**Embedding Assignments**

**Problem Statement:**

This assignment aims to explore various text embedding techniques using a chosen dataset. By implementing different embedding methods such as **Bag of Words, TF-IDF, Word2Vec, GloVe, and FastText**, students will gain a deeper understanding of how to convert text into numerical representations. These embeddings will be used to analyze and extract meaningful insights from the dataset.

**What is Embedding Techniques?**

Embeddings are a way to represent data in a continuous vector space, which is useful for machine learning models. Different embedding techniques capture various types of relationships in data, making them essential in many applications, especially in Natural Language Processing (NLP), recommendation systems, and more.

**What is Vector Space Models Concept?**

Vector Space Models (VSMs) are mathematical models used to represent text data (or other types of data) in a continuous vector space. These models are fundamental in various applications like Natural Language Processing (NLP), information retrieval, and recommendation systems.

**Representation**: In a VSM, words, sentences, documents, or other entities are represented as vectors (points) in a multi-dimensional space. Each dimension corresponds to a unique feature, which could be a word, term, or characteristic of the data.

**Vector Construction**: The vectors are typically constructed by assigning weights to each feature, which indicates the importance of that feature in the context of the entity being represented.

**Popular embedding techniques –**

**Bag of Words (BoW)**

Bag of Words is a simple and commonly used technique in natural language processing (NLP) to represent text data. In this model, a text (such as a sentence or document) is represented as a collection of words, ignoring grammar and word order but keeping the word frequency. The basic idea is to map each unique word in the text to a number that indicates its occurrence (or frequency) in the text**.**

**Steps to Create a Bag of Words Model:**

1. Tokenization: Split the text into individual words.
2. Vocabulary Creation: Create a list of all unique words in the entire text (corpus).
3. Vectorization: Convert each text into a vector where each element represents the frequency of a word in the vocabulary.

**Example with Code**

Let's implement this in Python using a simple example.

**Step 1: Import Required Libraries**

**from sklearn.feature\_extraction.text import CountVectorizer**

**# Example documents**

**documents = [**

**"Machine learning is fascinating",**

**"Machine learning is a subset of artificial intelligence",**

**"Deep learning is a subset of machine learning",**

**"Artificial intelligence is transforming the world"**

**]**

**Step 2: Initialize the CountVectorizer**

CountVectorizer is a tool in scikit-learn that converts a collection of text documents into a matrix of token counts (BoW model).

**# Initialize CountVectorizer**

**vectorizer = CountVectorizer()**

**Step 3: Fit and Transform the Documents**

Now, let's fit the vectorizer to our documents and transform the text into a BoW representation.

**# Fit and transform the documents**

**X = vectorizer.fit\_transform(documents)**

**# Convert to an array for better visualization**

**X\_array = X.toarray()**

**# Print the BoW matrix**

**print(X\_array)**

**Step 4: Output the Vocabulary**

Let's also print out the vocabulary that CountVectorizer has created.

**# Get the feature names (vocabulary)**

**vocab = vectorizer.get\_feature\_names\_out()**

**# Print the vocabulary**

**print(vocab)**

**Output Explanation**

When you run the above code, you'll get two main outputs:

1. BoW Matrix: A matrix where each row corresponds to a document, and each column corresponds to a word in the vocabulary. The values in the matrix represent the count of the word in that document.

[[0 0 1 0 0 0 0 1 1 1 0 0]

[1 1 0 0 0 0 0 1 1 0 1 1]

[1 0 0 1 0 0 0 1 1 0 0 0]

[0 1 0 0 1 1 1 0 0 0 0 0]]

1. **Vocabulary**: A list of all the unique words in the corpus, ordered as they appear in the matrix.

['artificial' 'deep' 'fascinating' 'intelligence' 'is' 'learning' 'machine' 'of' 'subset' 'the' 'transforming' 'world']

**How to Interpret the Matrix?**

1. Each row corresponds to one of the input documents.
2. Each column corresponds to one of the unique words (features) in the corpus.
3. The value at the intersection of a row and a column is the frequency of that word in the document.
   1. For example, the first row [0 0 1 0 0 0 0 1 1 1 0 0] tells us that the first document contains the words "fascinating", "of", "subset", "the" once each and does not contain any other words in the vocabulary.
   2. **Limitations of Bag of Words**
4. **Ignores Word Order**: BoW does not consider the order in which words appear, potentially losing important context.
5. **Ignores Semantics**: BoW does not capture the meaning of the words, only their frequency.
6. **High Dimensionality**: If the corpus is large, the vocabulary (and thus the vectors) can become very large, leading to high-dimensional data.

**TF-IDF**

**TF-IDF** stands for **Term Frequency-Inverse Document Frequency**. It's a statistical measure used to evaluate how important a word is to a document in a collection (or corpus). The importance increases proportionally to the number of times a word appears in the document but is offset by how common the word is in the entire corpus. This helps in highlighting words that are more unique to a particular document.

TF-IDF is particularly useful in information retrieval, text mining, and text classification.

**Components of TF-IDF**

1. **Term Frequency (TF)**: Measures how frequently a term appears in a document.

TF(t,d)=Total number of terms in document d/Number of times term t appears in a document d​

1. **Inverse Document Frequency (IDF)**: Measures how important a term is by reducing the weight of terms that appear frequently in many documents.

IDF(t)=log(Number of documents with term t/Total number of documents​)

1. **TF-IDF**: The product of TF and IDF, which gives the final score for each term in the document.

TF-IDF(t,d)=TF(t,d)×IDF(t)

**Code Implementation with Example**

from sklearn.feature\_extraction.text import TfidfVectorizer

# Example documents

documents = [

"Machine learning is fascinating",

"Machine learning is a subset of artificial intelligence",

"Deep learning is a subset of machine learning",

"Artificial intelligence is transforming the world"

]

Step 2: Initialize the TfidfVectorizer

TfidfVectorizer is a tool in scikit-learn that converts a collection of raw documents to a matrix of TF-IDF features.

# Initialize TfidfVectorizer

vectorizer = TfidfVectorizer()

**Step 3: Fit and Transform the Documents**

Now, let's fit the vectorizer to our documents and transform the text into a TF-IDF representation.

# Fit and transform the documents

X = vectorizer.fit\_transform(documents)

# Convert to an array for better visualization

X\_array = X.toarray()

# Print the TF-IDF matrix

print(X\_array)

**Step 4: Output the Vocabulary**

Let's also print out the vocabulary that TfidfVectorizer has created and map it to the TF-IDF matrix.

# Get the feature names (vocabulary)

vocab = vectorizer.get\_feature\_names\_out()

# Print the vocabulary

print(vocab)

**Output Explanation**

When you run the above code, you'll get two main outputs:

1. **TF-IDF Matrix**: A matrix where each row corresponds to a document, and each column corresponds to a word in the vocabulary. The values in the matrix represent the TF-IDF score of the word in that document.

Example output:

[[0. 0. 0.51785612 0. 0. 0. 0.

0.41428875 0.51785612 0.41428875 0. 0. ]

[0.45086156 0.45086156 0. 0. 0. 0. 0.

0.28847675 0.36007379 0.28847675 0.45086156 0.45086156]

[0.43779123 0. 0. 0. 0. 0.55422844 0.43779123

0.27922078 0.34852409 0.27922078 0. 0. ]

[0. 0.40204024 0. 0.53404633 0.53404633 0. 0.

0. 0. 0. 0.40204024 0.40204024]]

**2. Vocabulary**: A list of all the unique words in the corpus, ordered as they appear in the matrix.

['artificial' 'deep' 'fascinating' 'intelligence' 'is' 'learning' 'machine' 'of' 'subset' 'the' 'transforming' 'world']

**How to Interpret the Matrix?**

* **TF-IDF Values**: Each value in the matrix represents the TF-IDF score of a word in a specific document. A higher score indicates that the word is more relevant or unique to that document compared to others in the corpus.

For example, in the matrix:

* The first document has a high TF-IDF score for "fascinating," meaning this word is particularly important in that document.
* The words "artificial," "intelligence," and "transforming" have higher scores in the last document, indicating their significance in that document.

**Advantages of TF-IDF**

* **Balances Frequency and Uniqueness**: TF-IDF considers both the frequency of a term in a document and its rarity across all documents, making it a powerful tool for identifying significant words.
* **Improves Search Results**: In information retrieval, TF-IDF helps in ranking documents based on their relevance to a query.
* **Reduces Noise**: By down-weighting common words that appear in many documents, TF-IDF helps to focus on words that are more discriminative.

**Limitations of TF-IDF**

* **Ignores Word Order**: Like Bag of Words, TF-IDF does not consider the order of words, which can lead to a loss of context.
* **Vocabulary Size**: For large corpora, the vocabulary can be very large, leading to a high-dimensional space.
* **Static Representation**: TF-IDF does not capture the semantics or meaning of words; it only reflects their statistical properties.

**Word2Vec**

**Word2Vec** is a popular word embedding technique developed by Google in 2013. It is a type of neural network model that transforms words into continuous vector representations in a lower-dimensional space. These word embeddings capture semantic relationships between words, meaning that words with similar meanings are located close to each other in the vector space.

**Key Concepts in Word2Vec**

1. **Embedding**: Each word is represented as a dense vector of fixed size, where similar words have similar vectors.
2. **Context**: The context of a word refers to the words surrounding it in a sentence. Word2Vec uses the context to learn word embeddings.
3. **Models**:
   * **Continuous Bag of Words (CBOW)**: Predicts the current word based on the context (surrounding words).
   * **Skip-gram**: Predicts the context (surrounding words) based on the current word. This is the more commonly used model.

**Word2Vec Implementation**

We will use the gensim library to implement Word2Vec. Let's walk through the process.

**Step 1: Install Required Libraries**

If you don't have gensim installed, you can install it using pip:

bash

Copy code

pip install gensim

**Step 2: Import Required Libraries**

python

Copy code

from gensim.models import Word2Vec

from gensim.models import KeyedVectors

from nltk.tokenize import word\_tokenize

import nltk

# Download necessary NLTK data files

nltk.download('punkt')

**Step 3: Prepare the Data**

Let's create some sample sentences and tokenize them. In a real-world scenario, you would use a much larger corpus.

python

Copy code

# Sample corpus

sentences = [

"Machine learning is fascinating",

"Machine learning is a subset of artificial intelligence",

"Deep learning is a subset of machine learning",

"Artificial intelligence is transforming the world",

"The future of AI is exciting",

"Deep learning models are very powerful"

]

# Tokenize the sentences

tokenized\_sentences = [word\_tokenize(sentence.lower()) for sentence in sentences]

**Step 4: Train the Word2Vec Model**

We'll use the Skip-gram model in this example, but you can also use CBOW by setting sg=0.

python

Copy code

# Train the Word2Vec model

model = Word2Vec(sentences=tokenized\_sentences, vector\_size=100, window=3, min\_count=1, sg=1, workers=4)

# Save the model for later use

model.save("word2vec.model")

* **vector\_size=100**: The dimensionality of the word vectors.
* **window=3**: The maximum distance between the current and predicted word within a sentence.
* **min\_count=1**: Ignores all words with a total frequency lower than this.
* **sg=1**: Specifies Skip-gram model; use sg=0 for CBOW.

**Step 5: Use the Model**

Now that the model is trained, let's explore how to use it.

**1. Get Word Vectors**

You can retrieve the vector for any word in your vocabulary.

python

Copy code

# Get the vector for a specific word

vector = model.wv['learning']

print(vector)

**2. Find Similar Words**

You can find words similar to a given word based on cosine similarity.

python

Copy code

# Find similar words

similar\_words = model.wv.most\_similar('learning', topn=5)

print(similar\_words)

**3. Word Analogies**

Word2Vec can perform word analogies, such as "king - man + woman = queen".

python

Copy code

# Word analogy example

analogy = model.wv.most\_similar(positive=['king', 'woman'], negative=['man'], topn=1)

print(analogy)

**4. Visualize Word Embeddings**

You can visualize the word embeddings using dimensionality reduction techniques like PCA or t-SNE. Here's how to do it with PCA.

python

Copy code

import matplotlib.pyplot as plt

from sklearn.decomposition import PCA

# Get the word vectors

words = list(model.wv.index\_to\_key)

vectors = model.wv[words]

# Reduce the dimensionality to 2D using PCA

pca = PCA(n\_components=2)

pca\_result = pca.fit\_transform(vectors)

# Plot the words

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

plt.scatter(pca\_result[:, 0], pca\_result[:, 1])

for i, word in enumerate(words):

plt.annotate(word, xy=(pca\_result[i, 0], pca\_result[i, 1]))

plt.show()

**Output Explanation**

1. **Word Vectors**: The model.wv['learning'] output will give you the 100-dimensional vector representing the word "learning". This vector captures the semantic meaning of the word.
2. **Similar Words**: The most\_similar function will return a list of words similar to "learning". These are words that appear in similar contexts within the corpus.
3. **Word Analogies**: The analogy example will find a word that fits the relationship "king" is to "man" as "queen" is to "woman". This demonstrates the power of word embeddings in capturing relationships.
4. **Visualization**: The PCA plot will show the 2D representation of the word embeddings. Words with similar meanings or contexts will be close to each other in this plot.

**Advantages of Word2Vec**

* **Semantic Relationships**: Word2Vec captures complex semantic relationships between words.
* **Efficient**: The model is computationally efficient and can be trained on large corpora.
* **Versatile**: It can be used in various NLP tasks like sentiment analysis, machine translation, and more.

**Limitations of Word2Vec**

* **Requires Large Data**: Word2Vec requires a large corpus to produce high-quality word embeddings.
* **Context Ignorance**: It doesn't capture the context in which words are used, unlike more advanced models like BERT.
* **Out-of-Vocabulary (OOV) Words**: Words not seen during training are not represented.

**GloVe (Global Vectors for Word Representation)**

GloVe Overview:

1. It's based on the idea that word meanings can be derived from the statistical properties of their co-occurrence in a large corpus.

2. It combines the advantages of two major model families: global matrix factorization and local context window methods.

3. The main insight is that ratios of word-word co-occurrence probabilities have the potential to encode meaning components.

Let's start with a simple example to illustrate the concept, and then we'll move on to more complex implementations.

First, let's create a simple co-occurrence matrix:

```python

import numpy as np

# Simple co-occurrence matrix

co\_occurrence = np.array([

[0, 2, 1],

[2, 0, 3],

[1, 3, 0]

])

print("Co-occurrence matrix:")

print(co\_occurrence)

```

This code creates a simple 3x3 co-occurrence matrix. In a real-world scenario, this matrix would be much larger and would be created by counting word co-occurrences in a large corpus of text.

Now, let's implement a simplified version of the GloVe objective function:

```python

def glove\_loss(W, b, U, b\_tilde, co\_occurrence):

loss = 0

for i in range(co\_occurrence.shape[0]):

for j in range(co\_occurrence.shape[1]):

if i != j:

X\_ij = co\_occurrence[i, j]

if X\_ij > 0:

loss += (W[i].dot(U[j]) + b[i] + b\_tilde[j] - np.log(X\_ij))\*\*2

return loss

# Initialize random word vectors and biases

vocabulary\_size = 3

vector\_size = 2

W = np.random.randn(vocabulary\_size, vector\_size)

U = np.random.randn(vocabulary\_size, vector\_size)

b = np.random.randn(vocabulary\_size)

b\_tilde = np.random.randn(vocabulary\_size)

initial\_loss = glove\_loss(W, b, U, b\_tilde, co\_occurrence)

print(f"Initial loss: {initial\_loss}")

```

This code defines the GloVe loss function and initializes random word vectors and biases. The loss function calculates the difference between the dot product of word vectors (plus biases) and the logarithm of their co-occurrence count.

Now, let's implement a simple gradient descent to optimize these vectors:

```python

learning\_rate = 0.01

num\_iterations = 1000

for iteration in range(num\_iterations):

# Gradient descent update

for i in range(vocabulary\_size):

for j in range(vocabulary\_size):

if i != j:

X\_ij = co\_occurrence[i, j]

if X\_ij > 0:

diff = W[i].dot(U[j]) + b[i] + b\_tilde[j] - np.log(X\_ij)

# Update word vectors and biases

W[i] -= learning\_rate \* diff \* U[j]

U[j] -= learning\_rate \* diff \* W[i]

b[i] -= learning\_rate \* diff

b\_tilde[j] -= learning\_rate \* diff

# Print loss every 100 iterations

if (iteration + 1) % 100 == 0:

current\_loss = glove\_loss(W, b, U, b\_tilde, co\_occurrence)

print(f"Iteration {iteration + 1}, Loss: {current\_loss}")

print("\nFinal word vectors:")

print(W)

```

This code performs gradient descent to optimize the word vectors and biases. After training, we print the final word vectors.

In practice, GloVe would be applied to much larger vocabularies and co-occurrence matrices, and would often use more sophisticated optimization techniques. However, this simplified example demonstrates the core concepts:

1. We start with a co-occurrence matrix that captures word relationships in a corpus.

2. We define an objective function that aims to make the dot product of word vectors (plus biases) approximate the logarithm of the words' co-occurrence probability.

3. We use optimization techniques (in this case, simple gradient descent) to find word vectors that minimize this objective function.

**FastText**

FastText is an extension of Word2Vec that considers subword information, allowing it to capture the meaning of rare and out-of-vocabulary (OOV) words more effectively. Unlike Word2Vec, which treats each word as an atomic unit, FastText represents words as bags of character n-grams. This enables it to generate embeddings for words not seen during training by composing them from their subwords.

**Step 1: Preparing the Data**

First, let's prepare a small corpus and tokenize the words into character n-grams.

**python**

import numpy as np

import itertools

from collections import defaultdict

# Sample corpus

corpus = [

"I love machine learning",

"I love deep learning",

"deep learning is fun",

"I enjoy learning machine learning"

]

# Preprocess the corpus

words = list(itertools.chain(\*[sentence.lower().split() for sentence in corpus]))

vocab = set(words)

word\_to\_id = {word: i for i, word in enumerate(vocab)}

id\_to\_word = {i: word for word, i in word\_to\_id.items()}

# Function to generate character n-grams for a word

def get\_ngrams(word, n=3):

word = f"<{word}>"

ngrams = [word[i:i+n] for i in range(len(word)-n+1)]

return ngrams

# Example: n-grams for the word 'machine'

word = "machine"

ngrams = get\_ngrams(word)

print(f"Word: {word}, N-grams: {ngrams}")

**Step 2: Building the FastText Model**

The FastText model is built by training word and n-gram embeddings. For simplicity, we'll use a Continuous Bag of Words (CBOW) model where the task is to predict a word based on its context. We'll treat both words and their n-grams as input features.

**python**

class FastText:

def \_\_init\_\_(self, vocab\_size, embed\_size, n=3):

self.vocab\_size = vocab\_size

self.embed\_size = embed\_size

self.n = n

# Initialize word and n-gram embeddings

self.word\_embeddings = np.random.randn(vocab\_size, embed\_size)

self.ngram\_embeddings = defaultdict(lambda: np.random.randn(embed\_size))

def get\_embedding(self, word):

ngrams = get\_ngrams(word, self.n)

word\_vec = self.word\_embeddings[word\_to\_id[word]]

# Sum the word embedding and the n-gram embeddings

ngram\_vecs = np.sum([self.ngram\_embeddings[ngram] for ngram in ngrams], axis=0)

return word\_vec + ngram\_vecs

def fit(self, context\_pairs, epochs=100, learning\_rate=0.05):

for epoch in range(epochs):

total\_loss = 0

for context, target in context\_pairs:

context\_vecs = np.sum([self.get\_embedding(word) for word in context], axis=0)

target\_vec = self.word\_embeddings[word\_to\_id[target]]

# Calculate the prediction

score = context\_vecs.dot(target\_vec)

prob = 1 / (1 + np.exp(-score))

# Calculate the loss (negative log likelihood)

loss = -np.log(prob)

total\_loss += loss

# Compute gradients and update embeddings

grad = (prob - 1) \* context\_vecs

self.word\_embeddings[word\_to\_id[target]] -= learning\_rate \* grad

for word in context:

grad\_word = (prob - 1) \* self.word\_embeddings[word\_to\_id[target]]

self.word\_embeddings[word\_to\_id[word]] -= learning\_rate \* grad\_word

# Update n-gram embeddings

for ngram in get\_ngrams(word, self.n):

self.ngram\_embeddings[ngram] -= learning\_rate \* grad\_word

if epoch % 10 == 0:

print(f'Epoch: {epoch}, Loss: {total\_loss}')

def get\_word\_vector(self, word):

return self.get\_embedding(word)

# Generate context-target pairs for training (CBOW model)

context\_size = 2

context\_pairs = []

for sentence in corpus:

tokens = sentence.lower().split()

for i in range(len(tokens)):

target = tokens[i]

context = tokens[max(0, i - context\_size): i] + tokens[i + 1: min(len(tokens), i + 1 + context\_size)]

context\_pairs.append((context, target))

# Initialize FastText model

embed\_size = 10

fasttext = FastText(vocab\_size=len(vocab), embed\_size=embed\_size)

# Train the model on the context-target pairs

fasttext.fit(context\_pairs, epochs=100)

# Get the embedding for a specific word

word\_vector = fasttext.get\_word\_vector("learning")

print(f"Word Vector for 'learning': {word\_vector}")

**Step 3: Using the Word Embeddings**

Now that we have trained the FastText model, we can use the learned word vectors for various tasks, like finding similar words.

**python**

# Example: Compute the similarity between two words

def cosine\_similarity(vec1, vec2):

return np.dot(vec1, vec2) / (np.linalg.norm(vec1) \* np.linalg.norm(vec2))

word1, word2 = "machine", "learning"

similarity = cosine\_similarity(fasttext.get\_word\_vector(word1), fasttext.get\_word\_vector(word2))

print(f"Similarity between '{word1}' and '{word2}': {similarity}")

**Explanation**

* **Character N-Grams:** FastText represents words as bags of character n-grams (e.g., 'machine' could be represented by the n-grams <ma, mac, ach, chi, hin, ine, and ne>).
* **Embedding Learning:** FastText learns embeddings for both entire words and their n-grams. The final word embedding is a sum of the word's embedding and its n-grams' embeddings.
* **Handling OOV Words:** Because FastText uses n-grams, it can generate embeddings for words it has never seen during training by summing the embeddings of the word's n-grams.