Fraud Analytics (CS6890)

Assignment: 3

Title : Example-dependent cost-sensitive regression

Team Details :	Name	Roll Number
	Shreesh Gupta	*CS23MTECH12009*
	Hrishikesh Hemke	*CS23MTECH14003*
	Manan Patel	*CS23MTECH14006*
	Yug Patel	*CS23MTECH14019*
	Bhargav Patel	*CS23MTECH11026*

In [ ]:
```
# Required libraries
!pip install pandas numpy matplotlib scikit-learn torch torch-geometric ne

```

Cost-Sensitive Logistic Regression

Overview

This notebook implements various approaches to logistic regression, specifically focusing on cost-sensitive logistic regression. Cost-sensitive learning is a critical area in machine learning where the costs associated with misclassification vary between classes. This notebook provides a comprehensive look into implementing cost-sensitive logistic regression models using both gradient descent and genetic algorithms.

Libraries and Data Loading

We start by importing necessary libraries and loading the dataset:

In [1]:
```
import numpy as np
from scipy.special import expit as sigmoid  # Sigmoid function
from sklearn.base import BaseEstimator, ClassifierMixin
from scipy.optimize import minimize
from sklearn.metrics import accuracy_score, log_loss,  confusion_matrix, f
import pandas as pd
from sklearn.linear_model import LogisticRegression
import random
import warnings
from sklearn.neighbors import kneighbors_graph
import torch
from torch_geometric.data import Data
warnings.filterwarnings('ignore', category=RuntimeWarning)
import numpy as np
from sklearn.model_selection import train_test_split
import plotly.express as px
import pandas as pd
from sklearn.model_selection import train_test_split
import numpy as np
from scipy.optimize import minimize
from scipy.special import expit as sigmoid  # Stable sigmoid function
from scipy.special import gamma, gammaincinv, gammainc
import numpy as np
from scipy.special import expit as sigmoid
import scipy
import plotly.graph_objects as go
from sklearn.metrics import f1_score
import plotly.figure_factory as ff
import numpy as np
```

Load the data from a CSV file:

In [2]:
```
data_path = 'costsensitiveregression.csv'
data = pd.read_csv(data_path)

# Display the first few rows and columns to confirm its structure
data.head()
```

Out[2]:

	NotCount	YesCount	ATPM	PFD	PFG	SFD	SFG	WP	WS	AH	AN	Status	FNC
0	2	21	0.0	0.000	0.0	0.0	0.0	0.000000	0.0	0.0	0.0	0	0.0
1	23	0	0.0	0.044	0.0	0.0	0.0	0.306179	0.0	0.0	0.0	1	0.0
2	1	22	0.0	0.000	0.0	0.0	0.0	0.000000	0.0	0.0	0.0	0	0.0
3	5	18	0.0	0.000	0.0	0.0	0.0	0.000000	0.0	0.0	0.0	1	0.0
4	1	22	0.0	0.000	0.0	0.0	0.0	0.000000	0.0	0.0	0.0	0	0.0

Visualization

Visualize the distribution of labels using Plotly:

In [3]:
```
# Let's assume the DataFrame 'data' has a 'status' column with binary labe
# Let's replace dummy data with your specific column name if different
# data['Status'] = data['Status'].replace({1: 'label-1', 0: 'label-0'})  #

# Calculate the frequency of each label
label_counts = data['Status'].value_counts()

# Create a pie chart
fig = px.pie(
    label_counts,
    values=label_counts.values,
    names=label_counts.index,
    title='Distribution of Labels in Status Column',
    color_discrete_sequence=px.colors.sequential.RdBu
)

# Show the plot
fig.show()

```

Data Preprocessing

We preprocess the data by encoding labels and splitting the dataset into training and testing sets:

In [4]:
```
X = data.iloc[:, :-3]  # Assuming the last three columns are 'Status', 'FN
y = data['Status']
costs = data['FNC']
```

Model Implementation

Cost Matrix Definition

Define the cost matrix for cost-sensitive learning:

In [5]:
```
FP = np.full(len(y), 6)  # Replace len(y) with the actual length of your l
TP = np.full(len(y), 6)   # Same here
TN = np.zeros(len(y))     # And here
cost_matrix = np.column_stack((TP, data['FNC'], FP, TN)).astype(float)

# Split the data into train and test sets
X_train, X_test, y_train, y_test, cost_matrix_train, cost_matrix_test, cos
    X, y, cost_matrix, costs, test_size=0.20, random_state=42
)


```

Cost-Sensitive Logistic Regression Using Gradient Descent (Bahnsen approach)

Introduction

This implementation details a cost-sensitive logistic regression model that optimizes its parameters using gradient descent. Unlike traditional logistic regression, this model considers the varying costs of different types of classification errors, which can be crucial in domains like healthcare, finance, or any field where the consequences of errors are asymmetric.

