist()
		print(""Selected Features: "", selected_features)
		
		# Update X_train and X_test with selected features
		X_train_selected = X_train[selected_features]
		X_test_selected = X_test[selected_features]
		
		# SMOTE Balancing on training data
		smote = SMOTE(random_state=42)
		X_train_res, y_train_res = smote.fit_resample(X_train_selected, y_train)
		
		# Min-Max Scaling
		scaler = MinMaxScaler()
		X_train_scaled = scaler.fit_transform(X_train_res)
		X_test_scaled = scaler.transform(X_test_selected)
		
		# Define parameter grid for RandomForestClassifier
		param_grid = {
		'n_estimators': [50, 100, 200, 300],
		'max_depth': [3, 5, 7, 10, 15],
		'min_samples_split': [2, 5, 10, 15],
		'min_samples_leaf': [1, 2, 4, 6]
		}
		
		# Perform hyperparameter tuning without cross-validation
		best_f1_score = -1
		best_params = None
		
		for n_estimators in param_grid['n_estimators']:
		for max_depth in param_grid['max_depth']:
		for min_samples_split in param_grid['min_samples_split']:
		for min_samples_leaf in param_grid['min_samples_leaf']:
		# Train RandomForestClassifier with current parameters
		rf_clf = RandomForestClassifier(n_estimators=n_estimators,
		max_depth=max_depth,
		min_samples_split=min_samples_split,
		min_samples_leaf=min_samples_leaf,
		n_jobs=-1,
		class_weight='balanced',
		random_state=42)
		
		rf_clf.fit(X_train_scaled, y_train_res)
		
		# Predict on test data
		y_pred = rf_clf.predict(X_test_scaled)
		
		# Calculate F1 score
		f1 = f1_score(y_test, y_pred, average='weighted')
		
		# Check if current model is better than previous ones
		if f1 > best_f1_score:
		best_f1_score = f1
		best_params = {
		'n_estimators': n_estimators,
		'max_depth': max_depth,
		'min_samples_split': min_samples_split,

		'min_samples_leaf': min_samples_leaf
		}
		
		# Train final RandomForestClassifier with best parameters
		best_rf_clf =
		RandomForestClassifier(n_estimators=best_params['n_estimators'],
		max_depth=best_params['max_depth'],
		min_samples_split=best_params['min_samples_split'],
		min_samples_leaf=best_params['min_samples_leaf'],
		n_jobs=-1,
		class_weight='balanced',
		random_state=42)
		
		best_rf_clf.fit(X_train_scaled, y_train_res)
		
		# Predict on test data with the best model
		y_pred = best_rf_clf.predict(X_test_scaled)
		
		# Confusion Matrix
		conf_matrix = confusion_matrix(y_test, y_pred)
		tn, fp, fn, tp = conf_matrix.ravel()
		
		# Calculate Metrics
		accuracy = accuracy_score(y_test, y_pred)
		f1 = f1_score(y_test, y_pred, average='weighted')
		precision = precision_score(y_test, y_pred, average='weighted')
		recall = recall_score(y_test, y_pred, average='weighted')
		specificity = tn / (tn + fp)
		
		# Calculate Loss Cost
		cost_fp = 1000
		cost_fn = 5000
		loss_cost = cost_fp * fp + cost_fn * fn
		
		print(""Best Parameters:"", best_params)
		print(""Best F1 Weighted Score:"", best_f1_score)
		
		print(""Accuracy Score: "", accuracy)
		print(""F1 Score: "", f1)
		print(""Precision: "", precision)
		print(""Recall (Sensitivity): "", recall)
		print(""Specificity: "", specificity)
		print(""Loss Cost: "", loss_cost)
		
		print(""Confusion Matrix:"")
		print(conf_matrix)
		print(f""True Positives (TP): {tp}"")
		print(f""True Negatives (TN): {tn}"")
		print(f""False Positives (FP): {fp}"")
		print(f""False Negatives (FN): {fn}"")

After tuning-Model 2	import pandas as pd
	import numpy as np
	from sklearn.ensemble import RandomForestClassifier
	from imblearn.over_sampling import SMOTE
	from sklearn.preprocessing
		# Define fixed set of features
		fixed_features = ['feature17', 'feature41', 'feature60', 'feature66', 'feature342', 'feature427', 'feature442', 'feature563', 'elapsed_time']
		
		# Select only the fixed features
		X_train_selected = X_train[fixed_features]
		X_test_selected = X_test[fixed_features]

	import MinMaxScaler
	from sklearn.metrics import confusion_matrix, accuracy_score, f1_score, precision_score, recall_score
		# SMOTE Balancing on training data
		smote = SMOTE(random_state=42)
		X_train_res, y_train_res = smote.fit_resample(X_train_selected, y_train)
		
		# Min-Max Scaling
		scaler = MinMaxScaler()
		X_train_scaled = scaler.fit_transform(X_train_res)
		X_test_scaled = scaler.transform(X_test_selected)
		
		# Define parameter grid for RandomForestClassifier
		param_grid = {
		'n_estimators': [50, 100, 200, 300],
		'max_depth': [3, 5, 7, 10, 15],
		'min_samples_split': [2, 5, 10, 15],
		'min_samples_leaf': [1, 2, 4, 6]
		}
		
		# Perform hyperparameter tuning without cross-validation
		best_f1_score = -1
		best_params = None
		
		for n_estimators in param_grid['n_estimators']:
		for max_depth in param_grid['max_depth']:
		for min_samples_split in param_grid['min_samples_split']:
		for min_samples_leaf in param_grid['min_samples_leaf']:
		# Train RandomForestClassifier with current parameters
		rf_clf = RandomForestClassifier(n_estimators=n_estimators,
		max_depth=max_depth,
		min_samples_split=min_samples_split,
		min_samples_leaf=min_samples_leaf,
		n_jobs=-1,
		class_weight='balanced',
		random_state=42)
		
		rf_clf.fit(X_train_scaled, y_train_res)
		
		# Predict on test data
		y_pred = rf_clf.predict(X_test_scaled)
		
		# Calculate F1 score
		f1 = f1_score(y_test, y_pred, average='weighted')
		
		# Check if current model is better than previous ones
		if f1 > best_f1_score:
		best_f1_score = f1
		best_params = {
		'n_estimators': n_estimators,
		'max_depth': max_depth,
		'min_samples_split': min_samples_split,
		'min_samples_leaf': min_samples_leaf
		}
		
