Question 2: Naive Bayes

Importing necessary libraries

In [1]:
```
import numpy as np
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
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import random
import matplotlib.pyplot as plt
import numpy as np
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import roc_curve, auc, accuracy_score
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
from sklearn.naive_bayes import GaussianNB
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
```

Removing warnings

In [2]:
```
import warnings

# To ignore all warnings:
warnings.filterwarnings("ignore")

# To ignore a specific type of warning (e.g., DeprecationWarning):
warnings.filterwarnings("ignore", category=DeprecationWarning)
```

Part A: Probability:

In [3]:
```
def Experiment(k1, num_rolls1, num_trials1):
    # Set the number of trials and the number of dice rolls per trial
    k = k1
    num_trials = num_trials1
    num_rolls = num_rolls1

    results = []
    weight=[]
    weight.append(1 / (2 ** (k - 1)))
    for i in range(2, k + 1):
        weight.append(1 / (2 ** (i - 1)))

    print("Probabilities: ", weight)
    random.seed(50)
    for _ in range(num_trials):
        trial_sum = sum(random.choices(range(1, k + 1), k=num_rolls, weights=weight))
        results.append(trial_sum)

    # Plot a frequency distribution histogram
    plt.hist(results, bins=range(num_rolls, k * num_rolls + 2), align='left', rwidth=0.8)
    plt.title('Frequency Distribution (k={}, Rolls={})'.format(k, num_rolls))
    plt.xlabel('Sum of Upward Face Values')
    plt.ylabel('Frequency')
    plt.show()

    # Calculate and print the five-number summary
    min_val = np.min(results)
    q1 = np.percentile(results, 25)
    median = np.percentile(results, 50)
    q3 = np.percentile(results, 75)
    max_val = np.max(results)

    print("\nFive-number  summary  of  the  distribution.")
    print("Min: %d" % min_val)
    print("Q1: %d" % q1)
    print("Median: %d" % median)
    print("Q3: %d" % q3)
    print("Max: %d" % max_val)

    # Theoretical Expected Value
    expected_value = 0
```

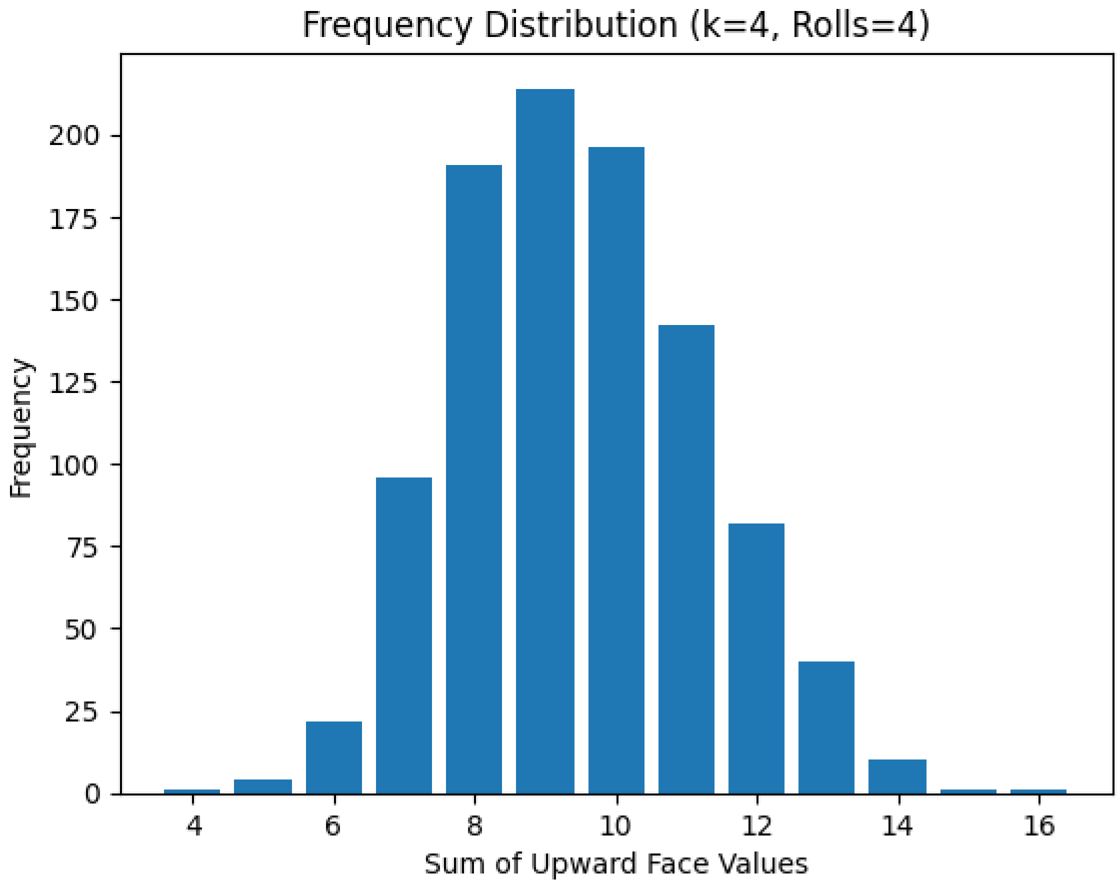
```python
    for i in range(1, k+1):
        #print(i, weight[i-1])
        expected_value+=i*weight[i-1]

    print("\nSimulated Expected Value: %.2f" % (np.mean(results)))
    print("Theoretical Expected Value: %.2f" % (expected_value*num_rolls))
```

1. Consider **k = 4** and randomly roll the die **4 times**

1. calculate the sum of the upward facevalue.
2. Repeat this **task 1000 times**
3. **plot a frequency distribution histogram**.
4. Print the **five-number summary** of the distribution.
5. Showing that the **theoretical Expected sum** of the event is close to the **actual sum** we got in the Python program simulation.

In [4]:
```python
Experiment(4, 4, 1000)
```

```
Probabilities:  [0.125, 0.5, 0.25, 0.125]
```



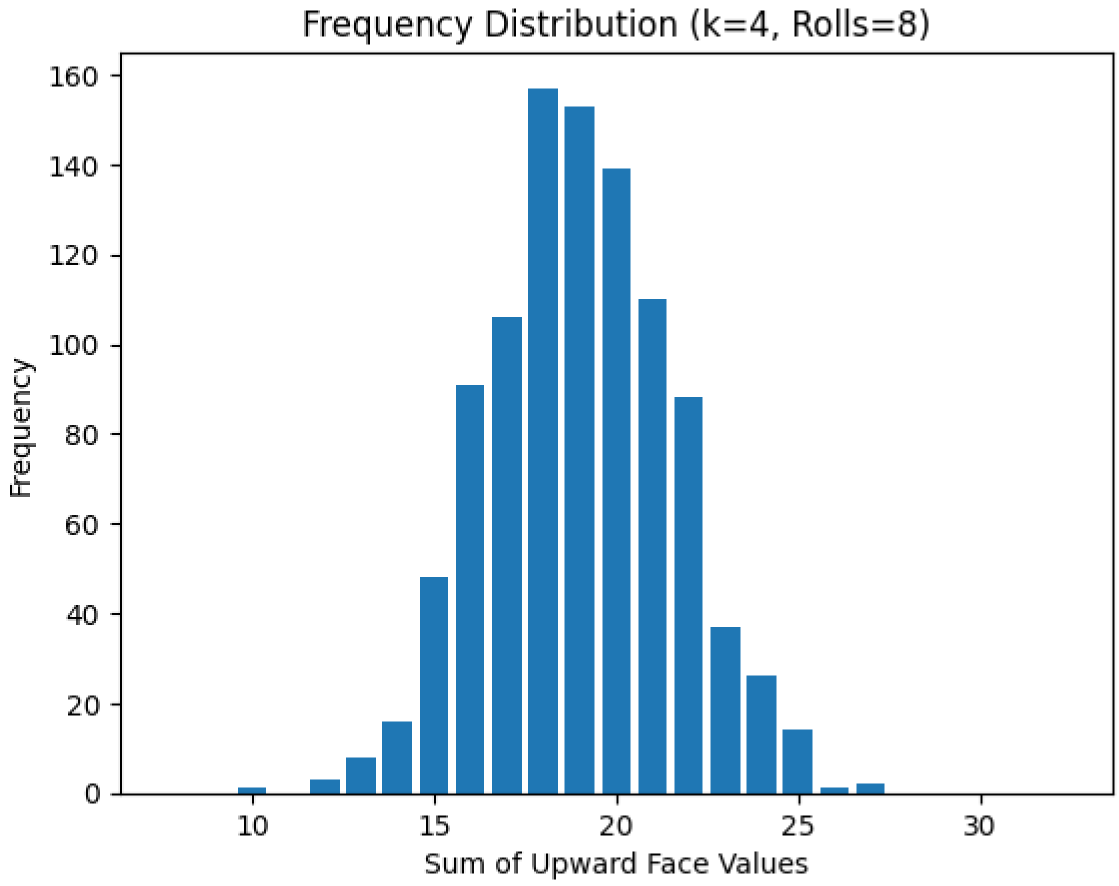

```
Five-number  summary  of  the  distribution.
Min: 4
Q1: 8
Median: 9
Q3: 11
Max: 16

Simulated Expected Value: 9.48
Theoretical Expected Value: 9.50
```

2. Consider **k = 4** and randomly roll the die **8 times**

1. calculate the sum of the upward facevalue.
2. Repeat this **task 1000 times**
3. **plot a frequency distribution histogram**.
4. Print the **five-number summary** of the distribution.
5. Showing that the **theoretical Expected sum** of the event is close to the **actual sum** we got in the Python program simulation.

