NLP assignment 1 report

Damiano Buzzo<damiano25@ru.is>

T-725-MALV Natural Language Processing

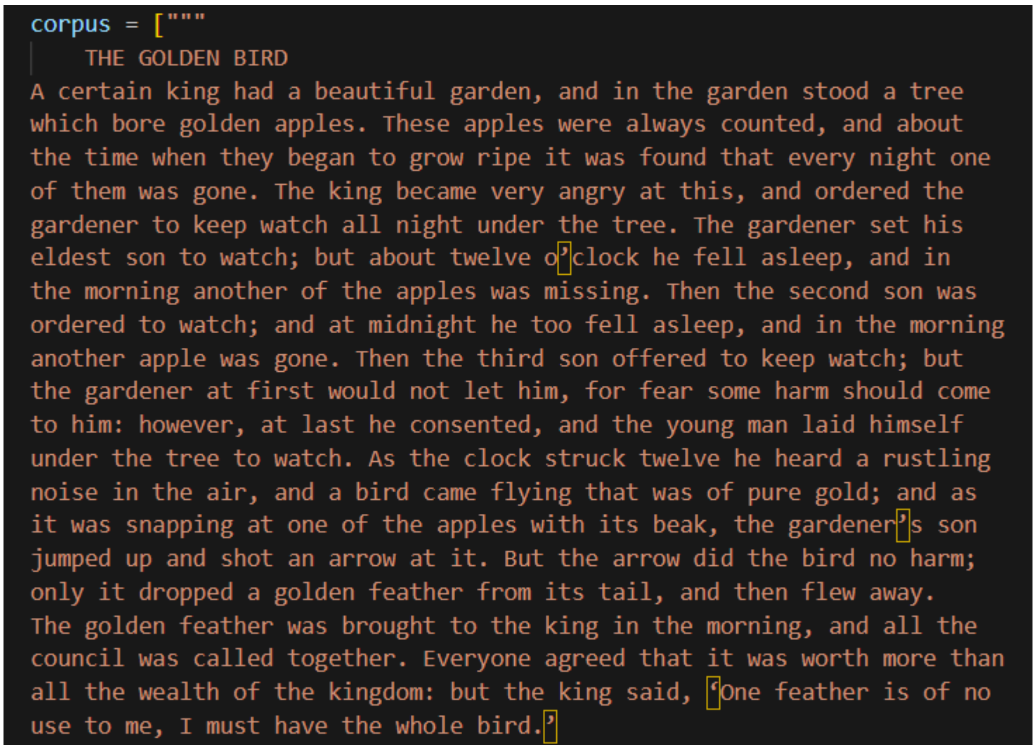
RU Computer Science

September 21, 2025

Contents

1 Part 2, Tokenization

For this task, I selected a small corpus consisting of short stories by the Brothers Grimm. I chose this corpus because fairy tales provide a rich and varied vocabulary, including both everyday words and archaic or domain-specific terms related to folklore, magic, and nature. This makes the dataset particularly interesting for studying how tokenization methods handle uncommon or compound words. In addition, the stories are short and self-contained, which makes the corpus easy to manage while still providing enough linguistic diversity to highlight differences between WordPiece and Unigram/SentencePiece. Beyond the experiment, such a tokenizer could be useful for tasks like building language models for story generation, automatic summarization of children's literature, or text analysis of folklore traditions.



```
corpus = ["""
    THE GOLDEN BIRD
A certain king had a beautiful garden, and in the garden stood a tree
which bore golden apples. These apples were always counted, and about
the time when they began to grow ripe it was found that every night one
of them was gone. The king became very angry at this, and ordered the
gardener to keep watch all night under the tree. The gardener set his
eldest son to watch; but about twelve o’clock he fell asleep, and in
the morning another of the apples was missing. Then the second son was
ordered to watch; and at midnight he too fell asleep, and in the morning
another apple was gone. Then the third son offered to keep watch; but
the gardener at first would not let him, for fear some harm should come
to him: however, at last he consented, and the young man laid himself
under the tree to watch. As the clock struck twelve he heard a rustling
noise in the air, and a bird came flying that was of pure gold; and as
it was snapping at one of the apples with its beak, the gardener’s son
jumped up and shot an arrow at it. But the arrow did the bird no harm;
only it dropped a golden feather from its tail, and then flew away.
The golden feather was brought to the king in the morning, and all the
council was called together. Everyone agreed that it was worth more than
all the wealth of the kingdom: but the king said, ‘One feather is of no
use to me, I must have the whole bird.’
```

Figure 1: Brother Grimm Corpus

1.1 What is the fundamental difference between each tokenization method?

The fundamental difference between each tokenization method is that WordPiece builds its vocabulary step by step by joining the most common pairs of characters or subwords it finds in the training data. Unigram takes another approach. It begins with a large set of possible tokens and then removes the ones that don't help much in representing the data.

Over time, Wordpiece creates tokens that match frequent patterns in the language, like endings or common word roots. Because of this, very frequent words or parts of words get their own tokens, while rare words are broken into smaller, reusable pieces.

Instead of always merging frequent pairs, Unigram tries to keep the tokens that best explain the text overall. This means there can be several valid ways to break down a word, and later the model can pick the most likely option depending on the sentence. In practice, this gives Unigram more flexibility, since it adapts better to different contexts, while WordPiece is more strict in its choices.

1.2 In the WordPiece script, what does the vocabsize variable do? What effect does it have on the output and the runtime to increase or decrease it by a factor of 10?

In the WordPiece algorithm, the parameter vocabsize defines the maximum number of tokens that will be included in the tokenizer's vocabulary. The merge of pairs continues until the vocabulary reach the desidered lenght.

A larger vocabulary allows the algorithm to keep more complete words and frequent subword units, which reduces the average number of tokens per sentence and produces segmentations that are closer to whole words.

Conversely, a smaller vocabulary forces the tokenizer to represent words using smaller and more frequent subparts, resulting in longer sequences.

Adjusting this parameter also affects runtime: increasing vocabsize by a large factor slows down the training of the tokenizer, since more merges must be computed and stored, and it leads to a larger embedding matrix in the final model, which increases memory usage and computational cost. Decreasing vocabsize makes training faster and the model lighter, but at the cost of more fragmented tokenization.

Below you will find the vocabulary defined with different vocabsize, the difference between us are visible looking at the different tokens expecially at the end of sequence.

```
100 = ['[PAD]', '[UNK]', '[CLS]', '[SEP]', '[MASK]', '!', '##D', '##E', '##H', '##I', '##
    ↪ L', '##N', '##O', '##R', '##a', '##b', '##c', '##d', '##e', '##f', '##g', '##h', '
    ↪ ##i', '##j', '##k', '##l', '##m', '##n', '##o', '##p', '##r', '##s', '##t', '##u',
    ↪  '##v', '##w', '##x', '##y', '##z', ',', '-', '.', ':', ';', '?', 'A', 'B', 'C', '
    ↪ D', 'E', 'G', 'H', 'I', 'L', 'N', 'O', 'P', 'S', 'T', 'W', 'Y', 'a', 'b', 'c', 'd'
    ↪ , 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
    ↪ 'u', 'v', 'w', 'y', ''', ''', '"', '"', 'ab', '##OL', '##IR', '##HE', '##EN', '##
    ↪ OLD', '##OLDEN', '##IRD', 'GOLDEN', 'BIRD', 'THE']
110 = ['[PAD]', '[UNK]', '[CLS]', '[SEP]', '[MASK]', '!', '##D', '##E', '##H', '##I', '##
    ↪ L', '##N', '##O', '##R', '##a', '##b', '##c', '##d', '##e', '##f', '##g', '##h', '
    ↪ ##i', '##j', '##k', '##l', '##m', '##n', '##o', '##p', '##r', '##s', '##t', '##u',
    ↪  '##v', '##w', '##x', '##y', '##z', ',', '-', '.', ':', ';', '?', 'A', 'B', 'C', '
    ↪ D', 'E', 'G', 'H', 'I', 'L', 'N', 'O', 'P', 'S', 'T', 'W', 'Y', 'a', 'b', 'c', 'd'
    ↪ , 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
    ↪ 'u', 'v', 'w', 'y', ''', ''', '"', '"', 'ab', '##OL', '##IR', '##HE', '##EN', '##
    ↪ OLD', '##OLDEN', '##IRD', 'GOLDEN', 'BIRD', 'THE', '##bb', 'Ev', 'up', '##bby', '
    ↪ ev', 'qu', 'of', '##pp', 'Bu', 'ob']
90 = ['[PAD]', '[UNK]', '[CLS]', '[SEP]', '[MASK]', '!', '##D', '##E', '##H', '##I', '##L
    ↪ ', '##N', '##O', '##R', '##a', '##b', '##c', '##d', '##e', '##f', '##g', '##h', '
    ↪ ##i', '##j', '##k', '##l', '##m', '##n', '##o', '##p', '##r', '##s', '##t', '##u',
    ↪  '##v', '##w', '##x', '##y', '##z', ',', '-', '.', ':', ';', '?', 'A', 'B', 'C', '
    ↪ D', 'E', 'G', 'H', 'I', 'L', 'N', 'O', 'P', 'S', 'T', 'W', 'Y', 'a', 'b', 'c', 'd'
    ↪ , 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
    ↪ 'u', 'v', 'w', 'y', ''', ''', '"', '"', 'ab']
```

And in the end we can see the difference between the same sentence tokenized with the different vocabularies, that makes obvious that a smaller vocabulary leads to longer output of the tokenizer, and viceversa.

```
originalSentence="THE GOLDEN BIRD: the biggest hobby of all!"
```

```
100=['THE', 'GOLDEN', 'BIRD', ':', 't', '##h', '##e', 'b', '##i', '##g', '##g', '##e', '
    ↪ ##s', '##t', 'h', '##o', '##b', '##b', '##y', 'o', '##f', 'a', '##l', '##l', '!']
110=['THE', 'GOLDEN', 'BIRD', ':', 't', '##h', '##e', 'b', '##i', '##g', '##g', '##e', '
    ↪ ##s', '##t', 'h', '##o', '##bby', 'of', 'a', '##l', '##l', '!']
90=['T', '##H', '##E', 'G', '##O', '##L', '##D', '##E', '##N', 'B', '##I', '##R', '##D',
    ↪ ':', 't', '##h', '##e', 'b', '##i', '##g', '##g', '##e', '##s', '##t', 'h', '##o',
    ↪  '##b', '##b', '##y', 'o', '##f', 'a', '##l', '##l', '!']
```

1.3 In the Unigram/SentencePiece script, what does the percenttoremove control? What happens if you change that value? Answer in general and for your corpus in particular.

In the Unigram/SentencePiece script the percentToRemove control determines how aggressively the vocabulary is pruned at each step. A higher value results in the removal of a larger fraction of the least useful tokens, which forces the algorithm to rely on smaller, more frequent subwords to represent rare or complex words. Consequently, tokenization becomes more fragmented, as words are split into smaller pieces.

Conversely, a lower percentToRemove value removes fewer tokens at each step, preserving larger subwords and allowing more words to remain as bigger, more whole-word-like units. Thus, adjusting this parameter directly influences the granularity of the resulting tokenization, balancing between finer subword splits and larger, more recognizable tokens. Testing the algorithm on my corpus and tokenizing the same phrase from the last subsection, we can see that the results remain the same even when varying the percentToremove. The only practical difference is in the execution time, which increases as the percentToremove value decreases. originalSentence="THE GOLDEN BIRD: the biggest hobby of all!"

```
0.1=['   ', 'T', 'H', 'E', '   ', 'G', 'O', 'L', 'D', 'E', 'N', '   ', 'B', 'I', 'R', 'D'
    ↪ , ':', ' the ', ' b ', 'i', 'g', 'g', 'e', 's', 't', 'y', ' of ', ' a ', 'll',
    ↪  '!'] -->12sec
0.2=['   ', 'T', 'H', 'E', '   ', 'G', 'O', 'L', 'D', 'E', 'N', '   ', 'B', 'I', 'R', 'D'
    ↪ , ':', ' the ', ' b ', 'i', 'g', 'g', 'e', 's', 't', 'y', ' of ', ' a ', 'll',
    ↪  '!'] -->6 sec
0.05=['   ', 'T', 'H', 'E', '   ', 'G', 'O', 'L', 'D', 'E', 'N', '   ', 'B', 'I', 'R', 'D
    ↪ ', ':', ' the ', ' b ', 'i', 'g', 'g', 'e', 's', 't', 'y', ' of ', ' a ', 'll'
    ↪ , '!'] -->24 sec
```

1.4 Which tokenization method was the most useful for your corpus and why?

In general, both WordPiece and SentencePiece achieve similar outcomes in terms of overall coverage, but they differ in the degree of fragmentation they introduce. WordPiece often produces more split units, which can increase the length of the tokenized sequence, while SentencePiece tends to generate more compact and consistent subwords.

In my corpus, this difference became clear: although both methods captured the main linguistic elements, SentencePiece yielded cleaner and more interpretable tokens, whereas WordPiece introduced a higher number of small fragments. As a result, SentencePiece provided a representation that was more faithful to the structure of the text and better suited to the characteristics of the data. For instance, in some passages WordPiece broke common words into many small pieces, while SentencePiece kept them as more coherent units, leading to shorter and more natural sequences. The results are visible in the precedent subsections.

2 Part 4, Bias in Word Embeddings

Word embeddings, such as those from GloVe Wiki and GloVe Twitter, capture statistical relationships from large text corpora, in order to represent mathematically the meaning of the world. Using this vectors is possible to calculate similarities between them considering their direction and position.

Bias in word embeddings is a direct consequence of the text corpus used to train them. Both the Wikipedia and Twitter GloVe embeddings encode a range of social biases, which can be systematically uncovered by analyzing the relationships between words representing gender, profession, ethnicity, and social class.

The simplest way to spot a bias is to analyze the similarities between words of the same type of bias, like showed in the code above:

```python
def bias_reference_similarity(wv, categories, reference_words):
    for category, words in categories.items():
        print(f"\n{category.capitalize()} bias (reference similarity):")
        for word in words + reference_words:
            if word in wv:
                sims = []
                for ref in reference_words:
                    if ref in wv:
                        sim = wv.similarity(word, ref)
                        sims.append((ref, sim))
                if sims:
                    print(f"Similarity of {word} to references: {sims}")
```

The results of this approach leads to find different type of bias. We are talking about **social bias** when people are judged or stereotyped because of their background, culture, or social group. In our code using similarities respect to man and woman, and a list of social terms we can observe that the terms boss appears for man but not for woman. This made clear that the embeddings are more used to a man boss respect to a woman boss.

```
Category: social, Reference: man
  servant (similarity: 0.623)
  boss (similarity: 0.583)
  poor (similarity: 0.532)
Category: social, Reference: woman
  servant (similarity: 0.592)
  employee (similarity: 0.511)
  poor (similarity: 0.508)
```

We can also observe a **profession bias** when certain jobs get linked more strongly to a specific social group, leading to stereotypes. In this case we have nurse similar to woman but not to man. A clear bias representing the fact that that work is usually seen as a woman-only job.

```
Category: profession, Reference: man
  doctor (similarity: 0.712)
  teacher (similarity: 0.639)
  lawyer (similarity: 0.607)
Category: profession, Reference: woman
  doctor (similarity: 0.725)
  nurse (similarity: 0.716)
  teacher (similarity: 0.685)
```

These biases are not limited to direct words similarities. We can also perform **analogies** exploiting the geometric properties of vector spaces to capture semantic relationships between concepts. By comparing directions and distances among these vectors, it becomes possible to model analogical reasoning, such as the classic example “king – man + woman queen”. This arises because embeddings learned from large datasets tend to encode semantic and syntactic patterns in their geometry, enabling arithmetic operations on vectors to reveal meaningful relationships. Above you can find the code used to perform the analogy comparison:

```
def bias_analogy_comparison(wv, analogy_pairs, topn=3):
    print("\nAnalogy comparisons (top results):")
    for (a, b, c) in analogy_pairs:
        if all(w in wv for w in [a, b, c]):
            results = wv.most_similar(positive=[b, c], negative=[a], topn=topn)
            print(f"{a}:{b} as {c}:? =>")
            for word, score in results:
                print(f"  {word} (score: {score:.3f})")
        else:
            missing = [w for w in [a, b, c] if w not in wv]
            print(f"Words missing for analogy: {a}, {b}, {c} (missing: {missing})")
```

Using analogies you can find **etnicity bias**, like the one resulting from “European : privileged as African : ?”, that shows how embeddings can reproduce bias. The model linked European with privileged and suggested words like educated, respected, and oppressed for African. These associations are not objective truths but instead reflect cultural stereotypes and historical inequalities encoded in the training data. In this case, the bias comes from how different groups are represented in text: Europeans are often described in positions of power or privilege, while Africans are more frequently associated with oppression or lack of access to education.

```
european:privileged as african:? =>
  educated (score: 0.747)
  respected (score: 0.712)
  oppressed (score: 0.701)
```

By comparing GloVe Wiki and GloVe Twitter embeddings, we can observe differences in bias. The nature and degree of bias differ significantly between Wikipedia and Twitter embeddings.

Wikipedia(**glove-wiki-gigaword-100**), as a curated and encyclopedic resource, tends to encode more formal, historical, and institutional biases. For example, professions may be more strongly gendered in line with traditional roles, and ethnic or religious terms may reflect long-standing societal narratives.

Twitter(**glove-twitter-100**), in contrast, is a dynamic and informal platform, capturing the language of everyday discourse, trending topics, and the opinions of a diverse user base. The biases present in Twitter embeddings are often more reflective of current events, popular culture, and the demographic makeup of active users. This can result in the amplification of certain stereotypes, the emergence of new associations, and sometimes the propagation of slang or derogatory terms.

For example Wikipedia embeddings link "rich" and "poor" to descriptive or neutral terms, instead Twitter embeddings show more emotional and informal associations, such as "rich" with "money" and "poor" with "shame".

```
Wikipedia (GloVe Wiki):

Most similar to rich:
[('natural', 0.756), ('especially', 0.728), ('particularly', 0.706), ('vast', 0.705), ('
    ↪ abundant', 0.701)]
```

```
Most similar to poor:
[('especially', 0.804), ('better', 0.770), ('bringing', 0.769), ('particularly', 0.768),
    ↪ ('lack', 0.765)]

Twitter (GloVe Twitter):

Most similar to rich:
[('grown', 0.838), ('young', 0.831), ('money', 0.822), ('fat', 0.791), ('small', 0.784)]
Most similar to poor:
[('shame', 0.804), ('well', 0.791), ('how', 0.783), ('such', 0.776), ('but', 0.774)]
```

Another example is Wikipedia embeddings that reflect stereotypical professional associations, such as "doctor" being close to "nurse" and "physician", and on the other side Twitter embeddings show more varied and less stereotypical associations, with "doctor" linked to "doc" and "death", and "nurse" to "receptionist" and "assistant".

```
Wikipedia (GloVe Wiki):

Most similar to doctor:
[('nurse', 0.798), ('physician', 0.797), ('patient', 0.761), ('child', 0.756), ('teacher'
    ↪ , 0.754)]
Most similar to nurse:
[('doctor', 0.798), ('nurses', 0.775), ('dentist', 0.773), ('pregnant', 0.746), ('
    ↪ pediatrician', 0.745)]

Twitter (GloVe Twitter):

Most similar to doctor:
[('doc', 0.826), ('has', 0.739), ('child', 0.737), ('dr.', 0.729), ('death', 0.728)]
Most similar to nurse:
[('nursing', 0.875), ('receptionist', 0.841), ('assistant', 0.812), ('physician', 0.806),
    ↪ ('therapist', 0.792)]
```

Regarding visualizations we can use methods such as **PCA**(Principal Component Analysis) that is a linear dimensionality reduction method that projects high-dimensional data onto directions of maximum variance. It is fast, easy to interpret, and preserves global structure, but may miss complex nonlinear relationships in embeddings. Using it we can reveal clustering and associations that reflect bias. Plotting embeddings can show how these concepts are grouped.

```
    def visualize_embeddings(wv, words, title, save_path=None, reference=None):
    words_present = [w for w in words if w in wv]
    if reference and reference in wv:
        # Sort by similarity to reference
        words_present.sort(key=lambda w: wv.similarity(w, reference), reverse=True)
    vectors = [wv[w] for w in words_present]
    pca = PCA(n_components=2)
    coords = pca.fit_transform(vectors)
    plt.figure(figsize=(8,6))
    for word, (x, y) in zip(words_present, coords):
        plt.scatter(x, y)
        plt.text(x+0.01, y+0.01, word)
    plt.title(title)
    if save_path:
        plt.savefig(save_path)
        plt.close()
    else:
        plt.show()
```

Such plots often show that gendered words and certain professions cluster together, visually demonstrating the bias encoded in the embeddings. As you can see above for example “man” is closer to “engineer” and “woman” to “nurse”.

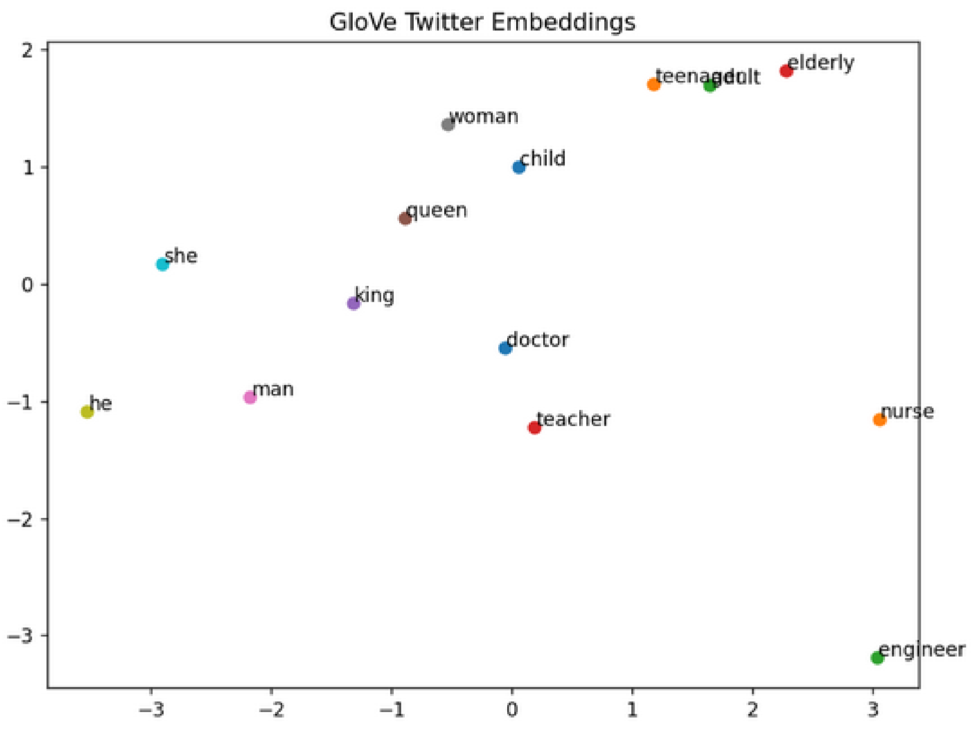
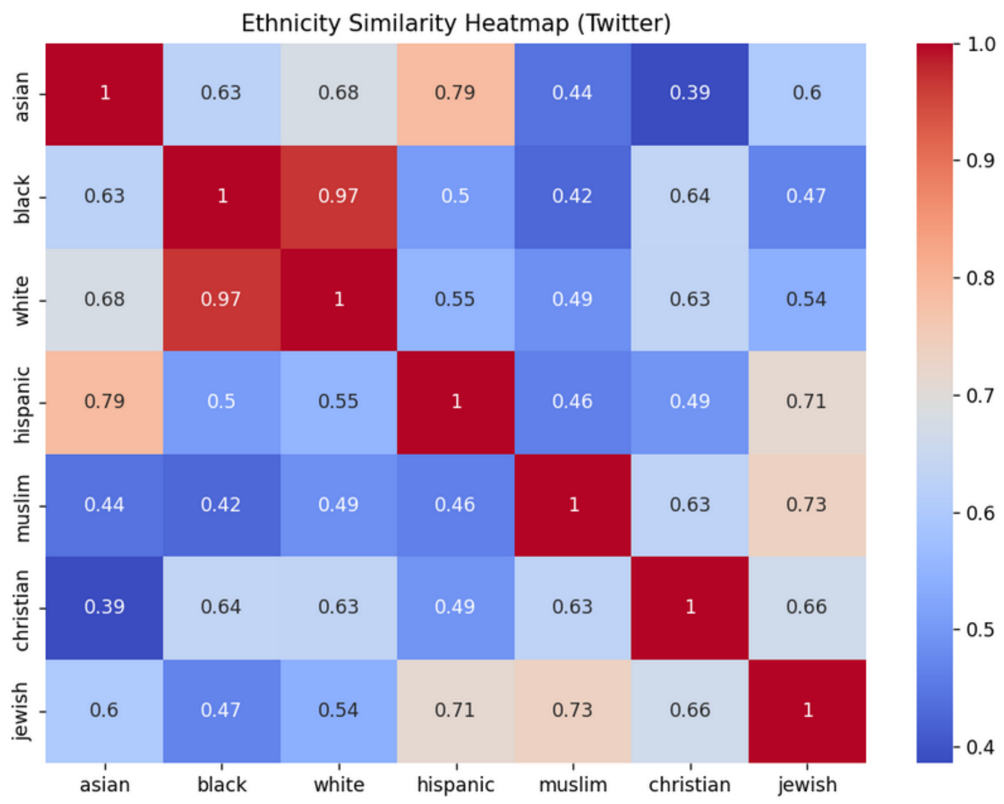


Figure 2: PCA

We can also use another 2d method called **t-SNE** that is a nonlinear method for visualizing high-dimensional data like embeddings. It preserves local relationships, so similar points cluster together in 2D or 3D. While great for spotting groups, it is sensitive to hyperparameters and may distort global distances. Another method of presenting biases may be a heatmap to visualize better which concepts are more related to each other.

```
    def visualize_heatmap(wv, words, title):
        matrix = np.zeros((len(words), len(words)))
        for i, w1 in enumerate(words):
                for j, w2 in enumerate(words):
                        if w1 in wv and w2 in wv:
                                matrix[i, j] = wv.similarity(w1, w2)
        plt.figure(figsize=(8,6))
        sns.heatmap(matrix, xticklabels=words, yticklabels=words, annot=True, cmap="
            ↪ coolwarm")
        plt.title(title)
        plt.tight_layout()
        plt.show()
```



	asian	black	white	hispanic	muslim	christian	jewish
asian	1	0.63	0.68	0.79	0.44	0.39	0.6
black	0.63	1	0.97	0.5	0.42	0.64	0.47
white	0.68	0.97	1	0.55	0.49	0.63	0.54
hispanic	0.79	0.5	0.55	1	0.46	0.49	0.71
muslim	0.44	0.42	0.49	0.46	1	0.63	0.73
christian	0.39	0.64	0.63	0.49	0.63	1	0.66
jewish	0.6	0.47	0.54	0.71	0.73	0.66	1

Figure 3: embeddings heatmap

Debiasing techniques aim to mitigate these unwanted associations, making NLP models fairer and more reliable.
A common debiasing approach is to identify a bias direction—such as the vector difference between "he" and "she" for gender—and then project word vectors orthogonally to this direction(Hard debiasing). This process removes the component of each word vector that aligns with the bias, neutralizing associations for words that should be unbiased (e.g., professions). In the code, the debiasWords function applies this method to a list of profession words, ensuring that their embeddings are less influenced by gender bias. This not only improves fairness but also helps downstream models make decisions based on meaning rather than stereotypes.

```python
def debias_embedding(wv, words, bias_directions):
   debiased_vectors = {}
   for word in words:
       if word in wv:
           vector = wv[word].copy()
           for direction in bias_directions:
               vector -= np.dot(vector, direction) * direction
           debiased_vectors[word] = vector
   return debiased_vectors
```

There are also others techniques:

- **Soft debiasing**: reduces but does not eliminate bias, allowing embeddings to retain some semantic distinctions
- **Balanced training data**: ensure equal representation of male/female examples or under-represented groups
- **Contextual embeddings (BERT, GPT)**: they adapt meaning based on context, which reduces but does not eliminate bias