Question 4

In the following parts, you will create a bigram and trigram model to classify if documents are written by humans or by ChatGPT. Download the datafile humvgpt.zip1. This file contains around 40, 000 human written and 20, 000 ChatGPT written documents. The data is stored in separate text files named hum.txt and gpt.txt respectively.

In [1]:
```
# Import libraries
from collections import defaultdict
import string
import re
import math
import numpy as np
```

a) Clean the data by removing all punctuation except ",.?!" and converting all words to lower case. You may also find it helpful to add special and tokens to each document for calculating N-gram probabilities. Partition the initial 90% of the data to a training set and final 10% to test set (the data has already been shuffled for you)

In [2]:
```
## Function to pre-process data
def pre_process_file(file):
    processed_documents = []

    with open(file, 'r+') as f:

        documents = f.read().splitlines()

        # Remove punctuation except ",.?!"
        punctuation_to_remove = ''.join(set(string.punctuation) - set(',.?!'))
        translation_table = str.maketrans('', '', punctuation_to_remove)

        for doc in documents:
            # remove punctuation and make all lower case
            modified_doc = doc.translate(translation_table).lower()

            # Add <START> before and <END> after the modified content
            processed_doc = '<START> ' + modified_doc.strip() + ' <END>'
            processed_documents.append(processed_doc)

    return processed_documents
```

In [3]:
```
# Read in the data
filepath_gpt = '/Users/anouckrietveld/Documents/Columbia/ML for DS/gpt.txt'
filepath_hum = '/Users/anouckrietveld/Documents/Columbia/ML for DS/hum.txt'

# Data pre-processing
file_gpt = pre_process_file(filepath_gpt)
file_hum = pre_process_file(filepath_hum)
```

In [4]:
```
# Partition the data into training and test sets
def train_test_split(file):
    train_size = int(0.9 * len(file))
    train_set = file[:train_size]
    test_set = file[train_size:]
    return train_set, test_set

gpt_train_set, gpt_test_set = train_test_split(file_gpt)
hum_train_set, hum_test_set = train_test_split(file_hum)
```

(b) Train a bigram and trigram model by calculating the N-gram frequencies per class in the training corpus. Calculate and report the percentage of bigrams/trigrams in the test set (counted with repeats) that do not appear in the training corpus (this is called the OOV rate). You should calculate two OOV rates, one for bigrams and one for trigrams.

First, we are going to calculate the N-gram frequencies per class in the training corpus

In [5]:
```
def calculate_ngram_frequency(file, n):
    ngram_freq = defaultdict(int)

    for document in file:
        words = document.split()
        for i in range(n - 1, len(words)):
            ngram = tuple(words[i - n + 1:i + 1])
            ngram_freq[ngram] += 1

    return ngram_freq

## Bigram
# Class gpt
train_bigram_frequency_gpt = calculate_ngram_frequency(gpt_train_set, 2)

# Class hum
train_bigram_frequency_hum = calculate_ngram_frequency(hum_train_set, 2)
```

```
## Trigram
# Class gpt
train_trigram_frequency_gpt = calculate_ngram_frequency(gpt_train_set, 3)

# Class hum
train_trigram_frequency_hum = calculate_ngram_frequency(hum_train_set, 3)
```

Now, we are going through the bigram and trigram frequencies of the test sets and count the number of times that the bigrams or trigrams are not present in the training set.

In [6]:
```
# Calculate and report the OOV rate

# Calculate the bigram frequencies for the test sets
test_bigram_frequency_gpt = calculate_ngram_frequency(gpt_test_set, 2)
test_bigram_frequency_hum = calculate_ngram_frequency(hum_test_set, 2)

# Calculate the trigram frequencies for the test sets
test_trigram_frequency_gpt = calculate_ngram_frequency(gpt_test_set, 3)
test_trigram_frequency_hum = calculate_ngram_frequency(hum_test_set, 3)

# Merge the two default dicts together and sum their values
def merge_and_sum(dict1, dict2):
    merged_dict = defaultdict(int)
    # Add values from dict1
    for key, value in dict1.items():
        merged_dict[key] += value
    # Add values from dict2
    for key, value in dict2.items():
        merged_dict[key] += value
    return merged_dict

# Bigram train and test
train_bigram_frequency = merge_and_sum(train_bigram_frequency_gpt, train_bigram_frequency_hum)
test_bigram_frequency = merge_and_sum(test_bigram_frequency_gpt, test_bigram_frequency_hum)

# Trigram train and test
train_trigram_frequency = merge_and_sum(train_trigram_frequency_gpt, train_trigram_frequency_hum)
test_trigram_frequency = merge_and_sum(test_trigram_frequency_gpt, test_trigram_frequency_hum)
```

