q-2

November 6, 2023

0.1 Imports

[9]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from ucimlrepo import fetch_ucirepo
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import␣
↪accuracy_score,precision_score,recall_score,f1_score,roc_curve,roc_auc_score
import matplotlib.pyplot as plt
```

0.2 (A) Probability

0.2.1 Random rolling of Dice

[184]:
```
def getRandomRolls(k:int,iter:int,iterLength=1):
    # k: types of outcomes
    # iter: number of rolls
    # return: list of rolls
    choices = np.arange(1,k+1)
    weights = [1/(2**(k-1))]
    for i in range(2,k+1):
        weights.append(1/(2**(i-1)))

    ans = []
    for i in range(0,iter):
        sum = 0
        for j in range(0,iterLength):
            sum += np.random.choice(choices, p=weights)
        ans.append(sum)
    return ans


## Calculation of theoretical expected value

def theoreticalExpectedValue(k:int,iterLength=1):
    choices = np.arange(1,k+1)
```

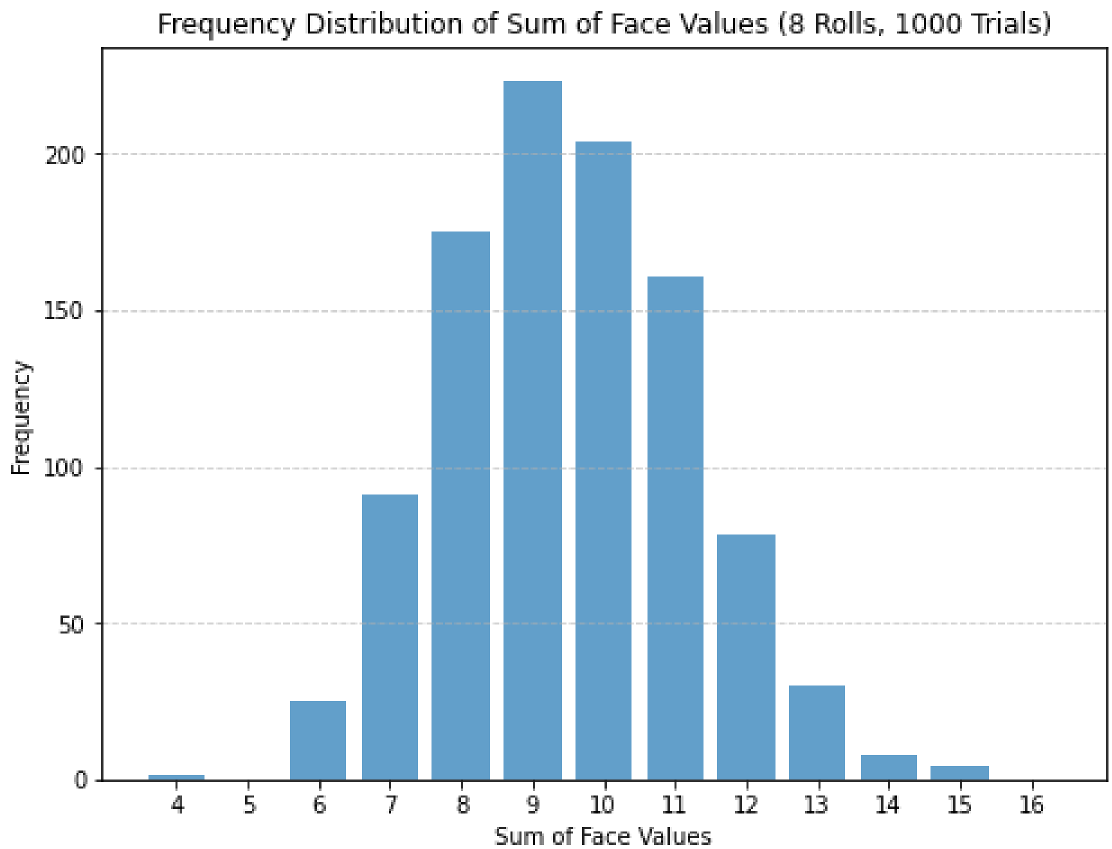
```
    weights = [1/(2**(k-1))]
    for i in range(2,k+1):
        weights.append(1/(2**(i-1)))
    ans = 0
    for i in range(0,k):
        ans += weights[i]*choices[i]
    return ans*iterLength
```

0.2.2 (a) K = 4 | Rolls = 4

[1]:
```
import random
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter

k=4
dice=list(range(k))
nums=list(range(k))
dice[0]=1/(2**(k-1))
for i in range(1,k):
    dice[i]=1/2**(i)

num_trials = 1000
num_rolls = 4

# Simulate rolling the die 'num_rolls' times for 'num_trials' trials
results = np.random.choice(range(1, len(dice) + 1), size=(num_trials,␣
 ↪num_rolls), p=dice)
# print(results)

# Calculate the sum of face values for each trial
sum_of_faces = np.sum(results, axis=1)

expec=0
for s in sum_of_faces:
  expec=expec+s
expec=expec/1000
print(f"Practical expected sum:{expec}")

# Plot the frequency distribution histogram
plt.figure(figsize=(8, 6))
plt.hist(sum_of_faces, bins=np.arange(num_rolls, num_rolls * len(dice) + 2),␣
 ↪rwidth=0.8, align='left', alpha=0.7)
plt.xlabel('Sum of Face Values')
plt.ylabel('Frequency')
plt.title('Frequency Distribution of Sum of Face Values (8 Rolls, 1000 Trials)')
```

```python
plt.xticks(np.arange(num_rolls, num_rolls * len(dice) + 1))
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()

# Calculate theoretical expected sum
theoretical_expected_sum = num_rolls * np.dot(range(1, len(dice) + 1), dice)

# Calculate five-number summary
min_value = np.min(sum_of_faces)
q1 = np.percentile(sum_of_faces, 25)
median = np.median(sum_of_faces)
q3 = np.percentile(sum_of_faces, 75)
max_value = np.max(sum_of_faces)

# Print results
print("Theoretical Expected Sum:", theoretical_expected_sum)
print("Minimum Value:", min_value)
print("Q1 (25th percentile):", q1)
print("Median (50th percentile):", median)
print("Q3 (75th percentile):", q3)
print("Maximum Value:", max_value)
```

