

Knowledge Discovery and Data Mining Finals Report

1st 522H0036 - Luong Canh Phong
Faculty of Information Technology
Ton Duc Thang University
Ho Chi Minh City, Vietnam
522H0036@student.tdtu.edu.vn

2nd 520H0341 - Nguyen Thai Bao
Faculty of Information Technology
Ton Duc Thang University
Ho Chi Minh City, Vietnam
520H0341@student.tdtu.edu.vn

3rd 522H0030 - Le Tan Huy
Faculty of Information Technology
Ton Duc Thang University
Ho Chi Minh City, Vietnam
522H0030@student.tdtu.edu.vn

4th 522H0008 - Dao Minh Phuc
Faculty of Information Technology
Ton Duc Thang University
Ho Chi Minh City, Vietnam
522H0008@student.tdtu.edu.vn

5th 522H0136 - Nguyen Nhat Phuong Anh
Faculty of Information Technology
Ton Duc Thang University
Ho Chi Minh City, Vietnam
5220136@student.tdtu.edu.vn

Abstract—Spam emails are a common problem seen on the internet as it is an annoyance in daily life and a cyber security risks to any sensitive and important data of a person or an organization/business. With the number of spam emails increasing more and more significantly over the past few years, many more algorithms are created and improved in spam detection efficiency. Overall, this paper goes through the basic understanding of spam emails, understanding the necessity of a spam classification algorithm, and learn more about the methodologies, its effectiveness and usefulness when detecting spam emails.

I. INTRODUCTION

Since the birth of the Internet, spam email has been a common occurrence. Along with the rapid growth and widespread of the Internet, the frequency has been increasing significantly, especially over the past decade. In addition to being nuisances, a waste of time and email storage, spam emails can be sent with malicious intent of stealing information, hijacking devices by storing malware within the content of the email itself. And with the nature of email spam being sent by botnets, it isn't easy to avoid the situation due to a new bot can be easily created in case another one got blocked or banned on the site. A common way how most platforms (such as Gmail, Yahoo!, Outlook) handle these spams is to develop a Machine Learning (ML) model to detect and get rid of the spam emails, lowering the number of spams getting into the inbox.

II. IMPORTANCE OF SPAM CLASSIFICATION

To understand why spam classification is important to our lives, we must first understand the spam emails and its impact on daily life and businesses.

A. Different Types of Spam Emails

There are various forms of spam, sent with different intentions and purposes. But they are commonly grouped into:

- Phishing Emails
- Email Spoofing
- Technical support scams

- Current event scams
- Marketing/advertising email
- Malware scam

B. Problems with Spam Emails

According to statistics report in 2023, 160 billion spam emails are sent every day, which is 46% of the 347 billion emails sent daily. Out of which, the most common type being marketing/advertising emails, which take up around 36%, followed up with promotional of adult content, around 31.7% of total spam emails. Despite scam and fraudulent emails being the least common type, over 70% of them are phishing emails, which is still over 6 billion phishing emails are being sent to users daily.

A single spam email carbon emission is almost 0.03g of Carbon Dioxide Equivalent (CO₂e). With the amount of spam being sent daily, it can easily get nearly 5 tonnes of CO₂e being released every day. Additionally, two-thirds of spam receivers have been reported to have their mental health affected due to the number of spam or phishing scams.

For businesses, this spam can be sent as a way to get businesses to invest in nonexistent organizations under the guise of an investment and promise a high payoff. For individuals, it would be under the form of bitcoin investment or for a charitable cause. Once the money is received, the sender would delete all traces and block the recipient contact.

III. METHODOLOGIES FOR SPAM CLASSIFICATION

Many methods to prevent spam are applied; a commonly used method is using Machine Learning models like Random Forest (RF), Support Vector Machine (SVM), Logistic Regression (LR), Naive Bayes (NB) or Deep Learning models such as Artificial Neural Network (ANN), (Explantation of used algorithms)

IV. RECENT ADVANCES IN SPAM CLASSIFICATION (TBA)

V. VISUALIZING THE DATA

(TBA)

VI. PREPROCESSING THE DATA

This section focuses on preparing the dataset for machine learning by cleaning and transforming the text data. It includes removing unnecessary stop words, tokenizing the emails, and vectorizing the emails.

A. Removing unnecessary stopwords

stop_words = set(stopwords.words('english')): This line creates a set called stop_words. It uses the nltk library's stopwords module to get a list of common English words like "the," "a," "is," etc., which are often removed from text data because they usually carry little meaningful information. These words are converted into a set for efficient lookup during the removal process.

punctuation = set(string.punctuation): This line creates a set called punctuation. It uses the string library's punctuation attribute to get a list of punctuation marks. Similar to stop_words, these punctuation marks are converted into a set for efficient removal.

def remove_stopwords(text): This line defines a function named remove_stopwords that takes a single argument, text, representing the input text to be processed.

return ' '.join([word for word in text.split() if word not in stop_words and word not in punctuation]): This line is the core of the function. It uses a list comprehension to filter out stopwords and punctuation from the input text:

- **text.split():** This splits the input text into a list of individual words.
- **if word not in stop_words and word not in punctuation:** This condition checks if a word is not in either the stop_words set or the punctuation set.
- **[word for ...if ...]:** This creates a new list containing only the words that passed the condition.
- **' '.join(...):** This joins the words back together into a single string, separated by spaces.

Finally, the function returns this processed string (with stopwords and punctuation removed). This step helps to reduce noise and focus on more meaningful words for further analysis.

B. Tokenizing the emails

Each email was then tokenized, which involves splitting the text into individual words or tokens.

import spacy: This line imports the spacy library, a popular Python library for NLP tasks.

nlp = spacy.load('en_core_web_sm'): This line loads a specific language model within spaCy called en_core_web_sm. This model is trained on a large English text dataset and is used for various NLP operations, including schematization

C. Defining and Applying the Lemmatization Function

The next step involves reducing words to their base or root form, known as lemmatization, using the spaCy library. This helps standardize words for better analysis.

def lemmatize(email): This line defines a function named lemmatize that takes a single argument, "email", representing the input email text to be processed.

doc = nlp(email): Inside the function, this line uses the previously loaded spaCy language model ("nlp") to process the input "email" text. The result, a spaCy Doc object containing tokens and their linguistic features, is stored in the "doc" variable.

return ' '.join(token.lemma_ for token in doc): This line performs the lemmatization and returns the result.

- **"token.lemma_ for token in doc":** This iterates through each token (word) in the processed "doc" object and extracts its lemma (base form). For example, "running" becomes "run," "studies" become "study."
- **" ' '.join(...)":** This joins the extracted lemmas back together into a single string, separated by spaces.

The function returns this lemmatized string.

df['email'] = df['email'].apply(lemmatize): This line applies the "lemmatize" function to each entry in the "email" column of the pandas DataFrame named "df". The original email text in the column is replaced by its lemmatized version.

This step is crucial for further analysis, as it allows the model to treat different forms of the same word (e.g., "run," "running," "ran") as a single concept, improving pattern recognition.

D. Removing duplicate words

This code snippet aims to remove duplicate words from each email string within the "email" column of the DataFrame "df", while importantly maintaining the original order of the remaining words.

df['email'] = ...: This part indicates that the result of the operation described on the right side will be assigned back to the "email" column of the "df" DataFrame, overwriting the previous content.

df['email'].apply(...): The "apply()" method is used here to apply a specific function to each element (each email string) in the "email" column.

lambda x: ...: This defines an anonymous inline function (a lambda function). The function takes one argument, "x", which represents an individual email string from the column during the "apply" operation.

x.split(): This part of the lambda function splits the input email string "x" into a list of individual words, using spaces as the default delimiter.

set(x.split()): This converts the list of words obtained from "x.split()" into a Python set. A key property of sets is that they only store unique elements, so this step automatically removes any duplicate words present in the list.

sorted(..., key=x.split().index): This is the core part for preserving order.

