Name: Vikas Vijaykumar Bastewad

Roll No.: 20CS10073

Question 2 : Naive Bayes Classifier

Part A: Probability

Part 1: Rolling the 4-faced die 4 times for 1000 trials and calculating the sum of the upward face values.

In [ ]:
```
# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
```

In [ ]:
```
# Defining the parameters
k = 4  # Number of faces on the die
num_rolls = 4 # Number of times the die is rolled
num_simulations = 1000 # Number of times the experiment is simulated
```

In [ ]:
```
# Calculating probabilities for each face value
for i in range(1, k+1):
    if i == 1:
        probabilities = [1 / (2 ** (k - 1))]
    else:
        probabilities.append(1 / (2 ** (i - 1)))
```

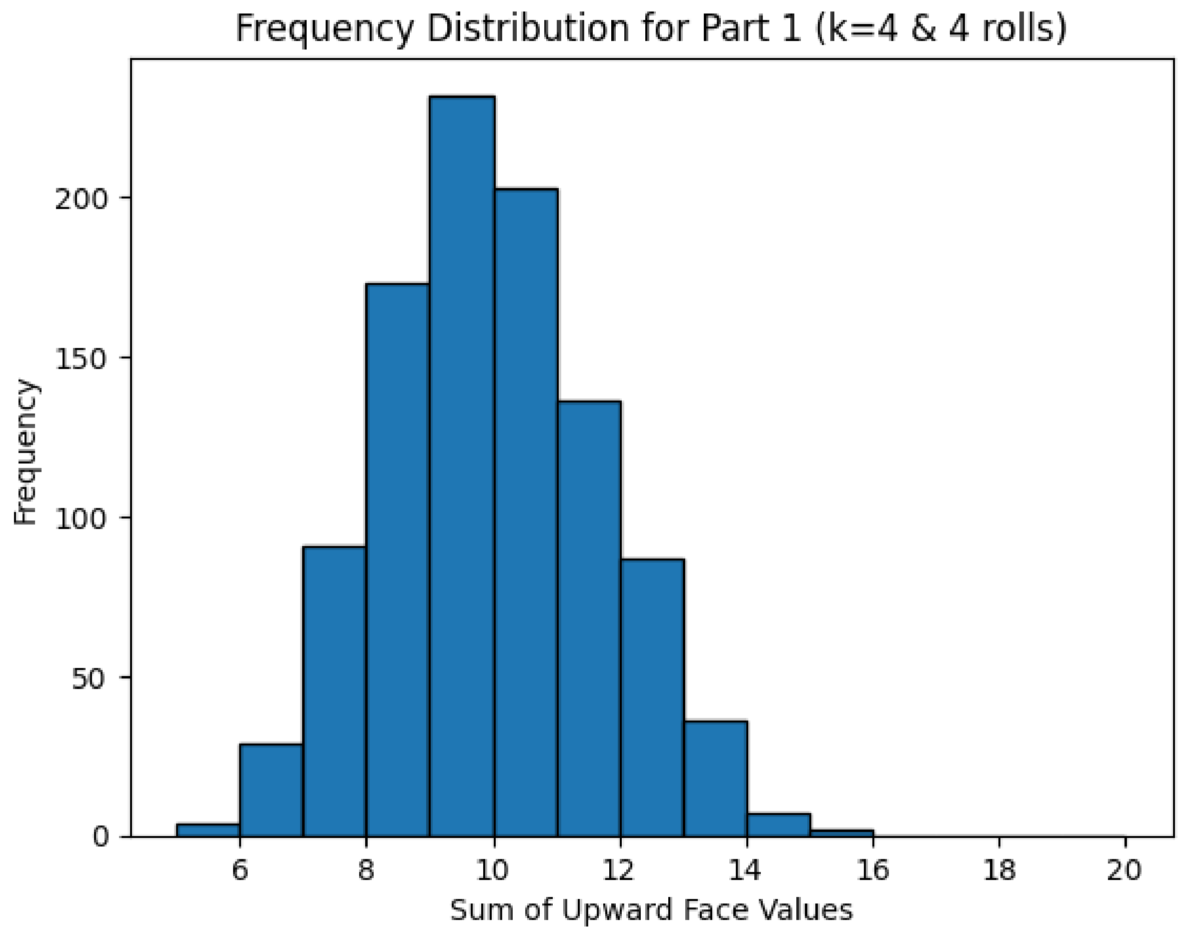
In [ ]:
```
# Normalizing probabilities to ensure they sum up to 1
probabilities /= np.sum(probabilities)
```

In [ ]:
```
# Simulating rolling the 4-faced die 4 times for 1000 trials
results_part1 = []      # List to store the results
for _ in range(num_simulations):
    rolls = np.random.choice(np.arange(1, k+1), size=num_rolls, p=probabilities)    # Rolling the die
    results_part1.append(np.sum(rolls))   # Summing up the results of the 4 rolls
```

In [ ]:
```
# Calculating the theoretical expected sum
expected_sum_part1 = (num_rolls)*(1 * (1 / (2 ** (k - 1))) + 2 * (1 / (2 ** (2 - 1))) + 3 * (1 / (2 *
```

In [ ]:
```
# Calculating actual mean and ploting the frequency distribution histogram
mean_part1 = np.mean(results_part1)         # Calculating the mean of the results obtained from the s
print('Part 1 results:')
print(f'Theoretical Expected Sum : {expected_sum_part1:.4f}')
print(f'Actual Mean of Sum : {mean_part1:.4f}')
print("-" * 40)
plt.hist(results_part1, bins=range(5, 21), edgecolor='black')
plt.xlabel('Sum of Upward Face Values')
plt.ylabel('Frequency')
plt.title('Frequency Distribution for Part 1 (k=4 & 4 rolls)')
plt.show()
```

```
Part 1 results:
Theoretical Expected Sum : 9.5000
Actual Mean of Sum : 9.4690
----------------------------------------
```



In this part, a 4-faced biased die was rolled 4 times in each simulation. The theoretical expected sum was calculated to be 9.5000, and the actual mean of the sum obtained from the simulations was 9.4690. The actual mean is very close to the theoretical expected sum, indicating that the simulation results align well with the theoretical calculations.

