Assignment 3

Name: Sourodeep Datta

Roll Number: 21CS10064

Importing the required libraries

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
import pandas as pd
```

Part A

Q1

Setting k = 4 and calculating corresponding probabilities

In [ ]:
```
k = 4

probabilities = [1/(2**(k - 1))]

for i in range(2, k+1):
    probabilities.append(1/(2**(i - 1)))

print(probabilities)
```

```
[0.125, 0.5, 0.25, 0.125]
```

Taking 1000 samples from the distribution

In [ ]:
```
np.random.seed(217)
samples = []

for i in range(1, 1001):
    sub_samples = np.random.choice(np.arange(1, k+1), size = 4, p=probabi
    samples.append(sub_samples.sum())

samples = np.array(samples)

freq = {}

for i in samples:
    if i in freq:
        freq[i] += 1
    else:
        freq[i] = 1
```

Defining necessary functions

In [ ]:
```
def print_five_number_summary(sample):
    minval = sample.min()
    lower_quartile = np.percentile(sample, 25)
    median = np.percentile(sample, 50)
    upper_quartile = np.percentile(sample, 75)
    maxval = sample.max()
    print("The five number summary is: ")
    print("Min: ", minval)
    print("Lower quartile: ", lower_quartile)
    print("Median: ", median)
    print("Upper quartile: ", upper_quartile)
    print("Max: ", maxval)

def plot_histogram(freq):
    plt.bar(list(freq.keys()), freq.values(), color='b')
    plt.xticks(list(freq.keys()))
    plt.title("Frequency Distribution of Random Die Rolls")
    plt.xlabel("Die Face")
    plt.ylabel("Frequency")
    plt.show()
```

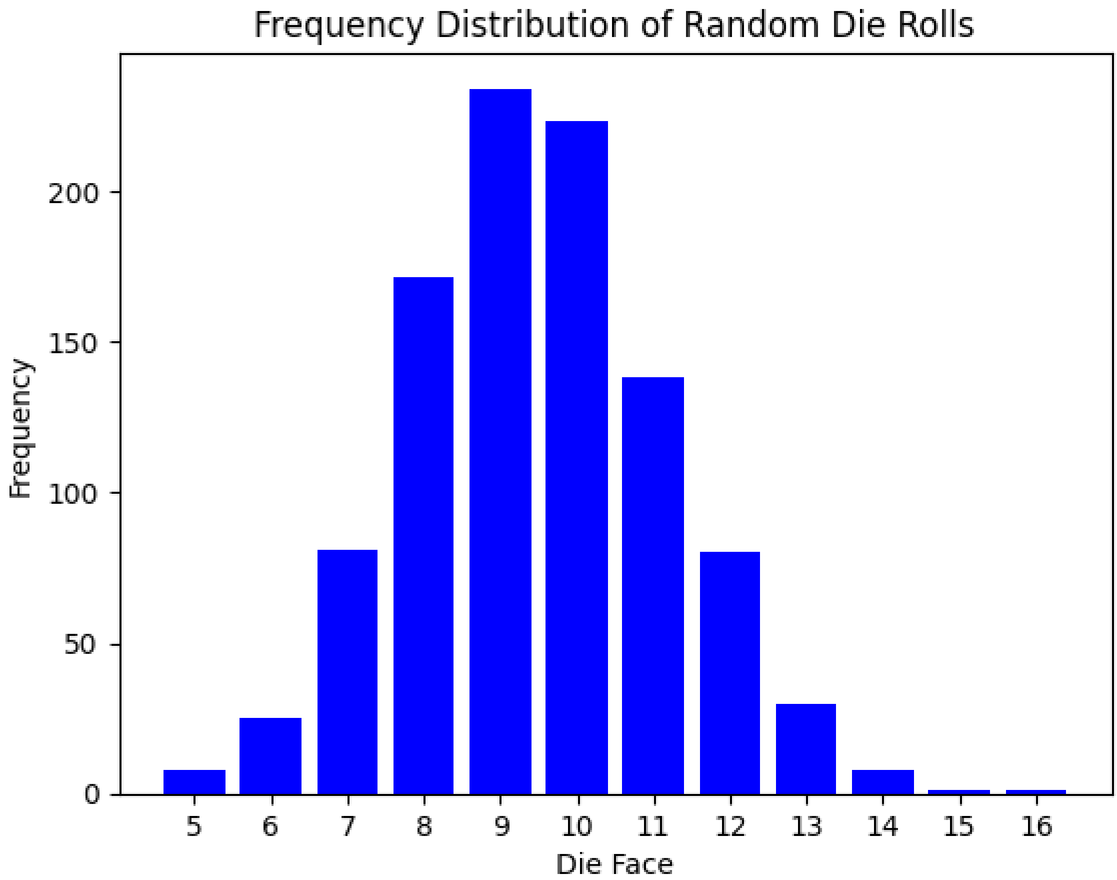
5 number summary of the samples

In [ ]:
```
print_five_number_summary(samples)
```

```
The five number summary is:
Min:  5
Lower quartile:  8.0
Median:  9.0
Upper quartile:  11.0
Max:  16
```

Histogram of the samples

In [ ]:
```
plot_histogram(freq)
```



In [ ]:
```
print("The estimated sample sum is", samples.sum())
```

```
The estimated sample sum is 9472
```

Let X_1, X_2, X_3 and X_4 be *iid* random variables having the probability distribution of the face value of the given biased 4-faced die. Let $Y = \sum_{i=1}^{4} X_i$ be a random variable. Then the expected value of Y is given by:

$$\mathbf{E}\left[Y\right] = \mathbf{E}\left[\sum_{i=1}^{4} X_i\right] = \sum_{i=1}^{4} \mathbf{E}\left[X_i\right]$$

As X_i are *iid* random variables, $\mathbf{E}\left[X_i\right] = \mathbf{E}\left[X_j\right] \ \forall \ i, j \in \{1, 2, 3, 4\}$. Let $\mathbf{E}\left[X_i\right] = \mathbf{E}\left[X\right] \ \forall \ i \in \{1, 2, 3, 4\}$. Then,

$$\mathbf{E}\left[Y\right] = \sum_{i=1}^{4} \mathbf{E}\left[X\right] = 4 \cdot \mathbf{E}\left[X\right]$$

Now, $\mathbf{E}\left[X\right]$ is the expected value of the face value of the given biased 4-faced die. Let p_i be the probability of getting the face value i for $i \in \{1, 2, 3, 4\}$. Then,

$$\mathbf{E}\left[X\right] = \sum_{i=1}^{4} i \cdot p_i$$

We know that $p_i = \dfrac{1}{2^{i-1}} \ \forall \ i \in \left[2, 4\right]$ and $p_1 = \dfrac{1}{2^3}$. Therefore,

$$\mathbf{E}[X] = \sum_{i=1}^{4} i \cdot p_i = \frac{1}{2^3} + \frac{2}{2^1} + \frac{3}{2^2} + \frac{4}{2^3} = \frac{1}{8} + \frac{2}{2} + \frac{3}{4} + \frac{4}{8} = \frac{19}{8} = 2.375$$

Thus, the expected value of Y is given by:

$$\mathbf{E}[Y] = 4 \cdot \mathbf{E}[X] = 4 \cdot 2.375 = 9.5$$

Now, let $Z = \sum_{i=1}^{n} Y_i$, where Y_i are *iid* random variables having the same probability distribution as Y. Then the expected value of Z is given by:

$$\mathbf{E}[Z] = \mathbf{E}[\sum_{i=1}^{n} Y_i] = \sum_{i=1}^{n} \mathbf{E}[Y_i]$$

As Y_i are *iid* random variables, $\mathbf{E}[Y_i] = \mathbf{E}[Y_j]\ \forall\, i, j \in \{1, 2, \ldots, n\}$. Let $\mathbf{E}[Y_i] = \mathbf{E}[Y]\ \forall\, i \in \{1, 2, \ldots, n\}$. Then,

$$\mathbf{E}[Z] = \sum_{i=1}^{n} \mathbf{E}[Y] = n \cdot \mathbf{E}[Y] = 9.5n$$

For n = 1000, we have:

$$\mathbf{E}[Z] = 9.5 \cdot 1000 = 9500$$

Thus, the expected value of Z is 9500, which is close to the experimentally calculated value of 9472 that we got.

Q2

Taking 1000 samples from the distribution

In [ ]:

