2023

DS 7333 | Quantifying The World

CASE STUDY 3 | SPAM OR NOT SPAM: EMAIL CLASSIFICATION

JOEY HERNANDEZ

Introduction

Background

Spam emails, unwanted and unsolicited messages that flood our inboxes, are a significant global digital concern. The management of email traffic often involves filtering systems, especially for individuals and organizations aiming to maintain efficient communication. One critical aspect of email management is the prevention of spam emails from reaching the primary inbox. Misclassifications can indicate various issues, such as inadequate filtering algorithms, evolving spam tactics post-filtering, or a lack of effective real-time monitoring. The goal in email management is to minimize the recurrence of unwanted events, in this case, spam emails.

Objective and Scope

The primary objective of this report is to develop a predictive model to determine whether an email is spam or not. Achieving this objective can offer several advantages, including improved email management, reduced time spent on irrelevant emails, and enhanced security for email users.

To accomplish our goal, we will employ clustering and naive bayes, widely used techniques for classification tasks. Clustering will help us group similar emails together, making it easier to identify potential spam patterns. On the other hand, naive bayes is well-suited for this task as it allows us to model the probability of an email being spam based on its content and other features. By analyzing these features, we can gain insights into the factors contributing to the likelihood of an email being classified as spam. Additionally, we will utilize techniques to handle any inconsistencies in the data, ensuring that our analysis is robust and representative of the email dataset.

Data Source

This case study will utilize 5 sets of emails, “easy_ham”,”easy_ham_2”,”hard_ham”,”spam”, and ”spam_2”. The data was provided to us in the Case study 3 Module.

Data Ingestion

The process of data ingestion involves obtaining raw email data and transforming it into a structured and usable format for further analysis. Given the diverse nature of emails, which can include various headers, content types, and encodings, it's crucial to have a systematic approach to extract relevant information, and to do so functions were created to handle the extraction of information.

Extracting Content from Emails:

1. **Function Definition**: The function extract_content_from_email is designed to process a raw email, represented as a string, and extract its headers and content.
2. **Email Parsing**: The function utilizes the message_from_string method to convert the raw email string into a message object, which allows for easier extraction of headers and content.
3. **Header Extraction**: The headers of the email are directly extracted into a dictionary format for easy access.
4. **Content Extraction**:
    - If the email is multi-part (i.e., it contains both text and attachments or multiple text parts), the function iterates through each part using the msg.walk() method.
    - For each part, the content type is identified, and the payload (actual content) is extracted.
    - The payload is then decoded using various character encodings. The function attempts to decode using 'utf-8' first. If that fails, it tries 'ascii', and finally, if both fail, it resorts to 'ISO-8859-1'.
    - If the content type indicates that the payload is HTML, the function uses BeautifulSoup to extract the text content from the HTML, ensuring that only the readable text is retained and HTML tags are discarded.
    - All the extracted content parts are then concatenated into a single string.
5. **Single-part Emails**: If the email is not multi-part, the function directly extracts the payload, decodes it using the same logic as above, and processes HTML content if necessary.
6. **Output**: The function returns two main components:
    - The headers of the email in dictionary format.
    - The cleaned and concatenated content of the email as a string.

By using the extract_content_from_email function, we can efficiently process raw email data, ensuring that we have a clean and structured dataset ready for the subsequent stages of our analysis.

Loading emails from folder location:

The next step is to load and preprocess the entire dataset. This involves reading emails from various folders and categorizing them as either spam or ham (non-spam). The provided functions facilitate this process.

1. **Function Definition**: The function load_emails_from_folder is designed to read and preprocess emails from a specified folder. It takes in the path to the folder and a label indicating whether the emails are spam or ham.
2. **Reading Emails**:
    - The function iterates over every file in the specified folder using os.listdir().

    - For each file, it attempts to read the content using different character encodings. The function first tries 'utf-8', then 'ascii', and finally 'ISO-8859-1' if the previous encodings result in a UnicodeDecodeError.
    - Once the email content is read, it is passed to the previously discussed extract_content_from_email function to extract headers and content.
3. **Storing Email Data**:
    - The extracted headers, content, and the provided label (indicating spam or ham) are stored in a dictionary.
    - This dictionary is appended to the emails list, which will contain all the processed emails from the folder.
4. **Loading All Emails**: The load_all_emails function is responsible for loading all emails from various folders.
    - It defines two lists: spam_folders and ham_folders, which contain the names of folders storing spam and ham emails, respectively.
    - For each folder in spam_folders, the function calls load_emails_from_folder with a label of 1 (indicating spam) and appends the results to the spam_emails list.
    - Similarly, for each folder in ham_folders, the function calls load_emails_from_folder with a label of 0 (indicating ham) and appends the results to the ham_emails list.
5. **Output**:
    - The load_all_emails function returns two lists: spam_emails and ham_emails, containing the processed emails categorized as spam and ham, respectively.
    - The final line of the provided code calls the load_all_emails function and stores the results in the spam_emails and ham_emails variables.

By using the load_emails_from_folder and load_all_emails functions, we can efficiently load and preprocess our entire email dataset, ensuring that we have a categorized and structured collection of emails ready for analysis.

Text Preprocessing

Text preprocessing is a crucial step in natural language processing tasks, especially when dealing with raw data like emails. The provided outline below discusses the steps taken to preprocess the email content, making it suitable for further analysis.

1. **Combining Datasets**:
    - The spam_emails and ham_emails lists are concatenated to form a single list called all_emails, which contains all the emails in the dataset.
2. **Checking Data Integrity**:
    - A loop iterates over each email in the all_emails list to ensure that the content of each email is a string. If any email's content is not a string, its index and content are printed. This step is essential to ensure that the subsequent preprocessing steps can be applied uniformly across all emails.

3. **Text Cleaning Function (clean_text)**:
    - **Remove URLs**: The function starts by removing any URLs present in the text using a regular expression.
    - **Convert to Lowercase**: The text is then converted to lowercase to ensure uniformity and to make the analysis case-insensitive.
    - **Remove Numbers and Special Characters**: Another regular expression is used to remove numbers and special characters, retaining only alphabetic characters and spaces.
    - **Tokenization**: The cleaned text is split into individual words or tokens.
    - **Remove Stopwords**: Common words (like "and", "the", "is", etc.) that don't add significant meaning in the context of text classification are removed. The stopwords.words('english') function from the NLTK library provides a list of these common words.
    - **Rejoin Words**: The cleaned and processed words are then joined back into a single string.
4. **Applying Text Cleaning**:
    - The clean_text function is applied to the content of each email in the all_emails list using a list comprehension. The result is a new list, all_emails_cleaned, which contains the preprocessed content of each email.
5. **Extracting Labels**:
    - Another list comprehension is used to extract the label (spam or ham) for each email in the all_emails list. The resulting list, labels, serves as the target variable (y) for subsequent modeling tasks.

In summary, the code process ensures that the email dataset is cleaned and preprocessed, removing any irrelevant information and transforming the content into a format that's more flexible to analysis. The result is a set of cleaned email contents (all_emails_cleaned) and their corresponding labels (labels).

Vectorization

Before diving into clustering or modeling, it's essential to convert the cleaned email content into a numerical format that can be processed by machine learning algorithms. One of the most popular methods for this conversion, especially in text data, is the Term Frequency-Inverse Document Frequency (TF-IDF) vectorization.

1. **What is TF-IDF?**:
    - **Term Frequency (TF)**: This measures the frequency of a word in a document. It's the ratio of the number of times a word appears in a document compared to the total number of words in that document.
    - **Inverse Document Frequency (IDF)**: This measures the importance of a word in the entire corpus. It's calculated as the logarithm of the ratio of the total number of documents to the number of documents containing the word. The idea is to

give lower weights to words that appear frequently across many documents as they might be less informative.

Using TF-IDF vectorization ensures that the features used for clustering or modeling capture the importance and relevance of each word in the context of the entire dataset. Words that are frequent in a particular email but not in the entire corpus will get higher weights, making them more influential in the subsequent analysis.

