AML 5283 | Natural Language Processing | Lab Final | Even Semester 2024

Instructions:

1. The exam is open book, notes, internet etc. You are welcome to refer to any non-human resource such as ChatGPT, Gemini, Grok, Bard etc., for answering the questions. In particular, the code closely follows the word2vec tutorial;
2. However, you must *not* discuss your questions or code with anyone else--inside or outside the class;
3. You should not share the code with anyone else; doing so will result in significant penalties for all involved.
4. By submitting your work, you are implicitly honoring the agreement above;
5. You might be called for a one-on-one during the exam after reviewing your submission to explain your code and answer additional questions. Failure to justify your code and answers will result in significant points docked from your exam score.
6. After finishing the exam, delete all codes related to the exam from the computer you are working on.

Upload PDF of your completed code clearly showing the output cells (go to file->print->save as PDF choosing Landscape orientation) with the naming convention example

$$\text{NLP\_LabFinal\_SudarsanAcharya.pdf}$$

by clicking here

Load libraries

```
## Load libraries
import numpy as np
import sys
import random
import matplotlib.pyplot as plt
import io
import re
import string
import tqdm

plt.style.use('dark_background')
%matplotlib inline

import tensorflow as tf
print(tf.__version__)
from tensorflow.keras import layers
```

```
2.15.0
```

Mount Google Drive

```
## Mount Google drive folder if running in Colab
if('google.colab' in sys.modules):
    from google.colab import drive
    drive.mount('/content/drive', force_remount = True)
    # Change path below starting from /content/drive/MyDrive/Colab Notebooks/
    # depending on how data is organized inside your Colab Notebooks folder in
    # Google Drive
    DIR = '/content/drive/MyDrive/NLP_Data_Set'
    DATA_DIR = DIR+'/Data/'
else:
    DATA_DIR = 'Data/'
```

```
Mounted at /content/drive
```

Load the Amazon reviews dataset which comprises a review per line with the corresponding review rating showing at the beginning of the line.

```
## Load the Amazon reviews dataset which comprises a review per line
## with the corresponding review rating showing at the beginning of the line.
FILENAME = DATA_DIR + 'amazonreviews.txt'
with open(FILENAME) as f:
  lines = f.read().splitlines()

# Print a few reviews to see how the review rating shows up in the
# beginning of each sentence
for line in lines[:5]:
  print(line)
```

```
__label__1 Dangerous Product!!!: I purchased the Vtech 2421 as a Christmas gift for my girlfriend. It worked for 3 days and then stopped. She then let it
__label__1 Very boring !: "Succubus" is incoherent, confusing, and, above all, very very boring. I mean i like weird, surreal movies, movies when it's ve
__label__2 A Terrific First Novel!: THE FIXER marks an impressive debut from this Boston author. Vampires, spies, lust, it's got a bit of everything! The
__label__2 so educational and yet fun: the crusin world game is so fun because you get 2 c different places while your still sitting on the floor playing
__label__1 This broke immediately: Very cute but unfortunately it is cheaply made. The handle broke immediately. Be aware that it is very tiny too.
```

(Q1) Compile the reviews as a list of tuples such that each review corresponds to a tuple with 2 elements like

(label1, Dangerous Product!!!: I purchased......)

(label1, Very boring !: "Succubus" is...........)

⋮

```
## Compile the reviews as a list of tuples
reviews = []
FILENAME = DATA_DIR + 'amazonreviews.txt'
with open(FILENAME) as f:
  lines = f.read().splitlines()
for line in lines:
  words = line.split()
  reviews.append((words[0], ' '.join(words[1:])))
print(reviews[0])
print(reviews[1])
```

```
('__label__1', 'Dangerous Product!!!: I purchased the Vtech 2421 as a Christmas gift for my girlfriend. It worked for 3 days and then stopped. She then l
('__label__1', 'Very boring !: "Succubus" is incoherent, confusing, and, above all, very very boring. I mean i like weird, surreal movies, movies when it
```

(Q2) Remove special characters and numerical values from each review and convert to lower case.

