q-2

November 6, 2023

1 Name: Pola Gnana Shekar

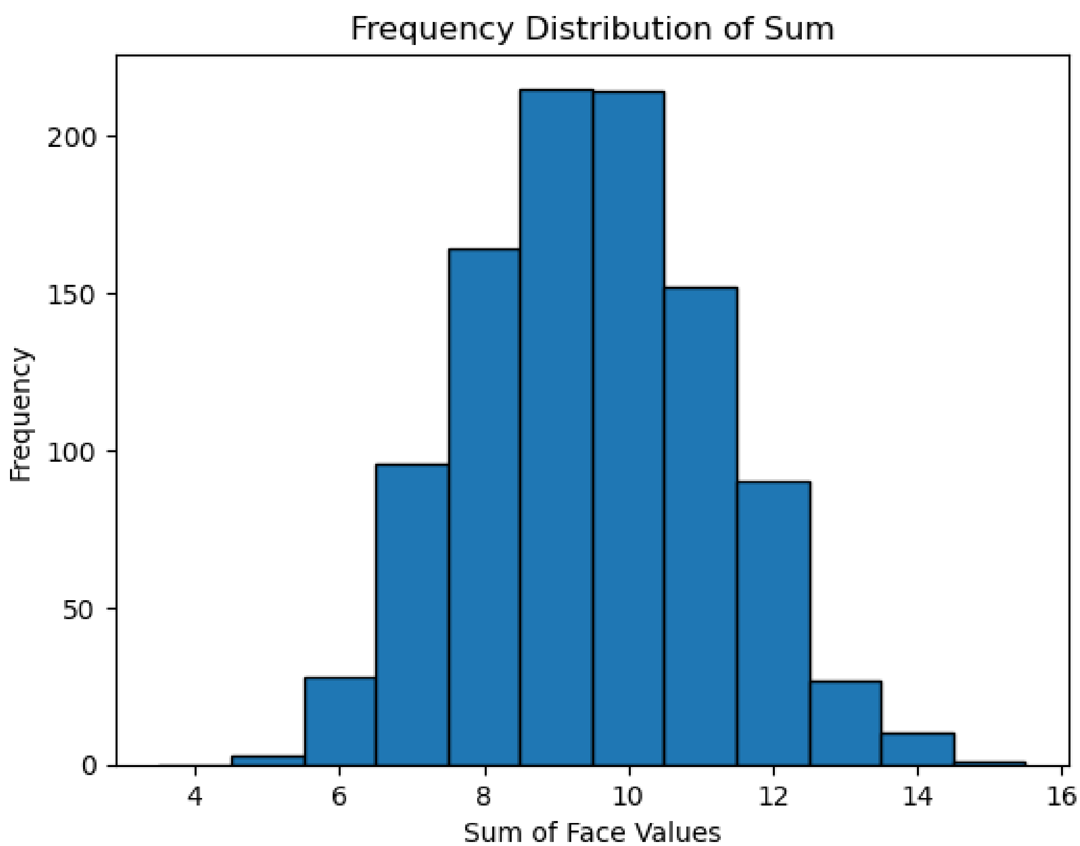
2 Roll No: 21CS10052

2.1 Probability

[1]:

```python
# Rolling a 4-faced die 4 times

import random
import matplotlib.pyplot as plt
import numpy as np

# Define the number of simulations and the number of dice rolls per simulation
num_simulations = 1000
num_rolls = 4
k = 4

# Initialize an empty list to store the sums of the upward face values
sums = []

# Create a dictionary to map face values to their weights
weights_dict = {i: 1 / (2 ** (i - 1)) for i in range(2, k + 1)}
weights_dict[1] = 1 / (2 ** (k - 1))

# Simulate rolling the biased die and calculating the sum
for _ in range(num_simulations):
    roll_sum = 0
    for _ in range(num_rolls):
        # Generate a random face value based on the corrected biased␣
↪probabilities
        face_value = random.choices(list(weights_dict.keys()),␣
↪list(weights_dict.values()))[0]
        roll_sum += face_value
    sums.append(roll_sum)

# Plot a histogram of the sums
```

```python
plt.hist(sums, bins=range(num_rolls, k * num_rolls + 1), edgecolor='k',␣
    ↪align='left')
plt.title("Frequency Distribution of Sum")
plt.xlabel("Sum of Face Values")
plt.ylabel("Frequency")
plt.show()

# Calculate the five-number summary
q1, median, q3 = np.percentile(sums, [25, 50, 75])
min_value, max_value = min(sums), max(sums)

# Calculate the mean of the sums list
mean_value = sum(sums) / len(sums)

print("Five-Number Summary:")
print(f"Minimum: {min_value}")
print(f"Q1 (25th percentile): {q1}")
print(f"Median (50th percentile): {median}")
print(f"Q3 (75th percentile): {q3}")
print(f"Maximum: {max_value}")
print(f"Mean: {mean_value}")
```



```
Five-Number Summary:
Minimum: 5
Q1 (25th percentile): 8.0
Median (50th percentile): 9.0
Q3 (75th percentile): 11.0
Maximum: 15
Mean: 9.5
```

[2]:
```
# theoretical calculation for the expected value.

def generatePermutationsWithRepetition(currentPermutation, n, k,␣
↪allPermutations):
    if n == 0:
        allPermutations.append(list(currentPermutation))
        return

    for i in range(1, k + 1):
        currentPermutation.append(i)
        generatePermutationsWithRepetition(currentPermutation, n - 1, k,␣
↪allPermutations)
        currentPermutation.pop()

def findProb(A, P, prob):
    sum_val = 0
    x = 1.0
    for permutation in A:
        for num in permutation:
            sum_val += num
            x *= prob[num - 1]
        P[sum_val] = P.get(sum_val, 0.0) + x
        sum_val = 0
        x = 1.0

if __name__ == '__main__':
    n, k = 4, 4
    allPermutations = []
    currentPermutation = []

    generatePermutationsWithRepetition(currentPermutation, n, k,␣
↪allPermutations)

    print("Probabilities of all permutations possible:")
    P = {}

    for i in range(n, n * k + 1):
        P[i] = 0.0
```