		# Train final RandomForestClassifier with best parameters
		best_rf_clf =
		RandomForestClassifier(n_estimators=best_params['n_estimators'],
		max_depth=best_params['max_depth'],
		min_samples_split=best_params['min_samples_split'],
		min_samples_leaf=best_params['min_samples_leaf'],

		n_jobs=-1,
		class_weight='balanced',
		random_state=42)
		
		best_rf_clf.fit(X_train_scaled, y_train_res)
		
		# Predict on test data with the best model
		y_pred = best_rf_clf.predict(X_test_scaled)
		
		# Confusion Matrix
		conf_matrix = confusion_matrix(y_test, y_pred)
		tn, fp, fn, tp = conf_matrix.ravel()
		
		# Calculate Metrics
		accuracy = accuracy_score(y_test, y_pred)
		f1 = f1_score(y_test, y_pred, average='weighted')
		precision = precision_score(y_test, y_pred, average='weighted')
		recall = recall_score(y_test, y_pred, average='weighted')
		specificity = tn / (tn + fp)
		
		# Calculate Loss Cost
		cost_fp = 1000
		cost_fn = 5000
		loss_cost = cost_fp * fp + cost_fn * fn
		
		print("Best Parameters:", best_params)
		print("Best F1 Weighted Score:", best_f1_score)
		
		print("Accuracy Score: ", accuracy)
		print("F1 Score: ", f1)
		print("Precision: ", precision)
		print("Recall (Sensitivity): ", recall)
		print("Specificity: ", specificity)
		print("Loss Cost: ", loss_cost)
		
		print("Confusion Matrix:")
		print(conf_matrix)
		print(f"True Positives (TP): {tp}")
		print(f"True Negatives (TN): {tn}")
		print(f"False Positives (FP): {fp}")
		print(f"False Negatives (FN): {fn}")

After tuning- Model 3	import pandas as pd
	import numpy as np
	from boruta import BorutaPy
	from sklearn.ensemble import RandomForestClassifier
	from imblearn.over_sampling import SMOTE
	from sklearn.preprocessing import MinMaxScaler
	from sklearn.model_selection import train_test_split
	from sklearn.metrics import confusion_matrix, accuracy_score, f1_score, precision_score, recall_score
	from sklearn.naive_bayes
		# Selecting specific features identified by Boruta
		selected_features = ['feature60', 'feature65', 'feature66', 'feature342', 'feature351', 'feature478', 'feature540', 'feature563']
		
		# Update X_train and X_test with selected features
		X_train_selected = X_train[selected_features]
		X_test_selected = X_test[selected_features]
		
		# SMOTE Balancing on training data
		smote = SMOTE(random_state=42)
		X_train_res, y_train_res = smote.fit_resample(X_train_selected, y_train)
		
		# Min-Max Scaling
		scaler = MinMaxScaler()
		X_train_scaled = scaler.fit_transform(X_train_res)
		X_test_scaled = scaler.transform(X_test_selected)

	import GaussianNB	
		# Define parameter grid for GaussianNB
		param_grid = {
		'var_smoothing': np.logspace(0,-9, num=100)
		}
		
		# Perform parameter tuning
		best_f1_score = -1
		best_params = None
		
		for var_smoothing in param_grid['var_smoothing']:
		# Train GaussianNB with current parameters
		nb_clf = GaussianNB(var_smoothing=var_smoothing)
		nb_clf.fit(X_train_scaled, y_train_res)
		
		# Predict on test data
		y_pred = nb_clf.predict(X_test_scaled)
		
		# Calculate F1 score
		f1 = f1_score(y_test, y_pred, average='weighted')
		
		# Check if current model is better than previous ones
		if f1 > best_f1_score:
		best_f1_score = f1
		best_params = {
		'var_smoothing': var_smoothing
		}
		
		# Train final GaussianNB with best parameters
		best_nb_clf =
		GaussianNB(var_smoothing=best_params['var_smoothing'])
		best_nb_clf.fit(X_train_scaled, y_train_res)
		
		# Predict on test data with the best model
		y_pred = best_nb_clf.predict(X_test_scaled)
		
		# Confusion Matrix
		conf_matrix = confusion_matrix(y_test, y_pred)
		tn, fp, fn, tp = conf_matrix.ravel()
		
		# Calculate Metrics
		accuracy = accuracy_score(y_test, y_pred)
		f1 = f1_score(y_test, y_pred, average='weighted')
		precision = precision_score(y_test, y_pred, average='weighted')
		recall = recall_score(y_test, y_pred, average='weighted')
		specificity = tn / (tn + fp)
		
		# Calculate Loss Cost
		cost_fp = 1000
		cost_fn = 5000
		loss_cost = cost_fp * fp + cost_fn * fn
		
		print("Best Parameters:", best_params)
		print("Best F1 Weighted Score:", best_f1_score)
		
		print("Accuracy Score: ", accuracy)
		print("F1 Score: ", f1)
		print("Precision: ", precision)