In [5]:
```python
Experiment(4, 8, 1000)
```

```
Probabilities:  [0.125, 0.5, 0.25, 0.125]
```



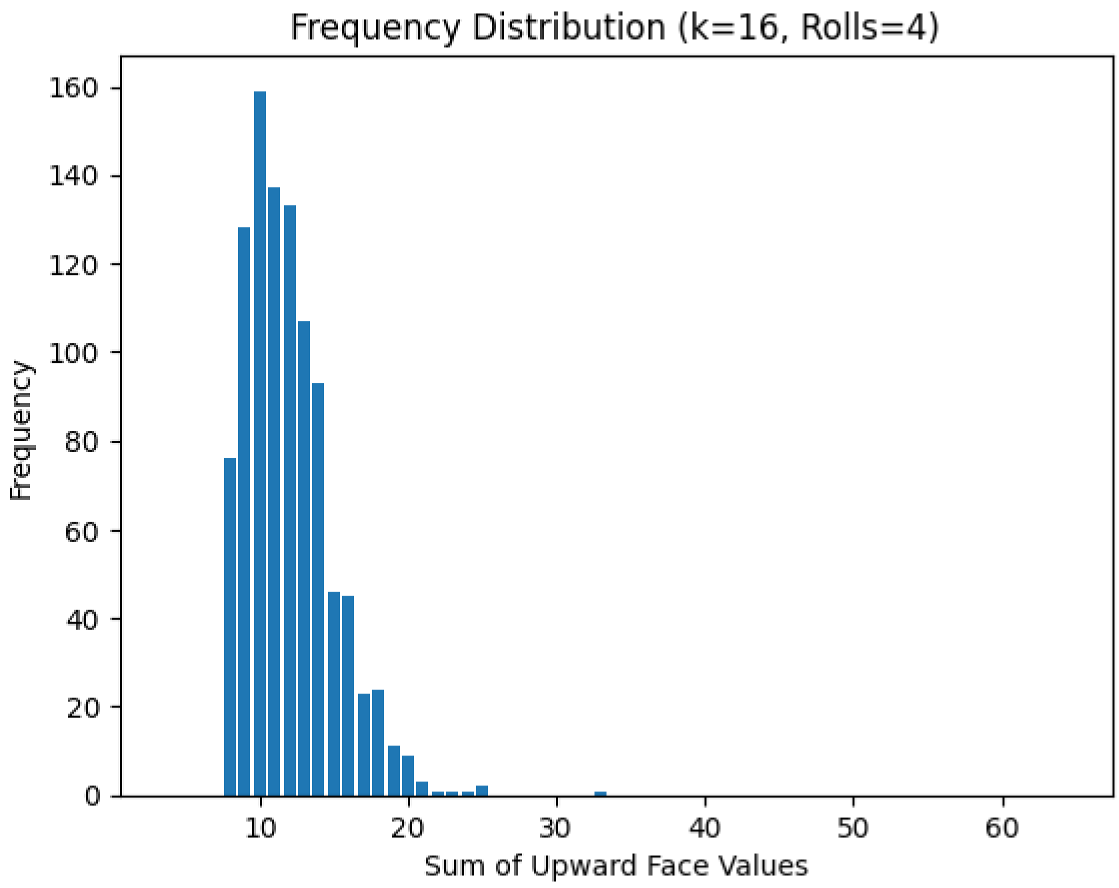

```
Five-number  summary  of  the  distribution.
Min: 10
Q1: 17
Median: 19
Q3: 21
Max: 27

Simulated Expected Value: 19.02
Theoretical Expected Value: 19.00
```

3. Consider **k = 16** and randomly roll the die **4 times**

1. calculate the sum of the upward facevalue.
2. Repeat this **task 1000 times**
3. **plot a frequency distribution histogram**.
4. Print the **five-number summary** of the distribution.
5. Showing that the **theoretical Expected sum** of the event is close to the **actual sum** we got in the Python program simulation.

In [6]:
```python
Experiment(16, 4, 1000)
```

```
Probabilities:  [3.0517578125e-05, 0.5, 0.25, 0.125, 0.0625, 0.03125, 0.015625, 0.0078125, 0.00390625, 0.001953125, 0.000976562
5, 0.00048828125, 0.000244140625, 0.0001220703125, 6.103515625e-05, 3.0517578125e-05]
```




```
Five-number  summary  of  the  distribution.
Min: 8
Q1: 10
Median: 11
Q3: 14
Max: 33

Simulated Expected Value: 11.98
Theoretical Expected Value: 12.00
```

Part B: Implementation of Naive Bayes (From Scratch)

