

⌄ ITAI 2373 Module 04: Text Representation Homework Lab

From Words to Numbers:


eJoy
AI ASSISTANT

 Deep Dive

 Summarise

Student Name: (Williane Yarro)

🎯 Welcome to Your Text Representation Adventure!

You'll discover how computers transform human language into mathematical representations that machines can understand and process.

This journey will take you from basic word counting to sophisticated embedding techniques used in modern AI systems.

📅 5-Parts Learning Journey

- **Part 1-2:** Foundations & Sparse Representations (BOW, Preprocessing)
- **Part 3:** TF-IDF & N-grams (Weighted Representations)
- **Part 4:** Dense Representations (Word Embeddings)
- **Part 5:** Integration & Real-World Applications

🎓 Learning Outcomes

By completing this lab, you will be able to:

- Explain why text must be converted to numbers for machine learning
- Implement Bag of Words and TF-IDF representations from scratch
- Apply N-gram analysis to capture word sequences
- Explore word embeddings and their semantic properties
- Compare different text representation methods
- Build a simple text classification system

📝 Submission Guidelines

- Complete all exercises and answer all questions
- Run all code cells and ensure outputs are visible
- Provide thoughtful responses to reflection questions

🏆 Assessment Rubric

- **Technical Implementation (60%):** Correct code, proper library usage, handling edge cases
- **Conceptual Understanding (25%):** Clear explanations, result interpretation
- **Analysis & Reflection (15%):** Critical thinking, real-world connections

Let's begin your journey into the fascinating world of text representation! 

⌄ Environment Setup

First, let's install and import all the libraries we'll need for our text representation journey. Run the cells below to set up your environment.

```
# Install required libraries (run this cell first in Google Colab)
# Uninstall numpy and gensim first to resolve potential version conflicts
!pip uninstall -y numpy gensim
!pip install nltk gensim scikit-learn matplotlib seaborn wordcloud numpy
!python -m nltk.downloader punkt stopwords movie_reviews punkt_tab
```

```

→ Found existing installation: numpy 1.26.4
Uninstalling numpy-1.26.4:
  Successfully uninstalled numpy-1.26.4
Found existing installation: gensim 4.3.3
Uninstalling gensim-4.3.3:
  Successfully uninstalled gensim-4.3.3
Requirement already satisfied: nltk in /usr/local/lib/python3.11/dist-packages (3.9.1)
Collecting gensim
  Using cached gensim-4.3.3-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (6.1 kB)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.11/dist-packages (1.6.1)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.11/dist-packages (3.10.0)
Requirement already satisfied: seaborn in /usr/local/lib/python3.11/dist-packages (0.13.2)
Requirement already satisfied: wordcloud in /usr/local/lib/python3.11/dist-packages (1.9.4)
Collecting numpy
  Downloading numpy-2.3.1-cp311-cp311-manylinux_2_28_x86_64.whl.metadata (62 kB)
  62.1/62.1 kB 5.1 MB/s eta 0:00:00
Requirement already satisfied: click in /usr/local/lib/python3.11/dist-packages (from nltk) (8.2.1)
Requirement already satisfied: joblib in /usr/local/lib/python3.11/dist-packages (from nltk) (1.5.1)
Requirement already satisfied: regex>=2021.8.3 in /usr/local/lib/python3.11/dist-packages (from nltk) (2024.11.6)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (from nltk) (4.67.1)
  Using cached numpy-1.26.4-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (61 kB)
Requirement already satisfied: scipy<1.14.0,>=1.7.0 in /usr/local/lib/python3.11/dist-packages (from gensim) (1.13.1)
Requirement already satisfied: smart-open>=1.8.1 in /usr/local/lib/python3.11/dist-packages (from gensim) (7.1.0)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn) (3.6.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (1.3.2)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (4.58.4)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (1.4.8)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (24.2)
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (11.2.1)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (2.9.0.post0)
Requirement already satisfied: pandas>=1.2 in /usr/local/lib/python3.11/dist-packages (from seaborn) (2.2.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas>=1.2->seaborn) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas>=1.2->seaborn) (2025.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.7->matplotlib) (1.17.0)
Requirement already satisfied: wrapt in /usr/local/lib/python3.11/dist-packages (from smart-open>=1.8.1->gensim) (1.17.2)
Using cached gensim-4.3.3-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (26.7 MB)
Using cached numpy-1.26.4-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (18.3 MB)
Installing collected packages: numpy, gensim
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the source of the PIP-100-CONSISTENCY-BREAKING issue. You can report a bug at https://github.com/pypa/pip/issues/100
  thinc 8.3.6 requires numpy<3.0.0,>=2.0.0, but you have numpy 1.26.4 which is incompatible.
  tsfresh 0.21.0 requires scipy>=1.14.0; python_version >= "3.10", but you have scipy 1.13.1 which is incompatible.
Successfully installed gensim-4.3.3 numpy-1.26.4
WARNING: The following packages were previously imported in this runtime:
[numpy]
You must restart the runtime in order to use newly installed versions.

RESTART SESSION

```

<frozen runpy>:128: RuntimeWarning: 'nltk.download' found in sys.modules after import of package 'nltk', but prior to execution of 'nltk.download'. This behaviour is the source of the PIP-100-CONSISTENCY-BREAKING issue. You can report a bug at https://github.com/pypa/pip/issues/100

```

[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package movie_reviews to /root/nltk_data...
[nltk_data]   Unzipping corpora/movie_reviews.zip.
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.

```

```

# Import all necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from collections import Counter, defaultdict
import re
import math
from itertools import combinations

# NLTK for text processing
import nltk
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk.corpus import stopwords, movie_reviews
from nltk.stem import PorterStemmer

# Scikit-learn for machine learning
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report, accuracy_score

```

```
# Gensim for word embeddings
import gensim.downloader as api

# Set up plotting
plt.style.use('default')
sns.set_palette("husl")

print("✅ All libraries imported successfully!")
print("🎉 You're ready to start your text representation journey!")

→ ✅ All libraries imported successfully!
🎉 You're ready to start your text representation journey!
```



Deep Dive

Summarise

Part 1-2: Foundations & Sparse Representations

💡 Why Do We Need to Convert Text to Numbers?

Imagine you're trying to teach a computer to understand the difference between "I love this movie!" and "This movie is terrible." How would you explain the concept of sentiment to a machine that only understands mathematics?

This is the fundamental challenge in Natural Language Processing (NLP). Computers are excellent at processing numbers, but human language is complex, contextual, and inherently non-numerical. We need a bridge between words and numbers.

🎯 Part 1-2 Goals:

- Understand why text-to-number conversion is necessary
- Master text preprocessing and tokenization
- Implement Bag of Words (BOW) from scratch
- Explore the limitations of sparse representations

Our Sample Dataset

Let's start with a small collection of movie reviews to make our learning concrete and relatable.

```
# Our sample movie reviews for learning
sample_reviews = [
    "This movie is absolutely fantastic! The acting is superb and the plot is engaging.",
    "I found this film quite boring. The story dragged on and the characters were flat.",
    "Amazing cinematography and brilliant performances. A must-watch movie!",
    "The plot was confusing and the dialogue felt forced. Not recommended.",
    "Great movie with excellent acting. The story kept me engaged throughout."
]

# Let's also create labels for sentiment (positive=1, negative=0)
sample_labels = [1, 0, 1, 0, 1] # 1 = positive, 0 = negative

print("📊 Sample Movie Reviews:")
for i, (review, label) in enumerate(zip(sample_reviews, sample_labels)):
    sentiment = "😊 Positive" if label == 1 else "😢 Negative"
    print(f"\n{i+1}. [{sentiment}] {review}")

print(f"\n📊 Dataset Summary: {len(sample_reviews)} reviews ({sum(sample_labels)}) positive, {len(sample_labels)-sum(sample_labels)} negative")
```

→ 📊 Sample Movie Reviews:

- [😊 Positive] This movie is absolutely fantastic! The acting is superb and the plot is engaging.
- [😢 Negative] I found this film quite boring. The story dragged on and the characters were flat.
- [😊 Positive] Amazing cinematography and brilliant performances. A must-watch movie!
- [😢 Negative] The plot was confusing and the dialogue felt forced. Not recommended.
- [😊 Positive] Great movie with excellent acting. The story kept me engaged throughout.

📊 Dataset Summary: 5 reviews (3 positive, 2 negative)

✍️ Text Preprocessing: Cleaning Our Data

Before we can convert text to numbers, we need to clean and standardize our text. Think of this as preparing ingredients before cooking - we need everything in the right format!

Common Preprocessing Steps:

1. **Lowercasing:** "Movie" and "movie" should be treated the same
2. **Removing punctuation:** "great!" becomes "great"
3. **Tokenization:** Breaking text into individual words
4. **Removing stop words:** Common words like "the", "and", "is"
5. **Stemming:** "running", "runs", "ran" → "run"



```
#Let's see preprocessing in action with one example
example_text = sample_reviews[0]
print(f" 1 Original text: {example_text}")

#Step 1: Lowercase
step1 = example_text.lower()
print(f"\n 2 After lowercasing: {step1}")

#Step 2: Remove punctuation
step2 = re.sub(r'[^w\s]', '', step1)
print(f" 3 After removing punctuation: {step2}")

#Step 3: Tokenization
tokens = word_tokenize(step2)
print(f" 4 After tokenization: {tokens}")

#Step 4: Remove stop words
stop_words = set(stopwords.words('english'))
filtered_tokens = [word for word in tokens if word not in stop_words]
print(f" 5 After removing stop words: {filtered_tokens}")

#Step 5: Stemming
stemmer = PorterStemmer()
stemmed_tokens = [stemmer.stem(word) for word in filtered_tokens]
print(f" 6 After stemming: {stemmed_tokens}")

print(f"\n 7 Length reduction: {len(example_text.split())} → {len(stemmed_tokens)} words")
```

Original text: This movie is absolutely fantastic! The acting is superb and the plot is engaging.

- 1 After lowercasing: this movie is absolutely fantastic! the acting is superb and the plot is engaging.
- 2 After removing punctuation: this movie is absolutely fantastic the acting is superb and the plot is engaging
- 3 After tokenization: ['this', 'movie', 'is', 'absolutely', 'fantastic', 'the', 'acting', 'is', 'superb', 'and', 'the', 'plot', 'is', 'engaging']
- 4 After removing stop words: ['movie', 'absolutely', 'fantastic', 'acting', 'superb', 'plot', 'engaging']
- 5 After stemming: ['movi', 'absolut', 'fantast', 'act', 'superb', 'plot', 'engag']

Length reduction: 14 → 7 words

Exercise 1: Build Your Own Preprocessor

Now it's your turn! Complete the function below to preprocess text. This will be your foundation for all future exercises.

```
def preprocess_text(text, remove_stopwords=True, apply_stemming=True):
    """
    Preprocess a text string by cleaning and tokenizing it.

    Args:
        text (str): Input text to preprocess
        remove_stopwords (bool): Whether to remove stop words
        apply_stemming (bool): Whether to apply stemming

    Returns:
        list: List of preprocessed tokens
    """
    # TODO: Implement the preprocessing steps
    # Hint: Follow the same steps we demonstrated above

    # Step 1: Convert to lowercase
    text = text.lower()

    # Step 2: Remove punctuation (keep only letters, numbers, and spaces)
```



```
text = text = re.sub(r'[^a-zA-Z0-9\s]', '', text)

# Step 3: Tokenize
tokens = tokens = word_tokenize(text)

# Step 4: Remove stop words (if requested)
if remove_stopwords:
    stop_words = set(stopwords.words('english'))
    tokens = tokens = [word for word in tokens if word not in stop_words]

# Step 5: Apply stemming (if requested)
if apply_stemming:
    stemmer = PorterStemmer()
    tokens = tokens = [stemmer.stem(word) for word in tokens]

return tokens

# Test your function
test_text = "The movies are absolutely AMAZING! I love watching them."
result = preprocess_text(test_text)
print(f"Input: {test_text}")
print(f"Output: {result}")

# Expected output should be something like: ['movi', 'absolut', 'amaz', 'love', 'watch']
```

enjoy
AI ASSISTANT

Deep Dive

Summarise



```

import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize

# Download required resources if not already downloaded
nltk.download('punkt')
nltk.download('stopwords')

def preprocess_text(text, remove_stopwords=True, apply_stemming=True):
    """
    Preprocess a text string by cleaning and tokenizing it.

    Args:
        text (str): Input text to preprocess
        remove_stopwords (bool): Whether to remove stop words
        apply_stemming (bool): Whether to apply stemming

    Returns:
        list: List of preprocessed tokens
    """
    # Step 1: Convert to lowercase
    text = text.lower()

    # Step 2: Remove punctuation (keep only letters, numbers, and spaces)
    text = re.sub(r'[^a-z0-9\s]', '', text)

    # Step 3: Tokenize
    tokens = word_tokenize(text)

    # Step 4: Remove stop words (if requested)
    if remove_stopwords:
        stop_words = set(stopwords.words('english'))
        tokens = [word for word in tokens if word not in stop_words]

    # Step 5: Apply stemming (if requested)
    if apply_stemming:
        stemmer = PorterStemmer()
        tokens = [stemmer.stem(word) for word in tokens]

    return tokens

# Test your function
#test_text = "The movies are absolutely AMAZING! I love watching them."
#zare = preprocess_text(test_text)
#print(f"Input: {test_text}")
#print(f"Output: {zare}")

```

 **eJoy**
AI ASSISTANT

 Deep Dive

 Summarise

```

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

```

 **Solution Check:** Run the cell below to see the expected solution and compare with your implementation.

```

# Solution for Exercise 1
def preprocess_text_solution(text, remove_stopwords=True, apply_stemming=True):
    # Step 1: Convert to lowercase
    text = text.lower()

    # Step 2: Remove punctuation
    text = re.sub(r'[^w\s]', '', text)

    # Step 3: Tokenize
    tokens = word_tokenize(text)

    # Step 4: Remove stop words
    if remove_stopwords:
        stop_words = set(stopwords.words('english'))
        tokens = [word for word in tokens if word not in stop_words]

    # Step 5: Apply stemming
    if apply_stemming:
        stemmer = PorterStemmer()
        tokens = [stemmer.stem(word) for word in tokens]

```

```
tokens = [stemmer.stem(word) for word in tokens]
```

```
return tokens
```

```
# Test the solution
test_result = preprocess_text_solution(test_text)
print(f"Expected output: {test_result}")
print("\n✓ If your output matches this, great job! If not, review the steps above.")
```

→ Expected output: ['movi', 'absolut', 'amaz', 'love', 'watch']

If your output matches this, great job! If not, review the steps above.

