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.

Documents are fed into magic NLP model capable to get, for instance, the sentiment of the original content 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.

In [1]:
```
import numpy as np
from w2v_utils import *
from sklearn.decomposition import PCA
```

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`.

In [2]:
```
def read_glove_vecs(filename, encoding='utf-8'):
    with open(filename, 'r', encoding=encoding) as file:
        words = set()
        word_to_vec_map = {}
        for line in file:
            line = line.strip().split()
            curr_word = line[0]
            words.add(curr_word)
            word_to_vec_map[curr_word] = np.array(line[1:], dtype=np.float64)
    return words, word_to_vec_map
```

In [3]:
```
words, word_to_vec_map = read_glove_vecs('E:/T2 2023/SIT799 Human/Vinit/Assignments/Task
```

You've loaded:

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

In [4]:
```
print("Example of words: ",list(words)[:10])
print("Vector for word 'person' = ", word_to_vec_map.get('person'))
```

```
Example of words:  ['busload', 'sheilas', '22min', 'matley', 'bugz', 'kulmiye', 'brancus
i', 'supersub', 'lags', 'ancrum']
Vector for word 'person' =  [ 0.61734    0.40035    0.067786  -0.34263    2.0647     0.6
0844
  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:

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

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.

Figure 1: The cosine of the angle between two vectors is a measure of how similar they are

---

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

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

In [5]:
```
def cosine_similarity(u, v):

    dot_product = np.dot(u, v)
    norm_u = np.linalg.norm(u)
    norm_v = np.linalg.norm(v)
    cosine_similarity = dot_product / (norm_u * norm_v)

    return cosine_similarity
```

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

In [6]:
```
def most_similar_word(word, word_to_vec_map):
    # Get the word vector for the given word
    word_vector = word_to_vec_map[word]

    # Initialize variables to keep track of the best similarity and the most similar wor
    # Set to a negative value to ensure any valid similarity will be higher
    best_similarity = -1
    best_word = None

    # Iterate over all words in the word_to_vec_map
    for key, value in word_to_vec_map.items():
        # Skip the same word
        if key == word:
            continue

        # Calculate cosine similarity between word_vector and each vector in the map
        similarity = cosine_similarity(word_vector, value)

        # Check if the current word has a higher similarity than the previous best
        if similarity > best_similarity:
            best_similarity = similarity
            best_word = key

    return best_word
```

Answer the questions below:

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

What is the similarity between the words brother and friend?

In [7]:
```
word1 = "brother"
word2 = "friend"
similarity = cosine_similarity(word_to_vec_map[word1], word_to_vec_map[word2])
print("The similarity between the words '{}' and '{}' is: {}".format(word1, word2, simil
```

```
The similarity between the words 'brother' and 'friend' is: 0.8713178668124657
```

What is the similarity between the words computer and kid?

In [8]:
```
word3 = "computer"
word4 = "kid"
similarity = cosine_similarity(word_to_vec_map[word3], word_to_vec_map[word4])
print("The similarity between the words '{}' and '{}' is: {}".format(word3, word4, simil
```

```
The similarity between the words 'computer' and 'kid' is: 0.4380016621036386
```

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

In [9]:
```
V1 = word_to_vec_map["france"] - word_to_vec_map["paris"]

V2 = word_to_vec_map["rome"] - word_to_vec_map["italy"]
```

```
# Calculate cosine similarity between V1 and V2
similarity = cosine_similarity(V1, V2)

print("The similarity between the words V1=(france - paris) and V2=(rome - italy) is:",
```

```
The similarity between the words V1=(france - paris) and V2=(rome - italy) is: -0.675147
9308174203
```

What is the most similar word to computer?

In [10]:
```
word_to_find = "computer"
most_similar = most_similar_word(word_to_find, word_to_vec_map)

print("The most similar word to '{}' is: {}".format(word_to_find, most_similar))
```

```
The most similar word to 'computer' is: computers
```

What is the most similar word to australia?

In [11]:
```
word_to_find_2 = "australia"
most_similar_2 = most_similar_word(word_to_find_2, word_to_vec_map)

print("The most similar word to '{}' is: {}".format(word_to_find_2, most_similar_2))
```

```
The most similar word to 'australia' is: zealand
```

What is the most similar word to python?

In [12]:
```
word_to_find_3 = "python"
most_similar_3 = most_similar_word(word_to_find_3, word_to_vec_map)

print("The most similar word to '{}' is: {}".format(word_to_find_3, most_similar_3))
```

```
The 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!