		print("Recall (Sensitivity): ", recall)
		print("Specificity: ", specificity)
		print("Loss Cost: ", loss_cost)
		print("Confusion Matrix:")
		print(conf_matrix)
		print(f"True Positives (TP): {tp}")
		print(f"True Negatives (TN): {tn}")
		print(f"False Positives (FP): {fp}")
		print(f"False Negatives (FN): {fn}")
Graphs-precision, f1, recall, lost... 3 models	import numpy as np	# Data for three models before and after tuning
	import matplotlib.pyplot as plt	models = ['Model 1', 'Model 2', 'Model 3']
		metrics = ['Accuracy', 'F1 Score', 'Precision', 'Recall (Sensitivity)', 'Loss Cost']
		metrics_values_before = np.array([
		[0.8086734694, 0.8494935529, 0.9134543088, 0.8086734694, 119000],
		[0.9107142857, 0.2553191489, 0.2857142857, 0.2307692308, 115000],
		[0.5816326531, 0.6856787933, 0.8740304301, 0.5816326531, 228000]
		])
		metrics_values_after = np.array([
		[0.8826530612, 0.8926764456, 0.904544958, 0.8826530612, 114000],
		[0.9285714286, 0.9184331797, 0.9126984127, 0.9285714286, 108000],
		[0.5816326531, 0.6856787933, 0.8740304301, 0.5816326531, 228000]
		])
		confusion_matrix_before = np.array([
		[302, 64, 11, 15],
		[274, 19, 19, 2],
		[218, 148, 16, 10]
		])
		confusion_matrix_after = np.array([
		[337, 29, 17, 9],
		[358, 8, 20, 6],
		[218, 148, 16, 10]
		])
		# Define colors for models
		colors = {
		'Model 1': {'Before Tuning': 'lightgreen', 'After Tuning': 'green'},
		'Model 2': {'Before Tuning': 'mediumseagreen', 'After Tuning': 'seagreen'}, # Using mediumseagreen for Model 2 before tuning
		'Model 3': {'Before Tuning': 'lightcoral', 'After Tuning': 'red'}
		}
		# Function to plot side-by-side comparison of metrics
		def plot_metrics_comparison(metrics, metrics_values_before, metrics_values_after):
		num_metrics = len(metrics)
		fig, axes = plt.subplots(2, 3, figsize=(18, 10))
		for i in range(num_metrics):
		row = i // 3
		col = i % 3
		ax = axes[row, col]
		index = np.arange(len(models))
		bar_width = 0.35

```
before_vals = metrics_values_before[:, i]
after_vals = metrics_values_after[:, i]

for j, model in enumerate(models):
ax.bar(index[j] - bar_width/2, before_vals[j], bar_width,
label='Before Tuning', color=colors[model]['Before Tuning'], alpha=0.6)
ax.bar(index[j] + bar_width/2, after_vals[j], bar_width,
label='After Tuning', color=colors[model]['After Tuning'], alpha=1.0)

if metrics[i] == 'Loss Cost': # Adjusting format for Loss Cost metric
ax.text(index[j] - bar_width/2, before_vals[j] + 5000, f'{before_vals[j]:,.0f}',
ha='center', va='bottom')
ax.text(index[j] + bar_width/2, after_vals[j] + 5000, f'{after_vals[j]:,.0f}',
ha='center', va='bottom')
else:
ax.text(index[j] - bar_width/2, before_vals[j] + 0.0005,
f'{before_vals[j]:.3f}', ha='center', va='bottom')
ax.text(index[j] + bar_width/2, after_vals[j] + 0.0005, f'{after_vals[j]:.3f}',
ha='center', va='bottom')

ax.set_xlabel('Models')
ax.set_ylabel(metrics[i])
ax.set_title(f'{metrics[i]} Comparison')
ax.set_xticks(index)
ax.set_xticklabels(models)

# Add legend outside the plot with adjusted parameters
handles = [
plt.Rectangle((0,0),1,1, color='lightgreen', alpha=0.6),
plt.Rectangle((0,0),1,1, color='green'),
plt.Rectangle((0,0),1,1, color='mediumseagreen'), # Using
mediumseagreen for Model 2 before tuning
plt.Rectangle((0,0),1,1, color='seagreen'),
plt.Rectangle((0,0),1,1, color='lightcoral', alpha=0.6),
plt.Rectangle((0,0),1,1, color='red')
]
labels = [
'Model 1 - Before Tuning', 'Model 1 - After Tuning',
'Model 2 - Before Tuning', 'Model 2 - After Tuning',
'Model 3 - Before Tuning', 'Model 3 - After Tuning'
]
fig.legend(handles=handles, labels=labels, loc='upper center',
bbox_to_anchor=(0.5, 1.15), ncol=3, fontsize='large') # Adjusted
bbox_to_anchor and fontsize

# Adjust subplot spacing
plt.subplots_adjust(bottom=0.15) # Increase bottom padding for the
legend

plt.tight_layout()
plt.show()


# Function to plot side-by-side comparison of confusion matrix
components
def plot_confusion_matrix_comparison(confusion_matrix_before,
confusion_matrix_after):
```

```
metrics_names = ['TN', 'FP', 'FN', 'TP']
num_metrics = len(metrics_names)
fig, axes = plt.subplots(1, num_metrics, figsize=(18, 6))

for i, metric_name in enumerate(metrics_names):
ax = axes[i]

index = np.arange(len(models))
bar_width = 0.35

before_vals = confusion_matrix_before[:, i]
after_vals = confusion_matrix_after[:, i]

for j, model in enumerate(models):
ax.bar(index[j] - bar_width/2, before_vals[j], bar_width,
label='Before Tuning', color=colors[model]['Before Tuning'], alpha=0.6)
ax.bar(index[j] + bar_width/2, after_vals[j], bar_width,
label='After Tuning', color=colors[model]['After Tuning'], alpha=1.0)

ax.text(index[j] - bar_width/2, before_vals[j] + 0.0005,
f'{before_vals[j]:.0f}', ha='center', va='bottom')
ax.text(index[j] + bar_width/2, after_vals[j] + 0.0005, f'{after_vals[j]:.0f}',
ha='center', va='bottom')

ax.set_xlabel('Models')
ax.set_ylabel(metric_name)
ax.set_title(f'{metric_name} Comparison')
ax.set_xticks(index)
ax.set_xticklabels(models)

# Add legend outside the plot with adjusted parameters
handles = [
plt.Rectangle((0,0),1,1, color='lightgreen', alpha=0.6),
plt.Rectangle((0,0),1,1, color='green'),
plt.Rectangle((0,0),1,1, color='mediumseagreen'), # Using
mediumseagreen for Model 2 before tuning
plt.Rectangle((0,0),1,1, color='lightcoral', alpha=0.6),
plt.Rectangle((0,0),1,1, color='red')
]
labels = [
'Model 1 - Before Tuning', 'Model 1 - After Tuning',
'Model 2 - Before Tuning', 'Model 2 - After Tuning',
'Model 3 - Before Tuning', 'Model 3 - After Tuning'
]


plt.tight_layout()
plt.show()

# Plotting metrics comparison
plot_metrics_comparison(metrics, metrics_values_before[:, :5],
metrics_values_after[:, :5])

# Plotting confusion matrix components comparison
plot_confusion_matrix_comparison(confusion_matrix_before,
confusion_matrix_after)
```