Part 2: Rolling the 4-faced die 8 times for 1000 trials and calculating the sum of the upward face values.

In [ ]:

```
# Redifiing the parameters for part 2
k = 4
```

```
num_rolls = 8
num_simulations = 1000
```

In [ ]:
```
# Calculating the probabilities for each face
for i in range(1, k+1):
    if i == 1:
        probabilities = [1 / (2 ** (k - 1))]
    else:
        probabilities.append(1 / (2 ** (i - 1)))
```

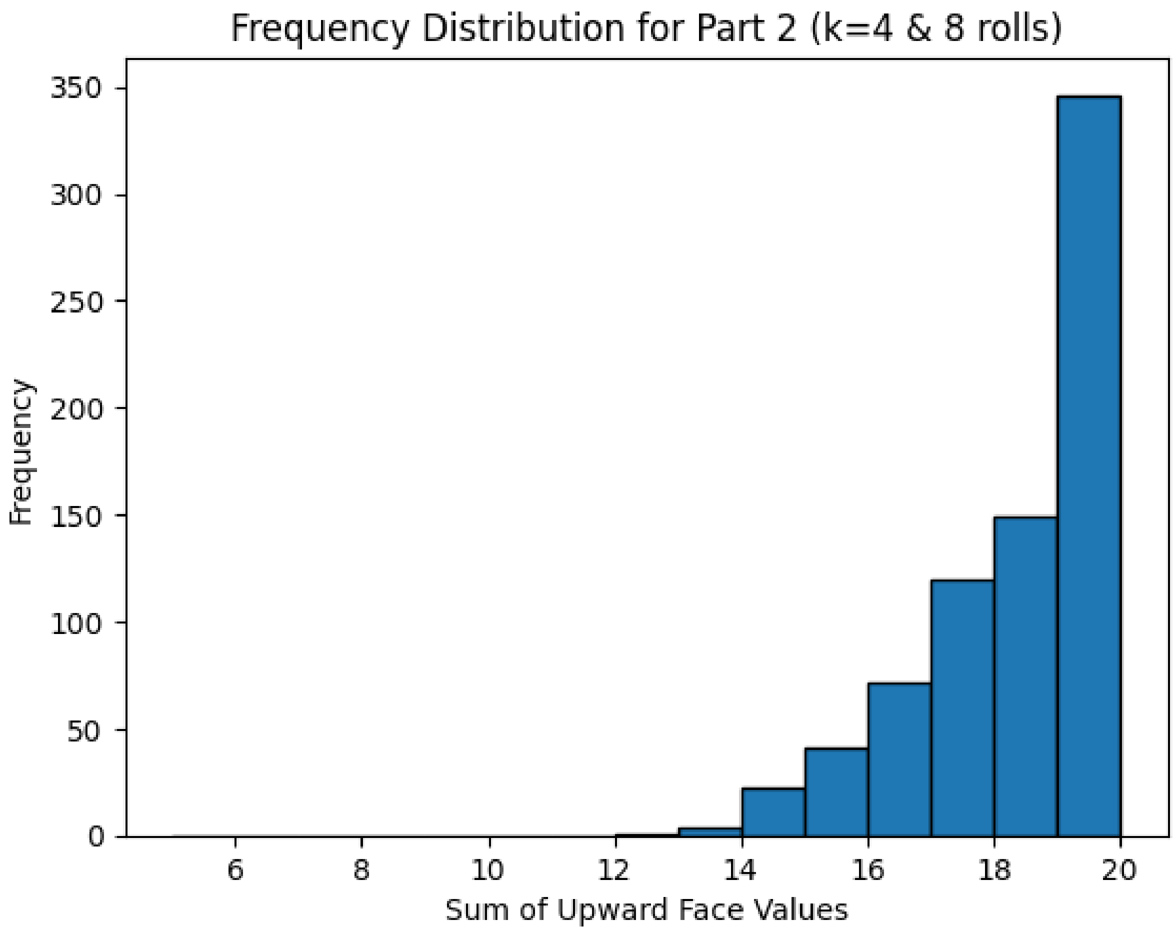
In [ ]:
```
# Normalizing probabilities to ensure they sum up to 1
probabilities /= np.sum(probabilities)
```

In [ ]:
```
# Simulating rolling the 4-faced die 8 times for 1000 trials
results_part2 = []                                                              # List to store t
for _ in range(num_simulations):
    rolls = np.random.choice(np.arange(1, k+1), size=num_rolls, p=probabilities)    # Rolling the die
    results_part2.append(np.sum(rolls))                                         # Summing up the
```

In [ ]:
```
# Calculating theoretical expected sum
expected_sum_part2 = (num_rolls)*(1 * (1 / (2 ** (4 - 1))) + 2 * (1 / (2 ** (2 - 1))) + 3 * (1 / (2 *
```

In [ ]:
```
# Calculate actual mean and plot frequency distribution histogram
mean_part2 = np.mean(results_part2)
print('Part 2 results:')
print(f'Theoretical Expected Sum : {expected_sum_part2:.4f}')
print(f'Actual Mean of Sum : {mean_part2:.4f}')

plt.hist(results_part2, bins=range(5, 21), edgecolor='black')
plt.xlabel('Sum of Upward Face Values')
plt.ylabel('Frequency')
plt.title('Frequency Distribution for Part 2 (k=4 & 8 rolls)')
plt.show()
```

```
Part 2 results:
Theoretical Expected Sum : 19.0000
Actual Mean of Sum : 19.0260
```



For this part, a 4-faced biased die was rolled 8 times in each simulation. The theoretical expected sum was 19.0000, and the actual mean of the sum obtained from the simulations was 19.0260. Similar to Part 1, the actual mean is very close to the theoretical expected sum, demonstrating that the simulation results are in good agreement with the theoretical calculations.

Part 3: Rolling the 16-faced die 4 times for 1000 trials and calculating the sum of the upward face values.