In [7]:
```
def oov_rate(test, train):
    oov_count = 0
    total_bigrams_test = sum(test.values())

    for bigram, freq in test.items():
```

```
        if not bigram in train.keys():
            oov_count += freq # counted with repeats
    oov_rate = (oov_count / total_bigrams_test) * 100

    return oov_rate
```

In [8]:
```
# Calculate bigram frequency
oov_rate_bigram = oov_rate(test_bigram_frequency, train_bigram_frequency)
print(f'The OVV rate of the bigram is {round(oov_rate_bigram,2)}%')

# Calculate trigram frequency
oov_rate_trigram = oov_rate(test_trigram_frequency, train_trigram_frequency)
print(f'The OVV rate of the trigram is {round(oov_rate_trigram,2)}%')
```

```
The OVV rate of the bigram is 9.64%
The OVV rate of the trigram is 35.27%
```

(c) Evaluate the two models on the test set and report the classification accuracy. Which model performs better and why? Your justification should consider the bigram and trigram OOV rate. This study will also tell you how difficult or easy it is to distinguish human vs. AI generated text!

In [9]:
```
# Calculate |V|, the number of unique words in the document

def count_unique_words(documents):
    # Concatenate all documents into a single string
    all_text = ' '.join(documents)

    # Tokenize the string into individual words
    words = all_text.split()

    # Use a set to store unique words
    unique_words = set(words)

    # Count the number of unique words
    num_unique_words = len(unique_words)

    return num_unique_words
```

In [10]:
```
## Incorrect n_minus_1_gram_count Calculation for Trigrams: In the trigram version of laplacian_smoothing,
## you must adjust the denominator to use the frequency of the two-word sequence preceding the third word in
## trigram, not just the first word. This requires changing how n_minus_1_gram_freq is accessed. For trigrams
## a single word (ngram[0]), but about the two-word sequence (ngram[:-1]).

def laplacian_smoothing(ngram, n_minus_1_gram_freq, ngram_freq, vocab_size, alpha=1):
```

```
    """
    Calculate the smoothed probability of a bigram given the class.

    Parameters:
    - bigram: The current bigram tuple (word_i, word_i+1).
    - unigram_freq: Frequency dictionary of unigrams for the class.
    - bigram_freq: Frequency dictionary of bigrams for the class.
    - vocab_size: The size of the vocabulary (|V|).
    - alpha: Smoothing parameter (default is 1 for add-one smoothing).

    Returns:
    - The smoothed probability of the bigram.
    """

    if len(ngram) == 2:  # For bigrams, the context is the first word
        context = ngram[:-1]  # This is equivalent to (ngram[0],)
    else:  # For trigrams (and potentially higher n-grams), the context is the first two words
        context = ngram[:-1]  # This slices everything but the last element, suitable for trigrams and adapta

    ngram_count = ngram_freq.get(ngram, 0) + alpha
    n_minus_1_gram_count = n_minus_1_gram_freq.get(context, 0) + vocab_size * alpha
    smoothed_probability = ngram_count / n_minus_1_gram_count

    return smoothed_probability
```

In [11]:

```
def prob_given_document(test_set_dict, n_minus1_gram_frequency_gpt, ngram_frequency_gpt, n_minus1_gram_freque
    document_probabilities = {}

    for document, true_label in test_set_dict.items():
        ngram_prob_gpt = 0
        ngram_prob_hum = 0

        # Split the document into tokens
        words = document.split()

        # Adjust the range to handle trigrams if n=3
        for i in range(len(words) - (n - 1)):
            # Adjust to create either bigram or trigram based on n
            ngram = tuple(words[i:i+n])

            # Calculate probability of the ngram given the class (human or GPT) using Laplacian smoothing
            # gpt
            laplacian_gpt = laplacian_smoothing(ngram, n_minus1_gram_frequency_gpt, ngram_frequency_gpt, abs_
            ngram_prob_gpt += math.log(laplacian_gpt)
```

```
            # hum
            laplacian_hum = laplacian_smoothing(ngram, n_minus1_gram_frequency_hum, ngram_frequency_hum, abs_
            ngram_prob_hum += math.log(laplacian_hum)

        # Add the log class prior probabilities
        prob_gpt_given_document = ngram_prob_gpt + math.log(P_gpt)
        prob_hum_given_document = ngram_prob_hum + math.log(P_hum)

        # Store the calculated probabilities for each document
        document_probabilities[document] = {'GPT': prob_gpt_given_document, 'Human': prob_hum_given_document}

    return document_probabilities
```