Clustering With DBSCAN

In this section, we employed the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm to cluster our TF-IDF vectorized email data. DBSCAN is a density-based clustering algorithm that groups together points that are close to each other based on a distance measurement (usually Euclidean distance) and a minimum number of points. It also marks as outliers the points that are in low-density regions.

1. **Parameter Tuning**:
    - Two primary parameters for DBSCAN are `eps` (the maximum distance between two samples for one to be considered as in the neighborhood of the other) and `min_samples` (the number of samples in a neighborhood for a point to be considered as a core point).
    - We defined a range of values for both `eps` and `min_samples` to search for the best combination. Specifically, we considered 20 values for `eps` between 0.1 and 2, and values for `min_samples` ranging from 3 to 14.
    - For each combination of `eps` and `min_samples`, we applied the DBSCAN clustering and calculated the silhouette score, which measures how similar an object is to its own cluster compared to other clusters. A higher silhouette score indicates better-defined clusters.
2. **Best Parameters**:
    - The combination of `eps` and `min_samples` that yielded the highest silhouette score was considered the best. In this case, the best values were `eps = 0.899` and `min_samples = 3`.
3. **Clustering with Best Parameters**:
    - Using the best parameters, we applied DBSCAN clustering to the entire dataset. The resulting clusters were stored in the `preds` variable.
    - The maximum cluster label in `preds` gives an idea of the total number of clusters formed.
    - The value `-1` in the `preds` indicates noise points or outliers, i.e., points that didn't belong to any cluster.
4. **Results**:
    - The output indicates that a significant number of emails (5,578) were marked as outliers or noise. This might suggest that a large portion of the dataset consists of unique or less common emails that didn't fit well into specific clusters.

- The next largest cluster had 121 emails, followed by clusters of sizes 41, 34, 28, and so on. Smaller clusters with just 3 emails were also formed.

Feature Engineering Using DBSCAN

After clustering the emails using DBSCAN, we can leverage the cluster labels as a new feature to enhance our dataset. This new feature can provide valuable information about the inherent groupings within the data, which can be beneficial for subsequent modeling tasks.

The cluster labels obtained from DBSCAN, stored in the `preds` variable, are converted into a numpy array. This array is then reshaped into a column matrix using `[:, np.newaxis]`. This transformation ensures that the cluster labels can be combined with the original TF-IDF vectors.

By incorporating the cluster labels as a feature, we're integrating a higher-level idea of the data into our dataset. This abstraction, derived from the density-based clustering, can provide models with insights about which emails are similar to each other in terms of content and structure.

Special Note:

When using DBSCAN for clustering, a unique aspect of its output is the assignment of a label of `-1` to data points considered as "noise" or outliers. These are points that don't belong to any specific cluster due to their low density. In our dataset, after incorporating the DBSCAN cluster labels as a feature, we have this `-1` value present.

Naive Bayes classifiers, especially the multinomial and Bernoulli variants, expect feature values to be non-negative. This is because they are designed to work with counts or binary data, respectively. Given that our new feature contains negative values (i.e., `-1` for noise), these variants of the Naive Bayes classifier would not be suitable.

However, the `GaussianNB` variant from the `sklearn` library is designed to handle continuous data and does not have the non-negativity constraint. It assumes that the continuous values associated with each class are distributed according to a Gaussian (normal) distribution. This makes `GaussianNB` an appropriate choice for our dataset, which now contains both the continuous TF-IDF values and the cluster labels, including the `-1` values for noise.

Naïve Bayes – With DBSCAN Feature Included

After the feature engineering phase, where DBSCAN cluster labels were incorporated into our dataset, we proceeded to model our data using the Gaussian Naive Bayes classifier.

Confusion Matrix:

A confusion matrix provides a summary of the prediction results on a classification problem. The number of correct and incorrect predictions is summarized with respect to each class.

In our matrix:

- **True Negative (TN)**: 1296 emails were correctly classified as ham.
- **False Positive (FP)**: 151 ham emails were incorrectly classified as spam.
- **False Negative (FN)**: Only 1 spam email was incorrectly classified as ham.
- **True Positive (TP)**: 450 emails were correctly classified as spam.

The heatmap visualization provides a clear representation of these results, with the darker shades indicating higher values.

Figure 1 : *Naïve Bayes Prediction Matrix*

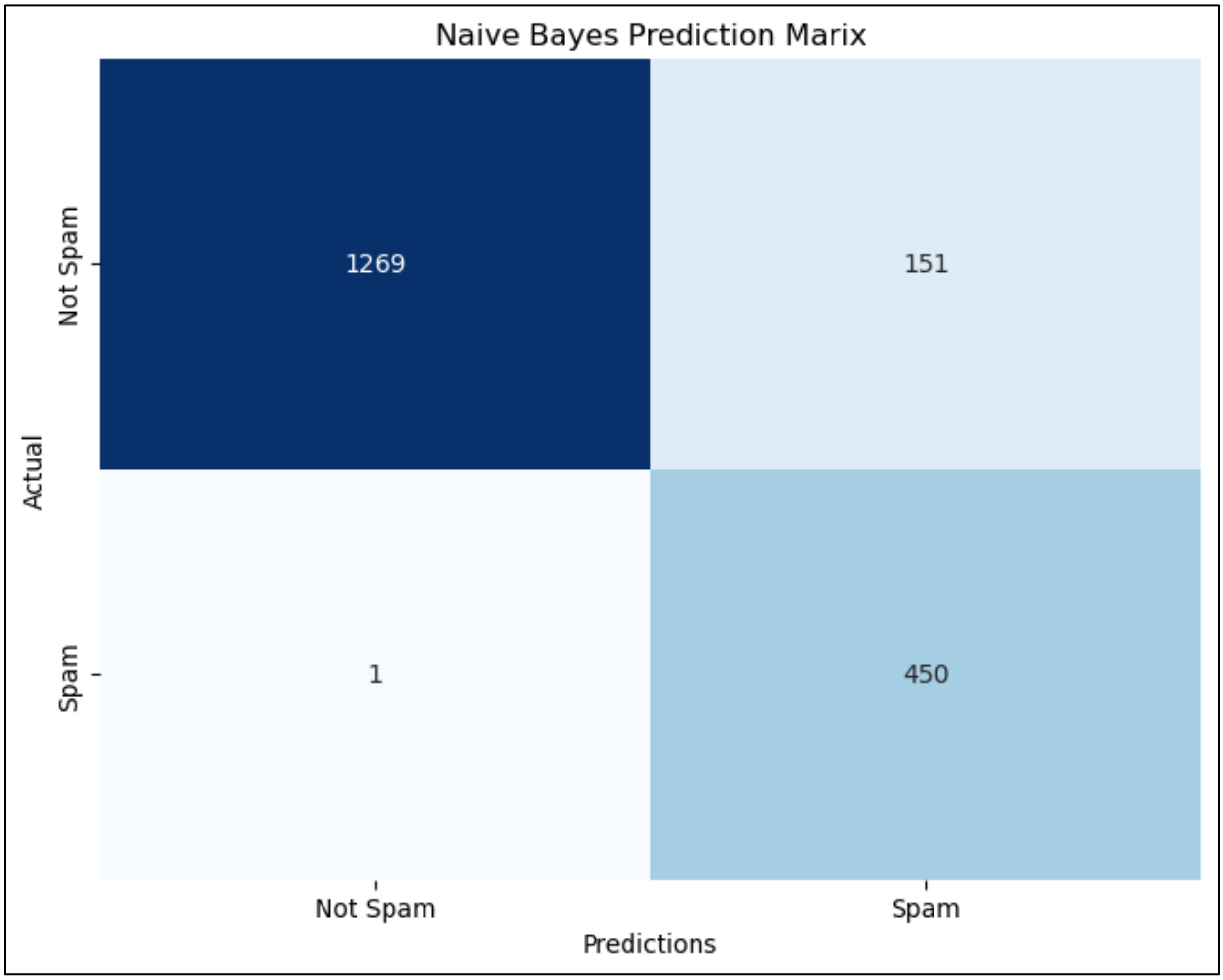


Figure 1 Description*: This heatmap visually represents the performance of the Naïve Bayes classifier in terms of its predictions for the test dataset. This matrix, commonly known as a confusion matrix, provides a clear breakdown of true positive, true negative, false positive, and false negative predictions made by the model.*

Classification Report:
The classification report provides key metrics about the model's performance, including:

- **Precision**: The ratio of correctly predicted positive observations to the total predicted positives. For our ham class, it's perfect at 1.00, while for the spam class, it's 0.75.
- **Recall**: The ratio of correctly predicted positive observations to all the actual positives. The recall for our ham class is 0.89, while it's perfect for the spam class at 1.00.
- **F1-Score**: The weighted average of Precision and Recall. It provides a balance between the two metrics, especially when the class distribution is uneven.

