

Pandas axis usage



```
X=df.drop('label',axis=1)
```

axis=1 what significance does it bear and if not use what will happen?

In the line:

```
python
```

```
X = df.drop('label', axis=1)
```

◆ `axis=1` Explanation:

- In **Pandas**, `axis=0` means **rows**, and `axis=1` means **columns**.
- So `axis=1` tells Pandas: "**Drop the 'label' column**" (not a row).

◆ What happens if you don't use `axis=1`?

If you write:

```
python
```

```
X = df.drop('label')
```

Then **by default, Pandas will assume `axis=0`**, i.e., it will try to drop **a row** with the index `'label'`.

This will:

- **Throw an error** if `'label'` is not an index in the rows:

```
vbnet
```

```
KeyError: "['label'] not found in axis"
```

✓ Summary:

Syntax	Effect
<code>df.drop('label', axis=1)</code>	✓ Drops the column named <code>'label'</code>
<code>df.drop('label')</code>	✗ Tries to drop a row named <code>'label'</code> , likely error

Would you like a visual example with a sample DataFrame?



messages.reset_index(inplace=True) what does inplace=True means??

Great question!

In this line:

```
python

messages.reset_index(inplace=True)
```

◆ What `inplace=True` means:

It tells Pandas to **perform the operation directly on the existing DataFrame** (i.e., **modify it in-place**) **without creating a new one**.

✓ What happens with and without `inplace=True`:

1. With `inplace=True`:

```
python

messages.reset_index(inplace=True)
```

- The index is reset **inside** the original `messages` DataFrame.
- No new variable is created.
- The `messages` object is **updated** directly.

2. Without `inplace=True`:

```
python

new_df = messages.reset_index()
```

- The operation returns a **new DataFrame** with the reset index.
- The original `messages` DataFrame stays unchanged unless you assign it.

🧠 In Short:

- `inplace=True`: **Modify the original** object directly.
- `inplace=False` (default): **Return a new** object, and the original stays the same.

Would you like a code example to visualize the difference?



```
## Applying Countvectorizer
# Creating the Bag of Words model
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer(max_features=5000,ngram_range=(1,3))
X = cv.fit_transform(corpus).toarray()
```

You're creating a **Bag of Words (BoW)** model using `CountVectorizer`, a common technique in natural language processing (NLP). Let's break it down:

What this code does:

```
python

from sklearn.feature_extraction.text import CountVectorizer

cv = CountVectorizer(max_features=5000, ngram_range=(1,3))
X = cv.fit_transform(corpus).toarray()
```

1. `CountVectorizer`:

- Converts a **collection of text documents** (`corpus`) into a **matrix of token counts** (i.e., how many times each word/phrase appears).
- This is your **Bag of Words** model.

2. Key Parameters Explained:

- `max_features=5000`:
 - Keeps only the **top 5000 most frequent words/phrases**.
 - Helps reduce dimensionality and computation cost.
- `ngram_range=(1,3)`:
 - Extracts **unigrams (1 word)**, **bigrams (2 words)**, and **trigrams (3 words)**.
 - For example, from `"I love NLP"`, it generates:
 - Unigrams: `"I"`, `"love"`, `"NLP"`
 - Bigrams: `"I love"`, `"love NLP"`
 - Trigram: `"I love NLP"`

3. `fit_transform(corpus).toarray()`:

- `fit_transform(corpus)`:
 - Learns the vocabulary and creates the **document-term matrix**.
- `.toarray()`:
 - Converts the sparse matrix to a **regular NumPy array**.

✓ Output:

- `x` becomes a **2D NumPy array** where:
 - Each row = one document (from `corpus`)
 - Each column = count of a specific word/ngram (from top 5000)

Would you like help visualizing how the output matrix looks or how to interpret it?



```
ngram_range=(1,3):
```

explain these with full example?

Absolutely! Let's break down `ngram_range=(1, 3)` with a **simple, full example** so you can understand how **unigrams, bigrams, and trigrams** are created from text using `CountVectorizer`.

♦ What is `ngram_range=(1, 3)`?

It means:

- Extract **unigrams** (single words),
- **bigrams** (two consecutive words),
- **trigrams** (three consecutive words)

All from the input text.

✓ Full Example:

✂ Sample input:

```
python

corpus = ["I love machine learning"]
```

⚙ Code:

```
python

from sklearn.feature_extraction.text import CountVectorizer

cv = CountVectorizer(ngram_range=(1, 3))
X = cv.fit_transform(corpus).toarray()
print(cv.get_feature_names_out())
print(X)
```

Output Breakdown:

 `cv.get_feature_names_out():`


This will show:

```
python

['i' 'i love' 'i love machine' 'learning' 'love' 'love machine' 'love machine learning' 'machine' 'r
```

These are:

- **Unigrams:** `'i'`, `'love'`, `'machine'`, `'learning'`
- **Bigrams:** `'i love'`, `'love machine'`, `'machine learning'`
- **Trigrams:** `'i love machine'`, `'love machine learning'`

 `X`:

This is a matrix with **1 row** (for 1 document) and **9 columns** (one for each n-gram):

```
python

[[1 1 1 1 1 1 1 1 1]]
```

Each `1` indicates the presence of the corresponding n-gram in the text.

Another Example:

Input:

```
python

corpus = ["AI is smart", "AI is the future"]
cv = CountVectorizer(ngram_range=(1, 2))
```

The n-grams will be:

- Unigrams: `'ai'`, `'is'`, `'smart'`, `'the'`, `'future'`
- Bigrams: `'ai is'`, `'is smart'`, `'is the'`, `'the future'`

Summary:

Range	Type of N-Gram	Example from <code>"I love ML"</code>
<code>(1, 1)</code>	Unigram only	<code>'I'</code> , <code>'love'</code> , <code>'ML'</code>
<code>(1, 2)</code>	Unigram + Bigram	<code>'I'</code> , <code>'love'</code> , <code>'I love'</code> , etc
<code>(1, 3)</code>	Unigram + Bigram + Trigram	<code>'I'</code> , <code>'I love'</code> , <code>'I love ML'</code>

Do you want a visual diagram showing how these n-grams are generated?



```
ngrams(2,2)
```

Great! Let's focus on this specific case:

◆ What does `ngram_range=(2, 2)` mean?

It tells `CountVectorizer` to extract **only bigrams** from the text.

- `2` is the minimum size of the n-gram
- `2` is also the maximum size

So only **2-word sequences** (bigrams) will be included.

✓ Full Example:

```
python

from sklearn.feature_extraction.text import CountVectorizer
corpus = ["I love machine learning and AI"]
cv = CountVectorizer(ngram_range=(2, 2)) # Only bigrams
X = cv.fit_transform(corpus).toarray()

print(cv.get_feature_names_out())
print(X)
```

📋 Output:

Bigrams generated:

```
python

['ai' 'and ai' 'learning and' 'love machine' 'machine learning']
```

(Actual output may show slightly different order depending on version.)

Each bigram is made by sliding a 2-word window over the text.

How they were created:

From `"I love machine learning and AI"`:

1. `I love`
2. `love machine`
3. `machine learning`
4. `learning and`

5. `and AI`

But:

- Some tokens may be filtered (like lowercase/stopwords) based on tokenizer or preprocessor settings.

Result Matrix (`x`):

This will show a **row vector** (for the document) with `1`s for present bigrams.

Example:

```
python

[[1 1 1 1 1]]
```

Each `1` means that the bigram exists in the document.

Summary:

Parameter	Result
`ngram_range=(2, 2)`	Only bigrams (two-word phrases)
Example	`"I love"` → bigram
Not Included	Unigrams (`"I"`, `"love"`) or trigrams (`"I love ML"`)

Would you like an example with multiple documents or visualization of the matrix?