1. Getting Dataset :

In [7]:
```python
from ucimlrepo import fetch_ucirepo

# fetch dataset
spambase = fetch_ucirepo(id=94)

# data (as pandas dataframes)
X = spambase.data.features
y = spambase.data.targets

# metadata
print(spambase.metadata)

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

{'uci_id': 94, 'name': 'Spambase', 'repository_url': 'https://archive.ics.uci.edu/dataset/94/spambase', 'data_url': 'https://arc
hive.ics.uci.edu/static/public/94/data.csv', 'abstract': 'Classifying Email as Spam or Non-Spam', 'area': 'Computer Science', 't
asks': ['Classification'], 'characteristics': ['Multivariate'], 'num_instances': 4601, 'num_features': 57, 'feature_types': ['In
teger', 'Real'], 'demographics': [], 'target_col': ['Class'], 'index_col': None, 'has_missing_values': 'no', 'missing_values_sym
bol': None, 'year_of_dataset_creation': 1999, 'last_updated': 'Mon Aug 28 2023', 'dataset_doi': '10.24432/C53G6X', 'creators':
['Mark Hopkins', 'Erik Reeber', 'George Forman', 'Jaap Suermondt'], 'intro_paper': None, 'additional_info': {'summary': 'The "sp
am" concept is diverse: advertisements for products/web sites, make money fast schemes, chain letters, pornography...\n\nThe cla
ssification task for this dataset is to determine whether a given email is spam or not.\n\t\nOur collection of spam e-mails came
from our postmaster and individuals who had filed spam. Our collection of non-spam e-mails came from filed work and personal e-
mails, and hence the word \'george\' and the area code \'650\' are indicators of non-spam. These are useful when constructing a
personalized spam filter. One would either have to blind such non-spam indicators or get a very wide collection of non-spam to
generate a general purpose spam filter.\n\nFor background on spam: Cranor, Lorrie F., LaMacchia, Brian A. Spam!, Communications
of the ACM, 41(8):74-83, 1998.\n\nTypical performance is around ~7% misclassification error. False positives (marking good mail
as spam) are very undesirable.If we insist on zero false positives in the training/testing set, 20-25% of the spam passed throug
h the filter. See also Hewlett-Packard Internal-only Technical Report. External version forthcoming. ', 'purpose': None, 'funded
_by': None, 'instances_represent': 'Emails', 'recommended_data_splits': None, 'sensitive_data': None, 'preprocessing_descriptio
n': None, 'variable_info': 'The last column of \'spambase.data\' denotes whether the e-mail was considered spam (1) or not (0),
i.e. unsolicited commercial e-mail. Most of the attributes indicate whether a particular word or character was frequently occur
ing in the e-mail. The run-length attributes (55-57) measure the length of sequences of consecutive capital letters. For the s
tatistical measures of each attribute, see the end of this file. Here are the definitions of the attributes:\r\n\r\n48 continuo
us real [0,100] attributes of type word_freq_WORD \r\n= percentage of words in the e-mail that match WORD, i.e. 100 * (number of
times the WORD appears in the e-mail) / total number of words in e-mail. A "word" in this case is any string of alphanumeric ch
aracters bounded by non-alphanumeric characters or end-of-string.\r\n\r\n6 continuous real [0,100] attributes of type char_freq_
CHAR] \r\n= percentage of characters in the e-mail that match CHAR, i.e. 100 * (number of CHAR occurences) / total characters in
e-mail\r\n\r\n1 continuous real [1,...] attribute of type capital_run_length_average \r\n= average length of uninterrupted seque
nces of capital letters\r\n\r\n1 continuous integer [1,...] attribute of type capital_run_length_longest \r\n= length of longest
uninterrupted sequence of capital letters\r\n\r\n1 continuous integer [1,...] attribute of type capital_run_length_total \r\n= s
um of length of uninterrupted sequences of capital letters \r\n= total number of capital letters in the e-mail\r\n\r\n1 nominal
{0,1} class attribute of type spam\r\n= denotes whether the e-mail was considered spam (1) or not (0), i.e. unsolicited commerci
al e-mail. \r\n', 'citation': None}}

	name	role	type	demographic	\
0	word_freq_make	Feature	Continuous	None	
1	word_freq_address	Feature	Continuous	None	
2	word_freq_all	Feature	Continuous	None	
3	word_freq_3d	Feature	Continuous	None	
4	word_freq_our	Feature	Continuous	None	
5	word_freq_over	Feature	Continuous	None	
6	word_freq_remove	Feature	Continuous	None	
7	word_freq_internet	Feature	Continuous	None	
8	word_freq_order	Feature	Continuous	None	
9	word_freq_mail	Feature	Continuous	None	
10	word_freq_receive	Feature	Continuous	None	
11	word_freq_will	Feature	Continuous	None	
12	word_freq_people	Feature	Continuous	None	
13	word_freq_report	Feature	Continuous	None	
14	word_freq_addresses	Feature	Continuous	None	
15	word_freq_free	Feature	Continuous	None	
16	word_freq_business	Feature	Continuous	None	
17	word_freq_email	Feature	Continuous	None	
18	word_freq_you	Feature	Continuous	None	
19	word_freq_credit	Feature	Continuous	None	
20	word_freq_your	Feature	Continuous	None	
21	word_freq_font	Feature	Continuous	None	
22	word_freq_000	Feature	Continuous	None	
23	word_freq_money	Feature	Continuous	None	
24	word_freq_hp	Feature	Continuous	None	
25	word_freq_hpl	Feature	Continuous	None	
26	word_freq_george	Feature	Continuous	None	
27	word_freq_650	Feature	Continuous	None	
28	word_freq_lab	Feature	Continuous	None	
29	word_freq_labs	Feature	Continuous	None	
30	word_freq_telnet	Feature	Continuous	None	
31	word_freq_857	Feature	Continuous	None	
32	word_freq_data	Feature	Continuous	None	
33	word_freq_415	Feature	Continuous	None	
34	word_freq_85	Feature	Continuous	None	
35	word_freq_technology	Feature	Continuous	None	
36	word_freq_1999	Feature	Continuous	None	
37	word_freq_parts	Feature	Continuous	None	
38	word_freq_pm	Feature	Continuous	None	
39	word_freq_direct	Feature	Continuous	None	
40	word_freq_cs	Feature	Continuous	None	
41	word_freq_meeting	Feature	Continuous	None	
42	word_freq_original	Feature	Continuous	None	
43	word_freq_project	Feature	Continuous	None	
44	word_freq_re	Feature	Continuous	None	
45	word_freq_edu	Feature	Continuous	None	
46	word_freq_table	Feature	Continuous	None	
47	word_freq_conference	Feature	Continuous	None	
48	char_freq_;	Feature	Continuous	None	
49	char_freq_(	Feature	Continuous	None	
50	char_freq_[	Feature	Continuous	None	
51	char_freq_!	Feature	Continuous	None	
52	char_freq_$	Feature	Continuous	None	
53	char_freq_#	Feature	Continuous	None	
54	capital_run_length_average	Feature	Continuous	None	
55	capital_run_length_longest	Feature	Continuous	None	
56	capital_run_length_total	Feature	Continuous	None	
57	Class	Target	Binary	None	

	description	units	missing_values
0	None	None	no
1	None	None	no

```
2                         None  None          no
3                         None  None          no
4                         None  None          no
5                         None  None          no
6                         None  None          no
7                         None  None          no
8                         None  None          no
9                         None  None          no
10                        None  None          no
11                        None  None          no
12                        None  None          no
13                        None  None          no
14                        None  None          no
15                        None  None          no
16                        None  None          no
17                        None  None          no
18                        None  None          no
19                        None  None          no
20                        None  None          no
21                        None  None          no
22                        None  None          no
23                        None  None          no
24                        None  None          no
25                        None  None          no
26                        None  None          no
27                        None  None          no
28                        None  None          no
29                        None  None          no
30                        None  None          no
31                        None  None          no
32                        None  None          no
33                        None  None          no
34                        None  None          no
35                        None  None          no
36                        None  None          no
37                        None  None          no
38                        None  None          no
39                        None  None          no
40                        None  None          no
41                        None  None          no
42                        None  None          no
43                        None  None          no
44                        None  None          no
45                        None  None          no
46                        None  None          no
47                        None  None          no
48                        None  None          no
49                        None  None          no
50                        None  None          no
51                        None  None          no
52                        None  None          no
53                        None  None          no
54                        None  None          no
55                        None  None          no
56                        None  None          no
57   spam (1) or not spam (0)  None          no
```

2. **Loading Dataset:** Load the data with a 70 : 15 : 15 split for train, validation, and testing

In [8]:
```python
# Split the data into train, validation, and test sets
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)
X_valid, X_test, y_valid, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

# Check the sizes of the sets

print("Train set size: {:.0f}%".format(X_train.shape[0]*100/X.shape[0]))
print("Validation set size: {:.0f}%".format(X_valid.shape[0]*100/X.shape[0]))
print("Test set size: {:.0f}%".format(X_test.shape[0]*100/X.shape[0]))
```

```
Train set size: 70%
Validation set size: 15%
Test set size: 15%
```

In [9]:
```python
# Count null values in X
null_values_in_X = X.isnull().sum().sum()

