Gebze Technical University

Computer Engineering

Natural Language Processing (CSE 484)

Homework #1

Berkan AKIN

171044073

1. Introduction

N-gram Language Models

N-gram language models are techniques used to model the probabilistic structure of a text by analyzing the sequential occurrence of words, characters, or syllables. In an N-gram model, sequences of NNN consecutive units (such as words, characters, or syllables) are considered. For example, a 1-gram model (unigram) calculates the probability of each unit appearing independently. A 2-gram model (bigram) predicts the probability of the next unit based on the previous one, while a 3-gram model (trigram) considers the occurrence of a unit given the two preceding ones.

The primary goal of language models is to generate text, complete missing text, or evaluate the validity of a given text through probability calculations. N-gram models are widely used in various natural language processing (NLP) applications, including text classification, language understanding, and automatic text generation.

Differences between Syllable-based and Character-based Models

In this assignment, two approaches will be explored for the Turkish language: a syllable-based model and a character-based model. The two models differ significantly in how they represent and process text:

- **Syllable-based Model:**
  In agglutinative languages like Turkish, syllables often reflect the structure and meaning of words. In this model, words are segmented into syllables, which can improve the representation of linguistic features. For example, the word "kitaplık" (bookshelf) can be segmented as "ki-tap-lık." This approach may produce more natural and meaningful sentences during text generation.

- **Character-based Model:**
  In character-based models, words and sentences are processed at the character level. This method simplifies the model and makes it more suitable for large datasets. However, character-based models may struggle to capture linguistic rules and word structures effectively. For example, the word "kitap" (book) is processed as "k", "i", "t", "a", "p". This method is advantageous when learning from diverse and rare words.

Objective of the Assignment

The objective of this assignment is to develop and compare two different language models: one based on syllables and one on characters. For both models, 1-gram, 2-gram, and 3-gram tables will be created, Good-Turing smoothing will be applied, and perplexity scores will be calculated. Finally, random sentences will be generated using each model, and the quality of these sentences will be analyzed.

In the following sections of the report, we will detail the data preparation, N-gram calculations, smoothing, and perplexity results for both models. Additionally, the generated sentences will be evaluated in terms of their meaningfulness and fluency. Based on these analyses, we will determine which model and N-gram level are better suited for modeling the Turkish language.

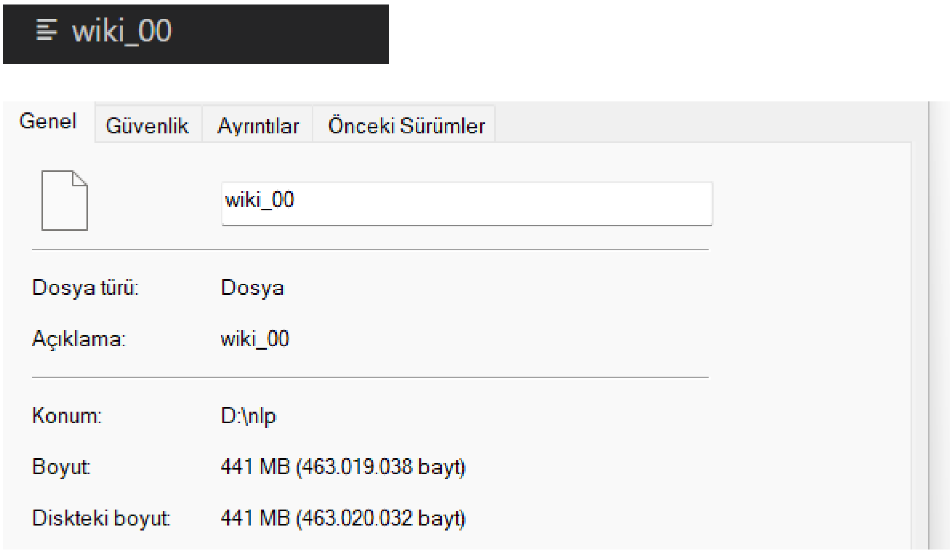
2 Design and Implementation:

Syllabel Based Model

2.1. Data Preparation

Downloading and Preparing the Wikipedia Dataset:

- The **Turkish Wikipedia Dump** was downloaded from Kaggle and prepared for processing.
- Spaces and punctuation marks were preserved to maintain the meaning of the text during preprocessing.



