# ****SPAM vs HAM****

**EMAIL CLASSIFICATION**

The goal of this project was to build a machine learning model that can automatically classify emails as **spam**, unwanted or malicious messages or **ham**, normal, useful emails. Spam emails often include scams, phishing links, or ads that waste time and can be risky for users.

By training a **supervised learning algorithm** on labeled email data, the **machine learns which words and patterns** are more common in spam compared to normal emails. For example, words like “**link”, “offer“ or “price”** often appear in spam, while neutral or conversational language appears in ham messages. Once trained, the model can then predict whether a new email is spam or not.

A white background with red text

AI-generated content may be incorrect.A white background with blue text

AI-generated content may be incorrect.

The project followed a **modular workflow in Python**, using the following main libraries and tools

* **pandas** | for reading and cleaning the datasets
* **scikit-learn** | for vectorization, TF-IDF, model training, and evaluation
* **MLflow** | for tracking experiments and comparing model results
* **matplotlib** and **seaborn** | for creating plots such as the ROC Curve, Precision–Recall Curve, and Confusion Matrix

The main algorithms tested were

A. **Multinomial Naïve Bayes**  
B. **Logistic Regression**  
C. **Linear Support Vector Machine | SVM**

The dataset was originally taken from **Kaggle’s “Spam or Not Spam” dataset** by *Hakan Ozler*, and I later added extra spam examples from another local dataset shared by a colleague to correct class imbalance. The main evaluation metrics were **Accuracy** and **F1-score**, which measure how well the model detects spam without misclassifying normal emails.

**METHODOLOGY**

A screenshot of a computer screen

AI-generated content may be incorrect.At first, I trained a quick baseline model using **Multinomial Naïve Bayes** on the original Kaggle dataset. The results looked good at first, but I noticed the data was **unbalanced** due to low F1 score. There were many more ham emails than spam. Because of that, the model showed high accuracy but failed to correctly identify many spam messages.

To solve this, I added a **second dataset shared by a colleague**, which contained more spam examples. Combining the two files was the hardest step. Each dataset had a different structure : some had no headers, others mixed labels and text in one column, and several used different separators or encodings.

To fix this, I wrote a set of small Python scripts

* **inspect\_data.py** | to explore columns, find errors, and check file structures
* **merge\_data.py** | to combine the two datasets into one master file
* **drop.py** | to remove empty, duplicate or corrupted rows
* **transform\_data.py** | to clean and standardize the text | uppercase, remove symbols, reduce spaces and fixing labels

After cleaning, I renamed the columns to **label** and **email**, made sure all labels were consistent : “spam” and “ham”, and saved the final balanced dataset as **MERGE\_DATA.csv** inside data/final/.

**A graph of ham vs spam

AI-generated content may be incorrect.A blue and green squares

AI-generated content may be incorrect.**

**VECTORIZATION**

I converted the cleaned emails into numbers using TF-IDF (Term Frequency–Inverse Document Frequency) so the machine could read and process them. This method highlights important words while downplaying very common ones like “THE” or “AND.”

The vectorizer() function in vectorize.py builds the TF-IDF model based on parameters defined in the YAML file, including n-gram range, The TF-IDF vectorizer used **n-grams with unigram\_min = 1** and **unigram\_max = 2**, so it captured both single words and two-word combinations from the emails., min\_df, and max\_df.

* **min\_df = 2** ignores words that appear in fewer than 2 emails | too rare to be useful.
* **max\_df = 0.9** ignores words that appear in more than 90% of the emails | too common to help classification.

The the\_vectorize\_fit() function then fits the vectorizer on the training data and applies the same transformation to the test data, ensuring both sets share the same vocabulary and weights. The fitted vectorizer was finally saved as **outputs/vectorizer.pkl** for reuse.

A black background with white dots

AI-generated content may be incorrect.