In [12]:
```
def calculate_accuracy(probabilities_file, test_set_dictionary):
    correct_predictions = 0
    total_predictions = len(probabilities_file)

    for document, probabilities in probabilities_file.items():

        # Determine the predicted label based on which probability is higher
        predicted_label = 'gpt' if probabilities['GPT'] > probabilities['Human'] else 'hum'

        # Get the true label from the test_set_dictionary
        true_label = test_set_dictionary[document]

        # Compare the predicted label to the true label
        if predicted_label == true_label:
            correct_predictions += 1

    # Calculate the accuracy
    accuracy = correct_predictions / total_predictions

    return accuracy
```

In [13]:
```
# Put the test set of hum and gpt together and add true_label to the dictionary

# Merge lists
test_set = gpt_test_set + hum_test_set

# Create dictionary where document:true_label
test_set_dict = {doc: 'gpt' if doc in gpt_test_set else 'hum' for doc in test_set}
```

In [14]:
```
# Calculate the absolute V value, which is the set of all unique words across all documents in the entire cor
corpus = file_gpt + file_hum
```

```
abs_V = count_unique_words(corpus)
```

In [15]:
```
# Calculate P(y) for every class
prob_gpt = len(file_gpt) / (len(file_gpt)+len(file_hum))
prob_hum = len(file_hum) / (len(file_gpt)+len(file_hum))
```

In [16]:
```
# Calculate the probability for gpt and hum given the document

# Calculate unigram frequencies gpt and hum
train_unigram_frequency_gpt = calculate_ngram_frequency(gpt_train_set, 1)
train_unigram_frequency_hum = calculate_ngram_frequency(hum_train_set, 1)

bigram_file_probabilities = prob_given_document(test_set_dict, train_unigram_frequency_gpt, train_bigram_freq

trigram_file_probabilities = prob_given_document(test_set_dict, train_bigram_frequency_gpt, train_trigram_fre
```

In [17]:
```
# Calculate the accuracy
bigram_accuracy = calculate_accuracy(bigram_file_probabilities, test_set_dict)
print(f"The classification accuracy for the bigram model is: {100*bigram_accuracy:.2f}%")

trigram_accuracy = calculate_accuracy(trigram_file_probabilities, test_set_dict)
print(f"The classification accuracy for the trigram model is: {100*trigram_accuracy:.2f}%")
```

```
The classification accuracy for the bigram model is: 95.86%
The classification accuracy for the trigram model is: 94.33%
```

Bigram Model Accuracy: The bigram model has higher accuracy compared to the trigram model. This could be because bigrams are simpler and more frequent in language, making it easier for the model to learn from the training data. Additionally, the OOV rate for bigrams is lower, indicating that a significant portion of the test set's bigrams were present in the training data, facilitating better classification.

Trigram Model Accuracy: The trigram model has lower accuracy, possibly due to the higher complexity of trigrams and a higher OOV rate. Trigrams are less frequent and may capture more nuanced patterns in the text, which might not generalize well to unseen data. The higher OOV rate for trigrams suggests that a larger portion of the test set's trigrams were not present in the training data, leading to difficulties in classification.

(iii) (a) Using T = 50, generate 5 sentences with 20 words each from the human and ChatGPT bigrams/trigrams (you should have 20 sentences total using T = 50). Which text corpus and N-gram model generates the best sentences? What happens when you increase or decrease the temperature? You should begin generation with N – 1 tokens and stop once you generate the token.

In [18]:
```
def softmax(word_counts, temperature):
    """Apply softmax to a list of frequencies with a temperature parameter."""

    # Extract just the frequencies from word_counts
    frequencies = [count for _, count in word_counts]

    # Apply the temperature scaling on the frequencies
    exp_frequencies = np.exp(np.array(frequencies) / temperature)

    # Normalize the exponentiated frequencies to get probabilities
    sum_exp_frequencies = np.sum(exp_frequencies)
    probabilities = exp_frequencies / sum_exp_frequencies

    # Pair each word with its corresponding probability
    next_word_probabilities = [(word, prob) for (word, _), prob in zip(word_counts, probabilities)]

    return next_word_probabilities
```

In [19]:
```
def get_next_word_probabilities(context, ngram_freq, temperature):
    """Given the context, return a dictionary of next word probabilities, with non-qualifying ngrams set to 0

    # Initialize all possible next words with a frequency of 0
    candidates = {bigram: 0 for bigram in ngram_freq}

    # Update the frequencies for ngrams that match the context
    for ngram, count in ngram_freq.items():
        if ngram[:-1] == context:
            candidates[ngram] = count

    word_counts = [(key[-1], value) for key, value in candidates.items()]

    # Now candidates have all possible words with their respective frequencies, zero if they don't follow the
    probabilities = softmax(word_counts, temperature)

    return probabilities
```

In [20]:
```
def generate_sentence(ngram_freq, temperature, n, max_length=20):
    """Generate a sentence using the N-gram frequencies with temperature scaling."""

    sentence = ['<START>'] * (n - 1)  # For bigram, this will be just ['<START>']

    while len(sentence) < max_length:
        context = tuple(sentence[-(n - 1):])
```

```
        word_probabilities = get_next_word_probabilities(context, ngram_freq, temperature)

        words, probs = zip(*word_probabilities)  # Unpack the list of tuples
        next_word = np.random.choice(words, p=probs)

        sentence.append(next_word)

        if next_word == '<END>':
            break

    return ' '.join(sentence)
```

In [21]:
```
def generate_sentences(ngram_freq, temperature, n, max_length, num_sentences):

    sentences = []
    for _ in range(num_sentences):
        sentence = generate_sentence(ngram_freq, temperature, n, max_length)
        sentences.append(sentence)
    return sentences
```

In [22]:
```
def print_generated_sentences(ngram_freq, temperature, n, max_length, num_sentences, description):
    """
    Generate and print sentences based on the n-gram frequencies.

    Parameters:
    - ngram_freq: Dictionary of n-gram frequencies
    - temperature: Temperature parameter for softmax
    - n: The n in n-gram (2 for bigram, 3 for trigram)
    - max_length: Maximum length of generated sentences
    - num_sentences: Number of sentences to generate
    - description: Description of the source and n-gram model
    """
    print(f"Generating sentences for: {description}")
    sentences = generate_sentences(ngram_freq, temperature, n, max_length, num_sentences)
    for sentence in sentences:
        print(sentence)
    print("\n")  # Print a newline for better separation between groups

# Now, we can call this function for each of your cases:
temperature = 50

# Bigram - GPT
print_generated_sentences(test_bigram_frequency_gpt, 20, 2, 20, 5, "Bigram - GPT")
```

```
# Bigram - Human
print_generated_sentences(test_bigram_frequency_hum, 20, 2, 20, 5, "Bigram - Human")

# Trigram - GPT
print_generated_sentences(test_trigram_frequency_gpt, 20, 3, 20, 5, "Trigram - GPT")

# Trigram - Human
print_generated_sentences(test_trigram_frequency_hum, 20, 3, 20, 5, "Trigram - Human")
```

```
Generating sentences for: Bigram - GPT
<START> can be a good tractor rgb social rises for the same they are a good and the united states
<START> the same amounts members the same that the same small management by the same reliable hit finds that
the
<START> services at bright hiding bulbs out snow a few wifi scanner provinces sump 505 healthy to the same da
ylight
<START> . <END>
<START> shopping managing is a good enough crowdfunding . <END>


Generating sentences for: Bigram - Human
<START> the same tanks and the same of the same much gets an moldy reaction lag let long goes founding
<START> the same it s to the same aca mines tie one so as a lot of the same from
<START> the same businesses and the same , and the same arbitrary them systems affecting and the same . <END>
<START> the same the same john criteria irritates that the same abbreviated rap mb gave ? gender farmland arb
iter they
<START> the same from the same this is a lot of the same can be bans bulb the same nothing


Generating sentences for: Trigram - GPT
<START> <START> certain causing agreement , surface being may the . . . by its people 12 , the when
<START> <START> , remember more by affected you and time people . of so need investment impaired radio to .
<START> <START> underwater be other place vietnam have what because and is various are temporary to to growls
this what
<START> <START> that on schools a many layer due it is important to . on has unexpected gasoline . .
<START> <START> its their in in . of of change platform stock to bodies divisions sales home considered what
by


Generating sentences for: Trigram - Human
<START> <START> what to time spicy if proved of to interest doing enjoy register . . . . . .
<START> <START> medical . deformed solution a that locked was default a . a to your , gear to s
<START> <START> <END>
<START> <START> eventually surfaces by i these <END>
<START> <START> different agree this the financial rate the , that the of bigger impulse you react tasks revi
se if
```

Decreasing Temperature: Lowering T results in more deterministic and coherent sentences. This reduction favors the generation of grammatically correct sequences, though they may lack creativity. **Increasing Temperature**: Enhancing T further amplifies randomness, potentially leading to nonsensical and highly unpredictable sentence structures

Bigram Models are likely to generate more grammatically coherent sentences at lower tem- peratures, though such sentences might be less interesting due to their simplicity. **Trigram Models**, with their capacity to understand more context, are better suited for gener- ating complex and nuanced sentences. However, the balance between coherence and creativity is crucial, especially as the temperature increases.

Also, refer to my written assignment for a more comprehensive answer.

In [ ]: