In [43]:
```
import spacy

nlp = spacy.load("en_core_web_sm")
piano_text = "Gus is learning piano"
piano_doc = nlp(piano_text)

for token in piano_doc:
    print(
        f"""
TOKEN: {token.text}
=====
{token.tag_ =}
{token.head.text =}
{token.dep_ =}"""
    )
```

```
TOKEN: Gus
=====
token.tag_ ='NNP'
token.head.text ='learning'
token.dep_ ='nsubj'

TOKEN: is
=====
token.tag_ ='VBZ'
token.head.text ='learning'
token.dep_ ='aux'

TOKEN: learning
=====
token.tag_ ='VBG'
token.head.text ='learning'
token.dep_ ='ROOT'

TOKEN: piano
=====
token.tag_ ='NN'
token.head.text ='learning'
token.dep_ ='dobj'
```

In [44]:
```
displacy.serve(piano_doc, style="dep")
```

```
C:\Users\akile\anaconda3\Lib\site-packages\spacy\displacy\__init__.py:106: UserWarning: [W011] It looks like you're calling displacy.serve from within a Jupyter notebook or a similar environment. This likely means you're already running a local web server, so there's no need to make displaCy start another one. Instead, you should be able to replace displacy.serve with displacy.render to show the visualization.
  warnings.warn(Warnings.W011)
```

displaCy

Gus PROPN is AUX learning VERB piano NOUN nsubj aux dobj

```
Using the 'dep' visualizer
Serving on http://0.0.0.0:5000 (http://0.0.0.0:5000) ...


127.0.0.1 - - [03/Jan/2024 10:20:30] "GET / HTTP/1.1" 200 3399
127.0.0.1 - - [03/Jan/2024 10:20:30] "GET /favicon.ico HTTP/1.1" 200 3399

Shutting down server on port 5000.
```

In [45]:
```
import spacy

nlp = spacy.load("en_core_web_sm")

one_line_about_text = (
    "Gus Proto is a Python developer"
    " currently working for a London-based Fintech company"
)

one_line_about_doc = nlp(one_line_about_text)

# Extract children of `developer`
print([token.text for token in one_line_about_doc[5].children])

# Extract previous neighboring node of `developer`
print(one_line_about_doc[5].nbor(-1))

# Extract next neighboring node of `developer`
print(one_line_about_doc[5].nbor())

# Extract all tokens on the left of `developer`
print([token.text for token in one_line_about_doc[5].lefts])

# Extract tokens on the right of `developer`
print([token.text for token in one_line_about_doc[5].rights])

# Print subtree of `developer`
print(list(one_line_about_doc[5].subtree))
```

```
['a', 'Python', 'working']
Python
currently
['a', 'Python']
['working']
[a, Python, developer, currently, working, for, a, London, -, based, Fintech,
company]
```

Named-Entity Recognition

In [46]:
```
import spacy

nlp = spacy.load("en_core_web_sm")

piano_class_text = (
    "Great Piano Academy is situated"
    " in Mayfair or the City of London and has"
    " world-class piano instructors."
)

piano_class_doc = nlp(piano_class_text)

for ent in piano_class_doc.ents:
    print(
        f"""
{ent.text =}
{ent.start_char =}
{ent.end_char =}
{ent.label_ =}
spacy.explain('{ent.label_}') = {spacy.explain(ent.label_)}"""
    )
```

```
ent.text ='Great Piano Academy'
ent.start_char =0
ent.end_char =19
ent.label_ ='ORG'
spacy.explain('ORG') = Companies, agencies, institutions, etc.

ent.text ='Mayfair'
ent.start_char =35
ent.end_char =42
ent.label_ ='GPE'
spacy.explain('GPE') = Countries, cities, states

ent.text ='the City of London'
ent.start_char =46
ent.end_char =64
ent.label_ ='GPE'
spacy.explain('GPE') = Countries, cities, states
```

In [47]:
```
displacy.serve(piano_class_doc, style="ent")
```

displaCy

Great Piano Academy ORG is situated in Mayfair GPE or the City of London GPE

and has world-class piano instructors.

```
Using the 'ent' visualizer
Serving on http://0.0.0.0:5000 (http://0.0.0.0:5000) ...


127.0.0.1 - - [03/Jan/2024 10:21:32] "GET / HTTP/1.1" 200 1425
127.0.0.1 - - [03/Jan/2024 10:21:32] "GET /favicon.ico HTTP/1.1" 200 1425

Shutting down server on port 5000.
```

In [48]:
```
survey_text = (
    "Out of 5 people surveyed, James Robert,"
    " Julie Fuller and Benjamin Brooks like"
    " apples. Kelly Cox and Matthew Evans"
    " like oranges."
)

def replace_person_names(token):
    if token.ent_iob != 0 and token.ent_type_ == "PERSON":
        return "[REDACTED] "
    return token.text_with_ws

def redact_names(nlp_doc):
    with nlp_doc.retokenize() as retokenizer:
        for ent in nlp_doc.ents:
            retokenizer.merge(ent)
    tokens = map(replace_person_names, nlp_doc)
    return "".join(tokens)

survey_doc = nlp(survey_text)
print(redact_names(survey_doc))
```

```
Out of 5 people surveyed, [REDACTED] , [REDACTED] and [REDACTED] like apples.
[REDACTED] and [REDACTED] like oranges.
```

INFERENCE:

In this NLP lab, we installed and utilized spaCy, exploring various text processing tasks. We began by creating a Doc object for processed text and implemented custom rules for sentence detection. Tokenization revealed insights into attributes such as alphanumeric nature, punctuation, and stop word status. The lab delved into dependency parsing, showcasing spaCy's ability to unveil syntactic relationships. Lemmatization was demonstrated for effective handling of word forms. Practical applications, including stop

word removal and lemmatization, were showcased to enhance data quality. The exploration covered word frequency analysis, part-of-speech tagging, and visualizations using displaCy. Preprocessing functions highlighted the importance of customized text cleaning. In summary, the lab provided a comprehensive understanding of spaCy's capabilities, encompassing advanced features like dependency parsing, for diverse aspects of natural language processing

In [ ]:

NLP LAB 3

AIM:

To explore various various techniques for word embedding, including One-Hot Encoding, Bag of Words with CountVectorizer, Tf-Idf (term frequency-inverse document frequency), Word2Vec, and GloVe etc.

One-Hot Encoding

One-hot encoding is a technique used in machine learning and natural language processing to represent categorical variables as binary vectors. It is a way to convert categorical data, which may take on values from a discrete set, into a binary matrix with a unique column for each category.

In [4]:
```
import numpy as np

# Step 1: Convert Text to lower case
text = "My Name is AKilesh"
text_lower = text.lower()

# Step 2: Tokenize the text
tokens = text_lower.split()

# Step 3: Get unique words
unique_words = set(tokens)

# Step 4: Sort the word list
sorted_words = sorted(unique_words)

# Step 5: Get the integer/position of the words
word_to_index = {word: index for index, word in enumerate(sorted_words)}

# Step 6 and 7: Create one-hot vectors and matrix
one_hot_matrix = np.zeros((len(tokens), len(unique_words)))

for i, token in enumerate(tokens):
    word_index = word_to_index[token]
    one_hot_matrix[i, word_index] = 1

# Display the results
print("Original Text:", text)
print("One-Hot Matrix:")
print(one_hot_matrix)
```

```
Original Text: My Name is AKilesh
One-Hot Matrix:
[[0. 0. 1. 0.]
 [0. 0. 0. 1.]
 [0. 1. 0. 0.]
 [1. 0. 0. 0.]]
```

Bag of Words

BoW is a representation model for text data in NLP.It treats a document as an unordered set or "bag" of words, disregarding grammar and word order but keeping track of the frequency of each word.The basic

idea is to convert text into numerical vectors, where each dimension of the vector corresponds to a unique word in the vocabulary, and the value in each dimension represents the frequency of that word in the document.

CountVectorizer

CountVectorizer is a specific implementation of the BoW model in the scikit-learn library, which is a popular machine learning library in Python.It converts a collection of text documents into a matrix of token counts.Each row of the matrix represents a document, and each column represents a unique word in the corpus. The matrix's entries are the word frequencies in each document.The output is a sparse matrix where most entries are zero, as not every word in the entire vocabulary appears in every document.

