به نام خدا

دانشگاه تهران

دانشکدگان فنی

دانشکده مهندسی برق و کامپیوتر

درس پردازش زبان طبیعی

پاسخ تمرین{2}

نام و نام خانوادگی: مریم حسینعلی

شماره دانشجویی: 610398209

QUESTION2

PART1 - DATASET PREPROCESSING

First, I uploaded the dataset into my Google Drive to access it in Google Colab:

```
[ ] from google.colab import drive
    drive.mount('/content/drive')

    Drive already mounted at /content/drive; to attempt to forcibly remount

    json_file_path = '/content/drive/My Drive/sarcasm.json'
```

Then I loaded the dataset into a pandas DataFrame:

```
df = pd.read_json('/content/drive/My Drive/sarcasm.json', lines=True)
```

To ensure the model focuses only on important words for detecting sarcasm, I applied a two-step cleaning process to the dataset headlines. First, I created a “clean_text” function to standardize and simplify the text. This function: 1) converts all text to lowercase. 2) removes any content within square brackets and hyperlinks.3) strips away special characters, punctuation, and numbers 4)removes line breaks. After that, I applied “clean_text” function to each headline in the dataset, for cleaning and standardizing the text.

Next, I removed stopwords.

```
# function for cleaning data
def clean_text(text):
    text = text.lower()
    text = re.sub(r'\[.*?\]', '', text)
    text = re.sub(r'http\S+', '', text)
    text = re.sub(r'[%s]' % re.escape("""!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~"""), '', text)
    text = re.sub(r'\n', '', text)
    text = re.sub(r'\w*\d\w*', '', text)
    return text

# apply cleaning function to each headline
df['cleaned_headline'] = df['headline'].apply(clean_text)

# removing stopwords from cleaned headlines
stop_words = set(stopwords.words('english'))
df['cleaned_headline'] = df['cleaned_headline'].apply(lambda x: ' '.join([word for word in x.split() if word not in stop_words]))
```

I extracted the “is_sarcastic” column from df dataframe, creating the array “y”.

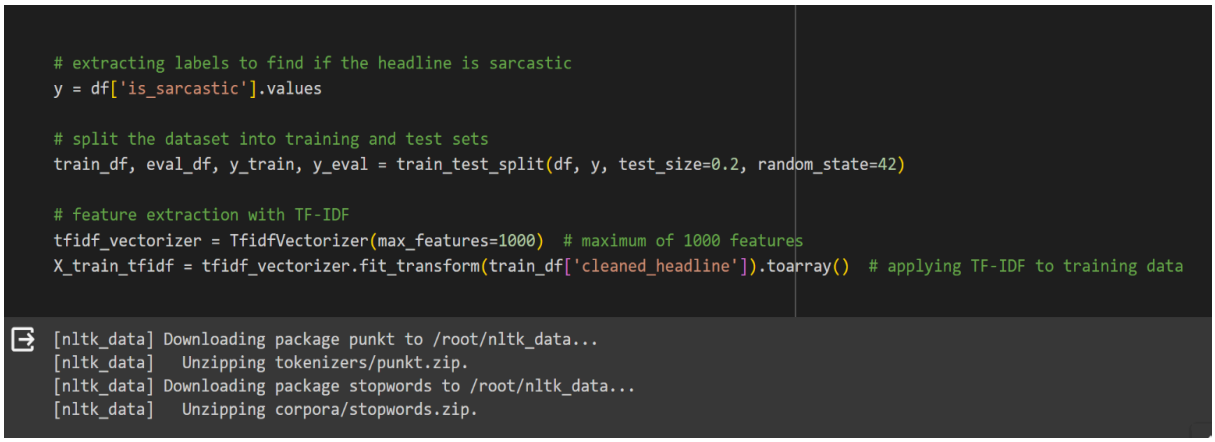
This array marks each headline with a 1 for sarcastic or a 0 for not sarcastic, which the model will use to make its predictions.

After that i splitted the dataset into training and test sets.

It was needed to convert the cleaned headlines into numbers so I used the TF-IDF vectorization method for this.

It selects up to 1000 of the most significant words from all the headlines. This method looks at how often a word appears in a headline but also checks if it's not too common across all headlines, making sure the words we focus on are truly special.

By doing this, every headline turned into a list of numbers that represents how important each of these top words is in that headline. So, the model can learn from the patterns in these numbers to detect sarcasm.



```
# extracting labels to find if the headline is sarcastic
y = df['is_sarcastic'].values

# split the dataset into training and test sets
train_df, eval_df, y_train, y_eval = train_test_split(df, y, test_size=0.2, random_state=42)

# feature extraction with TF-IDF
tfidf_vectorizer = TfidfVectorizer(max_features=1000)  # maximum of 1000 features
X_train_tfidf = tfidf_vectorizer.fit_transform(train_df['cleaned_headline']).toarray()  # applying TF-IDF to training data
```

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

PART2 - GLOVE

For easier access, I uploaded the GloVe 6B.zip file to my Google Drive and then unzipped it directly within the drive:

```
!unzip "/content/drive/My Drive/glove.6B.zip" -d "/content/drive/My Drive/glove.6B"
```

Before using GloVe embeddings it was essential to first process the cleaned headlines by tokenizing them.

To achieve this, I used the “Tokenizer” from TensorFlow's “keras.preprocessing.text” module, setting a limit for the top 1000 words based on their frequency in the dataset. This approach ensures that the model concentrates on the most relevant words.

By fitting the tokenizer on cleaned headlines, I transformed the textual data into sequences of integers, each representing a unique word in the dataset. This numerical representation was essential for embedding the text using GloVe vectors.

After tokenizing, the sequences had diffrent length because of diffrent headline sizes. To fix this, I applied padding to standardize the length of each sequence, ensuring that all input data has a uniform shape. The “pad_sequences” function was utilized to extend shorter sequences with zeros at the end, achieving a consistent length across the dataset.

The "word_index" created here maps each word to a number, preparing us to connect these words to GloVe vectors next.

This step prepares us to use GloVe vectors, making it easier to work with the detailed meanings in GloVe embeddings. It helps us get ready for the next steps in training and testing the model.

```
tokenizer = Tokenizer(num_words=1000)  # limit vocab to top 1000 words

# learning vocab from cleaned headlines in train_df
tokenizer.fit_on_texts(train_df['cleaned_headline'])

# converting the cleaned headlines into sequences of integers
X_train_seq = tokenizer.texts_to_sequences(train_df['cleaned_headline'])  # For training data
X_eval_seq = tokenizer.texts_to_sequences(eval_df['cleaned_headline'])  # For evaluation data

# find the maximum sequence length
max_length = max(max(len(seq) for seq in X_train_seq), max(len(seq) for seq in X_eval_seq))

# pad sequences
X_train_pad = pad_sequences(X_train_seq, maxlen=max_length, padding='post')  # training data
X_eval_pad = pad_sequences(X_eval_seq, maxlen=max_length, padding='post')  # evaluation data

# a mapping of each word to its unique index
word_index = tokenizer.word_index
```

PART3 - TRAINING MODEL

Next, we add GloVe embeddings:

First, we used a function called “load_glove_embeddings” to get word vectors from GloVe files. This function reads each line of the GloVe file and creates a dictionary that links each word to its vector . We looked at Glove vectors of different sizes 50, 100, 200, and 300 to find the best fit.

Next, we used the “create_embedding_matrix” function to connect our data's words with their GloVe vectors. This step matches every word in our data to its GloVe vector in a special matrix. By doing this, we help our model understand words better, improving how it learns from text.

```python
def load_glove_embeddings(file_path):
    # a dictionary to store the word embeddings.
    embedding_index = {}
    with open(file_path, 'r', encoding='utf-8') as f:
        for line in f:
            # split each line into word and its corresponding vector
            values = line.split()
            word = values[0]  # word is first value
            # rest of values form the vector, converted to floats
            coefs = np.asarray(values[1:], dtype='float32')
            # store the word and its vector
            embedding_index[word] = coefs
    return embedding_index
# dimensions of GloVe vectors
glove_dimensions = [50, 100, 200, 300]
glove_paths = {
    50: '/content/drive/My Drive/glove.6B/glove.6B.50d.txt',
    100: '/content/drive/My Drive/glove.6B/glove.6B.100d.txt',
    200: '/content/drive/My Drive/glove.6B/glove.6B.200d.txt',
    300: '/content/drive/My Drive/glove.6B/glove.6B.300d.txt'
}
# func to create embedding matrix
def create_embedding_matrix(dim, word_index):
    # load GloVe embeddings for specified dimension.
    embedding_index = load_glove_embeddings(glove_paths[dim])
    embedding_matrix = np.zeros((len(word_index) + 1, dim))
    for word, i in word_index.items():
        # corresponding GloVe vector
        embedding_vector = embedding_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector
    return embedding_matrix
```

The “get_average_embedding” function simplifies handling text by converting words into numerical values using GloVe.

This method ensures texts of any length are made uniform by averaging the meanings of the words they contain. This creates a numerical representation that captures the text's main idea, making it easier for analysis tools to interpret.

```python
# func to calculate average embedding for each sequence
def get_average_embedding(sequences, embedding_matrix):

    embeddings = np.zeros((len(sequences), embedding_matrix.shape[1]))
    for i, seq in enumerate(sequences):
        # filter out zeros to find actual word indices
        non_zero_elements = [idx for idx in seq if idx != 0]
        if non_zero_elements:
            # retrieve and average the embeddings for the non zero elements
            seq_embeddings = np.array([embedding_matrix[idx] for idx in non_zero_elements])
            embeddings[i] = np.mean(seq_embeddings, axis=0)
    return embeddings
```

In evaluating a Logistic Regression model with GloVe embeddings across dimensions: 50, 100, 200, and 300 from the GloVe6B dataset, I uncovered how embedding depth affects text comprehension. This process involved creating an embedding matrix for each dimension, ensuring the model had access to comprehensive, pre-trained word representations.

The critical step was transforming text into averaged GloVe embeddings, turning variable-length texts into uniform vectors that capture their semantic essence. This conversion to a numerical format allowed the Logistic Regression model to analyze text more deeply and effectively.

After training the model on these embeddings, it learned to identify patterns that link detailed word representations to specific outcomes. This training was essential for increasing the accuracy of model prediction.

```
for dim in glove_dimensions:
    print(f"Evaluating model with GloVe {dim}d embeddings...")
    # creating  embedding matrix for current dim
    embedding_matrix = create_embedding_matrix(dim, word_index)
    # converting training and evaluation sequences to their average embeddings
    X_train_avg = get_average_embedding(X_train_pad, embedding_matrix)
    X_eval_avg = get_average_embedding(X_eval_pad, embedding_matrix)

    # initialize and train Logistic Regression model.
    model = LogisticRegression(max_iter=1000)
    model.fit(X_train_avg, y_train)

    # predict and evaluate the model
    y_pred = model.predict(X_eval_avg)


    print(f"Results for GloVe {dim}d embeddings:")
    print(f"F1 Score: {f1_score(y_eval, y_pred)}")
    print(f"Precision: {precision_score(y_eval, y_pred)}")
    print(f"Recall: {recall_score(y_eval, y_pred)}\n")
```

here are the results:

```
Evaluating model with GloVe 50d embeddings...
Results for GloVe 50d embeddings:
F1 Score: 0.6027554535017221
Precision: 0.6307569082899479
Recall: 0.5771344814950531

Evaluating model with GloVe 100d embeddings...
Results for GloVe 100d embeddings:
F1 Score: 0.6162347560975611
Precision: 0.6419213973799127
Recall: 0.5925247343349213

Evaluating model with GloVe 200d embeddings...
Results for GloVe 200d embeddings:
F1 Score: 0.6387818041634541
Precision: 0.6738511590077267
Recall: 0.6071821179919384

Evaluating model with GloVe 300d embeddings...
Results for GloVe 300d embeddings:
F1 Score: 0.6594427244582043
Precision: 0.6986469864698647
Recall: 0.6244045437889337
```

It's obvious as we increase the dimensions from 50 to 300, there's a marked improvement in the model's performance metrics (F1 Score, Precision and Recall).

Conclusion:

Looking into various sizes of GloVe embeddings shows using larger embeddings can really help models predict better from text.

When we use more detailed GloVe embeddings, from smaller 50d to larger 300d, our model gets better at understanding and sorting text. For example, moving from 50d to 100d not only improves the overall score but also gets better at identifying right answers and avoiding wrong ones. This shows us that more details help the model make smarter decisions.

The larger embeddings pack in more information about how words are used and how they relate to each other. This richness helps the Logistic Regression model get a fuller picture of the text, making it better at working with new information it hasn't seen before. Seeing the biggest improvement with the 300d embeddings highlights how much depth matters in understanding text.

The results hint that using the biggest size available, like the 300d embeddings, might give the best performance for tasks involving understanding text. But, it's also important to consider how complex the text is, what resources are available, and how critical accuracy is for the task.

Analyzing each GloVe embedding dimension on its own offers a unique perspective on how well the Logistic Regression model performs with textual data. Taking the 100d embeddings as an example, we see a model that reaches a balanced level of understanding text. With an F1 Score of 0.616, Precision of 0.642, and Recall of 0.593. This score reflects a good performance, suggesting the model is fairly reliable at distinguishing relevant text data without too many errors.

General Analysis for 100d Embeddings:

- F1 Score: The score indicates a good balance between precision and recall, which means the model is quite adept at identifying relevant instances without overly misclassifying.

- Precision: This level of precision suggests that when the model predicts a text belongs to a certain category, it's right a significant portion of the time.

- Recall: The recall rate shows the model can find a considerable number of all relevant instances, though there's room for improvement in capturing more without sacrificing precision.

so looking into different sizes of GloVe embeddings shows that bigger sizes can really improve how well models can predict based on text. But, it's important to balance the need for detail with how much computing power is available.

QUESTION3

PART1 -

I began by downloading the dataset into my Google Colab, followed by a series of text cleaning steps (as usual :)):

```
!wget https://sherlock-holm.es/stories/plain-text/advs.txt -O sherlock_holmes_stories.txt
```

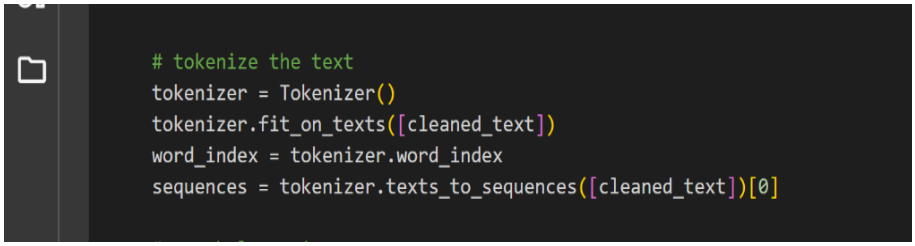
```
--2024-04-04 23:06:43--  https://sherlock-holm.es/stories/plain-text/advs.txt
Resolving sherlock-holm.es (sherlock-holm.es)... 157.90.249.21, 2a01:4f8:1c17:5725::1
Connecting to sherlock-holm.es (sherlock-holm.es)|157.90.249.21|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 610921 (597K) [text/plain]
Saving to: 'sherlock_holmes_stories.txt'

sherlock_holmes_sto 100%[===================>] 596.60K   757KB/s    in 0.8s

2024-04-04 23:06:45 (757 KB/s) - 'sherlock_holmes_stories.txt' saved [610921/610921]
```

```
import re
with open('sherlock_holmes_stories.txt', 'r', encoding='utf-8') as file:
    text = file.read()

text = text.lower()
text = re.sub(r'\[.*?\]', '', text)
text = re.sub(r'http\S+', '', text)
text = re.sub(r'[%s]' % re.escape("""!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~"""), '', text)
text = re.sub(r'\n', ' ', text)
text = re.sub(r'\w*\d\w*', '', text)

cleaned_text = text
```

Next - - - > SkipGram Model

SkipGram Model Implementation:

Preparing the Text Data: by using a “Tokenizer” we converted the text into a numerical format that our model can understand.

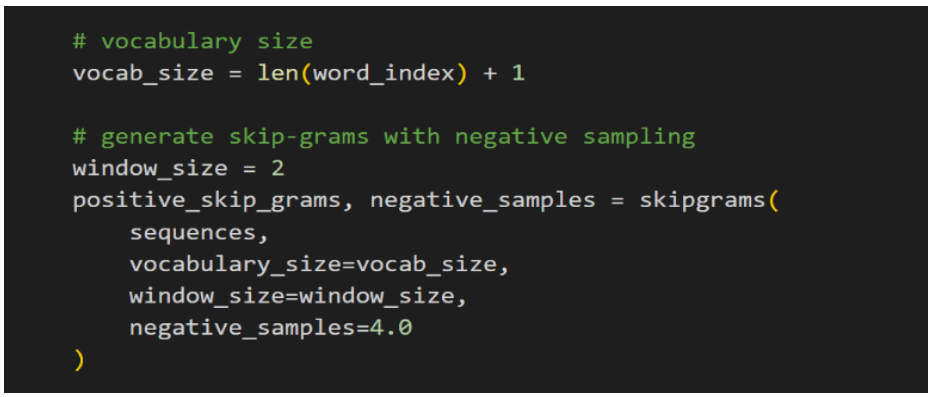


```
# tokenize the text
tokenizer = Tokenizer()
tokenizer.fit_on_texts([cleaned_text])
word_index = tokenizer.word_index
sequences = tokenizer.texts_to_sequences([cleaned_text])[0]
```

Generating Skip-grams with Negative Sampling

Next we generate skip-grams. A skip-gram is essentially a pair of words that appear close to each other in the text, which our model will use to learn the context of words

Negative sampling adds a twist by introducing random words that are not in the target word's context. This helps the model learn what context is not, which is just as important. We do this 4 times for each positive skip-gram to balance learning what context is and what it isn't.



```
# vocabulary size
vocab_size = len(word_index) + 1

# generate skip-grams with negative sampling
window_size = 2
positive_skip_grams, negative_samples = skipgrams(
    sequences,
    vocabulary_size=vocab_size,
    window_size=window_size,
    negative_samples=4.0
)
```

Building the SkipGram Model:

Next we build the SkipGram model, which has two main components:

1. Target Embedding Layer: This learns a dense vector for each target word.

2. Context Embedding Layer: This learns a dense vector for each context word.

When we feed a pair of words) into the model, it looks up these vectors and calculates a score that represents how well the context word fits with the target word. The higher the score, the better the fit.

```python
# SkipGram model
class SkipGram(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim):
        super(SkipGram, self).__init__()
        self.target_embedding = tf.keras.layers.Embedding(
            input_dim=vocab_size,
            output_dim=embedding_dim,
            input_length=1,
            name='w2v_embedding'
        )
        self.context_embedding = tf.keras.layers.Embedding(
            input_dim=vocab_size,
            output_dim=embedding_dim,
            input_length=1,
            name='context_embedding'
        )

    def call(self, pair):
        target, context = pair
        if len(target.shape) == 2:
            target = tf.squeeze(target, axis=-1)
        if len(context.shape) == 2:
            context = tf.squeeze(context, axis=-1)
        target_emb = self.target_embedding(target)
        context_emb = self.context_embedding(context)
        dots = tf.einsum('ik,ik->i', target_emb, context_emb)
        return dots
```

Training the Model:

With our model built, we then prepare our training data from the skip-grams we generated earlier. We feed these skip-grams into our model, which learns from them by adjusting the word vectors to maximize the score for matching pairs and minimize it for non-matching pairs.

```python
# compile the model
model = SkipGram(vocab_size, embedding_dim)
model.compile(optimizer='adam', loss=tf.keras.losses.BinaryCrossentropy(from_logits=True))

# prepare data for training

x_train = [np.array([pair[0] for pair in positive_skip_grams]),
           np.array([pair[1] for pair in positive_skip_grams])]
y_train = np.ones(len(positive_skip_grams))  # adjust labels based on positive and negative samples

# train the model
model.fit(x_train, y_train, epochs=1, batch_size=1024)
```

Extracting Word Embeddings:

After training, we have learned embeddings for each word in our vocabulary, which are dense vectors capturing the semantic meaning of words. By adding the target and context embeddings, we create a unified vector for each word, encapsulating its semantic essence.

```
# post-training
word_feature_vectors = model.target_embedding.get_weights()[0] + model.context_embedding.get_weights()[0]
```

the output here is:

```
2040/2040 [==============================] - 43s 21ms/step - loss: 0.1226
```

it means:

1. 2040/2040 indicates that the model processed all 2040 batches of data that were available for training. The first number is the current batch being processed, and the second number is the total number of batches. This means the training loop went through the entire dataset once.

2. 43s denotes that the entire training process for this epoch took 43 seconds to complete.

3. 21ms/step shows the average time taken to process each batch, which is 21 milliseconds.

4. loss: 0.1226 is a quantitative measure of how well the model's predictions match the actual labels. In the context of SkipGram and using binary cross-entropy, this value represents how effectively the model has learned to distinguish between positive and negative samples. A lower loss value suggests that the model is better at predicting the correct context of words.

PART2 -

In the second part of our exploration, we used the SkipGram model to investigate if our word embeddings could capture a well-known semantic relationship: "king" is to "man" as "queen" is to "woman." This was done through simple vector arithmetic and similarity measures.

Extracting Word Embeddings: First, we extracted embeddings for the words of interest from our trained model which gave us numerical vectors representing each word's semantic meaning as learned by the SkipGram model.

We then performed the following operation to theoretically derive the vector for "queen":

resultant_vec = king_vec - man_vec + woman_vec

finally to see how close our resultant vector was to the actual "queen" embedding, we used two measures:

- Dot Product Similarity: First, we calculated a basic similarity score with a dot product, which gave us a raw score indicating the vectors' directional alignment.

- Cosine Similarity: For a more normalized and interpretable measure, we calculated the cosine similarity.

The similarity score of approximately 3.9 indicates that the vector resulting from the operation "king" - "man" + "woman" is quite similar to the "queen" vector in our model's embedding space. This suggests that the embeddings have captured a meaningful semantic relationship that aligns with human understanding of these words' relationships.

also a cosine similarity score of approximately 0.962 is remarkably high, suggesting that our operation successfully moved the "king" vector close to "queen" in the model's semantic space.

```
def get_word_embedding(word, model, word_index):
    # retrieve the words index
    word_idx = word_index[word]
    # extract the embedding for the word
    return model.target_embedding(np.array([word_idx])).numpy()

# extract embeddings
queen_vec = get_word_embedding('queen', model, word_index)
king_vec = get_word_embedding('king', model, word_index)
man_vec = get_word_embedding('man', model, word_index)
woman_vec = get_word_embedding('woman', model, word_index)

# vector arithmetic
resultant_vec = king_vec - man_vec + woman_vec

# calculate similarity
similarity = np.dot(queen_vec.flatten(), resultant_vec.flatten())

print(f"Similarity: {similarity}")
```

```
Similarity: 3.913191795349121
```

```
[28] # compute cosine similarity
     cosine_similarity = np.dot(queen_vec.flatten(), resultant_vec.flatten()) / (np.linalg.norm(queen_vec) * np.linalg.norm(resultant_vec))
     print(f"Cosine Similarity: {cosine_similarity}")
```

```
Cosine Similarity: 0.9622769951820374
```

PART3 -

In Part 3, we visualized how our SkipGram model thinks about words by turning them into points on a simple graph. We looked words like "brother," "sister," "uncle," and "aunt" to see how the model sees the difference between male and female family members.

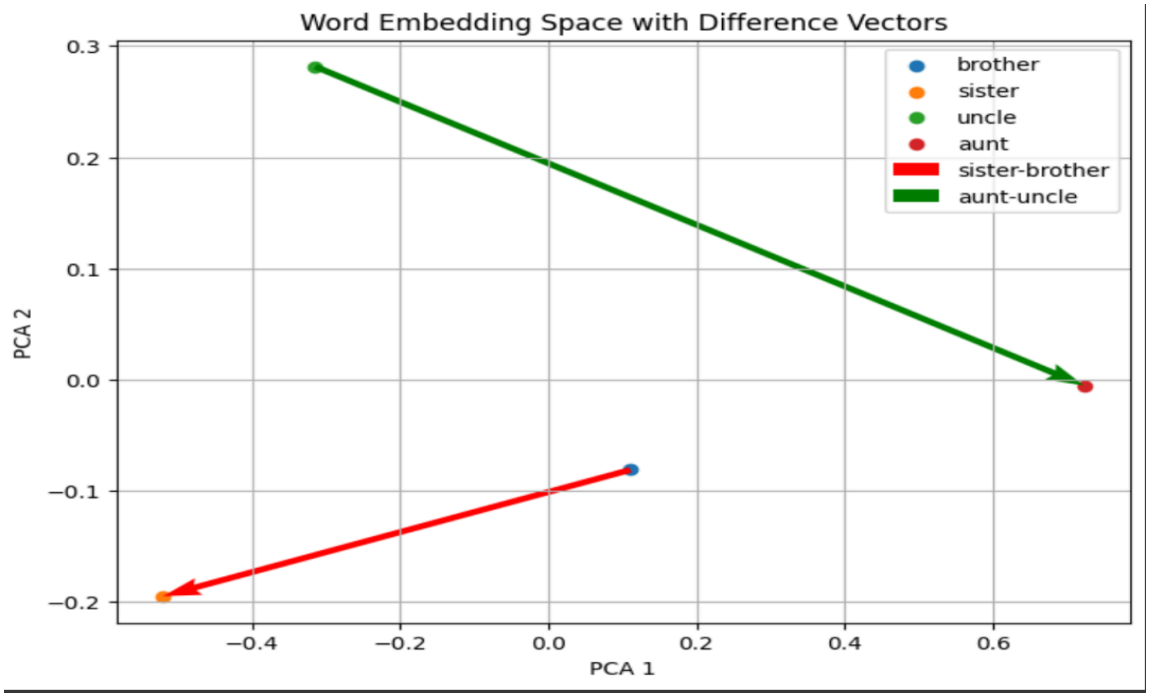
To do this, we used a special technique called PCA to squish the complex word maps from the model into a flat, easy to look at 2D picture. Then, we drew arrows to show the jump from male words to female words, like from "brother" to "sister."

When we looked at the picture, the arrows helped us see how much the model has to "move" to go from thinking about a "brother" to a "sister" or from an "uncle" to an "aunt." The way these arrows pointed and how long they were told us that the model has a pretty good idea of the difference between the words for male and female family members.

So, what we did was like drawing a family tree, but instead of lines, we used arrows on a flat map to show how these family words are connected in the model's mind.

```python
words = ['brother', 'sister', 'uncle', 'aunt']
embeddings = np.array([get_word_embedding(word, model, word_index).flatten() for word in words])

# apply PCA to reduce to 2 dimensions
pca = PCA(n_components=2)
reduced_embeddings = pca.fit_transform(embeddings)

# compute difference vectors
diff_vectors = np.array([reduced_embeddings[1] - reduced_embeddings[0],  # sister - brother
                         reduced_embeddings[3] - reduced_embeddings[2]]) # aunt - uncle


plt.figure(figsize=(8, 6))

# plot original word embeddings
for i, word in enumerate(words):
    plt.scatter(reduced_embeddings[i, 0], reduced_embeddings[i, 1], label=word)

# plot difference vectors
plt.quiver(*reduced_embeddings[0], *diff_vectors[0], color='r', scale=1, scale_units='xy', angles='xy', label='sister-brother')
plt.quiver(*reduced_embeddings[2], *diff_vectors[1], color='g', scale=1, scale_units='xy', angles='xy', label='aunt-uncle')

plt.legend()
plt.title('Word Embedding Space with Difference Vectors')
plt.xlabel('PCA 1')
plt.ylabel('PCA 2')
plt.grid(True)
plt.show()
```

the result:



PCA 1 (x-axis):This axis represents the direction of maximum variance in the dataset.

PCA 2 (y-axis): This axis represents the second most significant direction of variance.

From the provided visualization of word embeddings and their difference vectors, we can observe several things:

1. Relative Positioning: The points for "brother" and "sister" are distinct from "uncle" and "aunt," indicating that the model recognizes these as different pairs of words within the embedding space.

2. Difference Vectors: The arrows represent the difference vectors

3. Gender Relationship: If the arrows point in roughly the same direction and have similar lengths, it implies that the model encodes the gender relationship between these pairs in a consistent way.

4. PCA Interpretation: PCA has reduced the high-dimensional embeddings down to 2 dimensions for visualization.

From the plot, we can't directly read off the PCA values, but we can infer that the relative positions and directions of the vectors carry semantic meaning. The vectors suggest that the model sees "sister" as quite distinct from "brother" and "aunt" as quite distinct from "uncle".