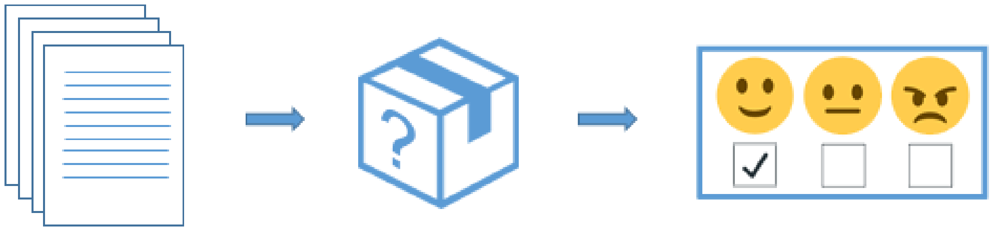
# 3.2HD - Discrimination and Bias in NLP

August 2, 2022

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.

## 1 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.

[1]:
```
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`.

[2]:
```
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.

[3]:
```
print("Example of words: ",list(words)[:10])
print("Vector for word 'person' = ", word_to_vec_map.get('person'))
```

```
Example of words:  ['torrelavega', 'kalle', 'catalysts', 'bodman', 'kathak',
'menelik', 'watsonia', 'ctp', 'cholestasis', 'unmeasurable']
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.

## 2 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:

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

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 $\theta$ 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.

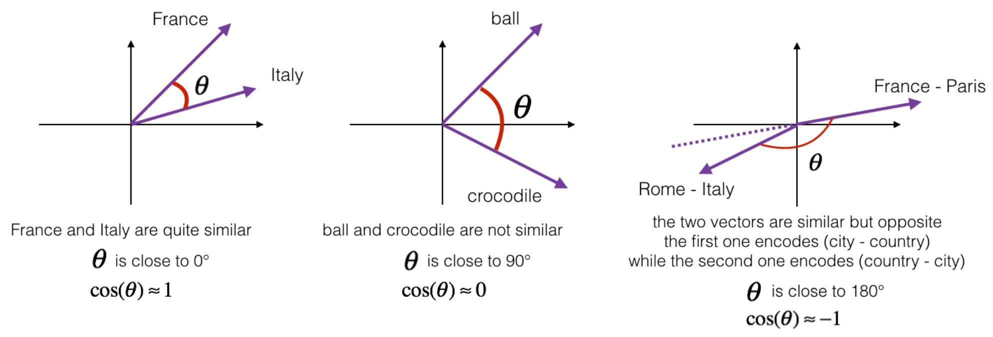

France
Italy
$\theta$
France and Italy are quite similar
$\theta$ is close to 0°
$\cos(\theta) \approx 1$
ball
crocodile
$\theta$
ball and crocodile are not similar
$\theta$ is close to 90°
$\cos(\theta) \approx 0$
France - Paris
Rome - Italy
$\theta$
the two vectors are similar but opposite
the first one encodes (city - country)
while the second one encodes (country - city)
$\theta$ is close to 180°
$\cos(\theta) \approx -1$

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 $||u||_2 = \sqrt{\sum_{i=1}^{n} u_i^2}$

[4]:
```python
## 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
    ## Task 1
# cosine_similarity
    dist = 0
    dot = u@v
    u_norm = np.sqrt(np.sum(u**2))
    v_norm = np.sqrt(np.sum(v**2))
    cosine_similarity = dot/(u_norm*v_norm)
    return cosine_similarity
    ## END
```

---

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

[19]:
```
## Task 2
# GRADED FUNCTION: most_similar_word
from heapq import *
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
    if word_to_vec_map == word_to_vec_map:
        x = word_to_vec_map.keys()
    vec = word_to_vec_map[word]
    sim = [(-100, '') for _ in range(1)]
    for w in x:
        if w == word:
            continue
        cos_sim = cosine_similarity(vec,word_to_vec_map[w])
        heappushpop(sim,(cos_sim, w))
    most_sim = [tup for tup in sim]
    _,best_words = zip(*most_sim)
    return best_words
    ## 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?

[6]:
```
brother = word_to_vec_map["brother"]
friend = word_to_vec_map["friend"]

print(cosine_similarity(brother,friend))
```

```
0.8713178668124657
```

What is the similarity between the words computer and kid?

[7]:
```
computer = word_to_vec_map["computer"]
kid = word_to_vec_map["kid"]

print(cosine_similarity(computer,kid))
```

```
0.4380016621036385
```

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

[17]:
```python
france = word_to_vec_map["france"]
paris = word_to_vec_map["paris"]
rome = word_to_vec_map["rome"]
italy = word_to_vec_map["italy"]
print(cosine_similarity(france - paris , rome - italy))
```

```
-0.6751479308174201
```

What is the most similar word to computer?

[20]:
```python
most_similar_word('computer',word_to_vec_map)
```

[20]:
```
('computers',)
```

What is the most similar word to australia?

[21]:
```python
most_similar_word('australia',word_to_vec_map)
```

[21]:
```
('zealand',)
```

What is the most similar word to python?

[22]:
```python
most_similar_word('python',word_to_vec_map)
```

[22]:
```
('reticulated',)
```

---

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

## 3 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!

---

[28]:
```python
## 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
    word_a, word_b, word_c = word_a.lower(), word_b.lower(), word_c.lower()

    va, vb, vc = word_to_vec_map[word_a], word_to_vec_map[word_b],␣
 ↪word_to_vec_map[word_c]

    wrd = word_to_vec_map.keys()

    max_cos= -100

    best_word = None

    for w in wrd:
        if w in [word_a, word_b, word_c]:
            continue
        cos_sim = cosine_similarity(vb - va, word_to_vec_map[w]-vc)
        if cos_sim > max_cos:
            max_cos = cos_sim
            best_word = w
    return best_word
    ##ENFD
```

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

[29]:
```
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
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->?.

## 4 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

---

[10]:
```python
## 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**

---

[11]:
```python
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 w in name_list:
  print (w, cosine_similarity(word_to_vec_map[w], g))

## 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 -0.16915471039231716
danielle 0.24393299216283895
reza -0.07930429672199553
katy 0.2831068659572615
yasmin 0.23313857767928758
```

---

**TASK 7: What do you observe?**

---

So, we can see that all 'male' names are associated with a negative value and all 'female' names are asscoiated with a positive value. However, since we are operating on cosine distance, the negative and positive values are indicators of gender but the numerical value, regardless polarity, still represents the distance from the g vector.

---

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

---

[12]:
```
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 w in word_list:
  print(w,cosine_similarity(word_to_vec_map[w],g))
```

```
## END
```

```
Other words and their similarities:
lipstick 0.2769191625638267
guns -0.1888485567898898
science -0.06082906540929701
arts 0.008189312385880337
literature 0.06472504433459932
warrior -0.20920164641125288
doctor 0.11895289410935041
tree -0.07089399175478091
receptionist 0.33077941750593737
technology -0.13193732447554302
fashion 0.03563894625772699
teacher 0.17920923431825664
engineer -0.0803928049452407
pilot 0.0010764498991916937
computer -0.10330358873850498
singer 0.1850051813649629
```

---

**TASK 9: What do you observe?**

---

As stated above, all 'male' names are associated with a negative value and all 'female' names are asscoiated with a positive value. So in this result, we can assume that any word that has a negative value is attributed to being identifiable with the male gender, examples includes; guns at -0.188 , warrior at -0.209 & technology at -0.131.

We can then also assume that any word that has a positive value is attributed to being identifiable with the female gender, examples includes; lipstick at 0.276 , fashion at 0.035 & singer at 0.185.

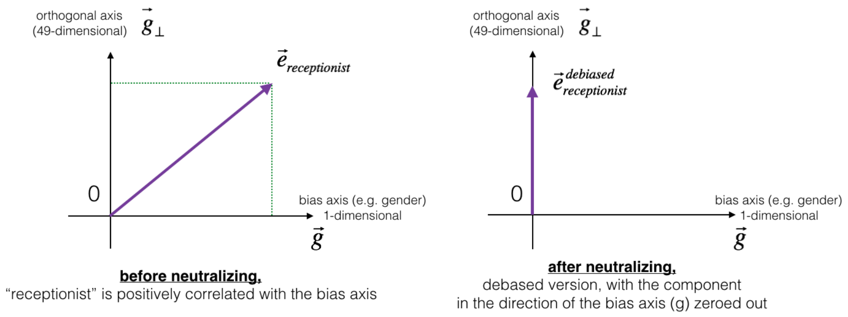
---

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.

