# University of Central Missouri
# Department of Computer Science & Cybersecurity

## CS5720 Neural network and Deep learning
## Spring 2025

## Home Assignment 4. (Cover Ch 9, 10)

## Student name:Tadikonda Gnandeep

### Submission Requirements:

- Total Points: 100
- Once finished your assignment push your source code to your repo (GitHub) and explain the work through the ReadMe file properly. Make sure you add your student info in the ReadMe file.
- Submit your GitHub link and video on the BB.
- Comment your code appropriately ***IMPORTANT.***
- Make a simple video about 2 to 3 minutes which includes demonstration of your home assignment and explanation of code snippets.
- Any submission after provided deadline is considered as a late submission.

**Q1: NLP Preprocessing Pipeline**

Write a Python function that performs basic NLP preprocessing on a sentence. The function should do the following steps:

1. **Tokenize** the sentence into individual words.
2. **Remove common English stopwords** (e.g., "the", "in", "are").
3. **Apply stemming** to reduce each word to its root form.

**Use the sentence:**

**"NLP techniques are used in virtual assistants like Alexa and Siri."**

The function should print:

- A list of all tokens
- The list after stop words are removed
- The final list after stemming

**Expected Output:**

Your program should print three outputs in order:

1. **Original Tokens** – All words and punctuation split from the sentence
2. **Tokens Without Stopwords** – Only meaningful words remain
3. **Stemmed Words** – Each word is reduced to its base/root form

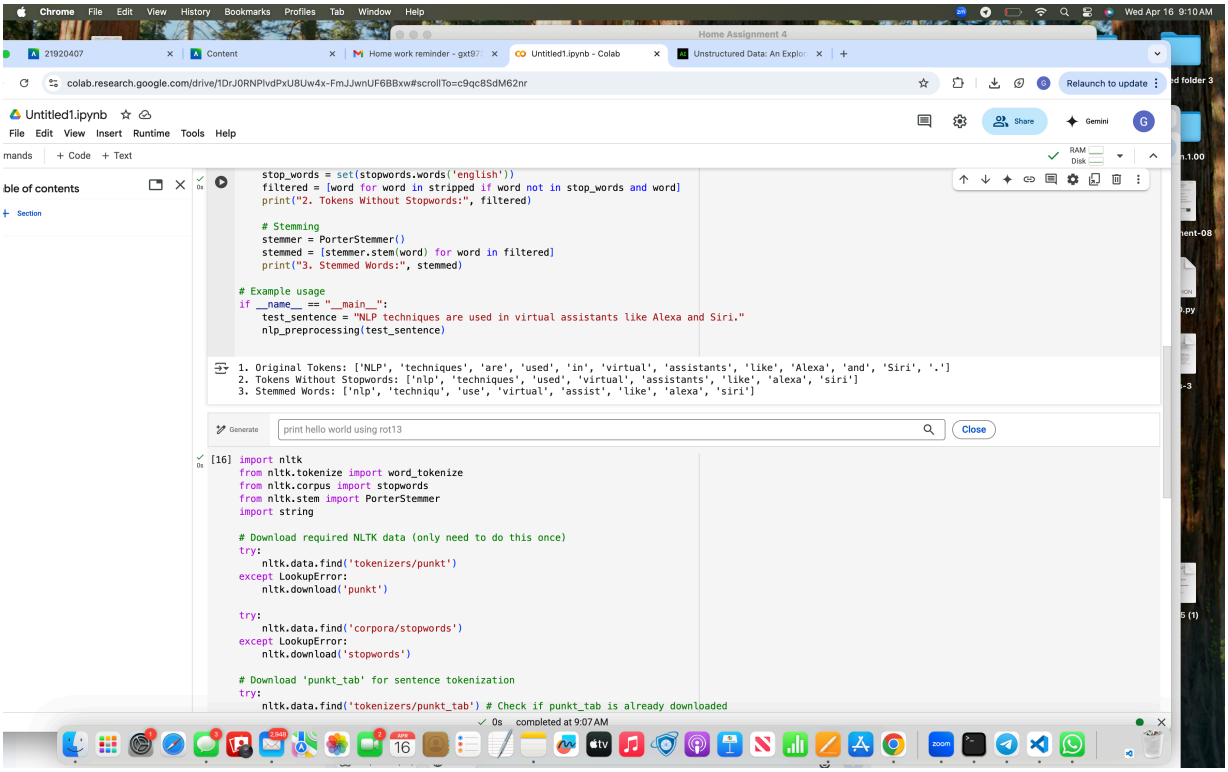
**Short Answer Questions:**

1. What is the difference between stemming and lemmatization? Provide examples with the word “running.”
2. Why might removing stop words be useful in some NLP tasks, and when might it actually be harmful?

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

```
import string

# Download required NLTK data (only need to do this
once)
try:
    nltk.data.find('tokenizers/punkt')
except LookupError:
    nltk.download('punkt')

try:
    nltk.data.find('corpora/stopwords')
except LookupError:
    nltk.download('stopwords')

def nlp_preprocessing(sentence):
    # Tokenization
    tokens = word_tokenize(sentence)
    print("1. Original Tokens:", tokens)

    # Remove punctuation and lowercase
    table = str.maketrans('', '', string.punctuation)
    stripped = [w.translate(table).lower() for w in
tokens if w.translate(table)]

    # Remove stopwords
    stop_words = set(stopwords.words('english'))
    filtered = [word for word in stripped if word not
in stop_words and word]
    print("2. Tokens Without Stopwords:", filtered)

    # Stemming
    stemmer = PorterStemmer()
    stemmed = [stemmer.stem(word) for word in filtered]
    print("3. Stemmed Words:", stemmed)

# Example usage
if __name__ == "__main__":
```

```
    test_sentence = "NLP techniques are used in virtual assistants like Alexa and Siri."
    nlp_preprocessing(test_sentence)
```




1ans)**The distinction between lemmatization and stemming:**

Lemmatization employs vocabulary and morphological analysis to provide the base dictionary form (lemma), whereas stemming removes word endings to obtain a root form (which might not be a true word).

Using "running" as an example:

"run" (or perhaps "runn") as the stem

**2Ans)Reduction of stop words:**

<u>Beneficial in:</u>

- concentrating on words with substance for tasks like topic modeling or information retrieval
- Text categorization with reduced dimensionality
- When stop words are insufficient to convey important information for the work at hand

harmful when:

- Working with syntactic analysis or phrase structure
- In situations where every word counts, such as language modeling or machine translation,
- When stop words—like "not" in sentiment analysis—have genuine meaning
- Small words might be important when answering questions ("to be or not to be").

**Q2: Named Entity Recognition with SpaCy**

**Task:** Use the spaCy library to extract **named entities** from a sentence. For each entity, print:

- The **entity text** (e.g., "Barack Obama")
- The **entity label** (e.g., PERSON, DATE)
- The **start and end character positions** in the string

Use the input sentence:

**"Barack Obama served as the 44th President of the United States and won the Nobel Peace Prize in 2009."**

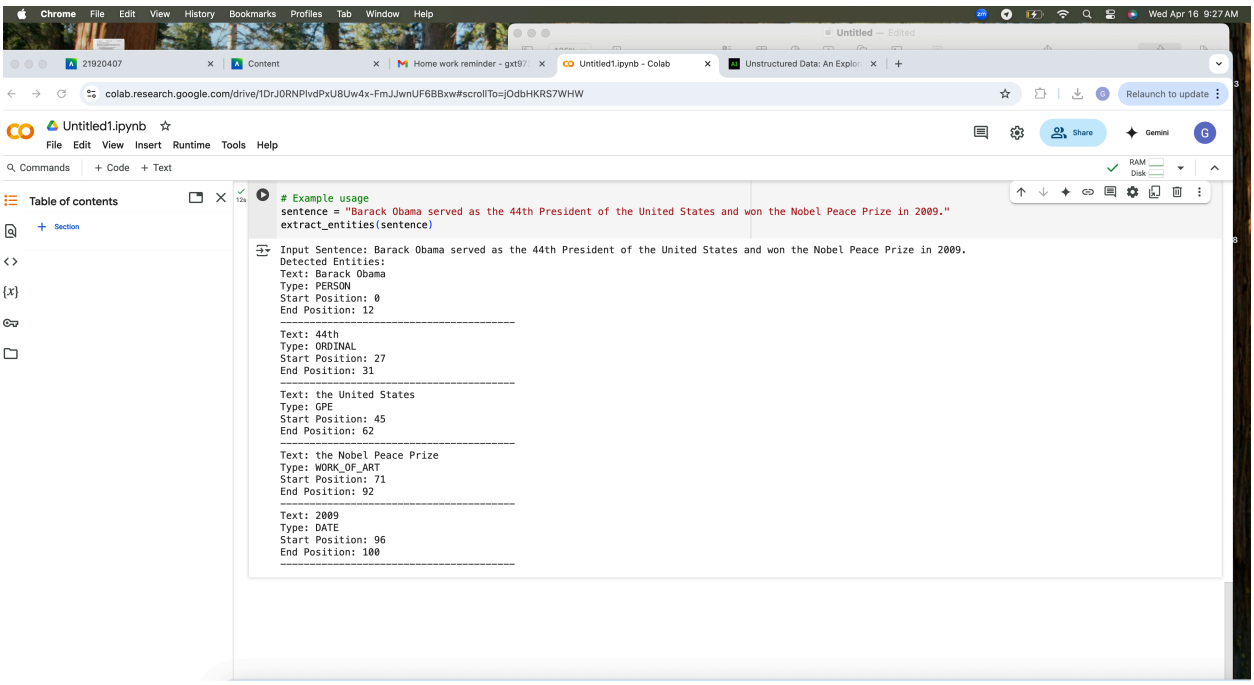
**Expected Output:**

Each line of the output should describe one entity detected

**Short Answer Questions:**

1. How does NER differ from POS tagging in NLP?
2. Describe two applications that use NER in the real world (e.g., financial news, search engines).

```python
import spacy

# Load the English language model
nlp = spacy.load("en_core_web_sm")

def extract_entities(sentence):
    doc = nlp(sentence)

    print("Input Sentence:", sentence)
    print("Detected Entities:")

    for ent in doc.ents:
        print(f"Text: {ent.text}")
        print(f"Type: {ent.label_}")
        print(f"Start Position: {ent.start_char}")
        print(f"End Position: {ent.end_char}")
        print("-" * 40)

# Example usage
sentence = "Barack Obama served as the 44th President of the United States and won the Nobel Peace Prize in 2009."
extract_entities(sentence)
```



1ans)

**I)Difference between NER and POS tagging :**

- **Named Entity Recognition, or NER**, recognizes and categorizes named entities in text, such as individuals, groups, places, dates, etc.
- **Part-of-Speech (POS) tagging** Each word's grammatical role—noun, verb, adjective, etc.—is indicated by tagging.

**Example**: ”Apple launched a new product”:
- "Apple" would be classified as an ORGANIZATION by NER.
- POS would mark "launched" as a verb and "apple" as a noun.

**II)Two real-world NER applications:**

1. Resume Parsing: HR software uses NER to automatically extract names, skills, education, and experience from resumes.
2. Customer Support Ticketing: Systems automatically detect product names, model numbers, and locations in support requests to route tickets appropriately.

**Q3: Scaled Dot-Product Attention**

**Task:** Implement the **scaled dot-product attention** mechanism. Given matrices Q (Query), K (Key), and V (Value), your function should:

- Compute the dot product of Q and $K^T$
- Scale the result by dividing it by $\sqrt{d}$ (where d is the key dimension)
- Apply softmax to get attention weights
- Multiply the weights by V to get the output

**Use the following test inputs:**

***Q = np.array([[1, 0, 1, 0], [0, 1, 0, 1]])***

***K = np.array([[1, 0, 1, 0], [0, 1, 0, 1]])***

***V = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])***

**Expected Output Description:**

Your output should display:

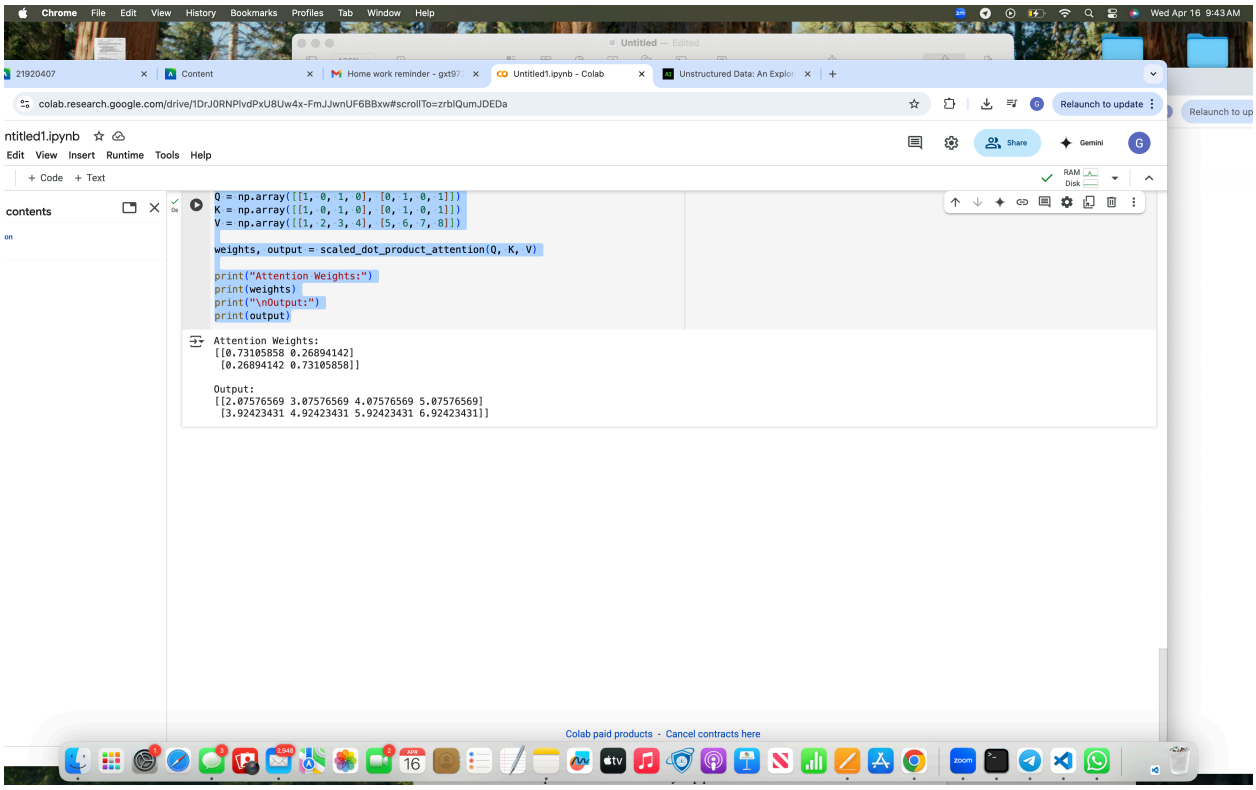
1. The **attention weights matrix** (after softmax)
2. The **final output matrix**

**Short Answer Questions:**

1. Why do we divide the attention score by √d in the scaled dot-product attention formula?
2. How does self-attention help the model understand relationships between words in a sentence?

```
def scaled_dot_product_attention(Q, K, V):
    """
    Q: Query matrix (n_q, d_k)
    K: Key matrix (n_k, d_k)
    V: Value matrix (n_k, d_v)
    """
    # 1. Compute dot product of Q and K^T
    d_k = K.shape[-1]
    scores = np.matmul(Q, K.T)  # (n_q, n_k)

    # 2. Scale by sqrt(d_k)
    scaled_scores = scores / np.sqrt(d_k)

    # 3. Apply softmax to get attention weights
    attention_weights = np.exp(scaled_scores) /
np.sum(np.exp(scaled_scores), axis=-1, keepdims=True)

    # 4. Multiply weights by V
    output = np.matmul(attention_weights, V)

    return attention_weights, output

# Test case
Q = np.array([[1, 0, 1, 0], [0, 1, 0, 1]])
```

```
K = np.array([[1, 0, 1, 0], [0, 1, 0, 1]])
V = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

weights, output = scaled_dot_product_attention(Q, K, V)

print("Attention Weights:")
print(weights)
print("\nOutput:")
print(output)
```




1. **Why divide by √d?**
   - The scaling factor √d (where d is the key dimension) prevents the dot products from becoming extremely large in magnitude when d is large. Without scaling:
     - Large dot product values would push the softmax into regions where it has extremely small gradients
     - This would make learning difficult (vanishing gradient problem)

- The scaling maintains stable gradients by keeping the attention scores at reasonable magnitudes

2. **How self-attention helps understand word relationships:**
   - Self-attention computes pairwise attention scores between all words in the input
   - This allows each word to directly attend to every other word in the sequence
   - Two key benefits:
     - Position-independent relationships: Can capture dependencies regardless of distance (unlike RNNs)
     - Interpretable patterns: The attention weights reveal which words the model considers important when processing each position
   - Example: For "The animal didn't cross the street because it was too tired", self-attention would learn to associate "it" with "animal" by giving them high attention scores

**Q4: Sentiment Analysis using HuggingFace Transformers**

**Task:** Use the HuggingFace transformers library to create a **sentiment classifier**. Your program should:

- Load a pre-trained sentiment analysis pipeline
- Analyze the following input sentence:
  **"Despite the high price, the performance of the new MacBook is outstanding."**
- Print:
  - **Label** (e.g., POSITIVE, NEGATIVE)
  - **Confidence score** (e.g., 0.9985)

**Expected Output**:

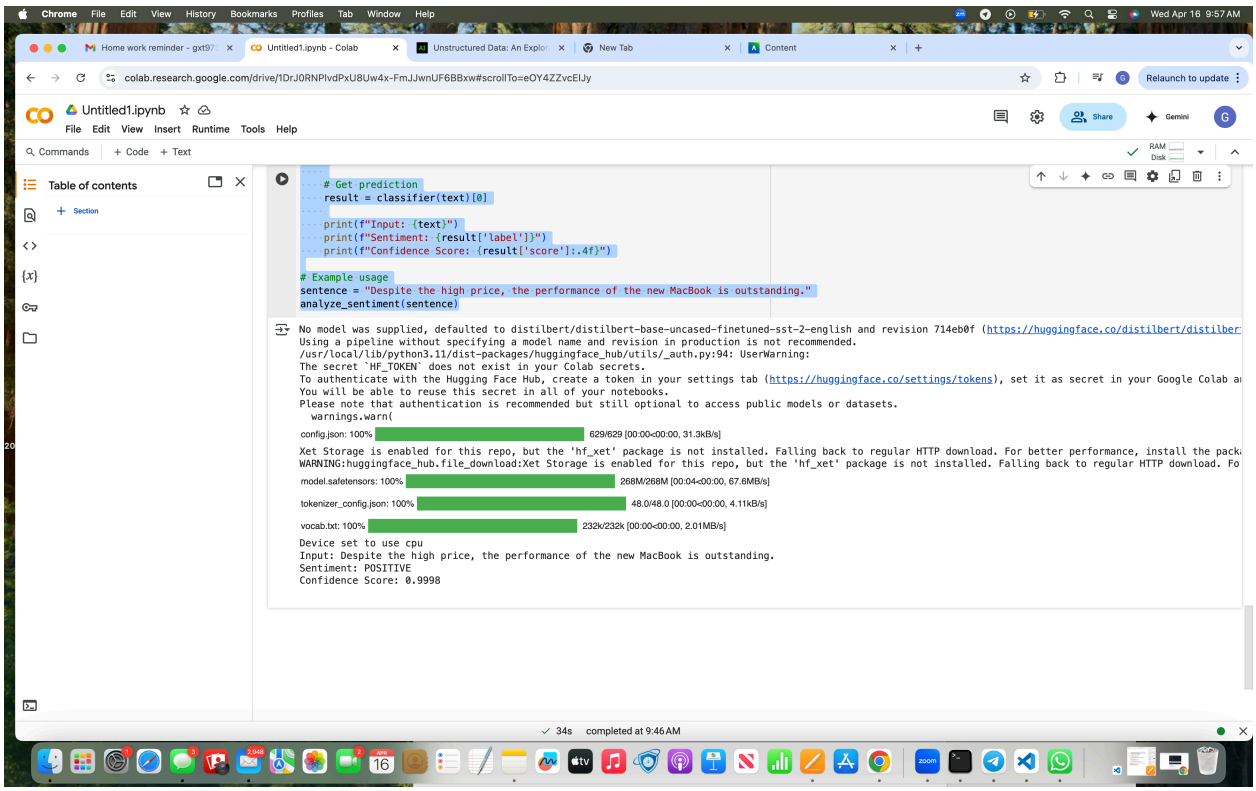
Your output should clearly display:

***Sentiment: [Label]***
***Confidence Score: [Decimal between 0 and 1]***

**Short Answer Questions:**

1. What is the main architectural difference between BERT and GPT? Which uses an encoder and which uses a decoder?
2. Explain why using pre-trained models (like BERT or GPT) is beneficial for NLP applications instead of training from scratch.

```
from transformers import pipeline

def analyze_sentiment(text):
    # Load pre-trained sentiment analysis pipeline
    classifier = pipeline("sentiment-analysis")

    # Get prediction
    result = classifier(text)[0]

    print(f"Input: {text}")
    print(f"Sentiment: {result['label']}")
    print(f"Confidence Score: {result['score']:.4f}")

# Example usage
sentence = "Despite the high price, the performance of the new MacBook is outstanding."
analyze_sentiment(sentence)
```



**1Ans)BERT vs GPT Architectural Difference:**

- **BERT** uses a transformer encoder architecture (bi-directional)
- Processes entire input sequence at once
- Sees both left and right context for each word

**GPT** uses a **transformer decoder** architecture (auto-regressive)
- Processes text sequentially from left to right
- Only sees previous words when predicting next word

**2Ans)Benefits of Pre-trained Models:**

**Transfer Learning**: Leverages knowledge from vast datasets (books, Wikipedia, etc.)
**Contextual Understanding**: Captures nuanced word meanings (e.g., "bank" as financial vs river)
**Efficiency**: Saves enormous compute resources vs training from scratch
**Performance**: Achieves state-of-the-art results with minimal task-specific data
**Multitask Capability**: Single model can handle multiple downstream tasks