211AI012

Akshara B

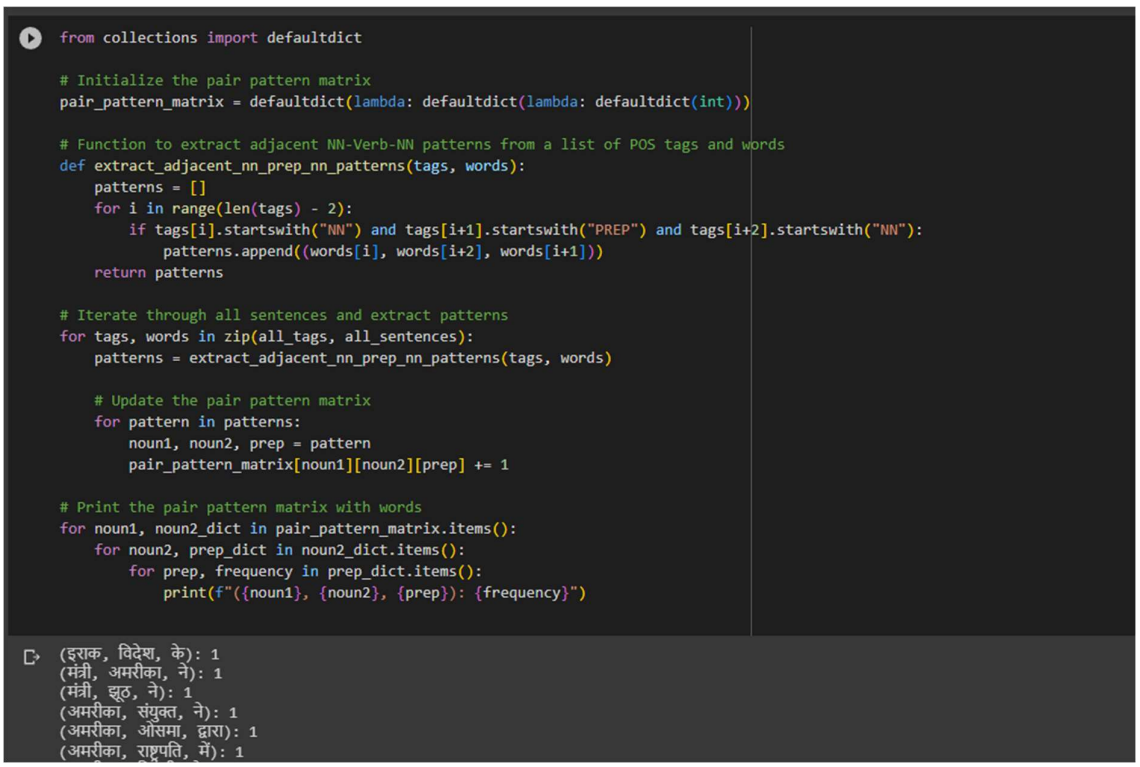
Assignment 2

1.

For the first part of the assignment to build a pair pattern matrix I tried implementing it with Hindi dataset with NN-PREP-NN pattern. Here I used the same dataset as Assignment 1

```
import re
import itertools

# Load the dataset
with open("/content/hindi.pos", "r", encoding="utf-8") as file:
    dataset_content = file.read()

# Extract sentences and their POS tags
sentences = re.findall(r'<Sentence id=\d+>.*?</Sentence>', dataset_content, re.DOTALL)

all_sentences = []
all_tags = []

for sentence in sentences:
    sentence_text = re.search(r'>(.*?)</Sentence>', sentence, re.DOTALL).group(1)
    tagged_words = sentence_text.strip().split()

    # Separate words and tags
    words = []
    tags = []
    for tagged_word in tagged_words:
        word, tag = tagged_word.rsplit("_", 1)
        words.append(word)
        tags.append(tag)

    all_sentences.append(words)
    all_tags.append(tags)

print(all_sentences)
print(tags)
```