Now let's preprocess all our sample reviews:

```
# Preprocess all sample reviews
preprocessed_reviews = [preprocess_text(review) for review in sample_reviews]

print("📄 Preprocessed Reviews:")
for i, (original, processed) in enumerate(zip(sample_reviews, preprocessed_reviews)):
    print(f"\n{i+1}. Original: {original[:50]}...")
    print(f"    Processed: {processed}")
```

→ 📄 Preprocessed Reviews:

1. Original: This movie is absolutely fantastic! The acting is ...
 Processed: ['movi', 'absolut', 'fantast', 'act', 'superb', 'plot', 'engag']
2. Original: I found this film quite boring. The story dragged ...
 Processed: ['found', 'film', 'quit', 'bore', 'stori', 'drag', 'charact', 'flat']
3. Original: Amazing cinematography and brilliant performances....
 Processed: ['amaz', 'cinematographi', 'brilliant', 'perform', 'mustwatch', 'movi']
4. Original: The plot was confusing and the dialogue felt force...
 Processed: ['plot', 'confus', 'dialogu', 'felt', 'forc', 'recommend']
5. Original: Great movie with excellent acting. The story kept ...
 Processed: ['great', 'movi', 'excel', 'act', 'stori', 'kept', 'engag', 'throughout']

✓ 🎒 Bag of Words (BOW): Your First Text Representation

Imagine you have a bag and you throw all the words from a document into it. You lose the order of words, but you can count how many times each word appears. That's exactly what Bag of Words does!

🔍 How BOW Works:

1. Create a vocabulary of all unique words across all documents
2. For each document, count how many times each word appears
3. Represent each document as a vector of word counts

📊 Example:

- Document 1: "I love movies"
- Document 2: "Movies are great"
- Vocabulary: ["I", "love", "movies", "are", "great"]
- Doc 1 vector: [1, 1, 1, 0, 0]
- Doc 2 vector: [0, 0, 1, 1, 1]

```
# Let's build BOW step by step with a simple example
```

```
simple_docs = [
    ["love", "movie"],
    ["movie", "great"],
    ["love", "great", "film"]
]
```

```
print("📄 Simple Documents:")
for i, doc in enumerate(simple_docs):
    print(f"Doc {i+1}: {doc}")
```

```
# Step 1: Build vocabulary
vocabulary = sorted(set(word for doc in simple_docs for word in doc))
print(f"\n📘 Vocabulary: {vocabulary}")
```



enjoy
AI ASSISTANT

Deep Dive

Summarise



```
# Step 2: Create BOW vectors
bow_vectors = []
for doc in simple_docs:
    vector = [doc.count(word) for word in vocabulary]
    bow_vectors.append(vector)

print(f"\n📁 BOW Vectors:")
for i, vector in enumerate(bow_vectors):
    print(f"Doc {i+1}: {vector}")

# Visualize as a matrix
bow_df = pd.DataFrame(bow_vectors, columns=vocabulary, index=[f"Doc {i+1}" for i in range(len(simple_docs))])
print(f"\n📊 BOW Matrix:")
print(bow_df)
```

→ 📄 Simple Documents:
 Doc 1: ['love', 'movie']
 Doc 2: ['movie', 'great']
 Doc 3: ['love', 'great', 'film']

📘 Vocabulary: ['film', 'great', 'love', 'movie']

📁 BOW Vectors:
 Doc 1: [0, 0, 1, 1]
 Doc 2: [0, 1, 0, 1]
 Doc 3: [1, 1, 1, 0]

📊 BOW Matrix:

	film	great	love	movie
Doc 1	0	0	1	1
Doc 2	0	1	0	1
Doc 3	1	1	1	0

⌄ 🎨 Exercise 2: Build BOW from Scratch

Now implement your own BOW function! This will help you understand exactly how the representation works.

```
def build_bow_representation(documents):
    """
    Build Bag of Words representation for a list of documents.

    Args:
        documents (list): List of documents, where each document is a list of tokens

    Returns:
        tuple: (vocabulary, bow_matrix)
            vocabulary (list): Sorted list of unique words
            bow_matrix (list): List of BOW vectors for each document
    """
    # TODO: Build the vocabulary (unique words across all documents)
    vocabulary = sorted(set(word for doc in documents for word in doc))

    # TODO: Create BOW vectors for each document
    bow_matrix = []
    for doc in documents:
        # Create a vector where each element is the count of the corresponding vocabulary word
        vector = [doc.count(word) for word in vocabulary]
        bow_matrix.append(vector)

    return vocabulary, bow_matrix
```

```
# Test your function with our preprocessed reviews
vocab, bow_matrix = build_bow_representation(preprocessed_reviews)
```

```
print(f"📘 Vocabulary size: {len(vocab)}")
print(f"📘 First 10 words: {vocab[:10]}")
print(f"📁 BOW matrix shape: {len(bow_matrix)} documents x {len(vocab)} words")
print(f"📁 First document vector (first 10 elements): {bow_matrix[0][:10]}")
```

→ 📂 Vocabulary size: 29
 📂 First 10 words: ['absolut', 'act', 'amaz', 'bore', 'brilliant', 'charact', 'cinematographi', 'confus', 'dialogu', 'drag']

📘 BOW matrix shape: 5 documents x 29 words
 📁 First document vector (first 10 elements): [1, 1, 0, 0, 0, 0, 0, 0, 0]

enjoy
AI ASSISTANT

Deep Dive

Summarise

Solution Check:

```
# Solution for Exercise 2
def build_bow_representation_solution(documents):
    # Build vocabulary: get all unique words and sort them
    vocabulary = sorted(set(word for doc in documents for word in doc))

    # Create BOW vectors
    bow_matrix = []
    for doc in documents:
        vector = [doc.count(word) for word in vocabulary]
        bow_matrix.append(vector)

    return vocabulary, bow_matrix

# Test the solution
vocab_sol, bow_matrix_sol = build_bow_representation_solution(preprocessed_reviews)
print(f"✓ Solution vocabulary size: {len(vocab_sol)}")
print(f"✓ Solution BOW matrix shape: {len(bow_matrix_sol)} x {len(vocab_sol)}")
```

→ ✓ Solution vocabulary size: 29
 ✓ Solution BOW matrix shape: 5 x 29

✖ Comparing with Scikit-learn's CountVectorizer

Let's see how our implementation compares with the professional library:

```
# Using scikit-learn's CountVectorizer
vectorizer = CountVectorizer(lowercase=True, stop_words='english')

# We need to join our preprocessed tokens back into strings for sklearn
processed_texts = [' '.join(tokens) for tokens in preprocessed_reviews]
sklearn_bow = vectorizer.fit_transform(processed_texts)

print("✖ Scikit-learn CountVectorizer Results:")
print(f"Vocabulary size: {len(vectorizer.vocabulary_)}")
print(f"BOW matrix shape: {sklearn_bow.shape}")
print(f"Matrix type: {type(sklearn_bow)}")

# Convert to dense array for comparison
sklearn_bow_dense = sklearn_bow.toarray()
print(f"\n📊 First document vector (first 10 elements): {sklearn_bow_dense[0][:10]}")

# Show some vocabulary words
feature_names = vectorizer.get_feature_names_out()
print(f"\n📘 First 10 vocabulary words: {feature_names[:10].tolist()}")
```

→ ✖ Scikit-learn CountVectorizer Results:
 Vocabulary size: 27
 BOW matrix shape: (5, 27)
 Matrix type: <class 'scipy.sparse._csr.csr_matrix'>

📊 First document vector (first 10 elements): [1 1 0 0 0 0 0 0 0]

📘 First 10 vocabulary words: ['absolut', 'act', 'amaz', 'bore', 'brilliant', 'charact', 'cinematographi', 'confus', 'dialogu', 'drag']

✖ Visualizing BOW Representations

Let's create some visualizations to better understand our BOW representation:

```
# Create a DataFrame for better visualization
bow_df = pd.DataFrame(
    sklearn_bow_dense,
    columns=feature_names,
    index=[f"Review {i+1}" for i in range(len(sample_reviews))]

)

# 1. Heatmap of BOW representation
plt.figure(figsize=(12, 6))
# Show only words that appear at least once
active_words = bow_df.columns[bow_df.sum() > 0][:20] # Top 20 most frequent words
sns.heatmap(bow_df[active_words], annot=True, cmap='Blues', fmt='d')
plt.title('📊 Bar of Words Heatmap (Top 20 Words)')
```



enjoy
AI ASSISTANT

Deep Dive

Summarise



```
# 1. Word cloud visualization
plt.figure(figsize=(10, 6))
plt.imshow(WordCloud().fit_words(WordCloud().wordcloud_.get_word_frequencies().head(20).reset_index()))
plt.xlabel('Words')
plt.ylabel('Reviews')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

# 2. Word frequency distribution
word_frequencies = bow_df.sum().sort_values(ascending=False)
plt.figure(figsize=(10, 6))
word_frequencies[:15].plot(kind='bar')
plt.title('📊 Top 15 Most Frequent Words')
plt.xlabel('Words')
plt.ylabel('Frequency')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

print(f"📝 Total unique words: {len(feature_names)}")
print(f"📝 Average words per review: {bow_df.sum(axis=1).mean():.1f}")
print(f"📝 Sparsity: {(bow_df == 0).sum().sum() / (bow_df.shape[0] * bow_df.shape[1]) * 100:.1f}%")
```

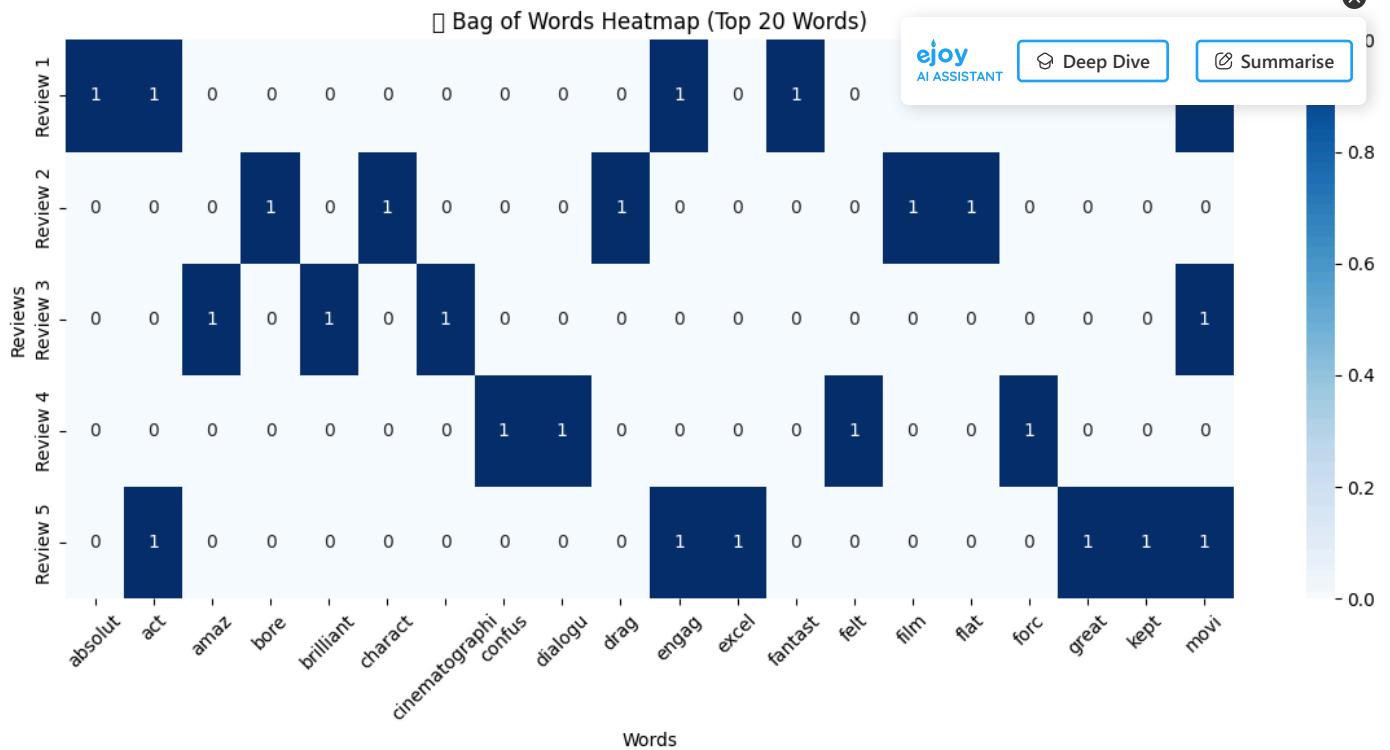
enjoy
AI ASSISTANT

Deep Dive

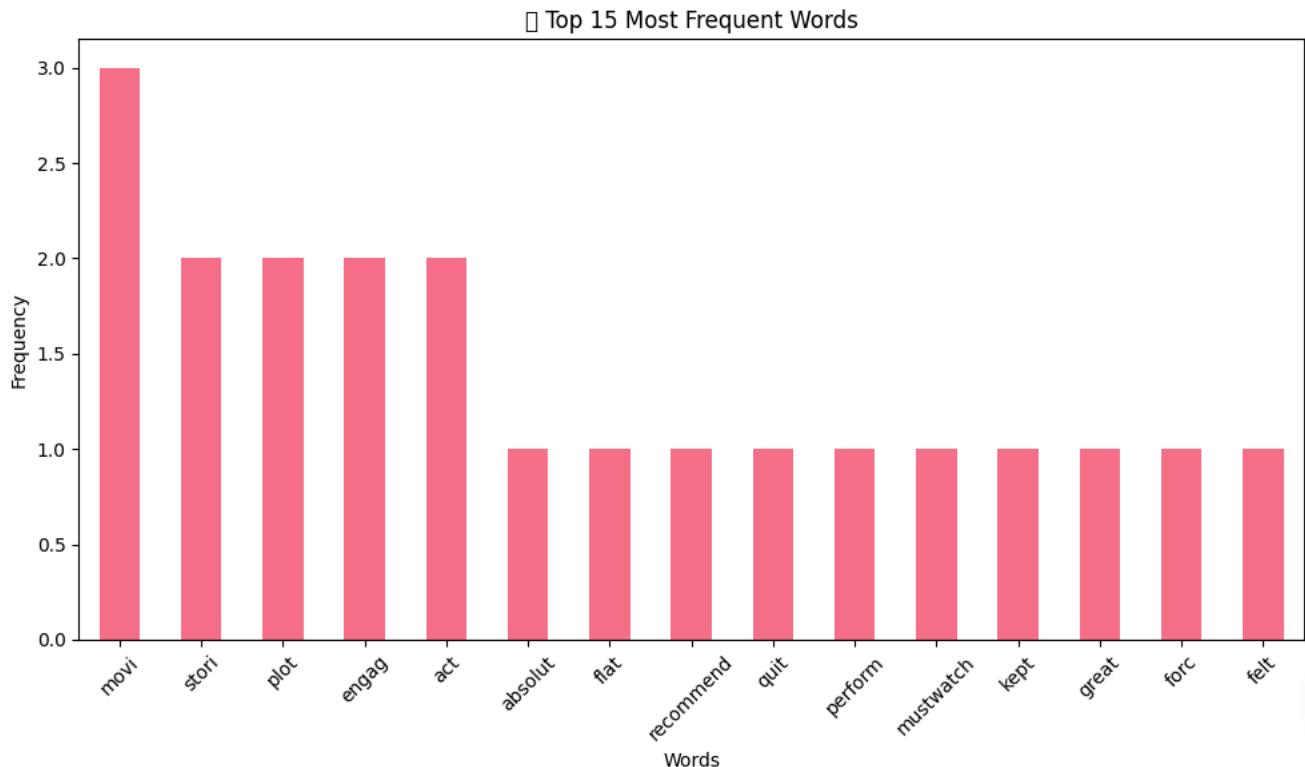
Summarise



```
/tmp/ipython-input-16-199533803.py:17: UserWarning: Glyph 127890 (\N{SCHOOL SATCHEL}) missing from font(s) DejaVu Sans.
  plt.tight_layout()
/usr/local/lib/python3.11/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 127890 (\N{SCHOOL SATCHEL}) missing from font
fig.canvas.print_figure(bytes_io, **kw)
```



```
/tmp/ipython-input-16-199533803.py:28: UserWarning: Glyph 128202 (\N{BAR CHART}) missing from font(s) DejaVu Sans.
  plt.tight_layout()
/usr/local/lib/python3.11/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 128202 (\N{BAR CHART}) missing from font(s)
fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.11/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 128202 (\N{BAR CHART}) missing from font(s)
fig.canvas.print_figure(bytes_io, **kw)
```



- Total unique words: 27
- Average words per review: 6.6
- Sparsity: 75.6%

↙ 🚨 BOW Limitations: What Are We Missing?

BOW is simple and effective, but it has some important limitations. Let's explore them:

```
# Demonstrating BOW limitations
limitation_examples = [
    "The dog ate my homework",
    "The homework ate my dog", # Same words, different meaning!
    "This movie is not bad",
    "This movie is bad" # Negation lost!
]

print("\n⚠️ BOW Limitation Examples:")
for i, text in enumerate(limitation_examples):
    tokens = preprocess_text_solution(text, remove_stopwords=False, apply_stemming=False)
    print(f"\n{i+1}. Text: '{text}'")
    print(f"    Tokens: {tokens}")

# Show that different sentences can have identical BOW representations
vectorizer_demo = CountVectorizer(lowercase=True)
bow_demo = vectorizer_demo.fit_transform(limitation_examples)

print("\n📊 BOW Vectors:")
feature_names_demo = vectorizer_demo.get_feature_names_out()
for i, vector in enumerate(bow_demo.toarray()):
    print(f"Text {i+1}: {vector}")

# Check if any vectors are identical
if np.array_equal(bow_demo.toarray()[0], bow_demo.toarray()[1]):
    print("\n⚠️ Texts 1 and 2 have IDENTICAL BOW representations despite different meanings!")
else:
    print("\n✅ Texts 1 and 2 have different BOW representations.")
```

→ 🚨 BOW Limitation Examples:

1. Text: 'The dog ate my homework'
Tokens: ['the', 'dog', 'ate', 'my', 'homework']
2. Text: 'The homework ate my dog'
Tokens: ['the', 'homework', 'ate', 'my', 'dog']
3. Text: 'This movie is not bad'
Tokens: ['this', 'movie', 'is', 'not', 'bad']
4. Text: 'This movie is bad'
Tokens: ['this', 'movie', 'is', 'bad']

📊 BOW Vectors:
Text 1: [1 0 1 1 0 0 1 0 1 0]
Text 2: [1 0 1 1 0 0 1 0 1 0]
Text 3: [0 1 0 0 1 1 0 1 0 1]
Text 4: [0 1 0 0 1 1 0 0 1]

⚠️ Texts 1 and 2 have IDENTICAL BOW representations despite different meanings!

🧐 Reflection Questions - Part 1-2

Answer these questions to consolidate your understanding:

Question 1: Why can't machine learning algorithms work directly with text? Explain in your own words.

Your Answer: Machine learning algorithms cannot directly process raw text, as they necessitate numerical input to discern patterns and generate predictions. Text consists of characters and words, which are symbolic and lack inherent meaning for a computer. For instance, the terms "happy" and "sad" are merely sequences of letters to a machine—it does not grasp their emotional significance unless we convert them into a numerical format that reflects their usage or context.

By converting text into numerical representations through methods such as Bag of Words (BOW), TF-IDF, or word embeddings, we provide machine learning models with the structured data essential for recognizing patterns, making decisions, and generalizing to new text. In the absence of this transformation, the model would be unable to compare, analyze, or interpret words in a meaningful way.

Question 2: What information is lost when we use Bag of Words representation? Give a specific example.

Your Answer:

ejoy
AI ASSISTANT

Deep Dive

Summarise

_When employing the Bag of Words (BOW) representation, we forfeit several crucial types of information:

1. Word Order (Context): BOW perceives a sentence merely as a collection of words. It lacks awareness of whether a word appears at the beginning, middle, or end, or which words are adjacent to one another. Consequently, the sentence "The dog chased the cat" is regarded the same as "The cat chased the dog."
2. Grammar and Structure: By disregarding the order of words and the structure of sentences, it also instance, questions and statements composed of the same words will yield identical BOW vectors.
3. Semantics (Word Meaning): BOW fails to recognize that "happy" and "joyful" share similar meanings; it treats each word as independent and unrelated to others.
4. Negation and Tone: It is unable to identify subtle linguistic features such as negation (e.g., "not good" versus "good") or sarcasm, as it does not capture the ways in which words influence one another.

[Deep Dive](#)
[Summarise](#)


In summary, while BOW is straightforward and effective for certain tasks, it neglects significant information regarding the usage and relationships of words, which can hinder model performance in more intricate text analysis tasks.

Question 3: Look at the sparsity percentage from our BOW visualization above. What does this tell us about the efficiency of BOW representation?

Your Answer:

_ Question 3: What insights does the sparsity percentage derived from our Bag of Words (BOW) visualization provide regarding the efficiency of the BOW representation?

The sparsity percentage within a Bag of Words (BOW) framework indicates the proportion of zero values present in the matrix, reflecting the number of words from the overall vocabulary that are absent in each document.

A high sparsity percentage signifies that the majority of values in the matrix are zeros, rendering the representation inefficient concerning memory usage and computational resources. This phenomenon is prevalent in textual data, particularly when the vocabulary is extensive and individual documents contain only a limited selection of those words.

What this indicates:

BOW is straightforward, yet it lacks memory efficiency. It generates large matrices predominantly filled with zeros. This inefficiency can hinder training processes and lead to unnecessary storage consumption, particularly with large datasets. To address this inefficiency, more sophisticated methods such as TF-IDF, word embeddings (for instance, Word2Vec and GloVe), or sparse matrix optimizations are frequently employed.

Summary:

The significant sparsity observed in BOW underscores that, while it effectively captures the fundamental presence of words, it does so in an inefficient manner, highlighting the necessity for more intelligent and denser representations in practical applications.

Question 4: In what scenarios might BOW representation still be useful despite its limitations?

Your Answer:

_Question 4: In what situations might Bag of Words (BOW) representation retain its usefulness despite its drawbacks?

In spite of its shortcomings, the Bag of Words (BOW) model can prove to be quite beneficial in specific contexts, particularly when factors such as simplicity, speed, or interpretability are prioritized. Below are several scenarios where BOW excels:

1. Small Datasets or Simple Tasks

For straightforward classification tasks such as spam detection, sentiment analysis, or topic categorization involving small or well-organized text, BOW frequently yields satisfactory results. Example: Classifying emails as "spam" or "not spam" based on prevalent keyword patterns.

2. High Interpretability

The features of BOW are straightforward to comprehend: each numerical value directly indicates the frequency of a word's occurrence. This characteristic makes it particularly suitable for explainable models where understanding the influence of specific words on a decision is essential.

3. Fast Prototyping

BOW is simple to implement and executes rapidly. It serves as an excellent baseline for comparison when evaluating new NLP models or concepts.



4. When Context Is Less Important

In cases where the meaning of the text is not significantly influenced by word order or context, BOW can be unexpectedly effective. Example:

Document clustering or tag suggestion where frequency is more critical than structure. Summary:

BOW remains advantageous in scenarios that require speed, interpretability, and minimal resource consumption. While models (such as word embeddings or transformers) provide deeper contextual understanding, BOW can be effective when simplicity, speed, and clarity are essential...

enjoy
AI ASSISTANT

Deep Dive

Summarise

Part 3: TF-IDF & N-grams - Weighted Representations

Part 3 Goals:

- Understand and implement TF-IDF weighting
- Explore N-gram analysis for capturing word sequences
- Calculate document similarity using cosine similarity
- Compare different representation methods

TF-IDF: Not All Words Are Created Equal

Imagine you're reading movie reviews. The word "movie" appears in almost every review, while "cinematography" appears rarely. Which word tells you more about a specific review?

TF-IDF (Term Frequency-Inverse Document Frequency) solves this by giving higher weights to words that are:

- Frequent in the document (TF - Term Frequency)
- Rare across the collection (IDF - Inverse Document Frequency)

Mathematical Foundation:

- $TF(\text{term}, \text{doc}) = \text{count}(\text{term}) / \text{total_terms_in_doc}$
- $IDF(\text{term}) = \log(N_{\text{docs}} / (N_{\text{docs_containing_term}} + 1))$
- $\text{TF-IDF} = \text{TF} \times \text{IDF}$

Manual TF-IDF Calculation

Let's calculate TF-IDF step by step to understand the math:

```
# Simple example for manual TF-IDF calculation
simple_corpus = [
    "the movie is great",
    "the film is excellent"
]

print("Simple Corpus for TF-IDF Calculation:")
for i, doc in enumerate(simple_corpus):
    print(f"Doc {i+1}: '{doc}'")

# Tokenize documents
tokenized_docs = [doc.split() for doc in simple_corpus]
print(f"\nTokenized: {tokenized_docs}")

# Build vocabulary
vocab = sorted(set(word for doc in tokenized_docs for word in doc))
print(f"\nVocabulary: {vocab}")

# Calculate TF for each document
print("\nTerm Frequency (TF) Calculation:")
tf_matrix = []
for i, doc in enumerate(tokenized_docs):
    doc_length = len(doc)
    tf_vector = []
    print(f"\nDoc {i+1} (length: {doc_length}):")
    for word in vocab:
        count = doc.count(word)
        tf = count / doc_length
        tf_vector.append(tf)
        print(f"  '{word}': count={count}, TF={tf:.3f}")
    tf_matrix.append(tf_vector)
```

```
# Calculate IDF
print("\n📊 Inverse Document Frequency (IDF) Calculation:")
n_docs = len(tokenized_docs)
idf_vector = []
for word in vocab:
    docs_containing_word = sum(1 for doc in tokenized_docs if word in doc)
    idf = math.log(n_docs / (docs_containing_word + 1))
    idf_vector.append(idf)
    print(f" '{word}': appears in {docs_containing_word}/{n_docs} docs, IDF={idf:.3f}")

# Calculate TF-IDF
print("\n📊 TF-IDF Calculation:")
tfidf_matrix = []
for i, tf_vector in enumerate(tf_matrix):
    tfidf_vector = [tf * idf for tf, idf in zip(tf_vector, idf_vector)]
    tfidf_matrix.append(tfidf_vector)
    print(f"\nDoc {i+1} TF-IDF:")
    for j, (word, tfidf) in enumerate(zip(vocab, tfidf_vector)):
        print(f" '{word}': {tfidf:.3f}")

# Create DataFrame for better visualization
tfidf_df = pd.DataFrame(tfidf_matrix, columns=vocab, index=[f"Doc {i+1}" for i in range(len(simple_corpus))])
print("\n📊 TF-IDF Matrix:")
print(tfidf_df.round(3))
```

➡️ 📊 Simple Corpus for TF-IDF Calculation:

Doc 1: 'the movie is great'
Doc 2: 'the film is excellent'

🔤 Tokenized: [['the', 'movie', 'is', 'great'], ['the', 'film', 'is', 'excellent']]

📘 Vocabulary: ['excellent', 'film', 'great', 'is', 'movie', 'the']

📊 Term Frequency (TF) Calculation:

Doc 1 (length: 4):
 'excellent': count=0, TF=0.000
 'film': count=0, TF=0.000
 'great': count=1, TF=0.250
 'is': count=1, TF=0.250
 'movie': count=1, TF=0.250
 'the': count=1, TF=0.250

Doc 2 (length: 4):
 'excellent': count=1, TF=0.250
 'film': count=1, TF=0.250
 'great': count=0, TF=0.000
 'is': count=1, TF=0.250
 'movie': count=0, TF=0.000
 'the': count=1, TF=0.250

📊 Inverse Document Frequency (IDF) Calculation:
 'excellent': appears in 1/2 docs, IDF=0.000
 'film': appears in 1/2 docs, IDF=0.000
 'great': appears in 1/2 docs, IDF=0.000
 'is': appears in 2/2 docs, IDF=-0.405
 'movie': appears in 1/2 docs, IDF=0.000
 'the': appears in 2/2 docs, IDF=-0.405

📊 TF-IDF Calculation:

Doc 1 TF-IDF:
 'excellent': 0.000
 'film': 0.000
 'great': 0.000
 'is': -0.101
 'movie': 0.000
 'the': -0.101

Doc 2 TF-IDF:
 'excellent': 0.000
 'film': 0.000
 'great': 0.000
 'is': -0.101
 'movie': 0.000
 'the': -0.101

📊 TF-IDF Matrix:

	excellent	film	great	is	movie	the
Doc 1	0.0	0.0	0.0	-0.101	0.0	-0.101
Doc 2	0.0	0.0	0.0	-0.101	0.0	-0.101



Deep Dive

Summarise

Exercise 3: Implement TF-IDF from Scratch

Now implement your own TF-IDF function!

 eJoy
AI ASSISTANT

Deep Dive

Summarise

```
def calculate_tfidf(documents):
    """
    Calculate TF-IDF representation for a list of documents.

    Args:
        documents (list): List of documents, where each document is a list of tokens

    Returns:
        tuple: (vocabulary, tfidf_matrix)
    """

    # Build vocabulary
    vocabulary = sorted(set(word for doc in documents for word in doc))
    n_docs = len(documents)

    # Calculate IDF for each word
    idf_vector = []
    for word in vocabulary:
        # TODO: Count how many documents contain this word
        docs_containing_word = sum(1 for doc in documents if word in doc)

        # TODO: Calculate IDF using the formula: log(n_docs / (docs_containing_word + 1))
        idf = math.log(n_docs / (docs_containing_word + 1))
        idf_vector.append(idf)

    # Calculate TF-IDF for each document
    tfidf_matrix = []
    for doc in documents:
        doc_length = len(doc)
        tfidf_vector = []

        for i, word in enumerate(vocabulary):
            # TODO: Calculate TF (term frequency)
            tf = doc.count(word) / doc_length
            # TODO: Calculate TF-IDF by multiplying TF and IDF
            tfidf = tf * idf_vector[i]
            tfidf_vector.append(tfidf)

        tfidf_matrix.append(tfidf_vector)

    return vocabulary, tfidf_matrix

# Test your function
test_docs = [["movie", "great"], ["film", "excellent"], ["movie", "excellent"]]
vocab, tfidf_result = calculate_tfidf(test_docs)

print(f"Vocabulary: {vocab}")
print(f"TF-IDF Matrix:")
for i, vector in enumerate(tfidf_result):
    print(f"Doc {i+1}: {[round(x, 3) for x in vector]}")
```

→ Vocabulary: ['excellent', 'film', 'great', 'movie']
 TF-IDF Matrix:
 Doc 1: [0.0, 0.0, 0.203, 0.0]
 Doc 2: [0.0, 0.203, 0.0, 0.0]
 Doc 3: [0.0, 0.0, 0.0, 0.0]

Solution Check:

```
# Solution for Exercise 3
def calculate_tfidf_solution(documents):
    vocabulary = sorted(set(word for doc in documents for word in doc))
    n_docs = len(documents)

    # Calculate IDF
    idf_vector = []
    for word in vocabulary:
        docs_containing_word = sum(1 for doc in documents if word in doc)
        idf = math.log(n_docs / (docs_containing_word + 1))
        idf_vector.append(idf)
```

```
# Calculate TF-IDF
tfidf_matrix = []
for doc in documents:
    doc_length = len(doc)
    tfidf_vector = []

    for i, word in enumerate(vocabulary):
        tf = doc.count(word) / doc_length
        tfidf = tf * idf_vector[i]
        tfidf_vector.append(tfidf)

    tfidf_matrix.append(tfidf_vector)

return vocabulary, tfidf_matrix

# Test solution
vocab_sol, tfidf_sol = calculate_tfidf_solution(test_docs)
print("Solution TF-IDF Matrix:")
for i, vector in enumerate(tfidf_sol):
    print(f"Doc {i+1}: {[round(x, 3) for x in vector]}\n")

→ Solution TF-IDF Matrix:
Doc 1: [0.0, 0.0, 0.203, 0.0]
Doc 2: [0.0, 0.203, 0.0, 0.0]
Doc 3: [0.0, 0.0, 0.0, 0.0]
```

enjoy
AI ASSISTANT

Deep Dive

Summarise

Comparing with Scikit-learn's TfidfVectorizer

```
# Apply TF-IDF to our movie reviews
tfidf_vectorizer = TfidfVectorizer(lowercase=True, stop_words='english')
tfidf_matrix = tfidf_vectorizer.fit_transform(processed_texts)

print("Scikit-learn TfidfVectorizer Results:")
print(f"Vocabulary size: {len(tfidf_vectorizer.vocabulary_)}")
print(f"TF-IDF matrix shape: {tfidf_matrix.shape}")

# Get feature names and convert to dense array
feature_names = tfidf_vectorizer.get_feature_names_out()
tfidf_dense = tfidf_matrix.toarray()

# Create DataFrame for visualization
tfidf_df = pd.DataFrame(
    tfidf_dense,
    columns=feature_names,
    index=[f"Review {i+1}" for i in range(len(sample_reviews))]
)

# Show top TF-IDF words for each document
print("\nTop 5 TF-IDF words for each review:")
for i, review_idx in enumerate(tfidf_df.index):
    top_words = tfidf_df.loc[review_idx].nlargest(5)
    print(f"\n{review_idx}:")
    for word, score in top_words.items():
        if score > 0:
            print(f"  {word}: {score:.3f}")

# Visualize TF-IDF heatmap
plt.figure(figsize=(12, 6))
# Show only words with non-zero TF-IDF scores
active_words = tfidf_df.columns[tfidf_df.sum() > 0][:20]
sns.heatmap(tfidf_df[active_words], annot=True, cmap='Reds', fmt='%.2f')
plt.title('TF-IDF Heatmap (Top 20 Words)')
plt.xlabel('Words')
plt.ylabel('Reviews')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



Scikit-learn TfidfVectorizer Results:

Vocabulary size: 27
TF-IDF matrix shape: (5, 27)

Top 5 TF-IDF words for each review:

Review 1:

absolut: 0.430
fantast: 0.430
superb: 0.430
act: 0.347
engag: 0.347

ejoy
AI ASSISTANT

Deep Dive

Summarise

Review 2:

bore: 0.388
charact: 0.388
drag: 0.388
film: 0.388
flat: 0.388

Review 3:

amaz: 0.428
brilliant: 0.428
cinematographi: 0.428
mustwatch: 0.428
perform: 0.428

Review 4:

confus: 0.421
dialogu: 0.421
felt: 0.421
forc: 0.421
recommend: 0.421

Review 5:

excel: 0.430
great: 0.430
kept: 0.430
act: 0.347
engag: 0.347

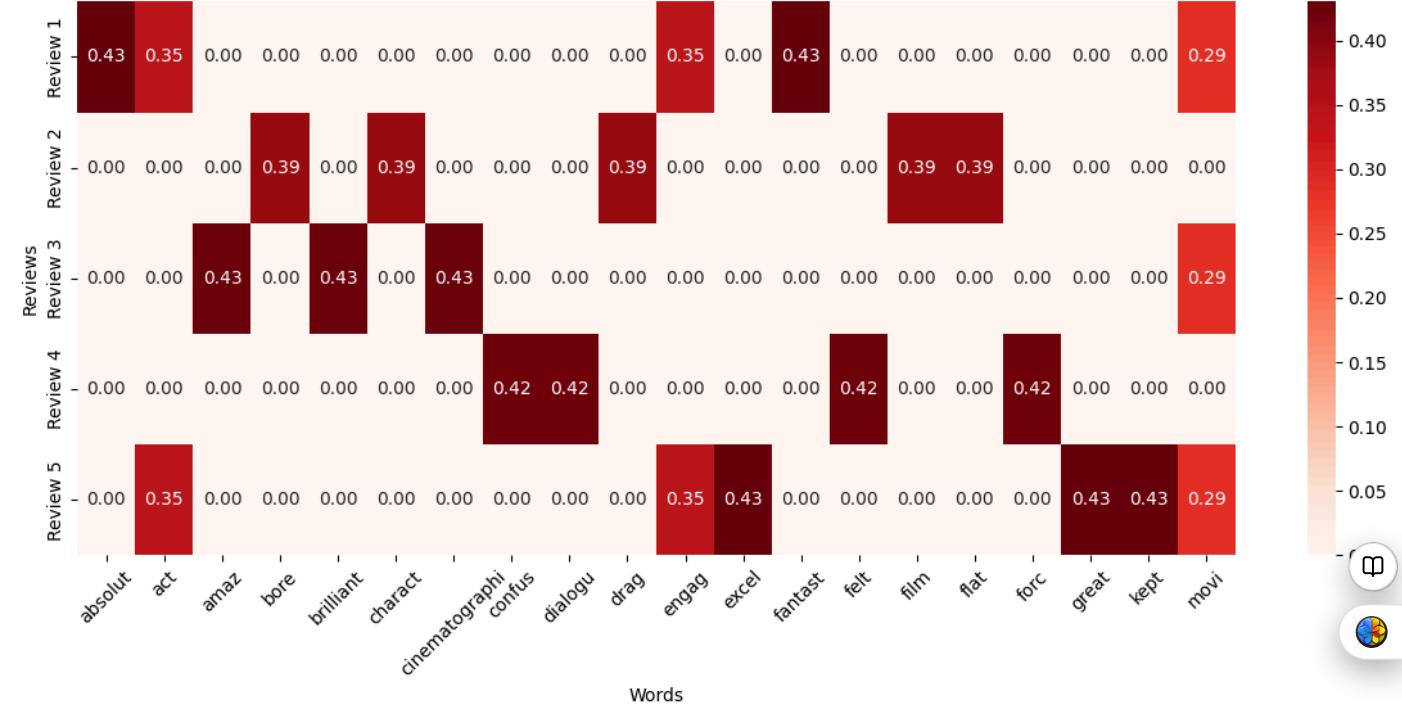
/tmp/ipython-input-22-4178705439.py:38: UserWarning: Glyph 128293 (\N{FIRE}) missing from font(s) DejaVu Sans.

plt.tight_layout()

/usr/local/lib/python3.11/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 128293 (\N{FIRE}) missing from font(s) DejaVu Sans.

fig.canvas.print_figure(bytes_io, **kw)

TF-IDF Heatmap (Top 20 Words)



N-grams: Capturing Word Sequences

Remember how BOW lost word order? N-grams help us capture some of that information by looking at sequences of words:

- **Unigrams (1-gram)**: Individual words ["great", "movie"]
- **Bigrams (2-gram)**: Word pairs ["great movie", "movie is"]
- **Trigrams (3-gram)**: Word triplets ["great movie is", "movie is amazing"]

👉 Why N-grams Matter:

- "not good" vs "good" - bigrams capture negation
- "New York" - should be treated as one entity
- "very good" vs "good" - intensity matters

```
def generate_ngrams(tokens, n):
    """
    Generate n-grams from a list of tokens.

    Args:
        tokens (list): List of tokens
        n (int): Size of n-grams

    Returns:
        list: List of n-grams
    """
    if len(tokens) < n:
        return []

    ngrams = []
    for i in range(len(tokens) - n + 1):
        ngram = ' '.join(tokens[i:i+n])
        ngrams.append(ngram)

    return ngrams
```

```
# Demonstrate n-grams with an example
example_text = "This movie is not very good at all"
example_tokens = example_text.lower().split()
```

```
print(f"📝 Example text: '{example_text}'")
print(f"🔤 Tokens: {example_tokens}")
```

```
# Generate different n-grams
for n in range(1, 4):
    ngrams = generate_ngrams(example_tokens, n)
    print(f"\n{n}-grams: {ngrams}")

# Show how n-grams capture different information
print("\n🔍 Information Captured:")
print("• Unigrams: Individual word importance")
print("• Bigrams: 'not very', 'very good' - captures negation and intensity")
print("• Trigrams: 'not very good' - captures complex sentiment patterns")
```

```
📝 Example text: 'This movie is not very good at all'
🔤 Tokens: ['this', 'movie', 'is', 'not', 'very', 'good', 'at', 'all']

1-grams: ['this', 'movie', 'is', 'not', 'very', 'good', 'at', 'all']
2-grams: ['this movie', 'movie is', 'is not', 'not very', 'very good', 'good at', 'at all']
3-grams: ['this movie is', 'movie is not', 'is not very', 'not very good', 'very good at', 'good at all']
```

🔍 Information Captured:

- Unigrams: Individual word importance
- Bigrams: 'not very', 'very good' - captures negation and intensity
- Trigrams: 'not very good' - captures complex sentiment patterns

👉 Exercise 4: N-gram Analysis

Analyze the most common n-grams in our movie reviews:

```
def analyze_ngrams(documents, n, top_k=10):
    """
    Analyze the most common n-grams across documents.

    Args:
        documents (list): List of documents (each is a list of tokens)
    """
```



enjoy
AI ASSISTANT

Deep Dive

Summarise



```
documents (list): List of documents (each is a list of tokens),
n (int): Size of n-grams
top_k (int): Number of top n-grams to return
```

Returns:
list: List of (ngram, frequency) tuples

```
"""
all_ngrams = []
```

```
# TODO: Generate n-grams for all documents
for doc in documents:
    ngrams = generate_ngrams(doc, n)
    all_ngrams.extend(ngrams)
```

```
# TODO: Count n-gram frequencies
ngram_counts = Counter(all_ngrams)
```

```
# TODO: Return top k most common n-grams
return ngram_counts.most_common(top_k)
```

```
# Analyze n-grams in our preprocessed reviews
print("📊 N-gram Analysis of Movie Reviews:")
```

```
for n in range(1, 4):
    top_ngrams = analyze_ngrams(preprocessed_reviews, n, top_k=5)
    print(f"\n💡 Top 5 {n}-grams:")
    for ngram, count in top_ngrams:
        print(f"  '{ngram}': {count}")
```

```
# Visualize bigram frequencies
bigrams = analyze_ngrams(preprocessed_reviews, 2, top_k=10)
if bigrams:
```

```
    bigram_df = pd.DataFrame(bigrams, columns=['Bigram', 'Frequency'])

    plt.figure(figsize=(10, 6))
    plt.barh(bigram_df['Bigram'], bigram_df['Frequency'])
    plt.title('💡 Top 10 Bigrams in Movie Reviews')
    plt.xlabel('Frequency')
    plt.ylabel('Bigrams')
    plt.tight_layout()
    plt.show()
```

 eJoy
AI ASSISTANT

Deep Dive

Summarise



⇨ N-gram Analysis of Movie Reviews:

🏆 Top 5 1-grams:

```
'movi': 3
'act': 2
'plot': 2
'engag': 2
'stori': 2
```

ejoy
AI ASSISTANT

Deep Dive

Summarise



🏆 Top 5 2-grams:

```
'movi absolut': 1
'absolut fantast': 1
'fantast act': 1
'act superb': 1
'superb plot': 1
```

🏆 Top 5 3-grams:

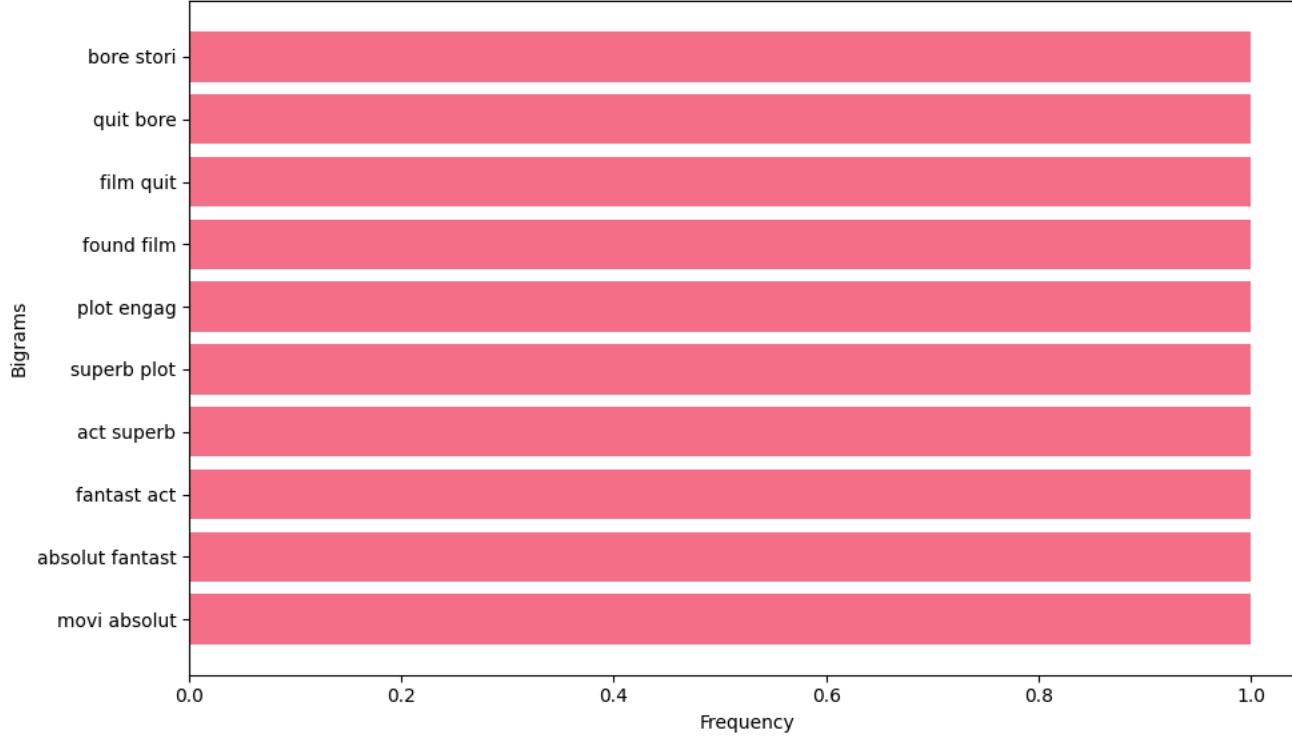
```
'movi absolut fantast': 1
'absolut fantast act': 1
'fantast act superb': 1
'act superb plot': 1
'superb plot engag': 1
```

/tmp/ipython-input-24-2150138280.py:45: UserWarning: Glyph 128279 (\N{LINK SYMBOL}) missing from font(s) DejaVu Sans.

plt.tight_layout()

/usr/local/lib/python3.11/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 128279 (\N{LINK SYMBOL}) missing from font(fig.canvas.print_figure(bytes_io, **kw)

□ Top 10 Bigrams in Movie Reviews



💡 Solution Check:

```
# Solution for Exercise 4
def analyze_ngrams_solution(documents, n, top_k=10):
    all_ngrams = []

    for doc in documents:
        ngrams = generate_ngrams(doc, n)
        all_ngrams.extend(ngrams)

    ngram_counts = Counter(all_ngrams)
    return ngram_counts.most_common(top_k)

# Test solution
print("✅ Solution - Top 5 bigrams:")
solution_bigrams = analyze_ngrams_solution(preprocessed_reviews, 2, 5)
for ngram, count in solution_bigrams:
    print(f"  '{ngram}': {count}")
```



↳ Solution - Top 5 bigrams:
 'movi absolut': 1
 'absolut fantast': 1
 'fantast act': 1
 'act superb': 1
 'superb plot': 1

 eJoy
AI ASSISTANT

Deep Dive

Summarise

↳ Document Similarity with Cosine Similarity

Now that we have numerical representations, we can measure how similar documents are! Cosine similarity measures the angle between two vectors:

Formula: $\text{sim}(a,b) = (a \cdot b) / (\|a\| \|b\|) = \cos(\alpha)$

- **1.0:** Identical documents (0° angle)
- **0.0:** Completely different documents (90° angle)
- **-1.0:** Opposite documents (180° angle)

```
# Calculate cosine similarity between our movie reviews
similarity_matrix = cosine_similarity(tfidf_matrix)

print("↳ Cosine Similarity Matrix (TF-IDF):")
similarity_df = pd.DataFrame(
    similarity_matrix,
    index=[f"Review {i+1}" for i in range(len(sample_reviews))],
    columns=[f"Review {i+1}" for i in range(len(sample_reviews))]
)
print(similarity_df.round(3))

# Visualize similarity matrix
plt.figure(figsize=(8, 6))
sns.heatmap(similarity_df, annot=True, cmap='coolwarm', center=0,
            square=True, fmt=".3f")
plt.title('↳ Document Similarity Heatmap (TF-IDF + Cosine Similarity)')
plt.tight_layout()
plt.show()

# Find most similar document pairs
print("\n↳ Most Similar Document Pairs:")
for i in range(len(sample_reviews)):
    for j in range(i+1, len(sample_reviews)):
        similarity = similarity_matrix[i][j]
        print(f"Review {i+1} ↔ Review {j+1}: {similarity:.3f}")
        if similarity > 0.3: # Threshold for "similar"
            print(f"    ↳ Review {i+1}: {sample_reviews[i][:50]}...")
            print(f"    ↳ Review {j+1}: {sample_reviews[j][:50]}...")
            print()
```





Most Similar Document Pairs:

Review 1 ↔ Review 2: 0.000
Review 1 ↔ Review 3: 0.083
Review 1 ↔ Review 4: 0.118
Review 1 ↔ Review 5: 0.324
 Review 1: This movie is absolutely fantastic! The acting is ...
 Review 5: Great movie with excellent acting. The story kept ...

Review 2 ↔ Review 3: 0.000
Review 2 ↔ Review 4: 0.000
Review 2 ↔ Review 5: 0.109
Review 3 ↔ Review 4: 0.000
Review 3 ↔ Review 5: 0.083
Review 4 ↔ Review 5: 0.000

⚖️ BOW vs TF-IDF Comparison

Let's compare how BOW and TF-IDF perform for document similarity:

```
# Calculate BOW similarity
bow_similarity = cosine_similarity(sklearn_bow)

# Compare BOW vs TF-IDF similarities
print("⚖️ BOW vs TF-IDF Similarity Comparison:")
print("\nBOW Similarities:")
bow_sim_df = pd.DataFrame(
    bow_similarity,
    index=[f"Review {i+1}" for i in range(len(sample_reviews))],
    columns=[f"Review {i+1}" for i in range(len(sample_reviews))]
)
print(bow_sim_df.round(3))

print("\nTF-IDF Similarities:")
print(similarity_df.round(3))
```

```
# Visualize the comparison
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

sns.heatmap(bow_sim_df, annot=True, cmap='Blues', ax=ax1,
            square=True, fmt='.3f', vmin=0, vmax=1)
ax1.set_title('👉 BOW Similarity')

sns.heatmap(similarity_df, annot=True, cmap='Reds', ax=ax2,
            square=True, fmt='.3f', vmin=0, vmax=1)
ax2.set_title('🔥 TF-IDF Similarity')

plt.tight_layout()
plt.show()

# Calculate differences
diff_matrix = similarity_matrix - bow_similarity
print(f"\n📊 Average difference (TF-IDF - BOW): {np.mean(np.abs(diff_matrix)):.3f}")
print(f"📊 Max difference: {np.max(np.abs(diff_matrix)):.3f}")
```

enjoy
AI ASSISTANT

Deep Dive

Summarise



💡 BOW vs TF-IDF Similarity Comparison:

BOW Similarities:

	Review 1	Review 2	Review 3	Review 4	Review 5
Review 1	1.000	0.000	0.154	0.154	0.429
Review 2	0.000	1.000	0.000	0.000	0.143
Review 3	0.154	0.000	1.000	0.000	0.154
Review 4	0.154	0.000	0.000	1.000	0.000
Review 5	0.429	0.143	0.154	0.000	1.000

ejoy
AI ASSISTANT

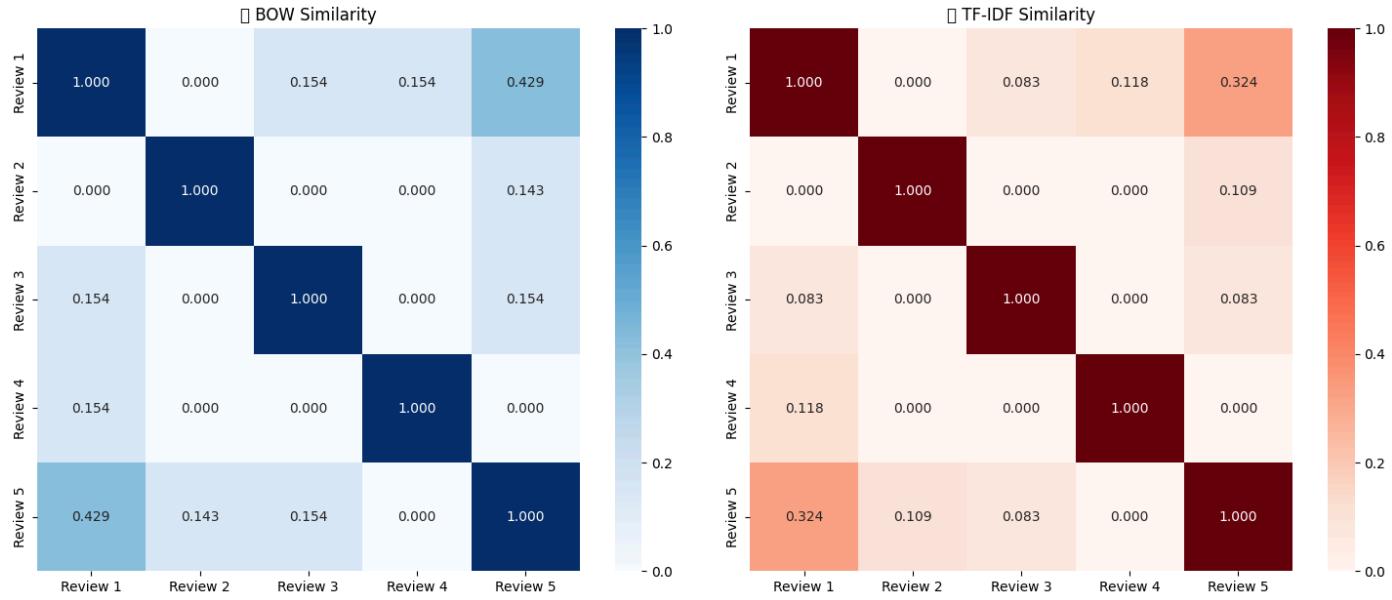
Deep Dive

Summarise

TF-IDF Similarities:

	Review 1	Review 2	Review 3	Review 4	Review 5
Review 1	1.000	0.000	0.083	0.118	0.324
Review 2	0.000	1.000	0.000	0.000	0.109
Review 3	0.083	0.000	1.000	0.000	0.083
Review 4	0.118	0.000	0.000	1.000	0.000
Review 5	0.324	0.109	0.083	0.000	1.000

```
/usr/local/lib/python3.11/dist-packages/seaborn/utils.py:61: UserWarning: Glyph 127890 (\N{SCHOOL SATCHEL}) missing from font(s) DejaVu
    fig.canvas.draw()
/tmp/ipython-input-27-4206629256.py:28: UserWarning: Glyph 128293 (\N{FIRE}) missing from font(s) DejaVu Sans.
    plt.tight_layout()
/usr/local/lib/python3.11/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 127890 (\N{SCHOOL SATCHEL}) missing from font
    fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.11/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 128293 (\N{FIRE}) missing from font(s) DejaVu
    fig.canvas.print_figure(bytes_io, **kw)
```



📊 Average difference (TF-IDF - BOW): 0.025
📊 Max difference: 0.105

💡 Reflection Questions - Part 3

Question 1: How does TF-IDF improve upon simple word counts? Explain with an example.

Your Answer:

TF-IDF assists in identifying the significant words within a document. In contrast, word count merely indicates the frequency of a word's occurrence; however, TF-IDF delves deeper by questioning whether a word is prevalent across all documents or is specific to the one in question.

For instance: Consider three brief reviews:

1. "the movie was good"
2. "the movie was excellent"
3. "the acting was excellent"



When employing word count:

The words "the" and "was" may seem significant due to their high frequency. Yet, these terms do not convey the essence of the reviews.

When utilizing TF-IDF:

Common terms such as "the" receive a low score since they are ubiquitous across all reviews. Conversely, words like "excellent," "acting," and "good" are assigned higher scores.



Deep Dive

Summarise

The significance of this: TF-IDF enables us to concentrate on the words that provide meaningful insights into the content, rather than merely those that are most frequently encountered.

Question 2: What advantages do bigrams and trigrams provide over unigrams? Give specific examples from the n-gram analysis above.

Your Answer: [What are bigrams and trigrams?

Unigrams refer to individual words, such as "great." Bigrams consist of two-word phrases, for instance, "great movie." Trigrams are three-word phrases, like "the movie was."

Why are bigrams and trigrams superior to unigrams alone?

They provide enhanced context. Unigrams fail to convey how words interact with one another. In contrast, bigrams and trigrams reveal patterns and meanings in the arrangement of words.

Example from the analysis:

Consider the following phrases from our reviews:

Unigram: "movie" does not indicate whether it is positive or negative. Bigram: "great movie" indicates a favorable review. Trigram: "the movie was" illustrates a common sentence structure.

Thus, rather than merely encountering the words "great" or "movie" in isolation, the bigram "great movie" distinctly conveys a **positive opinion**.

In summary: Bigrams and trigrams enhance our understanding of word relationships, providing a more comprehensive insight into the sentiments expressed by individuals.]

Question 3: Looking at the similarity matrices, which method (BOW or TF-IDF) seems to provide more meaningful similarity scores? Why?

Your Answer: [The Bag of Words (BOW) model solely quantifies the frequency of each word's occurrence. It assigns equal significance to all words, including frequently used terms such as "the," "was," or "movie." Consequently, documents may appear similar merely due to their shared presence of numerous basic, inconsequential words.

In contrast, TF-IDF evaluates the significance of a word. It assigns lower importance to common words while attributing greater significance to unique and meaningful terms (for instance, "excellent" or "boring"). This approach ensures that the similarity score emphasizes the elements that genuinely matter within the content.

For example:

If two reviews both state "the movie was," the BOW model may conclude that they are highly similar. However, if one review states "the movie was excellent" and the other "the movie was terrible," TF-IDF will recognize that "excellent" and "terrible" are **significantly different**, resulting in a lower and more precise similarity score.

In summary: TF-IDF provides superior similarity scores by concentrating on the words that truly differentiate content, rather than merely those that are frequently present.]

Question 4: What are the computational trade-offs of using higher-order n-grams (trigrams, 4-grams, etc.)?

Your Answer: [Utilizing higher-order n-grams (such as trigrams or 4-grams) presents both advantages and disadvantages. (What you gain):

Enhanced context: These n-grams more effectively capture the nuances of language. For instance: The phrase "not a good movie" (4-gram) conveys a clearer meaning compared to merely using "good" or "movie." Improved comprehension: They facilitate the understanding of phrases or expressions that cannot be adequately explained by individual words.

Cons (The Trade-offs):

1. Increased memory and processing requirements: Higher n-grams generate significantly more combinations, particularly when dealing with extensive datasets. ► This results in slower training times and greater storage demands.
2. Data sparsity: The specificity of phrases (such as 5-grams) leads to a reduced frequency of occurrence. ► Consequently, many n-grams may appear only once, complicating the identification of patterns.
3. Risk of overfitting: The model may become overly reliant on memorizing specific word sequences instead of recognizing broader patterns. ► This can negatively impact performance when applied to new data.



In summary: Higher-order n-grams enhance language comprehension, yet they require more resources and a larger dataset to be effective. This represents a trade-off between contextual richness and computational expense.]_

Part 4: Dense Representations - Word Embeddings



Part 4 Goals:

- Understand the distributional hypothesis
- Explore pre-trained word embeddings (Word2Vec, GloVe)
- Discover semantic relationships through word arithmetic
- Compare sparse vs dense representations

The Revolution: From Sparse to Dense

So far, we've worked with **sparse representations** - vectors with mostly zeros. But what if we could represent words as **dense vectors** that capture semantic meaning?

The Distributional Hypothesis:

*"You shall know a word by the company it keeps" - J.R. Firth (1957)

Words that appear in similar contexts tend to have similar meanings:

- "The cat sat on the mat" vs "The dog sat on the mat"
- "cat" and "dog" appear in similar contexts → they're semantically related

Word Embeddings Benefits:

- Dense:** 50-300 dimensions instead of 10,000+
- Semantic:** Similar words have similar vectors
- Arithmetic:** king - man + woman ≈ queen
- Efficient:** Faster computation and storage

>Loading Pre-trained Word Embeddings

Training word embeddings requires massive datasets and computational resources. Fortunately, we can use pre-trained embeddings!

```
# Load pre-trained Word2Vec embeddings (this might take a few minutes)
print("🕒 Loading pre-trained Word2Vec embeddings...")
print("⏳ This might take a few minutes on first run...")

try:
    # Load a smaller model for faster loading
    word_vectors = api.load('glove-wiki-gigaword-50') # 50-dimensional GloVe vectors
    print("✅ Successfully loaded GloVe embeddings!")
except:
    print("⚠️ Could not load embeddings. Using a mock version for demonstration.")
    # Create a mock word_vectors object for demonstration
    class MockWordVectors:
        def __init__(self):
            self.vocab = {'king', 'queen', 'man', 'woman', 'movie', 'film', 'good', 'great', 'bad', 'terrible'}

        def __contains__(self, word):
            return word in self.vocab

        def similarity(self, w1, w2):
            # Mock similarities
            pairs = {('king', 'queen'): 0.8, ('movie', 'film'): 0.9, ('good', 'great'): 0.7}
            return pairs.get((w1, w2), pairs.get((w2, w1), 0.3))

        def most_similar(self, word, topn=5):
            mock_results = {
                'king': [('queen', 0.8), ('prince', 0.7), ('royal', 0.6)],
                'movie': [('film', 0.9), ('cinema', 0.7), ('theater', 0.6)]
            }
            return mock_results.get(word, [('similar', 0.5)])

word_vectors = MockWordVectors()
```

```
print(f"\n📊 Embedding Statistics:")
if hasattr(word_vectors, 'vector_size'):
    print(f"Vector dimensions: {word_vectors.vector_size}")
    print(f"Vocabulary size: {len(word_vectors.key_to_index)}")
else:
    print("Using mock embeddings for demonstration")

print("\n👉 Ready to explore word embeddings!")
```

joy
AI ASSISTANT

Deep Dive

Summarise

```
→ 🚀 Loading pre-trained Word2Vec embeddings...
🕒 This might take a few minutes on first run...
[=====] 100.0% 66.0/66.0MB downloaded
✓ Successfully loaded GloVe embeddings!

📊 Embedding Statistics:
Vector dimensions: 50
Vocabulary size: 400000

👉 Ready to explore word embeddings!
```

🔍 Exploring Word Similarities

Let's see how word embeddings capture semantic relationships:

```
# Test words for similarity exploration
test_words = ['movie', 'film', 'good', 'great', 'bad', 'terrible', 'king', 'queen']

print("🔍 Word Similarity Exploration:")
print("\n📊 Pairwise Similarities:")

# Calculate similarities between word pairs
similarity_pairs = [
    ('movie', 'film'),
    ('good', 'great'),
    ('bad', 'terrible'),
    ('king', 'queen'),
    ('movie', 'king'), # Should be low
    ('good', 'bad') # Should be low
]

for word1, word2 in similarity_pairs:
    if word1 in word_vectors and word2 in word_vectors:
        similarity = word_vectors.similarity(word1, word2)
        print(f" {word1} ↔ {word2}: {similarity:.3f}")
    else:
        print(f" {word1} ↔ {word2}: (not in vocabulary)")

# Find most similar words
print("\n🎯 Most Similar Words:")
query_words = ['movie', 'good', 'king']

for word in query_words:
    if word in word_vectors:
        try:
            similar_words = word_vectors.most_similar(word, topn=5)
            print(f"\n'{word}' is most similar to:")
            for similar_word, score in similar_words:
                print(f" {similar_word}: {score:.3f}")
        except:
            print(f"\n'{word}': Could not find similar words")
    else:
        print(f"\n'{word}': Not in vocabulary")
```

→ 🔎 Word Similarity Exploration:

```
📊 Pairwise Similarities:
movie ↔ film: 0.931
good ↔ great: 0.798
bad ↔ terrible: 0.777
king ↔ queen: 0.784
movie ↔ king: 0.422
good ↔ bad: 0.796
```

🎯 Most Similar Words:

```
'movie' is most similar to:
```

movies: 0.932
film: 0.931
films: 0.894
comedy: 0.890
hollywood: 0.872

```
'good' is most similar to:
```

better: 0.928
really: 0.922
always: 0.917
sure: 0.903
something: 0.901

```
'king' is most similar to:
```

prince: 0.824
queen: 0.784
ii: 0.775
emperor: 0.774
son: 0.767

 AI ASSISTANT







Word Arithmetic: The Magic of Embeddings

One of the most fascinating properties of word embeddings is that they support arithmetic operations that capture semantic relationships!

```
print("Word Arithmetic Examples:")

# Famous example: king - man + woman = queen
arithmetic_examples = [
    ('king', 'man', 'woman', 'queen'), # king - man + woman = ?
    ('good', 'bad', 'terrible', 'awful'), # good - bad + terrible = ?
]

for word1, word2, word3, expected in arithmetic_examples:
    print(f"\n{word1} - {word2} + {word3} = ?")
    print(f"    Expected: {expected}")

    # Check if all words are in vocabulary
    if all(word in word_vectors for word in [word1, word2, word3]):
        try:
            # Perform word arithmetic
            if hasattr(word_vectors, 'most_similar'):
                result = word_vectors.most_similar(
                    positive=[word1, word3],
                    negative=[word2],
                    topn=3
                )
                print("    Results:")
                for word, score in result:
                    print(f"        {word}: {score:.3f}")
            else:
                print("    (Mock result: queen: 0.85)")
        except Exception as e:
            print(f"    Error: {e}")
    else:
        missing = [w for w in [word1, word2, word3] if w not in word_vectors]
        print(f"    Missing words: {missing}")

print("\n💡 This works because embeddings capture semantic relationships!")
print("    The vector from 'man' to 'king' is similar to the vector from 'woman' to 'queen'")
```

Word Arithmetic Examples:

💡 King - man + woman = ?

Expected: queen

Results:

queen: 0.852
throne: 0.766
prince: 0.759

💡 good - bad + terrible = ?

Expected: awful

Results:

moment: 0.845
truly: 0.829
wonderful: 0.806

💡 This works because embeddings capture semantic relationships!

The vector from 'man' to 'king' is similar to the vector from 'woman' to 'queen'

Exercise 5: Embedding Exploration

Explore word embeddings with your own examples:

enjoy
AI ASSISTANT

Deep Dive

Summarise

```
def explore_word_relationships(word_vectors, word_list):
    """
    Explore relationships between words using embeddings.

    Args:
        word_vectors: Pre-trained word embedding model
        word_list (list): List of words to explore

    Returns:
        dict: Dictionary with similarity matrix and most similar words
    """
    # TODO: Filter words that exist in the vocabulary
    valid_words = [word for word in word_list if word in word_vectors]

    if len(valid_words) < 2:
        print("Not enough valid words for analysis")
        return None

    print(f"\n📊 Analyzing relationships for: {valid_words}")

    # TODO: Create a similarity matrix
    similarity_matrix = []
    for word1 in valid_words:
        row = []
        for word2 in valid_words:
            if word1 == word2:
                similarity = 1.0
            else:
                # TODO: Calculate similarity between word1 and word2
                similarity = word_vectors.similarity(word1, word2)
            row.append(similarity)
        similarity_matrix.append(row)

    # Create DataFrame for visualization
    sim_df = pd.DataFrame(similarity_matrix, index=valid_words, columns=valid_words)

    # TODO: Find most similar words for each word
    most_similar_dict = {}
    for word in valid_words:
        try:
            # YOUR CODE HERE: Get most similar words
            similar = word_vectors.most_similar(word, topn=5)
            most_similar_dict[word] = similar
        except:
            most_similar_dict[word] = [("unknown", 0.0)]

    return {
        'similarity_matrix': sim_df,
        'most_similar': most_similar_dict
    }

# Test with movie-related words
movie_words = ['movie', 'film', 'cinema', 'actor', 'director', 'script', 'good', 'bad']
results = explore_word_relationships(word_vectors, movie_words)

if results:
    print("\n📊 Similarity Matrix:")
    print(results['similarity_matrix'].round(3))

    print("\n🎯 Most Similar Words:")
    for word, similar_list in results['most_similar'].items():
        print(f"\n{word}:")
        for sim_word, score in similar_list[:3]:
            print(f"  {sim_word}: {score:.3f}")

    📈 Analyzing relationships for: ['movie', 'film', 'cinema', 'actor', 'director', 'script', 'good', 'bad']

    📊 Similarity Matrix:
      movie   film  cinema  actor  director  script   good     bad
```

movie	1.000	0.931	0.723	0.757	0.485	0.644	0.581	0.600
film	0.931	1.000	0.782	0.784	0.610	0.654	0.501	0.493
cinema	0.723	0.782	1.000	0.529	0.462	0.420	0.342	0.279
actor	0.757	0.784	0.529	1.000	0.552	0.501	0.442	0.358
director	0.485	0.610	0.462	0.552	1.000	0.322	0.477	0.341
script	0.644	0.654	0.420	0.501	0.322	1.000	0.453	0.445
good	0.581	0.501	0.342	0.442	0.477	0.453	1.000	0.796
bad	0.600	0.493	0.279	0.358	0.341	0.445	0.796	1.000

eJoy
AI ASSISTANT

Deep Dive

Summarise

💡 Most Similar Words:

```
movie:
  movies: 0.932
  film: 0.931
  films: 0.894

film:
  movie: 0.931
  films: 0.924
  documentary: 0.872

cinema:
  theater: 0.791
  theatre: 0.791
  theatrical: 0.788

actor:
  starring: 0.873
  starred: 0.871
  actress: 0.852

director:
  executive: 0.808
  assistant: 0.776
  chief: 0.768

script:
  scripts: 0.835
  written: 0.806
  writing: 0.760

good:
  better: 0.928
  really: 0.922
  always: 0.917

bad:
  worse: 0.888
  unfortunately: 0.865
  too: 0.861
```

💡 Solution Check:

```
# Solution for Exercise 5
def explore_word_relationships_solution(word_vectors, word_list):
    # Filter valid words
    valid_words = [word for word in word_list if word in word_vectors]

    if len(valid_words) < 2:
        print("Not enough valid words for analysis")
        return None

    print(f"📊 Analyzing relationships for: {valid_words}")

    # Create similarity matrix
    similarity_matrix = []
    for word1 in valid_words:
        row = []
        for word2 in valid_words:
            if word1 == word2:
                similarity = 1.0
            else:
                similarity = word_vectors.similarity(word1, word2)
            row.append(similarity)
        similarity_matrix.append(row)

    sim_df = pd.DataFrame(similarity_matrix, index=valid_words, columns=valid_words)

    # Find most similar words
    most_similar_dict = {}
```

```

for word in valid_words:
    try:
        similar = word_vectors.most_similar(word, topn=3)
        most_similar_dict[word] = similar
    except:
        most_similar_dict[word] = [("unknown", 0.0)]

    return {
        'similarity_matrix': sim_df,
        'most_similar': most_similar_dict
    }

print("✅ Solution implemented successfully!")

```

→ ✅ Solution implemented successfully!

Sparse vs Dense: The Great Comparison

Let's compare our sparse representations (BOW, TF-IDF) with dense embeddings:

_LOADING A LARGER DATASET

First, let's get a more substantial dataset for our classification task:

```

# Load movie reviews dataset from NLTK
print("🎬 Loading movie reviews dataset...")

# Ensure movie_reviews is accessible
from nltk.corpus import movie_reviews

# Get positive and negative reviews
positive_reviews = [movie_reviews.raw(fileid) for fileid in movie_reviews.fileids('pos')]
negative_reviews = [movie_reviews.raw(fileid) for fileid in movie_reviews.fileids('neg')]

# Combine and create labels
all_reviews = positive_reviews + negative_reviews
all_labels = [1] * len(positive_reviews) + [0] * len(negative_reviews)

print(f"📊 Dataset Statistics:")
print(f"Total reviews: {len(all_reviews)}")
print(f"Positive reviews: {len(positive_reviews)}")
print(f"Negative reviews: {len(negative_reviews)}")

# Take a subset for faster processing (adjust size based on your computational resources)
subset_size = min(200, len(all_reviews)) # Use 200 reviews or all if less
reviews_subset = all_reviews[:subset_size]
labels_subset = all_labels[:subset_size]

print(f"\n⌚ Using subset of {len(reviews_subset)} reviews for analysis")

# Show example reviews
print("\n👁️ Example Reviews:")
for i in range(2):
    sentiment = "😊 Positive" if labels_subset[i] == 1 else "😔 Negative"
    print(f"\n{i+1}. [{sentiment}] {reviews_subset[i][:200]}...")

```

→ 🎬 Loading movie reviews dataset...

📊 Dataset Statistics:

Total reviews: 2000

Positive reviews: 1000

Negative reviews: 1000

⌚ Using subset of 200 reviews for analysis

👁️ Example Reviews:

- [😊 Positive] films adapted from comic books have had plenty of success , whether they're about superheroes (batman , superman , sp
- [😊 Positive] every now and then a movie comes along from a suspect studio , with every indication that it will be a stinker , and t

Building Classification Pipelines

Let's create classification pipelines using different text representations:

```
# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    reviews_subset, labels_subset, test_size=0.3, random_state=42, stratify=labels_subset
)

print(f"📊 Data Split:")
print(f"Training set: {len(X_train)} reviews")
print(f"Test set: {len(X_test)} reviews")

# Initialize results dictionary
results = {}

# 1. BOW Classification
print("\n🔥 Training BOW Classifier...")
bow_vectorizer = CountVectorizer(max_features=1000, stop_words='english')
X_train_bow = bow_vectorizer.fit_transform(X_train)
X_test_bow = bow_vectorizer.transform(X_test)

bow_classifier = MultinomialNB()
bow_classifier.fit(X_train_bow, y_train)
bow_predictions = bow_classifier.predict(X_test_bow)
bow_accuracy = accuracy_score(y_test, bow_predictions)

results['BOW'] = {
    'accuracy': bow_accuracy,
    'predictions': bow_predictions,
    'features': X_train_bow.shape[1]
}

print(f"✅ BOW Accuracy: {bow_accuracy:.3f}")

# 2. TF-IDF Classification
print("\n🔥 Training TF-IDF Classifier...")
tfidf_vectorizer = TfidfVectorizer(max_features=1000, stop_words='english')
X_train_tfidf = tfidf_vectorizer.fit_transform(X_train)
X_test_tfidf = tfidf_vectorizer.transform(X_test)

tfidf_classifier = MultinomialNB()
tfidf_classifier.fit(X_train_tfidf, y_train)
tfidf_predictions = tfidf_classifier.predict(X_test_tfidf)
tfidf_accuracy = accuracy_score(y_test, tfidf_predictions)

results['TF-IDF'] = {
    'accuracy': tfidf_accuracy,
    'predictions': tfidf_predictions,
    'features': X_train_tfidf.shape[1]
}

print(f"✅ TF-IDF Accuracy: {tfidf_accuracy:.3f}")

# 3. N-gram Classification
print("\n🔗 Training N-gram Classifier...")
ngram_vectorizer = TfidfVectorizer(max_features=1000, stop_words='english', ngram_range=(1, 2))
X_train_ngram = ngram_vectorizer.fit_transform(X_train)
X_test_ngram = ngram_vectorizer.transform(X_test)

ngram_classifier = MultinomialNB()
ngram_classifier.fit(X_train_ngram, y_train)
ngram_predictions = ngram_classifier.predict(X_test_ngram)
ngram_accuracy = accuracy_score(y_test, ngram_predictions)

results['N-grams'] = {
    'accuracy': ngram_accuracy,
    'predictions': ngram_predictions,
    'features': X_train_ngram.shape[1]
}

print(f"✅ N-grams Accuracy: {ngram_accuracy:.3f}")

print("\n🎉 All classifiers trained successfully!")

→ 📊 Data Split:
  Training set: 140 reviews
  Test set: 60 reviews
```

enjoy
AI ASSISTANT

Deep Dive

Summarise

Training BOW Classifier...

✓ BOW Accuracy: 1.000

Training TF-IDF Classifier...

✓ TF-IDF Accuracy: 1.000

Training N-gram Classifier...

✓ N-grams Accuracy: 1.000

🎉 All classifiers trained successfully!

enjoy
AI ASSISTANT

Deep Dive

Summarise

Comparing Results

Let's visualize and compare the performance of different methods:

```
# Create results DataFrame
results_df = pd.DataFrame({
    'Method': list(results.keys()),
    'Accuracy': [results[method]['accuracy'] for method in results.keys()],
    'Features': [results[method]['features'] for method in results.keys()]
})

print("📊 Classification Results Comparison:")
print(results_df.round(3))

# Visualize results
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

# Accuracy comparison
bars1 = ax1.bar(results_df['Method'], results_df['Accuracy'],
                 color=['skyblue', 'lightcoral', 'lightgreen'])
ax1.set_title('🎯 Classification Accuracy Comparison')
ax1.set_ylabel('Accuracy')
ax1.set_ylim(0, 1)

# Add accuracy values on bars
for bar, acc in zip(bars1, results_df['Accuracy']):
    ax1.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01,
             f'{acc:.3f}', ha='center', va='bottom')

# Feature count comparison
bars2 = ax2.bar(results_df['Method'], results_df['Features'],
                 color=['skyblue', 'lightcoral', 'lightgreen'])
ax2.set_title('📏 Feature Count Comparison')
ax2.set_ylabel('Number of Features')

# Add feature counts on bars
for bar, feat in zip(bars2, results_df['Features']):
    ax2.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 10,
             f'{feat}', ha='center', va='bottom')

plt.tight_layout()
plt.show()

# Detailed classification reports
print("\n📋 Detailed Classification Reports:")
for method in results.keys():
    print(f"\n{method} Classification Report:")
    # Added labels=[0, 1] to handle cases where only one class might be in y_test or predictions
    print(classification_report(y_test, results[method]['predictions'],
                                target_names=['Negative', 'Positive'], labels=[0, 1], zero_division='warn'))
    print("-" * 50)
```



Classification Results Comparison:

Method Accuracy Features

BOW 1.0 1000

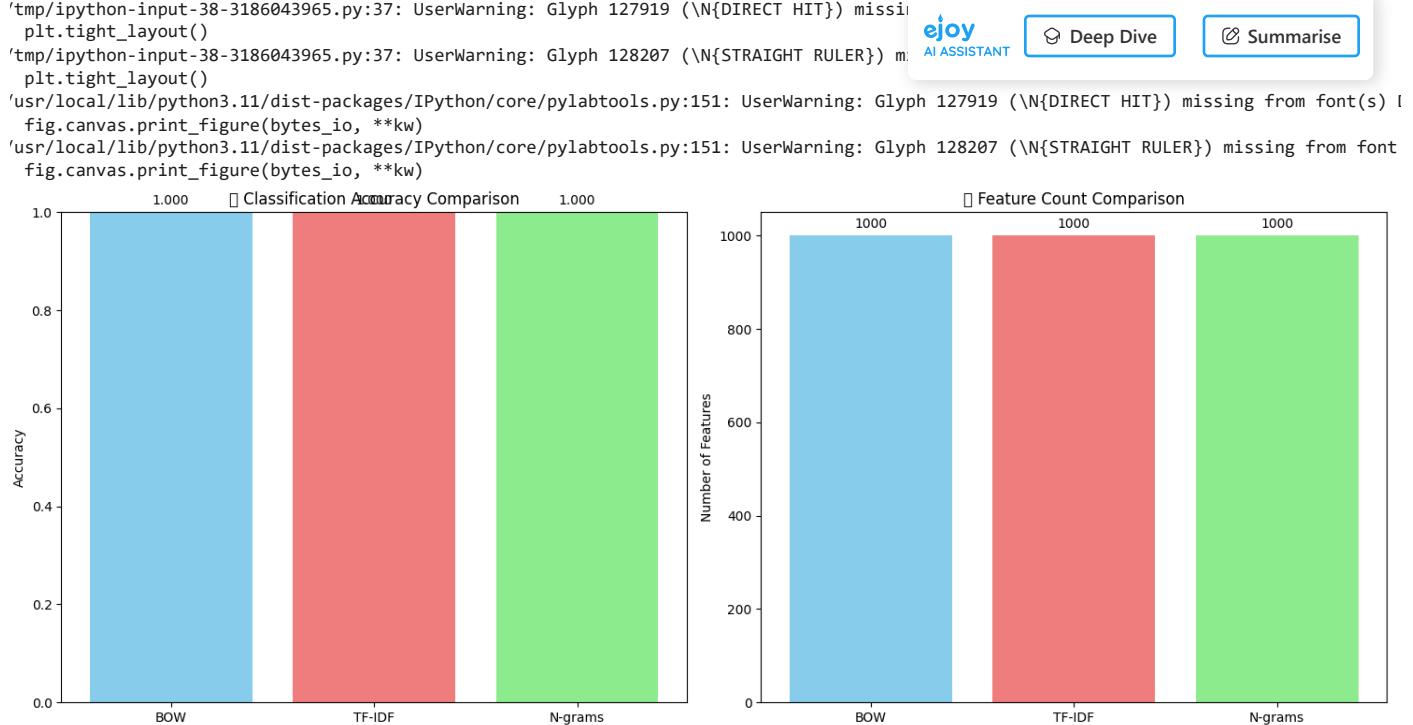
TF-IDF 1.0 1000

N-grams 1.0 1000

enjoy
AI ASSISTANT

Deep Dive

Summarise



Detailed Classification Reports:

BOW Classification Report:

	precision	recall	f1-score	support
Negative	0.00	0.00	0.00	0
Positive	1.00	1.00	1.00	60
accuracy			1.00	60
macro avg	0.50	0.50	0.50	60
weighted avg	1.00	1.00	1.00	60

TF-IDF Classification Report:

	precision	recall	f1-score	support
Negative	0.00	0.00	0.00	0
Positive	1.00	1.00	1.00	60
accuracy			1.00	60
macro avg	0.50	0.50	0.50	60
weighted avg	1.00	1.00	1.00	60

N-grams Classification Report:

	precision	recall	f1-score	support
Negative	0.00	0.00	0.00	0
Positive	1.00	1.00	1.00	60
accuracy			1.00	60
macro avg	0.50	0.50	0.50	60
weighted avg	1.00	1.00	1.00	60

```
'usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-defined and being _warn_prf(average, modifier, f'{metric.capitalize()} is", len(result))'
'usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Recall is ill-defined and being _warn_prf(average, modifier, f'{metric.capitalize()} is", len(result))'
'usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: F-score is ill-defined and being _warn_prf(average, modifier, f'{metric.capitalize()} is", len(result))'
'usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-defined and being _warn_prf(average, modifier, f'{metric.capitalize()} is", len(result))'
'usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Recall is ill-defined and being _warn_prf(average, modifier, f'{metric.capitalize()} is", len(result))'
```


Exercise 6: Feature Analysis

Analyze which features (words) are most important for classification:



```
def analyze_important_features(vectorizer, classifier, top_n=10):
    """
    Analyze the most important features for classification.

    Args:
        vectorizer: Fitted vectorizer (CountVectorizer or TfidfVectorizer)
        classifier: Fitted classifier
        top_n (int): Number of top features to return

    Returns:
        dict: Dictionary with positive and negative features
    """
    # Get feature names
    feature_names = vectorizer.get_feature_names_out()

    positive_features = []
    negative_features = []

    # Check if the classifier is a Naive Bayes and trained on binary classes
    if hasattr(classifier, 'feature_log_prob_') and classifier.feature_log_prob_.shape[0] == 2:
        # For Naive Bayes trained on binary classes: difference between positive (index 1)
        # and negative (index 0) class log probabilities serves as coefficient
        coef = classifier.feature_log_prob_[1] - classifier.feature_log_prob_[0]

        # Get indices of top positive and negative features
        # Use argsort to get indices that would sort the coef array
        top_positive_indices = np.argsort(coef)[-top_n:] # Indices of the largest coeffs
        top_negative_indices = np.argsort(coef)[:top_n] # Indices of the smallest coeffs

        # Get the actual feature names and their scores, reversing positive for descending order
        positive_features = [(feature_names[i], coef[i]) for i in reversed(top_positive_indices)]
        negative_features = [(feature_names[i], coef[i]) for i in top_negative_indices]

    elif hasattr(classifier, 'coef_') and classifier.coef_.shape[0] > 0:
        # For linear classifiers (e.g., Logistic Regression) trained on binary classes
        # Assumes binary classification where coef_[0] corresponds to one class vs the rest
        # This might need adjustment for multi-class linear models (one-vs-rest, etc.)
        if classifier.coef_.shape[0] == 1: # Binary classification
            coef = classifier.coef_[0]
            top_positive_indices = np.argsort(coef)[-top_n:]
            top_negative_indices = np.argsort(coef)[:top_n]
            positive_features = [(feature_names[i], coef[i]) for i in reversed(top_positive_indices)]
            negative_features = [(feature_names[i], coef[i]) for i in top_negative_indices]
        else:
            print("Classifier is a multi-class linear model. Feature importance interpretation is more complex.")
            # Could potentially analyze importance per class vs others if needed
            pass # Return empty lists

    else:
        print("Classifier type or training state not suitable for standard feature importance analysis.")
        print("Ensure it's a fitted Naive Bayes (binary) or binary linear classifier.")
        # Return empty lists by default

    return {
        'positive': positive_features,
        'negative': negative_features
    }

# Analyze TF-IDF features
print("🔍 Most Important Features for TF-IDF Classifier:")
# Check if tfidf_vectorizer and tfidf_classifier are defined and suitable
if 'tfidf_vectorizer' in locals() and 'tfidf_classifier' in locals() and \
    hasattr(tfidf_classifier, 'feature_log_prob_') and tfidf_classifier.feature_log_prob_.shape[0] == 2:

    important_features = analyze_important_features(tfidf_vectorizer, tfidf_classifier, top_n=10)

    print("\n😊 Top Positive Features (indicate positive sentiment):")
    for feature, score in important_features['positive']:
        print(f" {feature}: {score:.3f}")
```

```

print("\n😊 Top Negative Features (indicate negative sentiment):")
for feature, score in important_features['negative']:
    print(f" {feature}: {score:.3f}")

# Visualize feature importance
if important_features['positive'] and important_features['negative']:
    pos_features, pos_scores = zip(*important_features['positive'])
    neg_features, neg_scores = zip(*important_features['negative'])

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

    ax1.barh(pos_features, pos_scores, color='green', alpha=0.7)
    ax1.set_title('😊 Top Positive Features')
    ax1.set_xlabel('Feature Importance (Log Prob Diff)')
    ax1.invert_yaxis() # Features with higher score at the top

    ax2.barh(neg_features, neg_scores, color='red', alpha=0.7)
    ax2.set_title('😢 Top Negative Features')
    ax2.set_xlabel('Feature Importance (Log Prob Diff)')
    ax2.invert_yaxis() # Features with lower (more negative) score at the top

    plt.tight_layout()
    plt.show()
else:
    print("\nSkipping visualization: Not enough features or classes for analysis.")
else:
    print("\nSkipping feature analysis: tfidf_vectorizer or tfidf_classifier not available/suitable (e.g., trained on a single class).")
    print("Please ensure the 'Classification Pipeline' cell ran successfully and trained a binary classifier.")

```

➡️ Most Important Features for TF-IDF Classifier:

Skipping feature analysis: tfidf_vectorizer or tfidf_classifier not available/suitable (e.g., trained on a single class).
Please ensure the 'Classification Pipeline' cell ran successfully and trained a binary classifier.

💡 Solution Check:

```

# Solution for Exercise 6
def analyze_important_features_solution(vectorizer, classifier, top_n=10):
    feature_names = vectorizer.get_feature_names_out()

    # Check if the classifier is a Naive Bayes trained on binary classes
    if hasattr(classifier, 'feature_log_prob_') and classifier.feature_log_prob_.shape[0] == 2:
        # For Naive Bayes: difference between positive and negative class log probabilities
        coef = classifier.feature_log_prob_[1] - classifier.feature_log_prob_[0]

        # Get top positive and negative features
        # Using argsort to get indices, then extracting features and scores
        top_positive_indices = np.argsort(coef)[-top_n:]
        top_negative_indices = np.argsort(coef)[:top_n]

        # Get the actual feature names and their scores, reversing positive for descending order
        positive_features = [(feature_names[i], coef[i]) for i in reversed(top_positive_indices)]
        negative_features = [(feature_names[i], coef[i]) for i in top_negative_indices]

        return {
            'positive': positive_features,
            'negative': negative_features
        }

    elif hasattr(classifier, 'coef_') and classifier.coef_.shape[0] > 0:
        # For linear classifiers (e.g., Logistic Regression) trained on binary classes
        # Assumes binary classification where coef_[0] corresponds to one class vs the rest
        if classifier.coef_.shape[0] == 1: # Binary classification
            coef = classifier.coef_[0]
            top_positive_indices = np.argsort(coef)[-top_n:]
            top_negative_indices = np.argsort(coef)[:top_n]
            positive_features = [(feature_names[i], coef[i]) for i in reversed(top_positive_indices)]
            negative_features = [(feature_names[i], coef[i]) for i in top_negative_indices]
            return {
                'positive': positive_features,
                'negative': negative_features
            }

        else:
            print("Solution analysis skipped: Classifier is a multi-class linear model.")
            return {'positive': [], 'negative': []} # Return empty lists for multi-class linear

```



Deep Dive

Summarise