```
np.random.seed(217)
samples = []

for i in range(1, 1001):
    sub_samples = np.random.choice(np.arange(1, k+1), size = 8, p=probabi
    samples.append(sub_samples.sum())

samples = np.array(samples)

freq = {}

for i in samples:
    if i in freq:
        freq[i] += 1
    else:
        freq[i] = 1
```

5 number summary of the samples

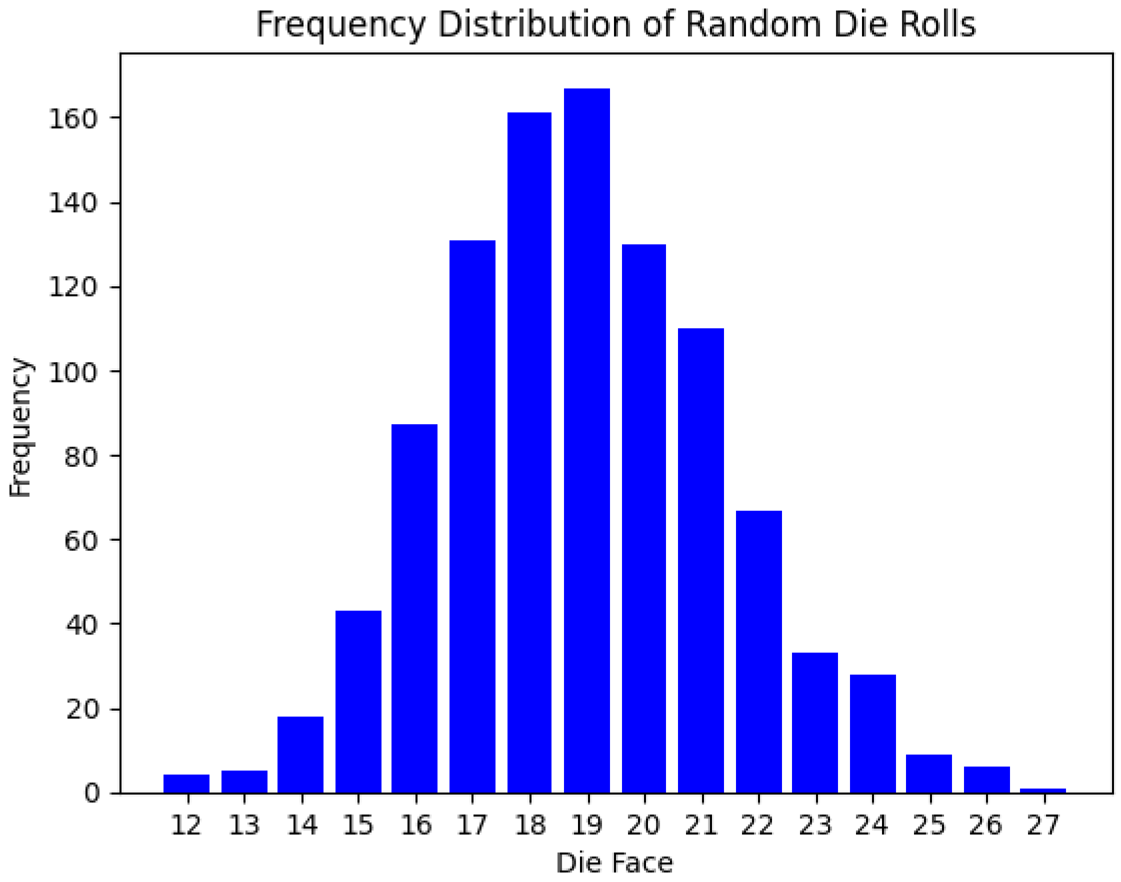
In [ ]:

```
print_five_number_summary(samples)
```

```
The five number summary is:
Min:  12
Lower quartile:  17.0
Median:  19.0
Upper quartile:  21.0
Max:  27
```

Histogram of the samples

In [ ]:
```
plot_histogram(freq)
```



In [ ]:
```
print("The estimated sample sum is", samples.sum())
```

```
The estimated sample sum is 18923
```

Let $X_1, X_2, X_3, \ldots X_8$ be *iid* random variables having the probability distribution of the face value of the given biased 4-faced die. Let $Y = \sum_{i=1}^{8} X_i$ be a random variable. Then the expected value of Y is given by:

$$\mathbf{E}[Y] = \mathbf{E}[\sum_{i=1}^{8} X_i] = \sum_{i=1}^{8} \mathbf{E}[X_i]$$

As X_i are *iid* random variables, $\mathbf{E}[X_i] = \mathbf{E}[X_j] \ \forall \ i, j \in [1, 8]$. Let $\mathbf{E}[X_i] = \mathbf{E}[X] \ \forall \ i \in [1, 8]$. Then,

$$\mathbf{E}[Y] = \sum_{i=1}^{8} \mathbf{E}[X] = 8 \cdot \mathbf{E}[X]$$

Now, $\mathbf{E}[X]$ is the expected value of the face value of the given biased 4-faced die. Let p_i be the probability of getting the face value i for $i \in \{1, 2, 3, 4\}$. Then,

$$\mathbf{E}[X] = \sum_{i=1}^{4} i \cdot p_i$$

We know that $p_i = \frac{1}{2^{i-1}}\ \forall\ i \in [2, 4]$ and $p_1 = \frac{1}{2^3}$. Therefore,

$$\mathbf{E}[X] = \sum_{i=1}^{4} i \cdot p_i = \frac{1}{2^3} + \frac{2}{2^1} + \frac{3}{2^2} + \frac{4}{2^3} = \frac{1}{8} + \frac{2}{2} + \frac{3}{4} + \frac{4}{8} = \frac{19}{8} = 2.375$$

Thus, the expected value of Y is given by:

$$\mathbf{E}[Y] = 8 \cdot \mathbf{E}[X] = 8 \cdot 2.375 = 19$$

Now, let $Z = \sum_{i=1}^{n} Y_i$, where Y_i are *iid* random variables having the same probability distribution as Y. Then the expected value of Z is given by:

$$\mathbf{E}[Z] = \mathbf{E}[\sum_{i=1}^{n} Y_i] = \sum_{i=1}^{n} \mathbf{E}[Y_i]$$

As Y_i are *iid* random variables, $\mathbf{E}[Y_i] = \mathbf{E}[Y_j]\ \forall\ i, j \in \{1, 2, \ldots, n\}$. Let $\mathbf{E}[Y_i] = \mathbf{E}[Y]\ \forall\ i \in \{1, 2, \ldots, n\}$. Then,

$$\mathbf{E}[Z] = \sum_{i=1}^{n} \mathbf{E}[Y] = n \cdot \mathbf{E}[Y] = 19n$$

For n = 1000, we have:

$$\mathbf{E}[Z] = 19 \cdot 1000 = 19000$$

Thus, the expected value of Z is 19000, which is close to the experimentally calculated value of 18923 that we got.

Q3

Setting k = 16 and calculating corresponding probabilities

In [ ]:
```
k = 16

probabilities = [1/(2**(k - 1))]

for i in range(2, k+1):
    probabilities.append(1/(2**(i - 1)))

print(probabilities)
```

```
[3.0517578125e-05, 0.5, 0.25, 0.125, 0.0625, 0.03125, 0.015625, 0.0078125,
0.00390625, 0.001953125, 0.0009765625, 0.00048828125, 0.000244140625, 0.00
01220703125, 6.103515625e-05, 3.0517578125e-05]
```

Taking 1000 samples from the distribution (sum of 4 rolls per sample)

In [ ]:
```
np.random.seed(217)
samples = []

for i in range(1, 1001):
    sub_samples = np.random.choice(np.arange(1, k+1), size = 4, p=probabi
    samples.append(sub_samples.sum())

samples = np.array(samples)

freq = {}

for i in samples:
    if i in freq:
        freq[i] += 1
    else:
        freq[i] = 1
```

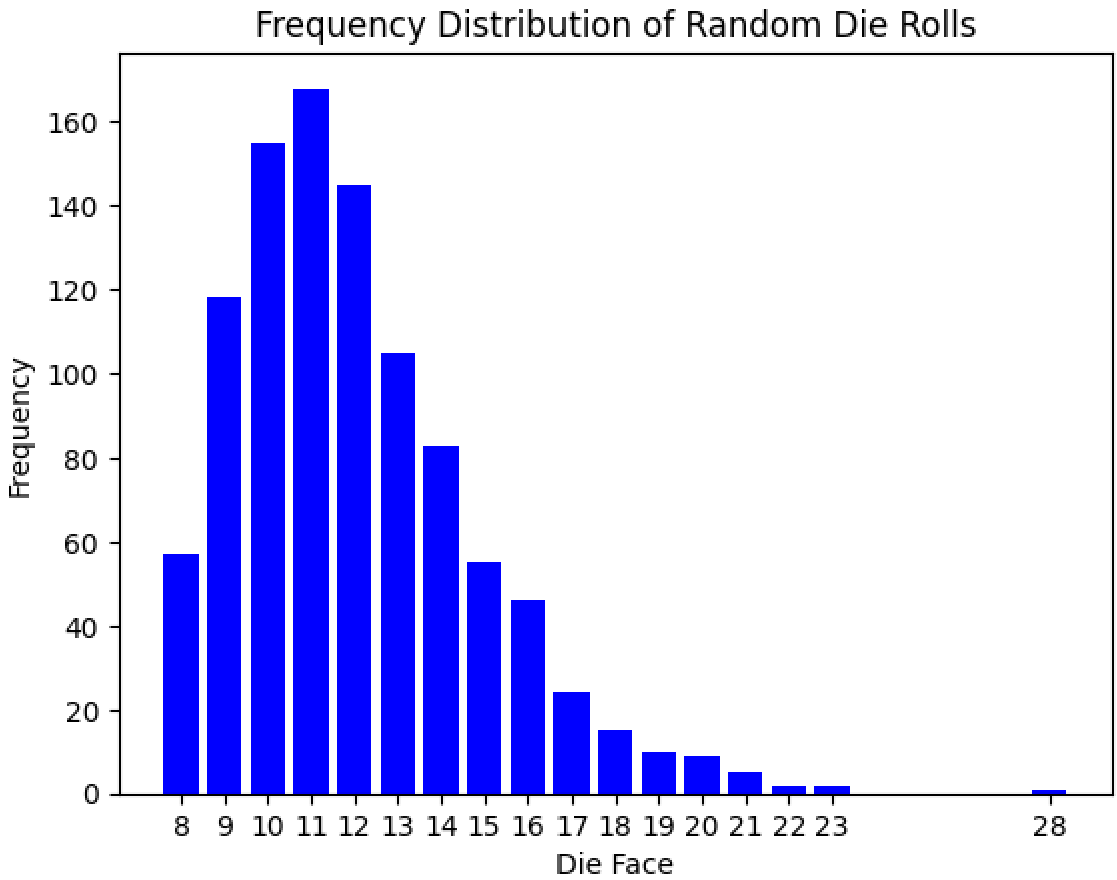
5 number summary of the samples

In [ ]:
```
print_five_number_summary(samples)
```

```
The five number summary is:
Min:  8
Lower quartile:  10.0
Median:  12.0
Upper quartile:  14.0
Max:  28
```

Histogram of the samples

In [ ]:
```
plot_histogram(freq)
```



In [ ]:
```
print("The estimated sample sum is", samples.sum())
```