- “sorted(...)”: This function sorts the elements of the set (which contains only unique words).
- “key=x.split().index”: This is the crucial argument. It tells “sorted” not to sort alphabetically, but based on the index (position) of each unique word in the *original* list created by “x.split()”. This effectively restores the initial sequence of words.

‘**.join(...)**’: Finally, this joins the sorted (by original position), unique words back into a single string, with words separated by spaces.

Removing duplicate words can be beneficial in text processing tasks as it helps reduce redundancy and noise in the data, potentially leading to improved model performance. Maintaining the original word order is important for preserving the contextual meaning of the email.

E. Vectorizing

To represent the processed email text numerically, a technique called vectorization was employed. Specifically, TF-IDF (Term Frequency-Inverse Document Frequency) vectorization was used to convert the text into a numerical matrix that machine learning models can understand. This step is crucial for training models to recognize patterns and make predictions based on email content.

Creating the Vectorizer

TfidfVectorizer: This line imports or initializes the “TfidfVectorizer” class from the “sklearn.feature_extraction.text” library. This class converts a collection of raw documents to a matrix of TF-IDF features.

min_df=1: This parameter is set during the initialization of “TfidfVectorizer”. It specifies that a word must appear in at least one document (email, in this case) to be considered part of the vocabulary. Setting it to 1 includes all words that appear at least once (unless filtered by other parameters like “max_features”).

max_features=1000: This parameter limits the size of the vocabulary to the 1000 most frequent words across all emails, based on term frequency. This helps manage the dimensionality of the resulting vector space and focuses on potentially more informative terms.

Fitting and Transforming the Data:

fit_transform(df['email']): This method of the “TfidfVectorizer” object is called on the “email” column of the DataFrame. It performs two actions in sequence:

- **fit**: It learns the vocabulary (the 1000 most frequent words meeting the “min_df” criteria) from the provided email data. It also calculates the Inverse Document Frequency (IDF) for each word in the learned vocabulary.
- **transform**: It converts each email in the “df['email']” column into a numerical vector. Each element in the vector corresponds to a word in the vocabulary, and its value is the TF-IDF score for that word in that specific email. The result is typically a sparse matrix (“vector_matrix” in the context description).

Converting to DataFrame:

vector_df = pd.DataFrame(vector_matrix.toarray(), columns=vectorizer.get_feature_names_out()): This line converts the numerical representation into a more readable pandas DataFrame.

- “vector_matrix.toarray()”: The TF-IDF process often returns a sparse matrix (efficient for storing data with many zeros). This method converts it into a standard dense NumPy array.
- “pd.DataFrame(...)”: This creates a pandas DataFrame from the dense array.
- “columns=vectorizer.get_feature_names_out()”: This sets the column names of the new DataFrame (“vector_df”) to be the actual words (features) that the vectorizer learned during the “fit” step.

This DataFrame (“vector_df”) now holds the vectorized data, where each row represents an email, each column represents a unique word (feature), and the cell values are the TF-IDF scores.

Previewing the Result:

vector_df.head(): This line calls the “head()” method on the newly created “vector_df” DataFrame. It displays the first few rows (typically 5) of the DataFrame, allowing for a quick inspection of the vectorized data structure and the TF-IDF values.

The values in the cells represent the TF-IDF score, indicating the calculated importance of each word (column) in each email (row), considering both its frequency in that email and its rarity across all emails. This numerical representation is the final input format used for training machine learning models for tasks like spam detection.

VII. MODELS IN USE

The project employs six different models for email classification, including five traditional machine learning models and one deep learning model. Below is a detailed explanation of each model, its algorithm, implementation, and results.

A. Random Forest Model

Random Forest is a popular machine learning model in the Ensemble Learning family, used for both classification and regression tasks. It consists of a collection of decision trees trained on different subsets of data, with the final prediction aggregated from these trees. The core idea is to combine multiple weak learners (individual decision trees) into a strong learner with higher accuracy and reduced overfitting compared to a single decision tree.

Random Forest relies on two key principles:

- **Bagging (Bootstrap Aggregating)**: Creates multiple data subsets by randomly sampling with replacement from the original dataset.
- **Feature Randomness**: At each split in a decision tree, only a random subset of features is considered.

1) Basic Components:

Decision Tree: The fundamental unit of Random Forest. Each tree is built independently on a data subset.

Forest: A collection of many decision trees (typically hundreds or thousands).

Bootstrap Sample: Each tree is trained on a randomly sampled subset of the original dataset.

Random Feature Subset: At each node, only a random subset of features is evaluated for the best split.

2) Detailed Algorithm:

Here are the specific steps to construct and use a Random Forest:

Step 1: Data Preparation

- Assume an original dataset D with N samples and M features.

Step 2: Create Bootstrap Samples

- Generate T subsets D_1, D_2, \dots, D_T (where T is the number of trees), each of size N , by randomly sampling with replacement from D .
- Due to sampling with replacement, some samples may appear multiple times, while others may not appear (approximately 36.8% of the data is left out, called Out-of-Bag (OOB) data).

Step 3: Build Each Decision Tree

- For each subset D_t :
 - a) Initialize a decision tree: Trees are grown fully without pruning.
 - b) At each node:
 - Randomly select a subset of m features from the total M features ($m < M$). Typically:
 - * For classification: $m = \sqrt{M}$
 - * For regression: $m = M/3$
 - Find the best feature among these m features to split the node, based on criteria like Gini Index, Entropy (classification), or Mean Squared Error (regression).
 - c) Repeat the splitting process until the tree is complete (reaches maximum depth or cannot split further).
- Result: T independent decision trees.

Step 4: Aggregate Results

For classification:

- Each tree predicts a class.
- The final class is the one with the most votes (Majority Voting).

For regression:

- Each tree predicts a numerical value.
- The final value is the average of all predictions.

Step 5: Model Evaluation (Using OOB Error)

- OOB data (not used to train a given tree) is used to test the performance of each tree.

- Aggregate OOB results to estimate the overall accuracy of the Random Forest without a separate test set.

3) Mathematical Formulas:

There are splitting criteria at each node in the tree, and it is different for each task.

For Classification:

- Gini Index:

$$Gini = 1 - \sum_{i=1}^C p_i^2$$

where p_i is the proportion of class i in the node.

- Entropy:

$$Entropy = - \sum_{i=1}^C p_i \log_2 p_i$$

- Final Result:

$$\hat{y} = \text{mode}(\hat{y}_1, \hat{y}_2, \dots, \hat{y}_T)$$

where \hat{y}_t is the prediction from a tree t .

For Regression:

- Splitting criterion: Minimize Mean Squared Error:

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

- Final Result:

$$\hat{y} = \frac{1}{T} \sum_{t=1}^T \hat{y}_t$$

4) Advantages:

- High accuracy due to combining multiple trees.
- Resistance to overfitting thanks to randomness and aggregation.
- Handles large datasets with many features and samples effectively.
- No need for data normalization since it is based on decision trees.
- Provides feature importance measurement based on impurity reduction.

5) Disadvantages:

- Resource-intensive: Requires significant memory and computation for large numbers of trees.
- Less interpretable than a single decision tree.
- Reduced performance with linear data, where linear regression might be more suitable.

B. Support Vector Machine (SVM)

SVM is a supervised machine learning algorithm used for classification and regression, though it is primarily applied to classification tasks. The core idea of SVM is to find the optimal hyperplane that best separates data points of different classes in a high-dimensional space, maximizing the margin between the classes. For cases where data is not linearly separable, SVM uses the kernel trick to transform the data into a higher-dimensional space where a linear boundary can be established.

SVM is known for its robustness and effectiveness in handling high-dimensional datasets and is widely used in tasks like text classification, image recognition, and bioinformatics.

1) Basic Components:

Hyperplane: A decision boundary that separates data points of different classes. In d -dimensional space, a hyperplane is defined by $w^T x + b = 0$, where w is the weight vector, b is the bias, and x is the input vector.

Margin: The distance between the hyperplane and the nearest data point from either class. SVM aims to maximize this margin.

Support Vectors: The data points closest to the hyperplane, which define the margin and are critical to determining the hyperplane's position.

Kernel Function: A function that transforms non-linearly separable data into a higher-dimensional space where a linear boundary can be found (e.g., linear, polynomial, or radial basis function (RBF) kernels).

Regularization Parameter (C): Controls the trade-off between maximizing the margin and minimizing classification errors.

Slack Variables: Allow for some misclassification in soft-margin SVM to handle non-linearly separable data.

2) Detailed Algorithm:

Step 1: Data Preparation

- Assume an original dataset

$$D = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$$

where $x_i \in \mathbb{R}^d$ (feature vector with d dimensions) and $y_i \in \{-1, +1\}$ (binary class labels).

Step 2: Define the Objective

Hard-Margin SVM (Linearly Separable Data):

- Find the hyperplane $w^T x + b = 0$ that separates the classes with the maximum margin.
- The margin is defined as the distance from the hyperplane to the nearest data point, given by $\frac{2}{\|w\|}$, where $\|w\| = \sqrt{w^T w}$.
- Objective: Maximize the margin, i.e., minimize $\frac{1}{2}\|w\|^2$, subject to:

$$y_i(w^T x_i + b) \geq 1, \quad \forall i = 1, \dots, N$$

Soft-Margin SVM (Non-Linearly Separable Data):

- Introduce slack variables $\xi_i \geq 0$ to allow some misclassification.
- Objective: Minimize:

$$\frac{1}{2}\|w\|^2 + C \sum_{i=1}^N \xi_i$$

Subject to:

$$y_i(w^T x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0, \quad \forall i$$

where C is the regularization parameter controlling the trade-off between margin maximization and classification error.

Step 3: Solve the Optimization Problem

The optimization problem is typically solved in its **dual form** using Lagrange multipliers to handle constraints efficiently.

- Dual Problem:

- Introduce Lagrange multipliers $\alpha_i \geq 0$.
- The dual optimization problem is:

Maximize

$$L(\alpha) = \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j (x_i^T x_j)$$

Subject to:

$$\sum_{i=1}^N \alpha_i y_i = 0, \quad 0 \leq \alpha_i \leq C, \quad \forall i$$

- The solution α_i determines the weight vector:

$$w = \sum_{i=1}^N \alpha_i y_i x_i$$

- The bias b is computed using support vectors (where $\alpha_i > 0$).
- Support vectors are the points where $y_i(w^T x_i + b) = 1$ (on the margin) or $\xi_i > 0$ (misclassified or within the margin).

Step 4: Handle Non-Linear Data with Kernels

For non-linearly separable data, map the input data to a higher-dimensional space using a kernel function $K(x_i, x_j)$.

- Common kernels:

- Linear Kernel:

$$K(x_i, x_j) = x_i^T x_j$$

- Polynomial Kernel:

$$K(x_i, x_j) = (x_i^T x_j + c)^d$$

- Radial Basis Function (RBF) Kernel:

$$K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)$$

- The dual problem becomes: Maximize

$$L(\alpha) = \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j K(x_i, x_j)$$

Subject to the same constraints as above.

Step 5: Prediction

For a new input x :

- Compute the decision function:

$$f(x) = \sum_{i \in SV} \alpha_i y_i K(x_i, x) + b$$

where SV is the set of support vectors.

- Predict the class:

$$\hat{y} = \text{sign}(f(x))$$

- For probability estimates (if enabled), use techniques like Platt scaling to convert $f(x)$ into probabilities

3) Advantages:

- Effective in High-Dimensional Spaces: Works well with datasets having many features (e.g., text classification).
- Robust to Outliers: Maximizing the margin focuses on support vectors, ignoring points far from the boundary.
- Flexible with Kernels: The kernel trick allows SVM to handle non-linearly separable data effectively.
- Global Optimization: The convex optimization problem ensures a unique solution.
- Sparse Solution: Only support vectors (a subset of the data) determine the model, making it memory-efficient.

4) Disadvantages:

- Computationally Expensive: Training time scales poorly with large datasets ($O(N^2)$ to $O(N^3)$ for solving the quadratic optimization problem).
- Sensitive to Parameter Tuning: Requires careful tuning of C and kernel parameters (e.g., for RBF kernel).
- Not Interpretable: The resulting hyperplane and support vectors are not as intuitive as decision trees.
- Poor with Noisy Data: Overlapping classes or noisy data can degrade performance.
- Not Ideal for Large Datasets: Due to high computational cost, SVM is less suitable for massive datasets compared to models like Random Forests

C. K-Nearest Neighbors (KNN)

KNN is a simple, non-parametric, and instance-based supervised machine learning algorithm used for both classification and regression. It works by finding the K closest data points (neighbors) to a new input in the feature space and making a prediction based on their labels (for classification) or values (for regression). KNN is often described as a “lazy learning” algorithm because it does not build an explicit model during training; instead, it stores the training data and performs calculations at prediction time.

The core idea is: “A point is likely to belong to the same class (or have a similar value) as its nearest neighbors.”

1) Basic Components:

Training Data: The entire dataset

$$D = (x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)$$

where $x_i \in \mathbb{R}^d$ is a feature vector with d dimensions, and y_i is the label (for classification, e.g., $y_i \in \{0, 1\}$) or value (for regression, e.g., $y_i \in \mathbb{R}$)

Distance Metric: A function to measure the “closeness” between points, typically Euclidean distance.

2) Detailed Algorithm:

Here are the steps to implement and use the KNN algorithm:

Step 1: Data Preparation

- Prepare the dataset

$$D = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$$

where x_i are feature vectors and y_i are labels (classification) or continuous values (regression).

- Normalize or standardize features, as KNN relies on distance calculations, and differing feature scales can skew results.

Step 2: Choose Parameters

- Select K , the number of neighbors to consider.
- Choose a **distance metric** (e.g., Euclidean, Manhattan).
- **For classification**, decide on a voting method (e.g., majority voting or weighted voting).
- **For regression**, decide on an aggregation method (e.g., mean or weighted mean).

Step 3: Prediction

- For a new input x :
 - a) Compute the distance between x and all points x_i in the training set using the chosen distance metric.
 - b) Identify the K nearest neighbors (the K points with the smallest distances).
 - c) **Classification:** Predict the class by majority voting among the K neighbors’ labels.
 - d) **Regression:** Predict the value by averaging the K neighbors’ values (or using a weighted average).
- **Weighted KNN:** Assign weights to neighbors based on their distance (e.g., inverse distance), giving closer neighbors more influence.

3) Mathematical Formulas:

KNN’s mathematics is centered around **distance calculations** and **aggregation** of neighbors’ outputs. Below are the key formulas:

- Euclidean Distance (L2 Norm):

$$d(x, x_i) = \sqrt{\sum_{j=1}^d (x_j - x_{i,j})^2}$$

where x_j and $x_{i,j}$ are the j -th features of points x and x_i , and d is the number of features.

- Manhattan Distance (L1 Norm):

$$d(x, x_i) = \sum_{j=1}^d |x_j - x_{i,j}|$$

- Minkowski Distance (Generalization):

$$d(x, x_i) = \left(\sum_{j=1}^d |x_j - x_{i,j}|^p \right)^{1/p}$$

- $p = 2$: Euclidean distance.
- $p = 1$: Manhattan distance.

- Cosine Similarity (for high-dimensional data):

$$d(x, x_i) = 1 - \text{cosine similarity} = 1 - \frac{x \cdot x_i}{\|x\| \|x_i\|}$$

- Weighted Distance (if features have different importance):

$$d(x, x_i) = \sqrt{\sum_{j=1}^d w_j (x_j - x_{i,j})^2}$$

where w_j is the weight for feature j .

Classification Prediction

- Majority Voting:

$$\hat{y} = \text{mode}(y_{i1}, y_{i2}, \dots, y_{iK})$$

where y_{i1}, \dots, y_{iK} are the labels of the K nearest neighbors.

- Weighted Voting:

$$\hat{y} = \arg \max_c \sum_{i \in \text{KNN}} w_i I(y_i = c)$$

where:

- $w_i = \frac{1}{d(x, x_i)}$ (inverse distance) or another weighting function.
- $I(y_i = c) = 1$ if $y_i = c$, else 0.
- c is a class label.

Regression Prediction

- Mean:

$$\hat{y} = \frac{1}{K} \sum_{i \in \text{KNN}} y_i$$

- Weighted Mean:

$$\hat{y} = \frac{\sum_{i \in \text{KNN}} w_i y_i}{\sum_{i \in \text{KNN}} w_i}$$

where $w_i = \frac{1}{d(x, x_i)}$ or similar.

Error Metrics

- Classification Error:

$$\text{Error} = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} I(\hat{y}_i \neq y_i)$$

- Regression Error (Mean Squared Error):

$$\text{MSE} = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} (\hat{y}_i - y_i)^2$$

4) Advantages:

- Simple and Intuitive: Easy to understand and implement.
- No Training Phase: No model is built, making it flexible for dynamic datasets.
- Non-Parametric: Makes no assumptions about data distribution, effective for non-linear patterns.
- Versatile: Works for both classification and regression.
- Robust to Multimodal Data: Can handle complex decision boundaries.

5) Disadvantages:

- Computationally Expensive at Prediction Time: Requires calculating distances to all training points ($O(Nd)$ per prediction, where N is the number of samples, d is the number of features).
- Memory-Intensive: Stores the entire training dataset.
- Sensitive to Noise and Outliers: Noisy points can skew predictions.
- Curse of Dimensionality: Performance degrades in high-dimensional spaces due to sparse data.
- Requires Feature Scaling: Distances are sensitive to feature scales.

D. Naive Bayes

Naive Bayes is a family of simple, probabilistic, supervised machine learning algorithms used primarily for classification tasks, though it can be adapted for other purposes. It is based on Bayes' Theorem and assumes that features are conditionally independent given the class label (the "naive" assumption). Despite this simplifying assumption, Naive Bayes performs surprisingly well in many real-world applications, especially in text classification and spam filtering.

The core idea is: Given a set of features, predict the most likely class by calculating probabilities based on prior observations.

1) Basic Components:

- **Training Data:**

$$D = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$$

where $x_i = (x_{i1}, x_{i2}, \dots, x_{id}) \in \mathbb{R}^d$ is a feature vector with d dimensions, and $y_i \in \{C_1, C_2, \dots, C_k\}$ is a class label from k classes.

- **Bayes' Theorem:** The foundation for calculating probabilities of classes given features.
- **Conditional Independence Assumption:** Assumes that features $x_{i1}, x_{i2}, \dots, x_{id}$ are independent given the class y_i .
- **Prior Probability:** The probability of each class before observing the features.
- **Likelihood:** The probability of observing the features given a class.
- **Posterior Probability:** The probability of a class given the observed features, which is used for prediction.
- **Smoothing:** A technique (e.g., Laplace smoothing) to handle zero probabilities for unseen feature-class combinations.

2) *Detailed Algorithm:* Here are the steps to implement and use the Naive Bayes algorithm:

Step 1: Data Preparation

- Prepare the dataset D , where each sample has features x_i and a class label y_i .
- For categorical features, Naive Bayes is straightforward. For continuous features, assume a distribution (e.g., Gaussian) or discretize the data.

Step 2: Training (Model Building)

- Estimate Prior Probabilities: Calculate the probability of each class C_j :

$$P(C_j) = \frac{\text{Number of samples with class } C_j}{\text{Total number of samples}}$$

- Estimate Likelihoods: For each feature x_m and class C_j , compute the conditional probability $P(x_m|C_j)$:
 - Categorical Features: Use frequency counts.
 - Continuous Features: Assume a distribution (e.g., Gaussian) and estimate parameters (mean, variance).
 - Apply smoothing (e.g., Laplace smoothing) to avoid zero probabilities.
- Store these probabilities for use during prediction.

Step 3: Prediction

- For a new input $x = (x_1, x_2, \dots, x_d)$:
 - a) Compute the posterior probability for each class C_j using Bayes' Theorem:

$$P(C_j|x) \propto P(C_j) \prod_{m=1}^d P(x_m|C_j)$$

(The proportionality \propto is used because the denominator $P(x)$ is constant across classes.)

- b) Predict the class with the highest posterior probability:

$$\hat{y} = \arg \max_{C_j} P(C_j) \prod_{m=1}^d P(x_m|C_j)$$

- c) Optionally, compute normalized probabilities by dividing by the sum of all class posteriors

3) Mathematical Formulas:

Naive Bayes is grounded in Bayes' Theorem and the conditional independence assumption. Below are the key formulas, explained intuitively.

Bayes' Theorem

For a class C_j and feature vector $x = (x_1, x_2, \dots, x_d)$:

$$P(C_j|x) = \frac{P(C_j)P(x|C_j)}{P(x)}$$

- $P(C_j|x)$: Posterior probability (probability of class C_j given features x).
- $P(C_j)$: Prior probability (probability of class C_j before seeing x).
- $P(x|C_j)$: Likelihood (probability of observing x given class C_j).
- $P(x)$: Evidence (probability of observing x , a normalizing constant).

Since $P(x)$ is the same for all classes, we can ignore it for classification:

$$P(C_j|x) \propto P(C_j)P(x|C_j)$$

Conditional Independence Assumption

Naive Bayes assumes that features are independent given the class:

$$P(x|C_j) = P(x_1, x_2, \dots, x_d|C_j) = \prod_{m=1}^d P(x_m|C_j)$$

Intuition: If you know the class (e.g., “spam email”), the presence of one feature (e.g., the word “free”) doesn’t affect the probability of another feature (e.g., the word “win”). This is often unrealistic but simplifies calculations.

Prior Probability

$$P(C_j) = \frac{\text{Count}(y = C_j)}{N}$$

where $\text{Count}(y = C_j)$ is the number of samples with class C_j , and N is the total number of samples.

Likelihood

- Categorical Features:

$$P(x_m = v|C_j) = \frac{\text{Count}(x_m = v, y = C_j)}{\text{Count}(y = C_j)}$$

where v is a specific value of feature x_m .

- Laplace Smoothing: To avoid zero probabilities:

$$P(x_m = v|C_j) = \frac{\text{Count}(x_m = v, y = C_j) + \alpha}{\text{Count}(y = C_j) + \alpha \cdot |\text{Values}(x_m)|}$$

where α (e.g., 1) is the smoothing parameter, and $|\text{Values}(x_m)|$ is the number of possible values for x_m .

- Continuous Features (Gaussian Naive Bayes): Assume feature x_m follows a Gaussian distribution for class C_j :

$$P(x_m|C_j) = \frac{1}{\sqrt{2\pi\sigma_{mj}^2}} \exp\left(-\frac{(x_m - \mu_{mj})^2}{2\sigma_{mj}^2}\right)$$

where:

- μ_{mj} : Mean of feature x_m for class C_j .
- σ_{mj}^2 : Variance of feature x_m for class C_j .

Error Metrics

- Classification Error:

$$\text{Error} = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} I(\hat{y}_i \neq y_i)$$

- Log Loss (if probabilities are used):

$$\text{Log Loss} = -\frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} \sum_{j=1}^k I(y_i = C_j) \log P(C_j|x_i)$$

4) Advantages:

- Simple and Fast: Easy to implement and computationally efficient, especially for training.
- Effective with Small Datasets: Performs well even with limited data, unlike complex models.
- Handles High-Dimensional Data: Common in text classification (e.g., bag-of-words models).
- Probabilistic Output: Provides probability estimates for each class, useful for decision-making.
- Robust to Irrelevant Features: The independence assumption mitigates the impact of irrelevant features.

5) *Disadvantages:*

- Naive Assumption: The conditional independence assumption is often unrealistic, leading to suboptimal performance when features are correlated.
- Sensitive to Zero Probabilities: Requires smoothing to handle unseen feature-class combinations.
- Poor with Continuous Features: Gaussian assumptions may not fit all data distributions.
- Outperformed by Complex Models: Often less accurate than SVM or Random Forest on complex datasets.
- Imbalanced Data Issues: May favor majority classes without proper adjustments.