Deep Learning Lab #4

Emily Bates, xzz320

Task 1a: Implement equation 1 in the `cosine_similarity()` function below.
Hint: check out the numpy documentation on np.dot, np.sum, and np.sqrt. Depending on how you choose to implement it, you can check out np.linalg.norm.

```
def cosine_similarity(vector1, vector2):
    """
    Calculates the cosine similarity of two word vectors - vector1 and
vector2
    Arguments:
        vector1 (ndarray): A word vector having shape (n,)
        vector2 (ndarray): A word vector having shape (n,)
    Returns:
        cosine_similarity (float): The cosine similarity between vector1
and vector2
    """

    # Start Code Here #
    # Compute the dot product between vector1 and vector2 (~ 1 line)
    dot = np.dot(vector1, vector2)

    # Compute the Euclidean norm or length of vector1 (~ 1 line)
    norm_vector1 = np.linalg.norm(vector1)

    # Compute the Euclidean norm or length of vector2 (~ 1 line)
    norm_vector2 = np.linalg.norm(vector2)

    # Compute the cosine similarity as defined in equation 1 (~ 1 line)
    cosine_similarity = dot / (norm_vector1 * norm_vector2)
    # End Code Here #

    return cosine_similarity
```

```
Cosine similarity between man and woman: 0.886033771849582
Cosine similarity between cat and dog: 0.9218005273769252
Cosine similarity between cat and cow: 0.40695688711826294
Cosine similarity between england - london and edinburgh - scotland: -
0.5203389719861108
```

Task 1b: In the code cell below, try out 3 of your own inputs here and report your inputs and outputs

```
# Start code here #
strawberry = word_to_vector_map["strawberry"]
apple = word_to_vector_map["apple"]
aunt = word_to_vector_map["aunt"]
```

```python
uncle = word_to_vector_map["uncle"]
coffee = word_to_vector_map["coffee"]
tea = word_to_vector_map["tea"]

print(f"Cosine similarity between strawberry and apple:
{cosine_similarity(strawberry,apple)}")
print(f"Cosine similarity between aunt and uncle:
{cosine_similarity(aunt,uncle)}")
print(f"Cosine similarity between coffee and tea:
{cosine_similarity(coffee,tea)}")
# End code here #
```

```
Cosine similarity between strawberry and apple: 0.44254816179185213
Cosine similarity between aunt and uncle: 0.7631033687184533
Cosine similarity between coffee and tea: 0.8079648365112425
```

As you can see, strawberry & apple have relatively low similarity, while aunt & uncle and coffee & tea have much higher similarity.

Task 2a: To perform word analogies, implement `answer_analogy()` below.

```python
def answer_analogy(word_i, word_j, word_k, word_to_vector_map):
    """
    Performs word analogy as described above
    Arguments:
        word_i (String): A word
        word_j (String): A word
        word_k (String): A word
        word_to_vector_map (Dict): A dictionary of words as key and its
associated embedding vector as value
    Returns:
        best_word (String): A word that fufils the relationship that e_j -
e_i as close as possible to e_l - e_k, as measured by cosine similarity
    """

    # Convert words to lowercase
    word_i = word_i.lower()
    word_j = word_j.lower()
    word_k = word_k.lower()

    # Start code here #
    try:
        # Get the embedding vectors of word_i (~ 1 line)
        embedding_vector_of_word_i = word_to_vector_map[word_i]
    except KeyError:
```

```
        print(f"{word_i} is not in our vocabulary. Please try a different 
word.")
        return

    try:
        # Get the embedding vectors of word_j (~ 1 line)
        embedding_vector_of_word_j = word_to_vector_map[word_j]
    except KeyError:
        print(f"{word_j} is not in our vocabulary. Please try a different 
word.")
        return

    try:
        # Get the embedding vectors of word_k (~ 1 line)
        embedding_vector_of_word_k = word_to_vector_map[word_k]
    except KeyError:
        print(f"{word_k} is not in our vocabulary. Please try a different 
word.")
        return
    # End code here #

    # Get all the words in our word to vector map i.e our vocabulary
    words = word_to_vector_map.keys()
    max_cosine_similarity = -1000                           # Initialize 
to a large negative number
    best_word = None                                        # Note: Do not 
change this None. Keeps track of the word that best answers the analogy.

    # Since we are looping through the whole vocabulary, if we encounter a 
word
    # that is the same as our input, that word becomes the best_word. To 
avoid
    # that we skip the input word.
    input_words = set([word_i, word_j, word_k])

    for word in words:
        if word in input_words:
            continue

        # Start code here #

        # Compute cosine similarity  (~ 1 line)
        similarity = cosine_similarity(embedding_vector_of_word_j - 
embedding_vector_of_word_i,
```

```
                                                 word_to_vector_map[word] -
embedding_vector_of_word_k)

        # Have we seen a cosine similarity bigger than
max_cosine_similarity?
            # then update the max_cosine_similarity to the current cosine
similarity
            # and update the best_word to the current word (~ 3 lines)
        if similarity > max_cosine_similarity:
            max_cosine_similarity = similarity
            best_word = word
        # End code here

    return best_word
```