- **Support**: The number of actual occurrences of the class in the dataset.

The report indicates that the model performs exceptionally well for the spam class, with a high recall, ensuring that almost all spam emails are identified.

Table 1 : *Naïve Bayes Classification Report*

Classification Report – Naïve Bayes				
	Precision	**Recall**	**F1- Score**	**Support**
Not Readmitted	1	0.89	0.94	1420
Readmitted	0.75	1.00	0.86	451
Accuracy			0.92	1871
Macro Average	0.87	0.95	0.90	1871
Weighted Average	0.94	0.92	0.92	1871

Table 1 Description: *classification report for the Naïve Bayes model, which was trained and evaluated on a dataset enriched with a feature engineered from the DBSCAN clustering predictions. This feature, derived from the inherent groupings discovered by DBSCAN, was integrated into the dataset to provide additional context and information for the classifier.*

Conclusion:

- The Gaussian Naive Bayes classifier, when combined with the features derived from DBSCAN clustering, demonstrates strong performance in distinguishing between spam and ham emails.
- The high recall for the spam class is particularly encouraging, as it suggests that the model is effective in catching most of the spam emails, a crucial requirement for a spam filter.

Clustering With KMEANS

In this section, KMeans was employed to segment our TF-IDF vectorized email data. The KMeans algorithm aims to partition the dataset into distinct, non-overlapping groups or clusters. The process and results of this clustering approach are detailed below:

1. **Determining Optimal Number of Clusters**:
    - One of the challenges with KMeans is determining the optimal number of clusters. To address this, we evaluated the silhouette score for different numbers of clusters.

    - The silhouette score measures how similar an object is to its own cluster compared to other clusters. A higher silhouette score indicates better-defined clusters.
    - We iteratively applied KMeans clustering for a range of cluster numbers, from 2 up to 100. For each iteration, the silhouette score was computed and stored in the sil_scores list.
2. **Visualizing Silhouette Scores**:
    - A plot was generated to visualize the silhouette scores against the number of clusters. This plot aids in identifying the optimal number of clusters by pinpointing where the silhouette score is maximized.
    - From the plot, we can infer the number of clusters that provides the best trade-off between maximizing the silhouette score and minimizing the complexity (number of clusters).
3. **Applying KMeans with Optimal Clusters**:
    - Based on the silhouette score plot, we chose 60 as the optimal number of clusters. KMeans was then applied to the entire dataset using this number.
    - The resulting clusters were stored in the preds variable.
4. **Feature Engineering Using KMeans Clusters**:
    - Similar to the approach with DBSCAN, the cluster labels obtained from KMeans were used to create a new feature.
    - The cluster labels in preds were converted into a column matrix, preds_matrix.
    - The hstack function was then used to horizontally stack the original TF-IDF vectors (new_vectors) and the preds_matrix. The result, combined_feat, is an enriched dataset that contains both the original TF-IDF vectors and the KMeans cluster labels.
5. **Significance of the New Feature**:
    - Incorporating the KMeans cluster labels as a feature embeds the structural information discovered by KMeans into the dataset. This can be particularly useful for models to discern patterns or relationships that might be associated with specific clusters.

Naïve Bayes – With KMeans Feature Included

After the feature engineering phase, where KMeans cluster labels were incorporated into our dataset, we proceeded to model our data using the Multinomial Naive Bayes classifier. This section provides an overview of the modeling process and its results.

1. **Data Splitting**:
    - The dataset (combined_feat), which includes the original TF-IDF vectors and the KMeans cluster labels, was split into training and test sets. This ensures that we have separate data for training our model and evaluating its performance.
2. **Model Initialization and Training**:

- We utilized the MultinomialNB classifier from the sklearn library. The Multinomial Naive Bayes classifier is suitable for classification with discrete features, such as word counts or the presence/absence of words.
- The model was trained using the training dataset (X_train) and the corresponding labels (y_train).

3. **Model Predictions and Accuracy**:
   - Predictions were made on the test dataset (X_test), and the accuracy of these predictions was computed against the actual labels (y_test).
   - The achieved accuracy was approximately 82.47%, indicating that the model correctly classified this percentage of emails as either spam or not spam in the test set.
4. **Confusion Matrix Visualization**:
   - A confusion matrix provides a summary of the prediction results on a classification problem. The number of correct and incorrect predictions is summarized with respect to each class.
   - In our matrix:
     - **True Negative (TN)**: 1416 emails were correctly classified as ham.
     - **False Negative (FN)**: 324 spam emails were incorrectly classified as ham.
     - **True Positive (TP)**: 127 emails were correctly classified as spam.
     - **False Positive (FP)**: 4 ham emails were incorrectly classified as spam.
   - The heatmap visualization provides a clear representation of these results, with the darker shades indicating higher values.