## 5 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.



orthogonal axis (49-dimensional) $\vec{g}_\perp$
$\vec{e}_{receptionist}$
0
bias axis (e.g. gender) 1-dimensional
$\vec{g}$
before neutralizing.
"receptionist" is positively correlated with the bias axis
orthogonal axis (49-dimensional) $\vec{g}_\perp$
$\vec{e}^{debiased}_{receptionist}$
0
bias axis (e.g. gender) 1-dimensional
$\vec{g}$
after neutralizing.
debased version, with the component in the direction of the bias axis (g) zeroed out

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.

---

[13]:
```
# 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_biascomp = np.divide(np.dot(e,g),np.linalg.norm(g)**2) * g
    e_debias = e - e_biascomp
    return e_debias
    ## END
```

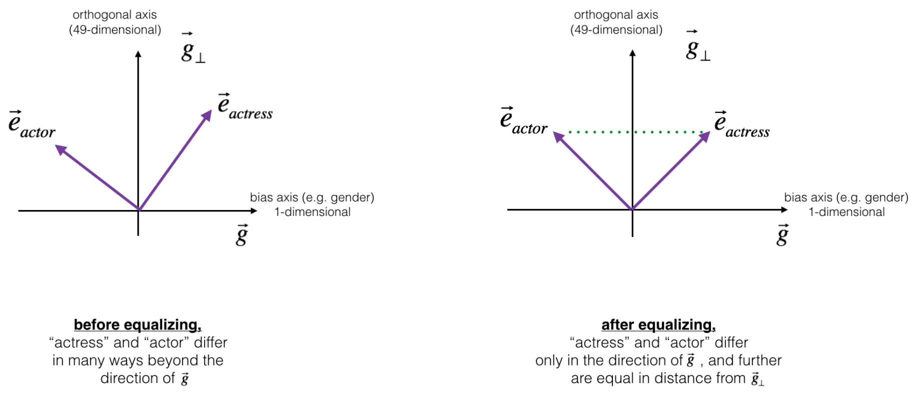
[14]:
```
e = "receptionist"
print("cosine similarity between " + e + " and g, before neutralizing: ",␣
 ↪cosine_similarity(word_to_vec_map["receptionist"], g))

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:
0.33077941750593737
cosine similarity between receptionist and g, after neutralizing:
-2.099120994400013e-17
```

## 6 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 equidistant 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:


orthogonal axis (49-dimensional)
$\vec{g}_\perp$
$\vec{e}_{actor}$
$\vec{e}_{actress}$
bias axis (e.g. gender) 1-dimensional
$\vec{g}$
before equalizing,
"actress" and "actor" differ in many ways beyond the direction of $\vec{g}$
orthogonal axis (49-dimensional)
$\vec{g}_\perp$
$\vec{e}_{actor}$
$\vec{e}_{actress}$
bias axis (e.g. gender) 1-dimensional
$\vec{g}$
after equalizing,
"actress" and "actor" differ only in the direction of $\vec{g}$, and further are equal in distance from $\vec{g}_\perp$

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!

---

[35]:
```
# 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
    ew1, ew2 = word_to_vec_map[w1], word_to_vec_map[w2]
    muu = (ew1 + ew2)/2.0
    muB = np.divide(np.dot(muu, bias_axis),np.linalg.
 ↪norm(bias_axis)**2)*bias_axis
    mu0 = muu - muB
    ew1B = np.divide(np.dot(ew1, bias_axis),np.linalg.
 ↪norm(bias_axis)**2)*bias_axis
    ew2B = np.divide(np.dot(ew2, bias_axis),np.linalg.
 ↪norm(bias_axis)**2)*bias_axis
    cew1B = np.sqrt(np.abs(1-np.sum(mu0**2)))*np.divide(ew1B-muB, np.
 ↪abs(ew1-mu0-muB))
    cew2B = np.sqrt(np.abs(1-np.sum(mu0**2)))*np.divide(ew2B-muB, np.
 ↪abs(ew2-mu0-muB))
    e1 = cew1B + mu0
    e2 = cew2B + mu0
    return e1, e2
    ##END
```

[36]:

```
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) =  -0.11711095765336832
cosine_similarity(word_to_vec_map["woman"], gender) =  0.35666618846270376

cosine similarities after equalizing:
cosine_similarity(e1, gender) =  -0.7165727525843935
cosine_similarity(e2, gender) =  0.7396596474928909
```

**TASK 12: What do you abserve?**

---

As we can see, before the equalisation, the cosine similarity for g was showing to be more bias towards 'man" as the distance between g and the man vector was smaller than g and the women vector (-0.117 is closer that 0.35).

However, after equalsing the word vectors, we created almost parity with both new vectors for man and woman becing about equal in cosine similarity (distance from the g vector). While the vector is still leaning towards man, the gap before equalisation has been reduced in distance in regards to the two old vectors and g.

---

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.

## 7 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.

[61]:
```
## START YOU CODE HERE
# 1. Race and Religion
rr = word_to_vec_map['war'] - word_to_vec_map['peace']

races_reli = ['white','black','asian','catholic','jewish','islam','christ']

## START YOU CODE HERE
for w in races_reli:
    print (w, cosine_similarity(word_to_vec_map[w], rr))

# negative values are more inclined towards war and positive are inclined␣
↪towards peace.
#Example: Black is assocaited the most with peace.
#Example: Christ has the highest association with war.
#Example: Catholics are the least inclined with war.
#Example: All religions are inclined toward war.

print()
# 2. Subspace Algorithm
### AVG of b1 = e(female) - e(male), b2 = E(she) - e(he) ...
eu1 = word_to_vec_map["female"]
eu2 = word_to_vec_map["male"]
eu3 = word_to_vec_map["she"]
eu4 = word_to_vec_map["he"]

b1 = eu1 - eu2
b2 = eu3 - eu4
g1 = (b1+b2)/2
print(g1) # This is the gender subspace and can be expanded for each␣
↪combination of gendered terms.
print()

# 3. Nuteralise & Equalise
e = "receptionist"
print("cosine similarity between " + e + " and g1, before neutralizing: ",␣
↪cosine_similarity(word_to_vec_map["receptionist"], g1))
e_debiased = neutralize("receptionist", g, word_to_vec_map)
print("cosine similarity between " + e + " and g1, after neutralizing: ",␣
↪cosine_similarity(e_debiased, g1))
print()
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"), g1, 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))
```

```
white 0.07408925169643356
black 0.1836786425996555
asian 0.02862493497512974
catholic -0.013299893361461437
```

```
jewish -0.014413991761022997
islam -0.07233419652085059
christ -0.15411671925939568

[ 0.088006    0.2618605   0.017345    0.1302     -0.02991     0.350925
 -0.1275096  -0.038395    0.390535   -0.100472   -0.038155   -0.06251
  0.298705    0.36842     0.19924    -0.0485695  -0.23613     0.112435
  0.358725   -0.100509    0.30948     0.377785   -0.00397     0.1239
  0.413735    0.0322     -0.106972    0.0951405  -0.04534    -0.242361
 -0.06985     0.1410445   0.0916415   0.083815   -0.145265    0.051218
 -0.1140125  -0.10261     0.057675   -0.25191    -0.0229275   0.0378925
  0.29886335 -0.547755    0.197972   -0.1703      0.135663   -0.296635
  0.0079655   0.2494135 ]

cosine similarity between receptionist and g1, before neutralizing:
0.41221218107115826
cosine similarity between receptionist and g1, after neutralizing:
0.2093472879244014

cosine similarities before equalizing:
cosine_similarity(word_to_vec_map["man"], gender) =  -0.11711095765336832
cosine_similarity(word_to_vec_map["woman"], gender) =  0.35666618846270376

cosine similarities after equalizing:
cosine_similarity(e1, gender) =  -0.03777400280259533
cosine_similarity(e2, gender) =  0.03688905113796094
```

## 8 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!

## 9 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/)

[ ]:

```
```