Task 2b: Test your implementation by running the code cell below. What are your observations? What do you observe about the last two outputs?

```
france -> french :: germany -> german
england -> london :: japan -> tokyo
boy -> girl :: man -> woman
man -> doctor :: woman -> nurse
small -> smaller :: big -> competitors
```

The first two analogies hold up, but the second two analogies are not accurate. The first inaccurate example of *man -> doctor :: woman -> nurse* shows bias in word embeddings. This perpetuates dated gender stereotypes. The second inaccurate example of *small -> smaller :: big -> competitor* shows a gap in the model's understanding. Here we would expect the result to be *bigger* not *competitor*. It did not return the intended analogy.

Task 2c: Try your own analogies by completing and executing the code cell below. Find 2 that works and one that doesn't. Report your inputs and outputs

```
my_analogies = [('north', 'south', 'east'), ('cereal', 'breakfast',
'pasta'), ('morning', 'a.m.', 'afternoon'), ('north', 'up', 'south')]
for analogy in my_analogies:
    best_word = answer_analogy(*analogy, word_to_vector_map)
    print(f"{analogy[0]} -> {analogy[1]} :: {analogy[2]} -> {best_word}")
```

```
north -> south :: east -> africa
cereal -> breakfast :: pasta -> dinner
morning -> a.m. :: afternoon -> p.m.
north -> up :: south -> down
```

The first example shown is incorrect. I was expecting *north -> south :: east -> west*. Instead, *africa* was returned instead of *west*. The other three are successful analogies.

Task 3a: Complete the `get_occupation_stereotypes()` below.

```
def get_occupation_stereotypes(she, he, occupations_file,
word_to_vector_map, verbose=False):
    """
    Computes the words that are closest to she and he in the GloVe
embeddings
    Arguments:
        she (String): A word
        he (String): A word
        occupations_file (String): The path to the occupation file
        word_to_vector_map (Dict): A dictionary mapping words to embedding
vectors
    Returns:
        most_similar_words (Tuple(List[Tuple(Float, String)],
List[Tuple(Float, String)])):
        A tuple of the list of the most similar occupation words to she
and he with their associated similarity
    """

    # Read occupations
    with open(occupations_file, 'r') as file_handle:
        occupations = json.load(file_handle)

    # Extract occupation words
    occupation_words = [occupation[0] for occupation in occupations]

    # Start code here #
    # Get embedding vector of she (~ 1 line)
    embedding_vector_she = word_to_vector_map[she]
    # Get embedding vector of he (~ 1 line)
    embedding_vector_he = word_to_vector_map[he]
    # Get the vector difference between embedding vectors of she and he (~
1 line)
    vector_difference_she_he = embedding_vector_she - embedding_vector_he
    # Get the normalized difference (~ 1 line)
    normalized_difference_she_he = vector_difference_she_he /
np.linalg.norm(vector_difference_she_he)
    # End code here #

    # Store the cosine similarities
    similarities = []

    for word in occupation_words:
        # Start code here #
```

```
        try:
            # Get the embedding vector of the current occupation word (~ 1 line)
            occupation_word_embedding_vector = word_to_vector_map[word]
            # Compute cosine similarity between embedding vector of the occupation word and normalized she - he vector (~ 1 line)
            similarity = cosine_similarity(occupation_word_embedding_vector,normalized_difference_she_he)
            similarities.append((similarity, word))
        except KeyError:
            if verbose:
                print(f"{word} is not in our vocabulary.")
        # End code here #

    most_similar_words = sorted(similarities)

    return most_similar_words[:20], most_similar_words[-20:]
```

Task 3b: Execute the cell below and report your results.

1) Does the GloVe word embeddings propagate bias? why?

2) From the list associated with she, list those that reflect gender stereotype.

3) Compare your list from 2 to the occupations closest to he. What are your conclusions?

Exclude businesswoman from your list.

```
Occupations closest to he:
(-0.3562123840342885, 'coach')
(-0.3369460967074499, 'caretaker')
(-0.316340865049864, 'captain')
(-0.3092732402211717, 'marshal')
(-0.3072913571995383, 'colonel')
(-0.30248716941585896, 'skipper')
(-0.30214385530420096, 'manager')
(-0.3016537316721372, 'midfielder')
(-0.29967111777167377, 'archbishop')
(-0.2944306421611404, 'commander')
(-0.29028676593363584, 'footballer')
(-0.288898501862417l, 'bishop')
(-0.2819996009929522, 'marksman')
(-0.27963882421385033, 'firebrand')
(-0.27878757562089745, 'provost')
(-0.27807933185669276, 'substitute')
(-0.272179724168074, 'lieutenant')
(-0.2719793191353711, 'custodian')
(-0.27191912253031186, 'superintendent')
```

```
(-0.2713465113055802, 'goalkeeper')

Occupations closest to she:
(0.3207273716226149, 'singer')
(0.3219568449876505, 'publicist')
(0.34405199459208385, 'nanny')
(0.34466952852982574, 'therapist')
(0.347465914896008, 'confesses')
(0.3557761074297246, 'businesswoman')
(0.3573048226883875, 'dancer')
(0.3645661876706785, 'hairdresser')
(0.3698354241676874, 'receptionist')
(0.3729106376588048, 'housekeeper')
(0.3730974791641408, 'homemaker')
(0.3812397049469024, 'housewife')
(0.38133986034619594, 'nurse')
(0.38926460609618757, 'narrator')
(0.41383366509680375, 'maid')
(0.42853139796990297, 'socialite')
(0.4434377286463443, 'waitress')
(0.4473291387166749, 'stylist')
(0.46861385366142183, 'ballerina')
(0.49840294304026306, 'actress')
```

1) Yes, the GloVe word embeddings propagate bias. We can see that occupations associated with “he” tend to be more traditionally male-dominated roles (like commander), while occupations associated with “she” tend to be more traditionally female-dominated roles (like nanny).
2) From the list associated with ‘she’, the following reflect gender stereotypes: nanny, therapist, dancer, hairdresser, receptionist, housekeeper, homemaker, nurse, stylist. However, I feel like an argument can be made for most, if not all, words on the list to involve gender stereotypes.
3) Comparing the occupations closest to "he" and "she", we see that the occupations associated with "she" reflect more traditional gender roles and stereotypes, such as caregiving, domestic work, and entertainment, while those associated with "he" include roles traditionally seen as more authoritative, leadership-oriented, or physically demanding. This reflects societal biases and stereotypes regarding gender roles and occupations, which are being perpetuated here by the underlying biases present in the GloVe word embeddings.

Task 4a: Run the cell below to computes the similarity between the gender embedding and the embedding vectors of male and female names. What can you observe?

```
Names and their similarities with simple gender subspace
mary 0.3457399102816379
john -0.17879783833420468
sweta 0.17016456601128147
david -0.1332261560078667
kazim -0.32658964009764835
```

```
angela 0.2600799146632235

Names and their similarities with PCA based gender subspace
mary 0.2637091204419718
john -0.3816839789078354
sweta 0.1773704777691709
david -0.3165647635266187
kazim -0.3249838182709315
angela 0.18623308926276097
```

With the simple gender subspace, the similarity scores between the gender embedding and the embedding vectors of male and female names vary. For example:

- Mary: positive similarity score, indicating a closer association with the female gender.
- John: negative similarity score, indicating a closer association with the male gender.
- Sweta, Angela: positive similarity scores, but lower than Mary, suggesting some association with the female gender, though weaker.
- David and Kazim: negative similarity scores, indicating a closer association with the male gender.

With the PCA-based gender subspace, the similarity scores between the gender embedding and the embedding vectors of male and female names also vary, but the magnitudes and directions of the scores differ from the simple gender subspace:

- Mary still has a positive similarity score, but it's lower than in the simple gender subspace, indicating a weaker association with the female gender.
- John's negative similarity score is even more negative, suggesting a stronger association with the male gender.
- Other names like Sweta, David, and Kazim show similar trends as in the simple gender subspace, but the magnitudes of their similarity scores differ slightly.
- Angela's positive similarity score is also lower compared to the simple gender subspace.

Overall, these observations suggest that the choice of method for identifying the gender subspace (simple vs. PCA-based) can affect the similarity scores between gender-specific names and the gender embedding, leading to variations in the strength and direction of the associations.

Task 4b: Quantify direct and indirect biases between words and the gender embedding by running the following cell. What is your observation?

```
engineer -0.2626286258749398
science -0.1202780958734538
pilot -0.1319833052724868
technology -0.1801116607819377
lipstick 0.4179404715417419
arts -0.04513818522820779
singer 0.16162975755073875
computer -0.16390549337211754
```

```
receptionist 0.3305284235998437
fashion 0.06913524872078784
doctor 0.02885191966409418
literature -0.08972688088254833
```

Lipstick, receptionist, and singer have strong positive similarity scores, indicating bias toward the female gender. Engineer, science, pilot, technology, and computer all have strong negative similarity scores, indicating bias toward the male gender. The other words have weaker bias.

Task 4c: Implement `neutralize()` below by implementing the formulas above. Hint see [np.sum]

```python
def neutralize(word, gender_direction, word_to_vector_map):
    """
    Project the vector of word onto the gender subspace to remove the bias of "word"
    Arguments:
        word (String): A word to debias
        gender_direction (ndarray): Numpy array of shape (embedding size (50), ) which is the bias axis
        word_to_vector_map (Dict): A dictionary mapping words to embedding vectors

    Returns:
        debiased_word (ndarray): the vector representation of the neutralized input word
    """

    # Start code here #
    # Get the vector representation of word (~ 1 line)
    embedding_of_word = word_to_vector_map[word]

    # Compute the projection of word onto gender direction. e.q. 3 (~ 1 line)
    projection_of_word_onto_gender = np.dot(embedding_of_word, gender_direction) / np.sum(gender_direction ** 2) * gender_direction

    # Neutralize word e.q 4 (~ 1 line)
    debiased_word  = embedding_of_word - projection_of_word_onto_gender
    # End code here #

    return debiased_word
```

Task 4d: Test your implementation by running the code cell below. What is your observation?

```
Before neutralization, cosine similarity between babysit and gender is:
0.2663444879209918
After neutralization, cosine similarity between babysit and gender is: -
1.3389570015765782e-17
```

Before neutralization, the cosine similarity between "babysit" and the gender embedding is approximately 0.266, indicating some level of association between "babysit" and female gender. After neutralization, the cosine similarity between "babysit" and gender is approximately -1.339e-17, which is very close to zero. This indicates that the association between "babysit" and gender has been effectively removed through neutralization, making “babysit” a gender neutral word.

Task 5a: Implement `equalization()` below by implementing the formulas above.

```
def equalization(equality_set, bias_direction, word_to_vector_map):
    """
    Equalize the pair of gender specific words in the equality set
ensuring that
    any neutral word is equidistant to all words in the equality set.
    Arguments:
        equality_set (Tuple(String, String)): a tuple of strings of gender
specific
        words to debias e.g ("grandmother", "grandfather")
        bias_direction (ndarray): numpy array of shape (embedding
dimension,). The
        embedding vector representing the bias direction
        word_to_vector_map (Dict):  A dictionary mapping words to
embedding vectors
    Returns:
        embedding_word_a (ndarray): numpy array of shape (embedding
dimension,). The
        embedding vector representing the first word
        embedding_word_b (ndarray): numpy array of shape (embedding
dimension,). The
        embedding vector representing the second word
    """

    # Start code here #
    # Get the vector representation of word pair by unpacking equality_set
(~ 3 line)
    word_a, word_b = equality_set
    embedding_word_a = word_to_vector_map[word_a.lower()]
    embedding_word_b = word_to_vector_map[word_b.lower()]

    # Compute the mean (eq. 5) of embedding_word_a and embedding_word_a (~
1 line)
```

```
    mean = (embedding_word_a + embedding_word_b) / 2

    # Compute the projection of mean representation onto the bias
direction (eq. 6) (~ 1 line)
    mean_B = np.dot(mean, bias_direction) / np.sum(bias_direction ** 2) *
bias_direction


    # Compute the projection onto the orthogonal subspace (eq. 7) (~ 1
line)
    mean_orthogonal = mean - mean_B

    # Compute the projection of th embedding of word a onto the bias
direction (eq. 8) (~ 1 line)
    embedding_word_a_on_bias_direction = np.dot(embedding_word_a,
bias_direction) / np.sum(bias_direction ** 2) * bias_direction

    # Compute the projection of th embedding of word b onto the bias
direction (eq. 9) (~ 1 line)
    embedding_word_b_on_bias_direction = np.dot(embedding_word_b,
bias_direction) / np.sum(bias_direction ** 2) * bias_direction

    # Re-embed embedding of word a using eq. 10 (~ 1 long line)
    new_embedding_word_a_on_bias_direction = np.sqrt(abs(1 -
np.sum(mean_orthogonal ** 2))) * (embedding_word_a_on_bias_direction -
mean_B) / np.linalg.norm(embedding_word_a - mean_orthogonal - mean_B)



    # Re-embed embedding of word b using eq. 11 (~ 1 long line)
    new_embedding_word_b_on_bias_direction = np.sqrt(abs(1 -
np.sum(mean_orthogonal ** 2))) * (embedding_word_b_on_bias_direction -
mean_B) / np.linalg.norm(embedding_word_b - mean_orthogonal - mean_B)

    # Equalize embedding of word a using eq. 12 (~ 1 line)
    embedding_word_a = mean_orthogonal +
new_embedding_word_a_on_bias_direction

    # Equalize embedding of word b using eq. 13 (~ 1 line)
    embedding_word_b = mean_orthogonal +
new_embedding_word_b_on_bias_direction

    # End code here #

    return embedding_word_a, embedding_word_b
```

Task 5b: Test your implementation by running the cell below.

```
print("Cosine similarity before equalization:")
print(f"(embedding vector of father, gender_direction):
{cosine_similarity(word_to_vector_map['father'], gender_direction)}")
print(f"(embedding vector of mother, gender_direction):
{cosine_similarity(word_to_vector_map['mother'], gender_direction)}")
print()

embedding_word_a, embedding_word_b  = equalization(("father", "mother"),
gender_direction, word_to_vector_map)
print("Cosine similarity after equalization:")
print(f"(embedding vector of father, gender_direction):
{cosine_similarity(embedding_word_a, gender_direction)}")
print(f"(embedding vector of mother, gender_direction):
{cosine_similarity(embedding_word_b, gender_direction)}")
```

```
Cosine similarity before equalization:
(embedding vector of father, gender_direction): -0.08502503175882657
(embedding vector of mother, gender_direction): 0.3332593015356538

Cosine similarity after equalization:
(embedding vector of father, gender_direction): -0.6639863783612923
(embedding vector of mother, gender_direction): 0.6639863783612926
```

Task 5c: Looking at the output of your implementation test above, what can you observe?

Before equalization, the cosine similarity between the embedding vector for "father" and gender direction is -0.085, while the cosine similarity between the embedding vector of "mother" and gender direction is 0.333. This shows that both "father" and "mother" have some bias, but "mother" has a stronger association with female gender compared to "father" and male gender. After equalization, the cosine similarity between the embedding vector of "father" and the gender direction (male) becomes -0.664, while the cosine similarity between the embedding vector of "mother" and the gender direction (female) becomes 0.664. These values indicate that after equalization, both "father" and "mother" are now equally distant, meaning that the gender bias present in both "father" and "mother" embeddings has been effectively mitigated through equalization. The goal has been met of ensuring that any neutral word is equidistant to all words in the set.