Trade off 1- Rates - 3

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# Define the values for Model 1, Model 2, and Model 3 after tuning
models = ['Model 1', 'Model 2', 'Model 3']
```

models		TP = np.array([9, 6, 10])
		TN = np.array([337, 358, 218])
		FP = np.array([29, 8, 148])
		FN = np.array([17, 20, 16])
		
		# Calculate rates
		total_positives = TP + FN
		total_negatives = TN + FP
		
		TP_rate = TP / total_positives
		FP_rate = FP / total_negatives
		FN_rate = FN / total_positives
		TN_rate = TN / total_negatives
		
		# Plotting the comparison graph
		rates = ['TP Rate', 'FP Rate', 'FN Rate', 'TN Rate']
		
		# Transpose the rates for plotting
		rate_values = np.array([TP_rate, FP_rate, FN_rate, TN_rate])
		
		plt.figure(figsize=(10, 6))
		
		# Plot each line separately for each model
		for i, model in enumerate(models):
		plt.plot(rates, rate_values[:, i], marker='o', linestyle='-', label=model)
		
		# Adding labels and title
		plt.xlabel('Rates')
		plt.ylabel('Value')
		plt.title('Comparison of TP, FP, FN, TN Rates After Tuning for Model 1, Model 2, and Model 3')
		plt.xticks(rotation=45)
		plt.legend()
		
		plt.grid(True)
		plt.tight_layout()
		plt.show()

6.6 Cross Validation and Evaluation

Modeling and Evaluation		
Purpose	**Library**	**Query**

Kfold- Model 1	import pandas as pd
	import numpy as np
	from boruta import BorutaPy
	from sklearn.ensemble import RandomForestClassifier
	from imblearn.over_sampling import SMOTE
	from sklearn.preprocessing import MinMaxScaler
	from sklearn.model_selection import train_test_split, StratifiedKFold
	from sklearn.metrics import confusion_matrix, accuracy_score, f1_score, precision_score, recall_score
		# Assuming 'merged_df' is your DataFrame with the relevant data
		X = merged_df.drop('feature591', axis=1) # Replace 'feature591' with your actual target column name
		y = merged_df['feature591']
		
		# Initialize k-fold cross-validation
		kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
		
		# Initialize lists to store evaluation metrics across folds
		accuracy_scores = []
		f1_scores = []
		precision_scores = []
		recall_scores = []
		specificity_scores = []
		loss_costs = []
		
		# Iterate over each fold
		for fold_idx, (train_idx, test_idx) in enumerate(kfold.split(X, y)):
		print(f"\nFold {fold_idx + 1}:")
		
		# Split data into training and testing sets for this fold
		X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
		y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]
		
		# Boruta Feature Selection
		rf_clf = RandomForestClassifier(n_jobs=-1, class_weight='balanced', max_depth=5, random_state=42)
		boruta_selector = BorutaPy(rf_clf, n_estimators='auto', verbose=0, random_state=42) # Set verbose to 0
		boruta_selector.fit(X_train.values, y_train)
		
		# Selected features
		selected_features = X_train.columns[boruta_selector.support_].tolist()
		print("Selected Features: ", selected_features)
		
		# Update X_train and X_test with selected features
		X_train_selected = X_train[selected_features]
		X_test_selected = X_test[selected_features]
		
		# SMOTE Balancing on training data
		smote = SMOTE(random_state=42)
		X_train_res, y_train_res = smote.fit_resample(X_train_selected, y_train)
		
		# Min-Max Scaling
		scaler = MinMaxScaler()
		X_train_scaled = scaler.fit_transform(X_train_res)
		X_test_scaled = scaler.transform(X_test_selected)
		
		# Define parameter grid for RandomForestClassifier
		param_grid = {
		'n_estimators': [50, 100, 200, 300],
		'max_depth': [3, 5, 7, 10, 15],

		'min_samples_split': [2, 5, 10, 15],
		'min_samples_leaf': [1, 2, 4, 6]
		}
		
		# Perform hyperparameter tuning without cross-validation
		best_f1_score = -1
		best_params = None
		
		for n_estimators in param_grid['n_estimators']:
		for max_depth in param_grid['max_depth']:
		for min_samples_split in param_grid['min_samples_split']:
		for min_samples_leaf in param_grid['min_samples_leaf']:
		# Train RandomForestClassifier with current parameters
		rf_clf = RandomForestClassifier(n_estimators=n_estimators,
		max_depth=max_depth,
		min_samples_split=min_samples_split,
		min_samples_leaf=min_samples_leaf,
		n_jobs=-1,
		class_weight='balanced',
		random_state=42)
		
		rf_clf.fit(X_train_scaled, y_train_res)
		
		# Predict on test data
		y_pred = rf_clf.predict(X_test_scaled)
		
		# Calculate F1 score
		f1 = f1_score(y_test, y_pred, average='weighted')
		
		# Check if current model is better than previous ones
		if f1 > best_f1_score:
		best_f1_score = f1
		best_params = {
		'n_estimators': n_estimators,
		'max_depth': max_depth,
		'min_samples_split': min_samples_split,
		'min_samples_leaf': min_samples_leaf
		}
		
		# Train final RandomForestClassifier with best parameters
		best_rf_clf =
		RandomForestClassifier(n_estimators=best_params['n_estimators'],
		max_depth=best_params['max_depth'],
		min_samples_split=best_params['min_samples_split'],
		min_samples_leaf=best_params['min_samples_leaf'],
		n_jobs=-1,
		class_weight='balanced',
		random_state=42)
		
		best_rf_clf.fit(X_train_scaled, y_train_res)
		
		# Predict on test data with the best model
		y_pred = best_rf_clf.predict(X_test_scaled)
		
		# Confusion Matrix
		conf_matrix = confusion_matrix(y_test, y_pred)
		tn, fp, fn, tp = conf_matrix.ravel()

		# Calculate Metrics
		accuracy = accuracy_score(y_test, y_pred)
		f1 = f1_score(y_test, y_pred, average='weighted')
		precision = precision_score(y_test, y_pred, average='weighted')
		recall = recall_score(y_test, y_pred, average='weighted')
		specificity = tn / (tn + fp)
		
		# Calculate Loss Cost
		cost_fp = 1000
		cost_fn = 5000
		loss_cost = cost_fp * fp + cost_fn * fn
		
		# Append scores to lists
		accuracy_scores.append(accuracy)
		f1_scores.append(f1)
		precision_scores.append(precision)
		recall_scores.append(recall)
		specificity_scores.append(specificity)
		loss_costs.append(loss_cost)
		
		# Print evaluation metrics for the fold
		print("\nEvaluation Metrics for Fold:")
		print("Best Parameters:", best_params)
		print("Best F1 Weighted Score:", best_f1_score)
		print("Accuracy Score: ", accuracy)
		print("F1 Score: ", f1)
		print("Precision: ", precision)
		print("Recall (Sensitivity): ", recall)
		print("Specificity: ", specificity)
		print("Loss Cost: ", loss_cost)
		
		print("Confusion Matrix:")
		print(conf_matrix)
		print(f"True Positives (TP): {tp}")
		print(f"True Negatives (TN): {tn}")
		print(f"False Positives (FP): {fp}")
		print(f"False Negatives (FN): {fn}")
		
		# Print average scores across all folds
		print("\nAverage Metrics Across All Folds:")
		print("Average Accuracy Score: ", np.mean(accuracy_scores))
		print("Average F1 Score: ", np.mean(f1_scores))
		print("Average Precision: ", np.mean(precision_scores))
		print("Average Recall (Sensitivity): ", np.mean(recall_scores))
		print("Average Specificity: ", np.mean(specificity_scores))
		print("Average Loss Cost: ", np.mean(loss_costs))

Kfold- Model 2	import pandas as pd
	import numpy as np
	from sklearn.impute import KNNImputer
	from sklearn.ensemble import RandomForestClassifier
	from imblearn.over_sampling import SMOTE
	from sklearn.preprocessing import MinMaxScaler
		# Initialize k-fold cross-validation
		kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
		
		# Initialize lists to store evaluation metrics across folds
		accuracy_scores = []
		f1_scores = []
		precision_scores = []
		recall_scores = []
		specificity_scores = []
		loss_costs = []
		
		# Assuming X_train, y_train, X_test, y_test are already defined

	from
	sklearn.model_selection
	import StratifiedKFold
	from sklearn.metrics import
	confusion_matrix,
	accuracy_score, f1_score,
	precision_score,
	recall_score
		X_train = X_train_cleaned[['feature17', 'feature41', 'feature60',
		'feature66', 'feature342', 'feature427', 'feature442', 'feature563',
		'elapsed_time']].copy()
		y_train = y_train.copy()
		X_test = X_test_cleaned[['feature17', 'feature41', 'feature60', 'feature66',
		'feature342', 'feature427', 'feature442', 'feature563',
		'elapsed_time']].copy()
		y_test = y_test.copy()
		
		# Perform KNN imputation within each fold
		for train_index, test_index in kfold.split(X_train, y_train):
		X_train_fold, X_val_fold = X_train.iloc[train_index],
		X_train.iloc[test_index]
		y_train_fold, y_val_fold = y_train.iloc[train_index],
		y_train.iloc[test_index]
		
		# KNN imputation on training fold
		imputer = KNNImputer(n_neighbors=5)
		X_train_imputed = imputer.fit_transform(X_train_fold)
		X_val_imputed = imputer.transform(X_val_fold)
		
		# SMOTE Balancing on training data
		smote = SMOTE(random_state=42)
		X_train_res, y_train_res = smote.fit_resample(X_train_imputed,
		y_train_fold)
		
		# Min-Max Scaling
		scaler = MinMaxScaler()
		X_train_scaled = scaler.fit_transform(X_train_res)
		X_val_scaled = scaler.transform(X_val_imputed)
		
		# Define parameter grid for RandomForestClassifier
		param_grid = {
		'n_estimators': [50, 100, 200, 300],
		'max_depth': [3, 5, 7, 10, 15],
		'min_samples_split': [2, 5, 10, 15],
		'min_samples_leaf': [1, 2, 4, 6]
		}
		
		# Perform hyperparameter tuning without cross-validation
		best_f1_score = -1
		best_params = None
		
		for n_estimators in param_grid['n_estimators']:
		for max_depth in param_grid['max_depth']:
		for min_samples_split in param_grid['min_samples_split']:
		for min_samples_leaf in param_grid['min_samples_leaf']:
		# Train RandomForestClassifier with current parameters
		rf_clf = RandomForestClassifier(n_estimators=n_estimators,
		max_depth=max_depth,
		min_samples_split=min_samples_split,
		min_samples_leaf=min_samples_leaf,
		n_jobs=-1,
		class_weight='balanced',
		random_state=42)
		rf_clf.fit(X_train_scaled, y_train_res)
		
		# Predict on validation data