Figure 2 : *Naïve Bayes Prediction Matrix*

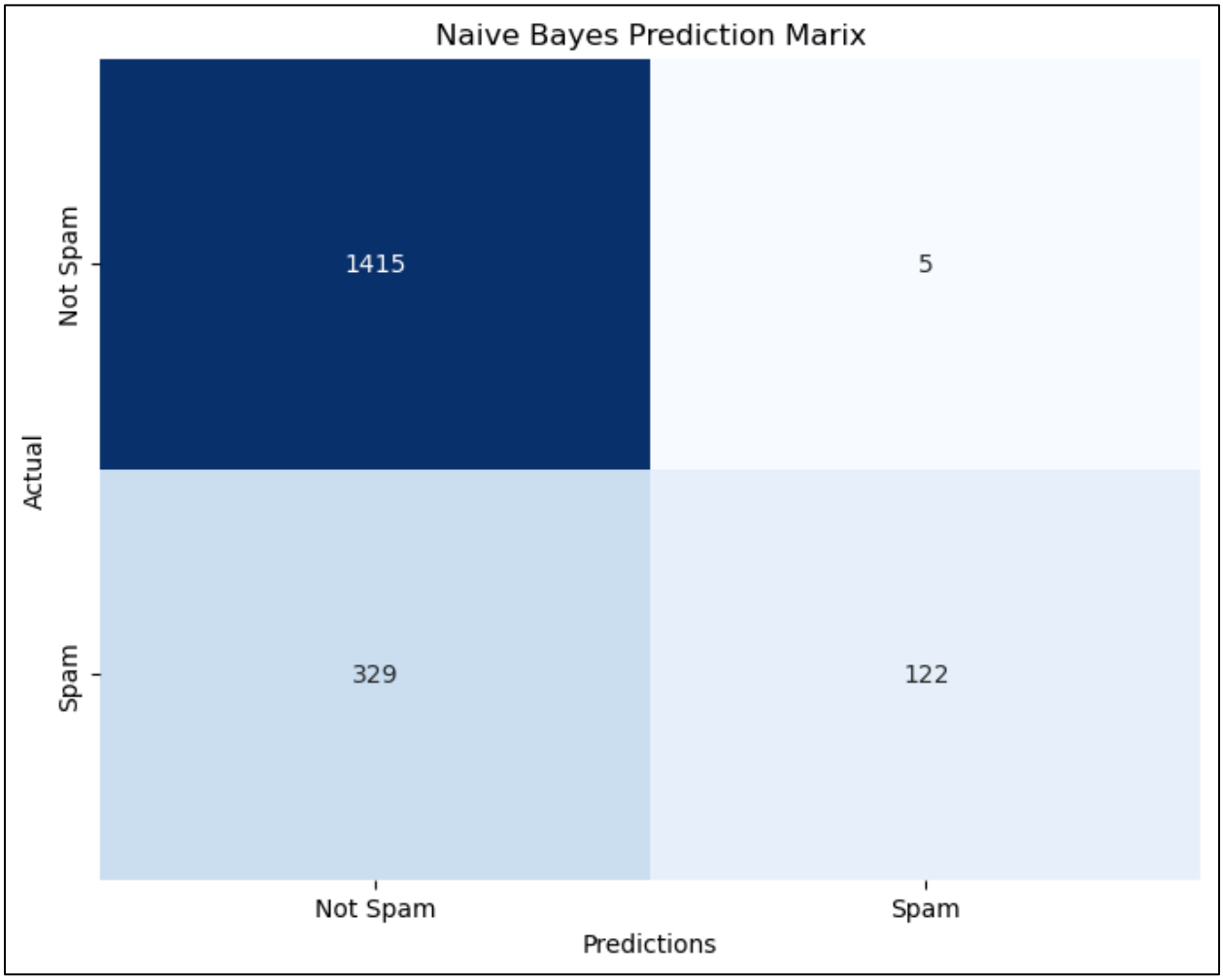


Figure 2 Description*: This heatmap visually represents the performance of the Naïve Bayes classifier in terms of its predictions for the test dataset. This matrix, commonly known as a confusion matrix, provides a clear breakdown of true positive, true negative, false positive, and false negative predictions made by the model.*

5. **Classification Report**:

   The classification report provides key metrics about the model's performance, including:

   - **Precision**: The ratio of correctly predicted positive observations to the total predicted positives. For our ham class, it's 0.81, while for the spam class, it's an impressive 0.96.
   - **Recall**: The ratio of correctly predicted positive observations to all the actual positives. The recall for our ham class is perfect at 1.00, but it's lower for the spam class at 0.27.

- **F1-Score**: The weighted average of Precision and Recall. It provides a balance between the two metrics, especially when the class distribution is uneven.
- **Support**: The number of actual occurrences of the class in the dataset.

The report indicates that while the model has a high precision for the spam class, its recall is low. This suggests that while the model is confident in its spam predictions, it's missing a significant number of actual spam emails.

Table 2 : *Naïve Bayes Classification Report*

Classification Report – Naïve Bayes				
	Precision	**Recall**	**F1- Score**	**Support**
Not Readmitted	0.81	1.00	0.87	1420
Readmitted	0.96	0.27	0.42	451
Accuracy			0.82	1871
Macro Average	0.89	0.95	0.66	1871
Weighted Average	0.85	0.82	0.78	1871

Table 2 Description: *classification report for the Naïve Bayes model, which was trained and evaluated on a dataset augmented with a feature derived from the KMeans clustering predictions. This engineered feature, originating from the distinct clusters identified by KMeans, was incorporated into the dataset to provide the classifier with added structural insights and context.*

6. **Conclusion**:
   - The Multinomial Naive Bayes classifier, when combined with the features derived from KMeans clustering, demonstrates a good performance in distinguishing between spam and ham emails.
   - However, the low recall for the spam class suggests there's room for improvement. It's crucial for a spam filter to identify as many spam emails as possible, so addressing this low recall should be a priority in future iterations.

In summary, the integration of KMeans-derived features and the Multinomial Naive Bayes classifier has proven to be an effective combination for email classification. The insights from the confusion matrix and classification report further emphasize the model's strengths and areas for improvement.

Conclusion

In this report, we developed a predictive model that could discern between spam and non-spam emails. The overarching objective was clear: to enhance email management, bolster security, and reduce the time individuals spend sifting through irrelevant emails. To achieve this, we employed a combination of clustering (both DBSCAN and KMeans) and the Naive Bayes classification technique.

Our approach was systematic. We began with data ingestion, ensuring that raw email data was transformed into a structured format suitable for analysis. Given the multifaceted nature of emails, our extraction function played a pivotal role in ensuring data integrity. Subsequent preprocessing steps further refined our dataset, stripping it of noise and irrelevant information.

The main portion of the analysis lay in the combination of clustering and classification. Clustering, through DBSCAN and KMeans, allowed us to discern inherent groupings within the emails, potentially highlighting patterns characteristic of spam messages. These cluster labels, when used as features, enriched our dataset, providing our Naive Bayes classifier with additional context.

The results were promising. Our confusion matrices and classification reports for both clustering methods provided a granular view of the model's performance. While the model showcased high precision in most cases, indicating a high trustworthiness in its positive predictions, certain challenges, like the recall for the spam class in the KMeans approach, highlighted areas for improvement.

Here's a comparative analysis of the results achieved:

1. **DBSCAN with Gaussian Naive Bayes**:
    - **Accuracy**: The model achieved an impressive accuracy of approximately 92%, indicating a high rate of correct predictions.
    - **Precision and Recall**: The model exhibited perfect precision for the non-spam class and a commendable precision of 0.75 for the spam class. The recall for the spam class was perfect, suggesting the model's proficiency at capturing most spam emails.
    - **F1-Score**: The balanced F1-Scores, especially a score of 0.86 for the spam class, highlight the model's balanced performance between precision and recall.
2. **KMeans with Multinomial Naive Bayes**:
    - **Accuracy**: This approach yielded an accuracy of approximately 82.47%, slightly lower than the DBSCAN method.
    - **Precision and Recall**: While the precision for the spam class was an impressive 0.96, the recall was notably lower at 0.27. This indicates that while the model's spam predictions were trustworthy, it missed a significant number of actual spam emails.
    - **F1-Score**: The F1-Score for the spam class was 0.42, reflecting the imbalance between precision and recall for this class.

The DBSCAN approach, when combined with Gaussian Naive Bayes, provided a more balanced performance between precision and recall, leading to a higher overall accuracy.

The KMeans approach, despite its lower recall for the spam class, showcased a notably high precision, indicating its potential if recall issues are addressed.

In essence, our work underscores the potential of combining traditional classification techniques with clustering insights. While our model achieved commendable accuracy, the journey of refining and perfecting it is ongoing. As spam tactics evolve, so must our strategies. The insights gained from this report lay a robust foundation for future iterations and refinements, bringing us one step closer to a more secure and efficient digital communication landscape.

Recommendations

Enriched Feature Engineering:

- **N-grams**: Instead of just using individual words (unigrams) in the TF-IDF vectorization, consider using bigrams or trigrams to capture more contextual information.
- **Email Metadata**: Explore features like email length, sender domain reputation, and time sent, which might provide additional insights into the nature of the email.

Addressing Class Imbalance:

- If the dataset has a significant class imbalance, techniques like SMOTE can be employed to balance the classes, potentially improving model performance, especially for the minority class.

By integrating these recommendations in future iterations of the project, there's potential to further enhance the accuracy, robustness, and generalizability of the spam detection system, ensuring a safer and more efficient email experience for users.