```
Practical expected sum:9.507
```




```
Theoretical Expected Sum: 9.5
Minimum Value: 4
Q1 (25th percentile): 8.0
Median (50th percentile): 9.0
Q3 (75th percentile): 11.0
Maximum Value: 15
```

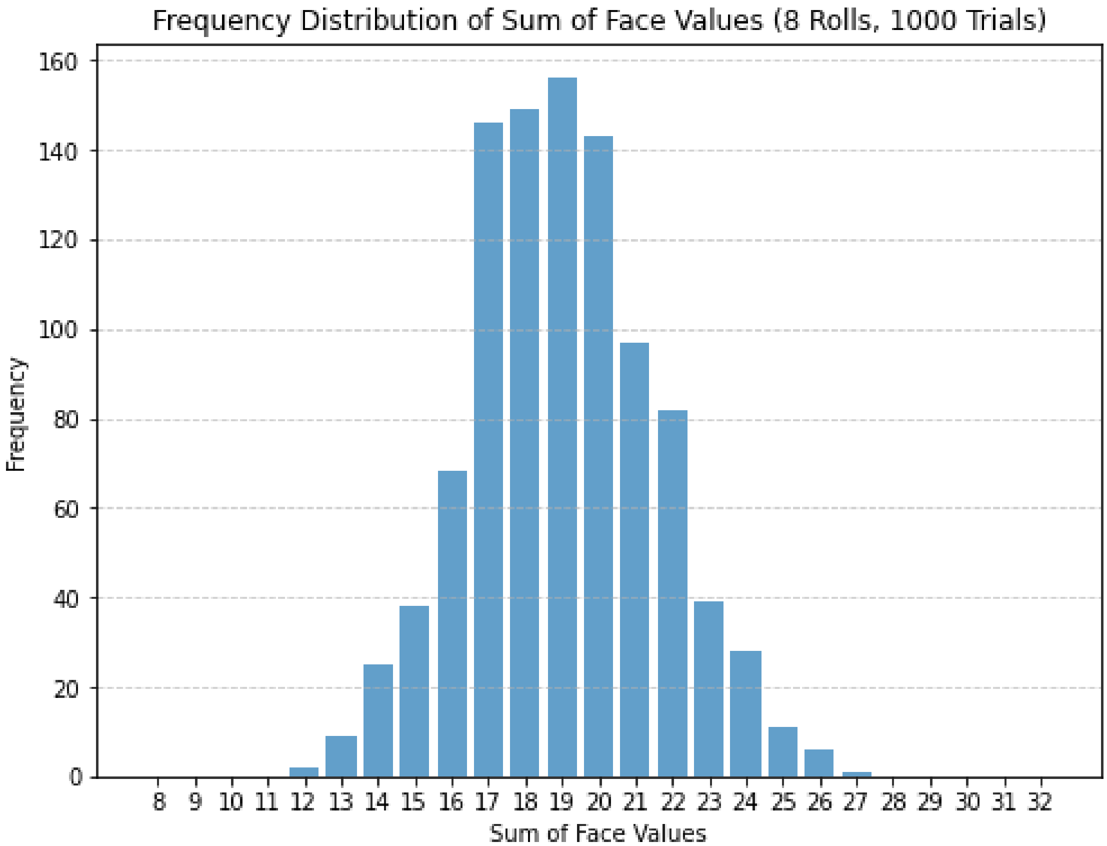
$$\mathbb{E}[X] = \sum_{x \in \mathcal{X}} x \cdot P(X = x)$$
$$E(X) = 1 \times 0.125 + 2 \times 0.5 + 3 \times 0.25 + 4 \times 0.125$$
$$E(X) = 0.125 + 1.0 + 0.75 + 0.5$$
$$E(X) = 2.375$$
$$ans = 4(E(X))$$
$$ans = 9.5$$

0.2.3 (b) K = 4 | Rolls = 8

[2]:
```
# Constants
num_trials = 1000
num_rolls = 8

# Simulate rolling the die 'num_rolls' times for 'num_trials' trials
results = np.random.choice(range(1, len(dice) + 1), size=(num_trials,␣
↪num_rolls), p=dice)
# print(results)

# Calculate the sum of face values for each trial
sum_of_faces = np.sum(results, axis=1)

expec=0
for s in sum_of_faces:
  expec=expec+s
expec=expec/1000
print(f"Practical expected sum:{expec}")

# Plot the frequency distribution histogram
plt.figure(figsize=(8, 6))
plt.hist(sum_of_faces, bins=np.arange(num_rolls, num_rolls * len(dice) + 2),␣
↪rwidth=0.8, align='left', alpha=0.7)
plt.xlabel('Sum of Face Values')
plt.ylabel('Frequency')
plt.title('Frequency Distribution of Sum of Face Values (8 Rolls, 1000 Trials)')
plt.xticks(np.arange(num_rolls, num_rolls * len(dice) + 1))
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()

# Calculate theoretical expected sum
theoretical_expected_sum = num_rolls * np.dot(range(1, len(dice) + 1), dice)

# Calculate five-number summary
min_value = np.min(sum_of_faces)
q1 = np.percentile(sum_of_faces, 25)
median = np.median(sum_of_faces)
q3 = np.percentile(sum_of_faces, 75)
max_value = np.max(sum_of_faces)

# Print results
print("Theoretical Expected Sum:", theoretical_expected_sum)
print("Minimum Value:", min_value)
print("Q1 (25th percentile):", q1)
print("Median (50th percentile):", median)
print("Q3 (75th percentile):", q3)
```