```
y_pred = rf_clf.predict(X_val_scaled)

# Calculate F1 score
f1 = f1_score(y_val_fold, y_pred, average='weighted')

# Check if current model is better than previous ones
if f1 > best_f1_score:
best_f1_score = f1
best_params = {
'n_estimators': n_estimators,
'max_depth': max_depth,
'min_samples_split': min_samples_split,
'min_samples_leaf': min_samples_leaf
}

# Train final RandomForestClassifier with best parameters on full
training fold
best_rf_clf =
RandomForestClassifier(n_estimators=best_params['n_estimators'],
max_depth=best_params['max_depth'],
min_samples_split=best_params['min_samples_split'],
min_samples_leaf=best_params['min_samples_leaf'],
n_jobs=-1,
class_weight='balanced',
random_state=42)

best_rf_clf.fit(X_train_scaled, y_train_res)

# Predict on test data with the best model
X_test_imputed = imputer.transform(X_test) # Impute test data
X_test_scaled = scaler.transform(X_test_imputed) # Scale test data
y_pred = best_rf_clf.predict(X_test_scaled)

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
tn, fp, fn, tp = conf_matrix.ravel()

# Calculate Metrics
accuracy = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred, average='weighted')
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
specificity = tn / (tn + fp)

# Calculate Loss Cost
cost_fp = 1000
cost_fn = 5000
loss_cost = cost_fp * fp + cost_fn * fn

# Append scores to lists
accuracy_scores.append(accuracy)
f1_scores.append(f1)
precision_scores.append(precision)
recall_scores.append(recall)
specificity_scores.append(specificity)
loss_costs.append(loss_cost)

