CS 4501 Natural Language Processing (Fall 2024)
University of Virginia
Instructor: Yu Meng
Assignment prepared by TAs: Zhepei Wei & Wenqian Ye

Assignment 2: N-gram Language Models (78 points)

Davis Wang

1. *Manually calculate N-gram Language Models* (15 pts).

```
[BOS] The cat sat on the mat [EOS]
[BOS] The mat is blue [EOS]
[BOS] The cat is also blue [EOS]
```

Given the above mini-corpus, answer the following questions. Note that:

- **The language models should be case-sensitive—for example, treating "The" and "the" as different words**;
- $p([\text{BOS}]) = 1$ **for bigram and trigram language models**.

(a) Unigram Model: Calculate the unigram probabilities for each word in the corpus. List all unigrams and their corresponding probabilities (3 pts).

- $p([\text{BOS}]) = \frac{3}{21}$
- $p(\text{"The"}) = \frac{3}{21}$
- $p([\text{EOS}]) = \frac{3}{21}$
- $p(\text{"mat"}) = \frac{2}{21}$
- $p(\text{"is"}) = \frac{2}{21}$
- $p(\text{"cat"}) = \frac{2}{21}$
- $p(\text{"sat"}) = \frac{1}{21}$
- $p(\text{"on"}) = \frac{1}{21}$
- $p(\text{"the"}) = \frac{1}{21}$
- $p(\text{"blue"}) = \frac{1}{21}$
- $p(\text{"also"}) = \frac{1}{21}$

(b) Bigram Model: Using a bigram language model to compute the probabilities:

- $p(\text{"The"}|[\text{BOS}]) = \frac{p(\text{"[BOS] The"})}{p([\text{BOS}])} = 1$
- $p(\text{"cat"}|\text{"The"}) = \frac{p(\text{"The cat"})}{p(\text{"The"})} = \frac{2}{3}$
- $p(\text{"blue"}|\text{"is"}) = \frac{p(\text{"is blue"})}{p(\text{"is"})} = \frac{1}{2}$

Show your calculations and the resulting probabilities (3 pts).

(c) Trigram Model: Calculate the trigram probability for the following sequence:

$$
\begin{aligned}
&p(\text{“[BOS] The mat is blue [EOS]”}) = \\
&\quad p([\text{BOS}]) \cdot p(\text{“The”} \mid [\text{BOS}]) \cdot p(\text{“mat”} \mid [\text{BOS}], \text{“The”}) \cdot \\
&\quad p(\text{“is”} \mid \text{“The”}, \text{“mat”}) \cdot p(\text{“blue”} \mid \text{“mat”}, \text{“is”}) \cdot p([\text{EOS}] \mid \text{“is”}, \text{“blue”}) \\
&= 1 \cdot 1 \cdot \frac{1}{3} \cdot 1 \cdot 1 \cdot 1 = \frac{1}{3}
\end{aligned}
$$

Show your calculations and the resulting probabilities (3 pts).

(d) Smoothing: Given the following sequence, compute the bigram probabilities with **AND** without add-one smoothing.

$$
\begin{aligned}
&p(\text{“[BOS] The cat sat on the blue mat [EOS]”}) = \\
&\quad p([\text{BOS}]) \cdot p(\text{“The”} \mid [\text{BOS}]) \cdot p(\text{“cat”} \mid \text{“The”}) \cdot \\
&\quad p(\text{“sat”} \mid \text{“cat”}) \cdot p(\text{“on”} \mid \text{“sat”}) \cdot p(\text{“the”} \mid \text{“on”}) \cdot \\
&\quad p(\text{“blue”} \mid \text{“the”}) \cdot p(\text{“mat”} \mid \text{“blue”}) \cdot p([\text{EOS}] \mid \text{“mat”})
\end{aligned}
$$

Without smoothing:

$$
\begin{aligned}
&= 1 \cdot 1 \cdot \frac{2}{3} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot 0 \cdot 0 \cdot \frac{1}{2} \\
&= 0
\end{aligned}
$$

With smoothing:

$$
\begin{aligned}
&= 1 \cdot \frac{3+1}{3+11} \cdot \frac{2+1}{3+11} \cdot \frac{1+1}{2+11} \cdot \frac{1+1}{1+11} \cdot \frac{1+1}{1+11} \cdot \frac{1}{1+11} \cdot \frac{1}{2+11} \cdot \frac{1+1}{2+11} \\
&= 2.5803068914 \times 10^{-7} \\
&\ln(0.00000100175306787) = -15.1701873089
\end{aligned}
$$

Show your calculations and the resulting probabilities (present the final result in natural log-scale if the resulting probability is not zero) (6 pts).

2. *Python Implementation of N-gram Language Models* (18 pts).
Follow the instructions in this Jupyter Notebook to download the specified dataset and split it into training and test sets.

   (a) Implement training for N-gram language models and compute the Unigram/Bigram/ Trigram probabilities (15 pts).

   Copy your code script and printed results to answer the question here.

   > Code script:

```
def train_unigram_lm(corpus):
    unigram_counts = {}
    unigram_model = {}

    for sentence in corpus:
        for word in sentence:
            if word not in unigram_counts:
                unigram_counts[word] = 1
            else:
                unigram_counts[word] += 1

    total_words = sum(unigram_counts.values())

    # TODO: Compute unigram probabilities based on unigram counts
    for word, count in unigram_counts.items():
        unigram_model[word] = count / total_words

    return unigram_counts, unigram_model

unigram_counts, unigram_model = train_unigram_lm(train_corpus)

def train_bigram_lm(corpus):
    unigram_counts, _ = train_unigram_lm(corpus)
    bigram_counts = {}
    bigram_model = {}

    for sentence in corpus:
        for i in range(len(sentence) - 1):
            bigram = (sentence[i], sentence[i + 1])
            if bigram not in bigram_counts:
                bigram_counts[bigram] = 1
            else:
                bigram_counts[bigram] += 1

    # TODO: Compute bigram probabilities based on bigram counts
    for bigram, count in bigram_counts.items():
        bigram_model[bigram] = count / unigram_counts[bigram[0]]

    return bigram_counts, bigram_model
```

```
bigram_counts, bigram_model = train_bigram_lm(train_corpus)

def train_trigram_lm(corpus):
    bigram_counts, _ = train_bigram_lm(corpus)
    trigram_counts = {}
    trigram_model = {}

    for sentence in corpus:
        for i in range(len(sentence) - 2):
            trigram = (sentence[i], sentence[i + 1], sentence[i + 2])
            if trigram not in trigram_counts:
                trigram_counts[trigram] = 1
            else:
                trigram_counts[trigram] += 1

    # TODO: Compute trigram probabilities based on trigram counts
    for trigram, count in trigram_counts.items():
        bigram = (trigram[0], trigram[1])
        if bigram in bigram_counts:
            trigram_model[trigram] = count / bigram_counts[bigram]
        else:
            trigram_model[trigram] = 0  # add smoothing here??

    return trigram_counts, trigram_model

trigram_counts, trigram_model = train_trigram_lm(train_corpus)

# Calculate the probability of a sample sentence using the Unigram, Bigram, and

def calulate_unigram_prob(sentence, unigram_model):
    for idx in range(len(sentence)):
        if idx == 0:
            prob = 1 # assume the sentence always starts with [BOS]
        else:
            prob *= unigram_model.get(sentence[idx], 0)
    return prob


def calculate_bigram_prob(sentence, bigram_model):
    for idx in range(len(sentence)):
        if idx == 0:
            prob = 1 # assume the sentence always starts with [BOS]
        else:
            prob *= bigram_model.get((sentence[idx-1], sentence[idx]), 0)
    return prob
```

```
def calculate_trigram_prob(sentence, trigram_model, bigram_model):
    for idx in range(len(sentence)):
        if idx == 0:
            prob = 1 # assume the sentence always starts with [BOS]
        elif idx == 1:
            prob *= bigram_model.get((sentence[0], sentence[1]), 0)
        else:
            prob *= trigram_model.get((sentence[idx-2], sentence[idx-1], senten
    return prob

sample_sentence = train_corpus[0]
print(f'Sample sentence: {sample_sentence}')

# Test the Unigram, Bigram, and Trigram models
unigram_prob = calulate_unigram_prob(sample_sentence, unigram_model)
bigram_prob = calculate_bigram_prob(sample_sentence, bigram_model)
trigram_prob = calculate_trigram_prob(sample_sentence, trigram_model, bigram_mo

print(f"Unigram probability: {unigram_prob}")
print(f"Bigram probability: {bigram_prob}")
print(f"Trigram probability: {trigram_prob}")

> Unigram probability: 7.605693014958497e-93
> Bigram probability: 1.7230259261125475e-28
> Trigram probability: 7.89141414141414e-08
```

(b) In one sentence, compare the probability of the given sample sentence estimated by the Unigram/Bigram/Trigram (*i.e.*, which probability is the highest and which is the lowest), and explain why this is the case (3 pts).

The Trigram probability is the highest and the Unigram probability is the lowest, because the unigram model does a very poor job of predicting proceedings words in a sequence, whereas the trigram model looks at previous tokens to attribute value to more than just one individual word at a time.

3. *Python Implementation of Smoothing Techniques* (21 pts).

(a) Using the previously trained N-gram models, implement basic smoothing techniques including add-k smoothing and interpolation for Bigram and Trigram models. (16 pts)

```
Copy your code script and printed results to answer the question here.

> Code script:
import math

def add_k_smoothing(unigram_counts, bigram_counts, k):
    smoothed_bigram_model = {}

    for word_1 in unigram_counts:
```

```
            for word_2 in unigram_counts:
                # TODO: Compute smoothed bigram probabilities
                numerator = bigram_counts.get( (word_1, word_2), 0) + k
                denominator = unigram_counts[word_1] + k * len(unigram_counts)
                smoothed_bigram_model[(word_1, word_2)] = numerator / denominator

        return smoothed_bigram_model

    smoothed_bigram_model = add_k_smoothing(unigram_counts, bigram_counts, 0.5)

    sample_sentence = test_corpus[5]

    bigram_prob = calculate_bigram_prob(sample_sentence, smoothed_bigram_model)

    print(f'sample_sentence: {sample_sentence}')
    print(f'Bigram probabilities (before smoothing): {calculate_bigram_prob(sample_
    print(f"Smoothed Bigram probabilities (add-k): {bigram_prob}")
    #print(f'Natural Log Representation: {math.log(bigram_prob)}')

    def interpolate_trigram_prob(sentence, trigram_model, bigram_model, unigram_mod
        for idx in range(len(sentence)):
            if idx == 0:
                prob = 1 # assume the sentence always starts with [BOS]
            elif idx == 1:
                prob *= bigram_model.get((sentence[0], sentence[1]), 0)
            else:
                # TODO: Compute interpolated trigram probabilities
                trigram_prob = trigram_model.get((sentence[idx-2], sentence[idx-1],
                bigram_prob = bigram_model.get((sentence[idx-1], sentence[idx]), 0)
                unigram_prob = unigram_model.get(sentence[idx], 0)

                prob = (lambdas[0] * unigram_prob + lambdas[1] * bigram_prob + lamb

        return prob

    print(f'Trigram probabilities (before interpolation): {calculate_trigram_prob(s
    print(f'Interpolated Trigram probabilities: {interpolate_trigram_prob(sample_se
    > Bigram probabilities (before smoothing): 0.0
    > Smoothed Bigram probabilities (add-k): 5.075930329757935e-43

    > Trigram probabilities (before interpolation): 0.0
    > Interpolated Trigram probabilities: 6.182331568255536e-36
```

(b) In one or two sentences, explain how to determine k in add-k smoothing and λ in interpolation smoothing for a better smoothing effect. Be specific about the evaluation metric you use. (5 pts)

When determining the 'k' value for add-k smoothing, it is necessary to target values that minimize perplexity as a measurement metric relative to the training data. Similarly, the lambda value for interpolation also can use perplexity as an evaluation metric, where testing multiple lambda values on a development set can help with minimizing perplexity.

4. *Language Model Evaluation* (24 pts).

(a) Evaluate the previously trained N-gram models with perplexity on the test set. (6 pts)

Copy your code script and printed results to answer the question here.

> Code script:

```
def calculate_bigram_perplexity(sentence, smoothed_bigram_model):
    """
    Calculates the perplexity of a sentence using the add-k smoothed bigram mod
    """
    perplexity = 0
    for idx in range(len(sentence)):
        if idx == 0:
            prob = 1  # assume the sentence always starts with [BOS]
        else:
            prob = smoothed_bigram_model.get((sentence[idx - 1], sentence[idx])

        perplexity += np.log(prob)

    # TODO: Compute perplexity
    #perplexityFinal = 2 ** (-1/len(sentence) * perplexity)
    perplexity = math.exp(-1/(len(sentence)) * perplexity)


    return perplexity

# Calculate PPL for add-k smoothed bigram model
bigram_perplexity = []
for sentence in test_corpus:
    perplexity = calculate_bigram_perplexity(sentence, smoothed_bigram_model)
    bigram_perplexity.append(perplexity)

print(f"Smoothed Bigram Perplexity (add-k): {np.mean(bigram_perplexity)}")

def calculate_trigram_perplexity(sentence, trigram_model, bigram_model, unigram
    """
    Calculates the perplexity of a sentence using the interpolated trigram mode
```

```
    """
    perplexity = 0
    for idx in range(len(sentence)):
        if idx == 0:
            prob = 1  # assume the sentence always starts with [BOS]
        elif idx == 1:
            prob = bigram_model.get((sentence[0], sentence[1]), 1e-10)  # use a
        else:
            prob = lambdas[0] * unigram_model.get(sentence[idx], 1e-10) + lambd

        perplexity += np.log(prob)

    # TODO: Compute perplexity
    perplexity = math.exp(-1/(len(sentence)) * perplexity)

    return perplexity


# Calculate PPL for interpolated trigram model
trigram_perplexity = []
for sentence in test_corpus:
    perplexity = calculate_trigram_perplexity(sentence, trigram_model, bigram_m
    trigram_perplexity.append(perplexity)

print(f"Interpolated Trigram Perplexity: {np.mean(trigram_perplexity)}")

> Smoothed Bigram Perplexity (add-k): 166678.44173551002
> Interpolated Trigram Perplexity: 5700.080387158461
```

(b) Implement text generation with N-gram models and obtain the generated texts. (15 pts)

```
Copy your code script and printed results to answer the question here.

> Code script:
def generate_text_unigram(unigram_model, max_seq_len):
    """
    Generates text using the unigram model with greedy decoding.
    """
    current_token = "[BOS]"
    generated_text = [current_token]
    for _ in range(max_seq_len - 1):
        max_prob = 0
        next_token = None
        for word in unigram_model:
            unigram_prob = unigram_model[word]
            # TODO: Select the next token using greedy decoding
            if unigram_prob > max_prob:
                max_prob = unigram_prob
```

```
                next_token = word

        generated_text.append(next_token)
        current_token = next_token

        if current_token == "[EOS]":
            break

    return generated_text

# Generate text using the unigram model
generated_text = generate_text_unigram(unigram_model, 10)
print(f"Unigram generated texts: {' '.join(generated_text)}")

def generate_text_bigram(bigram_model, unigram_model, max_seq_len):
    """
    Generates text using the bigram model with greedy decoding.
    """
    current_token = "[BOS]"
    generated_text = [current_token]
    for _ in range(max_seq_len - 1):
        max_prob = 0
        next_token = None
        for word in unigram_model:
            bigram_prob = bigram_model.get((current_token, word), 0)
            # TODO: Select the next token using greedy decoding
            if bigram_prob > max_prob:
                max_prob = bigram_prob
                next_token = word

        generated_text.append(next_token)
        current_token = next_token

        if current_token == "[EOS]":
            break

    return generated_text

# Generate text using the bigram model
generated_text = generate_text_bigram(bigram_model, unigram_model, 10)
print(f"Bigram generated texts: {' '.join(generated_text)}")

def generate_text_trigram(trigram_model, bigram_model, unigram_model, max_seq_l
    """
    Generates text using the trigram model with greedy decoding.
    """
    current_token_1 = "[BOS]"
    current_token_2 = None
```

```
    generated_text = [current_token_1]

    # Use bigram model to generate the second token
    max_prob = 0
    next_token = None
    for word in unigram_model:
        # TODO: Select the second token using greedy decoding
        bigram_prob = bigram_model.get((current_token_1, word), 0)
        if bigram_prob > max_prob:
            max_prob = bigram_prob
            next_token = word

    generated_text.append(next_token)
    current_token_2 = next_token

    if current_token_2 == "[EOS]":
        return generated_text

    # Use trigram model to generate the remaining tokens
    for _ in range(max_seq_len - 2):
        max_prob = 0
        next_token = None
        for word in unigram_model:
            # TODO: Select the next token using greedy decoding
            trigram_prob = trigram_model.get((current_token_1, current_token_2,
            if trigram_prob > max_prob:
              max_prob = trigram_prob
              next_token = word

        generated_text.append(next_token)
        current_token_1 = current_token_2
        current_token_2 = next_token

        if current_token_2 == "[EOS]":
            break

    return generated_text

# Generate text using the trigram model
generated_text = generate_text_trigram(trigram_model, bigram_model, unigram_mod
print(f"Trigram generated texts: {' '.join(generated_text)}")


> Unigram generated texts: Unigram generated texts: [BOS] . . . . . .
> Bigram generated texts: Bigram generated texts: [BOS] the u . [EOS]


> Trigram generated texts: Trigram generated texts: [BOS] the company said . [E
```

(c) In one or two sentences, compare the quality of texts generated by different N-gram models. (3 pts)

The unigram model does not generate a coherent sentence at all and simply outputs the highest probability token at any given time, and the bigram model does a bit better in performance with a complete beginning and end token. The trigram token does the best by far, with a subject and verb sequenced between a beginning and end token.