Operations-on-Word-Vectors

In this small post, I am applying some algebraic techniques to reduce bias (specifically gender bias) in word embeddings, which is an extremely important consideration in Machine Learning.

Motivation

The blind application of machine learning runs the risk of amplifying biases present in data. Such a danger is facing us with word embedding, a popular framework to represent text data as vectors which has been used in many machine learning and natural language processing tasks. Geometrically, gender bias is shown to be captured by a direction in the word embedding. Second, gender neutral words are shown to be linearly separable from gender definition words in the word embedding. Therefore, with a empirically sound technique, we could modify an embedding to remove gender stereotypes, such as the association between the word *receptionist* and *female*, while maintaining desired associations such as between the word *queen* and *female*.

What is Word Embedding?

A word embedding, trained on word co-occurrence in text corpora, represents each word (or common phrase) w as a d-dimensional word vector $\vec{w} \in \mathbb{R}^d$. It serves as a dictionary of sorts for computer programs that would like to use word meaning.

Word embeddings have two key properties:

1. Words with similar semantic meanings tend to have vectors that are close together.
2. The vector differences between words in embeddings have been shown to represent relationships between words.

Given an analogy puzzle, "man is to king as woman is to x" (denoted as $man : king :: woman : x$), simple arithmetic of the embedding vectors finds that x=queen is the best answer because:

$$\vec{\text{man}} - \vec{\text{woman}} \approx \vec{\text{king}} - \vec{\text{queen}}$$

Similarly, $x = Japan$ is returned for $Paris : France :: Tokyo : x$

However, it is also the case that

$$\vec{\text{man}} - \vec{\text{woman}} \approx \text{computer }\vec{\text{programmer}} - \vec{homemaker}$$

Methodology

To tackle this problem, we would use the **gender specific words** such as *brother, sister, businessman* and *businesswoman* to learn a gender subspace in the embedding, and the debiasing algorithm removes the bias only from the **gender neutral words** while respecting the definition of these gender specific words.

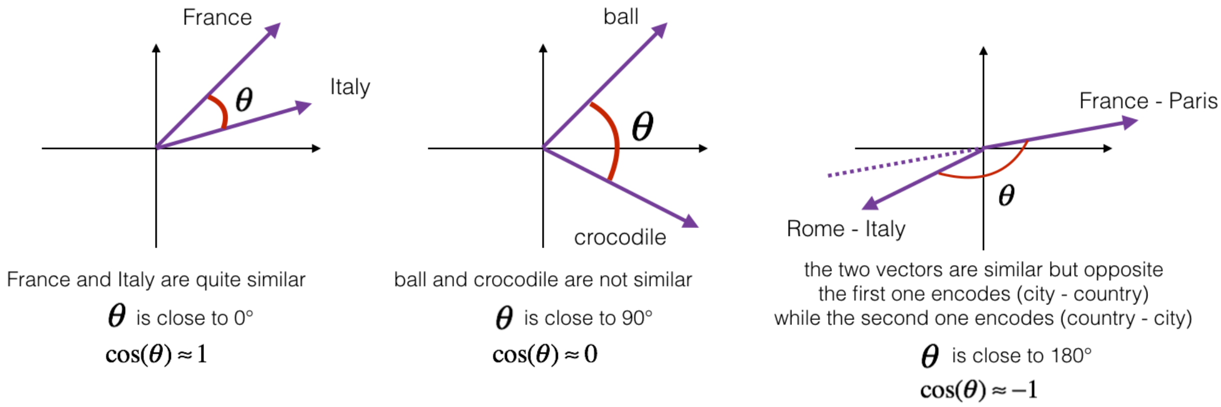
Cosine Similarity

To measure the similarity between 2 words is to measure the degree of similarity between 2 embedding vectors for the 2 words. Given two vectors u and v, cosine similarity is defined as follows:

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

Where:

- $u \cdot v$ is the dot product (or inner product) of two vectors
- $||u||_2$ is the norm (or length) of the vector u
- θ is the angle between u and v.
- The cosine 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.




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

        """

        # Special case. Consider the case u = [0, 0], v=[0, 0]

        if np.all(u == v):
                return 1

        dot = np.dot(u, v)

        norm_u = np.sqrt(np.sum(u**2))
        norm_v = np.sqrt(np.sum(v**2))

        # Avoid division by 0
        if np.isclose(norm_u * norm_v, 0, atol=1e-32):
                return 0

        cosine_similarity = dot / (norm_u * norm_v)

        return cosine_similarity
```

In the word analogy task, 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:

$$e_b - e_a \approx e_d - e_c$$

```
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
        """

        # Convert to lowercase

        word_a, word_b, word_c = word_a.lower(), word_b.lower(),
word_c.lower()

        e_a, e_b, e_c = word_to_vec_map[word_a], word_to_vec_map[word_b],
word_to_vec_map[word_c]

        words = word_to_vec_map.keys()

        max_cosine_sim = -100 # Initialize max_cosine_sim to a large
negative number

        best_word = None # Initialize best_word with None, it will help
keep track of the word to output

        # loop over the whole word vector set
        for w in words:
                if w == word_c:
                continue

        cosine_sim = cosine_similarity(e_b - e_a, word_to_vec_map[w] -
e_c)
        if cosine_sim > max_cosine_sim:
                max_cosine_sim = cosine_sim
                best_word = w

        return best_word
```

Cosine similarity is a relatively simple and intuitive, yet powerful, method you can use to capture nuanced relationships between words. Now, let's apply this formula to tackle a well-known problem.

Debiasing Algorithm

First, see how the GloVe word embeddings relate to gender. You'll begin by computing 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". Needless to say that you would 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.

```
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, consider the cosine similarity of different words with g. What does a positive value of similarity mean, versus a negative cosine similarity?

```
name_list = ['john', 'marie', 'sophie', 'ronaldo', 'priya', 'rahul',
'danielle', 'reza', 'katy', 'yasmin']

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

```
List of names and their similarities with constructed vector:
john -0.23163356145973724
marie 0.315597935396073
sophie 0.31868789859418784
ronaldo -0.31244796850329437
priya 0.17632041839009405
rahul -0.16915471039231725
danielle 0.24393299216283895
reza -0.07930429672199554
katy 0.2831068659572615
yasmin 0.2331385776792876
```

We see that female first names tend to have a positive cosine similarity with our constructed vector g, while male first names tend to have a negative cosine similarity. This is not surprising, and the result seems acceptable.

However, when we compute these words, we got:

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

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

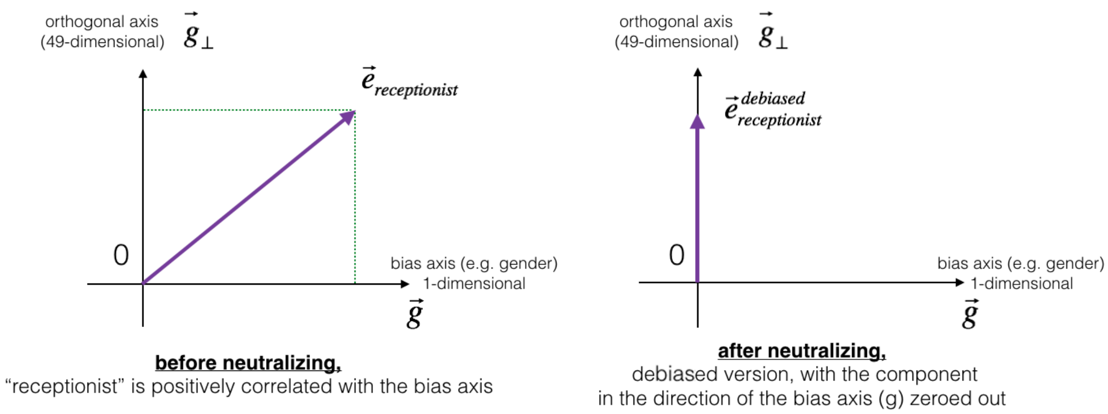
```
Other words and their similarities:
lipstick 0.2769191625638267
guns -0.1888485567898898
science -0.06082906540929701
arts 0.00818931238588035
literature 0.06472504433459932
warrior -0.2092016464112529
doctor 0.1189528941093504
tree -0.07089399175478088
receptionist 0.3307794175059374
technology -0.131937324475543
fashion 0.03563894625772699
teacher 0.1792092343182567
engineer -0.08039280494524072
pilot 0.001076449899191721
computer -0.1033035887385049
singer 0.1850051813649629
```

These results clearly reflect some biases. For example, while "computer" is negative and is closer in value to male first name, "literature" or "reception" are both positive and extremely close to the values with female first names.

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 is called $g_{\perp}$ here. 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 you can draw on a 2D screen, it's illustrated using a 1-dimensional axis below.



Given an input embedding e, you can use the following formulas to compute $e^{debiased}$:

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

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

With some knowledge in linear algebra, $e^{bias\_component}$ is the projection of e onto the direction g.

```
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"

        """

        e = word_to_vec_map[word]

        e_biascomponent = np.dot(e, g) / np.linalg.norm(g)**2 * g

        e_debiased = e - e_biascomponent

        return e_debiased
```

Let's apply on a word

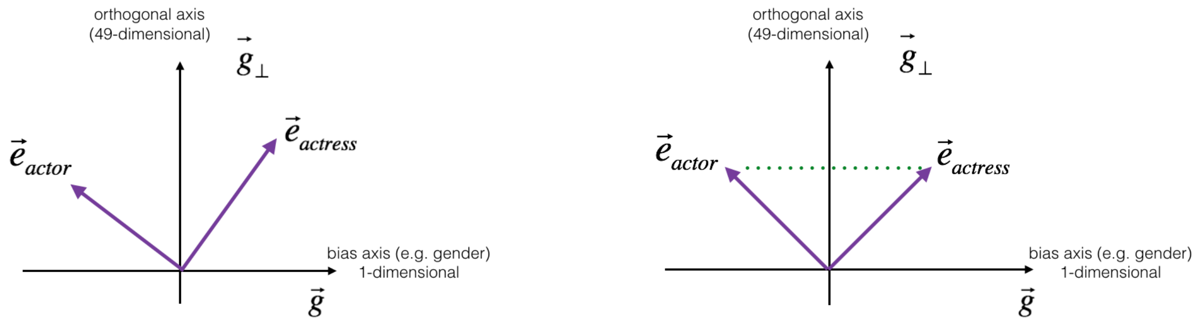
```
word = "receptionist"

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

e_debiased = neutralize(word, g_unit, word_to_vec_map_unit_vectors)

print("cosine similarity between " + word + " and g_unit, after
neutralizing: ", cosine_similarity(e_debiased, g_unit))
```

The results were unbelievably amazing:
cosine similarity between nurse and g, before neutralizing: 0.38030879680687524
cosine similarity between nurse and g_unit, after neutralizing: -1.0714992204190764e-16

Equalization Step

Next, let's 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 neutralization to "babysit," you 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 work that has been neutralized. Visually, this is how equalization works:



before equalizing,
“actress” and “actor” differ in many ways beyond the direction of $\vec{g}$

after equalizing,
“actress” and “actor” differ only in the direction of $\vec{g}$, and further are equal in distance from $\vec{g}_\perp$

Given a pair of words (w_1, w_2), their corresponding word vectors e_{w1} and e_{w2}, and a bias axis vector b, the equalize function performs the following steps:

1. **Compute the mean of the two word vectors:**

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

2. **Compute projections of μ on the bias axis and its orthogonal component:**

$$\mu_B = \frac{(\mu \cdot b)}{(b \cdot b)} b$$

$$\mu_\perp = \mu - \mu_B$$

3. **Compute projections of e_{w1} and e_{w2} on the bias axis:**

$$e_{w1B} = \frac{(e_{w1} \cdot b)}{(b \cdot b)} b$$

$$e_{w2B} = \frac{(e_{w2} \cdot b)}{(b \cdot b)} b$$

4. **Adjust the bias components:**

$$e_{w1B}^{\text{corrected}} = \sqrt{|1 - (\mu_\perp \cdot \mu_\perp)|} \cdot \frac{e_{w1B} - \mu_B}{|e_{w1} - \mu_\perp - \mu_B|}$$

$$e_{w2B}^{\text{corrected}} = \sqrt{|1 - (\mu_\perp \cdot \mu_\perp)|} \cdot \frac{e_{w2B} - \mu_B}{|e_{w2} - \mu_\perp - \mu_B|}$$

5. **Compute the debiased vectors:**

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

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

The function returns the debiased vectors e_1 and e_2.

```
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

        """

        # Step 1: Select word vector representation of "word". Use 
word_to_vec_map. (≈ 2 lines)
        w1, w2 = word_to_vec_map[pair[0]], word_to_vec_map[pair[1]]
        e_w1, e_w2 = w1, w2

        # Step 2: Compute the mean of e_w1 and e_w2 (≈ 1 line)
        mu = (e_w1 + e_w2) / 2.0

        # Step 3: Compute the projections of mu over the bias axis and the 
orthogonal axis (≈ 2 lines)
        mu_B = np.dot(mu, bias_axis) / np.linalg.norm(bias_axis)**2 * 
bias_axis
        mu_orth = mu - mu_B

        # Step 4: Compute e_w1B and e_w2B (≈2 lines)
        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

        # Step 5: Adjust the Bias part of e_w1B and e_w2B
        corrected_e_w1B = np.sqrt(np.abs(1 - np.sum(mu_orth**2))) * (e_w1B
- mu_B) / np.abs(e_w1 - mu_orth - mu_B)
        corrected_e_w2B = np.sqrt(np.abs(1 - np.sum(mu_orth**2))) * (e_w2B
- mu_B) / np.abs(e_w2 - mu_orth - mu_B)


        # Step 6: Debias by equalizing e1 and e2 to the sum of their
corrected projections (≈2 lines)
        e1 = corrected_e_w1B + mu_orth
        e2 = corrected_e_w2B + mu_orth

        return e1, e2
```

Let's evaluate the results

```
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_unit, word_to_vec_map_unit_vectors)
print("cosine similarities after equalizing:")
print("cosine_similarity(e1, gender) = ", cosine_similarity(e1, g_unit))
print("cosine_similarity(e2, gender) = ", cosine_similarity(e2, g_unit))
```

```
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) =  -0.6330327807618356
cosine_similarity(e2, gender) =  0.6613935733392551
```