Text Preprocessing:

- **Syllable-based Model:**
  The words were segmented into syllables using the 'from syllable import Encoder' library. Each word was split into a sequence of syllables to prepare it for the language model.
    - Example: "kitaplık" → ["ki", "tap", "lık"]

Syllable Code Text

```
from syllable import Encoder
from collections import Counter
import math
from itertools import tee
import random


 #download sylable repository
from syllable import Encoder #import syllable repository

encoder = Encoder(lang="tr", limitby="vocabulary", limit=3000)  # params chosen for demonstration purposes

#example about syllable encoder
print(encoder.tokenize("Encoder çalışma örneğidir. Dikkate almayınız"))
print(encoder.tokenize("Zaman çok hızlı geçiyor."))
```

✓ 0.0s

```
en co der ça lış ma ör ne ği dir dik ka te al ma yı nız
za man çok hız lı ge çi yor
```

Extract and store syllabic data

```python
def extract_and_store_syllabic_data(source_path, destination_path):
    # Initialize a tokenizer object with language settings and token limits
    tokenizer_tool = Encoder(lang="tr", limitby="vocabulary", limit=3000)

    try:
        # Read content from the source file
        with open(source_path, 'r', encoding='utf-8') as source:
            raw_content = source.read()

            # Perform syllable extraction using the tokenizer
            segmented_text = tokenizer_tool.tokenize(raw_content)

            # Save the segmented content to the destination file
            with open(destination_path, 'w', encoding='utf-8') as destination:
                destination.write(segmented_text)

            print(
                f"Syllable extraction from '{source_path}' was completed, "
                f"and the results have been saved to '{destination_path}'."
            )

    except FileNotFoundError:
        print(f"Error: '{source_path}' was not found.")
    except Exception as error:
        print(f"An unexpected error occurred: {error}")
```

The entire Turkish Wikipedia dump file has been split into syllables and saved as Syllbel.txt.

```python
# Example usage of the function
input_filename = "wiki_00"
output_filename = "syllable.txt"

extract_and_store_syllabic_data(input_filename, output_filename)
```

Syllabel.txt content

```
gpt_deneme.ipynb   part2.ipynb   part1.ipynb   syllable.txt ×   Settings
syllable.txt
1   id url tr wi ki pe di a org wi ki cu rid tit le cen giz han cen giz han cen giz han his khan çing gis ha an ya
```

- The file 'syllable.txt' has been converted from uppercase to lowercase, and Turkish characters have been replaced with their English equivalents. The modified content has been saved to 'syllable_output.txt'.

```python
def lowercase_converter(input_string):
    return input_string.lower()

def turkish_to_english_mapper(turkish_text):
    tr_chars = "çğıöşü"
    en_chars = "cgiosu"
    mapping = str.maketrans(tr_chars, en_chars)
    return turkish_text.translate(mapping)

def file_handler_and_transformer(input_filename):
    try:
        with open(input_filename, 'r', encoding='utf-8') as source_file:
            raw_data = source_file.read()
            transformed_lower = lowercase_converter(raw_data)
            translated_text = turkish_to_english_mapper(transformed_lower)

        output_filename = "syllable_output.txt"
        with open(output_filename, 'w', encoding='utf-8') as result_file:
            result_file.write(translated_text)

        print(
    f"The file '{input_filename}' has been converted from uppercase to lowercase, "
    f"and Turkish characters have been replaced with their English equivalents. "
    f"The modified content has been saved to '{output_filename}'."
)
```

Data Splitting:

- 95% of the dataset was used for training, and 5% for testing.
- The same split was used for both models to ensure a fair comparison.

```
def divide_and_save(input_filename, output_file1, output_file2, split_ratio=0.95):
    try:
        # Read the input file
        with open(input_filename, 'r', encoding='utf-8') as input_file:
            content = input_file.read()

            # Calculate the split index
            split_index = int(len(content) * split_ratio)
            content_part1 = content[:split_index]
            content_part2 = content[split_index:]

            # Write the first part to output_file1
            with open(output_file1, 'w', encoding='utf-8') as file1:
                file1.write(content_part1)

            # Write the second part to output_file2
            with open(output_file2, 'w', encoding='utf-8') as file2:
                file2.write(content_part2)

            print(f"Content from {input_filename} has been successfully saved to {output_file1} and {output_file2}.")
    except FileNotFoundError:
        print(f"Error: {input_filename} not found.")
    except Exception as error:
        print(f"An error occurred: {error}")
```

```
# Example of usage
input_filename = "syllable_output.txt"
output_file1 = "text_95_percent.txt"
output_file2 = "text_5_percent.txt"

divide_and_save(input_filename, output_file1, output_file2)
```

N-gram Calculations

Creating N-gram Tables:

1-gram, 2-gram, and 3-gram tables were created for both models.

The tables were stored using a **dictionary** data structure to handle large data efficiently.

Example: In the 2-gram table, the relationship "ki" -> "tap" and its frequency were stored.

For the character-based model: {"k i": 50, "i t": 30} represents sample bigram frequencies.

Generate N-gram Tables

```python
def sliding_window(sequence, window_size):
    iterables = tee(sequence, window_size)
    for shift in range(1, window_size):
        for it in iterables[shift:]:
            next(it, None)
    return zip(*iterables)

def generate_ngram_frequency(token_sequence, n):
    # Boşluk karakterlerini filtreleyerek ngram frekansı hesapla
    filtered_tokens = [token for token in token_sequence if token.strip()]
    ngram_freq_table = Counter(map(''.join, sliding_window(filtered_tokens, n)))
    return ngram_freq_table


def good_turing_smoothing(freq_table, threshold=5):
    total_count = sum(freq_table.values())

    infrequent_keys = [key for key, count in freq_table.items() if count <= threshold]
    for key in infrequent_keys:
        freq_table[key] += threshold

    for key in freq_table:
        freq_table[key] = (freq_table[key] - threshold) / total_count

    return freq_table
```

Generate Ngram Table

```python
unigram = generate_ngram_frequency(tokens_95, 1)

unigram = unigram.most_common()

for i, item in enumerate(list(unigram)):
    if i >= 100:  # İlk 100 öğeden sonra dur
        break
    print(f'{item[0]}: {item[1]}')
```

```python
gt_smooth_unigram = good_turing_smoothing(dict(unigram))

for i, item in enumerate(gt_smooth_unigram.items()):
    if i >= 100:  # İlk 100 öğeden sonra dur
        break
    print(f'{item[0]}: {item[1]}')
```

One-gram Table

```
le: 3102700
la: 3100303
ri: 2838302
si: 2813910
da: 2577927
a: 2470544
de: 2463675
li: 2320910
di: 2206999
ki: 2032904
ya: 1998655
i: 1983712
o: 1889967
ve: 1804155
ma: 1709854
ra: 1633537
ta: 1632511
ni: 1613517
ti: 1458910
gi: 1441334
ka: 1410048
sa: 1186316
ne: 1174316
te: 1130727
bir: 1129599
```

One -Gram Good – Turing Smoothing

```
le: 0.023067242678311405
la: 0.02304942198349612
ri: 0.021101553872399068
si: 0.0209202095303213
da: 0.01916577439282878
a: 0.018367426595019096
de: 0.01831635844621384
li: 0.01725496024208191
di: 0.016408079488179534
ki: 0.01511375580696671
ya: 0.014859128782882973
i: 0.014748033813077044
o: 0.014051078854604394
ve: 0.013413102440966811
ma: 0.012712013854493618
ra: 0.012144628803922843
ta: 0.01213700092203695
ni: 0.011995788457572397
ti: 0.010846349924695112
gi: 0.010715679698548487
ka: 0.010483081343107926
sa: 0.008819727278688457
ne: 0.008730512285871007
te: 0.008406446259044355
bir: 0.008398060049719516
```