Appendix

Case Study 3 - Spam vs. Ham

Import Packages

In [1]:
```python
import os
import pandas as pd
from bs4 import BeautifulSoup
from email import message_from_string
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.model_selection import train_test_split
from nltk.corpus import stopwords
import re
from email.message import Message
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
```

Define Functions For Extraction / Loading

Extracting Content From Email

In [2]:
```python
def extract_content_from_email(email_str):
    '''Extract the header & content from an email given as a string'''

    msg = message_from_string(email_str)

    # extract headers
    headers = dict(msg._headers)

    # If the email is multi-part
    if msg.is_multipart():
        content = []
        for part in msg.walk():
            content_type = part.get_content_type()
            payload = part.get_payload(decode=True)

            if payload is None:
                continue  # Skip and move on

            try:
                text = payload.decode('utf-8')
            except UnicodeDecodeError:
                try:
                    text = payload.decode('ascii')
                except:
                    text = payload.decode('ISO-8859-1')

            # Extract text from HTML
            if 'text/html' in content_type.lower():
                text = BeautifulSoup(text, 'lxml').get_text()

            content.append(text)
        content = ' '.join(content)
    else:
        content_type = headers.get('Content-Type', '')
        payload = msg.get_payload(decode=True)

        if payload is None:
            content = ""  # Empty string if the payload is None
        else:
            try:
                content = payload.decode('utf-8')
            except UnicodeDecodeError:
                try:
                    content = payload.decode('ascii')
                except:
                    content = payload.decode('ISO-8859-1')
            if 'text/html' in content_type.lower():
                content = BeautifulSoup(content, 'lxml').get_text()

    return headers, content
```

Loading Emails From Local Folders

In [3]:
```python
def load_emails_from_folder(folder_path, label):
    '''load and preprocess emails from folder'''
    emails = []
    for filename in os.listdir(folder_path):
        filepath = os.path.join(folder_path, filename)

        try:
            # Try reading with UTF-8 encoding first
            with open(filepath, 'r', encoding='utf-8') as f:
                email_str = f.read()
        except UnicodeDecodeError:
            try:
                # If UnicodeDecodeError, try reading with ASCII
                with open(filepath, 'r', encoding='ascii') as f:
                    email_str = f.read()
            except UnicodeDecodeError:
                # If still a UnicodeDecodeError, try ISO-8859-1
                with open(filepath, 'r', encoding='ISO-8859-1') as f:
                    email_str = f.read()

        headers, content = extract_content_from_email(email_str)
        emails.append({
            'headers': headers,
            'content': content,
            'label': label
        })
    return emails

def load_all_emails():
    spam_folders = ['spam', 'spam_2']
    ham_folders = ['easy_ham', 'easy_ham_2', 'hard_ham']

    spam_emails = []
    ham_emails = []

    # Load spam emails from all folders
    for folder in spam_folders:
        spam_emails.extend(load_emails_from_folder(folder, 1))

    # Load ham emails from all folders
    for folder in ham_folders:
        ham_emails.extend(load_emails_from_folder(folder, 0))

    return spam_emails, ham_emails

spam_emails, ham_emails = load_all_emails()
```

Spam Email and Ham Email now are two lists which contain all of the emails from the folders in the function load all emails.

Preprocessing Text

In [4]:
```python
all_emails = spam_emails + ham_emails
```

In [5]:
```python
for i, email in enumerate(all_emails):
    count = i
    if not isinstance(email.get('content'), str):
        print(f"Index {i}, place {count} has non-string content: {email.get('content')}")
```

In [6]:
```python
# Text Cleaning
def clean_text(text):
    # Remove URLs
    text = re.sub(r'http\S+', '', text)
    # Convert to lowercase
    text = text.lower()
    # Remove numbers, special characters
    text = re.sub(r'[^a-zA-Z\s]', '', text)
    # Tokenize
    words = text.split()
    # Remove stopwords
    words = [word for word in words if word not in stopwords.words('english')]
    return ' '.join(words)
```

In [7]:
```python
all_emails_cleaned = [clean_text(email['content']) for email in all_emails] # our x
labels = [email['label'] for email in all_emails] # our y(target)
```

Vectorization

TfidfVectorizer

- "Term Frequency - Inverse Document Frequency"
- Vectorization technique commonly used in NLP and text analysis.
- Converts documents into a matrix of TF-IDF values. Each row represents a document, and each column represents a unique word (or n-gram) in the entire corpus.
- Values are computed using a formula that combines term frequency(TF) with inverse document frequency(IDF). Thus, not only does it consider how often a word appears in a document (TF) but also how unique or important the wrod is accross all documents in the corpus(IDF). Common words like "The" are given lower weights, and rarer, more informative words are given higher weights.
- TfidfVectorizer normalizes the TF-IDF values to ensure that longer documents don't have an inherent advantage in terms of TF-IDF scores.

In [8]:
```python
tf_vectors = TfidfVectorizer()
new_vectors = tf_vectors.fit_transform(all_emails_cleaned)
```

In [9]:
```python
new_vectors.shape
```

Out[9]:
```
(9353, 85850)
```

DBSCAN Clustering

In the chunk below we will iterate through to find the best eps and min samples which yield the best silhouette score.

- The Silhouette score is a metric used to calculate the goodness of a clustering algorithm. it's value ranges from -1 to 1 where a higher value indicates the object is matched well to it's own cluster and poorly matched to neighboring clusters. If most objects have high values then the clustering configuration is appropriate. if too many are low then the clustering config may have too many or too few clusters.

Do Not Run The Commented Out Chunk

In [10]:
```python
# from sklearn.cluster import DBSCAN
# from sklearn.metrics import silhouette_score
# import numpy as np

# eps_values = np.linspace(.1,2,20)
# min_samples_values = list(range(3,15))

# best_score = -1
# best_eps = None
# best_min_samples = None

# for eps in eps_values:
#     for min_samples in min_samples_values:
#         db = DBSCAN(eps=eps, min_samples=min_samples)
#         preds = db.fit_predict(new_vectors)

#         n_clusters = len(set(preds)) -(1 if -1 in preds else 0)
#         if 1 < n_clusters < new_vectors.shape[0]:
#             score = silhouette_score(new_vectors, preds)
#             if score > best_score:
#                 best_eps, best_min_samples, best_score = eps, min_samples, score

# print(f'Best eps: {best_eps}')
# print(f'Best min_samples: {best_min_samples}')
# print(f'Best Silhouette Score: {best_score}')
```

Best eps: 0.8999999999999999

Best min_samples: 3

Best Silhouette Score: 0.05852089417844118

In [11]:
```python
from sklearn.cluster import DBSCAN
db = DBSCAN(eps = .899, min_samples = 3)
preds = db.fit_predict(new_vectors)
max(preds)
```

Out[11]:
```
715
```

In [12]:
```python
import numpy as np
from scipy.sparse import hstack
preds_matrix = np.array(preds)[:, np.newaxis]

combined_feat = hstack([new_vectors, preds_matrix])
```

Checking cluster counts

In [13]:
```python
DB_df = pd.DataFrame(preds)
DB_df[0].value_counts()
```

Out[13]:
```
-1      5578
 242     121
 15       41
 586      34
 0        28
        ...
 150       3
 51        3
 50        3
 250       3
 715       3
Name: 0, Length: 717, dtype: int64
```

DBSCAN Models

K-Nearest Neighbors

In [14]:
```python
X_train, X_test, y_train, y_test = train_test_split(combined_feat, labels,
                                                    test_size=.2,
                                                    random_state=12,
                                                    shuffle=True)
```

In [15]:
```python
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
```

Out[15]:
```
▾ KNeighborsClassifier
KNeighborsClassifier()
```

In [16]:
```python
predictions = knn.predict(X_test)
```

In [17]:
```python
from sklearn.metrics import accuracy_score
accuracy_score(y_test, predictions)
```

Out[17]:
```
0.9732763228220203
```

In [18]:
```python
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=10)
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)
accuracy_score(y_test, predictions)
```

