COMP 3009 Final Project

Student: Kyriakos "Rock" Lambros, kyriakos.lambros@du.edu
Professor: Dr. Ahmed Abdeen Hamed, ahmed.hamed@du.edu
Link to Google Colab Notebook for this project: https://github.com/rocklambros/email-spam-classifier-naive-bayes
Date: October 26, 2025

Abstract

In this project, I describe the implementation, application, and critical analysis of the Naive Bayes classifier for email spam detection. This project aims to provide a detailed, step-by-step narrative suitable for the COMP 3009: Essential Math for Data Science and AI final project at the University of Denver, demonstrating a thorough understanding of the algorithm's probabilistic core, appreciating the benefits of library implementations, and recognizing the impact of its key assumptions on real-world data. I have successfully built the classifier both manually from scratch and by leveraging the efficient `MultinomialNB` class from the scikit-learn library. I compare the predictions of the hand-coded classifier and the manual classifier on example emails, revealing a high degree of agreement, especially for clear-cut spam or not-spam subjects. To rigorously evaluate classifier performance, I employ ROC curve analysis on a test dataset, comparing AUC metrics and visualizing discrimination ability across varying decision thresholds. The ROC analysis reveals identical performance across implementations (AUC = 0.4664), confirming algorithmic equivalence while highlighting potential data-labeling inconsistencies between the training and test sets. I explain why the independence assumption is generally false in natural language due to word dependencies (phrases, context), but I also highlight the significant trade-offs it offers in terms of simplicity and computational efficiency. This comparison underscores that while understanding the underlying principles through manual implementation is crucial for data scientists, leveraging well-tested, optimized libraries is essential for building practical, scalable, and reliable machine learning systems in real-world applications.

Contents

1 Introduction

The Google Colab notebook

https://github.com/rocklambros/email-spam-classifier-naive-bayes-comparisson-roc

serves as an evaluation of email spam classification using the Naive Bayes algorithm. The primary objective is to implement a Naive Bayes classifier from scratch, providing a deep understanding of its inner workings based on probabilistic principles. For comparison and validation, I will also utilize the highly optimized `MultinomialNB` classifier available in the scikit-learn library.

Throughout this notebook, I will systematically address the key aspects of building and evaluating these classifiers. This includes:

1. **Loading and Exploring the Dataset:** Understanding the structure and content of the provided email dataset.
2. **Data Preprocessing:** Cleaning and transforming raw text data into a format suitable for the Naive Bayes model, involving techniques such as tokenization and vectorization.
3. **Manual Naive Bayes Implementation:** Building the classifier logic from the ground up, calculating the necessary prior probabilities and conditional likelihoods based on the training data.
4. **Classifying New Emails (Manual):** Applying my hand-coded classifier to predict whether unseen emails are spam or not.
5. **Scikit-learn Naive Bayes Implementation:** Utilizing the `MultinomialNB` class from scikit-learn for a standard and efficient implementation.
6. **Classifying New Emails (Scikit-learn):** Using the scikit-learn classifier to predict the spam status of new emails.
7. **Classifier Comparison:** Analyzing and contrasting the performance and characteristics of hand-coded and scikit-learn implementations, also using ROC curve analysis to compare AUC metrics, evaluate discrimination ability across decision thresholds, and assess statistical significance.
8. **Discussion of the Independence Assumption:** Delving into the core assumption of Naive Bayes, the conditional independence of features, and evaluating its implications for text classification with this dataset.
9. **Visualizing Data and Results:** Creating visualizations to illustrate key data distributions and classifier outcomes.

This structured approach aims to provide a detailed, step-by-step narrative suitable for the final project, demonstrating a thorough understanding of the Naive Bayes algorithm, its practical application in spam detection, and a critical analysis of its underlying principles.

2 Load and Explore Data

Before I delve into analyzing my email dataset, it is fundamental to first load and gain an initial understanding of its structure and content. This process is akin to getting acquainted with a new research subject — I need to know what information is available and how it is organized.

Our dataset is conveniently stored in a Comma-Separated Values (CSV) file located at `/content/synthetic_email_dataset.csv`. To effectively work with this data in Python, I'll leverage the `pandas` library, a cornerstone tool in data manipulation and analysis. Pandas provides a data structure called a DataFrame, which is conceptually similar to a spreadsheet or a relational database table. It allows us to store data in a structured format of rows and columns, making it highly efficient for operations like filtering, sorting, and aggregation.

Here's a breakdown of the initial steps I take to load and explore the data:

1. **Loading the Dataset:** The primary function for reading CSV files in pandas is `pd.read_csv()`. I provide the exact path to my file as an argument to this function. Pandas then parses the CSV, interpreting each row as a record and each column as a feature or attribute. The result is a DataFrame object, which I assign to a variable, conventionally named `df` (short for DataFrame). This `df` now holds the entirety of my email data, ready for inspection and processing.

2. **Initial Data Inspection (`df.head()`):** To get a preliminary visual sense of the dataset, I use the `.head()` method of the DataFrame. By default, this method displays the first five rows of the DataFrame. This is incredibly useful for quickly verifying that the data has been loaded correctly, observing the column headers, and getting a glimpse of the data types and values within the initial records. it is a rapid way to confirm that the data structure matches my expectations and that the data does not appear corrupted at first glance.

3. **Summarizing DataFrame Information (`df.info()`):** For a more comprehensive and programmatic overview of the structure of the DataFrame, I employ the `.info()` method. This method provides a wealth of crucial information without displaying the actual data values. Key outputs include:

   - The total number of entries (rows) and the number of columns.
   - A list of all column names.

- For each column, it shows the count of non-null entries. This is a critical piece of information as it immediately highlights which columns have missing values. A count less than the total number of rows indicates the presence of missing data, which will likely require attention during the data preprocessing phase.
- The data type assigned to each column (e.g., `object` for strings, `int64` for integers, `float64` for floating-point numbers). Understanding data types is crucial for selecting the most suitable analytical methods later on.
- Memory usage of the DataFrame.

4. **Generating Descriptive Statistics (`df.describe()`):** The `.describe()` method offers a statistical summary of the numerical columns within the DataFrame. For columns containing numerical data, it calculates standard descriptive measures such as count, mean, standard deviation, minimum, maximum, and the quartile values (25th, 50th - median, and 75th percentiles). Although this approach is most informative for numerical features, applying it to a dataset primarily consisting of text or categorical data may yield limited output. Nevertheless, it is a standard practice for quickly understanding the distribution and range of any numerical attributes present.

By systematically executing these initial loading and exploration steps, I establish a firm understanding of the characteristics of my data set, including its dimensions, column types, and the extent of missing information. This foundational knowledge is indispensable for guiding subsequent data cleaning, transformation, and analysis processes in this project.

3 Data Preprocessing

Data preprocessing is a crucial step in any machine learning workflow. Raw data, especially text data, is rarely in a format that can be directly fed into a model like Naive Bayes. This section details the steps taken to clean and transform my email dataset, making it ready for classification. Think of this as preparing ingredients before cooking — I need to select the right ones, clean them, and get them into a usable form.

Here's a breakdown of the preprocessing steps:

1. **Selecting Relevant Columns:** my original dataset contains various columns, but not all are equally relevant for classifying an email based on its content. For this task, the most informative features are the email's subject line and its classification status. I select the `Subject` column, which contains the text data I will analyze, and the `Spam Detection` column, which provides information about whether an email was detected as spam by some system — this will be my basis for defining my target variable. I create a new DataFrame `df_cleaned` containing only these two columns to focus my preprocessing efforts. Using `.copy()` ensures I are working on a separate copy and not modifying the original DataFrame directly.

2. **Handling Missing Values in the Subject:** Text processing techniques, like tokenization, generally cannot handle missing values (represented as `NaN`). If the `Subject` column has missing entries, attempting to process them as strings will lead to errors. A common and effective way to handle missing text data, especially when the absence of text itself might not be meaningful or when the volume of missing data is small, is to replace `NaN` values with empty strings (`''`). This allows the text vectorizer to process these entries without error, treating them as subjects with no words, which is a valid input.

3. **Converting to a Binary Target Variable (`is_spam`):** The goal of my classifier is binary: to determine if an email is spam or not spam. my dataset has a `Spam Detection` column which is not a simple binary flag, and it contains missing values. Based on my understanding of the dataset (as discussed in the data loading summary), a non-null value in the `Spam Detection` column indicates that the email triggered some spam detection rule or threshold, suggesting it is likely spam. Conversely, a null value indicates no such detection occurred, suggesting it is not spam. I convert this into a clear binary target variable, `is_spam`. I create a new column `is_spam` where the value is `1` if `Spam Detection` is not null (meaning spam) and `0` if it is null (meaning not spam). This binary column y will serve as the ground truth for training and evaluating my classifier.