# Print evaluation metrics for the fold
```

		print("\nEvaluation Metrics for Fold:")
		print("Best Parameters:", best_params)
		print("Best F1 Weighted Score:", best_f1_score)
		print("Accuracy Score: ", accuracy)
		print("F1 Score: ", f1)
		print("Precision: ", precision)
		print("Recall (Sensitivity): ", recall)
		print("Specificity: ", specificity)
		print("Loss Cost: ", loss_cost)
		
		print("Confusion Matrix:")
		print(conf_matrix)
		print(f"True Positives (TP): {tp}")
		print(f"True Negatives (TN): {tn}")
		print(f"False Positives (FP): {fp}")
		print(f"False Negatives (FN): {fn}")
		
		# Print average scores across all folds
		print("\nAverage Metrics Across All Folds:")
		print("Average Accuracy Score: ", np.mean(accuracy_scores))
		print("Average F1 Score: ", np.mean(f1_scores))
		print("Average Precision: ", np.mean(precision_scores))
		print("Average Recall (Sensitivity): ", np.mean(recall_scores))
		print("Average Specificity: ", np.mean(specificity_scores))
		print("Average Loss Cost: ", np.mean(loss_costs))

Kfold Volatility comparison for Model 1 and 2	import matplotlib.pyplot as plt	# Loss costs for each fold
		fold_loss_costs = [114000, 99000, 85000, 103000, 94000]
		
		# Calculate average loss cost
		avg_loss_cost = 99000
		
		# Labels for folds
		fold_labels = ['Fold 1', 'Fold 2', 'Fold 3', 'Fold 4', 'Fold 5']
		
		# Plotting the loss costs
		plt.figure(figsize=(10, 6))
		plt.bar(fold_labels, fold_loss_costs, color='skyblue', label='Loss Cost per Fold')
		plt.axhline(y=avg_loss_cost, color='orange', linestyle='--', label='Average Loss Cost')
		plt.xlabel('Folds')
		plt.ylabel('Loss Cost')
		plt.title('Loss Cost Comparison Across Folds')
		plt.ylim(0, max(fold_loss_costs) * 1.2) # Adjust ylim for better visualization
		plt.legend()
		plt.grid(True)
		plt.tight_layout()
		
		# Adding values above bars
		for i, v in enumerate(fold_loss_costs):
		plt.text(i, v + 5000, str(v), ha='center', va='bottom', fontsize=10)
		
		# Adding average value annotation
		plt.text(len(fold_loss_costs) - 0.5, avg_loss_cost + 5000, f'Avg: {avg_loss_cost}', ha='center', va='bottom', fontsize=10, color='orange')
		
		plt.show()

Trade off 2- Learning Curve for Model 1		

Before tuning	import pandas as pd
	import numpy as np
	import matplotlib.pyplot as plt
	from boruta import BorutaPy
	from sklearn.ensemble import RandomForestClassifier
	from imblearn.over_sampling import SMOTE
	from sklearn.preprocessing import MinMaxScaler
	from sklearn.model_selection import train_test_split, learning_curve, validation_curve
	from sklearn.metrics import confusion_matrix, accuracy_score, f1_score, precision_score, recall_score
		# Assuming 'merged_df' is your DataFrame with the relevant data
		# Replace 'merged_df' with your actual DataFrame containing the data
		# merged_df = pd.read_csv('your_data.csv') # Replace with your data loading code
		X = merged_df.drop('feature591', axis=1) # Replace 'feature591' with your actual target column name
		y = merged_df['feature591']
		
		# Split data into training and testing sets
		X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
		
		# Boruta Feature Selection
		rf_clf = RandomForestClassifier(n_jobs=-1, class_weight='balanced', max_depth=5, random_state=42)
		boruta_selector = BorutaPy(rf_clf, n_estimators='auto', verbose=0, random_state=42) # Set verbose to 0
		boruta_selector.fit(X_train.values, y_train)
		
		# Selected features
		selected_features = X_train.columns[boruta_selector.support_].tolist()
		print("Selected Features: ", selected_features)
		
		# Update X_train and X_test with selected features
		X_train_selected = X_train[selected_features]
		X_test_selected = X_test[selected_features]
		
		# SMOTE Balancing on training data
		smote = SMOTE(random_state=42)
		X_train_res, y_train_res = smote.fit_resample(X_train_selected, y_train)
		
		# Min-Max Scaling
		scaler = MinMaxScaler()
		X_train_scaled = scaler.fit_transform(X_train_res)
		X_test_scaled = scaler.transform(X_test_selected)
		
		# Define parameter grid for RandomForestClassifier
		param_grid = {
		'n_estimators': [50, 100, 200, 300],
		'max_depth': [3, 5, 7, 10, 15],
		'min_samples_split': [2, 5, 10, 15],
		'min_samples_leaf': [1, 2, 4, 6]
		}
		
		# Perform hyperparameter tuning without cross-validation
		best_f1_score = -1
		best_params = None
		
		for n_estimators in param_grid['n_estimators']:
		for max_depth in param_grid['max_depth']:
		for min_samples_split in param_grid['min_samples_split']:
		for min_samples_leaf in param_grid['min_samples_leaf']:
		# Train RandomForestClassifier with current parameters
		rf_clf = RandomForestClassifier(n_estimators=n_estimators,

```
max_depth=max_depth,
min_samples_split=min_samples_split,
min_samples_leaf=min_samples_leaf,
n_jobs=-1,
class_weight='balanced',
random_state=42)

rf_clf.fit(X_train_scaled, y_train_res)

# Predict on test data
y_pred = rf_clf.predict(X_test_scaled)

# Calculate F1 score
f1 = f1_score(y_test, y_pred, average='weighted')

# Check if current model is better than previous ones
if f1 > best_f1_score:
best_f1_score = f1
best_params = {
'n_estimators': n_estimators,
'max_depth': max_depth,
'min_samples_split': min_samples_split,
'min_samples_leaf': min_samples_leaf
}

# Train final RandomForestClassifier with best parameters
best_rf_clf =
RandomForestClassifier(n_estimators=best_params['n_estimators'],
max_depth=best_params['max_depth'],
min_samples_split=best_params['min_samples_split'],
min_samples_leaf=best_params['min_samples_leaf'],
n_jobs=-1,
class_weight='balanced',
random_state=42)

best_rf_clf.fit(X_train_scaled, y_train_res)

# Predict on test data with the best model
y_pred = best_rf_clf.predict(X_test_scaled)

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
tn, fp, fn, tp = conf_matrix.ravel()

# Calculate Metrics
accuracy = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred, average='weighted')
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
specificity = tn / (tn + fp)

# Calculate Loss Cost
cost_fp = 1000
cost_fn = 5000
loss_cost = cost_fp * fp + cost_fn * fn

print("Best Parameters:", best_params)
print("Best F1 Weighted Score:", best_f1_score)
print("Accuracy Score: ", accuracy)
print("F1 Score: ", f1)
print("Precision: ", precision)
print("Recall (Sensitivity): ", recall)
print("Specificity: ", specificity)
print("Loss Cost: ", loss_cost)

print("Confusion Matrix:")
print(conf_matrix)
print(f"True Positives (TP): {tp}")
print(f"True Negatives (TN): {tn}")
print(f"False Positives (FP): {fp}")
print(f"False Negatives (FN): {fn}")

# Plotting the learning curve
def plot_learning_curve(estimator, title, X, y, ylim=None, cv=None,
scoring='f1_weighted', n_jobs=-1, train_sizes=np.linspace(.1, 1.0, 5)):
plt.figure()
plt.title(title)
if ylim is not None:
plt.ylim(*ylim)
plt.xlabel("Training examples")
plt.ylabel("Score")
train_sizes, train_scores, test_scores = learning_curve(
estimator, X, y, cv=cv, scoring=scoring, n_jobs=n_jobs,
train_sizes=train_sizes)
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)
plt.grid()

plt.fill_between(train_sizes, train_scores_mean - train_scores_std,
train_scores_mean + train_scores_std, alpha=0.1,
color="r")
plt.fill_between(train_sizes, test_scores_mean - test_scores_std,
test_scores_mean + test_scores_std, alpha=0.1, color="g")
plt.plot(train_sizes, train_scores_mean, 'o-', color="r",
label="Training score")
plt.plot(train_sizes, test_scores_mean, 'o-', color="g",
label="Cross-validation score")

plt.legend(loc="best")
return plt

title = "Learning Curves (RandomForestClassifier)"
plot_learning_curve(best_rf_clf, title, X_train_scaled, y_train_res, cv=5,
scoring='f1_weighted')
plt.show()