Out[18]:
```
0.9722073757349011
```

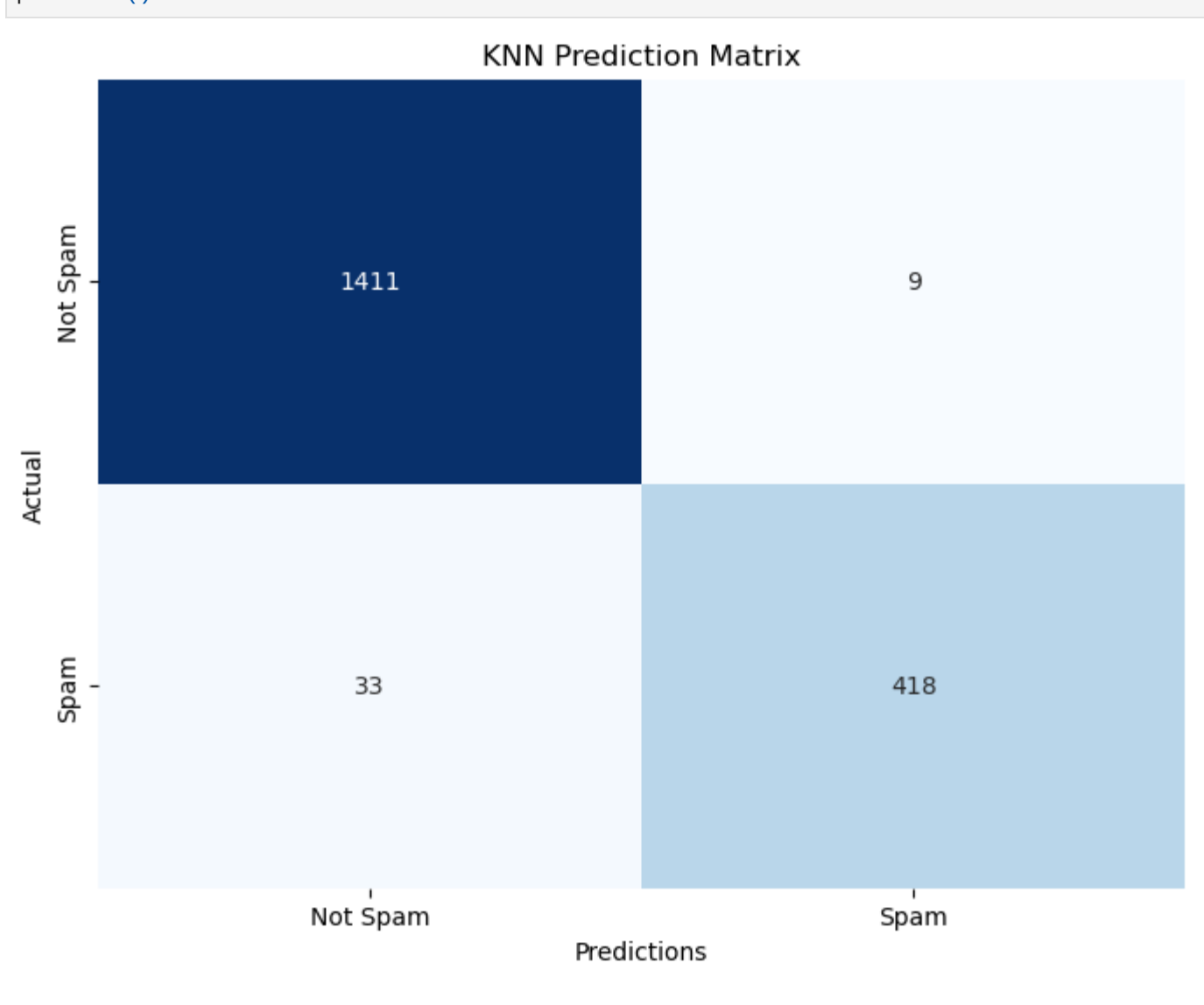
In [19]:
```python
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)
accuracy_score(y_test, predictions)
```

Out[19]:
```
0.9641902725815072
```

In [20]:
```python
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=2)
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)
accuracy_score(y_test, predictions)
```

Out[20]:
```
0.977552111170497
```

In [21]:
```python
## Confusion Matrix
# Calculate confusion matrix
knn_matrix = confusion_matrix(y_test, predictions)

# heatmap(matrix plot)
plt.figure(figsize=(8,6))
sns.heatmap(knn_matrix, annot=True, fmt ='d',cmap = 'Blues',cbar = False,
            xticklabels=['Not Spam', 'Spam'], yticklabels=['Not Spam', 'Spam'])
plt.xlabel('Predictions')
plt.ylabel('Actual')
plt.title('KNN Prediction Matrix')
plt.show()
```



In [22]:
```python
from sklearn.metrics import classification_report
classification_rep = classification_report(y_test, predictions)
print('KNN Classification Report:','\n',classification_rep)
```

```
KNN Classification Report:
               precision    recall  f1-score   support

           0       0.98      0.99      0.99      1420
           1       0.98      0.93      0.95       451

    accuracy                           0.98      1871
   macro avg       0.98      0.96      0.97      1871
weighted avg       0.98      0.98      0.98      1871
```

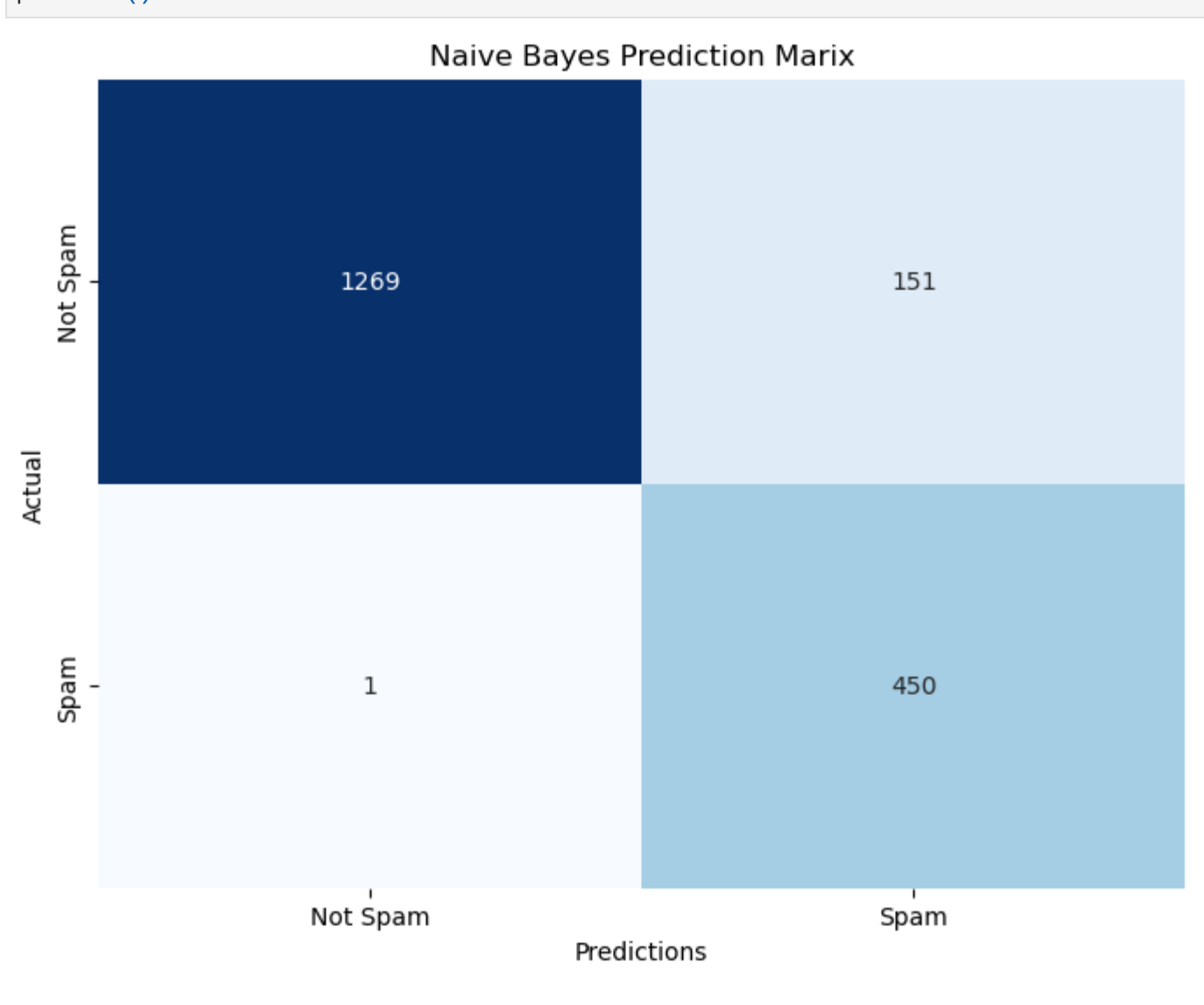
Naive Bayes

In [23]:
```python
from sklearn.naive_bayes import GaussianNB

gnb = GaussianNB()
gnb.fit(X_train.toarray(), y_train)  # Convert sparse matrix to dense if necessary

# Predictions
y_pred = gnb.predict(X_test.toarray())
```

In [24]:
```python
accuracy = accuracy_score(y_test, y_pred)
print('Naive Bayes (Gaussian) Accuracy:', accuracy)
```

```
Naive Bayes (Gaussian) Accuracy: 0.9187600213789417
```

In [25]:
```python
nb_confusion = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(8,6))
sns.heatmap(nb_confusion, annot=True, fmt = 'd', cmap = 'Blues', cbar = False,
            xticklabels=['Not Spam', 'Spam'], yticklabels=['Not Spam', 'Spam'])
plt.xlabel('Predictions')
plt.ylabel('Actual')
plt.title('Naive Bayes Prediction Marix')
plt.show()
```



In [26]:
```python
classification_rep = classification_report(y_test, y_pred)
print('Naive Bayes Classification Report:','\n',classification_rep)
```

```
Naive Bayes Classification Report:
               precision    recall  f1-score   support

           0       1.00      0.89      0.94      1420
           1       0.75      1.00      0.86       451

    accuracy                           0.92      1871
   macro avg       0.87      0.95      0.90      1871
weighted avg       0.94      0.92      0.92      1871
```

K-Means

Alt text

Alt text

Alt text

This chunk will take 65 min to run

In [27]:
```python
# from sklearn.cluster import KMeans
# from sklearn.metrics import silhouette_score
# import matplotlib.pyplot as plt

# sil_scores = []
# max_clusters = 100
# for i in range(2, max_clusters+1):  # Silhouette score requires at least 2 clusters
#     kmeans = KMeans(n_clusters=i, tol=.0001, max_iter=300, random_state=12, n_init=10)
#     kmeans_clusters = kmeans.fit_predict(new_vectors)
#     score = silhouette_score(new_vectors, kmeans_clusters)
#     sil_scores.append(score)

# plt.figure(figsize=(10,7))
# plt.plot(range(2, max_clusters+1), sil_scores, marker='o', linestyle='--')
# plt.title('Silhouette Score Plot')
# plt.xlabel('Number of Clusters')
# plt.ylabel('Silhouette Score')
# plt.grid(True)
# plt.show()
```

Chunk below takes 1 hour to run

In [28]:
```python
# from sklearn.cluster import KMeans
# import matplotlib.pyplot as plt

# wcss = []
# max_clusters = 100
# for i in range(1, max_clusters+1):
#     kmeans = KMeans(n_clusters=i, tol = .0001, max_iter=300, random_state=12, n_init=10)
#     kmeans.fit(new_vectors)
#     wcss.append(kmeans.inertia_)

# plt.figure(figsize=(10,7))
# plt.plot(range(1, max_clusters+1), wcss, marker = 'o', linestyle='--')
# plt.title('Elbow Plot')
# plt.xlabel('Number of Clusters')
# plt.ylabel('WCSS')
# plt.grid(True)
# plt.show()
```

In [29]:
```python
kmeans = KMeans(n_clusters=60, tol = .0001, max_iter=300)
preds = kmeans.fit_predict(new_vectors)
```

```
c:\Users\Joey\anaconda3\envs\ML\lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  warnings.warn(
```

In [30]:
```python
preds_matrix = np.array(preds)[:, np.newaxis]
combined_feat = hstack([new_vectors, preds_matrix])
```

Train Test Split

In [31]:
```python
X_train, X_test, y_train, y_test = train_test_split(combined_feat, labels,
                                                    test_size=.2,
                                                    random_state=12,
                                                    shuffle=True)
```

K-Neighbors Classifier

5 Nearest Neighbors

In [32]:
```python
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
```

Out[32]:
```
▾ KNeighborsClassifier
KNeighborsClassifier()
```

In [33]:
```python
predictions = knn.predict(X_test)
```

Old Accuracy Without The Added Clustering Feature: 0.4216996258685195

In [34]:
```python
from sklearn.metrics import accuracy_score
accuracy_score(y_test, predictions)
```

Out[34]:
```
0.8097274184927846
```

10 Nearest Neighbors

Old Accuracy Without The Added Clustering Feature: 0.3201496525921967

In [35]:
```python
knn = KNeighborsClassifier(n_neighbors=10)
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)
accuracy_score(y_test, predictions)
```

Out[35]:
```
0.7851416354890433
```

3 Nearest Neighbors

Old Accuracy Without The Added Clustering Feature: 0.5146980224478889

In [36]:
```python
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)
accuracy_score(y_test, predictions)
```

Out[36]:
```
0.8268305718866916
```

2 Nearest Neighbors

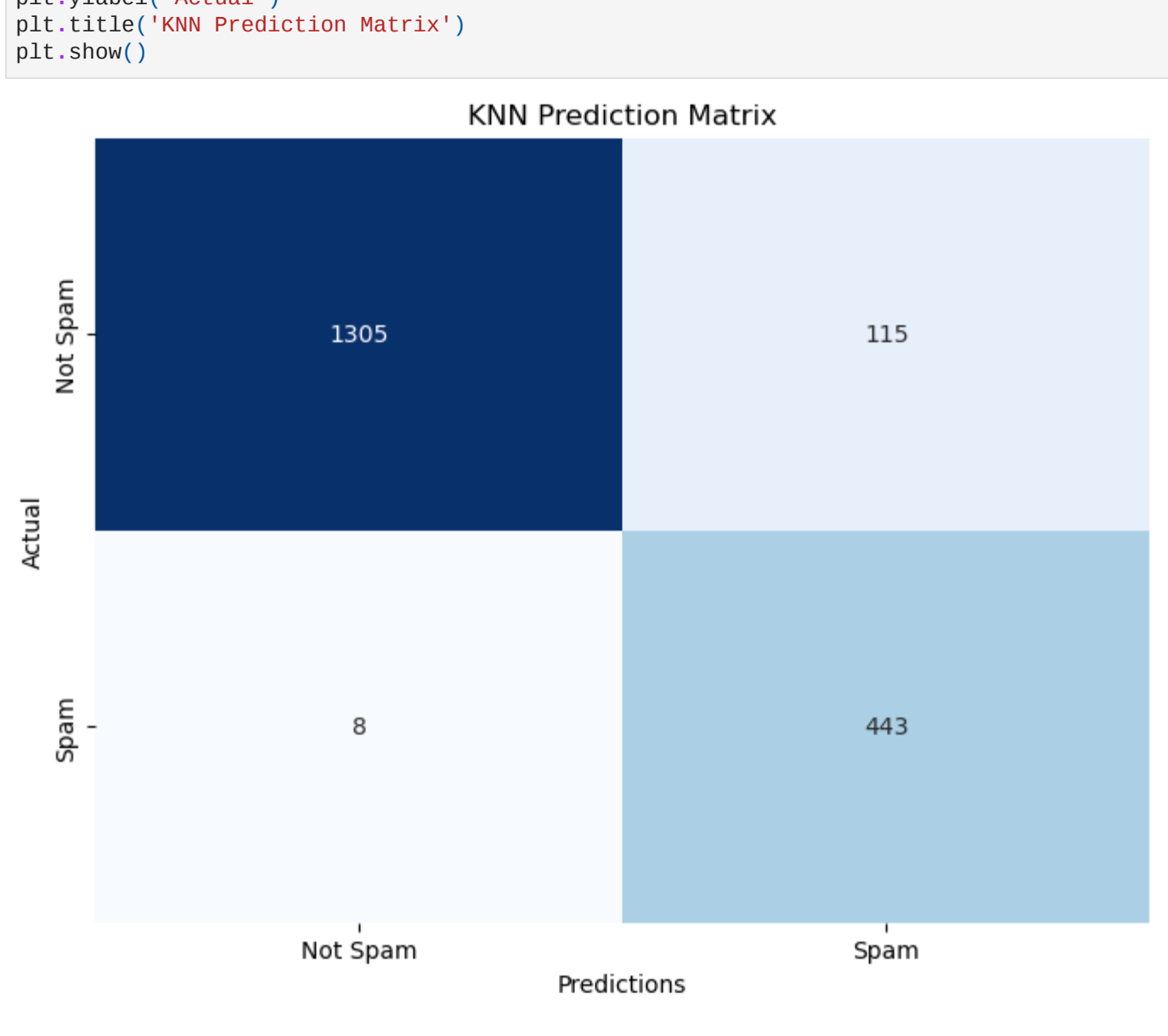
Old Accuracy Without Added Clustering Feature: 0.8508818813468734