Model Description

Sigmoid Function

The logistic regression model uses the sigmoid function to map the net input (z) (a linear combination of input features (X) and weights (w)) to probabilities between 0 and 1. The sigmoid function is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Cost-Sensitive Loss Function

The cost-sensitive loss function incorporates costs associated with four types of outcomes: true positives, true negatives, false positives, and false negatives. It is defined as follows:

$$L(y, \hat{y}, C) = \sum_{i=1}^{N} \left[y_i \times ((1 - \hat{y}_i) \times C_{FN} + \hat{y}_i \times C_{TP}) + (1 - y_i) \times (\hat{y}_i \times C_{FP} + (1 - \hat{y}_i) \times C \right.$$

Here:

- (y_i) is the actual label of the ith sample.
- (\hat{y}_i) is the predicted probability for the ith sample.
- (C_{TP}, C_{FN}, C_{FP}, C_{TN}) are the costs associated with true positives, false negatives, false positives, and true negatives, respectively.

Gradient Descent Optimization

The model parameters (weights (w) and bias (b)) are updated using gradient descent to minimize the cost-sensitive loss. The gradients for the weights and bias are computed as:

$$\frac{\partial L}{\partial w} = -\frac{1}{N} \sum_{i=1}^{N} X^T \times ((y - \hat{y}) \times (C_{TP} - C_{FP}))$$

$$\frac{\partial L}{\partial b} = -\frac{1}{N} \sum_{i=1}^{N} ((y - \hat{y}) \times (C_{TP} - C_{FP}))$$

The weights and bias are then updated iteratively:

$$w = w - \alpha \times \frac{\partial L}{\partial w}$$

$$b = b - \alpha \times \frac{\partial L}{\partial b}$$

where (\alpha) is the learning rate.

Implementation

```
class CostSensitiveLogisticRegressionL:
    def __init__(self, alpha=0.01, epochs=100):
        self.alpha = alpha
        self.epochs = epochs
        self.weights = None
        self.bias = None

    def fit(self, X, y, cost_matrix):
        self.weights, self.bias = gradient_descent(X, y, cost_matrix,
self.alpha, self.epochs)

    def predict_proba(self, X):
        probabilities = sigmoid(np.dot(X, self.weights) + self.bias)
        return probabilities

    def predict(self, X):
        probabilities = self.predict_proba(X)
        return (probabilities >= 0.5).astype(int)
```

In [6]:

```
# For Bahnsen et al.-inspired model
# Define the sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Define the cost-sensitive loss function
def calculate_loss(X, y, weights, bias, cost_matrix):
    # Predict probabilities
    probabilities = sigmoid(np.dot(X, weights) + bias)

    # Calculate cost-sensitive loss
    loss = y * ((1 - probabilities) * cost_matrix[:, 1] + probabilities *
           + (1 - y) * (probabilities * cost_matrix[:, 0] + (1 - probabili
    return np.sum(loss)

# Gradient descent function to minimize the cost-sensitive loss
def gradient_descent(X, y, cost_matrix, alpha, epochs):
    n_samples, n_features = X.shape
    weights = np.zeros(n_features)
    bias = 0

    for _ in range(epochs):
        probabilities = sigmoid(np.dot(X, weights) + bias)

        # Gradient calculation
        dw = -np.dot(X.T, (y - probabilities) * (cost_matrix[:, 2] - cost_
        db = -np.sum((y - probabilities) * (cost_matrix[:, 2] - cost_matri

        # Update weights and bias
        weights -= alpha * dw
        bias -= alpha * db

    return weights, bias

# Define the cost-sensitive logistic regression model
class CostSensitiveLogisticRegressionL:
    def __init__(self, alpha=0.01, epochs=100):
        self.alpha = alpha
        self.epochs = epochs
        self.weights = None
        self.bias = None

    def fit(self, X, y, cost_matrix):
        self.weights, self.bias = gradient_descent(X, y, cost_matrix, self

    def predict_proba(self, X):
        return sigmoid(np.dot(X, self.weights) + self.bias)

    def predict(self, X):
        probabilities = self.predict_proba(X)
        return (probabilities >= 0.5).astype(int)


```

Model Training and Evaluation Summary

- **Algorithm**: Cost-sensitive logistic regression optimized via genetic algorithm.
- **Generations**: Trained over 50 generations.
- **Evaluation**:
  - **Accuracy**: 29.9% on the test set, indicating low performance.
  - **Log Loss**: 6.009, suggesting poor prediction calibration.

