TC 5033

Word Embeddings

Activity 3a: Exploring Word Embeddings with GloVe and Numpy

- Objective:
  - To understand the concept of word embeddings and their significance in Natural Language Processing.
  - To learn how to manipulate and visualize high-dimensional data using dimensionality reduction techniques like PCA and t-SNE.
  - To gain hands-on experience in implementing word similarity and analogies using GloVe embeddings and Numpy.

- Instructions:
  - Download GloVe pre-trained vectors from the provided link in Canvas, the official public project: Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation https://nlp.stanford.edu/data/glove.6B.zip

  - Create a dictorionay of the embeddings so that you carry out fast look ups. Save that dictionary e.g. as a serialized file for faster loading in future uses.

  - PCA and t-SNE Visualization: After loading the GloVe embeddings, use Numpy and Sklearn to perform PCA and t-SNE to reduce the dimensionality of the embeddings and visualize them in a 2D or 3D space.

  - Word Similarity: Implement a function that takes a word as input and returns the 'n' most similar words based on their embeddings. You should use Numpy to implement this function, using libraries that already implement this function (e.g. Gensim) will result in zero points.

  - Word Analogies: Implement a function to solve analogies between words. For example, "man is to king as woman is to ____". You should use Numpy to implement this function, using libraries that already implement this function (e.g. Gensim) will result in zero points.

  - Submission: This activity is to be submitted in teams of 3 or 4. Only one person should submit the final work, with the full names of all team members included in a markdown cell at the beginning of the notebook.

- Evaluation Criteria:
  - Code Quality (40%): Your code should be well-organized, clearly commented, and easy to follow. Use also markdown cells for clarity.
  - Functionality (60%): All functions should work as intended, without errors.
    - Visualization of PCA and t-SNE (10% each for a total of 20%)
    - Similarity function (20%)
    - Analogy function (20%) |

Import libraries

```
import torch
import torch.nn.functional as F

from sklearn.manifold import TSNE
from sklearn.decomposition import PCA

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


import numpy as np
from numpy.linalg import norm

import pickle

import os.path
```

Load file

```
EMBEDDINGS_DIMENSION_SIZE = 50

INPUT_PATH = 'inputs'
INPUT_FILE_NAME = f'glove.6B.{EMBEDDINGS_DIMENSION_SIZE}d.txt'

OUTPUT_PATH = 'outputs'
OUTPUT_FILE_NAME = f'embeddings_dict_{EMBEDDINGS_DIMENSION_SIZE}d.pkl'

def create_emb_dictionary(path):
    """
    Creates a dictionary mapping words to their corresponding
embedding vectors
    from a given file.

    The file should be formatted such that each line contains a word
followed
    by its embedding values, separated by spaces.

    Args:
        path (str): The file path containing the word embeddings.
```

```
    Returns:
        dict: A dictionary where keys are words (str) and values are
NumPy arrays
              representing the embedding vectors.

    Example:
        embeddings = create_emb_dictionary("embeddings.txt")
        print(embeddings["hello"])  # Output: array([...],
dtype=float32)
    """
    embeddings_dict = {}
    with open(path, 'r', encoding='utf-8') as f:
        for line in f:
            values = line.split()
            word = values[0]
            vector = np.array(values[1:], dtype=np.float32)
            embeddings_dict[word] = vector
    return embeddings_dict

outputFilePath = os.path.join(OUTPUT_PATH, OUTPUT_FILE_NAME)

if os.path.isfile(outputFilePath):
    print("Found embedding dictionary file! Loading data...")
    with open(outputFilePath, 'rb') as f:
        embeddings_dict = pickle.load(f)
else:
    print("No embedding dictionary found, generating embedding cached
file...")
    embeddings_dict = create_emb_dictionary(os.path.join(INPUT_PATH,
INPUT_FILE_NAME))

    os.makedirs(OUTPUT_PATH)
    with open(outputFilePath, 'wb') as f:
        pickle.dump(embeddings_dict, f)
```

```
No embedding dictionary found, generating embedding cached file...
```