**MODEL TRAINING AND EVALUATION**

After vectorization, I retrained three models : **Multinomial Naïve Bayes**, **Logistic Regression**, and **Linear SVM** using the merged dataset. Each model was built in its own Python file : **model\_bayes.py, model\_logistic.py and model\_linear.py.**

| THE MODEL | DESCRIPTION | ADVANTAGE | |
| --- | --- | --- | --- |
| Multinomial Naïve Bayes | Uses word frequencies and probabilities | | Fast and great baseline |
| Logistic Regression | Linear model with weighted coefficients | Balanced performance | |
| Linear SVM | Finds best separating boundary | Strong on sparse text data | |

Each script defines two main functions:

* **the\_model()** | builds and configures the model using parameters from params.yaml
* **fit\_and\_predict()** | trains the model on the training data and predicts labels on the test set

For example, the **Naïve Bayes** model in **model\_bayes.py**

**A computer code on a black background

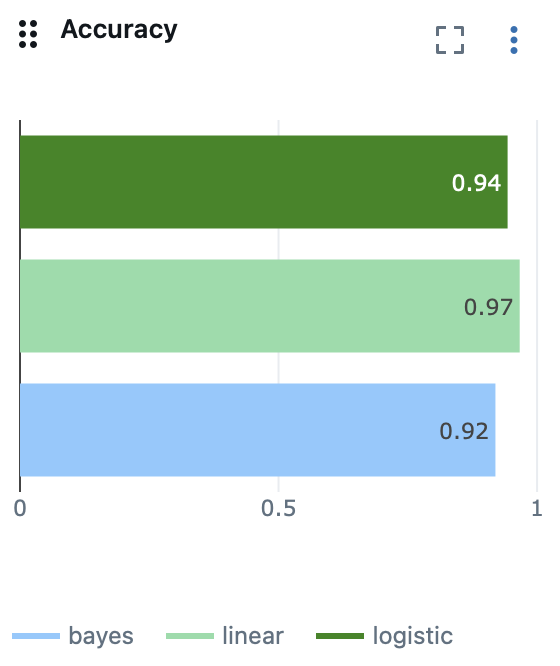
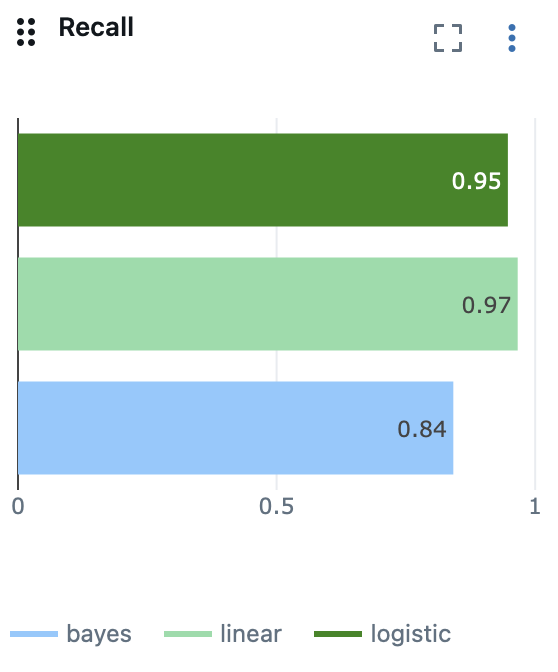
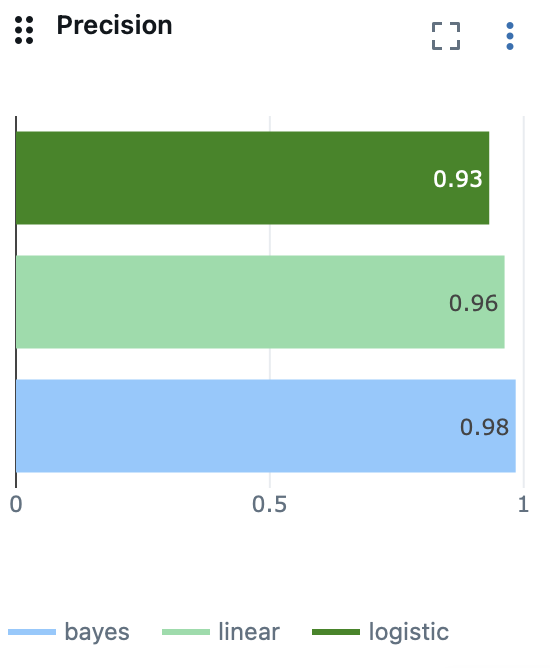
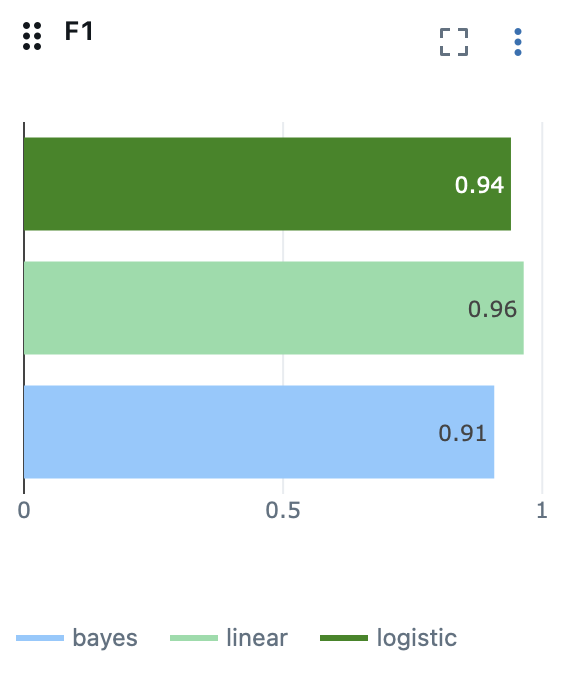
AI-generated content may be incorrect.**

**A computer screen with text

AI-generated content may be incorrect.**

**HYPERPARAMETER TUNING AND TRACKING**

I used **MLflow** to log the experiments, including model parameters, TF-IDF settings, and evaluation metrics such as Accuracy, F1, Precision, Recall and training time. All model results were then visualized in MLflow and exported as plots for easier comparison.



A graph of a line

AI-generated content may be incorrect.

The model with the highest **F1 score** was selected as the best overall, since it balances false positives and false negatives.

**Linear SVM** achieved the highest and most consistent F1-score with a strong precision–recall balance. **Naïve Bayes** had the highest precision but much lower recall, meaning it missed more spam messages, while **Logistic Regression** performed solidly but slightly below Linear SVM.

**CONFUSION MATRIX AND VISUALIZATION**

I plotted confusion matrices to see exactly how each model performed, how many spam and ham messages were correctly identified, and where errors happened. This gave a clear visual view of the trade-offs between models.

The confusion matrix represents the results of the **Linear SVM model**.

A blue squares with white text

AI-generated content may be incorrect.

**True Positives = 714 |** Spam emails correctly identified as spam.

**True Negatives = 811 |** Ham emails correctly identified as normal.

**False Positives = 28 |** Normal emails incorrectly flagged as spam.

**False Negatives = 25 |** Spam emails missed and classified as normal.

This shows that the Linear model accurately classified most emails, with only a few misclassifications. The small number of false positives and false negatives confirms its strong precision–recall balance.

**THE RESULTS**

| Model | Accuracy | F1-Score | Precision | Recall |
| --- | --- | --- | --- | --- |
| Linear SVM | 0.9664 | 0.9642 | 0.9623 | 0.9662 |
| Logistic Regression | 0.9430 | 0.9396 | 0.9321 | 0.9472 |
| Multinomial Naïve Bayes | 0.9195 | 0.9074 | 0.9842 | 0.8417 |

**REFLECTION**

The project began with the **Multinomial Naïve Bayes model**, which achieved an **F1 score** of only **0.38** on the original unbalanced Kaggle dataset. This showed that the main issue was not the algorithm itself but the data quality and imbalance. To address this, I merged an additional spam dataset and focused on cleaning and restructuring the files.

The **biggest challenge** was cleaning and merging the data, since the files had inconsistent formats, missing or improper headers, extra symbols inside the email text and mixed labels. It took several small Python scripts and multiple verification steps to make everything align correctly. Once the datasets were balanced, the same Naïve Bayes model’s F1 score increased to **0.90**, confirming how much **properly prepared data improves performance.**

Afterward, retraining all three models led to strong and consistent results, proving that **clean, balanced data often matters more than complex algorithms.**

In the future, I’d like to experiment with **higher n-grams, up to 3** and explore **deep learning approaches** to better capture small, less obvious relationships between words and how they connect within a sentence.