In [13]:
```
# GRADED FUNCTION: complete_analogy

def complete_analogy(word_a, word_b, word_c, word_to_vec_map):
    # Get word vectors for all the words
    word_a_vec = word_to_vec_map[word_a]
    word_b_vec = word_to_vec_map[word_b]
    word_c_vec = word_to_vec_map[word_c]

    # Perform the analogy calculation
    analogy_result = word_b_vec - word_a_vec + word_c_vec

    # Initialize variables to keep track of the best similarity and the most similar wor
    # Set to a negative value to ensure any valid similarity will be higher
    best_similarity = -1
    best_word = None

    # Iterate over all words in the word_to_vec_map
    for word, vec in word_to_vec_map.items():
        # Skip the same words
        if word in [word_a, word_b, word_c]:
            continue

        # Calculate cosine similarity between analogy_result and each vector in the map
        similarity = cosine_similarity(analogy_result, vec)

        # Check if the current word has a higher similarity than the previous best
        if similarity > best_similarity:
            best_similarity = similarity
            best_word = word

    return best_word
```

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

In [14]:
```
triads_to_try = [('italy', 'italian', 'spain'), ('india', 'delhi', 'japan'), ('man', 'wo
for triad in triads_to_try:
    print ('{} -> {} :: {} -> {}'.format( *triad, complete_analogy(*triad,word_to_vec_ma
```

```
italy -> italian :: spain -> spanish
india -> delhi :: japan -> tokyo
man -> woman :: boy -> girl
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->?.

In [15]:
```
triads_to_try = [('small', 'big', 'smaller')]
for triad in triads_to_try:
    print ('{} -> {} :: {} -> {}'.format( *triad, complete_analogy(*triad,word_to_vec_ma
```

```
small -> big :: smaller -> bigger
```

- The algorithm provides the incorrect result "bigger" for the analogy "small is to big as smaller is to bigger." The correct answer should be small -> smaller :: big -> bigger.

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

In [16]:
```
g = word_to_vec_map['woman'] - word_to_vec_map['man']

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

---

In [17]:
```
print ('List of names and their similarities with constructed vector:')
name_list = ['john', 'marie', 'sophie', 'ronaldo', 'priya', 'rahul', 'danielle', 'reza',

for i in name_list:
    print (i, cosine_similarity(word_to_vec_map[i], g))
```

```
List of names and their similarities with constructed vector:
john -0.23163356145973724
marie 0.315597935396073
sophie 0.31868789859418784
ronaldo -0.3124479685032943
priya 0.17632041839009405
rahul -0.1691547103923172
danielle 0.24393299216283892
reza -0.0793042967219955
katy 0.2831068659572615
yasmin 0.23313857767928753
```

TASK 7: What do you observe?

We can notice that the constructed vector exhibits a positive cosine similarity with all female first names, while it tends to show a negative cosine similarity with male first names. This observation seems to be as expected and accurate.

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

In [18]:
```
print('Other words and their similarities:')
word_list = ['lipstick', 'guns', 'science', 'arts', 'literature', 'warrior','doctor', 't
             'technology',  'fashion', 'teacher', 'engineer', 'pilot', 'computer', 'sing
```

```python
for i in word_list:
    print (i, cosine_similarity(word_to_vec_map[i], g))
```

```
Other words and their similarities:
lipstick 0.2769191625638267
guns -0.18884855678988974
science -0.06082906540929699
arts 0.008189312385880035
literature 0.06472504433459929
warrior -0.20920164641125288
doctor 0.11895289410935041
tree -0.07089399175478091
receptionist 0.33077941750593744
technology -0.131937324475543296
fashion 0.03563894625772699
teacher 0.17920923431825667
engineer -0.08039280494524072
pilot 0.0010764498991917154
computer -0.10330358873850497
singer 0.18500518136496288
```

TASK 9: What do you observe?

- The findings indicate that certain words are consistently stereotyped to particular genders. For instance, words like "computer" and "engineer" tend to be associated with the male gender (Negative Values), while "fashion" and "teacher" are linked to the female gender (Positive Values).
- To prevent discrimination and bias, it is crucial to neutralize these words and make them gender-neutral rather than specific to a particular gender.

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 $e_{receptionist}^{debiased}$.

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.

Figure 2: The word vector for "receptionist" represented before and after applying the neutralize operation.

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 $e^{debiased}$:

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

$$e^{debiased} = e - e^{bias\_component} \tag{3}$$

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.

