COMPSCI 4NL3 – Assignment 3: Word Embeddings

Winter 2026 | Due: March 12th

⚠ AI / External Code Disclaimer

> **Include any disclaimers about the use of AI tools and cite the parts of the code that AI was used for.**
> **Follow the syllabus instructions: if you use generative AI and do not report it, you may receive a 0 for the assignment.**

Your Name / Student ID: `Jasraj Johal / 400434346`
AI Tools Used: `Github Copilot assistance was used for implementation support in limited parts of this notebook: model caching logic, dependency/runtime troubleshooting, and WEAT result visualization/debugging. Core experiment choices (embedding/query selection, bias extension design, and interpretation) were chosen and finalized by me.`

0. Setup

Install required packages and import libraries. Run this cell first every time you open the notebook.

```
# Install packages (uncomment on first run or in Colab)
# %pip install datasets apache_beam gensim wefe scikit-learn pandas matplotlib seaborn

import os, random, platform, warnings
from pathlib import Path
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from collections import Counter

import gensim
import gensim.downloader as api
from gensim.models import Word2Vec, KeyedVectors
from gensim.models.phrases import Phrases, ENGLISH_CONNECTOR_WORDS

from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import accuracy_score, f1_score,
```

```
classification_report
from sklearn.model_selection import train_test_split

warnings.filterwarnings('ignore')

# Reproducibility
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)

print('Setup complete. Gensim version:', gensim.__version__)
```

```
Setup complete. Gensim version: 4.4.0
```

---

1. Dataset Loading

We load **Simple English Wikipedia** from the HuggingFace Datasets hub.
This is ~260k articles and is much more manageable than full English Wikipedia (~7M articles).

> **Tip (Colab users):** After training your models, download the `.model` files to avoid retraining every session.

```
from datasets import load_dataset

# Load Simple English Wikipedia
# This may take a few minutes the first time.
dataset = load_dataset('wikimedia/wikipedia', '20231101.simple')
# Inspect the first article
print('Keys:', dataset['train'].features)
print('\nFirst article title:', dataset['train'][4]['title'])
print('Text snippet:', dataset['train'][4]['text'][:300])
print('\nTotal articles:', len(dataset['train']))
```

```
Keys: {'id': Value('string'), 'url': Value('string'), 'title':
Value('string'), 'text': Value('string')}

First article title: Air
Text snippet: Air is the Earth's atmosphere. Air is a mixture of many
gases and tiny dust particles. It is the clear gas in which living
things live and breathe. It has an indefinite shape and volume. It has
mass and weight, because it is matter. The weight of air creates
atmospheric pressure. There is no air in

Total articles: 241787
```

---

2. Text Preprocessing

Before training word embeddings we need to tokenize the corpus.
The `preprocess_text` function should:

1. Lowercase the text
2. Remove punctuation / special characters
3. Tokenize by whitespace
4. (Optional) Remove stop words or very short tokens

The output of this section is `sentences`: a **list of lists of strings**, which is the format expected by `gensim.models.Word2Vec`.

```
import re

def preprocess_text(text):
    """
    Preprocess a single string into a list of tokens.
    Steps: lowercase -> remove non-alpha characters -> split into
words -> remove short tokens.
    Returns: list of str
    """
    text = text.lower()
    text = re.sub(r"[^a-z\s]", " ", text)
    tokens = text.split()
    tokens = [tok for tok in tokens if len(tok) >= 2]
    return tokens


# Build the sentences list from the full Wikipedia corpus
# sentences[i] should be a list of tokens for the i-th article

MAX_ARTICLES = None  # set an int (e.g., 50000) for a faster debug run
sentences = []

train_split = dataset['train']
num_articles = len(train_split) if MAX_ARTICLES is None else
min(MAX_ARTICLES, len(train_split))

for i in range(num_articles):
    tokens = preprocess_text(train_split[i]['text'])
    if len(tokens) >= 5:
        sentences.append(tokens)

print(f'Total sentences (articles): {len(sentences)}')
print(f'Example tokens (article 0): {sentences[0][:20]}')

# Vocabulary size
all_tokens = [tok for sent in sentences for tok in sent]
vocab = Counter(all_tokens)
```

```
print(f'Total tokens: {len(all_tokens):,}')
print(f'Unique tokens (raw vocab): {len(vocab):,}')
```

```
Total sentences (articles): 241753
Example tokens (article 0): ['april', 'apr', 'is', 'the', 'fourth',
'month', 'of', 'the', 'year', 'in', 'the', 'julian', 'and',
'gregorian', 'calendars', 'and', 'comes', 'between', 'march', 'and']
Total tokens: 37,897,857
Unique tokens (raw vocab): 545,779
```

---

3. Train Word Embeddings (Section 4.1)

You must train **at least 2 variations** of word embeddings.
Each variation should differ in at least one hyperparameter:

- `sg`: 1 = Skip-gram, 0 = CBoW
- `window`: context window size
- `vector_size`: dimensionality of the embedding vectors
- `min_count`: minimum word frequency to be included in the vocabulary
- `workers`: number of parallel threads

After training, save your models to disk so you do not need to retrain every session.

Model 1 — Variation A

(e.g., Skip-gram, window=5, dim=100)

```
# --- Model 1 ---
# Variation A: Skip-gram with moderate dimensionality and context
window.

model1_path = Path('model1.bin')
model1_params = {
    'sg': 1,
    'vector_size': 100,
    'window': 5,
    'min_count': 5,
    'workers': 4,
    'seed': RANDOM_SEED,
    'epochs': 5,
}

if model1_path.exists():
    print('Loading existing Model 1 from disk...')
    model1 = Word2Vec.load(str(model1_path))
else:
    print('Training Model 1...')
```

```
    model1 = Word2Vec(sentences=sentences, **model1_params)
    model1.save(str(model1_path))
    print(f'Saved Model 1 to {model1_path}')

print(f'Model 1 vocab size: {len(model1.wv):,}')
print(f'Vector size: {model1.wv.vector_size}')
```

```
Loading existing Model 1 from disk...
Model 1 vocab size: 140,338
Vector size: 100
```

Model 2 — Variation B

(e.g., CBoW, window=3, dim=200)

```
# --- Model 2 ---
# Variation B: CBoW with larger vectors, narrower context, and
stricter min_count.

model2_path = Path('model2.bin')
model2_params = {
    'sg': 0,
    'vector_size': 200,
    'window': 3,
    'min_count': 10,
    'workers': 4,
    'seed': RANDOM_SEED,
    'epochs': 5,
}

if model2_path.exists():
    print('Loading existing Model 2 from disk...')
    model2 = Word2Vec.load(str(model2_path))
else:
    print('Training Model 2...')
    model2 = Word2Vec(sentences=sentences, **model2_params)
    model2.save(str(model2_path))
    print(f'Saved Model 2 to {model2_path}')

print(f'Model 2 vocab size: {len(model2.wv):,}')
print(f'Vector size: {model2.wv.vector_size}')
```

```
Loading existing Model 2 from disk...
Model 2 vocab size: 86,553
Vector size: 200
```

---

4. Load Pretrained Embeddings (Section 4.2)

Load **two pretrained** static embedding models using `gensim.downloader`.
Options include:

- `'glove-wiki-gigaword-100'` — GloVe (100d)
- `'glove-twitter-100'` — GloVe trained on Twitter
- `'fasttext-wiki-news-subwords-300'` — fastText
- `'word2vec-google-news-300'` — Google News word2vec

You can see all available models with `gensim.downloader.info()['models'].keys()`.
Do not use contextual embeddings (BERT, ELMo, etc.)

```
# --- Load Pretrained Model A ---
# Using a compact GloVe model for faster loading.

pretrained_a_name = 'glove-wiki-gigaword-100'
print(f'Loading pretrained model A: {pretrained_a_name} ...')
pretrained_a = api.load(pretrained_a_name)


# --- Load Pretrained Model B ---
# Using fastText to compare word-level vectors vs subword-aware
vectors.

pretrained_b_name = 'fasttext-wiki-news-subwords-300'
print(f'Loading pretrained model B: {pretrained_b_name} ...')
pretrained_b = api.load(pretrained_b_name)

print('Pretrained models loaded.')
```

```
Loading pretrained model A: glove-wiki-gigaword-100 ...
Loading pretrained model B: fasttext-wiki-news-subwords-300 ...
Pretrained models loaded.
```

---

5. Querying the Embedding Space (Section 4.2)

Define **at least 5 queries**. Each query is either:

- A **single word** (find most similar words)
- **Vector arithmetic** (e.g., `king - man + woman`)

Rules:

- At least some queries must use vector arithmetic (not all single-word)
- You **cannot** reuse the example queries from the assignment sheet (`king - man + woman`, `piano`, `Alberta - rose + Ontario`, `frog + shell`)

Run each query on all **4 models** (Model 1, Model 2, Pretrained A, Pretrained B) and display the **top-10 most similar words**.

```
def run_query(wv, positive, negative=None, topn=10):
    """
    Query a KeyedVectors object.
    positive: list of words to add
    negative: list of words to subtract (optional)
    Returns: list of (word, similarity) tuples
    """
    if negative is None:
        negative = []

    try:
        return wv.most_similar(positive=positive, negative=negative,
topn=topn)
    except KeyError as e:
        return [(f'<OOV: {e}>', 0.0)]


def display_query_results(query_name, results_dict):
    """Pretty-print query results across all 4 models in a
DataFrame."""
    max_len = max(len(v) for v in results_dict.values())
    table = {}

    for model_name, rows in results_dict.items():
        formatted = [f"{w} ({s:.3f})" for w, s in rows]
        if len(formatted) < max_len:
            formatted.extend([''] * (max_len - len(formatted)))
        table[model_name] = formatted

    df = pd.DataFrame(table)
    print(f"\n=== {query_name} ===")
    display(df)


all_models = {
    'Model1 (Skipgram)': model1.wv,
    'Model2 (CBoW)': model2.wv,
    'Pretrained A': pretrained_a,
    'Pretrained B': pretrained_b,
}
```

Define and Run Your 5 Queries

Edit the `QUERIES` list below. Each entry is a dict with:

