Word2Vec

```
%pip install gensim
```

💡 **!!Restart the kernel after installing.**

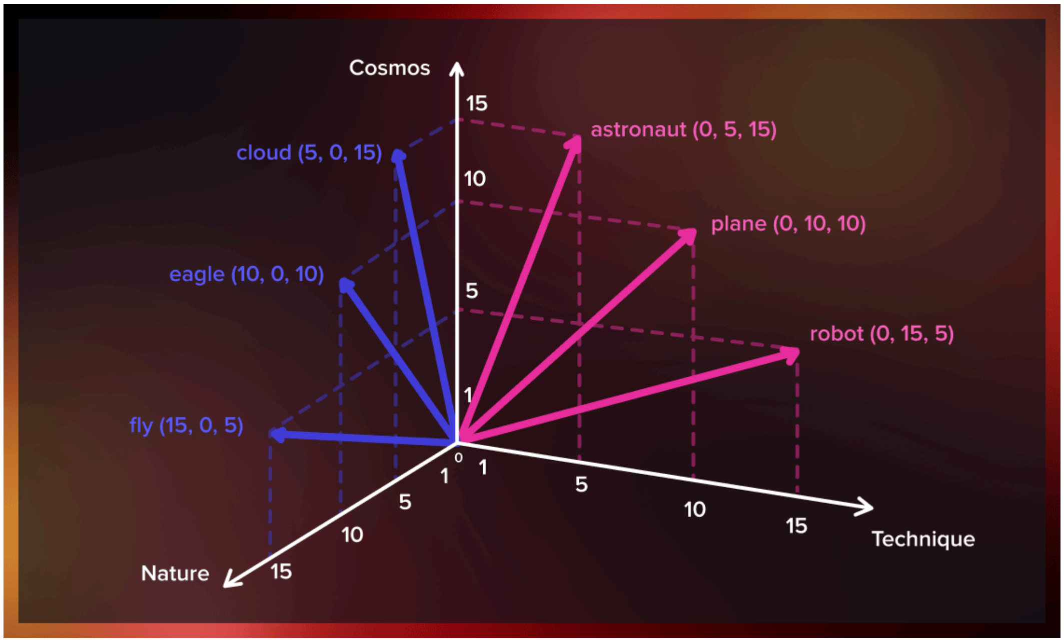
```
from gensim.models import Word2Vec
```

✅ What is Word2Vec?

> Word2Vec is a technique that turns words into numbers (vectors) in such a way that words with similar meanings are close together in vector space.

So:

- "king" and "queen" → have similar vectors
- "apple" and "banana" → also similar
- "apple" and "president" → far apart



🎯 WHY Word2Vec?

Before Word2Vec, we used **one-hot encoding**:

```
apple     = [1, 0, 0, 0, 0, 0]
banana    = [0, 1, 0, 0, 0, 0]
president = [0, 0, 0, 1, 0, 0]
```

- All vectors were same size as vocabulary
- No **meaning**, no **similarity**, just 1s and 0s

> Word2Vec solves this by learning **meaningful vectors** based on context.

