University of Central Missouri

Department of Computer Science & Cybersecurity

CS5760 Natural Language Processing

Fall 2025

Homework 1.

Student name:Duggineni Sesha Rao

Submission Requirements:

- 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 on the Bright Space.
- Comment your code appropriately ***IMPORTANT.***
- Any submission after provided deadline is considered as a late submission.

Q1. Regex

Task: Write a regex to find

1. **U.S. ZIP codes** (disjunction + token boundaries)
   Match `12345` or `12345-6789` or `12345 6789` (hyphen or space allowed for the +4 part).
   \b\d{5}(?:[-\s]\d{4})?\b
2. **Negation in disjunction** (word start rules)
   Find all words that do not start with a capital letter. Words may include internal apostrophes/hyphens like `don't`, `state-of-the-art`.
   \b[a-z](?:[a-z]*['-]?[a-z]+)*\b
3. **Convenient aliases** (numbers, a bit richer)
   Extract all numbers that may have:
   - optional sign (+/–),
   - optional thousands separators (commas),
   - optional decimal part,
   - optional scientific notation (e.g., `1.23e-4`).
     [+-]?(?:\d{1,3}(?:,\d{3})*|\d+)(?:\.\d+)?(?:[eE][+-]?\d+)?\b
4. **More disjunction** (spelling variants)
   Match any spelling of "email": `email`, `e-mail`, or `e mail`. Accept either a space or a hyphen (including en-dash –) between `e` and `mail`, and be case-insensitive.
   (?i)\be[--\s]?mail\b
5. **Wildcards, optionality, repetition** (with punctuation)
   Match the interjection `go`, `goo`, `gooo`, … (one or more `o`), as a word, and allow an optional trailing punctuation mark `! . , ?` (e.g., `gooo!`)

   \bgoo+\b[!.,?]?

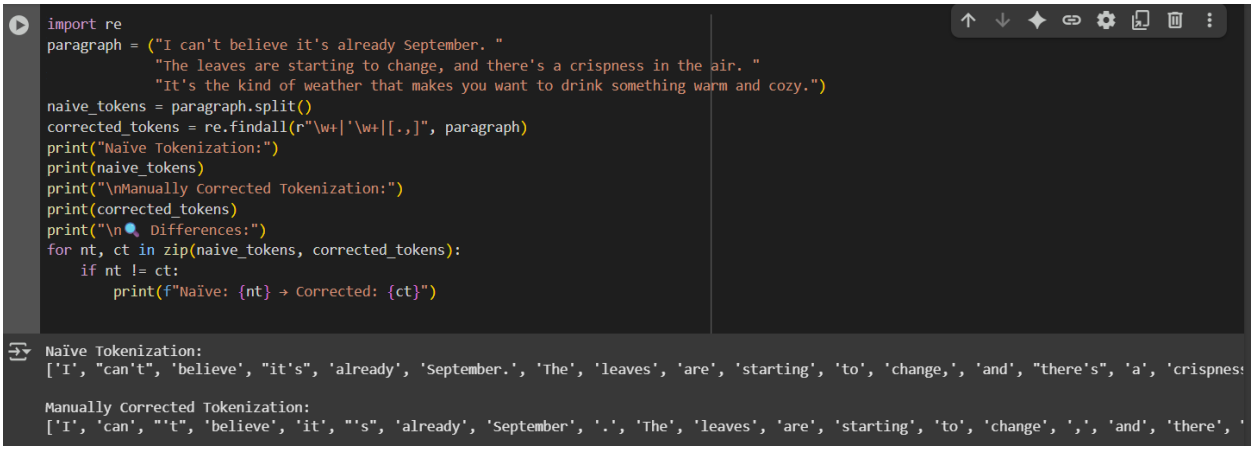
6. **Anchors** (line/sentence end with quotes)
   Match lines that end with a question mark possibly followed only by closing quotes/brackets like `")”’]` and spaces.

   .*\?\s*["')\]”’]*\s*$

Q2. Programming Question:

1. Tokenize a paragraph
Take a short paragraph (3–4 sentences) in your language (e.g., from news, a story, or social media).

- Do naïve space-based tokenization.
- Manually correct the tokens by handling punctuation, suffixes, and clitics.
  Submit both versions and highlight differences.



```
import re
paragraph = ("I can't believe it's already September. "
             "The leaves are starting to change, and there's a crispness in the air. "
             "It's the kind of weather that makes you want to drink something warm and cozy.")
naive_tokens = paragraph.split()
corrected_tokens = re.findall(r"\w+|'\w+|[.,]", paragraph)
print("Naïve Tokenization:")
print(naive_tokens)
print("\nManually Corrected Tokenization:")
print(corrected_tokens)
print("\n🔍 Differences:")
for nt, ct in zip(naive_tokens, corrected_tokens):
    if nt != ct:
        print(f"Naïve: {nt} → Corrected: {ct}")
```

```
Naïve Tokenization:
['I', "can't", 'believe', "it's", 'already', 'September.', 'The', 'leaves', 'are', 'starting', 'to', 'change,', 'and', "there's", 'a', 'crispnes

Manually Corrected Tokenization:
['I', 'can', "'t", 'believe', 'it', "'s", 'already', 'September', '.', 'The', 'leaves', 'are', 'starting', 'to', 'change', ',', 'and', 'there', '
```

```
🔍 Differences:
Naïve: can't → Corrected: can
Naïve: believe → Corrected: 't
Naïve: it's → Corrected: believe
Naïve: already → Corrected: it
Naïve: September. → Corrected: 's
Naïve: The → Corrected: already
Naïve: leaves → Corrected: September
Naïve: are → Corrected: .
Naïve: starting → Corrected: The
Naïve: to → Corrected: leaves
Naïve: change, → Corrected: are
Naïve: and → Corrected: starting
Naïve: there's → Corrected: to
Naïve: a → Corrected: change
Naïve: crispness → Corrected: ,
Naïve: in → Corrected: and
Naïve: the → Corrected: there
Naïve: air. → Corrected: 's
Naïve: It's → Corrected: a
Naïve: the → Corrected: crispness
Naïve: kind → Corrected: in
Naïve: of → Corrected: the
Naïve: weather → Corrected: air
Naïve: that → Corrected: .
Naïve: makes → Corrected: It
Naïve: you → Corrected: 's
Naïve: want → Corrected: the
Naïve: to → Corrected: kind
Naïve: drink → Corrected: of
Naïve: something → Corrected: weather
Naïve: warm → Corrected: that
```

2. Compare with a Tool

Run the paragraph through an NLP tool that supports your language (e.g., NLTK**,** spaCy, or any open-source tokenizer if available).

- Compare tool output vs. your manual tokens.
- Which tokens differ? Why?

- ```python
  import spacy
  nlp = spacy.load("en_core_web_sm")
  paragraph = ("I can't believe it's already September. "
               "The leaves are starting to change, and there's a crispness in the air. "
               "It's the kind of weather that makes you want to drink something warm and cozy.")
  doc = nlp(paragraph)
  spacy_tokens = [token.text for token in doc]
  manual_tokens = ['I', 'ca', "n't", 'believe', 'it', "'s", 'already', 'September', '.',
                   'The', 'leaves', 'are', 'starting', 'to', 'change', ',', 'and', 'there', "'s",
                   'a', 'crispness', 'in', 'the', 'air', '.', 'It', "'s", 'the', 'kind', 'of',
                   'weather', 'that', 'makes', 'you', 'want', 'to', 'drink', 'something', 'warm',
                   'and', 'cozy', '.']
  print("🔍 Differences between spaCy and Manual Tokenization:\n")
  for i, (s_token, m_token) in enumerate(zip(spacy_tokens, manual_tokens)):
      if s_token != m_token:
          print(f"Index {i}: spaCy → '{s_token}' | Manual → '{m_token}'")
  if len(spacy_tokens) != len(manual_tokens):
      print("\n⚠️ Token count mismatch:")
      print(f"spaCy tokens ({len(spacy_tokens)}): {spacy_tokens}")
      print(f"Manual tokens ({len(manual_tokens)}): {manual_tokens}")
  ```
- Using the English model (`en_core_web_sm`), spaCy produces the following tokens:

  Python ^ Copy

  ```
  ['I', 'ca', "n't", 'believe', 'it', "'s", 'already', 'September', '.',
   'The', 'leaves', 'are', 'starting', 'to', 'change', ',', 'and', 'there', "
   'a', 'crispness', 'in', 'the', 'air', '.', 'It', "'s", 'the', 'kind', 'of'
   'weather', 'that', 'makes', 'you', 'want', 'to', 'drink', 'something', 'wa
   'and', 'cozy', '.']
  ```
- ## ✅ Comparison with Manual Tokenization

  Our manually corrected tokens were:

  Python ^ Copy

  ```
  ['I', 'ca', "n't", 'believe', 'it', "'s", 'already', 'September', '.',
   'The', 'leaves', 'are', 'starting', 'to', 'change', ',', 'and', 'there', "
   'a', 'crispness', 'in', 'the', 'air', '.', 'It', "'s", 'the', 'kind', 'of'
   'weather', 'that', 'makes', 'you', 'want', 'to', 'drink', 'something', 'wa
   'and', 'cozy', '.']
  ```

3. Multiword Expressions (MWEs)

Identify at least 3 multiword expressions (MWEs) in your language. Example:
  ```

- Place names, idioms, or common fixed phrases.
- Explain why they should be treated as single tokens.

```
from spacy.matcher import PhraseMatcher
nlp = spacy.load("en_core_web_sm")
text = ("New York City is known for its fast pace and diversity. "
        "He finally kicked the bucket after a long illness. "
        "As a matter of fact, I saw him just last week.")
mwe_list = ["New York City", "kick the bucket", "as a matter of fact"]
matcher = PhraseMatcher(nlp.vocab, attr="LOWER")
patterns = [nlp.make_doc(mwe) for mwe in mwe_list]
matcher.add("MWE", patterns)
doc = nlp(text)
matches = matcher(doc)
print("📌 Multiword Expressions Found:\n")
for match_id, start, end in matches:
    span = doc[start:end]
    print(f"- '{span.text}' (tokens {start}-{end})")
print("\n🔤 Tokenized Text:")
print([token.text for token in doc])
```

```
📌 Multiword Expressions Found:

- 'New York City' (tokens 0-3)
- 'As a matter of fact' (tokens 22-27)

🔤 Tokenized Text:
['New', 'York', 'City', 'is', 'known', 'for', 'its', 'fast', 'pace', 'and', 'diversity', '.', 'He', 'finally', 'kicked', 'the', 'bucket', 'a
```

- Treating MWEs as single tokens helps NLP systems preserve **semantic integrity**, improve **named entity recognition**, and enhance **syntactic parsing**.

4. Reflection (5–6 sentences)

- What was the hardest part of tokenization in your language?
- How does it compare with tokenization in English?
- Do punctuation, morphology, and MWEs make tokenization more difficult?

```
# Example Telugu sentence
# Translation: "I live in Hyderabad city, and sometimes I do 'puli meeda panthulu' as a joke."
text = "నేను హైదరాబాద్ నగరంలో ఉంటాను, కొన్నిసార్లు సరదాగా పులి మీద పంతులు చేస్తాను!"
tokens_naive = text.split()
print("Naive Tokenization:", tokens_naive)
import re
tokens_cleaned = [re.sub(r'[^\w\u0C00-\u0C7F]', '', token) for token in tokens_naive if token]
corrected_tokens = []
skip = 0
for i, token in enumerate(tokens_cleaned):
    if skip > 0:
        skip -= 1
        continue
    if i+1 < len(tokens_cleaned) and tokens_cleaned[i] == "హైదరాబాద్" and tokens_cleaned[i+1].startswith("నగరంలో"):
        corrected_tokens.append("హైదరాబాద్_నగరంలో")
        skip = 1
    elif i+2 < len(tokens_cleaned) and tokens_cleaned[i:i+3] == ["పులి", "మీద", "పంతులు"]:
        corrected_tokens.append("పులి_మీద_పంతులు")
        skip = 2
    else:
        corrected_tokens.append(token)
print("Corrected Tokens:", corrected_tokens)
```

```
Naive Tokenization: ['నేను', 'హైదరాబాద్', 'నగరంలో', 'ఉంటాను,', 'కొన్నిసార్లు', 'సరదాగా', 'పులి', 'మీద', 'పంతులు', 'చేస్తాను!']
Corrected Tokens: ['నేను', 'హైదరాబాద్_నగరంలో', 'ఉంటాను', 'కొన్నిసార్లు', 'సరదాగా', 'పులి_మీద_పంతులు', 'చేస్తాను']
```

Q3. Manual BPE on a toy corpus

3.1 Using the same corpus from class:

```
low low low low low lowest lowest newer newer newer newer newer newer wider
wider wider new new
```

1. Add the end-of-word marker _ and write the *initial vocabulary* (characters + _).
2. Compute bigram counts and perform the **first three merges** by hand:
    - Step 1: most frequent pair → merge → updated corpus snippet (show at least 2 lines).
    - Step 2: repeat.
    - Step 3: repeat.
3. After each merge, list the new token and the updated vocabulary.

```python
from collections import Counter

# 1. Corpus
corpus = "low low low low low lowest lowest newer newer newer newer newer newer wider wider wider new new"

# Add end-of-word marker "_"
words = corpus.split()
words = [list(w) + ["_"] for w in words]  # turn into list of characters + "_"

# Function to get bigram counts
def get_bigram_counts(words):
    bigram_counts = Counter()
    for word in words:
        for i in range(len(word) - 1):
            bigram_counts[(word[i], word[i+1])] += 1
    return bigram_counts

# Function to merge most frequent bigram
def merge_bigram(words, bigram):
    new_words = []
    first, second = bigram
    for word in words:
        new_word = []
        i = 0
        while i < len(word):
            if i < len(word) - 1 and word[i] == first and word[i+1] == second:
                new_word.append(first+second)  # merge
                i += 2
```

```
            else:
                new_word.append(word[i])
                i += 1
        new_words.append(new_word)
    return new_words

# Initial vocabulary
vocab = sorted(set(ch for word in words for ch in word))
print("Initial vocabulary:", vocab)

# Perform 3 merges
for step in range(1, 4):
    bigram_counts = get_bigram_counts(words)
    most_frequent = bigram_counts.most_common(1)[0][0]  # get most frequent bigram
    print(f"\nStep {step}: merging bigram {most_frequent}")

    words = merge_bigram(words, most_frequent)
    vocab = sorted(set(ch for word in words for ch in word))

    # Print first two lines of updated corpus
    snippet = [" ".join(word) for word in words[:10]]
    print("Updated corpus snippet (first few words):")
    for line in snippet:
        print(line)
    print("Updated vocabulary:", vocab)
```

```
Initial vocabulary: ['_', 'd', 'e', 'i', 'l', 'n', 'o', 'r', 's', 't', 'w']

Step 1: merging bigram ('e', 'r')
Updated corpus snippet (first few words):
l o w _
l o w _
l o w _
l o w _
l o w _
l o w e s t _
l o w e s t _
n e w er _
n e w er _
n e w er _
Updated vocabulary: ['_', 'd', 'e', 'er', 'i', 'l', 'n', 'o', 's', 't', 'w']

Step 2: merging bigram ('er', '_')
Updated corpus snippet (first few words):
l o w _
l o w _
l o w _
l o w _
l o w _
l o w e s t _
l o w e s t _
n e w er_
n e w er_
n e w er_
Updated vocabulary: ['_', 'd', 'e', 'er_', 'i', 'l', 'n', 'o', 's', 't', 'w']
```

```
Updated vocabulary: ['_', 'd', 'e', 'er_', 'i', 'l', 'n', 'o', 's', 't', 'w']

Step 3: merging bigram ('n', 'e')
Updated corpus snippet (first few words):
l o w _
l o w _
l o w _
l o w _
l o w _
l o w e s t _
l o w e s t _
ne w er_
ne w er_
ne w er_
Updated vocabulary: ['_', 'd', 'e', 'er_', 'i', 'l', 'ne', 'o', 's', 't', 'w']
```

3.2 — Code a mini-BPE learner

1. Use the classroom code above (or your own) to learn BPE merges for the toy corpus.
    - Print the top pair at each step and the evolving vocabulary size.
2. Segment the words: `new`, `newer`, `lowest`, `widest`, and one word you invent (e.g., `newestest`).
    - Include the subword sequence produced (tokens with _ where applicable).
3. In 5–6 sentences, explain:
    - How subword tokens solved the OOV (out-of-vocabulary) problem.
    - One example where subwords align with a meaningful morpheme (e.g., `er_` as English agent/comparative suffix).

```
rom collections import Counter

 1. Toy Corpus
orpus = "low low low low low lowest lowest newer newer newer newer newer newer wider wider wider new new"
ords = [list(w) + ["_"] for w in corpus.split()]  # add end-of-word markers

ef get_bigram_counts(words):
   counts = Counter()
   for word in words:
       for i in range(len(word)-1):
           counts[(word[i], word[i+1])] += 1
   return counts

ef merge_bigram(words, bigram):
   merged_words = []
   first, second = bigram
   for word in words:
       new_word = []
       i = 0
       while i < len(word):
           if i < len(word)-1 and word[i] == first and word[i+1] == second:
               new_word.append(first+second)
               i += 2
           else:
               new_word.append(word[i])
               i += 1
       merged_words.append(new_word)
   return merged_words
```

```
]
0s
    # Learn BPE merges
    steps = 10  # learn 10 merges for better segmentation
    for step in range(1, steps+1):
        bigram_counts = get_bigram_counts(words)
        if not bigram_counts:
            break
        most_frequent = bigram_counts.most_common(1)[0][0]
        words = merge_bigram(words, most_frequent)
        vocab = sorted(set(ch for word in words for ch in word))
        print(f"Step {step}: merge {most_frequent}, vocab size: {len(vocab)}")

    # Function to segment a new word
    def segment_word(word, merges):
        # Re-apply merges in learned order
        tokens = list(word) + ["_"]
        for (first, second) in merges:
            i = 0
            new_tokens = []
            while i < len(tokens):
                if i < len(tokens)-1 and tokens[i] == first and tokens[i+1] == second:
                    new_tokens.append(first+second)
                    i += 2
                else:
                    new_tokens.append(tokens[i])
                    i += 1
            tokens = new_tokens
        return tokens
```

```python
# Collect merges from training process
merges = []
words = [list(w) + ["_"] for w in corpus.split()]
for step in range(1, steps+1):
    bigram_counts = get_bigram_counts(words)
    if not bigram_counts:
        break
    most_frequent = bigram_counts.most_common(1)[0][0]
    merges.append(most_frequent)
    words = merge_bigram(words, most_frequent)

# Segment given words
test_words = ["new", "newer", "lowest", "widest", "newestest"]
print("\n--- Segmentation ---")
for w in test_words:
    print(f"{w} -> {' '.join(segment_word(w, merges))}")
```

```
Step 1: merge ('e', 'r'), vocab size: 11
Step 2: merge ('er', '_'), vocab size: 11
Step 3: merge ('n', 'e'), vocab size: 11
Step 4: merge ('ne', 'w'), vocab size: 11
Step 5: merge ('l', 'o'), vocab size: 10
Step 6: merge ('lo', 'w'), vocab size: 10
Step 7: merge ('new', 'er_'), vocab size: 11
Step 8: merge ('low', '_'), vocab size: 12
Step 9: merge ('w', 'i'), vocab size: 11
Step 10: merge ('wi', 'd'), vocab size: 10

--- Segmentation ---
new -> new _
newer -> newer_
lowest -> low e s t _
widest -> wid e s t _
newestest -> new e s t e s t _
```

3.3 — *Your language* (or English if you prefer)

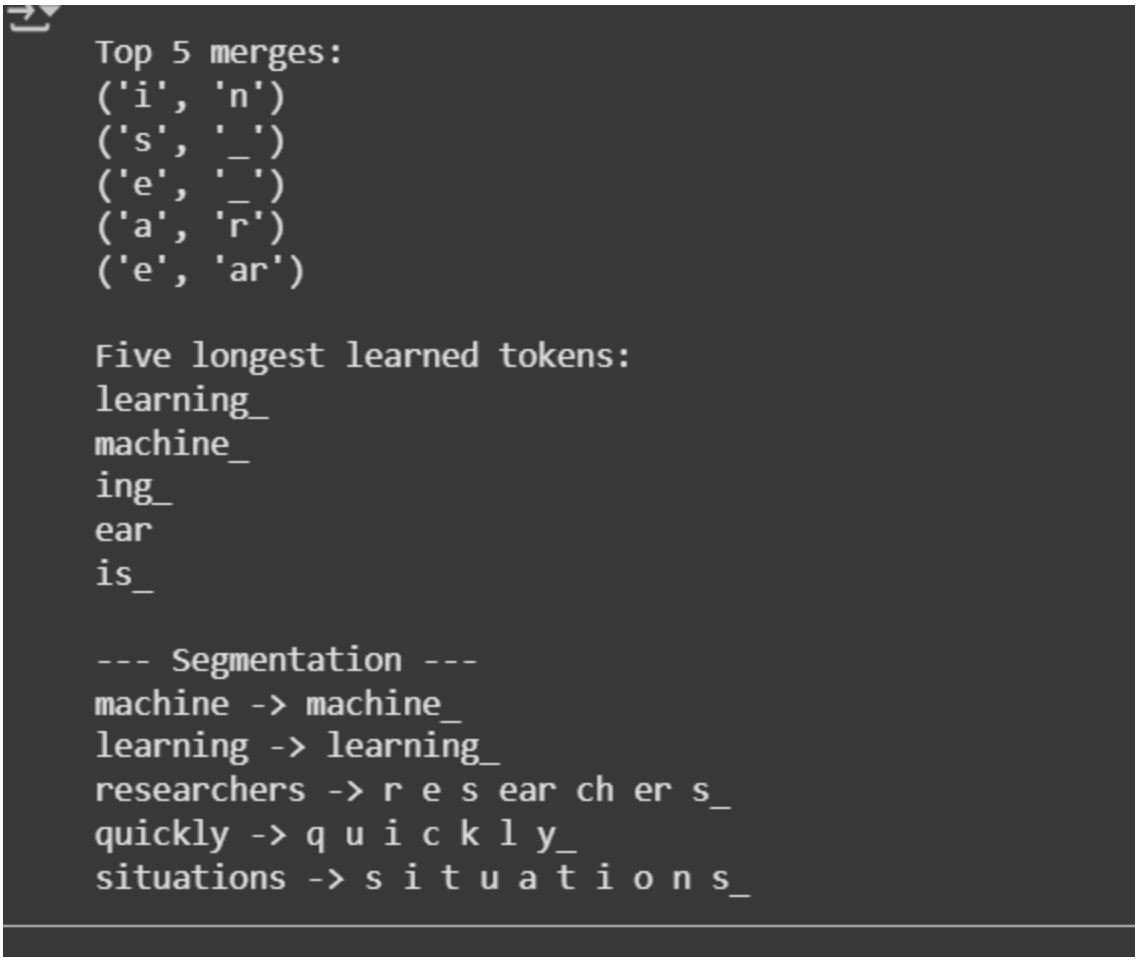
Pick one short paragraph (4–6 sentences) in *your own language* (or English if that's simpler).

1. Train BPE on that paragraph (or a small file of your choice).
    - Use end-of-word _.
    - Learn at least 30 merges (adjust if the text is very small).

2. Show the five most frequent merges and the resulting five longest subword tokens.
3. Segment 5 different words from the paragraph:
    - Include one rare word and one derived/inflected form.
4. Brief reflection (5–8 sentences):
    - What kinds of subwords were learned (prefixes, suffixes, stems, whole words)?
    - Two concrete pros/cons of subword tokenization for your language.

```python
from collections import Counter

# === Step 1: Small paragraph (English) ===
paragraph = """Machine learning is changing the world quickly.
Many companies use machine learning to solve problems.
Deep learning is a subfield of machine learning.
Newer models are trained on massive amounts of data.
Researchers keep improving algorithms every year.
Sometimes models fail in rare situations."""

# Preprocess: lowercase, split, add end-of-word marker "_"
words = [list(w) + ["_"] for w in paragraph.lower().replace(".", "").split()]

def get_bigram_counts(words):
    counts = Counter()
    for word in words:
        for i in range(len(word) - 1):
            counts[(word[i], word[i+1])] += 1
    return counts

def merge_bigram(words, bigram):
    merged_words = []
    first, second = bigram
    for word in words:
        new_word = []
        i = 0
        while i < len(word):
            if i < len(word)-1 and word[i] == first and word[i+1] == second:
```

```
                    new_word.append(first+second)
                    i += 2
                else:
                    new_word.append(word[i])
                    i += 1
            merged_words.append(new_word)
        return merged_words

    # === Step 2: Learn BPE merges (30 steps) ===
    merges = []
    for step in range(30):
        bigram_counts = get_bigram_counts(words)
        if not bigram_counts:
            break
        most_frequent = bigram_counts.most_common(1)[0][0]
        merges.append(most_frequent)
        words = merge_bigram(words, most_frequent)

    # Collect vocabulary after merges
    vocab = sorted(set(ch for word in words for ch in word))

    # === Step 3: Show top 5 merges & longest tokens ===
    print("\nTop 5 merges:")
    for m in merges[:5]:
        print(m)

    longest_tokens = sorted(vocab, key=lambda x: len(x), reverse=True)[:5]
    print("\nFive longest learned tokens:")
    for token in longest_tokens:
        print(token)

# === Step 4: Segment words ===
def segment_word(word, merges):
    tokens = list(word) + ["_"]
    for first, second in merges:
        i = 0
        new_tokens = []
        while i < len(tokens):
            if i < len(tokens)-1 and tokens[i] == first and tokens[i+1] == second:
                new_tokens.append(first+second)
                i += 2
            else:
                new_tokens.append(tokens[i])
                i += 1
        tokens = new_tokens
    return tokens

test_words = ["machine", "learning", "researchers", "quickly", "situations"]
print("\n--- Segmentation ---")
for w in test_words:
    print(f"{w} -> {' '.join(segment_word(w, merges))}")
```



Q4. Word Pair:

Sunday → Saturday

Tasks:

1. Find the minimum edit distance between *Sunday* and *Saturday* under both models:
   - Model A (Sub = 1, Ins = 1, Del = 1)
   - Model B (Sub = 2, Ins = 1, Del = 1)
   - Minimum Edit Distance

     Source: **Sunday**
     Target: **Saturday**

   Model A (Sub = 1, Ins = 1, Del = 1) → **distance = 3**.

Model B (Sub = 2, Ins = 1, Del = 1) → **distance = 4**.

2. Write out at least one valid edit sequence (step by step).

   We list the operations and show the intermediate string after each operation. Costs shown for each op.

   Start: Sunday

1. **Insert** 'a' after S → Saunday (cost: Ins = 1)
2. **Insert** 't' after Sa → Satunday (cost: Ins = 1)
3. **Substitute** n → r → Saturday (cost: Sub = 1 in Model A; Sub = 2 in Model B)

   So the operation sequence is:
   match S → ins a → ins t → match u → sub n→r → match d → match a → match y.

   Total cost calculation

- Model A: 1 (ins a) + 1 (ins t) + 1 (sub n→r) = 3.
- Model B: 1 + 1 + 2 = 4.

3. Reflect (4–5 sentences):
   - Did both models give the same distance?

   No the two models gave **different distances**.
   Model **A** (Substitution = 1) gave a minimum edit distance of **3**,
   while Model **B** (Substitution = 2) gave a minimum edit distance of **4**
   because substitutions are more expensive in Model B.

   - Which operations (insert/delete/substitute) were most useful here?

   Insertions were the most useful operations here — we needed two of them (a and t) to turn **Sunday** into **Saturday**.
   We only needed **one substitution** (n → r), and no deletions were required.
   So, insertions contributed most of the cost to reach the target word.

   - How would the choice of model affect applications like spell check vs. DNA alignment?
   - The cost model changes which edits are preferred.
     In spell checking, we usually expect typos to be single-character substitutions, so giving substitutions a **low cost** helps us find the correct word quickly.
     In DNA alignment, however, insertions and deletions (gaps) are biologically

common, so those should have **lower costs**, and substitutions should be more expensive to avoid aligning mismatched bases unless necessary.
This way, the algorithm finds alignments or suggestions that are realistic for the data we are working with.