After that, concatenate all cleaned reviews into one big string and tokenize that string.

Finally, create a vocabulary for the concatenated reviews which should help you identify the `vocab_size` (the vocabulary size) for the entire set of reviews which will be used later.

```
# Remove special characters and numerical values from each review
# and convert to lower case
reviews_cleaned = [re.sub(r'[^\w\s]', '', re.sub(r'\d+', '', review)).lower() for _, review in reviews]

# Concatenate all cleaned reviews into one string and tokenize that string
tokens = list(' '.join(reviews_cleaned).split())

# Create vocabulary for concatenated reviews
vocab, index = {}, 1  # start indexing from 1
vocab['<pad>'] = 0  # add a padding token
for token in tokens:
  if token not in vocab:
    vocab[token] = index
    index += 1
vocab_size = len(vocab)
print(vocab)
print(vocab_size)
```

```
{'<pad>': 0, 'dangerous': 1, 'product': 2, 'i': 3, 'purchased': 4, 'the': 5, 'vtech': 6, 'as': 7, 'a': 8, 'christmas': 9, 'gift': 10, 'for': 11, 'my': 12
9735
```

Create a tensorflow dataset object from the cleaned reviws and check that the elements of the tensorflow dataset object are the reviews stored as tensors.

```
## Create a tensorflow dataset from the cleaned reviews
text_ds = tf.data.Dataset.from_tensor_slices([x for x in reviews_cleaned])

# Check that the elements of the tensorflow dataset are the
# reviews stored as tensors
for element in text_ds:
  print(element)
```

```
tf.Tensor(b'cell  and unfinished novel ive been a king fan for many years this book was great  until you got to the end it just leaves you hanging neve
tf.Tensor(b'pretty good work for a teenager frankenstein is a little book that is a very creditable effort for a nineteenyearold author to have written
tf.Tensor(b'a step above cussler i read with interest the reviewer who dissed this book and couldnt disagree more as a once cussler fan i got bored wit
tf.Tensor(b'great mouse horrible with interference sadly this mouse at least the release i have is horrible at interference my version came with the mx
tf.Tensor(b'expected better not really impressed with the litter pan i mean shipping was prompt and even showed up earlier and it was well packaged i l
tf.Tensor(b'airborn audio m savvid and high priest after spending the past two years in the lab high priest and m savvid of anti pop consortium will re
```

User-defined function to generate skip-gram pairs with negative sampling for a list of sequences (int-encoded sentences) based on window size, number of negative samples and vocabulary size.

```
# Function to generates skip-gram pairs with negative sampling for a list of
# sequences (int-encoded sentences) based on window size, number of negative
# samples and vocabulary size.
def generate_training_data(sequences, window_size, num_ns, vocab_size, seed):
  # Elements of each training example are appended to these lists.
  targets, contexts, labels = [], [], []

  # Build the sampling table for `vocab_size` tokens.
  sampling_table = tf.keras.preprocessing.sequence.make_sampling_table(vocab_size)

  # Iterate over all sequences (sentences) in the dataset.
  for sequence in tqdm.tqdm(sequences):

    # Generate positive skip-gram pairs for a sequence (sentence).
    positive_skip_grams, _ = tf.keras.preprocessing.sequence.skipgrams(
          sequence,
          vocabulary_size = vocab_size,
          sampling_table = sampling_table,
          window_size = window_size,
          negative_samples = 0)

    # Iterate over each positive skip-gram pair to produce training examples
    # with a positive context word and negative samples.
    for target_word, context_word in positive_skip_grams:
      context_class = tf.expand_dims(
          tf.constant([context_word], dtype="int64"), 1)
      negative_sampling_candidates, _, _ = tf.random.log_uniform_candidate_sampler(
          true_classes=context_class,
          num_true = 1,
          num_sampled = num_ns,
          unique = True,
          range_max = vocab_size,
          seed = seed,
          name = "negative_sampling")

      # Build context and label vectors (for one target word)
      context = tf.concat([tf.squeeze(context_class,1), negative_sampling_candidates], 0)
      label = tf.constant([1] + [0]*num_ns, dtype="int64")

      # Append each element from the training example to global lists.
      targets.append(target_word)
      contexts.append(context)
      labels.append(label)
  return targets, contexts, labels
```

(Q3) Define the vocabulary size and the number of words in a sequence (that is, each sentence) that you identified in the earlier question.

