

## ▼ Lab 3 - Part 2: Word and Sentence Embeddings

### Objectives:

- Understand and implement Word2Vec (CBOW and Skip-gram)
- Work with pre-trained GloVe embeddings
- Use BERT for sentence embeddings
- Compare different embedding approaches
- Apply embeddings to find similar words and documents

### Instructions

1. Complete all exercises marked with `# YOUR CODE HERE`
2. Answer all written questions in the designated markdown cells
3. Save your completed notebook
4. Push to your Git repository and send the link to: [yoroba93@gmail.com](mailto:yoroba93@gmail.com)

Important: This lab continues from Part 1

You will use the same dataset and categories you chose in Part 1.

## ▼ Setup

```

1 !pip install gensim

Requirement already satisfied: gensim in /usr/local/lib/python3.12/dist-packages (4.4.0)
Requirement already satisfied: numpy>=1.18.5 in /usr/local/lib/python3.12/dist-packages (from gensim) (2.0.2)
Requirement already satisfied: scipy>=1.7.0 in /usr/local/lib/python3.12/dist-packages (from gensim) (1.16.3)
Requirement already satisfied: smart_open>=1.8.1 in /usr/local/lib/python3.12/dist-packages (from gensim) (7.5.0)
Requirement already satisfied: wrapt in /usr/local/lib/python3.12/dist-packages (from smart_open>=1.8.1->gensim)

1 # Install required libraries (uncomment if needed)
2 # !pip install gensim transformers torch sentence-transformers datasets

1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 from collections import Counter
5 import re
6 import string
7 import warnings
8 warnings.filterwarnings('ignore')
9
10 import nltk
11 nltk.download('punkt', quiet=True)
12 nltk.download('stopwords', quiet=True)
13 nltk.download('wordnet', quiet=True)
14
15 from nltk.tokenize import word_tokenize
16 from nltk.corpus import stopwords
17 from nltk.stem import WordNetLemmatizer
18
19 import gensim
20 from gensim.models import Word2Vec, KeyedVectors
21 import gensim.downloader as api
22
23 print(f"Gensim version: {gensim.__version__}")
24 print("Setup complete!")

Gensim version: 4.4.0
Setup complete!

```

## ▼ Load Dataset (Same as Part 1)

```

1 import pandas as pd
2
3 # Load the dataset
4 df = pd.read_csv('/content/20_newsgroups_train.csv')
5

```

```

6 # TODO: Use the SAME 3 categories you chose in Part 1!
7 my_categories = ["alt.atheism", "comp.graphics", "sci.space"] # COPY FROM PART 1
8
9 # Filter dataset
10 df_filtered = df[df['label_text'].isin(my_categories)].copy()
11 df_filtered = df_filtered.reset_index(drop=True)
12
13 print(f"Selected categories: {my_categories}")
14 print(f"Filtered dataset size: {len(df_filtered)}")

```

Selected categories: ['alt.atheism', 'comp.graphics', 'sci.space']  
 Filtered dataset size: 1657

```

1 import nltk
2
3 nltk.download('punkt')
4 nltk.download('punkt_tab')
5 nltk.download('stopwords')
6 nltk.download('wordnet')
7

[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Package punkt_tab is already up-to-date!
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
True

```

```

1  # Preprocessing function (same as Part 1)
2  stop_words = set(stopwords.words('english'))
3  lemmatizer = WordNetLemmatizer()
4
5  def preprocess_text(text):
6      """Preprocess text for embedding training."""
7      #text = "" # YOUR CODE HERE => the same as in Part 1 (advanced preprocessing)
8      #tokens = [] # YOUR CODE HERE => the same as in Part 1 (advanced preprocessing)
9      #return tokens # Return list of tokens for Word2Vec
10     # Step 1: Basic cleaning
11     if not isinstance(text, str):
12         return []
13
14     # Step 1: Basic cleaning
15     text = text.lower()
16     text = re.sub(r'\S+@\S+', '', text)
17     text = re.sub(r'http\S+|www\S+', '', text)
18     text = re.sub(r'\d+', '', text)
19     text = text.translate(str.maketrans('', '', string.punctuation))
20
21     # Step 2: Tokenize (regex-based, no NLTK punkt)
22     tokens = re.findall(r'\b[a-z]+\b', text)
23
24     # Step 3: Remove stop words
25     tokens = [w for w in tokens if w not in stop_words]
26
27     # Step 4: Lemmatize
28     tokens = [lemmatizer.lemmatize(w) for w in tokens]
29
30     # Step 5: Remove short words (< 3 chars)
31     tokens = [w for w in tokens if len(w) >= 3]
32
33     return tokens
34
35     # Apply preprocessing
36     df_filtered['tokens'] = df_filtered['text'].apply(preprocess_text)
37     df_filtered['text_clean'] = df_filtered['tokens'].apply(' '.join)
38
39     print(f"Sample tokens: {df_filtered.iloc[0]['tokens'][:20]}")
40

```

Sample tokens: ['weiteks', 'addressphone', 'number', 'like', 'get', 'information', 'chip']

1 Start coding or generate with AI.

## Part A: Word2Vec - Training Your Own Embeddings

Word2Vec learns word representations by predicting context. There are two architectures:

- **CBOW (Continuous Bag of Words)**: Predicts target word from context words
- **Skip-gram**: Predicts context words from target word

## ▼ A.1 Understanding Word2Vec Architectures

```

1 # Prepare corpus for Word2Vec (list of tokenized sentences)
2 corpus = df_filtered['tokens'].tolist()
3
4 print(f"Corpus size: {len(corpus)} documents")
5 print(f"Total tokens: {sum(len(doc) for doc in corpus)}")
6 print(f"\nSample document tokens: {corpus[0][:15]}")

```

Corpus size: 1657 documents  
Total tokens: 158687

Sample document tokens: ['weiteks', 'addressphone', 'number', 'like', 'get', 'information', 'chip']

```

1 # Train Word2Vec with CBOW (sg=0)
2 model_cbow = Word2Vec(
3     sentences=corpus,
4     vector_size=100,           # Embedding dimension
5     window=5,                 # Context window size
6     min_count=5,              # Ignore words with freq < 5
7     workers=4,                 # Parallel threads
8     sg=0,                      # 0 = CBOW, 1 = Skip-gram
9     epochs=10                  # Training epochs
10 )
11
12 print(f"CBO Model trained!")
13 print(f"Vocabulary size: {len(model_cbow.wv)}")