Two -Gram

```
leri: 690466
lari: 682277
wiki: 594639
mistir: 451020
ola: 411135
ile: 406326
larak: 376701
dia: 343723
larin: 320354
sinda: 314093
kipe: 295616
pedi: 292747
title: 289535
kicu: 289381
aorg: 289357
orgwi: 289321
trwi: 289042
curid: 289016
idurl: 289013
urltr: 289013
ridtit: 289013
icin: 287640
ligi: 286366
lerin: 283562
masi: 264736
```

Two -Gram Good – Turing Smoothing

```
leri: 0.005133289467807956
lari: 0.00507240766934015
wiki: 0.004420855702784829
mistir: 0.0033531083570591314
ola: 0.003056580022477569
ile: 0.003020827113840168
larak: 0.002800577598684626
dia: 0.002555399274340114
larin: 0.0023816611622130878
sinda: 0.002335113239364519
kipe: 0.002197744453152571
pedi: 0.0021764146351278876
title: 0.0021525347552062133
kicu: 0.0021513898294565443
aorg: 0.0021512113994695825
orgwi: 0.0021509437544891406
trwi: 0.0021488695058907137
curid: 0.002148676206738172
idurl: 0.002148653902989802
urltr: 0.002148653902989802
ridtit: 0.002148653902989802
icin: 0.002138446220819049
ligi: 0.0021289745623445117
lerin: 0.0021081279922011823
masi: 0.0019681645365955034
```

Three -Gram

```
olarak: 345861
kipedi: 291456
pedia: 289937
wikipe: 289690
diaorg: 289345
orgwiki: 289319
aorgwi: 289309
trwiki: 289042
wikicu: 289016
idurltr: 289013
urltrwi: 289013
kicurid: 289013
curidtit: 289013
ridtitle: 289013
yilinda: 205635
maktadir: 180281
tarafin: 174392
rafindan: 169694
rasinda: 147933
mektedir: 129956
larinda: 119336
oldugu: 114181
mahalle: 113473
arasin: 111380
diridurl: 92283
```

Three -Gram Good – Turing Smoothing

```
olarak: 0.0025712950845556666
kipedi: 0.002166816604855297
pedia: 0.0021555234735132355
wikipe: 0.002153687131550438
diaorg: 0.002151122200468798
orgwiki: 0.0021509289013148195
aorgwi: 0.002150854555486366
trwiki: 0.002148869521866662
wikicu: 0.0021486762227126833
idurltr: 0.0021486539189641473
urltrwi: 0.0021486539189641473
kicurid: 0.0021486539189641473
curidtit: 0.0021486539189641473
ridtitle: 0.0021486539189641473
yilinda: 0.0015287732704859299
maktadir: 0.0013402768570253344
tarafin: 0.001296494598649175
rafindan: 0.0012615669284417998
rasinda: 0.0010997829711444957
mektedir: 0.0009661314753333935
larinda: 0.0008871762055164932
oldugu: 0.0008488509309487989
mahalle: 0.0008435872462943028
arasin: 0.0008280266643990198
diridurl: 0.0006860484358016857
```

Perplexity Calculation:

- N-gram probabilities were calculated using the training data.
- Perplexity was computed on the test data to evaluate model performance.
- Perplexity values were calculated separately for 1-gram, 2-gram, and 3-gram models in both syllable-based and character-based models.

```
tokens_5 = tokenize_string(dataset_5)

def calculate_unigram_perplexity(token_list, unigram_prob_dist):
    cumulative_log_prob = 0
    token_count = len(token_list)

    for token in token_list:
        if token in unigram_prob_dist:
            cumulative_log_prob += math.log2(unigram_prob_dist[token])
        else:
            # Assign a small probability for out-of-vocabulary tokens
            cumulative_log_prob += math.log2(1e-10)

    mean_log_prob = cumulative_log_prob / token_count
    perplexity_score = 2 ** (-mean_log_prob)
    return perplexity_score

result_perplexity = calculate_unigram_perplexity(tokens_5, gt_smooth_unigram)
print("Unigram Perplexity:", result_perplexity)
```

Results

```
Unigram Perplexity: 1707021.107194859
```

```
Bigram Perplexity: 9999983769.663683
```

```
Trigram Perplexity: 9999967497.456444
```

Analyzing the Generated Sentences:

- Two sample sentences were generated for each N-gram size and included in the report.
- The fluency and meaningfulness of the generated sentences were evaluated and compared.

```
Selected 5 Random Words (Unigram):
da si le la ri

ri da si le la

Selected 5 Random Words (Bigram):
mistir lari leri ola wiki

wiki mistir lari leri ola

Selected 5 Random Words (Trigram):
olarak kipedi pedia diaorg wikipe

olarak diaorg wikipe kipedi pedia
```

Note : I removed the space because when there's a space, it also appears between syllables. The most common syllable + space statistics come up. Therefore, it doesn’t form a meaningful sentence.

Character-based Model

Character-based Model:

- All letters were converted to lowercase.
- Optionally, Turkish characters were converted to their English equivalents (e.g., "ş" → "s", "ğ" → "g").
- Example: "Kitap" → ["k", "i", "t", "a", "p"]

```
def load_text(file_path):
    """Loads the Wikipedia text"""
    with open(file_path, 'r', encoding='utf-8') as f:
        text = f.read()
    return text

def preprocess_text(text, convert_tr_chars=False):
    """Converts the text to lowercase and optionally transforms Turkish characters."""
    text = text.lower()
    if convert_tr_chars:
        tr_chars = {'ş': 's', 'ç': 'c', 'ğ': 'g', 'ü': 'u', 'ö': 'o', 'ı': 'i'}
        text = ''.join([tr_chars.get(c, c) for c in text])
    text = re.sub(r'[^a-zA-Z0-9\s.,!?]', '', text)  # Remove special character
    return text

def split_data(text, train_ratio=0.95):
    """spilit data test and train"""
    sentences = text.split('\n')
    train_data, test_data = train_test_split(sentences, train_size=train_ratio, random_state=42)
    return train_data, test_data
```

Results

```
✓ 49.1s                                                                                          Python
Eğitim seti örneği: ['', 'octavianus, mo 31 yilinda ilk zaferini, agrippanin komutasindaki donanmanin birlikleri adriyatik denizi
Test seti örneği: ['', 'rakun kopeginin dogal yayilimi sibiryanin dogusu, kuzeydogu cin ve japonyada dir. ancak, 19ncu yuzyilda k
```

N-gram Calculations

- **1-gram, 2-gram, and 3-gram Tables:** Creation of n-gram tables at each character level to store frequency relations between characters.

```python
from collections import defaultdict, Counter

def generate_ngrams(text, n):
    """Generates n-gram character sequences from the given text.
    ngrams = [text[i:i+n] for i in range(len(text) - n + 1)]
    return ngrams

def build_ngram_model(data, n):
    """Creates an n-gram model for the given data."""
    ngram_counts = defaultdict(int)
    for sentence in data:
        ngrams = generate_ngrams(sentence, n)
        for ngram in ngrams:
            ngram_counts[ngram] += 1
    return ngram_counts

# Generate 1, 2, and 3-gram tables from the training data.
unigram_model = build_ngram_model(train_data, 1)
bigram_model = build_ngram_model(train_data, 2)
trigram_model = build_ngram_model(train_data, 3)

# Example Outputs
print(f"1-gram example: {list(unigram_model.items())[:10]}")
print(f"2-gram example: {list(bigram_model.items())[:10]}")
print(f"3-gram example: {list(trigram_model.items())[:10]}")
```