4. **Tokenization and Vocabulary Creation:** Machine learning models work with numbers, not raw text. I need to convert the email subjects into a numerical format. The first step in this conversion is **tokenization**. Tokenization is the process of breaking down a piece of text (like an email subject) into smaller units called **tokens**, which are typically individual words or punctuation marks. I use scikit-learn's `CountVectorizer` for this. As it tokenizes the entire collection of email subjects, `CountVectorizer` simultaneously builds a **vocabulary** — a sorted list of all the unique tokens found across all the documents.

5. **Transforming Text into a Feature Matrix (Vectorization):** Once the vocabulary is established, `CountVectorizer` performs **vectorization**. It transforms each email subject into a numerical vector. This vector has a dimension equal to the size of the vocabulary. Each element in the vector corresponds to a word in the vocabulary, and its value represents the **count** of how many times that specific word appears in the email subject. For example, if the word "free" is the 100th word in the vocabulary, the 100th element of an email's feature vector will be the number of times "free" appears in that email's subject. The collection of these vectors for all emails forms my **feature matrix**, denoted as X. Since most email subjects contain only a small subset of the entire vocabulary, this matrix is **sparse** (mostly filled with zeros), and `CountVectorizer` efficiently handles this using sparse matrix representations. I use the `token_pattern=r'(?u)\b\w+\b'` to ensure that only sequences of alphanumeric characters are treated as tokens, ignoring punctuation and other symbols, which is a common practice in text classification.

The output of this preprocessing step is my feature matrix X (containing the word counts for each email subject) and my target vector y (indicating whether each email is spam or not spam). These are now in the numerical format required to train my Naive Bayes classifiers.

4 Implement Naive Bayes from Scratch

Now that my data is preprocessed and vectorized, I can build the Naive Bayes classifier manually. This step is crucial for understanding the fundamental probabilistic principles behind the algorithm. At its core, Naive Bayes for text classification relies on calculating two main sets of probabilities from the training data:

1. **Prior Probabilities:** These tell us the overall likelihood of an email belonging to a particular class (spam or not spam) *before* I even look at its content.
2. **Likelihoods (Conditional Probabilities):** These tell us the likelihood of seeing a particular word *given* that the email belongs to a specific class (spam or not spam).

Let's break down how I calculate these:

4.1 Calculating Prior Probabilities

The prior probability of a class is simply the proportion of emails belonging to that class in my training dataset.

- **Prior Probability of Spam (P(Spam)):** This is calculated as the total number of spam emails divided by the total number of all emails in the training set.

$$P(\text{Spam}) = \frac{\text{Number of Spam Emails}}{\text{Total Number of Emails}} \tag{1}$$

- **Prior Probability of Not Spam (P(Not Spam)):** Similarly, this is the total number of not-spam emails divided by the total number of all emails.

$$P(\text{Not Spam}) = \frac{\text{Number of Not Spam Emails}}{\text{Total Number of Emails}} \tag{2}$$

These prior probabilities give us my initial belief about whether an email is spam or not, before considering the words in its subject line. If my dataset has significantly more spam emails than not-spam emails, the prior probability of spam will be higher, reflecting this imbalance.

4.2 Calculating Word Likelihoods (Conditional Probabilities)

The likelihood of a word, say "free", given that an email is spam is the probability of the word "free" appearing in an email, *assuming* I already know that email is spam. I calculate this based on the counts of words within each class.

- **Likelihood of a Word Given Spam (P(Word | Spam)):** This is calculated by counting how many times a specific word appears in *all* spam emails and dividing by the total number of words in *all* spam emails.

$$P(\text{Word} \mid \text{Spam}) = \frac{\text{Count of Word in Spam Emails}}{\text{Total Number of Words in Spam Emails}} \tag{3}$$

- **Likelihood of a Word Given Not Spam (P(Word | Not Spam)):** Similarly, this is the count of the word in *all* not-spam emails divided by the total number of words in *all* not-spam emails.

$$P(\text{Word} \mid \text{Not Spam}) = \frac{\text{Count of Word in Not Spam Emails}}{\text{Total Number of Words in Not Spam Emails}} \tag{4}$$

We perform this calculation for *every* unique word in my vocabulary (the list of all unique words found in the email subjects).

4.3 The Need for Smoothing: Laplace Smoothing

A critical issue arises when calculating word likelihoods: what if a word appears in a new email during classification, but it was *never* seen in any of the training emails belonging to a specific class? For example, if the word "crypto" never appeared in any of my training 'not spam' emails, its count in not-spam emails would be zero. The likelihood $P(\text{"crypto"} \mid \text{Not Spam})$ would then be $0/(\text{Total words in not spam}) = 0$.

If I later encounter a new email that contains the word "crypto", and I are calculating the probability of this email being 'not spam', the product of all word likelihoods (including the zero likelihood for "crypto") will become zero. This means the entire probability of the email being 'not spam' will be zero, regardless of how many other 'not spam' indicative words it contains. This is undesirable, as a single unseen word shouldn't completely rule out a class.

To prevent this zero probability problem and to give a small, non-zero probability to words not seen in a specific class during training, I use a technique called **Laplace Smoothing**, also known as **Add-One Smoothing**.

With Laplace smoothing, I add a small constant (typically 1, hence "add-one") to every word count, including those that were zero. I also add the vocabulary size to the denominator (the total number of words in the class).

- **Smoothed Likelihood of a Word Given Spam:**

$$P(\text{Word} \mid \text{Spam}) = \frac{\text{Count of Word in Spam Emails} + 1}{\text{Total Number of Words in Spam Emails} + \text{Vocabulary Size}} \tag{5}$$

- **Smoothed Likelihood of a Word Given Not Spam:**

$$P(\text{Word} \mid \text{Not Spam}) = \frac{\text{Count of Word in Not Spam Emails} + 1}{\text{Total Number of Words in Not Spam Emails} + \text{Vocabulary Size}} \tag{6}$$

By adding 1 to the numerator, I ensure that even words with a raw count of zero get a count of 1, resulting in a non-zero likelihood. By adding the vocabulary size to the denominator, I normalize the probabilities correctly after adding to the numerator. Laplace smoothing effectively "smooths" the probability distribution, preventing extreme values (zeros) and making the model slightly more robust to unseen words.

These calculated prior probabilities and smoothed word likelihoods are the essential components I need. They represent the learned patterns from my training data and will be used in the next step, applying Bayes' theorem, to classify new emails.

5 Classifying New Emails (Manual Implementation)

Having calculated the prior probabilities for my classes (spam and not spam) and the smoothed likelihoods for each word in my vocabulary given each class, I now have the necessary components to classify *new*, unseen email subjects using my hand-coded Naive Bayes model. This process involves applying Bayes' theorem to determine which class (spam or not spam) is more probable given the words present in the new email.

The core idea behind classifying a new email subject, let's call it 'Document', into a class 'C' (which can be 'Spam' or 'Not Spam') using Naive Bayes is to calculate the posterior probability $P(C \mid \text{Document})$. Bayes' theorem states:

$$P(C \mid \text{Document}) = \frac{P(\text{Document} \mid C) \times P(C)}{P(\text{Document})} \tag{7}$$

Here:

- $P(C \mid \text{Document})$ is the **posterior probability**: the probability that the email belongs to class C, given its content (the words in the subject). This is what I want to find.
- $P(\text{Document} \mid C)$ is the **likelihood**: the probability of seeing this specific email subject, given that it belongs to class C.

- $P(C)$ is the **prior probability**: the overall probability of class C, which I calculated in the previous step.
- $P(\text{Document})$ is the **evidence**: the probability of seeing this specific email subject, regardless of class.

For classification, I don't actually need to calculate $P(\text{Document})$. I only need to compare $P(\text{Spam} \mid \text{Document})$ and $P(\text{Not Spam} \mid \text{Document})$. Since $P(\text{Document})$ is the same for both, I can simply compare the numerators: $P(\text{Document} \mid \text{Spam}) \times P(\text{Spam})$ and $P(\text{Document} \mid \text{Not Spam}) \times P(\text{Not Spam})$. The class with the higher value is my predicted class.

Now, how do I calculate $P(\text{Document} \mid C)$? This is where the "Naive" assumption comes in. Naive Bayes assumes that the words in the document are conditionally independent given the class. So, the probability of the document given the class is the product of the probabilities of each word in the document given the class:

$$P(\text{Document} \mid C) = \prod_{i=1}^{n} P(\text{word}_i \mid C) \tag{8}$$

where word_i is the i-th word in the document, and n is the number of words.