See some embeddings

```
def show_n_first_words(path, n_words):
    """
    Displays the first `n_words` lines of a word embedding file,
showing each word
    and the length of its corresponding embedding vector.

    Args:
        path (str): The file path containing the word embeddings.
        n_words (int): The number of words to display.

    Example:
```

```python
        show_n_first_words("embeddings.txt", 5)
        # Output:
        # ['word1', '0.1', '0.2', ...] vector_length
        # ['word2', '0.3', '0.4', ...] vector_length
    """
    with open(path, 'r') as f:
        for i, line in enumerate(f):
            print(line.split(), len(line.split()[1:]))
            if i>=n_words: break

show_n_first_words(os.path.join(INPUT_PATH, INPUT_FILE_NAME), 5)
```

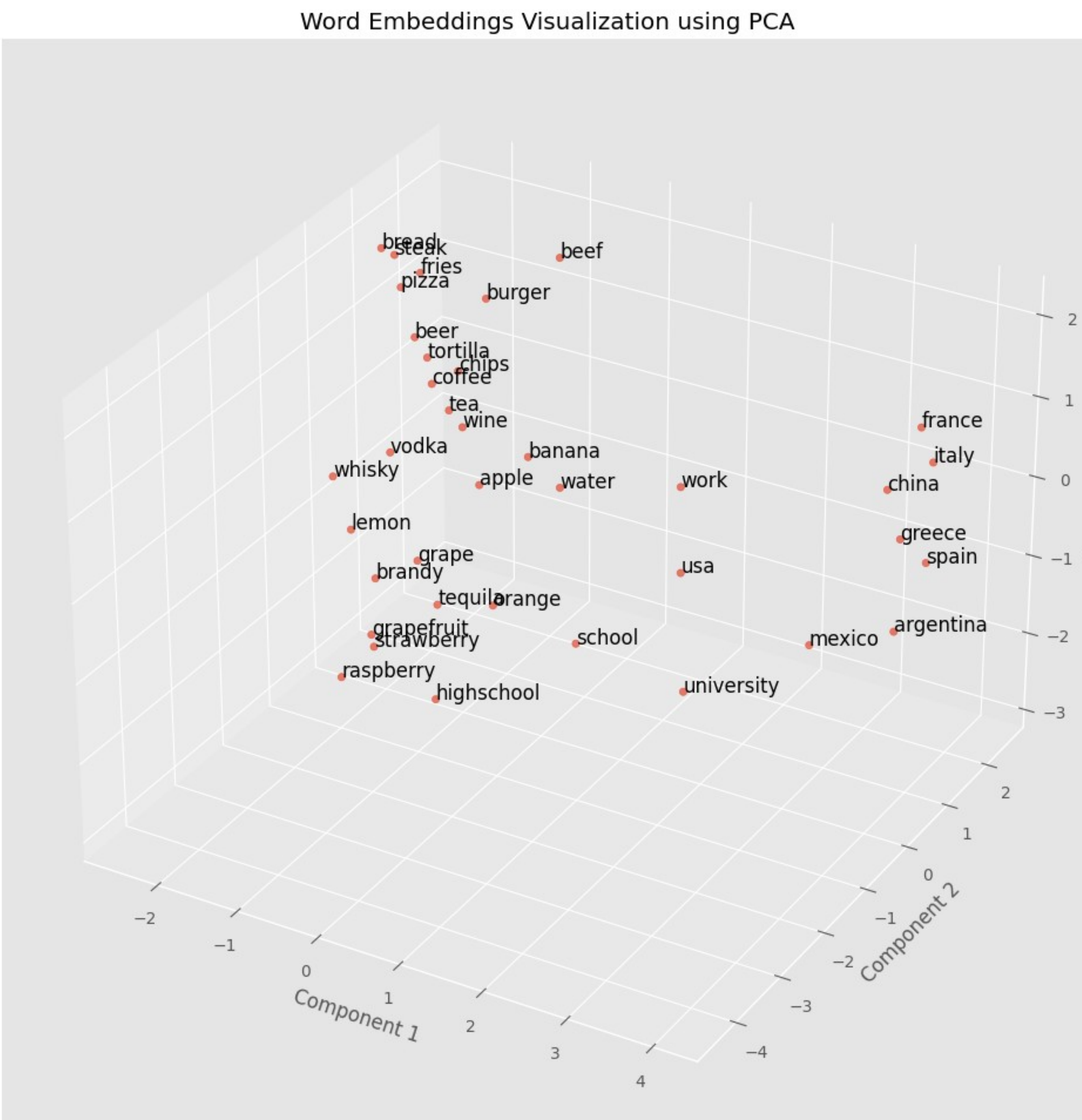
```
['the', '0.418', '0.24968', '-0.41242', '0.1217', '0.34527', '-
0.044457', '-0.49688', '-0.17862', '-0.00066023', '-0.6566',
'0.27843', '-0.14767', '-0.55677', '0.14658', '-0.0095095',
'0.011658', '0.10204', '-0.12792', '-0.8443', '-0.12181', '-0.016801',
'-0.33279', '-0.1552', '-0.23131', '-0.19181', '-1.8823', '-0.76746',
'0.099051', '-0.42125', '-0.19526', '4.0071', '-0.18594', '-0.52287',
'-0.31681', '0.00059213', '0.0074449', '0.17778', '-0.15897',
'0.012041', '-0.054223', '-0.29871', '-0.15749', '-0.34758', '-
0.045637', '-0.44251', '0.18785', '0.0027849', '-0.18411', '-0.11514',
'-0.78581'] 50
[',', '0.013441', '0.23682', '-0.16899', '0.40951', '0.63812',
'0.47709', '-0.42852', '-0.55641', '-0.364', '-0.23938', '0.13001', '-
0.063734', '-0.39575', '-0.48162', '0.23291', '0.090201', '-0.13324',
'0.078639', '-0.41634', '-0.15428', '0.10068', '0.48891', '0.31226',
'-0.1252', '-0.037512', '-1.5179', '0.12612', '-0.02442', '-0.042961',
'-0.28351', '3.5416', '-0.11956', '-0.014533', '-0.1499', '0.21864',
'-0.33412', '-0.13872', '0.31806', '0.70358', '0.44858', '-0.080262',
'0.63003', '0.32111', '-0.46765', '0.22786', '0.36034', '-0.37818', '-
0.56657', '0.044691', '0.30392'] 50
['.', '0.15164', '0.30177', '-0.16763', '0.17684', '0.31719',
'0.33973', '-0.43478', '-0.31086', '-0.44999', '-0.29486', '0.16608',
'0.11963', '-0.41328', '-0.42353', '0.59868', '0.28825', '-0.11547',