```
The estimated sample sum is 12015
```

Taking 1000 samples from the distribution (sum of 8 rolls per sample)

In [ ]:
```
np.random.seed(217)
samples = []

for i in range(1, 1001):
    sub_samples = np.random.choice(np.arange(1, k+1), size = 8, p=probabi
    samples.append(sub_samples.sum())

samples = np.array(samples)

freq = {}

for i in samples:
    if i in freq:
        freq[i] += 1
    else:
        freq[i] = 1
```

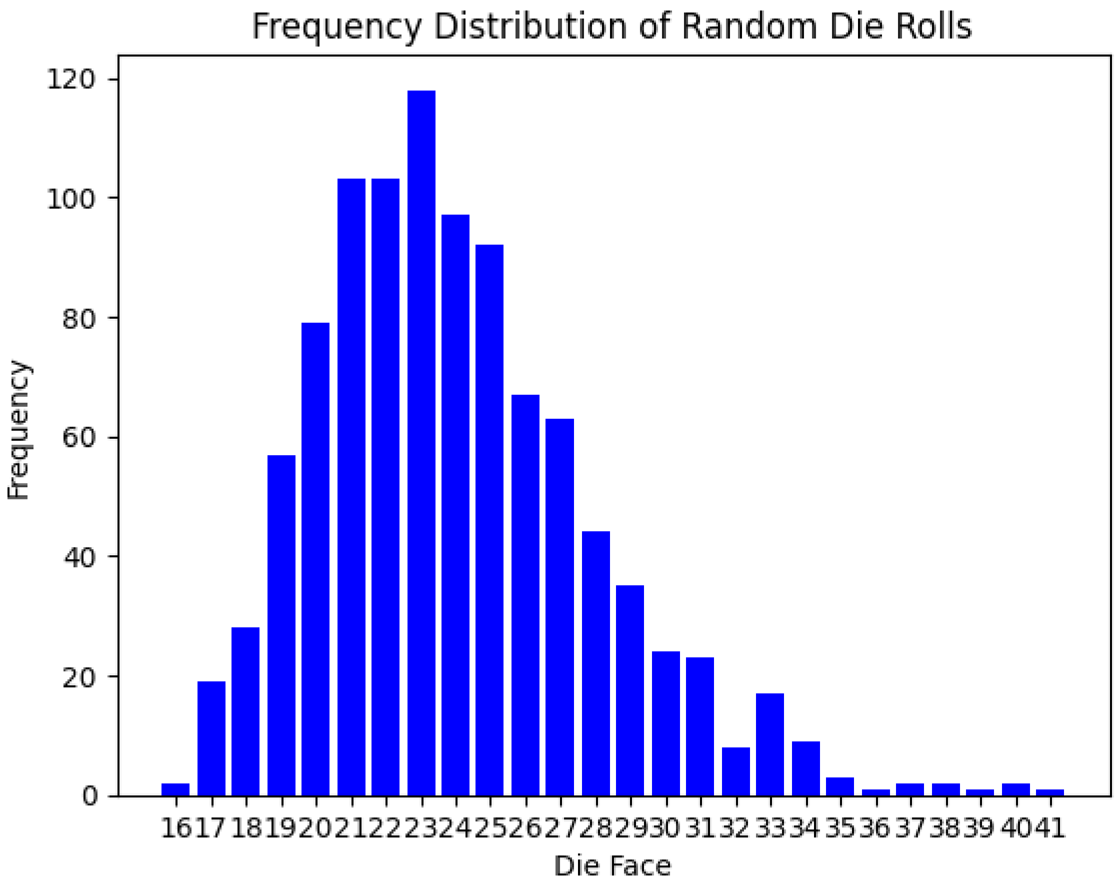
5 number summary of the samples

In [ ]:
```
print_five_number_summary(samples)
```

```
The five number summary is:
Min:  16
Lower quartile:  21.0
Median:  23.0
Upper quartile:  26.0
Max:  41
```

Histogram of the samples

In [ ]:
```
plot_histogram(freq)
```



In [ ]:
```
print("The estimated sample sum is", samples.sum())
```

```
The estimated sample sum is 23990
```

Part B

Loading the Dataset

In [ ]:
```
from ucimlrepo import fetch_ucirepo

spambase = fetch_ucirepo(id=94)

X = spambase.data.features
y = spambase.data.targets

X = pd.DataFrame(X)
y = pd.DataFrame(y)
```

In [ ]:
```
X.head()
```

Out[ ]:

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our
0	0.00	0.64	0.64	0.0	0.32
1	0.21	0.28	0.50	0.0	0.14
2	0.06	0.00	0.71	0.0	1.23
3	0.00	0.00	0.00	0.0	0.63
4	0.00	0.00	0.00	0.0	0.63

5 rows × 57 columns

In [ ]:
```
y.head()
```

Out[ ]:

	Class
0	1
1	1
2	1
3	1
4	1

Getting train-validation-test split

In [ ]:
```
X = np.array(X)
y = np.array(y)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
X_val, X_test, y_val, y_test = train_test_split(X_test, y_test, test_size
```

In [ ]:
```
print(X_train.shape)
print(X_val.shape)
print(X_test.shape)
```

```
(3220, 57)
(690, 57)
(691, 57)
```

Sampling 5 random columns

In [ ]:
```
column_idx = np.random.choice(np.arange(0, X_train.shape[1]), size = 5, r
columns = X_train[:, column_idx]
print(columns.shape)
```

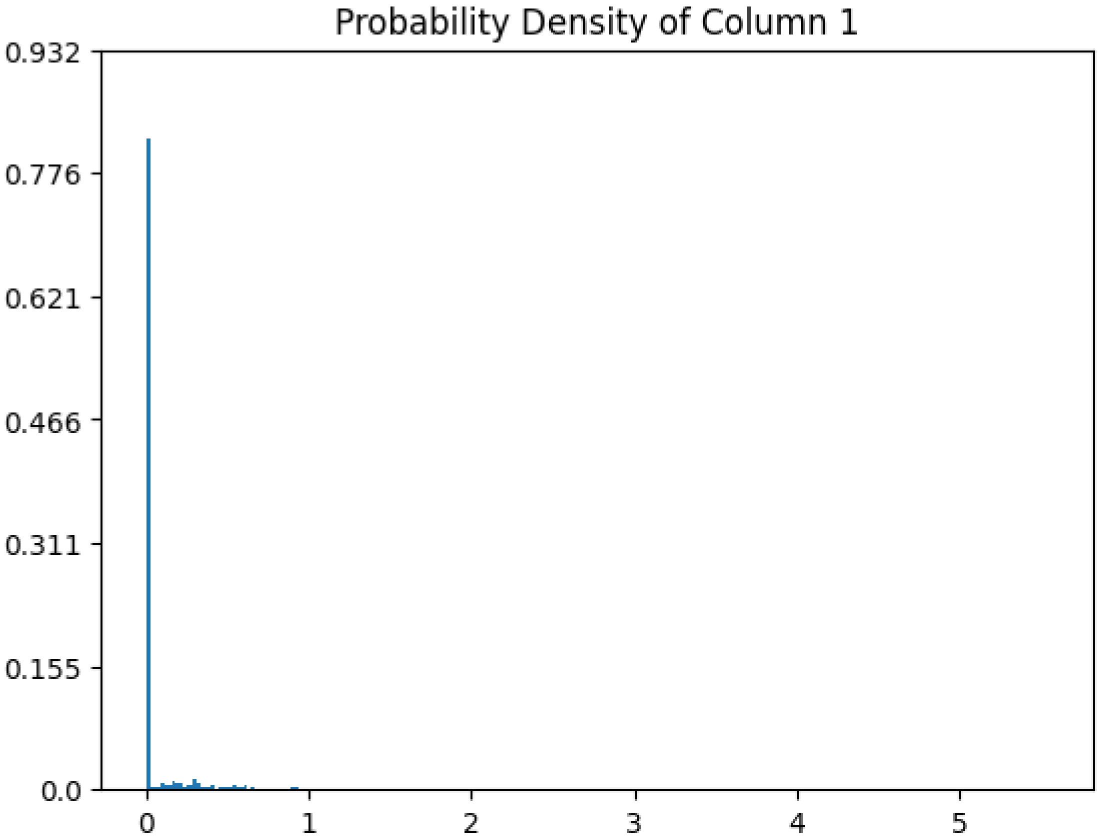
```
(3220, 5)
```

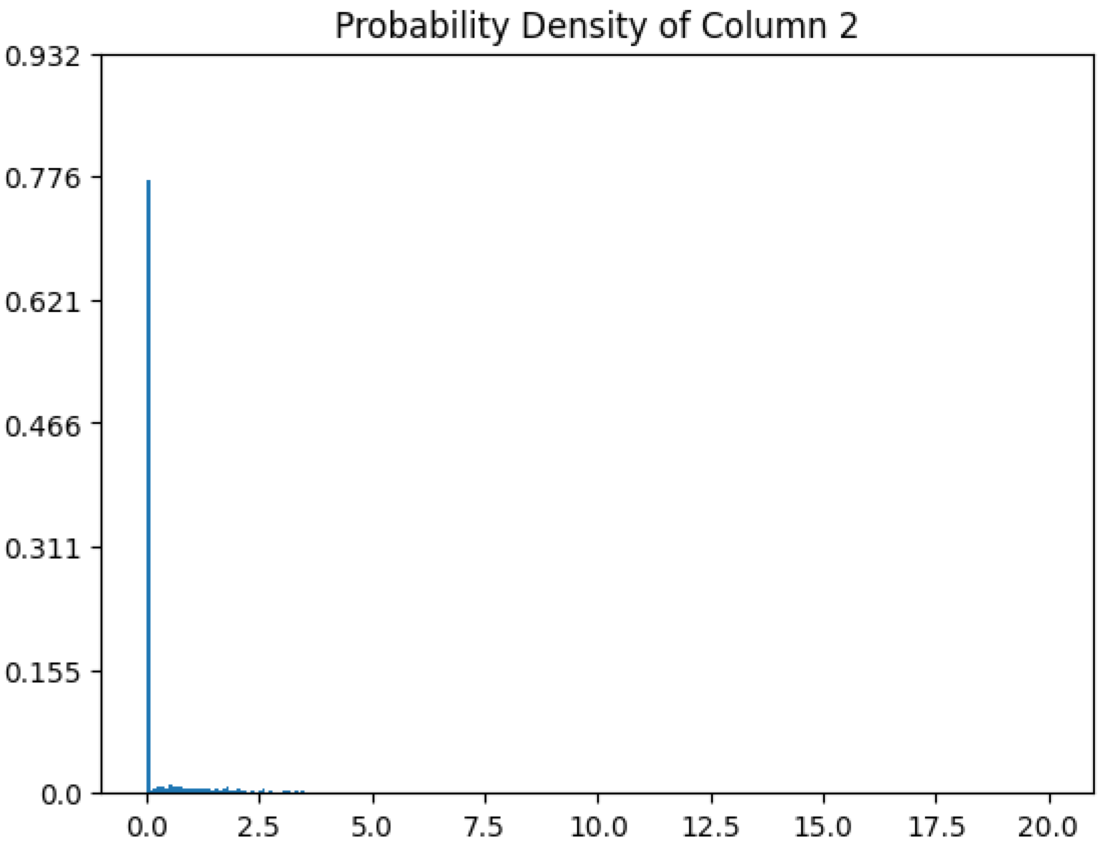
In [ ]:
```
def plot_probability_density_columns(columns):
    for i in range(columns.shape[1]):
        plt.hist(columns[:, i], bins=250, label="Column " + str(i), densi
        plt.title("Probability Density of Column " + str(i + 1))
        locs, _ = plt.yticks()
```

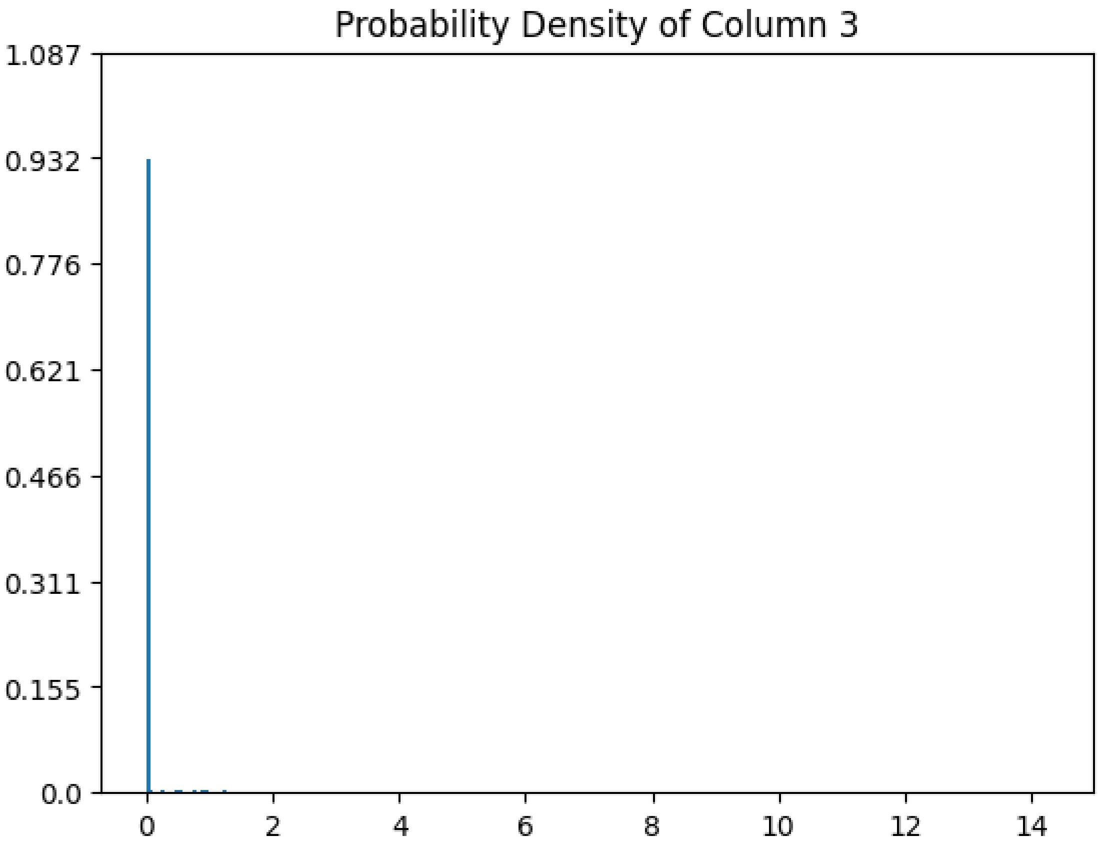
```
        plt.yticks(locs,np.round(locs/len(columns[:, i]),3))
        plt.show()
```

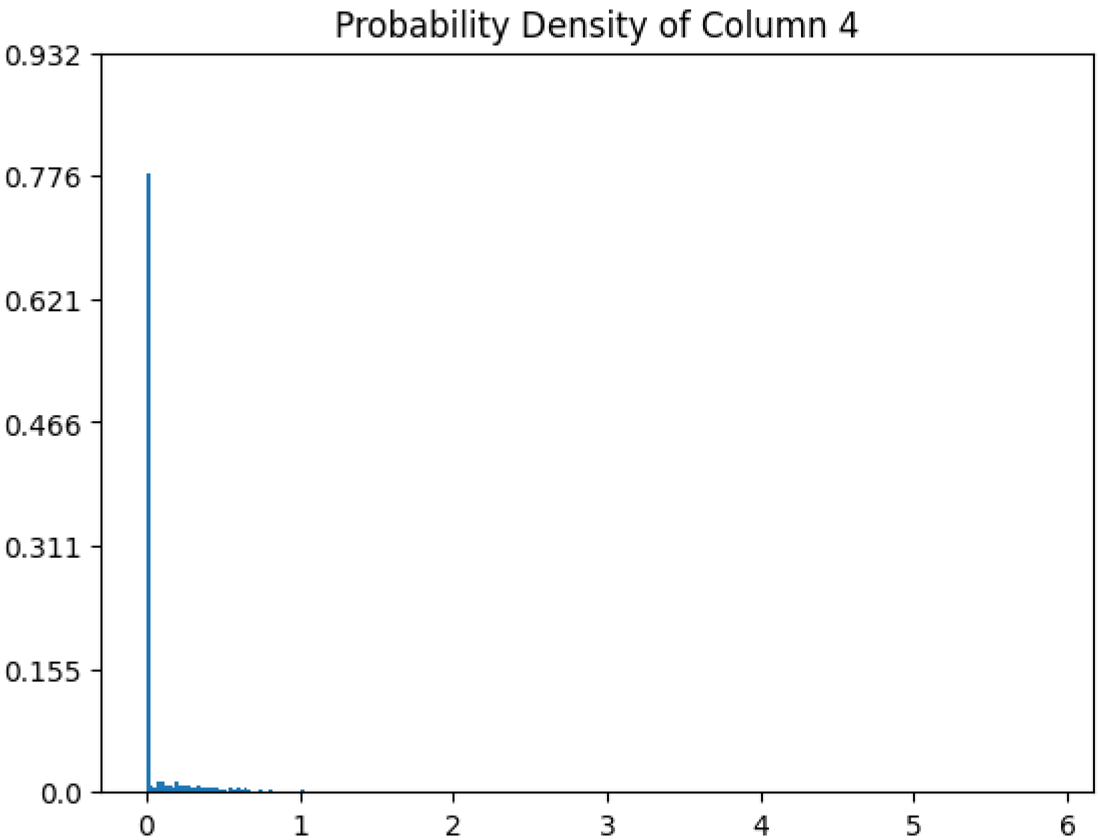
Plotting histograms for 5 sampled columns

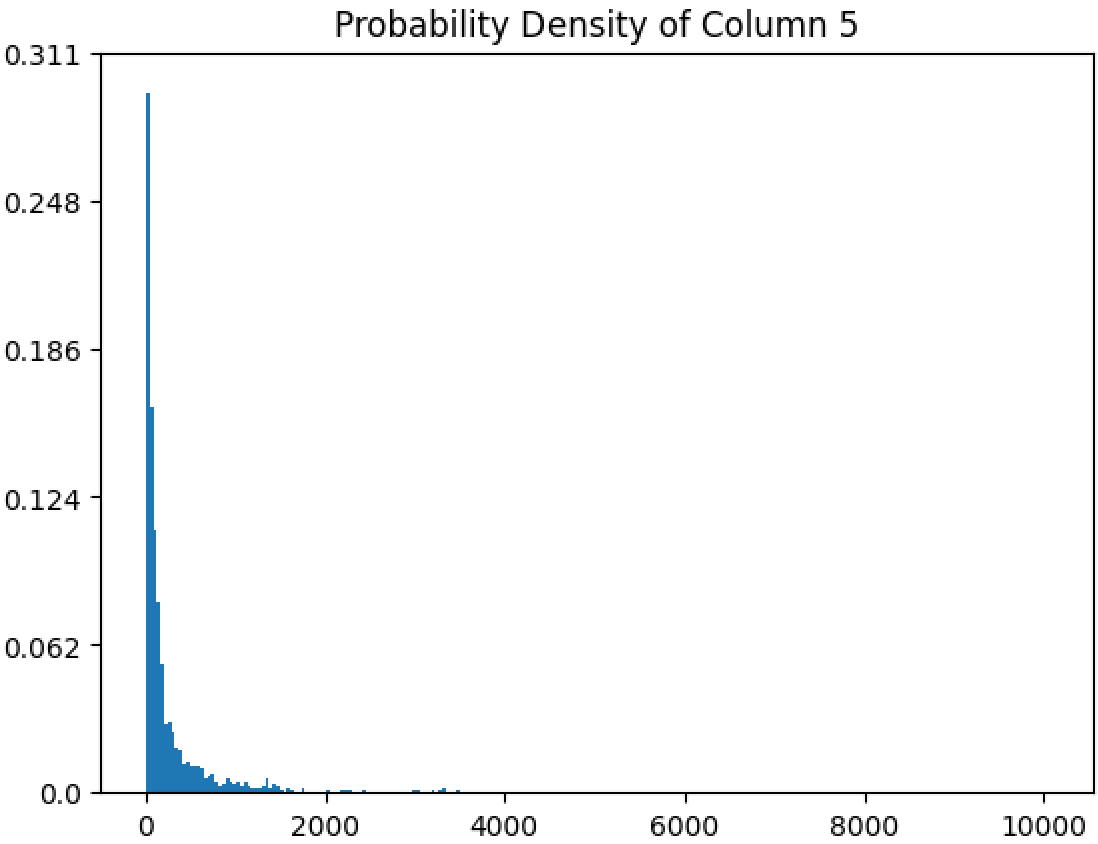
In [ ]:
```
plot_probability_density_columns(columns)
```






Probability Density of Column 3
1.087
0.932
0.776
0.621
0.466
0.311
0.155
0.0
0
2
4
6
8
10
12
14


Probability Density of Column 4
0.932
0.776
0.621
0.466
0.311
0.155
0.0
0
1
2
3
4
5
6



Creating the Model

In [ ]:
```
class NaiveBayes:
    def __init__(self, bins, classes):
        self.parameters = {}
        self.bin_edges = {}
        self.priors = {}
        self.bins = bins
        self.classes = np.arange(0, classes)

    def fit(self, X, y):
        for i, c in enumerate(self.classes):
            X_c = X[y[:, 0] == c, :]

            self.priors[c] = np.log(X_c.shape[0] / X.shape[0])   #Calcula
            self.parameters["ParameterProbs" + str(c)] = np.zeros((X.shap
            self.bin_edges["Bin Edges" + str(c)] = np.zeros((X.shape[1],

            for j in range(X.shape[1]):
                self.bin_edges["Bin Edges" + str(c)][j, :] = np.histogram
                self.parameters["ParameterProbs" + str(c)][j, 1:-1] = np.
                self.parameters["ParameterProbs" + str(c)][j, :] += 1
                #Storing Probabilites in Log Scale to avoid numerical und
                self.parameters["ParameterProbs" + str(c)][j, :] = np.log
                self.parameters["ParameterProbs" + str(c)][j, :] -= np.lo

    def predict(self, X):
        posteriors = []
        for x in X:
            posterior = []
            for c in self.classes:
                likelihood = 0
                for j in range(X.shape[1]):
```

```
                    bin_idx = np.digitize(x[j], self.bin_edges["Bin Edges
                    likelihood += self.parameters["ParameterProbs" + str(
                posterior.append(self.priors[c] + likelihood)
            posteriors.append(self.classes[np.argmax(posterior)])
        return posteriors
```

Training the Model

In [ ]:
```
model = NaiveBayes(150, 2)
model.fit(X_train, y_train)
```

Printing the Model Priors

In [ ]:
```
print("Model Priors are: ", {i : np.exp(model.priors[i]) for i in model.p
```

```
Model Priors are:  {0: 0.6006211180124224, 1: 0.39937888198757765}
```

Printing the total number of parameters

In [ ]:
```
print('The total number of parameters are: ', np.sum([model.parameters[i]
```

```
The total number of parameters are:  17330
```

In [ ]:
```
def accuracy(y_true, y_pred):
    return np.sum(y_true == y_pred) / len(y_true)

def precision(y_true, y_pred):
    tp = np.sum((y_true == 1) & (y_pred == 1))
    fp = np.sum((y_true == 0) & (y_pred == 1))
    return tp / (tp + fp)

def recall(y_true, y_pred):
    tp = np.sum((y_true == 1) & (y_pred == 1))
    fn = np.sum((y_true == 1) & (y_pred == 0))
    return tp / (tp + fn)

def f1_score(y_true, y_pred):
    prec = precision(y_true, y_pred)
    rec = recall(y_true, y_pred)
    return 2 * prec * rec / (prec + rec)
```

In [ ]:
```
predictions = np.array(model.predict(X_test))
```

Printing the necessary metrics

In [ ]:
```
print("Accuracy: ", accuracy(y_test[:, 0], predictions))
print("Precision: ", precision(y_test[:, 0], predictions))
print("Recall: ", recall(y_test[:, 0], predictions))
print("F1 Score: ", f1_score(y_test[:, 0], predictions))
```

```
Accuracy:  0.8610709117221418
Precision:  0.9453551912568307
Recall:  0.667953667953668
F1 Score:  0.7828054298642533
```

Taking the logarthims of the input data

In [ ]:
```
X_train_new = np.log(X_train + 0.0001)    #Adding 0.001 to avoid log(0)
X_val_new = np.log(X_val + 0.0001)
X_test_new = np.log(X_test + 0.0001)
```

Training model

In [ ]:
```
model_new = NaiveBayes(150, 2)
model_new.fit(X_train_new, y_train)
```

Printing the Model Priors

In [ ]:
```
print("Model Priors are: ", {i : np.exp(model_new.priors[i]) for i in mod
```

```
Model Priors are:  {0: 0.6006211180124224, 1: 0.39937888198757765}
```

In [ ]:
```
predictions = np.array(model_new.predict(X_test_new))
```

Printing the necessary metrics

In [ ]:
```
print("Accuracy: ", accuracy(y_test[:, 0], predictions))
print("Precision: ", precision(y_test[:, 0], predictions))
print("Recall: ", recall(y_test[:, 0], predictions))
print("F1 Score: ", f1_score(y_test[:, 0], predictions))
```

```
Accuracy:  0.9001447178002895
Precision:  0.8925619834710744
Recall:  0.833976833976834
F1 Score:  0.8622754491017964
```

The accuracy when taking the logarithms of the columns is more than the accuracy when not taking the logarithms of the columns. This is because the orginial data in the features is heavily clustered around 0. Taking the logarithms of the columns helps in spreading out the data among different bins, thus making it easier for the model to learn the features, improving the accuracy.

Part C

In [ ]:
```
model = GaussianNB()
model.fit(X_train, y_train[:, 0])

model_log = GaussianNB()
model_log.fit(X_train_new, y_train[:, 0])
```

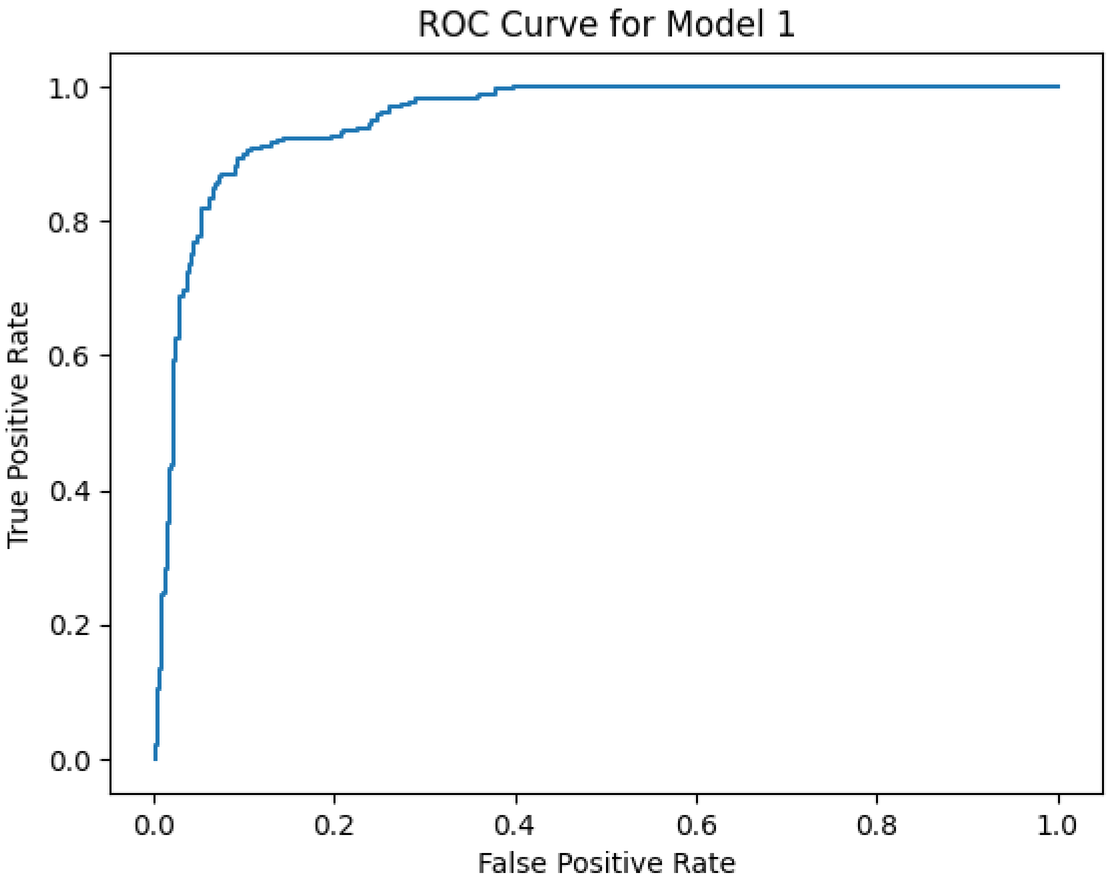
Out[ ]:
```
▾ GaussianNB
GaussianNB()
```

In [ ]:
```
def plot_ROC(fpr, tpr, label):
    plt.plot(fpr, tpr)
    plt.title("ROC Curve for " + str(label))
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.show()

def calculate_roc(y_true, y_pred):
```

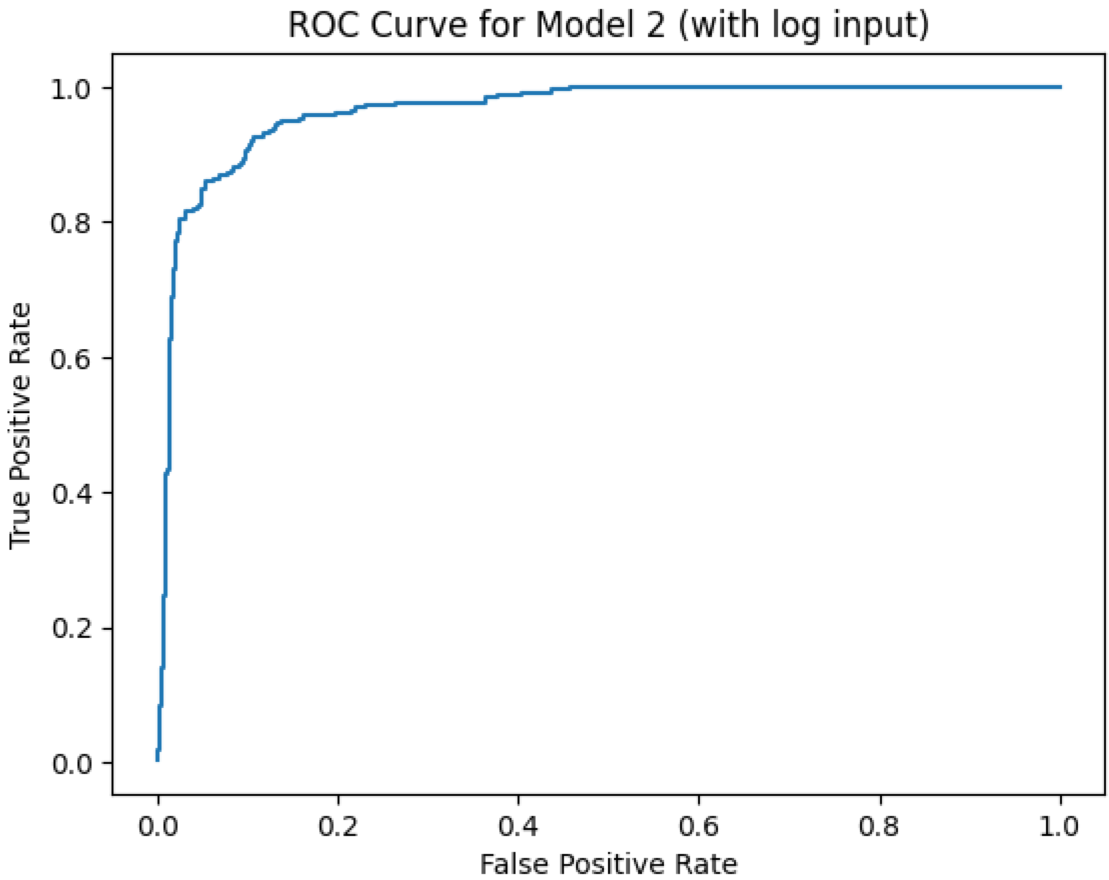
```
    data = list(zip(y_true, y_pred))

    data.sort(key=lambda x: x[1], reverse=True)

    num_positive = y_true.count(1)
    num_negative = y_true.count(0)

    tpr = []
    fpr = []
    true_positives = 0
    false_positives = 0

    for label, score in data:
        if label == 1:
            true_positives += 1
        else:
            false_positives += 1

        tpr.append(true_positives / num_positive)
        fpr.append(false_positives / num_negative)

    return fpr, tpr
```

In [ ]:
```
predictions = model.predict_proba(X_test)[:, 1]
predictions_log = model_log.predict_proba(X_test_new)[:, 1]
```

In [ ]:
```
fpr, tpr = calculate_roc(list(y_test[:, 0]), list(predictions))
plot_ROC(fpr, tpr, "Model 1")
```



In [ ]:
```
fpr, tpr = calculate_roc(list(y_test[:, 0]), list(predictions_log))
plot_ROC(fpr, tpr, "Model 2 (with log input)")
```



The second model is better than the first model. This is because the second model has an ROC curve that is closer to the top left corner of the graph, which is the ideal position for the ROC curve to be in, as it implies that the model will have a higher true positive rate for the same false positive rate compared to the second model.

Naive Bayes vs SVM

SVM:
It can be seen that the accuracy when using an SVM is higher than the accuracy when using a Naive Bayes classifier, when using specific kernels and values of C, with the best seen accuracy being for $C = 10$ using an rbf kernel, having an accuracy of 0.94.

Naive Bayes:
In general, it can be seen that Naive Bayes tends to perform worse than SVM according to accuracy. However, compared to the SVM, it requires much less time to train, and only has one hyperparameter, which is the number of bins, thus making it easier to optimize the hyperparameter. The best accuracy achieved was 0.90 using 150 bins.