Assignment 2

COMP-599 and LING-782/484

1. Learning Word Embeddings (50 pts)

In this part of the assignment, you will implement two models from the word2vec project: Continuous Bag-of-Words (CBOW) and Skip-Gram. The implementations follow from this paper: https://arxiv.org/pdf/1301.3781.pdf

1.1 Continuous Bag-of-Words and Skip-Gram Pre-processing (20 pts)

We have provided code to convert the initial text data (NLI dataset) into tokenized indices, similar to the model input for A1. Using that, implement the following functions in order to convert the sentences (mix of premises and hypotheses) into input indices and output labels for CBOW and Skip-Gram:

- `build_current_surrounding_pairs`
- `expand_surrounding_pairs`
- `cbow_preprocessing`
- `skipgram_preprocessing`

The detailed specifications for each function are provided below.

build_current_surrounding_pairs (5 pts)

A helper function that is applied to each sample in order to construct pairs of current word and its surrounding words. This will be used later in the assignment.

Concretely, this function pairs up each current word w(t) with its surrounding words (...w(t+2), w(t+1), w(t-1), (w(t-2)...), with respect to a window size.

Note**:** To ensure that `current_indices` has a consistent inner length, you will need to omit the start and end of the sample_indices list from the surrounding since it would otherwise have unbalanced sides.

Parameter	**Type**	**Description**
indices	list of int	The indices output by `tokens_to_ix` for a single sample (not for the entire dataset).

window_size	int	The number of indices from each side to choose from. The context should be 2 times the window size (e.g., 2 * 5 = 10). Takes a default value of 2.

Returns	**Type**	**Description**
surrounding_indices	list of list of int	Indices nearby the current word but does not include itself. Make sure that the length of the inner surrounding is consistent (i.e., 2 * window size).
current_indices	list of int	Indices of the current words in the middle of a context. Denoted at w(t) in the paper. The length of this list should be `sample_indices - 2 * window_size`

Example

```
>>> text = “dogs and cats are playing”.split()
>>> surroundings, currents = build_current_surrounding_pairs(
...     text, window_size=1
... )
>>> print(currents)
[‘and’, ‘cats’, ‘are’]
>>> print(surroundings)
[[‘dogs’, ‘cats’], [‘and’, ‘are’], [‘cats’, ‘playing’]]

>>> indices = [word_to_index[t] for t in text]
>>> surroundings, currents = build_current_surrounding_pairs(
...     indices, window_size=1
... )
>>> print(currents)
[3, 4887, 11]
>>> print(surroundings)
[[110, 4887], [3, 11], [4887, 31]]
```

expand_surrounding_words (5 pts)

A helper function used in Skip-Gram to expand a list of surroundings into pairs of a single surrounding with the target (the latter will be repeated).

Using the output of `build_current_surrounding_pairs`, convert the surrounding into pairs of context-target pair. The resulting lists should be longer.

Parameter	**Type**	**Description**

ix_surroundings	list of list of int	The context from a window. Those are the indices of words around the current word.
ix_current	list of int	The indices of the current words. Denoted as w(t) in the paper.

Returns	**Type**	**Description**
ix_surroundings_expanded	list of int	The indices of each surrounding word (after expansion).
ix_current	list of int	The indices of the current word (after expansion) matching each single surrounding word.

Example

```
>>> # dogs and cats are playing
>>> surroundings = [
...     [‘dogs’, ‘cats’], [‘and’, ‘are’], [‘cats’, ‘playing’]
... ]
>>> currents = [‘and’, ‘cats’, ‘are’]
>>> surrounding_expanded, current_expanded =
expand_surrounding_words(
...     surroundings, currents
... )
>>> print(surrounding_expanded)
[‘dogs’, ‘cats’, ‘and’, ‘are’ cats’, ‘playing’]
>>> print(current_expanded)
[‘and’, ‘and’, ‘cats’, ‘cats’, ‘are’, ‘are’]

>>> ix_surroundings =[[110, 4887], [3, 11], [4887, 31]]
>>> ix_currents = [3, 4887, 11]
>>> ix_surr_expanded, ix_curr_expanded = expand_surrounding_words(
...     ix_surroundings, ix_currents
... )
>>> print(ix_surr_expanded)
[110, 4887, 3, 11, 4887, 31]
>>> print(ix_curr_expanded)
[3, 3, 4887, 4887, 11, 11]
```

cbow_preprocessing (5 pts)

Preprocess the dataset to be ready for CBOW to use for training.

Concretely, use the `build_current_surrounding_pairs` function you implemented above to complete this function. The difference is that the input is a list of indices (so a nested list), but the output format should be the same.

Parameter	Type	Description
indices_list	list of list of int	The inner list contains the indices of a single sample. The outer list contains all the samples in the dataset.
window_size	int	The number of indices from each side to choose from.

Returns	Type	Description
sources	list of list of int	The inputs to the CBOW model. The inner list contains the indices of surrounding words and the outer list contains the samples (from a batch or from the whole dataset).
targets	list of int	The targets of the CBOW model. Contains the indices of the target word w(t).

skipgram_preprocessing (5 pts)

Preprocess the dataset to be ready for Skip-Gram to use for training.

Concretely, use the `build_current_surrounding_pairs` function you implemented above to complete this function. The difference is that the input is a list of indices (so a nested list), but the output format should be the same.

Note: Here, you need to return all possible pairs between a word w(t) and its surroundings. In the paper, it's a sampling method based on distance, but we will not do that for simplicity, instead we'll just use everything.

Parameter	Type	Description
indices_list	list of list of int	A nested list of indices. The inner list contains the indices of a single sample. The outer list contains all the samples in the dataset.

window_size	int	The number of indices from each side to choose from.

Returns	Type	Description
targets	list of int	The targets of the Skip-Gram model. List of indices of a single **surrounding** word.
sources	list of int	The inputs to the Skip-Gram model. List of indices of the **center** word w(t).

1.2 Continuous Bag-of-Words and Skip-Gram Models (15 pts)

Build CBOW and Skip-Gram as torch modules, and train them using the provided `train` function. You will need to write the following classes:

- `SharedNNLM`: A helper class (not a `nn.Module`) that loads the embedding and project layers and binds the weights (such that the weights are shared).
- `SkipGram`: The Skip-Gram model. It will use `SharedNNLM`. You only need to implement the forward pass.
- `CBOW`: The CBOW model. It will use `SharedNNLM`. You only need to implement the forward pass.

The detailed specifications for each class are provided below:

SharedNNLM.__init__ (5 pts)

Initializes `SharedNNLM` model. This class will be used by `SkipGram` and `CBOW`. This class is a simplification of the NNLM model (no hidden layer) and the input and output layers share the same weights. The projection in word2vec does not have a bias. Your `__init__` function should initialize the model's embeddings and create a projection layer (which does not have a bias term).

Parameter	Type	Description
num_words	int	The number of words in the vocabulary. Used to determine the size of the embedding table.
embed_dim	int	The dimension of embeddings to use.

SkipGram.forward (5 pts)

Executes the forward pass for the SkipGram model. Given the index of a target word w(t), this function returns predicted distribution of the index of a surrounding word.

Parameter	Type	Description
x	Tensor[batch_size]	The indices of the target word w(t).

Returns	Type	Description
output	Tensor[batch_size, num_words]	The predicted distribution of the index of a surrounding word.

CBOW.forward (5 pts)

Executes the forward pass for the CBOW model. Given the indices of the surrounding words, this function returns the predicted distribution of the index of w(t).

Parameter	Type	Description
x	Tensor[batch_size, 2 * window_size]	The indices of the surrounding words, i.e., w(t - window_size), …, w(t + window_size).

Returns	Type	Description
output	Tensor[batch_size, num_words]	The predicted distribution of the index of w(t).

1.3 Analysis (15 pts)

Build functions that lets you find the K most similar words:

- `compute_topk_similar`: Helper function used in `retrieve_similar_words` and `word_analogy`, it allows you to retrieve the K indices with highest cosine similarity given a vector and an embedding weight.
- `retrieve_similar_words`: A function that, given a model and index map, takes a word and finds K most similar words.
- `word_analogy`: A function that computes word analogies, e.g. *man is to woman what ? is to girl.*

The detailed specifications for each function are provided below.

compute_topk_similar (5 pts)

Helper function used in `retrieve_similar_words` and `word_analogy`, it allows you to retrieve the K indices with highest cosine similarity given a vector and an embedding weight.

Concretely, this function computes the cosine similarity between the embedding of a single word and the embedding of all words and then returns the indices of the top K most similar results (excluding the word itself).

Parameter	**Type**	**Description**
word_emb	Tensor[1, embed_dim]	The embedding representation of a single word.
w2v_emb_weight	Tensor[num_words, embed_dim]	The entire tensor representing the weight of the `nn.Embedding` of all words in the vocabulary.
k	int	The number of indices to retrieve.

Returns	**Type**	**Description**
topk_ix	list of int	The indices of the top K most similar words.

retrieve_similar_words (5 pts)

A function that, given a model and index map, takes a word and finds the K most similar words.

Specifically, given your implementation of `compute_topk_similar` and a word2vec model, this function finds the K most similar words from your vocabulary. Make sure you set your model to evaluation mode and disable gradient calculation.

Note: `compute_topk_similar` takes as input a `word_emb` of shape `[1, embed_dim]`. You will need to handle that.

Parameter	**Type**	**Description**
model	nn.Module	Either the SkipGram or CBOW model you created previously.
word	str	Some word that exists in `index_map` (or the list of its keys).
index_map	dict of {str:int}	A dictionary mapping a word to its index.
index_to_word	dict of {int:str}	A dictionary mapping an index to the word (reverse of `index_map`).

Returns	Type	Description
similar_words	list of str	The K most similar words to the input word.

word_analogy (5 pts)

A function that computes word analogies, e.g. *man is to woman what ? is to girl.*

Using your `compute_topk_similar` function and a word2vec model, this function will compute the following analogy:

> "word_a" is to "word_b" what "?" is to "word_c"

It can also be represented as as:

> word_a - word_b + word_c = ?

Your function will find the K most similar words to "?" from your vocabulary. Make sure you set your model to evaluation mode and disable gradient computation.

Parameter	Type	Description
model	nn.Module	Either the SkipGram or CBOW model you created previously.
word_a	str	Some word that exists in index_map (or the list of its keys).
word_b	str	Some word that exists in index_map (or the list of its keys).
word_c	str	Some word that exists in index_map (or the list of its keys).
index_map	dict of {str: int}	A dictionary mapping a word to its index.
index_to_word	dict of {int: str}	A dictionary mapping an index to the word (reverse of `index_map`).

Returns	Type	Description
similar_words	list of str	The K most similar words to the unknown word "?".

2. Measuring Gender Bias in Word Embeddings (30 pts)

In this part of the assignment, you will be investigating techniques for measuring gender bias in word embeddings. Specifically, you will be investigating gender bias in GloVe embeddings. We will use the 300 dimensional `glove.6B` vectors. The embeddings can be downloaded from here (download the `glove.6B.zip` file and use the `glove.6B.300d.txt` file contained within).

We have provided functions to load the GloVe embeddings and to load a list of gender attribute words (e.g., man/woman) for this section.

2.1 Estimating Gender Subspace (5 pts)

compute_gender_subspace

To begin, you will implement `compute_gender_subspace` which estimates the "gender direction" in an embedding space. To estimate this gender direction, you will use the gender attribute words we have provided. You will make use of the `sklearn` implementation of PCA to estimate the gender direction.

Concretely, your function will take a dictionary of word embeddings and a list of pairs of gender attribute words (e.g., man/woman) and estimate a gender subspace. The steps for estimating the gender subspace are:

1. Convert each pair of gendered words (e.g., *man/woman*) to their embeddings.
2. Normalize each pair of embeddings. That is, compute the mean embedding for each pair (e.g., `Mean(Emb(man), Emb(woman))`) and subtract the mean from each embedding in the pair (e.g., `Emb(man) - Mean`):

```
>>> mean = (word_to_embedding["man"]+ word_to_embedding["woman"]) / 2
>>> man_embedding = word_to_embedding["man"] - mean
>>> woman_embedding = word_to_embedding["woman"] - mean
```

3. Run PCA using the resulting list of normalized embeddings.

Note: In lecture, we used the difference between pairs of gendered embeddings as input to PCA. In this assignment you are only required to use the gendered embeddings themselves. Also, the principal components can be accessed through the `components_` attribute of the PCA class in `sklearn`.

The detailed specifications for this function are provided below.

Parameter	Type	Description
word_to_embedding	dict of {str: np.array}	A dictionary mapping from string to GloVe embedding.
gender_attribute_words	list of lists of strings	The gender attribute words (e.g., man/woman) used to estimate the gender subspace. Each element in the list is a pair of words which have similar meaning and differ primarily only with respect to gender (e.g., grandmother/grandmother).
n_components	int	The number of principal components to use for the estimated gender subspace.

Returns	Type	Description
gender_subspace	np.array[n_components, embed_dim]	The estimated gender subspace. Each element in the first dimension is a principal component for the estimated gender subspace.

Example

```
>>> gender_attribute_words = [
...     [“man”, “woman”],
...     [“boy”, “girl”]
... ]
>>> gender_subspace = compute_gender_subspace(
...     word_to_embedding=word_to_embedding,  #  GloVe embeddings
...     gender_attribute_words=gender_attribute_words,
...     n_components=1  # Single dimension
... )
>>> print(len(gender_subspace), type(gender_subspace[0]))
1
numpy.ndarray
```

2.2 Vector Projection (5 pts)

Now that you've estimated a "gender direction" in the GloVe embedding space, we will want to project embeddings onto this estimated direction. In this section, you will implement a function which computes a vector projection. The projection of a vector **a** onto a vector **b** is defined as:

$$\text{proj}_{\mathbf{b}}\mathbf{a} = \frac{\mathbf{a}\cdot\mathbf{b}}{\mathbf{b}\cdot\mathbf{b}}\mathbf{b}.$$

Your function will also need to return the scaling coefficient s:

$$s = \frac{\mathbf{a}\cdot\mathbf{b}}{\mathbf{b}\cdot\mathbf{b}}$$

which we will use later in the assignment. We denote the dot product between vectors **a** and **b** by **a·b.** The detailed specifications for the function are provided below:

Parameter	**Type**	**Description**
a	np.array[embed_dim]	The vector which you are projecting.
b	np.array[embed_dim]	The vector which you are projecting onto.

Returns	**Type**	**Description**
scalar	float	The scaling coefficient (s) in the vector projection.
vector_projection	np.array[embed_dim]	The vector projection of a onto b.

Example

```
>>> a = np.random.randn(768)
>>> b = np.random.randn(768)
>>> scalar, vector_projection = project(a, b)
>>> print(vector_projection.shape)
(768,)
```

2.3 BEC-Pro Profession Representations (4 pts)

Now that we've estimated a gender direction in the embedding space and implemented a function to project vectors onto this direction, we will investigate gender bias within

representations for *professions* (e.g., nurse, mechanic, doctor). In this section, you will compute representations for professions from the BEC-Pro dataset (Bartl et al., 2020).

Since professions sometimes contain multiple words (e.g., aerospace engineer), your function will need to average the GloVe embeddings for professions which consist of multiple words. You can split the professions into subwords using simple whitespace tokenization (e.g., `profession.split()`).

The specifications for this function are provided below:

Parameter	**Type**	**Description**
word_to_embedding	dict of {str: np.array[embed_dim]}	A dictionary mapping from string to GloVe embedding.
professions	list of strings	The list of professions to compute representations for.

Returns	**Type**	**Description**
profession_to_embedding	dict of {str: np.array[embed_dim]}	A dictionary mapping from profession strings to embeddings.

2.4 Words with Large Gender Components (4 pts)

You will now implement the function `compute_extreme_words` which, given a list of words, computes the K words with either the smallest or largest scalar coefficients onto a given gender direction. You will make use of the `project` function you previously implemented to get the scaling coefficients.

Concretely, for this function you will need to compute the scalar coefficients for each word embedding (corresponding to a word in `words`) onto the estimated gender direction. Then, you will return the words with either the largest scalar coefficients or the smallest. You can compute the scalar coefficients using only the first direction in your estimated gender subspace (e.g., the first principle component). The specifications for this function are provided below:

Parameter	**Type**	**Description**
words	list of strings	The list of words from which you select the K words with either the largest or the smallest scalar

		coefficients.
word_to_embedding	dict of {str: np.array[embed_dim]}	A dictionary mapping from strings to embeddings.
gender_subspace	np.array[n_components, embed_dim]	The estimated gender subspace. Each element in the first dimension is a principal component for the estimated gender subspace.
k	int	The number of words to select.
max_	bool	Whether to select the words with the most *positive* scalar coefficients (largest) or the most *negative* scalar coefficients (smallest). Defaults to true (largest).

Returns	**Type**	**Description**
extreme_words	list of strings	The K words with the most extreme scalar coefficients (either the smallest or largest scalar coefficients).

2.5 DirectBias Metric (4 pts)

You will now implement the DirectBias metric from [Bolukbasi et al., 2016 (Section 5.2)]:

$$\text{DirectBias}_c = \frac{1}{|N|} \sum_{w \in N} |\cos(w, g)|^c$$

Concretely, given a set of words *N* which we expect to be gender neutral. We compute the total absolute cosine similarity (raised to the power of *C*) between words from *N* and the estimated gender direction *g*.

To implement the DirectBias metric, you will need to implement two functions: `cosine_similarity` and `compute_direct_bias`. The specifications for these functions are given below:

cosine_similarity (2 pts)

A helper function for `compute_direct_bias` which computes the cosine similarity between two vectors. Note**:** Do not assume that `a` or `b` is a unit vector.

Parameter	Type	Description
a	np.array[embed_dim]	One of the vectors to compute the cosine similarity with.
b	np.array[embed_dim]	One of the vectors to compute the cosine similarity with.

Returns	Type	Description
cosine_similarity	float	The cosine similarity between a and b.

compute_direct_bias (2 pts)

For a given set of words and an estimated gender subspace, computes the DirectBias metric. You will compute the DirectBias metric using only the first direction in your estimated gender subspace (e.g., the first principle component).

Parameter	Type	Description
words	list of strings	The list of words to compute the direct bias measure with.
word_to_embedding	dict of {str: np.array[embed_dim]}	A dictionary mapping from strings to embeddings.
gender_subspace	np.array[n_components, embed_dim]	The estimated gender subspace. Each element in the first dimension is a principal component for the estimated gender subspace.
c	float	Parameter for the DirectBias metric. Determines the “strictness” of the measure.

Returns	Type	Description
direct_bias	float	The computed DirectBias metric.

2.6 Word Embedding Association Test (8 pts)

You will now implement the Word Embedding Association Test (WEAT; Caliskan et al., 2017). WEAT is used to test for social bias within word representations. To review WEAT, we refer you to slides 6 through 18 from the lecture.

You will be required to implement two functions: `weat_association` and `weat_differential_association`. The specifications for these functions are provided below.

weat_association (4 pts)

Computes a word's mean cosine similarity with the words from each attribute word set. Then, returns the difference between these two means. This function computes Equation 2 on slide 14 from the lecture.

Parameter	Type	Description
w	str	The word to compute the difference in mean cosine similarities for.
A	list of strings	One of the attribute word sets (e.g., *man, boy*).
B	list of strings	One of the attribute word sets (e.g., *woman, girl*).
word_to_embedding	dict of {str: np.array[embed_dim]}	A dictionary mapping from strings to embeddings.

weat_differential_association (4 pts)

This function computes the WEAT test statistic for given sets of target words (X, Y), attribute words (A, B), and embeddings. This function computes Equation 1 on slide 14 from the lecture using your implementation of `weat_association`.

Parameter	Type	Description
X	list of strings	One of the target word sets (e.g., *doctor, mechanic*).
Y	list of strings	One of the target word sets (e.g., *nurse, artist*).
A	list of strings	One of the attribute word sets (e.g., *man, boy*).
B	list of strings	One of the attribute word sets (e.g.,

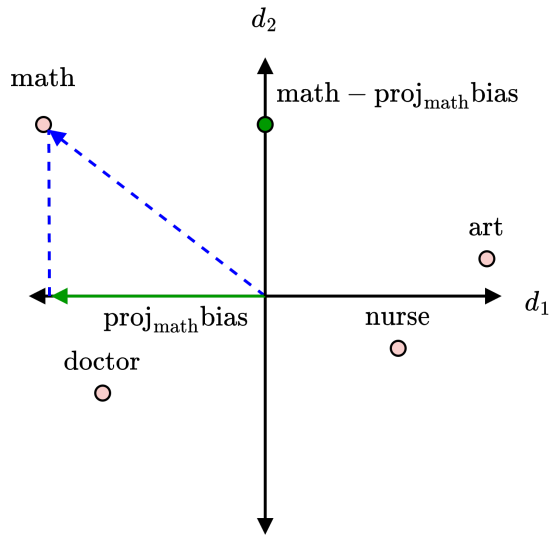
		woman, girl).
word_to_embedding	dict of {str: np.array[embed_dim]}	A dictionary mapping from strings to embeddings.
weat_association_func	Callable	Your implemented `weat_association` function. (This is to make testing of your solution easier.)

Returns	**Type**	**Description**
differential_association	float	The computed WEAT test statistic.

3. Mitigating Gender Bias in Word Embeddings (20 pts)

3.1 HardDebias (10 pts)

Now that you have implemented two metrics for gender bias in word embeddings, you will now implement HardDebias, a technique for mitigating bias in word embeddings. To review HardDebias, we refer you to slides 19 through 32 from the lecture.



To implement HardDebias, you will make use of your estimated gender direction from Part 2 of this assignment. You will be required to implement two functions for HardDebias:

`debias_word_embedding` and `hard_debias`. The specifications for both functions are given below.

debias_word_embedding (5 pts)

Given a word and an estimated gender subspace, this function subtracts the embedding's projection onto the estimated gender subspace from itself (making the embedding orthogonal to the estimated gender subspace). See the figure above for an example of this procedure.

Parameter	**Type**	**Description**
word	str	The word embedding to debias.
word_to_embedding	dict of {str: np.array[embed_dim]}	A dictionary mapping from strings to embeddings.
gender_subspace	np.array[n_components, embed_dim]	The estimated gender subspace. Each element in the first dimension is a principal component for the estimated gender subspace.

Returns	**Type**	**Description**
debiased_embedding	np.array[embed_dim]	The debiased word embedding.

hard_debias (5 pts)

Given a dictionary of word embeddings, this function uses `debias_word_embedding` to debias all of the word embeddings.

Parameter	**Type**	**Description**
word_to_embedding	dict of {str: np.array[embed_dim]}	A dictionary mapping from strings to embeddings.
gender_attribute_words	list of list of strings	The gender attribute words (e.g., man/woman) used to estimate the gender subspace. Each element in the list is a pair of words which have similar meaning and differ primarily only with respect to gender (e.g., grandmother/grandmother).

n_components	int	The number of principal components to use for the estimated gender subspace.

Returns	**Type**	**Description**
debiased_word_to_embedding	dict of {str: np.array[embed_dim]}	A dictionary mapping from strings to debiased embeddings.

3.2 Debiasing Evaluation (10 pts)

Given your implementation of HardDebias, you will now empirically evaluate the debiasing technique. The three required components for your report are detailed below. **This report component is completed on Gradescope.**

DirectBias Evaluation (2 pts)

First, using your implementation of `compute_direct_bias`, you will measure the gender bias in the profession embeddings *before* and *after* applying HardDebias. When computing DirectBias, you can use `c = 0.25`. **Report these DirectBias values on Gradescope and comment on what the *ideal* score for DirectBias** (one to two sentences).

WEAT Evaluation (2 pts)

Now, you will run a WEAT test using your GloVe embeddings *before* and *after* applying HardDebias. The target word sets and attribute word sets are provided below:

Math/Arts* and *Male/Female	
X	math, algebra, geometry, calculus, equations, computation, numbers, addition
Y	poetry, art, dance, literature, novel, symphony, drama, sculpture
A	male, man, boy, brother, he, him, his, son
B	female, woman, girl, sister, she, her, hers, daughter

We have provided a function `p_value_permutation_test` which runs an exact one-sided permutation test on the WEAT test statistic and returns the p-value.

Do you obtain statistically significant p-values (with significance level 0.05) for the test? What about after applying HardDebias? It is okay if you do not (or obtain a similar p-value). You will

provide the two p-values on Gradescope and briefly comment on your results (two to three sentences).

Gender Bias T-SNE Plot (6 pts)

Finally, you will create a 2D T-SNE plot which visualizes embeddings for professions with large gender components *before* applying HardDebias. You will be required to identify two sets of 10 professions each. To identify these sets of professions, you will use your previously estimated gender direction and your `compute_extreme_words` function:

```
>>> max_words = compute_extreme_words(
...     words=words,
...     word_to_embedding=word_to_embedding,
...     gender_subspace=gender_subspace,
...     k=k,
...     max_=True
... )

>>> min_words = compute_extreme_words(
...     words=words,
...     word_to_embedding=word_to_embedding,
...     gender_subspace=gender_subspace,
...     k=k,
...     max_=False
... )
```

One set of words will contain the professions with the smallest scalar coefficients onto the estimated gender direction and the other set of words will contain the words with the largest scalar coefficients onto the estimated gender direction.

You can use the `sklearn` implementation of T-SNE to create your plot. In the figure, make sure to plot the professions from each set with a different color. You can use any plotting library you like (e.g., `matplotlib`). You will **upload your plot to Gradescope**.

Optionally (**you are not required to submit this**), you can plot the same sets of professions *after* applying Hard-Debias. However, since we implemented a simplified form of Hard-Debias you may not be able to observe a substantial difference between the two plots.