ANLP 2024 Assignment 1

Marked anonymously: do not add your name(s) or ID numbers.

Use this template file for your solutions. Please start each question on a new page (as we have done here), do not remove the pagebreaks. Different questions may be marked by different markers so your answers should be self-contained.

Don't forget to copy your code for questions 1-5 into the Appendix of this file.

1 Preprocessing each line (10 marks)

```
def preprocess_line(line):
    line = re.sub(r'\d', '0', line) # Convert digits to '0'
    rule = re.compile(r'[^a-z0␣.]') # Define allowed characters,
       including '.'
    line = rule.sub('', line.lower()) # Remove disallowed characters and
        lowercase

    # Split the line into sentences based on '.' and process each
       sentence
    sentences = line.split('.')
    processed_sentences = [f"#{sentence.strip()}." for sentence in
       sentences if sentence.strip()]

    # Join the processed sentences back into a single line
    processed_line = '␣'.join(processed_sentences) + "##"

    return processed_line
```

2 Examining a pre-trained model (10 marks)

This model likely uses Maximum Likelihood Estimation (MLE) along with some smoothing techniques. It seems like MLE is used because the probabilities are based on how often the trigrams appear in the text. For example, the probability of 'e' coming after 'th' is 46.9%, which might be because the word "the" shows up a lot, so it gets counted more. Here are some examples:

thd 1.305e-03

the 4.690e-01

thf 2.139e-05

MLE alone can result in zero probabilities for unseen trigrams, but since we don't see any zero probabilities and several trigrams have the same small probability(for example, three consonants 'ngj' with probability 0.0001229), it suggests that add-alpha smoothing is being used as it ensures that all possible trigrams have non-zero probabilities by adding a small constant to each count.

We have been trying to identify a pattern to determine if other smoothing techniques such as interpolation, back-off, Good-Turing, or Kneser-Ney have been used in training the model. We only have the final model and do not know the training data or process. We have explored the model extensively but have not found anything. As a next step, we plan to implement each of these smoothing techniques on the trigram model and use the Wikipedia corpus to see if we can find a similar pattern.

3 Implementing a model: description and example probabilities (35 marks)

3.1 Model description

For the model implementation we used trigram model with add-α smoothing. We used smoothing, since trigram model trained on the given training tests would contain trigram combinations with 0 count and probability respectively. In terms of estimation method, we used Maximum Likelihood Estimation(MLE) assigning the count-based probabilities to the trigrams.

Furthermore, we incorporated α-smoothing to address the problem of unseen data using the "stealing from rich, giving to poor" method, which also allows us to assess the model's perplexity. We chose α-smoothing over other methods for several reasons:

1. Good-Turing smoothing: While effective, Good-Turing can be complex to implement and may not perform well with very sparse data. It also requires careful handling of zero-frequency events, which can be challenging in a trigram model.

2. Interpolation: While interpolation can be effective, it requires combining probabilities from lower-order n-grams (unigrams and bigrams). Given our focus on capturing trigram patterns for language identification, relying too heavily on lower-order n-grams could potentially weaken the unique trigram characteristics of each language.

3. Back-off: We didn't use back-off because our data is unbalanced with many trigrams having zero probability. This would cause frequent backing off to bigrams, which might also have zero probabilities, potentially leading to unreliable estimates and loss of contextual information.

In contrast, α-smoothing offers a simple yet effective solution with a single, easily optimizable parameter. We created three models with α-smoothing, one for each language, efficiently capturing language-specific patterns while addressing unseen data.

3.2 Model training

On this stage, we trained our model on given documents using maximum likelihood estimation.

Before we started scanning through our document, we decided to divided our data on 3 parts - **train, dev and test** with distributions of 90%, 5% and 5% respectively.

Also, we generated a template for all possible trigram combinations where we counted occurence of each combination.

Thus, we scanned our file using defined `preprocess_line` function. Going through 3-characters sized window, we counted each appearances of trigram as well as saved **bigram combinations count as history**. Since, there were trigrams not appeared in trained set, we added smoothing and our main formula become as below

$$P(w_i|w_{i-2}, w_{i-1}) = \frac{\text{Count}(w_{i-2}, w_{i-1}, w_i) + \alpha}{\text{Count}(w_{i-2}, w_{i-1}) + \alpha V}$$

Where:

- $P(w_i|w_{i-2}, w_{i-1})$ is the smoothed probability of the word w_i given the two preceding words w_{i-2} and w_{i-1},
- $\text{Count}(w_{i-2}, w_{i-1}, w_i)$ is the count of the trigram (w_{i-2}, w_{i-1}, w_i) in the training corpus,
- $\text{Count}(w_{i-2}, w_{i-1})$ is the count of the bigram (w_{i-2}, w_{i-1}) in the training corpus,
- V is the size of the vocabulary (i.e., the number of unique characters),
- α is the smoothing parameter, typically a small positive value.

α-smoothing helped us to balance the data redistributing the probabilities from too frequent to those that were referring to zero probabilities. Also, it helped us avoid overfitting.

Optimization of α-parameter. As written before, we divided three different datasets of each of the models - train, dev, test. We used dev test as dataset where we can check how models with different hyperparameters perform and compare their metric performance. For the metrics, we used **perplexity** defined in the **Paragraph 5**. Having our model trained on different hyperparameters and evaluating perplexity, we ran our alpha in range of values from `1` to `0.01`. Reading in a dev document and computing its perplexity under the estimated language models, we defined that α giving **minimum** perplexity is equal to

$$\alpha_{en} = 0.62 \qquad Perplexity_{en} = 8.6524$$

Doing similar for other models, we have following α-s

$$\alpha_{de} = 0.09 \qquad Perplexity_{de} = 5.96$$
$$\alpha_{es} = 0.08 \qquad Perplexity_{es} = 5.80$$

Assigning presented hyperparameters do not guarantee our model would perform best given experiment test data. Since for the configuration of the α for particular language model we used dev set of subsequent language (for example, training.es - dev.es), we expect our model acquire lower perplexity on test documents with the same languages.

For instance, if we test our spanish data model on the English Test set, we get the following perplexity

$$\alpha_{es\_en} = 0.08 \qquad Perplexity_{es\_en} = 28.56$$

Above perplexity demonstrates that usually it is surprise for Spanish model to observe English combinations. More about insights from the Language Generation using English, Spanish and German Language Models you can see in the chapter 4 of this paper.

3.3 Model excerpt

For this part of question, we printed out all our probabilities for all trigrams starting with 'ng', in other words we printed probabilities of all possible characters given history bigram 'ng'.

Trigram	**Probability**
ng	0.7874
nge	0.0855
ng.	0.0277
ngs	0.0214
ngt	0.0138
ngr	0.0126
ngo	0.0075
ngd	0.0050
nga	0.0038
ngl	0.0038
ngu	0.0038
ngf	0.0025
ngi	0.0025
ngn	0.0025
ngm	0.0013
ngz	0.0013
ng0	0.0013
ngw	0.0013
ngc	0.0013
ngv	0.0013
ngj	0.0013
ngp	0.0013
ngg	0.0013
ngh	0.0013
ngy	0.0013
ngx	0.0013
ngk	0.0013
ngq	0.0013
ngb	0.0013
ng#	0.0013
sum	**1.0000**

Table 1: Probabilities of Trigrams starting with "ng"

As we can have been expected from the model with smoothing, our output(see Table 1) probabilities sum up to 1.0000 showing that our model distributed the probability between frequent and rare occurred sequences. Besides, knowing basic English, we can determine that it is the most probable that after *ng* it usually goes " " (backspace) character, demonstrating that probably there are present participles presented frequently, appearing usually not in the end of the sentence.

4 Generating from models (15 marks)

Our implemented generateFromLM function works step-by-step as follows:

1. **Initialization**: The function starts with an initial sequence, typically '##', which represents the start of a sequence.

2. **Character Generation Loop:** The function by default generates 600 characters including '#' character. Then For the History Extraction, it extracts the last two characters of the current sequence to form the history for the next character prediction.

Next Character Probabilities: It collects possible next characters and their probabilities from the language model, based on the current history.

Fallback Mechanism: If no characters are found for the current history, it defaults to the most probable character in the entire model.

Sampling Method: Depending on the specified sampling method (top-k or top-p), it selects the next character: Top-k Sampling: Selects from the top k most probable characters. Top-p Sampling: Selects from characters whose cumulative probability exceeds p. **Sequence Update**: The selected character is appended to the sequence.

3. **Post-processing**: After generating the sequence, it removes all '#' characters and truncates the sequence to the first 300 characters.

Generated Sequences:

```
(a) Our model Top-p (p=0.62): e.kose struch which in the se show me
   region ser the of eurally and the she por portake the commint in the
   res the ission the seres onspearegion and the and so a fores the
   shavers a comming the the pre of the and in is an artion of postan of
    the re arliamen por the res of the the the con the pose th
(b) model-br.en Top-p(p=0.62): what there boy.you gone.that ant that the
    you cant is there a go that are doggie.whats a bung.you doggie.whats
    is the do sommy ther blookay.whats is he bood you wand that it.its
   there ballood you like the hats he bookay.ther fin that do you so it.
   the do you colood.the your nicks.whats that a do ther
```

After several attempts with different values for Top-k and Top-p methods we realized that Top-p sampling for p=0.62 seems to generate the most meaningful sequence, as it provides a more natural and diverse output while maintaining coherence. So we only analyze top-.62 for these two.

Analysis of Generated Sequences

The two generated sequences show distinct differences in vocabulary, structure, and style. Model (a) produces more formal, possibly political language (e.g., "region," "eurally," "arliamen," and "portake") with longer, complex phrases. In contrast, model (b) generates more colloquial, conversational text (e.g., "boy," "doggie," "bookay") with shorter, fragmented sentences often ending in periods. Model (b) also shows more recognizable patterns and repetitions (e.g., "doggie.whats," "do you"), while model (a) appears more varied but less structured.

These differences likely result from distinct training data sources: model (a) probably used formal written texts, while model (b) seems based on informal, conversational English, possibly including transcripts of spoken dialogue. Notably, neither model produces text with a meaningful overall structure, which is expected given that both are likely trained using trigram models operating at the character level, focusing on local patterns rather than broader context or semantics.

5 Computing perplexity (15 marks)

For calculating the perplexity of our models, we used the standard formula to measure the surprise the model experiences when processing the data. The formula for computing perplexity for a trigram model is as follows:

$$\text{Perplexity} = 2^{-\frac{1}{N}\sum_{i=1}^{N}\log_2 P(w_i|w_{i-2},w_{i-1})}$$

where N is the total number of words in the test dataset, and $P(w_i|w_{i-2}, w_{i-1})$ is the probability of word w_i given the preceding words w_{i-2} and w_{i-1}.

The above term of notation gives us the proper flexibility in terms of computations allowing us to process lines, get logarithm and sum the values up. We computed the perplexity for each of our three models, where each model's hyperparameters were trained on a development set. For the test datasets, we used 3 different texts with nearly same length in 3 different languages. You can observe results below

Model	**Standard Test**	**La Divina Comedia (es)**	**Josefine Mutzenbacher (de)**
Pretrained	25.754	60.388	53.211
English (en)	**8.001**	14.375	17.657
German (de)	29.318	17.145	**7.575**
Spanish (es)	29.331	**7.640**	40.686

Table 2: Perplexity results for different models across test sets.

From this data, we notice that our English model performs better on English test dataset, Spanish on Spanish Language dataset, and German on German dataset. That fact gives us a confidence that our model differentiates between languages of documents.

Thus, if we were given random dataset, our approach would be to check perplexities on all three of our model and choose the lowest one. The calculated perplexities would give us only an estimation of which model would have the least surprise from seeing the document. However, it is only an estimate, and we cannot completely rely on this estimation.

Testing our model on perplexity on different test sets, we may conclude that perplexity between model and same language document will have range up to 10.00. This perplexity would be based on factors as hyperparameter of α smoothing, size of training set, and even the at least similarity between topics of test document and train model. For example, if we were given document consisted of words and its perplexity under our English LM, and if the perplexity would be in range between $1 - 10$, it would be enough to conclude that it is in the English Language. From the table above and the text generation task defined in **Chapter 4**, we observe that although the Spanish texts, and German texts are similar in some morphemes and parts of the words with those in English document, their perplexity fluctuates between $14 - 17$.

6 Extra question (15 marks)

Appendix: your code

We enumerated subsections to be able to refer to them through our assignment.

6.1 Initial Version of the Trigram Model

```
def build_trigram_model(file_path):
    trigram_counts = defaultdict(int)
    bigram_counts = defaultdict(int)

    with open(file_path, 'r') as file:
        for line in file:
            line = preprocess_line(line)
            for i in range(len(line) - 2):
                bigram = line[i:i+2]
                trigram = line[i:i+3]
                bigram_counts[bigram] += 1
                trigram_counts[trigram] += 1

    trigram_probabilities = defaultdict(float)

    for ch1 in possible_characters:
        for ch2 in possible_characters:
            for ch3 in possible_characters:
                ch = ch1+ch2+ch3
                if ch not in trigram_probabilities:
                    trigram_probabilities[ch1+ch2+ch3] = 0

    for trigram in trigram_counts:
        bigram = trigram[:2]
        trigram_probabilities[trigram] = trigram_counts[trigram] / bigram_counts[bigra

    return trigram_probabilities
```

Above code gave us overfitted distributions and contained 0-probability combinations. For example,

```
##t :  0
##b :  0
##n :  0
### :  0
```

6.2 Trigram Model with Smoothing

```
def build_trigram_model_with_smoothing(file_path, alpha=0.1):
    trigram_counts = defaultdict(int)
    bigram_counts = defaultdict(int)

    # Define the complete set of possible characters

    for ch1 in possible_characters:
        for ch2 in possible_characters:
            for ch3 in possible_characters:
                trigram_counts[ch1+ch2+ch3] = 0

    with open(file_path, 'r') as file:
        for line in file:
            line = preprocess_line(line)
            for i in range(len(line) - 2):
                bigram = line[i:i+2]
                trigram = line[i:i+3]
                bigram_counts[bigram] += 1
                trigram_counts[trigram] += 1

    trigram_probabilities = defaultdict(float)
    vocabulary_size = len(possible_characters) # Use the complete set of
         possible characters

    # Calculate probabilities with smoothing
    for trigram in trigram_counts:
        bigram = trigram[:2]
        trigram_probabilities[trigram] = (trigram_counts[trigram] + alpha
            ) / (bigram_counts[bigram] + alpha * vocabulary_size)

    return trigram_probabilities
```

6.3 α-parameter optimization

In the below code, we show you example how we optimized parameter specifically for English Language Model.

```
train_file = base_url + 'training_train.en'
dev_file = base_url + 'training_dev.en'
mini = float('inf')
for i in range(100, 1, -1):
    coef = i/100
    trigram_probabilities_smoothing_de =
       build_trigram_model_with_smoothing(train_file, coef)
    perplexity = evaluate_perplexity(trigram_probabilities_smoothing_de,
         dev_file)
    mini = min(mini, perplexity)
    print(coef, perplexity)
print(mini)
```

6.4 Model Building

```
trigram_probabilities_smoothing_en = build_trigram_model_with_smoothing(
   enfile, 0.62)
trigram_probabilities_smoothing_de = build_trigram_model_with_smoothing(
   defile, 0.09)
trigram_probabilities_smoothing_es = build_trigram_model_with_smoothing(
   esfile, 0.08)
```

6.5 Generation

```
def generateFromLM(pretrained_set, initial_sequence='##', length=500,
   sampling_method='top-k', k=5, p=0.72):
    sequence = initial_sequence
    while len(sequence) < length:
        history = sequence[-2:]
        possible_next_chars = {}

        for trigram, prob in pretrained_set.items():
            if trigram.startswith(history):
                next_char = trigram[2]
                possible_next_chars[next_char] = prob
        if not possible_next_chars:
            # Fallback: Use the most probable characters in the entire
               model
            most_probable_trigram = max(pretrained_set.items(), key=
                lambda x: x[1])
            next_char = most_probable_trigram[0][2]
        else:
            if sampling_method == 'top-k':
                # Top-k sampling
                sorted_chars = sorted(possible_next_chars.items(), key=
                   lambda x: x[1], reverse=True)
                top_k_chars = sorted_chars[:k]
                chars, weights = zip(*top_k_chars)
                next_char = random.choices(chars, weights=weights, k=1)[0]
            elif sampling_method == 'top-p':
                # Top-p (nucleus) sampling
                sorted_chars = sorted(possible_next_chars.items(), key=
                   lambda x: x[1], reverse=True)
                cumulative_prob = 0.0
                top_p_chars = []
                for char, prob in sorted_chars:
                    cumulative_prob += prob
                    top_p_chars.append((char, prob))
                    if cumulative_prob >= p:
                        break
                chars, weights = zip(*top_p_chars)
                next_char = random.choices(chars, weights=weights, k=1)[0]
            else:
                # Default to random choice if no valid method is specified
                next_char = random.choices(list(possible_next_chars.keys()
                   ), weights=possible_next_chars.values(), k=1)[0]

        sequence += next_char

    # Remove all '#' characters
    sequence = sequence.replace('#', '')

    # Cut the first 300 characters
    sequence = sequence[:300]
```

```
    return sequence


# Iterate through p-values from 0.5 to 0.75 with 0.05 increments
for p_value in [round(0.5 + i * 0.05, 2) for i in range(6)]:
    generated_sequence_top_p = generateFromLM(
        trigram_probabilities_smoothing_en, sampling_method='top-p', p=
        p_value)
    print(f"(a) Our model Top-p (p={p_value}):",
        generated_sequence_top_p)

    generated_sequence_top_p = generateFromLM(pretrained_set,
        sampling_method='top-p', p=p_value)
    print(f"(b) model-br.en Top-p(p={p_value}):",
        generated_sequence_top_p)

    print() # Add a blank line for better readability between iterations
```

6.6 Evaluation of Perplexity

```
def evaluate_perplexity(trigram_probabilities, validation_file):
    log_prob_sum = 0
    trigram_count = 0
    with open(validation_file, 'r') as file:
        for line in file:
            line = preprocess_line(line)
            for i in range(len(line) - 2):
                trigram = line[i:i+3]
                trigram_count += 1
                probability = trigram_probabilities.get(trigram, 1e-10)
                log_prob_sum += log(probability, 2)
    avg_log_prob = log_prob_sum / trigram_count
    perplexity = 2 ** (-avg_log_prob)
    return perplexity
```