Yusuf Emre KILICER

210104004017

INTRODUCTION

The n gram language model is used to predict the next word or character in a sentence.It basically looks at the last few words or characters to make this prediction.

In a n-gram , it looks at the previous n-1 syllables (for a syllable-based model) or the previous n-1 characters (in a character-based model) to predict the next one.I'have already implemented this code, thus it will be detailed in the continuation of the article

However, it is not too easy to make this prediction. Thus , we first train the model on lots of text , then it learns which words or characters tend to follow each other. After making tones of probability calculation based on this training , it predicts the next most likely syllable or character.

Every step of training data ,testing data , calculating probabilites , smoothing techniques , generating sentences and every other thing will be explained in implementation and design section where it contains the b function implementations and their explanations.

Comparison of Syllable-based model vs Character-based model :

For the unit comparision , syllable based model has much more advantage then character based because syllable based treat each syllable as separate unit.This means that the model considers meaningful sections of language , which are more predictable than individual characters.

On the other hand ,character based models analyze text letter by letter , means each character is the unit.The problem is letters doesnot create enough information to predict.It may not reflect as much sharp meanin as syllables.

For complexity , syllable based segmentation process ,which makes the data processing more complex.However , it has also advantage, for example number of tokens are much much more less then character based models.This makes the model faster training and ngram calculations.

On the other hand , character based model need an too easy process which is separating the letters.However , to create meaningful predictions , it need much more number of n-gram than syllable based model.To increase the n-gram numbers , it needs much more data to understand.

For Correctness , syllable based models tend to generate much more readable text then character based model.The reason is that at the same number of n-grams , syllable based model have quite meaningful parts of words than character based model.

On the other hand , character based models could be better for sensitive predictions or maybe for the searching in youtube or google bar , which tries the predict your current word.Shortly , predicting meaningful sequences is more challencing than syllable based model , but this model can be more successful with complex word forms.

The part where I share the results , you are gonna realize that syllable based model have more success on generating sentence than character based model.

OBJECTIVES OF HOMEWORK

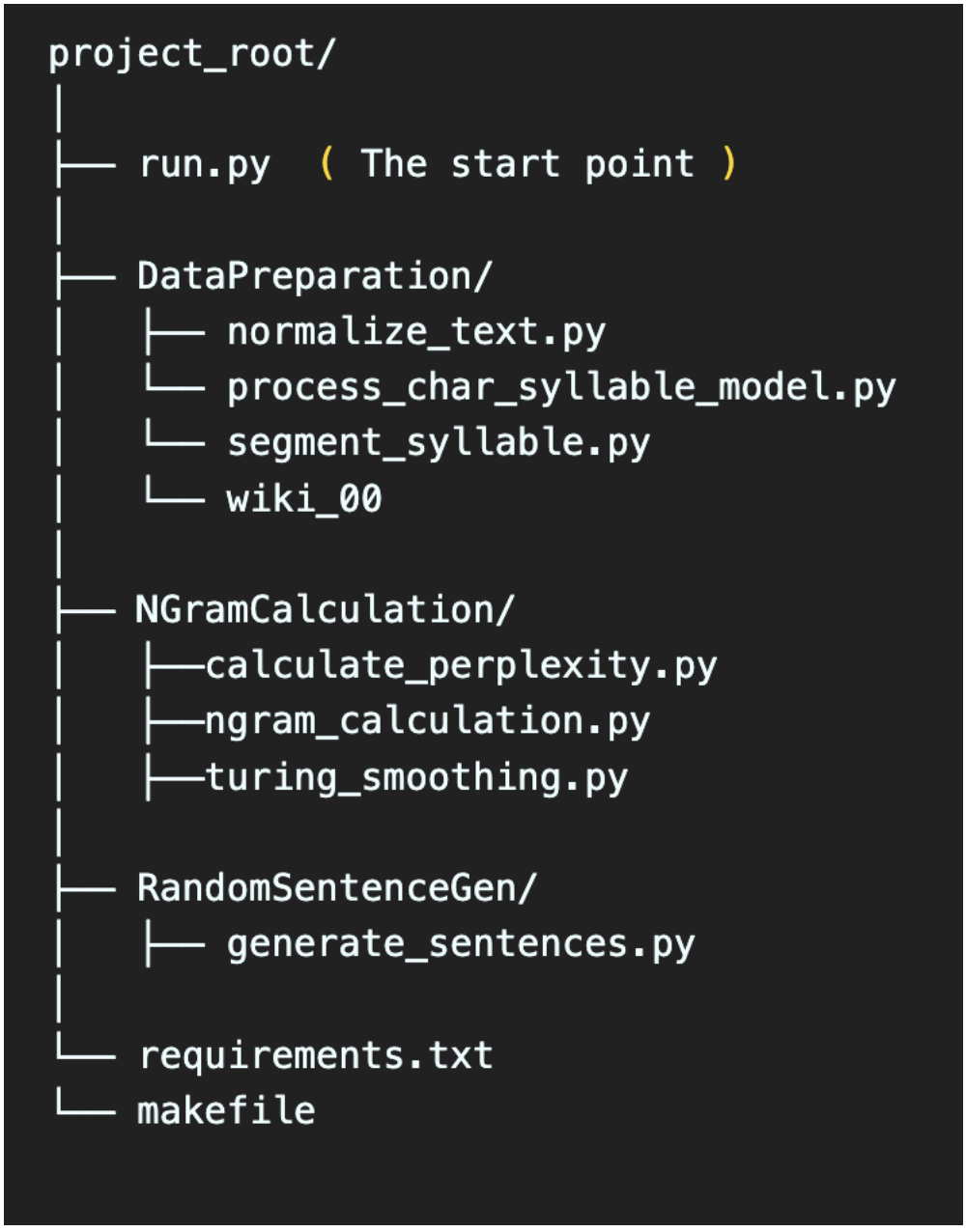
- Preprocess the data
- Implement the N-gram model
- Apply character-based and syllable-based segmentation
- Building N-gram tables for both model
- Applying Good-Turing smoothing and calculate perplexities
- Generating random sentences using processed n grams.
- Comparing and analyzing the results.

DESIGN & IMPLEMENTATION

In this part , I will explain an insanely detailed explanation of my projects design and code blocks.It includes everything I wrote on my project.

My whole project is organized into folders, with each function divided accordingly. However, I call every function in the run.py file which is my starting point.

