Debiasing Word Embeddings

What are Word Embeddings?

Word Embeddings are the result of applying dimensionality reduction techniques to words!

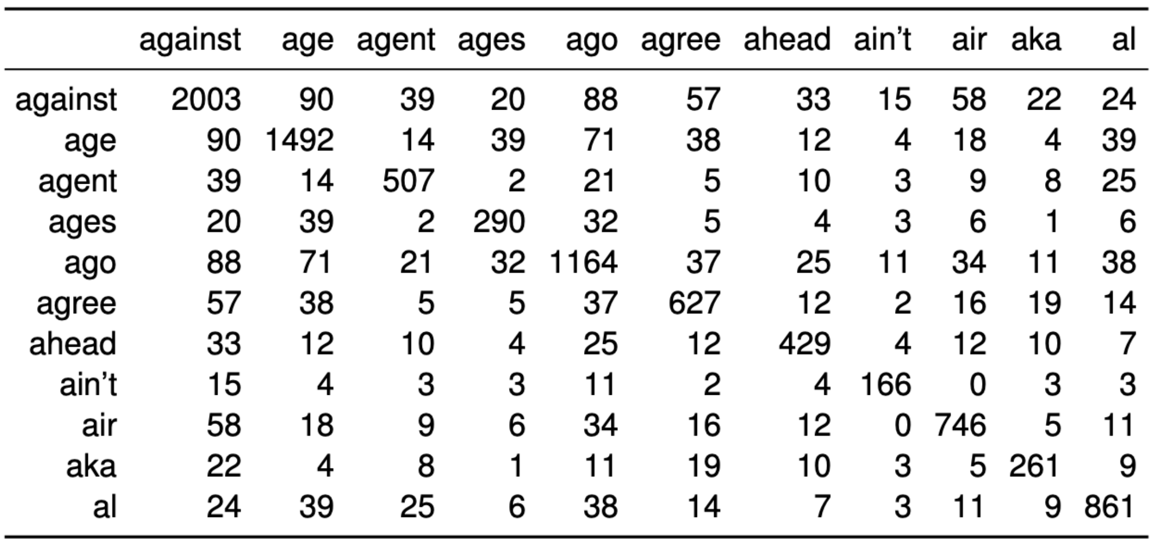
They give us dense representations of words, which we hope capture some syntactic and semantic information of the words. These dense representations are a natural tool for us to use if we want to pass words into neural models, and the field of Natural Language Processing (NLP) has used some variants of Word Embeddings extensively.

There are many, many, many ways to build word embeddings, but the key intuition comes from the notion that the meaning of a word can by inferred from the types of words it appears next to, or as put by John Firth in 1975: "You shall know a word by the company it keeps".

Typically, we start with a large corpora of text. We'll use this copora to give us counts of words which occur next to each other, which is a pretty good start. In this matrix, our rows correspond to an "embedding" of sorts, where each word's embedding is a word by word count of the words that appear next to it.

It's a hyperparameter, to choose how large of a window you use when considering words "next to each other", or if you want to do clever things, like weight each word by how distant it is from the word you're looking at.

Here's what a word co-occurence matrix looks like:



	against	age	agent	ages	ago	agree	ahead	ain’t	air	aka	al
against	2003	90	39	20	88	57	33	15	58	22	24
age	90	1492	14	39	71	38	12	4	18	4	39
agent	39	14	507	2	21	5	10	3	9	8	25
ages	20	39	2	290	32	5	4	3	6	1	6
ago	88	71	21	32	1164	37	25	11	34	11	38
agree	57	38	5	5	37	627	12	2	16	19	14
ahead	33	12	10	4	25	12	429	4	12	10	7
ain’t	15	4	3	3	11	2	4	166	0	3	3
air	58	18	9	6	34	16	12	0	746	5	11
aka	22	4	8	1	11	19	10	3	5	261	9
al	24	39	25	6	38	14	7	3	11	9	861

source:http://web.stanford.edu/class/cs224u/materials/cs224u-vsm-overview.pdf (http://web.stanford.edu/class/cs224u/materials/cs224u-vsm-overview.pdf)

Each row now captures *something* about the word it represents, in that words that appear in similar contexts will be closer together. Of course, they might not actually be *that* close together, since our space is the size of our vocabulary.