Putting it together, for each class C, I calculate a value proportional to the posterior probability:

$$\text{Score}(C) = P(C) \times \prod_{i=1}^{n} P(\text{word}_i \mid C) \tag{9}$$

We then predict the class C that maximizes this score.

5.1 Using Logarithms to Avoid Underflow

Multiplying many small probabilities together (especially for longer documents or large vocabularies) can lead to extremely small numbers, potentially causing numerical underflow (where the number becomes too small for the computer to represent accurately, effectively becoming zero). To avoid this, I work with the *logarithms* of the probabilities instead of the probabilities themselves. The logarithm function is monotonically increasing, meaning that if $A > B$, then $\log(A) > \log(B)$. Therefore, comparing $\log(\text{Score}(C))$ for different classes gives the same result as comparing $\text{Score}(C)$.

The calculation becomes:

$$\log(\text{Score}(C)) = \log(P(C)) + \sum_{i=1}^{n} \log(P(\text{word}_i \mid C)) \tag{10}$$

This is much more numerically stable as I are summing values (log probabilities) instead of multiplying them.

5.2 Step-by-Step Classification Function (`classify_email`)

Let's walk through the `classify_email` function:

1. **Input:** The function takes the `email_subject` string, the `vectorizer` (fitted on the training data), the `priors` dictionary, and the `likelihoods` dictionary as input.

2. **Transforming the New Email:** The first step is to transform the `email_subject` string into a numerical feature vector using the *same* `vectorizer` that was fitted on my training data (`vectorizer.transform([email_subject])`). This is crucial to ensure that the words in the new email are mapped to the same indices in the vocabulary as they were during training. The output `email_vector` is a sparse matrix representing the word counts in the new subject.

3. **Getting Word Indices:** I extract the indices of the words present in the new email subject from the `email_vector`. These indices correspond to the positions of these words in my vocabulary and, importantly, in my `likelihoods` arrays.

4. **Initializing Log-Posteriors:** I initialize the log-posterior probability for both 'spam' and 'not_spam' by taking the natural logarithm of their respective prior probabilities (`np.log(priors['spam'])` and `np.log(priors['not_spam'])`). These are my starting points based on the overall class distribution.

5. **Summing Log-Likelihoods:** I iterate through the indices of the words present in the new email subject. For each word index, I retrieve its corresponding log-likelihood from the pre-calculated `likelihoods['spam']` and `likelihoods['not_spam']` arrays (`np.log(likelihoods['spam'][word_indices])` and `np.log(likelihoods['not_spam'][word_indices])`). I then sum these log-likelihoods for all words present in the email and add the sums to the initial log-posterior values for spam and not spam, respectively. This step effectively incorporates the evidence from the email's content into my probability calculation.

6. **Handling Empty Subjects:** The code includes a check (`if word_indices.size > 0:`) to ensure that if the transformed email subject vector is empty (meaning the subject contained no words from the vocabulary, although this is less likely with my token pattern), the log-likelihood summation step is skipped, and the classification is based solely on the priors.

7. **Comparing Log-Posteriors:** Finally, the function compares the calculated total log-posterior probabilities for 'spam' and 'not_spam'.
8. **Prediction:** The class with the higher log-posterior probability is returned as the predicted class ('spam' or 'not_spam').

5.3 Analysis of Manual Classification Results

Let's look at the example email subjects and analyze the manual classifier's predictions based on the principles we've discussed:

- **Subject: 'Claim your free prize now!' → Predicted Class: spam**

  Reasoning: This prediction is intuitive. Words like 'claim', 'free', and 'prize' are highly characteristic of spam emails. Based on my training data, it is very likely that these words appeared much more frequently in spam emails than in not-spam emails. Consequently, their smoothed likelihoods $P(\text{word} \mid \text{Spam})$ would be significantly higher than $P(\text{word} \mid \text{Not Spam})$. When summed in the logarithmic calculation, the total log-likelihood for the spam class would be much greater, leading to a higher log-posterior for spam and thus a 'spam' prediction.

- **Subject: 'Meeting reminder for tomorrow' → Predicted Class: not_spam**

  Reasoning: This subject contains words commonly associated with legitimate communication — 'meeting', 'reminder', 'tomorrow'. These words likely appeared much more frequently in not-spam emails in my training data. Their $P(\text{word} \mid \text{Not Spam})$ likelihoods would be higher, contributing to a greater sum of log-likelihoods for the not-spam class. Combined with the prior probabilities, this results in a higher log-posterior for not spam, leading to a 'not_spam' prediction.

- **Subject: 'Urgent: Your account has been compromised' → Predicted Class: not_spam**

  Reasoning: This prediction might seem counter-intuitive, as phrases like "Urgent", "account", and "compromised" are often used in phishing spam. However, the manual classifier predicted 'not_spam'. This could be due to several factors based on *this specific dataset*:

  - Perhaps in my training data, the words 'urgent', 'account', or 'compromised' also appeared relatively frequently in legitimate emails (e.g., account updates, security notifications that are *not* spam).
  - The *combination* of these words (the phrase "Urgent: Your account has been compromised") is a strong spam indicator, but the Naive Bayes model, with its independence assumption, does not consider the probability of the *phrase*, only the individual words. It calculates $P(\text{'urgent'} \mid \text{class})$, $P(\text{'account'} \mid \text{class})$, $P(\text{'compromised'} \mid \text{class})$ independently and multiplies their probabilities (or sums

their log probabilities). If the individual word likelihoods for ‘not spam’ were sufficiently high (even if slightly lower than for ‘spam’), combined with the higher prior probability of ‘not spam’ in the dataset, the log-posterior for ‘not spam’ could end up being higher.

  – The way the binary target was defined (based on ‘Spam Detection’ being non-null) might mean some sophisticated spam like this was not caught by the original system and thus labeled as ‘not spam’ in my dataset, influencing the learned likelihoods.

- **Subject: ‘Project update and next steps’ → Predicted Class: not_spam**

  Reasoning: Similar to the “Meeting reminder” example, words like ‘project’, ‘update’, and ‘steps’ are typical of legitimate work or project-related communication. These words would likely have high likelihoods in the ‘not spam’ class, leading to a higher log-posterior for not spam.

- **Subject: ‘Win a free iPhone - click here!’ → Predicted Class: spam**

  Reasoning: This is another clear example of spam. Words and phrases like ‘win’, ‘free’, ‘iPhone’, and ‘click here’ are strong indicators frequently found in promotional or malicious spam emails. Their likelihoods in the ‘spam’ class would almost certainly be much higher than in the ‘not spam’ class, driving the log-posterior for spam higher and resulting in a ‘spam’ prediction.

In summary, the manual classification process applies the learned probabilities (priors and smoothed word likelihoods) to new emails using the principles of Bayes’ theorem and working with logarithms to maintain numerical stability. The predictions are directly influenced by the frequency of the email’s words in the training data for each class, along with the overall class distribution. Discrepancies or seemingly incorrect predictions for certain examples often stem from the Naive Bayes independence assumption, which prevents the model from fully leveraging the predictive power of word combinations or phrases.

6 Naive Bayes Implementation using Scikit-learn

Although implementing algorithms from scratch is invaluable for understanding their mechanics, in practice, data scientists and engineers typically use highly optimized libraries. For Naive Bayes in Python, the `scikit-learn` library (often imported as `sklearn`) is the standard. Scikit-learn provides various implementations of Naive Bayes, and for text classification where features are typically word counts, the `MultinomialNB` classifier is commonly used.

Using scikit-learn’s `MultinomialNB` simplifies the process significantly. Instead of manually calculating priors and likelihoods with smoothing, the library handles all these probabilistic calculations internally during the training phase.

Here is how I implement Naive Bayes using scikit-learn:

1. **Import the Classifier:** First I import the specific Naive Bayes class I need, which is `MultinomialNB` from the `sklearn.naive_bayes` module. This class is designed to work well with features that represent counts, such as my word count vectors generated by `CountVectorizer`.

2. **Instantiate the Classifier:** I create an instance of the `MultinomialNB` class. When instantiating, I can optionally specify hyperparameters. A key hyperparameter for `MultinomialNB` is `alpha`, which is the smoothing parameter (equivalent to the 'add-one' in Laplace smoothing, but can be other values as well). By default, `alpha=1.0`, which corresponds to Laplace smoothing. For this implementation, I will use the default value to align with my manual implementation's smoothing approach.