The folder structure is :



IMPORTS

```python
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~IMPORTING~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~§
import sys,os,re,random
#Need to add the parent directory to the sys.path in order to import the modules on other folders.
#import other folders at the same place with this current file

def append_path(folder_name):
    directory = os.path.abspath(os.path.join(os.path.dirname(__file__), folder_name))
    sys.path.append(directory)

append_path('RandomSentenceGen')
append_path('DataPreparation')
append_path('NGramCalculation')

from DataPreparation.normalize_text import normalize_the_file
from DataPreparation.process_char_syllable_model import process_data_syllable_model, process_data_char_based
from NGramCalculation.ngram_calculation import calculate_n_grams , get_frequency_of_ngrams
from NGramCalculation.turing_smoothing import good_turing_smoothing
from NGramCalculation.calculate_perplexity import  calculate_perplexity
from RandomSentenceGen.generate_sentences import generate_sentence
from sklearn.model_selection import train_test_split
from collections import Counter

file_path = 'DataPreparation/wiki_00'  # No leading space here
```

Then, I start with the normalize_the_file function, which normalizes the text file to Turkish by converting all uppercase letters to lowercase and replacing Turkish characters with their English equivalents.

```python
normalize_the_file(file_path)
```

NORMALIZATION AND READING FILE

```python
# This function replaces Turkish characters with English equivalent characters
def normalize_turkish_to_english(text):
    # Define mapping from Turkish characters to English characters
    mapping_turkish = str.maketrans("çğıöşüÇĞİÖŞÜ", "cgiosuCGIOSU")

    return text.translate(mapping_turkish).lower()  # Translate the text to English characters and convert it to lowercase

def normalize_the_file(file_path):
    try:
        # First, read the file content in read mode
        with open(file_path, 'r', encoding='utf-8') as file:
            lines = file.readlines()

        # Filter out lines containing unnecessary symbols like </doc> or <doc ...>
        cleaned_lines = [line for line in lines if not re.match(r"</?doc\b.*?>", line)]

        # Normalize the text
        normalized_text = normalize_turkish_to_english("".join(cleaned_lines))

        # Open the file in write mode and write the normalized text
        with open(file_path, 'w', encoding='utf-8') as file:
            file.write(normalized_text)

        print(f"File {file_path} is normalized from Turkish to English successfully.\n")
    except Exception as e:
        print(f"The file wasn't found. Check the path and filename: {e}")
```

First, I open the file and read it line by line. Note that I choose not to read lines that start with </doc> or <doc> because they significantly affect the generated sentences. First , I try to consider them , then I realize that they ruin the generated sentences.

I will compare the results later on this page. After reading the lines, I replace the Turkish characters 'çğıöşüÇĞİÖŞÜ' with their English equivalents.
I used the maketrans function, which is easiest way to handle this task..Then I write file to normalized text.

The important thing is that I've already normalized the text file, so there's no need to call this function every time. Therefore, I will simply read the lines.

```python
def read_file(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        lines = file.readlines()
    return lines
lines = read_file(file_path)
```

Basically, I open the file and read the lines.

SPLITTING DATASET INTO TRAIN AND TEST DATA

It is too easy and only one line :

```
train_data, test_data = train_test_split(lines, test_size=0.05, random_state=42)
```

I am splitting as train_data and test_data.

- Lines are the lines that I read from text file.
- test_size represents the proportion of the lines to be used as test data. In this case, 5% of the data will be used for testing.
- random_state is a number used to control the randomness of the split. By setting it to a specific value (42), we make sure that the data is split in the same way every time the code is run. Without it, the split would be different each time, because it would use a new random pattern.

SYLLABLE & CHARACTER BASED MODEL PROCESSING

To handle this task , I used 2 function.

- process_data_syllable_model segments words into syllables.
- process_data_char_based function is used for characted-based model.

```python
def process_data_syllable_model(data):
    processed_lines = []
    for line in data:  # Iterate over each line in the data
        if line.strip() == "":  # Skip empty lines
            continue
        words = line.split()  # First split the line into words
        syllable_segmented_line = " ".join([syllable for word in words for syllable in segment_syllables(word)])
        processed_lines.append(syllable_segmented_line)
    return processed_lines
```

This function processes data by segmenting each word into its syllables. It starts by initializing an empty list processed_lines, which will hold the syllable-segmented lines.The function iterates through each line in the data. Empty lines are skipped using an if condition.

Each line is split into individual words using line.split(). For each word, the segment_syllables function is called, which segments the word into its syllables. The segmented syllables are then joined back with spaces, treating each syllable as a separate word.Finally, the processed lines with syllables are appended to processed_lines, and the function returns this list.

Note that segment_syllables function is ready function that I found from github repo.

I didn't seperate the punctuation marks from the syllables , othewise the syllable-based model wasn't able to generate meaningful sentences.For example "gel " "dim." Is segmented syllable , not " gel " "dim " " . " .I chose that way , I hope , it is not problem.

I did not wanted to upload it because I didn't write my own function for segmenting syllables. If you want, I can send that function explicitly.

```python
# Function to process data for a character-based model
def process_data_char_based(data):
    processed_lines = []
    for line in data:  # Iterate over each line in the data
        if line.strip() == "":  # Skip empty lines
            continue
        # Treat each character as a token, join the characters with spaces
        char_segmented_line = " ".join(line)  # Add spaces between characters

        processed_lines.append(char_segmented_line)

    return processed_lines
```

This function processes data by treating each character as a token and splitting all characters in the line. Similar to the syllable model, it initializes an empty list processed_lines to store the character-segmented lines.Later , we are gonna return this.

It iterates through each line in the data, and if a line is empty, it skips it. Each line is split into individual characters by using " ".join(line), which inserts spaces between every character in the line. The processed lines are then appended to processed_lines, and the function returns this list where every character is treated as a separate token.

I call these functions to apply processing to my dataset in run.py file

```python
syllable_processed_train_data = process_data_syllable_model(train_data)
syllable_processed_test_data = process_data_syllable_model(test_data)

char_processed_train_data = process_data_char_based(train_data)
char_processed_test_data = process_data_char_based(test_data)
```

CALCULATING N GRAMS FOR BOTH MODEL

For N-gram calculation , I used ngrams library from nltk.util.It is pretty easy to implement.

```
def get_ngram_counts(data, n):
    ngram_counts = defaultdict(float)  #Will be storaged with defaultdict to avoid KeyError
    for line in data:
        tokens = line.split()
        ngrams_in_line = ngrams(tokens, n) # Generate n-grams for the current line
        for ngram in ngrams_in_line:
            ngram_counts[ngram] += 1 # Count the n-grams in the data and store them in a dictionary
                                     # with the n-gram as the key and the count as the value
    return ngram_counts
```

This function, get_ngram_counts, takes two arguments:

- **data** – the processed dataset based on either character or syllable models.
- **n** – the number of n-grams.