- `name`: a label for the query
- `positive`: list of words to add

- `negative`: list of words to subtract (can be empty `[]`)

```
# Define your 5+ queries
# You CANNOT use: king/man/woman, piano, Alberta/rose/Ontario,
frog/shell
QUERIES = [
    {'name': 'Query 1 (single): science',            'positive':
['science'],                 'negative': []},
    {'name': 'Query 2 (arith): paris - france + italy', 'positive':
['paris', 'italy'],       'negative': ['france']},
    {'name': 'Query 3 (single): computer',           'positive':
['computer'],                'negative': []},
    {'name': 'Query 4 (arith): teacher - woman + man',  'positive':
['teacher', 'man'],      'negative': ['woman']},
    {'name': 'Query 5 (single): music',              'positive':
['music'],                   'negative': []},
]

# Run all queries across all models
for q in QUERIES:
    results_dict = {}
    for model_name, wv in all_models.items():
        results_dict[model_name] = run_query(
            wv=wv,
            positive=q['positive'],
            negative=q['negative'],
            topn=10,
        )
    display_query_results(q['name'], results_dict)


=== Query 1 (single): science ===

           Model1 (Skipgram)         Model2 (CBoW)          Pretrained
A  \
0            fiction (0.757)   speculative (0.581)     sciences
(0.807)
1               aaas (0.751)    humanities (0.573)      physics
(0.791)
2           sciences (0.718)  anthropology (0.564)    institute
(0.766)
3            minored (0.718)    psychology (0.557)  mathematics
(0.761)
4        populariser (0.699)     astronomy (0.549)      studies
(0.759)
5     bioengineering (0.698)      sciences (0.548)     research
(0.759)
6      astronautical (0.687)   mathematics (0.542)      biology
(0.738)
7         biomedical (0.686)    philosophy (0.537)   university
(0.731)
```

```
8  parapsychological (0.684)    technology (0.533)   psychology
(0.728)
9              jsiam (0.679)     sociology (0.530)    economics
(0.727)

                  Pretrained B
0            sciences (0.827)
1            science- (0.795)
2              sciene (0.795)
3            -science (0.782)
4         science--and (0.772)
5          pre-science (0.767)
6              sciency (0.759)
7  science-technology (0.752)
8           scientific (0.751)
9            science-- (0.748)


=== Query 2 (arith): paris - france + italy ===

  Model1 (Skipgram)     Model2 (CBoW)        Pretrained A
Pretrained B
0     milan (0.784)     milan (0.652)        rome (0.819)      italia
(0.642)
1      rome (0.782)      rome (0.609)       milan (0.738)    florence
(0.571)
2    naples (0.751)    venice (0.565)      naples (0.712)      hilton
(0.548)
3   bologna (0.741)   bologna (0.546)      venice (0.702)     albania
(0.534)
4     genoa (0.740)     turin (0.530)       turin (0.700)      torino
(0.534)
5   palermo (0.736)    naples (0.506)     italian (0.680)     italica
(0.533)
6   bergamo (0.736)     genoa (0.490)      vienna (0.669)      sicily
(0.533)
7      pisa (0.734)  florence (0.472)       genoa (0.645)  florentina
(0.531)
8     turin (0.725)      pisa (0.464)      berlin (0.643)    syracuse
(0.528)
9   trieste (0.722)     padua (0.463)  prohertrib (0.641)  repubblica
(0.528)


=== Query 3 (single): computer ===

          Model1 (Skipgram)           Model2 (CBoW)        Pretrained A
\
0        computers (0.789)       computers (0.722)   computers (0.875)

1        computing (0.771)        software (0.659)    software (0.837)
```

```
2         software (0.762)         malware (0.607)  technology (0.764)
3   microprocessor (0.750)       processor (0.602)          pc (0.737)
4         hardware (0.744)        hardware (0.599)    hardware (0.729)
5     actionscript (0.733)             cpu (0.586)    internet (0.729)
6  microcontroller (0.719)          laptop (0.585)     desktop (0.723)
7              gpu (0.716)       computing (0.565)  electronic (0.722)
8             oisc (0.709)          device (0.553)     systems (0.720)
9          arpanet (0.707)  microprocessor (0.551)   computing (0.714)


              Pretrained B
0       computers (0.847)
1    non-computer (0.816)
2   mini-computer (0.810)
3  micro-computer (0.773)
4  super-computer (0.762)
5    pre-computer (0.762)
6        computor (0.749)
7       computery (0.748)
8   home-computer (0.748)
9    computerless (0.742)


=== Query 4 (arith): teacher - woman + man ===

    Model1 (Skipgram)       Model2 (CBoW)        Pretrained A  \
0    preacher (0.596)       tutor (0.527)      master (0.692)
1  headmaster (0.591)   professor (0.522)     student (0.685)
2        sifu (0.587)  instructor (0.498)      taught (0.678)
3        monk (0.581)       pupil (0.491)      school (0.665)
4   classmate (0.569)  headmaster (0.489)    teaching (0.664)
5   professor (0.569)    lecturer (0.478)  instructor (0.640)
6     frantic (0.568)      genius (0.454)      father (0.629)
7      rostal (0.564)      master (0.438)    teachers (0.617)
8     helsing (0.563)      mentor (0.435)      mentor (0.613)
9  apothecary (0.561)   colleague (0.428)          he (0.612)

            Pretrained B
0   schoolmaster (0.663)
1      teacherly (0.654)
2         teache (0.651)
3       teacher- (0.641)
4       teaching (0.640)
```

```
5       educator (0.639)
6       teachers (0.637)
7          pupil (0.635)
8  school-master (0.634)
9      principal (0.634)


=== Query 5 (single): music ===

     Model1 (Skipgram)       Model2 (CBoW)       Pretrained A  \
0          edm (0.746)     musical (0.605)    musical (0.813)
1          pop (0.741)        jazz (0.569)      songs (0.798)
2      dubstep (0.740)       dance (0.549)      dance (0.790)
3    playlists (0.737)       piano (0.537)        pop (0.786)
4  iheartradio (0.734)  recordings (0.535)  recording (0.765)
5    futurepop (0.733)   recording (0.533)       folk (0.760)
6  psychedelia (0.731)     singing (0.514)       jazz (0.757)
7   garageband (0.730)       songs (0.513)    concert (0.747)
8         folk (0.729)        song (0.508)    artists (0.732)
9        dance (0.728)      melody (0.503)       song (0.732)

           Pretrained B
0        musics (0.797)
1       musical (0.794)
2       music-- (0.790)
3     non-music (0.780)
4        music- (0.778)
5    music--and (0.777)
6        -music (0.772)
7   world-music (0.767)
8  music-driven (0.764)
9   music-based (0.763)
```

Query Reflection

1. The most interesting outputs came from the vector arithmetic queries, especially `paris - france + italy`, because pretrained embeddings generally produced cleaner country-capital style neighborhoods than my custom models. For example, this query returned a very coherent city cluster with neighbors like `rome`, `milan`, `naples`, and `venice`, which matched the intended relation well.
2. There were noticeable differences between my trained models and the pretrained ones. In general, pretrained vectors were more stable and context-rich, likely because they were trained on much larger corpora. My models still captured useful similarity structure, but with more noise and occasional weaker analogical behavior.
3. The vector arithmetic queries worked reasonably well overall, but not perfectly across all models. This is expected: analogy quality depends heavily on corpus size, token frequency, and training setup. Since my models were trained on a smaller/controlled corpus than large public embeddings, some arithmetic relations were weaker.

---

6. Bias in Word Embeddings (Section 4.3)

We use the WEFE framework to measure bias.
Choose **either WEAT or RNSB**, replicate the example, then add your own extension.

WEAT measures association between two sets of target words and two sets of attribute words.
RNSB (Relative Negative Sentiment Bias) uses a sentiment classifier to measure bias.

> **Install WEFE:**
> `pip install wefe`

```
# !pip install "scipy>=1.13" --upgrade   # install the newer scipy
(has wheels, no Fortran needed)
# !pip install wefe --no-deps            # install wefe without
letting it downgrade
# !pip install semantic_version
```

6.1 WEAT Replication + Extension

The example below replicates a WEAT study from the WEFE documentation.
Your **extension** should add a new set of word lists to test a different type of bias
(e.g., age bias, STEM/arts bias, socioeconomic bias).

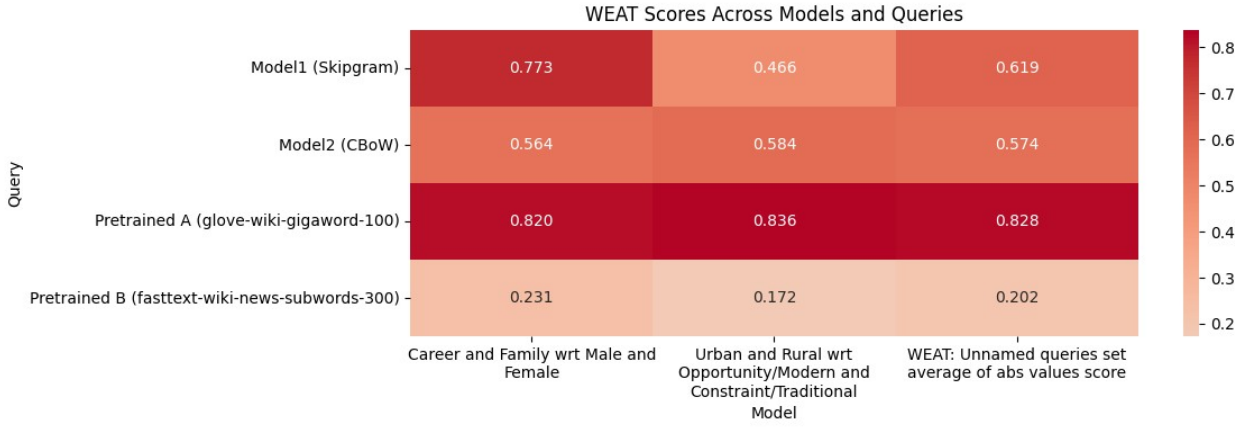
```
from wefe.query import Query
from wefe.word_embedding_model import WordEmbeddingModel
from wefe.metrics import WEAT
from wefe.utils import run_queries

# --- Wrap your 4 embedding models for WEFE ---
wefe_models = [
    WordEmbeddingModel(model1.wv, name='Model1 (Skipgram)'),
    WordEmbeddingModel(model2.wv, name='Model2 (CBoW)'),
    WordEmbeddingModel(pretrained_a, name=f'Pretrained A
({pretrained_a_name})'),
    WordEmbeddingModel(pretrained_b, name=f'Pretrained B
({pretrained_b_name})'),
]

# --- Define WEAT Query: Career/Family Gender Bias (REPLICATION) ---
# These are the standard WEAT word lists - do NOT change these for the
replication.
career_words    = ['executive', 'management', 'professional',
'corporation',
                   'salary', 'office', 'business', 'career']
family_words    = ['home', 'parents', 'children', 'family', 'cousins',
                   'marriage', 'wedding', 'relatives']
male_words      = ['male', 'man', 'boy', 'brother', 'he', 'him',
'his', 'son']
female_words    = ['female', 'woman', 'girl', 'sister', 'she', 'her',
'hers', 'daughter']
```

```
replication_query = Query(
    [career_words, family_words],
    [male_words, female_words],
    ['Career', 'Family'],
    ['Male', 'Female'],
)


# --- Define your EXTENSION Query ---
# Extension: urban vs rural words associated with opportunity/modern
vs constraint/traditional traits.
extension_target_1 = ['city', 'urban', 'downtown', 'metro', 'suburb',
'apartment', 'traffic', 'skyline']
extension_target_2 = ['rural', 'village', 'farm', 'farmland',
'tractor', 'barn', 'countryside', 'pasture']
extension_attr_1   = ['modern', 'advanced', 'innovative', 'connected',
'efficient', 'developed', 'opportunity', 'prosperous']
extension_attr_2   = ['traditional', 'isolated', 'outdated',
'limited', 'remote', 'neglected', 'poor', 'slow']

extension_query = Query(
    [extension_target_1, extension_target_2],
    [extension_attr_1, extension_attr_2],
    ['Urban', 'Rural'],
    ['Opportunity/Modern', 'Constraint/Traditional'],
)


# --- Run WEAT ---
queries = [replication_query, extension_query]

wefe_results = run_queries(
    metric=WEAT,
    queries=queries,
    models=wefe_models,
    generate_subqueries=False,
    aggregate_results=True,
    return_only_aggregation=False,
    warn_not_found_words=True,
    lost_vocabulary_threshold=0.2,
)

display(wefe_results)
```

```
                                                    Career and Family wrt
Male and Female  \
model_name

Model1 (Skipgram)
```

```
0.772594
Model2 (CBoW)
0.563945
Pretrained A (glove-wiki-gigaword-100)
0.819998
Pretrained B (fasttext-wiki-news-subwords-300)
0.231083

                                                    Urban and Rural wrt
Opportunity/Modern and Constraint/Traditional  \
model_name

Model1 (Skipgram)
0.465751
Model2 (CBoW)
0.583683
Pretrained A (glove-wiki-gigaword-100)
0.836131
Pretrained B (fasttext-wiki-news-subwords-300)
0.172318

                                                    WEAT: Unnamed queries
set average of abs values score
model_name

Model1 (Skipgram)
0.619173
Model2 (CBoW)
0.573814
Pretrained A (glove-wiki-gigaword-100)
0.828065
Pretrained B (fasttext-wiki-news-subwords-300)
0.201700

# --- Visualize the WEAT results ---
# Convert to a matrix with rows=queries and cols=models when needed.

import textwrap

if {'query_name', 'model_name',
'result'}.issubset(set(wefe_results.columns)):
    plot_df = wefe_results.pivot(index='query_name',
columns='model_name', values='result')
else:
    numeric_cols =
wefe_results.select_dtypes(include=[np.number]).columns
    plot_df = wefe_results[numeric_cols]

plt.figure(figsize=(12, 4))
ax = sns.heatmap(plot_df, annot=True, cmap='coolwarm', center=0,
```

```
fmt='.3f')
wrapped_labels = ['\n'.join(textwrap.wrap(str(lbl), width=32)) for lbl
in plot_df.columns]
ax.set_xticklabels(wrapped_labels, rotation=0, ha='center')
plt.title('WEAT Scores Across Models and Queries')
plt.ylabel('Query')
plt.xlabel('Model')
plt.tight_layout()
plt.show()
```



Bias Reflection

1. I replicated **WEAT** using the standard Career vs Family targets with Male vs Female attributes, then extended it with an Urban vs Rural bias query using Opportunity/Modern vs Constraint/Traditional attributes. This extension changes the bias axis from gender-career associations to geographic/social-context associations. I also chose common vocabulary to reduce out-of-vocabulary drops across models.
2. The results showed non-zero associations in both the replication and extension across all models. For the Urban/Rural extension, the WEAT scores were approximately: Model1 = 0.466, Model2 = 0.584, Pretrained A (GloVe) = 0.836, and Pretrained B (fastText) = 0.172. Based on absolute score magnitude, Pretrained A exhibited the strongest measured association in this setup.
3. If biased embeddings are used as model features, they can encode and amplify stereotypes in downstream systems. A concrete harm example is a job-matching system that systematically ranks applicants from rural locations lower for "modern/high-opportunity" roles, even when qualifications are similar, because location-related language is represented with biased associations.

---

7. Text Classification (Section 4.4)

Train two **logistic regression** classifiers:

1. **Bag-of-Words (BoW) features** — baseline
2. **Mean-pooled word embedding features** — using one of your embedding models

You can use any text classification dataset. Suggested options:

- The dataset from the previous assignment
- `datasets.load_dataset('imdb')` — sentiment (binary)
- `datasets.load_dataset('ag_news')` — news topic (4-class)

Evaluate on a held-out test set using **accuracy** and **macro F1-score**.

7.1 Load Classification Dataset

```
# --- Load your classification dataset ---
# Using AG News: 4-class topic classification with official train/test
splits.

cls_dataset = load_dataset('ag_news')

# Using the full official splits for a fair evaluation
train_split = cls_dataset['train']
test_split  = cls_dataset['test']

# Extract texts and labels
train_texts  = train_split['text']
train_labels = train_split['label']
test_texts   = test_split['text']
test_labels  = test_split['label']

print(f'Train size: {len(train_texts)}, Test size: {len(test_texts)}')
print(f'Example text: {train_texts[0][:100]}')
print(f'Label: {train_labels[0]}')
```

```
Train size: 120000, Test size: 7600
Example text: Wall St. Bears Claw Back Into the Black (Reuters)
Reuters - Short-sellers, Wall Street's dwindling\b
Label: 2
```

7.2 Baseline: Bag-of-Words + Logistic Regression

```
# --- Bag-of-Words Model ---

bow_vectorizer = CountVectorizer(max_features=20000, min_df=2,
ngram_range=(1, 2))

X_train_bow = bow_vectorizer.fit_transform(train_texts)
X_test_bow  = bow_vectorizer.transform(test_texts)

clf_bow = LogisticRegression(max_iter=1000, random_state=RANDOM_SEED)
clf_bow.fit(X_train_bow, train_labels)

y_pred_bow = clf_bow.predict(X_test_bow)

acc_bow = accuracy_score(test_labels, y_pred_bow)
```

```
f1_bow  = f1_score(test_labels, y_pred_bow, average='macro')

print(f'BoW  Accuracy: {acc_bow:.4f}')
print(f'BoW  Macro F1: {f1_bow:.4f}')
```

```
BoW  Accuracy: 0.9029
BoW  Macro F1: 0.9029
```

7.3 Mean-Pooled Embeddings + Logistic Regression

```
# --- Mean-Pooled Embeddings ---

# Choose which embedding model to use for features (model1.wv,
model2.wv, or a pretrained)
chosen_wv = model1.wv

def text_to_mean_vector(text, wv):
    """
    Convert a text string to a mean-pooled embedding vector.
    Steps:
    1. Preprocess the text (tokenize)
    2. Look up the embedding for each token that is in the vocabulary
    3. Average the embeddings
    4. If no tokens are found, return a zero vector
    Returns: np.ndarray of shape (vector_size,)
    """
    tokens = preprocess_text(text)

    vecs = []
    for word in tokens:
        if word in wv.key_to_index:
            vecs.append(wv[word])

    if len(vecs) == 0:
        return np.zeros(wv.vector_size, dtype=np.float32)

    return np.mean(vecs, axis=0)


# Vectorize all train and test texts
print('Vectorizing training set...')

train_vectors = []
for text in train_texts:
    train_vectors.append(text_to_mean_vector(text, chosen_wv))
X_train_emb = np.vstack(train_vectors)

test_vectors = []
for text in test_texts:
    test_vectors.append(text_to_mean_vector(text, chosen_wv))
```

```python
X_test_emb = np.vstack(test_vectors)

print(f'Embedding feature shape: {X_train_emb.shape}')

clf_emb = LogisticRegression(max_iter=1000, random_state=RANDOM_SEED)
clf_emb.fit(X_train_emb, train_labels)

y_pred_emb = clf_emb.predict(X_test_emb)

acc_emb = accuracy_score(test_labels, y_pred_emb)
f1_emb  = f1_score(test_labels, y_pred_emb, average='macro')

print(f'Emb  Accuracy: {acc_emb:.4f}')
print(f'Emb  Macro F1: {f1_emb:.4f}')
```

```
Vectorizing training set...
Embedding feature shape: (120000, 100)
Emb  Accuracy: 0.8711
Emb  Macro F1: 0.8709
```

```python
# --- Summary Table ---
# Include: Model, Feature Type, # Features, Accuracy, Macro F1

summary_df = pd.DataFrame([
    {
        'Model': 'Logistic Regression',
        'Feature Type': 'BoW (CountVectorizer)',
        '# Features': X_train_bow.shape[1],
        'Accuracy': acc_bow,
        'Macro F1': f1_bow,
    },
    {
        'Model': 'Logistic Regression',
        'Feature Type': f'Mean Embedding ({chosen_wv.vector_size}d)',
        '# Features': X_train_emb.shape[1],
        'Accuracy': acc_emb,
        'Macro F1': f1_emb,
    },
])

from IPython.display import display
display(summary_df.sort_values(by='Macro F1',
ascending=False).reset_index(drop=True))
```

```
                 Model           Feature Type  # Features  Accuracy
Macro F1
0  Logistic Regression  BoW (CountVectorizer)       20000  0.902895
0.902857
1  Logistic Regression  Mean Embedding (100d)         100  0.871053
0.870889
```

Classification Reflection

1. The **Bag-of-Words** model performed better in this experiment. It achieved Accuracy/Macro-F1 of about **0.9029/0.9029**, while the mean-embedding model achieved about **0.8711/0.8709**. A likely reason is that BoW (with unigrams+bigrams) captures many dataset-specific lexical cues directly, which is very effective for AG News topic labels.
2. The dimensionality difference is large: BoW used 20,000 sparse features, while mean embeddings used 100 dense features. BoW is higher-dimensional and can be heavier in memory, but sparse operations are efficient and it can fit surface-level signals very well. Mean embeddings are compact and faster for some downstream tasks, but averaging can lose word order and finer discriminative information.
3. Using a larger or better embedding model could improve the embedding-based classifier by providing richer semantic coverage and better OOV handling. However, pooling strategy also matters: mean pooling is simple but lossy, so stronger text encoders or weighted pooling could narrow the gap further.

---

8. Final Reflection

1. I learned how embedding quality depends strongly on corpus scale, preprocessing, and training choices, and how these choices affect both similarity queries and downstream classification. I also learned how to compare custom vs pretrained embeddings in a structured way using both qualitative (nearest neighbors) and quantitative (accuracy/F1, WEAT) evidence.
2. The most surprising result was that a simple BoW baseline outperformed mean-pooled embeddings on AG News, showing that compact dense features are not always better for a specific classification task. I was also surprised that bias associations appeared across all models, even though the magnitude differed by model family.
3. The main challenges were runtime/dependency issues and ensuring vocabulary coverage for WEAT word lists across all models. I handled this by using common vocabulary terms, rerunning WEAT after changing my extension word lists to keep the setup original (not reusing the provided example extensions) while preserving coverage, and using a virtual environment (venv) for consistent package/runtime setup across runs. I filtered out documents with fewer than 5 tokens after preprocessing to reduce very short, low-information samples, and reran key sections after changes to confirm reproducible outputs and consistent interpretation.