```
print("Maximum Value:", max_value)
```

```
Practical expected sum:19.005
```



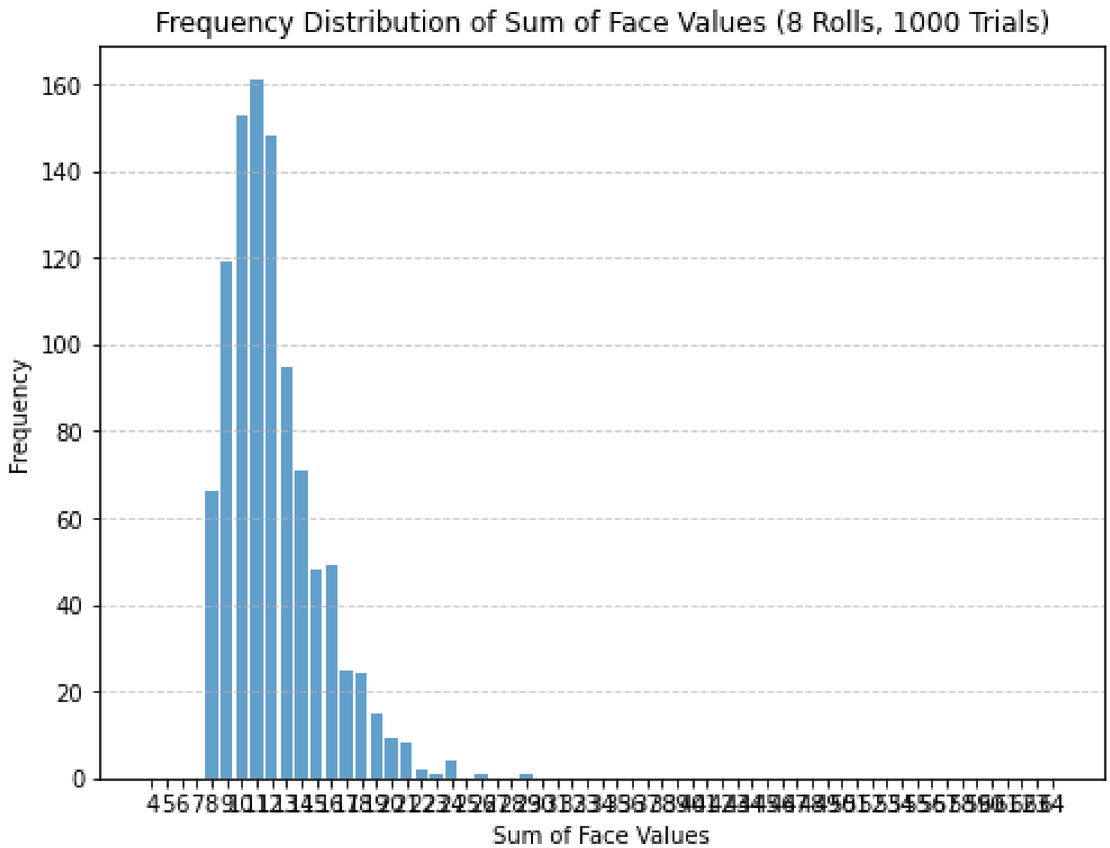

```
Theoretical Expected Sum: 19.0
Minimum Value: 12
Q1 (25th percentile): 17.0
Median (50th percentile): 19.0
Q3 (75th percentile): 21.0
Maximum Value: 27
```

0.2.4 (c) K = 16 | Rolls = 4

[3]:
```
k=16

dice=list(range(k))
nums=list(range(k))
dice[0]=1/(2**(k-1))
for i in range(1,k):
    dice[i]=1/2**(i)
```

```
num_trials = 1000
num_rolls = 4

# Simulate rolling the die 'num_rolls' times for 'num_trials' trials
results = np.random.choice(range(1, len(dice) + 1), size=(num_trials,␣
 ↪num_rolls), p=dice)
# print(results)