# Count null values in y
null_values_in_y = y.isnull().sum().sum()

# Print the counts
print("Number of null values in X:", null_values_in_X)
print("Number of null values in y:", null_values_in_y)
```

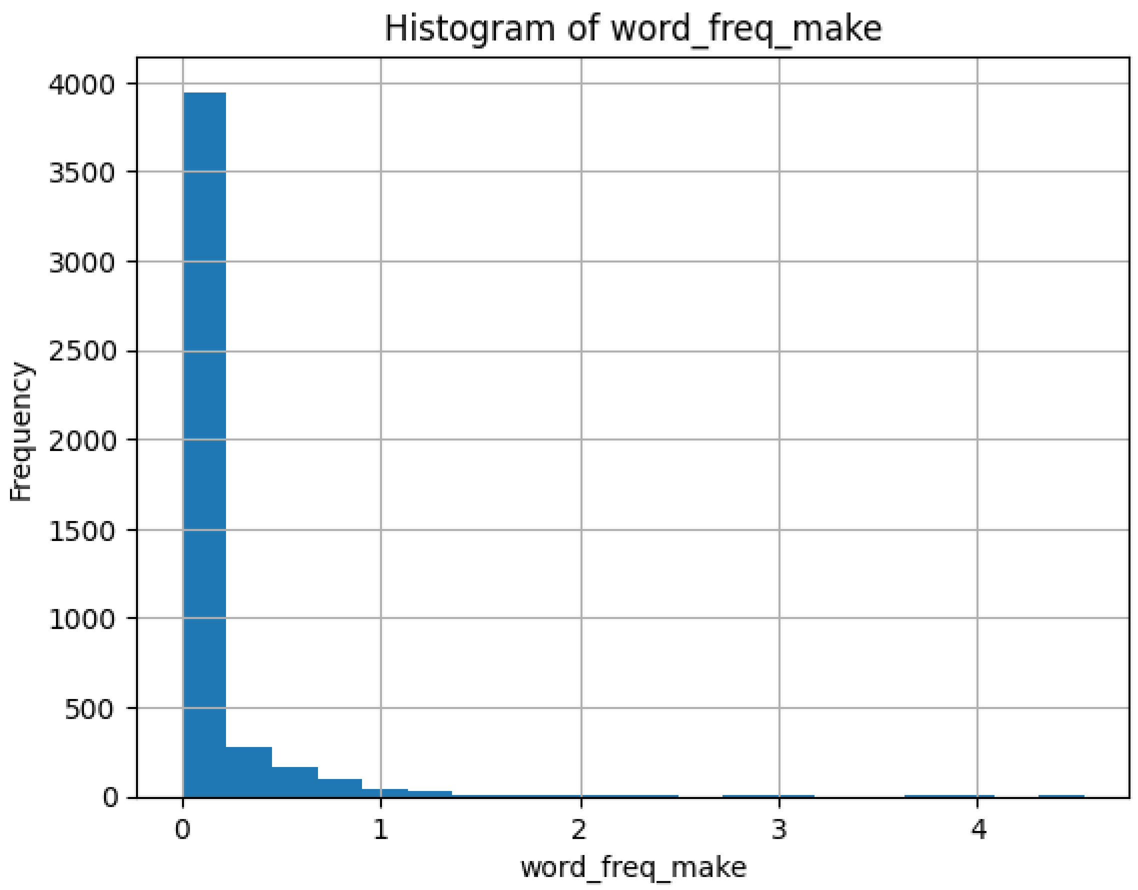
```
Number of null values in X: 0
Number of null values in y: 0
```

2. **Plot Distribution:** Choose some 5 columns from the dataset and plot the probability distribution.

In [10]:
```python
col = X.columns
column = col[0]
print(column)
X[col[0]].hist(bins=20)
```

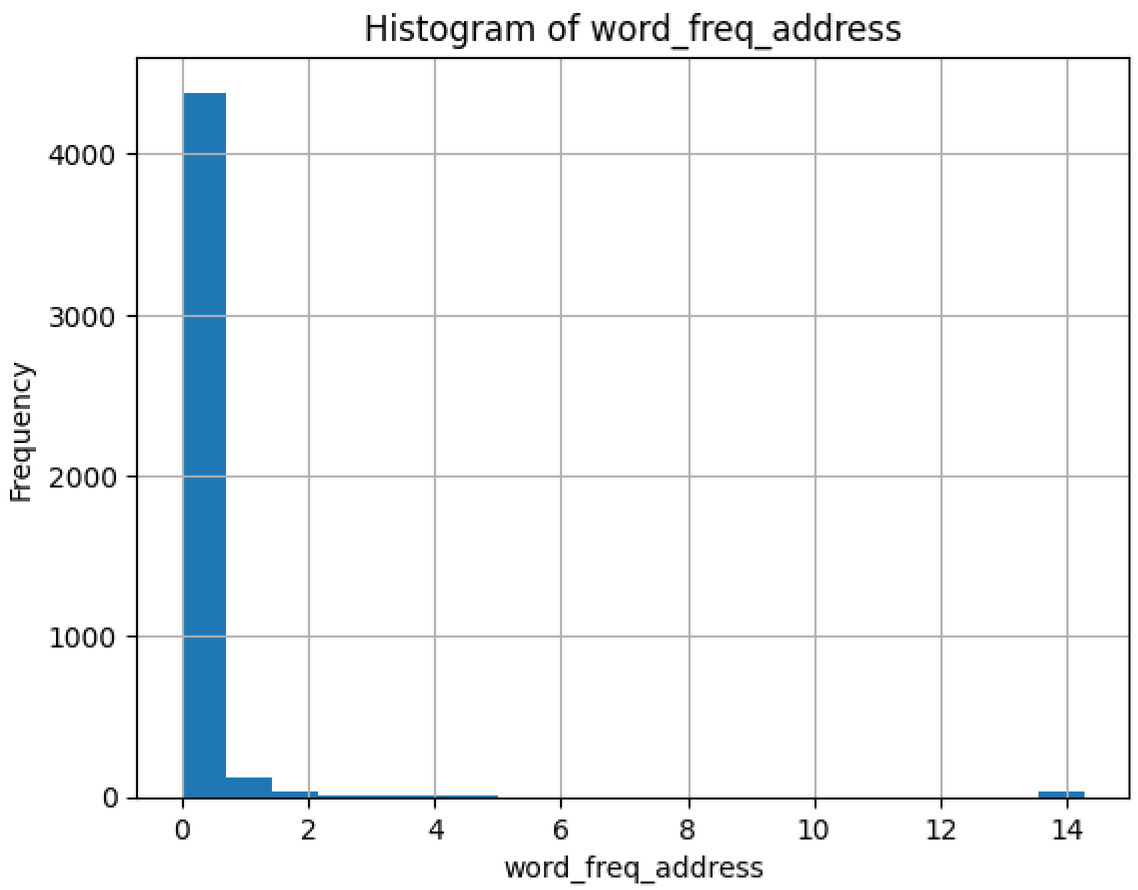
```
plt.title(f'Histogram of {column}')
plt.xlabel(column)
plt.ylabel('Frequency')
plt.show()
```

```
word_freq_make
```



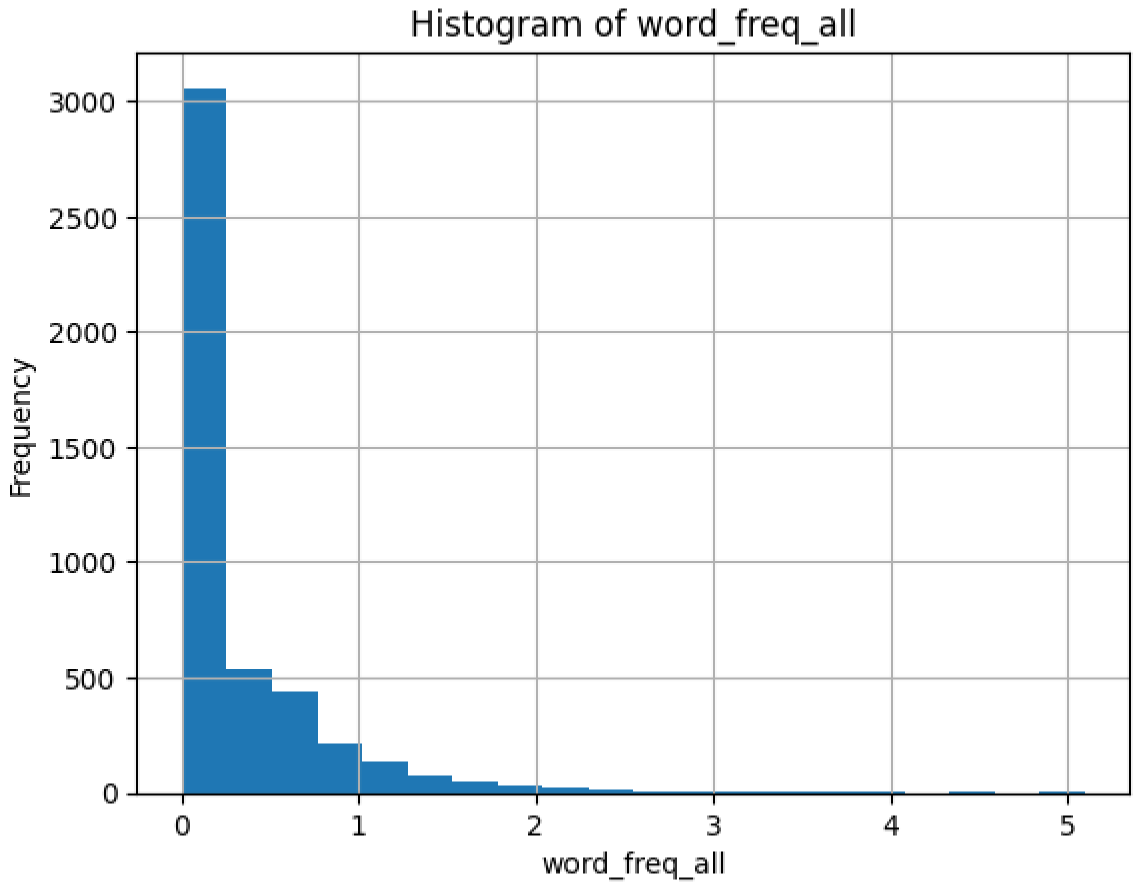
In [11]:
```
column = col[1]
print(column)
X[col[1]].hist(bins=20)
plt.title(f'Histogram of {column}')
plt.xlabel(column)
plt.ylabel('Frequency')
plt.show()
```