Following that, define a vectorization layer which will be used for preliminary integer-vectorizing the reviews

```
# Define the vocabulary size and the number of words in a sequence.
vocab_size = len(vocab)
sequence_length = 100

# Use the `TextVectorization` layer to integer-vectorize each cleaned review
vectorize_layer = layers.TextVectorization(
    max_tokens = vocab_size,
    output_mode = 'int',
    output_sequence_length = sequence_length)
```

Create and save the vocabulary, and vectorize the dataset.

```
# Call TextVectorization.adapt on the text dataset to create vocabulary.
batch_size = 1024
vectorize_layer.adapt(text_ds.batch(batch_size))

# Save the created vocabulary for reference.
inverse_vocab = vectorize_layer.get_vocabulary()

# Vectorize the data in dataset
text_vector_ds = text_ds.batch(batch_size).prefetch(tf.data.AUTOTUNE).map(vectorize_layer).unbatch()
```

Flatten the dataset into a list of sentence vector sequences. Note that each sequence corresponds to a review.

```
## Flatten the dataset into a list of sentence vector sequences
sequences = list(text_vector_ds.as_numpy_iterator())
```

Inspect a few vectorized examples from `sequences`

```
## Inspect a few vectorized examples from `sequences`:
for seq in sequences[:5]:
  print(f"{seq} => {[inverse_vocab[i] for i in seq]}")
```

```
[8570   67    3  272    2 4345   24    5  408  546   12   15 2441    7
  413   12  296    4  138  652   90  138  383    7 1592   12  542  143
 2668    6   71    7  172    4   43   90  217    7   14   28 6224    2
 1521   40   41 2005   95  339    3  511    2   67   56  270   64   14
    5  302  783    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0] => ['dangerous', 'product', 'i', 'purchased', 'the', 'vtech', 'as', 'a', 'christmas', 'gift', 'for', 'my', 'girlfriend', 'it', 'worked', 'for
[  29  261 2905   10 3504 1005    4 1027   27   29   29  261    3  732
    3   30 1167 4864  267  267   43   34   29  187   12    2 1647    6
 5304    2 1209   37    2 1122   21    9   75 1045  221 4686   92  188
    3  147    7   49   25   35  292    5 1253   75   70    3   14  526
   11  254 2905   10   16    5 1253   75   35 9594 1331   10 1019    9
   35   27 8859   56  453   12  178   87 4899   37 7237    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0] => ['very', 'boring', 'succubus', 'is', 'incoherent', 'confusing', 'and', 'above', 'all', 'very', 'very', 'boring', 'i', 'mean', 'i', 'like',
[   5 2112   66  348    2 7894 6753   44 1867 3844   37    9 2015  229
 4420 5062 6816   34   97    5  223    8  362    2  278   10  657    4
 7856   28    7  175   12    5   29 5399   42    9   22   10    5  181
 1066    4    3  169  393 4690    2  247 1502   10 1071  120    5 1112
  235    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0] => ['a', 'terrific', 'first', 'novel', 'the', 'fixer', 'marks', 'an', 'impressive', 'debut', 'from', 'this', 'boston', 'author', 'vampires',
[  28 1549    4  239  181    2 8614  215  117   10   28  181   84   17
   50 1604  191 1443  168   57  133 2171   19    2 1273  513    5  315
 7736   50 1604 1443    4   30 6567    4    2  349    8 4769   36   20
   11   13 1132   39 1590   39 8722   98    7   10   94   65    2   82
 8613 1653   84    7   46  116  191 3941    8 1139    4   17   61 1304
    2  252   17  124    6  251   11    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0] => ['so', 'educational', 'and', 'yet', 'fun', 'the', 'crusin', 'world', 'game', 'is', 'so', 'fun', 'because', 'you', 'get', 'c', 'different',
[   9  485  870   29  593   21  377    7   10 3948  129    2  497  485
  870   25 1148   13    7   10   29 1674   73    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0] => ['this', 'broke', 'immediately', 'very', 'cute', 'but', 'unfortunately', 'it', 'is', 'cheaply', 'made', 'the', 'handle', 'broke', 'immedia
```

(Q4) Generate training examples from `sequences`.

```
SEED = 42
## Generate training examples from `sequences`
targets, contexts, labels = generate_training_data(
    sequences = sequences,
    window_size = 128,
    num_ns = 20,
    vocab_size = vocab_size,
    seed = SEED)

targets = np.array(targets)
contexts = np.array(contexts)
labels = np.array(labels)

print('\n')
print(f"targets.shape: {targets.shape}")
print(f"contexts.shape: {contexts.shape}")
print(f"labels.shape: {labels.shape}")
```

```
100%|██████████| 1000/1000 [13:24<00:00,  1.24it/s]


targets.shape: (888284,)
contexts.shape: (888284, 21)
labels.shape: (888284, 21)
```

Setup the dataset to perform efficient batching for the potentially large number of training examples when training the word2vec model.

```
## Configure the dataset for performance
BATCH_SIZE = 1024
BUFFER_SIZE = 10000
dataset = tf.data.Dataset.from_tensor_slices(((targets, contexts), labels))
dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)
dataset = dataset.cache().prefetch(buffer_size = tf.data.AUTOTUNE)
```

Word2Vec model defined using the Keras Subclassing API

```
## Word2Vec model defined using the Keras Subclassing API
## (https://www.tensorflow.org/guide/keras/custom_layers_and_models)
class Word2Vec(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim):
    super(Word2Vec, self).__init__()
    self.target_embedding = layers.Embedding(vocab_size,
                                             embedding_dim,
                                             name="w2v_embedding")
    self.context_embedding = layers.Embedding(vocab_size,
                                              embedding_dim)

  def call(self, pair):
    target, context = pair
    # target: (batch, dummy?)  # The dummy axis doesn't exist in TF2.7+
    # context: (batch, context)
    if len(target.shape) == 2:
      target = tf.squeeze(target, axis=1)
    # target: (batch,)
    word_emb = self.target_embedding(target)
    # word_emb: (batch, embed)
    context_emb = self.context_embedding(context)
    # context_emb: (batch, context, embed)
    dots = tf.einsum('be,bce->bc', word_emb, context_emb)
    # dots: (batch, context)
    return dots
```

(Q5) Define loss function and compile model

```
## Define loss function and compile model
embedding_dim = 128
word2vec = Word2Vec(vocab_size, embedding_dim)
word2vec.compile(optimizer='adam',
                 loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
                 metrics=['accuracy'])
```

(Q6) Train the model on the `dataset`

```
# Train the model on the `dataset` for some number of epochs
word2vec.fit(dataset, epochs = 20, batch_size = batch_size)
```

```
Epoch 1/20
867/867 [==============================] - 37s 40ms/step - loss: 2.9972 - accuracy: 0.0867
Epoch 2/20
867/867 [==============================] - 3s 3ms/step - loss: 2.8935 - accuracy: 0.1070
Epoch 3/20
867/867 [==============================] - 3s 3ms/step - loss: 2.6788 - accuracy: 0.1861
Epoch 4/20
867/867 [==============================] - 4s 4ms/step - loss: 2.3166 - accuracy: 0.3293
Epoch 5/20
867/867 [==============================] - 3s 3ms/step - loss: 1.9994 - accuracy: 0.4196
Epoch 6/20
867/867 [==============================] - 3s 3ms/step - loss: 1.7937 - accuracy: 0.4659
Epoch 7/20
867/867 [==============================] - 3s 4ms/step - loss: 1.6611 - accuracy: 0.4928
Epoch 8/20
867/867 [==============================] - 3s 3ms/step - loss: 1.5692 - accuracy: 0.5113
Epoch 9/20
867/867 [==============================] - 3s 3ms/step - loss: 1.5012 - accuracy: 0.5249
Epoch 10/20
867/867 [==============================] - 4s 4ms/step - loss: 1.4488 - accuracy: 0.5354
Epoch 11/20
867/867 [==============================] - 3s 3ms/step - loss: 1.4069 - accuracy: 0.5441
Epoch 12/20
867/867 [==============================] - 3s 3ms/step - loss: 1.3728 - accuracy: 0.5514
Epoch 13/20
867/867 [==============================] - 3s 3ms/step - loss: 1.3445 - accuracy: 0.5573
Epoch 14/20
867/867 [==============================] - 4s 4ms/step - loss: 1.3206 - accuracy: 0.5622
Epoch 15/20
867/867 [==============================] - 3s 3ms/step - loss: 1.3003 - accuracy: 0.5663
Epoch 16/20
867/867 [==============================] - 3s 3ms/step - loss: 1.2828 - accuracy: 0.5698
Epoch 17/20
867/867 [==============================] - 3s 3ms/step - loss: 1.2676 - accuracy: 0.5728
Epoch 18/20
867/867 [==============================] - 4s 4ms/step - loss: 1.2543 - accuracy: 0.5754
Epoch 19/20
867/867 [==============================] - 3s 3ms/step - loss: 1.2425 - accuracy: 0.5777
Epoch 20/20
867/867 [==============================] - 3s 3ms/step - loss: 1.2321 - accuracy: 0.5796
<keras.src.callbacks.History at 0x7c76eaa16290>
```

Obtain the weights and the corresponding words from the model.

```
## Obtain the weights and the corresponding words from the model
weights = word2vec.get_layer('w2v_embedding').get_weights()[0]
vocab = vectorize_layer.get_vocabulary()

# Print a few words and their embeddings
for index, word in enumerate(vocab):
  if index == 0:
    continue  # skip 0, it's padding.

  if index in [10, 50, 100]:
   print(word)
   print(weights[index])
```

```
  1.19417816e-01  4.54035491e-01 -3.47425863e-02 -1.69408068e-01
  2.35649422e-01 -4.64829803e-01  1.35699302e-01 -5.24406612e-01]
get
[ 0.32412106 -0.26867813 -0.51757956  0.29829293  0.5028762   0.3358776
```

```
  2.40182742e-01 -5.19716591e-02  3.32006693e-01 -1.73560396e-01
  2.79580206e-01 -3.18618059e-01  2.66103595e-01  2.08188698e-01
 -2.07699195e-01  1.01884462e-01  2.27957338e-01  3.49910498e-01
 -4.32413593e-02 -2.71940976e-01  2.48790354e-01 -3.52767467e-01
  5.51318645e-01 -2.65631855e-01 -1.68014213e-01 -2.35113338e-01
  8.85301381e-02 -1.52057707e-01  3.44156891e-01  2.52940178e-01
 -7.83764869e-02 -6.18920475e-02 -5.62379062e-01  1.83441207e-01
  4.57281396e-02 -3.03052098e-01 -5.48661470e-01 -2.77032763e-01
  8.77584666e-02 -5.24212681e-02  5.55879951e-01 -2.37805285e-02
  4.45256829e-01 -1.29289135e-01  2.04189539e-01  2.16256455e-01
 -4.44533944e-01 -2.31991187e-02  2.78312951e-01 -1.79005302e-02
 -6.55845940e-01  6.88154772e-02  1.04755819e+00  4.65804577e-01
  4.36798334e-01  4.37678508e-02 -3.16271901e-01  5.53678945e-02
  9.62024182e-02 -4.57684606e-01  1.66577160e-01  8.80848318e-02
 -1.62925243e-01  3.49295735e-01 -7.69593418e-02  2.62079090e-01
  2.92586654e-01  3.28991324e-01 -6.40128255e-01 -1.36998951e-01
 -7.93183818e-02  4.39568311e-01  3.76329757e-02 -4.58860248e-01
 -3.39909464e-01 -1.57049641e-01  3.58893871e-01 -1.12000376e-01
  3.95395905e-02  3.42403531e-01  5.91424823e-01 -1.09443560e-01
 -5.71536243e-01 -2.19262436e-01  1.39294416e-01  1.35411128e-01
  3.36748391e-01 -1.18725814e-01 -7.79685557e-01 -2.82706946e-01
 -2.77753145e-01 -1.85699001e-01 -2.64769673e-01 -8.80588070e-02
 -1.20732069e-01  1.07201844e-01  1.65599450e-01  1.07509471e-01]
```

(Q7) Compute the average vector embedding for each sentence, that is, for each cleaned review.

This involves identifying the words in the vocabulary for each review whose embeddings have to be averaged.

```
## Compute the average vector embedding for each sentence, that
## is, for each cleaned review
average_embeddings = []

for review in reviews_cleaned:
    # Initialize a variable to store the sum of embeddings for words in the review
    review_embedding_sum = np.zeros(embedding_dim)  # Assuming embedding_dim is defined

    # Initialize a variable to count the number of words in the review that have embeddings
    num_words_with_embeddings = 0

    # Iterate through each word in the cleaned review
    for word in review.split():
        # Check if the word exists in the vocabulary
        if word in vocab:
            # Retrieve the embedding vector for the word from the weights array
            word_index = vocab.index(word)
            embedding_vector = weights[word_index]
            # Add the embedding vector to the sum
            review_embedding_sum += embedding_vector
            # Increment the count of words with embeddings
            num_words_with_embeddings += 1

    # If there are words with embeddings in the review, compute the average embedding
    if num_words_with_embeddings > 0:
        average_embedding = review_embedding_sum / num_words_with_embeddings
    else:
        # If no words in the review have embeddings, use a zero vector as the average embedding
        average_embedding = np.zeros(embedding_dim)

    # Append the average embedding for the review to the list
    average_embeddings.append(average_embedding)

# Convert the list of average embeddings to a numpy array
average_embeddings = np.array(average_embeddings)

# Print the shape of the resulting array of average embeddings
print("Shape of average embeddings array:", average_embeddings.shape)
```

```
Shape of average embeddings array: (1000, 128)
```

```
average_embeddings
```

```
array([[ 0.02933319, -0.03873046, -0.00230069, ...,  0.07121062,
        -0.04227232, -0.06334046],
       [ 0.15826841, -0.11459609,  0.00399778, ...,  0.0716318 ,
         0.12584683, -0.04665625],
       [ 0.0611398 , -0.02978408, -0.1450664 , ..., -0.18191394,
        -0.16540123, -0.09069099],
       ...,
       [ 0.09307673, -0.06982495, -0.05410084, ...,  0.05237945,
         0.0044741 , -0.07498172],
       [ 0.07491208, -0.03315069, -0.0407722 , ...,  0.01629238,
        -0.05055378, -0.05890643],
       [-0.10618844, -0.12592619,  0.07831757, ...,  0.08477453,
        -0.16435302, -0.24528601]])
```

(Q8) Apply Kmeans to the average vector embedding to cluster the cleaned reviews. Color code the clusters using the review labels. How can you relate the reviews w.r.t. the cluster they are falling into?

If they fall into the same cluster then it means they are similar to their average content and not similar if they do not fall into same cluster

```
## Apply Kmeans to the average vector embedding to cluster the cleaned
## reviews. Color code the clusters using the review labels.
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Initialize KMeans with the desired number of clusters
num_clusters = 5  # Adjust this based on your preference or analysis
kmeans = KMeans(n_clusters=num_clusters, random_state=42)

# Fit KMeans to the average vector embeddings
kmeans.fit(average_embeddings)

# Predict the cluster labels for each average vector embedding
```