In [7]:
```
# Initialize and fit the model
cost_sensitive_lr_bun = CostSensitiveLogisticRegressionL()
cost_sensitive_lr_bun.fit(X_train, y_train, cost_matrix_train)

# Predictions and evaluations
predictions_cslr_bun = cost_sensitive_lr_bun.predict(X_test)
accuracy_cslr_bun  = accuracy_score(y_test, predictions_cslr_bun)
loss_cslr_bun = log_loss(y_test, cost_sensitive_lr_bun.predict_proba(X_tes

print("Accuracy:", accuracy_cslr_bun)
print("Log Loss:", loss_cslr_bun)

```

```
Accuracy: 0.2987334055811433
Log Loss: 0.6931471805599452
```

Nikou Gunnemann's approach

Customized Logistic Loss Function Explanation

The `logistic_loss` function in Python computes a variant of the logistic regression loss, incorporating costs that vary based on different model variants. This adaptation allows the model to differently weigh the importance of misclassifications according to their associated costs, crucial in scenarios where different types of errors have significantly different implications.

General Setup for Logistic Regression

Logistic regression predicts the probability (p) that a given instance (x) belongs to the positive class (i.e., (y = 1)):

$$p = \sigma(\beta^T x)$$

where (\sigma) is the sigmoid activation function defined by:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The conventional logistic loss function, used for training logistic regression models, penalizes predictions based on the log likelihood of the true classes:

$$L(y, p) = -\left[y \log(p) + (1 - y) \log(1 - p)\right]$$

This function is summed over all instances, where being wrong and confident leads to higher penalties.

Variant-Specific Cost-Sensitive Loss Functions

Variant A: Linear Weighting

- **Weights ((a_i)):** Directly proportional to the costs associated with each instance, (a_i = c_i).
- **Exponents ((b_i)):** Set to 1, maintaining the standard logistic loss form but scaled by cost.

$$L_A = \sum c_i \left[y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \right]$$

Variant B: Exponential Weighting

- **Weights ((a_i)):** All set to 1.
- **Exponents ((b_i)):** Determined by the inverse gamma function of the costs, modifying the sensitivity of the loss to errors, (b_i = \Gamma^{-1}(c_i) - 1).

$$L_B = \sum \left[y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \right]^{b_i}$$

Variant C: Ratio Control

- **Weights ((a_i)) and Exponents ((b_i)):** Both parameters are tailored to manage the ratio of costs between different types of misclassifications. Calculations involve solving equations to balance the ratios appropriately:
  - (b_i) is calculated such that it satisfies a specific cost-benefit ratio.
  - (a_i = 1 / \Gamma(b_i + 1)).

$$L_C = \sum \frac{1}{\Gamma(b_i + 1)} \left[y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \right]^{b_i}$$

Variant D: Maintaining Constant Correct Classification Cost

- **Weights ((a_i)) and Exponents ((b_i)):** Designed to keep the cost of correct classifications constant while varying the penalties for misclassifications according to their costs.
  - (a_i) and (b_i) are adjusted to ensure that the losses for correct classifications remain constant across different costs.
  - (a_i = \frac{0.5}{\Gamma(b_i + 1) - \Gamma(b_i + 1, 0.6931)}).

$$L_D = \sum \frac{0.5}{\Gamma(b_i + 1) - \Gamma(b_i + 1, 0.6931)} \left[y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \right]^{b_i}$$

These variants modify the standard logistic loss to integrate the cost directly into the model training, making the logistic regression sensitive to the cost associated with incorrect

In [8]:
```
def logistic_loss(beta, X, y, costs, variant):
    predictions = sigmoid(np.dot(X, beta))
    m = len(y)  # Number of instances

    if variant == 'A':
        # Variant A: Linear weights as per cost
        ai = costs
        bi = np.ones_like(costs)
    elif variant == 'B':
        # Variant B: Exponential weights determined by the inverse gamma f
        ai = np.ones_like(costs)
        bi = gammaincinv(costs, 0.5) - 1  # This needs validation for corr
    elif variant == 'C':
        # Variant C: Control the ratio with adjusted ai and bi
        bi = np.array([gammaincinv(1, (1 + 0.5 * ci) / (ci + 1)) for ci in
        ai = 1 / gamma(bi + 1)
    elif variant == 'D':
        # Variant D: Similar to C but keeps the correct classification los
        bi = np.array([gammaincinv(1, (1 + 0.5 * ci) / (ci + 1)) for ci in
        ai = 0.5 / (gamma(bi + 1) - gammainc(bi + 1, 0.6931))

    # Calculate the customized loss
    loss = np.sum(ai * (y * (-np.log(predictions + 1e-15))**bi + (1 - y) *
    return loss