In [37]:
```python
knn = KNeighborsClassifier(n_neighbors=2)
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)
accuracy_score(y_test, predictions)
```

Out[37]:
```
0.93425975414217
```

Confusion Matrix

In [38]:
```python
# Calculate confusion matrix
knn_matrix = confusion_matrix(y_test, predictions)

# heatmap(matrix plot)
plt.figure(figsize=(8,6))
sns.heatmap(knn_matrix, annot=True, fmt ='d',cmap = 'Blues',cbar = False,
            xticklabels=['Not Spam', 'Spam'], yticklabels=['Not Spam', 'Spam'])
plt.xlabel('Predictions')
plt.ylabel('Actual')
plt.title('KNN Prediction Matrix')
plt.show()
```



Classification Report

In [39]:
```python
from sklearn.metrics import classification_report
classification_rep = classification_report(y_test, predictions)
print('KNN Classification Report:','\n',classification_rep)
```

```
KNN Classification Report:
               precision    recall  f1-score   support

           0       0.99      0.92      0.95      1420
           1       0.79      0.98      0.88       451

    accuracy                           0.93      1871
   macro avg       0.89      0.95      0.92      1871
weighted avg       0.95      0.93      0.94      1871
```

Naive Bayes

In [40]:
```python
from sklearn.naive_bayes import MultinomialNB

naive_bayes = MultinomialNB()

naive_bayes.fit(X_train, y_train)
```

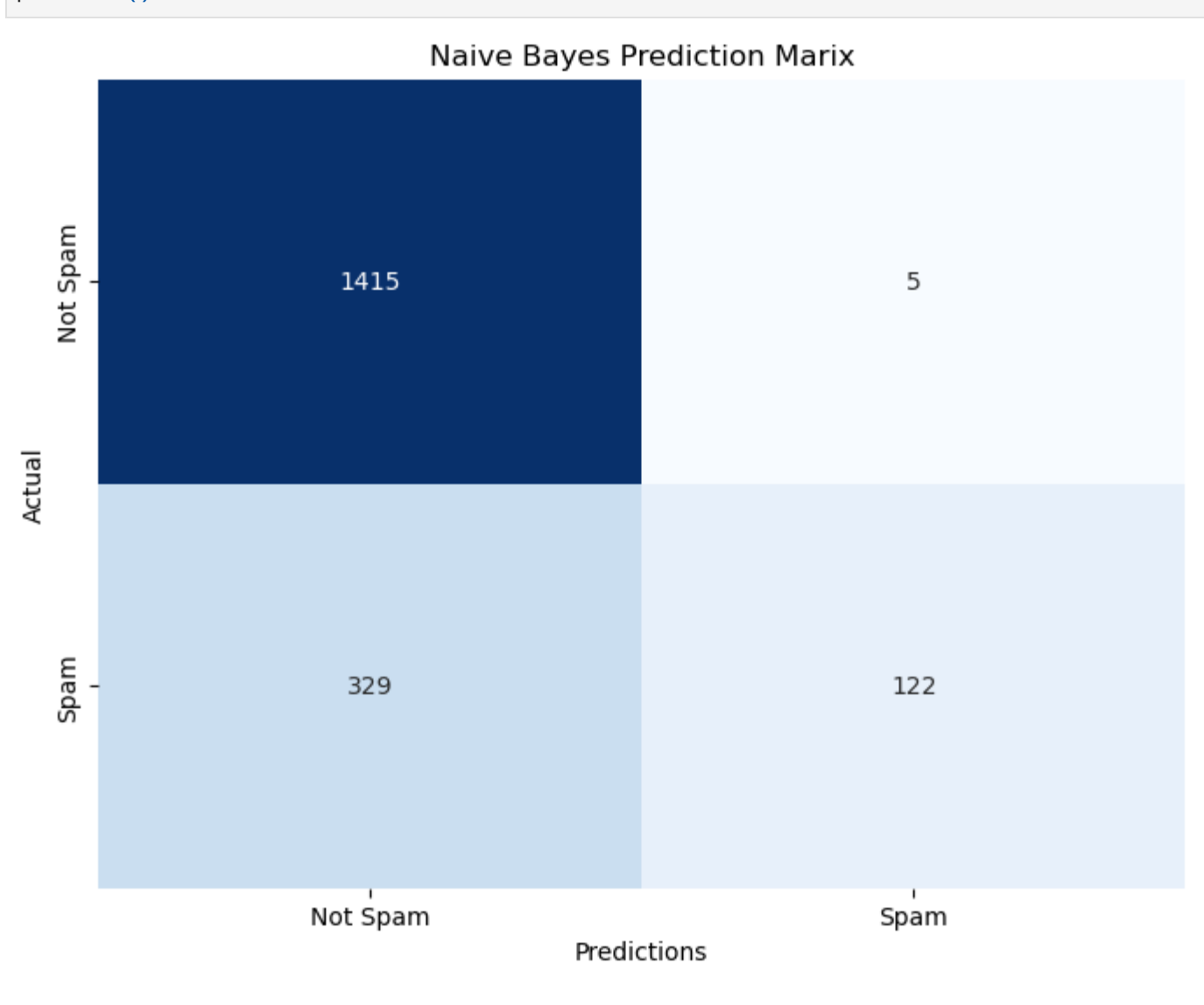
Out[40]:
```
▾ MultinomialNB
MultinomialNB()
```

In [41]:
```python
predictions = naive_bayes.predict(X_test)
accuracy = accuracy_score(y_test, predictions)
print('Naive Bayes Accuracy:', accuracy)
```

```
Naive Bayes Accuracy: 0.8214858364510956
```

Confusion Matrix - Naive Bayes

In [42]:
```python
nb_confusion = confusion_matrix(y_test, predictions)

plt.figure(figsize=(8,6))
sns.heatmap(nb_confusion, annot=True, fmt = 'd', cmap = 'Blues', cbar = False,
            xticklabels=['Not Spam', 'Spam'], yticklabels=['Not Spam', 'Spam'])
plt.xlabel('Predictions')
plt.ylabel('Actual')
plt.title('Naive Bayes Prediction Marix')
plt.show()
```



In [45]:
```python
from sklearn.metrics import classification_report
classification_rep = classification_report(y_test, predictions)
print('Naive Bayes Classification Report:','\n',classification_rep)
```

```
Naive Bayes Classification Report:
               precision    recall  f1-score   support

           0       0.81      1.00      0.89      1420
           1       0.96      0.27      0.42       451

    accuracy                           0.82      1871
   macro avg       0.89      0.63      0.66      1871
weighted avg       0.85      0.82      0.78      1871
```