**🔹 1. Training Data**

* **Purpose**: Used to **train** the model—i.e., to help it learn patterns and relationships in the data.
* **What happens**: The model adjusts its internal weights or parameters based on this data.
* **Example**: In supervised learning, the model sees both features (inputs) and labels (targets).

**🔹 2. Validation Data**

* **Purpose**: Used to **tune** hyperparameters and evaluate the model during training, but **not used for training** itself.
* **What happens**:
  + Used to monitor performance after each training epoch (especially in deep learning).
  + Helps avoid **overfitting** by checking if the model is generalizing well to unseen data.
* **Optional but important**: Especially used in cross-validation or when you want early stopping.
* **Never used** in final model testing.

**🔹 3. Testing Data**

* **Purpose**: Used to **evaluate the final performance** of the model after training and tuning are complete.
* **What happens**:
  + It's like a final exam—only used once.
  + Must be **completely unseen** during training and validation.
* **Used to report**: accuracy, precision, recall, F1 score, etc.

If you're working with data where labels are genuinely unknown and can never be known, **you can't use supervised evaluation metrics** like accuracy or F1. Instead, you'd need:

* **Unsupervised methods** (like clustering metrics),
* Or **human-in-the-loop validation**.

### Bottom line:

* **If labels exist or can be created**, supervised learning is often **preferred** because it's usually **more accurate**.
* **If labels don’t exist**, unsupervised (or semi-supervised/weakly-supervised) methods are used.

**WORD TOKENIZE**

**It is a python function that splits a given sentence into words using the NLTK library. It returns a list of words separated by comma. It also assumes punctuation as a token.**

### ✅ So, the point of pos\_tag is:

To **improve the quality of lemmatization** by telling the lemmatizer what kind of word it's dealing with — verb, noun, adjective, etc.

Even though we don’t keep the tags in the final result, we **use them in the background** to get better, cleaner base words.

### 🧠 Bag of Words (BoW) — Explained Simply

**Bag of Words** is a text representation technique used in Natural Language Processing (NLP) and machine learning. It **converts text into numerical features** by focusing only on **word occurrence**, ignoring grammar and word order.

### 🧱 How It Works:

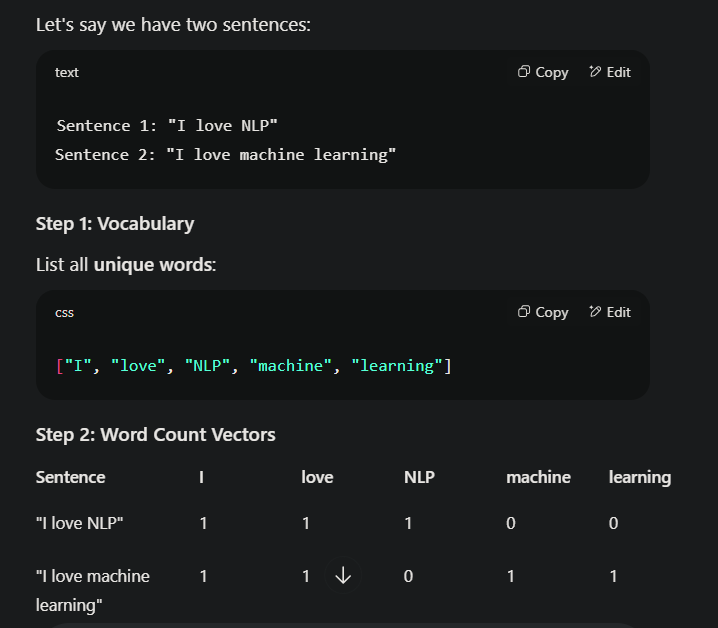
1. **Build a vocabulary** of all unique words in your dataset.
2. **Count** how many times each word appears in each document.
3. Represent each document as a **vector of word counts**.

### 🧾 Pros:

* Simple and easy to implement.
* Works well with small datasets.

### ⚠️ Cons:

* **Ignores word order** (semantics).
* Can create very **sparse vectors** (many 0s) in large vocabularies.
* Doesn’t capture meaning or context (e.g., "good" and "great" are treated as different words with no relation).

****

## 📊 What is **Class Imbalance**?

In a classification problem, class imbalance occurs when **some classes have many more samples than others**.

### Example:

| **Label** | **Count** |
| --- | --- |
| Class 0 | 900 |
| Class 1 | 100 |

Here, Class 0 dominates. A model trained on this data might just **predict everything as Class 0** and still achieve 90% accuracy — but that’s misleading.

## ⚠️ Why is Class Imbalance a Problem?

* The model **learns to favor** the majority class.
* It leads to **poor recall or precision** for the minority class (which is often more important — e.g., fraud detection, disease diagnosis).
* **Accuracy becomes a bad metric.**

### 🎯 Imagine This:

You're building a system to **detect spam emails**.

Your model will sometimes:

* ✅ **Correctly identify spam** (Good!)
* ❌ **Miss some spam** (Bad – spam goes to inbox)
* ❌ **Wrongly mark good emails as spam** (Bad – your boss’s email ends up in junk!)

Now, let’s look at how we evaluate your system:

## 📌 1. **Precision** — "How often is it right when it says it's spam?"

Out of all the emails **you marked as spam**, how many were **actually spam**?

### ✅ High Precision:

Your model is **very careful** — only calls something spam when it's **very sure**.

* Great when **false alarms are costly**.
* Example: You don’t want your boss’s email in the spam folder!

## 📌 2. **Recall** — "How many of the actual spam emails did you catch?"

Out of all the real spam emails, how many did your model **correctly catch**?

### ✅ High Recall:

Your model **catches most of the spam**, but might sometimes wrongly catch a few good ones too.

* Great when **missing something is risky**.
* Example: In medical tests, you'd rather **flag more people for further testing** than miss someone who actually has the disease.

## 📌 3. **F1 Score** — "A balance between precision and recall."

It's like the **average performance** when you care about both **being right** and **not missing anything**.

### ✅ Use F1 when:

* You need a **balance** between precision and recall.
* Especially when your data is **imbalanced** (e.g., very few spam emails compared to normal ones).

## 📌 4. **ROC AUC** — "How well can your model separate spam from not-spam?"

### Let’s say:

* Your model gives **scores** (like confidence):
  + Spam = 0.9
  + Not spam = 0.1
  + Some emails = 0.5

**ROC AUC** looks at **how well these scores rank** emails:

* Are **spams getting higher scores** than non-spams?

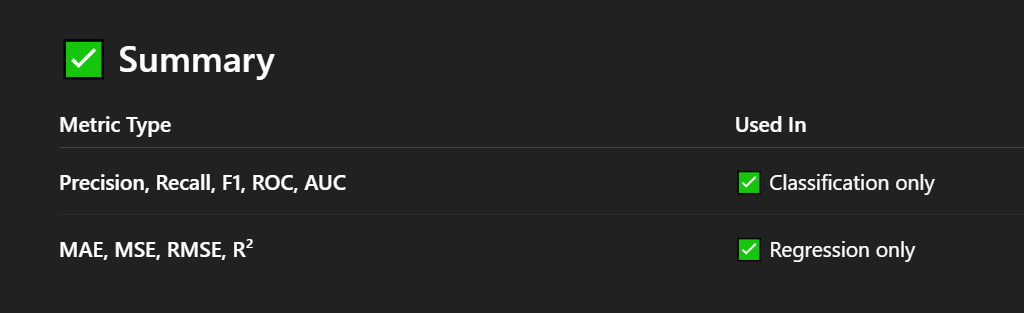
AUC = 1 → Perfect separation  
AUC = 0.5 → Random guessing  
AUC = 0 → Totally wrong

### ✅ Use ROC AUC when:

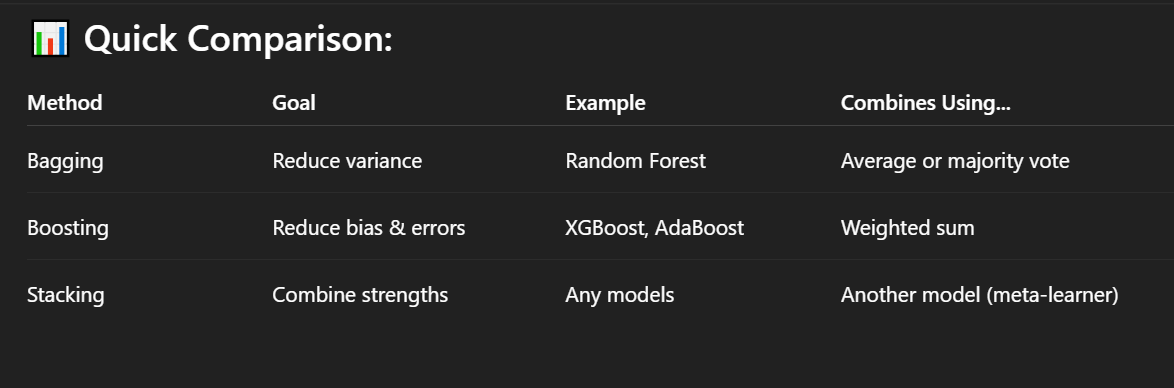
* You want to measure how **well the model distinguishes classes**, regardless of what threshold you choose.

## 🧠 Quick Recap Table:

| **Metric** | **Tells You...** | **You Want It High When...** |
| --- | --- | --- |
| **Precision** | Out of predicted positives, how many are correct | False positives are expensive (e.g., spam filter) |
| **Recall** | Out of actual positives, how many you found | False negatives are risky (e.g., disease detection) |
| **F1 Score** | Balance of precision & recall | You care about both equally |
| **ROC AUC** | Overall ranking ability of the model | You want general performance, not tied to a specific threshold |

****

**Quick comparison between Ensemble Models**

****

**BoW (Bag of Words)** is a popular technique used in **Natural Language Processing (NLP)** to convert text into numerical features that machine learning models can understand. Here's a simple explanation:

### 🔹 What is BoW?

The **Bag of Words** model represents text data by **counting how often each word appears** in the document, without caring about:

* **Grammar**
* **Word order**
* **Context**

It treats the text as just a **"bag" of words**, hence the name.

### 🔹 How it Works — Step-by-Step

Let's say we have two sentences:

1. "I love data science."
2. "Data science is fun."

#### **Step 1: Create Vocabulary**

List all **unique words** from all documents:

css

CopyEdit

["I", "love", "data", "science", "is", "fun"]

#### **Step 2: Vectorize Sentences**

Each sentence becomes a vector based on **word counts**:

| **Word** | **I** | **love** | **data** | **science** | **is** | **fun** |
| --- | --- | --- | --- | --- | --- | --- |
| Sentence 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| Sentence 2 | 0 | 0 | 1 | 1 | 1 | 1 |

### 🔹 Where BoW is Used

* **Text classification** (e.g. spam detection)
* **Sentiment analysis**
* **Topic modeling**
* **Document similarity**

### 🔹 Pros

* Simple and easy to implement
* Works well with many traditional ML models

### 🔹 Cons

* **Ignores word order** (so "not good" and "good not" are the same)
* **Sparse vectors** (most entries are 0)
* Doesn't capture **semantics** or **context**