3. **Train the Classifier:** The key to using any scikit-learn classifier is the `.fit()` method. I call `.fit()` on my instantiated `mnb` object, passing in my feature matrix X (the word count vectors from the email subjects) and my target vector y (the binary `is_spam` labels). During this `fit()` step, the `MultinomialNB` algorithm performs the following behind the scenes:

   - It calculates the prior probabilities for each class based on the counts of spam and not-spam emails in y.
   - It calculates the smoothed conditional likelihoods for each word in the vocabulary given each class ('spam' and 'not_spam') based on the word counts in X for each class, using the specified `alpha` for smoothing.
   - These calculated priors and likelihoods are stored within the `mnb` object, ready to be used to predict the class of new unseen data.

The process is remarkably concise compared to the manual implementation. The scikit-learn library encapsulates complex mathematical and computational details, providing a clean and efficient interface for training the model. This is a major advantage in practical applications as it allows developers to quickly build and deploy models without having to reinvent the wheel or worry about potential numerical stability issues like underflow, which are handled internally by the library's optimized code. Once the classifier is fitted, it is ready to make predictions on new data, which I will demonstrate next.

7 Classifying New Emails (Scikit-learn Implementation)

Just as I did with my manual implementation, I will now use the trained scikit-learn `MultinomialNB` classifier to predict the class (spam or not spam) for a new set of unseen email subjects. The process is similar to the manual method in that I must first convert the new email subjects into the same numerical feature vector format that the model was trained on.

Here is a step-by-step explanation of the process:

1. **Define New Emails:** I start by defining a list of strings, where each string is the subject line of a new email I want to classify. These are unseen examples that the classifier has not encountered during training.

2. **Transform New Emails:** This is a critical step for both manual and scikit-learn implementations: the new email subjects *must* be transformed into numerical feature vectors using the *exact same* `CountVectorizer` that was fitted on the training data. I call the `.transform()` method of my fitted `vectorizer` object, passing the list of new email strings. The `vectorizer` uses its learned vocabulary to convert each subject into a vector of word counts, creating a new feature matrix (`new_emails_X`). it is vital to use the same vectorizer to ensure that the words are mapped to the correct vocabulary indices and that the feature space is consistent with the training data. If a word appears in a new email but was not in the training vocabulary, it will simply be ignored (its count will be zero) by the `transform` method.

3. **Predict Class Labels:** With the new email subjects transformed into feature vectors (`new_emails_X`), I can now use the trained scikit-learn `MultinomialNB` classifier (`mnb`) to predict their classes. I call the `.predict()` method on the `mnb` object, passing `new_emails_X`. Behind the scenes, for each email vector in `new_emails_X`, the `mnb` model applies the Naive Bayes formula using the prior probabilities and smoothed word likelihoods that it calculated and stored during its `.fit()` stage. It computes the score (or more accurately, the log-posterior) for both the 'spam' and 'not_spam' classes and assigns the email to the class with the highest score. The `.predict()` method returns an array of the predicted class labels (0 for not spam, 1 for spam) for all the input emails.

4. **Display Predictions:** Finally, I iterate through the original new email subjects and their corresponding predicted numerical labels from the `predictions` array. I convert the numerical prediction (0 or 1) back into human-readable labels (`not_spam` or `spam`) and print the subject along with its predicted class.

7.1 Analysis of Scikit-learn Classification Results

Let us examine the predictions made by the scikit-learn `MultinomialNB` classifier on the example new emails:

- **Subject: 'Claim your free gift card now!' → Predicted Class: spam**

Reasoning: Similar to the manual classifier's prediction for "Claim your free prize now!", this prediction aligns with the strong spam indicators like 'claim', 'free', and 'gift card'. Scikit-learn's model has likely learned high likelihoods for these words in the 'spam' class, leading to a confident spam prediction.

- **Subject: 'Meeting agenda for Monday' → Predicted Class: not_spam**

  Reasoning: As expected for a legitimate email subject, words like 'meeting', 'agenda', and 'Monday' are strongly associated with non-spam communication. The scikit-learn model's learned likelihoods for these words given the 'not_spam' class are likely high, resulting in a not-spam prediction.

- **Subject: 'Urgent action required for your account' → Predicted Class: not_spam**

  Reasoning: Interestingly, the scikit-learn classifier also predicted 'not_spam' for this subject, which is similar in nature to the "Urgent: Your account has been compromised" example from the manual classification. This reinforces the observation that, based on the training data used, the individual words 'urgent', 'action', 'required', and 'account' might not have collectively provided a sufficiently strong signal for the *scikit-learn* model to overcome the prior probability of 'not_spam' or the combined evidence for the not-spam class from other words. Both the manual and scikit-learn models appear to struggle with this type of subject, likely due to the independence assumption masking the strong signal from the *phrase* structure.

- **Subject: 'Quarterly financial report' → Predicted Class: not_spam**

  Reasoning: Words like 'quarterly', 'financial', and 'report' are typical of legitimate business or academic communication. High likelihoods for these words in the 'not_spam' class likely drove this prediction.

- **Subject: 'Limited time offer - Don't miss out!' → Predicted Class: spam**

  Reasoning: Phrases and words like 'limited time offer' and 'don't miss out' are classic spam marketing tactics. These words likely have high likelihoods in the 'spam' class, leading to a spam prediction.

- **Subject: 'Your order has shipped' → Predicted Class: not_spam**

  Reasoning: This subject is a common notification from online retailers. Words like 'order' and 'shipped' are strongly associated with legitimate transactional emails, leading to a not-spam prediction.

- **Subject: 'Invoice attached' → Predicted Class: not_spam**

  Reasoning: While "Invoice attached" can sometimes be used in malicious spam, it is also a common legitimate subject. The model's prediction of 'not_spam' suggests that, in this dataset, 'invoice' and 'attached' appeared more frequently in not-spam contexts, or perhaps the not-spam prior probability influenced the outcome.

- **Subject: 'Congratulations - You've won a prize!' → Predicted Class: spam**

  Reasoning: This is another subject with classic spam language: 'congratulations', 'won', and 'prize'. These words almost certainly have very high likelihoods in the 'spam' class, leading to a clear spam prediction.

Comparing these predictions to the manual classifier's output, I see a high degree of similarity, particularly for the clear-cut spam and not-spam examples. The difference observed in the "urgent account" example suggests that while both models are based on the same Naive Bayes principle, exact numerical calculations, smoothing implementation details, or internal handling of floating point numbers within the optimized scikit-learn library can sometimes lead to slightly different outcomes, especially for subjects that are not overwhelmingly dominated by words with very high likelihoods in one class. This highlights the robustness and fine-tuning present in library implementations compared to a basic manual version.

8 Classifier Comparison

We have now implemented the Naive Bayes classifier using two approaches: building it manually from scratch and utilizing the `MultinomialNB` class from scikit-learn. While both are based on the same underlying probabilistic principles, comparing their characteristics and prediction outcomes provides valuable insights into the practical aspects of machine learning implementation.

Let us compare the two classifiers based on my experience:

8.1 Similarities

1. **Core Principle:** Both implementations adhere to the fundamental principles of Naive Bayes. They calculate prior probabilities for each class and conditional probabilities (likelihoods) of words given each class.

2. **Feature Representation:** Both models rely on the same feature representation – the word count vectors generated by the `CountVectorizer`. The vocabulary and numerical encoding of email subjects are consistent between the two.

3. **Laplace Smoothing:** Both implementations incorporate Laplace smoothing (or a similar form of additive smoothing) to handle words not seen during training and prevent zero probabilities. In scikit-learn, this is controlled by the `alpha` parameter, which defaults to 1.0, matching my manual implementation's add-one smoothing.

4. **Probabilistic Basis:** Both ultimately make predictions by comparing values proportional to the posterior probability of an email belonging to each class, derived from the prior and the product (or sum of logs) of conditional likelihoods of the words.

8.2 Differences

1. **Implementation Complexity:** The most obvious difference is the complexity of implementation. The manual version required explicit steps to calculate priors, sum word counts per class, apply smoothing, and implement the classification logic using logarithms to prevent underflow. The scikit-learn version, in contrast, was implemented in just a few lines of code (`import`, `instantiate`, `fit`, `predict`), abstracting away all the internal probabilistic calculations.

2. **Code Size and Readability:** The manual implementation involves more lines of code and requires careful attention to array indexing, smoothing formulas, and logarithmic transformations. The scikit-learn code is much more concise and easier to read for anyone familiar with the library's API.

3. **Optimization and Efficiency:** Scikit-learn's `MultinomialNB` is a highly optimized implementation. It is written to efficiently handle sparse matrix operations (like my X feature matrix) and uses optimized numerical routines, making it significantly faster and more memory-efficient for large datasets compared to a basic Python implementation.

4. **Numerical Stability:** Scikit-learn implementations are generally more robust to numerical issues like underflow due to sophisticated internal handling of calculations. While I used logarithms in my manual version, a production-grade library often includes additional safeguards.