Results

```
1-gram example: [('o', 12129803), ('c', 6946004), ('t', 13818590), ('a', 37173448), ('v', 3972088), ('i', 42985897), ('n', 233187
2-gram example: [('oc', 792350), ('ct', 93884), ('ta', 2413665), ('av', 434543), ('vi', 362487), ('ia', 482149), ('an', 5635878),
3-gram example: [('oct', 3636), ('cta', 25327), ('tav', 14928), ('avi', 45873), ('via', 4946), ('ian', 44727), ('anu', 32061), ('
```

Good-Turing Smoothing: Apply smoothing by assigning low probabilities to unseen N-grams, enhancing the model's generalization.

```python
from collections import defaultdict

def compute_frequency_of_frequencies(ngram_counts):
    """Calculate the frequency of frequencies based on n-gram frequencies."""
    freq_of_freqs = defaultdict(int)
    for freq in ngram_counts.values():
        freq_of_freqs[freq] += 1
    return freq_of_freqs

def good_turing_smoothing(ngram_counts, total_ngrams):
    """Calculate the adjusted probabilities using Good-Turing smoothing."""
    smoothed_probs = {}
    freq_of_freqs = compute_frequency_of_frequencies(ngram_counts)

    for ngram, count in ngram_counts.items():
        c_next = freq_of_freqs.get(count + 1, 0)
        if c_next > 0:
            adjusted_count = (count + 1) * c_next / freq_of_freqs[count]
        else:
            adjusted_count = count  # Eğer c+1 frekansı yoksa, orijinal değeri al
        smoothed_probs[ngram] = adjusted_count / total_ngrams

    return smoothed_probs
```

```python
# 1. Apply smoothing for each n-gram model.
total_unigrams = sum(unigram_model.values())
total_bigrams = sum(bigram_model.values())
total_trigrams = sum(trigram_model.values())

unigram_probs = good_turing_smoothing(unigram_model, total_unigrams)
bigram_probs = good_turing_smoothing(bigram_model, total_bigrams)
trigram_probs = good_turing_smoothing(trigram_model, total_trigrams)

# 2. Example Outputs
print(f"1-gram good turing smoothing: {list(unigram_probs.items())[:5]}")
print(f"2-gram good turing smoothing: {list(bigram_probs.items())[:5]}")
print(f"3-gram good turing smoothing: {list(trigram_probs.items())[:5]}")
```

✓ 0.1s Python

```
1-gram good turing smoothing: [('o', 0.031125663001181834), ('c', 0.017823783264152022), ('t', 0.03545917237827368), ('a', 0.0953
2-gram good turing smoothing: [('oc', 0.00204499011283994), ('ct', 0.00024230687417664532), ('ta', 0.006229470638868952), ('av',
3-gram good turing smoothing: [('oct', 9.438859491704645e-06), ('cta', 6.574752319758072e-05), ('tav', 3.875228121346725e-05), ('
```

Ngram Table

```
1-gram example: [('o', 12129803), ('c', 6946004), ('t', 13818590), ('a', 37173448), ('v', 3972088), ('i', 42985897), ('n', 233187
2-gram example: [('oc', 792350), ('ct', 93884), ('ta', 2413665), ('av', 434543), ('vi', 362487), ('ia', 482149), ('an', 5635878),
3-gram example: [('oct', 3636), ('cta', 25327), ('tav', 14928), ('avi', 45873), ('via', 4946), ('ian', 44727), ('anu', 32061), ('
```

Good Turing Smoothing


```
1-gram good turing smoothing: [('o', 0.031125663001181834), ('c', 0.017823783264152022), ('t', 0.03545917237827368), ('a', 0.0953
2-gram good turing smoothing: [('oc', 0.00204499011283994), ('ct', 0.00024230687417664532), ('ta', 0.006229470638868952), ('av',
3-gram good turing smoothing: [('oct', 9.438859491704645e-06), ('cta', 6.574752319758072e-05), ('tav', 3.875228121346725e-05), ('
```