```

Fitting Cost-Sensitive Logistic Regression Models

The `fit_model` function is designed to fit logistic regression models by optimizing a custom logistic loss function that takes into account different cost structures depending on the specified variant. The function employs the `minimize` method from `scipy.optimize` to find the best model coefficients (`beta`) that minimize the loss.

Function Definition

The `fit_model` function takes the following parameters:

- **X (array-like)**: Feature matrix for the training data.
- **y (array-like)**: Target vector for the training data.
- **costs (array-like)**: Vector of costs associated with each instance in the training data.
- **variant (str)**: Specifies the variant of the cost-sensitive logistic regression to be used.

It initializes the model coefficients to zeros, then uses the BFGS algorithm (a quasi-Newton method) to optimize the coefficients by minimizing the custom logistic loss:

In [9]:
```
def fit_model(X, y, costs, variant):
    beta_init = np.zeros(X.shape[1])
    result = minimize(logistic_loss, beta_init, args=(X, y, costs, variant
    return result.x



# Fit models for each variant
variants = ['A', 'B', 'C', 'D']
models_nik = {}
for variant in variants:
    models_nik[variant] = fit_model(X_train, y_train, costs_train, variant
    print(f"Model coefficients for variant {variant}: {models_nik[variant]
```

```
Model coefficients for variant A: [ 0.1190579  -0.15401323  0.05821664  0.219
04436 -0.42776506 -0.01477515
  0.00892466 -0.05023754 -0.11274034 -0.06083723]
Model coefficients for variant B: [-0.00328367  0.00659971 -0.28634056  0.021
09848  0.02959005 -0.02277006
 -0.03964703 -0.00038196 -0.02422468  0.01262785]
Model coefficients for variant C: [ 1.67472449e-01 -1.46788075e-01 -1.4522074
5e+00  1.76324331e-02
  1.49062543e-01  4.66728940e-02 -4.56879000e+01 -1.76797565e-02
 -9.67787998e-02  3.04078471e-01]
Model coefficients for variant D: [ 1.69746606e-01 -1.53645542e-01 -1.3599763
0e+00  2.23429085e-02
  1.46982817e-01  3.17544436e-02 -4.29806934e+01 -1.77911989e-02
 -8.88658171e-02  2.86332200e-01]
```

Logistic Regression Overview and Performance Evaluation

Logistic Regression Explanation

Logistic Regression is a statistical method used for binary classification that predicts the probability of the target variable's categories. It models the probability of the default class (usually "1") using the logistic function.

Equation of the Logistic Regression Model

The logistic regression model calculates probabilities using the logistic function:

[\text{logit}(p) = \log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1x_1 + \beta_2x_2 + \ldots + \beta_nx_n]

Where:

- (p) is the probability of belonging to the positive class.
- (\beta_0, \beta_1, ..., \beta_n) are the coefficients of the model.
- (x_1, x_2, ..., x_n) are the predictor variables.

Probability Prediction Using the Sigmoid Function

The logistic function or the sigmoid function is used to convert the linear regression output to a probability:

[p = \sigma(z) = \frac{1}{1 + e^{-z}}]

where (z) is the linear combination of the features and coefficients, (z = \beta_0 + \beta_1x_1 + \ldots + \beta_nx_n).

Model Training and Performance

Training Process

A standard logistic regression model was trained using the `LogisticRegression` class from sklearn, configured with a maximum of 500 iterations to ensure convergence.

```
standard_lr = LogisticRegression(max_iter=500)
standard_lr.fit(X_train, y_train)
```

Results

- **Accuracy**: The accuracy achieved was 86.58%, indicating the percentage of correctly predicted instances in the test data.
- **Log Loss**: The log loss recorded was 0.3056, reflecting the model's confidence in its predictions. Lower log loss values are better as they indicate higher confidence and accuracy in the predictions.