# Plotting the validation curve
def plot_validation_curve(estimator, title, X, y, param_name,
param_range, ylim=None, cv=None,
scoring="f1_weighted", n_jobs=-1):
plt.figure()
plt.title(title)
if ylim is not None:
```

		plt.ylim(*ylim)
		plt.xlabel(param_name)
		plt.ylabel("Score")
		train_scores, test_scores = validation_curve(
		estimator, X, y, param_name=param_name,
		param_range=param_range,
		cv=cv, scoring=scoring, n_jobs=n_jobs)
		train_scores_mean = np.mean(train_scores, axis=1)
		train_scores_std = np.std(train_scores, axis=1)
		test_scores_mean = np.mean(test_scores, axis=1)
		test_scores_std = np.std(test_scores, axis=1)
		plt.grid()
		
		plt.semilogx(param_range, train_scores_mean, label="Training score",
		color="darkorange", lw=2)
		plt.fill_between(param_range, train_scores_mean - train_scores_std,
		train_scores_mean + train_scores_std, alpha=0.2,
		color="darkorange", lw=2)
		plt.semilogx(param_range, test_scores_mean, label="Cross-validation
		score",
		color="navy", lw=2)
		plt.fill_between(param_range, test_scores_mean - test_scores_std,
		test_scores_mean + test_scores_std, alpha=0.2,
		color="navy", lw=2)
		plt.legend(loc="best")
		return plt
		
		title = "Validation Curve (RandomForestClassifier)"
		param_name = "n_estimators"
		param_range = [50, 100, 200, 300] # Specify the range of n_estimators
		plot_validation_curve(best_rf_clf, title, X_train_scaled, y_train_res,
		param_name=param_name,
		param_range=param_range, cv=5, scoring="f1_weighted")
		plt.show()

After tuning	import pandas as pd
	import numpy as np
	import matplotlib.pyplot as plt
	from boruta import BorutaPy
	from sklearn.ensemble import RandomForestClassifier
	from imblearn.over_sampling import SMOTE
	from sklearn.preprocessing import MinMaxScaler
	from sklearn.model_selection import train_test_split, learning_curve
	from sklearn.metrics import confusion_matrix, accuracy_score, f1_score, precision_score, recall_score
		# Assuming 'merged_df' is your DataFrame with the relevant data
		X = merged_df.drop('feature591', axis=1) # Replace 'feature591' with
		your actual target column name
		y = merged_df['feature591']
		
		# Split data into training and testing sets
		X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
		random_state=42)
		
		# Boruta Feature Selection
		rf_clf = RandomForestClassifier(n_jobs=-1, class_weight='balanced',
		max_depth=5, random_state=42)
		boruta_selector = BorutaPy(rf_clf, n_estimators='auto', verbose=0,
		random_state=42) # Set verbose to 0
		boruta_selector.fit(X_train.values, y_train)
		
		# Selected features
		selected_features = X_train.columns[boruta_selector.support_].tolist()
		print("Selected Features: ", selected_features)
		
		# Update X_train and X_test with selected features
		X_train_selected = X_train[selected_features]
		X_test_selected = X_test[selected_features]

```
# SMOTE Balancing on training data
smote = SMOTE(random_state=42)
X_train_res, y_train_res = smote.fit_resample(X_train_selected, y_train)

# Min-Max Scaling
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train_res)
X_test_scaled = scaler.transform(X_test_selected)

# Train Random Forest Classifier
rf_clf.fit(X_train_scaled, y_train_res)
y_pred = rf_clf.predict(X_test_scaled)

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
tn, fp, fn, tp = conf_matrix.ravel()

# Calculate Metrics
accuracy = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred, average='weighted')
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
specificity = tn / (tn + fp)

# Calculate Loss Cost
cost_fp = 1000
cost_fn = 5000
loss_cost = cost_fp * fp + cost_fn * fn

print("Accuracy Score: ", accuracy)
print("F1 Score: ", f1)
print("Precision: ", precision)
print("Recall (Sensitivity): ", recall)
print("Specificity: ", specificity)
print("Loss Cost: ", loss_cost)

print("Confusion Matrix:")
print(conf_matrix)
print(f"True Positives (TP): {tp}")
print(f"True Negatives (TN): {tn}")
print(f"False Positives (FP): {fp}")
print(f"False Negatives (FN): {fn}")

# Learning Curve
train_sizes, train_scores, test_scores = learning_curve(
rf_clf, X_train_scaled, y_train_res, cv=5, scoring='f1_weighted',
n_jobs=-1,
train_sizes=np.linspace(0.1, 1.0, 10))

# Calculate mean and standard deviation of training scores and test
scores
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

# Plotting the learning curve
plt.figure(figsize=(10, 6))
```

		plt.title("Learning Curve (RandomForestClassifier)")
		plt.xlabel("Training examples")
		plt.ylabel("Score")
		
		plt.grid()
		plt.fill_between(train_sizes, train_scores_mean - train_scores_std, train_scores_mean + train_scores_std, alpha=0.1,
		color="r")
		plt.fill_between(train_sizes, test_scores_mean - test_scores_std, test_scores_mean + test_scores_std, alpha=0.1, color="g")
		plt.plot(train_sizes, train_scores_mean, 'o-', color="r", label="Training score")
		plt.plot(train_sizes, test_scores_mean, 'o-', color="g", label="Cross-validation score")
		plt.legend(loc="best")
		
		plt.show()

After CV	import pandas as pd
	import numpy as np
	from boruta import BorutaPy
	from sklearn.ensemble import RandomForestClassifier
	from imblearn.over_sampling import SMOTE
	from sklearn.preprocessing import MinMaxScaler
	from sklearn.model_selection import train_test_split, StratifiedKFold, learning_curve
	from sklearn.metrics import confusion_matrix, accuracy_score, f1_score, precision_score, recall_score
	import matplotlib.pyplot as plt
		# Assuming 'merged_df' is your DataFrame with the relevant data
		X = merged_df.drop('feature591', axis=1) # Replace 'feature591' with your actual target column name
		y = merged_df['feature591']
		
		# Initialize k-fold cross-validation
		kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
		
		# Initialize lists to store evaluation metrics across folds
		accuracy_scores = []
		f1_scores = []
		precision_scores = []
		recall_scores = []
		specificity_scores = []
		loss_costs = []
		
		# Initialize lists to store learning curve data
		train_sizes_all = []
		train_scores_mean_all = []
		train_scores_std_all = []
		test_scores_mean_all = []
		test_scores_std_all = []
		
		# Iterate over each fold
		for fold_idx, (train_idx, test_idx) in enumerate(kfold.split(X, y)):
		print(f"\nFold {fold_idx + 1}:")
		
		# Split data into training and testing sets for this fold
		X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
		y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]
		
		# Boruta Feature Selection
		rf_clf = RandomForestClassifier(n_jobs=-1, class_weight='balanced', max_depth=5, random_state=42)
		boruta_selector = BorutaPy(rf_clf, n_estimators='auto', verbose=0, random_state=42) # Set verbose to 0
		boruta_selector.fit(X_train.values, y_train)
		
		# Selected features
		selected_features = X_train.columns[boruta_selector.support_].tolist()
		print("Selected Features: ", selected_features)