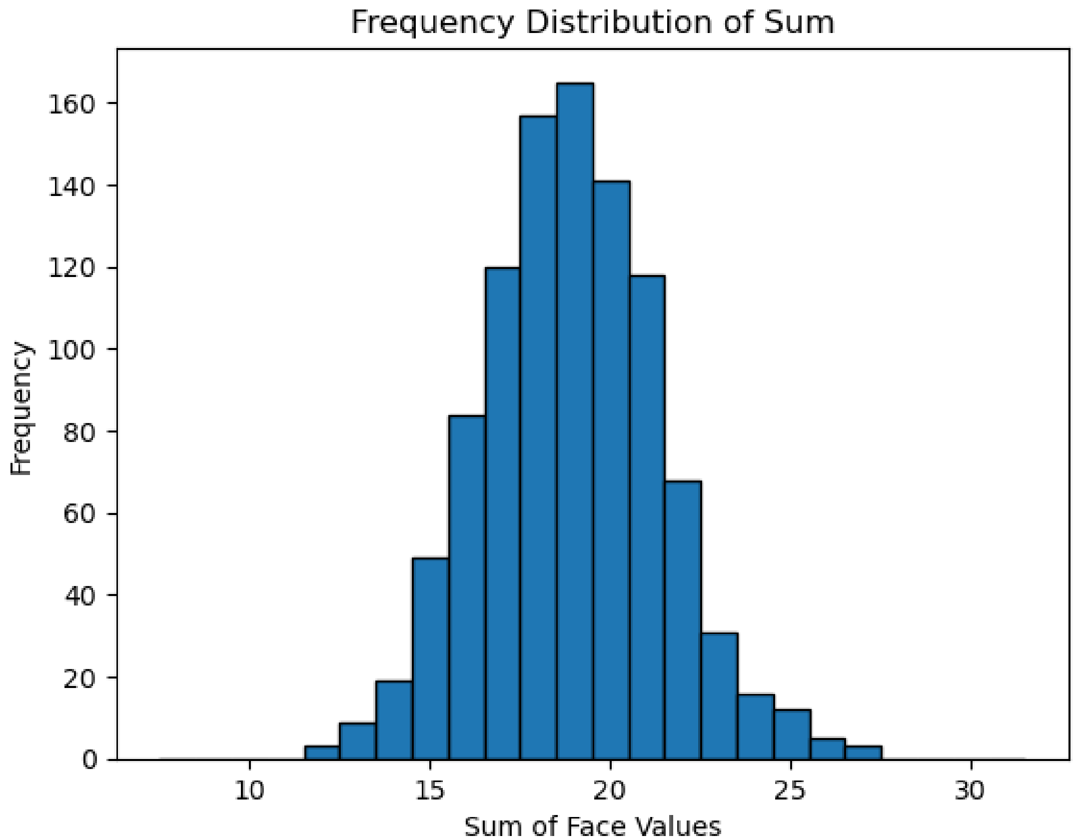
```
    prob = [0.125, 0.5, 0.25, 0.125]
    findProb(allPermutations, P, prob)

    for i in range(n, n * k + 1):
        print(f"P[{i}] = {P[i]}")

    ExValue = 0.0

    # using the formula E[X]= sum for all xi(xi*P(X=xi))
    for i in range(n, n * k + 1):
        ExValue += i * P[i]

    print(f"\nTheoretical Expected sum of the upward face value : {ExValue}")
```

```
Probabilities of all permutations possible:
P[4] = 0.000244140625
P[5] = 0.00390625
P[6] = 0.025390625
P[7] = 0.0869140625
P[8] = 0.173828125
P[9] = 0.224609375
P[10] = 0.21240234375
P[11] = 0.1484375
P[12] = 0.080078125
P[13] = 0.0322265625
P[14] = 0.009765625
P[15] = 0.001953125
P[16] = 0.000244140625

Theoretical Expected sum of the upward face value : 9.5
```

The theortical value of Expected sum of upward face value is 9.5 which is close to what we have got in the python simualtion that is 9.423

[3]:
```
# rolling a 4 faced die 8 times

import random
import matplotlib.pyplot as plt
import numpy as np

# Define the number of simulations and the number of dice rolls per simulation
num_simulations = 1000
num_rolls = 8
k = 4

# Initialize an empty list to store the sums of the upward face values
sums = []
```

```python
# Create a dictionary to map face values to their weights
weights_dict = {i: 1 / (2 ** (i - 1)) for i in range(2, k + 1)}
weights_dict[1] = 1 / (2 ** (k - 1))

# Simulate rolling the biased die and calculating the sum
for _ in range(num_simulations):
    roll_sum = 0
    for _ in range(num_rolls):
        # Generate a random face value based on the corrected biased␣
↪probabilities
        face_value = random.choices(list(weights_dict.keys()),␣
↪list(weights_dict.values()))[0]
        roll_sum += face_value
    sums.append(roll_sum)

# Plot a histogram of the sums
plt.hist(sums, bins=range(num_rolls, k * num_rolls + 1), edgecolor='k',␣
↪align='left')
plt.title("Frequency Distribution of Sum")
plt.xlabel("Sum of Face Values")
plt.ylabel("Frequency")
plt.show()

# Calculate the five-number summary
q1, median, q3 = np.percentile(sums, [25, 50, 75])
min_value, max_value = min(sums), max(sums)

# Calculate the mean of the sums list
mean_value = sum(sums) / len(sums)

print("Five-Number Summary:")
print(f"Minimum: {min_value}")
print(f"Q1 (25th percentile): {q1}")
print(f"Median (50th percentile): {median}")
print(f"Q3 (75th percentile): {q3}")
print(f"Maximum: {max_value}")
print(f"Mean: {mean_value}")
```




```
Five-Number Summary:
Minimum: 12
Q1 (25th percentile): 17.0
Median (50th percentile): 19.0
Q3 (75th percentile): 21.0
Maximum: 27
Mean: 18.901
```