In [10]:
```
standard_lr = LogisticRegression(max_iter=500)
standard_lr.fit(X_train, y_train)
predictions_lr = standard_lr.predict(X_test)

accuracy_lr  = accuracy_score(y_test, predictions_lr)
loss_lr = log_loss(y_test, standard_lr.predict_proba(X_test))

print("Accuracy:", accuracy_lr)
print("Log Loss:", loss_lr)
```

```
Accuracy: 0.8661609319967488
Log Loss: 0.30547488662730277
```

Mathematical Explanation and Results Discussion for Cost-Sensitive Logistic Regression

Mathematical Foundation

- **Objective**: Minimize both predictive error and the financial/practical impact of errors, quantified by a cost matrix.
- **Cost Matrix**: Assigns costs to different types of classification errors:
  - **True Positives (TP)**
  - **False Positives (FP)**
  - **True Negatives (TN)**
  - **False Negatives (FN)**
- **Cost-Sensitive Loss Function**:

  The loss function for cost-sensitive logistic regression, accounting for different types of errors, is defined as:

$$L(y, \hat{y}, C) = \sum_{i=1}^{N} \left[y_i \times ((1 - \hat{y}_i) \times C_{FN} + \hat{y}_i \times C_{TP}) + (1 - y_i) \times (\hat{y}_i \times C_{FP} + (1 - \hat{y}_i) \times C \right.$$

This function:

In [11]:
```
def cost_sensitive_logistic_loss(y_true, y_pred, cost_matrix):
    total_loss = y_true * ((1 - y_pred) * cost_matrix[:, 1] + y_pred * cos
    return np.sum(total_loss)
```

In [12]:
```
def evaluate_loss(models, X, y, costs, variant):
    beta = models[variant]
    predictions = sigmoid(X.dot(beta))
    # Calculate the logistic regression loss
    loss = logistic_loss(beta, X, y, costs, variant)
    return loss

def gamma_func(x):
    return scipy.special.gamma(x)
```

In [13]:
```
cost_cslr_bun = cost_sensitive_logistic_loss(y_test, predictions_cslr_bun,
print("Bahnsen approach Cost of cost-sensitive logistic regression:", cost

losses = {}
for variant in variants:
    losses[variant] = evaluate_loss(models_nik, X_test, y_test, costs_test
    print(f"Nikou Gunnemann's approach Cost-sensitive Logistic Loss for va

cost_lr = cost_sensitive_logistic_loss(y_test, predictions_lr, cost_matrix
print("Cost of Standard logistic regression:", cost_lr)
```

```
Bahnsen approach Cost of cost-sensitive logistic regression: 177168.0
Nikou Gunnemann's approach Cost-sensitive Logistic Loss for variant A: 486973
5.186412062
Nikou Gunnemann's approach Cost-sensitive Logistic Loss for variant B: 8226.0
9071459818
Nikou Gunnemann's approach Cost-sensitive Logistic Loss for variant C: 7998.8
377675579395
Nikou Gunnemann's approach Cost-sensitive Logistic Loss for variant D: 5092.2
63001465815
Cost of Standard logistic regression: 814303.592301
```

Cost Comparison between Standard and Cost-Sensitive Logistic Regression Models

The bar chart illustrates a stark contrast in the total misclassification costs incurred by two logistic regression models: the standard LR and the cost-sensitive LR.

- **Standard Logistic Regression (Blue Bar)**:
  - The cost incurred by the standard logistic regression model is substantially higher, with a total of 812,473.36. This suggests that while the model might predict accurately, it does not account for the varying costs of different types of classification errors, leading to a less economically optimal outcome.
- **Cost-Sensitive Logistic Regression (Green Bar)**:
  - In contrast, the cost-sensitive logistic regression model shows a significantly reduced cost of 177,162.0. This model incorporates a cost matrix that assigns specific costs to different types of errors, such as false positives and false negatives. By doing so, it is able to minimize the more economically impactful errors, thus reducing the overall cost.

In [14]:
```
# Data for plotting
costs = [cost_lr, cost_cslr_bun] + [losses[v] for v in variants]
labels = ['Standard LR', 'Bahnsen approach Cost-Sensitive LR'] + [f"Nikou

# Create a bar chart
fig = go.Figure(go.Bar(
    x=labels,
    y=costs,
    text=costs,
    textposition='auto',
    marker_color=['red', 'green'] + ['blue', 'purple', 'yellow', 'orange']
))

# Add titles and labels
fig.update_layout(
    title='Comparison of Logistic Regression Costs Across Different Approa
    xaxis_title='Model Type',
    yaxis_title='Cost',
    template='plotly_white'
)

# Improve resolution by setting the width and height
fig.update_layout(width=1000, height=600)  # Adjusted for better visualiz

# Show the plot
fig.show()

```

F1 Score Comparison Between Standard and Cost-Sensitive Logistic Regression

F1 Score is the harmonic mean of precision and recall, offering a balance between the two by taking both false positives and false negatives into account. It is particularly useful when the class distribution is imbalanced.