```
word_freq_address
```



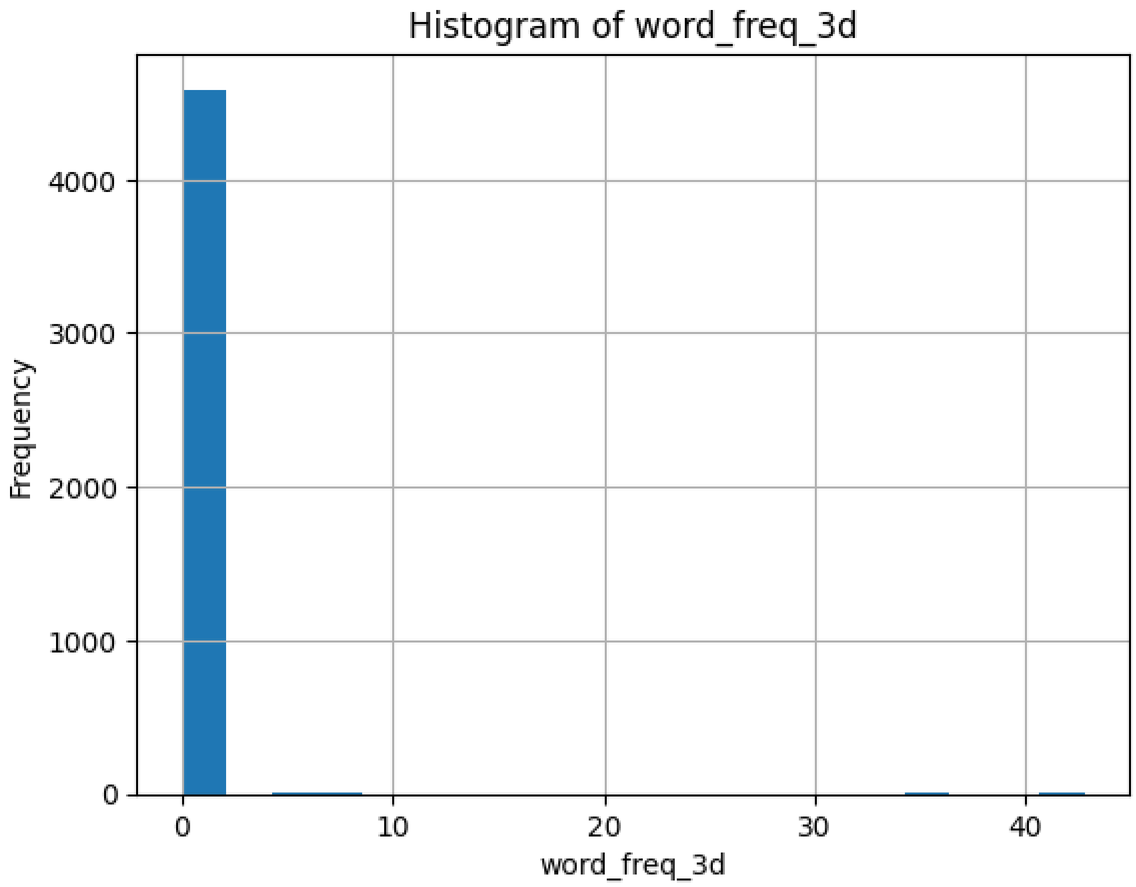
In [12]:
```
column = col[2]
print(column)
X[col[2]].hist(bins=20)
plt.title(f'Histogram of {column}')
plt.xlabel(column)
plt.ylabel('Frequency')
plt.show()
```

```
word_freq_all
```



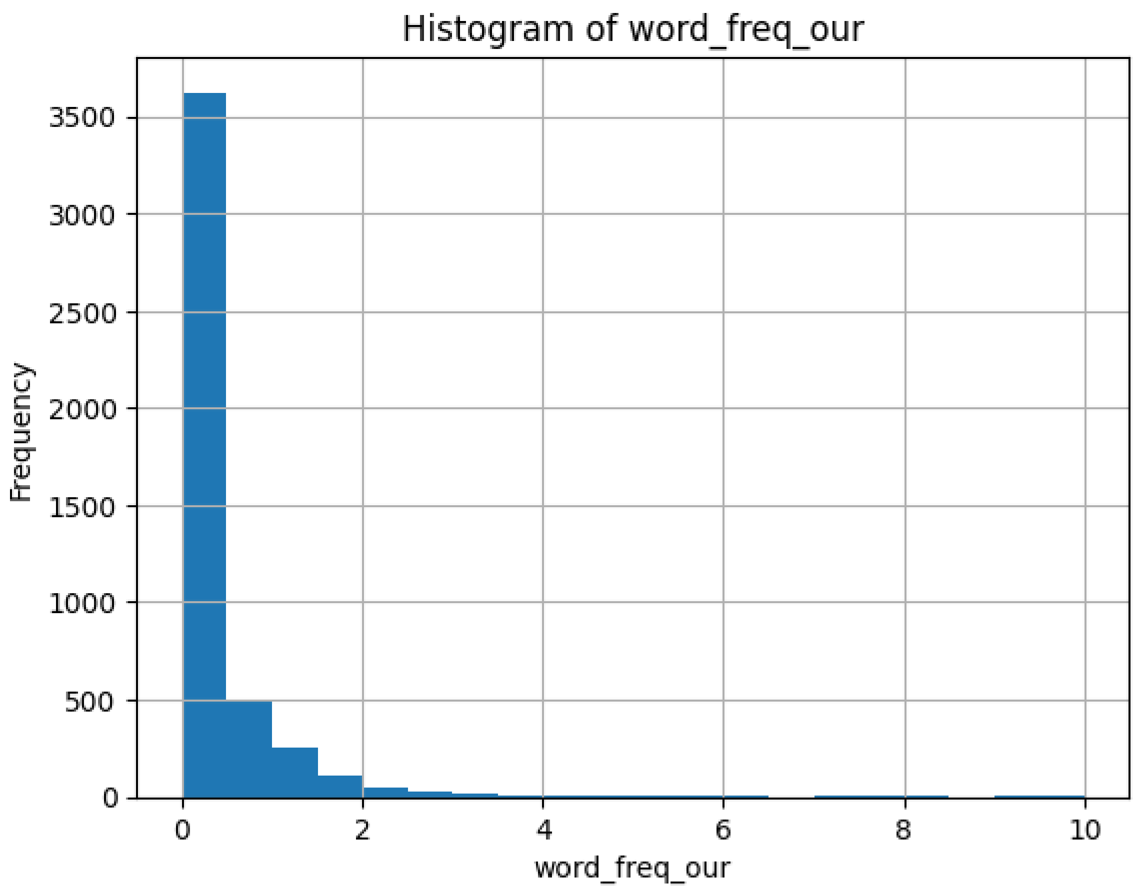
In [13]:
```
column = col[3]
print(column)
X[col[3]].hist(bins=20)
plt.title(f'Histogram of {column}')
plt.xlabel(column)
plt.ylabel('Frequency')
plt.show()
```

```
word_freq_3d
```



In [14]:
```
column = col[4]
print(column)
X[col[4]].hist(bins=20)
plt.title(f'Histogram of {column}')
plt.xlabel(column)
plt.ylabel('Frequency')
plt.show()
```

```
word_freq_our
```



In [15]:
```
print(y_train)
```

```
      Class
132       1
1358      1
2569      0
2842      0
4536      0
...     ...
1861      0
2366      0
266       1
277       1
3114      0

[3220 rows x 1 columns]
```

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

5. **Train Model:** Implement the Naive Bayes algorithm from scratch (preferably object-orientedimplementation with fit and predict function, this will make your later questions easier tohandle). Also, mention the total number of parameters needed to be stored for the model.

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

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

In [16]:
```
# Laplace smoothing parameter (you can adjust this value)
alpha = 1.0

class GaussianNaiveBayes:
    def fit(self, X, y):
        self.X = X
        self.y = y
        self.classes = np.unique(y)
        self.class_priors = self.calculate_class_priors()
        self.means, self.variances = self.calculate_class_stats()

    def calculate_class_priors(self):
        class_priors = {}
        total_samples = len(self.y)
        for c in self.classes:
            class_priors[c] = (len(self.y[self.y == c]) + alpha) / (total_samples + alpha * len(self.classes))
        return class_priors

    def calculate_class_stats(self):
        means = {}
        variances = {}
        for c in self.classes:
            class_samples = self.X[self.y == c]
            means[c] = np.mean(class_samples, axis=0)
            variances[c] = np.var(class_samples, axis=0)
        return means, variances

    def calculate_likelihood(self, x, mean, variance):
        exponent = np.exp(-(x - mean) ** 2 / (2 * variance))
        return (1 / (np.sqrt(2 * np.pi * variance))) * exponent

    def predict(self, X_test):
        predictions = []
        for x in X_test:
```

```
            posteriors = {}
            for c in self.classes:
                prior = np.log(self.class_priors[c])
                x = x.astype(np.float64)
                likelihoods = np.log(self.calculate_likelihood(x, self.means[c], self.variances[c]))
                posteriors[c] = np.sum(likelihoods) + prior
            predictions.append(max(posteriors, key=posteriors.get))
        return predictions

    def print_class_priors(self):
        print("Class Priors:")
        for c in self.classes:
            print(f"Class {c}: {self.class_priors[c]}")

    def print_class_priors_and_attributes(self):
        print("Class Priors:")
        for c in self.classes:
            print(f"Class {c}: {self.class_priors[c]}")
        print("\nClass-Specific Means and Variances (Parameters):")
        for c in self.classes:
            print(f"Class {c}:")
            print("Means:", self.means[c])
            print("Variances:", self.variances[c])

    def total_parameters(self):
        num_params = len(self.classes)  # For class priors
        for c in self.classes:
            num_params += len(self.means[c]) + len(self.variances[c])  # For means and variances
        return num_params