In [ ]:
```
# Redefining parameters for part 3
k = 16
```

```
num_rolls = 4
num_simulations = 1000
```

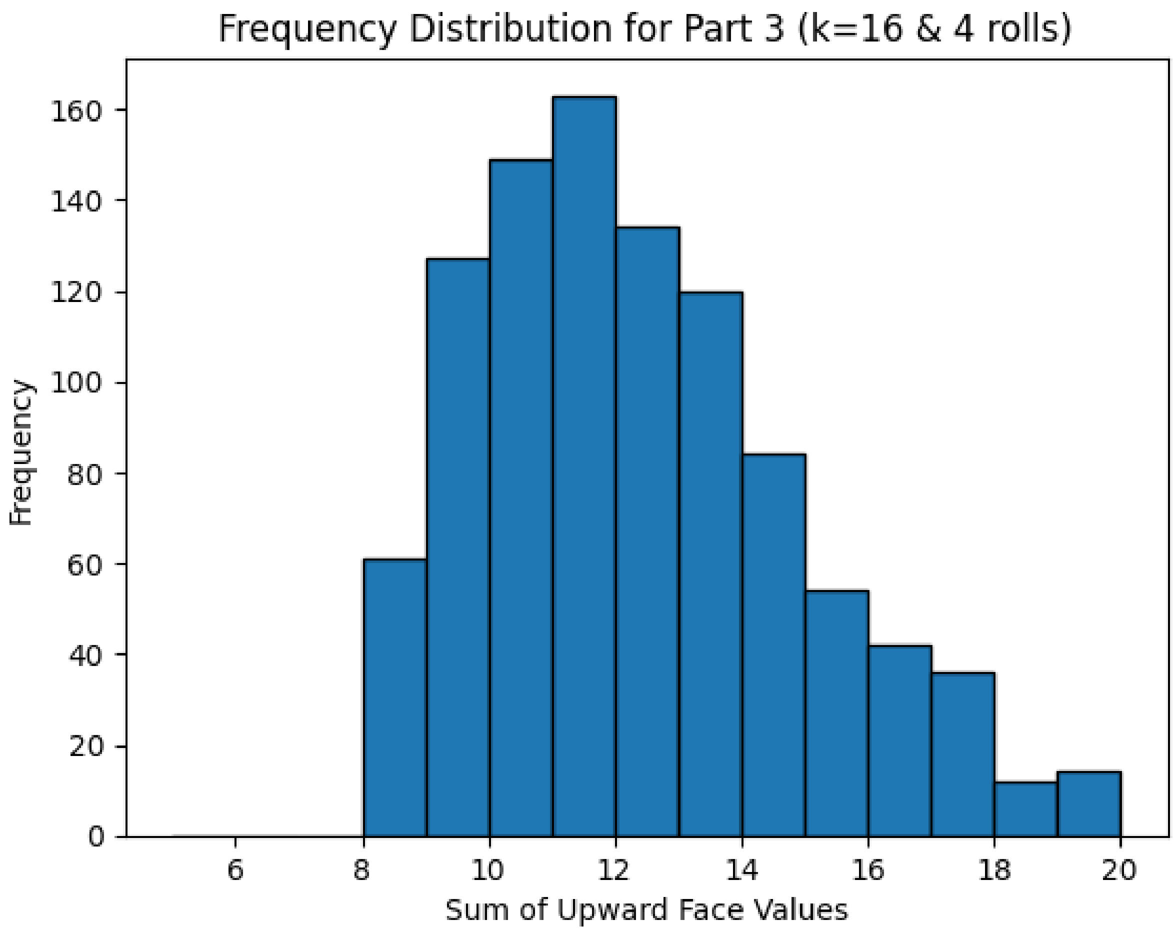
In [ ]:
```
# Calculating probabilities for each face
for i in range(1, k+1):
    if i == 1:
        probabilities = [1 / (2 ** (k - 1))]
    else:
        probabilities.append(1 / (2 ** (i - 1)))

# Normalizing the probabilities to ensure they sum up to 1
probabilities /= np.sum(probabilities)
```

In [ ]:
```
# Simulating rolling the 16-faced die 4 times for 1000 trials
results_part3 = []                                                                  # List to sto
for _ in range(num_simulations):
    rolls = np.random.choice(np.arange(1, k+1), size=num_rolls, p=probabilities)        # Rolling the
    results_part3.append(np.sum(rolls))                                                # Summing up
```

In [ ]:
```
# Calculating hte theoretical expected sum
expected_sum_part3 = (num_rolls)*(1 * (1 / (2 ** (k - 1))) + 2 * (1 / (2 ** (2 - 1))) + 3 * (1 / (2 *
```

In [ ]:
```
# Calculating actual mean and ploting frequency distribution histogram
mean_part3 = np.mean(results_part3)         # Calculating the mean of the results obtained from the s
print('Part 3 results:')
print(f'Theoretical Expected Sum : {expected_sum_part3:.4f}')
print(f'Actual Mean of Sum : {mean_part3:.4f}')

plt.hist(results_part3, bins=range(5, 21), edgecolor='black')
plt.xlabel('Sum of Upward Face Values')
plt.ylabel('Frequency')
plt.title('Frequency Distribution for Part 3 (k=16 & 4 rolls)')
plt.show()
```

```
Part 3 results:
Theoretical Expected Sum : 9.0001
Actual Mean of Sum : 11.9260
```



In this part, a 16-faced biased die was rolled 4 times in each simulation. The theoretical expected sum was 9.0001, and the actual mean of the sum obtained from the simulations was 11.9260. The actual mean deviates more from the theoretical expected sum in this case. This discrepancy could be due to the increased complexity introduced by the higher number of faces on the die, leading to a larger variance in the simulation results.

Part B : Implementation of Naive Bayes (From Scratch)

In [ ]:
```python
# Importing required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from ucimlrepo import fetch_ucirepo
# Importing Scikit learn
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

2. Loading Dataset: Load the data with a 70:15:15 split for train, validation, and testing (You may use sklearn for this part).

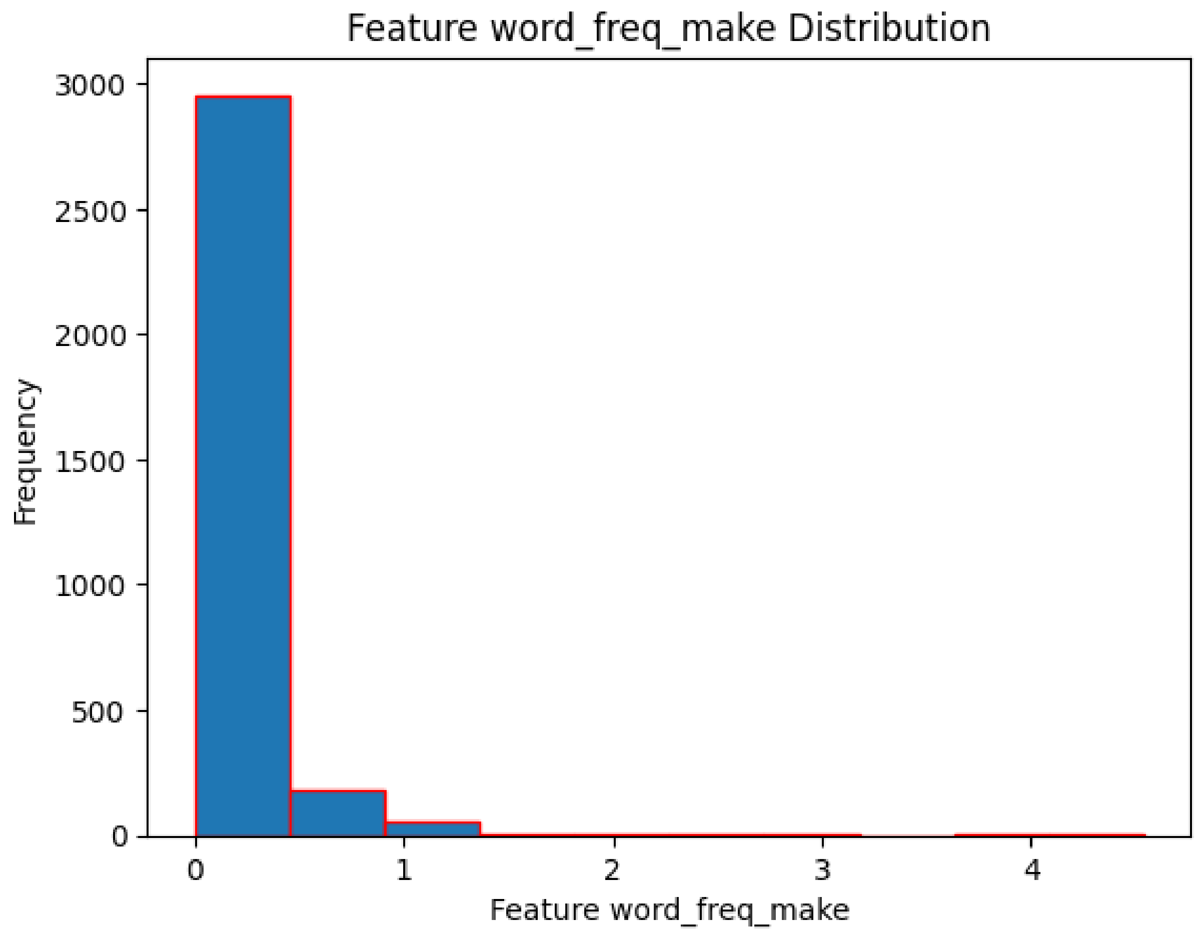
In [ ]:
```python
spambase = fetch_ucirepo(id=94)
X = spambase.data.features
y = spambase.data.targets.values.ravel()

# Spliting the dataset into training, validation and testing sets (70:15:15)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)
```

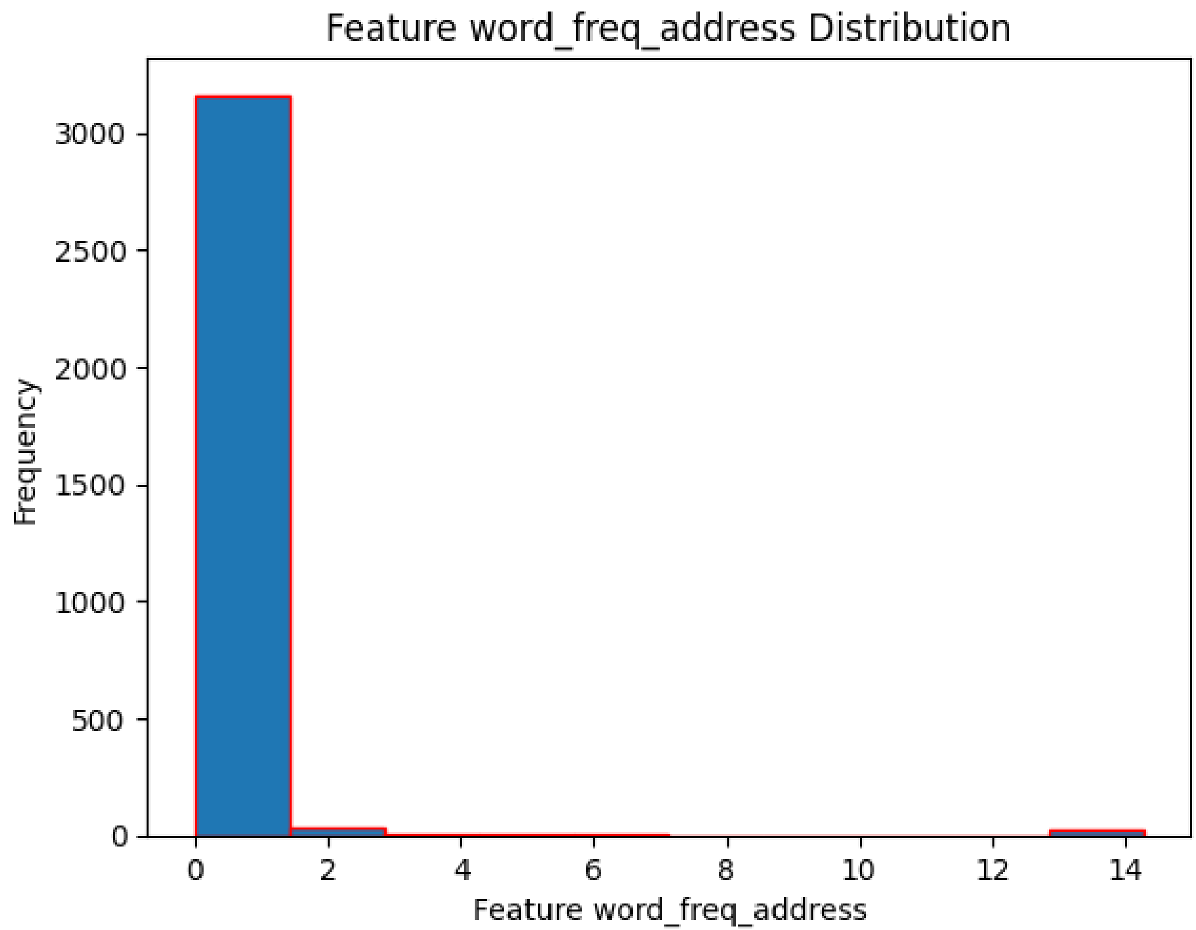
3. Plot Distribution: Choose some 5 columns from the dataset and plot the probability

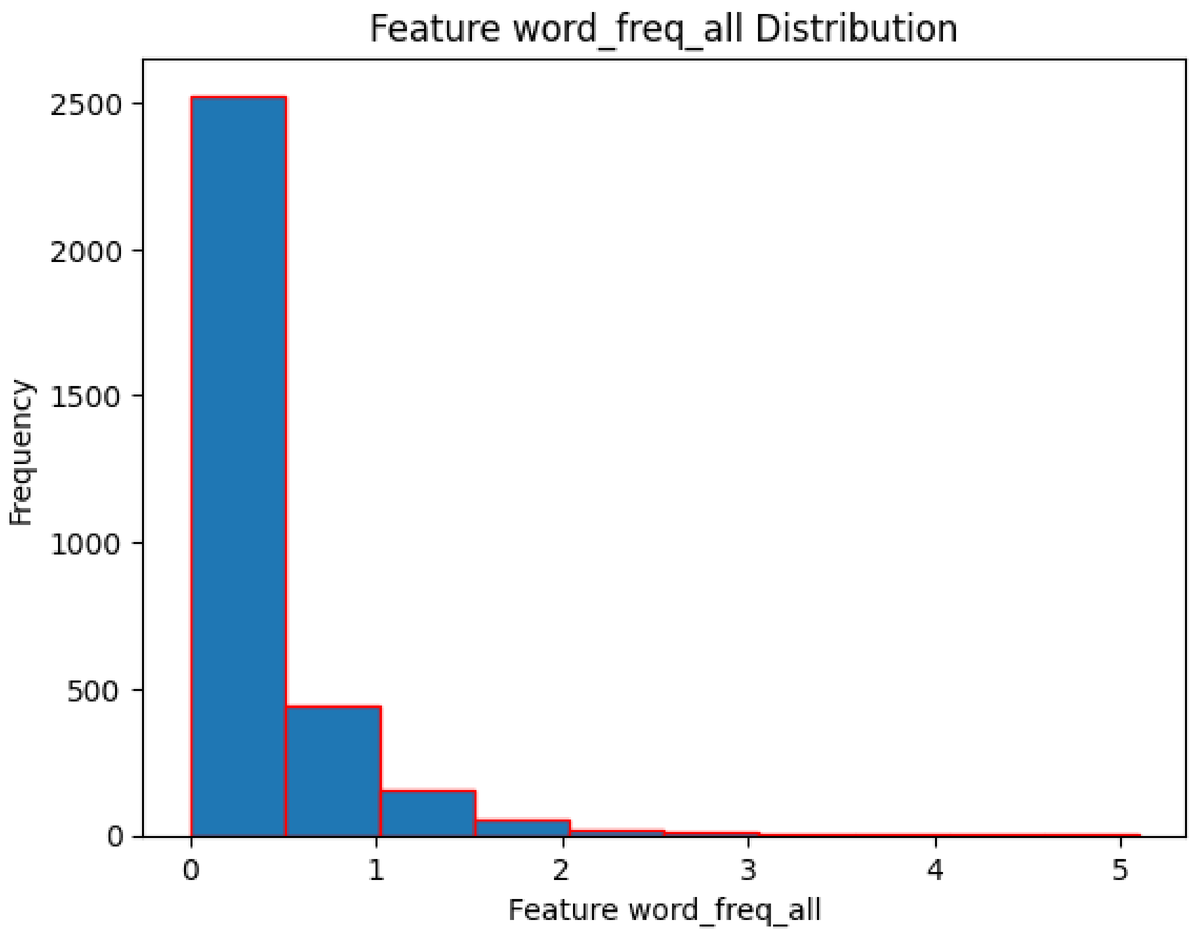
distribution.

In [ ]:
```python
selected_columns = X_train.iloc[:, :5]  # Selecting first 5 columns
print(selected_columns.columns)         # Printing column names
for i in range(selected_columns.shape[1]):  # Plotting histogram for each column
    plt.hist(selected_columns.iloc[:, i], bins=10, edgecolor='red')
    plt.xlabel(f'Feature {selected_columns.columns[i]}')
    plt.ylabel('Frequency')
    plt.title(f'Feature {selected_columns.columns[i]} Distribution')
    plt.show()
```

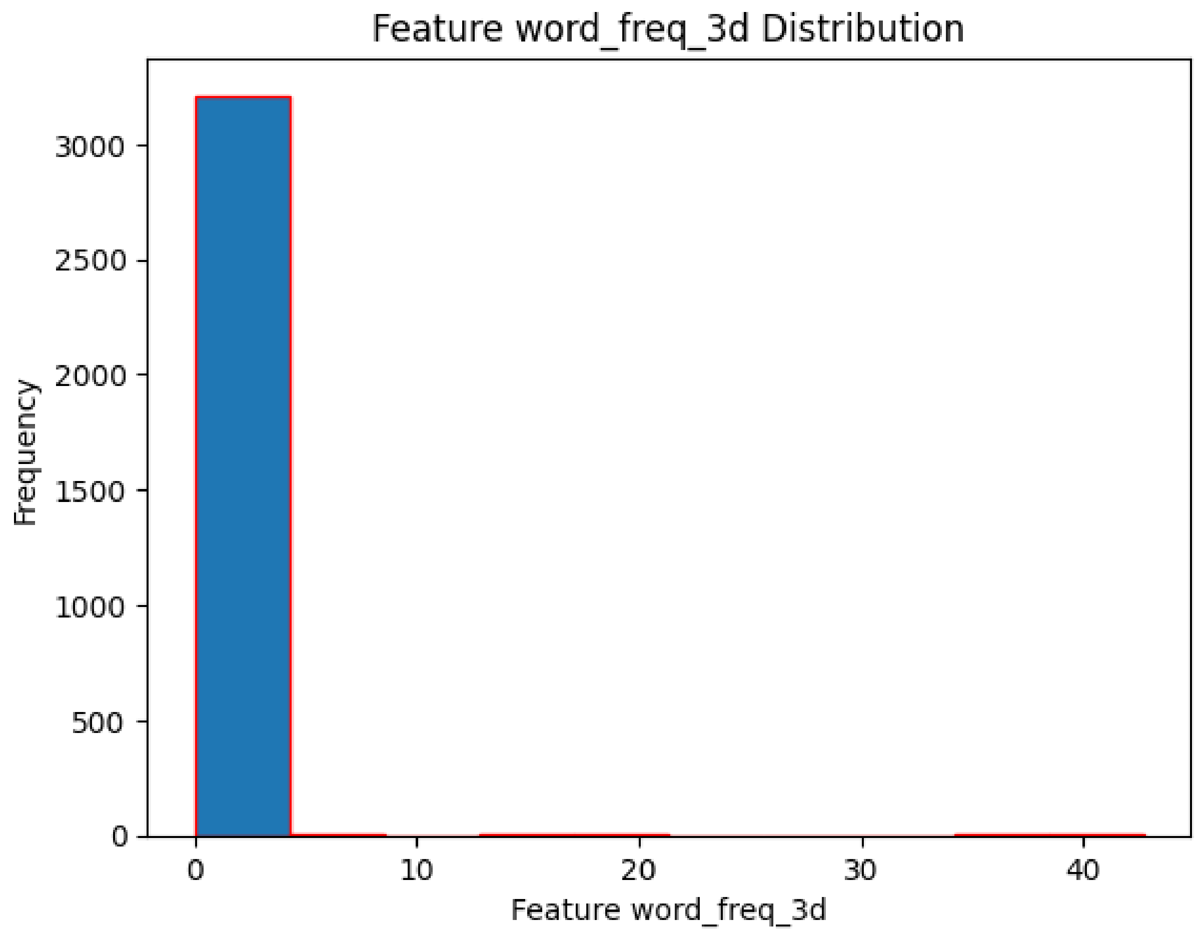
```
Index(['word_freq_make', 'word_freq_address', 'word_freq_all', 'word_freq_3d',
       'word_freq_our'],
      dtype='object')
```

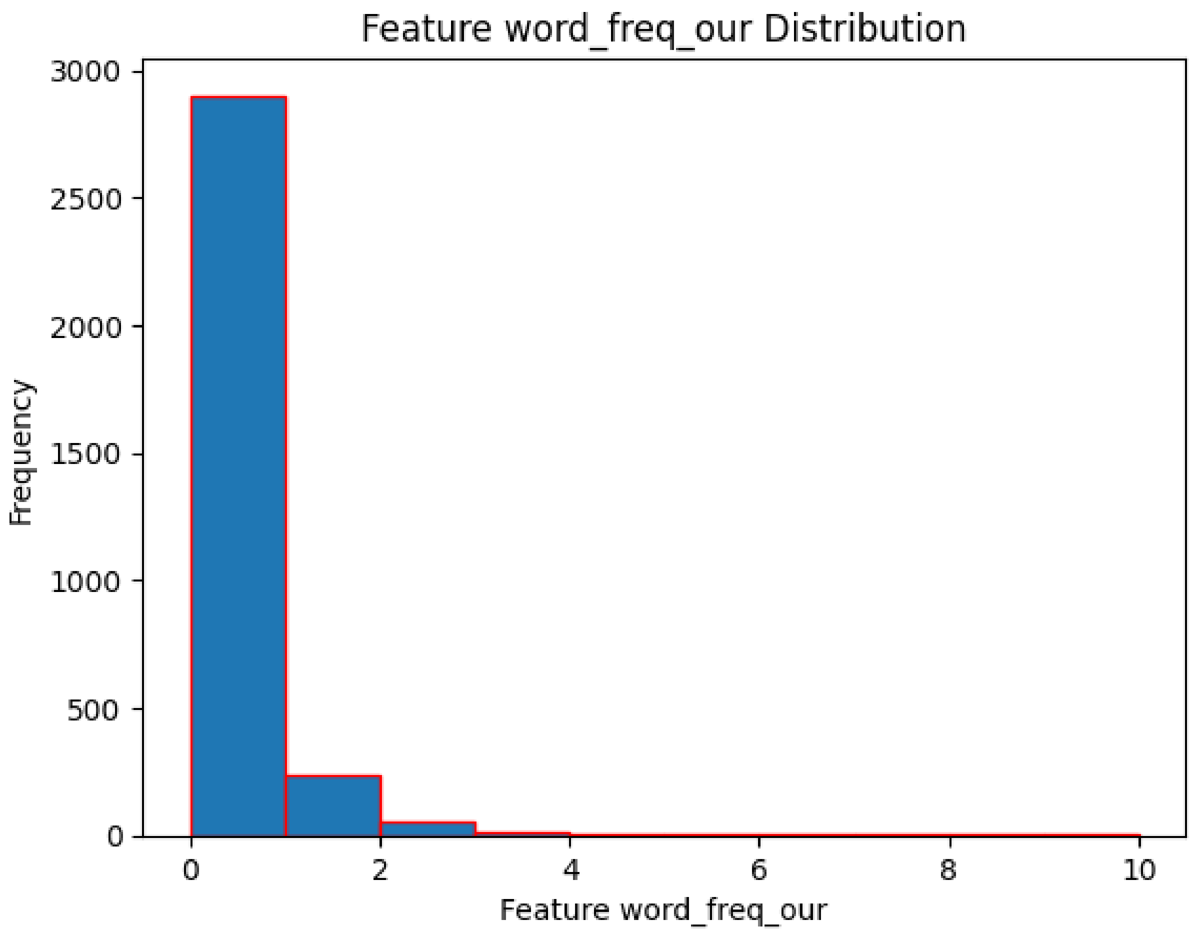

Feature word_freq_make Distribution
3000
2500
2000
1500
1000
500
0
Frequency
0
1
2
3
4
Feature word_freq_make


Feature word_freq_address Distribution
Frequency
0
500
1000
1500
2000
2500
3000
0
2
4
6
8
10
12
14
Feature word_freq_address



Feature word_freq_all Distribution
Frequency
0
500
1000
1500
2000
2500
0
1
2
3
4
5
Feature word_freq_all


Feature word_freq_3d Distribution
Frequency
3000
2500
2000
1500
1000
500
0
0
10
20
30
40
Feature word_freq_3d



4. Priors: Calculate and print the priors of classes.

In [ ]:
```
# Calculating the priors of classes
def priors(y):
    unique, counts = np.unique(y, return_counts=True)
    priors = dict(zip(unique, counts / len(y)))
    return priors

# printing the priors of classes
print(f'Priors of classes: {priors(y_train)}')
```

```
Priors of classes: {0: 0.6161490683229813, 1: 0.3838509316770186}
```

5. Train Model: Implement the Naive Bayes algorithm from scratch

In [ ]:

```
# Naive Bayes Classifier
class NaiveBayes:
    def __init__(self): # Initializing the model
        self.priors = None
        self.means = None
        self.variances = None
        self.classes = None

    def fit(self, X, y): # Fitting the model
        self.classes = np.unique(y)
        self.priors = priors(y)
        self.means = np.zeros((len(self.classes), X.shape[1]))
        self.variances = np.zeros((len(self.classes), X.shape[1]))

        for i, c in enumerate(self.classes): # Calculating the means and variances of each class
            X_c = X[y == c]
            self.means[i, :] = X_c.mean(axis=0)
            self.variances[i, :] = X_c.var(axis=0)

    def predict(self, X): # Predicting the model
        y_pred = []
        for _, sample in X.iterrows():  # Calculating the likelihood of each class
            class_probs = []
            for class_label, class_prob in self.priors.items(): # Calculating the likelihood of each
                feature_probs = self._pdf(class_label, sample)
                class_likelihood = 1
                for feature_prob in feature_probs:
                    class_likelihood *= feature_prob
                class_probs.append(class_likelihood * class_prob)
            y_pred.append(self.classes[np.argmax(class_probs)]) # Predicting the class with maximum 
        return y_pred


    def _pdf(self, class_idx, x):       # helper function to calculate the likelihood of each feature
        mean = self.means[class_idx]
```

```
        var = self.variances[class_idx]
        numerator = np.exp(-(x - mean) ** 2 / (2 * var))
        denominator = np.sqrt(2 * np.pi * var)
        return numerator / denominator
```

5.1 mention the total number of parameters needed to be stored for the model.

Total number of parameters = 2 * (number of classes) * (number of features) + (number of classes)

For this dataset, we have 2 classes and 57 features. Hence, total number of parameters = 2 * 2 * 57 + 2 = 232

6. Prediction and Evaluation: Implement functions to generate predictions on the test set and calculate accuracy, precision, recall, and F1-score for the Naive Bayes model.

In [ ]:
```
# Creating the Naive Bayes classifier
nb = NaiveBayes()
nb.fit(X_train, y_train)    # Fitting the model
```

In [ ]:
```
y_pred = nb.predict(X_test) # Predicting the model
```

In [ ]:
```
# Calculating the accuracy
accuracy = np.sum(y_pred == y_test) / len(y_test)
print(f'Accuracy: {accuracy}')

# Calculating the precision
precision = precision_score(y_test, y_pred)
print(f'Precision: {precision}')

# Calculating the recall
recall = recall_score(y_test, y_pred)
print(f'Recall: {recall}')

# Calculating the f1-score
f1 = f1_score(y_test, y_pred)
print(f'F1-Score: {f1}')
```

```
Accuracy: 0.41389290882778584
Precision: 0.41389290882778584
Recall: 1.0
F1-Score: 0.5854657113613102
```

7. Log Transformation: Apply log transformation to all the columns of the dataset. Then again train the Naive Bayes Classifier and do the evaluations the same as earlier. (Note: Train/Test splits remain the same)

In [ ]:
```python
# Applying Log transformation
X_train_log = np.log(X_train + 1)
X_test_log = np.log(X_test + 1)
```

In [ ]:
```python
# Create Naive Bayes classifier
nb = NaiveBayes()
nb.fit(X_train_log, y_train)    # Fitting the model
```

In [ ]:
```python
# Predicting the model
y_pred = nb.predict(X_test_log)
```

In [ ]:
```python
# Calculating the accuracy
accuracy = np.sum(y_pred == y_test) / len(y_test)
print(f'Accuracy: {accuracy}')

# Calculating the precision
precision = precision_score(y_test, y_pred)
print(f'Precision: {precision}')

# Calculating the recall
recall = recall_score(y_test, y_pred)
print(f'Recall: {recall}')

# Calculating the f1-score
f1 = f1_score(y_test, y_pred)
print(f'F1-Score: {f1}')
```

```
Accuracy: 0.41389290882778584
Precision: 0.41389290882778584
Recall: 1.0
F1-Score: 0.5854657113613102
```

8. Discuss: Explain the changes you noticed in the results before and after modifying the dataset.

The accuracy, precision, recall and f1-score values are same before and after log transformation. The log transformation is used to convert skewed data to normal distribution. The data in this dataset is already in normal distribution. So, the log transformation does not affect the results.

Part C: Implemention of Naive Bayes (sklearn)

In [ ]:
```python
# Importing the libraries
from ucimlrepo import fetch_ucirepo
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import roc_curve, auc, accuracy_score
import matplotlib.pyplot as plt
import numpy as np
```

In [ ]:
```python
# Fetching the dataset from UCIML repo
spambase = fetch_ucirepo(id=94)
```

In [ ]:
```python
# Data (as pandas dataframes)
X = spambase.data.features
y = spambase.data.targets.values.ravel()

