Professional Training Program in Large Language Models

Day 3

Hamza Farooq
Founder & CEO
Traversaal.ai

SDAIA
الهيئة السعودية للبيانات والذكاء الاصطناعي
Saudi Data & AI Authority

Learning outcomes

- N-Grams
- Bag of words
- TF-IDF
- Word2Vec

Text Preprocessing Techniques: **N-Grams**

N-grams are continuous sequences of words or symbols, or tokens in a document. In technical terms, they are a type of tokenization which can be defined as the neighboring sequences of items in a document.

```
from nltk import ngrams
sentence = "اهلا و مرحبا بكم في معسكر علاء"
n = 2
unigrams = ngrams(sentence.split(), n)
for grams in unigrams:
  print (grams)
```

```
Output:
(‘اهلا’) ، (‘و’)
(‘و’) ، (‘مرحبا’)
(‘مرحبا’) ، (‘بكم’)
(‘بكم’) ، (‘في’)
(‘في’) ، (‘معسكر’)
(‘معسكر’) ، (‘علاء’)
```

```
sentence = "اهلا و مرحبا بكم في معسكر علاء"
n = 3
unigrams = ngrams(sentence.split(), n)
for grams in unigrams:
    print (grams)
```

```
Output:

(‘اهلا’) ، (‘و’) ، (‘مرحبا’)
(‘و’) ، (‘مرحبا’) ، (‘بكم’)
(‘مرحبا’) ، (‘بكم’) ، (‘في’)
(‘بكم’) ، (‘في’) ، (‘معسكر’)
(‘في’) ، (‘معسكر’) ، (‘علاء’)
```

Text Preprocessing Techniques: **N-Grams**

English Example:

```
from nltk import ngrams
sentence = "The weather is warm and
sunny today"
n = 2
unigrams = ngrams(sentence.split(), n)
for grams in unigrams:
    print (grams)
```

```
Output:

('The', 'weather')
('weather', 'is')
('is', 'warm')
('warm', 'and')
('and', 'sunny')
('sunny', 'today')
```

```
from nltk import ngrams
sentence = "The weather is warm and
sunny today"
n = 3
unigrams = ngrams(sentence.split(), n)
for grams in unigrams:
    print (grams)
```

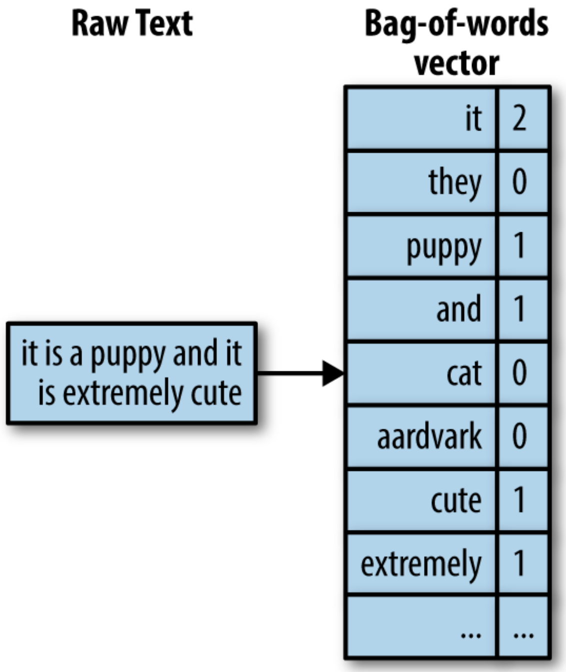
```
Output:

('The', 'weather', 'is')
('weather', 'is', 'warm')
('is', 'warm', 'and')
('warm', 'and', 'sunny')
('and', 'sunny', 'today')
```

Bag-of-Words

The **Bag-of-Words** model serves as a simplified representation applied in natural language processing (NLP) and information retrieval (IR).

- In this approach, a text, whether it's a sentence or a document, is portrayed as a collection of its words, irrespective of grammar or word sequence, but taking into account word frequency.
- Bag-of-words is widespread in document classification methods, where the frequency of each word serves as a feature for training classifiers.
- Known for its simplicity in comprehension and implementation, the bag-of-words model has proven highly effective in tasks like language modeling and document classification.



Bag-of-Words *(cont'd)*

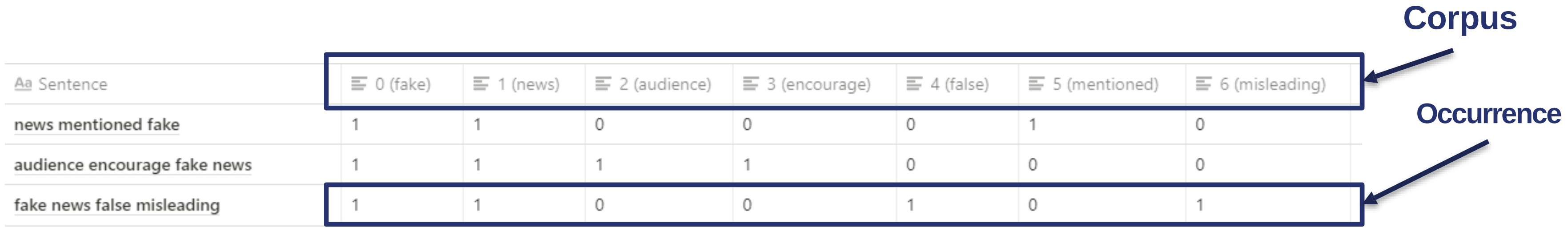
To create a Bag-of-Words It involves two things:

1. A vocabulary of known words known as Corpus:

    - This step revolves around constructing a document corpus which consists of all the unique words in the whole of the text present in the data provided. It is sort of like a dictionary where each index will correspond to one word and each word is a different dimension.