X_train_df = pd.DataFrame(X_train)
y_train_df = pd.DataFrame(y_train)

# Concatenate feature data and target labels into a single DataFrame
data = pd.concat([X_train_df, y_train_df], axis=1)

# Convert the combined DataFrame to a NumPy array
data_array = data.to_numpy()

# Split the data into features (X_train) and labels (y_train)
X_train_final = data_array[:, :-1]  # All columns except the last one are features
y_train_final = data_array[:, -1]  # The last column is the target variable


X_valid_df = pd.DataFrame(X_valid)
y_valid_df = pd.DataFrame(y_valid)

# Concatenate feature data and target labels into a single DataFrame
data = pd.concat([X_valid_df, y_valid_df], axis=1)

# Convert the combined DataFrame to a NumPy array
data_array = data.to_numpy()

# Split the data into features (X_train) and labels (y_train)
X_valid_final = data_array[:, :-1]  # All columns except the last one are features
y_valid_final = data_array[:, -1]

# Initialize and train the Gaussian Naive Bayes classifier
nb_classifier = GaussianNaiveBayes()
nb_classifier.fit(X_train_final, y_train_final)

# Print the class priors
nb_classifier.print_class_priors()

X_test_df = pd.DataFrame(X_test)
y_test_df = pd.DataFrame(y_test)

# Concatenate feature data and target labels into a single DataFrame
data = pd.concat([X_test_df, y_test_df], axis=1)

# Convert the combined DataFrame to a NumPy array
data_array = data.to_numpy()

# Split the data into features (X_train) and labels (y_train)
X_test_final = data_array[:, :-1]  # All columns except the last one are features
y_test_final = data_array[:, -1]

# Evaluate the model on the test set
y_pred = nb_classifier.predict(X_test_final)

accuracy = accuracy_score(y_test_final, y_pred)

# Calculate precision
precision = precision_score(y_test_final, y_pred)

# Calculate recall
recall = recall_score(y_test_final, y_pred)

# Calculate F1-score
f1 = f1_score(y_test_final, y_pred)
```

```
Class Priors:
Class 0.0: 0.6058348851644941
Class 1.0: 0.39416511483550587
```

5. **Train Model:** Implement the Naive Bayes algorithm from scratch (preferably object-orientedimplementation with fit and predict function, this will make your later questions easier tohandle). Also, mention the total number of parameters needed to be stored for the model.

In [17]:
```python
nb_classifier.print_class_priors_and_attributes()
# Calculate the total number of parameters needed to be stored for the model
num_parameters = nb_classifier.total_parameters()
print("\nTotal Number of Parameters:", num_parameters)
```

```
Class Priors:
Class 0.0: 0.6058348851644941
Class 1.0: 0.39416511483550587

Class-Specific Means and Variances (Parameters):
Class 0.0:
Means: [7.43618657e-02 2.54623270e-01 1.93885187e-01 1.06611994e-03
 1.73772424e-01 4.25627883e-02 1.07585853e-02 3.90415172e-02
 3.82214249e-02 1.44869298e-01 2.48744234e-02 5.45376730e-01
 6.28856996e-02 5.37365454e-02 6.75550999e-03 8.12506407e-02
 4.76576115e-02 1.04864172e-01 1.26810354e+00 7.27319323e-03
 4.34848795e-01 5.23065095e-02 7.40133265e-03 2.10917478e-02
 9.11599180e-01 4.49989749e-01 1.25886725e+00 2.15612506e-01
 1.68836494e-01 1.60194772e-01 1.10235777e-01 8.01742696e-02
 1.54941056e-01 8.03434136e-02 1.86371092e-01 1.47447463e-01
 2.00650948e-01 1.78728857e-02 1.10896976e-01 8.39876986e-02
 6.05176832e-02 2.36232701e-01 6.80830343e-02 1.25633009e-01
 4.25135828e-01 2.69354177e-01 7.50896976e-03 5.35366479e-02
 5.19538698e-02 1.61447975e-01 2.32147617e-02 1.15513583e-01
 1.16919528e-02 1.78933880e-02 2.29566479e+00 1.74249103e+01
 1.57713480e+02]
Variances: [9.02344339e-02 2.82452245e+00 2.53954664e-01 6.23773691e-04
 3.18067163e-01 5.25905618e-02 1.56040376e-02 5.81779127e-02
 4.51809986e-02 2.22656146e-01 2.86115552e-02 9.65189902e-01
 7.44433898e-02 1.60975223e-01 7.09295868e-03 5.17470091e-01
 4.64378243e-02 1.81628980e-01 3.28047564e+00 9.50686488e-03
 1.02796275e+00 4.45421231e-01 5.38489224e-03 1.02093580e-01
 4.48648581e+00 1.36883352e+00 1.76188660e+01 4.73547639e-01
 6.29633551e-01 3.25202422e-01 3.01662476e-01 1.91131184e-01
 5.49148215e-01 1.91255597e-01 5.58299696e-01 2.83223315e-01
 2.48175025e-01 6.98803037e-02 2.00897760e-01 1.95477179e-01
 1.68144632e-01 1.02491984e+00 7.67977604e-02 4.62881493e-01
 1.49937977e+00 1.20588959e+00 8.45314895e-03 1.64959609e-01
 1.04824290e-01 7.63215471e-02 2.30370994e-02 8.85212040e-01
 5.43974878e-03 3.00372075e-02 4.16726655e+00 7.54748718e+02
 1.29257927e+05]
Class 1.0:
Means: [1.58613081e-01 1.65697400e-01 4.00740741e-01 1.47635934e-01
 5.31323877e-01 1.81449961e-01 2.85153664e-01 1.92923562e-01
 1.73293932e-01 3.59219858e-01 1.25973207e-01 5.55610717e-01
 1.45807723e-01 8.02758077e-02 1.09700552e-01 5.09282900e-01
 2.96918834e-01 3.27501970e-01 2.25037037e+00 2.17856580e-01
 1.38118991e+00 1.80543735e-01 2.48873128e-01 2.09141056e-01
 2.16942474e-02 1.11583924e-02 2.07249803e-03 2.23167849e-02
 4.64933018e-04 8.24271080e-03 1.82033097e-03 3.70370370e-04
 1.60677699e-02 6.69818755e-04 5.37431048e-03 2.88337273e-02
 4.42237983e-02 4.77541371e-03 1.22379827e-02 3.94405043e-02
 7.88022065e-05 2.61623325e-03 8.79432624e-03 6.85579196e-03
 1.22356186e-01 1.11505122e-02 1.02442868e-03 2.42710796e-03
 1.82033097e-02 1.15275020e-01 8.76516942e-03 5.09695035e-01
 1.78569740e-01 7.86414500e-02 1.03185595e+01 1.06438928e+02
 4.52844760e+02]
Variances: [1.07477509e-01 1.24822938e-01 2.26922068e-01 4.01090111e+00
 5.08674528e-01 1.12721144e-01 3.55155630e-01 2.00223683e-01
 1.28604595e-01 4.08566688e-01 7.29422562e-02 4.15466077e-01
 1.37536562e-01 9.26933045e-02 1.42168232e-01 1.01131903e+00
 4.31277268e-01 4.72228196e-01 2.41211137e+00 7.29519125e-01
 1.41937691e+00 1.30141649e+00 2.55908817e-01 2.98881138e-01
 3.42615109e-02 1.24381380e-02 1.56615392e-03 1.27707950e-01
 4.05245780e-05 1.51610569e-02 1.81867221e-03 1.73936900e-04
 1.40665707e-02 2.07037552e-04 2.05039181e-03 2.14017919e-02
 6.78637986e-02 2.99847202e-03 8.69774951e-03 2.56946436e-02
 7.87401086e-06 8.21681722e-04 2.81217913e-03 4.33897132e-03
 8.77475611e-02 9.64847932e-03 2.82323280e-04 9.12785270e-04
 6.80265055e-03 1.07103682e-01 2.77212295e-03 5.04020374e-01
 1.55282300e-01 4.89373092e-01 3.07867121e+03 1.13370637e+05
 6.22561186e+05]

Total Number of Parameters: 230
```

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

In [18]:
```python
print("\nAccuracy: {:.2f}".format(accuracy))
print("Precision: {:.2f}".format(precision))
print("Recall: {:.2f}".format(recall))
print("F1 Score: {:.2f}".format(f1))
print()
```

```
Accuracy: 0.82
Precision: 0.69
Recall: 0.97
F1 Score: 0.81
```

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

In [19]:
```python
def log_transform(X):
    return np.log1p(X)
X_train_log_transformed = log_transform(X_train_final)
X_test_log_transformed = log_transform(X_test_final)
X_valid_log_transformed = log_transform(X_valid_final)
print(X_train_log_transformed, X_train_final)
```