First, I create a dictionary using the defaultdict from the collections library, initializing it to avoid KeyErrors. Initially, I used Counter, but I switched to defaultdict to facilitate sparse data representation.

Next, I split each line into tokens using the line.split() function. Then, I pass these tokens and the value of n (the number of n-grams) to the ngrams function to generate the n-grams. For each n-gram found, I increment its count in
the ngram_counts dictionary.ngrams is a ready function from nltk.util and it handles ngrams.

Finally, the function returns the ngram_counts dictionary, where the keys are the n-grams and the values are their respective counts.

I need both train and test data ngrams.On perplexity function , I am gonna use both processed test data and train data.

```
print("*_* Calculating the 1,2 and 3 grams for both syllable and char based models.... for train and test data.")

syllable_1gram_train_counts = calculate_n_grams(syllable_processed_train_data, 1) #return a dictionary
syllable_1gram_test_counts = calculate_n_grams(syllable_processed_test_data, 1)
char_1gram_train_counts = calculate_n_grams(char_processed_train_data, 1)
char_1gram_test_counts = calculate_n_grams(char_processed_test_data, 1)

syllable_2gram_train_counts = calculate_n_grams(syllable_processed_train_data, 2)
syllable_2gram_test_counts = calculate_n_grams(syllable_processed_test_data,2)
char_2gram_train_counts = calculate_n_grams(char_processed_train_data, 2)
char_2gram_test_counts = calculate_n_grams(char_processed_test_data, 2)

syllable_3gram_train_counts = calculate_n_grams(syllable_processed_train_data, 3)
syllable_3gram_test_counts = calculate_n_grams(syllable_processed_test_data,3)
char_3gram_train_counts = calculate_n_grams(char_processed_train_data, 3)
char_3gram_test_counts = calculate_n_grams(char_processed_test_data, 3)
```

SMOOTHING THE N-GRAMS

The code of the function is :

```
def good_turing_smoothing(ngram_counts, total_ngrams):

    frequency_of_n_grams = Counter(ngram_counts.values()) #it count the frequency of each n-gram in the n-gram counts dictionary
    smoothed_ngram_counts = defaultdict(float) #it will be used to store the smoothed n-gram counts

    for ngram, count in ngram_counts.items(): #ngram_counts = {('I', 'am'): 3, ('am', 'awesome'): 2}
        if count + 1 in frequency_of_n_grams: #if there is an n-gram with count + 1
            # Adjust the count using Good-Turing formula
            adjusted_count = (count + 1) * frequency_of_n_grams[count + 1] / frequency_of_n_grams[count]
        else:
            adjusted_count = count  # If there's no N-gram with count + 1, leave it unchanged

        # Normalize by the total number of N-grams
        smoothed_ngram_counts[ngram] = adjusted_count / total_ngrams #store the smoothed n-gram count in the dictionary

    # Calculate probability for unseen N-grams (those with 0 count)
    N1 = frequency_of_n_grams.get(1, 0)  # N1 is the number of N-grams that occur exactly once
    unseen_prob = N1 / total_ngrams if total_ngrams > 0 else 0  # Good-Turing estimate for unseen N-grams

    return smoothed_ngram_counts, unseen_prob #return the smoothed n-gram counts and the probability of unseen n-grams
```

In this part , I used the good_turing_smoothing function to smooth n-gram counts by adjusting the probabilities of seen n-grams and assigning a probability to unseen ones:

1. **ngram_counts** is a dictionary where the keys are n-grams, and the values represent the counts of those n-grams.
2. **total_ngrams** the total number of n-grams in the dataset.as I said before, we found the total number of n-grams by using sum to use the exactly here.

To start, I calculate the "frequency of n-grams" using Counter. This step counts the frequency of each n-gram in the n-gram counts dictionary. For example, we are working with 1-gram. The counts are (‘la’) : 10 , (‘le’): 12, (‘ri’) 12 , the frequencies are : (10 : 1 , 12 : 2).

```
frequency_of_n_grams = Counter(ngram_counts.values())  # Counts the frequency of each n-gram count in the data
```

Next, I initialize smoothed_ngram_counts, a dictionary that will store the adjusted counts after smoothing. After this , looping through each n-gram and its count, then apply the Good-Turing smoothing formula.

```
smoothed_ngram_counts = defaultdict(float) # I will  stored the smoothed probabilities of n-grams later.
```

Specifically, for each count, I check if an n-gram occurs with a count of count + 1 in the frequency of n-grams . If it does, I adjust the count with the following formula:

```
# Adjust the count using Good-Turing formula
adjusted_count = (count + 1) * frequency_of_n_grams.get(count + 1, 0) / frequency_of_n_grams.get(count, 1)
# frequency_of_n_grams.get(count + 1, 0) returns the frequency of N-grams with count + 1
```

This formula helps to increase the probability of seeing new or rare n-grams. In this part, it adjusts the number of n-grams, assigning it a more realistic value. According The

Good-Turing method slightly lowers the probability of seen n-grams, leaving a small chance for n-grams we haven't seen. This helps the model predict new, unseen n-grams better.

```
N1 = frequency_of_n_grams.get(1, 0)# N1 is the number of N-grams that occur exactly once.
```

To handle unseen n-grams (the ones with a count of 0), I use N1, which represents the number of n-grams that appear exactly once. The get(1, 0) function gets the number of n-grams if there is an n-gram with frequency 1, otherwise it returns 0.

```
unseen_prob = N1 / total_ngrams if total_ngrams > 0 else 0  # Good-Turing estimate for unseen N-grams
```

The probability for unseen n-grams is estimated as N1 / total_ngrams, provided that total_ngrams is greater than zero. If the total number of n-grams (total_ngrams) is greater than zero, unseen_prob is calculated as the value of N1 divided by the total number of n-grams. This gives the probability of n-grams that were never observed. If the total number of n-grams is zero, the probability of unseen n-grams is set to 0.

In the end, the function returns:

- **smoothed_ngram_counts**: contains the smoothed probabilities for the observed n-grams.
- **unseen_prob**: the estimated probability for n-grams that were not observed in the data.

Then I call the good_turing_smoothing function for each model and each n-gram, and I also get their unseen probabilities where I need them to compute perplexities.