'-0.041848', '-0.67989', '-0.25063', '0.18472', '0.086876', '0.46582',
'0.015035', '0.043474', '-1.4671', '-0.30384', '-0.023441', '0.30589',
'-0.21785', '3.746', '0.0042284', '-0.18436', '-0.46209', '0.098329',
'-0.11907', '0.23919', '0.1161', '0.41705', '0.056763', '-6.3681e-05',
'0.068987', '0.087939', '-0.10285', '-0.13931', '0.22314', '-
0.080803', '-0.35652', '0.016413', '0.10216'] 50
['of', '0.70853', '0.57088', '-0.4716', '0.18048', '0.54449',
'0.72603', '0.18157', '-0.52393', '0.10381', '-0.17566', '0.078852',
'-0.36216', '-0.11829', '-0.83336', '0.11917', '-0.16605', '0.061555',
'-0.012719', '-0.56623', '0.013616', '0.22851', '-0.14396', '-
0.067549', '-0.38157', '-0.23698', '-1.7037', '-0.86692', '-0.26704',
'-0.2589', '0.1767', '3.8676', '-0.1613', '-0.13273', '-0.68881',
'0.18444', '0.0052464', '-0.33874', '-0.078956', '0.24185', '0.36576',
'-0.34727', '0.28483', '0.075693', '-0.062178', '-0.38988', '0.22902',
'-0.21617', '-0.22562', '-0.093918', '-0.80375'] 50
['to', '0.68047', '-0.039263', '0.30186', '-0.17792', '0.42962',
```

```
'0.032246', '-0.41376', '0.13228', '-0.29847', '-0.085253', '0.17118',
'0.22419', '-0.10046', '-0.43653', '0.33418', '0.67846', '0.057204',
'-0.34448', '-0.42785', '-0.43275', '0.55963', '0.10032', '0.18677',
'-0.26854', '0.037334', '-2.0932', '0.22171', '-0.39868', '0.20912',
'-0.55725', '3.8826', '0.47466', '-0.95658', '-0.37788', '0.20869', '-
0.32752', '0.12751', '0.088359', '0.16351', '-0.21634', '-0.094375',
'0.018324', '0.21048', '-0.03088', '-0.19722', '0.082279', '-0.09434',
'-0.073297', '-0.064699', '-0.26044'] 50
['and', '0.26818', '0.14346', '-0.27877', '0.016257', '0.11384',
'0.69923', '-0.51332', '-0.47368', '-0.33075', '-0.13834', '0.2702',
'0.30938', '-0.45012', '-0.4127', '-0.09932', '0.038085', '0.029749',
'0.10076', '-0.25058', '-0.51818', '0.34558', '0.44922', '0.48791', '-
0.080866', '-0.10121', '-1.3777', '-0.10866', '-0.23201', '0.012839',
'-0.46508', '3.8463', '0.31362', '0.13643', '-0.52244', '0.3302',
'0.33707', '-0.35601', '0.32431', '0.12041', '0.3512', '-0.069043',
'0.36885', '0.25168', '-0.24517', '0.25381', '0.1367', '-0.31178', '-
0.6321', '-0.25028', '-0.38097'] 50
```

Plot some embeddings

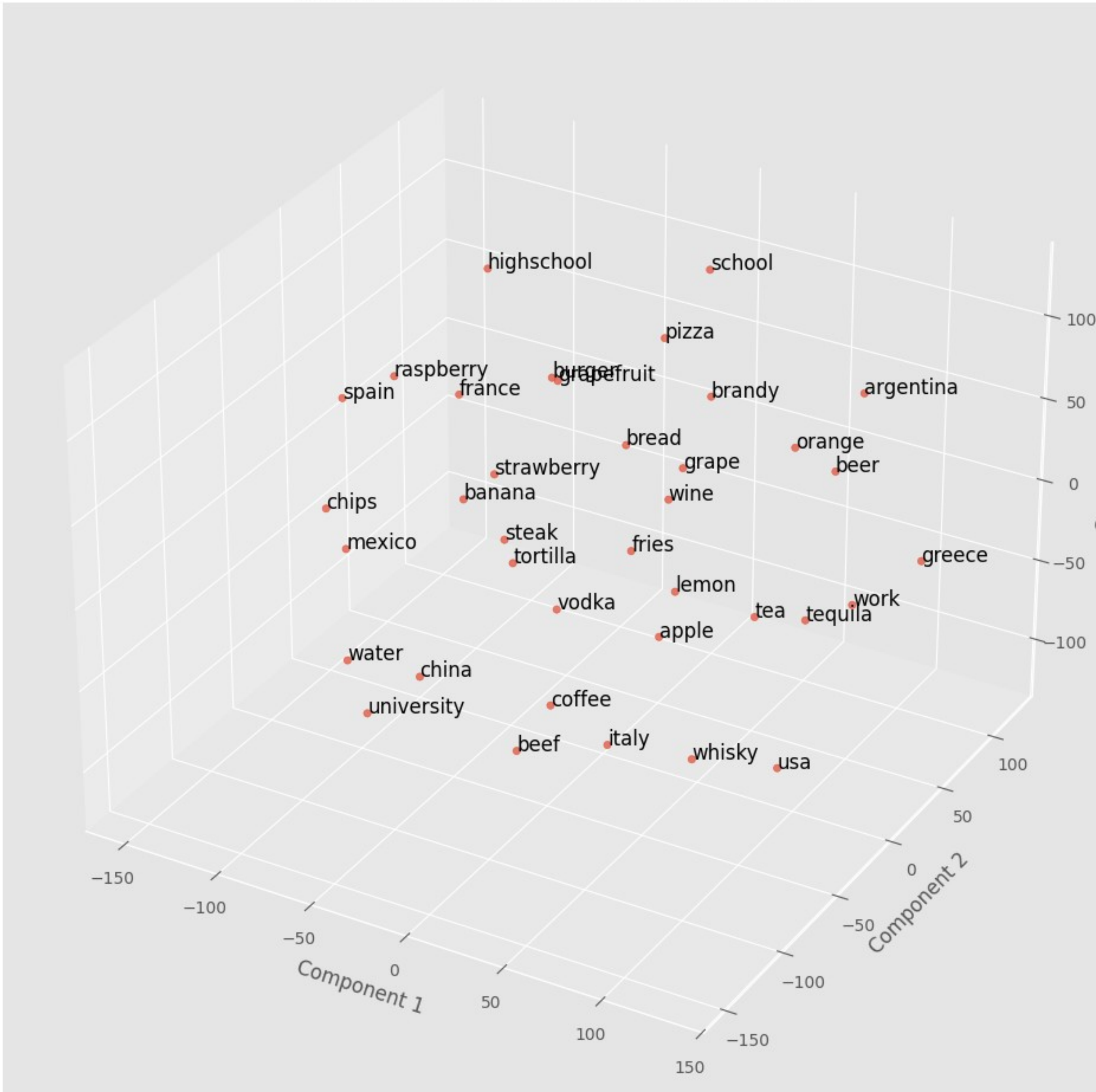
```
def plot_embeddings(words2show, embeddings_dict, func=PCA):
    """
    Plots a 3D visualization of word embeddings using dimensionality
reduction.

    This function reduces the dimensionality of the given word
embeddings using
    a specified technique (e.g., PCA or t-SNE) and plots the
transformed embeddings
    in a 3D space.

    Args:
        words2show (list of str): A list of words to visualize.
        embeddings_dict (dict): A dictionary mapping words to their
embedding vectors.
        func (callable, optional): A dimensionality reduction function
(e.g., PCA, TSNE).
                                   Defaults to PCA.

    Returns:
        None: Displays a 3D scatter plot of the reduced embeddings
with annotated words.

    Example:
        plot_embeddings("embeddings.txt", ["king", "queen", "man",
"woman"], 300, embeddings_dict, PCA)
    """
    word_vectors = np.array([embeddings_dict[word] for word in
words2show if word in embeddings_dict])
    words_filtered = [word for word in words2show if word in
```

```
embeddings_dict]

    if word_vectors.shape[0] == 0:
        print("No words found in the embeddings dictionary.")
        return

    reducer = func(n_components=3)
    reduced_embeddings = reducer.fit_transform(word_vectors)

    fig = plt.figure(figsize=(12, 14))
    ax = fig.add_subplot(111, projection='3d')

    ax.scatter(reduced_embeddings[:, 0], reduced_embeddings[:, 1],
reduced_embeddings[:, 2], alpha=0.7)

    for i, word in enumerate(words_filtered):
        ax.text(reduced_embeddings[i, 0], reduced_embeddings[i, 1],
reduced_embeddings[i, 2], word, fontsize=12)

    ax.set_title(f"Word Embeddings Visualization using
{func.__name__}")
    ax.set_xlabel("Component 1")
    ax.set_ylabel("Component 2")
    ax.set_zlabel("Component 3")

    plt.show()

words = ['burger', 'tortilla', 'bread', 'pizza', 'beef', 'steak',
'fries', 'chips',
            'argentina', 'mexico', 'spain', 'usa', 'france', 'italy',
'greece', 'china',
            'water', 'beer', 'tequila', 'wine', 'whisky', 'brandy',
'vodka', 'coffee', 'tea',
            'apple', 'banana', 'orange', 'lemon', 'grapefruit',
'grape', 'strawberry', 'raspberry',
            'school', 'work', 'university', 'highschool']

plot_embeddings(words, embeddings_dict, PCA)
```




```
# t-SNE dimensionality reduction for visualization
plot_embeddings(words, embeddings_dict, TSNE)
```

Word Embeddings Visualization using TSNE



Let us compute analogies

We'll use the cosine similarity formula in order to compute the angle between the vectors to get thge similar words.

```
def find_most_similar_vector(vector, embeddings_dict, top_n=10):
    """
    Finds the most similar word vectors to a given vector based on
cosine similarity.

    Args:
        vector (numpy.ndarray): The reference vector to compare
against.
        embeddings_dict (dict): A dictionary where keys are words and
values are their corresponding embedding vectors.
        top_n (int, optional): The number of most similar words to
return. Defaults to 10.

    Returns:
        list of tuples: A sorted list of tuples containing the most
similar words and their cosine similarity scores.
                        Format: [(word1, similarity1), (word2,
similarity2), ...]

    Example:
        similar_words =
find_most_similar_vector(embeddings_dict["king"], embeddings_dict,
top_n=5)
        print(similar_words)
        # Output: [('queen', 0.89), ('prince', 0.85), ('monarch',
0.83), ...]
    """
    similarities = {}
    for other_word, other_vector in embeddings_dict.items():
        similarity = np.dot(vector, other_vector) /
(np.linalg.norm(vector) * np.linalg.norm(other_vector))
        similarities[other_word] = similarity
```

```
    sorted_similarities = sorted(similarities.items(), key=lambda x:
x[1], reverse=True)
    return sorted_similarities[1:top_n+1]
```

One surprising aspect of GloVe vectors is that the directions in the embedding space can be meaningful. The structure of the GloVe vectors certain analogy-like relationship like this tend to hold: `king − man + woman ≈ queen`

```
# analogy
def analogy(word1, word2, word3, embeddings_dict):
    """
    Solves a word analogy of the form: word1 is to word2 as word3 is
to X.
    This is done by performing a vector arithmetic operation and
finding the most
    similar word to the resulting vector.

    Args:
        word1 (str): The first word in the analogy.
        word2 (str): The second word in the analogy.
        word3 (str): The third word in the analogy.
        embeddings_dict (dict): A dictionary where keys are words and
values are their corresponding embedding vectors.

    Returns:
        list: A list of the most similar words to the resulting
analogy vector, based on cosine similarity.

    Example:
        analogy_result = analogy("king", "queen", "man",
embeddings_dict)
        print(analogy_result)
        # Output: [('woman', 0.95), ...]
    """
    if word1 not in embeddings_dict or word2 not in embeddings_dict or
word3 not in embeddings_dict:
        return "One or more words not found in the dictionary."

    vec1 = embeddings_dict[word1]
    vec2 = embeddings_dict[word2]
    vec3 = embeddings_dict[word3]
    analogy_vector = vec2 - vec1 + vec3

    return find_most_similar_vector(analogy_vector, embeddings_dict)

analogy('man', 'king', 'woman', embeddings_dict)
```

```
[('queen', np.float32(0.86095804)),
 ('daughter', np.float32(0.7684512)),
 ('prince', np.float32(0.7640699)),
```

```
 ('throne', np.float32(0.76349705)),
 ('princess', np.float32(0.7512728)),
 ('elizabeth', np.float32(0.75064886)),
 ('father', np.float32(0.73144966)),
 ('kingdom', np.float32(0.7296158)),
 ('mother', np.float32(0.728001)),
 ('son', np.float32(0.72795373))]
```

```
def find_most_similar(word, embeddings_dict, top_n=10):
    """
    Finds the most similar words to a given word based on its
embedding vector's
    cosine similarity with other word vectors in the dictionary.

    Args:
        word (str): The word to find similarities for.
        embeddings_dict (dict): A dictionary where keys are words and
values are their corresponding embedding vectors.
        top_n (int, optional): The number of most similar words to
return. Defaults to 10.

    Returns:
        list: A list of the most similar words and their cosine
similarity scores,
              sorted in descending order of similarity.
              Format: [(word1, similarity1), (word2,
similarity2), ...]

    Example:
        similar_words = find_most_similar("king", embeddings_dict,
top_n=5)
        print(similar_words)
        # Output: [('queen', 0.89), ('prince', 0.85), ('monarch',
0.83), ...]
    """
    if word not in embeddings_dict:
        return "Word not found in dictionary."

    word_vector = embeddings_dict[word]
    return find_most_similar_vector(word_vector, embeddings_dict,
top_n)

most_similar = find_most_similar('mexico', embeddings_dict)

for i, w in enumerate(most_similar, 1):
    print(f'{i} ---> {w[0]}')
```

```
1 ---> mexican
2 ---> venezuela
3 ---> colombia
```

```
4 ---> peru
5 ---> chile
6 ---> puerto
7 ---> rico
8 ---> cuba
9 ---> guatemala
10 ---> panama
```