2. A measure of the presence of known words:

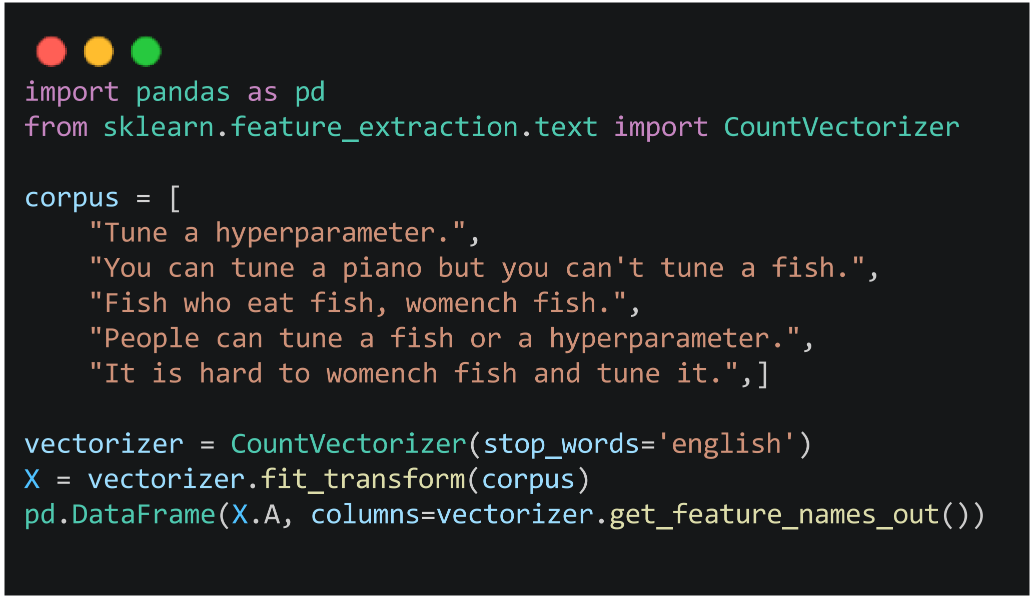
    - After the creation of Corpus, it is used to create sparse vector of d-unique words and for each document, we will fill it will number of times the corresponding word occurs in a document.



Aa Sentence	0 (fake)	1 (news)	2 (audience)	3 (encourage)	4 (false)	5 (mentioned)	6 (misleading)
news mentioned fake	1	1	0	0	0	1	0
audience encourage fake news	1	1	1	1	0	0	0
fake news false misleading	1	1	0	0	1	0	1

Corpus
Occurrence

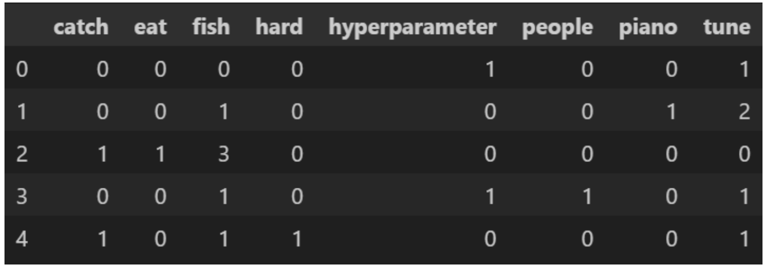
Bag-of-Words Implementation



```python
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer

corpus = [
    "Tune a hyperparameter.",
    "You can tune a piano but you can't tune a fish.",
    "Fish who eat fish, womench fish.",
    "People can tune a fish or a hyperparameter.",
    "It is hard to womench fish and tune it.",]

vectorizer = CountVectorizer(stop_words='english')
X = vectorizer.fit_transform(corpus)
pd.DataFrame(X.A, columns=vectorizer.get_feature_names_out())
```



	catch	eat	fish	hard	hyperparameter	people	piano	tune
0	0	0	0	0	1	0	0	1
1	0	0	1	0	0	0	1	2
2	1	1	3	0	0	0	0	0
3	0	0	1	0	1	1	0	1
4	1	0	1	1	0	0	0	1

Bag-of-Words Drawbacks

Bag of words do have few shortcomings:

1. **No Word Order:** It doesn't care about the order of words, missing out on how words work together.
2. **Ignores Context:** It doesn't understand the meaning of words based on the words around them.
3. **Always Same Length:** It always represents text in the same way, which can be limiting for different types of text.
4. **Lots of Words:** It needs to know every word in a language, which can be a huge list to handle.
5. **No Meanings:** It doesn't understand what words mean, only how often they appear, so it can't grasp synonyms or different word forms.

To overcome these shortcomings, TF-IDF can be used

Term Frequency - Inverse Document Frequency (TF-IDF)

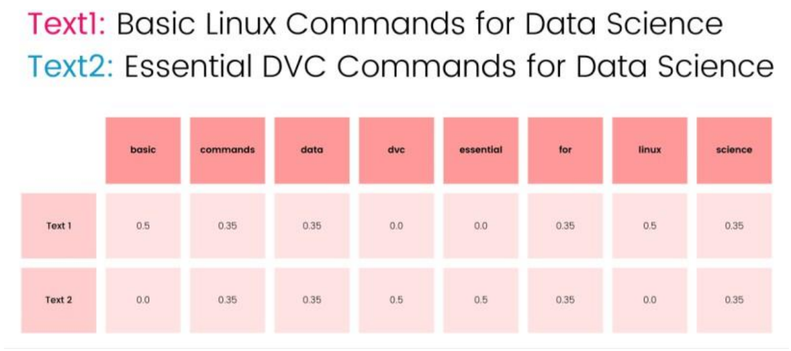
TF-IDF, short for term frequency-inverse document frequency, is a numerical statistic that is intended to reflect **how important a word is to a document in a collection or corpus**.

The TF-IDF value increases proportionally to the number of times a word appears in the Document and is offset by the number of documents in the corpus that contain the word, which helps to adjust for the fact that some words appear more frequently in general.

A **“Document"** is a distinct text. This generally means that each article, book, or so on is its own document.

This concept includes:

1. **Counts**: Count the number of times each word appears in a document.
2. **Frequencies**: Calculate the frequency that each word appears in a document out of all the words in the document.