The F1 Score can be mathematically represented as:

$$F1 = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

where:

- **Precision** is the ratio of true positive predictions to the total predicted positives:
$$\text{Precision} = \frac{TP}{TP + FP}$$
- **Recall** (also known as sensitivity) is the ratio of true positive predictions to the actual positives:
$$\text{Recall} = \frac{TP}{TP + FN}$$

Results

- **Standard Logistic Regression**:
  - **F1 Score**: 0.763, indicating a relatively high balance between precision and recall. This model may predict positive classes more reliably than the cost-sensitive counterpart.
- **Cost-Sensitive Logistic Regression**:
  - **F1 Score**: 0.460, significantly lower than the standard model, which implies a trade-off between precision and recall that favors minimizing costlier errors over achieving a balance.

Discussion

- The higher F1 Score of the standard LR model indicates better performance in terms of balanced precision and recall. However, it doesn't consider the different costs of misclassification.
- The cost-sensitive LR model's lower F1 Score suggests that while it may be optimizing for cost, it does so at the expense of either precision, recall, or both.
- It's important to note that a lower F1 Score in the cost-sensitive model does not necessarily indicate poorer overall performance, as it may be strategically accepting lower precision and/or recall to reduce more costly errors, aligning with its optimization objectives.

The choice between these models should be guided by the specific context and objectives:

In [15]:
```
# Assuming predictions for each variant are available as predictions_nik[v
predictions_nik = {}  # Dictionary to store predictions
variants = ['A', 'B', 'C', 'D']  # Define your variants
for variant in variants:
    # Generate predictions for variant; replace this with your actual pred
    predictions_nik[variant] = np.round(sigmoid(np.dot(X_test, models_nik[

# Calculate F1 scores
f1_lr = f1_score(y_test, predictions_lr)
f1_cslr_bun = f1_score(y_test, predictions_cslr_bun)
f1_scores_nik = {variant: f1_score(y_test, predictions_nik[variant]) for v

# Print F1 scores
print("Standard LR F1 Score:", f1_lr)
print("Bahnsen approach Cost-Sensitive LR F1 Score:", f1_cslr_bun)
for variant in variants:
    print(f"Nikou Gunnemann's approach Variant {variant} F1 Score:", f1_sc

```

```
Standard LR F1 Score: 0.7636363636363638
Bahnsen approach Cost-Sensitive LR F1 Score: 0.46003807139690733
Nikou Gunnemann's approach Variant A F1 Score: 0.756384222509384
Nikou Gunnemann's approach Variant B F1 Score: 0.1850216662578694
Nikou Gunnemann's approach Variant C F1 Score: 0.7663705359769372
Nikou Gunnemann's approach Variant D F1 Score: 0.7667671216687348
```

In [16]:

```
# Compute the cost according to some method outlined in the PDF
def compute_cost(predictions, true_labels, costs):
    cm = confusion_matrix(true_labels, predictions)
    tp_cost = costs[:, 0] * cm[1, 1]  # True Positives
    fn_cost = costs[:, 1] * cm[1, 0]  # False Negatives
    fp_cost = costs[:, 2] * cm[0, 1]  # False Positives
    tn_cost = costs[:, 3] * cm[0, 0]  # True Negatives
    total_cost = tp_cost.sum() + fn_cost.sum() + fp_cost.sum() + tn_cost.s
    return total_cost

# Compute costs for standard logistic regression and Bahnsen's approach
cost_lr = compute_cost(predictions_lr, y_test, cost_matrix_test)
cost_cslr_bun = compute_cost(predictions_cslr_bun, y_test, cost_matrix_tes

# Dictionary to store costs for Nikou Gunnemann's approach variants
costs_nik = {}
variants = ['A', 'B', 'C', 'D']
for variant in variants:
    predictions_variant = np.round(sigmoid(np.dot(X_test, models_nik[varia
    costs_nik[variant] = compute_cost(predictions_variant, y_test, cost_ma

# Print results
print("Standard LR Cost:", cost_lr)
print("Bahnsen approach Cost-Sensitive LR Cost:", cost_cslr_bun)
for variant in variants:
    print(f"Nikou Gunnemann's approach Variant {variant} Cost:", costs_nik

