Tutoral 6: CN7050: Intelligent Systems

Text represenation for AI and introducing LLMs

In this tutorial, you will learn about:

```
How text is represented in AI systems

Word embedding models and their role in NLP

Sentence embeddings using Transformer models

How large language models generate text
```

Dr Shaheen Khatoon

Ph.D. | PMP | PMI-ACP | ITIL | PgCert TLHE

Senior Lecturer in Data Science (Computer Science and Digital Technologies) School of Architecture, Computing and Engineering

Room: EB.1.99

(e) S.Khatoon@uel.ac.uk

(w) https://uel.ac.uk/about-uel/staff/shaheen-khatoon '''

How to represent text for AI

Baisc preprocessing

```
# Install dependencies if not available
!pip install -q nltk pandas

# Import libraries
import nltk
import pandas as pd
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer

# Download required NLTK resources
nltk.download('punkt')
```

```
nltk.download('stopwords')
nltk.download('punkt_tab')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
True
```

```
# ----------------------------
# Text Preprocessing Function
# ----------------------------
def preprocess_text(sentence):
    # Tokenization
    tokens = word_tokenize(sentence.lower())

    # Stopword removal
    stop_words = set(stopwords.words('english'))
    filtered_tokens = [w for w in tokens if w.isalnum() and w not in st

    # Stemming
    ps = PorterStemmer()
    stemmed_tokens = [ps.stem(w) for w in filtered_tokens]

    return " ".join(stemmed_tokens), stemmed_tokens
```

```
# Example Documents
documents = [
    "Should we go to a pizzeria or do you prefer a restaurant?",
    "Natural Language Processing is a key part of artificial intelligen
    "Machine learning enables computers to learn from data.",
    "AI and data science are transforming healthcare analytics."
]

# Preprocess each document
processed_docs = []
for doc in documents:
    cleaned_text, tokens = preprocess_text(doc)
    processed_docs.append({
        "Original": doc,
        "Cleaned_Text": cleaned_text,
        "Tokens": tokens
    })

# Display preprocessed results
df = pd.DataFrame(processed_docs)
print("Preprocessed DataFrame:")
display(df)
```

Preprocessed DataFrame:

	Original	Cleaned_Text	Tokens
0	Should we go to a pizzeria or do you prefer a ...	go pizzeria prefer restaur	[go, pizzeria, prefer, restaur]
1	Natural Language Processing is a key part of a...	natur languag process key part artifici intellig	[natur, languag, process, key, part, artifici,...
2	Machine learning enables	machin learn enabl	[machin, learn, enabl,

Next steps: Generate code with df | New interactive sheet

One-hot encoding

```
#one-hot encoding for a single document
import numpy as np
def one_hot_encoding(sentence):
    words = sentence.lower().split()
    vocabulary = sorted(set(words))
    word_to_index = {word: i for i, word in enumerate(vocabulary)}
    one_hot_matrix = np.zeros((len(words), len(vocabulary)), dtype=int)
    for i, word in enumerate(words):
        one_hot_matrix[i, word_to_index[word]] = 1

    return one_hot_matrix, vocabulary

# Example of usage
sentence = "Should we go to a pizzeria or do you a prefer a restaurant?
one_hot_matrix, vocabulary = one_hot_encoding(sentence)
print("Vocabulary:", vocabulary)
print("One-Hot Encoding Matrix:\n", one_hot_matrix)
```

```
Vocabulary: ['a', 'do', 'go', 'or', 'pizzeria', 'prefer', 'restaurant?', 'sho
One-Hot Encoding Matrix:
 [[0 0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0]
 [1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0 0]]
```

```
# One-Hot Encoding for Entire Vocabulary
```

```
import numpy as np

# Flatten tokens from all documents to build the full vocabulary
all_tokens = [token for sublist in df["Tokens"] for token in sublist]
vocabulary = sorted(set(all_tokens))

# Create a mapping from word to index
word_to_index = {word: i for i, word in enumerate(vocabulary)}

# Initialize matrix (documents × vocabulary)
one_hot_matrix = np.zeros((len(df), len(vocabulary)), dtype=int)

# Fill one-hot matrix for each document
for i, tokens in enumerate(df["Tokens"]):
    for token in tokens:
        if token in word_to_index:
            one_hot_matrix[i, word_to_index[token]] = 1

# Create a labeled DataFrame for better readability
one_hot_df = pd.DataFrame(one_hot_matrix, columns=vocabulary)
one_hot_df.index = [f"Doc_{i+1}" for i in range(len(df))]

print(" Vocabulary:", vocabulary)
print("\n One-Hot Encoding Matrix for All Documents:")
display(one_hot_df)
```

```
Vocabulary: ['ai', 'analyt', 'artifici', 'comput', 'data', 'enabl', 'go', 'h

One-Hot Encoding Matrix for All Documents:
```

	ai	analyt	artifici	comput	data	enabl	go	healthcar	intellig	ke
Doc_1	0	0	0	0	0	0	1	0	0	
Doc_2	0	0	1	0	0	0	0	0	1	
Doc_3	0	0	0	1	1	1	0	0	0	
Doc_4	1	1	0	0	1	0	0	1	0	

4 rows × 21 columns

Bag-of-words

BoW is an algorithm for extracting features from text that preserves this frequency property. BoW is a very simple algorithm that ignores the position of words in the text and only considers this frequency property. The name “bag” comes precisely from the fact that any information concerning sentence order and structure is not preserved by the algorithm. For BoW, we only need a vocabulary and a way to be able to count words. In this case, the idea is to create document vectors: a single vector represents a document and the frequency of words contained in the vocabulary.

```
def bag_of_words(sentences):
    """
    Creates a bag-of-words representation of a list of documents.
    """
    tokenized_sentences = [sentence.lower().split() for sentence in sen
    flat_words = [word for sublist in tokenized_sentences for word in s
    vocabulary = sorted(set(flat_words))
    word_to_index = {word: i for i, word in enumerate(vocabulary)}

    bow_matrix = np.zeros((len(sentences), len(vocabulary)), dtype=int)
    for i, sentence in enumerate(tokenized_sentences):
        for word in sentence:
            if word in word_to_index:
                bow_matrix[i, word_to_index[word]] += 1

    return vocabulary, bow_matrix

# Example of usage
corpus = ["This movie is awesome awesome", "I do not say is good, but n
            "Awesome? Only a fool can say that"]
vocabulary, bow_matrix = bag_of_words(corpus)
print("Vocabulary:", vocabulary)
print("Bag of Words Matrix:\n", bow_matrix)
```

```
Vocabulary: ['a', 'awesome', 'awesome?', 'but', 'can', 'do', 'fool', 'good,',
Bag of Words Matrix:
 [[0 2 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1]
 [0 1 0 1 0 1 0 1 1 1 0 1 1 0 1 0 0]
 [1 0 1 0 1 0 1 0 0 0 0 0 0 1 1 1 0]]
```

TF-IDF

In BoW, we obtained a document-term matrix. However, the raw frequency is very skewed and does not always allow us to discriminate between two documents.

The document-term matrix was born in information retrieval to find documents, though words such as “good” or “bad” are not very discriminative since they are often used in text with a generic meaning. In contrast, words with low frequency are much more informative, so we are interested more in relative than absolute frequency:

Instead of using raw frequency, we can use the logarithm in base 10, because a word that occurs 100 times in a document is not 100 times more relevant to its meaning in the document. Of course, since vectors can be very sparse, we assign 0 if the frequency is 0.

Second, we want to pay more attention to words that are present only in some documents. These words will be more relevant to the meaning of the document, and we want to preserve this information. To do this, we normalize by IDF. IDF is defined as the ratio of the total number of documents in the corpus to how many documents a term is present in. To summarize, to obtain the TF-IDF, we multiply TF by the logarithm of IDF.

```
def compute_tf(sentences):
    """Compute the term frequency matrix for a list of sentences."""
    vocabulary = sorted(set(word for sentence in sentences for word in
    word_index = {word: i for i, word in enumerate(vocabulary)}
    tf = np.zeros((len(sentences), len(vocabulary)), dtype=np.float32)
    for i, sentence in enumerate(sentences):
        words = sentence.lower().split()
        word_count = len(words)
        for word in words:
            if word in word_index:
                tf[i, word_index[word]] += 1 / word_count
    return tf, vocabulary

def compute_idf(sentences, vocabulary):
    """Compute the inverse document frequency for a list of sentences."
    num_documents = len(sentences)
    idf = np.zeros(len(vocabulary), dtype=np.float32)
    word_index = {word: i for i, word in enumerate(vocabulary)}
    for word in vocabulary:
        df = sum(1 for sentence in sentences if word in sentence.lower(
        idf[word_index[word]] = np.log(num_documents / (1 + df)) + 1  #
    return idf

def tf_idf(sentences):
    """Generate a TF-IDF matrix for a list of sentences."""
    tf, vocabulary = compute_tf(sentences)
    idf = compute_idf(sentences, vocabulary)
    tf_idf_matrix = tf * idf
    return vocabulary, tf_idf_matrix

vocabulary, tf_idf_matrix = tf_idf(corpus)
print("Vocabulary:", vocabulary)
print("TF-IDF Matrix:\n", tf_idf_matrix)
```

```
Vocabulary: ['a', 'awesome', 'awesome?', 'but', 'can', 'do', 'fool', 'good,',
TF-IDF Matrix:
 [[0.         0.4        0.         0.         0.         0.
  0.         0.         0.         0.2        0.28109303 0.
  0.         0.         0.         0.         0.28109303]
 [0.         0.11111111 0.         0.1561628  0.         0.1561628
  0.         0.1561628  0.1561628  0.11111111 0.         0.1561628
  0.1561628  0.         0.11111111 0.         0.         ]
 [0.20078073 0.         0.20078073 0.         0.20078073 0.
  0.20078073 0.         0.         0.         0.         0.
  0.         0.20078073 0.14285715 0.20078073 0.        ]]
```

Use of TFIDF in IR systems

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
```

```
#this for unzip and read the file
try:
    df=pd.read_csv("IMDB Dataset.csv")
except:
    !wget https://github.com/SalvatoreRa/tutorial/blob/main/datasets/IM
    !unzip IMDB.zip?raw=true
```

```
--2025-11-03 14:55:23--  https://github.com/SalvatoreRa/tutorial/blob/main/da
Resolving github.com (github.com)... 140.82.112.4
Connecting to github.com (github.com)|140.82.112.4|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://github.com/SalvatoreRa/tutorial/raw/refs/heads/main/dataset
--2025-11-03 14:55:23--  https://github.com/SalvatoreRa/tutorial/raw/refs/hea
Reusing existing connection to github.com:443.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/SalvatoreRa/tutorial/refs/heads/m
--2025-11-03 14:55:23--  https://raw.githubusercontent.com/SalvatoreRa/tutori
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.10
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.1
HTTP request sent, awaiting response... 200 OK
Length: 26962657 (26M) [application/zip]
Saving to: ‘IMDB.zip?raw=true’

IMDB.zip?raw=true   100%[===================>]  25.71M  --.-KB/s    in 0.1s

2025-11-03 14:55:24 (269 MB/s) - ‘IMDB.zip?raw=true’ saved [26962657/26962657

Archive:  IMDB.zip?raw=true
  inflating: IMDB Dataset.csv
```

```
# Load the dataset
df = pd.read_csv("IMDB Dataset.csv")
df = df.iloc[:5000,:]

# Display the first few rows of the dataframe
print(df.head())

# Initialize the TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer(max_features=1000)

# Fit and transform
tfidf_matrix = tfidf_vectorizer.fit_transform(df['review'])
```

```
# Convert the TF-IDF matrix to a DataFrame
tfidf_df = pd.DataFrame(tfidf_matrix.toarray(), columns=tfidf_vectorize
```

```
                                                review sentiment
0  One of the other reviewers has mentioned that ...  positive
1  A wonderful little production. <br /><br />The...  positive
2  I thought this was a wonderful way to spend ti...  positive
3  Basically there's a family where a little boy ...  negative
4  Petter Mattei's "Love in the Time of Money" is...  positive
```

```
# Define the query
query = "I want to watch a movie about zombie"

# Transform the query using the same TF-IDF vectorizer
query_tfidf = tfidf_vectorizer.transform([query])

# Calculate cosine similarity between the query and all documents
cosine_similarities = cosine_similarity(query_tfidf, tfidf_matrix).flat

df['cosine_similarity'] = cosine_similarities

reranked_df = df.sort_values(by='cosine_similarity',
                             ascending=False).reset_index(drop=True)

reranked_df
```

	review	sentiment	cosine_similarity
0	If you want to waste a small portion of your l...	negative	0.589198
1	This movie is god awful. Not one quality to th...	negative	0.483662
2	Now, I'm a big fan of Zombie movies. I admit Z...	negative	0.463848
3	"Revolt of the Zombies" proves that having the...	negative	0.360548
4	I haven't had a chance to view the previous fi...	negative	0.343682
...	...	...	...
4995	i see hundreds of student films- this is tops....	positive	0.000000
4996	Although I'm a girl, thankfully I have a sense...	positive	0.000000

Next steps: Generate code with `reranked_df` New interactive sheet

```
#give a look in detail to the first example
reranked_df.iloc[0,0]
```

```
'If you want to waste a small portion of your life sit in front of this pred
ictable zombie film. It fails at the first post by not being scary OR funny.
It is a dull grey movie that I guess went straight to video. Hammy and tongu
e in cheek acting leave a sour taste in the mouth. If you want to watch a po
or but still watchable recent zombie film watch Diary of the Dead. Poor spec
```

Word Embedding

Generate vectors that are small in size, composed of real (dense) numbers, and that preserve this contextual information.

Thus, the purpose is to have vectors of limited size that can represent the meaning of a word. The scattered vectors we obtained earlier cannot be used efficiently for mathematical operations or downstream tasks. Also, the more words there are in the vocabulary, the larger the size of the vectors we get. Therefore, we want dense vectors (with real numbers) that are small in size and whose size does not increase as the number of words in the vocabulary increases.

these dense vectors can be used for different operations because they better represent the concept of similarity between words. These dense vectors are called word embeddings.

Data load and preprocessing

```
!pip install -q gensim umap-learn tqdm adjustText
```

```
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 27.9/27.9 MB 86.5 MB/s eta 0:00:0
```

```
import numpy as np
import pandas as pd
import os
import re
import time
import nltk
from gensim.models import Word2Vec
from tqdm import tqdm
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.cluster.hierarchy import dendrogram, linkage
from adjustText import adjust_text
from umap import UMAP
```

```
def preprocessing_reviews(reviews):

    """
    simple preprocessing: splitting on the space and remove word less t
    """

    processed_reviews = []

    for review in tqdm(reviews):
        review = re.sub('<[^>]+>', '', review)
        processed = re.sub('[^a-zA-Z ]', '', review)
        words = processed.split()
        processed_reviews.append(' '.join([word.lower() for word in wor
    return processed_reviews

df['reviews_processed'] = preprocessing_reviews(df['review'])
df['tokens'] = df['reviews_processed'].apply(nltk.word_tokenize)
df.head()
```

```
100%|██████████| 5000/5000 [00:01<00:00, 3207.38it/s]
```

	review	sentiment	cosine_similarity	reviews_processed	tokens
0	One of the other reviewers has mentioned that ...	positive	0.025828	one of the other reviewers has mentioned that ...	[one, of, the, other, reviewers, has, mentione...
1	A wonderful little production...	positive	0.021080	wonderful little production the filming...	[wonderful, little, production...

Word2vec

```
start_time = time.time()
# embedding
model = Word2Vec(sentences=df['tokens'].tolist(),
                 sg=1,
                 vector_size=100,
                 window=5,
                 workers=4)

print(f'Time needed : {(time.time() - start_time) / 60:.2f} mins')
```

```
Time needed : 0.33 mins
```

Once we have our embedding, we can visualize it. For example, if we try clustering the vectors of some words, words that have similar meanings should be closer together:

```
all_words = list(model.wv.index_to_key)

# Convert to DataFrame for easier viewing
embedding_df = pd.DataFrame(embeddings, index=all_words)

# Display first few embeddings
print("\n=== Word Embeddings Sample ===")
display(embedding_df.head(10))

# display embedding for a specific word
word = 'love'  # <-- change to any word in your vocab
if word in model.wv:
    print(f"\nEmbedding for '{word}':\n", model.wv[word])
else:
    print(f"\nWord '{word}' not in vocabulary.")
```

```
=== Word Embeddings Sample ===
```

	0	1	2	3	4	5	6	
the	0.012338	0.313313	0.145100	0.037314	0.015930	-0.175705	0.170597	0
and	0.134682	0.325658	0.001203	0.135575	0.015877	-0.302835	-0.169765	0
of	0.052011	0.152977	-0.119471	-0.035030	-0.171498	-0.111423	-0.239469	0
to	0.077519	0.208892	0.039928	0.028621	0.136316	-0.422598	0.240107	0
is	-0.359902	-0.046754	-0.185639	0.071479	-0.211031	-0.243895	0.023164	0
in	-0.065287	0.276323	0.189219	-0.160902	-0.350229	-0.232909	-0.043510	0
it	0.145376	0.353638	0.093785	-0.088343	-0.213174	-0.376860	0.319348	0
this	0.256897	0.091520	0.276289	0.211247	-0.391885	-0.309951	0.140323	0
that	-0.130609	0.395545	-0.063280	-0.040831	-0.051134	-0.123723	0.114772	0
was	-0.262752	0.218048	-0.325372	0.440369	-0.399793	-0.320941	-0.046805	0

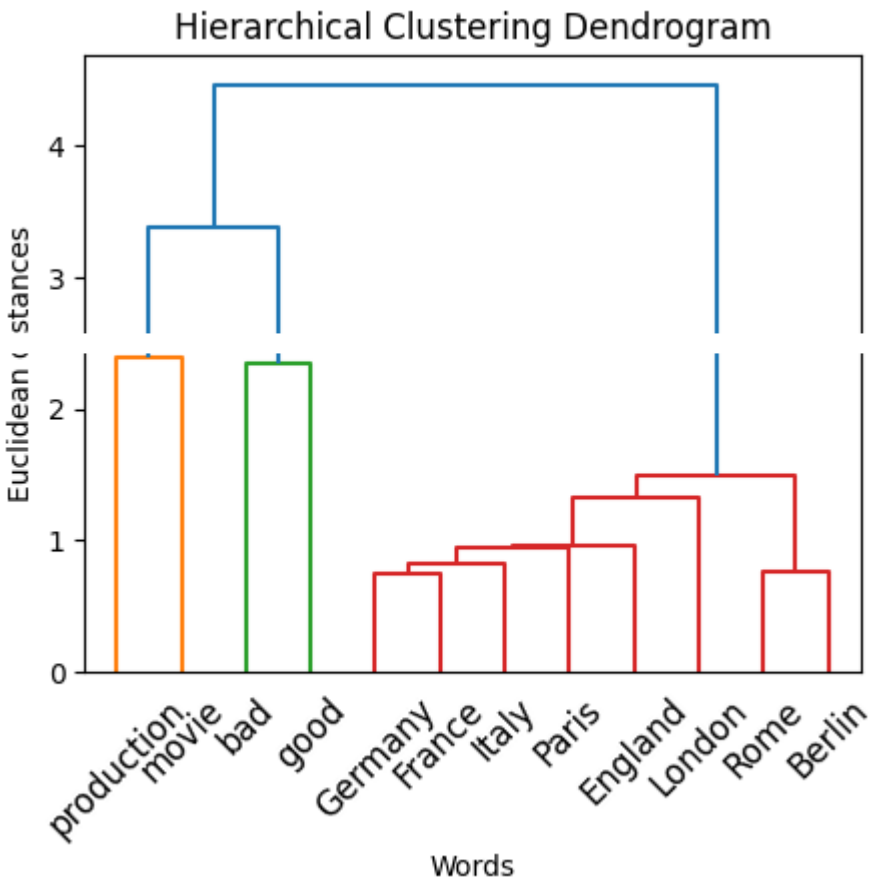
10 rows × 100 columns

```
Embedding for 'love':
 [ 0.0628069   0.4717827  -0.15149786  0.39491978 -0.0494485   0.01760162
  0.551305    0.3791465  -0.1780295  -0.2721025  -0.3487414  -0.39225456
 -0.25332513  0.26279914  0.05920301 -0.06113205 -0.38240504  0.59075826
 -0.10978506 -0.63008577 -0.14544414 -0.11722335  0.21067198 -0.26478222
 -0.03347863 -0.15971985 -0.2884307  -0.25170007 -0.07063447  0.04461943
  0.27017832 -0.05391076  0.03013221  0.01196067 -0.47779933  0.19314584
  0.05132506 -0.37791973 -0.10676209  0.11874396 -0.35057503 -0.31774685
 -0.06149433  0.15963562 -0.01191935 -0.49332827  0.2178178  -0.26135173
  0.08778144  0.4831367   0.43137723 -0.16528608  0.2321634  -0.00317804
  0.09479047 -0.09362503  0.21500687 -0.16549475 -0.0446067   0.25380078
 -0.09130144  0.13089773 -0.24086294 -0.30961266 -0.17270392  0.2563059
  0.12764916 -0.07372927 -0.105375    0.1316033  -0.22527225  0.44114193
  0.17851904  0.09044652  0.13025734  0.6231276  -0.05908431  0.03555044
 -0.27003828  0.1341773  -0.5789958  -0.4467749   0.10572103  0.18362162
 -0.08812474 -0.03886231  0.54479957  0.08028372  0.11609267  0.291487
  0.38396877  0.49682525  0.30779132  0.28518745  0.6092599   0.57112145
 -0.0490026   0.0777132  -0.50499487 -0.4391271 ]
```

Once we have our embedding, we can visualize it. For example, if we try clustering the vectors of some words, words that have similar meanings should be closer together:

```
# Entire set of words in the model
all_words = list(model.wv.index_to_key)
all_vectors = np.array([model.wv[word] for word in all_words])

# Highlighted words and their vectors
highlight_words = ['Berlin', 'Paris', 'London','Rome', 'Italy',
                   'France', 'Germany', 'England', 'movie', 'production
highs = [w.lower() for w in highlight_words]
indices = [all_words.index(word) for word in highs if word in all_words
```

```
highlight_vectors = np.array([all_vectors[index] for index in indices])

linked = linkage(highlight_vectors, 'ward')

plt.figure(figsize=(5, 4))
dendrogram(linked,
           orientation='top',
           labels=highlight_words,
           distance_sort='descending',
           show_leaf_counts=True)
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('Words')
plt.ylabel('Euclidean distances')
plt.xticks(rotation=45)
plt.savefig('word_dendrogram.jpg', format='jpeg', bbox_inches='tight')
plt.show()
```



Another way to visualize vectors is to use dimensionality reduction techniques.

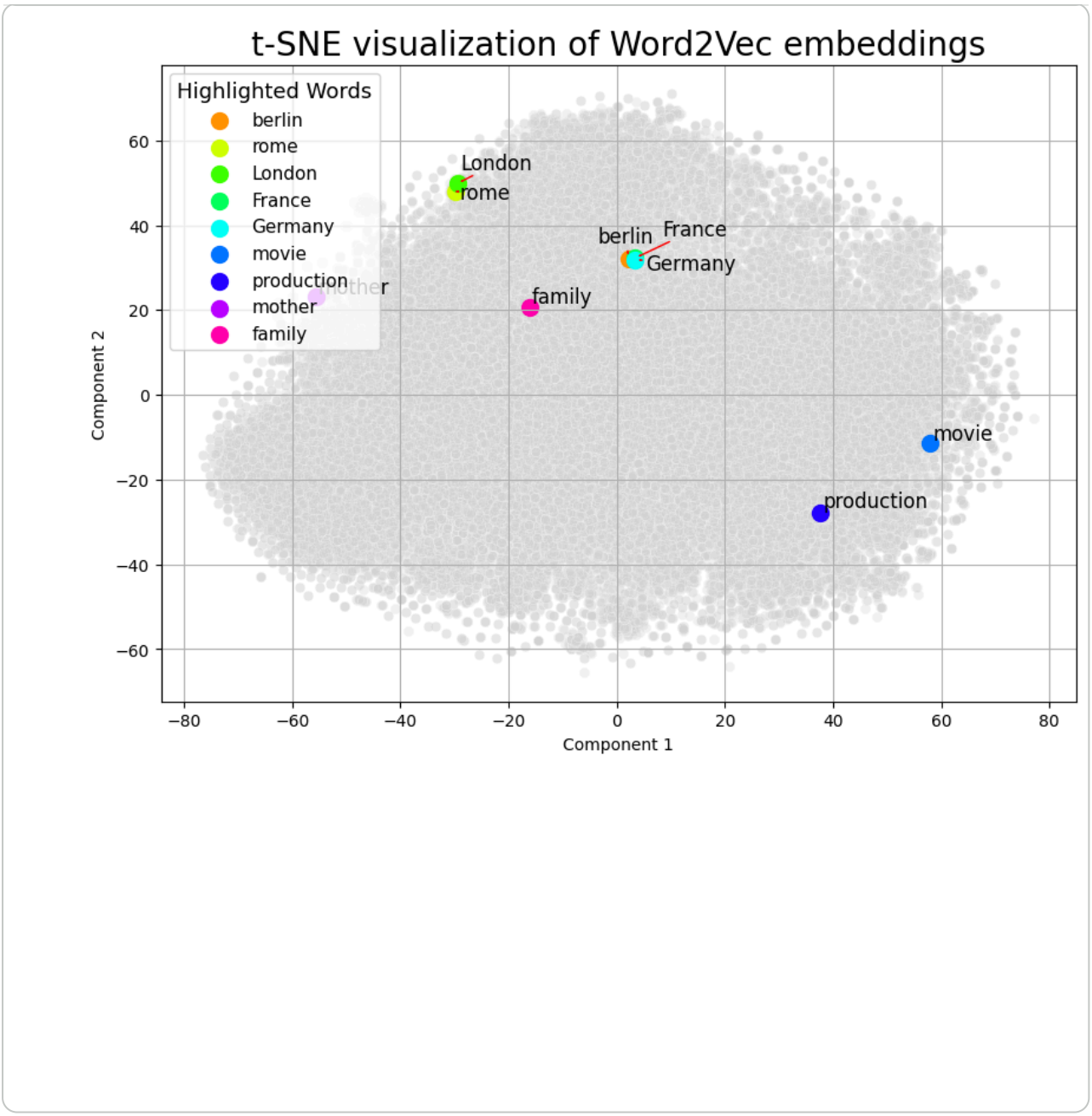
Vectors are multidimensional (100-1,024), so it is more convenient to reduce them to two or three dimensions so that they can be visualized more easily.

Some of the most commonly used techniques are principal component analysis (PCA) and t-distributed stochastic neighbor embedding (t-SNE).

Uniform Manifold Approximation and Projection (UMAP), on the other hand, is a technique that has become the first choice for visualizing multidimensional data in

recent years:

```
# Apply t-SNE to the entire set of vectors
tsne = TSNE(n_components=2, random_state=0)
Y_tsne = tsne.fit_transform(all_vectors)

highlight_words = ['berlin', 'rome', 'London', 'France', 'Germany',
                   'movie', 'production', 'mother', 'family']

highs = [w.lower() for w in highlight_words]
indices = [all_words.index(word) for word in highs if word in all_words
highlight_vectors = np.array([all_vectors[index] for index in indices])
Y_highlight = Y_tsne[indices]


plt.figure(figsize=(10, 7))


sns.scatterplot(x=Y_tsne[:, 0], y=Y_tsne[:, 1], color="lightgrey", alph

# Plot highlighted words
palette = sns.color_palette("hsv", len(highlight_words))
texts = []
for i, word in enumerate(highlight_words):
    plt.scatter(Y_highlight[i, 0], Y_highlight[i, 1], color=palette[i],
    # adjust text
    texts.append(plt.text(Y_highlight[i, 0], Y_highlight[i, 1], word, f

adjust_text(texts, arrowprops=dict(arrowstyle='->', color='red'))

plt.title('t-SNE visualization of Word2Vec embeddings', fontsize=20)
plt.xlabel('Component 1')
plt.ylabel('Component 2')


plt.grid(True)
plt.legend(title='Highlighted Words', title_fontsize='13', fontsize='11
plt.savefig('word_tsne.jpg', format='jpeg')
plt.show()
```



calculate similarity between words

Once we have obtained vector representations, we need a method to calculate the similarity between them.

This is crucial in many applications—for instance, to find words in an embedding space that are most similar to a given word, we compute the similarity between its vector and those of other words. Similarly, given a query sentence, we can retrieve the most relevant documents by comparing its vector with document embeddings and selecting those with the highest similarity.

Most similarity measures are based on the dot product. This is because the dot product is high when the two vectors have values in the same dimension. In contrast, vectors that have zero alternately will have a dot product of zero, thus orthogonal or

dissimilar. This is why the dot product was used as a similarity measure for word co-occurrence matrices or with vectors derived from document TF matrices:

The normalized dot product is equivalent to the cosine of the angle between the two vectors, hence cosine similarity:

$$cos\Theta = \frac{\boldsymbol{a} \bullet \boldsymbol{b}}{\boldsymbol{a}|\,\boldsymbol{b}|} = \frac{\sum_{i=1}^{N} a_i \times b_i}{\sqrt[2]{\sum_{i=1}^{N} a_i^2}\sqrt[2]{\sum_{i=1}^{N} b_i^2}}$$

Cosine similarity has some interesting properties:

It is between -1 and 1. Opposite or totally dissimilar vectors will have a value of -1, 0 for orthogonal vectors (or totally dissimilar for scattered vectors), and 1 for perfectly similar vectors. Since it measures the angle between two vectors, the interpretation is easier and is within a specific range, so it allows one intuitively to understand the similarity or dissimilarity.

It is fast and cheap to compute.

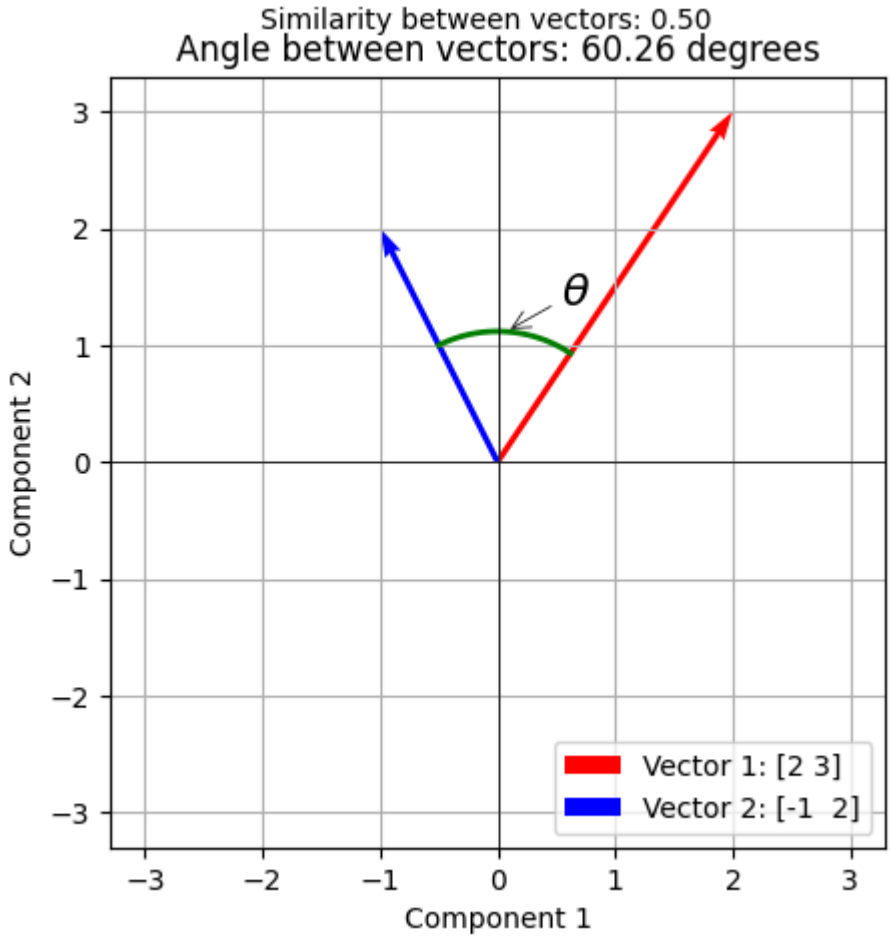
It is less sensitive to word frequency and, thus, more robust to outliers.

It is scale-invariant, meaning that it is not influenced by the magnitude of the vectors. Being normalized, it can also be used with high-dimensional data.

```
    def plot_vectors_and_angle(v1, v2):

        dot_product = np.dot(v1, v2)
        norm_v1 = np.linalg.norm(v1)
        norm_v2 = np.linalg.norm(v2)
        cosine_similarity = dot_product / (norm_v1 * norm_v2)
        angle_radians = np.arccos(cosine_similarity)
        angle_degrees = np.degrees(angle_radians)


        fig, ax = plt.subplots(figsize=(5, 5))


        ax.quiver(0, 0, v1[0], v1[1], angles='xy', scale_units='xy', scale=
        ax.quiver(0, 0, v2[0], v2[1], angles='xy', scale_units='xy', scale=


        start_angle = np.arctan2(v1[1], v1[0])
        if np.cross(v1, v2) < 0:
            angle_radians = -angle_radians


        theta = np.linspace(start_angle, start_angle + angle_radians, 100)
```

```
    r = 0.5 * min(np.linalg.norm(v1), np.linalg.norm(v2))
    x = r * np.cos(theta)
    y = r * np.sin(theta)


    ax.plot(x, y, linestyle='-', color='green', lw=2)


    midpoint = (start_angle + angle_radians / 2)
    ax.annotate(r'$\theta$', xy=(r * np.cos(midpoint), r * np.sin(midpo
                textcoords='offset points', fontsize=16, arrowprops=dic


    max_range = np.max(np.abs(np.vstack([v1, v2, [x.max(), y.max()]])))
    ax.set_xlim([-max_range, max_range])
    ax.set_ylim([-max_range, max_range])


    plt.grid(True)
    plt.axhline(0, color='black', linewidth=0.5)
    plt.axvline(0, color='black', linewidth=0.5)
    plt.title(f'Angle between vectors: {angle_degrees:.2f} degrees')
    plt.suptitle(f'Similarity between vectors: {cosine_similarity:.2f}'
    plt.xlabel('Component 1')
    plt.ylabel('Component 2')
    plt.legend(loc='lower right')
    plt.savefig('cosine_similarity.jpg', format='jpeg', bbox_inches='ti
    plt.show()

    return cosine_similarity, angle_degrees

# Example usage
v1 = np.array([2, 3])
v2 = np.array([-1, 2])
cos_sim, angle = plot_vectors_and_angle(v1, v2)
```

```
/tmp/ipython-input-1974943856.py:19: DeprecationWarning: Arrays of 2-dimensio
  if np.cross(v1, v2) < 0:
```



Properties of embeddings

Embeddings are a surprisingly flexible method and manage to encode different syntactic and semantic properties that can both be visualized and exploited for different operations.

Once we have a notion of similarity, we can search for the words that are most similar to a word w. Note that similarity is defined as appearing in the same context window; the model cannot differentiate synonyms and antonyms.

In addition, the model is also capable of representing grammatical relations such as superlatives or verb forms.

Another interesting relationship we can study is analogies. The parallelogram model is a system for representing analogies in a cognitive space. The classic example is king:queen::man:? (which in a formula would be a:b::a*:?). Given that we have vectors, we can turn this into an a-a*+b operation.

We can test this in Python using the embedding model we have trained:

We can check the most similar words We can test the analogy We can then test the capacity to identify synonyms and antonyms The code for this process is as follows:

```
word_1 = "good"
syn = "great"
ant = "bad"
most_sim =model.wv.most_similar("good")
print("Top 3 most simalr words to {} are :{}".format(word_1, most_sim[:

synonyms_dist = model.wv.distance(word_1, syn)
antonyms_dist = model.wv.distance(word_1, ant)
print("Synonyms {}, {} have cosine distance: {}".format(word_1, syn, sy
print("Antonyms {}, {} have cosine distance: {}".format(word_1, ant, an
a = 'king'
a_star = 'man'
b = 'woman'
b_star= model.wv.most_similar(positive=[a, b], negative=[a_star])
print("{} is to {} as {} is to: {} ".format(a, a_star, b, b_star[0][0])
```

```
Top 3 most simalr words to good are :[('decent', 0.7567450404167175), ('great
Synonyms good, great have cosine distance: 0.28625917434692383
Antonyms good, bad have cosine distance: 0.316448450088501
king is to man as woman is to: dunne
```

Using Sentence Transformer to understand Embeddings

The model "all-MiniLM-L6-v2" is a sentence embedding model,you can use other embedding models from HuggingFace

https://huggingface.co/sentence-transformers

```
!pip install -q sentence-transformers scikit-learn matplotlib
```

```
# --- Step 2: Import Libraries ---
from sentence_transformers import SentenceTransformer
from sklearn.decomposition import PCA
from sklearn.metrics.pairwise import cosine_similarity
import matplotlib.pyplot as plt
import numpy as np
```

```
#Load Pretrained Embedding Model ---
# You can replace this model with others like "all-MiniLM-L6-v2" or "pa
model = SentenceTransformer('all-MiniLM-L6-v2')
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: Us
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings ta
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access p
  warnings.warn(
modules.json: 100%                                  349/349 [00:00<00:00, 34.6kB/s]
config_sentence_transformers.json: 100%             116/116 [00:00<00:00, 14.9kB/s]
README.md:         10.5k/? [00:00<00:00, 1.05MB/s]
sentence_bert_config.json: 100%                     53.0/53.0 [00:00<00:00, 6.92kB/s]
config.json: 100%                                   612/612 [00:00<00:00, 57.5kB/s]
model.safetensors: 100%                             90.9M/90.9M [00:00<00:00, 162MB/s]
tokenizer_config.json: 100%                         350/350 [00:00<00:00, 29.7kB/s]
vocab.txt:         232k/? [00:00<00:00, 8.85MB/s]
tokenizer.json:         466k/? [00:00<00:00, 27.8MB/s]
special_tokens_map.json: 100%                       112/112 [00:00<00:00, 11.9kB/s]
config.json: 100%                                   190/190 [00:00<00:00, 14.1kB/s]
```

```
#Example Sentences ---
sentences = [
    "Cats are wonderful pets.",
    "Dogs are loyal and friendly.",
    "The sky is blue and clear today.",
    "I love to play with my dog.",
    "Artificial intelligence is transforming healthcare."
]
```

```
#display embedding of a single word
word = "pets"
# Generate embedding
embedding = model.encode(word)

print(f"Shape of embedding vector: {embedding.shape}")
print("Embedding values (first 10 dimensions):", embedding[:10])
```

```
Shape of embedding vector: (384,)
Embedding values (first 10 dimensions): [ 7.98705500e-03  1.01199448e-02  3.9
 -1.06959045e-01  9.81669800e-05  7.06559941e-02 -5.11679165e-02
  5.48836812e-02  2.15079561e-02]
```

```
sentence_embeddings = model.encode(sentences)
for i, sentence in enumerate(sentences):
    print(f"Sentence: {sentence}")
```

```
        print(f"Sentence embedding shape: {sentence_embeddings[i].shape}")
        print(f"First 10 dimensions: {sentence_embeddings[i][:10]}\n")

        # Generate embeddings for each word (approximation using the same m
        words = sentence.split()
        for word in words:
            word_embedding = model.encode(word)
            print(f"  Word: {word}")
            print(f"  Embedding shape: {word_embedding.shape}")
            print(f"  First 10 dimensions: {word_embedding[:10]}\n")

        print("-" * 80)
```

```
 0.0683144   0.0059432  -0.00331671  0.05505008]

 Word: transforming
 Embedding shape: (384,)
 First 10 dimensions: [-0.01721075  0.0921106  -0.01783849 -0.01299875 -0.(
 0.00853476 -0.02988112  0.04077945 -0.0169568 ]

 Word: healthcare.
 Embedding shape: (384,)
 First 10 dimensions: [-0.0023908   0.06769317  0.01712604  0.046023   -0.(
 0.10530414 -0.00095508 -0.00768973 -0.00287271]
```

```
# Generate Sentence Embeddings ---
#embeddings = model.encode(sentences)

print("Embeddings Generated!")
print(f"Shape of embedding matrix: {sentence_embeddings.shape}")
```

```
Embeddings Generated!
Shape of embedding matrix: (5, 384)
```

```
#Calculate similarities in sentenses
similarity_matrix = model.similarity(sentence_embeddings, sentence_embe
print("\nCosine Similarity Between Sentences:\n")
print(similarity_matrix)
```

```
Cosine Similarity Between Sentences:

tensor([[ 1.0000,  0.5055,  0.0583,  0.4395,  0.1039],
        [ 0.5055,  1.0000,  0.0176,  0.5127, -0.0222],
        [ 0.0583,  0.0176,  1.0000, -0.0037,  0.0909],
        [ 0.4395,  0.5127, -0.0037,  1.0000,  0.1084],
        [ 0.1039, -0.0222,  0.0909,  0.1084,  1.0000]])
```

Observe sentence similraity for example in first row showing similarity of sentence 1 to other four sentences. for example

Similarity Interpretation S1–S2 0.5055 Moderately related — e.g., “Cats are wonderful pets” vs. “Dogs are loyal and friendly.” (both about pets/animals)

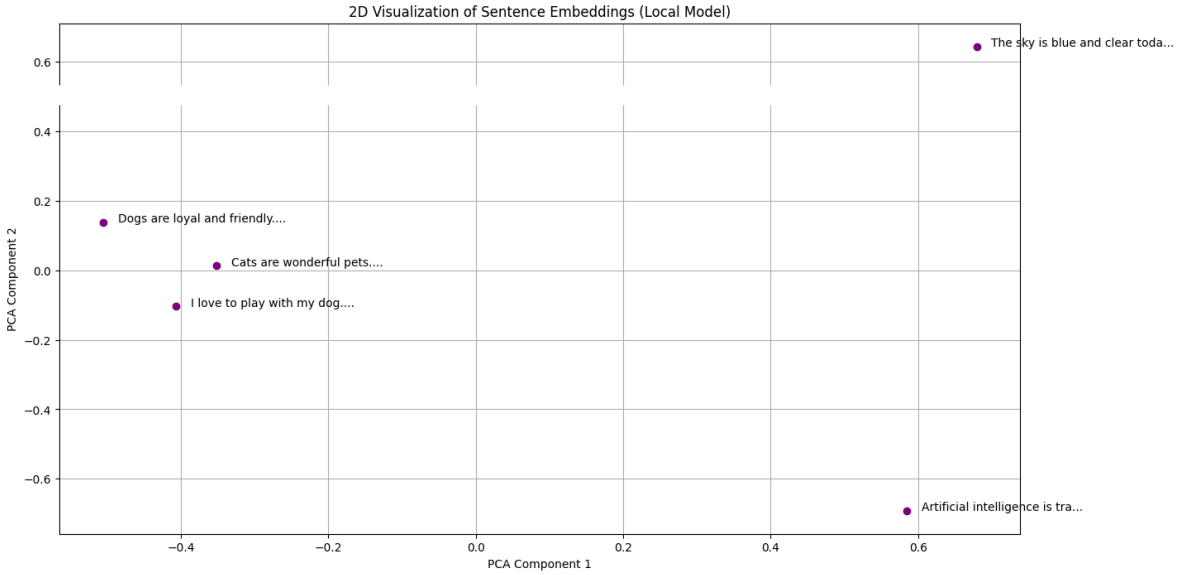
S1–S3 0.0583 Unrelated — “Cats are wonderful pets” vs. “The sky is blue”

S1–S4 0.4395 Somewhat related — “Cats are wonderful pets” vs. “I love to play with my dog.” (both about domestic animals)

S1–S5 0.1039 Very weakly related — “Cats…” vs. “AI in healthcare.” (completely different domains)

```
# Visualize in 2D Using PCA ---
pca = PCA(n_components=2)
reduced_embeddings = pca.fit_transform(sentence_embeddings)
```

```
plt.figure(figsize=(15,8))
plt.scatter(reduced_embeddings[:,0], reduced_embeddings[:,1], color='pu

for i, sentence in enumerate(sentences):
    plt.annotate(sentence[:30] + "...", (reduced_embeddings[i,0]+0.02,

plt.title("2D Visualization of Sentence Embeddings (Local Model)")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
plt.grid(True)
plt.show()
```



Understanding Text Generation Models

When you use an LLM, two models are loaded:

The generative model itself

Its underlying tokenizer

The tokenizer is in charge of splitting the input text into tokens before feeding it to the generative model.

You can find the tokenizer and model on the Hugging Face site and only need the corresponding IDs to be passed.

In this case, we use “microsoft/Phi-3-mini-4k-instruct” as the main path to the model.

Running the code will start downloading the model and depending on your internet connection can take a couple of minutes.

```
 %%capture
!pip install -q transformers>=4.40.1 accelerate>=0.27.2
```

```
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    torch_dtype="auto",
    trust_remote_code=False,
)
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-inst
```

```
config.json: 100%                                                 967/967 [00:00<00:00, 27.3kB/s]
`torch_dtype` is deprecated! Use `dtype` instead!
model.safetensors.index.json:         16.5k/? [00:00<00:00, 397kB/s]
Fetching 2 files: 100%                                              2/2 [01:30<00:00, 90.11s/it]
model-00002-of-                                          2.67G/2.67G [01:23<00:00, 45.7MB/s]
00002.safetensors: 100%
model-00001-of-                                          4.97G/4.97G [01:29<00:00, 258MB/s]
00002.safetensors: 100%
Loading checkpoint shards: 100%                                     2/2 [00:00<00:00,  3.99it/s]
generation_config.json: 100%                                      181/181 [00:00<00:00, 22.6kB/s]
tokenizer_config.json:          3.44k/? [00:00<00:00, 351kB/s]
tokenizer.model: 100%                                           500k/500k [00:00<00:00, 1.96MB/s]
tokenizer.json:          1.94M/? [00:00<00:00, 41.1MB/s]
added_tokens.json: 100%                                           306/306 [00:00<00:00, 820B/s]
```

```
print(model)
# Explore models architecture
```

```
Phi3ForCausalLM(
  (model): Phi3Model(
    (embed_tokens): Embedding(32064, 3072, padding_idx=32000)
    (layers): ModuleList(
      (0-31): 32 x Phi3DecoderLayer(
        (self_attn): Phi3Attention(
          (o_proj): Linear(in_features=3072, out_features=3072, bias=False)
          (qkv_proj): Linear(in_features=3072, out_features=9216, bias=False)
        )
        (mlp): Phi3MLP(
          (gate_up_proj): Linear(in_features=3072, out_features=16384, bias=F
          (down_proj): Linear(in_features=8192, out_features=3072, bias=False
          (activation_fn): SiLUActivation()
        )
        (input_layernorm): Phi3RMSNorm((3072,), eps=1e-05)
        (post_attention_layernorm): Phi3RMSNorm((3072,), eps=1e-05)
        (resid_attn_dropout): Dropout(p=0.0, inplace=False)
        (resid_mlp_dropout): Dropout(p=0.0, inplace=False)
      )
    )
    (norm): Phi3RMSNorm((3072,), eps=1e-05)
    (rotary_emb): Phi3RotaryEmbedding()
  )
  (lm_head): Linear(in_features=3072, out_features=32064, bias=False)
)
```

How Tokenizers Prepare the Inputs to the Language Model

You can find an example showing the tokenizer of GPT-4 on the

https://platform.openai.com/tokenizer

We can then proceed to the actual generation. We first declare our prompt, then tokenize it, then pass those tokens to the model, which generates its output. In this case, we're asking the model to only generate 20 new tokens:

```
# Define prompt
prompt = "Write an email apologizing to Sarah for the tragic gardening

# Tokenize input properly
inputs = tokenizer(prompt, return_tensors="pt")

# Generate text
generation_output = model.generate(
    input_ids=inputs["input_ids"],
    max_new_tokens=20,
    do_sample=True,
    temperature=0.7
)

# Decode and print result
output_text = tokenizer.decode(generation_output[0], skip_special_token
print(output_text)
```

```
The attention mask is not set and cannot be inferred from input because pad t
Write an email apologizing to Sarah for the tragic gardening mishap. Explain


D
```

```
print(inputs["input_ids"])
```

```
tensor([[14350,   385,  4876, 27746,  5281,   304, 19235,   363,   278, 25305
          293, 16423,   292,   286,   728,   481, 29889, 12027,  7420,   920
          372,  9559, 29889, 32001]])
```

```
# Print token IDs and decoded tokens for the input
print("\nToken IDs:", inputs["input_ids"][0].tolist())
print("\nDecoded tokens:")
for token_id in inputs["input_ids"][0]:
    print(f"{token_id.item()} -> {tokenizer.decode(token_id)}")
```

```

Token IDs: [14350, 385, 4876, 27746, 5281, 304, 19235, 363, 278, 25305, 293,

Decoded tokens:
14350 -> Write
385 -> an
```

```
4876 -> email
27746 -> apolog
5281 -> izing
```