```
# Update X_train and X_test with selected features
X_train_selected = X_train[selected_features]
X_test_selected = X_test[selected_features]

# SMOTE Balancing on training data
smote = SMOTE(random_state=42)
X_train_res, y_train_res = smote.fit_resample(X_train_selected, y_train)

# Min-Max Scaling
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train_res)
X_test_scaled = scaler.transform(X_test_selected)

# Define parameter grid for RandomForestClassifier
param_grid = {
'n_estimators': [50, 100, 200, 300],
'max_depth': [3, 5, 7, 10, 15],
'min_samples_split': [2, 5, 10, 15],
'min_samples_leaf': [1, 2, 4, 6]
}

# Perform hyperparameter tuning without cross-validation
best_f1_score = -1
best_params = None

for n_estimators in param_grid['n_estimators']:
for max_depth in param_grid['max_depth']:
for min_samples_split in param_grid['min_samples_split']:
for min_samples_leaf in param_grid['min_samples_leaf']:
# Train RandomForestClassifier with current parameters
rf_clf = RandomForestClassifier(n_estimators=n_estimators,
max_depth=max_depth,
min_samples_split=min_samples_split,
min_samples_leaf=min_samples_leaf,
n_jobs=-1,
class_weight='balanced',
random_state=42)

rf_clf.fit(X_train_scaled, y_train_res)

# Predict on test data
y_pred = rf_clf.predict(X_test_scaled)

# Calculate F1 score
f1 = f1_score(y_test, y_pred, average='weighted')

# Check if current model is better than previous ones
if f1 > best_f1_score:
best_f1_score = f1
best_params = {
'n_estimators': n_estimators,
'max_depth': max_depth,
'min_samples_split': min_samples_split,
'min_samples_leaf': min_samples_leaf
}

# Train final RandomForestClassifier with best parameters
```

```
best_rf_clf =
RandomForestClassifier(n_estimators=best_params['n_estimators'],
max_depth=best_params['max_depth'],
min_samples_split=best_params['min_samples_split'],
min_samples_leaf=best_params['min_samples_leaf'],
n_jobs=-1,
class_weight='balanced',
random_state=42)

best_rf_clf.fit(X_train_scaled, y_train_res)

# Predict on test data with the best model
y_pred = best_rf_clf.predict(X_test_scaled)

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
tn, fp, fn, tp = conf_matrix.ravel()

# Calculate Metrics
accuracy = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred, average='weighted')
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
specificity = tn / (tn + fp)

# Calculate Loss Cost
cost_fp = 1000
cost_fn = 5000
loss_cost = cost_fp * fp + cost_fn * fn

# Append scores to lists
accuracy_scores.append(accuracy)
f1_scores.append(f1)
precision_scores.append(precision)
recall_scores.append(recall)
specificity_scores.append(specificity)
loss_costs.append(loss_cost)

# Print evaluation metrics for the fold
print("\nEvaluation Metrics for Fold:")
print("Best Parameters:", best_params)
print("Best F1 Weighted Score:", best_f1_score)
print("Accuracy Score: ", accuracy)
print("F1 Score: ", f1)
print("Precision: ", precision)
print("Recall (Sensitivity): ", recall)
print("Specificity: ", specificity)
print("Loss Cost: ", loss_cost)

print("Confusion Matrix:")
print(conf_matrix)
print(f"True Positives (TP): {tp}")
print(f"True Negatives (TN): {tn}")
print(f"False Positives (FP): {fp}")
print(f"False Negatives (FN): {fn}")

# Calculate learning curve
train_sizes, train_scores, test_scores, fit_times, _ = learning_curve(
best_rf_clf, X_train_scaled, y_train_res, cv=kfold, n_jobs=-1,
scoring='f1_weighted', train_sizes=np.linspace(.1, 1.0, 5),
return_times=True)

# Store learning curve data
train_sizes_all.append(train_sizes)
train_scores_mean_all.append(np.mean(train_scores, axis=1))
train_scores_std_all.append(np.std(train_scores, axis=1))
test_scores_mean_all.append(np.mean(test_scores, axis=1))
test_scores_std_all.append(np.std(test_scores, axis=1))

# Print average scores across all folds
print("\nAverage Metrics Across All Folds:")
print("Average Accuracy Score: ", np.mean(accuracy_scores))
print("Average F1 Score: ", np.mean(f1_scores))
print("Average Precision: ", np.mean(precision_scores))
print("Average Recall (Sensitivity): ", np.mean(recall_scores))
print("Average Specificity: ", np.mean(specificity_scores))
print("Average Loss Cost: ", np.mean(loss_costs))

# Plot learning curve
plt.figure()
plt.title("Learning Curve")
plt.xlabel("Training examples")
plt.ylabel("Score")

plt.grid()

train_scores_mean_all = np.array(train_scores_mean_all)
train_scores_std_all = np.array(train_scores_std_all)
test_scores_mean_all = np.array(test_scores_mean_all)
test_scores_std_all = np.array(test_scores_std_all)

plt.fill_between(train_sizes_all[0], train_scores_mean_all.mean(axis=0) -
train_scores_std_all.mean(axis=0),
train_scores_mean_all.mean(axis=0) +
train_scores_std_all.mean(axis=0), alpha=0.1,
color="r")
plt.fill_between(train_sizes_all[0], test_scores_mean_all.mean(axis=0) -
test_scores_std_all.mean(axis=0),
test_scores_mean_all.mean(axis=0) +
test_scores_std_all.mean(axis=0), alpha=0.1, color="g")
plt.plot(train_sizes_all[0], train_scores_mean_all.mean(axis=0), 'o-',
color="r",
label="Training score")
plt.plot(train_sizes_all[0], test_scores_mean_all.mean(axis=0), 'o-',
color="g",
label="Cross-validation score")

plt.legend(loc="best")
plt.show()
```