```

```
Standard LR Cost: 39207831424.13624
Bahnsen approach Cost-Sensitive LR Cost: 5231416704.0
Nikou Gunnemann's approach Variant A Cost: 42817146185.82045
Nikou Gunnemann's approach Variant B Cost: 104513950986.5759
Nikou Gunnemann's approach Variant C Cost: 37255560550.78639
Nikou Gunnemann's approach Variant D Cost: 37630307950.05574
```

In [17]:
```
def plot_confusion_matrix(cm, class_names, title='Confusion Matrix'):
    """
    Plots a confusion matrix using Plotly's Figure Factory.

    Parameters:
    cm (array-like): A confusion matrix of shape (n_classes, n_classes).
    class_names (list of strings): List of class names, in the order they
    title (str): Title of the heatmap. Default is 'Confusion Matrix'.
    """
    # Convert the confusion matrix to a DataFrame
    data = np.array(cm)

    # Create the heatmap
    fig = ff.create_annotated_heatmap(
        z=data,
        x=class_names,
        y=class_names,
        annotation_text=data.astype(str),
        showscale=True,
        colorscale='Viridis'
    )

    # Add titles and labels
    fig.update_layout(
        title=title,
        xaxis=dict(title='Predicted Label'),
        yaxis=dict(title='True Label')
    )

    # Improve the xaxis and yaxis visibility
    fig.update_xaxes(side="bottom")

    # Add colorbar
    fig['data'][0]['showscale'] = True

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

In [18]:
```
class_names = ['Negative', 'Positive']
```

Confusion Matrix Comparison

In [19]:
```
cm_cslr_bun = confusion_matrix(y_test, predictions_cslr_bun)
plot_confusion_matrix(cm_cslr_bun, class_names, title='Bahnsen approach Co
```

In [20]:
```
variants = ['A', 'B', 'C', 'D']  # Define your variants

for variant in variants:
    # Generate predictions for each variant; ensure you have models for ea
    predicted_labels_variant = np.round(sigmoid(np.dot(X_test, models_nik|

    # Calculate confusion matrix for the variant
    cm_variant = confusion_matrix(y_test, predicted_labels_variant)

    # Plot the confusion matrix
    plot_confusion_matrix(cm_variant, class_names, title=f"Variant {variar


```

In [21]:
```
cm_lr = confusion_matrix(y_test, predictions_lr)
plot_confusion_matrix(cm_lr, class_names, title='Standard Logestic Regress
```

Cost Matrix Analysis for Logistic Regression Models

In [22]:

```
def plot_cost_matrix(cost_matrix, title='Cost Matrix'):
    """
    Plots a cost matrix using Plotly's Figure Factory.

    Parameters:
    cost_matrix (DataFrame): A DataFrame representing the cost matrix.
    title (str): Title of the heatmap.
    """
    # Convert the DataFrame to a numpy array for plotting
    data = cost_matrix.values
    class_names = cost_matrix.columns.tolist()  # Assumes columns are name

    # Create the heatmap
    fig = ff.create_annotated_heatmap(
        z=data,
        x=class_names,
        y=cost_matrix.index.tolist(),
        annotation_text=data.astype(str),
        showscale=True,
        colorscale='Blues'
    )

    # Add titles and labels
    fig.update_layout(
        title=title,
        xaxis=dict(title='Predicted Condition'),
        yaxis=dict(title='Actual Condition')
    )

    # Improve the xaxis and yaxis visibility
    fig.update_xaxes(side="bottom")

    # Add colorbar
    fig['data'][0]['showscale'] = True

    # Show the figure
    fig.show()

```

In [23]:

```
# Assume costs for FP and FN are given or calculated elsewhere in the anal
cost_fp = 6  # Example cost for False Positives
cost_fn = 6  # Example cost for False Negatives
cost_tp = 0  # Usually, the cost of True Positives is zero
cost_tn = 0  # Usually, the cost of True Negatives is zero

def calculate_cost_matrix(predictions, y_test):
    cm = confusion_matrix(y_test, predictions, labels=[1, 0])