🧠 Key Idea: *Meaning Comes from Context*

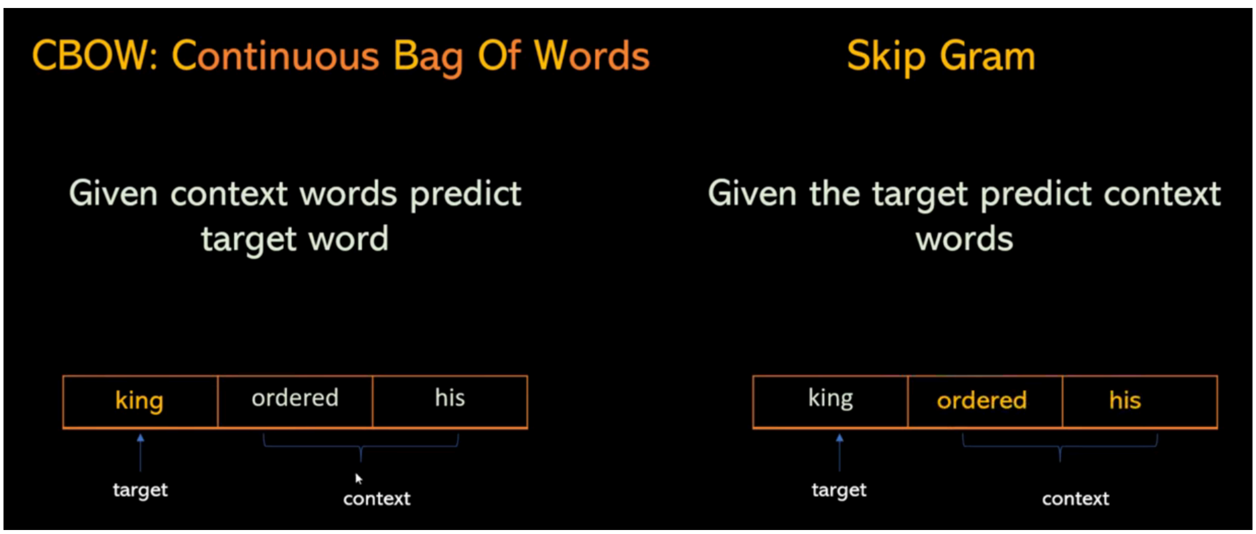
> “You shall know a word by the company it keeps.” — John Firth

- So, if “king” often appears near “royal”, “palace”, “queen”, “monarch”...
- And “queen” appears near the same words...
- Then **king and queen must be related**.

🔧 How It Works?

Two Architectures

Model	Description	Example Use Case
CBOW	Predicts a target word from context words	Faster training
Skip-gram	Predicts context words from a target word	Better for rare words



1. CBOW (Continuous Bag of Words)

> Predict the target word from context words

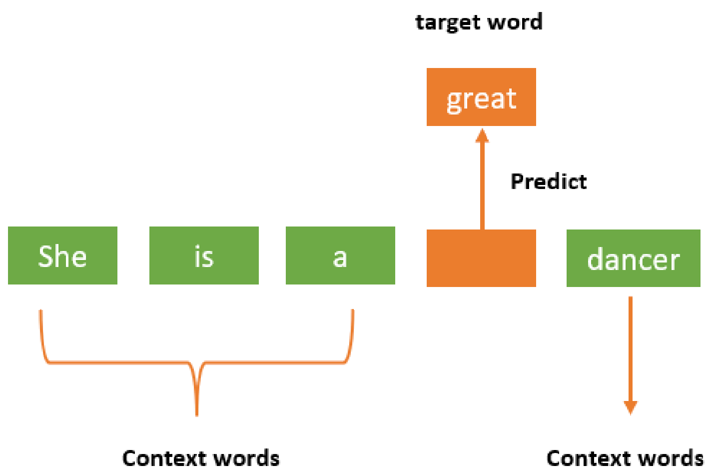
💡 **CBOW is a fully connected neural network.**

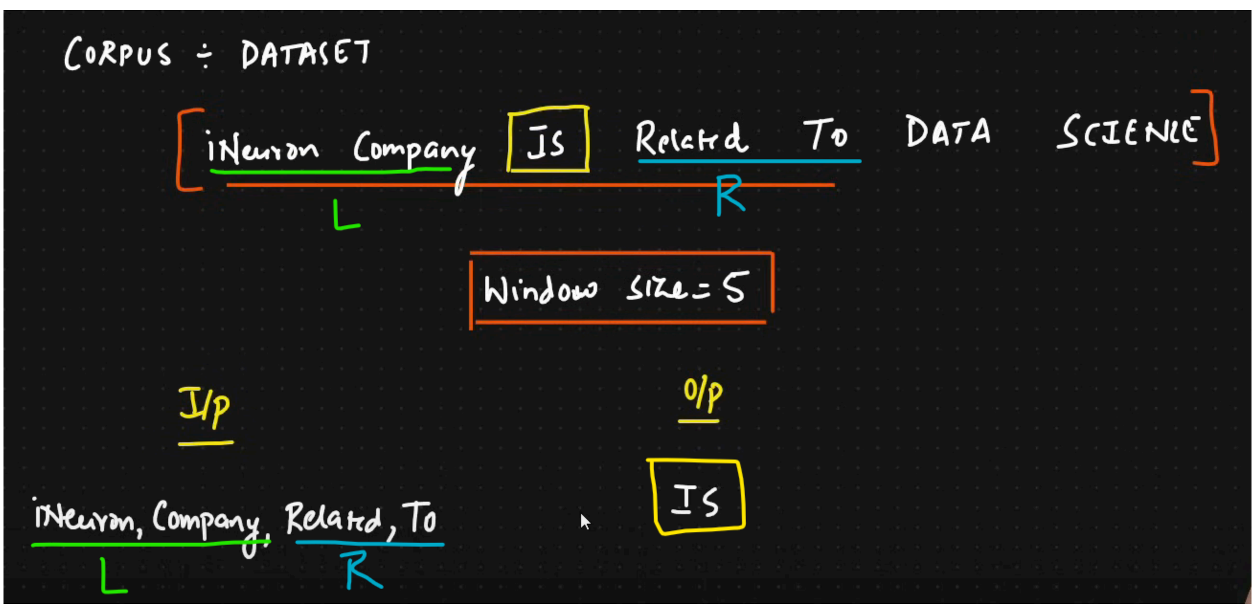
📘 Example:

```
Input  : ["I", "love", "to", ___, "apples"]
Output : "eat"
```

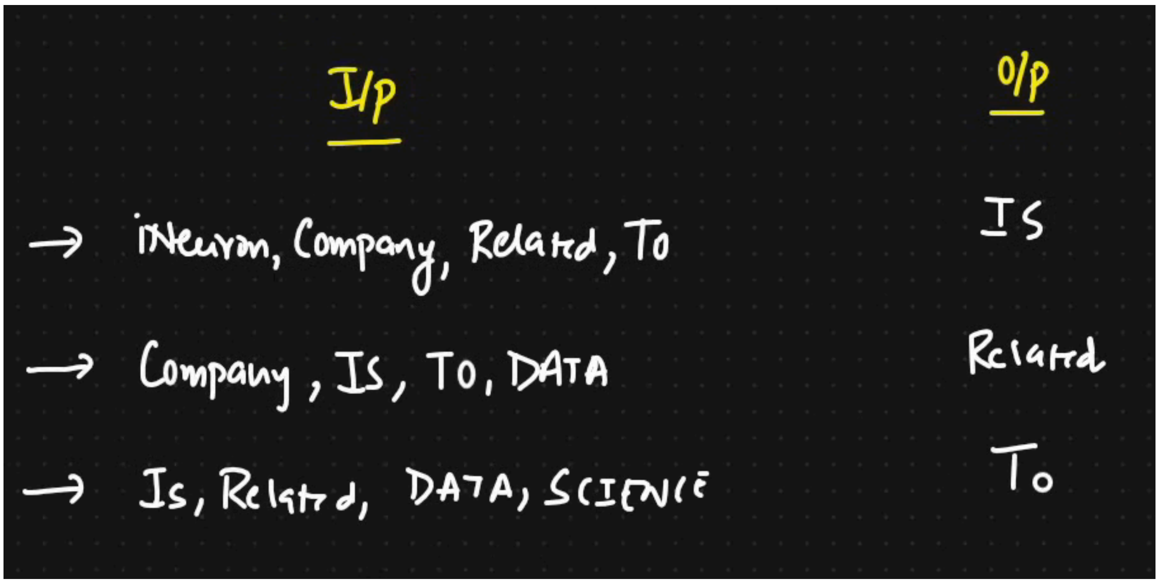
It learns to say:

"If the words around it are these, the center word is probably 'eat'."



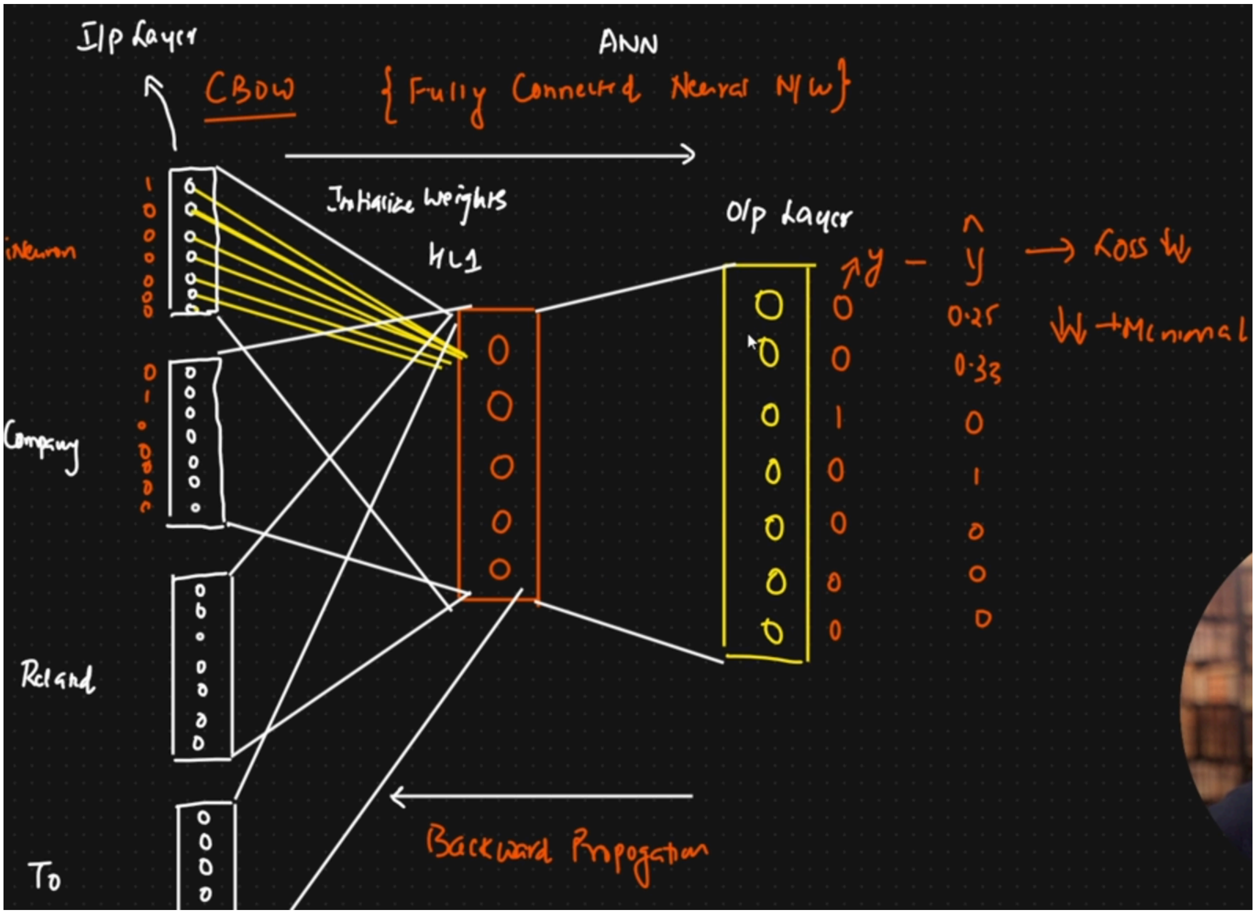


- We define a window size
- It takes words to left an right of the **output word**
    - In above example, **“is”** is **output word**
- It does this to to know what words are in the forward & backword context.
- We move this window and take the next 5 words.



- 👆We need to convert this into vectors to sent them to neural networks.
    - We do it with **OHE**

- After this, you’ll for a fully connected neural network



- Last layer is **softmax** layer
- In this way, the prediction for the hidden word is done.

2. Skip-Gram

- It is opposite of CBOW
- Here also you have window of number size & you slide this window

> Predict context words from the center word

📘 Example:

```
Input  : "eat"
Output : ["I", "love", "to", "apples"]
```

It learns to say:

"If the center word is ‘eat’, the words around it are likely to be this."

> 💡 **The architecture of skip-gram is reverse of CBOW.**

When to use what?

- Small data→ CBOW
- Large data → Skip-Gram

Improve the performance

- Increase the training data
- Increase the vector dimension
- Increase the window size (More time)

🧱 What does the model look like?

It’s a **tiny neural network with 1 hidden layer**.

- **Input**: **One-hot encoded word**
- **Hidden**: **Embedding layer** (the goal)
- **Output**: **Prediction** of target/context word

> After training, we take the weights from the hidden layer — those are the word vectors!

📌 Shape of the word vector?

Typically:

- `100` or `300` dimensions (you choose)
- **A smaller number than vocabulary size**
- Each word gets a dense, meaningful vector

🔥 Real Example: Vector Arithmetic

After training:

```
vector("king") - vector("man") + vector("woman") ≈ vector("queen")
```

Python Code:

🔧 1. Install Gensim

```
pip install gensim
```

Using Gensim (Recommended)

```
from gensim.models import Word2Vec

# Sample sentences
sentences = [
    ["the", "cat", "sat", "on", "the", "mat"],
    ["dogs", "are", "good", "pets"]
]

# Train model
model = Word2Vec(sentences, vector_size=100, window=5, min_count=1, sg=
1)  # sg=1 for skip-gram (default sg=0)

# Get word vector
cat_vector = model.wv["cat"]  # shape: (100,)

# Find similar words
```

```
similar_words = model.wv.most_similar("cat", topn=3)
# Output: [('dog', 0.92), ('mat', 0.85), ('sat', 0.78)]

print(similar_words)
```

```
[('good', 0.19912061095237732),
 ('are', 0.07497556507587433),
 ('mat', 0.0605918429791927 34)]
```

Parameter	Effect	Typical Value
`vector_size`	Dimension of word vectors	50-300
`window`	How many words around the target to use	2-10
`min_count`	Ignores rare words below this count	5-20
`sg`	0=CBOW, 1=Skip-gram	0 or 1

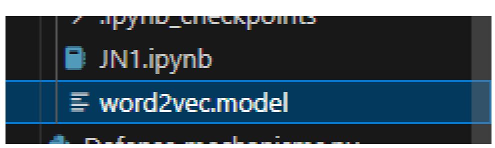
Example 2:

```
from gensim.models import Word2Vec

# Sample data
sentences = [["this", "is", "a", "sample"], ["we", "are", "learning", "word2vec"],
["the", "cat", "sat", "on", "the", "mat"],
   ["dogs", "are", "good", "pets"]]

# Train the model
model = Word2Vec(sentences, vector_size=100, window=5, min_count=1, wor
kers=4)

# Save the model
model.save("word2vec.model")
```



- **Now use the saved model:**

```
# Load model
model = Word2Vec.load("word2vec.model")

# Get vector for a word
vector = model.wv['sample']

# Find similar words
print(model.wv.most_similar('learning'))

# Word analogy
print(model.wv.most_similar(positive=['king', 'woman'], negative=['man']))
```

```
[('sample', 0.12813477218151093), ('we', 0.10941850394010544), ('sat', 0.1088925376534462), ('the', 0.06285077333450317), ('on', 0.0504
[('pets', 0.13342435657978058), ('are', 0.09861470758914948), ('word2vec', 0.09118345379829407), ('mat', 0.07821463793516159), ('good',
```

🔧 Gensim `Word2Vec` Default Parameters:

```
Word2Vec(
    sentences=None,       # Iterable of tokenized sentences
    vector_size=100,      # Dimensionality of word vectors
    window=5,             # Max distance between current and predicted word
    min_count=5,          # Ignores words with total frequency lower than this
    workers=3,            # Number of threads to use during training
    sg=0,                 # 0 = CBOW, 1 = Skip-gram
    hs=0,                 # 1 = Hierarchical Softmax, 0 = negative sampling
    negative=5,           # If > 0, negative sampling will be used
    epochs=5,             # Number of iterations (epochs) over the corpus
    seed=1,               # Random seed for reproducibility
)
```

Parameter	Description
`vector_size`	Size of each word vector (embedding dimension).

Parameter	Description
`window`	Context window size — how many words left and right to consider.
`min_count`	Words with lower frequency than this are ignored.
`workers`	CPU cores to use for training (parallelism).
`sg`	Training algorithm: 0 = CBOW (default), 1 = Skip-gram.
`hs`	Use Hierarchical Softmax instead of negative sampling.
`negative`	Number of negative samples used (only if `hs=0`).
`epochs`	Training passes over the data.
`seed`	Random seed for reproducibility

Option 2: Use it in Keras Embedding layer

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding

# Suppose embedding_matrix is loaded from Word2Vec (shape: vocab_size x 300)

model = Sequential()
model.add(Embedding(
    input_dim=vocab_size,
    output_dim=300,
    weights=[embedding_matrix],
    trainable=False  # keep pretrained vectors fixed
))
```

Defaults:

```
keras.layers.Embedding(
    input_dim,
    output_dim,
    embeddings_initializer="uniform",
    embeddings_regularizer=None,
```

```
    embeddings_constraint=None,
    mask_zero=False,
    weights=None,
    lora_rank=None,
    **kwargs
)
```

📚**Option 3: Use Pretrained Models (Optional)**

```
from gensim.models import KeyedVectors

# Download Google News pretrained Word2Vec (3M words, 300-dim)
# Note: ~1.5GB
model = KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bin', binary=True)

print(model.most_similar('computer'))
```

📁 Use Cases

- NLP classification (sentiment, spam, etc.)
- Document similarity
- Chatbots
- Search engines

🧠 Trivia

- Invented by **Google in 2013**
- Learns using **negative sampling** or **hierarchical softmax** (for speed)
- Trained on **Google News** dataset (100B words)

Advantages vs. Limitations

Pros	Cons
Simple and fast to train	Can't handle out-of-vocabulary words
Captures semantic relationships	No contextual meaning (unlike BERT)
Works well with small data	Single vector per word (no polysemy

Phone Reviews Word2Vec

```
import gensim
import pandas as pd
```

```
# Use the raw URL for the file
url = "https://raw.githubusercontent.com/masta-g3/amazon_nlp/master/Cell_Phones_and_Accessories_5.json.gz"

# Read the gzip file directly
df = pd.read_json(url, lines=True)
df
```

	reviewerID	asin	reviewerName	helpful	reviewText	overall	summary	unixReviewTime	reviewTime
0	A30TL5EWN6DFXT	120401325X	christina	[0, 0]	They look good and stick good! I just don't li...	4	Looks Good	1400630400	05 21, 2014
1	ASY55RVNIL0UD	120401325X	emily l.	[0, 0]	These stickers work like the review says they ...	5	Really great product.	1389657600	01 14, 2014
2	A2TMXE2AFO7ONB	120401325X	Erica	[0, 0]	These are awesome and make my phone look so st...	5	LOVE LOVE LOVE	1403740800	06 26, 2014
3	AWJ0WZQYMYFQ4	120401325X	JM	[4, 4]	Item arrived in great time and was in perfect ...	4	Cute!	1382313600	10 21, 2013
4	ATX7CZYFXI1KW	120401325X	patrice m rogoza	[2, 3]	awesome! stays on, and looks great. can be use...	5	leopard home button sticker for iphone 4s	1359849600	02 3, 2013
...	...	...	...	...	...	...	...	...	...
194434	A1YMNTFLNDYQ1F	B00LORXVUE	eyeused2loveher	[0, 0]	Works great just like my original one. I reall...	5	This works just perfect!	1405900800	07 21, 2014

194439 rows × 9 columns

- We are interested only in **reviewText** column

- We need to remove the stop words, convert to lower case, remove punctuations
    - Use → `gensim.utils.simple_preprocess()`

```
gensim.utils.simple_preprocess("They look good and stick good! I just don't li
ke the rounded shape because I was always bumping it and Siri kept popping
up and it was irritating. I just won't buy a product like this again")
```

```
['they',
 'look',
 'good',
 'and',
 'stick',
 'good',
 'just',
 'don',
 'like',
 'the',
 'rounded',
 'shape',
 'because',
 'was',
 'always',
 'bumping',
 'it',
 'and',
 'siri',
 'kept',
 'popping',
 'up',
 'and',
 'it',
 'was',
...
```

Apply this on the entire column:

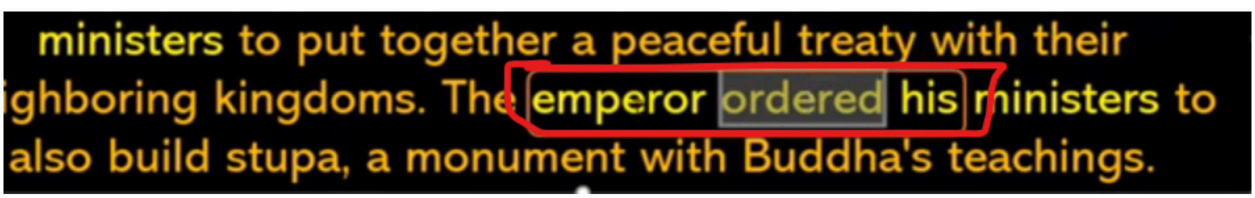
```
review_text = df.reviewText.apply(gensim.utils.simple_preprocess)
review_text
```

```
0         [they, look, good, and, stick, good, just, don...
1         [these, stickers, work, like, the, review, say...
2         [these, are, awesome, and, make, my, phone, lo...
3         [item, arrived, in, great, time, and, was, in,...
4         [awesome, stays, on, and, looks, great, can, b...
                                ...
194434    [works, great, just, like, my, original, one, ...
194435    [great, product, great, packaging, high, quali...
194436    [this, is, great, cable, just, as, good, as, t...
194437    [really, like, it, becasue, it, works, well, w...
194438    [product, as, described, have, wasted, lot, of...
Name: reviewText, Length: 194439, dtype: object
```

Create a **Word2Vec** model:

```
model= gensim.models.Word2Vec(
    window=10,
    min_count=2,
    workers=4
)
```

`window=10` → Take 10 words before & after the target word



`min_count=2` → At least 2 words for a review to be qualified as a sentence

`workers=4` → CPU threads

- Build a unique words' vocabulary.

```
model.build_vocab(review_text)
```

Train the model:

`model.corpus_count` = 194439 (No. of reviews)

`model.epochs` = 5

```
model.train(review_text, total_examples=model.corpus_count, epochs=5)
```

```
(61499861, 83868975)
```

Save the model:

```
model.save('wvmodel.model')
```

- You can use anywhere else

Find out most similar words:

```
model.wv.most_similar('buy')
```

```
[('purchase', 0.7886890769004822),
 ('buying', 0.6954835057258606),
 ('reorder', 0.6249381899833679),
 ('order', 0.6123434901237488),
 ('bet', 0.6065912842750549),
 ('purchasing', 0.5726720094680786)
 ('invest', 0.5687201619148254),
 ('hesitate', 0.5634164810180664),
 ('ordering', 0.5627203583717346),
 ('spend', 0.5543853640556335)]
```

Find out similarity score:

```
model.wv.similarity('great', 'good')
```

```
0.77926606
```

Average Word2Vec

> 💡 Creates **document-level embeddings**

Average Word2Vec means:

- Take the **vector** of each word in a sentence (from Word2Vec),
- Then **average** all those vectors (element-wise),
- So you get **one final vector** representing the entire sentence.
- This approach extends word-level embeddings to longer texts while preserving much of the semantic meaning.

🧠 **This is often used for:**

- **Sentence classification**,
- **Spam detection**,
- **Similarity comparison**,
- When you need a fixed-size input for ML models (like logistic regression or SVM).

🎯 Why Do We Need This?

Because:

- Word2Vec gives a **vector per word** (e.g. 300 values).
- But ML models need **one vector per sentence/document**.

So we do:

```
sentence vector = average of all its word vectors
```

This is called **document embedding** or **sentence embedding**, made using Word2Vec.

🔣 Example:

Sentence:

```
"I love pizza"
```

Step-by-step:

1. Get vectors from pre-trained Word2Vec:

```
vec("I")      → [0.01, -0.12, ..., 0.88]
vec("love")   → [0.55,  0.32, ..., 0.74]
vec("pizza")  → [0.29, -0.10, ..., 0.45]
```

1. **Add all word vectors:**

```
python
CopyEdit
total_vector = vec("I") + vec("love") + vec("pizza")
```

2. Divide by number of words:

```
avg_vector = total_vector / 3
```

Now `avg_vector` represents the sentence "I love pizza".

🧮 Python Code Example (Using Gensim Word2Vec)

```
from gensim.models import Word2Vec
import numpy as np

# Load pre-trained Word2Vec model (Google News or train your own)
model = Word2Vec.load("your_model_path.model")

# Tokenized sentence
sentence = ['i', 'love', 'pizza']

# Get vectors for words that are in the vocabulary
vectors = [model.wv[word] for word in sentence if word in model.wv]

# Take average
if vectors:
    avg_vector = np.mean(vectors, axis=0)
else:
    avg_vector = np.zeros(model.vector_size)  # fallback
```

Step	Explanation
`model.wv[word]`	Gives the Word2Vec vector of that word
`np.mean(vectors, axis=0)`	Calculates the average of all word vectors — element by element
`np.zeros(...)`	Used if no word was found in vocab

Each vector is e.g. **300 numbers**, so the final vector is also 300-dimensional.

`if vectors:`

- If `vectors` has at least one word vector → the condition is **True**.
- If `vectors = []` → the condition is **False**.
- 🧠 **Why check this?**
    - Because if **no word** from the sentence is found in Word2Vec, we can't calculate an average. There's nothing to average.

- `axis=0` means: **column-wise averaging** (i.e., across words, not across vector dimensions)

```
vec1 = [1, 2, 3]
vec2 = [4, 5, 6]

avg_vector = [(1+4)/2, (2+5)/2, (3+6)/2] = [2.5, 3.5, 4.5]
```

`avg_vector = np.zeros(model.vector_size)`

- If **none of the sentence words are found** in Word2Vec (i.e., `vectors == []`), we return a **zero vector**.

```
[0.0, 0.0, 0.0, ..., 0.0]  # 300 times
```

⚠️ Limitations

Issue	Why?
❌ Ignores word order	“I love pizza” vs “Pizza loves me” look the same
❌ Treats all words equally	“the”, “and”, “not” get same weight as “pizza”
❌ No context-awareness	“bank” in “river bank” and “money bank” gets same vector

So it's **simple but shallow**. For deeper understanding, we move to **BERT**, **Doc2Vec**, or **RNN-based models**.

✅ Use Cases

- **Baseline classifier** (e.g., logistic regression)
- **Text clustering**
- **Sentence similarity**
- **Plagiarism detection**
- **News categorization**

🧠 Trivia

- Average Word2Vec is like a **bag-of-meanings** (not bag of words)
- Fast, simple, and works surprisingly well for many tasks
- You can **weigh words differently**, like using TF-IDF before averaging

Example 2:

```
from gensim.models import Word2Vec
import numpy as np

# Sample sentences (pre-tokenized)
sentences = [
    ["natural", "language", "processing"],
    ["word", "embeddings", "are", "useful"],
    ["deep", "learning", "for", "nlp"]
]

# Train or load a Word2Vec model
model = Word2Vec(sentences, min_count=1)

def average_word2vec(doc):
    """Calculate average Word2Vec vector for a document"""
    vectors = []
    for word in doc:
        if word in model.wv:
            vectors.append(model.wv[word])
    if vectors:
        return np.mean(vectors, axis=0)
    return np.zeros(model.vector_size)

# Get document embeddings
doc_embeddings = [average_word2vec(doc) for doc in sentences]
```

```
import numpy as np
np.array(doc_embeddings).shape

Output:
(3,100)
```