# Calculate the sum of face values for each trial
sum_of_faces = np.sum(results, axis=1)

expec=0
for s in sum_of_faces:
  expec=expec+s
expec=expec/1000
print(f"Practical expected sum:{expec}")

# Plot the frequency distribution histogram
plt.figure(figsize=(8, 6))
plt.hist(sum_of_faces, bins=np.arange(num_rolls, num_rolls * len(dice) + 2),␣
 ↪rwidth=0.8, align='left', alpha=0.7)
plt.xlabel('Sum of Face Values')
plt.ylabel('Frequency')
plt.title('Frequency Distribution of Sum of Face Values (8 Rolls, 1000 Trials)')
plt.xticks(np.arange(num_rolls, num_rolls * len(dice) + 1))
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()

# Calculate theoretical expected sum
theoretical_expected_sum = num_rolls * np.dot(range(1, len(dice) + 1), dice)

# Calculate five-number summary
min_value = np.min(sum_of_faces)
q1 = np.percentile(sum_of_faces, 25)
median = np.median(sum_of_faces)
q3 = np.percentile(sum_of_faces, 75)
max_value = np.max(sum_of_faces)

# Print results
print("Theoretical Expected Sum:", theoretical_expected_sum)
print("Minimum Value:", min_value)
print("Q1 (25th percentile):", q1)
print("Median (50th percentile):", median)
print("Q3 (75th percentile):", q3)
print("Maximum Value:", max_value)
```

```
Practical expected sum:12.117
```




```
Theoretical Expected Sum: 11.9979248046875
Minimum Value: 8
Q1 (25th percentile): 10.0
Median (50th percentile): 12.0
Q3 (75th percentile): 14.0
Maximum Value: 29
```

0.2.5 Results

The Theoretical and practical value expectation of the sum of dice are very similar

0.3 (B) Naive Bayes Implementation (Manual)

[10]:
```
# fetch dataset
spambase = fetch_ucirepo(id=94)
```

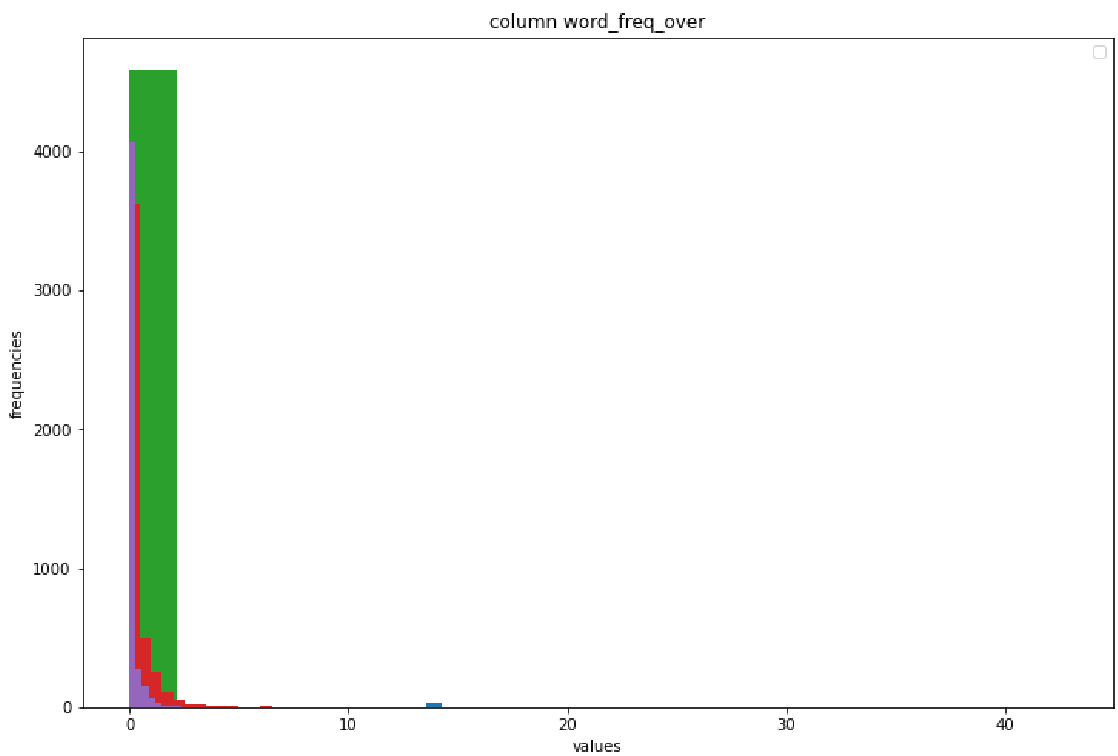
[11]:
```
# data (as pandas dataframes)
# loading as dataframe
X_df = spambase.data.features
```

```
X = X_df.to_numpy()
Y = spambase.data.targets.to_numpy()
Y = Y.ravel()
# scaler = StandardScaler()
# X = scaler.fit_transform(X)
```

[15]:
```
cols = [1,2,3,4,5]

columns = X_df.iloc[:,cols]

print('column distributions')

plt.figure(figsize=(12, 8))
for column in columns:
    plt.hist(columns[column],bins = 20)
    plt.title('column '+str(column))
    plt.xlabel('values')
    plt.ylabel('frequencies')
plt.legend()
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label
start with an underscore are ignored when legend() is called with no argument.
```

```
column distributions
```



[191]:
```
x_train, x_test, y_train, y_test = train_test_split(X, Y,test_size=0.3,␣
 ↪random_state=42)
x_val, x_test, y_val, y_test = train_test_split(x_test, y_test,test_size=0.5,␣
 ↪random_state=42)
```

[192]:
```
class naiveBayes():
    def __init__(self):
        self.bias = 1e-250
        pass
    def fit(self,x_train,y_train):
        self.Py = []
        self.priors = []
        self.features = x_train.shape[1]
        #count no of zeroes in y_train
        self.Py.append((y_train==0).sum()/len(y_train))
        self.Py.append((y_train==1).sum()/len(y_train))

        for i in range(0,self.features):
            X0 = [x_train[j][i] for j in range(0,len(y_train)) if y_train[j]==0]
            X1 = [x_train[j][i] for j in range(0,len(y_train)) if y_train[j]==1]

            mean0 = sum(X0)/len(X0)
            var0 = sum([(x-mean0)**2 for x in X0])/len(X0) + self.bias

            mean1 = sum(X1)/len(X1)
            var1 = sum([(x-mean1)**2 for x in X1])/len(X1) + self.bias
            self.priors.append( [[mean0,var0],[mean1,var1]] )
            # print(i,var0,var1)

    def gaussian(self,x,mean_var):
        mean = mean_var[0]
        var = mean_var[1]
        exponent = np.exp( -( ((x-mean)**2)/(2*var) ) )
        base = 1/((2*np.pi*var)**(0.5))
        ans = base*exponent + self.bias
        # print(ans)
        return np.log(ans)


    def predict(self,x_test):
        y_pred = []
        for i in range(0,len(x_test)):
            prob0 = np.log(self.Py[0])
            prob1 = np.log(self.Py[1])
```