In [19]:
```
# TASK 10
# GRADED neutralize

def neutralize(word, g, word_to_vec_map):
    e = word_to_vec_map[word]
    bias_component = np.dot(e, g) / np.linalg.norm(g)**2 * g
    e_debiased = e - bias_component
    return e_debiased
```

In [20]:
```
e = "receptionist"
print("cosine similarity between " + e + " and g, before neutralizing: ", cosine_similar

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

```
cosine similarity between receptionist and g, before neutralizing:  0.3307794175059374
cosine similarity between receptionist and g, after neutralizing:  -5.256929099191626e-
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 $e_{receptionist}^{debiased}$, 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:

$$\mu = \frac{e_{w1} + e_{w2}}{2} \tag{4}$$

$$\mu_{B} = \frac{\mu \cdot bias\_axis}{||bias\_axis||_2^2} * bias\_axis \tag{5}$$

$$\mu_{\perp} = \mu - \mu_{B} \tag{6}$$

$$e_{w1B} = \frac{e_{w1} \cdot bias\_axis}{||bias\_axis||_2^2} * bias\_axis \tag{7}$$

$$e_{w2B} = \frac{e_{w2} \cdot bias\_axis}{||bias\_axis||_2^2} * bias\_axis \tag{8}$$

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

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

$$e_1 = e_{w1B}^{corrected} + \mu_{\perp} \tag{11}$$

$$e_2 = e_{w2B}^{corrected} + \mu_{\perp} \tag{12}$$

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

In [21]:
```
def equalize(pair, bias_axis, word_to_vec_map):

    # Get the word vectors for the pair
    w1, w2 = pair
    e_w1, e_w2 = word_to_vec_map[w1], word_to_vec_map[w2]

    # Compute the mean vector
    mu = (e_w1 + e_w2) / 2

    # Compute the bias components
    mu_B = (np.dot(mu, bias_axis) / np.linalg.norm(bias_axis)**2) * bias_axis
    mu_ortho = mu - mu_B

    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

    # Correct the bias components
    correction_term = np.sqrt(abs(1 - np.linalg.norm(mu_ortho)**2))
    e_w1B_corrected = correction_term * (e_w1B - mu_B) / np.linalg.norm((e_w1 - mu_ortho
    e_w2B_corrected = correction_term * (e_w2B - mu_B) / np.linalg.norm((e_w2 - mu_ortho

    # Compute the final equalized word vectors
    e1 = e_w1B_corrected + mu_ortho
    e2 = e_w2B_corrected + mu_ortho

    return e1, e2
```

In [22]:
```
print("cosine similarities before equalizing:")
print("cosine_similarity(word_to_vec_map[\"man\"], gender) = ", cosine_similarity(word_t
print("cosine_similarity(word_to_vec_map[\"woman\"], gender) = ", cosine_similarity(word
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) =  0.3566661884627037

cosine similarities after equalizing:
cosine_similarity(e1, gender) =  -0.7004364289309389
cosine_similarity(e2, gender) =  0.7004364289309386
```

TASK 12: What do you abserve?

1. **Before applying the equalization process, the word vectors for "man" and "woman" had limited associations with the gender direction vector, showing slightly negative and slightly positive correlations, respectively. This suggested the presence of some gender bias in the original word embeddings.**

2. **However, after applying the equalization algorithm, the equalized word vectors, denoted as e1 and e2, exhibited a strong and equal correlation with the gender direction vector. Both e1 and e2 became equidistant from the gender direction, with cosine similarities of approximately 0.7 in magnitude. This successful equalization process effectively eliminated the gender bias from the word vectors, rendering them more neutral in relation to the specified gender bias axis.**

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.

In [23]:
```
# function to compute the bias direction g by averaging multiple gender-specific word pa

def compute_gender_direction(gender_specific_word_pairs, word_to_vec_map):

    gender_direction_sum = np.zeros_like(word_to_vec_map["woman"])
    num_pairs = len(gender_specific_word_pairs)

    for pair in gender_specific_word_pairs:
        w1, w2 = pair
        e_w1, e_w2 = word_to_vec_map[w1], word_to_vec_map[w2]
        gender_direction_sum += e_w1 - e_w2

    gender_direction = gender_direction_sum / num_pairs
    return gender_direction
```

In [24]:
```
# Example
gender_specific_word_pairs = [("woman", "man"), ("mother", "father"), ("girl", "boy")]
gender_direction = compute_gender_direction(gender_specific_word_pairs, word_to_vec_map)

# Debias the word vectors for "actress" and "actor":
e_actress_debiased = neutralize("actress", gender_direction, word_to_vec_map)
e_actor_debiased = neutralize("actor", gender_direction, word_to_vec_map)

# And let's equalize the pair "actress" and "actor":
e1, e2 = equalize(("actress", "actor"), gender_direction, word_to_vec_map)
```

In [25]:
```
print("Cosine similarity between e_actress_debiased and gender:", np.dot(e_actress_debia
print("Cosine similarity between e_actor_debiased and gender:", np.dot(e_actor_debiased,
print("Cosine similarity between e1 and gender:", np.dot(e1, gender_direction) / (np.lin
print("Cosine similarity between e2 and gender:", np.dot(e2, gender_direction) / (np.lin
```

```
Cosine similarity between e_actress_debiased and gender: 3.715764414110377e-17
Cosine similarity between e_actor_debiased and gender: -3.7097052530631785e-17
Cosine similarity between e1 and gender: 0.6423428174500829
Cosine similarity between e2 and gender: -0.642342817450083
```

Observation

1. **After debiasing, the word vectors for "actress" and "actor" show cosine similarities very close to zero with the gender direction vector "gender." This implies that these word vectors are now almost independent of the gender direction, successfully achieving gender neutrality.**

2. **Regarding the equalized word vectors e1 and e2, their cosine similarities with the gender direction "gender" are approximately 0.642, and they exhibit opposite signs. The equalization process effectively eliminated the gender bias introduced by gender-specific associations, ensuring that both words now have equal impact on the gender direction.**

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.