    # Calculating costs based on the confusion matrix
    tp_cost = cm[0, 0] * cost_tp
    fn_cost = cm[0, 1] * cost_fn
    fp_cost = cm[1, 0] * cost_fp
    tn_cost = cm[1, 1] * cost_tn

    cost_matrix = pd.DataFrame({
        'Predicted Positive': [tp_cost, fp_cost],
        'Predicted Negative': [fn_cost, tn_cost]
    }, index=['Actual Positive', 'Actual Negative'])

    return cost_matrix

# Assume `models_nik` and `X_test` are defined
variants = ['A', 'B', 'C', 'D']  # Define your variants
cost_matrices_nik = {}  # Dictionary to store cost matrices for each varia

for variant in variants:
    predicted_labels_variant = np.round(sigmoid(np.dot(X_test, models_nik[
    cost_matrices_nik[variant] = calculate_cost_matrix(predicted_labels_va

# Calculate cost matrices for both models
cost_matrix_lr = calculate_cost_matrix(predictions_lr, y_test)
cost_matrix_cslr_bun = calculate_cost_matrix(predictions_cslr_bun, y_test)

print("Cost Matrix for Standard Logistic Regression:")
plot_cost_matrix(cost_matrix_lr, title="Standard Logistic Regression Cost

print("Bahnsen approach Cost Matrix for Cost-Sensitive Logistic Regressior
plot_cost_matrix(cost_matrix_cslr_bun, title="Bahnsen Approach Cost-Sensit

for variant in variants:
    print(f"Nikou Gunnemann's approach Cost Matrix for Variant {variant}:"
    plot_cost_matrix(cost_matrices_nik[variant], title=f"Variant {variant}
```

```
Cost Matrix for Standard Logistic Regression:

Bahnsen approach Cost Matrix for Cost-Sensitive Logistic Regression:

Nikou Gunnemann's approach Cost Matrix for Variant A:

Nikou Gunnemann's approach Cost Matrix for Variant B:

Nikou Gunnemann's approach Cost Matrix for Variant C:

Nikou Gunnemann's approach Cost Matrix for Variant D:
```

Savings Calculation Explanation

The `calculate_savings` function compares the cost efficiency between standard logistic regression and another classification technique.

Function Details

- **Purpose**: To quantify the economic benefit of using an alternative technique over standard logistic regression in terms of cost savings.
- **Parameters**:
  - `cost_lr` : The total misclassification cost from using standard logistic regression.
  - `cost_x` : The total misclassification cost from using the alternative technique.

Savings Calculation

The savings are calculated using the formula:

$$\text{Savings} = \frac{\text{Cost}_{\text{LR}} - \text{Cost}_{\text{X}}}{\text{Cost}_{\text{LR}}}$$

- **Interpretation**: The savings score represents the proportion of costs saved by using the new technique. A higher score indicates greater savings.

In [24]:

```
def calculate_savings(cost_lr, cost_x):
    """
    Calculate the savings of a technique compared to logistic regression.

    Parameters:
    cost_lr (float): The misclassification cost using logistic regression.
    cost_x (float): The misclassification cost using the new technique.

    Returns:
    float: The savings score.
    """
    savings = (cost_lr - cost_x) / cost_lr
    return savings

```

In [25]:
```
cost_savings_bun = calculate_savings(cost_lr, cost_cslr_bun)
print(f"Cost savings for Bahnsen approach: {cost_savings_bun}")

# Dictionary to store cost savings for Nikou Gunnemann's approach variants
cost_savings_nik = {}

for variant in variants:
    # Assume cost_matrices_nik contains the cost matrices for each variant
    cost_savings_nik[variant] = calculate_savings(cost_lr, losses[variant]
    print(f"Cost savings for Nikou Gunnemann's approach Variant {variant}:
```

```
Cost savings for Bahnsen approach: 0.8665721486248904
Cost savings for Nikou Gunnemann's approach Variant A: 0.999875796875024
Cost savings for Nikou Gunnemann's approach Variant B: 0.9999997901926627
Cost savings for Nikou Gunnemann's approach Variant C: 0.9999997959887738
Cost savings for Nikou Gunnemann's approach Variant D: 0.999999870121279
```

In [26]:
```
# Prepare the labels and values for plotting
labels = ['Bahnsen approach'] + [f'Variant {v}' for v in variants]
savings = [cost_savings_bun] + [cost_savings_nik[v] for v in variants]

import plotly.graph_objects as go

# Create a bar chart
fig = go.Figure(go.Bar(
    x=labels,
    y=savings,
    text=[f"{s:.2f}%" for s in savings],  # Formatting the text on the bar
    textposition='auto',
    marker_color='blue'  # Color of the bars
))

# Add titles and labels
fig.update_layout(
    title='Cost Savings Compared to Standard Logistic Regression',
    xaxis_title='Model Variant',
    yaxis_title='Savings (%)',
    template='plotly_white'
)

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