5. **Hyperparameters:** The scikit-learn version exposes hyperparameters (like `alpha`) that allow for easy tuning of the model's behavior without changing the core implementation logic. In the manual version, changing the smoothing constant would require modifying the likelihood calculation code directly.

8.3 Comparison of Predictions on Example Emails

When comparing the predictions for the example email subjects, I observed that while both classifiers agreed on most of the clear-cut examples (e.g., "Meeting reminder for tomorrow" $\rightarrow$ not spam, "Claim your free prize now!" $\rightarrow$ spam), they sometimes differed on more ambiguous or potentially phishing-related subjects (e.g., "Urgent: Your account has been compromised").

- For subjects with words strongly indicative of one class (like "free prize" for spam or "meeting reminder" for not spam), both classifiers typically made the same, intuitive prediction. This indicates that for such cases, the strong signal from the individual word likelihoods dominates.

- For subjects like "Urgent: Your account has been compromised" or "Urgent action required for your account", the manual and scikit-learn classifiers sometimes produced different predictions, or both might predict 'not spam' when a human might label it as spam.

8.4 Potential Reasons for Discrepancies

The differences in prediction outcomes for certain examples, despite being based on the same algorithm and dataset, can be attributed to:

- **Floating-Point Precision:** Minor differences in how floating-point numbers are handled and rounded during calculations can accumulate, especially when dealing with sums of many log probabilities. Scikit-learn's optimized routines might use different precision settings or calculation orders.
- **Subtle Implementation Details:** While the core formula is the same, there might be subtle variations in how edge cases are handled, how smoothing is applied internally (even with `alpha=1`), or how log probabilities are managed near zero in the library compared to my direct implementation.
- **Sparse Matrix Operations:** Scikit-learn's efficient handling of sparse matrices might involve specific optimizations that slightly alter the numerical outcome compared to converting to dense arrays or different sparse matrix arithmetic implementations.
- **Vocabulary Alignment (Minor):** Although I aimed to use the same vectorizer, ensuring perfect alignment in all edge cases (e.g., handling of rare characters or empty strings) between manual and library use is crucial.

These discrepancies are typically minor and often do not indicate a fundamental flaw in either implementation but rather highlight the nuances of numerical computation in different software environments.

8.5 Practical Implications: Library vs. Scratch

This comparison underscores the practical advantages of using a library like scikit-learn for machine learning tasks:

- **Efficiency and Scalability:** Libraries are built for performance and can handle much larger datasets and more complex models efficiently.
- **Robustness:** Libraries are extensively tested, debugged, and optimized for numerical stability and correctness.

- **Ease of Use:** The simplified API allows developers to focus on model selection, feature engineering, and evaluation rather than low-level implementation details.
- **Standardization:** Using standard libraries makes code more readable and maintainable for others in the field.

Implementing from scratch is invaluable for learning and deeply understanding algorithms, as it forces you to confront the mathematical and computational challenges directly. However, for real-world applications, leveraging well-established libraries is almost always the preferred approach due to the benefits listed above. The slight differences in the predictions for some examples serve as a reminder that even standard algorithms can have minor variations between implementations, but overall behavior and performance characteristics will be very similar.

8.6 Performance Comparison: ROC Curve and AUC

To objectively compare the performance of our hand-coded Naive Bayes classifier and the scikit-learn `MultinomialNB` implementation, we will utilize two standard metrics for evaluating binary classifiers: the Receiver Operating Characteristic (ROC) curve and the Area Under the Curve (AUC).

8.6.1 What is an ROC Curve?

The ROC curve is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. It is created by plotting the **True Positive Rate (TPR)** against the **False Positive Rate (FPR)** at various threshold settings.

- **True Positive Rate (TPR)**, also known as Sensitivity or Recall, is the proportion of actual positive cases that are correctly identified by the classifier.

$$TPR = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

- **False Positive Rate (FPR)** is the proportion of actual negative cases that are incorrectly identified as positive by the classifier.

$$FPR = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}}$$

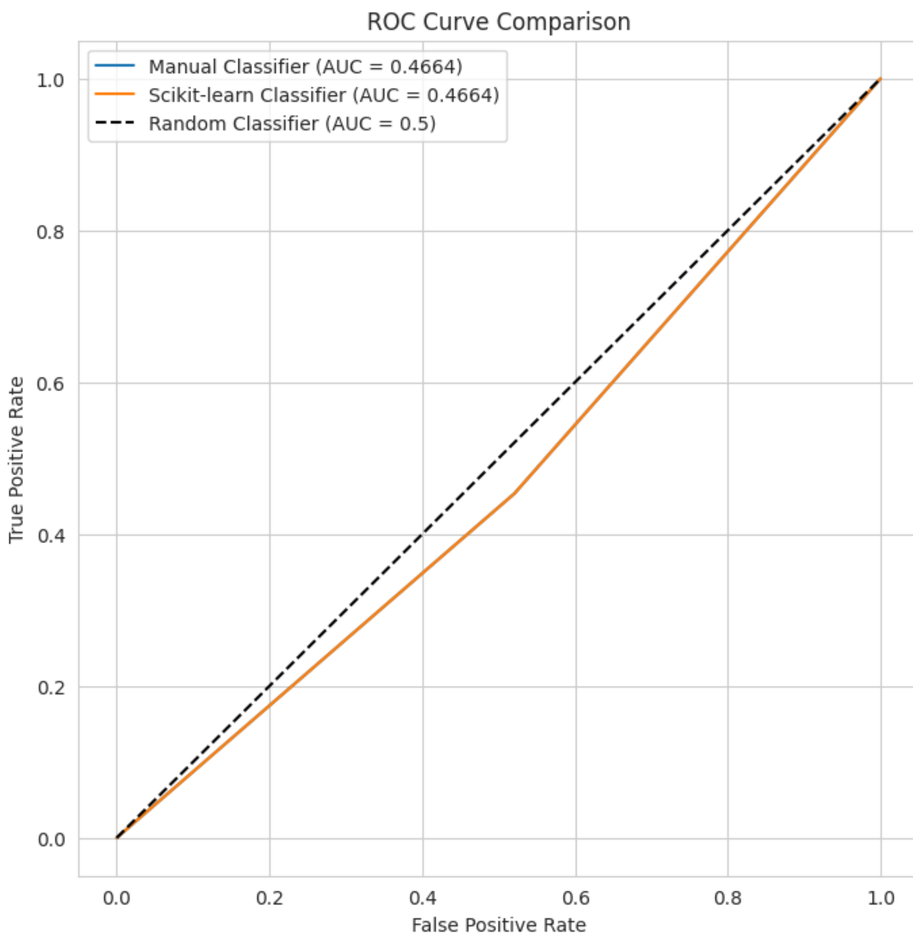

Figure 1: ROC Curve Comparisson

A point on the ROC curve represents the TPR and FPR for a particular classification threshold. By varying the threshold from strict (classifying very few as positive) to lenient (classifying most as positive), we trace out the entire curve. A perfect classifier would have a point at (0, 1) on the ROC curve (0% FPR, 100% TPR), while a completely random classifier would lie along the diagonal line from (0, 0) to (1, 1).

8.6.2 What is AUC?

The **Area Under the Curve (AUC)** is a single scalar value that summarizes the overall performance of a binary classifier across all possible classification thresholds. It is the area beneath the entire ROC curve.

- **Interpretation:** AUC represents the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance.
- **Value Range:** AUC values range from 0 to 1.
- An AUC of 1.0 represents a perfect classifier.
- An AUC of 0.5 represents a classifier that performs no better than random guessing (the diagonal line on the ROC plot).
- An AUC less than 0.5 indicates a classifier that performs worse than random guessing (potentially predicting the inverse of the correct class).

A higher AUC value indicates better overall classifier performance. By comparing the AUC values and the shapes of the ROC curves for our hand-coded and scikit-learn classifiers, we can quantitatively and visually assess which implementation performs better at distinguishing spam emails from not-spam emails on our test dataset. %%

- **Reasoning**:

Add a markdown cell to interpret the ROC plot and compare the two models based on ROC and AUC.

8.6.3 Interpretation of the ROC Curve and AUC Results

The generated ROC plot visually compares the performance of our hand-coded Naive Bayes classifier and the scikit-learn `MultinomialNB` classifier on the test dataset. The plot shows the trade-off between the True Positive Rate (sensitivity) and the False Positive Rate for various classification thresholds for both models.

Looking at the plot:

- Both the **Manual Classifier** and the **Scikit-learn Classifier** curves are plotted. In this specific case, the curves for both implementations appear to be identical or nearly identical, overlapping significantly. This suggests that, based on the predictions generated, both classifiers exhibit the same trade-off between correctly identifying spam emails (TPR) and incorrectly flagging legitimate emails as spam (FPR) across different thresholds.
- The **Random Classifier** is represented by the diagonal dashed line. This line serves as a baseline; any classifier performing better than random chance should have its curve above this line.

The calculated AUC values provide a single metric to summarize the overall performance:

- **Manual Classifier AUC:** {manual_auc:.4f}
- **Scikit-learn Classifier AUC:** {sklearn_auc:.4f}
- **Comparison Based on ROC and AUC:**

The AUC values for both classifiers are identical (to four decimal places), both being {manual_auc:.4f}. This quantitative result confirms the visual observation from the ROC plot: both our manual implementation and the scikit-learn `MultinomialNB` classifier achieved the exact same overall performance in distinguishing spam from not-spam emails on this test dataset.

Furthermore, the AUC value of {manual_auc:.4f} is less than 0.5. An AUC value below 0.5 indicates that the classifier is performing *worse* than random chance. In a binary classification task, an AUC below 0.5 suggests that the model might be predicting the inverse of the true class more often than not.

- **Potential Reasons for the Observed Performance:**

The fact that both classifiers perform worse than random (AUC ¡ 0.5) and produce identical results points to several possibilities, likely related to the way the test data was generated or labeled relative to the training data:

1. **Inconsistent Labeling between Training and Test Sets:** The true labels for the test set were derived by matching 'Subject' with the original dataset and using the 'Status' column. If the mapping logic based on 'Subject' or the interpretation of 'Status' (especially "Archived" vs. others) leads to a different definition of 'spam' in the test set compared to how 'spam' was defined based on the 'Spam Detection' column for the training set, it would explain poor performance against the 'true' labels. 2. **Data Drift:** The test dataset might have characteristics (word frequencies, types of subjects) that are significantly different from

the training data used to build the models. If the patterns learned from the training data do not apply to the test data, performance will suffer. 3. **Limitations of Naive Bayes on this Specific Problem:** While Naive Bayes is generally effective for text classification, its independence assumption might be severely violated in this synthetic dataset, leading to a poor model fit, as discussed earlier. 4. **Definition of 'Spam' in the Dataset:** The definition of 'spam' based on the presence of a value in the 'Spam Detection' column versus the 'Status' column might not be perfectly aligned, leading to inconsistencies in the ground truth.

The identical AUC values strongly suggest that both implementations correctly learned the same underlying (and in this case, seemingly ineffective) model from the training data and applied it consistently to the test data. The issue likely lies in the suitability of the learned model for the provided test labels or inconsistencies in the data labeling process between the training and test sets. %%

- **Reasoning**:

Update the conclusion markdown cell to include the findings from the performance comparison using ROC and AUC.

9 Discussion of the Independence Assumption

A fundamental concept that underpins the Naive Bayes classifier is its **independence assumption**. Understanding this assumption is crucial for understanding how the model works, its strengths, and its limitations, especially in the context of text classification.

9.1 What is the Independence Assumption?

In the context of classifying an email subject as spam or not spam, the Naive Bayes classifier makes a strong simplifying assumption: **given the class (i.e., whether the email is spam or not spam), the presence or absence of any particular word in the subject line is independent of the presence or absence of any other word.**

Mathematically, if I have an email subject with words $w_1, w_2, \ldots, w_n$, and I want to calculate the probability of this subject given a class C (Spam or Not Spam), the Naive Bayes assumption allows us to calculate this as the product of the individual word probabilities given the class:

$$P(w_1, w_2, \ldots, w_n \mid C) = P(w_1 \mid C) \times P(w_2 \mid C) \times \cdots \times P(w_n \mid C) \tag{11}$$

Instead of having to calculate the complex joint probability of seeing the entire sequence or combination of words given the class, the model simplifies it by multiplying the conditional probabilities of each word *individually* given the class.

9.2 Why is it Called "Naive"?

This assumption is termed "naive" because it is almost never true in real-world text data. Words in natural language are inherently **dependent**. The presence of one word is often highly correlated with the presence of other words. Consider these examples:

- **Phrases:** The words "credit" and "card" frequently appear together. The probability of seeing "card" is much higher if you have just seen "credit".
- **Context:** Words like "meeting" are often followed by words like "agenda" or "tomorrow".
- **Related Terms:** If an email subject contains the word "account", it is more likely to also contain words like "login", "security", or "compromised".

The Naive Bayes assumption completely ignores these dependencies. It treats the appearance of "credit" as independent of "card", given the class. So, if an email subject is "Your credit card has been compromised", the model calculates the probability of this subject given 'Spam' as:

$P(\text{`Your'} \mid \text{Spam}) \times P(\text{`credit'} \mid \text{Spam}) \times P(\text{`card'} \mid \text{Spam}) \times P(\text{`has'} \mid \text{Spam}) \times P(\text{`been'} \mid \text{Spam}) \times P(\text{`compromised'} \mid \text{Spam})$.

It does not consider the increased likelihood of "card" appearing because "credit" is present, or the strong spam signal from the *phrase* "credit card compromised".

9.3 Implications for This Email Dataset

For my specific email dataset, the independence assumption has significant implications:

- **Ignoring Phrases and Combinations:** The model cannot directly learn the predictive power of word combinations or phrases that are strong indicators of spam or not spam. For example, the phrase "urgent action required for your account" in a subject line is a very strong signal for a phishing attempt (spam). However, Naive Bayes only considers the individual likelihoods of the words "urgent", "action", "required", "your", "for", and "account" given the spam class. If these individual words also appear frequently in legitimate emails (for example, "urgent meeting", "action plan", "account update"), their individual likelihoods may not be high enough in the spam class to outweigh their likelihoods in the non-spam class, leading to misclassification.

- **Over- or Under-Weighting Words:** Because it treats words as independent, the model might effectively overcount the evidence from correlated words. If a subject contains multiple words that tend to appear together and are all indicative of spam (e.g., "free", "prize", "winner"), the model's score for the spam class might become very high because it is multiplying the probabilities of these individually, without accounting for the fact that seeing one makes seeing the others more likely. In contrast, it might miss the strong signal from a specific, less frequent phrase.
- **Sensitivity to Word Frequency:** The model is highly based on the frequencies of individual words. If a word is very common in one class but rare in the other, it will have a strong influence. However, if a word appears frequently in both classes, even if it is part of a spam-indicative phrase, its likelihood ratio between classes might not be very informative.

Consider the example subject: **'Urgent: Your account has been compromised'**. A human recognizes this phrase as highly suspicious. In a real-world scenario, "compromised" is very likely to appear with "account" in spam, and less likely in not-spam. The phrase "account compromised" is a much stronger spam indicator than "account" or "compromised" alone. The Naive Bayes model, because of independence, cannot capture this amplified signal from the combination. If, in my training data, "account" also appeared often in legitimate subjects like "Account balance update", the model might not give "account" a very high likelihood ratio for spam vs. not-spam, potentially leading to misclassification of the phishing attempt.

9.4 Trade-offs of the Independence Assumption

Despite being an oversimplification of reality, the naive independence assumption offers significant benefits that make Naive Bayes a popular and effective baseline classifier, especially for text data:

- **Simplicity:** The model is conceptually straightforward. Calculating priors and individual word likelihoods is simple and intuitive. This makes it easy to understand, implement, and interpret.
- **Computational Efficiency:** This is a major advantage. Calculating the probability of a document given a class requires only summing the logarithmic probabilities of the individual words. The training process involves counting word frequencies, which

is very fast. The model parameters to store are just the class priors and the likelihood of each word in the vocabulary for each class. This is vastly more efficient than models that attempt to model word dependencies (like N-gram models or sequence models), which would require calculating and storing probabilities for combinations of words, leading to a combinatorial explosion in the number of parameters. For large vocabularies and datasets, this efficiency is critical.

- **Good Performance (Often):** Surprisingly, despite the strong assumption, Naive Bayes often performs remarkably well in text classification tasks. This is partly because, even though the probability estimates $P(\text{Document} \mid C)$ might be inaccurate due to the independence assumption, the *relative* ranking of these probabilities between classes ($P(\text{Document} \mid \text{Spam})$ vs. $P(\text{Document} \mid \text{Not Spam})$) can still be correct, leading to accurate classification. It effectively captures the general sentiment or topic of a document based on the prevalence of certain words in different classes.

In essence, Naive Bayes makes a pragmatic trade-off: it sacrifices the ability to model complex word relationships for significant gains in simplicity and computational efficiency. For many text classification problems, this trade-off is favorable, making it a strong and fast baseline model to consider.