For smoothing function , I need both n-gram counts and total numbers of n-grams.

```
char_1gram_smoothed , char_1gram_unseen = good_turing_smoothing(char_1gram_counts, total_char_1gram_counts)
char_2gram_smoothed , char_2gram_unseen = good_turing_smoothing(char_2gram_counts, total_char_2gram_counts)
char_3gram_smoothed , char_3gram_unseen = good_turing_smoothing(char_3gram_counts, total_char_3gram_counts)

syllable_1gram_smoothed , syllable_1gram_unseen = good_turing_smoothing(syllable_1gram_counts, total_syllable_1gram_counts)
syllable_2gram_smoothed , syllable_2gram_unseen = good_turing_smoothing(syllable_2gram_counts, total_syllable_2gram_counts)
syllable_3gram_smoothed , syllable_3gram_unseen = good_turing_smoothing(syllable_3gram_counts, total_syllable_3gram_counts)
```

Their results will be published on table and result part.

```python
def print_smoothed_ngrams(ngram_counts):

    ngram_counter = Counter(ngram_counts)

    # Print the top 10 n-grams
    for ngram, count in ngram_counter.most_common(10):

        # If the n-gram is a 1-gram, print it as a tuple
        if len(ngram) == 1:
            print(f"('{ngram[0]}')", count)
        else:
            print(ngram, count)
```

I wrote a basic print function for smoothed ngrams. I just put a condition for length , but is not necessary.

I need this print function to see the results.

```python
print("The smoothed n-grams for char 1-gram:")
print_smoothed_ngrams(char_1gram_smoothed)

print("The smoothed n-grams for char 2-gram:")
print_smoothed_ngrams(char_2gram_smoothed)

print("The smoothed n-grams for char 3-gram:")
print_smoothed_ngrams(char_3gram_smoothed)

print("The smoothed n-grams for syllable 1-gram:")
print_smoothed_ngrams(syllable_1gram_smoothed)

print("The smoothed n-grams for syllable 2-gram:")
print_smoothed_ngrams(syllable_2gram_smoothed)

print("The smoothed n-grams for syllable 3-gram:")
print_smoothed_ngrams(syllable_3gram_smoothed)
```

Every result of these implementations will be published in table and result part.

CALCULATING PERPLEXITY

Before we try to generate sentences , we should calculate the perplexities of all n-grams.Before the explanation of my code , you should know these information about perplexity :

- Perplexity is a measure of how well a probability model predicts a sample.
- It is calculated as the inverse probability of the test set, normalised by the number of words.
- The lower the perplexity, the better the model predicts the sample.
- Perplexity is calculated by summing the negative log probabilities of each n-gram in the test data and then exponentiating the result.
- This is done to avoid numerical underflow, as the product of many probabilities can become very small.

The code of the function is :

```python
import math

def calculate_perplexity(ngrams_in_test_data, smoothed_probabilities, unseen_prob):


    log_prob_sum = 0  # Sum of log probabilities
    total_ngrams_counted = len(ngrams_in_test_data)  # Total length of n-grams in the test data
    # Calculate log probabilities for each n-gram
    for ngram in ngrams_in_test_data:
        # Use smoothed probability if available, else use the unseen probability
        prob = smoothed_probabilities.get(ngram, unseen_prob)
        log_prob_sum += -math.log(prob)

    # Compute perplexity
    perplexity = math.exp(log_prob_sum / total_ngrams_counted)
    return perplexity
```

I used this function to calculate perplexity to evaluate the performance of our n-gram models. It basically calculates the perplexity of a given test dataset.

It takes three parameters:ngrams_in_test_data , which contains the sentences to be evaluated; smoothed_probabilities , a dictionary of n-gram probabilities from the training data and unseen probability , the estimated probability for unseen n-grams. All the three parameters are already calculated. Thus, it is easy to calculate perplexity.

First, I start by initializing log prob. The log prob sum will accumulate the negative logarithmic probabilities of the n-grams found in the test data.At the same time , I need total_ngrams_counted to calculate perplexity at the end.I'll explain how will I use.

Note that I've already processed the test data based on character and syllable model. Thus, no need to worrying about process them again.

```
# Calculate log probabilities for each n-gram
for ngram in ngrams_in_test_data:
    # Use smoothed probability if available, else use the unseen probability
    prob = smoothed_probabilities.get(ngram, unseen_prob)
    log_prob_sum += -math.log(prob)
```

In this part, the iterating of whole ngrams in test data is essential because I must calculate log probabilities for each n-gram.

First , I use the smoothed probability for prob variable .However, if does not exist , then prob will be unseen probability.

After I've got the negative logarithm of the probability, just add it to the log prob sum. Note that I must use the negative log probability to prevent numerical underflow otherwise the product of many probabilities can become very small and cause numerical underflow.

```
log_prob_sum += -math.log(prob)
```

Then I calculate perplexity by taking the exponent of the average of the negative log probabilities. Now , it is time to calculate perplexity.

```
# Compute perplexity
perplexity = math.exp(log_prob_sum / total_ngrams_counted)
```

In that part , I just compute perplexity based on formula , nothing else.

There is one part that chatgpt recommend me.

```
if total_ngrams_counted == 0:
    return float('inf')  # Avoid division by zero if no n-grams in test data
```

It is important because avoiding division by zero is obligation , otherwise there could be math.domain error.

In the end , I just call the function to see results.

```
print("Calculating perplexity of the test data for each n-gram model...")
char_1gram_perplexity = calculate_perplexity(char_1gram_test_counts, char_1gram_smoothed,char_1gram_unseen)
char_2gram_perplexity = calculate_perplexity(char_2gram_test_counts, char_2gram_smoothed,char_2gram_unseen)
char_3gram_perplexity = calculate_perplexity(char_3gram_test_counts, char_3gram_smoothed,char_3gram_unseen)
syllable_1gram_perplexity = calculate_perplexity(syllable_1gram_test_counts, syllable_1gram_smoothed,syllable_1gram_unseen)
syllable_2gram_perplexity = calculate_perplexity(syllable_2gram_test_counts, syllable_2gram_smoothed,syllable_2gram_unseen)
syllable_3gram_perplexity = calculate_perplexity(syllable_3gram_test_counts, syllable_3gram_smoothed,syllable_3gram_unseen)
```

Every result will be shown in results and table section.

RANDOM SENTENCE GENERATION

Now , it is time to generate sentences.

While generating sentences, I use one helper function that checks if each word in the n-gram contains only alphabetic characters , punctuation marks and numbers. (a-z , A-Z , or . , ! ? ... , 0 – 9)

Is Valid N Gram ? :

```
def is_valid_ngram(ngram):
    """ Check if the n-gram contains only valid characters: alphabetic, punctuation, and numbers. """
    return all(re.match(r'^[a-z0-9.,!?;:\'\"()\-]+$', word) for word in ngram)
```

The logic is too easy. It uses a generator expression to iterate over each word in the ngram. The re.match function checks if the words consisty only of alphabetic characters punctuations and digits.I got this regex from chatgpt.

“All “ function returns true if and only if all words in the n-gram are valid for the regex.

The code of function that generates sentences :

```
def generate_sentence(ngram_counts, n, top_n=5, max_sentence_length=50):
    sentence = []
    # Count the n-grams
    ngram_counter = Counter(ngram_counts)

    # Filter valid n-grams for the initial selection
    valid_top_ngrams = [ngram for ngram, _ in ngram_counter.most_common() if is_valid_ngram(ngram)]
    if not valid_top_ngrams:
        return "No valid n-grams to generate a sentence."

    # Choose an initial n-gram from the top 5 valid options
    current_ngram = random.choice(valid_top_ngrams[:top_n])
    sentence.extend(current_ngram)

    while len(sentence) < max_sentence_length:
        # Define context for the next n-gram
        context = tuple(sentence[-(n-1):]) if n > 1 else ()
        possible_ngrams = [ngram for ngram, _ in ngram_counter.items() if ngram[:n-1] == context]

        # Filter valid n-grams and take the top 5
        valid_possible_ngrams = [ngram for ngram in possible_ngrams if is_valid_ngram(ngram)]
        top_valid_ngrams = valid_possible_ngrams[:top_n]

        if not top_valid_ngrams:
            break  # No more valid n-grams

        # Select the next n-gram randomly from the top 5 valid options
        next_ngram = random.choice(top_valid_ngrams)
        sentence.append(next_ngram[-1])  # Add only the new word to the sentence

    return ' '.join(sentence)
```

The function have 4 parameters :

- ngram_counts: A list of n-grams of training data from which to generate a sentence.Before using it , we preprocessed the data , than find the n gramcounts.Everything we did was for this generate sentence function. Important thing is it is we use training data for generating sentence because it represents the patterns and common sequences in our data that my model has learned, we use test data to evaluate the model's accuracy or perplexity on unseen data, not for generating sentences.
- n: The size of the n-grams .1 for unigram , 2 for bigrams , 3 trigrams etc.
- top_n: The number of top n-grams to consider for sentence generation.The default is 5 for the homework thus I assignt default 5.
- max_sentence_length: The maximum length of the generated sentence (default is 50 words).

Initializing the sentence :

```
def generate_sentence(ngram_counts, n, top_n=5, max_sentence_length=50):
    sentence = []
    # Count the n-grams
    ngram_counter = Counter(ngram_counts)
```

- sentence: An empty list to hold the words of the generated sentence.
- ngram_counter: A Counter object that counts the occurrences of each n-gram in ngram_counts.
- We use counter because unlike defaultdict, Counter has additional functions like most_common, elements, subtract.It just makes easier.We are gonna use most_common function for this code.

Filtering the valid n-grams :

```
# Filter valid n-grams for the initial selection
valid_top_ngrams = [ngram for ngram, _ in ngram_counter.most_common() if is_valid_ngram(ngram)]
```

Now time to use this helper function. This line filters the top n-grams to only include those that are valid.We are using the is_valid_ngram function.We exactly use most_common function here.

Then , we gonna initialize the current n-gram :

A random chosen valid n-gram is selected as the starting point for the sentence.Importan thing is we gonna continue from only with the valid ngrams.]

```
# Choose an initial n-gram from the top 5 valid options
current_ngram = random.choice(valid_top_ngrams[:top_n])
sentence.extend(current_ngram) # Add the initial n-gram to the sentence
```

Then the selected n-gram is extended to the sentence list.

Now , it is time to generate sentence :

I have one while loop that generate sentences until it reaches the maximum length.

```
while len(sentence) < max_sentence_length:
    # Define context for the next n-gram
    context = tuple(sentence[-(n-1):]) if n > 1 else ()
```

Context is the variable captures the last n-1 words of the current sentence to provide context for generating the next word.

```
possible_ngrams = [ngram for ngram in ngram_counts if ngram[:n-1] == context] # Get possible n-grams that match the context
```

In this line , I find all possible n-grams that match the current context.

```
# Filter valid possible n-grams
valid_possible_ngrams = [ng for ng in possible_ngrams if is_valid_ngram(ng)]
```

After getting possible_ngrams , I filter out invalid possible n-grams.

Now , time to select next n-gram and append it.

```
top_valid_ngrams = valid_possible_ngrams[:top_n]

if not top_valid_ngrams:
    break  # No more valid n-grams

# Select the next n-gram randomly from the top 5 valid options
next_ngram = random.choice(top_valid_ngrams)
```

The first line creates a new list named as top_valid_ngrams that contains only the first top_n (5) elements from the valid_possible_ngrams list.Then the condition checks if the top_valid_ngrams list is empty. If it is empty, it indicates that there are no valid n-grams available to choose from for sentence generatin.

In the line with random choice a new n-gram is randomly selected from the top_valid_ngrams list. The random.choice() function ensures that the selection is made randomly from the filtered options, introducing variability into the sentence generation process.

By choosing randomly from the top valid n-grams, the function allows for creative and varied sentence constructions while still maintaining context based on frequency and validity.

In the end , we add the new word to the sentence.

```
sentence.append(next_ngram[-1])  # Add only the new word to the sentence
```

The words in the sentence list are joined into a single string, separated by spaces to improve readibility and returned.

```
return ' '.join(sentence)
```

After this , the generated sentences are ready.

RESULTS & TABLES

(The analysis of these results will be presented in the Analysis and Conclusion section.This part only includes results.)

In this section, I will show my result of ngram probabilities , unseen probabilities , perplexities , generated sentences and their analyses.

First , I will start with smoothing probabilities for both character-based model and syllable-based model.

Smoothing Probabilities of Character-Based Model.

I print top-10 probabilities from higher to lower as it was written on the assignment pdf.

