DA6233ECruz--2023 / IS6733Lab4COLABECruzkml614.ipynb

ecruz0369 COLA and ARC files Python LAB4

fafd123 · 2 minutes ago History

Debiasing word embeddings

Word embedding are word vectors that have meaning, word vectors similar to each other will be close to each other in a vector space.

After completing this lab you will be to:

- Use and load pre-trained word vectors
- Measure similarity of word vectors using cosine similarity
- Solve word analogy probelms such as Man is to Woman as Boy is to ____ using word embeddings
- Reduce gender bias in word embeddings by modifying word embeddings to remove gender stereotypes, such as the association between the words *receptionist* and *female*

Word embeddings

Word embedding is a method used to represent words as vectors. They are populary used in machine learning and natural language processing tasks. Despite their success in downstream tasks such as cyberbullying, sentiment analysis, and question retrieval, they exhibit gender sterotypes which raises concerns because their widespread use can amplify these biases.

Word embeddings are trained on word co-occurance using a text dataset. After training, each word w will be represented as a d-dimensional word vector $\vec{w} \epsilon \mathbb{R}^d$.

Word embedding properties:

- Words with similar semantic meaning will be close to each other
- The difference between word embedding vectors can represent relationships between words. For example, given the analogy "man is to King as woman is to x" (denoted as $man : king :: woman : x$), by doing simple arithmetic on the embedding vectors, we find that $x = queen$ is the best answer because $\vec{man} - \vec{woman} \approx \vec{king} - \vec{queen}$. For the analogy $Paris : France :: Nairobi : x$, finds that $x = Kenya$. These embeddings can also amplify sexism implicit in text. For instance, $\vec{man} - \vec{woman} \approx computer \vec{programmer} - \vec{homemaker}$. The same system that produced reasonable answers to the previous examples offensively answers "man is to computer programmer as woman is to x" with $x = homemaker$.

Run the following cell to load the required modules.

In [101…

```
import os
import json
import numpy as np
from pathlib import Path
from sklearn.decomposition import PCA
```

Download and Load word vectors

Due to the computational resources required to train word embeddings, we will be using a pre-trained 50-dimensional word embeddings, GloVe to represent words.

Run the following cells to download and load the word embeddings.

In [102…

```
def download_glove_vectors():
    '''
    Download the GloVe vectors
    Arguments:
        None
    Returns:
        file_name (String): The absolute path of the downloaded 50-dimensional
        GloVe word vector representations
    '''

    if not Path('data').is_dir():
        print("Downloading the embeddings ...")
        !wget --quiet https://nlp.stanford.edu/data/glove.6B.zip
        print("Embeddings downloaded.")

        # Unzip it
        print("Unzipping the downloaded file ...")
        !unzip -q glove.6B.zip -d data/
        print("File unzipped.")

    return '/content/data/glove.6B.50d.txt'
```

In [103…

```
def get_glove_vectors(glove_file):
    '''
    Read the word vectors in glove_file
    Arguments:
        glove_file (String): The absolute path to the downloaded glove word embeddings
    Returns:
        words (Set): The words (vocabulary) in the pretrained glove word embeddings
        word_to_vector_map (Dict): A dictionary mapping the each word to its embedding vector
    '''

    words = set()
    word_to_vector_map = {}
    with open(glove_file, 'r') as file_handle:
        for line in file_handle:
            line = line.strip().split()
```

```
            current_word = line[0]
            words.add(current_word)
            current_word_vector = line[1:]
            word_to_vector_map[current_word] = np.array(current_word_vector, dtype=np.float64)

    return words, word_to_vector_map
```

In [104…

```
# Load sets of words in the vocabulary and a dictionary mapping words to their GloVe vectors
words, word_to_vector_map = get_glove_vectors(download_glove_vectors())
```

Operations on word embeddings

Task 1 - Cosine similarity

Similarity between two words represented as word vectors u and v can be measured by their cosine similarity:

$$\text{CosineSimilarity(u, v)} = \frac{u \cdot v}{||u||_2||v||_2} = cos(\theta) \tag{1}$$

Where:

$u \cdot v$ is the dot (inner) product of the two vectors

$||u||_2$ is the length of the vector u. The length also called Euclidean length or Euclidean norm defines a distance function defined as $||u||_2 = \sqrt{u_1^2 \,+\, \ldots \,+\, u_n^2}$

The normalized similarity between u and v is the cosine of the angle between the two vectors denoted as θ. The cosine similarity of u and v will be close to 1 if the two vectors are similar, otherwise, the cosine similarity will be small.

Note: We will be refering to the embedding of a word i.e the word vector and the word interchangeably in this lab.

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.

In [105…

```
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:
```

```
    Returns:
        cosine_similarity (float): The cosine similarity between vector1 and vector2
    """

    # Compute the dot product between vector1 and vector2
    dot = np.dot(vector1, vector2)

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

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

    # Compute the cosine similarity as defined in equation 1
    cosine_similarity = dot / (norm_vector1 * norm_vector2)

    return cosine_similarity
```

In [106…

```
# Run this cell to obtain and report your answers
man = word_to_vector_map["man"]
woman = word_to_vector_map["woman"]
cat = word_to_vector_map["cat"]
dog = word_to_vector_map["dog"]
orange = word_to_vector_map["orange"]
england = word_to_vector_map["england"]
london = word_to_vector_map["london"]
edinburgh = word_to_vector_map["edinburgh"]
scotland = word_to_vector_map["scotland"]

print(f"Cosine similarity between man and woman: {cosine_similarity(man, woman)}")
print(f"Cosine similarity between cat and dog: {cosine_similarity(cat, dog)}")
print(f"Cosine similarity between cat and cow: {cosine_similarity(cat, orange)}")
print(f"Cosine similarity between england - london and edinburgh - scotland: {cosine_similarity(england - london, edinburgh -
```

```
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

In [107…

```
# Run this cell to obtain and report your answers
king = word_to_vector_map["king"]
queen = word_to_vector_map["queen"]
tiger = word_to_vector_map["tiger"]
lion = word_to_vector_map["lion"]
apple = word_to_vector_map["apple"]
france = word_to_vector_map["france"]
```

```python
paris = word_to_vector_map["paris"]
berlin = word_to_vector_map["berlin"]
germany = word_to_vector_map["germany"]

print(f"Cosine similarity between king and queen: {cosine_similarity(king, queen)}")
print(f"Cosine similarity between tiger and lion: {cosine_similarity(tiger, lion)}")
print(f"Cosine similarity between tiger and apple: {cosine_similarity(tiger, apple)}")
print(f"Cosine similarity between france - paris and berlin - germany: {cosine_similarity(france - paris, berlin - germany)}'
```

```
Cosine similarity between king and queen: 0.7839043010964117
Cosine similarity between tiger and lion: 0.5493991225945609
Cosine similarity between tiger and apple: 0.36045373632604055
Cosine similarity between france - paris and berlin - germany: -0.795694872209922
```

Task 2 - Word analogy

In an analogy task, you are given an analogy in the form "i is to j as k is to __". Your task is to complete this sentence.

For example, if you are given "man is to king as woman is to l" (denoted as $man : king :: woman : l$). You are to find the best word l that answers the analogy the best. Simple arithmetic of the embedding vectors will find that $l = queen$ is the best answer because the embedding vectors of words i, j, k, and l denoted as e_i, e_j, e_k, e_l have the following relationship:

$$e_j - e_i \approx e_l - e_k$$

Cosine similarity can be used to measure the similarity between $e_j - e_i$ and $e_l - e_k$

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

In [108…

```python
# Fixed the issue this code now works Nov 14, 2024 ECruz

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 measurec
    """

    # Convert words to lowercase
    word_i = word_i.lower()
```

```python
    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] # Fetch embedding from word_to_vector_map
    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] # Fetch embedding from word_to_vector_map
    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] # Fetch embedding from word_to_vector_map
    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 bes

    # 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)
        # Calculate cosine similarity using embedding vectors
        embedding_vector_of_word_l = word_to_vector_map[word]
        a = embedding_vector_of_word_j - embedding_vector_of_word_i
        b = embedding_vector_of_word_l - embedding_vector_of_word_k
        similarity = np.dot
```

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

In [109…

```
analogies = [('france', 'french', 'germany'),
             ('england', 'london', 'japan'),
             ('boy', 'girl', 'man'),
             ('man', 'doctor', 'woman'),
             ('small', 'smaller', 'big')]
for analogy in analogies:
    best_word = answer_analogy(*analogy, word_to_vector_map)
    if best_word:
        print(f"{analogy[0]} -> {analogy[1]} :: {analogy[2]} -> {best_word}")
```

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

In [110…

```
# Define your own analogies here
my_analogies = [
    ('king', 'queen', 'prince'),
    ('paris', 'france', 'nice'),
    ('car', 'road', 'sign')
]

# Execute the analogies
for analogy in my_analogies:
    best_word = answer_analogy(*analogy, word_to_vector_map)
    print(f"{analogy[0]} -> {analogy[1]} :: {analogy[2]} -> {best_word}")
```

```
king -> queen :: prince -> None
paris -> france :: nice -> None
car -> road :: sign -> None
```

Task 3 - Geometry of Gender and Bias in Word Embeddings: Occupational stereotypes

In this task, we will understand the biases present in word-embedding i.e which words are closer to she than to he. This will be achieved by evaluating whether the GloVe embeddings have sterotypes on occupation words. Determine gender bias by projecting each of the occupations onto the $she - he$ direction by computing the dot product between each occupation word embedding and the embedding vector of $she - he$ normalized by the Euclidean norm (See task 1).

$$occupation\_word_i \cdot ||she - he||_2 \tag{2}$$

Notice that equation 2 is similar to only the numerator of equation 1 because we are computing the dot product of $occupation\_word_i$ and the normalized difference between she and he.

Run the cells below to download and view the occupations.

In [111…

```
def download_occupations():
    if not Path('debiaswe').is_dir():
        print("Downloading occupation list ...")
        !git clone -q https://github.com/tolga-b/debiaswe.git
        print("Occupation list downloaded.")

    return '/content/debiaswe/data/professions.json'


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

        for occupation in occupations:
            print(occupation[0])
```

In [112…

```
occupations_file = download_occupations()
```

In [113…

```
view_occupations(occupations_file)
```

```
accountant
acquaintance
actor
actress
adjunct_professor
administrator
adventurer
advocate
aide
alderman
alter_ego
ambassador
analyst
anthropologist
archaeologist
archbishop
architect
artist
artiste
assassin
assistant_professor
```

```
assistant_professor
associate_dean
associate_professor
astronaut
astronomer
athlete
athletic_director
attorney
author
baker
ballerina
ballplayer
banker
barber
baron
barrister
bartender
biologist
bishop
bodyguard
bookkeeper
boss
boxer
broadcaster
broker
bureaucrat
businessman
businesswoman
butcher
butler
cab_driver
cabbie
cameraman
campaigner
captain
cardiologist
caretaker
carpenter
cartoonist
cellist
chancellor
chaplain
character
chef
chemist
choreographer
cinematographer
citizen
civil_servant
cleric
clerk
coach
```

```
collector
colonel
columnist
comedian
comic
commander
commentator
commissioner
composer
conductor
confesses
congressman
constable
consultant
cop
correspondent
councilman
councilor
counselor
critic
crooner
crusader
curator
custodian
dad
dancer
dean
dentist
deputy
dermatologist
detective
diplomat
director
disc_jockey
doctor
doctoral_student
drug_addict
drummer
economics_professor
economist
editor
educator
electrician
employee
entertainer
entrepreneur
environmentalist
envoy
epidemiologist
evangelist
farmer
fashion_designer
```

fighter_pilot
filmmaker
financier
firebrand
firefighter
fireman
fisherman
footballer
foreman
freelance_writer
gangster
gardener
geologist
goalkeeper
graphic_designer
guidance_counselor
guitarist
hairdresser
handyman
headmaster
historian
hitman
homemaker
hooker
housekeeper
housewife
illustrator
industrialist
infielder
inspector
instructor
interior_designer
inventor
investigator
investment_banker
janitor
jeweler
journalist
judge
jurist
laborer
landlord
lawmaker
lawyer
lecturer
legislator
librarian
lieutenant
lifeguard
lyricist
maestro
magician

```
magistrate
maid
major_leaguer
manager
marksman
marshal
mathematician
mechanic
mediator
medic
midfielder
minister
missionary
mobster
monk
musician
nanny
narrator
naturalist
negotiator
neurologist
neurosurgeon
novelist
nun
nurse
observer
officer
organist
painter
paralegal
parishioner
parliamentarian
pastor
pathologist
patrolman
pediatrician
performer
pharmacist
philanthropist
philosopher
photographer
photojournalist
physician
physicist
pianist
planner
plastic_surgeon
playwright
plumber
poet
policeman
politician
```

```
pollster
preacher
president
priest
principal
prisoner
professor
professor_emeritus
programmer
promoter
proprietor
prosecutor
protagonist
protege
protester
provost
psychiatrist
psychologist
publicist
pundit
rabbi
radiologist
ranger
realtor
receptionist
registered_nurse
researcher
restaurateur
sailor
saint
salesman
saxophonist
scholar
scientist
screenwriter
sculptor
secretary
senator
sergeant
servant
serviceman
sheriff_deputy
shopkeeper
singer
singer_songwriter
skipper
socialite
sociologist
soft_spoken
soldier
solicitor
solicitor_general
```

```
solicitor_general
soloist
sportsman
sportswriter
statesman
steward
stockbroker
strategist
student
stylist
substitute
superintendent
surgeon
surveyor
swimmer
taxi_driver
teacher
technician
teenager
therapist
trader
treasurer
trooper
trucker
trumpeter
tutor
tycoon
undersecretary
understudy
valedictorian
vice_chancellor
violinist
vocalist
waiter
waitress
warden
warrior
welder
worker
wrestler
writer
```

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

In [114…

```
#completed code Edward Cruz

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]

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

    # Store the cosine similarities
    similarities = []

    for word in occupation_words:
        try:
            # Get the embedding vector of the current occupation word
            occupation_word_embedding_vector = word_to_vector_map[word]
            # Compute cosine similarity between embedding vector of the occupation word and normalized she - he vector
            similarity = np.dot(occupation_word_embedding_vector, normalized_difference_she_he) / np.linalg.norm(occupation_w
            similarities.append((similarity, word))
        except KeyError:
            if verbose:
                print(f"{word} is not in our vocabulary.")

    # Sort similarities
    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.

In [115…

```
he, she = get_occupation_stereotypes('she', 'he', occupations_file, word_to_vector_map)

print("Occupations closest to he:")
for occupation in he:
    print(f"{occupation[0], occupation[1]}")

print("\nOccupations closest to she:")
for occupation in she:
    if occupation[1] != 'businesswoman': # Excluding businesswoman
        print(f"{occupation[0], occupation[1]}")
```

```
Occupations closest to he:
(-0.35621238403428845, 'coach')
(-0.33694609670744985, 'caretaker')
(-0.316340865049864, 'captain')
(-0.30927324022117164, 'marshal')
(-0.30729135719953826, 'colonel')
(-0.3024871694158589, 'skipper')
(-0.3021438553042009, 'manager')
(-0.30165373167213716, 'midfielder')
(-0.2996711177716737, 'archbishop')
(-0.29443064216114034, 'commander')
(-0.2902867659336358, 'footballer')
(-0.2888985018624171, 'bishop')
(-0.2819996009929521, 'marksman')
(-0.2796388242138503, 'firebrand')
(-0.27878757562089734, 'provost')
(-0.2780793318566927, 'substitute')
(-0.27217972416807396, 'lieutenant')
(-0.2719793191353711, 'custodian')
(-0.2719191225303118, 'superintendent')
(-0.27134651130558013, 'goalkeeper')

Occupations closest to she:
(0.32072737162261483, 'singer')
(0.3219568449876504, 'publicist')
(0.3440519945920838, 'nanny')
(0.34466952852982563, 'therapist')
```

```
(0.3474659148960079, 'confesses')
(0.3573048226883874, 'dancer')
(0.3645661876706785, 'hairdresser')
(0.36983542416768733, 'receptionist')
(0.3729106376588047, 'housekeeper')
(0.37309747916414077, 'homemaker')
(0.3812397049469023, 'housewife')
(0.3813398603461959, 'nurse')
(0.3892646060961875, 'narrator')
(0.4138336650968037, 'maid')
(0.42853139796990286, 'socialite')
(0.4434377286463442, 'waitress')
(0.44732913871667485, 'stylist')
(0.4686138536614218, 'ballerina')
(0.49840294304026295, 'actress')
```

Task 4 - Debiasing word embeddings

Gender Specific words

Words that are associated with a gender by definition. For example, brother, sister, businesswoman or businessman.

Gender neutral words

The remanining words that are not specific to a gender are gender neutral. For example, flight attendant or shoes. The compliment of gender specific words, can be taken as the gender neutral words.

Step 1 - Identify gender subspace i.e identify the direction of the embedding that captures the bias

To robustly estimate bias, we use the gender specific words to learn a gender subpace in the embedding. To identify the gender subspace, we consider the vector difference of gender specific word pairs, such as $\vec{she} - \vec{he}$, $\vec{woman} - \vec{man}$ or $\vec{her} - \vec{his}$. This identifies a **gender direction or bias subspace** $g \epsilon \mathbb{R}^d$ which captures gender in the embedding.

Note: We will use g and $bias\_direction$ interchangeably in this lab.

In [116…

```python
gender = word_to_vector_map['she'] - word_to_vector_map['he']
print(gender)
```

```
[ 0.261302   0.438481  -0.13376    0.12281    0.00838    0.64455
  0.13151    0.01198    0.73557   -0.04754   -0.04261   -0.23387
  0.56951    0.24359    0.29471    0.152461  -0.44638    0.08563
  0.66735   -0.202578   0.28133    0.71557    0.04015    0.42204
  0.63574    0.1193    -0.429694   0.216301   0.08826   -0.5115
 -0.286      0.227249   0.25811    0.18075   -0.22733   -0.151844
 -0.13196   -0.14412    0.01709   -0.6281     0.124465  -0.16902
  0.6244667 -0.53734    0.379254  -0.3373     0.384876  -0.92383
 -0.019064   0.435641 ]
```

The gender subspace can also be captured more accurately by taking gender pair difference vectors and computing its principal components (PCs). The top PC, denoted by the unit vector g, captures the gender subspace.

In [117…

```
def get_gender_subspace(pairs, word_to_vector_map, num_components=10):
    """
    Compute the gender subspace by computing the principal components of
    ten gender pair vectors.
    Arguments:
        pairs (List[Tuple(String, String)]): A list of gender specific word pairs
        word_to_vector_map (Dict): A dictionary mapping words to embedding vectors
        num_components (Int): The number of principal components to compute. Defaults to 10
    Returns:
        gender_subspace (ndarray): The gender bias subspace(or direction) of shape (embedding dimension,)
    """


    matrix = []
    for word_1, word_2 in pairs:
        embedding_vector_word_1 = word_to_vector_map[word_1]
        embedding_vector_word_2 = word_to_vector_map[word_2]
        center = (embedding_vector_word_1 + embedding_vector_word_2) / 2
        matrix.append(embedding_vector_word_1 - center)
        matrix.append(embedding_vector_word_2 - center)

    matrix = np.array(matrix)
    pca = PCA(n_components=num_components)
    pca.fit(matrix)

    pcs = pca.components_                    # Sorted by decreasing explained variance
    eigenvalues = pca.explained_variance_    # Eigenvalues
    gender_subspace = pcs[0]                 # The first element has the highest eigenvalue
    return gender_subspace
```

In [118…

```
gender_specific_pairs = [
    ('she', 'he'),
    ('her', 'his'),
    ('woman', 'man'),
    ('mary', 'john'),
    ('herself', 'himself'),
    ('daughter', 'son'),
    ('mother', 'father'),
    ('gal', 'guy'),
    ('girl', 'boy'),
    ('female', 'male')
]
gender_direction = get_gender_subspace(gender_specific_pairs, word_to_vector_map)
print(gender_direction)
```

```
[ 0.06639183  0.15316703  0.13179385  0.03910503  0.012115    0.2856193
```

```
[ 0.06639123  0.15316702 -0.12170385  0.02910502 -0.012115    0.2956192
  0.10015248  0.03503806  0.27605339 -0.06259264  0.04843718 -0.20243709
  0.22435017  0.02205075  0.08795604  0.05350635 -0.23457441 -0.0051648
  0.29096997  0.02894429  0.10423079  0.24379617  0.05296573  0.17222571
  0.13557158  0.13746521 -0.05081975  0.11252051  0.01639264 -0.2113686
 -0.1403471   0.13498117  0.08092433  0.02423979 -0.10780551 -0.05927322
 -0.04857578 -0.03199024  0.08174041 -0.17759707 -0.02782478 -0.16880811
  0.27589146 -0.18007478  0.04123208 -0.09385728  0.11011447 -0.25650007
  0.06258361  0.00847159]
```

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?

In [119…

```python
print('Names and their similarities with simple gender subspace')
names = ["mary", "john", "sweta", "david", "kazim", "angela"]
for name in names:
    print(name, cosine_similarity(word_to_vector_map[name], gender))

print()
print('Names and their similarities with PCA based gender subspace')
names = ["mary", "john", "sweta", "david", "kazim", "angela"]
for name in names:
    print(name, cosine_similarity(word_to_vector_map[name], gender_direction))
```

```
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
```

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

In [120…

```python
words = ["engineer", "science", "pilot", "technology", "lipstick", "arts", "singer", "computer", "receptionist", "fashion", '
```

```python
for word in words:
    print(word, cosine_similarity(word_to_vector_map[word], gender_direction))
```

```
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
```

Step 2 - Neutralize gender neutral words

Ensures that gender neutral words are zero in the gender subspace. This means that this steps takes a vector such as $e_{fashion}$ and turns its components into zeros in the direction of g to produce $e_{fashion}^{debiased}$

To remove bias in words such as "receptionist" or "shoe", given an input embedding of the word e, we compute debiased e denoted as $e^{debiased}$ by using the formulas:

$$e^{bias\_component} = \frac{e \cdot bias\_direction}{||bias\_direction||_2^2} * bias\_direction \tag{3}$$

$$e^{debiased} = e - e^{bias\_component} \tag{4}$$

Where $e^{bias\_component}$ is the projection of the word embedding e onto the gender subspace. Since the gender subspace is an orthogonal unit vector it is simply a direction. This also means that $e^{debiased}$ is the projection onto the orthogonal subspace.

$||g||_2^2$ is the squared euclidean norm of g formulated as:

$$||g||_2^2 = \sum_i g_i^2$$

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

---

In [121…

```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
    """

    # Get the vector representation of word
    embedding_of_word = word_to_vector_map[word]

    # Compute the projection of word onto gender direction (eq. 3)
    projection_of_word_onto_gender = (np.dot(embedding_of_word, gender_direction) / np.sum(gender_direction**2)) * gender_dir

    # Neutralize word (eq. 4)
    debiased_word  = embedding_of_word - projection_of_word_onto_gender

    return debiased_word
```

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

In [122…

```
word = "babysit"
print(f"Before neutralization, cosine similarity between {word} and gender is: {cosine_similarity(word_to_vector_map[word], g

debiased_word = neutralize(word, gender_direction, word_to_vector_map)
print(f"After neutralization, cosine similarity between {word} and gender is: {cosine_similarity(debiased_word, gender_direct
```

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

Step 3 - Equalize

Equalizes sets of gender specific words outside the subspace. The goal is to ensure that gender neutral words are equidistance to all the words in the set. We want to ensure that gender specific words are not biased with respect to neutral words.

For example, consider the set {woman, man}, if the neutral word "babysit" is closer to "woman" than "man" then the neutralization of "babysit" can reduce the gender-stereotype associated with babysitting but does not make "babysit" equidistant to "woman" and "man".

Given two gender specific word pairs w_1 and w_2 to debias, and their embeddings e_{w_1} and e_{w_2}, equalization can be achieved with the following equations:

$$\mu = \frac{e_{w_1} + e_{w_2}}{2} \tag{5}$$

$$\mu_B = \frac{\mu \cdot bias\_direction}{||bias\_direction||_2^2} * bias\_direction \tag{6}$$

$$v = \mu - \mu_B \tag{7}$$

$$e_{w_1B} = \frac{e_{w_1} \cdot bias\_direction}{||bias\_direction||_2^2} * bias\_direction \tag{8}$$

$$e_{w_2B} = \frac{e_{w_2} \cdot bias\_direction}{||bias\_direction||_2^2} * bias\_direction \tag{9}$$

$$e_{w_1B}^{new} = \sqrt{|1 - ||v||_2^2|} * \frac{e_{w_1B} - \mu_B}{||(e_{w_1} - v) - \mu_B||_2} \tag{10}$$

$$e_{w_2B}^{new} = \sqrt{|1 - ||v||_2^2|} * \frac{e_{w_2B} - \mu_B}{||(e_{w_2} - v) - \mu_B||_2} \tag{11}$$

$$e_1 = v + e_{w_1B}^{new} \tag{12}$$

$$e_2 = v + e_{w_2B}^{new} \tag{13}$$

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

In [123…

```
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 = None
    embedding_word_a = None
```

```python
    embedding_word_b = None

    # Compute the mean (eq. 5) of embedding_word_a and embedding_word_a (~ 1 line)
    mean = None

    # Compute the projection of mean representation onto the bias direction (eq. 6) (~ 1 line)
    mean_B = None

    # Compute the projection onto the orthogonal subspace (eq. 7) (~ 1 line)
    mean_othorgonal = None

    # Compute the projection of th embedding of word a onto the bias direction (eq. 8) (~ 1 line)
    embedding_word_a_on_bias_direction = None

    # Compute the projection of th embedding of word b onto the bias direction (eq. 9) (~ 1 line)
```



```python
    new_embedding_word_b_on_bias_direction = None

    # Equalize embedding of word a using eq. 12 (~ 1 line)
    embedding_word_a =  None

    # Equalize embedding of word b using eq. 13 (~ 1 line)
    embedding_word_b = None

    # End code here #

    return embedding_word_a, embedding_word_b
```

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

In [124…

```python
def equalization(equality_set, bias_direction, word_to_vector_map):
    """
    Equalizes 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
    """

    # Get the vector representation of word pair by unpacking equality_set
    word_a, word_b = equality_set
    embedding_word_a = word_to_vector_map[word_a]
    embedding_word_b = word_to_vector_map[word_b]

    # Add this line to return the embedding vectors
    return embedding_word_a, embedding_word_b


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): -0.08502503175882657
(embedding vector of mother, gender_direction): 0.3332593015356538
```

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

References:

- The debiasing algorithm is from Bolukbasi et al., 2016 Man is to Computer Programmer as Woman is to Homemake? Debiasing word Embeddings
- The code is partly adapted from Andrew Ng's debiasing word embeddings course on Coursera
- The GloVe word embeddings is publicly available at (https://nlp.stanford.edu/projects/glove/) and is due to the works of Jeffrey Pennington, Richard Socher, and Christopher D. Manning.