Spam msg detection project

Code notes

1. **[from sklearn.preprocessing import LabelEncoder**

encoder = LabelEncoder()]

Explanation of this code -

`from sklearn.preprocessing import LabelEncoder`

- You import the `LabelEncoder` class from `sklearn.preprocessing`.
- `LabelEncoder` is used when you have **categorical data (like strings)** and you want to convert them into **numeric values**.

Example:

```
["red", "green", "blue", "green"]
```

will become:

```
[2, 1, 0, 1]
```

- **`encoder = LabelEncoder()`**
- Here you create an **instance (object)** of the LabelEncoder class called encoder.
- This object can now be used to **fit** and **transform** your categorical data.
- **How to use it:**

```
data = ["red", "green", "blue", "green"]
encoded = encoder.fit_transform(data)
print(encoded)
```

Output:

```
[2 1 0 1]
```

- `fit_transform()` learns the mapping (like "blue" -> 0, "green" -> 1, "red" -> 2) and applies it to your data.
- You can also decode back with:

```
encoder.inverse_transform([2, 1, 0])
```

which gives:

```
['red', 'green', 'blue']
```

2. `[df['target'] = encoder.fit_transform(df['target'])]`

Code explanation -

1. **`df['target']`**
    a. Refers to the column named **`target`** in your pandas DataFrame `df`.
    b. Suppose it has categorical labels like:

```
spam
ham
spam
ham
```

2. **`encoder.fit_transform(df['target'])`**
    a. `fit_transform()` first **learns the mapping** from labels → numbers (fit)
    b. Then it **applies the transformation** (transform).

Example:

```
"ham"  -> 0
"spam" -> 1
```

So the column becomes something like:

```
1
0
```

```
1
0
```

3. `df['target'] = ...`
    a. This replaces the original `target` column with the new **numeric encoded version**.
    b. After execution, `df['target']` will contain only numbers (no strings).

✅ **In short:**
That line **encodes the categorical target column into numeric values**, making it usable for machine learning models (which usually require numeric input).

3. **df.isnull().sum()**

1. `df.isnull()`
    a. Checks the entire DataFrame for missing values (`NaN` or `None`).
    b. Returns a DataFrame of the same shape with `True` where the value is missing, and `False` otherwise.

Example:

```
A      B
1      NaN
2      5
NaN    7
```

`df.isnull()` →

```
   A      B
0  False   True
1  False  False
2   True  False
```

2. **`.sum()`**
    a. Since `True = 1` and `False = 0` in Python, summing counts the number of `True` values (i.e., missing values) in each column.

Result:

```
A    1
B    1
dtype: int64
```

✅ **In short:**
`df.isnull().sum()` tells you how many missing values are in **each column** of your DataFrame.

4. **`df = df.drop_duplicates(keep='first')`**

- **`df.drop_duplicates()`**
- Removes duplicate **rows** from your DataFrame.
- A duplicate row means **all column values are the same** as another row.
- **`keep='first'`**
- If duplicates exist, this tells pandas to **keep the first occurrence** and **remove the rest**.
- Options you can use:
    - `keep='first'` → keep the first row, drop the rest (default).
    - `keep='last'` → keep the last row, drop earlier ones.
    - `keep=False` → drop **all** duplicates (nothing is kept).
- **`df = ...`**
- Reassigns the result back to `df`, so your DataFrame is updated without duplicates.

1. 5. `plt.pie(df['target'].value_counts(), labels['ham','spam'],autopct="%0.2f")`
`plt.show()`

    `df['target'].value_counts()`
    a. `This counts how many times each value appears in the target column.`
    b. `Suppose target has two classes: ham and spam.`
    `Example:`

2. ham   4825
   spam   747
   Name: target, dtype: int64

    a. So df['target'].value_counts() → [4825, 747] (counts of ham and spam).
3.
4. **plt.pie(...)**
    a. Creates a pie chart using those counts.
    b. Parameters:
        i. **df['target'].value_counts()** → sizes of each slice.
        ii. **labels=['ham','spam']** → labels for the slices.
        iii. **autopct="%0.2f"** → displays percentages with 2 decimal places inside the chart.
5. Example: if ham=4825 and spam=747, percentages are:
    a. Ham = 86.58%
    b. Spam = 13.42%

6. plt.show() - displays the chart

Now after the chart is displayed I can see the the data is imbalanced so we've to use the below techniques

- In classification, each class (label) should ideally have a **similar number of samples**.
- If one class has **much more data** than the other, the dataset is called **imbalanced**.

◇ Example with your spam dataset:

Suppose your target distribution is:

```
ham  = 4825 samples (≈ 86.6%)
spam =  747 samples (≈ 13.4%)
```

Here:

- ham dominates the dataset.
- spam is much less represented.

👉 This is **imbalanced data.**

◇ Why is imbalance a problem?

1. **Model bias**:
    a. The model may learn to just predict the majority class ("ham" all the time) and still get high accuracy (~86%).
    b. But it will fail to correctly identify the minority class ("spam"), which might actually be the most important!
2. **Metrics get misleading**:
    a. Accuracy looks good (because most predictions are ham), but **precision, recall, F1-score** for spam will be poor.

◇ How to handle imbalance?

You don't always delete data. Instead, you can:

1. **Resampling techniques**
    a. **Oversampling** the minority class (e.g., duplicate or use SMOTE).
    b. **Undersampling** the majority class.
2. **Class weights**
    a. Tell the model to pay more attention to the minority class.
    b. Example in scikit-learn:

```
model = LogisticRegression(class_weight='balanced')
```

3. **Use better metrics**
    a. Instead of accuracy, use **precision, recall, F1-score, ROC-AUC.**

6. What is NLTK?

- It is a **Python library** used for working with **natural language processing (NLP)** tasks.
- NLTK provides tools for:
    - Text preprocessing

- Tokenization (splitting text into words/sentences)
- Stemming & Lemmatization (reducing words to their root form)
- Stopword removal (removing common words like *the, is, and*)
- Part-of-speech (POS) tagging
- Named Entity Recognition (NER)
- Building simple NLP models

◇ Why use NLTK?

- It's one of the **earliest and most popular NLP libraries** in Python.
- Good for **learning NLP concepts** and small projects.
- Comes with many **preloaded datasets & corpora** (like WordNet, stopwords, etc.).

6. PUNKT

- **punkt** is a **pre-trained tokenizer model** that comes with NLTK.
- It is used for **sentence tokenization** and **word tokenization**.
- Without downloading punkt, functions like word_tokenize() or sent_tokenize() won't work.

◇ Why download?

NLTK has a lot of built-in **datasets & models** (stopwords, WordNet, POS taggers, etc.). They're not installed by default, so you need to download them once.

◇ Example usage:

```
import nltk
from nltk.tokenize import sent_tokenize, word_tokenize

nltk.download('punkt')

text = "Hello! How are you doing today? NLTK is great for NLP."
print("Sentences:", sent_tokenize(text))
print("Words:", word_tokenize(text))
```

Output:

```
Sentences: ['Hello!', 'How are you doing today?', 'NLTK is great for NLP.']
Words: ['Hello', '!', 'How', 'are', 'you', 'doing', 'today', '?', 'NLTK', 'is',
'great', 'for', 'NLP', '.']
```

✅ **In short:**
nltk.download('punkt') downloads the **tokenizer model** so you can split text into **words and sentences.**

8. df['num_characters'] = df['text'].apply(len)

1. **df['text']**
   a. Refers to the column named **text** in your DataFrame.
   b. Suppose it contains SMS/email messages like:

```
"Hello there"
"Free offer!!!"
"Good morning"
```

2. **.apply(len)**
   a. .apply() applies a function to every element of the column.
   b. Here, the function is Python's built-in **len()**, which gives the **length of a string** (number of characters, including spaces and punctuation).

Example:

```
len("Hello there")  → 11
len("Free offer!!!") → 13
```

3. **df['num_characters'] = ...**
   a. Creates a **new column** called num_characters and stores the calculated lengths there.

◇ Example:

```
import pandas as pd

df = pd.DataFrame({'text': ["Hello there", "Free offer!!!", "Good morning"]})
```

```
df['num_characters'] = df['text'].apply(len)
print(df)
```

Output:

```
          text  num_characters
0    Hello there              11
1  Free offer!!!              13
2  Good morning               12
```

✅ **In short:**
That line adds a **new column** (num_characters) which stores the **length of each text message** in your dataset.

```
C:\Users\hrith\AppData\Local\Temp\ipykernel_14360\253964734.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
‘’’Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df['num_characters'] = df['text'].apply(len)’’’
```

```
df['num_words'] = df['text'].apply(lambda x:len(nltk.word_tokenize(x)))
```

1. **df['text']**
    a. Refers to the text column (messages, sentences, etc.).
    b. Example:

```
"Hello there"
"Free offer!!!"
```

2. **lambda x: ...**
    a. A **lambda function** means "an anonymous function".
    b. For each text x, it applies the expression inside.
3. **nltk.word_tokenize(x)**
    a. Splits the text into words (using NLTK’s tokenizer).
    b. Example:

```
nltk.word_tokenize("Hello there!") → ['Hello', 'there', '!']
```

4. **len(...)**
    a. Counts how many tokens (words + punctuation) are there.
    b. Example:

len(['Hello', 'there', '!']) → 3

5. **df['num_words'] = ...**
   a. Creates a new column num_words and stores the word counts.

◇ Example:

```
import pandas as pd
import nltk
nltk.download('punkt')

df = pd.DataFrame({'text': ["Hello there", "Free offer!!!", "Good morning"]})
df['num_words'] = df['text'].apply(lambda x: len(nltk.word_tokenize(x)))

print(df)
```

Output:

```
          text  num_words
0    Hello there          2
1  Free offer!!!          3
2  Good morning          2
```

✅ **In short:**
This line adds a new column num_words that stores the **number of tokens (words) in each text message.**

```
df[df['target'] == 0][['num_characters','num_words','num_sentences']].describe()
```

Step 1: df['target'] == 0

This creates a **boolean mask** that checks which rows in df have target == 0.

- If target = 0 → that row is selected
- If target ≠ 0 → that row is ignored

👉 In your spam dataset:

- 0 usually means **ham (not spam)**
- 1 means **spam**

So this step filters only **ham messages**.

◇ Step 2: df[...]

When you wrap the mask inside df[...], pandas returns only the rows where the condition is True.
So now we have a DataFrame with only **ham messages**.

◇ Step 3: [['num_characters','num_words','num_sentences']]

This selects only these **three columns**:

- num_characters → length of message in characters
- num_words → number of words in the message
- num_sentences → number of sentences in the message

So now we have a smaller DataFrame with only **ham messages** and only **3 numerical columns**.

◇ Step 4: .describe()

This gives **summary statistics** for those numeric columns.
Typical output looks like:

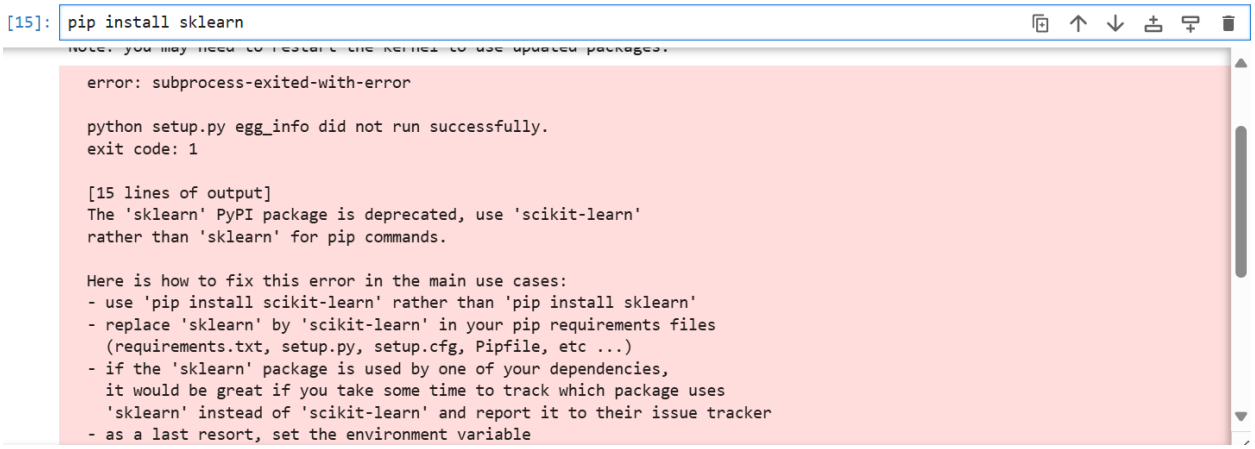
	num_characters	num_words	num_sentences
count	4825.000000	4825.000000	4825.000000
mean	71.451234	15.612345	1.345678
std	59.345678	12.123456	0.678901
min	2.000000	1.000000	1.000000
25%	30.000000	7.000000	1.000000
50%	60.000000	14.000000	1.000000
75%	100.000000	22.000000	2.000000
max	910.000000	160.000000	12.000000

◇ Meaning

This tells you:

- How **long** ham messages are compared to spam (later you'll probably compare with target == 1)
- The **distribution** of message length in characters, words, and sentences
- Useful for understanding dataset imbalance and feature engineering

✅ **In short:**
That line of code filters only **ham messages** (target=0), selects length-related features, and prints descriptive statistics (count, mean, std, min, max, quartiles).



Implementing using matplotlib first-

Density curves with Matplotlib (plt.hist + density=True)

For ham

```
plt.hist(df[df['target'] == 0]['num_characters'],

         bins=100, density=True, alpha=0.5, label='Ham')
```

1. **df[df['target'] == 0]**
    a. This selects only the rows from the DataFrame df where the column target is equal to 0.
    b. Likely, in your dataset, target = 0 represents **Ham (non-spam messages)**.
2. **['num_characters']**
    a. From those filtered rows, it extracts the column num_characters.
    b. This column probably represents the length of each message (number of characters).
3. **plt.hist(..., bins=100, density=True, alpha=0.5, label='Ham')**
    a. plt.hist(...) → Plots a histogram of the data.

b. bins=100 → Splits the range of num_characters into 100 intervals (bars).
c. density=True → Normalizes the histogram so that the total area = 1 (useful for comparing distributions).
d. alpha=0.5 → Sets transparency to 50%, so if you plot another histogram on top, both will be visible.
e. label='Ham' → Adds a legend label "Ham" to this histogram.

In short:

👉 This plots a **normalized histogram of message lengths (in characters) for Ham messages** with 100 bins, slightly transparent, and labeled "Ham".

```
plt.figure(figsize=(12,6))

sns.histplot(df[df['target'] == 0]['num_words'])

sns.histplot(df[df['target'] == 1]['num_words'],color='red')
```

1. **plt.figure(figsize=(12,6))**
   a. Creates a new figure for plotting.
   b. The figure size is set to **12 inches wide** and **6 inches tall**, which gives you a wide rectangular plot.

2. **df[df['target'] == 0]['num_words']**
   a. This filters your DataFrame df to include only rows where target == 0.
   b. Most likely, target = 0 means **Ham (non-spam)** messages in your dataset.
   c. Then, it selects the column num_words, which probably contains the number of words in each message.

✅ So this gives you a list of word counts for all **Ham** messages.

3. **sns.histplot(df[df['target'] == 0]['num_words'])**
   a. Plots a histogram of **Ham messages' word counts**.
   b. By default, Seaborn decides the number of bins and styling.
   c. It shows how often messages of different word lengths appear.

4. **sns.histplot(df[df['target'] == 1]['num_words'], color='red')**
   a. Same as above, but for rows where target == 1.
   b. Here, target = 1 probably represents **Spam messages**.
   c. This histogram is drawn in **red**, so you can compare Spam vs Ham distributions on the same plot.

```
sns.pairplot(df,hue='target')
```

🔎 Step-by-step Explanation

1. **sns.pairplot(df, ...)**
    a. Creates a **pairwise plot (scatterplot matrix)** from the DataFrame df.
    b. It plots **every numerical column against every other numerical column** in a grid of scatterplots.
    c. On the **diagonal**, it shows the distribution of each variable (usually as a histogram or KDE plot).

2. **hue='target'**
    a. Colors the points in the plots according to the target column.
    b. In your dataset, target probably has two values:
        i. 0 → **Ham**
        ii. 1 → **Spam**
    c. This way, you can compare how Ham and Spam distribute across all numerical features.

🎯 What the plot tells you

- Each cell (off-diagonal) is a scatterplot of one feature vs another.
- Diagonal cells show the distribution (histogram/density) of that feature.
- The **colors** (from hue='target') let you see if Ham and Spam cluster differently.
    - Example: If Spam points are concentrated in a different region of a scatterplot than Ham, that feature might be useful for classification.

✅ **In short:**
pairplot is a quick way to explore **relationships between all numerical features** in your dataset, with color-coded groups (here, Spam vs Ham).

```
sns.heatmap(df.corr(),annot=True)
```

ValueError: could not convert string to float: 'Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there got amore wat...'

```
sns.heatmap(df.corr(numeric_only=True),annot=True)
```

1. **df.corr()**
    a. This calculates the **correlation matrix** of your DataFrame df.

b. Correlation measures the linear relationship between two numerical variables:
    i. **+1** → Perfect positive correlation (when one increases, the other always increases).
    ii. **-1** → Perfect negative correlation (when one increases, the other always decreases).
    iii. **0** → No linear relationship.
c. The result is a square table where rows and columns are features, and each cell shows the correlation between two features.

2. **sns.heatmap(...)**
    a. Creates a **heatmap** visualization of the correlation matrix.
    b. Each cell is colored based on the correlation value:
        i. Dark colors (near -1) → strong negative correlation.
        ii. Light colors (near +1) → strong positive correlation.
        iii. Medium colors (near 0) → weak or no correlation.

3. **annot=True**
    a. This means **annotate the heatmap with numbers**.
    b. So instead of just colors, you'll also see the exact correlation values inside each cell.

🎯 What this tells you

- You can quickly see which features are **highly related** to each other.
- Example in your dataset:
    - num_characters and num_words might have a strong positive correlation (longer messages usually have more words).
    - If two features are **very highly correlated** (close to 1), they might be redundant for ML models.
    - Correlation with target can hint at which features separate Ham vs Spam better.

✅ **In short:**
This line plots a heatmap of the correlation matrix, showing **how strongly each numerical feature in your dataset is related to the others**, with exact values written inside the cells.

```
df['transformed_text'] = df['text'].apply(transform_text)
```

suppose your DataFrame looks like:

```
import pandas as pd

data = {'text': [
    "Go until jurong point, crazy.. Available only in bugis n great world la e
buffet...",
    "Ok lar... enjoy yourself",
    "Free entry in 2 a wkly comp..."
]}
df = pd.DataFrame(data)

df['transformed_text'] = df['text'].apply(transform_text)
print(df)
```

Output:

```
                                                text                transformed_text
0  Go until jurong point, crazy.. Available only...  go jurong point crazi avail bugi n
great world la e buffet
1                               Ok lar... enjoy yourself                  ok lar enjoy
2                     Free entry in 2 a wkly comp...                   free entri 2
wkli comp
```

👉 **So this line is basically saying:**
"Take each text in the text column, clean it with transform_text(), and save the result in a new column called transformed_text."

- The transformation **happens in memory** (RAM).
- As long as your **kernel/session is running**, the new column transformed_text exists in df.
- But if you **restart the kernel** (or close Jupyter/Python), all variables, DataFrames, and transformations in memory are lost.

Save the dataframe with

```
df.to_csv("transformed_spam.csv", index=False)
```

reload it with:

```
df = pd.read_csv("transformed_spam.csv")
```

The **wordcloud** module in Python is a library that helps you **visualize text data.**

🔎 What it does

- It takes a big chunk of text (or processed text data, like your transformed_text column).
- It counts how often each word appears.
- It generates an image where:
  - **More frequent words appear larger and bolder.**
  - **Less frequent words appear smaller.**

This is called a **Word Cloud** (also known as a Tag Cloud).

1. `spam_wc = wc.generate(df[df['target'] == 1]['transformed_text'].str.cat(sep=" "))`

   `df['target'] == 1`
   a. This checks which rows in your DataFrame belong to **spam** messages.
   b. Usually, in spam datasets:
      i. target = 1 → spam
      ii. target = 0 → ham (not spam).
2. ✅ So this filters only spam rows.
3.
4. **`df[df['target'] == 1]`**
   a. This selects all rows where the target is spam.
   b. Basically, you're making a smaller DataFrame of **only spam messages**.
5.
6. **`['transformed_text']`**
   a. From that spam-only DataFrame, pick the column that contains the **cleaned text** (after preprocessing).
7.
8. **`.str.cat(sep=" ")`**
   a. This takes the entire column of spam messages (which is many rows of text)
   b. and concatenates them into **one big string**, separated by spaces.
   c. Example:
9. "win prize money" + "free entry today" + "claim reward"
10. → "win prize money free entry today claim reward"

11. ✅ Now the word cloud can process the full spam dataset as a single text input.
12.
13. **wc.generate(...)**
    a. Feeds that big spam-text string into the WordCloud generator.
    b. It counts word frequencies (like "free", "win", "money") and makes the word cloud image.
14.
15. **spam_wc = ...**
    a. Stores the generated word cloud in the variable spam_wc.
16.

17. ☑ In plain English:

18. 👉 "Take all spam messages, combine them into one big text, generate a word cloud from it, and store it in spam_wc."
19. So you'll see common spammy words like **"free"**, **"win"**, **"claim"**, **"prize"** appear big in the word cloud. 🎉

```
spam_corpus = []
for msg in df[df['target'] == 1]['transformed_text'].tolist():
    for word in msg.split():
        spam_corpus.append(word)
```

1. **spam_corpus = []**
    a. Create an **empty list** to store all the words that appear in spam messages.

2. **df[df['target'] == 1]['transformed_text']**
    a. Select only the spam rows (target == 1).
    b. From those rows, take the **transformed_text** column (which contains cleaned text).

3. **.tolist()**
    a. Convert that Pandas Series into a **Python list**.
    b. So instead of a column of spam texts, you get something like:

```
[
  "free entri win cash prize",
```

```
  "claim reward now",
  "congrat win lotteri"
]
```

4. **for msg in ...:**
    a. Loop through each spam message (each string in the list).

5. **for word in msg.split():**
    a. .split() splits the message string into individual words (tokens) separated by spaces.
    b. Example: "free entri win cash prize".split() → ['free', 'entri', 'win', 'cash', 'prize'].

6. **spam_corpus.append(word)**
    a. Each word is added to the spam_corpus list.
    b. Over time, this builds up a list of **all words across all spam messages**.

☑ End Result

spam_corpus becomes a flat list of **every word from all spam messages.**

```
from collections import Counter word_counts =
pd.DataFrame(Counter(spam_corpus).most_common(30),columns=['word','count'])
'''sns.barplot( pd.DataFrame(Counter(spam_corpus).most_common(30))[0],
pd.DataFrame(Counter(spam_corpus).most_common(30))[1] )''' plt.figure(figsize=(12,6))
sns.barplot(x='word', y='count', data= word_counts) plt.xticks(rotation = 90) plt.show()
```

Creating the DataFrame

```
word_counts = pd.DataFrame(Counter(spam_corpus).most_common(30),
                           columns=['word','count'])
```

- Counter(spam_corpus) → counts how many times each word appears in your **spam_corpus.**
- .most_common(30) → keeps only the **top 30 most frequent words.**

- pd.DataFrame(..., columns=['word','count']) → converts the result into a pandas DataFrame with two columns:
    - word → the word itself
    - count → how many times it appeared

✅ Example output might look like:

word	count
free	250
win	180
prize	150
call	140
claim	120

2. Plot setup

```
plt.figure(figsize=(12,6))
```

- This makes the plot **12 inches wide** and **6 inches tall**, so it's readable.

3. Barplot

```
sns.barplot(x='word', y='count', data=word_counts,
            hue='word', palette="tab20", legend=False)
```

- sns.barplot() → draws a bar chart.
- x='word' → words go on the **X-axis**.
- y='count' → frequency goes on the **Y-axis**.
- data=word_counts → uses the DataFrame we made earlier.
- hue='word' → assigns each bar a **different color** based on the word.
- palette="tab20" → picks a set of 20 distinct colors.
- legend=False → hides the legend (otherwise it would just repeat all the words again).

4. Beautifying the plot

```
plt.xticks(rotation=90)
plt.xlabel("Words")
```

```
plt.ylabel("Frequency")
plt.title("Top 30 Spam Words")
plt.show()
```

- plt.xticks(rotation=90) → rotates the words on the X-axis **vertically** so they don't overlap.
- plt.xlabel("Words") → labels the X-axis.
- plt.ylabel("Frequency") → labels the Y-axis.
- plt.title("Top 30 Spam Words") → adds a title at the top.
- plt.show() → finally **displays the graph**.

✅ **Result:**
You get a **bar graph of the 30 most frequent spam words**, each bar in a different color, with words on the X-axis and their counts on the Y-axis.

```
from sklearn.feature_extraction.text import CountVectorizer,TfidfVectorizer
cv = CountVectorizer()
tfidf = TfidfVectorizer(max_features=3000)
```

- Both CountVectorizer and TfidfVectorizer are tools in **scikit-learn** that transform text (sentences, documents, etc.) into **numerical feature vectors**.
- These vectors are then used as input to ML models (like Naive Bayes, Logistic Regression, SVM, etc.).

◇ 2. CountVectorizer

```
cv = CountVectorizer()
```

- This converts text into a **Bag of Words (BoW)** representation.
- It builds a vocabulary of all words in your dataset and counts **how many times each word appears** in a document.
- Example:
  Suppose you have 2 sentences:

```
"Free money now"
"Win money free prize"
```

The vocabulary = [free, money, now, win, prize]

- First sentence → [1,1,1,0,0]
- Second sentence → [1,1,0,1,1]
  Each row = document, each column = word count.

The vectorizer scans all the text and finds **unique words.**
Here, unique words are:

TfidfVectorizer

```
tfidf = TfidfVectorizer(max_features=3000)
```

- TF-IDF = **Term Frequency – Inverse Document Frequency.**
- Unlike CountVectorizer, which just counts, TF-IDF **weighs words** by importance:
  - **Term Frequency (TF):** How often a word appears in a document.
  - **Inverse Document Frequency (IDF):** How rare a word is across all documents.
- This way, **common words like "the, is, a" get lower weight**, while rare but important words like "prize, free, urgent" get higher weight.

✅ **Better than BoW** when you want to highlight important words instead of just counts.

- max_features=3000 → limits the vocabulary size to the **top 3000 words** (by frequency/importance).
  - This reduces **dimensionality** (since vocab can be huge in real-world text).

[free, money, now, win, prize]

This becomes our **vocabulary.**
👉 Each word gets a column index:

- free → 0
- money → 1
- now → 2
- win → 3
- prize → 4

📌 Step 2: Encode Sentence 1 → "Free money now"

- Does it contain **free**? ✅ yes → 1
- Does it contain **money**? ✅ yes → 1
- Does it contain **now**? ✅ yes → 1
- Does it contain **win**? ❌ no → 0

- Does it contain **prize**? ❌ no → 0

👉 Vector = [1, 1, 1, 0, 0]

📌 Step 3: Encode Sentence 2 → "Win money free prize"

- **free**? ✅ yes → 1
- **money**? ✅ yes → 1
- **now**? ❌ no → 0
- **win**? ✅ yes → 1
- **prize**? ✅ yes → 1

👉 Vector = [1, 1, 0, 1, 1]

📌 Step 4: Combine into Matrix

So the two sentences become:

```
[1, 1, 1, 0, 0]   ← Sentence 1 ("Free money now")
[1, 1, 0, 1, 1]   ← Sentence 2 ("Win money free prize")
```

This is a **document-term matrix**.

- Each **row** = a document (sentence).
- Each **column** = a word in the vocabulary.
- Each **cell** = how many times the word appears in that document.

X = tfidf.fit_transform(df['transformed_text']).toarray()

tfidf

Earlier you defined:

```
tfidf = TfidfVectorizer(max_features=3000)
```

- This is a **TF-IDF Vectorizer** (Term Frequency – Inverse Document Frequency).
- It converts text into numerical vectors, but instead of raw counts (like CountVectorizer), it **weights words** based on importance:

    - Common words across many documents get **lower weight**.
    - Rare but important words get **higher weight**.

2. fit_transform(df['transformed_text'])

- **fit** → learns the vocabulary from your transformed_text column (all unique words, up to max_features=3000).
- **transform** → converts each document into a TF-IDF vector.
- So now you get a **sparse matrix** (efficient storage of mostly zeros).

For example:
If df['transformed_text'] has 3 documents:

```
0   "free money now"
1   "win money free prize"
2   "only in jurong point"
```

TF-IDF might look like (numbers are weights):

```
[0.57, 0.57, 0.57, 0.00, 0.00, ...]   ← Doc 1
[0.46, 0.46, 0.00, 0.63, 0.46, ...]   ← Doc 2
[0.00, 0.00, 0.00, 0.00, 0.71, ...]   ← Doc 3
```

3. .toarray()

- The result of fit_transform is a **sparse matrix** (not stored as a normal array, saves memory).
- .toarray() converts it into a **dense NumPy array** so you can use it like a regular dataset (rows = documents, columns = features).

4. X = ...

Now X is your **feature matrix**:

- Shape: (number_of_documents, number_of_features)
- Each row = a document.
- Each column = a word in the vocabulary.
- Each cell = TF-IDF score of that word in that document.

✅ This X can now be fed directly into a machine learning model like **Logistic Regression, Naive Bayes, SVM, Random Forest, etc.** for classification.

```
y=df['target'].values
```

- df → is your pandas DataFrame (contains your dataset).
- df['target'] → selects the **"target" column** (labels of your dataset, usually 0 = ham, 1 = spam).
- .values → converts the pandas **Series** into a **NumPy array**.

So, if your dataset looks like this:

text	target
"Win money now"	1
"How are you?"	0
"Claim your free prize today"	1

✅ Why do we use .values?
Because most machine learning models (like scikit-learn classifiers) expect the target labels y as a **NumPy array**, not a pandas Series.

👉 So in short:
y = the labels of your dataset (0 = ham, 1 = spam) stored as a NumPy array for training.

```
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,random_state=2)
```

- **train_test_split** → function from sklearn.model_selection that splits your dataset into **training data** and **testing data**.
- **Inputs**:
  - X → features (your TF-IDF vectors of messages).
  - y → labels (0 = ham, 1 = spam).
- **Parameters**:
  - test_size=0.2 → 20% of the dataset will be used for **testing**, and 80% for **training**.
    - Example: if you have 5000 messages → 4000 go to training, 1000 go to testing.
  - random_state=2 → ensures **reproducibility**. If you run the split multiple times, you'll always get the same split of train/test data. (If you don't fix random_state, each run could shuffle differently).
- **Outputs**:

- X_train → TF-IDF features for training messages.
- X_test → TF-IDF features for testing messages.
- y_train → labels for training (spam/ham).
- y_test → labels for testing.

```
from sklearn.naive_bayes import GaussianNB,MultinomialNB,BernoulliNB from sklearn.metrics import accuracy_score,confusion_matrix,precision_score
```

These are **Naive Bayes classifiers** from scikit-learn. All are based on the **Naive Bayes theorem**, but they work best for **different types of data**:

1. **GaussianNB**
   a. Assumes that the features follow a **normal (Gaussian) distribution**.
   b. Used for **continuous data** (e.g., age, salary, height).
   c. Example: if features are real numbers, like [1.7, 3.2, 5.8].
2. **MultinomialNB**
   a. Best for **discrete counts** (like word frequencies in text).
   b. Works very well in **text classification problems** (spam detection, sentiment analysis).
   c. Example: "Free money now" → [1,1,1,0,0] (counts of words).
3. **BernoulliNB**
   a. Best for **binary features** (0 or 1 → whether a word appears or not).
   b. Example: "Free money now" → [1,1,1,0,0] (presence/absence of words).
   c. Often used when you care only about **whether a word exists**, not how many times.

📦 2. from sklearn.metrics import accuracy_score, confusion_matrix, precision_score

These are **evaluation metrics** used to measure model performance:

1. **accuracy_score(y_true, y_pred)**
   a. Tells you the % of correct predictions.
   b. Formula:

$$Accuracy = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$Accuracy=Total PredictionsCorrect Predictions

   c. Example: If out of 100 SMS, 90 are classified correctly → accuracy = 0.9 (90%).

2. **confusion_matrix(y_true, y_pred)**
    a. Shows a **table** of predictions vs actual values.
    b. For binary classification (spam/ham), it looks like this:

	Predicted Ham	**Predicted Spam**
Actual Ham	True Negative (TN)	False Positive (FP)
Actual Spam	False Negative (FN)	True Positive (TP)

    c. Example: if the model mislabels some spam as ham, you'll see it in FN.

3. **precision_score(y_true, y_pred)**
    a. Precision answers: **Of all predicted spams, how many were actually spam?**
    b. Formula:

*Precision=TPTP+FP|text{Precision} = |frac{TP}{TP + FP}*Precision=TP+FPTP

    c. Example: If model predicts 50 spams, but only 40 are actually spam → precision = 40/50 = 0.8.

✅ In summary:

- **Naive Bayes classifiers** → build the model.
- **Metrics** → check how good the model is.

Naive Bayes

https://www.geeksforgeeks.org/machine-learning/naive-bayes-classifiers/

Naive Bayes Classifiers – GeeksforGeeks Summary

1. Core Principle (Bayes' Theorem + Independence)

Naive Bayes is a **probabilistic classifier** built on Bayes' Theorem. It computes the probability that a data point belongs to a certain class given its features. The "naive" part reflects its strong assumption that **all features are conditionally independent given**

the class, even though this is often unrealistic—yet it simplifies computation significantly GeeksforGeeks+1.

2. Conditional Independence Simplifies Probability Calculation

The algorithm models the features independently:

$P(x|y)=\prod\alpha P(x\alpha|y)P(x \mid y) = \prod_ \alpha P(x_ \alpha \mid y)$P(x|y)=α∏ P(xα |y)

This greatly reduces complexity while still performing well in practice GeeksforGeeksScikit-learn.

3. Popular Variants

- **GaussianNB** – Assumes continuous features follow a **normal distribution**; suitable for real-valued data (e.g., heights, sensor readings) GeeksforGeeks+1.
- **MultinomialNB** – Ideal for modeling **word counts** or frequency data (common in text analysis such as spam detection) CodefinityIBM.
- **BernoulliNB** – Works with **binary features** (presence or absence of a feature); useful for document classification where words are either present or not CodefinityIBM.

4. Strengths of Naive Bayes

- **Simple and fast** to train and predict—even with large datasets IBMScikit-learnPickl.AI.
- **Effective with high-dimensional data** (like text), since we rarely run into performance issues despite the many features Pickl.AIIBM.
- Still performs **well in practice**, especially in domains like document classification and spam filtering, even when independence assumptions are slightly violated GeeksforGeeksPickl.AIScikit-learn.

5. Limitations

- The **feature independence assumption** may not hold for real-world data—features often interact or correlate, which can reduce accuracy Codefinityheadgym.com.

- The **zero-frequency problem**: If a feature appears in test data but not in training data, the probability becomes zero. This can be handled with techniques like **Laplace smoothing** upGrad.

6. Real-World Uses

Naive Bayes is widely applied where the combination of simplicity, speed, and effectiveness matters:

- **Spam detection**
- **Sentiment analysis**
- **Document categorization**
- **Medical diagnosis**
- **Recommender systems** Pickl.AIIBMheadgym.com.

Summary Table

Aspect	Description
Basis	Bayes' Theorem with conditional independence
Strengths	Fast, simple, scalable to high-dimensional text
Variants	Gaussian, Multinomial, Bernoulli
Limitations	Independence assumption; zero-frequency issue
Typical Use Cases	Spam filtering, sentiment & document classification, diagnostics

gnb = GaussianNB()

- **Gaussian Naive Bayes**
- Used when your features are **continuous values** (e.g., heights, weights, temperatures).
- It assumes each feature follows a **Gaussian (normal) distribution** within each class.
- Example:
  If you want to classify flowers based on petal length and width (numeric), GaussianNB is appropriate.

👉 Formula:

$P(x|y) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$P(x|y)=2πσ2 1 exp(−2σ2(x−μ)2)

where μ (mu) and σ (sigma) are the mean and standard deviation of the feature.

2. mnb = MultinomialNB()

- **Multinomial Naive Bayes**
- Used when your features are **counts or frequencies** (non-negative integers).
- Very common in **text classification** tasks.
- Example: Spam detection — each email is represented by word counts (bag-of-words).
  - "Free money now" → [1,1,1,0,0]
  - "Win free prize" → [1,0,0,1,1]

👉 MultinomialNB works well here because it expects discrete count features.

3. bnb = BernoulliNB()

- **Bernoulli Naive Bayes**
- Used when your features are **binary (0 or 1)**.
- Example: Whether a word is **present (1)** or **absent (0)** in a document (not the count).
- Works well when the **presence of a feature matters more than its frequency**.

👉 For spam detection:

- Email A: "Free money now" → [1,1,1,0,0] (word present or not)
- Email B: "Win free prize" → [1,0,0,1,1]

🚀 In Short:

Classifier	Works Best With	Example Use Case
GaussianNB	Continuous features	Iris dataset, sensor data
MultinomialNB	Word counts / frequencies	Text classification, spam detection
BernoulliNB	Binary features (0/1)	Document classification (word present/absent)

```
gnb.fit(X_train,y_train) y_pred1 = gnb.predict(X_test)
print(accuracy_score(y_test,y_pred1)) print(confusion_matrix(y_test,y_pred1))
print(precision_score(y_test,y_pred1))
```

gnb.fit(X_train, y_train)

- **Training the model.**
- X_train: Features for training (TF-IDF vectors or count vectors in your case).
- y_train: Labels for training (spam = 1, ham = 0).
- The Naive Bayes classifier **learns the probability distribution** of features for each class here.
    - For spam detection, it learns things like:
        - Probability of seeing "free" given spam.
        - Probability of seeing "hello" given ham.

2. y_pred1 = gnb.predict(X_test)

- **Prediction step.**
- The trained model looks at unseen X_test (features of test dataset) and predicts labels (0 = ham, 1 = spam).
- y_pred1: An array of predicted labels for the test set.

3. accuracy_score(y_test, y_pred1)

- **Accuracy = (Correct Predictions / Total Predictions).**
- Example:
    - Test set = 100 messages.
    - Correctly predicted = 90.
    - Accuracy = 90%.

⚠️ But accuracy can be misleading if your dataset is **imbalanced** (e.g., 80% ham, 20% spam).

4. confusion_matrix(y_test, y_pred1)

This gives you a **2x2 matrix**:

[TNFPFNTP]\begin{bmatrix} TN & FP \\ FN & TP \end{bmatrix}[TNFN FPTP]

- **TN (True Negative):** Model predicted ham, actually ham.
- **FP (False Positive):** Model predicted spam, actually ham.
- **FN (False Negative):** Model predicted ham, actually spam.
- **TP (True Positive):** Model predicted spam, actually spam.

👉 Helps you see where the model is making mistakes.

5. precision_score(y_test, y_pred1)

- **Precision = TP / (TP + FP)**
- Out of all the messages the model predicted as spam, how many were actually spam?
- Precision is very important in spam detection.
    - High precision = Almost everything flagged as spam is actually spam.
    - Low precision = Model often marks ham as spam (bad user experience).

☑ Why this approach?

You are:

1. Training (fit) the Naive Bayes model on training data.
2. Testing (predict) on unseen test data.
3. Evaluating using multiple metrics:
    a. **Accuracy** → Overall performance.
    b. **Confusion Matrix** → Detailed breakdown of errors.
    c. **Precision** → Reliability of spam predictions.

Linear Models

1. **LogisticRegression**
    a. A linear model for classification.
    b. Works well for high-dimensional data like text (Bag of Words / TF-IDF).
    c. Often a very strong baseline.
2. **SVC (Support Vector Classifier)**
    a. Powerful classifier that finds the best boundary (hyperplane) between classes.
    b. Works well on text but can be slow on very large datasets.

◇ Naive Bayes

3. **MultinomialNB**
    a. Designed for text classification (word counts, TF-IDF).
    b. Very fast and efficient.
    c. Often used as the first model for spam detection.

◇ Tree-based Models

4. **DecisionTreeClassifier**
    a. Simple model based on splitting data by feature values.
    b. Easy to interpret, but can **overfit** easily.
5. **KNeighborsClassifier (KNN)**
    a. Classifies based on nearest neighbors in feature space.
    b. Usually not great for text data (too slow and memory heavy).

◇ Ensemble Models

6. **RandomForestClassifier**
    a. An ensemble of decision trees (bagging).
    b. More robust than a single decision tree.
7. **AdaBoostClassifier**
    a. Boosting method: builds multiple weak learners sequentially, each fixing errors of the last.
    b. Good for improving performance on tricky datasets.
8. **BaggingClassifier**
    a. Trains multiple classifiers on random subsets of data (bagging).
    b. Helps reduce overfitting.
9. **ExtraTreesClassifier**
    a. Similar to RandomForest but with more randomness in splits.
    b. Often faster and sometimes more accurate.
10. **GradientBoostingClassifier**

- Boosting method where trees are added sequentially to minimize errors.
- More powerful than AdaBoost in many cases.

11. **XGBClassifier (Extreme Gradient Boosting)**

- Advanced version of Gradient Boosting.
- Very popular in Kaggle competitions because of speed + accuracy.
- Usually gives excellent results on structured/tabular data, and can work well with TF-IDF features.

```
svc = SVC(kernel='sigmoid', gamma=1.0) knc = KNeighborsClassifier() mnb = MultinomialNB()
dtc = DecisionTreeClassifier(max_depth=5) lrc = LogisticRegression(solver='liblinear',
penalty='l1') rfc = RandomForestClassifier(n_estimators=50, random_state=2) abc =
```

```
AdaBoostClassifier(n_estimators=50, random_state=2) bc =
BaggingClassifier(n_estimators=50, random_state=2) etc =
ExtraTreesClassifier(n_estimators=50, random_state=2) gbdt =
GradientBoostingClassifier(n_estimators=50,random_state=2) xgb =
XGBClassifier(n_estimators=50,random_state=2)
```

Support Vector Classifier

```
svc = SVC(kernel='sigmoid', gamma=1.0)
```

- kernel='sigmoid': Uses a sigmoid kernel (acts like a neural net activation).
- gamma=1.0: Controls how much influence a single training point has.

◇ K-Nearest Neighbors

```
knc = KNeighborsClassifier()
```

- Default KNN classifier.
- Classifies based on closest neighbors in feature space.

◇ Naive Bayes

```
mnb = MultinomialNB()
```

- Good for text data (word counts / TF-IDF).
- Works fast and is often a strong baseline.

◇ Decision Tree

```
dtc = DecisionTreeClassifier(max_depth=5)
```

- Single tree with depth limited to 5.
- Limiting depth helps prevent overfitting.

◇ Logistic Regression

```
lrc = LogisticRegression(solver='liblinear', penalty='l1')
```

- solver='liblinear': Solver for small datasets.
- penalty='l1': Lasso regularization (forces some coefficients to 0 → feature selection).

◇ Random Forest

```
rfc = RandomForestClassifier(n_estimators=50, random_state=2)
```

- An ensemble of 50 decision trees.
- random_state=2 → ensures reproducibility.

◇ AdaBoost

```
abc = AdaBoostClassifier(n_estimators=50, random_state=2)
```

- Boosting algorithm (sequentially fixes mistakes).
- 50 weak learners (usually decision stumps).

◇ Bagging Classifier

```
bc = BaggingClassifier(n_estimators=50, random_state=2)
```

- Bagging = bootstrap aggregating.
- Trains 50 base estimators on random subsets of data.

◇ Extra Trees

```
etc = ExtraTreesClassifier(n_estimators=50, random_state=2)
```

- Like Random Forest, but splits are more random.
- Often faster and sometimes more accurate.

◇ Gradient Boosting

```
gbdt = GradientBoostingClassifier(n_estimators=50, random_state=2)
```

- Boosting algorithm where each tree tries to fix the previous tree's errors.
- Slower than Random Forest but often more accurate.

◇ XGBoost

```
xgb = XGBClassifier(n_estimators=50, random_state=2)
```

- Optimized Gradient Boosting implementation.
- Very powerful (often wins Kaggle competitions).

- Keys ('SVC', 'KN', …) → short names for your models.
- Values (svc, knc, …) → actual model objects you defined earlier.

This is super useful because now you can **loop through all models** easily instead of writing code for each one.

✅ Example usage:

```
from sklearn.metrics import accuracy_score, precision_score, confusion_matrix

results = []

for name, model in clfs.items():
    model.fit(X_train, y_train)             # Train
```

```
    y_pred = model.predict(X_test)           # Predict
    acc = accuracy_score(y_test, y_pred)     # Accuracy
    prec = precision_score(y_test, y_pred)   # Precision
    results.append([name, acc, prec])        # Save results

# Convert results into a nice DataFrame
import pandas as pd
results_df = pd.DataFrame(results, columns=['Model', 'Accuracy', 'Precision'])
print(results_df)
```

👉 Output will look like:

Model	Accuracy	Precision
SVC	0.92	0.89
KN	0.85	0.82
NB	0.91	0.87
…	…	…

What it does:

1. **Inputs**
    a. clf → the classifier (SVM, Naive Bayes, Random Forest, etc.)
    b. X_train, y_train → training data and labels
    c. X_test, y_test → test data and labels
2. **Train the model**

```
clf.fit(X_train, y_train)
```

    a. The model learns patterns from the training set.
3. **Make predictions**

```
y_pred = clf.predict(X_test)
```

    a. The trained model predicts labels for the test data.
4. **Evaluate performance**
    a. **Accuracy** → % of predictions that are correct

```
accuracy = accuracy_score(y_test, y_pred)
```

b. **Precision** → Out of all predicted "positives", how many were actually positive?
Useful when false positives are costly.

```
precision = precision_score(y_test, y_pred)
```

5. **Return results**
   a. The function gives back (accuracy, precision) for easy comparison across classifiers.

✅ Example usage:

```
acc, prec = train_classifier(clfs['SVC'], X_train, y_train, X_test, y_test)
print("SVC Accuracy:", acc)
print("SVC Precision:", prec)
```

So basically this function is a **reusable evaluation tool** for any classifier you plug in.

```
accuracy_scores = [] precision_scores = []

for name,clf in clfs.items():

current_accuracy,current_precision = train_classifier(clf, X_train,y_train,X_test,y_test)

print("For ",name)
print("Accuracy - ",current_accuracy)
print("Precision - ",current_precision)

accuracy_scores.append(current_accuracy)
precision_scores.append(current_precision)
```

1. **Initialize empty lists**

```
accuracy_scores = []
precision_scores = []
```

   a. These will store accuracy and precision values for each classifier.

2. **Loop through classifiers**

```
for name, clf in clfs.items():
```

a. clfs is your dictionary of classifiers:

```
clfs = {
  'SVC': svc, 'KN': knc, 'NB': mnb, 'DT': dtc,
  'LR': lrc, 'RF': rfc, 'AdaBoost': abc,
  'BgC': bc, 'ETC': etc, 'GBDT': gbdt, 'xgb': xgb
}
```

b. name → string key (like "SVC", "NB", etc.)
c. clf → actual classifier object (like SVC(kernel='sigmoid', gamma=1.0))

3. Train & evaluate each model

```
current_accuracy, current_precision = train_classifier(clf, X_train, y_train, X_test,
y_test)
```

a. Calls the function you wrote earlier.
b. train_classifier trains the model and returns **accuracy** & **precision** for this classifier.

4. Print results for each classifier

```
print("For ", name)
print("Accuracy - ", current_accuracy)
print("Precision - ", current_precision)
```

a. Helps you see performance model by model in the console.

5. Store results for later comparison

```
accuracy_scores.append(current_accuracy)
precision_scores.append(current_precision)
```

a. Saves all accuracy and precision values into lists, in the **same order** as clfs.

☑ Example output (say for 3 classifiers)

```
For  SVC
Accuracy -  0.94
Precision -  0.93

For  NB
Accuracy -  0.87
Precision -  0.85

For  RF
Accuracy -  0.96
Precision -  0.95
```

And after the loop:

```
accuracy_scores = [0.94, 0.87, 0.96, ...]
precision_scores = [0.93, 0.85, 0.95, ...]
```

```
performance_df =
pd.DataFrame({'Algorithm':clfs.keys(),'Accuracy':accuracy_scores,'Precision':precision_sc
ores}).sort_values('Precision',ascending=False)
```

1. **pd.DataFrame({...})**
    a. Creates a Pandas **DataFrame** (like a table).
    b. The dictionary has 3 keys: "Algorithm", "Accuracy", "Precision".
    c. Each key maps to a list/sequence of values.
    d. So you get a table like:

Algorithm	Accuracy	Precision
SVC	0.93	0.95
NB	0.85	0.88
RF	0.97	0.96
…	…	…

    e. Here:
        i. clfs.keys() → list of all algorithm names (SVC, NB, RF, etc.)
        ii. accuracy_scores → list of accuracy values for each classifier
        iii. precision_scores → list of precision values for each classifier
2. **.sort_values('Precision', ascending=False)**
    a. Sorts the DataFrame **by Precision column.**
    b. ascending=False means **highest precision comes first** (descending order).

Example output after sorting:

Algorithm	Accuracy	Precision
RF	0.97	0.96
SVC	0.93	0.95
NB	0.85	0.88

🎯 Why do this?

- You now have a clean table of all models and their performance.
- By sorting on **precision**, you can quickly see which model gives the **best precision** (important in spam detection, because you don't want to mark "ham" messages as spam).

What pd.melt does

- **melt** converts a **wide DataFrame** into a **long/tidy format**.
- It takes columns and "melts" them into **key-value pairs**.

Example

Suppose performance_df looks like this 👇

Algorithm	Accuracy	Precision
SVC	0.93	0.95
NB	0.85	0.88
RF	0.97	0.96

After melting:

```
performance_df1 = pd.melt(performance_df, id_vars="Algorithm")
```

You'll get:

Algorithm	variable	value

SVC	Accuracy	0.93
NB	Accuracy	0.85
RF	Accuracy	0.97
SVC	Precision	0.95
NB	Precision	0.88
RF	Precision	0.96

```
performance_df1 = pd.melt(performance_df, id_vars = "Algorithm")
```

☑ Why do this?

- The **melted format** is easier for plotting libraries (like **Seaborn** or **Matplotlib**) because you can compare **Accuracy vs Precision** side by side for each algorithm.
- Example visualization:

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.barplot(x="Algorithm", y="value", hue="variable", data=performance_df1)
plt.show()
```

This will give you grouped bars:

- One bar for **Accuracy**
- One bar for **Precision**
  for each algorithm.

```
temp_df=pd.DataFrame({'Algorithm':clfs.keys(),'Accuracy_max_ft_3000':accuracy_scores,'Precision_max_ft_3000':precision_scores}).sort_values('Precision_max_ft_3000',ascending=False)
```

Explanation:

1. **pd.DataFrame({...})**
   a. You are creating a new DataFrame using a dictionary.
   b. Each dictionary key becomes a column name in the DataFrame, and the values become the column data.

Columns being created:

c. "Algorithm" → takes the keys from your clfs dictionary (names of classifiers: 'SVC', 'KN', 'NB', etc.)
d. "Accuracy_max_ft_3000" → filled with values from your accuracy_scores list (accuracy of each classifier).
e. "Precision_max_ft_3000" → filled with values from your precision_scores list (precision of each classifier).

👉 The _max_ft_3000 suffix likely indicates that you trained these models using **a maximum of 3000 features** (like from TfidfVectorizer(max_features=3000) or similar).

2. **.sort_values('Precision_max_ft_3000', ascending=False)**
   a. This sorts the entire DataFrame **based on the "Precision_max_ft_3000" column.**
   b. ascending=False means it sorts from **highest precision to lowest precision.**
   c. This way, the classifier with the **best precision** appears first in the DataFrame.

☑ Example (imagine the data)

clfs.keys() → ['SVC','KN','NB']
accuracy_scores → [0.92, 0.88, 0.85]
precision_scores → [0.95, 0.80, 0.83]

Your DataFrame before sorting:

Algorithm	Accuracy_max_ft_3000	Precision_max_ft_3000
	0	0
SVC	0.92	0.95
KN	0.88	0.80
NB	0.85	0.83

After .sort_values('Precision_max_ft_3000', ascending=False):

Algorithm	Accuracy_max_ft_3000	Precision_max_ft_3000
	0	0
SVC	0.92	0.95
NB	0.85	0.83
KN	0.88	0.80

👉 So basically:
This code is building a **performance comparison table** of all classifiers (with accuracy & precision), and **ranking them by precision** (best → worst).

```
temp_df =
pd.DataFrame({'Algorithm':clfs.keys(),'Accuracy_scaling':accuracy_scores,'Precision_scaling':precision_scores}).sort_values('Precision_scaling',ascending=False)
```

```
new_df = performance_df.merge(temp_df,on='Algorithm')

new_df_scaled = new_df.merge(temp_df,on='Algorithm')
```

First line

```
new_df = performance_df.merge(temp_df,on='Algorithm')
```

- **performance_df** → your original DataFrame (probably without scaling), something like:

Algorithm	Accuracy	Precision
SVC	0.92	0.93
KN	0.87	0.79
NB	0.84	0.88

- **temp_df** → your DataFrame **after scaling**, something like:

Algorithm	Accuracy_scaling	Precision_scaling
SVC	0.91	0.92
KN	0.88	0.81
NB	0.86	0.89

- **.merge(temp_df, on='Algorithm')** → joins both DataFrames **row by row** based on the Algorithm column.

👉 Result (new_df) will combine both versions side by side:

Algorithm	Accuracy	Precision	Accuracy_scaling	Precision_scaling
SVC	0.92	0.93	0.91	0.92
KN	0.87	0.79	0.88	0.81
NB	0.84	0.88	0.86	0.89

2️⃣ Second line

```
new_df_scaled = new_df.merge(temp_df,on='Algorithm')
```

- Here, you are **again merging temp_df onto new_df**, still on the Algorithm column.
- But temp_df already got merged into new_df in the first step.

So after this line, you'll end up **duplicating** the scaling columns, with _x and _y suffixes that Pandas automatically adds to avoid name clashes.

👉 Example:

Algorithm	Accuracy	Precision	Accuracy_scaling_x	Precision_scaling_x	Accuracy_scaling_y	Precision_scaling_y
SVC	0.92	0.93	0.91	0.92	0.91	0.92
KN	0.87	0.79	0.88	0.81	0.88	0.81
NB	0.84	0.88	0.86	0.89	0.86	0.89

☑️ In summary

- The **first merge** (new_df) is correct → it lets you compare performance **before vs. after scaling** in one table.
- The **second merge** (new_df_scaled) just duplicates columns unnecessarily → probably not what you want.

```
temp_df =
pd.DataFrame({'Algorithm':clfs.keys(),'Accuracy_num_chars':accuracy_scores,'Precision_num
_chars':precision_scores}).sort_values('Precision_num_chars',ascending=False)
```

1. **pd.DataFrame({...})**
    a. You're creating a new pandas DataFrame with 3 columns:
        i. **Algorithm** → the names of your models (clfs.keys()), e.g. ['SVC','KN','NB',...].

ii. **Accuracy_num_chars** → the list of accuracy values stored in accuracy_scores.
iii. **Precision_num_chars** → the list of precision values stored in precision_scores.

Example (before sorting):

Algorithm	Accuracy_num_chars	Precision_num_chars
SVC	0.95	0.94
KN	0.88	0.83
NB	0.91	0.89
DT	0.85	0.80

2. **.sort_values('Precision_num_chars', ascending=False)**
   a. This sorts the DataFrame **by the precision column** (Precision_num_chars) in **descending order**.
   b. So the algorithm with the **highest precision** appears at the top.

Example (after sorting):

Algorithm	Accuracy_num_chars	Precision_num_chars
SVC	0.95	0.94
NB	0.91	0.89
KN	0.88	0.83
DT	0.85	0.80

☑ Why the column names have _num_chars

- This tells you **what feature set or experiment setup** these scores came from.
- Looks like here, you trained your classifiers using the **num_chars feature** (maybe message length for spam detection).
- That's why you named the columns Accuracy_num_chars and Precision_num_chars, so later you can compare with other setups (like scaling, max_ft_3000, etc.).

👉 In short: this code creates a **results table for your classifiers when trained on the num_chars feature**, and sorts it so you can quickly see **which algorithm gave the best precision**.

```
new_df_scaled.merge(temp_df,on='Algorithm')
```

1. **ew_df_scaled**
    a. This is already a DataFrame you built earlier by merging performance_df and temp_df.
    b. It likely contains columns like:

```
Algorithm | Accuracy | Precision | Accuracy_scaling | Precision_scaling
```

2. **.merge(temp_df, on='Algorithm')**
    a. You're merging (joining) another DataFrame (temp_df) with new_df_scaled.
    b. The join key is **Algorithm** → meaning rows will align based on the algorithm names (like SVC, NB, etc.).
    c. temp_df in your case has columns like:

```
Algorithm | Accuracy_num_chars | Precision_num_chars
```

3. **Result after merge**
    a. After this merge, you'll get a **bigger DataFrame** containing results from both DataFrames:

Example:

Algorithm	Accuracy	Precision	Accuracy_scaling	Precision_scaling	Accuracy_num_chars	Precision_num_chars
SVC	0.95	0.94	0.96	0.95	0.92	0.91
NB	0.91	0.89	0.90	0.87	0.89	0.88
KN	0.88	0.83	0.85	0.81	0.84	0.80

☑ Why do this?

- This way you're **consolidating all experiment results (different feature engineering techniques)** into a **single comparison table**.
- Now you can easily compare:
    - Accuracy vs. Accuracy_scaling vs. Accuracy_num_chars
    - Precision vs. Precision_scaling vs. Precision_num_chars

Voting Classifier

svc = SVC(kernel='sigmoid', gamma=1.0,probability=True) mnb = MultinomialNB() etc = ExtraTreesClassifier(n_estimators=50, random_state=2)

from sklearn.ensemble import VotingClassifier

👉 Basically, this line is just **combining multiple experiment results into one big DataFrame so you can analyze them side by side.**

- A **VotingClassifier** is an **ensemble method** in machine learning.
- It combines the predictions of multiple models (called *base learners*) to make a final decision.
- Instead of relying on a single classifier, it uses the **"wisdom of the crowd"** idea: combining several models usually gives better performance and stability.

✨ Two Main Types of Voting

1. **Hard Voting (default)**
    a. Each classifier makes a prediction (e.g., spam vs not spam).
    b. The final prediction is the **majority vote**.
    c. Example:
        i. SVC → Spam
        ii. Naive Bayes → Spam
        iii. ExtraTrees → Not Spam
        → Final Result = Spam (since 2 out of 3 say Spam).
2. **Soft Voting**
    a. Works when classifiers can output **probability scores** (that's why you set probability=True for SVC ✅).
    b. Instead of simple majority, it takes the **average probability** of each class across all models and picks the class with the highest probability.
    c. This is usually more accurate, because it considers the **confidence** of each model.

🧩 Why use VotingClassifier here?

You defined three different models:

```
svc = SVC(kernel='sigmoid', gamma=1.0, probability=True)
mnb = MultinomialNB()
etc = ExtraTreesClassifier(n_estimators=50, random_state=2)


#this 3 models gave me the most accuracy and precision
```

- Each model has **different strengths**:
  - **SVC**: good with complex decision boundaries.
  - **MultinomialNB**: fast and works well with text data (especially word frequencies).
  - **ExtraTrees**: a strong ensemble tree-based method.

By combining them in a **VotingClassifier**, you reduce the chance that a single weak model will make the wrong prediction. The final model is usually:

- **More robust**
- **Less prone to overfitting**
- **Better generalization on unseen data**

☑ Example of usage:

```
voting_clf = VotingClassifier(
    estimators=[('svc', svc), ('mnb', mnb), ('etc', etc)],
    voting='soft'  # 'hard' for majority voting
)

voting_clf.fit(X_train, y_train)
y_pred = voting_clf.predict(X_test)
```

👉 So, in short:
You use **VotingClassifier** to **combine multiple algorithms** into one strong classifier, leveraging their **diverse strengths** to achieve better accuracy and precision.

```
voting = VotingClassifier(estimators=[('svm', svc), ('nb', mnb), ('et', etc)],voting='soft')
```

1. **otingClassifier(...)**

a. You are creating an **ensemble classifier** that combines multiple different models.
b. Instead of relying on one algorithm, it will use all three and make a final decision.

2. **estimators=[('svm', svc), ('nb', mnb), ('et', etc)]**
    a. You pass a list of models to the VotingClassifier.
    b. Each entry is a tuple:
        i. First part = a **name** (string identifier for the model).
        ii. Second part = the actual **model object** you defined earlier.

So here:

c. 'svm' → svc (Support Vector Classifier with sigmoid kernel).
d. 'nb' → mnb (Multinomial Naive Bayes).
e. 'et' → etc (Extra Trees Classifier).

These are your **base learners**.

3. **voting='soft'**
    a. This tells the VotingClassifier to use **soft voting** instead of hard voting.
    b. **Soft voting**:
        i. Each model outputs **probabilities** for each class.
        ii. The classifier then takes the **average probability** across models and selects the class with the highest probability.
        iii. Example:
            1. SVC says: [Spam: 0.6, Not Spam: 0.4]
            2. NB says: [Spam: 0.8, Not Spam: 0.2]
            3. ET says: [Spam: 0.4, Not Spam: 0.6]
            4. Average: [Spam: (0.6+0.8+0.4)/3 = 0.6, Not Spam: 0.4]
            5. Final Prediction = Spam ✅
    c. This usually works better than **hard voting** (which just takes the majority vote) because it considers the **confidence level** of each model.

☑ So in simple words:

This line creates an **ensemble classifier** that:

- Combines **SVC**, **Naive Bayes**, and **Extra Trees**.
- Uses **soft voting** (averaging probabilities) to decide the final prediction.
- Makes your model more **robust** and often **more accurate** than using just one classifier.

```
y_pred = voting.predict(X_test) print("Accuracy",accuracy_score(y_test,y_pred))
print("Precision",precision_score(y_test,y_pred))
```

1. **y_pred = voting.predict(X_test)**
    a. Here, you use your trained **VotingClassifier** (voting) to make predictions on the **test data** (X_test).
    b. y_pred will be an array of predicted labels (spam/ham, or 0/1 depending on your dataset).

Example:

```
y_pred = [0, 1, 0, 0, 1, ...]
```

2. **accuracy_score(y_test, y_pred)**
    a. This compares the **true labels** (y_test) with the **predicted labels** (y_pred).
    b. Formula for accuracy:

*Accuracy=Number of correct predictionsTotal number of predictions\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}*Accuracy=Total number of predictionsNumber of correct predictions

    c. If you had 100 test samples and 90 were predicted correctly → Accuracy = 0.90 (90%).

3. **precision_score(y_test, y_pred)**
    a. Precision measures **how many of the predicted positives were actually correct.**
    b. Formula:

*Precision=TPTP+FP\text{Precision} = \frac{TP}{TP + FP}*Precision=TP+FPTP

where:

        i. **TP (True Positives)** = Correctly predicted positive cases (e.g., correctly predicted spam).
        ii. **FP (False Positives)** = Incorrectly predicted positive cases (e.g., said "spam" but it was not spam).
    c. High precision means your model makes **few false alarms.**
    d. Example: If the model predicts 100 emails as spam, and 90 are actually spam → Precision = 0.90.

☑ In simple words:

- `voting.predict(X_test)` → predicts the labels for unseen data.
- `accuracy_score(...)` → tells you **how many predictions were correct overall.**
- `precision_score(...)` → tells you **how reliable the positive (spam) predictions are.**

Examples -

Spam msg - URGENT! You have won a 1 week FREE vacation to Bahamas. Reply YES to claim.
This is returning spam

congratulations!! you've won a free ticket. call now!!!-this should return spam but its returning not spam.

WINNER!! As a valued customer you have been selected to receive a FREE $1000 Walmart gift card. Call 1800123456 now!this is returning spam

Get loans approved instantly! No credit check required. Apply now!!! This should return spam but its returning not spam

improving this model

Explanation

1. Imports

- WordNetLemmatizer: Used to reduce words to their base form (e.g., *running → run, mice → mouse*).
- stopwords: Common words like *the, is, and* that don't add meaning for classification.
- string: Gives punctuation characters so we can remove them.

2. Download resources

- punkt: tokenizer for splitting text into words.
- stopwords: list of common useless words.
- wordnet & omw-1.4: dictionaries needed for lemmatizer.

3. Lowercasing

```
text = text.lower()
```

👉 Converts everything to lowercase so Spam and spam are treated the same.

4. Tokenization

```
text = nltk.word_tokenize(text)
```

👉 Breaks the input string into a list of words (tokens).
Example: "Free money now!" → ["free", "money", "now", "!"]

5. Keep only alphanumeric

```
y = []
for i in text:
    if i.isalnum():
        y.append(i)
```

👉 Removes emojis, symbols, and other junk. Keeps only letters/numbers.
 Example: "free!" → "free"

6. Remove stopwords & punctuation

```
text = []
for i in y:
    if i not in stopwords.words('english') and i not in string.punctuation:
        text.append(i)
```

👉 Removes words like *"is", "the", "and"* and punctuation.
 Example: "you are the winner!" → ["winner"]

7. Lemmatization

```
lemmatized_words = []
for i in text:
    lemmatized_words.append(lemmatizer.lemmatize(i))
```

👉 Converts words to their **base dictionary form**:

- *“running” → “run”*
- *“mice” → “mouse”*
- *“better” → “good”*

8. Reconstruct the text

```
return " ".join(lemmatized_words)
```

👉 Joins all cleaned words back into one string.
Example: "You are running faster than mice!" → "run fast mouse"

☑ Why this is useful

- Reduces noise (removes stopwords & symbols).
- Normalizes words (lemmatization groups variations together).
- Makes your ML model focus on **meaningful tokens**, improving accuracy.