[4]:

```
# theoretical calculation for the expected value.

def generatePermutationsWithRepetition(currentPermutation, n, k,␣
↪allPermutations):
    if n == 0:
        allPermutations.append(list(currentPermutation))
        return

    for i in range(1, k + 1):
        currentPermutation.append(i)
        generatePermutationsWithRepetition(currentPermutation, n - 1, k,␣
↪allPermutations)
```

```
        currentPermutation.pop()

def findProb(A, P, prob):
    sum_val = 0
    x = 1.0
    for permutation in A:
        for num in permutation:
            sum_val += num
            x *= prob[num - 1]
        P[sum_val] = P.get(sum_val, 0.0) + x
        sum_val = 0
        x = 1.0

if __name__ == '__main__':
    n, k = 8, 4
    allPermutations = []
    currentPermutation = []

    generatePermutationsWithRepetition(currentPermutation, n, k,␣
 ↪allPermutations)

    print("Probabilities of all permutations possible:")
    P = {}

    for i in range(n, n * k + 1):
        P[i] = 0.0

    prob = [0.125, 0.5, 0.25, 0.125]
    findProb(allPermutations, P, prob)

    for i in range(n, n * k + 1):
        print(f"P[{i}] = {P[i]}")

    ExValue = 0.0

    # using the formula E[X]= sum for all xi(xi*P(X=xi))
    for i in range(n, n * k + 1):
        ExValue += i * P[i]

    print(f"\nTheoretical Expected sum of the upward face value : {ExValue}")
```

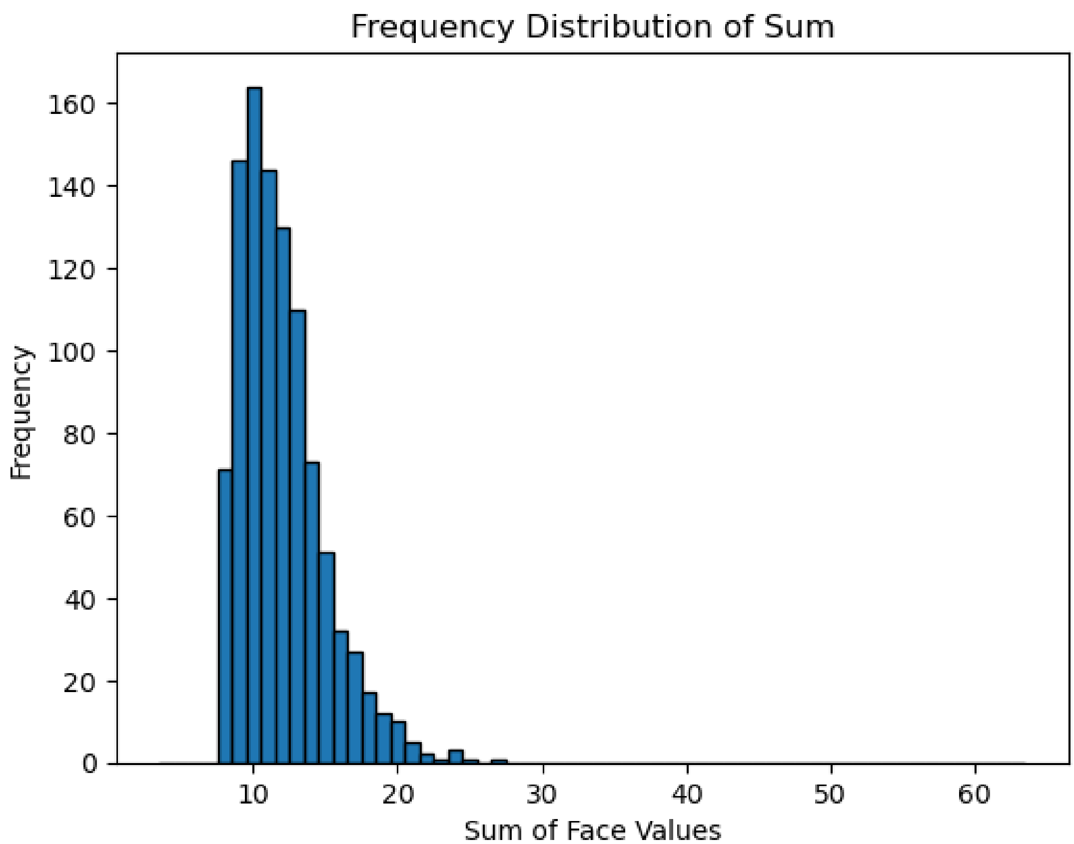
```
Probabilities of all permutations possible:
P[8] = 5.960464477539063e-08
P[9] = 1.9073486328125e-06
P[10] = 2.765655517578125e-05
P[11] = 0.00024080276489257812
P[12] = 0.0014085769653320312
```

```
P[13] = 0.005881309509277344
P[14] = 0.018239736557006836
P[15] = 0.043354034423828125
P[16] = 0.08124446868896484
P[17] = 0.12318754196166992
P[18] = 0.1544179916381836
P[19] = 0.16265392303466797
P[20] = 0.14574849605560303
P[21] = 0.11203193664550781
P[22] = 0.07427501678466797
P[23] = 0.0425715446472168
P[24] = 0.02109050750732422
P[25] = 0.008999824523925781
P[26] = 0.003286123275756836
P[27] = 0.00101470947265625
P[28] = 0.00026035308837890625
P[29] = 5.3882598876953125e-05
P[30] = 8.58306884765625e-06
P[31] = 9.5367431640625e-07
P[32] = 5.960464477539063e-08

Theoretical Expected sum of the upward face value : 19.0
```

The theortical value of Expected sum of upward face value is 19 which is close to what we have got in the python simualtion that is 18.971

[5]:
```python
# rolling a 16 faced die 4 times

import random
import matplotlib.pyplot as plt
import numpy as np

# Define the number of simulations and the number of dice rolls per simulation
num_simulations = 1000
num_rolls = 4
k = 16

# Initialize an empty list to store the sums of the upward face values
sums = []

# Create a dictionary to map face values to their weights
weights_dict = {i: 1 / (2 ** (i - 1)) for i in range(2, k + 1)}
weights_dict[1] = 1 / (2 ** (k - 1))

# Simulate rolling the biased die and calculating the sum
for _ in range(num_simulations):
    roll_sum = 0
    for _ in range(num_rolls):
```

```python
        # Generate a random face value based on the corrected biased␣
↪probabilities
        face_value = random.choices(list(weights_dict.keys()),␣
↪list(weights_dict.values()))[0]
        roll_sum += face_value
    sums.append(roll_sum)

# Plot a histogram of the sums
plt.hist(sums, bins=range(num_rolls, k * num_rolls + 1), edgecolor='k',␣
↪align='left')
plt.title("Frequency Distribution of Sum")
plt.xlabel("Sum of Face Values")
plt.ylabel("Frequency")
plt.show()

# Calculate the five-number summary
q1, median, q3 = np.percentile(sums, [25, 50, 75])
min_value, max_value = min(sums), max(sums)

# Calculate the mean of the sums list
mean_value = sum(sums) / len(sums)

print("Five-Number Summary:")
print(f"Minimum: {min_value}")
print(f"Q1 (25th percentile): {q1}")
print(f"Median (50th percentile): {median}")
print(f"Q3 (75th percentile): {q3}")
print(f"Maximum: {max_value}")
print(f"Mean: {mean_value}")
```




```
Five-Number Summary:
Minimum: 8
Q1 (25th percentile): 10.0
Median (50th percentile): 11.0
Q3 (75th percentile): 13.0
Maximum: 27
Mean: 11.884
```

2.2 Implementation of Naive Bayes (From Scratch)

[6]:
```
#importing the data set

from ucimlrepo import fetch_ucirepo
import pandas as pd
# fetch dataset
spambase = fetch_ucirepo(id=94)

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

```python
# saving pandas dataframe
Xpd=X
ypd=y

# conversion into numpy arrays
X = X.values
y = y.values

y = y.flatten()
```

[7]:
```python
# printing the features and target values
print("Features:\n",X)
print("Target:\n",y)
```

```
Features:
 [[0.000e+00 6.400e-01 6.400e-01 … 3.756e+00 6.100e+01 2.780e+02]
 [2.100e-01 2.800e-01 5.000e-01 … 5.114e+00 1.010e+02 1.028e+03]
 [6.000e-02 0.000e+00 7.100e-01 … 9.821e+00 4.850e+02 2.259e+03]
 …
 [3.000e-01 0.000e+00 3.000e-01 … 1.404e+00 6.000e+00 1.180e+02]
 [9.600e-01 0.000e+00 0.000e+00 … 1.147e+00 5.000e+00 7.800e+01]
 [0.000e+00 0.000e+00 6.500e-01 … 1.250e+00 5.000e+00 4.000e+01]]
Target:
 [1 1 1 … 0 0 0]
```

[8]:
```python
# spliting the data
from sklearn.model_selection import train_test_split

# Split the data into training (70%), validation (15%), and testing (15%) sets
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3,␣
↪random_state=42)
X_valid, X_test, y_valid, y_test = train_test_split(X_temp, y_temp, test_size=0.
↪5, random_state=42)

print("Shape of X_train:", X_train.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of X_valid:", X_valid.shape)
print("Shape of y_valid:", y_valid.shape)
print("Shape of X_test:", X_test.shape)
print("Shape of y_test:", y_test.shape)
```

```
Shape of X_train: (3220, 57)
Shape of y_train: (3220,)
Shape of X_valid: (690, 57)
Shape of y_valid: (690,)
Shape of X_test: (691, 57)
```

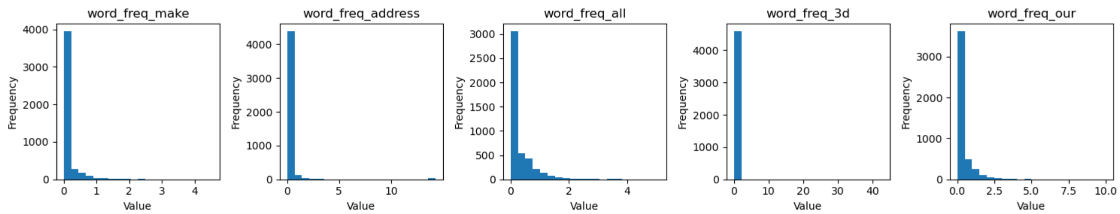
```
Shape of y_test: (691,)
```

[9]:
```
# selecting 5 columns

# Select 5 columns by specifying their indices (0 to 4)
selected_columns = Xpd.iloc[:, [0, 1, 2, 3, 4]]

# Print the selected columns
print(selected_columns)
```

```
      word_freq_make  word_freq_address  word_freq_all  word_freq_3d  \
0               0.00               0.64           0.64           0.0
1               0.21               0.28           0.50           0.0
2               0.06               0.00           0.71           0.0
3               0.00               0.00           0.00           0.0
4               0.00               0.00           0.00           0.0
…                …                 …            …            …
4596            0.31               0.00           0.62           0.0
4597            0.00               0.00           0.00           0.0
4598            0.30               0.00           0.30           0.0
4599            0.96               0.00           0.00           0.0
4600            0.00               0.00           0.65           0.0

      word_freq_our
0              0.32
1              0.14
2              1.23
3              0.63
4              0.63
…             …
4596           0.00
4597           0.00
4598           0.00
4599           0.32
4600           0.00

[4601 rows x 5 columns]
```

[10]:
```
# plotting the probability curves for the 5 columns selected

import matplotlib.pyplot as plt

# Define the number of bins and boundaries for the histograms
num_bins = 20  # You can adjust this value as needed
x_min = selected_columns.min()
x_max = selected_columns.max()
```

```
# Create subplots for each selected column
fig, axes = plt.subplots(1, 5, figsize=(15, 3))

# Plot histograms for each selected column
for i, column in enumerate(selected_columns.columns):
    ax = axes[i]
    ax.hist(selected_columns[column], bins=num_bins, range=(x_min[column],␣
 ↪x_max[column]))
    ax.set_title(column)
    ax.set_xlabel('Value')
    ax.set_ylabel('Frequency')

plt.tight_layout()
plt.show()
```



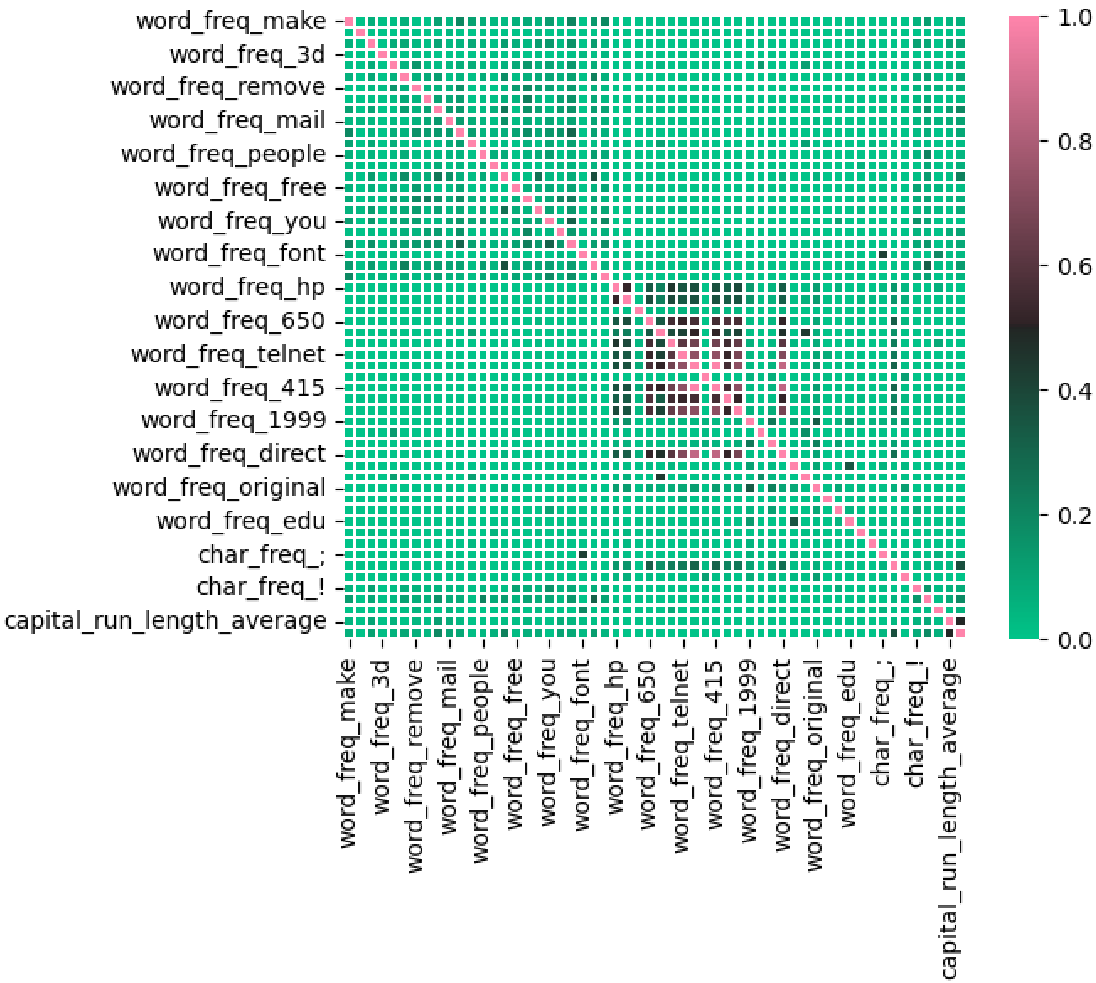
[11]:
```
import pandas as pd

class_priors = ypd.value_counts(normalize=True)

# Print the priors for the classes
print("Class Priors:")
print(class_priors)
```

```
Class Priors:
Class
0        0.605955
1        0.394045
dtype: float64
```

[12]:
```
# to find the correlation between the features.

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# to view the correlation between the features
corr = Xpd.iloc[:,:-1].corr(method="pearson")
```

```python
cmap= sns.diverging_palette(150,354,100,70,center='dark',as_cmap=True)
sns.heatmap(corr,vmax=1,vmin=0,cmap=cmap,square=True,linewidths=.2)
```

[12]: <AxesSubplot:>



- Most of the cells in the correlation matrix are having value as zero.
- Features are independent and the assumption of Naive Bayes is valid here.

[13]:
```python
# Naive bayes implementation from scratch without log transformation
import numpy as np

class NaiveBayes:

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self._classes = np.unique(y)
```

```
        n_classes = len(self._classes)

        # calculate mean, var, and prior for each class
        self._mean = np.zeros((n_classes, n_features), dtype=np.float64)
        self._var = np.zeros((n_classes, n_features), dtype=np.float64)
        self._priors = np.zeros(n_classes, dtype=np.float64)

        for idx, c in enumerate(self._classes):
            X_c = X[y == c]
            self._mean[idx, :] = X_c.mean(axis=0)
            self._var[idx, :] = X_c.var(axis=0)
            self._priors[idx] = X_c.shape[0] / float(n_samples)


    def predict(self, X):
        y_pred = [self._predict(x) for x in X]
        return np.array(y_pred)

    def _predict(self, x):
        posteriors = []

        # calculate posterior probability for each class
        for idx, c in enumerate(self._classes):
            prior = self._priors[idx]
            likelihood = np.prod(self._pdf(idx, x))
            posterior = prior * likelihood
            posteriors.append(posterior)

        # return class with the highest posterior
        return self._classes[np.argmax(posteriors)]

    def _pdf(self, class_idx, x):
        mean = self._mean[class_idx]
        var = self._var[class_idx] + 1e-10
        numerator = np.exp(-((x-mean)**2) / (2 * var))
        denominator = np.sqrt(2 * np.pi * var) + 1e-10
        return (numerator / denominator + 1e-10)
```

[14]:

```
# prediction and reporting the metrics.

from sklearn.metrics import precision_score, recall_score, f1_score

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

```python
nb = NaiveBayes()
nb.fit(X_train, y_train)
predictions = nb.predict(X_test)
predictions_validation = nb.predict(X_valid)

# Calculate accuracy
accuracy_test = accuracy(y_test,predictions)
accuracy_validation = accuracy(y_valid, predictions_validation)

# Calculate precision
precision_test = precision_score(y_test, predictions)
precision_validation = precision_score(y_valid, predictions_validation)

# Calculate recall
recall_test = recall_score(y_test, predictions)
recall_validation = recall_score(y_valid, predictions_validation)

# Calculate F1-score
f1_test = f1_score(y_test, predictions)
f1_validation = f1_score(y_valid, predictions_validation)

print("Without Log Transformation:")
print("Test Set Metrics:")
print("Accuracy:", accuracy_test)
print("Precision:", precision_test)
print("Recall:", recall_test)
print("F1 Score:", f1_test)

print("\nValidation Set Metrics:")
print("Accuracy:", accuracy_validation)
print("Precision:", precision_validation)
print("Recall:", recall_validation)
print("F1 Score:", f1_validation)
```

```
Without Log Transformation:
Test Set Metrics:
Accuracy: 0.6599131693198264
Precision: 0.5493230174081238
Recall: 0.993006993006993
F1 Score: 0.7073474470734745

Validation Set Metrics:
Accuracy: 0.6768115942028986
Precision: 0.5669291338582677
Recall: 0.9896907216494846
F1 Score: 0.7209011264080101
```

[15]:

```python
# Naive Bayes implementation from scratch with log transformation.

import numpy as np

class NaiveBayes_LT:

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self._classes = np.unique(y)
        n_classes = len(self._classes)

        # calculate mean, var, and prior for each class
        self._mean = np.zeros((n_classes, n_features), dtype=np.float64)
        self._var = np.zeros((n_classes, n_features), dtype=np.float64)
        self._priors = np.zeros(n_classes, dtype=np.float64)

        for idx, c in enumerate(self._classes):
            X_c = X[y == c]
            self._mean[idx, :] = X_c.mean(axis=0)
            self._var[idx, :] = X_c.var(axis=0)
            self._priors[idx] = X_c.shape[0] / float(n_samples)


    def predict(self, X):
        y_pred = [self._predict(x) for x in X]
        return np.array(y_pred)

    def _predict(self, x):
        posteriors = []

        for idx, c in enumerate(self._classes):
            prior = np.log(self._priors[idx])
            posterior = np.sum(np.log(self._pdf(idx, x)))
            posterior = posterior + prior
            posteriors.append(posterior)

        # return class with the highest posterior
        return self._classes[np.argmax(posteriors)]

    def _pdf(self, class_idx, x):
        mean = self._mean[class_idx]
        var = self._var[class_idx] + 1e-10
        numerator = np.exp(-((x-mean)**2) / (2 * var))
        denominator = np.sqrt(2 * np.pi * var) + 1e-10
        return (numerator / denominator + 1e-10)
```

[16]:

```python
# prediction and reporting the metrics.

from sklearn.metrics import precision_score, recall_score, f1_score

def accuracy(y_true, y_pred):
    accuracy = np.sum(y_true == y_pred) / len(y_true)
    return accuracy

X_train_log = np.log(X_train + 1e-10)  # Adding a small value to avoid log(0)
X_valid_log = np.log(X_valid + 1e-10)
X_test_log = np.log(X_test + 1e-10)

nblt = NaiveBayes_LT()
nblt.fit(X_train_log, y_train)
predictions = nblt.predict(X_test_log)
predictions_validation = nblt.predict(X_valid_log)

# Calculate accuracy
accuracy_test = accuracy(y_test,predictions)
accuracy_validation = accuracy(y_valid, predictions_validation)

# Calculate precision
precision_test = precision_score(y_test, predictions)
precision_validation = precision_score(y_valid, predictions_validation)

# Calculate recall
recall_test = recall_score(y_test, predictions)
recall_validation = recall_score(y_valid, predictions_validation)

# Calculate F1-score
f1_test = f1_score(y_test, predictions)
f1_validation = f1_score(y_valid, predictions_validation)

print("With Log Transformation:")
print("Test Set Metrics:")
print("Accuracy:", accuracy_test)
print("Precision:", precision_test)
print("Recall:", recall_test)
print("F1 Score:", f1_test)

print("\nValidation Set Metrics:")
print("Accuracy:", accuracy_validation)
print("Precision:", precision_validation)
print("Recall:", recall_validation)
print("F1 Score:", f1_validation)
```

```
With Log Transformation:
```

```
Test Set Metrics:
Accuracy: 0.8075253256150506
Precision: 0.687041564792176
Recall: 0.9825174825174825
F1 Score: 0.8086330935251799

Validation Set Metrics:
Accuracy: 0.8028985507246377
Precision: 0.6832151300236406
Recall: 0.993127147766323
F1 Score: 0.8095238095238095
```

2.2.1 Discussion due to log transformation.

Without Log Transformation:

- The model exhibits moderate accuracy on both the test and validation sets (around 66-68%).
- Precision and recall scores are balanced, indicating a reasonable trade-off between correctly classifying spam emails and minimizing false positives.
- The F1 scores are approximately 0.70, reflecting the harmonic mean of precision and recall.

With Log Transformation:

- Log transformation of the data leads to a significant boost in accuracy, with values around 80% for both test and validation sets.
- Precision is notably improved, indicating a higher rate of correct spam email predictions.
- The model maintains a high recall, effectively capturing most of the actual spam emails.
- The F1 scores increase to around 0.81, suggesting a more balanced performance between precision and recall.

In essence, applying log transformation enhances the Naive Bayes model's performance, making it more effective in classifying spam emails by improving accuracy, precision, and F1 scores.

2.3 Implementation of Naive Bayes (sklearn)

[17]:
```python
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score

# Without log transformation
gnb = GaussianNB()
gnb.fit(X_train, y_train)
y_valid_pred = gnb.predict(X_valid)
y_test_pred = gnb.predict(X_test)

# Calculate accuracy
accuracy_test = accuracy_score(y_test, y_test_pred)
accuracy_validation = accuracy_score(y_valid, y_valid_pred)
```

```python
# Calculate precision
precision_test = precision_score(y_test, y_test_pred)
precision_validation = precision_score(y_valid, y_valid_pred)

# Calculate recall
recall_test = recall_score(y_test, y_test_pred)
recall_validation = recall_score(y_valid, y_valid_pred)

# Calculate F1-score
f1_test = f1_score(y_test, y_test_pred)
f1_validation = f1_score(y_valid, y_valid_pred)

print("Without Log Transformation:")
print("Test Set Metrics:")
print("Accuracy:", accuracy_test)
print("Precision:", precision_test)
print("Recall:", recall_test)
print("F1 Score:", f1_test)

print("\nValidation Set Metrics:")
print("Accuracy:", accuracy_validation)
print("Precision:", precision_validation)
print("Recall:", recall_validation)
print("F1 Score:", f1_validation)
```

```
Without Log Transformation:
Test Set Metrics:
Accuracy: 0.8277858176555717
Precision: 0.7191601049868767
Recall: 0.958041958041958
F1 Score: 0.8215892053973014

Validation Set Metrics:
Accuracy: 0.8217391304347826
Precision: 0.7222222222222222
Recall: 0.9381443298969072
F1 Score: 0.8161434977578477
```

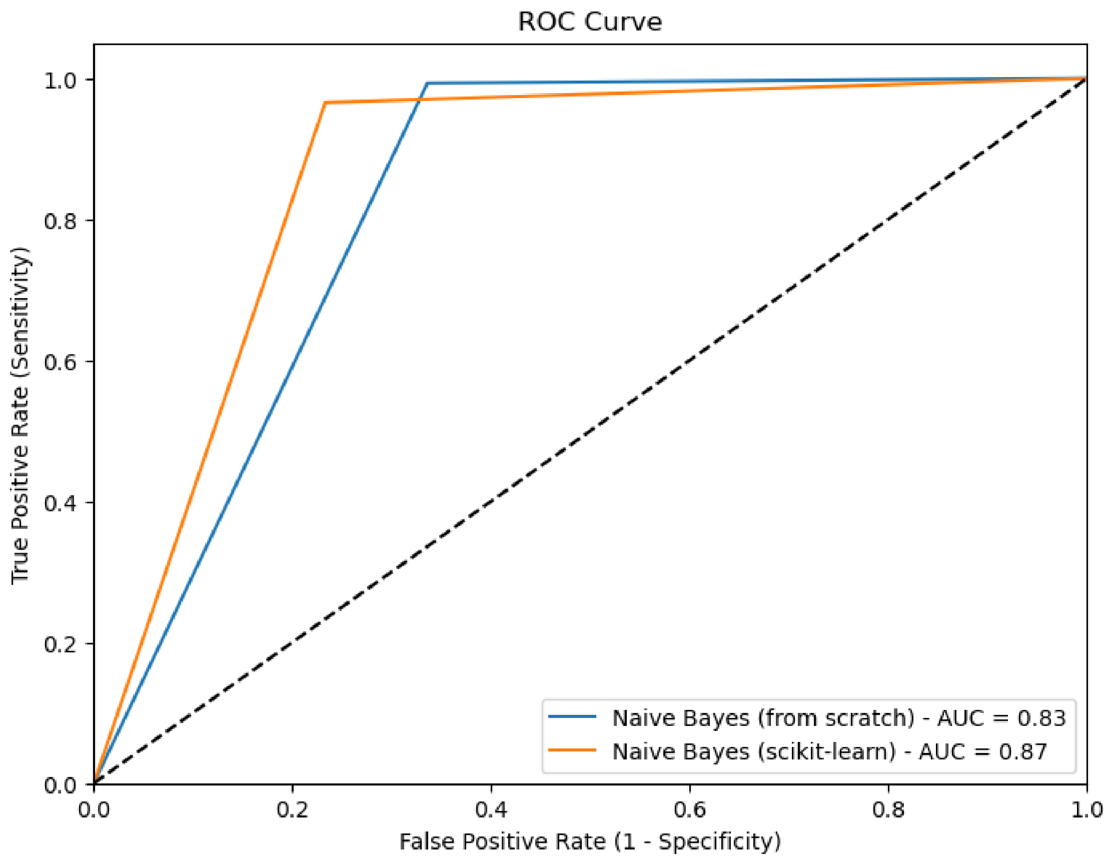
[18]:
```python
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score

# With log transformation
X_train_log = np.log(X_train + 1e-10)  # Adding a small value to avoid log(0)
X_valid_log = np.log(X_valid + 1e-10)
X_test_log = np.log(X_test + 1e-10)

gnb_log = GaussianNB()
```

```python
gnb_log.fit(X_train_log, y_train)
y_valid_pred_log = gnb_log.predict(X_valid_log)
y_test_pred_log = gnb_log.predict(X_test_log)

# Calculate accuracy
accuracy_test = accuracy_score(y_test, y_test_pred_log)
accuracy_validation = accuracy_score(y_valid, y_valid_pred_log)

# Calculate precision
precision_test = precision_score(y_test, y_test_pred_log)
precision_validation = precision_score(y_valid, y_valid_pred_log)

# Calculate recall
recall_test = recall_score(y_test, y_test_pred_log)
recall_validation = recall_score(y_valid, y_valid_pred_log)

# Calculate F1-score
f1_test = f1_score(y_test, y_test_pred_log)
f1_validation = f1_score(y_valid, y_valid_pred_log)

print("With Log Transformation:")
print("Test Set Metrics:")
print("Accuracy:", accuracy_test)
print("Precision:", precision_test)
print("Recall:", recall_test)
print("F1 Score:", f1_test)

print("\nValidation Set Metrics:")
print("Accuracy:", accuracy_validation)
print("Precision:", precision_validation)
print("Recall:", recall_validation)
print("F1 Score:", f1_validation)
```

```
With Log Transformation:
Test Set Metrics:
Accuracy: 0.8480463096960926
Precision: 0.7465940054495913
Recall: 0.958041958041958
F1 Score: 0.8392036753445635

Validation Set Metrics:
Accuracy: 0.8507246376811595
Precision: 0.7513368983957219
Recall: 0.9656357388316151
F1 Score: 0.8451127819548871
```

[19]:

```python
# plotting ROC curves

import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, roc_auc_score

X_train_log = np.log(X_train + 1e-10)  # Adding a small value to avoid log(0)
X_valid_log = np.log(X_valid + 1e-10)
X_test_log = np.log(X_test + 1e-10)

nb_from_scratch = NaiveBayes_LT()
nb_sklearn = GaussianNB()

# Train the models and make predictions on the validation set
nb_from_scratch.fit(X_train_log, y_train)
nb_sklearn.fit(X_train_log, y_train)

y_pred_from_scratch = nb_from_scratch.predict(X_valid_log)
y_pred_sklearn = nb_sklearn.predict(X_valid_log)

# Calculate ROC curve and AUC for both models
fpr_from_scratch, tpr_from_scratch, _ = roc_curve(y_valid, y_pred_from_scratch)
fpr_sklearn, tpr_sklearn, _ = roc_curve(y_valid, y_pred_sklearn)

auc_from_scratch = roc_auc_score(y_valid, y_pred_from_scratch)
auc_sklearn = roc_auc_score(y_valid, y_pred_sklearn)

# Plot ROC curves
plt.figure(figsize=(8, 6))
plt.plot(fpr_from_scratch, tpr_from_scratch, label=f"Naive Bayes (from scratch)␣
 ↪- AUC = {auc_from_scratch:.2f}")
plt.plot(fpr_sklearn, tpr_sklearn, label=f"Naive Bayes (scikit-learn) - AUC =␣
 ↪{auc_sklearn:.2f}")
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate (1 - Specificity)')
plt.ylabel('True Positive Rate (Sensitivity)')
plt.title('ROC Curve')
plt.legend(loc="lower right")
plt.show()
```



The ROC curve is comparing two Naive Bayes models: one built from scratch and one from the scikit-learn library. The Area Under the Curve (AUC) is a measure of the overall performance of the model. The higher the AUC, the better the model is at distinguishing between positive and negative classes.

Based on the AUC values from the image: - The model built from scratch has an AUC of 0.83 - The scikit-learn model has an AUC of 0.87

Therefore, the scikit-learn model appears to be the better model out of the two, as it has a higher AUC value. This suggests that it has a better trade-off between sensitivity and specificity, and is more capable of distinguishing between the positive and negative classes.

2.4 Comparision between Naive Bayes and SVM:

In the comparison between Naive Bayes and Support Vector Machines (SVM) with different kernels and regularization, we can observe the following:

Naive Bayes (with Log Transformation):

- Test Set Accuracy: 0.848
- Validation Set Accuracy: 0.851

The Naive Bayes model with log transformation shows a reasonable level of accuracy on both the test and validation sets. It is particularly strong in terms of recall, indicating that it correctly identifies a significant portion of true positive cases.

SVM (with Different Kernels):

- Linear Kernel Accuracy: 0.925
- Polynomial (Degree 2) Kernel Accuracy: 0.839
- Polynomial (Degree 3) Kernel Accuracy: 0.764
- Sigmoid Kernel Accuracy: 0.889
- RBF Kernel Accuracy: 0.935

The SVM model, particularly with the RBF kernel, demonstrates high accuracy on the test set. However, some other kernel functions like the polynomial (degree 3) kernel have lower accuracy, which might indicate overfitting.

SVM (with Different Regularization):

- Regularization (C) = 0.001: Accuracy = 0.890
- Regularization (C) = 0.1: Accuracy = 0.922
- Regularization (C) = 1: Accuracy = 0.925
- Regularization (C) = 10: Accuracy = 0.923
- Regularization (C) = 100: Accuracy = 0.921

The SVM model's accuracy remains relatively stable across a range of regularization values, with the best performance observed with C = 1.

In summary, the SVM with the RBF kernel achieves the highest accuracy, making it the preferred model among the options. However, it's important to note that the choice of kernel and regularization parameters plays a significant role in the performance of SVM. Naive Bayes, although not as accurate as the best-performing SVM model, still demonstrates reasonable accuracy, especially when considering recall. The choice between these models should take into account the specific goals of the classification task, computational resources, and other trade-offs such as precision and recall.