# Spliting the data into train, validation, and test sets (70:15:15 split)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)
```

In [ ]:
```
# Training the Gaussian Naive Bayes model
nb_model = GaussianNB()     # Instantiating the model
nb_model.fit(X_train, y_train)     # Fitting the model on the training data
```

Out[ ]:
```
▾ GaussianNB
GaussianNB()
```

In [ ]:
```
# Training Gaussian Naive Bayes model after log transformation of the data
X_train_log = np.log1p(X_train)  # Applying log transformation
X_val_log = np.log1p(X_val)  # Applying log transformation
nb_model_log = GaussianNB() # Instantiating the model
nb_model_log.fit(X_train_log, y_train)  # Fitting the model on the training data
```

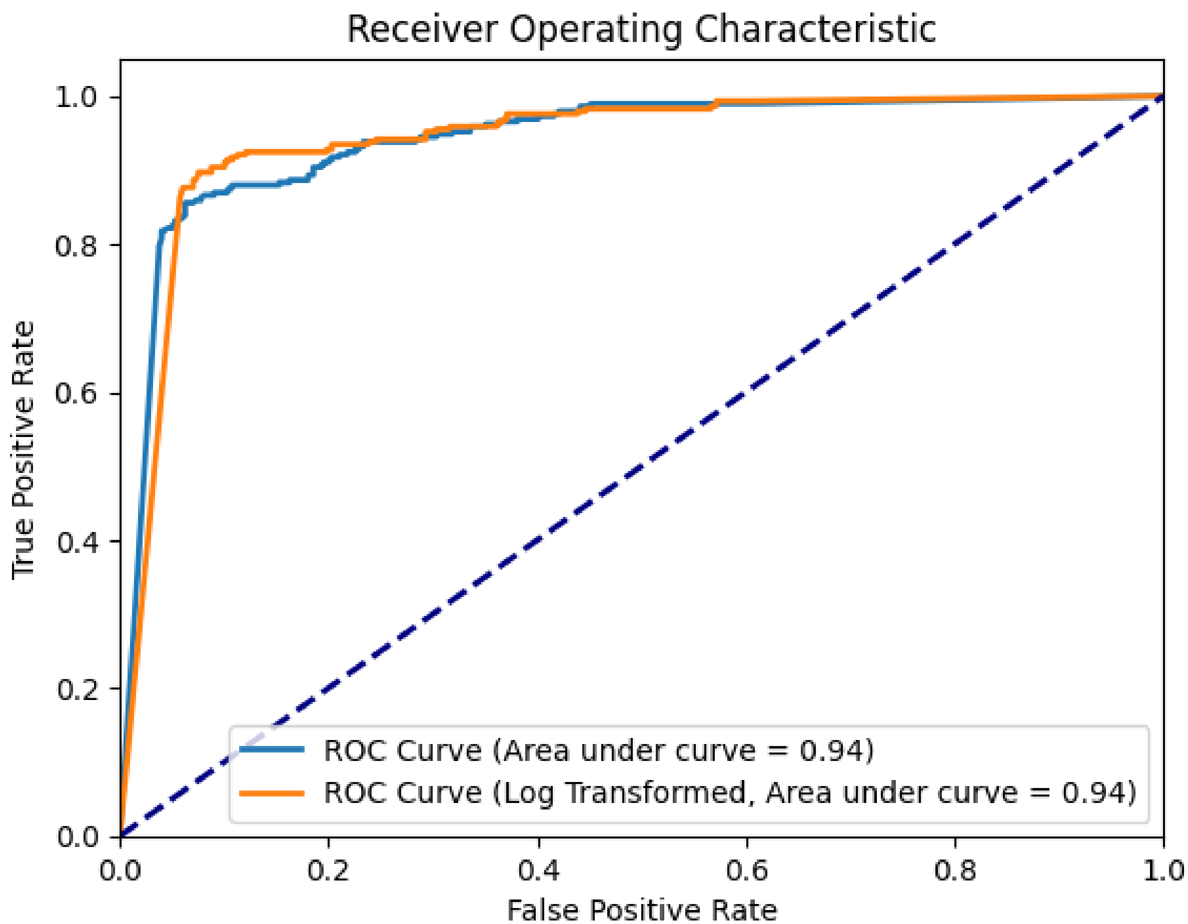
Out[ ]:
```
▾ GaussianNB
GaussianNB()
```

In [ ]:
```
# Predicting the probabilities for validation set
y_prob = nb_model.predict_proba(X_val)[:, 1]
y_prob_log = nb_model_log.predict_proba(X_val_log)[:, 1]
```

In [ ]:
```
# Calculating the ROC curve and AUC for both models
fpr, tpr, _ = roc_curve(y_val, y_prob)
fpr_log, tpr_log, _ = roc_curve(y_val, y_prob_log)
roc_auc = auc(fpr, tpr)
roc_auc_log = auc(fpr_log, tpr_log)
```

In [ ]:
```
# Ploting the ROC curves
plt.figure()
plt.plot(fpr, tpr, lw=2, label=f'ROC Curve (Area under curve = {roc_auc:.2f})')
plt.plot(fpr_log, tpr_log, lw=2, label=f'ROC Curve (Log Transformed, Area under curve = {roc_auc_log
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')
```

```python
plt.title('Receiver Operating Characteristic')
plt.legend(loc='lower right')
plt.show()
```



In [ ]:

```python
# Choosing the best model based on ROC curve
best_model = nb_model_log if roc_auc_log > roc_auc else nb_model

# Predicting on test set and calculating the accuracy of the chosen model
X_test_log = np.log1p(X_test)  # Apply log transformation to test set features
y_pred = best_model.predict(X_test_log)  # Predict using the chosen model
accuracy = accuracy_score(y_test, y_pred)  # Calculate accuracy
print(f'Accuracy of the Naive bias model: {accuracy:.2f}')
```

```
Accuracy of the Naive bias model: 0.79
```

Naive Bayes Model:

```
    Accuracy: 0.79
```

SVM Models:

```
    Regularization parameter: 0.001
        Accuracy: 0.87

    Regularization parameter: 0.1
        Accuracy: 0.93

    Regularization parameter: 1
        Accuracy: 0.92

    Regularization parameter: 10
        Accuracy: 0.91

    Regularization parameter: 100
        Accuracy: 0.90
```

As it can be easily seen that the SVM model outperformed the Naive Bayes model in terms of accuracy. The SVM model with regularization parameter 0.1 performed the best with an accuracy of 0.93.