How big is our Vocabulary? Well, that's another hyperparameter. You can decide to filter out words that don't occur that often, to potentially get rid of noise, etc. But generally, it's going to be close to 300K or 400K. That means our word vectors are of dimension 400,000! Luckily, we can use dimensionality reduction techniques, like the ones you've seen already, to learn a low dimensional representation of this co-occurrence matrix. This will give us low-dimensional (200-300d), dense word vectors to use, so we can operate over them efficiently and pass them into models such as neural networks!

Which dimensionality reduction technique should we use? Should we normalize counts? Convert them to probability distributions and minimize things like KL divergence? These are all design choices which can have a big effect on the quality and output of your word vectors.

For this assignment, though, we'll be using a very standard set of Word Embeddings, called GloVe (Global Vectors for Word Representations https://nlp.stanford.edu/projects/glove/ (https://nlp.stanford.edu/projects/glove/)) These word embeddings were standard in state of the art english NLP models, until very recently.

Let's load them up and take a look at them.

In [1]:
```
import numpy as np
from numpy.linalg import norm
```

In [2]:
```
def load_vecs(path):
    """ Loads in word vectors from path.
    Will return a dictionary of word to index, and a matrix of vectors (each
    """
    vecs = []
    w2i = {}

    with open(path, 'r') as inp:
        for line in inp.readlines():
            line = line.strip().split()
            word = str(line[0])
            w2i[word] = len(vecs)
            vecs.append(np.array(line[1:], dtype=float))
        vecs = np.array([v / norm(v) for v in vecs])
        print(f'Read in {vecs.shape[0]} words of size {vecs.shape[1]}')
    return w2i, vecs
```

In [3]:
```
# This might take a little bit to run!
indxr, wembs = load_vecs('data/glove.6B.100d.txt')
```

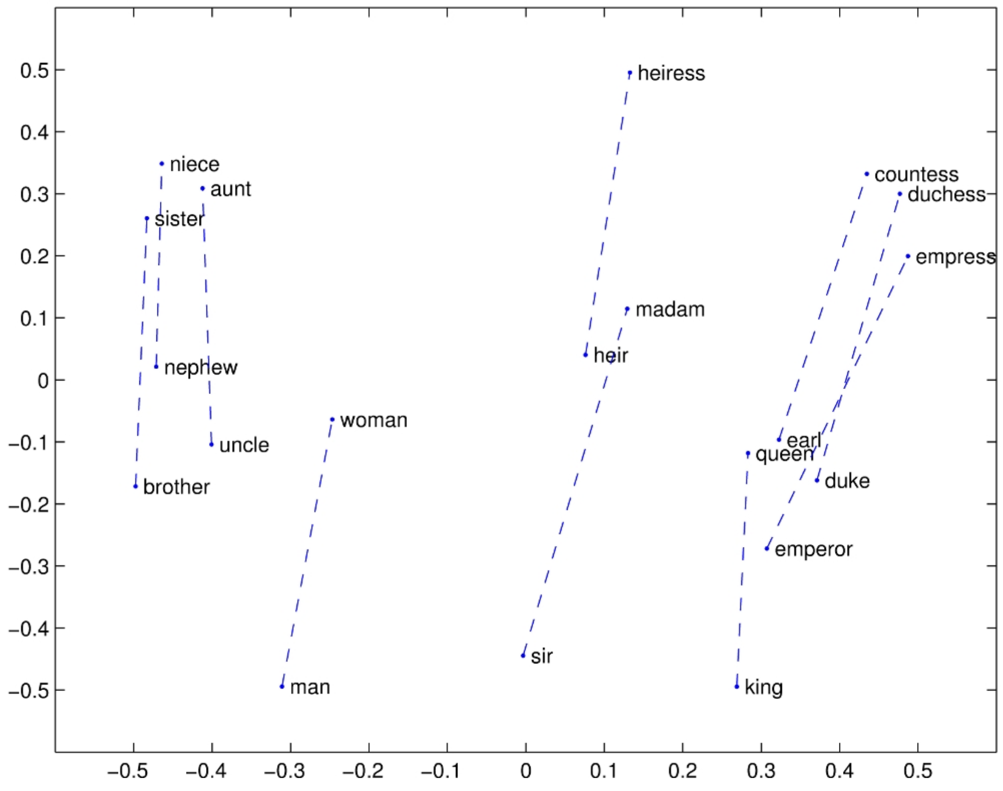
```
Read in 400000 words of size 100
```

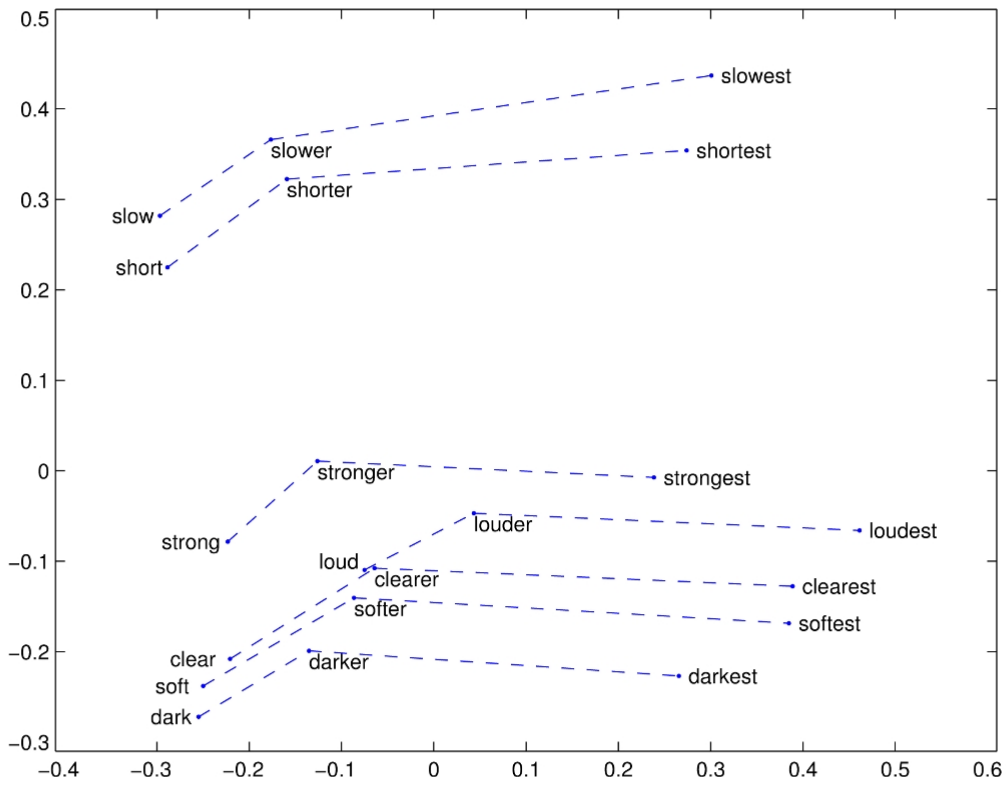
These word vectors capture some interesting semantic information!

Somewhat importantly, there seems to be some notion of semantics captured in the *vector difference* between two words. This led to the very popular "analogy" game, where it was found you could take the difference of two vectors, add it to another vector, and get this sort of analogy of the first two, compared to the second.

The canonical example here is "Man is to King as Woman is to"... And it turns out, when you use GloVe word embeddings to do this task, you get "Queen"! Which is pretty cool!

Here are some examples of the embedding space, and you can kind of see why this works!





The distance between man and woman and king and queen looks similar, but so does the direction! So it makes sense that the vector difference between woman and man, added to king, looks like queen.

Let's do this ourselves, and take a loot at it.

We've given you the similarity function ~

Implement the `analogy` method in the cell below. You are free to write helper functions as you wish, as well. The `analogy` function should take in 4 arguments: `n, word1, word2, word3` which reflects the following analogy: "word1 is to word3 as word2 is to... result".

Your function should return the top `n` vectors, in order of **most similar to least, as measured by cosine distance** which reflect this analogy and **are not word1, word2, or word3**.

Remember that the analogy intuition comes from the fact that $w1 - w2 \approx w3 - w4$, so your function should be searching for vectors which are similar to $w4 \approx w2 - w1 + w3$!

In [4]:
```
def similarity(v1, v2):
    return np.dot(v1, v2)

def bump(l, newb, pos, n):
    if len(l) < n:
        return np.concatenate((l[:pos], [newb], l[pos:]))
    else:
        return np.concatenate((l[:pos], [newb], l[pos:n - 1]))


def find_neighbors(k, neighborhood, new):
    sims = [similarity(new, neighborhood[0])]
    nearest = np.zeros(1, dtype=int)
    for n in range(1, len(neighborhood)):
        for i in range(len(sims)):
            this_sim = similarity(neighborhood[n], new)
            if this_sim > sims[i]:
                sims = bump(sims, this_sim, i, k)
                nearest = bump(nearest, n, i, k)
                break
    return nearest


def find_words(neighbors):
    words = []
    for ind in neighbors:
        words.append(list(indxr.keys())[ind])
    return words


def analogy(n, word1, word2, word3):
    emb1 = wembs[indxr[word1]]
    emb2 = wembs[indxr[word2]]
    emb3 = wembs[indxr[word3]]
    emb4 = emb2 - emb1 + emb3
    neighbors = find_neighbors(n+3, wembs, emb4)
    top_words = find_words(neighbors)
    for w in [word1, word2, word3]:  # filter out any repeated words - bori
        if top_words.__contains__(w):
            top_words.remove(w)
    return top_words[:n]
```

Once you've implemented your function, you can run the cell below to test it.

For reference, the top result should be queen!!

Do not change the below cell when turning in the notebook. You should run this cell as is, and leave the output when you have the function correct!

```
In [5]: print(analogy(10, "man", "woman", "king"))
```

```
['queen', 'monarch', 'throne', 'daughter', 'princess', 'prince', 'elizabe
th', 'mother', 'emperor', 'wife']
```

We are going to use the cell above to help us evaluate your function.

However, you should be curious about this "analogy" function! Play around with it in the cell below. Try different words! (Remember that we have a limited vocabulary... You don't need to handle OOV words nicely, but know that your code can crash occasionally if you pass in a word that's not in our vocabulary!)

```
In [6]: print(analogy(10, "man", "woman", "strong"))
        print(analogy(10, "obama", "merkel", "america"))
```

```
['stronger', 'weak', 'robust', 'strongest', 'despite', 'support', 'growin
g', 'concern', 'particularly', 'reflected']
['europe', 'germany', 'european', 'france', 'scandinavia', 'german', 'e
u', 'world', 'continent', 'baltic']
```

In this cell, tell us about the most interesting analogy you've found!

It can be any analogy you've discovered, whether you think it's interesting, wrong, or even problematic! Give a short description about why you chose this analogy. You should limit your analogies to be chosen from the top 10 results of your analogy function (that is, with `n=10`).

Analogy: "Man is to strong as woman is to *stronger*" (top result)

Discussion: The future is female.

Analogy: "Obama is to America as Merkel is to *Europe*" (top result, *Germany* second result)

Discussion: Interesting that it's pretty successful in this analogy involving proper nouns. As a follow up check, to see what kind of resolution this line of analogies would have, I also substituted napoleon in for obama, and rather than giving the same result it broadened to "global", and then some broad terms involving politics/ economics, not picking up on Germany at all (which it shouldn't considering Napoleon's connection to America is not similar to Merkel's connection to Germany).

(Gender) Bias in Word Embeddings

Word embeddings are a very useful tool in NLP, and they have often helped researchers boost their performance in a variety of tasks. However, recently a large problem has been discovered in these types of word embeddings. They inherit some biases from the data they are trained on which is very harmful, such as mysoginistic or racist stereotypes. This is a **huge** problem, because models which use these embeddings are being deployed into the real word to assist with automation strategies!

Let's take a look at some examples of gender bias in our word embeddings, which we'll be focusing on for the rest of this assignment.

Do not change the below cell! Make sure you run it as is before turning in your notebook.

In [7]:
```
print(analogy(10, "man", "woman", "programmer"))
print(analogy(10, "man", "woman", "doctor"))
# Even names contain these biases!
print(analogy(10, "john", "mary", "doctor"))
```

```
['educator', 'programmers', 'linguist', 'technician', 'freelance', 'anima
tor', 'translator', 'software', 'psychotherapist', 'technologist']
['nurse', 'physician', 'doctors', 'patient', 'dentist', 'pregnant', 'medi
cal', 'nursing', 'mother', 'hospital']
['nurse', 'mother', 'nursing', 'woman', 'dentist', 'pregnant', 'hospita
l', 'patient', 'girl', 'grandmother']
```

What does it look like if we swap the analogy around?

Do not change the below cell! Make sure you run it as is before turning in your notebook.

In [8]:
```
print(analogy(10, "woman", "man", "programmer"))
print(analogy(10, "woman", "man", "doctor"))
print(analogy(10, "mary", "john", "doctor"))
```

```
['programmers', 'software', 'computer', 'animator', 'engineer', 'setup',
'mechanic', 'compiler', 'animators', 'developer']
['dr.', 'brother', 'him', 'he', 'himself', 'physician', 'father', 'maste
r', 'friend', 'taken']
['physician', 'he', 'dr.', 'surgeon', 'himself', 'him', 'asked', 'man',
'expert', 'agent']
```

This is obviously problematic. Yet, these word embeddings (the ones you're using right now) **have been used in multiple state of the art systems in NLP!** This is a hot area of research right now, because this poses a huge potential problem as more and more AI systems are starting to be deployed into the real world.

We're going to take a look at one method of *debiasing* these word embeddings, which attempts to *remove gender stereotypes* while keeping in useful gender information such as "king and queen" and "boy and girl". The debiasing method we're going to look at is described in this paper (https://arxiv.org/abs/1607.06520), which is one of the seminal papers on exposing stereotypes and bias in this form.

While they use different word embeddings, we've seen that our GloVe embeddings contain similar biases. Their method behaves as follows.

They first define a "gender subspace" $\mathcal{B} = (b_1, b_2, \ldots, b_k)$ composed of *orthogonal vectors*. k is a hyperparameter we choose.

$\mathcal{B}$ is built from a set of pairs of gendered items. The idea here is that $\mathcal{B}$ captures some notion of the "direction" of gender which we're trying to capture.

The set of pairs, $S = p_1, p_2, \ldots, p_n$, is given to you. Each pair contains two words which are considered gendered words whose relation captures some notion of gender, i.e. S ={("woman", "man"), ("she", "he")...}

Building $\mathcal{B}$ goes as follows:

1. Build up matrix $\mathbf{C} := \sum_{i=1}^{n} (\vec{w_1} - \vec{w_2})(\vec{w_1} - \vec{w_2})^T + (\vec{w_2} - \vec{w_1})(\vec{w_2} - \vec{w_1})^T$, for $(w_1, w_2) \in p_i$.

That is, for each pair, subtract each word vector from the other and take the outer product of each resulting vector and add it to $\mathbf{C}$. In our case, $\mathbf{C}$'s dimensionality should be 100 by 100.

2. Compute the SVD of $\mathbf{C}$.

You can use numpy's `numpy.linalg.svd` method for this.

3. This will give you a decomposition $\mathbf{U\Sigma V} = \mathbf{C}$. Take the top-$k$ vectors from the decomposition of $\mathbf{C}$ as the orthogonal vectors defining the space $\mathcal{B} = (b_1, \ldots, b_k)$. That is, you should take the first k columns of $\mathbf{U}$.

Again, the intuition here is that we now have some set of vectors which, together, capture some notion of the direction of gender.

In [9]:
```
# Copy biased embeddings into a new object.
debiased_wembs = np.copy(wembs)
```

In [10]:
```
from numpy.linalg import svd
gender_pairs = [('she', 'he'), ('her', 'his'), ('woman', 'man'), ('mary', '

def build_gender_subspace(k):
    C = np.zeros([100, 100])
    for pair in gender_pairs:
        emb1 = np.array([wembs[indxr[pair[0]]]]).T
        emb2 = np.array([wembs[indxr[pair[1]]]]).T
        onetwo = np.matmul(emb1-emb2, np.transpose(emb1-emb2), np.zeros(C.sh
        twoone = np.matmul(emb2-emb1, np.transpose(emb2-emb1), np.zeros(C.sh
        C = np.add(C, np.add(np.matmul(emb1-emb2, np.transpose(emb1-emb2)),
    [u, _, _] = svd(C)
    return u[:, :k]
```

Now let's build the subspace with $k = 10$. You can check that things seem ok by making sure that the dot product between all your b_i vectors is close to zero, since they should be orthogonal.

In [11]:
```
B = build_gender_subspace(10)
```

We'll only implement the neutralize" portion of the hard-debiasing method in the paper, if you're following along. We won't implement equalize or the Soft-Debiasing method, to keep things short :)

Once we have our gender subspace $\mathcal{B}$ composed of our k orthogonal vectors, we can select some choice word w to debias as follows:

1. Select the embedding $\vec{w}$ of word w from our regular, biased embeddings.
2. Compute
$$\overrightarrow{w_{\mathcal{B}}} = \sum_{j=i}^{k} (\vec{w} \cdot \overrightarrow{b_j}) * \overrightarrow{b_j}$$
.
3. Compute the new, debiased embedding as
$$\overrightarrow{w_{ub}} = (\vec{w} - \overrightarrow{w_{\mathcal{B}}}) \; / \; ||\vec{w} - \overrightarrow{w_{\mathcal{B}}}||$$

Intuitively, what we are doing is projecting our biased vector $\vec{w}$ into our gender subspace, and then subtracting the result from $\vec{w}$.

You should implement a function `debias_word(word)` , which takes one argument: the word to debias. It should use our previously defined subspace `B` to compute $\overrightarrow{w_{ub}}$, and you should store the result in the new `debiased_wembs` matrix defined above, **in the same index that the word is in the original `wemb` matrix**.

That is, please do not change the `wembs` matrix directly, but save your debiased embeddings in the copy we created.

In [12]:
```python
def debias_word(word):
    emb = wembs[indxr[word]]
    wb = np.zeros(emb.shape)
    for j in range(B.shape[1]):
        wb = np.add(wb, np.dot(emb, B[:,j]) * B[:,j])
    wub = np.subtract(emb, wb)/norm(np.subtract(emb, wb))
    debiased_wembs[indxr[word]] = wub
    return debiased_wembs
```

Lastly, let's build a *new* analogy function, called `debiased_analogy` which operates exactly the same the your original analogy function, but uses the `debiased_wembs` instead.

In [13]:
```python
def debiased_analogy(n, word1, word2, word3):
    emb1 = debiased_wembs[indxr[word1]]
    emb2 = debiased_wembs[indxr[word2]]
    emb3 = debiased_wembs[indxr[word3]]
    emb4 = emb2 - emb1 + emb3
    neighbors = find_neighbors(n+3, debiased_wembs, emb4)
    top_words = find_words(neighbors)
    for w in [word1, word2, word3]:  # filter out any repeated words - bori
        if top_words.__contains__(w):
            top_words.remove(w)
    return top_words
```

Now we're ready to start debiasing our word vectors!

We've picked out a few choice words to debiase for the purpose of our example. Let's debias them, and then revisit our "man is to doctor as woman is to" analogy.

Do not change the below cell! Make sure you run it as is before turning in your notebook. It will be used for grading.

In [14]:
```
debias_word("doctor")
debias_word("doctors")
debias_word("nurse")
debias_word("dentist")
debias_word("patient")
debias_word("physician")
debias_word("dr.")
debias_word("boss")


print(debiased_analogy(10, "man", "woman", "doctor"))
print(debiased_analogy(10, "woman", "man", "doctor"))
```

```
['physician', 'nurse', 'patient', 'dr.', 'doctors', 'medical', 'pregnan
t', 'pregnancy', 'pediatrician', 'nursing', 'psychiatrist', 'pharmacist']
['physician', 'nurse', 'patient', 'dr.', 'doctors', 'his', 'dentist', 'br
other', 'he', 'described', 'admitted']
```

This is looking better! Not only is the top result a lot better, but there seems to be much less of a difference between the swapped results as well. Very nice!

Our solution doesn't scale, obviously. We can't expect ourselves to manually type each word that we think should be debiased. However, this is a decent start!

There is a lot of research in this area right now, and people are constantly coming up with better and more scalable methods to solve this very real problem!

To close things off, provide us with one more example of an analogy you found that seemed gender-biased.

Debias the words that you think are necessary to debias the analogy, and then report the new analogy in the cells below.

Biased Analogy you found: "Mary is to schoolteacher as John is to *businessman/ bricklayer/ politician*", while "John is to schoolteacher as Mary is to *homemaker/ housewife/ widowed*"

Debias the words below, and then print out the new debiased analogy and it's gender-swap!

In [15]:
```
debias_word("schoolteacher")
print(debiased_analogy(10, "mary", "john", "schoolteacher"))
print(debiased_analogy(10, "john", "mary", "schoolteacher"))
```

```
['businessman', 'bricklayer', 'loner', 'electrician', 'mechanic', 'stockb
roker', 'banker', 'politician', 'veteran', 'rancher', 'accountant', 'cong
ressman']
['homemaker', 'housewife', 'seamstress', 'tomboy', 'stepdaughter', 'spins
ter', 'pensioner', 'widowed', 'grandmother', 'midwife', 'housekeeper', 'm
other']
```

Unfortunately, looks like this one was not successfully debiased using the same means as were used to debias "doctor". Perhaps there is more bias on this one - I personally do know many more female doctors than I do male schoolteachers. It also may not help that "schoolteacher" is a slightly outdated term. I'd say that gender bias in writing has gotten a bit better with time, but if a term stopped being used before this change arose then it's safe to expect it to reflect greater bias.