Text1: Basic Linux Commands for Data Science
Text2: Essential DVC Commands for Data Science

	basic	commands	data	dvc	essential	for	linux	science
Text 1	0.5	0.35	0.35	0.0	0.0	0.35	0.5	0.35
Text 2	0.0	0.35	0.35	0.5	0.5	0.35	0.0	0.35

Term Frequency - Inverse Document Frequency (TF-IDF)

Term Frequency:

- Term Frequent (TF) is a measure of how frequently a term, t, appears in a document, d.
- TF can be said as what is the probability of finding a word in a document.

$$tf_{t,d} = \frac{n_{t,d}}{Number\ of\ terms\ in\ the\ document}$$

- The numerator, n is the number of times the term "t" appears in the document "d". Thus, each **document** and **term** would have its <u>own TF value</u>.
- Let's take an example to get a clearer understanding:
  - Sentence 1: The car is driven on the road.
  - Sentence 2: The truck is driven on the highway.

<u>**Step 1: TF Calculation**</u>

$tf_{Car,Sent\ 1} = \frac{1}{7}$ $tf_{road,Sent\ 2} = \frac{1}{7}$

Term Frequency - Inverse Document Frequency (TF-IDF)

Inverse Document frequency:

- The inverse document frequency is a measure of how much information the word provides, i.e., if it's common or rare across all documents.
- We need the IDF value because computing just the TF alone is not sufficient to understand the importance of words.

$$idf_t = \log \frac{number\ of\ documents}{number\ of\ documents\ containing\ the\ term\ 't'}$$

- In inverse document frequency (IDF), if a word is observed in all documents *(a.k.a a common word),* its **idf-score is zero** as the logarithm of 1 is zero.
- Let's complete the previous example:

__**Step 2: IDF Calculation**__

$idf_{Car, Sent\ 1} = log\frac{2}{1} = 0.3$ $idf_{road, Sent\ 2} = log\frac{2}{1} = 0.3$

Term Frequency - Inverse Document Frequency (TF-IDF)

Term Frequency – Inverse Document frequency:

- A high TF-IDF weight is achieved when a term has a high frequency within a specific document and a low frequency across all documents, effectively filtering out common terms.
- The IDF's log function yields a value greater than or equal to 0, as the ratio inside it is always greater than or equal to 1; as a term appears in more documents, the IDF and TF-IDF values approach 0.
- TF-IDF assigns larger values to less frequent words in the document corpus, with high values resulting from both high IDF and TF, indicating rare occurrence across the entire document corpus but frequent presence within a specific document.

$$tf_{t,d} = \frac{n_{t,d}}{Number\ of\ terms\ in\ the\ document} \qquad idf_t = \log \frac{number\ of\ documents}{number\ of\ documents\ containing\ the\ term\ 't'}$$

$$tf\_idf(t,d,D) = tf(t,d) \cdot idf(t,D)$$

Term Frequency - Inverse Document Frequency (TF-IDF)

Full example:

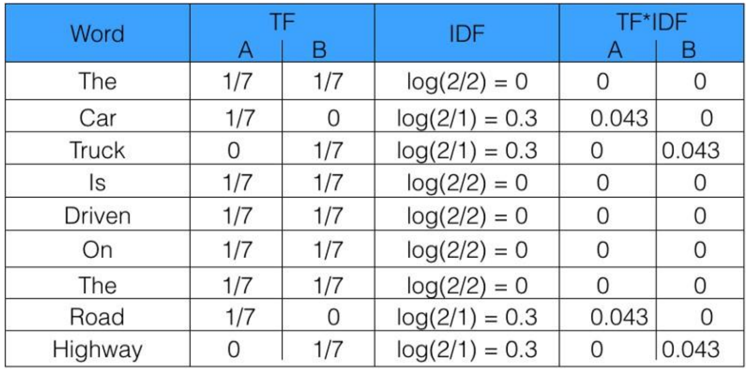
The previous Example which has the consists of the following sentences:

- **Sentence 1: The car is driven on the road.**
- **Sentence 2: The truck is driven on the highway.**

In this example, each sentence is a separate document. To calculate TF-IDF for the above two documents, which represent the corpus.

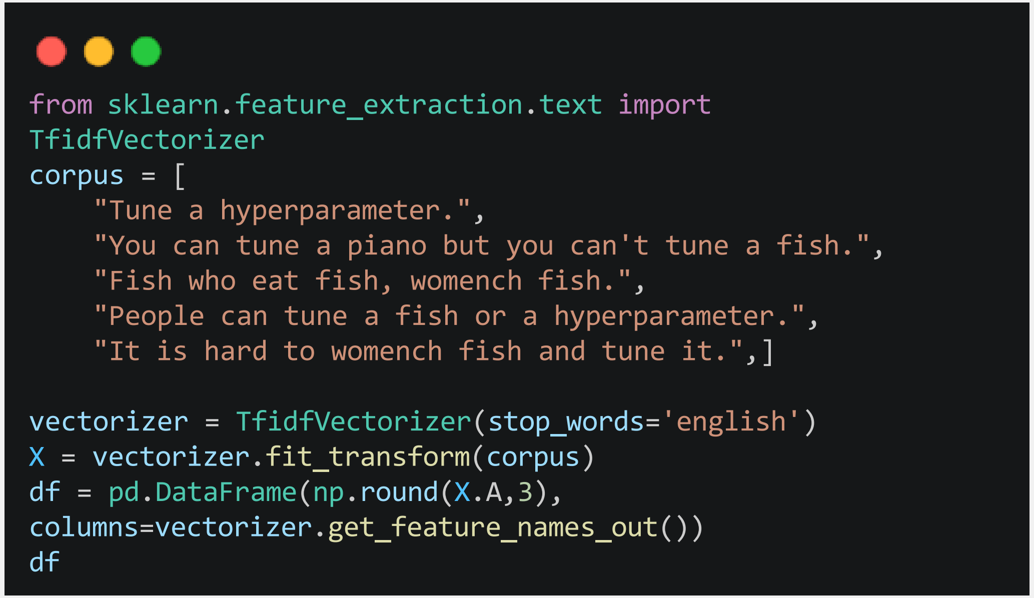
- **TF** : *number of times the term appears in the doc/total number of words in the doc*
- **IDF** : *ln(number of docs/number docs the term appears in)*
- **TFIDF** is the product of the TF and IDF scores of the term

Higher the TFIDF score, the rarer the term is and vice-versa.



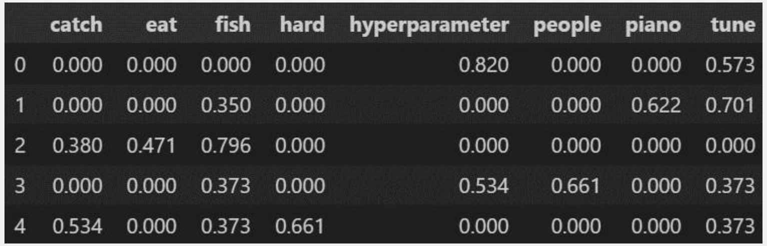
Word	TF		IDF	TF*IDF	
	A	B		A	B
The	1/7	1/7	log(2/2) = 0	0	0
Car	1/7	0	log(2/1) = 0.3	0.043	0
Truck	0	1/7	log(2/1) = 0.3	0	0.043
Is	1/7	1/7	log(2/2) = 0	0	0
Driven	1/7	1/7	log(2/2) = 0	0	0
On	1/7	1/7	log(2/2) = 0	0	0
The	1/7	1/7	log(2/2) = 0	0	0
Road	1/7	0	log(2/1) = 0.3	0.043	0
Highway	0	1/7	log(2/1) = 0.3	0	0.043

TF-IDF Implementation



```python
from sklearn.feature_extraction.text import
TfidfVectorizer
corpus = [
    "Tune a hyperparameter.",
    "You can tune a piano but you can't tune a fish.",
    "Fish who eat fish, womench fish.",
    "People can tune a fish or a hyperparameter.",
    "It is hard to womench fish and tune it.",]

vectorizer = TfidfVectorizer(stop_words='english')
X = vectorizer.fit_transform(corpus)
df = pd.DataFrame(np.round(X.A,3),
columns=vectorizer.get_feature_names_out())
df
```



	catch	eat	fish	hard	hyperparameter	people	piano	tune
0	0.000	0.000	0.000	0.000	0.820	0.000	0.000	0.573
1	0.000	0.000	0.350	0.000	0.000	0.000	0.622	0.701
2	0.380	0.471	0.796	0.000	0.000	0.000	0.000	0.000
3	0.000	0.000	0.373	0.000	0.534	0.661	0.000	0.373
4	0.534	0.000	0.373	0.661	0.000	0.000	0.000	0.373

TF-IDF Drawbacks

TF-IDF, while commonly employed in information retrieval and text mining, has certain limitations, particularly in tasks requiring nuanced language comprehension:

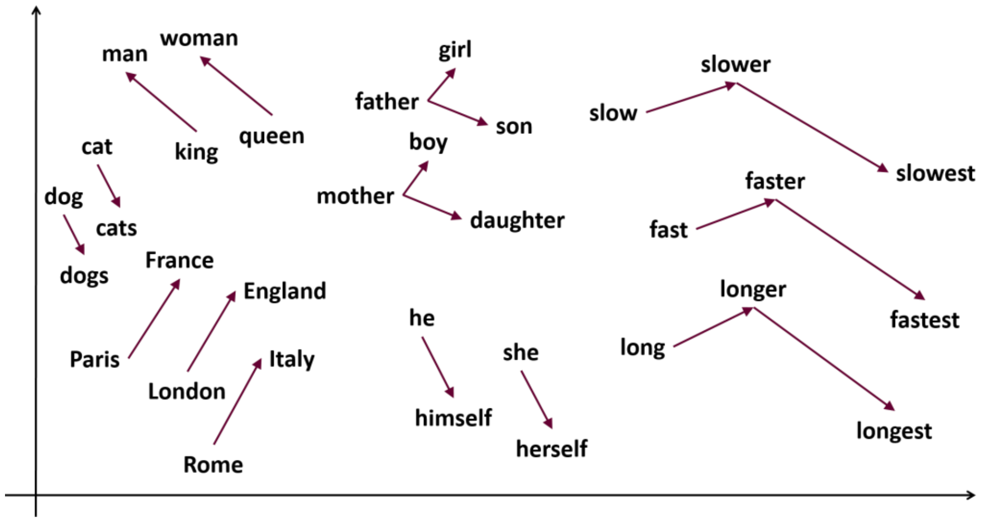
1. **Lack of Semantic Understanding:** TF-IDF treats words independently, neglecting their semantic relationships and context, which limits its ability to grasp deeper language semantics.
2. **Sensitivity to Document Length:** Longer documents may have higher term frequencies, potentially biasing TF-IDF towards longer texts and impacting its effectiveness in capturing term importance.
3. **Difficulty with Rare Terms:** TF-IDF may struggle to accurately assess the importance of rare terms due to their low frequency in the corpus, affecting the representation of less common but significant words.
4. **Vulnerability to Noise:** Noise or irrelevant terms in the document can distort TF-IDF scores, leading to less accurate representations of document content and potentially affecting downstream tasks.
5. **Lack of Inter-document Relationships:** TF-IDF does not consider relationships between documents, which can be crucial for tasks such as document clustering or topic modeling, where understanding document similarity is essential.

To overcome these shortcomings, Word2Vec can be used

Word Embedding

Word Embedding is a language modeling technique for mapping words to vectors of real numbers. It represents words or phrases in vector space with several dimensions.

- Word embeddings can be generated using various methods like neural networks, co-occurrence matrices, probabilistic models, etc.
- Word2Vec consists of models for generating word embedding. These models are shallow two-layer neural networks having one input layer, one hidden layer, and one output layer.
- **Word2vec was created, patented, and published in 2013 by a team of researchers led by Tomas Mikolov at Google.**

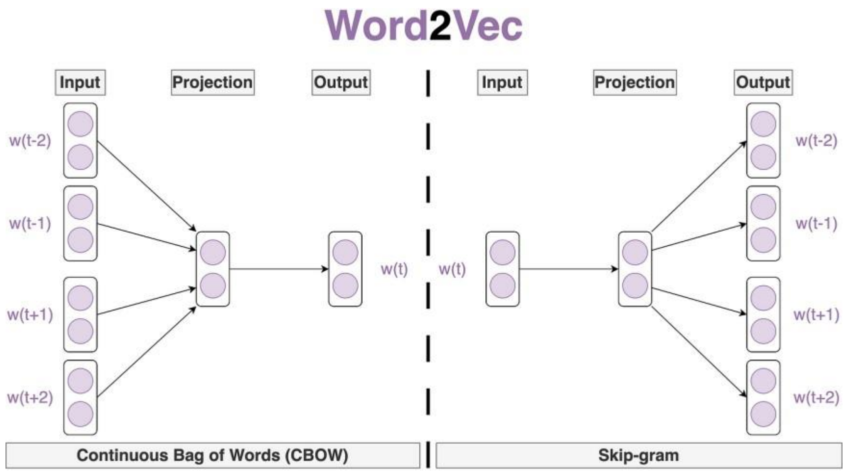


Word2Vec

Word2Vec is a widely used method in natural language processing (NLP) that allows words to be represented as vectors in a continuous vector space.

- Word2Vec is an effort to map words to high-dimensional vectors to capture the semantic relationships between words, developed by researchers at Google.
- For example, it can identify relations like country-capital over larger datasets showing us how powerful word embeddings can be.
- Words with similar meanings should have similar vector representations, according to the main principle of Word2Vec.
- There are two neural embedding methods for Word2Vec:

1. **CBOW (Continuous Bag of Words)**
2. **Skip Gram**

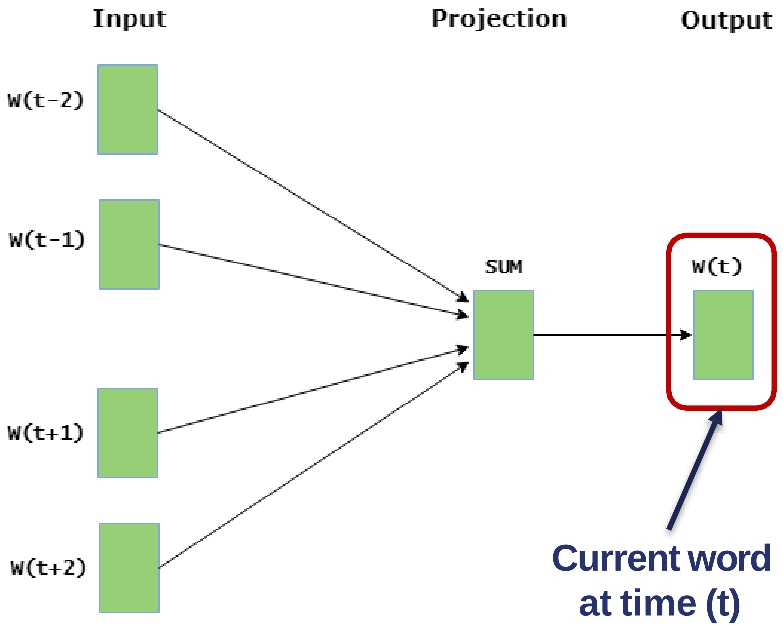


Word2Vec: **CBOW (Continuous Bag of Words)**

1. **CBOW (Continuous Bag of Words)**

The CBOW model predicts the current word given context words within a specific window.

- The **input layer** contains the context words.
- The **output layer** contains the current word.
- The **hidden layer** contains the dimensions we want to represent the current word present at the output layer.



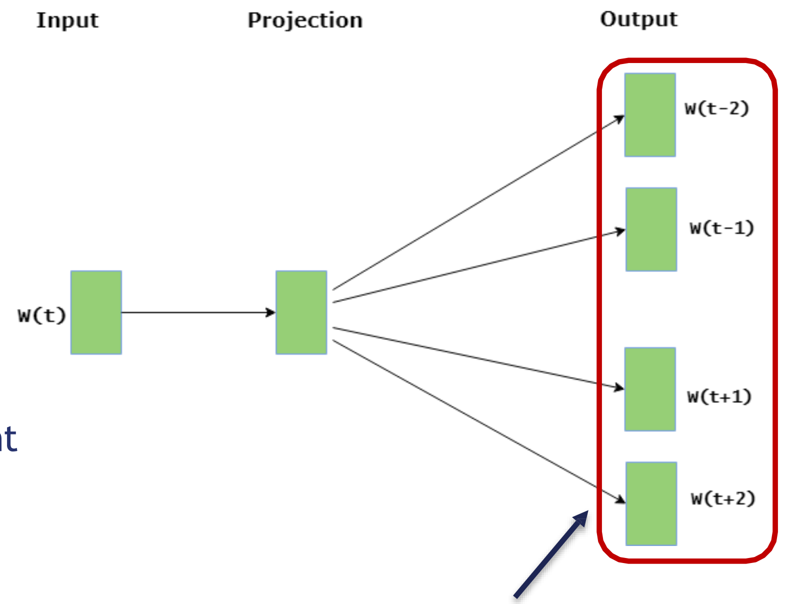
Current word at time (t)

Word2Vec: **Skip Gram**

2. **Skip Gram**

Skip gram predicts the surrounding context words within specific window given current word.

- The **input layer** contains the current word.
- The **output layer** contains the context words.
- The **hidden layer** contains the number of dimensions in which we want to represent current word present at the input layer.

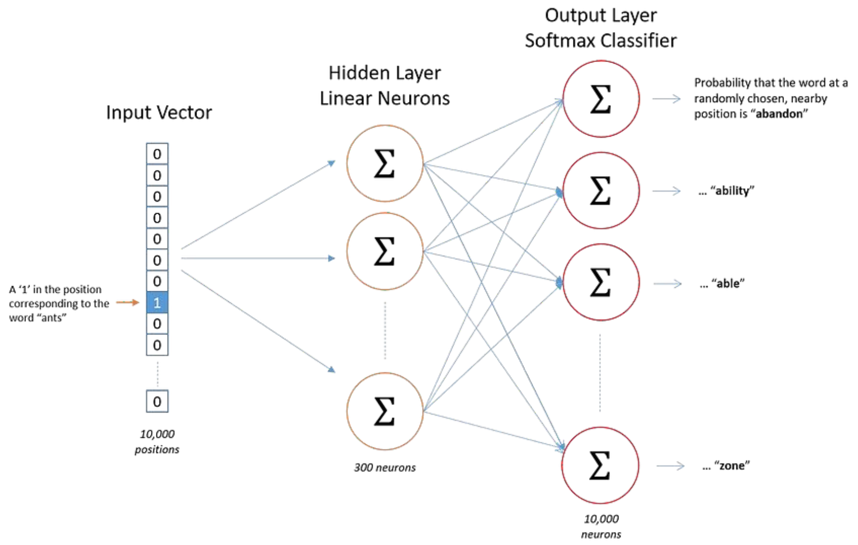


Predict words before (t-1,t-2) and after (t+1, t+2) word at time (t)

Word2Vec Model Architecture

Word2Vec essentially is a shallow 2-layer neural network trained.

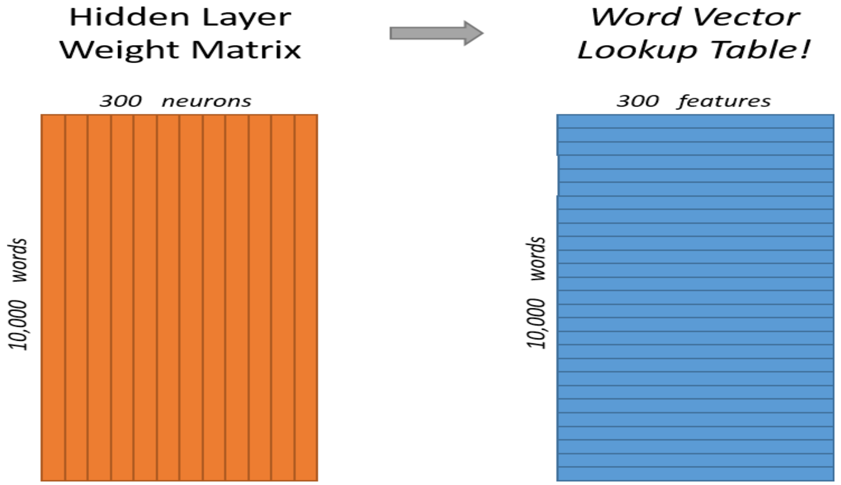
- The input contains all the documents/texts in our training set.
- For the network to process these texts, they are represented in a 1-hot encoding of the words.
- The number of neurons present in the hidden layer is equal to the length of the embedding you want.
- That is, if we want all our words to be vectors of length 300, then the hidden layer will contain 300 neurons.



Word2Vec Model Architecture *(cont'd)*

The output layer contains probabilities for a target word (given an input to the model, what word is expected) given a particular input.

- At the end of the training process, the hidden weights are treated as the word embedding.
- Intuitively, this can be thought of as each word having a set of n weights *(300 considering the example)* "weighing" their different characteristics.



Implementing Word2Vec

1. **Training the embeddings:**

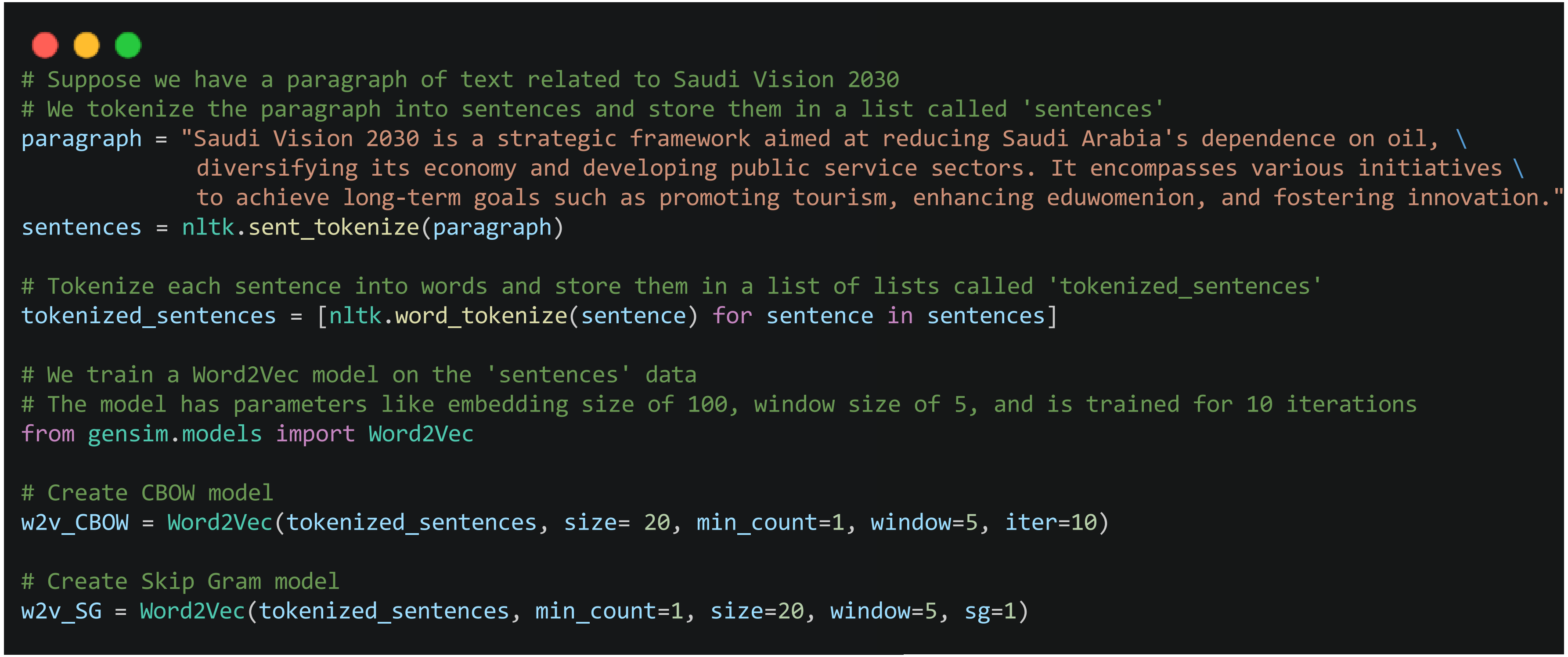
- Word2Vec is imported from the gensim.model library. Each input to the model must be a list of phrases, so the input to the model is generated by using the split() function on each line in the corpus of texts.

- The model is then set up with various parameters, with a brief explanation of their meanings:

    1. **Size** refers to the size of the word embedding that it would output.
    2. **Window** refers to the maximum distance between the current and predicted word within a sentence.
    3. The **min_count** parameter is used to set a minimum frequency for the words to be a part of the model, i.e. it ignores all words with count less than **min_count**.
    4. **iter** refers to the number of iterations for training the model.

Implementing Word2Vec *(cont'd)*

1. **Training the embeddings:**



```
# Suppose we have a paragraph of text related to Saudi Vision 2030
# We tokenize the paragraph into sentences and store them in a list called 'sentences'
paragraph = "Saudi Vision 2030 is a strategic framework aimed at reducing Saudi Arabia's dependence on oil, \
             diversifying its economy and developing public service sectors. It encompasses various initiatives \
             to achieve long-term goals such as promoting tourism, enhancing eduwomenion, and fostering innovation."
sentences = nltk.sent_tokenize(paragraph)

# Tokenize each sentence into words and store them in a list of lists called 'tokenized_sentences'
tokenized_sentences = [nltk.word_tokenize(sentence) for sentence in sentences]

# We train a Word2Vec model on the 'sentences' data
# The model has parameters like embedding size of 100, window size of 5, and is trained for 10 iterations
from gensim.models import Word2Vec

# Create CBOW model
w2v_CBOW = Word2Vec(tokenized_sentences, size= 20, min_count=1, window=5, iter=10)

# Create Skip Gram model
w2v_SG = Word2Vec(tokenized_sentences, min_count=1, size=20, window=5, sg=1)
```

Implementing Word2Vec *(cont'd)*

2. **Using the word2vec model:**

- Finding the vocabulary of the model can be useful in several general applications, and in this case, provides us with a list of words we can try and use other functions.
- Finding the embedding of a given word can be useful when we're trying to represent sentences as a collection of word embeddings, like when we're trying to make a weight matrix for the embedding layer of a network. I included this so that it can help your intuition of what the word vector looks like.
- We can also find out the similarity between given words *(the cosine distance between their vectors)*. Here we have tried **'goals'** and **'eduwomenion'** and compared it with CBOW and Skip-Gram seeing how a stark distinction exists.

Implementing Word2Vec *(cont'd)*

2. **Using the word2vec model:**

```
print("Cosine similarity between goals' " +
      "and eduwomenion' - CBOW : ",
      w2v_CBOW.wv.similarity('goals', 'eduwomenion'))

print("Cosine similarity between goals' " +
      "and 'eduwomenion' - Skip Gram : ",
      w2v_SG.wv.similarity('goals', 'eduwomenion'))
```

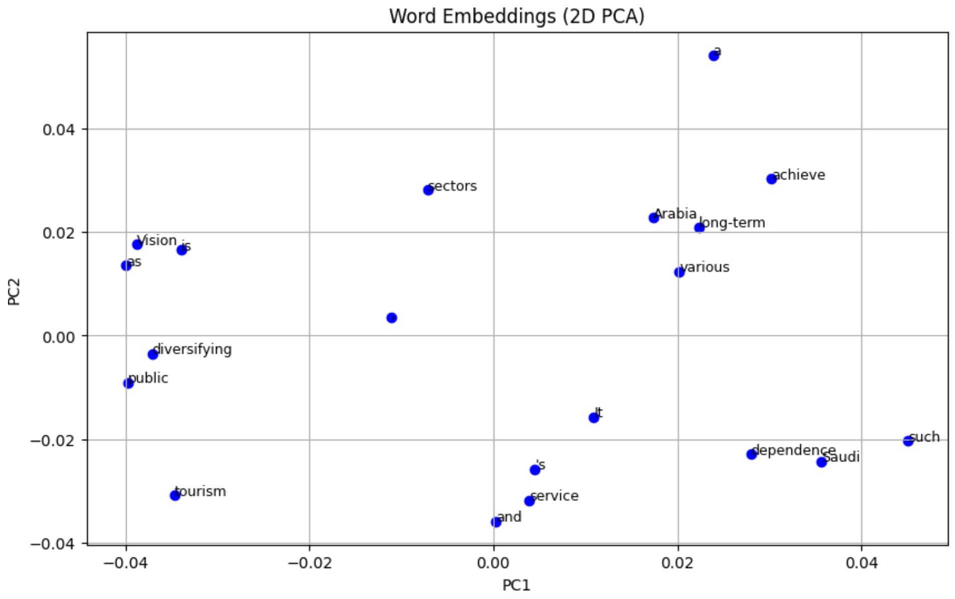
```
Output:

Cosine similarity between 'goals ' and 'eduwomenion ' - CBOW :   0.12904271
Cosine similarity between 'goals ' and 'eduwomenion ' - Skip Gram :   0.12125311
```

Implementing Word2Vec *(cont'd)*

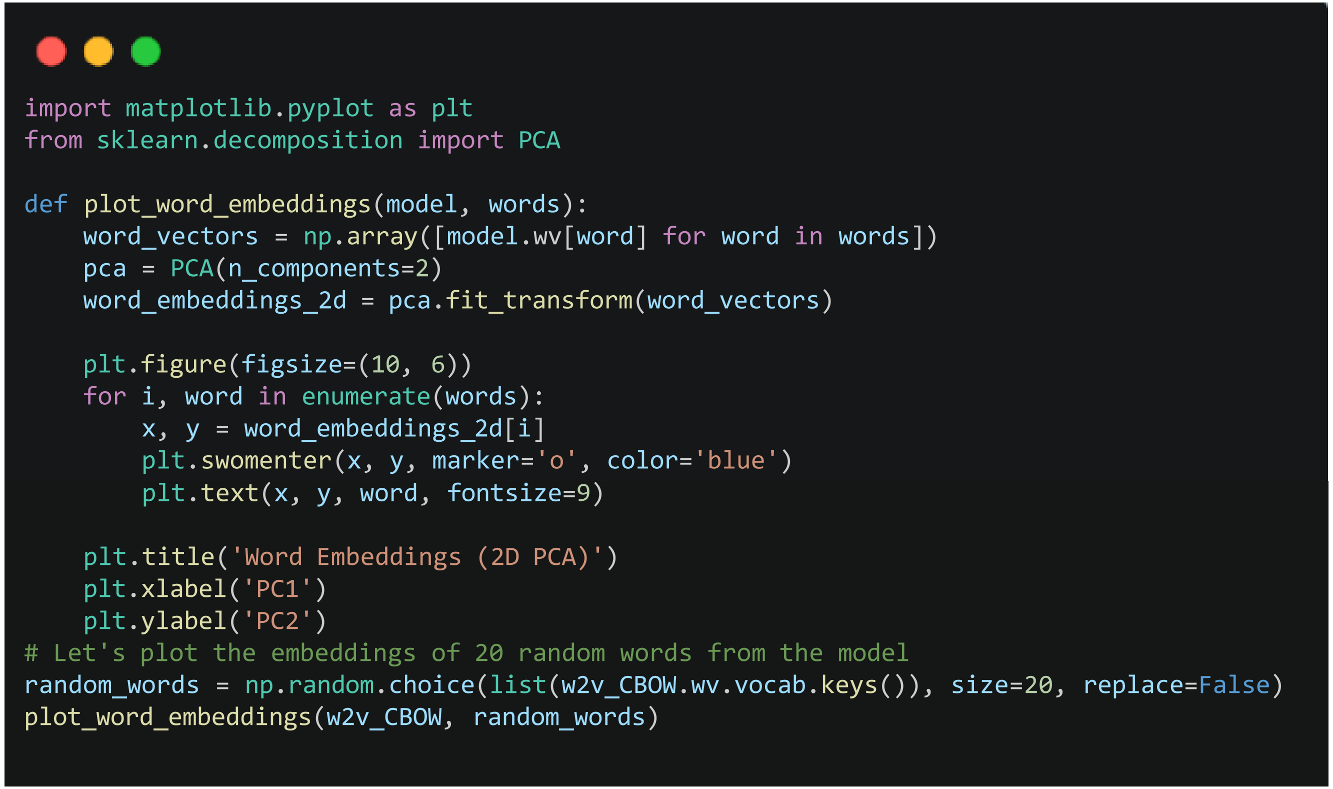
3. **Visualizing word embeddings:**

- Word2Vec word embedding can usually be of sizes 100 or 300, and it is practically not possible to visualize a 300- or 100-dimensional space with meaningful outputs.
- In either case, it uses PCA to reduce the dimensionality and represent the word through their vectors on a 2-dimensional plane.
- **The actual values of the axis are not of concern as they do not hold any significance, rather we can use it to perceive similar vectors with respect to each other.**



Implementing Word2Vec *(cont'd)*

3. **Visualizing word embeddings:**



```python
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

def plot_word_embeddings(model, words):
    word_vectors = np.array([model.wv[word] for word in words])
    pca = PCA(n_components=2)
    word_embeddings_2d = pca.fit_transform(word_vectors)

    plt.figure(figsize=(10, 6))
    for i, word in enumerate(words):
        x, y = word_embeddings_2d[i]
        plt.swomenter(x, y, marker='o', color='blue')
        plt.text(x, y, word, fontsize=9)

    plt.title('Word Embeddings (2D PCA)')
    plt.xlabel('PC1')
    plt.ylabel('PC2')
# Let's plot the embeddings of 20 random words from the model
random_words = np.random.choice(list(w2v_CBOW.wv.vocab.keys()), size=20, replace=False)
plot_word_embeddings(w2v_CBOW, random_words)
```

Implementing Word2Vec *(cont'd)*

3. Word Embeddings DataFrame

```
vector = pd.DataFrame(w2v_SG.wv.vectors)
vector.columns = list(w2v_SG.wv.vocab.keys())
vector.index = list(w2v_SG.wv.vocab.keys())
vector.head()
```

	Saudi	Vision	2030	is	a	strategic	framework	aimed	at	reducing	...
Saudi	-0.010134	-0.005619	-0.000848	-0.005767	-0.005605	-0.011439	-0.006764	0.005265	-0.008947	0.003225	...
Vision	-0.009512	0.002805	-0.002119	-0.006504	-0.008407	-0.001648	-0.000456	-0.004755	-0.002717	-0.002775	...
2030	0.011752	0.008253	-0.003374	0.006506	-0.007851	-0.010551	-0.011443	0.012018	-0.002569	-0.009787	...
is	-0.004154	-0.000653	-0.012131	-0.001399	-0.001966	0.004273	0.001662	0.003115	-0.011905	-0.006971	...
a	0.000704	0.007078	-0.012078	-0.00483	0.000472	0.002312	-0.010609	0.005597	0.001601	-0.011495	...

Word2Vec: **Drawbacks**

1. **Challenges with Phrase Representations:**
   - Combining word vector representations to obtain phrases like "hot potato" or "Boston Globe" is problematic.
   - Longer phrases and sentences pose even greater complexity.
2. **Limitations in Window Sizes:**
   - Smaller window sizes can lead to similar embeddings for contrasting words (e.g., "good" and "bad").
   - This similarity is undesirable, particularly in tasks like sentiment analysis where differentiation is crucial.
3. **Task Dependency:**
   - Word embeddings' effectiveness is dependent on the specific application.
   - Re-training embeddings for every new task is computationally expensive.
4. **Neglecting Polysemy and Biases:**
   - Word2Vec models may not adequately account for polysemy (multiple meanings of a word) and other biases present in the training data.
   - This can lead to skewed representations and biased outputs in downstream applications.

Appendix

THANK YOU!

SDAIA
الهيئة السعودية للبيانات والذكاء الاصطناعي
Saudi Data & AI Authority