Natural Language Processing 2025-1A Homework 3

Vector Semantics, Word2Vec and LLMs

Deadline: 1 October (23:59)

Questions: Post them in the homework discussion on Canvas, sent them to nlp-course@utwente.nl or ask us during the practical sessions.

How to submit: Please answer the questions directly in this notebook and submit it before the deadline.

Please Write your group number, your names with student IDs Here: (double click here!)

Make sure that the following libraries are up-to-date in your computation envrionment. It is highly recommended to work on this assignment in UT's JupyterLab.

In [ ]:
```
!python -m pip install --upgrade pip
!pip3 install gensim nltk scikit-learn numpy pandas scipy
!pip install  --upgrade gensim nltk scikit-learn numpy pandas scipy ### U
```

We'll need these libraries later.

In [ ]:
```
import pandas, numpy, scipy, math
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectori
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import nltk
nltk.download('stopwords')
#nltk.download('punkt')
nltk.download('punkt_tab')
from sklearn.metrics.pairwise import cosine_similarity
from gensim.test.utils import datapath
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
import codecs
import time
```

In this assignment, you will first explore two types of word vectors: those generated using co-occurrence–based methods and those produced by the local-context predictive model Word2Vec. You will then apply and evaluate an NLP task powered by a Large Language Model (LLM).

Note on Terminology:

- The terms "word" and "term" are used interchangeably in this context, referring to unique tokens that you aim to represent as vectors. These tokens can be individual words, n-grams, phrases, or even identifiers, but for this assignment, we will focus on individual words.
- Though "word vectors" and "word embeddings" are often used synonymously, they have distinct meanings. According to Wikipedia, conceptually, word embedding "*involves the mathematical embedding from space with many dimensions per word to a continuous vector space with a much lower dimension*".

Part I. Co-occurrence count-based vectors

Let's start with this corpus consisting of 10 sentences.

In [ ]:
```
sents=[
    "The warm sun melts the icy snow on the mountain.",
    "A warm cup of tea felt perfect in the cool morning air.",
    "Her warm smile brightened the cold winter day.",
    "I love the contrast of a warm blanket on a cold night.",
    "The cold wind chilled me, but the warm fire offered comfort.",
    "After a cold swim, the warm towel felt like heaven.",
    "The warm colors of the sunset clashed with the cold breeze.",
    "The chilly floor left her longing for the cozy comfort of slippers."
    "A gentle breeze eased the bite of the cold ocean waves.",
    "Cold hands found solace in the warm pockets of his coat."
]
```

Exercise 1.1.1 Construct the vocabulary (0.5 point)

Before we construct co-occurrence matrices, we need to identify unique terms in the corpus, i.e. construct the vocabulary. You can remove stop words and apply other text normalisation operations before constructing the vocabulary.

Tip: Sort your vocabulary alphabetically!

In [ ]:
```
# Collect unique terms in the corpus

# your code starts here



# your code ends here

print('The size of the vocabulary is', len(vocab))
print('The word in the vocabulary are', vocab)
```

Co-Occurrence

A co-occurrence matrix counts how often terms co-occur in certain context. The context can be a complete document, a sentence, or a sliding window.

Tip: Check out the sklearn.feature_extraction.text submodule that gathers utilities to build feature vectors from text documents.

Exercise 1.1.2 Term-document occurrence matrix and term-term co-occurrence matrix (0.5 point)

Let's first consider **each sentence** in the above corpus to be the context where the (co-)occurrences are counted. For example, the words *warm*, *sun*, *icy* and *snow* occur in the first sentence, therefore, they occur in this document and co-occur with each other. Going through all the sentences, you can construct the term-document occurrence matrix and term-term co-occurrence matrix.

In [ ]:
```
# Construct the term-document occurrence matrix

# your code starts here


# your code ends here

print('The shape of the term-document matrix is', tdMatrix.shape)
tdMatrix_pd = pandas.DataFrame(tdMatrix, index=vocab, columns=list(range(
tdMatrix_pd
```

The term–term co-occurrence matrix can be computed directly from the term–document occurrence matrix. When doing so, pay close attention to the diagonal entries — they indicate self-co-occurrences, which may need to be removed or adjusted depending on your application.

In [ ]:
```
# Construct the term-term co-occurrence matrix
# Be sure to handle the diagonal elements appropriately

# your code starts here



# your code ends here
print('The shape of the term-term matrix is', ttMatrix.shape)
ttMatrix_pd = pandas.DataFrame(ttMatrix, index=vocab, columns=vocab)
ttMatrix_pd
```

Based on term-term co-occurrence matrix, which pair(s) of words co-occur the most?

YOUR ANSWER:

Exercise 1.2 Cosine similarity

The benefit of vector semantics is that the similarity of two words can be computed as the cosine similarity between their vectors. Let's now compare how similar two words are.

Exercise 1.2.1 Calculate cosine similarity between words (0.5 point)

What is the cosine similarity between "cold" and "warm" if 1) using term-document occurrence matrix 2) using term-term co-occurrence matrix?

You may write your own cosine similarity function or use `sklearn.metrics.pairwise.cosine_similarity` to calculate the pair-wise cosine similarity among all the words.

In [ ]:
```
# Calculate the cosine similarity between "cold" and "ward" using 1) usin
# and 2) term-term co-occurrence matrix

# your code starts here



# your code ends here
```

Exercise 1.2.2 (0.5 point)

Now we can calculate cosine similarity between words using a co-occurrence matrix. You can choose any previously constructed matrix for the similarity calculation. Rank all the words based on their similarity to the word *cold*.

In [ ]:
```
# Rank all the words by their similarity to word "cold"

# your code starts here



# your code ends here
```

The calculated cosine similarity does not appear to capture semantic similarity or relatedness reliably. How might we obtain more meaningful similarity measures?

YOUR ANSWER:

Exercise 1.3 TF-IDF

Excercise 1.3.1 (0.5 point)

For the above corpus, construct a TF-IDF weighted term-document matrix, using `sklearn.feature_extraction.text.TfidfVectorizer`.

In [ ]:
```
# Construct a TF-IDF weighted term-document matrix

# your code starts here



```

```
# your code ends here
```

Compute and rank the words in descending order based on their similarity to *cold*.

In [ ]:
```
# Compute and rank the words in descending order based on their similarit

# Your code starts here


# Your code ends here
```

Exercise 1.3.2 (0.5 point)

Let's use a bigger dataset which contains 2225 BBC news articles to construct TF-IDF term-document matrix.

In [ ]:
```
sents=codecs.open('bbc-text.csv','r', encoding='utf-8').readlines() # loa

# your code starts here


# your code ends here

print('The size of the vocabulary is', len(vocab))
print('The shape of the term-document matrix is', term_doc_matrix.shape)
```

We can compute which words are most similar to *cold*. Does this list of words make more sense now and why?

YOUR ANSWER:

In [ ]:
```
# Find the top 10 words that are most similar to word "cold"

# your code starts here


# your code ends here
```

Find another 3 pairs of words whose cosine similarity makes sense to you.

In [ ]:
```
# Look for 3 pairs of words whose cosine similarities reflect their seman
```

Part II. Word2Vec word vectors

Here, we explore the embeddings produced by word2vec. Please read J&M 6.8 or the original paper if you are interested in the details of the algorithm.

Exercise 2.1 Pre-train word2vec model

Run the following script to load the word2vec vectors into memory. **Note**: This might take several minutes. If you run out of memory, try closing other applicaions or restart your machine to free more memory.

Please note, the following experiments run with Gensim 4.3.3. If you are still running an old version of Gensim, please upgrade your Gensim library or check Migrating from Gensim 3.x to 4 to adapt your code.

In [ ]:
```
# Load 3 million Word2Vec Vectors, pre-trained on Google news, each with
# This model may take a few minutes to load.

import gensim.downloader as api
start_time = time.time()
w2v_google = api.load("word2vec-google-news-300")
print("--- %s seconds ---" % (time.time() - start_time))
```

In [ ]:
```
print("Loaded vocab size {}".format(len(w2v_google.index_to_key)))
```

Once the model is loaded, you can extract the vector for individual words directly using `wv_google['']`

In [ ]:
```
w2v_google['cold']
```

One of the property of semantic embedding is that similar words are embedded close to each other. Use `w2v_google.most_similar()` to identify the most similar words to *north*. Does this list make more sense to you?

In [ ]:
```
start_time = time.time()
for w,c in w2v_google.most_similar('cold'):
    print(w,c)
print("--- %s seconds ---" % (time.time() - start_time))
```

Check a few more words to see whether their most similar words make sense to you and explain why.