First I downloaded the Hindi corpus and separated sentences and tags and printed them as shown.



```
from collections import defaultdict

# Initialize the pair pattern matrix
pair_pattern_matrix = defaultdict(lambda: defaultdict(lambda: defaultdict(int)))

# Function to extract adjacent NN-Verb-NN patterns from a list of POS tags and words
def extract_adjacent_nn_prep_nn_patterns(tags, words):
    patterns = []
    for i in range(len(tags) - 2):
        if tags[i].startswith("NN") and tags[i+1].startswith("PREP") and tags[i+2].startswith("NN"):
            patterns.append((words[i], words[i+2], words[i+1]))
    return patterns

# Iterate through all sentences and extract patterns
for tags, words in zip(all_tags, all_sentences):
    patterns = extract_adjacent_nn_prep_nn_patterns(tags, words)

    # Update the pair pattern matrix
    for pattern in patterns:
        noun1, noun2, prep = pattern
        pair_pattern_matrix[noun1][noun2][prep] += 1

# Print the pair pattern matrix with words
for noun1, noun2_dict in pair_pattern_matrix.items():
    for noun2, prep_dict in noun2_dict.items():
        for prep, frequency in prep_dict.items():
            print(f"({noun1}, {noun2}, {prep}): {frequency}")
```

```
(इराक, विदेश, के): 1
(मंत्री, अमरीका, ने): 1
(मंत्री, झूठ, ने): 1
(अमरीका, संयुक्त, ने): 1
(अमरीका, ओसमा, द्वारा): 1
(अमरीका, राष्ट्रपति, में): 1
```

Now we create pair patter matrix. The matrix is implemented as 3 dimensional dictionary (due to memory constraint and complexity for creating an actual matrix using lists)

First we extract Noun-Preposition-Noun triplets from all the available sentences. If I was doing this in English I would've used Noun-Verb-Noun triplet as suggested by sir (ex: carpenter cuts wood) but since this combination is not commonly occurred in Hindi language I used Noun-Preposition-Noun triplets (ex : सूत्रों के हवाले)

Now after extracting these triplet pairs we calculate document frequency of each triplet (how many documents does the pattern occur) and print it.

```
(शेरॉन, कोशिश, को): 1
(हिंसा, घटनाएं, की): 1
(हिंसा, स्तर, के): 1
(हिंसा, बढ़ोतरी, में): 1
(फिलिस्तीनियों, हमले, के): 1
(कार्यालय, कार्यभार, में): 1
(संसद, बहुमत, में): 1
(किनारों, इजराइलियों, पर): 1
(पार्टी, शेरोन, द्वारा): 1
(पार्टी, बैठक, के): 1
(दिल्ली, भूकंप, में): 1
(भूकंप, झटके, के): 1
(भूकंप, तीव्रता, की): 1
(भूकंप, वाला, में): 1
(उपमहाद्वीप, यात्रा, की): 2
(मार्च, न्यूयार्क, को): 2
(मार्च, बंग्लादेश, को): 1
(मार्च, भारत, तक): 1
(मार्च, सेंट, को): 1
(मार्च, दिल्ली, से): 1
(पाक, राष्ट्रपति, के): 1
(पाक, आवास, के): 1
(तनावों, मद्देनजर, के): 1
(एडिलेड, गुरुवार, में): 1
(ब्रैडमैन, पुत्र, के): 1
(ब्रैडमैन, श्रद्धांजलि, को): 1
(ब्रैडमैन, बल्लेबाजी, की): 1
```

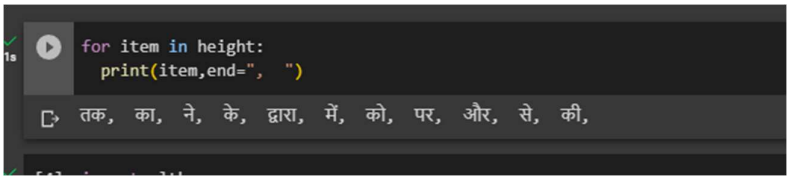
Since the sentences aren't too large or similar we get document frequency as 1, 2 or 3

This is just a model of how 3d pair pattern matching can be done.

```
# Initialize a set to store unique elements
rows = set()
columns=set()
height=set()

for noun1, noun2_dict in pair_pattern_matrix.items():
    for noun2, prep_dict in noun2_dict.items():
        for prep in prep_dict.keys():
            rows.add(noun1)
            columns.add(noun2)
            height.add(prep)



# Print the size of the matrix
print("Size of the matrix:",len(rows)," x",len(columns)," x ",len(height))
```

```
Size of the matrix: 355  x 382  x  11
```