The smoothing probabilities of 1-gram character-based model :

1-gram	*Good Turing Smoothing Probability Char-Based Model*
(' i ')	*0.12340872771157227*
(' a ')	*0.11163881385405836*
(' e ')	*0.08544406113809157*
(' n ')	*0.07080009736960534*
(' r ')	*0.06721629745924947*
(' l ')	*0.06561949968124815*
(' s ')	*0.04759756749424475*
(' u ')	*0.043891277133815715*
(' k ')	*0.04081525263404291*
(' d ')	*0.04068789192264918*

The smoothing probabilities of 2-gram character-based model :

2-gram	Good Turing Smoothing Probability Char-Based Model
('i', 'n')	0.024233982878532966
('a', 'r')	0.018108435443460545
('a', 'n')	0.016985122834085223
('l', 'a')	0.016154761204981473
('e', 'r')	0.015046621613769927
('i', 'r')	0.0148717954811721
('r', 'i')	0.014204148730948793
('i', 'l')	0.014019158898429623
('l', 'e')	0.013659234907349296
('s', 'i')	0.013543934167002615

The smoothing probabilities of 3-gram character-based model :

3-gram	Good Turing Smoothing Probability Char-Based Model
('l', 'a', 'r')	0.006959219421415907
('i', 'n', 'd')	0.0069468785317187915
('l', 'e', 'r')	0.005438953968071928
('e', 'r', 'i')	0.00532493568777075
('n', 'd', 'a')	0.005291949091453649
('a', 'r', 'i')	0.005180524321405561
('i', 'r', ':')	0.004963209257734635
('a', 'r', 'a')	0.004816105107996621

('i', 'n', 'i')	0.0046383720985352225
('r', 'i', 'n')	0.00437311432260205

Smoothing Probabilities of Syllable-Based Model :

The smoothing probabilities of Syllable-Based 1 Gram Model :

1-gram	Good Turing Smoothing Probability Syllable-Based Model
('la')	0.023046241092183766
('le')	0.020654672615362617
('ri')	0.019690295595200233
('si')	0.018661555685559897
('da')	0.018330313110498115
('de')	0.01703424090377464
('li')	0.01623304950756636
('ya')	0.014094130303935526
('ve')	0.013674536429932441
('a')	0.013013278700823952

The smoothing probabilities of Syllable-Based 2 Gram Model :

2-gram	Good Turing Smoothing Probability Syllable-Based Model
('le', 'ri')	0.0045415911841754555
('la', 'ri')	0.0045322439322271245
('mis', 'tir.')	0.003184359323345361
('o', 'la')	0.003180552046990679

('la', 'rak')	0.0028126992215627227
('i', 'le')	0.0027801586636578097
('la', 'rin')	0.002365577348439943
('i', 'cin')	0.0021527894499719534
('sin', 'da')	0.002146736657563184
('le', 'rin')	0.0020496278006635663

The smoothing probabilities of Syllable-Based 3 Gram Model :

3-gram	Good Turing Smoothing Probability Syllable-Based Model
('o', 'la', 'rak')	0.0026318079278473032
('yi', 'lin', 'da')	0.0015411914698329676
('ta', 'ra', 'fin')	0.001393649422592502
('ra', 'fin', 'dan')	0.0013340486372000636
('mak', 'ta', 'dir.')	0.0012639962681550226
('ra', 'sin', 'da')	0.0011226559136240833
('mek', 'te', 'dir.')	0.0008906433488057696
('a', 'ra', 'sin')	0.0008906197383005303
('ol', 'du', 'gu')	0.0008716290219197097
('ma', 'hal', 'le')	0.0008330258458534333

The unseen probabilities for Char-Based Model:

Char-Based Model	*Unseen Probability*
1 -gram	*4.618457141318067e-06*
2-gram	*8.041761355924608e-05*
3-gram	*0.0004297378239098862*

The unseen probabilities for Syllable-Based Model:

Char-Based Model	*Unseen Probability*
1 -gram	*0.003247986147326424*
2-gram	*0.028201987789862 71*
3-gram	*0.09929745052820006*

PERPLEXITY VALUES FOR BOTH MODELS

The perplexity values of char-based model : # will be arranged.

Char-Based Model	*Perplexity Probability*
1 -gram	*6054524.165742934*
2-gram	*2701373.30357834*
3-gram	*2327.0001949116927*

The perplexity values of syllable-based model :

Syllable -Based Model	*Perplexity Probability*
1 -gram	*758805.1711627281*
2-gram	*381615.21103128284*
3-gram	*151949.686101408*

GENERATED SENTENCES :

I arrange max-length as 50 words.You can change it if you want.

N-grams	*Generated Sentence*
Syllable-Based 3 Gram	*ta ra fin ke penk a ci sin dan do la sir ken is ti yo rum. so nun da,da kim li gi ni dev ret mis tir. yi lin da top lu mun u lus -sim di ki a la rak ge ci len e di son); e lekt rik*
SyllableBased 2 Gram	*la ri i cin gno sis te pe ran hu di an der lecht ta ki sa nat ci haz red ver si ko cak, bu ku ru ma, sa hil ler den ler de ve di an cak tir. 1951'den be ke da, ya li bir koy i se*
Syllable-Based 1 Gram	*le vi cor ra cor cor vi ro po vi cor po ra cor cor cor ro vi cor vi ro cor vi cor cor ro ra vi ro ra ra ro po vi ro cor ro ra vi ra cor cor cor po vi po vi ra ro ro*
Chart-Based 3 Gram	*l e r k u s t a s m u n u 9 1 y o t a l i g a z d i j u r n u k k o p y a r d s c r) , s u n u k e*
Char-Based 2 Gram	*l a l i c a k u c a l i c i o n c e d a k o j e n ; h i , y e g a k u c t l i r p e n l a y i c a y*
Char-Based 1 Gram	*n v i c v v v i c v r i c r v c r r i v i i c v i o r r c c c c r c c r o v r v r r o i v v o v v v*

N-grams	*Generated Sentence*
Syllable-Based 3 Gram	*o la rak ge ri al di gi i se is te yen ai le ri mac ta, gok ce pe ri fe mi nen 15 da ka i le ko nus ma ci lar ta ki miy la is pan ya, tur ku la' ta ra fin da*
SyllableBased 2 Gram	*la ri ve hiz met ri sin day sap lan 8-4 yen bir lik la di. yi su re ye ye ku ru ma, hi o zel lik la rin dan son 'in ilk ya la ri da bur sas por' la rin de ilk bas la ma ya yo*
Syllable-Based 1 Gram	*ri po cor vi cor po ra ro ra ro po ro po cor ra ro ro ra po ro vi po vi po po po ro cor cor ro po po po ra vi po cor cor cor cor ra ra po po ra po ro ra cor ra*
Chart-Based 3 Gram	*l e r i m d u v e t i o l o s t u n s i r k o c u o l l e n b o n ; b u r k u r t a , e l g u t a s*
Char-Based 2 Gram	*a r a m v l i , y i s e d u r s e n l e t i , g o n d i o j a m a k u s i c e t a m v y i r p o j i*
Char-Based 1 Gram	*n v r i r r r o i o i v i r c c v o i v r i v i i r r c i o i c c v r v v o o c r i o o o i i v v i*

N-grams	*Generated Sentence*
Syllable-Based 3 Gram	*ta ra fin ta si var di, jack 'in av kos ku' ne uy mak tay ken, bu gun i se de na co pe' da go ze carp mis ka ra buk 3 mart 2012 dar be si ust len mis, ta ra fin dan hep zen gin*
SyllableBased 2 Gram	*mis tir. ko ru lan bu ra dev ral dir. tu nun 9. yuz de ilk a ko no mo to, ta rim ve tek no lo bo lun me tek no mo to, ei da bas hay ret ti me,yi lin da ni mar ka tes lim o du*
Syllable-Based 1 Gram	*da ro ra cor ra ro cor vi po ro ro po ra ro po po po vi po ra cor ro ro ro vi vi cor ro*

	ro ra ro po po vi ra ro po po ra ro ro cor ra po ro vi ra vi ra po
Char-Based 3 Gram	***l a r s u r k o p y i s u k m , t b o t t i . " h m u s m i o g r e n b o l l u m l e r i m v a m e***
Char-Based 2 Gram	***l a l e g r a k n o l i c e g a y a y i s a m i r a k a l o n c t i o r a k u n c e t r m e n d m e***
Char-Based 1 Gram	***i i v c o i i i v c c o o o v c i o o v i c c i r o c o i r r c c o r c o i c r i r i c i i o c i o***

N-grams	***Generated Sentence (15 WORDS)***
Syllable-Based 3 Gram	***ra fin dan yap ti gi ni e les ti ren be le di ye***
SyllableBased 2 Gram	***o la son ra dan ci kar si las mak u re ka ma si***
Syllable-Based 1 Gram	***la bo de rak ri de ci zil yon la ti mey hin da ta***
Chart-Based 3 Gram	***l e r k u s t a s m u n u 9 1 y o t a l i g a z d i j u r n u k k o p y a r d s c r) , s u n u k e***
Char-Based 2 Gram	***e r i a k a z i c i l a y e n***
Char-Based 1 Gram	***v i v o i o o i i r i c r o i***

Analysis & Conclusion

Analysis of smoothing probabilities of char-based model :

The most clear thing is that the 1-gram probabilities are the highest among all n-grams.Characters like 'i', 'a', 'e', and 'n' have significant probabilities.Probability is pretty higher then other n-grams , but it has less context.

When we look at 2-gram model ,the probabilities decrease compared to 1-grams.It is obvius that ('i', 'n'), ('a', 'r'), and ('e', 'r') have lower probabilities than single characters.It is expected since pairing characters reduces the overall frequency of each unique pair.

The 2-gram model reflects common character combinations.The advantge is the adding slightly more context than 1-grams alone.

In the 3-gram model, probabilities drop further. Sequences like ('l', 'a', 'r') and ('i', 'n', 'd') represent specific combinations but appear less frequently. The reduction in probabilities is typical for higher n-grams due to the increased specificity of each sequence, which naturally seems less often. The 3-gram model provides more contextual nformation about character seuences but the frequencies are limited than 1 and 2-gram models.

Shortly , the value of n increases in a character-based model, the probability of each n-gram decreases. This decrease shows that longer character sequences are more specific and occur less often which is the most expected thing ever.

Analysis of smoothing probabilities of syllable-based model :

In the syllable model , first thing I can say is that the 1-gram probabilities are higher than more number of n grams. The syllables like ('la'), ('le'), and ('ri') showing significant values. This indicates these syllables are common in the language.

In the 2-gram mode , the probabilities for common syllable pairs like (‘le’ , ‘ri’) , (‘la’ , ‘ri’) indicates these syllables are common in the Turkish wikipedia.Also , syllable 2-gram model capture more context than 1-gram syllable model.

The 3-gram syllable probabilities are the lowest, with sequences like ('o', 'la', 'rak') showing a marked drop in frequency. It is obvius because the 3-gram sylalble model is much more spesific, which decreases the smoothing probabilities.It is normal because it has more context than the 1 and 2 gram models.

In summary , I can say that the probabilities decrease across n- grams.This decreasing means that the more n gram the more spesific values and this decreases their probabilities.However ,it also improves the ability of prediction.

Comparison between syllable-based model and character-based model:

For both character-based and syllable-based model , the smoothing probabilities of n grams are decreasing while the number of n grams are increasing.However , the char-based model have more probabilities then syllable based model , because syllable based model captures more spesific ngrams than char-based model.

You are also gonna see the difference when the sentences are generated , but let me give a spoiler.The syllabel based can make much more strong prediction than char based model.

Analysis of unseen probabilities :

Unseen probabilities represent the likelihood of encountering n-grams that were not present in the training data. These probabilities mean that how well the model can generalize to new data. The unseen probability for syllable 1-grams is significantly higher than that of character 1-grams. This may be proof of beign individual characters may rarely appear in new contexts.When we compare the other 2 and 3 n-grams , always unseen probabilities of syllabes based model is much more higher than char-based model.

In summary , unseen probabilities for syllable n-grams are significantly higher than for character n-grams across all n values.I think I can say that my model may be better to handle unseen syllable combinations than character combinations.I think ,it is an expected result since syllables tend to vary more than character based model.

Perplexity Analysis :

The character-based model has much lower perplexity scores than the syllable-based model, meaning it's generally better at predicting the next character in a sequence. In contrast, the syllable-based model has higher perplexity because there are fewer possible syllables than characters ,and the are more spesific than char based model, making it harder for the model to predict.

However , in practice , the syllable-based model produces sentences better than char based model , they are natural and make more sense. If we compare 3-grams with both char-based model and syllable-based model , it is too hard to create a meaningful words or phrases since short character sequences dont have enough information to generate sentence.

Shortly , while character-based models are better at prediction accuracy, syllable-based models generate more meaningful sentences for this project.Therefore, for tasks that require natural sentence structure, syllable-based models with larger n-grams may be a better choice, even with their higher perplexity scores.