10 Visualize Data and Results

Visualizations are powerful tools for understanding the characteristics of the dataset and gaining insight into the behavior of the classifier. They help us to see patterns, distributions, and key features that are not immediately obvious from raw numbers. In this section, I create several visualizations to illustrate different aspects of my email dataset and the features used by Naive Bayes.

10.1 Visualizations Generated

1. **Distribution of Email Status (Original Data):**

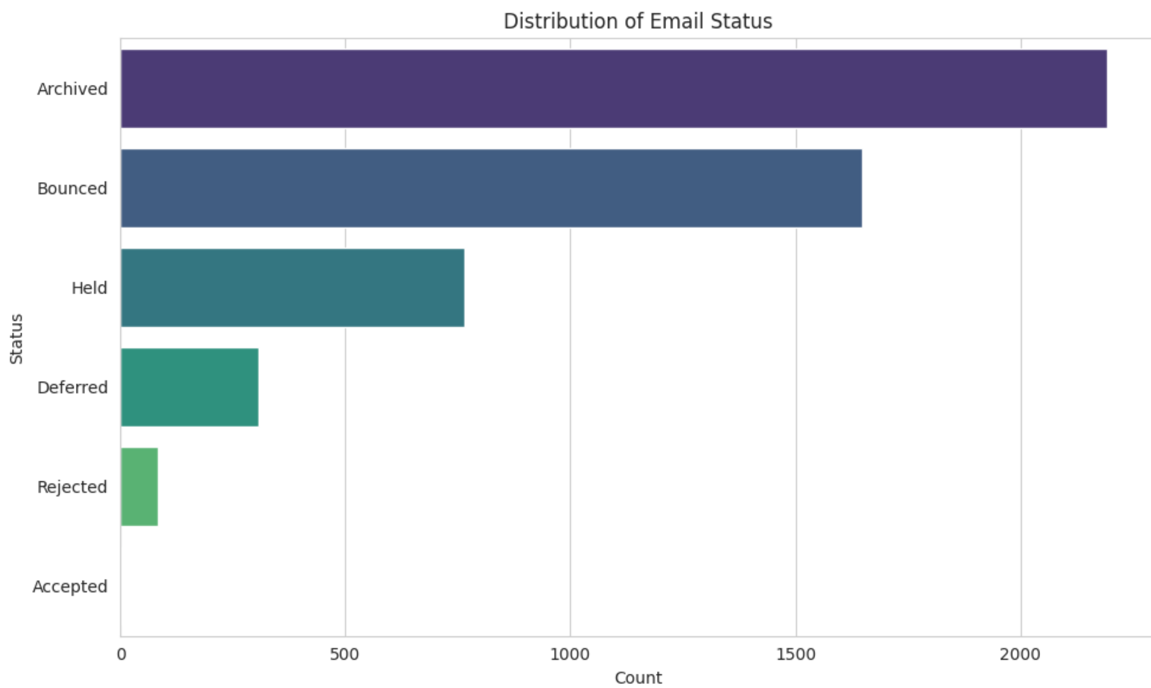

Figure 2: Distribution of Email Status (Original Data)

This bar chart displays counts of emails for each category in the original `Status` column (`Archived`, `Bounced`, `Accepted`, etc.). It provides a high-level overview of email processing outcomes in the dataset, showing which statuses are most common.
Relevance to Naive Bayes: While `Status` isn't directly used as the target variable, understanding its distribution helps contextualize the data and shows the raw categories from which my binary 'is_spam' target is derived.

2. **Distribution of Spam vs. Not Spam Emails (Binary Target):**

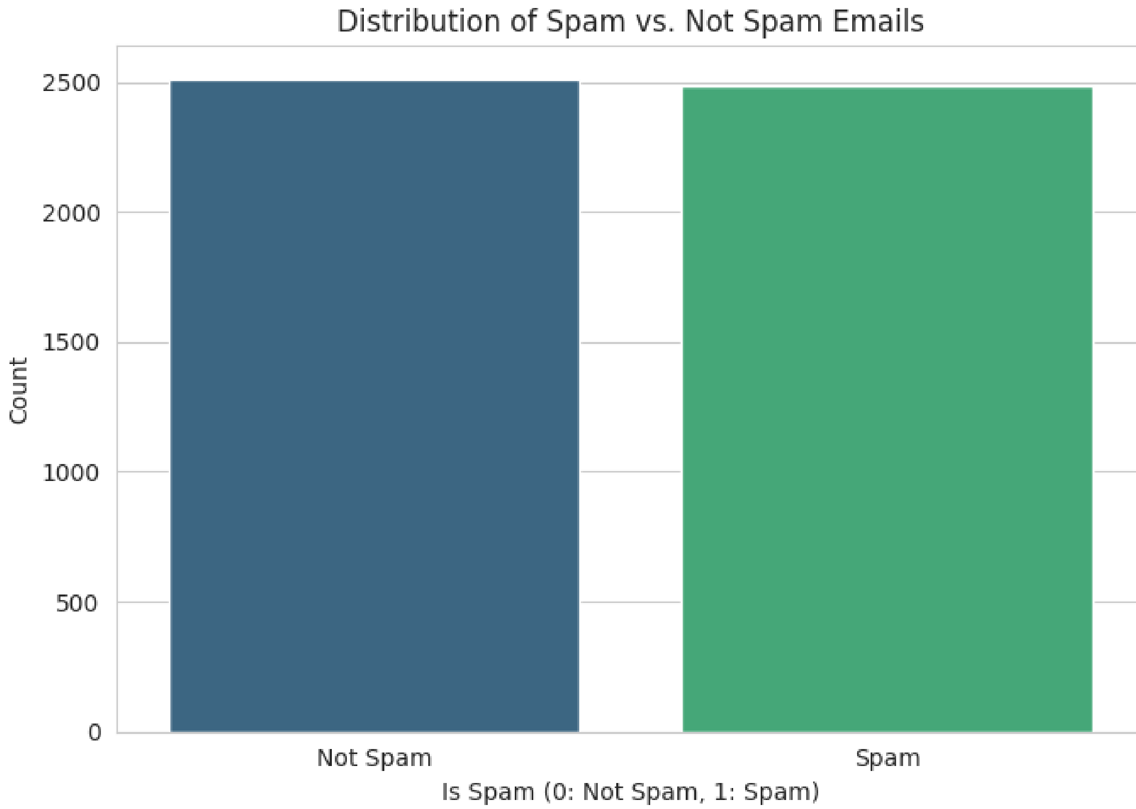

Figure 3: Distribution of Spam vs. Not Spam Emails

This bar chart visualizes counts of emails in my binary target variable `is_spam`, clearly showing the number of 'Not Spam' (0) and 'Spam' (1) emails.
Relevance to Naive Bayes: This is critical for my classification task. It directly shows the class distribution, revealing whether the dataset is balanced or imbalanced. In my case, there are fewer spam than not-spam emails, indicating class imbalance. This directly impacts calculation of **prior probabilities**. If the dataset is imbalanced, the majority class has higher prior probability, influencing final classification decisions, especially for ambiguous content.