In [26]:
```python
# Race_Religion-related words
race_religion_words = ['christian', 'muslim', 'hindu', 'jews', 'muslim', 'black', 'asian

# Calculate cosine similarity between two word vectors
def cosine_similarity(word_vector1, word_vector2):
    cosine_sim = np.dot(word_vector1, word_vector2) / (np.linalg.norm(word_vector1) * np
    return cosine_sim

# List of example words to test for Race & religion bias
example_words = ['criminal', 'homeless', 'war', 'greedy', 'labour', 'intellectually', 'u

# Find the word in religion_words with the highest cosine similarity for each example wo
for example_word in example_words:
    max_similarity = -1
    most_similar_word = None

    example_word_vector = word_to_vec_map[example_word]

    for race_religion_word in race_religion_words:
        race_religion_word_vector = word_to_vec_map[race_religion_word]
        similarity = cosine_similarity(example_word_vector, race_religion_word_vector)

        if similarity > max_similarity:
            max_similarity = similarity
            most_similar_word = race_religion_word

    print(f"The word in 'race_religion_words' with the highest cosine similarity to '{ex
    print('--------------------------------------------------------------')
```

```
The word in 'race_religion_words' with the highest cosine similarity to 'criminal' is 'm
uslim' with a similarity score of 0.44
--------------------------------------------------------------
The word in 'race_religion_words' with the highest cosine similarity to 'homeless' is 'j
ews' with a similarity score of 0.467
--------------------------------------------------------------
The word in 'race_religion_words' with the highest cosine similarity to 'war' is 'musli
m' with a similarity score of 0.56
--------------------------------------------------------------
The word in 'race_religion_words' with the highest cosine similarity to 'greedy' is 'jew
s' with a similarity score of 0.196
--------------------------------------------------------------
The word in 'race_religion_words' with the highest cosine similarity to 'labour' is 'chr
istian' with a similarity score of 0.41
--------------------------------------------------------------
The word in 'race_religion_words' with the highest cosine similarity to 'intellectually'
is 'jews' with a similarity score of 0.168
--------------------------------------------------------------
The word in 'race_religion_words' with the highest cosine similarity to 'uneducated' is
'caucasian' with a similarity score of 0.395
--------------------------------------------------------------
The word in 'race_religion_words' with the highest cosine similarity to 'servicemen' is
'muslim' with a similarity score of 0.454
--------------------------------------------------------------
```

Here is an demonstration of Race & Religion Bias from the word2vec:

- The word 'Criminal' has the highest similarity (similarity score: 0.44) with the word 'muslim', suggesting a certain association between the two in the word vector space.
- The word 'Homeless' shows greater similarity (similarity score: 0.467) with the word 'jews' in the word vector space.
- The word 'War' exhibits a higher similarity (similarity score: 0.56) with the word 'muslim' in the word vector space.
- The word 'Greedy' has somewhat more similarity (similarity score: 0.196) with the word 'jews' in the word vector space.
- The word 'Labour' displays higher similarity (similarity score: 0.41) with the word 'christian' in the word vector space.
- The word 'Intellectually' is somewhat more similar (similarity score: 0.168) with the word 'jews' in the word vector space.
- The word 'Uneducated' is more similar (similarity score: 0.395) with the word 'caucasian' in the word vector space.
- The word 'Servicemen' has somewhat more similarity (similarity score: 0.454) with the word 'muslim' in the word vector space.

```
In [27]: #Performing word analogies to demonstrate race and religion bias

         religious_triads_to_try = [('jew', 'greedy', 'muslim'),('muslim', 'powerless', 'jew'),('
                                     ('muslim', 'warzone', 'christian'),('muslim', 'uneducated', '

         for triad in religious_triads_to_try:
             print ('{} -> {} :: {} -> {}'.format( *triad, complete_analogy(*triad,word_to_vec_ma
```

```
jew -> greedy :: muslim -> warlords
muslim -> powerless :: jew -> unemployable
christian -> familial :: muslim -> vendettas
muslim -> warzone :: christian -> zuleika
muslim -> uneducated :: christian -> naïve
christian -> intellectually :: muslim -> mentally
```

- Thus the word analogies demonstrate the race and religion bias in the word2vec space.

```
In [28]: #Implement the algorithm used to identify the bias subspace described by Manzini et al.

         # Given target words related to races and religions
         target_words = ['asian', 'african', 'caucasian', 'muslim', 'christian', 'hindu' , 'jew']

         # Given attribute words (neutral)
         attribute_words = ['computer', 'science', 'music', 'food', 'book', 'water','dance']

         # Compute the mean vector for target words
         mean_target_vector = np.mean([word_to_vec_map[word] for word in target_words], axis=0)

         # Compute the mean vector for attribute words
         mean_attribute_vector = np.mean([word_to_vec_map[word] for word in attribute_words], axi
```

```
In [29]: #Implement the algorithm used to identify the bias subspace described by Manzini et al.

         # Compute the bias subspace using PCA
         def compute_bias_subspace(target_vectors, attribute_vectors, num_components):
             all_vectors = np.vstack((target_vectors, attribute_vectors))
             pca = PCA(n_components=num_components)
             pca.fit(all_vectors)
             bias_subspace = pca.components_
             return bias_subspace

         # Set the number of components for the bias subspace
         num_components = 10

         # Compute the bias subspace using PCA
         bias_subspace = compute_bias_subspace([word_to_vec_map[word] for word in target_words],
                                               [word_to_vec_map[word] for word in attribute_words
                                               num_components)
```

```
In [30]: print(bias_subspace)
```

```
[[ 1.07314233e-02 -8.66829787e-02 -1.01744847e-01 -2.08941816e-01
   2.85773637e-01  1.71047862e-01  1.10909897e-01  6.01538215e-02
  -2.48640958e-01 -1.10256442e-01  9.16552589e-03 -2.57757119e-01
   4.14646879e-02 -3.40094735e-02 -8.80672177e-02  7.15423106e-03
   6.75616891e-02 -1.42633252e-01  1.91279029e-02  2.91683120e-01
  -2.77685195e-01  1.31958220e-02 -1.07836105e-01  1.79437795e-01
  -7.46029854e-02 -4.05776678e-02  8.22602301e-03 -8.36757907e-02
  -1.10128802e-01  4.73040251e-02 -3.05051363e-01  7.37566958e-02
  -1.02875646e-01  1.75194557e-01 -3.08212469e-02 -1.12457377e-01
  -2.37478164e-02 -2.51836442e-01 -2.08755513e-01  1.78209994e-01
  -8.36462710e-02  5.61087742e-02 -7.44272463e-02  1.62677871e-02
   2.90861252e-02 -5.71881719e-02 -1.82906475e-01  7.49000377e-03
  -2.56618486e-02 -2.30858560e-01]
```

```
[-1.31503653e-01  1.21795393e-01 -2.87179601e-02 -1.19110294e-01
  3.66229777e-02  1.05924266e-01 -8.55542842e-02 -2.72894320e-03
 -2.24305851e-01  2.52636171e-01 -1.58658619e-02  6.95882289e-02
 -2.16994170e-01  1.45777324e-01  2.64677143e-02 -1.79371983e-01
 -8.65732497e-02 -2.49401214e-02  4.83804983e-02  9.68171922e-02
  8.53242669e-02  9.43783433e-02 -1.41000869e-01  1.58633269e-01
  9.46022652e-02 -1.56451584e-02 -3.30471638e-01 -1.34731602e-01
 -7.85234462e-02 -2.80141713e-01 -1.50901316e-01 -2.10583657e-01
  4.81807238e-02 -1.47363086e-01 -8.90894786e-02 -4.37517925e-02
  1.80446168e-01  3.44715394e-02 -3.24132084e-02  2.78713063e-02
  2.04618808e-01 -4.72156020e-02 -2.56786307e-01  1.03739053e-02
 -1.62872208e-01 -6.49343971e-02  1.63631214e-01 -9.56683160e-02
 -2.14043999e-01  2.25935073e-01]
[-2.25805596e-01  1.12613818e-01 -3.74121499e-01  1.95704896e-01
 -2.13549041e-01 -9.60812075e-03  4.76055384e-02  1.81503686e-01
 -1.10090452e-01  7.48855092e-02  1.68943842e-01 -7.84787646e-02
  8.63691892e-02  1.39317772e-01 -1.20922923e-01 -5.90924650e-02
  1.55643326e-01  1.66112896e-02 -1.07175069e-01 -1.69420611e-01
 -1.83627925e-02  1.29813052e-01  3.99875015e-02  4.09261763e-02
 -1.98573860e-02  2.62209678e-01 -4.42164387e-02  3.05865750e-02
 -2.09435678e-01 -8.14137886e-02  1.19244715e-01  1.43759761e-01
  1.07620674e-01  1.27802067e-01  7.20869579e-02 -1.06170523e-01
 -4.02542563e-02 -1.00457215e-01 -1.93832694e-01 -1.37152246e-01
 -1.85979329e-01 -4.60932384e-02  9.58305006e-02 -3.38593312e-01
  2.96680555e-02 -5.89568042e-02  7.23472637e-02 -1.73105983e-01
 -1.54429734e-01 -2.74051966e-02]
[ 1.98691365e-02  1.13400679e-01 -1.00883522e-01 -2.67275258e-01
 -3.27537725e-03  1.73354555e-01  2.74416391e-02  2.61698841e-01
  6.56052619e-03 -4.69037580e-02  8.59537618e-02  3.72528969e-02
  2.15384391e-01  3.14587498e-02  4.07918272e-02 -2.84314084e-02
  1.29555964e-01  2.07228751e-02  7.21478403e-02 -1.33028045e-01
 -1.15110692e-01 -5.34110088e-02  1.44599400e-01  3.18178045e-02
 -5.49339633e-02  8.17528242e-02  5.66990430e-03  3.56728448e-01
  4.34157941e-01 -8.04700792e-02 -2.50873669e-01 -4.50643381e-02
 -8.04964388e-02  1.31391205e-01 -5.50595301e-02  8.84588451e-03
  6.45045801e-02  1.05268778e-01  2.39191613e-01  7.10722445e-02
  1.79241106e-01  1.16222890e-02  1.54518546e-01 -1.51881782e-01
 -1.04680686e-01 -1.49661645e-01 -8.33870442e-02 -8.44406100e-02
 -3.91291280e-02  1.93275849e-01]
[-1.19890683e-01  1.01110771e-02  1.52976084e-01  1.64137298e-01
  4.24587700e-02  2.62427604e-02  5.84545016e-02 -7.31418394e-02
 -1.54683275e-01 -2.31059296e-01  1.59575580e-01 -2.48781106e-02
 -2.09124891e-02  1.27749117e-01 -7.28187287e-02 -7.77315974e-02
 -1.42139892e-01  1.17425952e-01  4.07459914e-02  1.57373517e-01
 -2.02535749e-01 -6.06775147e-02 -2.39226687e-02 -3.30969227e-02
  5.93145853e-04 -5.97587570e-02  3.43236781e-02 -4.92446398e-02
 -5.38849883e-02  1.07038421e-02 -3.01346307e-01 -4.45633130e-02
  2.57736868e-02 -1.41315336e-01  4.78414135e-02 -1.37573655e-01
 -2.87759558e-01  1.99454130e-01 -1.71733626e-02 -2.22678388e-01
 -3.49758132e-04 -1.09025831e-01  4.01372846e-01 -7.83770312e-02
 -7.06181114e-02 -6.11251651e-02  2.67561914e-01  2.85256788e-01
 -1.13697067e-02  1.65481224e-01]
[ 7.34990509e-02 -6.79899964e-04  1.60874357e-01  8.13111482e-02
 -2.05936752e-01  7.66296402e-02  2.82063671e-01  1.50285514e-02
 -1.95596396e-01  1.39174498e-02 -3.13821039e-02 -3.79771422e-02
  1.84681832e-01  1.17803468e-01 -2.53470480e-01  1.63437076e-01
  2.08501185e-02  3.43090654e-01 -1.92059115e-01  1.73959737e-02
  5.48456809e-02 -8.15328367e-02  3.87211462e-02 -8.24240147e-02
  2.48452535e-01  8.51140959e-02 -4.21988610e-02  1.15974651e-01
 -4.51938774e-02 -1.15268065e-01 -1.67109708e-01 -5.80468257e-02
 -5.96386503e-02  1.87922115e-03  3.60433507e-02  1.30228147e-01
  8.11166379e-02 -6.17180962e-02  1.25159882e-01  8.58920524e-02
 -1.55371834e-01 -9.33996628e-02 -1.54634107e-01  2.56671051e-01
 -1.54261544e-01 -1.24557535e-01  2.04472723e-01 -1.21585678e-01
  1.57881081e-01 -2.26642679e-01]
[ 4.19649191e-02  3.10141887e-01 -2.43482102e-01 -1.56484326e-01
```

```
  -7.06217653e-02  1.19187234e-01 -8.24986961e-02 -1.67952242e-01
   7.90609509e-02 -1.87398325e-01 -8.72042262e-02  1.81762287e-01
  -6.32125895e-02 -2.24080052e-01 -1.15467408e-01 -1.32885639e-01
   3.08373570e-01 -2.26707993e-02 -1.40108726e-01  9.28294235e-02
  -1.08563153e-01  9.65187168e-02  1.13465425e-01 -1.89812479e-01
   1.44741692e-01  4.68440375e-02 -7.04391900e-03  7.51785647e-03
  -2.98935287e-01  1.24433386e-01 -1.66214094e-02 -1.07047087e-01
  -1.67828039e-01 -2.10974430e-01 -8.47853215e-02 -5.02634930e-02
  -4.41378774e-02  4.67643899e-02  1.55905630e-01  8.03411733e-02
   5.68030890e-02 -8.46301849e-02 -1.98464172e-02 -1.27045877e-01
  -2.49359785e-01  1.66576245e-01 -4.49048526e-02  1.01995887e-01
   7.36200384e-02 -1.05790301e-01]
 [-1.73266258e-01  2.28212801e-01 -2.20113239e-01 -8.65147970e-03
   1.80357490e-01 -8.74144838e-02  1.38742919e-01 -6.11108845e-03
  -6.56666412e-02  4.30061099e-02 -1.41803161e-01  1.03097504e-01
   4.07731825e-02  2.02523516e-01  2.85040113e-01  8.30915206e-02
   2.24515826e-01 -3.00770416e-02 -1.94254765e-01  2.43108211e-01
   3.34605583e-01 -9.77204944e-02  5.31040858e-02 -6.65360925e-02
   1.08540209e-01 -2.16865447e-01  4.72746073e-02 -1.72638554e-02
   7.30399435e-02  4.33099505e-02 -1.21388229e-01  7.28800709e-02
  -1.72385706e-02 -2.53418951e-02 -1.48870857e-01 -5.10674924e-02
   9.44129349e-02  1.02478036e-01 -9.06112422e-02 -6.29112736e-02
  -2.13736190e-01 -1.09943907e-01  7.13103875e-02  5.65010289e-02
   2.58973984e-01 -1.11247067e-01 -5.12279706e-02  3.61887268e-03
   2.22244069e-01  1.47865182e-01]
 [ 1.57426028e-01 -5.52667553e-03 -5.41311558e-02  1.09846805e-01
   1.44766423e-02  1.22356567e-01 -5.19237545e-02 -6.41460832e-02
   4.44326684e-02 -1.03693162e-01 -3.05808244e-01  1.80751653e-01
   4.78773801e-02 -1.38820362e-01  3.30239226e-01 -1.23007042e-01
  -1.90539037e-01 -1.25707301e-01 -3.67465193e-02 -3.56233648e-02
  -3.15258740e-02 -1.79659651e-01  1.30979268e-01 -2.35224978e-02
   1.64076626e-01  1.02366163e-01 -2.58924496e-01  1.52535427e-01
   1.06015365e-02 -3.09497414e-02 -1.26367579e-01 -1.68191373e-01
   1.38099594e-01  3.44556474e-01  2.73659106e-02 -1.29310356e-02
  -7.12829913e-02 -1.55682395e-01 -2.42546714e-01 -2.90151639e-01
   1.46686757e-02 -2.82168079e-02 -6.36270404e-04 -3.68068384e-02
  -9.53171320e-02 -1.05839819e-02  1.21526478e-01 -2.78534546e-03
   7.64419196e-02 -1.77654334e-01]
 [ 2.10944315e-04  3.42736472e-02 -1.35397523e-01 -7.48986851e-02
  -3.19716222e-02 -4.57323978e-02  1.74732044e-01  2.08127316e-02
   3.06035705e-01 -5.59329381e-02 -1.76706785e-01 -1.99421045e-01
   1.57203694e-01 -5.94559170e-03 -1.12406747e-01  7.99573717e-03
  -9.06564050e-02  1.28180083e-01  2.17200890e-01  2.57701371e-01
   2.20797886e-01  8.86425973e-02  2.95413730e-01  4.74136526e-02
  -1.06275095e-01 -2.35171496e-02 -4.18030690e-01  1.11828327e-02
   3.14183367e-02 -4.21352702e-02  4.94520454e-02  1.72824948e-02
   3.21296865e-02 -5.11425573e-02  1.23913202e-01  1.38068430e-01
  -1.45142284e-01 -9.76864259e-03 -2.88448214e-02  1.84062273e-01
  -1.71993360e-01  2.91136497e-02 -1.45259315e-02 -4.42719529e-02
  -2.21790945e-02 -8.95725217e-03  2.49387365e-02  2.88272169e-01
  -2.08704153e-01  6.77938943e-02]]
```

In [31]:

```python
#3. Use the Neutralize and Equalize debiasing debiasing you have implemented above and s

#Neutralising

g = word_to_vec_map['greedy']

e = "jew"
print("cosine similarity between " + e + " and greedy, before neutralizing: ", cosine_si

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

```
cosine similarity between jew and greedy, before neutralizing:  0.34333884817210963
cosine similarity between jew and greedy, after neutralizing:  -6.255810186786585e-17
```

In [32]:
```
#Equalisation

print("cosine similarities before equalizing:")
print("cosine_similarity(word_to_vec_map[\"jew\"], greedy) = ", cosine_similarity(word_t
print("cosine_similarity(word_to_vec_map[\"science\"], greedy) = ", cosine_similarity(wo
print()

e1, e2 = equalize(("jew", "science"), g, word_to_vec_map)
print("cosine similarities after equalizing:")
print("cosine_similarity(jew, greedy) = ", cosine_similarity(e1, g))
print("cosine_similarity(science, greedy) = ", cosine_similarity(e2, g))
```

```
cosine similarities before equalizing:
cosine_similarity(word_to_vec_map["jew"], greedy) =  0.34333884817210963
cosine_similarity(word_to_vec_map["science"], greedy) =  0.020128803443215697

cosine similarities after equalizing:
cosine_similarity(jew, greedy) =  0.21731550346808143
cosine_similarity(science, greedy) =  -0.21731550346808168
```

In [33]:
```
# function to compute the bias direction g by averaging multiple religion-specific word

def compute_race_religion_direction(race_religion_specific_word_pairs, word_to_vec_map):

    race_religion_direction_sum = np.zeros_like(word_to_vec_map["greedy"])
    num_pairs = len(race_religion_specific_word_pairs)

    for pair in race_religion_specific_word_pairs:
        w1, w2 = pair
        e_w1, e_w2 = word_to_vec_map[w1], word_to_vec_map[w2]
        race_religion_direction_sum += e_w1 - e_w2

    race_religion_direction = race_religion_direction_sum / num_pairs
    return race_religion_direction
```

In [34]:
```
#Debiasing for religion

race_religion_specific_word_pairs = [("greedy", "jew"), ("black", "homeless"), ("asian"
race_religion_direction = compute_race_religion_direction(race_religion_specific_word_pa

# Debias the word vectors for "jew" and "science":
e_jew_debiased = neutralize("jew", race_religion_direction, word_to_vec_map)
e_science_debiased = neutralize("science", race_religion_direction, word_to_vec_map)

# And let's equalize the pair "jew" and "science":
e1, e2 = equalize(("jew", "science"), race_religion_direction, word_to_vec_map)
```

In [35]:
```
print("Cosine similarity between e_jew_debiased and religion:", np.dot(e_jew_debiased, r
print("Cosine similarity between e_science_debiased and religion:", np.dot(e_science_deb
print("Cosine similarity between jew and religion:", np.dot(e1, race_religion_direction)
print("Cosine similarity between science and religion:", np.dot(e2, race_religion_direct
```

```
Cosine similarity between e_jew_debiased and religion: 6.583770996789907e-17
Cosine similarity between e_science_debiased and religion: -5.631244470936987e-17
Cosine similarity between jew and religion: -0.3998117730321888
Cosine similarity between science and religion: 0.399811773032189
```

Observation

1. **After debiasing, the word vectors for "jew" and "science" show cosine similarities very close to zero with the race-religion direction vector. This implies that these word vectors are now almost independent of the race-religion direction, successfully achieving religion neutrality.**

2. **Regarding the equalized word vectors e1 and e2, their cosine similarities with the race-religion direction are approximately 0.39, and they exhibit opposite signs. The equalization process effectively eliminated the religion bias introduced by religion-specific associations, ensuring that both words now have equal impact on the race-religion direction.**

In [36]:
```
#Debiasing for race

# Example
race_religion_specific_word_pairs = [("greedy", "jew"), ("black", "homeless"), ("asian"
race_religion_direction = compute_race_religion_direction(race_religion_specific_word_pa

# Debias the word vectors for "black" and "asian":
e_black_debiased = neutralize("black", race_religion_direction, word_to_vec_map)
e_asian_debiased = neutralize("asian", race_religion_direction, word_to_vec_map)

# And let's equalize the pair "black" and "asian":
e1, e2 = equalize(("black", "asian"), race_religion_direction, word_to_vec_map)
```

In [37]:
```
print("Cosine similarity between e_black_debiased and race:", np.dot(e_black_debiased, r
print("Cosine similarity between e_asian_debiased and race:", np.dot(e_asian_debiased, r
print("Cosine similarity between black and race:", np.dot(e1, race_religion_direction) /
print("Cosine similarity between asian and race:", np.dot(e2, race_religion_direction) /
```

```
Cosine similarity between e_black_debiased and race: -4.327548373928614e-17
Cosine similarity between e_asian_debiased and race: -8.621157349891453e-17
Cosine similarity between black and race: -0.19068849238412622
Cosine similarity between asian and race: 0.19068849238412594
```

Observation

1. **After debiasing, the word vectors for "black" and "asian" show cosine similarities very close to zero with the race-religion direction vector. This implies that these word vectors are now almost independent of the race-religion direction, successfully achieving race neutrality.**

2. **Regarding the equalized word vectors e1 and e2, their cosine similarities with the race-religion direction are approximately 0.19, and they exhibit opposite signs. The equalization process effectively eliminated the racial bias introduced by race-specific associations, ensuring that both words now have equal impact on the race-religion direction.**

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/)