Quality of Generated Sentences :

Character-Based Model:

- 1-gram:
    - Output: The generated text is fully random, with each character disconnected from the next.
    - Explanation: Since each character is generated independently, the model lacks any context to predict what might logically follow, resulting in sequences that look like random character streams. For example, a 1-gram model might produce something like "i i v c o i i i v c c o o o v," which doesn't form words or even recognizable syllables.
    - Conclusion: The 1-gram model provides no useful output because of its inability to capture character sequences common in real words.Also , it is because top 5 is pretty low value for sentences ,every output of generated 1 -gram sentence is almost same.The sentences are gonna start with " i " " a" "e " "n ", it is expected because these are the most common ones of ngram table.

- 2-gram:
    - Output: The output starts to show occasional short words or letter pairs common in Turkish, such as "bi" or "ya." However, it is still not enough and does not form meaningful sentences.
    - Explanation: A 2-gram model considers pairs of characters, so it can form frequent two-character combinations. But without the context provided by longer n-grams, it's unable to form longer, complex wrods.
    - Example: "l a l e g r a k n o" – some parts might resemble syllables but fall short of forming real Turkish words.Maybe "lale " is mentioned here , but to be honest I am not sure.
    - Conclusion: The 2-gram character-based model can capture very short sequences but still unable to create meaningful sentences.
- 3-gram:
    - Output: The model now produces some recognizable words or parts of words, such as "gel," "ler," and "dayan." These words are the Turkish words , I can understand.

- Explanation: The 3-gram model can capture common three-character sequences, which are more likely to represent parts of real words. However, the model still lacks the context for full sentences, often generating words that are not related to each other.
- Conclusion: While the 3-gram model can generate recognizable words, it does not produce meaningful sentences. The sentences are almost always starting with “lar” “ler “ .It is expected because these are the most common ones of the ngram table.We can see the example of this situation in every n-gram generated sentence.

Syllable-Based Model:

- 1-gram:
  - Output: The output consists of random syllables that are disconnected and lack coherence.
  - Explanation: Since each syllable is generated independently, there’s no connection between them, making it impossible to form meaningful phrases or sentences.
  - Example: "da ro ra cor ra ro cor vi po ro" .Each syllable is randomly selected, leading to a meaningful sentence.
  - Conclusion: The 1-gram syllable model is not succesed at producing useful language structure because it doesn’t capture relationships between syllables.
- 2-gram:
  - Output: The 2-gram syllable model begins to produce more realistic combinations, like some syllable pairs make sense together. However, the sentences remain incoherent overall.
  - Explanation: By considering two-syllable pairs, the model can form more realistic small phrases but lacks the context for full sentences.
  - Example: "mis tir. ko ru lan bu ra" or “ .These short phrases could theoretically appear in Turkish but do not flow naturally together.
  - Conclusion: Although the 2-gram model creates some familiar-sounding phrases, it’s still not enough for meaningful sentences.
- 3-gram:

- Output: This model shows the best results among syllable-based models. It generates meaningful phrases and even some coherent structures, such as “Yılında sabit telefon vardır ancak.”
- Explanation: A 3-gram model captures three-syllable sequences, which often correspond to complete words or word pairs in Turkish. While this setup still doesn’t achieve perfect fluency, it creates segments that are closer to actual language usage.
- Example: "ta ra fin ta si var di, jack 'in av kos ku' ne uy mak tay ken” or “tarafin kepenk acisindan dolasirken istiyorum.” although some words make sense.
- Conclusion: The 3-gram syllable model performs significantly better than 1-gram and 2-gram models, capturing more language structure and generating sequences.3 gram pretty understandable.

Shortly , the syllable-based models capture more language structure than the character-based ones.However, still I can’t say that syllable model is good ,1-gram and 2-gram of syllable based model is like impossible to understand.However, I think 3 grams are starting to producing meaningful sentences.

Even in the character model's 3-gram output, while sequences looks closer to Turkish, meaningful phrases are quite rare.

No matter how 1-gram or 2-gram of syllable-based model is bad , it is still much much more better than char based model.

To generate meaningful sentences , more than 4 -gram would be quite advantage since even 3 -gram started to generate pretty meaningful words.However, I couldn’t compile it these kind high amount ngrams , my computer wasn’t handle with it.

As I explained before , 3 grams for character based model is not enough to create meaningful sentences.

Also , the problem is with 1-grams.We always choosing randomly top_5 , it meas there will be only 5 different character or syllable.This is the bigges problem of 1 -gram.It kills to chance of generating different sentences.You can obviusly see that 1-grams are %99 similar to each other.

AS RESULT

I can clearly say that syllable-based model is much more suitable for Turkish , especially for the more number of n-grams.

TABLE ON LLM USAGE & REFERENCES

THE PARTS I TOOK HELP FROM CHATGPT :

SEGMENT SYLLABLE

I found a built in function from github for segmenting syllable.It is not from chatGPT.It does not use any external library.This function segments words into syllables by first creating a "bit pattern" for vowels (1) and consonants (0). It then uses specific patterns (like 101 or 1001) to identify syllable boundaries based on Turkish syllable rules, cutting the word accordingly and appending each syllable to the syllables list.

#source: https://gist.github.com/miratcan/9196ae2591b1f34ab645520a767ced17

I didnot change anything since it works awesome for this project.

TAKING HELP FOR GOOD TURING SMOOTHING FUNCTION

While calculating adjusted counts and calculating unseen probabilities .The formula was kind of confusing , it just helped me how to integrate formula to the function.

TAKING HELP FOR RANDOM SENTENCE GENERATOR FUNCTION

To check if an n-gram is valid, I used a regex pattern suggested by ChatGPT. I wasn't sure how to design a regex that would match only specific valid characters in each word of an n-gram, so ChatGPT provided this pattern for me.

Also , ChatGPT suggested creating a context tuple to capture the last (n-1) syllable for syllable-based model or the last n-1 char for char-based model as context for selecting the next n-gram.I know how to choose random from top_5 n grams , I could not capture the words.

I also used various resources, including ChatGPT, to understand how to effectively use n-gram libraries and structure n-gram counts and selections. I am gonna share the sources that I also took help to implement the code.

- https://medium.com/nlplanet/two-minutes-nlp-perplexity-explained-with-simple-probabilities-6cdc46884584

- https://medium.com/@ompramod9921/exploring-n-grams-the-building-blocks-of-natural-language-understanding-9caa01552571

- https://github.com/wolferobert3/nlp_ngram_smoothing/blob/main/good_turing_smoothing_word_bigrams.py (It did not help me too much)
- https://www.youtube.com/watch?v=GwP8gKa-ij8&t=111s

AND OF COURSE , the slides that we used in the lecture and your video recordings from previous years on Youtube.