In [5]:
```
from sklearn.feature_extraction.text import CountVectorizer

# Example Sentences
sentence1 = "My Name is AKilesh"
sentence2 = "I am a student at AVV chennai"

# Step 1: Convert to lowercase
sentence1 = sentence1.lower()
sentence2 = sentence2.lower()

# Step 2: Remove special characters and stopwords (optional)
# In this example, we'll skip this step for simplicity.

# Step 3: Create a list of sentences
corpus = [sentence1, sentence2]

# Step 4: Use CountVectorizer to create Bag of Words model
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)

# Display the vocabulary and the Bag of Words matrix
print("Vocabulary:", vectorizer.get_feature_names_out())
print("Bag of Words Matrix:")
print(X.toarray())
```

```
Vocabulary: ['akilesh' 'am' 'at' 'avv' 'chennai' 'is' 'my' 'name' 'student']
Bag of Words Matrix:
[[1 0 0 0 0 1 1 1 0]
 [0 1 1 1 1 0 0 0 1]]
```

Tf-Idf (term frequency-inverse document frequency)

TF-IDF, or Term Frequency-Inverse Document Frequency, is a numerical statistic that measures the importance of a word in a document relative to a collection of documents. It combines two components: Term Frequency (TF), which quantifies the frequency of a term in a document, and Inverse Document Frequency (IDF), which measures the importance of a term across the entire document collection.

In [7]:
```
import pandas as pd
import numpy as np

# Construct a small corpus
corpus = ['data science is one of the most important fields of science',
          'this is one of the best data science courses',
          'data scientists analyze data']

# Create a word set for the corpus
words_set = set()

for doc in corpus:
    words = doc.split(' ')
    words_set = words_set.union(set(words))

print('Number of words in the corpus:', len(words_set))
print('The words in the corpus:\n', words_set)

# Computing Term Frequency (TF)
n_docs = len(corpus)
n_words_set = len(words_set)

df_tf = pd.DataFrame(np.zeros((n_docs, n_words_set)), columns=list(words_set))

# Compute Term Frequency (TF)
for i in range(n_docs):
    words = corpus[i].split(' ')  # Words in the document
    for w in words:
        df_tf[w][i] = df_tf[w][i] + (1 / len(words))

print("Term Frequency (TF):")
print(df_tf)

# Computing Inverse Document Frequency (IDF)
print("\nInverse Document Frequency (IDF):")
idf = {}

for w in words_set:
    k = 0  # number of documents in the corpus that contain this word

    for i in range(n_docs):
        if w in corpus[i].split():
            k += 1

    idf[w] = np.log10(n_docs / k)
    print(f'{w:>15}: {idf[w]:>10}')

# Putting it Together: Computing TF-IDF
df_tf_idf = df_tf.copy()

for w in words_set:
    for i in range(n_docs):
        df_tf_idf[w][i] = df_tf[w][i] * idf[w]

print("\nTF-IDF:")
print(df_tf_idf)
```

```
Number of words in the corpus: 14
The words in the corpus:
 {'is', 'one', 'most', 'this', 'of', 'best', 'scientists', 'the', 'courses', 'important',
'fields', 'data', 'analyze', 'science'}
Term Frequency (TF):
         is       one      most      this        of      best  scientists  \
0  0.090909  0.090909  0.090909  0.000000  0.181818  0.000000        0.00
1  0.111111  0.111111  0.000000  0.111111  0.111111  0.111111        0.00
2  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000        0.25

        the   courses  important    fields      data  analyze   science
0  0.090909  0.000000   0.090909  0.090909  0.090909     0.00  0.181818
1  0.111111  0.111111   0.000000  0.000000  0.111111     0.00  0.111111
2  0.000000  0.000000   0.000000  0.000000  0.500000     0.25  0.000000

Inverse Document Frequency (IDF):
            is: 0.17609125905568124
           one: 0.17609125905568124
          most: 0.47712125471966244
          this: 0.47712125471966244
            of: 0.17609125905568124
          best: 0.47712125471966244
    scientists: 0.47712125471966244
           the: 0.17609125905568124
       courses: 0.47712125471966244
     important: 0.47712125471966244
        fields: 0.47712125471966244
          data:        0.0
       analyze: 0.47712125471966244
       science: 0.17609125905568124

TF-IDF:
         is       one      most      this        of      best  scientists  \
0  0.016008  0.016008  0.043375  0.000000  0.032017  0.000000     0.00000
1  0.019566  0.019566  0.000000  0.053013  0.019566  0.053013     0.00000
2  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000     0.11928

        the   courses  important    fields  data  analyze   science
0  0.016008  0.000000   0.043375  0.043375   0.0  0.00000  0.032017
1  0.019566  0.053013   0.000000  0.000000   0.0  0.00000  0.019566
2  0.000000  0.000000   0.000000  0.000000   0.0  0.11928  0.000000
```

WORD2VEC

Word2Vec is a popular technique in natural language processing (NLP) that is used to represent words as dense vectors of real numbers.Word2Vec is a shallow neural network-based model designed to learn distributed representations of words in a continuous vector space.It captures semantic relationships between words by placing similar words close to each other in the vector space.

In [29]:
```
pip install gensim
```

```
Requirement already satisfied: gensim in c:\users\akile\anaconda3\lib\site-packages (4.
3.0)
Requirement already satisfied: numpy>=1.18.5 in c:\users\akile\anaconda3\lib\site-packa
ges (from gensim) (1.24.3)
Requirement already satisfied: scipy>=1.7.0 in c:\users\akile\anaconda3\lib\site-packag
es (from gensim) (1.11.1)
Requirement already satisfied: smart-open>=1.8.1 in c:\users\akile\anaconda3\lib\site-p
ackages (from gensim) (5.2.1)
Collecting FuzzyTM>=0.4.0 (from gensim)
  Downloading FuzzyTM-2.0.5-py3-none-any.whl (29 kB)
Requirement already satisfied: pandas in c:\users\akile\anaconda3\lib\site-packages (fr
om FuzzyTM>=0.4.0->gensim) (2.0.3)
Collecting pyfume (from FuzzyTM>=0.4.0->gensim)
  Downloading pyFUME-0.2.25-py3-none-any.whl (67 kB)
     ---------------------------------------- 0.0/67.1 kB ? eta -:--:--
     ---------------------------------- --- 61.4/67.1 kB 1.7 MB/s eta 0:00:01
     ---------------------------------------- 67.1/67.1 kB 1.8 MB/s eta 0:00:00
Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\akile\anaconda3\lib\s
ite-packages (from pandas->FuzzyTM>=0.4.0->gensim) (2.8.2)
```

In [31]:
```
import gensim
from gensim.models import KeyedVectors

import gensim.downloader as api

# Download the pre-trained Word2Vec model from Google (takes some time)
# This downloads a large file, so be patient.
word2vec_model_path = api.load("word2vec-google-news-300", return_path=True)

# Load the Word2Vec model using gensim
word2vec_model = gensim.models.KeyedVectors.load_word2vec_format(word2vec_model_path, binar

# Example: Get the word embedding for the word "king"
word_embedding = word2vec_model["king"]

# Print the dimensionality of the word embedding
print("Dimensionality of word embedding:", len(word_embedding))

# Example: Get the most similar words to "king"
similar_words = word2vec_model.most_similar("king")
print("Words most similar to 'king':", similar_words)

# Example: Calculate the similarity between two words
similarity_score = word2vec_model.similarity("king", "queen")
print("Similarity between 'king' and 'queen':", similarity_score)

# Example: Calculate the vector representing the combination of words "king" and "man" minus
result_vector = word2vec_model.most_similar(positive=["king", "man"], negative=["woman"], t
print("Vector representation of 'king - man + woman':", result_vector)
```

```
Dimensionality of word embedding: 300
Words most similar to 'king': [('kings', 0.7138046622276306), ('queen', 0.651095628738403
3), ('monarch', 0.6413194537162781), ('crown_prince', 0.6204220056533813), ('prince', 0.61
59993410110474), ('sultan', 0.5864824056625366), ('ruler', 0.5797566771507263), ('prince
s', 0.5646552443504333), ('Prince_Paras', 0.5432944297790527), ('throne', 0.54221045970916
75)]
Similarity between 'king' and 'queen': 0.6510957
Vector representation of 'king - man + woman': [('kings', 0.6490575075149536)]
```

In [16]:
```
pip install spacy
```

```
Requirement already satisfied: spacy in c:\users\akile\anaconda3\lib\site-packages (3.
7.2)
Requirement already satisfied: spacy-legacy<3.1.0,>=3.0.11 in c:\users\akile\anaconda3
\lib\site-packages (from spacy) (3.0.12)
Requirement already satisfied: spacy-loggers<2.0.0,>=1.0.0 in c:\users\akile\anaconda3
\lib\site-packages (from spacy) (1.0.5)
Requirement already satisfied: murmurhash<1.1.0,>=0.28.0 in c:\users\akile\anaconda3\li
b\site-packages (from spacy) (1.0.10)
Requirement already satisfied: cymem<2.1.0,>=2.0.2 in c:\users\akile\anaconda3\lib\site
-packages (from spacy) (2.0.8)
Requirement already satisfied: preshed<3.1.0,>=3.0.2 in c:\users\akile\anaconda3\lib\si
te-packages (from spacy) (3.0.9)
Requirement already satisfied: thinc<8.3.0,>=8.1.8 in c:\users\akile\anaconda3\lib\site
-packages (from spacy) (8.2.2)
Requirement already satisfied: wasabi<1.2.0,>=0.9.1 in c:\users\akile\anaconda3\lib\sit
e-packages (from spacy) (1.1.2)
Requirement already satisfied: srsly<3.0.0,>=2.4.3 in c:\users\akile\anaconda3\lib\site
-packages (from spacy) (2.4.8)
Requirement already satisfied: catalogue<2.1.0,>=2.0.6 in c:\users\akile\anaconda3\lib
```

In [20]:
```
!python -m spacy download en_core_web_md
```

```
Collecting en-core-web-md==3.7.1
  Downloading https://github.com/explosion/spacy-models/releases/download/en_core_web_m
d-3.7.1/en_core_web_md-3.7.1-py3-none-any.whl (https://github.com/explosion/spacy-model
s/releases/download/en_core_web_md-3.7.1/en_core_web_md-3.7.1-py3-none-any.whl) (42.8 M
B)
     ---------------------------------------- 0.0/42.8 MB ? eta -:--:--
     ---------------------------------------- 0.0/42.8 MB ? eta -:--:--
     ---------------------------------------- 0.1/42.8 MB 660.6 kB/s eta 0:01:05
     ---------------------------------------- 0.5/42.8 MB 3.5 MB/s eta 0:00:13
      --------------------------------------- 1.0/42.8 MB 5.7 MB/s eta 0:00:08
     - -------------------------------------- 1.5/42.8 MB 7.2 MB/s eta 0:00:06
     - -------------------------------------- 2.0/42.8 MB 7.7 MB/s eta 0:00:06
     -- ------------------------------------- 2.7/42.8 MB 8.8 MB/s eta 0:00:05
     --- ------------------------------------ 3.6/42.8 MB 10.4 MB/s eta 0:00:04
     ---- ----------------------------------- 4.4/42.8 MB 10.7 MB/s eta 0:00:04
     ---- ----------------------------------- 5.2/42.8 MB 11.8 MB/s eta 0:00:04
     ----- ---------------------------------- 5.9/42.8 MB 11.9 MB/s eta 0:00:04
     ------ --------------------------------- 6.9/42.8 MB 12.6 MB/s eta 0:00:03
     ------- -------------------------------- 7.6/42.8 MB 12.8 MB/s eta 0:00:03
```

GLOVE

GloVe, which stands for Global Vectors for Word Representation, is an unsupervised learning algorithm for obtaining vector representations (embeddings) of words.GloVe is a word embedding model that learns vector representations for words based on their global co-occurrence statistics within a given corpus.It leverages the idea that the meaning of words can be inferred from the contexts in which they frequently appear.

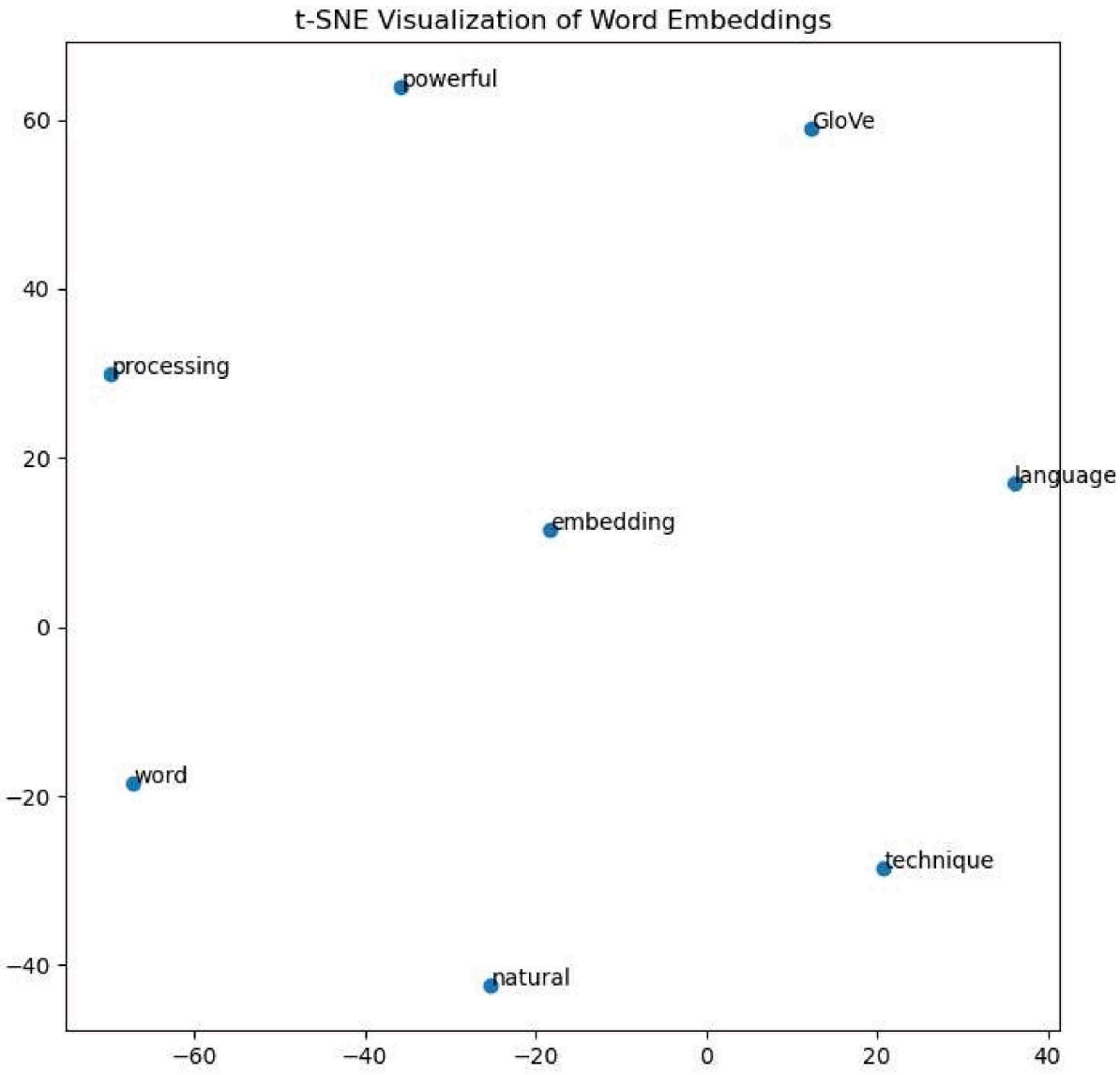
In [21]:
```
import spacy

# Load the pre-trained GloVe model
nlp = spacy.load("en_core_web_md")

# Test sentences
sentence1 = "I love natural language processing."
sentence2 = "GloVe is a powerful word embedding technique."

# Tokenize and get word embeddings for each sentence
doc1 = nlp(sentence1)
doc2 = nlp(sentence2)

# Print word embeddings for each word in the sentences
print("Word embeddings for sentence 1:")
for token in doc1:
    print(f"{token.text}: {token.vector}")

print("\nWord embeddings for sentence 2:")
for token in doc2:
    print(f"{token.text}: {token.vector}")
```

```
Word embeddings for sentence 1:
I: [ -1.8607     0.15804   -4.1425    -8.6359   -16.955      1.157
  -1.588      5.6609   -12.03      16.417      4.1907     5.5122
  -0.11932   -6.06       3.8957    -7.8212     3.6736   -14.824
  -7.6638     2.5344     7.9893     3.6785     4.3296   -11.338
  -3.5506    -5.899      1.0998     3.4515    -5.4191     1.8356
  -2.902     -7.9294    -1.1269     8.4124     5.1416    -3.1489
  -4.2061    -1.459      7.8313     0.27859   -4.3832     8.0756
  -0.94784   -6.1214     8.2792     5.0529    -8.3611    -6.0743
  -0.53773    2.7538     3.8162    -4.1612     0.7591    -2.8374
  -6.4851    -3.3435     3.2703     2.759      2.6645     4.0013
  13.381     -5.2907    -3.133      4.5374   -11.899     -6.716
  -0.041939  -2.0879     3.0101    10.3        2.6835     2.7265
   8.3018    -4.4563    14.43       3.9642    -4.8287    -5.648
  -7.2597   -11.475     -2.6171     0.3325    14.454     -5.155
   0.93722   -2.6187    -1.783      3.8711     1.4681    -6.705
  -4.0953    -0.22536    9.444    -10.305     -0.13202   -2.5534
   0.36113   -8.539      2.6755    -2.5872     2.8679     9.7515
  -2.1221     0.82061  -10.319      1.1547    -6.5808     4.9236
```

In [6]:
```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

# Define a list of strings as your sample text
x = ['text', 'the', 'leader', 'prime', 'natural', 'language']

# Create the tokenizer
tokenizer = Tokenizer()
tokenizer.fit_on_texts(x)

# Number of unique words in the dictionary
print("Number of unique words in the dictionary =", len(tokenizer.word_index))
print("Dictionary is =", tokenizer.word_index)


# Function to create an embedding matrix for the vocabulary
def embedding_for_vocab(filepath, word_index, embedding_dim):
    vocab_size = len(word_index) + 1

    # Adding again 1 because of reserved 0 index
    embedding_matrix_vocab = np.zeros((vocab_size, embedding_dim))

    with open(filepath, encoding="utf8") as f:
        for line in f:
            word, *vector = line.split()
            if word in word_index:
                idx = word_index[word]
                embedding_matrix_vocab[idx] = np.array(vector, dtype=np.float32)[:embedding

    return embedding_matrix_vocab

# Define the embedding dimension
embedding_dim = 100

# Generate the embedding matrix for the given GloVe file
embedding_matrix_vocab = embedding_for_vocab('glove.6B.100d.txt', tokenizer.word_index, emb

# Print the dense vector for the first word in the vocabulary
```

```
Number of unique words in the dictionary = 6
Dictionary is = {'text': 1, 'the': 2, 'leader': 3, 'prime': 4, 'natural': 5, 'language':
6}
```

In [28]:
```
import spacy
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
import numpy as np

# Load the pre-trained GloVe model
nlp = spacy.load("en_core_web_md")

# List of words for visualization
words_to_visualize = ["natural", "language", "processing", "GloVe", "powerful", "word", "em

# Get word embeddings for the selected words
word_vectors = [nlp(word).vector for word in words_to_visualize]

# Convert word_vectors to a NumPy array
word_vectors_array = np.array(word_vectors)

# Ensure perplexity is less than the number of samples
perplexity = min(30, len(word_vectors_array) - 1)

# Apply t-SNE to reduce dimensionality to 2D
tsne = TSNE(n_components=2, perplexity=perplexity, random_state=42)
word_vectors_2d = tsne.fit_transform(word_vectors_array)

# Plot the words in 2D space
plt.figure(figsize=(8, 8))
plt.scatter(word_vectors_2d[:, 0], word_vectors_2d[:, 1])

# Annotate each point with the corresponding word
for i, word in enumerate(words_to_visualize):
    plt.annotate(word, (word_vectors_2d[i, 0], word_vectors_2d[i, 1]))

plt.title("t-SNE Visualization of Word Embeddings")
plt.show()
```


t-SNE Visualization of Word Embeddings
powerful
GloVe
processing
language
embedding
word
technique
natural
60
40
20
0
−20
−40
−60
−40
−20
0
20
40

FAST TEXT

In [21]:
```
from nltk.tokenize import word_tokenize
from gensim.models import FastText

# Dummy sentences
sentences = [
    "This is a sample sentence for FastText.",
    "FastText is a powerful word embedding model.",
    "We can train FastText on custom sentences too.",
    "Creating a simple example for demonstration.",
    "Let's check the similarity of some words.",
]

# Tokenize the sentences
tokenized_sentences = [word_tokenize(sentence.lower()) for sentence in sentences]

# Train FastText model
model = FastText(
    sentences=tokenized_sentences,
    vector_size=128,
    window=5,
    min_count=1,  # Adjust min_count as needed
    workers=4,
    epochs=10,
    seed=42,
    sg=1,
)

# Save the trained model
model.save('dummy_fasttext_model')

# Access the word vectors
ftext = model.wv

# Check the number of unique words in the model
print("Number of unique words:", len(ftext.index_to_key))

# Check the vector size
print("Vector size:", ftext.vector_size)

# Get the vector for a specific word
word_vector = ftext["fasttext"]
print("Vector for 'fasttext':", word_vector)

# Find similar words
similar_words_mantap = ftext.similar_by_word("mantap", topn=5)
print("Similar words to 'mantap':", similar_words_mantap)

similar_words_bagus = ftext.similar_by_word("bagus", topn=5)
print("Similar words to 'bagus':", similar_words_bagus)
```

```
Number of unique words: 31
Vector size: 128
Vector for 'fasttext': [-4.98031732e-04  1.72681102e-04 -8.47386254e-05  2.27859756e-03
  8.92896263e-04  3.49581242e-04 -8.55057617e-04 -3.98599310e-04
  9.64064617e-04 -6.45174470e-04  2.88390846e-04  5.65262453e-05
 -4.23705875e-04 -6.82594546e-04  8.28830409e-04  1.00176083e-03
 -2.00148323e-03  1.73650682e-03 -1.29093463e-03  3.55957483e-04
  4.24317222e-05 -8.21459282e-04 -2.77078041e-04 -6.49649184e-04
  8.73517536e-04 -3.08861985e-04 -7.99137109e-04 -1.82663347e-03
  1.87556288e-04  1.25339910e-04 -2.63656111e-04 -8.55624385e-04
  5.55895036e-04  7.09098938e-04  7.98384601e-04 -1.77128322e-03
  6.98002870e-04 -5.98518935e-04  1.86768814e-03  6.84162660e-04
  4.83201758e-04  1.39986238e-04 -6.21701882e-04 -5.26229327e-04
 -1.06013427e-03  4.31456428e-04 -2.71964062e-04  1.07383115e-04
  6.34527169e-05  1.15913659e-04  3.46912799e-04 -1.37849152e-03
 -5.68136165e-04 -1.15422753e-03  5.61428722e-04  1.03809393e-03
 -7.31351494e-04 -2.82177760e-04 -1.61603617e-04 -1.71460980e-03
  2.27081095e-04  1.22794649e-03 -4.77720925e-04  1.09660043e-03
 -1.10715651e-03 -6.86682761e-04  3.63592221e-06  1.42164971e-03
 -8.17802647e-05 -9.90196713e-05 -8.21492904e-06 -1.56539353e-03
 -2.33937244e-04 -2.44581490e-04  1.13282258e-04 -1.19333400e-03
  1.54490208e-05  2.90335825e-04 -1.68535579e-03 -2.28905745e-04
 -1.09496767e-04 -2.17412511e-04  2.84353446e-04 -1.82656950e-04
  9.56507050e-04  5.58205415e-04  3.45374050e-04 -9.04890825e-04
  2.67956173e-04  2.33425890e-04 -7.72091735e-04 -1.10911625e-03
 -9.95779759e-04 -6.88049186e-04 -7.29771346e-05 -2.17395660e-04
 -3.56130797e-04  2.34601932e-04 -4.45069891e-04 -1.79911859e-03
  3.20740714e-04  1.12061936e-03 -1.71231237e-04 -6.26352092e-04
  6.84442493e-05 -1.43895345e-03 -1.28856732e-03 -1.39677944e-03
  1.31438964e-03 -1.68813963e-03  9.59930883e-04  1.13626511e-03
 -2.43216811e-04  5.61613997e-04 -2.47450283e-04 -3.73065908e-04
 -2.10669474e-04  7.40772055e-04 -1.05676451e-03  1.26514002e-03
 -8.93100398e-04  4.29932610e-04  1.13889975e-04  1.58490706e-03
 -9.06038913e-04  8.66347633e-04 -4.16930037e-04 -5.06301702e-04]
Similar words to 'mantap': [('on', 0.15716801583766937), ('fasttext', 0.1413688659667968
8), ('let', 0.13120952248573303), ('of', 0.1057160273194313), ('this', 0.101848550140857
7)]
Similar words to 'bagus': [('for', 0.1979062855243683), ('too', 0.09124556928873062), ('em
bedding', 0.08523477613925934), ('custom', 0.06899155676364899), ('powerful', 0.0575192682
4450493)]
```

LAB-4

BUILDING LANGUAGE MODELS

N gram model

In [19]:
```
def readData():
    data = ['This is a  dog','This is a cat','I love my cat','This is my name ']
    dat=[]
    for i in range(len(data)):
        for word in data[i].split():
            dat.append(word)
    print(dat)
    return dat

def createBigram(data):
   listOfBigrams = []
   bigramCounts = {}
   unigramCounts = {}
   for i in range(len(data)-1):
      if i < len(data) - 1 and data[i+1].islower():

         listOfBigrams.append((data[i], data[i + 1]))

         if (data[i], data[i+1]) in bigramCounts:
            bigramCounts[(data[i], data[i + 1])] += 1
         else:
            bigramCounts[(data[i], data[i + 1])] = 1

      if data[i] in unigramCounts:
         unigramCounts[data[i]] += 1
      else:
         unigramCounts[data[i]] = 1
   return listOfBigrams, unigramCounts, bigramCounts


def calcBigramProb(listOfBigrams, unigramCounts, bigramCounts):
    listOfProb = {}
    for bigram in listOfBigrams:
        word1 = bigram[0]
        word2 = bigram[1]
        listOfProb[bigram] = (bigramCounts.get(bigram))/(unigramCounts.get(word1))
    return listOfProb


if __name__ == '__main__':
    data = readData()
    listOfBigrams, unigramCounts, bigramCounts = createBigram(data)

    print("\n All the possible Bigrams are ")
    print(listOfBigrams)

    print("\n Bigrams along with their frequency ")
    print(bigramCounts)

    print("\n Unigrams along with their frequency ")
    print(unigramCounts)

    bigramProb = calcBigramProb(listOfBigrams, unigramCounts, bigramCounts)

    print("\n Bigrams along with their probability ")
    print(bigramProb)
    inputList="This is my cat"
    splt=inputList.split()
    outputProb1 = 1
    bilist=[]
    bigrm=[]

    for i in range(len(splt) - 1):
```

```
        if i < len(splt) - 1:

            bilist.append((splt[i], splt[i + 1]))

    print("\n The bigrams in given sentence are ")
    print(bilist)
    for i in range(len(bilist)):
        if bilist[i] in bigramProb:

            outputProb1 *= bigramProb[bilist[i]]
        else:

            outputProb1 *= 0
```

```
['This', 'is', 'a', 'dog', 'This', 'is', 'a', 'cat', 'I', 'love', 'my', 'cat', 'This', 'i
s', 'my', 'name']

 All the possible Bigrams are
[('This', 'is'), ('is', 'a'), ('a', 'dog'), ('This', 'is'), ('is', 'a'), ('a', 'cat'),
('I', 'love'), ('love', 'my'), ('my', 'cat'), ('This', 'is'), ('is', 'my'), ('my', 'nam
e')]

 Bigrams along with their frequency
{('This', 'is'): 3, ('is', 'a'): 2, ('a', 'dog'): 1, ('a', 'cat'): 1, ('I', 'love'): 1,
('love', 'my'): 1, ('my', 'cat'): 1, ('is', 'my'): 1, ('my', 'name'): 1}

 Unigrams along with their frequency
{'This': 3, 'is': 3, 'a': 2, 'dog': 1, 'cat': 2, 'I': 1, 'love': 1, 'my': 2}

 Bigrams along with their probability
{('This', 'is'): 1.0, ('is', 'a'): 0.6666666666666666, ('a', 'dog'): 0.5, ('a', 'cat'): 0.
5, ('I', 'love'): 1.0, ('love', 'my'): 1.0, ('my', 'cat'): 0.5, ('is', 'my'): 0.3333333333
333333, ('my', 'name'): 0.5}

 The bigrams in given sentence are
[('This', 'is'), ('is', 'my'), ('my', 'cat')]
```

In [1]:

```
from nltk.corpus import reuters
from nltk import bigrams, trigrams
from collections import Counter, defaultdict

# Create a placeholder for model
model = defaultdict(lambda: defaultdict(lambda: 0))

# Count frequency of co-occurance
for sentence in reuters.sents():
    for w1, w2, w3 in trigrams(sentence, pad_right=True, pad_left=True):
        model[(w1, w2)][w3] += 1

# Let's transform the counts to probabilities
for w1_w2 in model:
    total_count = float(sum(model[w1_w2].values()))
    for w3 in model[w1_w2]:
        model[w1_w2][w3] /= total_count
```

In [2]:
```
dict(model["today","the"])
```

Out[2]:
```
{'public': 0.05555555555555555,
 'European': 0.05555555555555555,
 'Bank': 0.05555555555555555,
 'price': 0.1111111111111111,
 'emirate': 0.05555555555555555,
 'overseas': 0.05555555555555555,
 'newspaper': 0.05555555555555555,
 'company': 0.16666666666666666,
 'Turkish': 0.05555555555555555,
 'increase': 0.05555555555555555,
 'options': 0.05555555555555555,
 'Higher': 0.05555555555555555,
 'pound': 0.05555555555555555,
 'Italian': 0.05555555555555555,
 'time': 0.05555555555555555}
```

In [3]:
```
import random

# starting words
text = ["today", "the"]
sentence_finished = False

while not sentence_finished:
  # select a random probability threshold
  r = random.random()
  accumulator = .0

  for word in model[tuple(text[-2:])].keys():
      accumulator += model[tuple(text[-2:])][word]
      # select words that are above the probability threshold
      if accumulator >= r:
          text.append(word)
          break

  if text[-2:] == [None, None]:
      sentence_finished = True

print (' '.join([t for t in text if t]))
```

```
today the emirate , largest producer , was previously split on whether the tariffs were on
ly limited sales were expected to fall to 17 . 9 pct , according to private banks .
```

In [16]:
```
import numpy as np
import pandas as pd
from keras.utils import to_categorical
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import LSTM, Dense, GRU, Embedding
from keras.callbacks import EarlyStopping, ModelCheckpoint
```

In [17]:
```
data_text = """The unanimous Declaration of the thirteen united States of America, When in t
We hold these truths to be self-evident, that all men are created equal, that they are endou
He has refused his Assent to Laws, the most wholesome and necessary for the public good.
He has forbidden his Governors to pass Laws of immediate and pressing importance, unless sus
He has refused to pass other Laws for the accommodation of large districts of people, unless
He has called together legislative bodies at places unusual, uncomfortable, and distant fron
He has dissolved Representative Houses repeatedly, for opposing with manly firmness his inva
He has refused for a long time, after such dissolutions, to cause others to be elected; whe
He has endeavoured to prevent the population of these States; for that purpose obstructing t
He has obstructed the Administration of Justice, by refusing his Assent to Laws for establis
He has made Judges dependent on his Will alone, for the tenure of their offices, and the amo
He has erected a multitude of New Offices, and sent hither swarms of Officers to harrass our
He has kept among us, in times of peace, Standing Armies without the Consent of our legislat
He has affected to render the Military independent of and superior to the Civil power.
He has combined with others to subject us to a jurisdiction foreign to our constitution, and
For Quartering large bodies of armed troops among us:
For protecting them, by a mock Trial, from punishment for any Murders which they should comm
For cutting off our Trade with all parts of the world:
For imposing Taxes on us without our Consent:
For depriving us in many cases, of the benefits of Trial by Jury:
For transporting us beyond Seas to be tried for pretended offences
For abolishing the free System of English Laws in a neighbouring Province, establishing thei
For taking away our Charters, abolishing our most valuable Laws, and altering fundamentally
For suspending our own Legislatures, and declaring themselves invested with power to legisla
He has abdicated Government here, by declaring us out of his Protection and waging War agair
He has plundered our seas, ravaged our Coasts, burnt our towns, and destroyed the lives of o
He is at this time transporting large Armies of foreign Mercenaries to compleat the works of
He has constrained our fellow Citizens taken Captive on the high Seas to bear Arms against t
He has excited domestic insurrections amongst us, and has endeavoured to bring on the inhabi
In every stage of these Oppressions We have Petitioned for Redress in the most humble terms
Nor have We been wanting in attentions to our Brittish brethren. We have warned them from ti
We, therefore, the Representatives of the united States of America, in General Congress, Ass
```

In [18]:
```
import re

def text_cleaner(text):
    # lower case text
    newString = text.lower()
    newString = re.sub(r"'s\b","",newString)
    # remove punctuations
    newString = re.sub("[^a-zA-Z]", " ", newString)
    long_words=[]
    # remove short word
    for i in newString.split():
        if len(i)>=3:
            long_words.append(i)
    return (" ".join(long_words)).strip()

# preprocess the text
data_new = text_cleaner(data_text)
```

In [19]:
```
def create_seq(text):
    length = 30
    sequences = list()
    for i in range(length, len(text)):
        # select sequence of tokens
        seq = text[i-length:i+1]
        # store
        sequences.append(seq)
    print('Total Sequences: %d' % len(sequences))
    return sequences

# create sequences
sequences = create_seq(data_new)
```

```
Total Sequences: 7052
```

In [7]:
```
chars = sorted(list(set(data_new)))
mapping = dict((c, i) for i, c in enumerate(chars))

def encode_seq(seq):
    sequences = list()
    for line in seq:
        # integer encode line
        encoded_seq = [mapping[char] for char in line]
        # store
        sequences.append(encoded_seq)
    return sequences

# encode the sequences
sequences = encode_seq(sequences)
```

In [27]:
```
def encode_seq(seq, mapping):
    encoded_seqs = []
    for line in seq:
        # integer encode line
        encoded_seq = [mapping[char] for char in line]
        # store
        encoded_seqs.append(encoded_seq)
    return np.array(encoded_seqs)

# encode the sequences
sequences = encode_seq(sequences, mapping)
```

In [20]:
```
sequences
```

```
 'en the course human events beco',
 'n the course human events becom',
 ' the course human events become',
 'the course human events becomes',
 'he course human events becomes ',
 'e course human events becomes n',
 ' course human events becomes ne',
 'course human events becomes nec',
 'ourse human events becomes nece',
 'urse human events becomes neces',
 'rse human events becomes necess',
 'se human events becomes necessa',
 'e human events becomes necessar',
 ' human events becomes necessary',
 'human events becomes necessary ',
 'uman events becomes necessary f',
 'man events becomes necessary fo',
 'an events becomes necessary for',
 'n events becomes necessary for ',
 ' events becomes necessary for o',
```

In [28]:
```
# create X and y
X, y = sequences[:, :-1], sequences[:, -1]
# one hot encode y
y = to_categorical(y, num_classes=vocab)
```

In [29]:
```
from sklearn.model_selection import train_test_split

# vocabulary size
vocab = len(mapping)
sequences = np.array(sequences)
# create X and y

# create train and validation sets
X_tr, X_val, y_tr, y_val = train_test_split(X, y, test_size=0.1, random_state=42)

print('Train shape:', X_tr.shape, 'Val shape:', X_val.shape)
```

```
Train shape: (6346, 30) Val shape: (706, 30)
```

In [30]:
```
from sklearn.model_selection import train_test_split
from keras.utils import to_categorical
import numpy as np

# Assuming you have a 'mapping' variable defined elsewhere
# vocab = len(mapping)

# Assuming you have a 'sequences' variable defined elsewhere
# sequences = np.array(sequences)

# Reshape sequences to make it 2-dimensional
sequences = sequences.reshape((len(sequences), sequences.shape[1]))

# create X and y
X, y = sequences[:, :-1], sequences[:, -1]

# one hot encode y
y = to_categorical(y, num_classes=vocab)

# create train and validation sets
X_tr, X_val, y_tr, y_val = train_test_split(X, y, test_size=0.1, random_state=42)

print('Train shape:', X_tr.shape, 'Val shape:', X_val.shape)
```

```
Train shape: (6346, 30) Val shape: (706, 30)
```

In [31]:
```
model = Sequential()
model.add(Embedding(vocab, 50, input_length=30, trainable=True))
model.add(GRU(150, recurrent_dropout=0.1, dropout=0.1))
model.add(Dense(vocab, activation='softmax'))
print(model.summary())

# compile the model
model.compile(loss='categorical_crossentropy', metrics=['acc'], optimizer='adam')
# fit the model
model.fit(X_tr, y_tr, epochs=100, verbose=2, validation_data=(X_val, y_val))
```

```
199/199 - 6s - loss: 0.3939 - acc: 0.8727 - val_loss: 2.6548 - val_acc: 0.4844 - 6s/epo
ch - 30ms/step
Epoch 61/100
199/199 - 6s - loss: 0.3865 - acc: 0.8747 - val_loss: 2.7005 - val_acc: 0.4830 - 6s/epo
ch - 28ms/step
Epoch 62/100
199/199 - 6s - loss: 0.3957 - acc: 0.8713 - val_loss: 2.6988 - val_acc: 0.4745 - 6s/epo
ch - 28ms/step
Epoch 63/100
199/199 - 6s - loss: 0.3826 - acc: 0.8743 - val_loss: 2.7569 - val_acc: 0.4830 - 6s/epo
ch - 32ms/step
Epoch 64/100
199/199 - 7s - loss: 0.3782 - acc: 0.8760 - val_loss: 2.8102 - val_acc: 0.4802 - 7s/epo
ch - 34ms/step
Epoch 65/100
199/199 - 7s - loss: 0.3708 - acc: 0.8763 - val_loss: 2.8133 - val_acc: 0.4802 - 7s/epo
ch - 33ms/step
Epoch 66/100
199/199 - 6s - loss: 0.3690 - acc: 0.8776 - val_loss: 2.8537 - val_acc: 0.4788 - 6s/epo
ch - 29ms/step
```

In [32]:
```
def generate_seq(model, mapping, seq_length, seed_text, n_chars):
    in_text = seed_text
    # generate a fixed number of characters
    for _ in range(n_chars):
        # encode the characters as integers
        encoded = [mapping[char] for char in in_text]
        # truncate sequences to a fixed length
        encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
        # predict character
        yhat = model.predict_classes(encoded, verbose=0)
        # reverse map integer to character
        out_char = ''
        for char, index in mapping.items():
            if index == yhat:
                out_char = char
                break
        # append to input
        in_text += char
    return in_text
```

In [39]:
```
import numpy as np

def generate_seq(model, mapping, seq_length, seed_text, n_chars):
    in_text = seed_text
    # generate a fixed number of characters
    for _ in range(n_chars):
        # encode the characters as integers
        encoded = [mapping[char] for char in in_text]
        # truncate sequences to a fixed length
        encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
        # predict character
        yhat = model.predict(encoded, verbose=0)
        # get the index of the character with the highest probability
        yhat = np.argmax(yhat)
        # reverse map integer to character
        out_char = ''
        for char, index in mapping.items():
            if index == yhat:
                out_char = char
                break
        # append to input
        in_text += out_char
    return in_text
```

In [48]:
```
inp='Large armies of'
print(len(inp))
print(generate_seq(model,mapping,50,inp.lower(),15))
```

```
15
large armies offect their safe
```

INFERENCE:

In this NLP lab, we covered several techniques for word representation, starting with One-Hot Encoding. We transformed a given text into lowercase, tokenized it, obtained unique words, assigned integer positions to words, and created a one-hot matrix. Next, we explored Bag of Words using CountVectorizer, applying it to example sentences. We proceeded to Tf-Idf (term frequency-inverse document frequency), constructing a small corpus and computing TF and IDF values. We also delved into Word2Vec, leveraging the gensim library to download and use a pre-trained model, demonstrating word embeddings and similarity calculations. Additionally, we explored GloVe, utilizing spaCy to load a pre-trained model and showcasing word embeddings for sample sentences. Finally, we implemented

text embedding using GloVe for custom sentences and visualized word embeddings in 2D space using t-SNE. These techniques enhance our understanding of how to represent words in a format suitable for natural language processing task.In addition to this I have implemented an n-gram language model specifically designed for predictive next-word tasks. This involved considering sequences of n words within the text to discern patterns and relationships, enabling the model to predict the next word in a given context

In [ ]:

LAB - 5

```
import nltk
from nltk.tokenize import word_tokenize

# nltk.download('punkt')  # Ensure 'punkt' tokenizer is downloaded

text = word_tokenize("And now for something completely different")
tags = nltk.pos_tag(text)
print(tags)
```

```
[('And', 'CC'), ('now', 'RB'), ('for', 'IN'), ('something', 'NN'), ('completely', 'RB'), ('different', 'JJ')]
```

```
text = word_tokenize("They refuse to permit us to obtain the refuse permit")
nltk.pos_tag(text)
```

```
[('They', 'PRP'),
 ('refuse', 'VBP'),
 ('to', 'TO'),
 ('permit', 'VB'),
 ('us', 'PRP'),
 ('to', 'TO'),
 ('obtain', 'VB'),
 ('the', 'DT'),
 ('refuse', 'NN'),
 ('permit', 'NN')]
```

```
# import nltk
# nltk.download('brown')

text = nltk.Text(word.lower() for word in nltk.corpus.brown.words())
text.similar('woman')
text.similar('bought')
text.similar('over')
text.similar('the')
```

```
man time day year car moment world house family child country boy
state job place way war girl work word
made said done put had seen found given left heard was been brought
set got that took in told felt
in on to of and for with from at by that into as up out down through
is all about
a his this their its her an that our any all one these my in your no
some other and
```

2 Tagged Corpora

```
tagged_token = nltk.tag.str2tuple('fly/NN')
print(tagged_token)
print(tagged_token[0])
print(tagged_token[1])
```

```
('fly', 'NN')
fly
NN
```

```
import nltk

sent = '''
The/AT grand/JJ jury/NN commented/VBD on/IN a/AT number/NN of/IN
other/AP topics/NNS ,/, AMONG/IN them/PPO the/AT Atlanta/NP and/CC
Fulton/NP-tl County/NN-tl purchasing/VBG departments/NNS which/WDT it/PPS
said/VBD ``/`` ARE/BER well/QL operated/VBN and/CC follow/VB generally/RB
accepted/VBN practices/NNS which/WDT inure/VB to/IN the/AT best/JJT
interest/NN of/IN both/ABX governments/NNS ''/'' ./.
'''

tagged_tokens = [nltk.tag.str2tuple(t) for t in sent.split()]
print(tagged_tokens)
```

```
[('The', 'AT'), ('grand', 'JJ'), ('jury', 'NN'), ('commented', 'VBD'), ('on', 'IN'), ('a', 'AT'), ('number', 'NN'), ('of', 'IN'), ('othe
```

```
import nltk

# Using the default Brown corpus tagset
tagged_words_default = nltk.corpus.brown.tagged_words()
print(tagged_words_default)

# Using the Universal Tagset for the Brown corpus
tagged_words_universal = nltk.corpus.brown.tagged_words(tagset='universal')
print(tagged_words_universal)
```

```
[('The', 'AT'), ('Fulton', 'NP-TL'), ...]
[('The', 'DET'), ('Fulton', 'NOUN'), ...]
```

```
import nltk

# Access tagged words from the NPS Chat corpus
nps_chat_tagged_words = nltk.corpus.nps_chat.tagged_words()
print(nps_chat_tagged_words)

# Access tagged words from the CoNLL 2000 corpus
conll2000_tagged_words = nltk.corpus.conll2000.tagged_words()
print(conll2000_tagged_words)

# Access tagged words from the Treebank corpus
treebank_tagged_words = nltk.corpus.treebank.tagged_words()
print(treebank_tagged_words)
```

```
[('now', 'RB'), ('im', 'PRP'), ('left', 'VBD'), ...]
[('Confidence', 'NN'), ('in', 'IN'), ('the', 'DT'), ...]
[('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ...]
```

```
import nltk

# Access tagged words from the Brown corpus with the Universal Tagset
brown_tagged_words_universal = nltk.corpus.brown.tagged_words(tagset='universal')
print(brown_tagged_words_universal)

# Access tagged words from the Treebank corpus with the Universal Tagset
treebank_tagged_words_universal = nltk.corpus.treebank.tagged_words(tagset='universal')
print(treebank_tagged_words_universal)
```

```
[('The', 'DET'), ('Fulton', 'NOUN'), ...]
[('Pierre', 'NOUN'), ('Vinken', 'NOUN'), (',', '.'), ...]
```

```
import nltk

# Access tagged words from the Sinica Treebank corpus
sinica_treebank_tagged_words = nltk.corpus.sinica_treebank.tagged_words()
print(sinica_treebank_tagged_words)

# Access tagged words from the Indian corpus
indian_tagged_words = nltk.corpus.indian.tagged_words()
print(indian_tagged_words)

# Access tagged words from the Mac-Morpho corpus
mac_morpho_tagged_words = nltk.corpus.mac_morpho.tagged_words()
print(mac_morpho_tagged_words)

# Access tagged words from the CoNLL 2002 corpus
conll2002_tagged_words = nltk.corpus.conll2002.tagged_words()
print(conll2002_tagged_words)

# Access tagged words from the CESS-CAT corpus
cess_cat_tagged_words = nltk.corpus.cess_cat.tagged_words()
print(cess_cat_tagged_words)
```

```
[('一', 'Neu'), ('友情', 'Nad'), ('嘉珍', 'Nba'), ...]
[('মহিষের', 'NN'), ('সন্তান', 'NN'), (':', 'SYM'), ...]
```

```
[('Jersei', 'N'), ('atinge', 'V'), ('média', 'N'), ...]
[('Sao', 'NC'), ('Paulo', 'VMI'), ('(', 'Fpa'), ...]
[('El', 'da0ms0'), ('Tribunal_Suprem', 'np0000o'), ...]
```

```
from nltk.corpus import brown
brown_news_tagged = brown.tagged_words(categories='news', tagset='universal')
tag_fd = nltk.FreqDist(tag for (word, tag) in brown_news_tagged)
tag_fd.most_common()
[('NOUN', 30640), ('VERB', 14399), ('ADP', 12355), ('.', 11928), ('DET', 11389),
 ('ADJ', 6706), ('ADV', 3349), ('CONJ', 2717), ('PRON', 2535), ('PRT', 2264),
 ('NUM', 2166), ('X', 106)]
```

```
[('NOUN', 30640),
 ('VERB', 14399),
 ('ADP', 12355),
 ('.', 11928),
 ('DET', 11389),
 ('ADJ', 6706),
 ('ADV', 3349),
 ('CONJ', 2717),
 ('PRON', 2535),
 ('PRT', 2264),
 ('NUM', 2166),
 ('X', 106)]
```

```
import nltk

# Load the Brown corpus and filter it to use only the 'news' category
brown_news_tagged = nltk.corpus.brown.tagged_words(categories='news')

# Generate bigrams from the tagged words in the 'news' category
word_tag_pairs = nltk.bigrams(brown_news_tagged)

# Filter for cases where the second item in the bigram is tagged as a noun ('NOUN')
noun_preceders = [a[1] for (a, b) in word_tag_pairs if b[1] == 'NOUN']

# Create a frequency distribution of the tags that precede nouns
fdist = nltk.FreqDist(noun_preceders)

# Extract the most common tags that precede nouns
most_common_tags = [tag for (tag, _) in fdist.most_common()]
print(most_common_tags)
```

```
[]
```

```
import nltk

# Access tagged words from the Treebank corpus with the Universal Tagset
wsj = nltk.corpus.treebank.tagged_words(tagset='universal')

# Create a frequency distribution of word-tag pairs
word_tag_fd = nltk.FreqDist(wsj)

# Extract the most common words tagged as verbs
verbs = [wt[0] for (wt, _) in word_tag_fd.most_common() if wt[1] == 'VERB']
print(verbs)
```

```
['is', 'said', 'was', 'are', 'be', 'has', 'have', 'will', 'says', 'would', 'were', 'had', 'been', 'could', "'s", 'can', 'do', 'say', 'ma
```

```
import nltk

# Assuming 'wsj' contains the Wall Street Journal portion of the Treebank corpus
# If 'wsj' is not defined yet, load the Wall Street Journal portion of the Treebank corpus
wsj = nltk.corpus.treebank.tagged_words()

# Create a ConditionalFreqDist object
cfd1 = nltk.ConditionalFreqDist(wsj)

# Check the distribution of parts of speech for the word 'yield'
yield_pos_distribution = cfd1['yield'].most_common()
print("Distribution of 'yield':", yield_pos_distribution)

# Check the distribution of parts of speech for the word 'cut'
cut_pos_distribution = cfd1['cut'].most_common()
print("Distribution of 'cut':", cut_pos_distribution)
```

```
Distribution of 'yield': [('VB', 28), ('NN', 20)]
Distribution of 'cut': [('VB', 12), ('VBD', 10), ('VBN', 3), ('NN', 3)]
```

```
import nltk

# Assuming 'wsj' contains the Wall Street Journal portion of the Treebank corpus
# If 'wsj' is not defined yet, load the Wall Street Journal portion of the Treebank corpus
wsj = nltk.corpus.treebank.tagged_words()

# Create a ConditionalFreqDist object by swapping word and tag positions
cfd2 = nltk.ConditionalFreqDist((tag, word) for (word, tag) in wsj)

# List words tagged as 'VBN' (past participle)
vbn_words = list(cfd2['VBN'])
print(vbn_words[:20])  # Display the first 20 words for brevity
```

```
['been', 'expected', 'made', 'compared', 'based', 'used', 'priced', 'sold', 'named', 'designed', 'held', 'fined', 'taken', 'paid', 'trac
```

```
import nltk

# Assuming 'wsj' contains the Wall Street Journal portion of the Treebank corpus
# If 'wsj' is not defined yet, load the Wall Street Journal portion of the Treebank corpus
wsj = nltk.corpus.treebank.tagged_words()

# Create a ConditionalFreqDist object 'cfd1'
cfd1 = nltk.ConditionalFreqDist((word, tag) for (word, tag) in wsj)

# Find words that are tagged as both 'VBD' (past tense) and 'VBN' (past participle)
words_with_vbd_and_vbn = [w for w in cfd1.conditions() if 'VBD' in cfd1[w] and 'VBN' in cfd1[w]]
print(words_with_vbd_and_vbn[:10])  # Display the first 10 words for brevity

# Find context surrounding 'kicked' as 'VBD'
idx1 = wsj.index(('kicked', 'VBD'))
print(wsj[idx1-4:idx1+1])

# Find context surrounding 'kicked' as 'VBN'
idx2 = wsj.index(('kicked', 'VBN'))
print(wsj[idx2-4:idx2+1])
```

```
['named', 'used', 'caused', 'reported', 'said', 'stopped', 'heard', 'studied', 'led', 'replaced']
[('While', 'IN'), ('program', 'NN'), ('trades', 'NNS'), ('swiftly', 'RB'), ('kicked', 'VBD')]
[('head', 'NN'), ('of', 'IN'), ('state', 'NN'), ('has', 'VBZ'), ('kicked', 'VBN')]
```

```
import nltk

def findtags(tag_prefix, tagged_text):
    cfd = nltk.ConditionalFreqDist((tag, word) for (word, tag) in tagged_text
                                   if tag.startswith(tag_prefix))
    return {tag: cfd[tag].most_common(5) for tag in cfd.conditions()}

tagged_news = nltk.corpus.brown.tagged_words(categories='news')
tagdict = findtags('NN', tagged_news)

for tag in sorted(tagdict):
    print(tag, tagdict[tag])
```

```
 NN [('year', 137), ('time', 97), ('state', 88), ('week', 85), ('man', 72)]
 NN$ [("year's", 13), ("world's", 8), ("state's", 7), ("nation's", 6), ("city's", 6)]
 NN$-HL [("Golf's", 1), ("Navy's", 1)]
 NN$-TL [("President's", 11), ("Administration's", 3), ("Army's", 3), ("League's", 3), ("University's", 3)]
 NN-HL [('sp.', 2), ('problem', 2), ('Question', 2), ('cut', 2), ('party', 2)]
 NN-NC [('ova', 1), ('eva', 1), ('aya', 1)]
 NN-TL [('President', 88), ('House', 68), ('State', 59), ('University', 42), ('City', 41)]
 NN-TL-HL [('Fort', 2), ('Mayor', 1), ('Commissioner', 1), ('City', 1), ('Oak', 1)]
 NNS [('years', 101), ('members', 69), ('people', 52), ('sales', 51), ('men', 46)]
 NNS$ [("children's", 7), ("women's", 5), ("men's", 3), ("janitors'", 3), ("taxpayers'", 2)]
 NNS$-HL [("Dealers'", 1), ("Idols'", 1)]
 NNS$-TL [("Women's", 4), ("States'", 3), ("Giants'", 2), ("Princes'", 1), ("Bombers'", 1)]
 NNS-HL [('Wards', 1), ('deputies', 1), ('bonds', 1), ('aspects', 1), ('Decisions', 1)]
 NNS-TL [('States', 38), ('Nations', 11), ('Masters', 10), ('Communists', 9), ('Rules', 9)]
 NNS-TL-HL [('Nations', 1)]
```

```
import nltk

# Assuming 'brown_learned_text' contains the words from the 'learned' category of the Brown corpus
brown_learned_text = nltk.corpus.brown.words(categories='learned')

# Find words that occur after 'often' in the 'learned' category of the Brown corpus
words_after_often = sorted(set(b for (a, b) in nltk.bigrams(brown_learned_text) if a == 'often'))
print(words_after_often)
```

```
[',', '.', 'accomplished', 'analytically', 'appear', 'apt', 'associated', 'assuming', 'became', 'become', 'been', 'began', 'call', 'call
```

```
brown_lrnd_tagged = brown.tagged_words(categories='learned', tagset='universal')
tags = [b[1] for (a, b) in nltk.bigrams(brown_lrnd_tagged) if a[0] == 'often']
>>> fd = nltk.FreqDist(tags)
>>> fd.tabulate()
```

```
 VERB  ADV  ADP  ADJ    .  PRT
   37    8    7    6    4    2
```

```
from nltk.corpus import brown
import nltk

def process(sentence):
    for (w1, t1), (w2, t2), (w3, t3) in nltk.trigrams(sentence):
        if t1.startswith('V') and t2 == 'TO' and t3.startswith('V'):
            print(w1, w2, w3)

# Iterate through tagged sentences in the Brown corpus and apply the 'process' function
for tagged_sent in brown.tagged_sents():
    process(tagged_sent)
```

```
like to talk
ceased to grumble
tending to bid
start to pay
failed to measure
going to take
needed to push
trying to get
try to get
want to increase
wants to bring
threaten to strike
begun to grow
combine to serve
helping to strengthen
designed to promote
threatening to expand
seeks to get
begin to see
continue to expand
failing to render
decided to tackle
expects to sign
tends to become
came to understand
deserve to breathe
advised to seek
attempting to make
try to gun
began to fill
proposes to preserve
asked to approve
seeking to break
tends to spread
want to amend
rejected to seek
continued to speak
trying to make
expected to head
tempted to let
appear to cost
attempt to shore
seeking to achieve
expected to report
```

```
from nltk.corpus import brown
import nltk

# Extract tagged words from the 'news' category of the Brown corpus using the Universal Tagset
brown_news_tagged = brown.tagged_words(categories='news', tagset='universal')

# Create a ConditionalFreqDist object based on the tagged words
data = nltk.ConditionalFreqDist((word.lower(), tag) for (word, tag) in brown_news_tagged)

# Iterate through words in sorted order and print words with more than three different tags
for word in sorted(data.conditions()):
    if len(data[word]) > 3:
        tags = [tag for (tag, _) in data[word].most_common()]
        print(word, ' '.join(tags))
```

```
best ADJ ADV VERB NOUN
close ADV ADJ VERB NOUN
open ADJ VERB NOUN ADV
present ADJ ADV NOUN VERB
that ADP DET PRON ADV
```

```
pos = {}  # Creating an empty dictionary named 'pos'
pos['colorless'] = 'ADJ'  # Adding 'colorless' with the tag 'ADJ' to the dictionary
pos['ideas'] = 'N'  # Adding 'ideas' with the tag 'N' to the dictionary
pos['sleep'] = 'V'  # Adding 'sleep' with the tag 'V' to the dictionary
pos['furiously'] = 'ADV'  # Adding 'furiously' with the tag 'ADV' to the dictionary
pos
```

```
{'colorless': 'ADJ', 'ideas': 'N', 'sleep': 'V', 'furiously': 'ADV'}
```

```
pos['ideas']
```

```
'N'
```

```
pos['colorless']
```

```
'ADJ'
```

```
list(pos)  # [1]
sorted(pos)  # [2]
[w for w in pos if w.endswith('s')]  # [3]
```

```
['colorless', 'ideas']
```

```
for word in sorted(pos):
    print(word + ":", pos[word])
```

```
colorless: ADJ
furiously: ADV
ideas: N
sleep: V
```

```
# Creating a dictionary 'pos' with words as keys and their associated parts of speech as values
pos = {
    'colorless': 'ADJ',
    'ideas': 'N',
    'sleep': 'V',
    'furiously': 'ADV'
}

# Retrieving and displaying keys, values, and items from the 'pos' dictionary
print(list(pos.keys()))  # Retrieve keys
print(list(pos.values()))  # Retrieve values
print(list(pos.items()))  # Retrieve key-value pairs

# Iterating through sorted key-value pairs and printing them
for key, val in sorted(pos.items()):
    print(key + ":", val)
```

```
['colorless', 'ideas', 'sleep', 'furiously']
['ADJ', 'N', 'V', 'ADV']
[('colorless', 'ADJ'), ('ideas', 'N'), ('sleep', 'V'), ('furiously', 'ADV')]
colorless: ADJ
furiously: ADV
ideas: N
sleep: V
```

```
pos['sleep'] = 'V'
pos['sleep']
```

```
'V'
```

```
pos['sleep'] = 'N'
pos['sleep']
```

```
'N'
```

```
pos = {'colorless': 'ADJ', 'ideas': 'N', 'sleep': 'V', 'furiously': 'ADV'}
```

```
pos = dict(colorless='ADJ', ideas='N', sleep='V', furiously='ADV')
```

```
from collections import defaultdict

frequency = defaultdict(int)
frequency['colorless'] = 4
frequency['ideas']  # Accessing a key that doesn't exist defaults to 0
# Output: 0
```

```
0
```