🔶 1. Bayesian Analysis – Core Idea

Bayesian analysis is based on **Bayes' Theorem**, which updates the **probability** of a hypothesis as more **evidence** or **data** becomes available.

👉 Bayes' Theorem:

$$P(H|E) = \frac{P(E|H) \cdot P(H)}{P(E)}$$

Where:

- $P(H|E)$ = Posterior probability (after seeing evidence)
- $P(E|H)$ = Likelihood (how likely is the evidence given the hypothesis)
- $P(H)$ = Prior probability (belief before evidence)
- $P(E)$ = Marginal likelihood (total probability of evidence)

- You start with a **belief** (prior).
- You observe **new data** (evidence).
- You **update your belief** to get a new belief (posterior).

✅ Example: Medical Testing

You want to know if someone has a disease, given they tested positive.

- Disease prevalence $P(D)$ = 1%
- Test is 99% accurate:
  - $P(\text{Positive}|D) = 0.99$
  - $P(\text{Positive}|\neg D) = 0.05$

If the person tests positive, what is the probability they actually have the disease?

$$P(D|\text{Positive}) = \frac{0.99 \cdot 0.01}{0.99 \cdot 0.01 + 0.05 \cdot 0.99} \approx 0.166$$

So even though the test is **99% accurate**, the chance of actually having the disease is **only ~16.6%** — because the disease is rare!

Uncertainty Quantification(UQ) helps us understand **how confident we are** in our models' predictions and results.

Domain for UQ:

Domain	Use Case
💉 Healthcare	Diagnosing diseases with confidence levels
🚗 Autonomous Vehicles	Predicting pedestrian movements with safety margins
🌦️ Weather Forecasting	Showing probability of rain rather than binary prediction
📈 Finance	Risk assessment, portfolio optimization
🚀 Engineering	Simulation modeling, aerospace design validation
🤖 Machine Learning	Active learning, model interpretability, safe AI

Naïve Bayes Classifier

The **Naïve Bayes Classifier** is a **supervised machine learning algorithm** based on **Bayes' Theorem**, with the **naïve assumption** that features are **conditionally independent** given the class label.

$$P(C \mid x_1, x_2, ..., x_n) \propto P(C) \cdot \prod_{i=1}^{n} P(x_i \mid C)$$

Symbol	Meaning
C	A class (e.g., "spam" or "not spam")
$x_1, x_2, ..., x_n$	Feature values (e.g., words in a sentence)
$P(C \mid x_1, x_2, ..., x_n)$	Posterior probability: how likely class C is, given features
$P(C)$	Prior probability: how common is class C overall
$P(x_i \mid C)$	Likelihood: how likely is feature x_i, assuming class C
$\prod$	Product over all features i from 1 to n

🧠 Meaning in Simple Words:

To **predict the class** C given a set of features $x_1, x_2, ..., x_n$:

1. **Start with your prior belief** in each class: $P(C)$
2. **Multiply by the likelihood** of each feature given the class: $P(x_1|C) \cdot P(x_2|C) \cdot ... \cdot P(x_n|C)$
3. **Ignore the denominator** $P(x_1, ..., x_n)$ (since it's the same for all classes), so we use **proportionality** ($\propto$)

Then, select the class with the **highest resulting score**.

Example: Spam Detection

Let's say we have:

- **Class**: Spam or Not Spam
- **Features**: Words like "free", "money", "win"

Suppose you're given this email:

Text: "win free money"

You want to calculate:

$$P(\text{spam} \mid \text{win}, \text{free}, \text{money}) \propto P(\text{spam}) \cdot P(\text{win} \mid \text{spam}) \cdot P(\text{free} \mid \text{spam}) \cdot P(\text{money} \mid \text{spam})$$

Then do the same for **not spam**:

$$P(\text{not spam} \mid \text{win}, \text{free}, \text{money}) \propto P(\text{not spam}) \cdot P(\text{win} \mid \text{not spam}) \cdot P(\text{free} \mid \text{not spam}) \cdot P(\text{money} \mid \text{not spam})$$

Then compare the two results and choose the **class with the higher score**.

💡 Why Multiply Probabilities?

The "naïve" assumption is that **each feature is conditionally independent** given the class. So we can multiply their individual probabilities.

This makes the computation **very fast**, even for lots of features (like words in text).

This formula assumes independence:

$$P(x_1, x_2, ..., x_n \mid C) = P(x_1 \mid C) \cdot P(x_2 \mid C) \cdot \cdots \cdot P(x_n \mid C)$$

In reality, features may not be truly independent — but this **still works surprisingly well** in practice, especially for text classification.

Scenario: Medical Diagnosis (Cancer)

Let's use **Bayesian analysis** to determine the probability of having **cancer** given a **positive result from a diagnostic test**.

Given:

- **Prior Probability (P(Cancer))**: The probability of a person having cancer before we even conduct the test. Based on general data, we know 1% of people have cancer.

$$P(\text{Cancer}) = 0.01$$

- **Test Sensitivity (P(Pos | Cancer))**: The probability that the test will be positive given the person has cancer. This is 95% accurate.

$$P(\text{Pos} \mid \text{Cancer}) = 0.95$$

- **Test Specificity (P(Pos | No Cancer))**: The probability that the test will be positive given the person does **not** have cancer. This is a **false positive rate** of 5%.

$$P(\text{Pos} \mid \text{No Cancer}) = 0.05$$

Prior Probability of No Cancer (P(No Cancer)):

$$P(\text{No Cancer}) = 1 - P(\text{Cancer}) = 0.99$$

Step 1: Calculate the Total Probability of a Positive Test (P(Pos))

The total probability of a positive test result is:

$$P(\text{Pos}) = P(\text{Pos} \mid \text{Cancer}) \cdot P(\text{Cancer}) + P(\text{Pos} \mid \text{No Cancer}) \cdot P(\text{No Cancer})$$

Substituting the values:

$$P(\text{Pos}) = (0.95 \times 0.01) + (0.05 \times 0.99) = 0.0095 + 0.0495 = 0.059$$

Step 2: Apply Bayes' Theorem to Find P(Cancer | Pos)

Now, we use Bayes' Theorem to calculate the **posterior probability** that the person actually has cancer given the positive test result:

$$P(\text{Cancer} \mid \text{Pos}) = \frac{P(\text{Pos} \mid \text{Cancer}) \cdot P(\text{Cancer})}{P(\text{Pos})}$$

Substituting the values:

$$P(\text{Cancer} \mid \text{Pos}) = \frac{(0.95 \times 0.01)}{0.059} = \frac{0.0095}{0.059} \approx 0.161$$

Even though the test is 95% sensitive, the probability of having cancer given a positive test result is only about **16.1%**. This relatively low probability is due to the **low prevalence of cancer** (only 1%) and the **false positive rate** (5%).

2. Naïve Bayes Classifier: Email Spam Classification

Now let's use **Naïve Bayes** for classifying **spam** vs. **non-spam** emails based on the presence of specific words.

Scenario: Classifying Spam Emails

Suppose we want to classify whether an email is **spam** or **not spam** based on certain words in the email.

Given:

- **Prior Probabilities**:
  - P(Spam) = 0.4 (40% of emails are spam)
  - P(Not Spam) = 0.6 (60% of emails are not spam)
- **Likelihoods (Word Probabilities Given Class)**:
  - **For Spam**:
    - P(free | Spam) = 0.8
    - P(money | Spam) = 0.7

- **For Not Spam**:
  - P(free | Not Spam) = 0.1
  - P(money | Not Spam) = 0.05

Step 1: Apply Naïve Bayes Formula to Calculate Posteriors

Let's say we have an email that contains the words **"free"** and **"money"**. We want to calculate the probability that the email is **spam** vs. **not spam**.

For Spam:

P(Spam | free, money) ∝ P(Spam) * P(free | Spam) * P(money | Spam)

Substituting the values:

P(Spam | free, money) ∝ 0.4 * 0.8 * 0.7 = 0.224

For Not Spam:

P(Not Spam | free, money) ∝ P(Not Spam) * P(free | Not Spam) * P(money | Not Spam)

Substituting the values:

P(Not Spam | free, money) ∝ 0.6 * 0.1 * 0.05 = 0.003

Step 2: Compare the Posteriors

Now, we compare the probabilities:

- **Spam**: 0.224
- **Not Spam**: 0.003

Since **0.224 > 0.003**, we classify the email as **spam**.

Linear Discriminant Analysis (LDA)

Linear Discriminant Analysis (LDA) is a statistical technique used for classification and dimensionality reduction. It is a supervised learning algorithm that works well when the classes (groups) are linearly separable.

Key Concepts of LDA

1. **Discriminant Functions**: LDA seeks to find a linear combination of features that best separates two or more classes. **The goal is to maximize the** separation **between classes while minimizing the variance within each class.**

- **Maximizing the distance (separation) between the means of different classes** – so classes are far apart.
- **Minimizing the spread (variance) within each class** – so each class is tightly grouped together

2. **Class Separation**: LDA finds the axes that maximize the distance between the means of the classes, while minimizing the spread (variance) of each class.
3. **Assumptions in LDA**:
    - The data from each class is normally distributed (Gaussian).
    - All classes have the same **covariance matrix** (homoscedasticity).
    - The relationship between the features and the classes is linear.
4. **Objective**: LDA computes a set of axes (discriminants) that represent the directions in which the class separation is maximized.

Steps in LDA

1. **Compute the Mean Vectors**: For each class, calculate the mean vector of the features.
2. **Compute the Scatter Matrices**:
    - **Within-class scatter matrix (Sw)**: Measures how much the samples of each class vary around their own class mean.
    - **Between-class scatter matrix (Sb)**: Measures how much the class means differ from the overall mean.
3. **Compute the Eigenvectors and Eigenvalues**: By solving the generalized eigenvalue problem for the matrix `inv(Sw) * Sb`, we obtain the eigenvectors and eigenvalues, which represent the directions (axes) that maximize class separation.
4. **Select the Most Significant Eigenvectors**: Based on the eigenvalues, select the top eigenvectors (those with the largest eigenvalues) to form the new lower-dimensional space.
5. **Project the Data**: The data is then projected onto these eigenvectors to create a lower-dimensional representation of the original data, maximizing the class separation.

$J(w) = (w^T \cdot Sb \cdot w) / (w^T \cdot Sw \cdot w)$

This is the **objective function** for **Linear Discriminant Analysis (LDA)**.

It is used to find the best **projection vector** `w` that **maximizes class separability**.

1. w

- A **projection vector** (also called a weight vector).
- This is what we are trying to **find**.
- Once we find the best `w`, we project the high-dimensional data onto this direction.

2. w^T

- The **transpose** of `w`.
- So if `w` is a column vector, w^T is a row vector.

3. `Sb` (Between-class scatter matrix)

- Measures **how far apart** the **means** of different classes are.
- The more separated the class centers are, the better.
- High `Sb` ⇒ classes are well separated.

4. `Sw` (Within-class scatter matrix)

- Measures the **spread** of data **within each class**.
- The less spread (more compact), the better.
- Low `Sw` ⇒ each class is tightly grouped.

- **Maximize J(w)**:
  - Maximize the **numerator** → increase the separation between classes.
  - Minimize the **denominator** → reduce the spread within each class.

- find the direction `w` that gives **maximum separation and minimum overlap**.

Term	Represents	Goal
$w^T \cdot Sb \cdot w$	Separation between class centers	Maximize
$w^T \cdot Sw \cdot w$	Spread within each class	Minimize
`J(w)`	Ratio of the two	Maximize

K-Nearest Neighbors (K-NN) Algorithm?

K-NN is a **non-parametric, lazy learning** algorithm used for **classification and regression**.

- **"Non-parametric"**: It doesn't make assumptions about the data distribution.
- **"Lazy"**: It doesn't learn a model during training—it just stores the data and does all computations during prediction.

K-NN Work?

1. **Choose a value for `k`** (the number of neighbors to consider).
2. For a **new data point**:

    - Compute the **distance** (usually Euclidean) to all other points in the training data.
    - **Select the `k` closest data points** (neighbors).
3. **For classification**:
    - Assign the class that is **most common** among the `k` neighbors.
4. **For regression**:
    - Take the **average** of the `k` nearest neighbors' values.

Let's say you want to classify a fruit based on weight and color.

You already have a dataset of fruits (apple, banana, orange) with known weights and colors.

1. You get a new fruit with weight = 150g and color = "yellow".
2. You calculate its distance to every known fruit in your dataset.
3. Choose `k = 3`.
4. If 2 of the 3 nearest fruits are bananas, the new fruit is classified as a **banana**.

Support Vector Machines (SVM)

Support Vector Machines (SVM) are a powerful supervised machine learning algorithm used primarily for **classification** tasks, though they can also be used for regression.

SVMs are particularly effective for high-dimensional spaces and situations where the number of features is greater than the number of samples.

The basic idea behind **SVM** is to find a **hyperplane** that separates different classes in the dataset.

The hyperplane divides the feature space into two halves, each belonging to a different class.

The main goal of SVM is to find the hyperplane that maximizes the **margin** between the two classes. The margin is the distance between the hyperplane and the closest data point from either class

(these closest points are called **support vectors**).

- **Hyperplane**:
    - A **hyperplane** in a 2D space is just a line that divides the space into two regions. In higher dimensions, it is a plane or hyperplane.
- **Support Vectors**:

- These are the data points closest to the hyperplane. These points are the most critical in determining the position and orientation of the hyperplane.

- **Maximizing the Margin**:
  - SVM aims to maximize the **margin**—the distance between the hyperplane and the support vectors. A larger margin generally means better generalization and fewer errors on unseen data.

The general equation for the hyperplane is given by:

$$\mathbf{w}^T\mathbf{x} + b = 0$$

Where:

- **w** is the weight vector perpendicular to the hyperplane.
- **x** is a data point (feature vector).
- **b** is the bias term, which shifts the hyperplane.

Linear Classification: Linear SVM

Nonlinear Classification : Polynomial and Radial Basis function Kernel (RBF Kernel)