In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

Dataset Exploration

The dataset we are working with is structured as a CSV file with 5171 entries, each representing an email message. It contains the following columns:

- **unnamed**: an index or identifier for each message.
- **label**: This is a string indicating whether the message is 'spam' or 'ham'
- **text**: The content of the email message
- **label_num**: A numerical representation of the label column, where '0' corresponds to 'ham' and '1' corresponds to 'spam'

  For our machine learning task, the text column will serve as the feature (input), while label_num will be the target (output) we want our model to predict.

In [2]:
```
df = pd.read_csv('../dataset/spam_ham_dataset.csv')
df.head()
```

Out[2]:

	Unnamed: 0	label	text	label_num
0	605	ham	Subject: enron methanol ; meter # : 988291\r\n...	0
1	2349	ham	Subject: hpl nom for january 9 , 2001\r\n(see...	0
2	3624	ham	Subject: neon retreat\r\nho ho ho , we ' re ar...	0
3	4685	spam	Subject: photoshop , windows , office . cheap ...	1
4	2030	ham	Subject: re : indian springs\r\nthis deal is t...	0

In [3]:
```
df.shape
```

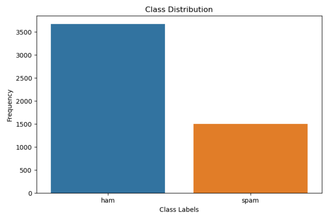
Out[3]:
```
(5171, 4)
```

In this section, we'll take a closer look at our dataset to understand the distribution of classes (spam vs. ham), identify any patterns or anomalies, and prepare the data for the machine learning model. This will involve statistical analysis, visualization, and preprocessing.

We'll start by visualizing the distribution of spam and ham messages in the dataset, and then we'll perform some basic text analysis. Let's begin with the distribution of classes.

In [4]:
```
# class balance count
class_count = df['label'].value_counts()

# bar plot
plt.figure(figsize=(8,5))
sns.barplot(x=class_count.index, y=class_count.values)
plt.title("Class Distribution")
plt.xlabel("Class Labels")
plt.ylabel("Frequency")
plt.show()
```



The bar plot above illustrates the distribution of spam and ham messages within our dataset. From this visualization, we can observe whether there's a significant imbalance between the two classes.

Based on the plot, it looks like there are more ham messages than spam messages, which is typical in real-world scenarios where legitimate messages usually outnumber spam. It's important to consider this imbalance when training our machine learning model, as it may lead to a model that's biased towards predicting the majority class. To address this, we might explore techniques such as resampling the dataset, using different evaluation metrics, or adjusting the class weights in the machine learning algorithm.

Word Frequency Analysis:

In [5]:
```
from collections import Counter

def get_most_common_words(class_label, num_words, column_name, dataset):
    text = " ".join(message for message in dataset[dataset['label'] ==class_label][column_name])
    words = text.lower().split()
    return Counter(words).most_common(num_words)
```

In [6]:
```
get_most_common_words("spam", 20, "text", df)
```

Out[6]:
```
[('.', 19399),
 (',', 11227),
 ('-', 7934),
 ('the', 7254),
 ('/', 5707),
 ('to', 5160),
 ('and', 4903),
 ('a', 4622),
 ('of', 4490),
 ('$', 3787),
 ('in', 3129),
 ('=', 2089),
 ('your', 2794),
 ('for', 2523),
 ('!', 2448),
 ('this', 2283),
 ('is', 2256),
 ('your', 1946),
 (')', 1927),
 ('*', 1762)]
```

For ham messages, the most common items are often punctuation marks and common English words, along with some email-specific terms like 'ect' (possibly a truncated version of 'etcetera' or a specific term) and 'enron'.

For spam messages, again punctuation marks and common English words are frequent, but there are also indicators of spam content such as exclamation marks and sales or urgency-related words like 'free', '!', and '7'.

Text Length Analysis

In [7]:
```
df['text_length'] = df['text'].apply(len)
df.head(1)
```

Out[7]:

	Unnamed: 0	label	text	label_num	text_length
0	605	ham	Subject: enron methanol ; meter # : 988291\r\n...	0	327

In [8]:
```
df.groupby('label')['text_length'].describe()
```

Out[8]:

	count	mean	std	min	25%	50%	75%	max
label								
ham	3672.0	977.008170	1382.827493	18.0	231.75	530.0	1227.25	32258.0
spam	1499.0	1223.256171	1825.986210	11.0	283.00	576.0	1253.50	22073.0

The descriptive statistics for the length of messages in each class show that spam messages tend to be slightly longer on average compared to ham messages. However, both types of messages have a wide range of lengths, as indicated by the standard deviation and the maximum length.

Ham messages have a mean length of 977 characters and a maximum length of 32,258 characters. Spam messages have a mean length of 1223 characters and a maximum length of 22,073 characters.

Cleaning punctuation and most common words

In [9]:
```
import string
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
```

In [10]:
```
#ENGLISH_STOP_WORDS
```

In [11]:
```
string.punctuation
```

Out[11]:
```
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

In [12]:
```
def preprocess_text(text):
    text = text.lower()
    text = text.translate(str.maketrans("","", string.punctuation))
    words = text.split()
    words = [word for word in words if word not in ENGLISH_STOP_WORDS]
    text = " ".join(words)
    return text
```

In [13]:
```
df['cleaned_text'] = df['text'].apply(preprocess_text)
df.head(1)
```

Out[13]:

	Unnamed: 0	label	text	label_num	text_length	cleaned_text
0	605	ham	Subject: enron methanol ; meter # : 988291\r\n...	0	327	subject enron methanol meter 988291 follow not...

K-Nearest Neighbors (KNN)

The K-Nearest Neighbors (KNN) algorithm is a simple, yet effective machine learning algorithm used for classification and regression tasks. In the context of our spam detection problem, we will be using it for classification. KNN works on the principle of feature similarity: a new instance is classified by a majority vote of its neighbors, with the instance being assigned to the class most common among its k nearest neighbors.

For KNN to work with text data, we first need to convert the text into a set of numerical features. This is typically done using techniques like Bag of Words or TF-IDF. We will use the Bag of Words model, which involves the following steps:

1. Tokenization: Splitting text into individual words.
2. Vocabulary Building: Creating a vocabulary of all the unique words in the dataset.
3. Encoding: Transforming each text into a numerical vector based on the vocabulary.

The value of k (the number of neighbors to consider) is a hyperparameter that can be tuned. A small value for k can make the algorithm sensitive to noise in the data, while a large value makes it computationally expensive and may include features that are less relevant.

In [14]:
```
from sklearn.feature_extraction.text import CountVectorizer
```

In [15]:
```
vectorizer = CountVectorizer() #token
X = vectorizer.fit_transform(df['cleaned_text'])
X
```

Out[15]:
```
<5171x50179 sparse matrix of type '<class 'numpy.int64'>'
	with 338374 stored elements in Compressed Sparse Row format>
```

The transformation of the text data into numerical vectors has resulted in a sparse matrix X with 5171 rows, which corresponds to the number of messages, and 50179 columns, each representing a unique word in the vocabulary created from our dataset.

With the text data now in a format suitable for machine learning, we are ready to proceed to the next step where we will train and test the KNN classifier.

In [16]:
```
y = df['label_num']
```

Training and Testing the System

To train and test our KNN model, we'll follow these steps:

- Split the Data: Divide the dataset into a training set and a testing set. This allows us to train the model on one set of data and then test it on a separate set to evaluate its performance.
- Initialize the KNN Classifier: Choose a value for

k and initialize the classifier.

- Train the Classifier: Fit the classifier to the training data.
- Test the Classifier: Use the trained classifier to predict the labels of the testing data.
- Evaluate Performance: Compare the predicted labels to the true labels of the testing set to evaluate the model.

In [17]:
```
from sklearn.model_selection import train_test_split
```

In [18]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
X_train.shape
```

Out[18]:
```
(4136, 50179)
```

In [19]:
```
X_test.shape
```

Out[19]:
```
(1035, 50179)
```

In [20]:
```
from sklearn.neighbors import KNeighborsClassifier
```

In [21]:
```
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
```

Out[21]:
```
KNeighborsClassifier(n_neighbors=3)
```

In [22]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score
```

In [23]:
```
prediction = knn.predict(X_test)
```

In [24]:
```
accuracy_score(y_test, prediction)
```

Out[24]:
```
0.8328502415458937
```

In [25]:
```
precision_score(y_test, prediction)
```

Out[25]:
```
0.6339285714285714
```

In [26]:
```
recall_score(y_test, prediction)
```

Out[26]:
```
0.9692832764505119
```

Creating an OpenAI ChatGPT Version of the System

To create a version of our spam detection system using OpenAI's ChatGPT, we would utilize the OpenAI API to send messages to the model and receive predictions on whether a message is spam or ham. This approach would involve setting up an API call that passes the message text to ChatGPT, which has been fine-tuned on a diverse range of internet text and can perform tasks like text classification when prompted correctly.

There are some caveats to consider:

- OpenAI's models, including ChatGPT, are not specialized for spam detection out of the box and may require fine-tuning on a specific spam detection dataset to achieve optimal performance.
- The API usage comes with associated costs and latency that depend on the number of requests made and the computational resources required for processing.
- The API's performance would depend on the quality and format of the prompts given to the model.

In [27]:
```
#pip install openai
```

In [28]:
```
from openai import OpenAI

# Replace 'your_api_key_here' with your actual OpenAI API key
client = OpenAI(api_key='gpt4_api')

def classify_email(email):

    completion = client.chat.completions.create(
        model="gpt-4o",
        messages=[
            {"role": "developer", "content": "Act as a spam detection algorithm for email classification task. Classify the email into spam or not spam."},
            {
                "role": "user", "content": email
            }
        ]
    )
    #return response.choices[0].email.content.strip()
    print(completion.choices[0].message)
```

In [29]:
```
#classify_email(df['text'].values[10])
```

In [ ]:
```

```