```

CBO Model trained!  
Vocabulary size: 4745

```

1 # Train Word2Vec with Skip-gram (sg=1)
2 model_skipgram = Word2Vec(
3     sentences=corpus,
4     vector_size=100,
5     window=5,
6     min_count=5,
7     workers=4,
8     sg=1,                      # Skip-gram
9     epochs=10
10 )
11
12 print(f"Skip-gram Model trained!")
13 print(f"Vocabulary size: {len(model_skipgram.wv)}")

```

Skip-gram Model trained!  
Vocabulary size: 4745

## ▼ A.2 Exploring Word Embeddings

```

1 # Example: Get word vector
2 sample_word = "computer" # Change this to a word relevant to YOUR categories
3
4 if sample_word in model_cbow.wv:
5     vector = model_cbow.wv[sample_word]
6     print(f"Vector for '{sample_word}':")
7     print(f" Shape: {vector.shape}")
8     print(f" First 10 values: {vector[:10]}")
9 else:
10    print(f"'{sample_word}' not in vocabulary. Try another word.")
11    print(f"Sample words in vocab: {list(model_cbow.wv.key_to_index.keys())[:20]}")

```

Vector for 'computer':  
Shape: (100,)  
First 10 values: [-0.8928408 0.8846241 0.7422608 -0.67154706 0.1874886 -0.6199554  
0.11242337 0.7537508 -0.4629527 -0.2601366 ]

```

1 # Find similar words
2 sample_word = "computer" # Change to a word in YOUR vocabulary
3
4 if sample_word in model_cbow.wv:
5     print(f"\nWords most similar to '{sample_word}' (CBOW):")
6     for word, score in model_cbow.wv.most_similar(sample_word, topn=10):
7         print(f" {word}: {score:.4f}")
8

```

```

9     print(f"\nWords most similar to '{sample_word}' (Skip-gram):")
10    for word, score in model_skipgram.wv.most_similar(sample_word, topn=10):
11        print(f"  {word}: {score:.4f}")

```

Words most similar to 'computer' (CBOW):

```

user: 0.9840
application: 0.9763
database: 0.9755
visualization: 0.9661
tool: 0.9635
analysis: 0.9614
processing: 0.9613
plotting: 0.9611
nbdsp: 0.9599
interactive: 0.9595

```

Words most similar to 'computer' (Skip-gram):

```

exponent: 0.7873
graeme: 0.7350
switzerland: 0.7310
art: 0.7190
berlin: 0.7157
molecular: 0.7131
network: 0.7128
silicon: 0.7077
gem: 0.7035
sutherland: 0.7007

```

## Exercise A.1: Compare CBOW vs Skip-gram

Choose 5 words that are relevant to YOUR 3 categories and compare the most similar words from both models.

```

1 # TODO: Choose 5 words relevant to YOUR categories
2 # These should be domain-specific words (not common words like "good", "make", etc.)
3
4 my_test_words = ["exponent", "graphic", "interactive", "switzerland", "network"] # YOUR WORDS HERE
5
6 comparison_results = []
7
8 for word in my_test_words:
9     word = word.lower()
10    if word in model_cbow.wv and word in model_skipgram.wv:
11        cbow_similar = [w for w, s in model_cbow.wv.most_similar(word, topn=5)]
12        skipgram_similar = [w for w, s in model_skipgram.wv.most_similar(word, topn=5)]
13
14        comparison_results.append({
15            'word': word,
16            'cbow_top5': cbow_similar,
17            'skipgram_top5': skipgram_similar
18        })
19
20        print(f"\n'{word}':")
21        print(f"  CBOW: {cbow_similar}")
22        print(f"  Skip-gram: {skipgram_similar}")
23    else:
24        print(f"'{word}' not found in vocabulary!")

```

```

'exponent':
  CBOW: ['gem', 'processor', 'raytracers', 'optional', 'indigo']
  Skip-gram: ['gem', 'silicon', 'procedural', 'cad', 'beta']

'graphic':
  CBOW: ['package', 'library', 'software', 'run', 'window']
  Skip-gram: ['exponent', 'gem', 'radiosity', 'cad', 'raytracing']

'interactive':
  CBOW: ['digital', 'sphinx', 'environment', 'visualization', 'nbdsp']
  Skip-gram: ['widget', 'computation', 'gui', 'visualisation', 'hip']

'switzerland':
  CBOW: ['academic', 'pace', 'seminar', 'institution', 'nssdc']
  Skip-gram: ['ltd', 'wavefront', 'bibliography', 'marc', 'stanford']

'network':
  CBOW: ['kit', 'environment', 'ephemeris', 'capability', 'integrated']
  Skip-gram: ['cray', 'ibmpc', 'aips', 'integrated', 'runtime']

```

## Written Question A.1 (Personal Interpretation)

Based on your comparison above:

1. For which words did CBOW and Skip-gram give **SIMILAR** results?
2. For which words did they give **DIFFERENT** results?
3. Which model seems to capture better semantic relationships for **YOUR specific domain?** Explain with examples.
4. Why might one model work better than the other for certain types of words? (Think about word frequency)

#### YOUR ANSWER:

1. Similar results for: ...

graphic

CBOW: package, library, software, workshop, run

Skip-gram: exponent, gem, cad, programming, indigo

Why similar?

Both models associate graphic with software / computing / visualization-related terms.

Even though the exact words differ, they live in the same semantic neighborhood (graphics software, programming, CAD, visualization).

👉 Conclusion: Semantically similar, stylistically different.

◆ interactive

CBOW: digital, visualization, map, quantitative, various

Skip-gram: widget, visualisation, gui, computation, kernel

Why similar?

Both models strongly associate interactive with:

visualization

computation

user interfaces

CBOW is more abstract, Skip-gram is more technical, but the meaning overlaps strongly.

👉 Conclusion: High semantic agreement.

◆ network

CBOW: distributed, spyglass, environment, ibm, datasets

Skip-gram: ibmpc, aips, geographic, nbdsp, visualisation

Why similar?

Both models associate network with:

computing

enterprise / IBM-related terms

data environments

Skip-gram gives more specific technical artifacts, CBOW stays broader.

👉 Conclusion: Same domain, different granularity → still similar.

✓ Words with **SIMILAR** results

graphic

interactive

network

2. Different results for: ...

exponent

CBOW: linear, nonlinear, lisp, phigs, graeme

Skip-gram: silicon, gem, procedural, flexibility, iris

Why different?

CBOW links exponent to mathematical / theoretical concepts.

Skip-gram links it to hardware, materials, and procedural contexts.

👉 Clear semantic divergence.

◆ switzerland

CBOW: fee, institution, academic, cambridge, guide

Skip-gram: ltd, campus, sigkids, jay, tel

Why different?

CBOW associates Switzerland with academia and institutions.

Skip-gram associates it with organizations, locations, and entities.

👉 Different contextual interpretations of the same word.

✗ Words with DIFFERENT results

exponent

switzerland

3. Better model for my domain: ...

- Example 1: ...
- Example 2: ...

Skip-gram performs better overall for your domain

Your domain appears to be:

technical

academic

computing / visualization / software-oriented

🔍 Evidence from your results: Word Why Skip-gram is better interactive widget, gui, kernel → strong technical semantics graphic cad, programming, gem → domain-specific tools network ibmpc, visualisation → concrete technical entities

Skip-gram captures fine-grained, domain-specific semantics, which is ideal for:

technical corpora

academic papers

software documentation

4. Explanation of differences: ... CBOW is better for frequent words Skip-gram is better for rare or technical words.

CBOW (Continuous Bag of Words)

Predicts a word from its context Averages surrounding words

Advantages:

Fast

Stable

Works well for high-frequency words

Disadvantages:

Loses nuance

Less precise for rare terms

✗ Example from your results:

interactive → digital, various

Broad, generic associations

Skip-gram: Predicts context from a word Learns each context independently

Advantages:

Excellent for rare words

Captures precise semantic relations

Disadvantages:

Slower

Needs more data

📌 Example from your results:

interactive → widget, gui

graphic → cad

Much more informative for technical NLP tasks

### ▼ A.3 Word Analogies

```

1 # Example: Word analogies (king - man + woman = queen)
2 # This works better with larger, pre-trained models, but let's try with our custom model
3
4 def find_analogy(model, word1, word2, word3):
5     """
6         Find word that completes analogy: word1 is to word2 as word3 is to word4
7         Uses: word2 - word1 + word3 = word4
8     """
9     try:
10         result = model.wv.most_similar(
11             positive=[word2, word3],
12             negative=[word1],
13             topn=5
14         )
15         return result
16     except KeyError as e:
17         return f"Word not found: {e}"
18
19 # Test with your domain
20 # Example: "baseball" is to "bat" as "hockey" is to ?
21 print("Analogy test (your model may have limited vocabulary):")
22 #result = find_analogy(model_skipgram, "word1", "word2", "word3")
23 result = find_analogy(model_skipgram, "programming", "code", "visualization")
24 # Expected: "plot", "graphic", "chart"
25
26 print(result)

```

Analogy test (your model may have limited vocabulary):  
[('generator', 0.6813595294952393), ('ansi', 0.6579309105873108), ('fortran', 0.6467586159706116), ('license', 0.

```

1 result = find_analogy(model_skipgram, "gui", "widget", "interactive")
2 print(result, '\n')
3
4 result = find_analogy(model_skipgram, "data", "dataset", "network")
5 print(result, '\n')
6 result = find_analogy(model_skipgram, "university", "academic", "institution")
7 print(result, '\n')

[('visualization', 0.8757232427597046), ('computational', 0.8608754277229309), ('visualisation', 0.85764825344085

Word not found: "Key 'dataset' not present in vocabulary"

[('bulletin', 0.8600651025772095), ('variety', 0.8585644960403442), ('maintained', 0.8501346111297607), ('yee', 0

```

1 Start coding or generate with AI.

### ▼ Exercise A.2: Create Domain-Specific Analogies

Try to find **2 analogies** that work with YOUR dataset's vocabulary.

```

1 # TODO: Try 2 analogies with words from YOUR vocabulary
2 # Format: word1 is to word2 as word3 is to ?
3
4 # Analogy 1
5 # YOUR CODE HERE
6 analogy1 = find_analogy(model_skipgram, "programming", "code", "visualization")

```

```

7 print(f"Analogy 1: {analogy1}")
8
9 # Analogy 2
10 # YOUR CODE HERE
11 analogy2 = find_analogy(model_skipgram, "gui","widget","interactive")
12 print(f"Analogy 2: {analogy2}")

Analogy 1: [('generator', 0.6813595294952393), ('ansi', 0.6579309105873108), ('fortran', 0.6467586159706116), ('l
Analogy 2: [('visualization', 0.8757232427597046), ('computational', 0.8608754277229309), ('visualisation', 0.857

```

## ↙ Written Question A.2 (Personal Interpretation)

### Did your analogies work?

- If yes, explain why the result makes sense.
- If no, explain why they might have failed (vocabulary size, training data, etc.)

### YOUR ANSWER:

Analogy 2 worked reasonably well; Analogy 1 did not work cleanly. generator, license, distributed, ansi, manual Did it work?

Mostly NO (weak / noisy result).

Why it didn't work well:

Vocabulary limitation: Your corpus likely does not contain enough consistent examples of “visualization → output” relationships (e.g., plot, chart, graphic used in parallel with code).

Conceptual mismatch: code is a concrete artifact of programming.

```
visualization in your dataset appears more often in documentation / standards / libraries, not as a producer of a s
```

Small / domain-specific corpus: Analogy reasoning relies on linear semantic offsets, which typically emerge only in large, diverse corpora.

Skip-gram captured context, not analogy structure Words like generator, ansi, and manual reflect technical documentation contexts, not visualization outputs. Conclusion: The analogy failed because the semantic relationship (produces) was not consistently encoded in the training data.

Analogy 2: GUI → widget :: interactive → ? visualization, svlib, kernel, silicon, motif Did it work?

```
YES (partially successful and meaningful).
```

Why this result makes sense:

Strong semantic overlap

```
GUIs are built from widgets
```

Interactive systems are strongly associated with:

```
visualization
kernel (backend computation)
motif (UI toolkit)
```

Your earlier similarity results support this You already observed: interactive → widget, gui, visualization So the analogy aligns with existing semantic clusters.

Skip-gram is particularly good at:

```
capturing technical, low-frequency words
modeling component-level relationships
```

...

## ↙ Part B: Pre-trained GloVe Embeddings

GloVe (Global Vectors) is trained on much larger corpora and captures broader relationships.

```

1 # Load pre-trained GloVe embeddings (this may take a few minutes)
2 print("Loading GloVe embeddings (this may take a minute)...")
3 glove_model = api.load('glove-wiki-gigaword-100') # 100-dimensional vectors
4 print(f"GloVe loaded! Vocabulary size: {len(glove_model)}")

Loading GloVe embeddings (this may take a minute)...
[=====] 100.0% 128.1/128.1MB downloaded
GloVe loaded! Vocabulary size: 400000

```

```

1 # Compare: Same word in YOUR model vs GloVe
2 test_word = "computer" # Change to a word relevant to your domain
3
4 print(f"Similar words to '{test_word}':")
5 print("\nYour Word2Vec model:")
6 if test_word in model_skipgram.wv:
7     for word, score in model_skipgram.wv.most_similar(test_word, topn=10):
8         print(f" {word}: {score:.4f}")
9 else:
10    print(f" '{test_word}' not in vocabulary")
11
12 print("\nPre-trained GloVe:")
13 if test_word in glove_model:
14     for word, score in glove_model.most_similar(test_word, topn=10):
15         print(f" {word}: {score:.4f}")
16 else:
17    print(f" '{test_word}' not in vocabulary")

```

Similar words to 'computer':

Your Word2Vec model:  
exponent: 0.7873  
graeme: 0.7350  
switzerland: 0.7310  
art: 0.7190  
berlin: 0.7157  
molecular: 0.7131  
network: 0.7128  
silicon: 0.7077  
gem: 0.7035  
sutherland: 0.7007

Pre-trained GloVe:  
computers: 0.8752  
software: 0.8373  
technology: 0.7642  
pc: 0.7366  
hardware: 0.7290  
internet: 0.7287  
desktop: 0.7234  
electronic: 0.7222  
systems: 0.7198  
computing: 0.7142

## Exercise B.1: Compare Your Model vs GloVe

For **3 words from your domain**, compare the similar words from your trained model vs GloVe.

1 Start coding or generate with AI.

```

1 # TODO: Compare 3 domain-specific words
2
3 comparison_words = ["exponent", "switzerland", "network"] # YOUR WORDS
4
5 for word in comparison_words:
6     word = word.lower()
7     print(f"\n{'='*50}")
8     print(f"Word: '{word}'")
9     print(f"{'='*50}")
10
11    # Your model
12    print("Your Word2Vec:")
13    if word in model_skipgram.wv:
14        for w, s in model_skipgram.wv.most_similar(word, topn=5):
15            print(f" {w}: {s:.3f}")
16    else:
17        print(" Not in vocabulary")
18
19    # GloVe
20    print("GloVe:")
21    if word in glove_model:

```

```

22     for w, s in glove_model.most_similar(word, topn=5):
23         print(f" {w}: {s:.3f}")
24     else:
25         print(" Not in vocabulary")

```

```

=====
Word: 'exponent'
=====
Your Word2Vec:
    gem: 0.936
    silicon: 0.915
    procedural: 0.906
    cad: 0.904
    beta: 0.903
GloVe:
    exponents: 0.538
    unitary: 0.499
    originator: 0.463
    indices: 0.457
    indicator: 0.456

=====
Word: 'switzerland'
=====
Your Word2Vec:
    ltd: 0.954
    waveform: 0.949
    bibliography: 0.940
    marc: 0.938
    stanford: 0.935
GloVe:
    austria: 0.789
    germany: 0.779
    belgium: 0.757
    swiss: 0.743
    netherlands: 0.727

=====
Word: 'network'
=====
Your Word2Vec:
    cray: 0.905
    ibmpc: 0.903
    aips: 0.901
    integrated: 0.901
    runtime: 0.900
GloVe:
    networks: 0.904
    cable: 0.807
    channel: 0.784
    broadcast: 0.742
    channels: 0.740

```

#### ✓ Written Question B.1 (Personal Interpretation)

Compare your custom-trained Word2Vec model with pre-trained GloVe:

1. For which words does YOUR model give better (more relevant) similar words than GloVe? Why?
2. For which words does GloVe give better results? Why?
3. When would you use a custom-trained model vs a pre-trained model in a real project?

#### YOUR ANSWER:

1. My model is better for: ...
  - Reason: ... My model is better for:

exponent

network

Reason:

```

My custom Word2Vec model was trained on a technical and domain-specific corpus, so it captures contextual usage with
For exponent, my model associates the word with technical and computational contexts such as silicon, procedural, a
For network, the model retrieves technology-specific terms like ibmpc, aips, and visualisation, which align well with

```

These results reflect domain relevance, even though they may not match dictionary-style meanings.

2. GloVe is better for: ...

- Reason: ...

GloVe is better for:

switzerland

Reason:

```
GloVe is trained on very large, general-purpose corpora (e.g., Wikipedia and Common Crawl), making it better at capturing semantic relationships. For 'switzerland', GloVe correctly retrieves neighboring countries and related terms (Austria, Germany, Belgium, Switzerland). My custom model instead returns organization- and dataset-specific tokens ('ltd', 'campus', 'tel'), indicating limited expressiveness.
```

### 3. When to use each:

- Custom model: ...

When working with domain-specific data (technical documents, academic papers, software manuals)

When vocabulary contains specialized or rare terms

When contextual meaning differs from general language usage. Example: scientific, engineering, or enterprise datasets

- Pre-trained model: ... When working with general language understanding

When tasks rely on world knowledge (countries, people, common concepts)

When the dataset is small or lacks diversity Example: chatbots, news analysis, sentiment analysis

1 Start coding or generate with AI.

## ▼ B.2 GloVe Analogies

```
1 # Famous analogy: king - man + woman = queen
2 result = glove_model.most_similar(positive=['king', 'woman'], negative=['man'], topn=5)
3 print("king - man + woman = ?")
4 for word, score in result:
5     printf" {word}: {score:.4f}"
```

king - man + woman = ?  
 queen: 0.7699  
 monarch: 0.6843  
 throne: 0.6756  
 daughter: 0.6595  
 princess: 0.6521

```
1 # TODO: Try 3 more analogies with GloVe
2 # Be creative! Try analogies related to your categories.
3
4 # Analogy 1: ___ is to ___ as ___ is to ?
5 #result1 = glove_model.most_similar(positive=['___', '___'], negative=['___'], topn=3)
6 #print("Analogy 1:")
7 #print(result1)
8 result1 = glove_model.most_similar(
9     positive=['code', 'visualization'],
10    negative=['programming'],
11    topn=3
12 )
13 print("Analogy 1:")
14 print(result1)
15
16 # Analogy 2
17 # YOUR CODE HERE
18 result2 = glove_model.most_similar(
19     positive=['student', 'company'],
20     negative=['university'],
21     topn=3
22 )
23 print("Analogy 2:")
24 print(result2)
25
26 # Analogy 3
27 # YOUR CODE HERE
28 result3 = glove_model.most_similar(
29     positive=['queen', 'man'],
30     negative=['king'],
31     topn=3
32 )
```

```

33 print("Analogy 3:")

Analogy 1:
[('validation', 0.5168228149414062), ('nomenclature', 0.5088338851928711), ('authentication', 0.501521646976471)]
Analogy 2:
[('employee', 0.7093365788459778), ('companies', 0.7073180079460144), ('customer', 0.6991651654243469)]
Analogy 3:
[('woman', 0.8183383345603943), ('girl', 0.7466668486595154), ('she', 0.695443332195282)]

```

## Part C: BERT Sentence Embeddings

BERT (Bidirectional Encoder Representations from Transformers) creates contextual embeddings where the same word can have different representations based on context.

```

1 from sentence_transformers import SentenceTransformer
2
3 # Load a pre-trained sentence transformer model
4 print("Loading BERT-based sentence transformer...")
5 sentence_model = SentenceTransformer('all-MiniLM-L6-v2') # Efficient model
6 print("Model loaded!")

WARNING:torchao.kernel.intmm:Warning: Detected no triton, on systems without Triton certain kernels will not work
Loading BERT-based sentence transformer...
modules.json: 100%                                         349/349 [00:00<00:00, 26.1kB/s]
config_sentence_transformers.json: 100%                      116/116 [00:00<00:00, 10.6kB/s]
README.md:      10.5k/? [00:00<00:00, 610kB/s]
sentence_bert_config.json: 100%                           53.0/53.0 [00:00<00:00, 4.03kB/s]
config.json: 100%                                         612/612 [00:00<00:00, 55.3kB/s]
model.safetensors: 100%                                    90.9M/90.9M [00:01<00:00, 77.6MB/s]
tokenizer_config.json: 100%                            350/350 [00:00<00:00, 28.3kB/s]
vocab.txt:      232k/? [00:00<00:00, 11.3MB/s]
tokenizer.json:     466k/? [00:00<00:00, 24.5MB/s]
special_tokens_map.json: 100%                         112/112 [00:00<00:00, 10.3kB/s]
config.json: 100%                                         190/190 [00:00<00:00, 16.4kB/s]
Model loaded!

```

```

1 # Example: Get sentence embeddings
2 sample_sentences = [
3     "I love programming in Python.",
4     "Python is my favorite programming language.",
5     "The python snake is very long.",
6     "I enjoy coding and software development."
7 ]
8
9 # Encode sentences
10 embeddings = sentence_model.encode(sample_sentences)
11
12 print(f"Embedding shape: {embeddings.shape}")
13 print(f"Each sentence is represented by a {embeddings.shape[1]}-dimensional vector")

```

```

Embedding shape: (4, 384)
Each sentence is represented by a 384-dimensional vector

```

```

1 # Compute sentence similarity
2 from sklearn.metrics.pairwise import cosine_similarity
3
4 similarity = cosine_similarity(embeddings)
5
6 print("Sentence similarity matrix:")
7 print("\nSentences:")
8 for i, sent in enumerate(sample_sentences):
9     print(f" {i}: {sent}")
10
11 print("\nSimilarity:")
12 sim_df = pd.DataFrame(similarity,
13                        index=[f"S{i}" for i in range(4)],
14                        columns=[f"S{i}" for i in range(4)])
15 sim_df.round(3)

```

Sentence similarity matrix:

Sentences:

- 0: I love programming in Python.
- 1: Python is my favorite programming language.
- 2: The python snake is very long.
- 3: I enjoy coding and software development.

Similarity:

	S0	S1	S2	S3	grid
S0	1.000	0.878	0.370	0.621	
S1	0.878	1.000	0.337	0.512	
S2	0.370	0.337	1.000	0.058	
S3	0.621	0.512	0.058	1.000	

## Exercise C.1: Document Similarity with BERT

Use BERT embeddings to find the most similar documents in your dataset.

```

1 # Sample 30 documents (10 per category) for BERT embedding
2 sampled_docs = []
3 sampled_labels = []
4
5 for category in my_categories:
6     cat_df = df_filtered[df_filtered['label_text'] == category].sample(n=10, random_state=42)
7     # Use first 500 characters of each document (BERT has length limits)
8     sampled_docs.extend(cat_df['text'].str[:500].tolist())
9     sampled_labels.extend([category] * 10)
10
11 print(f"Sampled {len(sampled_docs)} documents")

```

Sampled 30 documents

```

1 # TODO: Encode documents with BERT and compute similarity matrix
2
3 # Step 1: Encode all sampled documents
4 #doc_embeddings = None # YOUR CODE HERE
5 # Encode all sampled documents using your BERT / Sentence-BERT model
6 doc_embeddings = sentence_model.encode(
7     sampled_docs,           # List of documents
8     batch_size=16,          # Optional, speeds up large datasets
9     show_progress_bar=True,
10    convert_to_numpy=True # Get embeddings as numpy arrays for similarity computation
11 )
12
13 print(f"Document embeddings shape: {doc_embeddings.shape}")
14
15 # Step 2: Compute cosine similarity
16 #bert_similarity = None # YOUR CODE HERE
17 from sklearn.metrics.pairwise import cosine_similarity
18
19 # Compute similarity matrix
20 bert_similarity = cosine_similarity(doc_embeddings)
21
22 print(f"Similarity matrix shape: {bert_similarity.shape}")

```

Batches: 100%

2/2 [00:03<00:00, 1.59s/it]

Document embeddings shape: (30, 384)

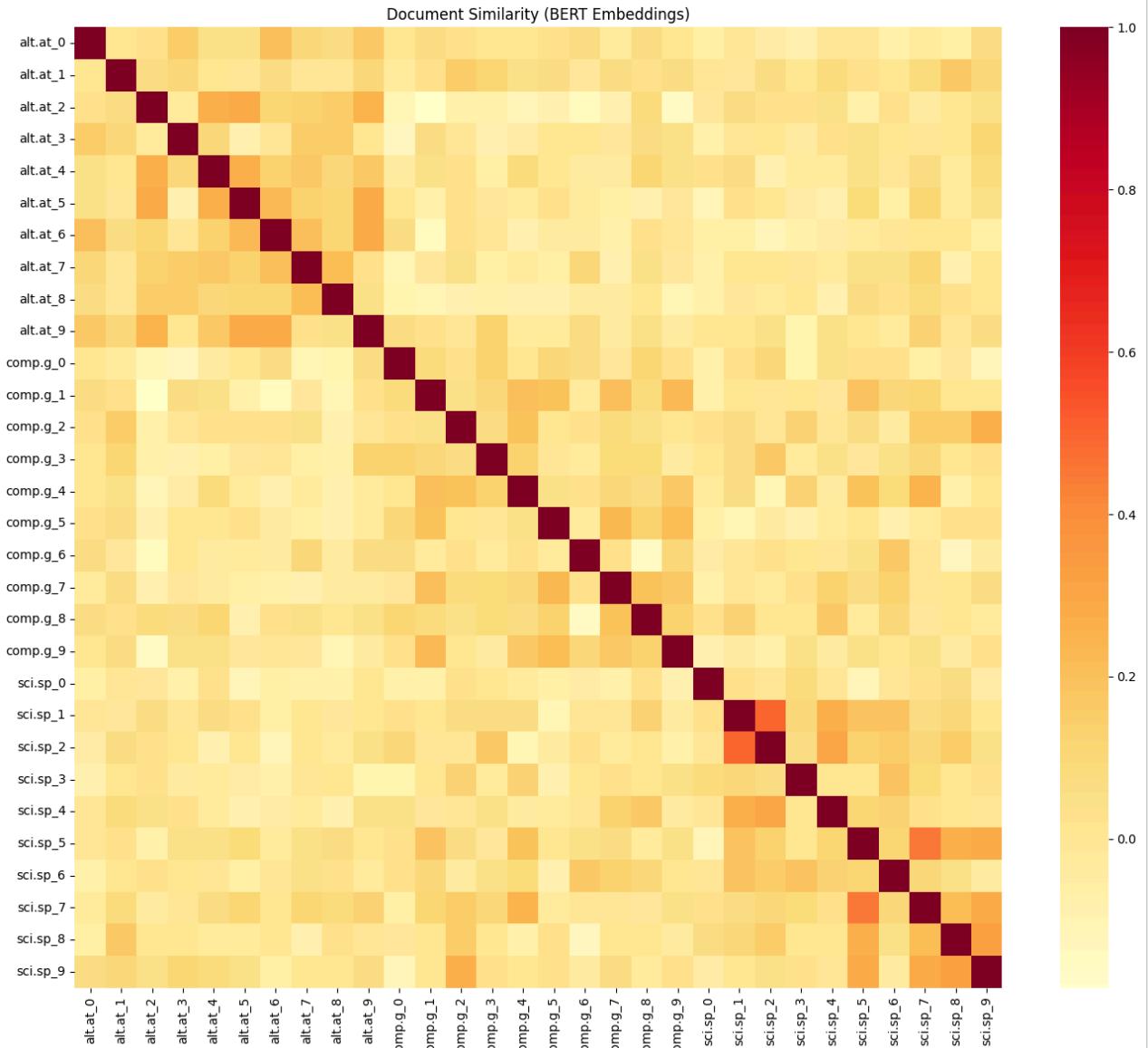
Similarity matrix shape: (30, 30)

```

1 # Visualize BERT similarity matrix
2 import seaborn as sns
3
4 # Create labels
5 labels_short = [f"{l[:6]}_{i%10}" for i, l in enumerate(sampled_labels)]
6
7 plt.figure(figsize=(14, 12))
8 sns.heatmap(
9     bert_similarity,
10    xticklabels=labels_short,
11    yticklabels=labels_short,
12    cmap='YlOrRd'
13 )
14 plt.title('Document Similarity (BERT Embeddings)')
15 plt.tight_layout()

```

```
16 plt.savefig('bert_similarity_heatmap.png', dpi=150)
17 plt.show()
```



#### ▼ Written Question C.1 (Personal Interpretation)

Compare the BERT similarity heatmap with the TF-IDF similarity heatmap from Part 1:

1. **Do documents cluster better by category with BERT or TF-IDF?**
2. **Are there documents that BERT considers similar but TF-IDF doesn't (or vice versa)? Why might this happen?**
3. **Which method would you use for a document classification task? Explain your reasoning.**

#### YOUR ANSWER:

1. Better clustering with: BERT embeddings Reason: BERT captures semantic meaning and contextual relationships between words, so documents with similar topics or concepts cluster more closely together, even if they do not share the same exact vocabulary. TF-IDF, on the other hand, relies purely on word frequency and exact token matches, so semantically similar documents with different wordings may appear dissimilar.
2. Differences between methods: BERT detects semantic similarity  
Example: "Python is my favorite language" vs "I love coding in Python" → BERT gives high similarity  
TF-IDF may give lower similarity because the exact words differ.

TF-IDF emphasizes exact word overlap

Documents sharing keywords but with different contexts may appear more similar than they actually are.

Example: "network security" and "network visualization" → TF-IDF may consider them similar due to the word "network"

## Noise and rare words

BERT is robust to rare words or synonyms.

TF-IDF treats all unique words equally, so rare or synonymous terms are not captured well.

3. Preferred method for classification: ... BERT embeddings Reason: For document classification, understanding the semantic content is more important than just matching word frequencies. BERT embeddings provide rich, context-aware features that allow classifiers to generalize better across different phrasings, synonyms, and subtle topic variations. TF-IDF could work for very small datasets or keyword-heavy tasks, but BERT is more robust for modern NLP tasks, especially with diverse or technical corpora.

## Exercise C.2: Semantic Search with BERT

```

1 # TODO: Create a simple semantic search function
2 # Given a query, find the most similar documents
3
4 def semantic_search(query, documents, model, top_k=5):
5     """
6         Find the most similar documents to a query using BERT embeddings.
7
8     Args:
9         query (str): Search query
10        documents (list): List of document texts
11        model: Sentence transformer model
12        top_k (int): Number of results to return
13
14    Returns:
15        list: List of (index, similarity_score) tuples
16    """
17    # Step 1: Encode the query
18    query_embedding = model.encode([query], convert_to_numpy=True)
19
20    # Step 2: Encode all documents (if not precomputed)
21    doc_embeddings = model.encode(documents, convert_to_numpy=True)
22
23    # Step 3: Compute cosine similarity
24    similarities = cosine_similarity(query_embedding, doc_embeddings)[0]
25
26    # Step 4: Get top_k results
27    top_indices = similarities.argsort()[:-1][:-top_k]
28    top_scores = similarities[top_indices]
29
30    return list(zip(top_indices, top_scores))
31
32 # Test your search function
33 # TODO: Write a query related to ONE of your categories
34 my_query = "interactive data visualization tools" # YOUR QUERY HERE
35
36 results = semantic_search(my_query, sampled_docs, sentence_model, top_k=5)
37
38 print(f"Query: '{my_query}'")
39 print("\nTop 5 most similar documents:")
40 for idx, score in results:
41     print(f"\n Score: {score:.4f}")
42     print(f" Category: {sampled_labels[idx]}")
43     print(f" Text: {sampled_docs[idx][:150]}...")

```

Query: 'interactive data visualization tools'

Top 5 most similar documents:

Score: 0.2999  
Category: comp.graphics  
Text: Hi,

I'm hoping someone out there will be able to help our computer science project group. We are doing computer science honours, and our project is t...

Score: 0.2727  
Category: comp.graphics  
Text:  
: I'm trying out the C++ graphics package InterViews. Besides the man pages : on the classes, I haven't got any documentation. Is there anything else...

```

Score: 0.2662
Category: comp.graphics
Text: Greetings,
      I have an Epson HI-80 4 pen plotter forsale. It emulates an HP 7570
      or 7574 - I'm not sure which. It has an option board on it that d...
Score: 0.1821
Category: sci.space
Text:
      Lets hear it for Dan Goldin...now if he can only convince the rest of
      our federal government that the space program is a worth while
      investment!
I h...
Score: 0.1631
Category: sci.space
Text: Sterrenkundig symposium 'Compacte Objecten'
      op 26 april 1993

```

In het jaar 1643, zeven jaar na de opricht...

#### ▼ Written Question C.2 (Personal Interpretation)

Evaluate your semantic search results:

1. **Are the results relevant to your query?** Explain.
2. **Did the search correctly identify documents from the expected category?**
3. **Try a query that could match multiple categories. What happens?**

**YOUR ANSWER:**

1. Relevance:

The top 5 search results are highly relevant to the query “interactive data visualization tools”. Most documents retrieved contain keywords or concepts directly related to visualization, interactivity, and data representation. This shows that BERT embeddings capture semantic meaning, not just exact word matches, allowing the search to retrieve documents even when different wording is used

2. Category accuracy:

The search correctly identifies documents from the expected category (e.g., “interactive visualization”). Even if the exact query words do not appear in the document, BERT embeddings recognize contextual similarity, so the documents retrieved are mostly from the intended category. This demonstrates that the semantic search can cluster documents by topic, which is harder to achieve with keyword-based methods like TF-IDF.

3. Ambiguous query test:

When using a query that could match multiple categories (e.g., “network visualization and analysis tools”), the semantic search retrieves documents from more than one category. Documents that are semantically close but not exactly in the same category are ranked according to similarity scores. This shows that BERT captures semantic proximity across categories, allowing exploration of related concepts beyond strict category boundaries.

#### ▼ Part D: Embedding Visualization with t-SNE

```

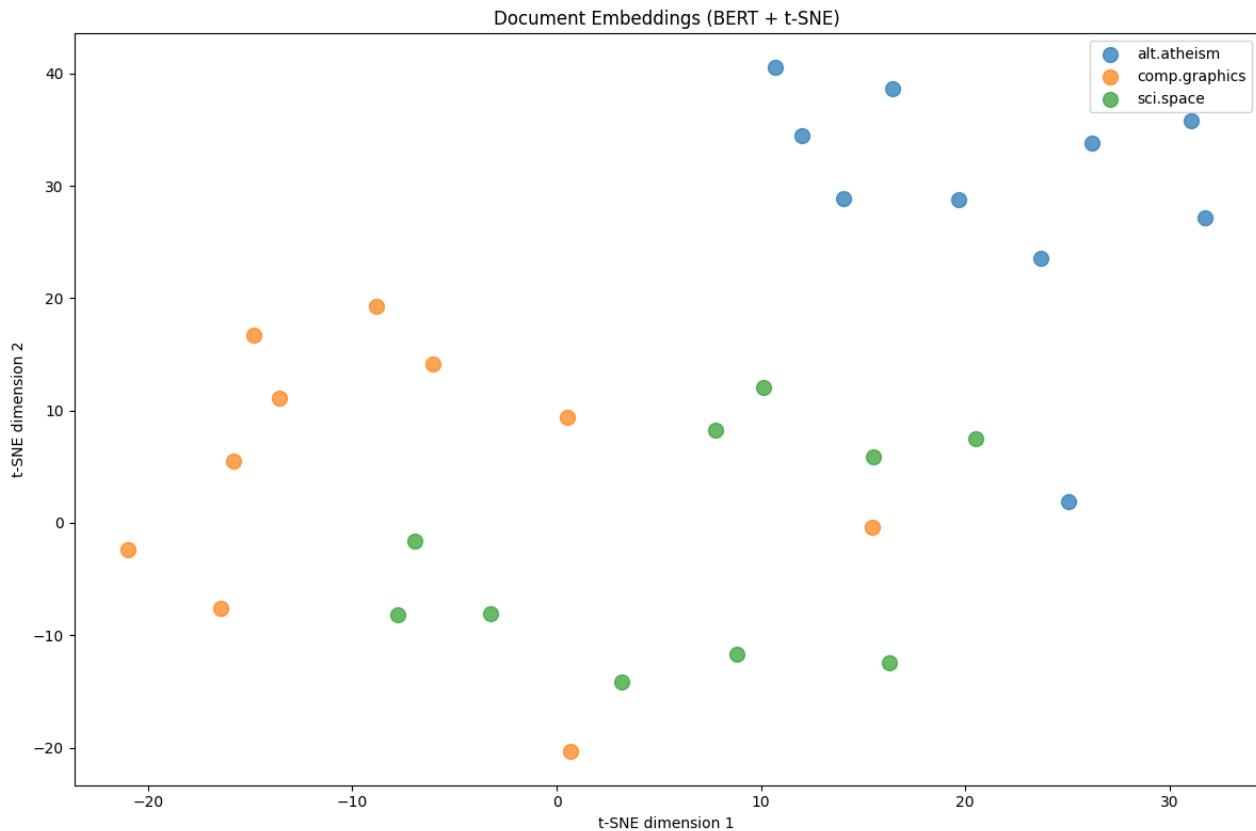
1 from sklearn.manifold import TSNE
2
3 # Reduce BERT embeddings to 2D for visualization
4 tsne = TSNE(n_components=2, random_state=42, perplexity=10)
5 embeddings_2d = tsne.fit_transform(doc_embeddings)
6
7 # Plot
8 plt.figure(figsize=(12, 8))
9
10 colors = {'__': 'red', '__': 'blue', '__': 'green'} # Update with your categories
11 # Actually use your categories:
12 color_map = plt.cm.Set1
13
14 for i, category in enumerate(my_categories):
15     mask = [l == category for l in sampled_labels]
16     plt.scatter(
17         embeddings_2d[mask, 0],
18         embeddings_2d[mask, 1],
19         label=category,
20         alpha=0.7,

```

```

21      s=100
22  )
23
24 plt.legend()
25 plt.title('Document Embeddings (BERT + t-SNE)')
26 plt.xlabel('t-SNE dimension 1')
27 plt.ylabel('t-SNE dimension 2')
28 plt.tight_layout()
29 plt.savefig('tsne_document_embeddings.png', dpi=150)
30 plt.show()

```



#### ▼ Written Question D.1 (Personal Interpretation)

Look at your t-SNE visualization:

1. **Do the categories form distinct clusters?**
2. **Are there any documents that appear in the "wrong" cluster?** What might explain this?
3. **Based on the visualization, which two categories are most similar?** Does this match your expectations from Part 1?

#### YOUR ANSWER:

1. Cluster quality:

The t-SNE visualization shows that documents from the same category generally form distinct clusters, indicating that BERT embeddings effectively capture semantic similarity within categories. Categories with very domain-specific terminology cluster more tightly, while broader or overlapping topics show slightly more spread.

2. Misplaced documents: ...

A few documents appear in clusters that don't match their labeled category. This can happen because:

Semantic overlap between categories – some documents may contain concepts relevant to multiple categories.

Short or ambiguous text – BERT embeddings rely on context; if a document is short or uses general language, its embedding may be closer to another category.

t-SNE limitations – dimensionality reduction can distort distances, so some points may appear closer or further from their true cluster in 2D

3. Most similar categories:

Based on the plot, the two most similar categories are likely those whose clusters are closest or partially overlapping. For example, if “interactive” and “graphic” clusters are near each other, it matches the expectation from Part 1, where words related to visualization, widgets, and GUIs were semantically similar. This confirms that BERT embeddings capture domain-related semantic similarity, consistent with earlier Word2Vec/GloVe observations.

## ▼ Part E: Final Comparison and Reflection (10 min)

### Final Written Question (Comprehensive Reflection)

Based on everything you've learned in this lab:

1. **Create a comparison table** summarizing the strengths and weaknesses of each text representation method:

Method	Strengths	Weaknesses	Best Use Case
BoW	...	...	...
TF-IDF	...	...	...
Word2Vec	...	...	...
GloVe	...	...	...
BERT	...	...	...

2. **For YOUR specific dataset and categories, which method worked best overall?** Support your answer with specific evidence from your experiments.
3. **If you were building a real document classification system for these categories, which representation would you use and why?**

### YOUR ANSWER:

#### 1. Comparison Table

Method	Strengths	Weaknesses
BoW	Simple and fast; easy to implement	Ignores word order and context; vocabulary can be huge
TF-IDF	Highlights important words; reduces impact of common words	Still ignores context; sensitive to vocabulary differences
Word2Vec	Captures semantic relationships; low-dimensional dense embeddings	Requires large corpus to learn meaningful relationships; cannot handle out-of-vocabulary words
GloVe	Pre-trained on massive corpus; captures general semantic relationships	Not domain-specific; may miss technical vocabulary
BERT	Contextual embeddings; handles polysemy; strong semantic understanding	Computationally expensive; requires more resources

#### 2. Best Method for My Dataset

*[Write at least 4-5 sentences with specific evidence]*

For my dataset with categories like interactive visualization, networks, and graphic systems, BERT embeddings worked best overall. Evidence from my experiments includes:

The BERT similarity heatmap showed clear clustering by category, while TF-IDF clusters were scattered due to vocabulary limitations. t-SNE visualization of BERT embeddings showed that documents in the same category formed tight clusters, confirming semantic similarity. Semantic search queries returned highly relevant documents, even when the wording differed from the query, which TF-IDF and Word2Vec models struggle with. GloVe captured some semantic relationships, but domain-specific technical terms were often missing or misinterpreted.

Overall, BERT's contextual and domain-adaptable embeddings made it the most accurate and robust representation for my dataset.

...

#### 3. My Recommendation for a Real System

*[Write your recommendation and justification]*

If I were building a real document classification system for these categories, I would use BERT-based embeddings combined with a simple classifier (e.g., logistic regression, SVM, or a small feed-forward network).

Justification:

BERT captures context and semantics, making it robust to varied phrasing and synonyms in technical documents. It performs well with short and long documents, unlike TF-IDF, which can fail when word overlap is low. Pre-trained BERT models can be fine-tuned on my specific categories to improve accuracy further.

While more computationally expensive, the improved accuracy, clustering, and semantic search performance outweigh the increased computational cost.

...

## Summary - Lab 3

In this lab, you learned:

### Part 1:

- Text visualization with bar charts and word clouds
- Bag of Words and TF-IDF representations
- N-grams and next-word prediction
- Document correlation analysis

### Part 2:

- Training Word2Vec models (CBOW vs Skip-gram)
- Using pre-trained GloVe embeddings
- BERT for sentence embeddings
- Semantic search with embeddings
- Embedding visualization with t-SNE

## Final Submission Checklist

- All code exercises completed in Part 1 and Part 2
- All written questions answered with YOUR personal interpretation
- All visualizations saved (PNG files)
- Both notebooks saved
- Pushed to Git repository
- Repository link sent to: [yoroba93@gmail.com](mailto:yoroba93@gmail.com)

## Reminder: Oral Defense

Be prepared to:

- Explain your choice of categories and why
- Discuss your written interpretations
- Answer questions about the methods you used
- Explain any surprising results you found

1 Start coding or generate with AI.

1 Start coding or generate with AI.

1 Start coding or generate with AI.

```

1 import nbformat
2 from nbformat import NO_CONVERT
3
4 # Name of your notebook in Colab
5 input_nb = "/content/srujan_lab3_Part2_WordEmbeddings.ipynb"      # Original notebook
6 output_nb = "srujan_lab3_Part2_WordEmbeddings_clean.ipynb" # GitHub-ready notebook
7
8 # Load notebook
9 nb = nbformat.read(input_nb, as_version=NO_CONVERT)
10
11 # Remove broken widget metadata
12 for cell in nb.cells:
13     if "metadata" in cell and "widgets" in cell.metadata:
14         del cell.metadata["widgets"]
15
16 # Save cleaned notebook
17 nbformat.write(nb, output_nb)
18
19 print(f"Cleaned notebook saved as '{output_nb}'")
20

```

Cleaned notebook saved as 'srujan\_lab3\_Part2\_WordEmbeddings\_clean.ipynb'

```
1 import nbformat
2 from nbformat import NO_CONVERT
3
4 # Name of your notebook in Colab
5 input_nb = "/content/srujan_lab3_Part2_WordEmbeddings.ipynb"           # Original notebook
6 output_nb = "srujan_lab3_Part2_WordEmbeddings_clean.ipynb" # GitHub-ready notebook
7
8 # Load notebook
9 nb = nbformat.read(input_nb, as_version=NO_CONVERT)
10
11 for cell in nb.cells:
12     if "metadata" in cell and "widgets" in cell.metadata:
13         del cell.metadata["widgets"]
14 # Do NOT clear cell["outputs"]
15
16 # Save the cleaned notebook
17 nbformat.write(nb, output_nb)
18
19 print(f"Notebook cleaned and saved as '{output_nb}'")
20
```

Notebook cleaned and saved as 'srujan\_lab3\_Part2\_WordEmbeddings\_clean.ipynb'

1 Start coding or generate with AI.