Step 1: Dataset Selection I am using the Ionosphere dataset from the UCI Machine Learning Repository - https://archive.ics.uci.edu/dataset/52/ionosphere This dataset is for binary classification tasks, as it involves distinguishing between 'good' and 'bad' radar returns. It contains 351 instances with 34 continuous features.

```
!pip install ucimlrepo
```

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
```

```
from ucimlrepo import fetch_ucirepo

# fetch dataset
ionosphere = fetch_ucirepo(id=52)

# data (as pandas dataframes)
x = ionosphere.data.features
y = ionosphere.data.targets

# metadata
print(ionosphere.metadata)

# variable information
print(ionosphere.variables)
```

```
16  Attribute17  Feature   Continuous        None        None  None
17  Attribute18  Feature   Continuous        None        None  None
18  Attribute19  Feature   Continuous        None        None  None
19  Attribute20  Feature   Continuous        None        None  None
20  Attribute21  Feature   Continuous        None        None  None
21  Attribute22  Feature   Continuous        None        None  None
22  Attribute23  Feature   Continuous        None        None  None
23  Attribute24  Feature   Continuous        None        None  None
24  Attribute25  Feature   Continuous        None        None  None
25  Attribute26  Feature   Continuous        None        None  None
26  Attribute27  Feature   Continuous        None        None  None
27  Attribute28  Feature   Continuous        None        None  None
28  Attribute29  Feature   Continuous        None        None  None
29  Attribute30  Feature   Continuous        None        None  None
30  Attribute31  Feature   Continuous        None        None  None
31  Attribute32  Feature   Continuous        None        None  None
32  Attribute33  Feature   Continuous        None        None  None
33  Attribute34  Feature   Continuous        None        None  None
34        Class   Target  Categorical        None        None  None

   missing_values
0              no
1              no
2              no
3              no
```

```
30             no
31             no
32             no
33             no
34             no
```

Step 2: Binary Output Variable The Ionosphere dataset already has a binary output variable with classes 'good' and 'bad'. I am mapping these to numerical values: 'good' is 1 and 'bad' is 0

Step 3: Data Transformation

Missing Values: The dataset doesn't contain missing values, so no transformation is necessary.

Step 4: Business Scenario

Let's say for example if we're working with a space agency monitoring ionospheric conditions using radar systems then accurately classifying radar returns as 'good' or 'bad' is crucial for determining ionospheric stability, which impacts satellite communications and navigation systems. Timely identification of 'bad' conditions allows for preventive measures, ensuring uninterrupted services.

Step 5: False Positives and False Negatives

In this context:

False Positive (FP): Classifying a 'bad' radar return as 'good'. This could lead to undetected ionospheric disturbances, potentially disrupting satellite operations.

False Negative (FN): Classifying a 'good' radar return as 'bad'. This might result in unnecessary preventive actions, leading to operational inefficiencies.

Given the potential high costs associated with satellite disruptions, false positives are costlier than false negatives. I would assign a cost ratio of 4:1 for FP to FN, reflecting the higher impact of false positives.

This is because we would be better off predicting a failure when there was not one as it might only cause a minor disruption, but if we failed to predict a disruption then it would disturb all satelite activities and could result in huge losses.

Step 6: Cost Function

Now I will define a function to calculate the total cost based on false positives and false negatives:

```
def calculate_cost(y_actual, y_pred):

    fp_cost=6
    fn_cost=1
    y_actual = np.array(y_actual, dtype=int)
    y_pred = np.array(y_pred, dtype=int)
    # False Positives: Predicted 1 but actual is 0
    FP = sum((y_actual == 0) & (y_pred == 1))

    # False Negatives: Predicted 0 but actual is 1
    FN = sum((y_actual == 1) & (y_pred == 0))

    # Total cost calculation
    cost = (fp_cost * FP) + (fn_cost * FN)
    return cost
```

Step 7: Threshold Generation

Now I generate 100 candidate thresholds evenly spaced between 0 and 1:

```
thresholds = np.linspace(0, 1, 100)
```

Step 8: Cost Matrix Initialization

Initialize a 100x10 matrix to store costs for each threshold and fold:

```
out = np.zeros((100, 10))
```

Step 9: Cross-Validation Folds

Assign each data point to one of 10 cross-validation folds:

```
n = np.ceil(len(y) / 10)
fold_vec = np.concatenate([np.arange(1, 11)] * int(n))[:len(y)]
np.random.seed(247)
fold_vec = np.random.permutation(fold_vec)
```

Step 10: Logistic Regression and Cost Evaluation

Perform logistic regression and evaluate the prediction cost for each threshold using 10-fold cross-validation:

```
for i in range(10):
    test_data = np.where(fold_vec == i + 1)
    train_data = np.where(fold_vec != i + 1)
    label_map = {'g': 1, 'b': 0}

    # Split data into training and testing sets
    x_train = x.iloc[train_data]
    y_train = y.iloc[train_data]
    x_test = x.iloc[test_data]
    y_test = y.iloc[test_data]

    # Extract the label column if y_test is a DataFrame
    if isinstance(y_test, pd.DataFrame):
        y_test = y_test['Class']

    # Remove 'Class' and reset index
    y_test = y_test[y_test != 'Class']
    y_test = y_test.reset_index(drop=True)

    # Perform the logistic regression
    mod = LogisticRegression(max_iter=1000)
    mod.fit(x_train, y_train.values.ravel())

    # Predict probabilities
    y_pred_prob = mod.predict_proba(x_test)[:, 1]

    # Map y_test to numeric labels
    y_test = [label_map.get(label, -1) for label in y_test]

    # Evaluate predictions
    for j, threshold in enumerate(thresholds):
        y_pred = [1 if prob >= threshold else 0 for prob in y_pred_prob]
        # Uncomment below when calculate_cost is ready
        out[j, i] = calculate_cost(y_test, y_pred)
```

Step 11: Optimal Threshold Selection

Determine the threshold with the lowest average cost across folds:

```
mean_costs = out.mean(axis=1)
optimal_threshold_index = np.argmin(mean_costs)
optimal_threshold = thresholds[optimal_threshold_index]
optimal_cost = mean_costs[optimal_threshold_index]

print (optimal_cost, optimal_threshold)
```

```
15.2 0.8686868686868687
```

Step 12: Threshold Refinement

Since the threshold is around 0.86 and our cost ratio is 6:1 that would imply that we are much more likely to classify something as positive as opposed to negative, which is inline with our thought process of there being more cost attached to a false positive.