In [ ]:
```
w2v_google.most_similar('black')
```

Word analogies

An analogy explains one thing in terms of another to highlight the ways in which they are alike. For example, *paris* is similar to *france* in the same way that *rome* is to *italy*. Word2Vec vectors sometimes shows the ability of solving analogy problem of the form **a is to b as a* is to what?**.

In the cell below, we show you how to use word vectors to find x. The `most_similar` function finds words that are most similar to the words in the `positive` list and most dissimilar from the words in the `negative` list. The

answer to the analogy will be the word ranked most similar (largest numerical value). In the case below, the top one word *italy* is the answer, so this analogy is solved successfully.

In [ ]:
```
# Run this cell to answer the analogy -- paris : france :: rome : x
print(w2v_google.most_similar(positive=['rome', 'france'], negative=['par
```

Exercise 2.1.1 (0.5 point)

Look for one analogy that can be solved successfully and one analogy that could not be solved using this pre-trained Word2Vec model. Check out this paper for inspirations.

In [ ]:
```
# Your successful case goes here


# Your failed case goes here
```

Visualising word analogies

The following cell shows you how to use tSNE to visualise a set of words based on their embeddings. You can also apply other dimensionality reduction methods (e.g. sklearn.decomposition.TruncatedSVD) to reduce the vectors from 300-dimensional to 2 dimensional.

Please note, reducing dimensionality from 300 to 2 is a very challenging task. You can try different parameters in the tSNE and see their effects on the final visualisation. In particular, the visualisation is very sensitive to the perplexity value that you give. Please try a few different perplexity valuse and keep the one that gives the most reasonable visusalisation.

In [ ]:
```
from sklearn.manifold import TSNE
import numpy as np
import matplotlib.pyplot as plt

def tsne_plot(model, wordlist, p):
    labels = []
    tokens = []

    for word in wordlist:
        tokens.append(model[word])
        labels.append(word)

    tokens = np.array(tokens)

    tsne_model = TSNE(perplexity=p, n_components=2, init='pca', max_iter=
    new_values = tsne_model.fit_transform(tokens)

    x = []
    y = []
    for value in new_values:
        x.append(value[0])
```

```
        y.append(value[1])

    plt.figure(figsize=(18, 18))
    for i in range(len(x)):
        plt.scatter(x[i], y[i])
        plt.annotate(labels[i],
                     xy=(x[i], y[i]),
                     xytext=(5, 2),
                     textcoords='offset points',
                     ha='right',
                     va='bottom')
    plt.show()

wordlist = ['man', 'woman', 'nephew', 'niece', 'brother', 'sister', 'uncl
tsne_plot(w2v_google, wordlist, len(wordlist) - 1)
```

Exercise 2.1.2 (0.5 point)

Find another group analogies (at least 3 pairs of words) and see how they are visualised.

In [ ]:
```
# prepare at least 3 pairs of words

# your answer goes here

wordlist=[]
p=len(wordlist)-1
tsne_plot(w2v_google,wordlist,p)
```

Exercise 2.1.3 Synonyms and antonyms (0.5 point)

Find three words (w1, w2, w3) so that

- w1 and w2 are synonyms,
- w1 and w3 are antonyms,
- cosine_distance(w1, w2) > cosine_distance(w1, w3) or cosine_distance(w1, w2) ≈ cosine_distance(w1, w3).

Please give a possible explanation for why this has happened.

You can use `w2v_google.distance()` function to compute the cosine distance between two words.

In [ ]:
```
# Replace XXX, YYY and ZZZ with your chosen words

w1='XXX'
w2='YYY'
w3='ZZZ'

print("Synonyms {}, {} have cosine distance: {}".format(w1, w2, w2v_googl
print("Antonyms {}, {} have cosine distance: {}".format(w1, w3, w2v_googl
```

Your answer:

Exercise 2.1.4 Polysemous Words (0.5 point)

Some words are polysemous, i.e. they have multiple meanings. For example the word *bank* can be a financial institute or the rising ground bordering a lake or river. Find a polysemous word whose top most similar words contains related words from multiple meanings. You should use the the `wv_google.most_similar()` function to compute the closet neighbours of the word. You may increase the number of neighbours in order to identify multiple groups of meanings. Submit the ranked word list and explained how the words are grouped into different meanings.

In [ ]:
```
w2v_google.most_similar('crane',topn=50)
```

In [ ]:
```
focusword='crane'
wordlist=[focusword]
for w in w2v_google.most_similar(focusword,topn=100):
    wordlist.append(w[0])
print(wordlist)
tsne_plot(w2v_google,wordlist,50)
```

Look into literature and describe potential methods to address this polysymy issue in word embeddings. Please cite the papers that you refer to.

YOUR ANSWER:

Exercise 2.2 Self-trained Word2Vec model

The word2vec model that we have been using so far is pre-trained on Google news. This is suitable for applications involving general topics. However, for special domains, such as scientific or medical domain, some domain-specific semantics could not be captured in the pre-trained model. Fortunately, word2vec is pretty efficient in training from scratch. We will use two different datasets to observer the effect on the input corpus.

Importance parameters are highlighted in bold. Please choose a few different values and see their effects.

class gensim.models.word2vec.Word2Vec(sentences=None, corpus_file=None, **vector_size=100**, alpha=0.025, **window=5**, **min_count=5**, max_vocab_size=None, sample=0.001, seed=1, workers=3, min_alpha=0.0001, **sg=0**, hs=0, **negative=5**, ns_exponent=0.75, cbow_mean=1, hashfxn=, epochs=5, null_word=0, trim_rule=None, sorted_vocab=1, batch_words=10000, compute_loss=False, callbacks=(), comment=None, max_final_vocab=None, shrink_windows=True)

Please check the gensim documentation for more assistance.

In [ ]:
```
# the most similar words to 'young' in Google news
w2v_google.most_similar('young')
```

Exercise 2.2.1 (1 point)

We first train a word2vec model on the corpus consisting the abstracts from 111K astrophysics/astronomy articles.

In [ ]:
```
# This might take up a few minutes to train.
from gensim.models.word2vec import LineSentence, Word2Vec
sentences=LineSentence('astro_norm.txt')

start_time = time.time()
# Train a word2vec model using the astro dataset
# your code starts here



# your code ends here
print("--- %s seconds ---" % (time.time() - start_time))
```

In [ ]:
```
w2v_astro.wv.most_similar('young')
```

If all goes well, you may see *pms*, *proto* or *yso* among the top 10 most similar words to *young*. If you are curious, protostars and pre-main-sequence (PMS) stars are all Young Stella Objects (YSOs). Here, "young" means pre-main-sequence. For low-mass stars, this means ages of 10^5 to 10^8 years. Ref

We then train a word2vec model on the corpus consisting of nearly 479K Medline articles. Note, this corpus is rather big. If this is too much for your local machine, use UT's JupyterLab or Google Colab.

In [ ]:
```
# This might take up half an hour to train!

from gensim.models.word2vec import LineSentence, Word2Vec
sentences=LineSentence('medline_norm.txt')

start_time = time.time()
# Train a word2vec model using the astro dataset
# your code starts here



# your code ends here
print("--- %s seconds ---" % (time.time() - start_time))
```

In [ ]:
```
w2v_medline.wv.most_similar('young')
```

Find another word and compute its most similar words based on different models. Please explain why this happens.

YOUR ANSWER:

In [ ]:
```
w2v_google.most_similar('XXX')  # Replace XXX with your chosen word
```

In [ ]:
```
w2v_astro.wv.most_similar('XXX')
```

In [ ]:
```
w2v_medline.wv.most_similar('XXX')
```

YOUR ANSWER to Why this happens:

Exercise 2.2.2 (1 point)

Experiment with different parameters, for example, the vector size, the window size, the minimal count, skip-gram or CBOW, etc. Observe their effects on the quality of the word embeddings and/or computational cost.

You can apply intrinsic evaluations to compare the quality of your models. For example, your can check the correlation with human opinion on word similarity or on word analogies. Check gensim documentations for more options. For example, evaluate_word_analogies and evaluate_word_pairs.

YOUR ANSWER:

What are your observations?

YOUR ANSWER:

Part III. Exploring and Evaluating a Large Language Model (LLM)

In this part, you will apply what you've learned about representing and working with language to a modern NLP tool — a Large Language Model (LLM), — using free, local models via Hugging Face. You will design prompts, run them through an LLM, collect the outputs, and evaluate them critically.

> ⚠️ **Part III Instructions:** You may either **keep your code and results directly in this Jupyter notebook** or **organise them in a separate document and upload it to Canvas**.

Minimal environment setup (CPU, text-only)

In [ ]:
```
!pip install --upgrade torch --index-url https://download.pytorch.org/whl
!pip install --upgrade transformers pandas tqdm
```

Set these before any transformers import to avoid TensorFlow/torchvision and accelerate vs. transformers collisions:

In [ ]:
```
import os, sys, site, subprocess

# Block optional backends before *any* transformers import
os.environ["TRANSFORMERS_NO_TF"] = "1"
```

```
os.environ["TRANSFORMERS_NO_TORCHVISION"] = "1"
os.environ["TRANSFORMERS_NO_JAX"] = "1"

# (Optional but helpful) install a clean CPU torch + stable transformers
def pipi(args): subprocess.check_call([sys.executable, "-m", "pip", "inst
pipi(["--index-url", "https://download.pytorch.org/whl/cpu", "torch==2.5.
pipi(["transformers==4.45.2"])

# Prefer user site-packages in this session
user_site = site.getusersitepackages()
if user_site not in sys.path:
    sys.path.insert(0, user_site)
```

Warm-up Exercise: Playing with Small LLMs

Before diving into the main assignment, try running one of these lightweight models locally. You'll get a feel for how different architectures respond to prompts.

Option 1 – Seq2Seq Model (encoder-decoder)

How it works: First encodes the input text into a hidden representation, then decodes it into an output sequence. The input and output can be different in length and form.

Strengths: Great at transforming text from one form to another (e.g., summarization, translation, question answering).

Examples: T5, BART, FLAN-T5.

In [ ]:
```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

# 1. Pick your model
seq2seq_name = "google/flan-t5-small" # Options: "google/flan-t5-small" o

# 2. Load with CPU optimization
seq_tok = AutoTokenizer.from_pretrained(seq2seq_name)
seq_model = AutoModelForSeq2SeqLM.from_pretrained(seq2seq_name)

# 3. Prompt the model to provide an answer
x = seq_tok("Write a haiku about AI", return_tensors="pt")
y = seq_model.generate(**x, max_new_tokens=400, do_sample=True, temperatu
print(seq_tok.decode(y[0], skip_special_tokens=True))
```

Option 2 - Causal Model (decoder-only)

How it works: Predicts the next token in a sequence based only on the tokens before it, generating text step-by-step.

Strengths: Ideal for free-form generation where you want the model to continue or expand on a prompt (e.g., storytelling, dialogue, creative writing).

Examples: GPT-2, GPT-Neo, LLaMA, Mistral.

In [ ]:
```
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch

causal_name = "TinyLlama/TinyLlama-1.1B-Chat-v1.0"  # or "microsoft/phi-2
causal_tok = AutoTokenizer.from_pretrained(causal_name)
causal_model = AutoModelForCausalLM.from_pretrained(causal_name)

prompt = "Write a haiku about AI:" # What happens if you remove the colon
ids = causal_tok(prompt, return_tensors="pt")
out = causal_model.generate(
    **ids,
    max_new_tokens=80,
    do_sample=True, temperature=0.8, top_p=0.95,
    pad_token_id=causal_tok.eos_token_id or tok.pad_token_id  # important
)
print(causal_tok.decode(out[0], skip_special_tokens=True))
```

Exercise 3.1 (1 point)

Experiment with different prompt design techniques to explore how LLM responses vary, evaluate their effectiveness, and develop best practices for a chosen application scenario (e.g. summarisation, question answering, text classification, instruction following, etc.)

Describe your NLP application and justify your choice of models - indicate whether you are using Seq2Seq (encoder–decoder) or Causal (decoder-only) models, explain why they are suited to your application, and mention any constraints (e.g., speed, resources, interpretability).

YOUR ANSWER:

Exercise 3.2 (2 point)

Try at least three different prompt techniques (zero-shot, few-shot, chain-of-thought, role prompting, etc.) and compare the results. Document the exact prompts text, the model's raw output, your reflection on effectiveness (e.g., factual accuracy, consistency, clarity, creativity, etc.)

YOUR ANSWER:

In [1]:
```
# This is an example NLP scenario

# Step 1: Define your task and prompt variants
TASK = "Write a short museum label (80–120 words) for Van Gogh's 'Sunflow
PROMPTS = [
    {"name": "zero-shot", "system": "You are a helpful museum guide.", "u
    {"name": "role+style", "system": "You are a witty museum educator for
    {"name": "cot-steps", "system": "You are a precise art historian.", "
]
```

In [ ]:
```

```