```
[[0.         0.         0.75141609 ... 1.21194097 2.99573227 4.97673374]
 [0.27763174 0.         0.49469624 ... 1.5288784  4.15888308 5.7651911 ]
 [0.         0.         0.         ... 0.88418068 2.19722458 4.52178858]
 ...
 [0.         0.         0.         ... 1.90016534 3.68887945 4.52178858]
 [0.         0.45107562 0.45107562 ... 2.84589727 4.99721227 5.9242558 ]
 [0.         0.         0.         ... 1.25276297 2.77258872 4.18965474]] [[0.0000e+00 0.0000e+00 1.1200e+00 ... 2.3600e+00 1.90
00e+01 1.4400e+02]
 [3.2000e-01 0.0000e+00 6.4000e-01 ... 3.6130e+00 6.3000e+01 3.1800e+02]
 [0.0000e+00 0.0000e+00 0.0000e+00 ... 1.4210e+00 8.0000e+00 9.1000e+01]
 ...
 [0.0000e+00 0.0000e+00 0.0000e+00 ... 5.6870e+00 3.9000e+01 9.1000e+01]
 [0.0000e+00 5.7000e-01 5.7000e-01 ... 1.6217e+01 1.4700e+02 3.7300e+02]
 [0.0000e+00 0.0000e+00 0.0000e+00 ... 2.5000e+00 1.5000e+01 6.5000e+01]]
```

In [20]:
```python
# Initialize and train the Gaussian Naive Bayes classifier
nb_classifier = GaussianNaiveBayes()
nb_classifier.fit(X_train_log_transformed, y_train_final)

# Print the class priors
nb_classifier.print_class_priors()

# Evaluate the model on the test set
y_pred = nb_classifier.predict(X_test_log_transformed)

accuracy = accuracy_score(y_test_final, y_pred)

# Calculate precision
precision = precision_score(y_test_final, y_pred)

# Calculate recall
recall = recall_score(y_test_final, y_pred)

# Calculate F1-score
f1 = f1_score(y_test_final, y_pred)

print("\nAccuracy: {:.2f}".format(accuracy))
print("Precision: {:.2f}".format(precision))
print("Recall: {:.2f}".format(recall))
print("F1 Score: {:.2f}".format(f1))
print()
nb_classifier.print_class_priors_and_attributes()

# Calculate the total number of parameters needed to be stored for the model
num_parameters = nb_classifier.total_parameters()
print("\nTotal Number of Parameters:", num_parameters)
```

```
Class Priors:
Class 0.0: 0.6058348851644941
Class 1.0: 0.39416511483550587

Accuracy: 0.84
Precision: 0.71
Recall: 0.97
F1 Score: 0.82

Class Priors:
Class 0.0: 0.6058348851644941
Class 1.0: 0.39416511483550587

Class-Specific Means and Variances (Parameters):
Class 0.0:
Means: [5.14142096e-02 7.35123602e-02 1.27531844e-01 8.48728406e-04
 1.08781303e-01 3.09101336e-02 7.10666350e-03 2.63718515e-02
 2.72110198e-02 9.16962009e-02 1.74377849e-02 3.09423516e-01
 4.40938823e-02 2.86680274e-02 4.82454027e-03 4.15234335e-02
 3.46422208e-02 6.53761500e-02 5.87281656e-01 5.02003411e-03
 2.40755601e-01 1.42250411e-02 5.82291259e-03 1.05553076e-02
 3.82330615e-01 2.28687657e-01 3.25531567e-01 1.19121698e-01
 8.49654012e-02 9.38378493e-02 6.21493896e-02 4.53311428e-02
 8.07876259e-02 4.54490698e-02 1.04306076e-01 8.99362105e-02
 1.33364765e-01 8.72808249e-03 6.72445517e-02 4.79134291e-02
 3.28230017e-02 1.06679241e-01 4.72054440e-02 6.50507763e-02
 2.17174368e-01 1.24505422e-01 5.20836099e-03 3.00059996e-02
 3.34908086e-02 1.30467071e-01 1.80404945e-02 6.63806199e-02
 9.92003429e-03 1.32857005e-02 1.11076319e+00 2.32839363e+00
 3.99839510e+00]
Variances: [3.02273457e-02 1.26642586e-01 7.64380690e-02 3.63341928e-04
 7.33151671e-02 1.55202280e-02 5.05016888e-03 1.67033122e-02
 1.52838676e-02 6.45457937e-02 1.04896558e-02 2.04955847e-01
 2.52630806e-02 2.89844323e-02 2.86511810e-03 3.75793860e-02
 1.84088981e-02 4.87580473e-02 4.12801586e-01 3.14220352e-03
 1.78321469e-01 3.06774991e-02 2.42828419e-03 1.05392693e-02
 3.80661959e-01 2.02409034e-01 5.20027092e-01 1.06959566e-01
 8.52063292e-02 7.61681082e-02 5.40162896e-02 4.09980452e-02
 7.81785518e-02 4.10711824e-02 8.76891553e-02 6.64743618e-02
 7.78757332e-02 9.79012385e-03 5.33956840e-02 4.24489881e-02
 3.23095179e-02 1.25767710e-01 2.85679833e-02 6.42470576e-02
 1.85412884e-01 1.36196304e-01 3.40279834e-03 2.65304566e-02
 2.13739167e-02 3.31004769e-02 6.92716289e-03 3.98782509e-02
 2.79727352e-03 5.65323076e-03 1.26988998e-01 1.07833382e+00
 2.04930079e+00]
Class 1.0:
Means: [1.20133376e-01 1.24262245e-01 2.89513382e-01 2.69394278e-02
 3.47295011e-01 1.36878962e-01 1.89705672e-01 1.34188856e-01
 1.26685390e-01 2.36963597e-01 9.81360419e-02 3.72639484e-01
 1.06639237e-01 5.45223376e-02 7.13465102e-02 3.03596638e-01
 1.89596247e-01 2.07117373e-01 1.05024597e+00 1.15624441e-01
 7.47380910e-01 5.99245033e-02 1.69305149e-01 1.44317882e-01
 1.40653547e-02 7.66064071e-03 1.59671332e-03 9.85453573e-03
 4.45849722e-04 5.12775178e-03 1.28714640e-03 3.03595272e-04
 1.19814659e-02 5.84280152e-04 4.58678320e-03 2.17901482e-02
 2.86883913e-02 3.84134235e-03 9.46127589e-03 3.08508019e-02
 7.51065247e-05 2.28074235e-03 7.69093649e-03 5.41212288e-03
 9.12029531e-02 8.36359008e-03 9.10729514e-04 2.07931534e-03
 1.55953155e-02 9.55393706e-02 7.71429986e-03 3.47122798e-01
 1.37392283e-01 4.46237522e-02 1.65871074e+00 3.74502839e+00
 5.37612112e+00]
Variances: [4.59144916e-02 4.83572348e-02 8.78099502e-02 6.15899263e-02
 1.39389939e-01 5.19825605e-02 9.74715250e-02 6.72870847e-02
 5.63877753e-02 1.16154720e-01 3.51386213e-02 1.28244358e-01
 4.70195506e-02 3.47315362e-02 4.96573868e-02 1.66839976e-01
 1.08748264e-01 1.20861209e-01 2.86781704e-01 1.04270872e-01
 2.44522559e-01 1.01761227e-01 8.71160975e-02 6.78738556e-02
 1.02721011e-02 5.19424021e-03 7.46085208e-04 1.15689017e-02
 3.67468470e-05 4.07156147e-03 8.19919871e-04 1.16871673e-04
 6.14721201e-03 1.55074665e-04 1.35375308e-03 1.08204645e-02
 2.07891557e-02 1.48329650e-03 4.37905219e-03 1.29052092e-02
 7.15277538e-06 5.99185973e-04 1.88228186e-03 2.32777831e-03
 4.07297603e-02 4.19279898e-03 2.03169909e-04 6.04192855e-04
 4.24363786e-03 1.84933374e-02 1.76741478e-03 1.07109494e-01
 4.06694348e-02 2.91219165e-02 5.67021847e-01 1.53356282e+00
 1.47023234e+00]

Total Number of Parameters: 230
```

8. **Discuss:** Changes in the result is that the result improved for all accuracy, precision, recall and f1-score.

Part C: Implemention of Naive Bayes (sklearn)

1. **Train the model:** Import GaussianNB from sklearn.naivebayes. Train the model with the ***actually loaded** dataset and again after **log transformation**.

In [21]:
```python
# Train the model with the original dataset
nb_original = GaussianNB()
nb_original.fit(X_train_final, y_train_final)
```

```
# Train the model with the log-transformed dataset
nb_log_transformed = GaussianNB()
nb_log_transformed.fit(X_train_log_transformed, y_train_final)
```

Out[21]:

```
▾ GaussianNB
GaussianNB()
```

2. High precision:

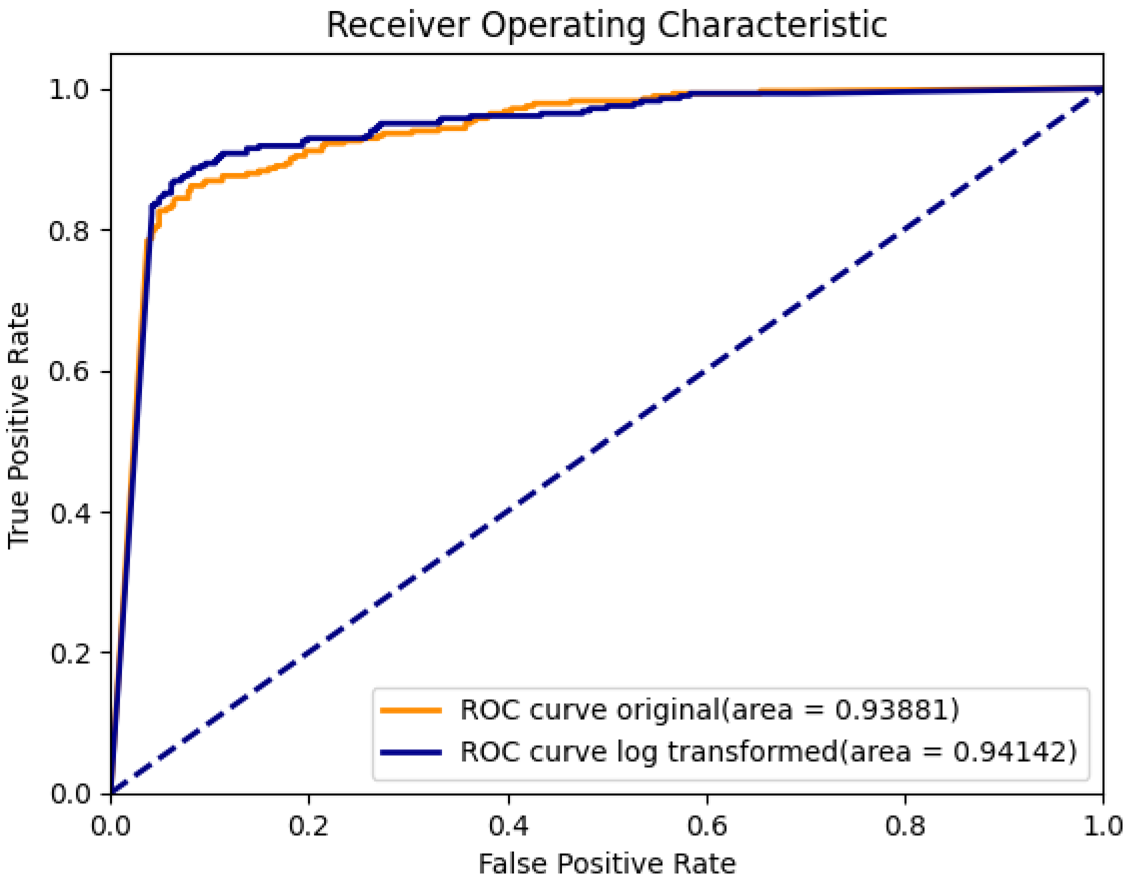
1. Drawing a ROC curve for two models we got previously.
2. As we understand the importance of emails, **we don't want not spam mail classified as spam (However very littleerror is acceptable)**.
3. So **Best model has High Precision or LOW FPR value**
4. Choosing one best model from the ROC curve

In [22]:

```
# Predict probabilities for the original and log-transformed models
y_scores_original = nb_original.predict_proba(X_valid_final)
y_scores_log_transformed = nb_log_transformed.predict_proba(X_valid_log_transformed)

y_pred_original = nb_original.predict(X_test_final)

# Calculate accuracy for the original model
accuracy_original = accuracy_score(y_test_final, y_pred_original)

# Calculate precision for the original model
precision_original = precision_score(y_test_final, y_pred_original)

# Calculate recall for the original model
recall_original = recall_score(y_test_final, y_pred_original)

# Calculate F1-score for the original model
f1_original = f1_score(y_test_final, y_pred_original)

print("Precision (Original Data): {:.2f}".format(precision_original))


y_pred_log_transformed = nb_log_transformed.predict(X_test_log_transformed)

# Calculate accuracy for the log-transformed model
accuracy_log_transformed = accuracy_score(y_test_final, y_pred_log_transformed)

# Calculate precision for the log-transformed model
precision_log_transformed = precision_score(y_test_final, y_pred_log_transformed)

# Calculate recall for the log-transformed model
recall_log_transformed = recall_score(y_test_final, y_pred_log_transformed)

# Calculate F1-score for the log-transformed model
f1_log_transformed = f1_score(y_test_final, y_pred_log_transformed)

print("Precision (Log Transformed Data): {:.2f}\n".format(precision_log_transformed))



# Compute ROC curve and AUC for the original model
fpr_original, tpr_original, _ = roc_curve(y_valid, y_scores_original[:, 1])
roc_auc_original = auc(fpr_original, tpr_original)

# Compute ROC curve and AUC for the log-transformed model
fpr_log_transformed, tpr_log_transformed, _ = roc_curve(y_valid_final, y_scores_log_transformed[:, 1])

roc_auc_log_transformed = auc(fpr_log_transformed, tpr_log_transformed)

roc_auc_original = auc(fpr_original, tpr_original)


# Assuming you have computed ROC curves for your models (e.g., fpr_log_transformed, tpr_log_transformed)
# Compute AUC for the log-transformed model
roc_auc_log_transformed = auc(fpr_log_transformed, tpr_log_transformed)

# Define the desired TPR
desired_tpr = 0.95  # Adjust this value based on your preference

# Find the FPR closest to the desired TPR for the original model
idx_original = np.argmax(tpr_original >= desired_tpr)
fpr_at_desired_tpr_original = fpr_original[idx_original]

# Find the FPR closest to the desired TPR for the log-transformed model
idx_log_transformed = np.argmax(tpr_log_transformed >= desired_tpr)
fpr_at_desired_tpr_log_transformed = fpr_log_transformed[idx_log_transformed]

# Choose the best model based on low FPR at the desired TPR
best_model = "Original" if fpr_at_desired_tpr_original < fpr_at_desired_tpr_log_transformed else "Log-Transformed"
best_auc = roc_auc_original if fpr_at_desired_tpr_original < fpr_at_desired_tpr_log_transformed else roc_auc_log_transformed
if(best_model == "Original"):
    best_precision = precision_original
else:
    if (best_model == "Log-Transformed"):
        best_precision = precision_log_transformed
```

```
print("The best model is the {} model\nBased on a LOW FPR and HIGH PRECISION at TPR = {:.3f}\nWith AUC as {:.3f} and Precsion a



# Plot ROC curves
plt.figure()
plt.plot(fpr_original, tpr_original, color='darkorange', lw=2, label='ROC curve original(area = {:.5f})'.format(roc_auc_origina
plt.plot(fpr_log_transformed, tpr_log_transformed, color='darkblue', lw=2, label='ROC curve log transformed(area = {:.5f})'.for
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')
plt.title('Receiver Operating Characteristic')
plt.legend(loc='lower right')
plt.show()
```

```
Precision (Original Data): 0.70
Precision (Log Transformed Data): 0.71

The best model is the Log-Transformed model
Based on a LOW FPR and HIGH PRECISION at TPR = 0.950
With AUC as 0.941 and Precsion as 0.713.
```



3. Compare Accuracy: Comparing and discuss the accuracy of Naive Bayes and SVM.

In [23]:

```
# Train the Naive Bayes model
nb_classifier = GaussianNB()
nb_classifier.fit(X_train, y_train)

# Train the SVM model
svm_classifier = SVC()
svm_classifier.fit(X_train, y_train)

# Evaluate Naive Bayes accuracy
y_pred_nb = nb_classifier.predict(X_test)
accuracy_nb = accuracy_score(y_test, y_pred_nb)

# Evaluate SVM accuracy
y_pred_svm = svm_classifier.predict(X_test)
accuracy_svm = accuracy_score(y_test, y_pred_svm)

# Compare and discuss accuracy
print("Naive Bayes Accuracy:", accuracy_nb)
print("SVM Accuracy:", accuracy_svm)

if accuracy_nb > accuracy_svm:
    print("\nNaive Bayes has higher accuracy.")
else:
    print("\nSVM has higher accuracy.")
```

```
Naive Bayes Accuracy: 0.829232995658466
SVM Accuracy: 0.7163531114327062

Naive Bayes has higher accuracy.
```

In [ ]: