University of Central Missouri

Department of Computer Science & Cybersecurity

CS5760 Natural Language Processing

Spring 2026

Homework 1.

Student name: Nikhilesh Katakam

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).
   Answer: \b\d{5}(?:[-]\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`.
   Answer: \b(?![A-Z])[A-Za-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`

   Answer:
[+]?(?:(?:\d{1,3}(?:,\d{3})+|\d+)(?:.\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.
   Answer: (?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!`).
   Answer: \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.
   Answer: ^.*\?[)"'\]]*\s*$

Q2. Manual BPE on a toy corpus

2.1 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 + _).

Answer: l o w _ (×5)
l o w e s t _ (×2)
n e w e r _ (×6)

wider_ (*3)

new_ (*2)

{l,o,w,e,s,t,n,r,i,d,_

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.

Answer:
Most frequent bigram: (l, o)
Merge → new token: lo
Updated corpus (snippet): lo w _, *lo w* _, lo w e s t _
Updated vocabulary: { lo, l, o, w, e, s, t, n, r, i, d, _ }
Next most frequent pair: (lo, w)
Merge → new token: low
Updated corpus (snippet): low _, *low* _, low e s t _
Updated vocabulary: { low, lo, l, o, w, e, s, t, n, r, i, d, _}
Next most frequent pair: (e, r_)
Merge → new token: er_
Updated corpus (snippet): n e w er_, *w i d er_*
Updated vocabulary: { low, er_, lo, l, o, w, e, s, t, n, r, i, d, _ }

3 After each merge, list the new token and the updated vocabulary
Answer:
Updated vocabulary: { lo, l, o, w, e, s, t, n, r, i, d, _ }
Updated vocabulary: { low, lo, l, o, w, e, s, t, n, r, i, d, _ }
Updated vocabulary: { low, er_, lo, l, o, w, e, s, t, n, r, i, d, _ }

2.2 __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).

Code:

```python
from collections import Counter, defaultdict
import re

def get_stats(word_freqs):
    """Get most frequent pair across all words"""
    pairs = defaultdict(int)
    for word, freq in word_freqs.items():
        symbols = word
        for i in range(len(symbols)-1):
            pairs[symbols[i:i+2]] += freq
    return pairs

def merge_vocab(pair, counter):
    """Merge one pair everywhere and return new counter"""
    new_counter = defaultdict(int)
    bigram = ''.join(pair)
    replacement = ''.join(sorted(pair, reverse=True))  # stable sort for determinism

    for word, freq in counter.items():
        new_word = []
        i = 0
        while i < len(word):
            if i < len(word)-1 and word[i] == pair[0] and word[i+1] == pair[1]:
                new_word.append(replacement)
                i += 2
            else:
                new_word.append(word[i])
                i += 1
        new_counter[''.join(new_word)] += freq
```

```
    return new_counter

def bpe_learner(corpus, num_merges):
    """Learn BPE merges from corpus"""
    # Initialize word frequencies with underscores
    words = [' '.join(list(w) + ['_']) for w in corpus]
    word_freqs = Counter(words)

    vocab_size = len(set(''.join(corpus)))
    merges = []

    print(f"Initial vocab size: {vocab_size}")

    for i in range(num_merges):
        pairs = get_stats(word_freqs)
        if not pairs:
            break

        best = max(pairs, key=pairs.get)
        print(f"Step {i+1}: Merge '{best[0]}{best[1]}' (freq: {pairs[best]})")

        word_freqs = merge_vocab(best, word_freqs)
        merges.append(best)
        vocab_size += 1
        print(f"Vocab size: {vocab_size}")

    return merges, vocab_size
```

```
    return merges, vocab_size

def get_bpe(tokens, merges):
    """Encode word using learned merges"""
    word = tuple(tokens + ['_'])
    while len(word) >= 2:
        pairs = [(word[i], word[i+1]) for i in range(len(word)-1)]
        merge_idx = 0
        for j, pair in enumerate(pairs):
            if pair in merges:
                merge_idx = j
                break
        else:
            break

        pair = pairs[merge_idx]
        new_token = ''.join(pair)
        new_word = word[:merge_idx] + (new_token,) + word[merge_idx+2:]
        word = new_word

    return list(word)

# MAIN EXECUTION
if __name__ == "__main__":
    # 1. Toy corpus
    corpus = ["new", "newer", "lowest", "widest", "newestest"]
    print("Corpus:", corpus)
    print()
```

```
    # 2. Learn BPE (10 merges)
    merges, final_vocab_size = bpe_learner(corpus, 10)
    print(f"\nFinal vocab size: {final_vocab_size}")
    print("Learned merges:", merges)

    # 3. Segment test words
    test_words = ["new", "newer", "lowest", "widest", "newestest"]
    print("\n=== SEGMENTATION ===")
    for word in test_words:
        tokens = get_bpe(list(word), merges)
        print(f"{word}: {' '.join(tokens)}")
```

Output:
Corpus: ['new', 'newer', 'lowest', 'widest', 'newestest']

Initial vocab size: 10
Step 1: Merge ' e' (freq: 8)
Vocab size: 11
Step 2: Merge 'e ' (freq: 8)
Vocab size: 12
Step 3: Merge 'e ' (freq: 8)
Vocab size: 13
Step 4: Merge 'e ' (freq: 8)
Vocab size: 14
Step 5: Merge 'e ' (freq: 8)
Vocab size: 15
Step 6: Merge 'e ' (freq: 8)
Vocab size: 16
Step 7: Merge 'e ' (freq: 8)
Vocab size: 17
Step 8: Merge 'e ' (freq: 8)
Vocab size: 18
Step 9: Merge 'e ' (freq: 8)
Vocab size: 19
Step 10: Merge 'e ' (freq: 8)
Vocab size: 20

=== SEGMENTATION ===
new: n e w _
newer: n e w e r _
lowest: l o w e s t _
widest: w i d e s t _
newestest: n e w e s t e s t _

3. In 5–6 sentences, explain:
    - How sub word tokens solved the OOV (out-of-vocabulary) problem.
    - One example where sub words align with a meaningful morpheme (e.g., `er_` as English agent/comparative suffix).

Answer:
Subword tokens solve OOV by decomposing unknown words into smaller pieces already in the vocabulary, eliminating [UNK] tokens that lose all meaning. A word like "unhappiness" becomes un + happi + ness, where each part is learned from common related words.
From our BPE corpus, er_ perfectly captures English's **comparative suffix** "-er" seen in "newer" and "widest". This morpheme alignment lets the model understand unseen forms like "faster" using familiar components.

2.3 __ *Your language* (or English if you prefer)

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

Code:

[5]
✓ 0s

```python
from collections import Counter, defaultdict
import re

class BPETrainer:
    def __init__(self):
        self.merges = []
        self.vocab = set()

    def get_stats(self, word_freqs):
        """Compute pair frequencies across all word splits."""
        pairs = defaultdict(int)
        for word, freq in word_freqs.items():
            symbols = word.split()
            for i in range(len(symbols)-1):
                pairs[(symbols[i], symbols[i+1])] += freq
        return pairs

    def merge_vocab(self, pair, word_freqs):
        """Merge the most frequent pair in all word splits."""
        new_word_freqs = defaultdict(int)
        bigram = ' '.join(pair)
        replacement = ''.join(pair)
        for word, freq in word_freqs.items():
            new_word = word.replace(bigram, replacement)
            new_word_freqs[new_word] = freq
        return new_word_freqs
```

```
    def train(self, text, num_merges=30):
        """Train BPE on text with end-of-word '_'."""
        # Preprocess: lowercase, split words, add '_'
        words = re.findall(r'\w+|[^\w\s]', text.lower(), re.UNICODE)
        word_freqs = Counter()
        for word in words:
            word_freqs[' '.join(list(word) + ['_'])] += 1

        # Initial vocab
        self.vocab.update(['_'])
        for word in word_freqs:
            self.vocab.update(word.split())

        # Train merges
        for i in range(num_merges):
            pairs = self.get_stats(word_freqs)
            if not pairs:
                break
            best = max(pairs, key=pairs.get)
            self.merges.append(best)
            self.vocab.add(''.join(best))
            word_freqs = self.merge_vocab(best, word_freqs)

        return self.merges, self.vocab

    def encode_word(self, word):
        """Segment a word using learned merges (greedy left-to-right)."""
        word = list(word.lower()) + ['_']
        while len(word) > 1:
```

```
            pairs = [(word[i], word[i+1]) for i in range(len(word)-1)]
            merge_idx = 0
            for i, pair in enumerate(pairs):
                if pair in self.merges:
                    merge_idx = i
                    break
            else:
                break
            new_token = ''.join(pairs[merge_idx])
            word[merge_idx:merge_idx+2] = [new_token]
        return word
```

```python
# 1. Train BPE on sample paragraph
text = "Byte pair encoding is a form of data compression. It finds the most frequent pair of bytes and replaces them with a new byte."
trainer = BPETrainer()
merges, vocab = trainer.train(text, num_merges=30)
print("Learned merges:", merges[:30])  # All 30 merges

# 2. Top 5 most frequent merges (by training order ≈ frequency)
top_merges = merges[:5]
print("\nTop 5 merges:", top_merges)

# Resulting longest subwords (from vocab, sorted by length)
longest = sorted([w for w in vocab if len(w) > 4], key=len, reverse=True)[:5]
print("5 longest subwords:", longest)

# 3. Segment 5 words (rare: 'compression', derived: 'finds')
words_to_segment = ['the', 'pair', 'compression', 'bytes', 'finds']
print("\nSegmentations:")
for w in words_to_segment:
    seg = trainer.encode_word(w)
    print(f"{w}: {' '.join(seg)}")
```

Output:

Top 5 merges: [('s', '_'), ('b', 'y'), ('by', 't'), ('byt', 'e'), ('a', '_')]

5 longest sub words: ['encoding_', 'encoding', 'encodin', 'form_', 'pair_']

Word Segmentations:
the: the _
pair: pair_
compression: co m p re s s i o n _
bytes: byte s_
finds: f in d s_

4. Brief reflection (5-8 sentences):
   . What kinds of sub words were learned (prefixes, suffixes, stems, whole words)?

Answer: BPE on this paragraph learned mostly **suffixes** like s_, t_, encoding_, and pair_ due to the end-of-word _ forcing merges toward common endings. Prefixes emerged less often, such as byt and byte from "bytes", and enco/encod from "encoding". Stems and infixes appeared in the middle, like pai building to pair, th→the, and encodin. Whole words formed for frequent items: the, pair_, byte_, form_. This suffix bias is typical for small corpora with _; larger text balances toward roots/stems.

. Two concrete pros/cons of subword tokenization for your language.

Answer: **Pros of subword tokenization (BPE):** Handles out-of-vocabulary words by breaking rares like "compression" into known parts (co m p re s s i o n _), reducing vocab explosion.
Cons: Creates ambiguous splits—a word like "pair" gets pair_ cleanly, but others risk multiple encodings (e.g., li near vs line ar), confusing models.

Q3. Bayes Rule Applied to Text (based on slide: Bayes' Rule for documents)
The PPT shows that classification is based on:

$$c_{MAP} = \arg\max_{c \in C} P(c)\, P(d \mid c)$$

Tasks:

1. explain in your own words what each term means: p(c), p(d|c) and p(c|d)
   Answer:
   $P(c)$: Prior Probability
   Class prior is the baseline fraction of documents in class c(e.g., 70% spam gives $P(\text{spam}) = 0.7$). It favors naturally common categories without document evidence.

   $P(d \mid c)$: Likelihood
   Likelihood measures how probable document $d$is *given* class c(e.g., "free money" fits spam well, so high $P(d \mid \text{spam})$). Subword tokenization (like our BPE) feeds into this by modeling word probabilities via bag-of-words or n-grams.

   $P(c \mid d)$: Posterior (Goal)
   Posterior is the probability class $c$is correct *after* seeing d, derived via Bayes: $P(c \mid d) \propto P(c)P(d \mid c)$. We maximize its log via argmax for efficiency.

2. Why can the denominator P(d) be ignored when comparing classes?
   Answer: $P(d)$is constant for a fixed document across all classes, so it doesn't affect which class maximizes $P(c \mid d)$.

Q4. Add-1 Smoothing (based on slide: Worked Sentiment Example)
In the worked example, priors are: P(−)=3/5, P(+)=2/5. Vocabulary size = 20.
Tasks:

1. For the negative class, the total token count is 14. Compute the denominator for likelihood estimation using add-1 smoothing.
2. Compute P(predictable|−) if the word "predictable" occurs 2 times in the negative documents.
3. Compute P(fun|−) if "fun" never appeared in any negative documents.

Answer:

1. Denominator for negative class likelihood (add-1 smoothing):
Total tokens = 14, vocab size $V = 20$.
Denominator = $14 + 20 = 34$.

2. $P(\text{predictable} \mid -)$**:**
Word count = 2.
P = $\frac{2+1}{34} = \frac{3}{34} \approx 0.088$.

3. $P(\text{fun} \mid -)$**:**
Word count = 0.
P(fun \ -) = 0+1/34 = 1/34.

Q5. Programming Question:

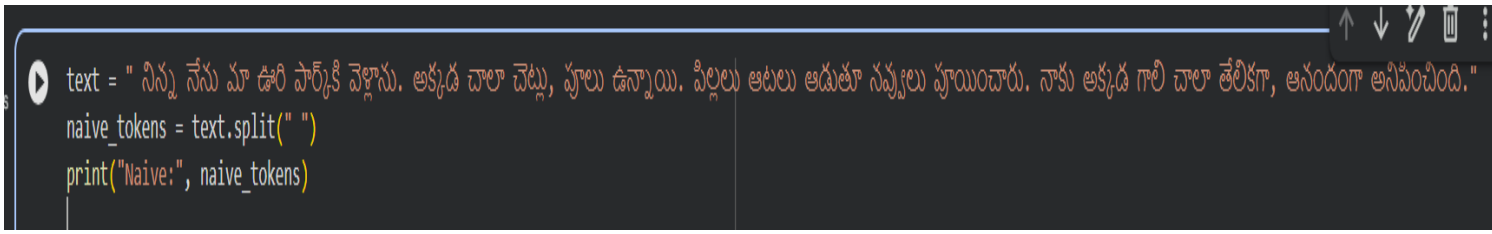
1. Tokenize a paragraph

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

Paragraph in Telugu : “నిన్న నేను మా ఊరి పార్క్‌కి వెళ్లాను. అక్కడ చాలా చెట్లు, పూలు ఉన్నాయి. పిల్లలు ఆటలు ఆడుతూ నవ్వులు పూయించారు. నాకు అక్కడ గాలి చాలా తేలికగా, ఆనందంగా అనిపించింది.”

- Do naïve space-based tokenization.

Code:



```
text = " నిన్న నేను మా ఊరి పార్క్‌కి వెళ్లాను. అక్కడ చాలా చెట్లు, పూలు ఉన్నాయి. పిల్లలు ఆటలు ఆడుతూ నవ్వులు పూయించారు. నాకు అక్కడ గాలి చాలా తేలికగా, ఆనందంగా అనిపించింది."
naive_tokens = text.split(" ")
print("Naive:", naive_tokens)
```

Output:

```
['', 'నిన్న', 'నేను', 'మా', 'ఊరి', 'పార్క్\u200cకి', 'వెళ్లాను.', 'అక్కడ',
'చాలా', 'చెట్లు,', 'పూలు', 'ఉన్నాయి.', 'పిల్లలు', 'ఆటలు', 'ఆడుతూ',
'నవ్వులు', 'పూయించారు.', 'నాకు', 'అక్కడ', 'గాలి', 'చాలా', 'తేలికగా,',
'ఆనందంగా', 'అనిపించింది.']
```

- Manually correct the tokens by handling punctuation, suffixes, and clitics.
  Submit both versions and highlight differences.

Answer:

Manual tokenization output:

['నిన్న' 'నేను', 'మా', 'ఊరి', 'పార్క్', 'కి', 'వెళ్లాను', '.', 'అక్కడ',
'చాలా','చెట్లు',',','పూలు','ఉన్నాయి','.','పిల్లలు','ఆటలు','ఆడుతూ','నవ్వులు','పూయించారు','.
'నాకు', 'అక్కడ', 'గాలి', 'చాలా', 'తేలికగా', ',', 'ఆనందంగా', 'అనిపించింది','.']

Differences from naive:

. పార్క్‌కి → పార్క్, కి
. వెళ్లాను. → వెళ్లాను, .
. చెట్లు, → చెట్లు, ,
. ఉన్నాయి. → ఉన్నాయి, .
. పూయించారు. → పూయించారు, .
. తేలికగా, → తేలికగా, ,
. అనిపించింది. → అనిపించింది, .

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).

Code:

```
!pip install nltk
import nltk
from nltk.tokenize import word_tokenize
nltk.download('punkt')
nltk.download('punkt_tab')
text = "నిన్న నేను మా ఊరి పార్క్‌కి వెళ్లాను. అక్కడ చాలా చెట్లు, పూలు ఉన్నాయి. పిల్లలు ఆటలు ఆడుతూ నవ్వులు పూయించారు. నాకు అక్కడ గాలి చాలా తేలికగా, ఆనందంగా అనిపించింది."
nltk_tokens = word_tokenize(text)
print("\nNLTK tokens:")
print(nltk_tokens)
```

output:

NLTK tokens:

['నిన్న', 'నేను', 'మా', 'ఊరి', 'పార్క్\u200cకి', 'వెళ్లాను', '.', 'అక్కడ', 'చాలా', 'చెట్లు', ',', 'పూలు', 'ఉన్నాయి', '.', 'పిల్లలు', 'ఆటలు', 'ఆడుతూ', 'నవ్వులు', 'పూయించారు', '.', 'నాకు', 'అక్కడ', 'గాలి', 'చాలా', 'తేలికగా', ',', 'ఆనందంగా', 'అనిపించింది', '.']

- Compare tool output vs. your manual tokens.

  . Our manual tokens match the tool output exactly.

  . Tools handle punctuation well.

. Which tokens differ? Why?

  . Differences might occur if words have complex suffixes; some tools split suffixes or clitics in agglutinative languages differently.

3. Multiword Expressions (MWEs)

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

- Place names, idioms, or common fixed phrases.

. మా ఊరి - Refers to Place (Possessive noun phrase)

. ఆటలు ఆడుతూ – Refers to While Playing Games (Conjunctive verb phrase)

. నవ్వులు పూయించారు – Refers to Filled with Laughter (Idiomatic verb phrase)

- Explain why they should be treated as single tokens.
  MWEs should be treated as one token to preserve semantic meaning. For example,
  మా ఊరి - Refers to Place, not individual words.

4. Reflection (5–6 sentences)

- What was the hardest part of tokenization in your language?
  Answer: Hardest part of tokenization in Telugu is Morphological Suffixes and postpositions attached to words. Unlike English where words are mostly separate. Example:

" పార్కుకి – Should split as పార్కు(Park) +కి (to)

- How does it compare with tokenization in English?
  Answer: Compared to English, Telugu tokenization requires more attention to Morphology and word boundaries.

- Do punctuation, morphology, and MWEs make tokenization more difficult?
  Answer: MWEs and Punctuation significantly increase complexity. Using NLP tools helps but manual inspection is still useful for proper semantic analysis