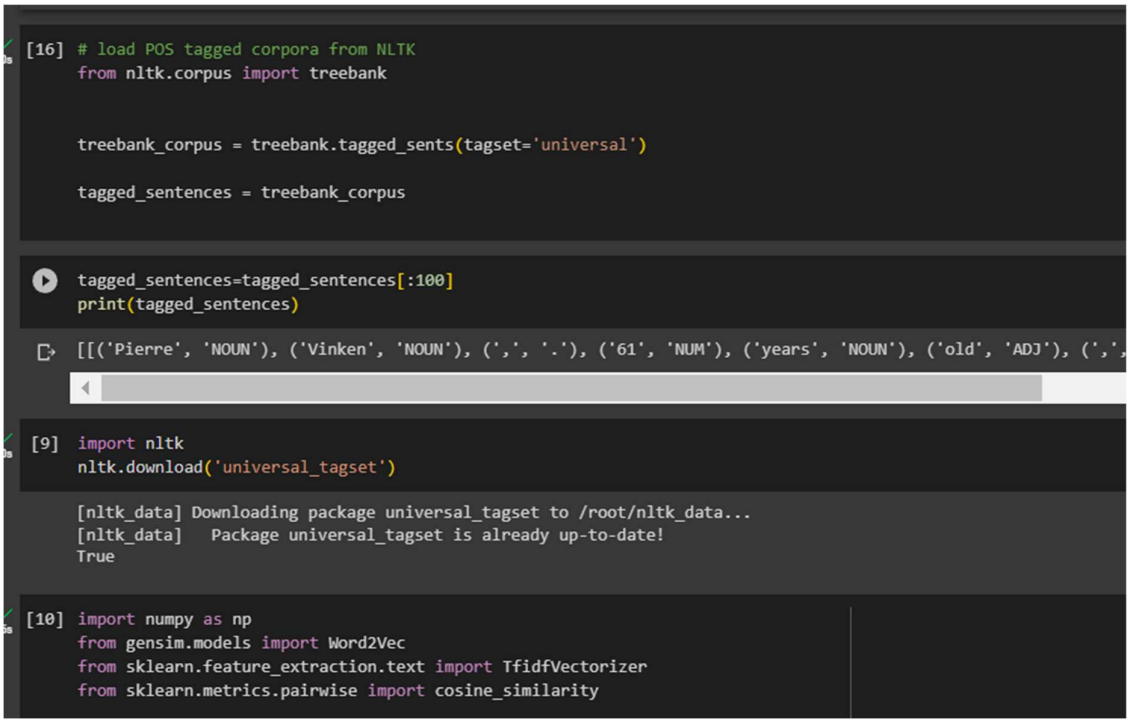
Here we can see that the dimensions of are matrix are 355 nouns x 382 nouns x 11 prepositions



It can also be inferred that these 11 prepositions usually occur in-between nouns.

2.

Now here to calculate cosine similarity I used English corpus from treebank. Since word2vec conversion is too complicated for Hindi language



```python
import numpy as np
from gensim.models import Word2Vec
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# Convert tagged sentences to plain text sentences
def tagged_to_text(tagged_sentence):
    return ' '.join([word for word, tag in tagged_sentence])

plain_text_sentences = [tagged_to_text(tagged_sentence) for tagged_sentence in tagged_sentences]

# Train a Word2Vec model on the plain text sentences
tokenized_sentences = [sentence.split() for sentence in plain_text_sentences]
word2vec_model = Word2Vec(tokenized_sentences, vector_size=100, window=5, min_count=1, sg=0)

# Convert plain text sentences to TF-IDF vectors
vectorizer = TfidfVectorizer()
tfidf_matrix = vectorizer.fit_transform(plain_text_sentences)

# Calculate Cosine Similarity
cosine_similarities = cosine_similarity(tfidf_matrix, tfidf_matrix)

# Define a function to calculate soft cosine similarity (same as before)
def soft_cosine_similarity(sentence1, sentence2):
    tokens1 = sentence1.split()
    tokens2 = sentence2.split()

    vector1 = np.mean([word2vec_model.wv[token] for token in tokens1 if token in word2vec_model.wv], axis=0)
    vector2 = np.mean([word2vec_model.wv[token] for token in tokens2 if token in word2vec_model.wv], axis=0)

    soft_cosine = cosine_similarity(vector1.reshape(1, -1), vector2.reshape(1, -1), dense_output=False)

    return soft_cosine[0, 0]

# Calculate Soft Cosine Similarity for all sentence pairs (same as before)
soft_cosine_similarities = np.zeros((len(plain_text_sentences), len(plain_text_sentences)))
for i in range(len(plain_text_sentences)):
    for j in range(len(plain_text_sentences)):
```

```python
plain_text_sentences = [tagged_to_text(tagged_sentence) for tagged_sentence in tagged_sentences]

# Train a Word2Vec model on the plain text sentences
tokenized_sentences = [sentence.split() for sentence in plain_text_sentences]
word2vec_model = Word2Vec(tokenized_sentences, vector_size=100, window=5, min_count=1, sg=0)

# Convert plain text sentences to TF-IDF vectors
vectorizer = TfidfVectorizer()
tfidf_matrix = vectorizer.fit_transform(plain_text_sentences)

# Calculate Cosine Similarity
cosine_similarities = cosine_similarity(tfidf_matrix, tfidf_matrix)

# Define a function to calculate soft cosine similarity (same as before)
def soft_cosine_similarity(sentence1, sentence2):
    tokens1 = sentence1.split()
    tokens2 = sentence2.split()

    vector1 = np.mean([word2vec_model.wv[token] for token in tokens1 if token in word2vec_model.wv], axis=0)
    vector2 = np.mean([word2vec_model.wv[token] for token in tokens2 if token in word2vec_model.wv], axis=0)

    soft_cosine = cosine_similarity(vector1.reshape(1, -1), vector2.reshape(1, -1), dense_output=False)

    return soft_cosine[0, 0]

# Calculate Soft Cosine Similarity for all sentence pairs (same as before)
soft_cosine_similarities = np.zeros((len(plain_text_sentences), len(plain_text_sentences)))
for i in range(len(plain_text_sentences)):
    for j in range(len(plain_text_sentences)):
        soft_cosine_similarities[i, j] = soft_cosine_similarity(plain_text_sentences[i], plain_text_sentences[j])

# Print the results
print("Cosine Similarity:")
print(cosine_similarities)
print("\nSoft Cosine Similarity:")
print(soft_cosine_similarities)
```

After downloading corpus and extracting POS tagged sentences, we do the following steps:

1.Converting Tagged Sentences to Plain Text Sentences:

we start by defining a function tagged_to_text(tagged_sentence) that takes a tagged sentence (a list of word-tag pairs) and converts it into a plain text sentence by extracting only the words. The words are joined into a single string using ' '.join().

2. Creating Plain Text Sentences:

The list plain_text_sentences is created by applying the tagged_to_text() function to each tagged sentence in tagged_sentences. This list now contains plain text versions of the sentences.

3. Training a Word2Vec Model:

Then we trains a Word2Vec model using the plain text sentences. The Word2Vec model is created from tokenized_sentences, where each sentence has been split into a list of words. The key parameters used for the model are:

vector_size: The dimensionality of the word vectors (in this case, 100).

window: The maximum distance between the current and predicted word within a sentence (in this case, 5).

min_count: Ignores all words with a total frequency lower than this (in this case, 1).

sg: Defines the training algorithm (0 for CBOW, 1 for Skip-gram).

4. Converting to TF-IDF Vectors:

Next we use scikit-learn's TfidfVectorizer to convert the plain_text_sentences into TF-IDF vectors. This vectorization process assigns numerical values to each word in the sentences based on their importance in the corpus.

5. Calculating Cosine Similarity:

The cosine_similarity() function from scikit-learn is used to calculate the cosine similarity between all pairs of sentences represented as TF-IDF vectors. This results in the cosine_similarities matrix, where each element [i, j] represents the cosine similarity between sentences i and j.

6. Defining a Function for Soft Cosine Similarity:

Next we define a function soft_cosine_similarity(sentence1, sentence2 to calculate the soft cosine similarity between two sentences. It does this by:

Splitting the sentences into tokens (words).

Calculating the word vectors for each token based on the Word2Vec model.

Computing the cosine similarity between the word vectors, which accounts for the semantic similarity between words.

The soft_cosine_similarities matrix is populated with these similarity values, where each element [i, j] represents the soft cosine similarity between sentences i and j.

```
    for j in range(len(plain_text_sentences)):
        soft_cosine_similarities[i, j] = soft_cosine_similarity(plain_text_sentences[i], plain_text_ser

# Print the results
print("Cosine Similarity:")
print(cosine_similarities)
print("\nSoft Cosine Similarity:")
print(soft_cosine_similarities)
```

```
Cosine Similarity:
[[1.         0.10975085 0.20688955 ... 0.0401786  0.05979196 0.00861912]
 [0.10975085 1.         0.08424405 ... 0.         0.07424789 0.01073415]
 [0.20688955 0.08424405 1.         ... 0.01064619 0.04202038 0.0157862 ]
 ...
 [0.0401786  0.         0.01064619 ... 1.         0.32942614 0.17782338]
 [0.05979196 0.07424789 0.04202038 ... 0.32942614 1.         0.15440939]
 [0.00861912 0.01073415 0.0157862  ... 0.17782338 0.15440939 1.        ]]

Soft Cosine Similarity:
[[1.         0.53598654 0.68954694 ... 0.64418817 0.64355147 0.58837479]
 [0.53598654 0.99999994 0.52711624 ... 0.4423753  0.62997693 0.44538668]
 [0.68954694 0.52711624 1.00000012 ... 0.74750006 0.74553049 0.64483285]
 ...
 [0.64418817 0.4423753  0.74750006 ... 0.99999994 0.75899851 0.71077746]
 [0.64355147 0.62997693 0.74553049 ... 0.75899851 1.         0.68074602]
 [0.58837479 0.44538668 0.64483285 ... 0.71077746 0.68074602 0.99999988]]
```

```
[14] import numpy as np
     import matplotlib.pyplot as plt
     from sklearn.cluster import KMeans
     from sklearn.manifold import TSNE
```

So finally we get cosine_similarities based on TF-IDF vectors and soft_cosine_similarities based on semantic similarity of words using Word2Vec. These matrices provide a measure of how similar each pair of sentences is in the corpus, considering both textual content and semantic meaning.

3.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.manifold import TSNE

dissimilarity_matrix = 1 - cosine_similarities

num_clusters = 3

# Perform K-Means clustering
kmeans = KMeans(n_clusters=num_clusters)
cluster_labels = kmeans.fit_predict(dissimilarity_matrix)

# Perform t-SNE dimensionality reduction
tsne = TSNE(n_components=2)  # You can also use n_components=3 for 3D visualization
tsne_embeddings = tsne.fit_transform(dissimilarity_matrix)

# Visualize the clustered sentences using a scatter plot
plt.figure(figsize=(10, 8))
for cluster_id in range(num_clusters):
    plt.scatter(
        tsne_embeddings[cluster_labels == cluster_id, 0],
        tsne_embeddings[cluster_labels == cluster_id, 1],
        label=f'Cluster {cluster_id + 1}',
    )

plt.title('t-SNE Visualization of Clusters')
plt.legend()
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `
  warnings.warn(
```

1. Dissimilarity Matrix:

We first Compute a dissimilarity matrix from cosine similarities by subtracting each similarity from 1.

2. Number of Clusters:

We Define the number of clusters (num_clusters) for K-Means clustering (I took number of clusters to be 3 c).

3. K-Means Clustering:

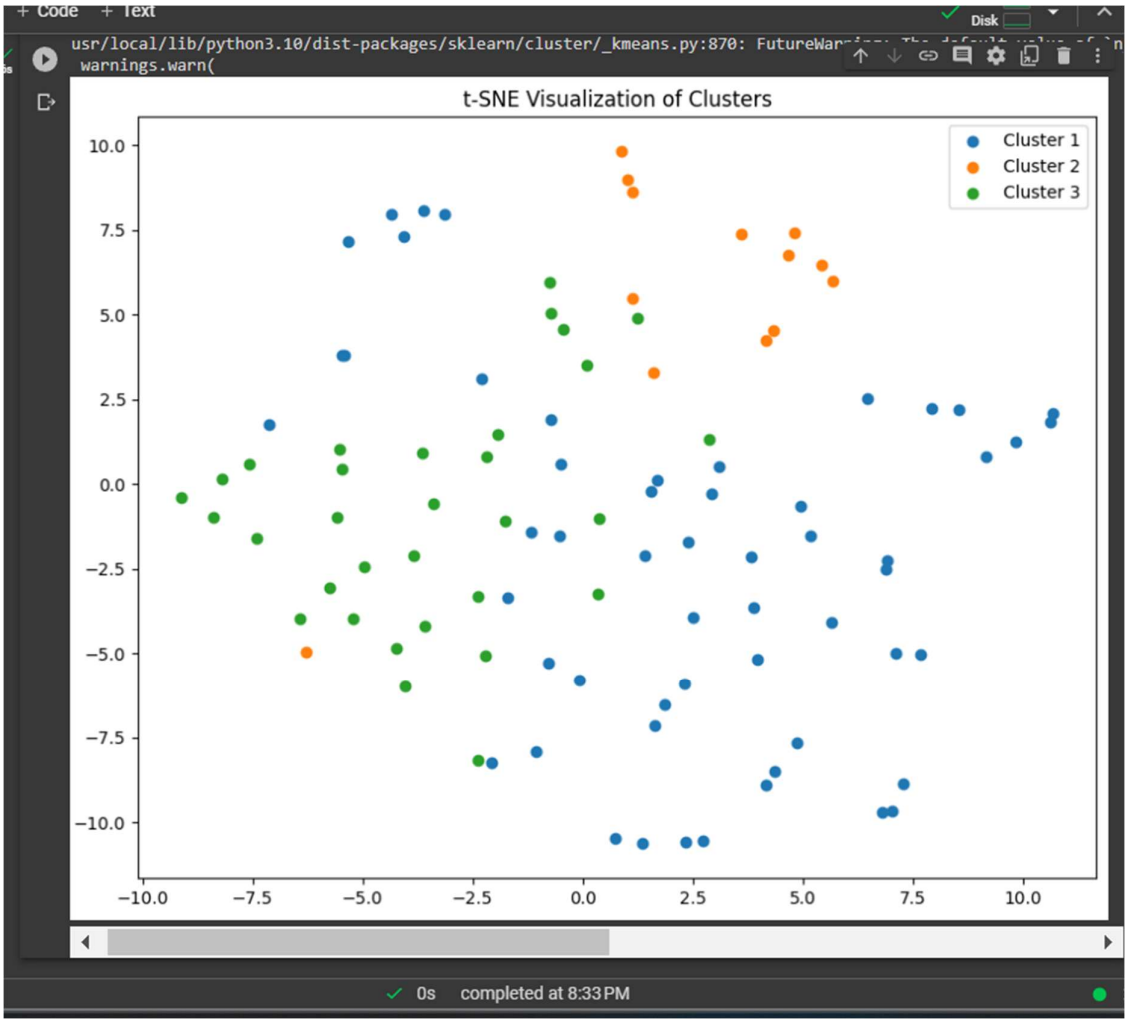
Then we apply K-Means clustering to the dissimilarity matrix, assigning sentences to clusters. And then we get cluster labels (cluster_labels) for each sentence.

4. t-SNE Dimensionality Reduction:

we use t-SNE to reduce high-dimensional dissimilarity matrix to 2D (or 3D) space. we store the reduced coordinates in tsne_embeddings.

5. Scatter Plot Visualization:

Finally we create a scatter plot to visualize sentences in the lower-dimensional space. Each cluster is plotted with a distinct color and labeled accordingly.



The result is a visual representation of how sentences are grouped into clusters in the lower-dimensional t-SNE space. Each point on the plot represents a sentence, and sentences with similar semantic and content characteristics tend to be closer to each other in this space, making it easier to identify patterns and similarities among sentences.