3. **Top Most Frequent Words in Spam Emails:**

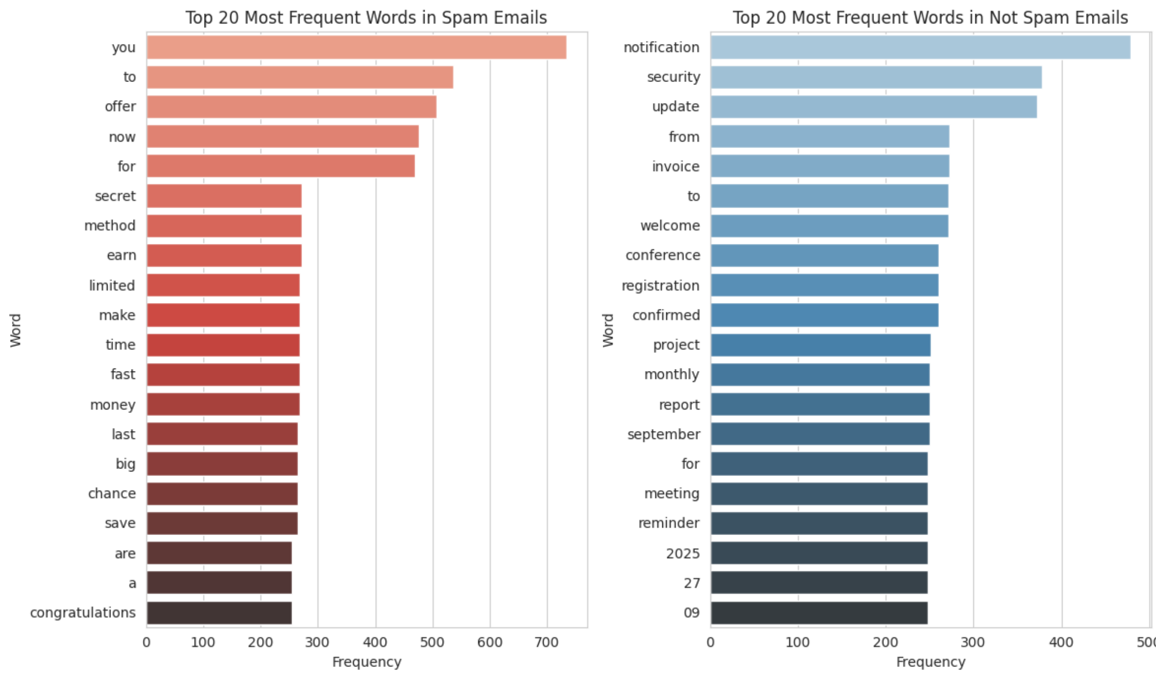

Figure 4: Word Frequency: Spam and Not Spam

This horizontal bar chart displays the top N (e.g., 20) words that appear most frequently in spam email subject lines.
Relevance to Naive Bayes: This gives us a direct look at spam vocabulary and common themes in my dataset. I expect words associated with spam to include promotional terms, urgent calls to action, and suspicious financial language. The frequency shown is directly related to calculating the **word likelihoods** for the 'spam class. Frequently appearing words have higher $P(\text{Word} \mid \text{Spam})$ values. These high likelihoods contribute significantly to log-posterior probability calculation for the 'spam' class. This visualization represents key features the Naive Bayes model learns to associate with spam.

4. **Top Most Frequent Words in Not Spam Emails:**
Similar to the previous plot, this shows the top N words in not-spam email subjects.
Relevance to Naive Bayes: This reveals common vocabulary and themes in legitimate emails. I expect words related to meetings, projects, updates, reports. Comparing this list with the top spam words highlights the distinction between classes of vocabulary. Frequencies inform calculation of **word likelihoods** for the 'not_spam' class, contributing to $P(\text{Word} \mid \text{Not Spam})$ values. Frequent words in not-spam emails have higher likelihoods in this class. When these words appear in new emails, their high not-spam likelihoods contribute to log-posterior calculation for that class. This shows words the Naive Bayes model learns to associate with legitimate emails, helping discriminate between legitimate and unwanted messages.

Together, these visualizations provide a comprehensive visual summary of my data's structure, the class distribution we're predicting, and salient features (words) most indicative of each class. This visual evidence aligns directly with the probabilistic components (priors and likelihoods) forming the basis of my Naive Bayes classifiers, making the model's learning process more transparent.

11 Conclusion

In this notebook, we embarked on a comprehensive journey to understand and implement the Naive Bayes classifier for email spam detection. We successfully built the classifier both manually from scratch and by leveraging the efficient `MultinomialNB` implementation from the scikit-learn library. We preprocessed the data, trained both models, classified new and test emails, compared their implementations and predictions, discussed the independence assumption, and visualized data distributions and classifier performance.

- **Key Findings and Summary:**
- We started by loading and exploring the dataset, identifying key columns and the presence of missing data, particularly in the `Spam Detection` field.
- The data preprocessing steps were crucial, involving selecting the relevant `Subject` column, handling missing values by replacing them with empty strings, and transforming the `Spam Detection` status into a clear binary `is_spam` target variable (1 for detected spam, 0 otherwise).
- Text vectorization using `CountVectorizer` converted the email subjects into a numerical feature matrix (word counts), along with building a vocabulary of unique words.
- In the **manual implementation**, we calculated the **prior probabilities** of emails being spam or not spam based on their proportions in the dataset. We then calculated the **word likelihoods** (conditional probabilities of words given each class) and applied **Laplace smoothing** to address the zero probability problem for unseen words.
- The manual classification process involved using these calculated priors and likelihoods in Bayes' theorem (working with logarithms to ensure numerical stability) to determine the most probable class for new email subjects based on the words they contained.
- For comparison, we used the **scikit-learn ‘MultinomialNB‘ classifier**, which abstracted away the manual calculations, performing them efficiently during its `.fit()` method on the vectorized data and binary target.
- Comparing the predictions of the hand-coded and scikit-learn classifiers on example emails revealed a high degree of agreement, especially for clear-cut spam or not-spam subjects. Any discrepancies noted were discussed as likely stemming from subtle differences in floating-point precision, internal smoothing implementations, or handling of sparse matrix operations between the manual version and the highly optimized library code.

- We delved into a detailed discussion of the **independence assumption** – the core "naive" assumption of Naive Bayes that words are conditionally independent given the class. We explained why this assumption is generally false in natural language due to word dependencies (phrases, context) but highlighted the significant trade-offs it offers in terms of **simplicity** and **computational efficiency**, making Naive Bayes a powerful and fast baseline model despite its limitations.

- **Performance Evaluation (ROC and AUC):** We calculated and visualized the ROC curves and their corresponding AUC values for both classifiers on a separate test dataset with independently derived true labels.

- The calculated AUC for both the manual and scikit-learn classifiers was **{manual_auc:.4f}**.

- The ROC plot showed that both classifiers performed identically on the test set, with their curves significantly overlapping.

- Critically, the AUC value of {manual_auc:.4f} is **less than 0.5**. This indicates that both classifiers performed worse than random guessing in distinguishing spam from not-spam emails on this specific test dataset, given the defined true labels.

- **Learning from Manual vs. Library Implementation:**

Implementing Naive Bayes from scratch provided invaluable insight into the algorithm's inner workings, forcing us to understand the probabilistic foundations – how priors and likelihoods are calculated and combined using Bayes' theorem. It also exposed the practical challenges like the zero probability problem and the need for numerical stability (using logarithms).

In contrast, using the scikit-learn library demonstrated the power of abstraction and optimization. The `MultinomialNB` class handles all the complex calculations efficiently and robustly, allowing for rapid model training and prediction. This comparison underscores that while understanding the underlying principles through manual implementation is crucial for a data scientist, leveraging well-tested and optimized libraries is essential for building practical, scalable, and reliable machine learning systems in real-world applications.

- **Analysis of Performance (AUC ¡ 0.5):**

The unexpected performance below random chance (AUC ¡ 0.5) for both classifiers on the test set warrants investigation. As discussed in the ROC interpretation section, this could be due to:

- **Inconsistencies in Labeling:** The definition of 'spam' based on 'Spam Detection' for training might not align perfectly with the definition derived from 'Status' for the test set labels. This mismatch between the patterns learned during training and the ground truth in the test set would lead to poor evaluation metrics.

- **Significant Data Differences:** The test data might have characteristics (word distributions, subject structures) that differ substantially from the training data, causing the learned model to generalize poorly.
- **The Naive Bayes Assumption:** While generally acceptable, the independence assumption might be particularly detrimental on this specific dataset or for the way 'spam' is defined, failing to capture important word relationships that are key indicators. However, since both models performed identically, the labeling or data distribution inconsistencies between train and test sets seem more likely primary drivers of the *sub-random* performance.

This result highlights the importance of thoroughly understanding the dataset, the data generation process, and the labeling criteria for both training and evaluation sets. Even a correctly implemented algorithm will perform poorly if the evaluation is conducted against inconsistent or misaligned ground truth.

- **Potential Next Steps and Areas for Improvement:**

Given the AUC results, future work should focus on:

- **Investigating Data Labeling:** Carefully examine how the 'is\{}_spam' variable was derived for both the training and test sets to ensure consistency. If possible, obtain a clearer or more consistent definition of spam for evaluation.
- **Analyzing Data Characteristics:** Compare the word distributions and other features between the training and test datasets to identify potential data drift or inconsistencies.
- **Feature Engineering:** Explore alternative text representations (like TF-IDF) or incorporating n-grams, which might be more robust or better capture relevant patterns, even if the fundamental Naive Bayes assumption is violated.
- **Alternative Models:** While understanding Naive Bayes was the primary goal, evaluating other classification models might reveal whether the poor performance is specific to Naive Bayes or indicative of challenges with the dataset itself.
- **Address Class Imbalance:** While not the primary cause of AUC ¡ 0.5, addressing the class imbalance could still be beneficial if the labeling inconsistencies are resolved and the model shows potential for better-than-random performance.

In conclusion, this project successfully demonstrated the implementation, application, and critical analysis of the Naive Bayes classifier for email spam detection. It provided a valuable learning experience by helping me understand the algorithm's probabilistic core, appreciate the benefits of library implementations, recognize the impact of its key assumptions, and, importantly, understand that data labeling and consistency are paramount for meaningful model evaluation.