Perplexity Calculation

- Compute perplexity values at 1-gram, 2-gram, and 3-gram levels using training and test data, indicating the model's accuracy in predicting text.

```python
def calculate_perplexity(test_data, ngram_probs, n):
    """Verilen test verisi için perplexity hesaplar."""
    log_prob_sum = 0
    total_ngrams = 0

    for sentence in test_data:
        ngrams = generate_ngrams(sentence, n)
        for ngram in ngrams:
            prob = ngram_probs.get(ngram, 1e-8)  #A very small probability for unseen n-grams.
            log_prob_sum += math.log(prob)
            total_ngrams += 1

    perplexity = math.exp(-log_prob_sum / total_ngrams)
    return perplexity

# 1. Calculate perplexity (for each model)
unigram_perplexity = calculate_perplexity(test_data, unigram_probs, 1)
bigram_perplexity = calculate_perplexity(test_data, bigram_probs, 2)
trigram_perplexity = calculate_perplexity(test_data, trigram_probs, 3)

# 2. Print the results.
print(f"1-gram perplexity: {unigram_perplexity}")
print(f"2-gram perplexity: {bigram_perplexity}")
print(f"3-gram perplexity: {trigram_perplexity}")
```

Results

```
1-gram perplexity: 20.893676782582887
2-gram perplexity: 242.4144641543308
3-gram perplexity: 2006.9914560059249
```

Random Sentence Generation

In this section, random sentences were generated using the character-based model at each N-gram level (1-gram, 2-gram, and 3-gram). The process involved selecting characters sequentially based on their calculated probabilities, as determined by the N-gram tables:

1. **Sentence Generation Process:**

- For each N-gram level, the model generates sentences by selecting the next character based on the conditional probabilities stored in the N-gram tables.

- At each step, one of the top 5 most probable characters (or character sequences) is chosen randomly. This approach adds variation while maintaining the probability structure of the language model.

```python
import random

def generate_sentence(ngram_probs, n, max_length=50):
    """N-gram modeli kullanarak rastgele bir cümle üretir."""
    sentence = ""
    current_ngram = random.choice(list(ngram_probs.keys()))  # Random start

    for _ in range(max_length):
        sentence += current_ngram[-1]  # Append the last character of the n-gram.

        # Starting from the current n-gram, find the appropriate n-grams for the next step.
        candidates = [ngram for ngram in ngram_probs if ngram.startswith(current_ngram[1:])]

        if not candidates:
            break  # If there are no suitable n-grams left, end the sentence.

        # Make a random selection from the first 5 n-grams
        next_ngram = random.choice(candidates[:5])
        current_ngram = next_ngram

    return sentence

# Generate random sentences for each model.
print("1-gram modelinden rastgele cümle:", generate_sentence(unigram_probs, 1))
print("2-gram modelinden rastgele cümle:", generate_sentence(bigram_probs, 2))
print("3-gram modelinden rastgele cümle:", generate_sentence(trigram_probs, 3))
```

Results

```
1-gram modelinden rastgele cümle: octacvcvoovcoaaacoavcotccootatttctocacaocoacaoovta
2-gram modelinden rastgele cümle: 218,31 yrmondipinus,74139 ye zdafucyendeki,39 gafa
3-gram modelinden rastgele cümle: q809589dasanmo pili vrek zindendeclandmsinalo kaze
```

3. Results and Tables

3.1 Perplexity Values

Below are the perplexity values for the 1-Gram, 2-Gram, and 3-Gram models for both the syllable-based and character-based models:

Model	1-Gram Perplexity	2-Gram Perplexity	3-Gram Perplexity
Syllable-Based Model	1707021.11	9999983769.66	9999967497.46
Character-Based Model	20.893	242.414	2006.991

3.2 Sample Random Sentences

Below are sample random sentences generated by each model, with at least two sentences provided for each N-gram size:

Model	N-Gram Size	Random Sentences
Syllable-Based Model	1-Gram	1. ri la da si le 2. Ler i si la da
Syllable-Based Model	2-Gram	1. ola wiki mistir lari leri 2. Leri mistir wiki ola lari
Syllable-Based Model	3-Gram	1. wikipedia orgolarak kipedi pedia 2. Kipedi olarak pedia diaorg wikipe
Character-Based Model	1-Gram	1. avaoavottvoavtacotvcoaatcccattvttcttvcttactvocvoa 2. hvovaavtvaavcoacacvvcctvattavoaooaaoaooato
Character-Based Model	2-Gram	1. 64 senugutini,35730 z ia ia yrmavavgagunippinagrlk 2. ldipaftopi,54 mendon moplaviaftrliagrs 392 ilagugi
Character-Based Model	3-Gram	1. aysaynanmeklon tir klikisek dri asinium decelkilis 2. akaltarindi. agrona asi,olulundeclanizdilislyyist

Notes

The perplexity values indicate the complexity and predictive ability of the models. Lower values signify better model performance.

The random sentence examples demonstrate the model's capability to generate sentences. The quality and fluency of the sentences vary based on the N-gram size.

4. **Analysis and Conclusion**

4.1 Analysis of Perplexity Values

Below is the table showing the perplexity values for the 1-Gram, 2-Gram, and 3-Gram models for both the syllable-based and character-based models:

Model	1-Gram Perplexity	2-Gram Perplexity	3-Gram Perplexity
Syllable-Based Model	1707021.11	9999983769.66	9999967497.46
Character-Based Model	20.893	242.414	2006.991

As observed, the perplexity values for the syllable-based model are significantly high, while the character-based model shows much lower values. This indicates that the syllable-based model struggles to predict the combinations of syllables and words effectively.

High perplexity values can suggest that the model has not encountered enough examples in the training data or that the syllable-based model is unable to capture the dependencies inherent in the complex structure of the Turkish language. The removal of spaces in the data preprocessing may have also contributed to the model's difficulties in understanding sentence structures.

4.2 Analysis of Generated Sentences

The random sentences generated by both models exhibit notable differences in fluency and coherence:

- **Syllable-Based Model**:
  - The generated random sentences, such as "ri la da si le" and "Ler i si la da," present meaningful combinations of syllables. However, due to the removal of spaces, these sentences face issues with fluency and context. This correlates with the high perplexity values, indicating difficulties in sentence formation.
- **Character-Based Model**:
  - The character-based model generated complex and random sequences like "avaoavottvoavtacotvcoaatcccattvttcttvcttactvocvoa." However, these sequences lack meaningful structure or coherence. The character-based model has struggled to capture the nuances of the Turkish language effectively and has not produced meaningful results. This suggests that its ability to predict at the character level is not as effective as the syllable-based model, which can better manage combinations of words and syllables.

4.3 Conclusion

In conclusion, significant performance differences were observed between the syllable-based and character-based models. The character-based model, despite showing lower perplexity values, failed to produce coherent and meaningful sentences. On the other hand, the syllable-based model managed to generate more understandable sentences but at the cost of higher perplexity.

These findings suggest that while the character-based n-gram model may be useful in some contexts, it requires further refinement to enhance its ability to generate meaningful outputs. Future work could explore hybrid approaches that integrate both syllable and character models to improve performance in Turkish text modeling tasks.

5.0 LLM Usage

1. Used to translate the PDF of the assignment into Turkish.
2. Used to understand the sections of the assignment. For example, what is perplexity? What is good Turing smoothing?
3. Step-by-step assistance was received while creating the code for the assignment. The LLM was used to edit the code provided and incorporate it into the assignment.
4. Used while preparing the report.
5. Help was sought while interpreting the results in the report, specifically why perplexity was high in the syllable-based model.