```
            for j in range(0,self.features):
                prob0 += self.gaussian(x_test[i][j],self.priors[j][0])
                prob1 += self.gaussian(x_test[i][j],self.priors[j][1])
            if prob0 > prob1:
                y_pred.append(0)
            else:
                y_pred.append(1)
        return y_pred
```

[193]:
```
def getPerformanceMetrics(y_test, y_pred):
    accuracy = accuracy_score(y_test, y_pred)*100
    precision = precision_score(y_test, y_pred)*100
    recall = recall_score(y_test, y_pred)*100
    f1 = f1_score(y_test, y_pred)*100
    return [accuracy, precision, recall, f1]
```

[194]:
```
model = naiveBayes()
model.fit(x_train,y_train)
y_pred = model.predict(x_test)
model_metrics = getPerformanceMetrics(y_test, y_pred)

print("Metrics for the Naive Bayes model : ")
print("Accuracy: ",model_metrics[0])
print("Precision: ",model_metrics[1])
print("Recall: ",model_metrics[2])
print("F1 Score: ",model_metrics[3])
```

```
Metrics for the Naive Bayes model :
Accuracy:  71.49059334298119
Precision:  59.2901878914405
Recall:  99.3006993006993
F1 Score:  74.24836601307192
```

0.3.1 (b) Log Transforming the data

[195]:
```
bias = 1e-250
x_train_log = np.array([np.log(x_train[i]+bias) for i in range(0,len(x_train))])
x_test_log = np.array([np.log(x_test[i]+bias) for i in range(0,len(x_test))])
x_val_log = np.array([np.log(x_val[i]+bias) for i in range(0,len(x_val))])

model_log = naiveBayes()
model_log.fit(x_train_log,y_train)
y_pred_log = model_log.predict(x_test_log)
model_metrics_log = getPerformanceMetrics(y_test, y_pred_log)


print("Metrics for the Naive Bayes model after applying log transformation : ")
```

```
print("Accuracy: ",model_metrics_log[0])
print("Precision: ",model_metrics_log[1])
print("Recall: ",model_metrics_log[2])
print("F1 Score: ",model_metrics_log[3])
```

```
Metrics for the Naive Bayes model after applying log transformation :
Accuracy:  74.8191027496382
Precision:  62.5560538116592
Recall:  97.55244755244755
F1 Score:  76.22950819672131
```

0.3.2 Results

The performance of the model has increased after applying the log trasnformation of the data. Accuracy, Precision and F1 Score of the model are increased. Precision has slightly decreased but it is still very high at 97.5%.

Conclusion : Applying the log transformation of the given dataset increases the performance of the model

0.4 (C) Sklearn Implementation

[196]:
```
from sklearn.naive_bayes import GaussianNB
```

[197]:
```
model = GaussianNB()
model.fit(x_train,y_train)
y_pred = model.predict(x_test)
model_metrics = getPerformanceMetrics(y_test, y_pred)

print("Metrics for the Naive Bayes model : ")
print("Accuracy: ",model_metrics[0])
print("Precision: ",model_metrics[1])
print("Recall: ",model_metrics[2])
print("F1 Score: ",model_metrics[3])
```

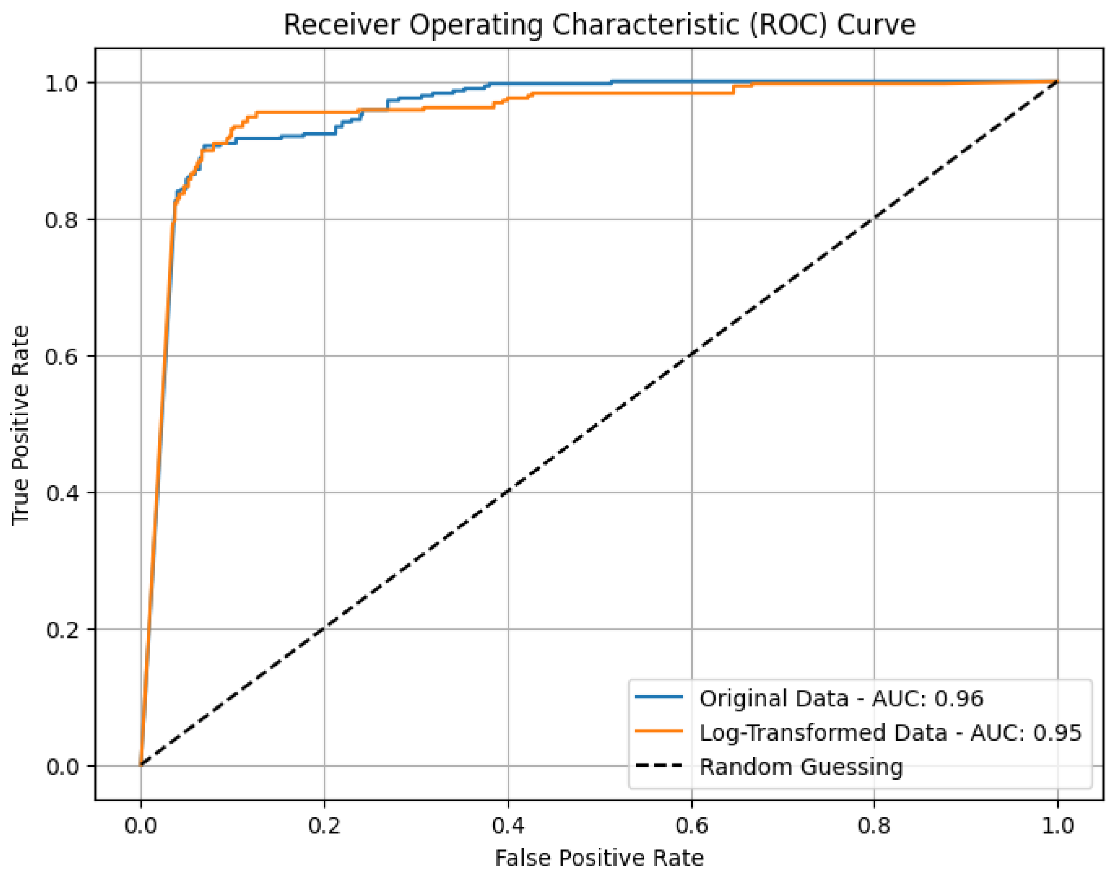
```
Metrics for the Naive Bayes model :
Accuracy:  82.77858176555716
Precision:  71.91601049868767
Recall:  95.8041958041958
F1 Score:  82.15892053973015
```

[198]:
```
model_log = GaussianNB()
model_log.fit(x_train_log,y_train)
y_pred_log = model_log.predict(x_test_log)
model_metrics_log = getPerformanceMetrics(y_test, y_pred)

print("Metrics for the Naive Bayes model after applying log transformation : ")
print("Accuracy: ",model_metrics_log[0])
```

```python
print("Precision: ",model_metrics_log[1])
print("Recall: ",model_metrics_log[2])
print("F1 Score: ",model_metrics_log[3])
```

```
Metrics for the Naive Bayes model after applying log transformation :
Accuracy:  82.77858176555716
Precision:  71.91601049868767
Recall:  95.8041958041958
F1 Score:  82.15892053973015
```

[199]:
```python
y_prob_original = model.predict_proba(x_test)[:, 1]
y_prob_log = model_log.predict_proba(x_test_log)[:, 1]

# Compute ROC curve and AUC for both models
fpr_original, tpr_original, _ = roc_curve(y_test, y_prob_original)
fpr_log, tpr_log, _ = roc_curve(y_test, y_prob_log)

auc_original = roc_auc_score(y_test, y_prob_original)
auc_log = roc_auc_score(y_test, y_prob_log)

# Plot ROC curves
plt.figure(figsize=(8, 6))
plt.plot(fpr_original, tpr_original, label=f'Original Data - AUC: {auc_original:
 ↪.2f}')
plt.plot(fpr_log, tpr_log, label=f'Log-Transformed Data - AUC: {auc_log:.2f}')
plt.plot([0, 1], [0, 1], linestyle='--', color='black', label='Random Guessing')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend()
plt.grid()
plt.show()
```


Receiver Operating Characteristic (ROC) Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Original Data - AUC: 0.96
Log-Transformed Data - AUC: 0.95
Random Guessing

[ ]: