University of Central Missouri

Department of Computer Science & Cybersecurity

CS5760 Natural Language Processing

Fall 2025

Homework 1.

Student name:

Sai Siva Shankara Vara Prasad Kopparthi

Student ID: 700765221

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). Make sure you only match whole tokens (not inside longer strings).

   A) r'\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`.

   A) r'\b(?![A-Z])[A-Za-z'\'-]+\b'

3. **Convenient aliases** (numbers, a bit richer)
   Extract all numbers that may have:
   a. optional sign (+/−),
   b. optional thousands separators (commas),
   c. optional decimal part,
   d. optional scientific notation (e.g., `1.23e-4`).

   A) r'[+-]?\d{1,3}(?:,\d{3})*(?:\.\d+)?(?:[eE][+-]?\d+)?'

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.

   A) r'(?i)e[--]?mail'

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!`).

   A) r'\bgo+[,!?.]?\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.

   A) r'\?['")\]\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.

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?

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.

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?

A) Code: Code is submitted in Homework1.ipynb file.

Reflection:

- Tokenization in Telugu is harder than English because of compound words and rich morphology.
- Naïve splitting by spaces often leaves punctuation attached to words.
- Manual correction improves results, but still misses some cases.
- Tools like NLTK handle Unicode but may not fully capture Telugu MWEs, so extra care is needed.

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.

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).

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

Answer)

3.1 Using the toy corpus

Corpus:

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

1. Add end-of-word marker _ and write initial vocabulary

```
l o w _         : 5
l o w e s t _   : 2
n e w e r _     : 6
w i d e r _     : 3
n e w _         : 2
```

2. Compute bigram counts and perform first three merges

- **Step 1:** merge `e r` → new token `er`
- `newer → n e w er _`
- `wider → w i d er _`
- **Step 2:** merge `er _` → new token `er_`
- `newer → n e w er_`
- `wider → w i d er_`
- **Step 3:** merge `n e` → new token `ne`
- `newer → ne w er_`
- `new → ne w _`

3. After each merge, list new token and updated vocabulary

```
After Step 1: er
After Step 2: er_
After Step 3: ne
```

Updated vocabulary after 3 merges:

```
l o w _         : 5
l o w e s t _   : 2
ne w er_        : 6
w i d er_       : 3
ne w _          : 2
```

3.2 Mini-BPE Learner

1. Top pair at each step and evolving vocabulary size (first 10 merges):

```
1. ('e','r')
2. ('er','_')
```

```
3. ('n','e')
4. ('ne','w')
5. ('l','o')
6. ('lo','w')
7. ('new','er_')
8. ('low','_')
9. ('w','i')
10. ('wi','d')
```

2. Segment the words:

```
new       → ['new', '_']
newer     → ['newer_']
lowest    → ['low', 'e', 's', 't', '_']
wider     → ['wid', 'er_']
newestest → ['new', 'e', 's', 't', 'e', 's', 't', '_']
```

3. Reflection (5–6 sentences):

1. Subword tokens solve the OOV problem by splitting unknown words into familiar parts.
2. For example, `er_` aligns with the English comparative/agent suffix.
3. This reduces vocabulary size while keeping coverage.
4. It also allows rare words like “newestest” to be represented using common subunits.
5. A benefit is better generalization, but sometimes BPE splits words into unnatural fragments.

3.3 BPE on a Paragraph

Paragraph (English, 5 sentences):
"Natural language processing studies how to teach computers to understand and generate human language. Tokenization and subword segmentation are important preprocessing steps. Byte-Pair Encoding is a popular subword algorithm because it balances vocabulary size and coverage. Subwords help handle rare words and morphological variants. This example shows how BPE merges common character sequences."

1. Five most frequent merges:

```
1. ('a', 'n')
2. ('e', '_')
3. ('s', '_')
4. ('d', '_')
5. ('o', 'r')
```

2. Five longest subword tokens:

```
subwor
and_
```

```
age_
wor
how
```

3. Segment 5 different words:

```
processing    → ['pr', 'o', 'ce', 's', 'si', 'ng_']
tokenization  → ['to', 'k', 'en', 'i', 'z', 'at', 'i', 'on_']
subword       → ['subwor', 'd_']
morphological → ['m','or','p','ho','l','o','g','i','c','al','_']
coverage      → ['co','v','er','age_']
```

4. Reflection (5–8 sentences):

1. BPE learned frequent prefixes (e.g., "sub"), suffixes (e.g., "age_"), and stems (e.g., "wor").
2. This shows that BPE often captures morpheme-like units in real text.
3. A key advantage is that it reduces vocabulary size while handling rare or derived words.
4. It also generalizes better to unseen words by combining familiar subunits.
5. A drawback is that meaningful words may be split into too many pieces, increasing sequence length.
6. Another issue is that some subwords are arbitrary fragments with little meaning.
7. For English this tradeoff works well, but in highly agglutinative languages the splits may become excessive.

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)
2. Write out at least one valid edit sequence (step by step).
3. Reflect (4–5 sentences):
   - Did both models give the same distance?
   - Which operations (insert/delete/substitute) were most useful here?
   - How would the choice of model affect applications like spell check vs. DNA alignment?

Answer)

1) **Minimum edit distance (Model A: Sub=1, Ins=1, Del=1)**

   Target: Sunday → Saturday

   One possible sequence:

   1. `Sunday` → `S**a**unday` (Insert **a**)
   2. `Saunday` → `Saturnday` (Insert **tur**)
   3. `Saturnday` → `Saturday` (Delete extra **n**)

   Total number of edits are 3
   For Model A the Distance is 3

2) **Minimum edit distance (Model B: Sub=2, Ins=1, Del=1)**

   Here substitutions are more expensive, so insertions/deletions are preferred.

   One possible sequence:

   1. `Sunday` → `S**a**unday` (Insert **a**)
   2. `Saunday` → `Saturnday` (Insert **tur**)
   3. `Saturnday` → `Saturday` (Delete **n**)

   Total number of edits are 4
   For Model B the Distance is 4

3) **Reflection**:

   1. Model A result: Distance = 3.
   2. Model B result: Distance = 4.
   3. The difference comes from the fact that substitution is cheaper in Model A, so the algorithm takes fewer steps.
   4. In Model B, substitution is costly, leading to more insertions and deletions.
   5. This is important in real-world applications:
      - Spell check systems should use cheap substitutions since typos often replace one character with another.
      - DNA and protein sequence alignment should favor cheaper insertions and deletions because mutations typically involve insertions or deletions rather than substitutions.