Welcome to your assignment this week!

To better understand bias and discrimination in AI, in this assignment, we will look at a Natural Language Processing use case.

Natural Language Pocessing (NLP) is a branch of Artificial Intelligence (AI) that helps computers to understand, to interpret and to manipulate natural (i.e. human) language. Imagine NLP-powered machines as black boxes that are capable of understanding and evaluating the context of the input documents (i.e. collection of words), outputting meaningful results that depend on the task the machine is designed for.

Just like any other machine learning algorithm, biased data results in biased outcomes. And just like any other algorithm, results debiasing is painfully annoying, to the point that it might be simpler to unbias the society itself.

The big deal: word embeddings

Words must be represented as **numeric vectors** in order to be fed into machine learning algorithms. One of the most powerful (and popular) ways to do it is through **Word Embeddings**. In word embedding models, each word in a given language is assigned to a high-dimensional vector, such that **the geometry of the vectors captures relations between the words**.

Because word embeddings are very computionally expensive to train, most ML practitioners will load a pre-trained set of embeddings.

After this assignment you will be able to:

- Load pre-trained word vectors, and measure similarity using cosine similarity
- Use word embeddings to solve word analogy problems such as Man is to Woman as King is to _____.
- Modify word embeddings to reduce their gender bias

Run the following cell to load the packages you will need.

```
import numpy as np
from w2v_utils import *
```

Next, lets load the word vectors. For this assignment, we will use 50-dimensional GloVe vectors to represent words. Run the following cell to load the `word_to_vec_map`.

```
# original code but i change my file directory
# words, word_to_vec_map = read_glove_vecs('data/glove.6B.50d.txt')
words, word_to_vec_map =
read_glove_vecs('../../../data/glove.6B.50d.txt')
```

You've loaded:

- `words`: set of words in the vocabulary.
- `word_to_vec_map`: dictionary mapping words to their GloVe vector representation.

```
print("Example of words: ",list(words)[:10])
print("Vector for word 'person' = ", word_to_vec_map.get('person'))
```

```
Example of words:  ['wazzani', 'ogive', 'dubbers', 'hesler', '-32',
'jundullah', 'pannier', 'mushake', 'parichay', 'matchday']
Vector for word 'person' =  [ 0.61734    0.40035    0.067786  -0.34263
2.0647     0.60844
  0.32558    0.3869     0.36906    0.16553    0.0065053 -0.075674
  0.57099    0.17314    1.0142    -0.49581   -0.38152    0.49255
 -0.16737   -0.33948   -0.44405    0.77543    0.20935    0.6007
  0.86649   -1.8923    -0.37901   -0.28044    0.64214   -0.23549
  2.9358    -0.086004  -0.14327   -0.50161    0.25291   -0.065446
  0.60768    0.13984    0.018135  -0.34877    0.039985   0.07943
  0.39318    1.0562    -0.23624   -0.4194    -0.35332   -0.15234
  0.62158    0.79257  ]
```

GloVe vectors provide much more useful information about the meaning of individual words. Lets now see how you can use GloVe vectors to decide how similar two words are.

Cosine similarity

To measure how similar two words are, we need a way to measure the degree of similarity between two embedding vectors for the two words. Given two vectors u and v, cosine similarity is defined as follows:

\begin{equation} \text{CosineSimilarity(u, v)} = \frac {u . v} {||u||_2 ||v||_2} = cos(\theta) \tag{1} \ end{equation}

where $u.v$ is the dot product (or inner product) of two vectors, $||u||_2$ is the norm (or length) of the vector u, and θ is the angle between u and v. This similarity depends on the angle between u and v. If u and v are very similar, their cosine similarity will be close to 1; if they are dissimilar, the cosine similarity will take a smaller value.

Task 1: Implement the function `cosine_similarity()` to evaluate similarity between word vectors.

Reminder: The norm of u is defined as $||u||_2 = \sqrt{\sum_{i=1}^{n} u_i^2}$

```
## Task 1
# cosine_similarity
```

```
def cosine_similarity(u, v):
    """
    Cosine similarity reflects the degree of similariy between u and v

    Arguments:
        u -- a word vector of shape (n,)
        v -- a word vector of shape (n,)

    Returns:
        cosine_similarity -- the cosine similarity between u and v
defined by the formula above.
    """
    ## START YOU CODE HERE
    simililarity = np.dot(u, v) / (np.linalg.norm(u) *
np.linalg.norm(v))
    return simililarity
    ## END
```

Task 2: Implement `most_similar_word` which returns the most similar word to a word.

```
## Task 2
# GRADED FUNCTION: most_similar_word

def most_similar_word(word, word_to_vec_map):
    """
    Most similar word return the most similar word to the word u.

    Arguments:
        word -- a word, string
        word_to_vec_map -- dictionary that maps words to their
corresponding vectors.

    Returns:
    best_word --  the most similar word to u as measured by cosine
similarity
    """
    ## START YOU CODE HERE
    # Vector for the given word
    word_vec = word_to_vec_map[word]

    best_word = None
    max_similarity = -1  # initialize with very low similarity

    for candidate_word, candidate_vec in word_to_vec_map.items():
        if candidate_word == word:
            continue  # skip the same word
```

```
        similarity = cosine_similarity(word_vec, candidate_vec)

        if similarity > max_similarity:
            max_similarity = similarity
            best_word = candidate_word

    return best_word

    ## END
```

Answer the questions below:

TASK 3: Write a code the answer the following questions:

What is the similarity between the words brother and friend?

```
similarity1 = cosine_similarity(word_to_vec_map['brother'],
word_to_vec_map['friend'])
print("Similarity between 'brother' and 'friend' =", similarity1)
```

```
Similarity between 'brother' and 'friend' = 0.8713178668124658
```

What is the similarity between the words computer and kid?

```
similarity2 = cosine_similarity(word_to_vec_map['computer'],
word_to_vec_map['kid'])
print("Similarity between 'computer' and 'kid' =", similarity2)
```

```
Similarity between 'computer' and 'kid' = 0.4380016621036385
```

What is the similarity between the words V1=(france - paris) and V2=(rome - italy)?

```
france_vec = word_to_vec_map["france"]
paris_vec = word_to_vec_map["paris"]
rome_vec = word_to_vec_map["rome"]
italy_vec = word_to_vec_map["italy"]

V1 = france_vec - paris_vec
V2 = rome_vec - italy_vec

similarity3 = cosine_similarity(V1, V2)
print("Similarity between 'V1=(france - paris)' and 'V2=(rome -
italy)' =", similarity3)
```

```
Similarity between 'V1=(france - paris)' and 'V2=(rome - italy)' = -
0.6751479308174202
```

What is the most similar word to computer?

```
most_similar1 = most_similar_word("computer", word_to_vec_map)
print("Most similar word to 'computer' is =", most_similar1)
```

```
Most similar word to 'computer' is = computers
```

What is the most similar word to australia?

```
most_similar2 = most_similar_word("australia", word_to_vec_map)
print("Most similar word to 'australia' is =", most_similar2)
```

```
Most similar word to 'australia' is = zealand
```

What is the most similar word to python?

```
most_similar3 = most_similar_word("python", word_to_vec_map)
print("Most similar word to 'python' is =", most_similar3)
```

```
Most similar word to 'python' is = reticulated
```

Playing around the cosine similarity of other inputs will give you a better sense of how word vectors behave.

Word analogy task

In the word analogy task, we complete the sentence "*a* is to *b* as *c* is to ____". An example is '*man* is to *woman* as *king* is to *queen*' . In detail, we are trying to find a word *d*, such that the associated word vectors e_a, e_b, e_c, e_d are related in the following manner: $e_b - e_a \approx e_d - e_c$. We will measure the similarity between $e_b - e_a$ and $e_d - e_c$ using cosine similarity.

Task 4: Complete the code below to be able to perform word analogies!

```
## Task 4
# GRADED FUNCTION: complete_analogy

def complete_analogy(word_a, word_b, word_c, word_to_vec_map):
    """
    Performs the word analogy task as explained above: a is to b as c
is to ____.

    Arguments:
    word_a -- a word, string
    word_b -- a word, string
    word_c -- a word, string
```

```
    word_to_vec_map -- dictionary that maps words to their
corresponding vectors.

    Returns:
    best_word --  the word such that v_b - v_a is close to v_best_word
- v_c, as measured by cosine similarity
    """
    ## START YOU CODE HERE

    # Get the vectors for the words
    e_a = word_to_vec_map[word_a]
    e_b = word_to_vec_map[word_b]
    e_c = word_to_vec_map[word_c]

    # Compute the target vector
    target_vec = e_b - e_a + e_c

    max_cosine_sim = -np.inf
    best_word = None

    for word, vec in word_to_vec_map.items():
        # Skip input words
        if word in [word_a, word_b, word_c]:
            continue

        cosine_sim = cosine_similarity(target_vec, vec)

        if cosine_sim > max_cosine_sim:
            max_cosine_sim = cosine_sim
            best_word = word

    return best_word

    ##ENFD
```

Run the cell below to test your code, this may take 1-2 minutes.

```
triads_to_try = [('italy', 'italian', 'spain'), ('india', 'delhi',
'japan'), ('man', 'woman', 'boy'), ('small', 'smaller', 'large')]
for triad in triads_to_try:
    print ('{} -> {} :: {} -> {}'.format( *triad,
complete_analogy(*triad,word_to_vec_map)))
```

```
italy -> italian :: spain -> spanish
india -> delhi :: japan -> tokyo
india -> delhi :: japan -> tokyo
man -> woman :: boy -> girl
man -> woman :: boy -> girl
small -> smaller :: large -> larger
small -> smaller :: large -> larger
```

Once you get the correct expected output, please feel free to modify the input cells above to test your own analogies. Try to find some other analogy pairs that do work, but also find some where the algorithm doesn't give the right answer: For example, you can try small->smaller as big->?.

```
# Try additional analogy pairs, including some that may not work as
expected
analogy_tests = [
        ('man', 'woman', 'king'),
        ('paris', 'france', 'rome'),
        ('walk', 'walking', 'run'),
        ('good', 'better', 'bad'),
        ('small', 'smaller', 'big'),
        ('cat', 'kitten', 'dog'),
        ('doctor', 'nurse', 'engineer'),
        ('happy', 'happier', 'sad'),
        ('fast', 'faster', 'slow'),
        ('apple', 'apples', 'banana'),
        ('boy', 'girl', 'brother'),
        ('rich', 'richer', 'poor'),
        ('teacher', 'school', 'doctor'),
        ('sun', 'day', 'moon'),
        ('car', 'driver', 'plane'),
        ('strong', 'stronger', 'weak'),
        ('long', 'longer', 'short'),
        ('big', 'bigger', 'small'),
        ('cold', 'colder', 'hot'),
        ('child', 'children', 'woman')
    ]

for triad in analogy_tests:
    try:
        result = complete_analogy(*triad, word_to_vec_map)
        print(f"{triad[0]} -> {triad[1]} :: {triad[2]} -> {result}")
    except Exception as e:
        print(f"{triad[0]} -> {triad[1]} :: {triad[2]} -> Error: {e}")
```

```
man -> woman :: king -> queen
paris -> france :: rome -> italy
paris -> france :: rome -> italy
walk -> walking :: run -> running
walk -> walking :: run -> running
good -> better :: bad -> worse
good -> better :: bad -> worse
small -> smaller :: big -> bigger
small -> smaller :: big -> bigger
cat -> kitten :: dog -> puppy
cat -> kitten :: dog -> puppy
doctor -> nurse :: engineer -> mechanic
doctor -> nurse :: engineer -> mechanic
```

```
happy -> happier :: sad -> sadder
happy -> happier :: sad -> sadder
fast -> faster :: slow -> slower
fast -> faster :: slow -> slower
apple -> apples :: banana -> bananas
apple -> apples :: banana -> bananas
boy -> girl :: brother -> cousin
boy -> girl :: brother -> cousin
rich -> richer :: poor -> poorer
rich -> richer :: poor -> poorer
teacher -> school :: doctor -> medical
teacher -> school :: doctor -> medical
sun -> day :: moon -> eve
sun -> day :: moon -> eve
car -> driver :: plane -> pilot
car -> driver :: plane -> pilot
strong -> stronger :: weak -> weaker
strong -> stronger :: weak -> weaker
long -> longer :: short -> usually
long -> longer :: short -> usually
big -> bigger :: small -> larger
big -> bigger :: small -> larger
cold -> colder :: hot -> cooler
cold -> colder :: hot -> cooler
child -> children :: woman -> mother
child -> children :: woman -> mother
```

Debiasing word vectors

In the following exercise, you will examine gender biases that can be reflected in a word embedding, and explore algorithms for reducing the bias. In addition to learning about the topic of debiasing, this exercise will also help hone your intuition about what word vectors are doing. This section involves a bit of linear algebra, though you can probably complete it even without being expert in linear algebra, and we encourage you to give it a shot.

Lets first see how the GloVe word embeddings relate to gender. You will first compute a vector $g = e_{woman} - e_{man}$, where e_{woman} represents the word vector corresponding to the word *woman*, and e_{man} corresponds to the word vector corresponding to the word *man*. The resulting vector g roughly encodes the concept of "gender". (You might get a more accurate representation if you compute $g_1 = e_{mother} - e_{father}$, $g_2 = e_{girl} - e_{boy}$, etc. and average over them. But just using $e_{woman} - e_{man}$ will give good enough results for now.)

Task 5: Compute the bias vector using woman - man

```
## START YOU CODE HERE

g = word_to_vec_map['woman'] - word_to_vec_map['man']

## END

print(g)
```

```
[-0.087144    0.2182     -0.40986    -0.03922    -0.1032      0.94165
 -0.06042     0.32988     0.46144    -0.35962     0.31102    -0.86824
  0.96006     0.01073     0.24337     0.08193    -1.02722    -0.21122
  0.695044   -0.00222     0.29106     0.5053     -0.099454    0.40445
  0.30181     0.1355     -0.0606     -0.07131    -0.19245    -0.06115
 -0.3204      0.07165    -0.13337    -0.25068714 -0.14293    -0.224957
 -0.149       0.048882    0.12191    -0.27362    -0.165476   -0.20426
  0.54376    -0.271425   -0.10245    -0.32108     0.2516     -0.33455
 -0.04371     0.01258   ]
```

Now, you will consider the cosine similarity of different words with g. Consider what a positive value of similarity means vs a negative cosine similarity.

Task 6: Compute and print the similarity between g and the words in name_list

---

```
print ('List of names and their similarities with constructed
vector:')
name_list = ['john', 'marie', 'sophie', 'ronaldo', 'priya', 'rahul',
'danielle', 'reza', 'katy', 'yasmin']

## START YOU CODE HERE
for name in name_list:
    similarity = cosine_similarity(word_to_vec_map[name], g)
    print(f"{name}: {similarity}")

## END
```

```
List of names and their similarities with constructed vector:
john: -0.23163356145973724
marie: 0.315597935396073
sophie: 0.31868789859418784
ronaldo: -0.31244796850329437
priya: 0.17632041839009402
rahul: -0.1691547103923172
danielle: 0.24393299216283895
reza: -0.07930429672199553
katy: 0.2831068659572615
yasmin: 0.23313857767928758
```

TASK 7: What do you observe?

The cosine similarity values show that names typically associated with females (e.g., marie, sophie, danielle, katy, yasmin) have positive similarity with the gender bias vector, while names typically associated with males (e.g., john, ronaldo, rahul, reza) have negative similarity. This indicates that the word embeddings capture gender-related information, reflecting societal biases present in the training data.

Task 8: Compute and print the similarity between g and the words in word_list:

```
print('Other words and their similarities:')
word_list = ['lipstick', 'guns', 'science', 'arts', 'literature',
'warrior','doctor', 'tree', 'receptionist',
             'technology',  'fashion', 'teacher', 'engineer', 'pilot',
'computer', 'singer']

## START YOU CODE HERE
for word in word_list:
    similarity = cosine_similarity(word_to_vec_map[word], g)
    print(f"{word}: {similarity}")

## END
```

```
Other words and their similarities:
lipstick: 0.27691916256382665
guns: -0.1888485567898898
science: -0.06082906540929699
arts: 0.008189312385880344
literature: 0.0647250443345993
warrior: -0.20920164641125288
doctor: 0.11895289410935043
tree: -0.07089399175478092
receptionist: 0.3307794175059374
technology: -0.13193732447554296
fashion: 0.035638946257727
teacher: 0.1792092343182567
engineer: -0.08039280494524072
pilot: 0.0010764498991917074
computer: -0.10330358873850498
singer: 0.18500518136496297
```

TASK 9: What do you observe?

The results show that words stereotypically associated with females (e.g., lipstick, fashion, receptionist, singer, teacher) tend to have positive cosine similarity with the gender bias vector, while words associated with males (e.g., guns, warrior, engineer, pilot) have negative similarity. Neutral or less gendered words (e.g., science, literature, computer, tree) show values closer to zero. This demonstrates that word embeddings encode and reflect societal gender stereotypes, even for professions and activities that should ideally be gender-neutral.

We'll see below how to reduce the bias of these vectors, using an algorithm due to Boliukbasi et al., 2016. Note that some word pairs such as "actor"/"actress" or "grandmother"/"grandfather" should remain gender specific, while other words such as "receptionist" or "technology" should be neutralized, i.e. not be gender-related. You will have to treat these two type of words differently when debiasing.

Neutralize bias for NON-GENDER specific words

The figure below should help you visualize what neutralizing does. If you're using a 50-dimensional word embedding, the 50 dimensional space can be split into two parts: The bias-direction g, and the remaining 49 dimensions, which we'll call $g_\perp$. In linear algebra, we say that the 49 dimensional $g_\perp$ is perpendicular (or "orthogonal") to g, meaning it is at 90 degrees to g. The neutralization step takes a vector such as $e_{receptionist}$ and zeros out the component in the direction of g, giving us $e^{debiased}_{receptionist}$.

Even though $g_\perp$ is 49 dimensional, given the limitations of what we can draw on a screen, we illustrate it using a 1 dimensional axis below.

TASK 10: Implement `neutralize()` to remove the bias of words such as "receptionist" or "scientist". Given an input embedding e, you can use the following formulas to compute $e^{debiased}$:

\begin{equation} e^{bias_component} = \frac{e \cdot g}{||g||_2^2} * g\tag{2} \end{equation}

\begin{equation} e^{debiased} = e - e^{bias_component}\tag{3} \end{equation}

If you are an expert in linear algebra, you may recognize $e^{bias_component}$ as the projection of e onto the direction g. If you're not an expert in linear algebra, don't worry about this.

```
# TASK 10
# GRADED neutralize

def neutralize(word, g, word_to_vec_map):
    """
    Removes the bias of "word" by projecting it on the space
orthogonal to the bias axis.
    This function ensures that gender neutral words are zero in the
gender subspace.

    Arguments:
        word -- string indicating the word to debias
        g -- numpy-array of shape (50,), corresponding to the bias
axis (such as gender)
        word_to_vec_map -- dictionary mapping words to their
corresponding vectors.

    Returns:
        e_debiased -- neutralized word vector representation of the
input "word"
    """
    ## START YOU CODE HERE

    e = word_to_vec_map[word]
    e_bias_component = (np.dot(e, g) / np.linalg.norm(g)**2) * g
    e_debiased = e - e_bias_component

    return e_debiased

    ## END


e = "receptionist"
print("cosine similarity between " + e + " and g, before neutralizing:
", cosine_similarity(word_to_vec_map["receptionist"], g))

e_debiased = neutralize("receptionist", g, word_to_vec_map)
print("cosine similarity between " + e + " and g, after neutralizing:
", cosine_similarity(e_debiased, g))
```

```
cosine similarity between receptionist and g, before neutralizing:
0.3307794175059374
cosine similarity between receptionist and g, after neutralizing:  -
3.8581292784004314e-17
```

Equalization algorithm for GENDER-SPECIFIC words

Next, lets see how debiasing can also be applied to word pairs such as "actress" and "actor." Equalization is applied to pairs of words that you might want to have differ only through the gender property. As a concrete example, suppose that "actress" is closer to "babysit" than "actor." By applying neutralizing to "babysit" we can reduce the gender-stereotype associated with babysitting. But this still does not guarantee that "actor" and "actress" are equidistant from "babysit." The equalization algorithm takes care of this.

The key idea behind equalization is to make sure that a particular pair of words are equi-distant from the 49-dimensional $g_\perp$. The equalization step also ensures that the two equalized steps are now the same distance from $e^{debiased}_{receptionist}$, or from any other word that has been neutralized. In pictures, this is how equalization works:

The derivation of the linear algebra to do this is a bit more complex. (See Bolukbasi et al., 2016 for details.) But the key equations are:

\begin{equation} \mu = \frac{e_{w1} + e_{w2}}{2}\tag{4} \end{equation}

\begin{equation} \mu_{B} = \frac {\mu \cdot bias_axis}{||bias_axis||_2^2} *bias_axis \tag{5} \ end{equation}

\begin{equation} \mu_{\perp} = \mu - \mu_{B} \tag{6} \end{equation}

\begin{equation} e_{w1B} = \frac {e_{w1} \cdot bias_axis}{||bias_axis||_2^2} *bias_axis \tag{7} \ end{equation}

\begin{equation} e_{w2B} = \frac {e_{w2} \cdot bias_axis}{||bias_axis||_2^2} *bias_axis \tag{8} \ end{equation}

\begin{equation} e_{w1B}^{corrected} = \sqrt{ |{1 - ||\mu_{\perp} ||^2_2} |} * \frac{e_{\text{w1B}} - \mu_B} {||(e_{w1} - \mu_{\perp}) - \mu_B||} \tag{9} \end{equation}

\begin{equation} e_{w2B}^{corrected} = \sqrt{ |{1 - ||\mu_{\perp} ||^2_2} |} * \frac{e_{\text{w2B}} - \mu_B} {||(e_{w2} - \mu_{\perp}) - \mu_B||} \tag{10} \end{equation}

\begin{equation} e_1 = e_{w1B}^{corrected} + \mu_{\perp} \tag{11} \end{equation}

\begin{equation} e_2 = e_{w2B}^{corrected} + \mu_{\perp} \tag{12} \end{equation}

TASK 11: Implement the function below. Use the equations above to get the final equalized version of the pair of words. Good luck!

```
# TASK 11
# GRADED equalize
```

```
def equalize(pair, bias_axis, word_to_vec_map):
    """
    Debias gender specific words by following the equalize method
described in the figure above.

    Arguments:
    pair -- pair of strings of gender specific words to debias, e.g.
("actress", "actor")
    bias_axis -- numpy-array of shape (50,), vector corresponding to
the bias axis, e.g. gender
    word_to_vec_map -- dictionary mapping words to their corresponding
vectors

    Returns
    e_1 -- word vector corresponding to the first word
    e_2 -- word vector corresponding to the second word
    """
    ## START YOU CODE HERE

    w1, w2 = pair
    e_w1 = word_to_vec_map[w1]
    e_w2 = word_to_vec_map[w2]

    # Compute mean vector μ
    mu = (e_w1 + e_w2) / 2.0

    # Project μ on bias axis (μ_B)
    mu_B = (np.dot(mu, bias_axis) / np.linalg.norm(bias_axis)**2) *
bias_axis

    # Orthogonal component of μ (μ_perp)
    mu_orth = mu - mu_B

    # Project e_w1 and e_w2 on bias axis
    e_w1B = (np.dot(e_w1, bias_axis) / np.linalg.norm(bias_axis)**2) *
bias_axis
    e_w2B = (np.dot(e_w2, bias_axis) / np.linalg.norm(bias_axis)**2) *
bias_axis

    numerator = e_w1B - mu_B
    denominator = abs(np.linalg.norm((e_w1 - mu_orth) - mu_B))
    scale = np.sqrt(abs(1 - abs(np.linalg.norm(mu_orth)**2)))
    e_w1B_corrected = scale * numerator / denominator

    numerator = e_w2B - mu_B
    denominator = abs(np.linalg.norm((e_w2 - mu_orth) - mu_B))
    e_w2B_corrected = scale * numerator / denominator

    # Final equalized vectors (equations 11 and 12)
    e_1 = e_w1B_corrected + mu_orth
```

```
    e_2 = e_w2B_corrected + mu_orth

    return e_1, e_2

    ##END

print("cosine similarities before equalizing:")
print("cosine_similarity(word_to_vec_map[\"man\"], gender) = ",
cosine_similarity(word_to_vec_map["man"], g))
print("cosine_similarity(word_to_vec_map[\"woman\"], gender) = ",
cosine_similarity(word_to_vec_map["woman"], g))
print()
e1, e2 = equalize(("man", "woman"), g, word_to_vec_map)
print("cosine similarities after equalizing:")
print("cosine_similarity(e1, gender) = ", cosine_similarity(e1, g))
print("cosine_similarity(e2, gender) = ", cosine_similarity(e2, g))
```

```
cosine similarities before equalizing:
cosine_similarity(word_to_vec_map["man"], gender) =  -
0.1171109576533683
cosine_similarity(word_to_vec_map["woman"], gender) =
0.35666618846270376

cosine similarities after equalizing:
cosine_similarity(e1, gender) =  -0.7004364289309388
cosine_similarity(e2, gender) =  0.7004364289309387
```

TASK 12: What do you observe?

After applying the equalization algorithm, the cosine similarities of gender-specific word pairs (such as "man" and "woman") with the gender bias vector become equal in magnitude but opposite in sign. This means both words are now symmetrically positioned with respect to the gender direction, reducing bias and ensuring fairness in their representation. The algorithm successfully removes the imbalance, but some residual bias may remain due to limitations in how the bias direction is defined.

Please feel free to play with the input words in the cell above, to apply equalization to other pairs of words.

These debiasing algorithms are very helpful for reducing bias, but are not perfect and do not eliminate all traces of bias. For example, one weakness of this implementation was that the bias direction g was defined using only the pair of words *woman* and *man*. As discussed earlier, if g were defined by computing $g_1 = e_{woman} - e_{man}$; $g_2 = e_{mother} - e_{father}$; $g_3 = e_{girl} - e_{boy}$; and so on and averaging over them, you would obtain a better estimate of the "gender" dimension in the 50 dimensional word embedding space. Feel free to play with such variants as well.

Detecting and Removing Multiclass Bias in Word Embeddings

The method above introduced by Bolukbasi et al. 2016 is a method to debias embeddings by removing components that lie in stereotype-related embedding subspaces. They demonstrate the effectiveness of the approach by removing gender bias from word2vec embeddings, preserving the utility of embeddings and potentially alleviating biases in downstream tasks. However, this method was only for binary labels (e.g., male/female), whereas most real-world demographic attributes, including gender, race, religion, are not binary but continuous or categorical, with more than two categories.

In the following, you are asked to implement the work by Manzini et al. 2019, which is a generalization of Bolukbasi et al.'s (2016) that enables multiclass debiasing, while preserving utility of embeddings.

- Manzini, Thomas and Lim, Yao Chong and Tsvetkov, Yulia and Black, Alan W, **Black is to Criminal as Caucasian is to Police: Detecting and Removing Multiclass Bias in Word Embeddings**, in NAACL 2019. https://arxiv.org/pdf/1904.04047.pdf

You will have to:

1. Demonstrate examples of Race and Religion bais from word2vec.
2. Implement the algorithm used to identify the bias subspace described by **Manzini et al. 2019**.
3. Use the Neutralize and Equalize debiasing debiasing you have implemented above and show the results on a few examples.

```
# 1. Demonstrate examples of Race and Religion bias
race_words = ['black', 'white', 'asian', 'caucasian', 'hispanic',
'indian']
religion_words = ['christian', 'muslim', 'jewish', 'buddhist',
'hindu', 'sikh']

bias_targets = ['criminal', 'police', 'doctor', 'teacher']

print('Race bias:')
for target in bias_targets:
    for race in race_words:
        try:
            sim = cosine_similarity(word_to_vec_map[target],
word_to_vec_map[race])
            print(f"{target} <-> {race}: {sim:.4f}")
        except:
            pass
    print()

print('Religion bias:')
for target in bias_targets:
    for religion in religion_words:
```

```
        try:
            sim = cosine_similarity(word_to_vec_map[target],
word_to_vec_map[religion])
            print(f"{target} <-> {religion}: {sim:.4f}")
        except:
            pass
    print()

# 2. Identify multiclass bias subspace using mean-difference vectors
# For race: use mean of each group and difference
race_vecs = [word_to_vec_map[w] for w in race_words if w in
word_to_vec_map]
race_mean = np.mean(race_vecs, axis=0)
religion_vecs = [word_to_vec_map[w] for w in religion_words if w in
word_to_vec_map]
religion_mean = np.mean(religion_vecs, axis=0)

# Use difference vectors as bias axes
race_bias_axis = race_mean - religion_mean
religion_bias_axis = religion_mean - race_mean
print('Race bias axis (first 5 values):', race_bias_axis[:5])
print('Religion bias axis (first 5 values):', religion_bias_axis[:5])

# 3. Neutralize and Equalize for multiclass bias
# Example: Neutralize 'teacher' with respect to both race and religion
bias axes
teacher_vec = word_to_vec_map['teacher']
for axis in [race_bias_axis, religion_bias_axis]:
    teacher_vec = teacher_vec - (np.dot(teacher_vec, axis) /
np.linalg.norm(axis)**2) * axis
print('Neutralized teacher vector (first 5 values):', teacher_vec[:5])

# Example: Equalize 'doctor' and 'criminal' with respect to race bias
axis
vec1 = word_to_vec_map['doctor']
vec2 = word_to_vec_map['criminal']
mu = (vec1 + vec2) / 2
axis = race_bias_axis
mu_B = (np.dot(mu, axis) / np.linalg.norm(axis)**2) * axis
mu_orth = mu - mu_B
vec1B = (np.dot(vec1, axis) / np.linalg.norm(axis)**2) * axis
vec2B = (np.dot(vec2, axis) / np.linalg.norm(axis)**2) * axis
scale = np.sqrt(abs(1 - np.linalg.norm(mu_orth)**2))
vec1B_corr = scale * (vec1B - mu_B) / abs(np.linalg.norm((vec1 -
mu_orth) - mu_B))
vec2B_corr = scale * (vec2B - mu_B) / abs(np.linalg.norm((vec2 -
mu_orth) - mu_B))
equalized_vec1 = vec1B_corr + mu_orth
equalized_vec2 = vec2B_corr + mu_orth
print('Equalized doctor vector (first 5 values):', equalized_vec1[:5])
```

```
print('Equalized criminal vector (first 5 values):',
equalized_vec2[:5])

Race bias:
criminal <-> black: 0.3730
criminal <-> white: 0.3538
criminal <-> asian: 0.2771
criminal <-> caucasian: 0.0343
criminal <-> hispanic: 0.3182
criminal <-> indian: 0.3811

police <-> black: 0.4548
police <-> white: 0.4407
police <-> asian: 0.2918
police <-> caucasian: 0.0730
police <-> hispanic: 0.3032
police <-> indian: 0.5374

doctor <-> black: 0.3287
doctor <-> white: 0.2464
doctor <-> asian: 0.1517
doctor <-> caucasian: 0.0749
doctor <-> hispanic: 0.1577
doctor <-> indian: 0.3667

teacher <-> black: 0.3757
teacher <-> white: 0.3076
teacher <-> asian: 0.2488
teacher <-> caucasian: 0.1539
teacher <-> hispanic: 0.3817
teacher <-> indian: 0.4037

Religion bias:
criminal <-> christian: 0.2543
criminal <-> muslim: 0.4399
criminal <-> jewish: 0.3748
criminal <-> buddhist: 0.2157
criminal <-> hindu: 0.1801
criminal <-> sikh: 0.1590

police <-> christian: 0.3432
police <-> muslim: 0.5525
police <-> jewish: 0.4079
police <-> buddhist: 0.2841
police <-> hindu: 0.3342
police <-> sikh: 0.2560

doctor <-> christian: 0.3784
doctor <-> muslim: 0.3340
doctor <-> jewish: 0.3239
```

```
doctor <-> buddhist: 0.2211
doctor <-> hindu: 0.2001
doctor <-> sikh: 0.1927

teacher <-> christian: 0.4704
teacher <-> muslim: 0.3704
teacher <-> jewish: 0.4548
teacher <-> buddhist: 0.3718
teacher <-> hindu: 0.2222
teacher <-> sikh: 0.2669

Race bias axis (first 5 values): [-0.6287535   0.23254917  0.1831725
1.17943333 -0.71855   ]
Religion bias axis (first 5 values): [ 0.6287535  -0.23254917 -
0.1831725  -1.17943333  0.71855   ]
Neutralized teacher vector (first 5 values): [-1.02820613  1.01297731
-0.55200634 -1.23401686  0.69389191]
Equalized doctor vector (first 5 values): [ 0.47184932 -0.3078345  -
0.49421817 -0.34457712  0.60543196]
Equalized criminal vector (first 5 values): [ 0.35095172 -0.26311962 -
0.45899751 -0.11779404  0.46726817]

criminal <-> black: 0.3730
criminal <-> white: 0.3538
criminal <-> asian: 0.2771
criminal <-> caucasian: 0.0343
criminal <-> hispanic: 0.3182
criminal <-> indian: 0.3811

police <-> black: 0.4548
police <-> white: 0.4407
police <-> asian: 0.2918
police <-> caucasian: 0.0730
police <-> hispanic: 0.3032
police <-> indian: 0.5374

doctor <-> black: 0.3287
doctor <-> white: 0.2464
doctor <-> asian: 0.1517
doctor <-> caucasian: 0.0749
doctor <-> hispanic: 0.1577
doctor <-> indian: 0.3667

teacher <-> black: 0.3757
teacher <-> white: 0.3076
teacher <-> asian: 0.2488
teacher <-> caucasian: 0.1539
teacher <-> hispanic: 0.3817
teacher <-> indian: 0.4037
```

```
Religion bias:
criminal <-> christian: 0.2543
criminal <-> muslim: 0.4399
criminal <-> jewish: 0.3748
criminal <-> buddhist: 0.2157
criminal <-> hindu: 0.1801
criminal <-> sikh: 0.1590

police <-> christian: 0.3432
police <-> muslim: 0.5525
police <-> jewish: 0.4079
police <-> buddhist: 0.2841
police <-> hindu: 0.3342
police <-> sikh: 0.2560

doctor <-> christian: 0.3784
doctor <-> muslim: 0.3340
doctor <-> jewish: 0.3239
doctor <-> buddhist: 0.2211
doctor <-> hindu: 0.2001
doctor <-> sikh: 0.1927

teacher <-> christian: 0.4704
teacher <-> muslim: 0.3704
teacher <-> jewish: 0.4548
teacher <-> buddhist: 0.3718
teacher <-> hindu: 0.2222
teacher <-> sikh: 0.2669

Race bias axis (first 5 values): [-0.6287535   0.23254917  0.1831725
1.17943333 -0.71855   ]
Religion bias axis (first 5 values): [ 0.6287535  -0.23254917 -
0.1831725  -1.17943333  0.71855   ]
Neutralized teacher vector (first 5 values): [-1.02820613  1.01297731
-0.55200634 -1.23401686  0.69389191]
Equalized doctor vector (first 5 values): [ 0.47184932 -0.3078345  -
0.49421817 -0.34457712  0.60543196]
Equalized criminal vector (first 5 values): [ 0.35095172 -0.26311962 -
0.45899751 -0.11779404  0.46726817]
```

Congratulations!

You've come to the end of this assignment, and have seen a lot of the ways that word vectors can be used as well as modified. Here are the main points you should remember:

- Cosine similarity a good way to compare similarity between pairs of word vectors. (Though L2 distance works too.)

- For NLP applications, using a pre-trained set of word vectors from the internet is often a good way to get started.
- Bias in data is an important problem.
- Neutralize and equalize allow to reduce bias in the data.
- Detecting and Removing Multiclass Bias in Word Embeddings.

Congratulations on finishing this notebook!

References

- The debiasing algorithm is from Bolukbasi et al., 2016, Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings
- The GloVe word embeddings were due to Jeffrey Pennington, Richard Socher, and Christopher D. Manning. (https://nlp.stanford.edu/projects/glove/)